JAVA FOR THE STUDY OF EVOLUTION

VOLUME III version 2

Where are the bugs?

José del Carmen Rodríguez Santamaría
The EvolJava Community

Contents

Preface v

Introduction vii

1 Evolution is not teleological 1
- 1.1 No bugs . . . 1
- 1.2 The morphs of Dawkins . . . 2
- 1.3 Menus . . . 38
- 1.4 Saving a graphic . . . 47
- 1.5 Lindenmayer systems . . . 51
- 1.6 Conclusion . . . 80

2 Evolution as an style 81
- 2.1 From feelings to measures . . . 81
- 2.2 Evolution in the world of software . . . 82
- 2.3 Divide and conquer . . . 84
- 2.4 Diverse evolutionary styles . . . 96
- 2.5 The NK model . . . 102
- 2.6 Conclusion . . . 110

3 Evolution as a necessity 111
- 3.1 The Networks of life . . . 111
- 3.2 Conclusion . . . 130

4 Software discontinuity and semantics 131
- 4.1 From Grammar to semantics . . . 131
- 4.2 Semantics and genetics . . . 133
- 4.3 The complexity of a semantics . . . 135
- 4.4 Context dependence . . . 137
- 4.5 A discontinuous semantics . . . 160
- 4.6 Conclusion . . . 170

5 Is there any limit to evolution? 173
- 5.1 SAT as an abstraction from language . . . 173
- 5.2 Diffusion in logical space . . . 175
- 5.3 Solving SAT . . . 183
- 5.4 SAT dares evolution . . . 191

5.5 The arising of functions 217
5.6 Algorithmic complexity theory 250
5.7 Summary on complexity vs evolution 279
5.8 Conclusion 282

6 Conclusions **283**

7 Glossary **287**

8 Answers to exercises **293**

Preface

The series *Java for the Study of Evolution* is directed to scientists that want to manage a serious but not excessively expensive tool to study evolution by direct experimentation under perfectly controlled conditions. These requirements cannot be met in nature but only in simulations and mathematical models. In consequence, the series has three main purposes:

1. To endow the community of **researchers in biology and evolution** with *high level programming*, enabling an accurate study of models and simulations of the most diverse nature.

2. To clearly show how this tool is used to study the fundamental questions of evolution.

3. To suggest that the study of Java could be *very fruitful* for **undergraduates** in biological sciences even more than calculus alone.

We studied the fundamentals of Java, Science and Evolution in the first volume. The second volume was consecrated to study the proposal that the genome is software and that evolution is a software developer.

This is the third volume, whose purpose is to introduce a discussion of some of the hard objections to volume II, the first of which is: the closest reality to a software designer is that his or her work is always plagued with unavoidable bugs, errors of programming that are ensued by more corrections that cause more bugs. So, if we pretend that the genome of living beings is the product of evolution, which is a software developer, then we must ask: where are the bugs committed by evolution? Version 2: less errors and less bugs + migration to NetBeans + all programs are accepted by Java 8.

Most possibly, it would be difficult to enjoy this volume without having worked out volume II, but and effort to make explicit and clear the important points is promised to be done.

Bogotá, Colombia, *José Rodríguez*
Nov 2016

Introduction

We studied in Vol II of this series a proposal that has been maturing during the last century: the genome is **software** and evolution is a software **developer**. We were able to study an elementary and mandatory test of that idea: if evolution is a software developer, then we can use evolution to design software to fill in our needs and desires, much the same as a farmer can use artificial evolution to select seeds of dwarf fruit trees, which are good to resist strong winds and hurricanes.

We were in fact able to tame evolution to produce some pieces of software but our work allowed us to appreciate at once the burden of complexity: a lot of time is necessary to achieve rather humble results. So, we have been unable to see the marvels expected from evolution. Nevertheless, in spite of the shallowness of our work, we found ourselves involved in delicate questions about the origin of the genetic code: our work predicted that the genetic code must observe a rich variability, but this prediction is not backed by nature: we have a **genetic code** with minor variations that can be explained by descend and that do not correspond to the expected high level of diversity. This discrepancy is enough to consider our new insight of evolution, which presents it as a software developer, as a precious tool to generate science, a fact that we promised to illustrate in other forms in the present volume. Our first step was to discuss the next objection: If evolution is a software developer, where are the *bugs* of evolution?

The official answer to the question of bugs is presented in Chapter I, *Evolution is not teleological*: a bug is a contravention against a predefined goal of software design, so evolution cannot commit bugs because evolution has no purpose at all. The lack of goals is apparent if we present evolution as a diffusive process no more mysterious than the process of diffusion observed by ink through a paper. Evolution has two means to diffuse: mutation and recombination. We illustrate this with the help of two very nice models, the biomorphs of Dawkins and the L-grammars.

Evolution can be used to design software and its most precious virtue for computing science is presented in chapter II: *Evolution as an style* allows us to conclude that evolution is not only a process but it is additionally an style of software design, which is today highly appreciated and is under serious research by computing scientists.

Evolution is currently used to correctly explain many things, but how do life and evolution appear? An important proposal made by Kauffman ([38] 1995) is reviewed in chapter III: *Evolution as a necessity.* His idea is that under some initial conditions, it is almost certain that life and evolution shall emerge. We verify in our simulations the main points of this theory but our analysis leads us to believe that this nice proposal suffers of self-clogging and

self-digestion and has mandatory predictions that are not observed in nature.

Our experience with genetic programming shows that one must battle with the tremendous load of computations that might lead to nowhere. In trying to understand why, we have said that this is due to the discontinuity of software, which is the fact that tiny changes in software may produce terrible changes in the output else produce the dead of the program. In chapter IV, *Software discontinuity and semantics*, we make a linguistic study of this topic. Linguistics is also at the heart of the next chapter V: given the finiteness of natural resources predicted by the dominant cosmological model, is there any limit to evolution? In this regard, we study **SAT**, the satisfiability problem of mathematical logic, as an appropriate model to study the boundedness of the possibilities of evolution as a diffusive process.

Chapter 1

Evolution is not teleological

`Evolution has no purpose`

1 ***Motivation*** *The genome is software and evolution is a software developer. Under this insight, the evolutionary theory reads: evolution is the software developer that is responsible for the existence of the genome of alive beings. So, it is urgent to ask: where are the bugs of evolution?*

2 ***Purpose*** *We explain in this chapter one of the tenets of the evolutionary theory: evolution does not happen with a purpose, there is no design in nature; evolution just happens as a diffusive process of recombinant variants that are filtered by the milieu. As a consequence, evolution cannot commit bugs or errors against a predefined purpose.*

3 ***More on Java*** *At the same time that we learn more on evolution, we also learn how to build a* ***GUI (Graphical User Interface)*** *with buttons, menus and all that. We use Swing that was the Java tool used before 2015 but afterwards one might prefer JavaFX which can be found in Volumes XII and beyond. Swing is, anyway, a tool that one can use at any moment.*

1.1 No bugs

If evolution is a software developer, where are the *bugs* of evolution? Where are the multitude of errors committed along the evolutionary process that begins from no knowledge and zero understanding and that ends with so high quality software as that we see in the genome of each alive being? Can we say, for instance, that the genomic bugs caused species to disappear and that dinosaurs were more bugged than mammals?

Let us begin the discussion of this theme recalling the form as the word *bug* arrived to us:

”The late U.S Navy Admiral Grace Hopper was working with the Mark II computer in 1945. The machine used racks of relays that closed to create circuits as required. (There were no integrated circuits in 1945, of course.) But the relays were exposed to the environment, and one day a big fat moth settled on a relay contact just before it closed. Its body prevented the circuit from closing, of course, since moths are poor conductors of electricity, and so the computer did not behave as expected and the system had a bug in it.” (Lauren Wiener, 1993).

Ever since that time, the word **bug** has been used to designate misbehavior of a program, i.e. an error against a predefined function.

The correct scientific reaction against our question about evolutionary bugs results from a right understanding of the terms: a bug is an error in programming committed by a software designer against an objective that he or she has determined beforehand. So, bugs are options reserved to entities with freewill to elect aims and purposes. A **purpose** is a determination of a free will to achieve something. But evolution in nature, as it is conceived in science, has no purpose whatsoever and so it cannot commit bugs.

There is a fundamental difference between design and diffusion in terms of probability. As fiction writers have shown us, life can have infinitely many forms quite diverse from the one we know. So, when we analyze diffusion as a candidate to explain life, any structure that has the properties of life is good. But in design, apart from hitting a living form, one must usually be restricted to a specific form of life, the one that found a place in the mind of the designer. So, random assembling + mutation + recombination represents a huge gain of probability over design.

We see in this chapter two examples that illustrates the power of mutation + recombination. And at the same time we learn more Java. The first example presents a version of the biomorphs of Dawkins, by now a well known entity in the literature of evolutionary biology. The second example offers a version of a Lindenmayer system. Everything is presented through visual facilities that make ideas easily understandable even for babies. In our examples, the User plays the role of artificial selection, i.e. a mechanism that says what must be reproduced more than the rest.

1.2 The morphs of Dawkins

Richard Dawkins published in 1986 the celebrated book *The blind watchmaker, [18]*. The eternal actuality of this book is a challenge for us.

4 ***Why a challenge?***

This book enjoys our respect because of three reasons. First: it studies the very questions that modern biology and great religions have in common, say, Is the extreme complexity and beauty of alive beings a sign of design by God? Second: far from dismissing the question, this book allows everyone to look at it with enhanced admiration and respect for the marvel of life. Third: the Reader is offended with concrete and simple mechanisms to kill and explain complexity.

That book is precisely a challenge for us because of its mechanistic approach. Thus, we have on one hand that Java is the master of mechanistic thought and, on the other, that science is a fabric in which mechanisms are deeply studied and rejected or refined. That is why the challenge found in the book of Dawkins represents the type of challenges that we must learn to formulate and discuss.

At this moment in time, the very good idea is simply to present one of the proposed mechanisms found in *the blind watchmaker*. The **morphs of Dawkins** are our election, which was presented in chapter 3, *Accumulating small change*. His original presentation is

entirely evolutionary and directed to show that evolution has plenty of power to explain some aspects of biological morphology. Since that time on, the idea has been elaborated variously. We took the version elaborated in Java by Alain Gogniat ([28], 1998).

The code by Gogniat is updated below (Java also evolves) and adapted to help the Java student to make a path among the wilderness and to learn the use of graphic interfaces. The code may be used to show that simple combinatorics plus mutation has a lot to say about the origin of morphology and, eventually, of biodiversity. Artificial selection is exercised by the User.

5 *The code of biomorphs*

The present code is the result of an attempt at modernizing a very nice program that was composed using the **AWT (Abstract Window Toolkit)**, the initial facility of Java to make GUIs. But Java evolves, and today we have the option of making a GUI by using the very modern Swing package (Loy et al, [44] 2003) that was released in April 1997 by the Sun's Java Group. There is very simple rule to distinguish the two facilities: those items that come from Swing has a J as a prefix to the correspondent items offered by AWT. Example: a Swing button is JButton while an AWT button is Button and so on. Anyway, Swing needs the support of AWT, so AWT is not completely outmoded. But Swing is better and that is why it is a bit more complex.

The next code uses two parts, the ancient one, based on AWT, and a modern Swing oriented one. Such hybrids are not the modern way of making things, which is based entirely on Swing. Nevertheless, this program might help us to keep in mind that evolution may result in the existence of hybrids that contain various styles of solving problems.

The code follows:

```
//Program C5, Biomorphs
//This program is a generator of
//patterns that somehow resemble
//insects or other alive beings.
//Its generating mechanism is
//evolution= random combinations + random mutation
//+ reproduction +  selection by the User.

//Adapted from a program by Alain Cognait (1997)
//http://www.cs.bgu.ac.il/~sipper/biomorphs/evolution.html
package ejvol3v2p;

//Needed tools
import java.applet.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Arrays;
import java.util.Random;
import java.util.ArrayList;
```

```
import java.util.List;

import javax.swing.Box;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JColorChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JRadioButton;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

//Declaration of the class Biomorphs
public class Biomorphs extends Applet {

    private static final long serialVersionUID = 1L;
    static final int FRAMEHEIGHT = 650;
    static final int FRAMEWIDTH = 700;
    static final int RECTHEIGHT = 260;

    static final int RECTWIDTH = 100;
    static final int ECART = 30;
    static final int BASE = 30;
    static final int TEXTHIGH = 30;
    static final int OFFSET_TOP = 70;

    static int iTop = 0;
    static int iLeft = 10;

    static int iTop1 = OFFSET_TOP;
    int iTop2 = iTop1 + RECTHEIGHT + ECART;
    int iTop3 = iTop1 + 2 * (RECTHEIGHT + ECART);

    static int iOnecol = RECTWIDTH + ECART;
    static int iOneline = RECTHEIGHT + ECART;
    int iLeftParent = iLeft + (4 * iOnecol) + 20;

    boolean newGeneration;
    String w8[] = new String[9];
    String w2[] = new String[2];
    static final int NBMORPHS = 8;
    private int selected[];
    private int nbSelected;
    boolean bMigration;
    private int nGeneration;
    private genome Genome[];
```

```
    private morph Morph[];

    String sDNA0, sDNA1;

    private Checkbox cb[];
    private boolean bDisplayTxtGene = false;
    private TextField txtGene[];

    private morph Parent[];
    private Checkbox cbParent[];
    private TextField txtParentGene[];
    private String[] sParent = {"Father", "Mother"};
    static final int FATHER = 100;
    static final int MOTHER = 101;

    private Button btnGo;
    private Button btnMigrate;
    private Button btnShowGenome;
    private Button btnControls;
    private Label lblMutation;
    private Label lblTranscription;

    private Color frameColor;
    private Color frameColor1;
    private Color frameColor2;
    private Color backGround;
    private Color backGround1;
    private Color backGround2;
    private Color foreGround;
    private Color foreGround1;
    private Color foreGround2;

    private boolean changedf;
    private boolean changedb;
    private boolean changedo;

//Initial colors
    int DEF_FRAME_COLOR = 10;
    int DEF_BACK_COLOR = 12;
    int DEF_FORE_COLOR = 0;

// ***************** default parameters
    public int NBVAR = 8;
    private int OptimumOrder = 7;
    private int MaxSeverity = 3;
    private int TypeTranscription = 1;
```

```
    private int NbCrossover = 3;

    private Random rnd;

//*****************************************
//**************Constructor**************
//*****************************************
    public Biomorphs() {
        rnd = new Random();
        nGeneration = 0;
        bMigration = false;
        Genome = new genome[NBMORPHS];
        Morph = new morph[NBMORPHS];
        selected = new int[2];
        selected[0] = 0;
        selected[1] = 0;
        nbSelected = 0;
        Parent = new morph[2];

    }

//*****************************************
//**************INITIALIZATION*************
//*****************************************
//init() is the first method to be executed.
//It presents the first screen,
//where the User can ask for an action.
//Clicks over the first screen are sensed
//at the method action()
//and the corresponding answers are invoked.
//The various control services are autonomous
//with  sensoring  and action system  packed
//in the corresponding classes or methods.
//Once init() has been executed,
//paint() is automatically called.
    @Override
    public void init() {
        // lightGray = (192, 192, 192);
        frameColor = new Color(0);
        backGround = new Color(0);
        foreGround = new Color(0);
        frameColor = fixColor(DEF_FRAME_COLOR);
        frameColor1 = fixColor(DEF_FRAME_COLOR);
        frameColor2 = fixColor(DEF_FRAME_COLOR);
        backGround = fixColor(DEF_BACK_COLOR);
        backGround1 = fixColor(DEF_BACK_COLOR);
```

```
        backGround2 = fixColor(DEF_BACK_COLOR);
        foreGround = fixColor(DEF_FORE_COLOR);
        foreGround1 = fixColor(DEF_FORE_COLOR);
        foreGround2 = fixColor(DEF_FORE_COLOR);

        changedf = false;
        changedb = false;
        changedo = false;

        newGeneration = true;

        Genese();
        DrawFrames();
        dispOperation();
        resetCB();
        lblTranscription
                .setText("Transcription = forced order");
        TypeTranscription = 1;
    }

    private void Genese() {
        for (int i = 0; i < NBMORPHS; i++) {
            Genome[i] = new genome(rnd);
            Morph[i] = new morph(Genome[i], TypeTranscription,
                    NBVAR, OptimumOrder);
        }
    }

//**********************************
//****Graphic administration*********
//**********************************
//*********AWT based code************
//Planning of the first screen
    private void DrawFrames() {
        setLayout(null);
        resize(FRAMEWIDTH, FRAMEHEIGHT);

// ************ label Mutation
        lblMutation = new Label();
        add(lblMutation);
        lblMutation.setBackground(frameColor);
        lblMutation.setBounds(iLeft + 2 * iOnecol, iTop, 200, 20);

// ************ label Transcription
        lblTranscription = new Label();
        add(lblTranscription);
```

```
        lblTranscription.setBackground(frameColor);
        lblTranscription.setBounds(iLeft + 2 * iOnecol, iTop + 20,
                200, 20);

// ************ button GO (REPRODUCTION, CLONE, SELECT)
        btnGo = new Button("REPRODUCTION");
        add(btnGo);
        btnGo.setBounds(iLeft, iTop + 45, iOnecol + RECTWIDTH,
                20);

//************ button MUTATE
        btnMigrate = new Button("M U T A T E");
        add(btnMigrate);
        btnMigrate.setBounds(iLeft + 2 * iOnecol, iTop + 45,
                RECTWIDTH, 20);

// ************ Show & Hide Genome
        btnShowGenome = new Button("Show Genome");
        add(btnShowGenome);
        btnShowGenome.setBounds(iLeft + 3 * iOnecol, iTop + 45,
                RECTWIDTH, 20);

//************ Controls
        btnControls = new Button("Controls");
        add(btnControls);
        btnControls.setBounds(3 * iLeft + 4 * iOnecol, iTop + 45,
                RECTWIDTH, 20);

// ************ textfields to display genome
        txtGene = new TextField[NBMORPHS];
        txtParentGene = new TextField[2];

// ************ checkboxes to select parents
        cb = new Checkbox[NBMORPHS];
// lblGene = new Label[NBMORPHS];
        cbParent = new Checkbox[2];

// ************ Init textfields + checkboxes
        int No = 0;
        for (int line = 0; line < 2; line++) {
            /*
 if (line == 0)
   {
 cbParent[line] = new Checkbox(FATHER);
}
 else {
```

```
 cbParent[line] = new Checkbox(MOTHER);
 }    */
            cbParent[line] = new Checkbox(sParent[line]);
            add(cbParent[line]);
            cbParent[line].setBackground(frameColor);
            cbParent[line].setBounds(iLeftParent + ECART,
                    2 + iTop1 + RECTHEIGHT + line * iOneline,
                    60, 20);
            txtParentGene[line] = new TextField();
            add(txtParentGene[line]);
            txtParentGene[line].setVisible(false);
            txtParentGene[line].setBounds(iLeftParent,
                    2 + iTop1 + RECTHEIGHT
                    + line * iOneline - TEXTHIGH,
                    RECTWIDTH, TEXTHIGH);

            for (int col = 0; col < NBMORPHS / 2; col++) {
                cb[No] = new Checkbox("" + (No + 1));
                cb[No].setBackground(frameColor);
                add(cb[No]);
                cb[No].setBounds(iLeft + col * iOnecol + ECART,
                        2 + iTop1 + RECTHEIGHT + line * iOneline,
                        40, 20);
                txtGene[No] = new TextField();
                add(txtGene[No]);
                txtGene[No].setEditable(false);
                txtGene[No].setVisible(bDisplayTxtGene);
                txtGene[No].setBounds(iLeft + col * iOnecol,
                        2 + iTop1 + RECTHEIGHT
                        + line * iOneline - TEXTHIGH,
                        RECTWIDTH, TEXTHIGH);

                No++;
            }
        }

        lblMutation.setText("Mutation = hot");
        MaxSeverity = 3;
    }

//**********************************
//***********PAINT******************
//**********************************
//The screen is painted or repainted
    @Override
    public void paint(Graphics g) {
```

```
//Debugging instruction
//System.out.println("paint");

//**** let us take care of repaint
        if (changedf == true) {
            frameColor = frameColor2;
            frameColor2 = frameColor1;
            //System.out.println("F");
            changedf = false;
        } else {
            frameColor = frameColor2;
        }

        if (changedb == true) {
            backGround = backGround2;
            backGround2 = backGround1;
            //System.out.println("B");
            changedb = false;
        } else {
            backGround = backGround2;
        }

        if (changedo == true) {
            foreGround = foreGround2;
            foreGround2 = foreGround1;
//System.out.println("O");
            changedo = false;
        } else {
            foreGround = foreGround2;
        }

//paint() properly
        g.setColor(frameColor);
        g.fillRect(0, 0, FRAMEWIDTH, FRAMEHEIGHT);
        if (nGeneration == 0) {
            g.fillRect(0, 0, FRAMEWIDTH, FRAMEHEIGHT);
        }
        g.draw3DRect(iLeft, iTop, 2 * RECTWIDTH + ECART, 40, true);
        Font f = getFont();
        g.setColor(Color.black);
        g.setFont(new Font("Helvetica", Font.BOLD, 24));
        g.drawString("GENERATION  " + nGeneration, iLeft + 5, iTop + 32);
        g.setFont(f);

//Draw the two parents
        for (int line = 0; line < 2; line++) {
```

```
            if (nGeneration != 0) {
//Draw frames
                g.setColor(frameColor);
                g.draw3DRect(iLeftParent,
                        iTop1 + line * iOneline,
                        RECTWIDTH, RECTHEIGHT, false);
                g.setColor(backGround);
                g.fillRect(iLeftParent,
                        iTop1 + line * iOneline,
                        RECTWIDTH,
                        RECTHEIGHT);
                g.setColor(foreGround);
//Draw morphs
                Parent[line].phenotype(g,
                        iLeftParent + RECTWIDTH / 2,
                        iTop1 + RECTHEIGHT - BASE + line * iOneline,
                        NBVAR, OptimumOrder);

                if (newGeneration) {
                    txtParentGene[line].setText(Parent[line].toString()
                    txtParentGene[0].setText(sDNA0);
                    txtParentGene[1].setText(sDNA1);

                    w2[line] = txtParentGene[line].getText();
                    txtParentGene[line].setVisible(bDisplayTxtGene);
                } else {
                    txtParentGene[line].setText(w2[line]);
                    txtParentGene[line].setVisible(bDisplayTxtGene);
                }
            }
        }

//Draw the eight children
//Draw boxes
        for (int line = 0; line < 2; line++) {
            for (int col = 0; col < NBMORPHS / 2; col++) {
                g.setColor(frameColor);
                g.draw3DRect(iLeft + col * iOnecol,
                        iTop1 + line * iOneline,
                        RECTWIDTH,
                        RECTHEIGHT, false);
                g.setColor(backGround);
                g.fillRect(iLeft + col * iOnecol,
                        iTop1 + line * iOneline,
                        RECTWIDTH,
                        RECTHEIGHT);
```

```
            }
        }

//Draw morphs and texts of offspring
        int j = 0;
        g.setColor(foreGround);
//draws two raws of children
//there are four children in each row.
        for (int line = 0; line < 2; line++) {
            for (int col = 0; col < NBMORPHS / 2; col++) {
                Morph[j].phenotype(g,
                        iLeft + col * iOnecol + RECTWIDTH / 2,
                        iTop1 + line * iOneline + RECTHEIGHT - BASE,
                        NBVAR, OptimumOrder);

                if (newGeneration) {
                    txtGene[j].setText(Morph[j].toString());
                    w8[j] = txtGene[j].getText();
                } else {
                    txtGene[j].setText(w8[j]);
                }
                j++;
            }
        }
        bMigration = false;
        newGeneration = false;
    }

//The clicks of the User are listened here
    @Override
    public boolean action(Event event, Object obj) {
        Object oTarget = event.target;

//Is there any click?
        if (oTarget instanceof Button) {
            Button button = (Button) oTarget;
//Migration required?
            if (button == btnMigrate) {
                newGeneration = true;
                NewGenese();
                bMigration = true;
                repaint();
                //newGeneration=false;
                bMigration = false;
                return true;
            } //Reproduction allowed?
```

```
            else if (button == btnGo) {
                newGeneration = true;
                NewGeneration();
                repaint();
                return true;
            } //Show or hide genomes?
            else if (button == btnShowGenome) {
                newGeneration = false;
                flipShowGenome();

                return true;
            } //Access to controls?
            else if (button == btnControls) {
                newGeneration = false;
                choices();
                return true;
            }
        } //Which morph do you choose?
        else if (oTarget instanceof Checkbox) {
            Checkbox checkbox = (Checkbox) oTarget;
            String label = checkbox.getLabel();
            if (checkbox.getState() == true) {
                addSelected(label);
                return true;
            } else if (checkbox.getState() == false) {
                delSelected(label);
            }
            return true;
        }
        return false;
    }

//*************************************
//************* REINITIALIZATION*********
//*************************************
    private void NewGenese() {
        Genese();
        resetCB();
        bMigration = true;
        repaint();
    }

    private void NewGeneration() {
        nGeneration++;
        bMigration = false;
        zygote Zygote;
```

```
        morph father, mother;
        sDNA0 = "sDNA0 NewGen";
        sDNA1 = "tDNA1 NewGen";

//Genome  of the new father
        switch (selected[0]) {
            //If father was selected,
            case FATHER:
                father = Parent[0];
                sDNA0 = txtParentGene[0].getText();
                //System.out.println(sDNA0);
                //System.out.println(" FF NewGen");
                break;
            //if mother was selected
            case MOTHER:
                father = Parent[1];
                sDNA0 = txtParentGene[1].getText();
                break;
            //if a child was selected
            default:
                father = Morph[selected[0] - 1];

                sDNA0 = txtGene[selected[0] - 1].getText();
                //System.out.println(" GG sDNA0" +sDNA0);
                break;
        }
        System.out.println(" GG sDNA0" + sDNA0);

//The genome of the new mother -if she is required
        if (nbSelected > 1) {
            switch (selected[1]) {
                case FATHER:
                    mother = Parent[0];
                    sDNA1 = txtParentGene[0].getText();
                    break;
                case MOTHER:
                    mother = Parent[1];
                    sDNA1 = txtParentGene[1].getText();
                    break;
                default:
                    mother = Morph[selected[1] - 1];
                    sDNA1 = txtGene[selected[1] - 1].getText();
                    //System.out.println(" GG sDNA1" +sDNA1);
                    break;
            }
        } else {
```

```
            mother = father;
            sDNA1 = sDNA0;
        }
        System.out.println(" GG sDNA1" + sDNA1);

//System.out.println(sDNA0 + " sDNA0");
//System.out.println(sDNA1 + " sDNA1");
//Morph  (phenotype) of father
        Parent[0] = new morph(father.Genome, TypeTranscription,
                NBVAR, OptimumOrder);
        if (Parent[0].Isvalid() == false) {
            Parent[0] = new morph(sDNA0, TypeTranscription,
                    NBVAR, OptimumOrder);
        }

//Morph of mother
        Parent[1] = new morph(mother.Genome, TypeTranscription,
                NBVAR, OptimumOrder);

        if (Parent[1].Isvalid() == false) {

            Parent[1] = new morph(sDNA1, TypeTranscription,
                    NBVAR, OptimumOrder);
        }

//Genomes and morphs (phenotypes) of Children
        for (int i = 0; i < NBMORPHS; i++) {
            Zygote = new zygote(Parent[0].Genome, Parent[1].Genome);
            Zygote.crossing_over(rnd, NbCrossover);
            Morph[i] = new morph(Zygote.getChild(rnd),
                    TypeTranscription, NBVAR, OptimumOrder);
            Morph[i].mutate(MaxSeverity, rnd);
        }
        resetCB();

//System.out.println(sDNA0 + " father  NewGen");
//System.out.println(sDNA1+ " mother  NewGen" );
        newGeneration = true;
    }

//*************************************
//*************Subordinated methods************
//*************************************
//Transforms numbers to colors
    private Color fixColor(int selection) {
        Color c = new Color(0);
```

```
        switch (selection) {
            case 0:
                c = Color.black;
                break;
            case 1:
                c = Color.blue;
                break;
            case 2:
                c = Color.cyan;
                break;
            case 3:
                c = Color.darkGray;
                break;
            case 4:
                c = Color.gray;
                break;
            case 5:
                c = Color.green;
                break;
            case 6:
                c = Color.lightGray;
                break;
            case 7:
                c = Color.magenta;
                break;
            case 8:
                c = Color.orange;
                break;
            case 9:
                c = Color.pink;
                break;
            case 10:
                c = Color.red;
                break;
            case 11:
                c = Color.white;
                break;
            case 12:
                c = Color.yellow;
                break;
        }
        return c;
    }

//'Show genome' vs 'hide genome'
    private void flipShowGenome() //OK
```

```
    {
        bDisplayTxtGene = !bDisplayTxtGene;
        if (bDisplayTxtGene) {
            btnShowGenome.setLabel("Hide Genome");
        } else {
            btnShowGenome.setLabel("Show Genome");
        }

        /*Debugging instruction
  * int j=3;
 System.out.println(txtGene[j]+ " flipflop");
         */
        for (int i = 0; i < NBMORPHS; i++) {
            txtGene[i].setVisible(bDisplayTxtGene);
        }
        //System.out.println(txtGene[j]+ "flipflop");
        if (nGeneration > 0) {
            txtParentGene[0].setVisible(bDisplayTxtGene);
            txtParentGene[1].setVisible(bDisplayTxtGene);
        }
    }

//CB = checkBox
    private void resetCB() {
        selected[0] = 0;
        selected[1] = 0;
        nbSelected = 0;
        for (int i = 0; i < NBMORPHS; i++) {
            cb[i].setState(false);
        }
        cbParent[0].setState(false);
        cbParent[0].setVisible(nGeneration != 0);
        cbParent[1].setState(false);
        cbParent[1].setVisible(nGeneration != 0);
        dispOperation();
    }

    private void dispTranscription() {
        switch (TypeTranscription) {
            case 0:
                lblTranscription.setText(
                        "Transcription = random order");
                break;
            case 1:
                lblTranscription.setText(
                        "Transcription = forced order");
```

```
                break;
        }

        if (nGeneration != 0) {
            for (int i = 0; i < NBMORPHS; i++) {
                Morph[i].reTranscript(TypeTranscription, NBVAR,
                        OptimumOrder);
            }
            Parent[0].reTranscript(TypeTranscription, NBVAR,
                    OptimumOrder);
            Parent[1].reTranscript(TypeTranscription, NBVAR,
                    OptimumOrder);
        }

    }

//The type of action of the select button is decided
    private void dispOperation() {
        switch (nbSelected) {
            case 0:
                btnGo.setEnabled(false);
                btnGo.setLabel("SELECT");
                break;
            case 1:
                btnGo.setEnabled(true);
                btnGo.setLabel("ASEXUAL REPRODUCTION");
                break;
            case 2:
                btnGo.setEnabled(true);
                btnGo.setLabel("SEXUAL REPRODUCTION");
                break;
        }
    }

//The User can play with clicking
//over  the 8 checkboxes
    private void addSelected(String s) {
        Integer i;
        if (s.equals(sParent[0])) {
            i = FATHER;
        } else if (s.equals(sParent[1])) {
            i = MOTHER;
        } else {
            i = new Integer(s);
        }
        if (nbSelected == 0) {
```

```
            selected[0] = i;
            //increment the nbSelected variable by 1
            nbSelected++;
        } else if (nbSelected == 1) {
            selected[1] = i;
            nbSelected++;
        } else if (nbSelected > 1) {
            switch (selected[0]) {
                case FATHER:
                    cbParent[0].setState(false);
                    break;
                case MOTHER:
                    cbParent[1].setState(false);
                    break;
                default:
                    cb[selected[0] - 1].setState(false);
                    break;
            }
            selected[0] = selected[1];
            selected[1] = i;
        }
        //System.out.println(selected[0]  );
        //System.out.println(selected[1]  );
        //System.out.println(nbSelected);
        dispOperation();
    }

//If a third checkbox is clicked upon
//the first one must be deleted.
//Double clicking corresponds to deleting
    private void delSelected(String s) {
        Integer i;
        if (s.equals(sParent[0])) {
            i = FATHER;
        } else if (s.equals(sParent[1])) {
            i = MOTHER;
        } else {
            i = new Integer(s);
        }

        if (selected[0] == i) {
            selected[0] = selected[1];
            //decrement the nbSelected variable by 1
            nbSelected--;
        } else if (selected[1] == i) {
            switch (i) {
```

```
                case FATHER:
                    cbParent[0].setState(false);
                    break;
                case MOTHER:
                    cbParent[0].setState(false);
                    break;
                default:
                    cb[selected[0]].setState(false);
                    break;
            }
            nbSelected--;
        }
        selected[1] = 0;
        dispOperation();
    }

//*************************************
//************** CONTROL SERVICE**********
//*************************************
//************** Swing based code*********
//*************************************
//Choices of the Control button
    public void choices() {
// Allowed choices
        JRadioButton choice1, choice2, choice3, choice4,
                choice5, choice6, choice7;
        choice1 = new JRadioButton("Frame Color");
        choice1.setActionCommand("frameColor");
        choice2 = new JRadioButton("Background Color");
        choice2.setActionCommand("backgroundColor");
        choice3 = new JRadioButton("Morphs Color");
        choice3.setActionCommand("morphsColor");
        choice4 = new JRadioButton("Richness");
        choice4.setActionCommand("richness");
        choice5 = new JRadioButton("Transcription Order");
        choice5.setActionCommand("transcriptionOrder");
        choice6 = new JRadioButton("Max Mutation rate");
        choice6.setActionCommand("maxMutationrate");
        choice7 = new JRadioButton("FAQs");
        choice7.setActionCommand("faqs");

// A group, to ensure a single choice.
        final ButtonGroup group = new ButtonGroup();
        group.add(choice1);
        group.add(choice2);
        group.add(choice3);
```

```
        group.add(choice4);
        group.add(choice5);
        group.add(choice6);
        group.add(choice7);

//  ActionListener: reacts to User inputs
        class VoteActionListener implements ActionListener {

            @Override
            public void actionPerformed(ActionEvent ev) {
                String orderChoice = group.getSelection().
                        getActionCommand();

//System.out.println("mutationChoise : " + choice);
                if ("frameColor".equals(orderChoice)) {
                    colorFrame();
                }

                if ("morphsColor".equals(orderChoice)) {
                    colorBioMorphs();
                }

                if ("backgroundColor".equals(orderChoice)) {
                    colorBackGround();
                }

                if ("richness".equals(orderChoice)) {
                    richness();
                }

                if ("transcriptionOrder".equals(orderChoice)) {
                    orderFrame();
                    dispTranscription();
                }

                if ("maxMutationrate".equals(orderChoice)) {
                    mutationFrame();
                }

                if ("faqs".equals(orderChoice)) {
                    faqs();
                }

            }
        }
```

```
// Add listeners to each button
        ActionListener alisten = new VoteActionListener();
        choice1.addActionListener(alisten);
        choice2.addActionListener(alisten);
        choice3.addActionListener(alisten);
        choice4.addActionListener(alisten);
        choice5.addActionListener(alisten);
        choice6.addActionListener(alisten);
        choice7.addActionListener(alisten);

// The frame
        JFrame frame = new JFrame();

        frame.setTitle("Controls");
//frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = frame.getContentPane();
        c.setLayout(new GridLayout(0, 1));
        c.add(new JLabel("Controls"));

        c.add(choice1);
        c.add(choice2);
        c.add(choice3);
        c.add(choice4);
        c.add(choice5);
        c.add(choice6);
        c.add(choice7);

        frame.pack();
        frame.setVisible(true);

    }

//*****************************
    public class colorFrame extends JFrame {

        private static final long serialVersionUID = 1L;

        public colorFrame() {
            super("Frame Color");
            super.setSize(200, 110);

            final Container contentPane = super.getContentPane();
            final JButton go = new JButton("Show FrameColorChooser");
            contentPane.setBackground(frameColor);

            go.addActionListener((ActionEvent e) -> {
```

```
                Color c;
                c = JColorChooser.showDialog(
                        ((Component) e.getSource()).getParent(),
                        "FrameColorChooser", Color.gray);
                contentPane.setBackground(c);
                frameColor1 = c;
                changedf = true;
            });

            contentPane.add(go, BorderLayout.SOUTH);
//setDefaultCloseOperation(EXIT_ON_CLOSE);
        }
    }

    public void colorFrame() {
        colorFrame cp = new colorFrame();
        cp.setVisible(true);
        repaint();
    }

//*****************************************
    public class colorBackGround extends JFrame {

        private static final long serialVersionUID = 1L;

        public colorBackGround() {
            super("BackGround Color");
            super.setSize(220, 100);
            final Container contentPane = super.getContentPane();
            final JButton go
                    = new JButton("Show BackGroundColorChooser");
            contentPane.setBackground(backGround);
            go.addActionListener((ActionEvent e) -> {
                Color c;
                c = JColorChooser.showDialog(
                        ((Component) e.getSource()).getParent(),
                        "BackGroundColorChooser", Color.yellow);
                contentPane.setBackground(c);
                backGround1 = c;
                changedb = true;
            });

            contentPane.add(go, BorderLayout.SOUTH);
            //setDefaultCloseOperation(EXIT_ON_CLOSE);
        }
    }
```

```
    public void colorBackGround() {
        colorBackGround cp = new colorBackGround();
        cp.setVisible(true);
        repaint();
    }

//*************************************
    public class colorBioMorphs extends JFrame {

        private static final long serialVersionUID = 1L;

        public colorBioMorphs() {
            super("BioMorphs Color");

            super.setSize(190, 125);
            final Container contentPane = super.getContentPane();
            final JButton go
                    = new JButton("Show BioMorphsColorChooser");
            contentPane.setBackground(foreGround);
            go.addActionListener((ActionEvent e) -> {
                Color c;

                c = JColorChooser.showDialog(
                        ((Component) e.getSource()).getParent(),
                        "BioMorphsColorChooser", Color.yellow);
                contentPane.setBackground(c);

                foreGround1 = c;
                changedo = true;
            });

            contentPane.add(go, BorderLayout.SOUTH);
//setDefaultCloseOperation(EXIT_ON_CLOSE);
        }

    }

    public void colorBioMorphs() {
        colorBioMorphs cp = new colorBioMorphs();
        cp.setVisible(true);
        repaint();
    }

//*************************************
//Level of richness of morphs
```

```
    public void richness() {
// Allowed choices
        JRadioButton choice1, choice2, choice3, choice4, choice5;
        choice1 = new JRadioButton("4");
        choice1.setActionCommand("four");
        choice2 = new JRadioButton("5");
        choice2.setActionCommand("five");
        choice3 = new JRadioButton("6");
        choice3.setActionCommand("six");
        choice4 = new JRadioButton("7");
        choice4.setActionCommand("seven");
        choice5 = new JRadioButton("8");
        choice5.setActionCommand("eight");

// A group, to ensure a single choice.
        final ButtonGroup group = new ButtonGroup();
        group.add(choice1);
        group.add(choice2);
        group.add(choice3);
        group.add(choice4);
        group.add(choice5);

//  ActionListener: reacts to User inputs
        class VoteActionListener implements ActionListener {

            @Override
            public void actionPerformed(ActionEvent ev) {
                String orderChoice = group.getSelection().getActionComma

                if ("four".equals(orderChoice)) {
                    OptimumOrder = 4;
                }
                if ("five".equals(orderChoice)) {
                    OptimumOrder = 5;
                }
                if ("six".equals(orderChoice)) {
                    OptimumOrder = 6;
                }
                if ("seven".equals(orderChoice)) {
                    OptimumOrder = 7;
                }
                if ("eight".equals(orderChoice)) {
                    OptimumOrder = 8;
                }

            }
```

```
        }

// Adds listeners to each button
        ActionListener alisten = new VoteActionListener();
        choice1.addActionListener(alisten);
        choice2.addActionListener(alisten);
        choice3.addActionListener(alisten);
        choice4.addActionListener(alisten);
        choice5.addActionListener(alisten);

// The frame
        JFrame frame = new JFrame();

        frame.setTitle("Richness");
//frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = frame.getContentPane();
        c.setLayout(new GridLayout(1, 1));
        c.add(new JLabel("Richness"));

        c.add(choice1);
        c.add(choice2);
        c.add(choice3);
        c.add(choice4);
        c.add(choice5);

        frame.pack();
        frame.setVisible(true);

    }

//**************************************
//Order of transcription
    public void orderFrame() {
// Allowed choices
        JRadioButton choice1, choice2;
        choice1 = new JRadioButton("Random Order ");
        choice1.setActionCommand("randomOrder");
        choice2 = new JRadioButton("Forced Order ");
        choice2.setActionCommand("forcedOrder");

// A group, to ensure a single choice.
        final ButtonGroup group = new ButtonGroup();
        group.add(choice1);
        group.add(choice2);

//  ActionListener: reacts to User inputs
```

```
        class VoteActionListener implements ActionListener {

            @Override
            public void actionPerformed(ActionEvent ev) {
                String orderChoice
                        = group.getSelection().getActionCommand();

                if ("randomOrder".equals(orderChoice)) {
                    lblTranscription.setText("Transcription = random orc
                    TypeTranscription = 0;
                }

                if ("forcedOrder".equals(orderChoice)) {
                    lblTranscription.setText("Transcription = forced orc
                    TypeTranscription = 1;
                }
            }
        }

//Adds listeners to each button
        ActionListener alisten = new VoteActionListener();
        choice1.addActionListener(alisten);
        choice2.addActionListener(alisten);

//The frame
        JFrame frame = new JFrame();

        frame.setTitle("Transcription order");
//frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = frame.getContentPane();
        c.setLayout(new GridLayout(1, 1));
        c.add(new JLabel("Transcription"));

        c.add(choice1);
        c.add(choice2);

        frame.pack();
        frame.setVisible(true);

    }

//*************************************
//Max level of mutation
    public void mutationFrame() {
// Allowed choices
        JRadioButton choice1, choice2, choice3, choice4;
```

```
        choice1 = new JRadioButton("Low");
        choice1.setActionCommand("low");
        choice2 = new JRadioButton("Medium");
        choice2.setActionCommand("medium");
        choice3 = new JRadioButton("High");
        choice3.setActionCommand("high");
        choice4 = new JRadioButton("Hot");
        choice4.setActionCommand("hot");

//A group, to ensure a single choice.
        final ButtonGroup group = new ButtonGroup();
        group.add(choice1);
        group.add(choice2);
        group.add(choice3);
        group.add(choice4);

//  ActionListener: reacts to  input  of the User
        class VoteActionListener implements ActionListener {

            @Override
            public void actionPerformed(ActionEvent ev) {
                String mutationChoice
                        = group.getSelection().getActionCommand();

                if ("low".equals(mutationChoice)) {
                    lblMutation.setText("Mutation = low");
                    MaxSeverity = 0;
                }
                if ("medium".equals(mutationChoice)) {
                    lblMutation.setText("Mutation = medium");
                    MaxSeverity = 1;
                }
                if ("high".equals(mutationChoice)) {
                    lblMutation.setText("Mutation = high");
                    MaxSeverity = 2;
                }
                if ("hot".equals(mutationChoice)) {
                    lblMutation.setText("Mutation = hot");
                    MaxSeverity = 3;
                }
            }

        }

// Add listeners to each button
        ActionListener alisten = new VoteActionListener();
```

```
        choice1.addActionListener(alisten);
        choice2.addActionListener(alisten);
        choice3.addActionListener(alisten);
        choice4.addActionListener(alisten);

// The frame
        JFrame frame = new JFrame();
        frame.setTitle("Max Mutation Rate");
//frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = frame.getContentPane();
        c.setLayout(new GridLayout(0, 1));
        c.add(new JLabel("Max Mutation rate"));
        c.add(choice1);
        c.add(choice2);
        c.add(choice3);
        c.add(choice4);
        frame.pack();
        frame.setVisible(true);
    }

//*****************************************
//Displays FAQS
    public class faqs extends JFrame {

        private static final long serialVersionUID = 1L;
        private JTextArea textArea1;

        public faqs() {
            Box box = Box.createHorizontalBox(); // create box
            String s1 = "How can I stop proliferating Applets?\n"
                    + "After using a Control service, \n"
                    + "close the corresponding applet (X or alt+F4)\n"
                    + "and also close the Control applet (X or alt+F4).\
                    + "\n"
                    + "What is this?\n"
                    + "This is a pattern generator whose mechanism \n"
                    + "imitates the evolutionary process. \n"
                    + "The next generation is produced by mutation \n"
                    + "+ optional recombination of the genomes of \n"
                    + "the present one + spontaneous generation.\n"
                    + "Natural selection is simulated by the User  \n"
                    + "that selects ONE genome to be cloned, \n"
                    + "else TWO  individuals,  whose genomes \n"
                    + "must be recombined to \n"
                    + "produce the new generation.\n"
                    + "There is also a purifying selection to keep \n"
```

```
                    + "genomes of the same length (8 letters).\n"
                    + "It is possible to set a new beginning\n"
                    + "by clicking on the MUTATE button."
                    + "\n \n"
                    + "What was this application built for? \n"
                    + "It was designed to show the potentialities  \n"
                    + "of evolution to generate patterns that somehow  \n"
                    + "resemble alive beings, say, insects.\n \n"
                    + "Who did make this? \n"
                    + "The original idea was of Richard Dawkins , \n"
                    + "in his famous book The 'Blind Watchmaker' (1986). \n"
                    + "We follow a Java implementation made"
                    + "by Alain Gogniat in 1997.  \n"
                    + "Does this program show that life arouse by evolution?\n
                    + "No: produced patterns are not alive,  \n"
                    + "say, they do not feel the environment anyhow. \n \n"
                    + "Does this program show that species arouse by evolution
                    + "\n"
                    + "No,  the program contains a purifying selection  \n"
                    + "that has just one purpose:  \n"
                    + "to preserve the identity of the species. \n"
                    + "In that regard, the code is anti-evolutionary.\n \n"
                    + "What does this program shows? \n"
                    + "This program shows that  \n"
                    + "Java for the study of evolution \n"
                    + "could be a brilliant profession.\n \n "
                    + "Can I use this application for my own profit and sake?\
                    + "Yes, you can and it is completely free.";

            //TextAreaFrame constructor. Create textarea1
            textArea1 = new JTextArea(s1, Font.PLAIN, 14);

            box.add(new JScrollPane(textArea1)); // add scrollpane
            super.add(box);
            textArea1.setFont(new Font("TimesRoman",
                    Font.PLAIN, 14));
        }
    }

    private void faqs() {
        faqs textAreaFrame = new faqs();
        textAreaFrame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
        textAreaFrame.setSize(460, 200); // set frame size
        textAreaFrame.setVisible(true); // display frame
    }
```

```
// ************************************************
//
//  CLASS morph:from genome to phenotype
//
// ************************************************
    class morph {

        private int nDirection = 0;
        private int nOrder;

        private int MaxY;

        public int dx[], dy[];
        private List<Dimension> FromPt = new ArrayList<>();
        private List<Dimension> ToPt = new ArrayList<>();
        ;
 public genome Genome;
        private int gene[];
        private boolean computed;
        private int transcription;

//builds local variables
//Overloaded constructor
//(various forms of definition).
        public morph(genome Gen, int Transcript,
                int NBVAR, int OptimumOrder) {
            Genome = Gen;
            gene = Genome.chromosome;
            dx = new int[NBVAR];
            dy = new int[NBVAR];

            MaxY = 0;
            transcription = Transcript;
            computed = false;
        }

//builds local variables
        public morph(String S, int Transcript, int NBVAR,
                int OptimumOrder) {
            Genome = new genome(S);
            gene = Genome.chromosome;
            dx = new int[NBVAR];
            dy = new int[NBVAR];

            MaxY = 0;
```

```
            transcription = Transcript;
            computed = false;
            //System.out.println(S + " morph");
        }

//defines when a genome is valid
        public boolean Isvalid() {
            return Genome.Isvalid();
        }

//calls mutation
        public void mutate(int severity, Random r) {
            Genome.mutate(severity, r);
            computed = false;
        }

//conditions transcript
        public void reTranscript(int Transcript, int NBVAR,
                int OptimumOrder) {
            computed = false;
            transcription = Transcript;
            transcript(NBVAR, OptimumOrder);
        }

//Converts numbers  to string
        @Override
        public String toString() {
            return Genome.toString();
        }

//From genes to geometry
        private void transcript(int NBVAR, int OptimumOrder) {
            dx[0] = -gene[1] / 4;
            dx[1] = -gene[0] / 4;
            dx[2] = gene[3] / 2;
            dx[3] = gene[0] / 2;
            dx[4] = gene[1] / 2;
            dx[5] = gene[2] / 2;
            dx[6] = -dx[2] / 2;
            dx[7] = -dx[5] / 2;

            dy[0] = (gene[5]) / 2;
            dy[1] = (gene[4]);
            dy[2] = (gene[3]);
            dy[3] = -(gene[4]);
            dy[4] = -(gene[5]);
```

```
            dy[5] = (gene[6]);
            dy[6] = (gene[0]);
            dy[7] = -dy[5];

            if (transcription == 1) {
                gene[8] = OptimumOrder;
            } else {
                //takes the absolute value of gene[8]:
                //If gene[8] < 0 is true, gene[8]= -gene[8]
                //else gene[8]=gene[8]
                gene[8] = (gene[8] < 0) ? -gene[8] : gene[8];
                //gene[8] = max(3,gene[8]):
                //If gene[8] < 3 is true, gene[8]= 3]
                //else gene[8]=gene[8]
                gene[8] = (gene[8] < 3) ? 3 : gene[8];
            }

            nOrder = gene[8];
            //builds a tree
            tree(0, 0, nOrder, nDirection, NBVAR);
        }

//Draws the branches at (x,y)
//A morph is a tree, whose branches are bent.
        public void phenotype(Graphics g, int x, int y,
                int NBVAR, int OptimumOrder) {
            if (computed == false) {
                transcript(NBVAR, OptimumOrder);
            }
//Dimension = couple of numbers to denote
//the dimensions of a rectangle, width and height,
//or the coordinates of a vector in the plane.
            Dimension Pt1, Pt2;
//Draws two branches at (x,y)
            for (int i = 0; i < FromPt.size(); i++) {
                Pt1 = (Dimension) FromPt.get(i);
                Pt2 = (Dimension) ToPt.get(i);
                //draw  two segments of a symmetric morph
                g.drawLine(x + Pt1.width, y - MaxY + Pt1.height,
                        x + Pt2.width,
                        y - MaxY + Pt2.height);
                g.drawLine(x - Pt1.width, y - MaxY + Pt1.height,
                        x - Pt2.width,
                        y - MaxY + Pt2.height);
            }
        }
```

```
        private void PushGene(int x1, int y1, int x2, int y2) {
            //calculates the maximum value of all y's
            if (y1 > MaxY) {
                MaxY = y1;
            } else if (y2 > MaxY) {
                MaxY = y2;
            }

            //Adds (x1,y1) to vector FromPt
            Dimension dimension1 = new Dimension(x1, y1);
            dimension1.height = x1;
            dimension1.width = y1;
            FromPt.add(new Dimension(x1, y1));

            //Adds (x2,y2) to vector toPt
            dimension1.height = x2;
            dimension1.width = y2;
            ToPt.add(new Dimension(x2, y2));

        }

//Construction of a symmetric tree
//branch by branch
        private void tree(int x, int y, int order,
                int dir, int NBVAR) {
            int Xnew, Ynew;
            if (dir < 0) {
                //increment of dir = NBVAR
                //dir = dir+NBVAR;
                dir += NBVAR;
            } else if (dir >= NBVAR) {
                //decrement of dir = NBVAR
                //dir = dir-NBVAR;
                dir -= NBVAR;
            }
            Xnew = x + order * dx[dir];
            Ynew = y + order * dy[dir];
            if (Ynew > 100) {
                Ynew = 100;
            }
            PushGene(x, y, Xnew, Ynew);
            if (order > 0) {
                tree(Xnew, Ynew, order - 1, dir - 1, NBVAR);
                tree(Xnew, Ynew, order - 1, dir + 1, NBVAR);
            }
```

```
            computed = true;
        }
    }
}

// ************************************************
//
//  CLASS genome: fills the chromosome
//
// ************************************************
class genome {

    int NBGENE = 9;
    int MAXLEN = 7;
    public int chromosome[];

    private final boolean valid;

    public genome(Random r) {
        chromosome = new int[NBGENE];
        GetVal(r);
        valid = true;
    }

    public genome(int c[]) {
        chromosome = new int[NBGENE];
        //c is copied into chromosome
        System.arraycopy(c, 0, chromosome, 0, NBGENE);
        valid = true;
    }

    public genome(String S) {
        Random rr = new Random();
        chromosome = new int[NBGENE];

        valid = S.trim().length() == NBGENE;
        if (valid) {
            char c[];
            c = S.toUpperCase().trim().toCharArray();
            for (int i = 0; i < NBGENE - 1; i++) {
                chromosome[i] = ((int) c[i] - 71) % MAXLEN;
            }
            chromosome[NBGENE - 1]
                    = ((int) c[NBGENE - 1] - 71) % (MAXLEN + 1);
            if (chromosome[NBGENE - 1] < 3) {
                chromosome[NBGENE - 1] = 3;
```

```
            }
        } else {
            GetVal(rr);
        }
    }

    public boolean Isvalid() {
        //System.out.println(valid +    " Isvalid ");

        return valid;

    }

//Mutations are done at vector chromosome
    public void mutate(int maxSeverity, Random r) {
        if (maxSeverity > 0) {
            //p is a random positive integer less than NBGENE
            int p = r.nextInt() % (NBGENE);
            p = (p < 0) ? -p : p;

            //severity is another random integer
            int severity = r.nextInt();

            //mute is an integer, whose absolute value
            // is less or equal than maxSeverity
            int mute = severity % maxSeverity + 1;
            System.out.println("mute = " + mute);
            //mutation is done
            //chromosome[p] =chromosome[p] + mute;
            chromosome[p] += mute;
            if (p < (NBGENE - 1)) {
                chromosome[p] %= (MAXLEN);
            } else {
                chromosome[p] %= (MAXLEN + 1);
            }
            System.out.println("chromosome[p] = " + chromosome[p]);
        }

    }

//vector chromosome is filled in
//random numbers less than MAXLEN
    private void GetVal(Random r) {

        for (int i = 0; i < NBGENE - 1; i++) {
            chromosome[i] = r.nextInt() % MAXLEN;
```

```
        }
        chromosome[NBGENE - 1] = r.nextInt() % (MAXLEN + 1);
    }

    @Override
    public String toString() {
        String S0;
        char c[];
        c = new char[NBGENE];
        for (int i = 0; i < NBGENE; i++) {
            c[i] = ((char) (chromosome[i] + 71));
        }
        S0 = " " + Arrays.toString(c);
        //System.out.println(S0 +" toString");
        return S0;
    }
}

// ************************************************
//
//  CLASS zygote: two parents produce  a child
//
// ************************************************
class zygote {

    private final genome male, female;

    public zygote(genome mG, genome fG) {
        male = new genome(mG.chromosome);
        female = new genome(fG.chromosome);
    }

    public genome getMale() {
        return male;
    }

    public genome getFemale() {
        return female;
    }

    public genome getChild(Random r) {
        int p;
        p = r.nextInt();
        if ((p) < 0) {
            return female;
        } else {
```

```
            return male;
        }
    }

    public void crossing_over(Random r, int nCross) {
        genome genomeMale = male;
        int lenGenome = genomeMale.NBGENE;

        int i, p, k;
        int temp;

        for (k = 0; k < nCross; k++) {
            p = r.nextInt() % lenGenome;
            p = p < 0 ? -p : p;
            for (i = p; i < lenGenome; i++) {
                temp = male.chromosome[i];
                male.chromosome[i] = female.chromosome[i];
                female.chromosome[i] = temp;
            }
        }
    }

}//End of zygote

//End of Program C5, Biomorphs
```

6 ***Exercise*** *Run the code, play with it and try to understand it. To avoid typing the code, import the corresponding zip file with all programs of this volume. NetBeans 8 and Java 8 are needed.*

7 ***Exercise*** *Study the FAQs displayed by the program. Agree or disagree with the answers. If you disagree, modify the program to fill in your own ideas. Remember that you enjoy an unconditional license to use and misuse all our software: our interest is to facilitate the formation of a strong evolJava community.*

1.3 Menus

A **menu** is a graphic form of allowing the election of one among various command options. For a GUI, a menu is frequently associated with the upper bar of the main window.

8 ***A code with menus***

In the next program we can see a GUI that shows the form as **menus** operate.

```
//Program C8, MenuShow
//Introduction to SWING menus
package ejvol3v2p;

import java.awt.Color;

import java.awt.Container;
import java.awt.Font;
import java.awt.BorderLayout;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;

import javax.swing.JColorChooser;
import javax.swing.JFrame;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JLabel;

import javax.swing.ButtonGroup;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JMenuBar;

public class MenuShow extends JFrame {

    private static final long serialVersionUID = 1L;
    public Color colorF = Color.gray;
    private final Color COLORVALUES[]
            = {Color.BLACK, Color.YELLOW, Color.MAGENTA,
                Color.GREEN};

    private final JRadioButtonMenuItem COLORITEMS[];
    private final JRadioButtonMenuItem FONTS[];
    //font style menu items
    private final JCheckBoxMenuItem STYLEITEMS[];
    //displays a text
    private final JLabel DISPLAYJLABEL;
    private final ButtonGroup FONTBUTTONGROUP;
    private final ButtonGroup COLORBUTTONGROUP;
    //Style of font
    private int style;
```

```
public MenuShow() {
    //Title
    super("menuShow");
    //Background color
    super.getContentPane().setBackground(colorF);
    //Initialization
    DISPLAYJLABEL = new JLabel("Lindenmayer Systems");
    DISPLAYJLABEL.setForeground(Color.BLACK);
    DISPLAYJLABEL.setFont(new Font("SanSerif", Font.PLAIN, 12));
    super.add(DISPLAYJLABEL, BorderLayout.CENTER);

    //*****************************************
    //The first column of  the menu is worked out
    //Creates file menu
    JMenu fileMenu = new JMenu("File");
    //File menu is enabled  with Alt+ F
    fileMenu.setMnemonic('F');

    //The first menu item is added to the first column
    JMenuItem aboutItem = new JMenuItem("Purpose...");
    //If FILE IS enabled, About is open with Alt + A
    aboutItem.setMnemonic('A');
    //Adds about item to first column of the menu
    fileMenu.add(aboutItem);
    aboutItem.addActionListener(//Strucutre to answer to the User
            (ActionEvent event) -> {
                JOptionPane.showMessageDialog(MenuShow.this,
                        "We learn how to implement a menu bar. \n"
                        + "In a next program,"
                        + " we will use a turtle to implement "
                        + "\nLindenmayer systems.",
                        "About", JOptionPane.PLAIN_MESSAGE
                );
            } //Displays message dialog
    );

    //The second menu item is added to the first column
    JMenuItem saveItem = new JMenuItem("Save");
    saveItem.setMnemonic('x');
    fileMenu.add(saveItem);
    saveItem.addActionListener((ActionEvent event) -> {
        JOptionPane.showMessageDialog(MenuShow.this,
                "This option could be used to "
                + "save \na graphic.",
                "Save", JOptionPane.PLAIN_MESSAGE
        );
```

```
        } // Displays message dialog
        );

        //The third menu item is added to the first column
        JMenuItem exitItem = new JMenuItem("Exit");
        exitItem.setMnemonic('x');
        fileMenu.add(exitItem);
        exitItem.addActionListener((ActionEvent event) -> {
            System.exit(0);
        } // terminates application
        );

        /* ***************************************
      * A new  column is added to the menu
         */
        JMenu demoMenu = new JMenu("Demo");
        demoMenu.setMnemonic('D');

        //The first menu item is added to the Demo menu
        JMenuItem KochItem = new JMenuItem("Koch...");
        KochItem.setMnemonic('K');
        demoMenu.add(KochItem);
        KochItem.addActionListener((ActionEvent event) -> {
            JOptionPane.showMessageDialog(MenuShow.this,
                    "Koch system \nLindenmayer systems",
                    "Koch", JOptionPane.PLAIN_MESSAGE
            );
        });

        //The second menu item is added to the Demo menu
        JMenuItem HilbertItem = new JMenuItem("Hilbert");
        HilbertItem.setMnemonic('z');
        demoMenu.add(HilbertItem);
        HilbertItem.addActionListener((ActionEvent event) -> {
            JOptionPane.showMessageDialog(MenuShow.this,
                    "Hilbert system \nLindenmayer systems",
                    "Hilbert", JOptionPane.PLAIN_MESSAGE
            );
        });

        /* ***************************************
      * A new  column is added to the menu
         */
        JMenu evolutionMenu = new JMenu("Evolution");
        demoMenu.setMnemonic('D');
```

```
  //The first menu item is added
  JMenuItem eItem = new JMenuItem("E...");
  eItem.setMnemonic('K');
  evolutionMenu.add(eItem);
  eItem.addActionListener((ActionEvent event) -> {
      JOptionPane.showMessageDialog(MenuShow.this,
              "E?",
              "E", JOptionPane.PLAIN_MESSAGE
      );
  } //Displays message dialog
  );

  //The second menu item is added
  JMenuItem hItem = new JMenuItem("H");
  hItem.setMnemonic('h');
  evolutionMenu.add(hItem);
  hItem.addActionListener((ActionEvent event) -> {
      JOptionPane.showMessageDialog(MenuShow.this,
              "H  \n?",
              "H ", JOptionPane.PLAIN_MESSAGE
      );
  });

  /* ****************************************
* A new  column will be added to the menu,
   */
  //Hierarchy and names of menus and submenus:
  //fathers and sons are Jemnu
  //grandsons are JmenuItem:
  //artMenu, flowersMenu and branchesMenu are Jmenu
  //colorFlower and nameFlower are JmenuItem
  JMenu artMenu = new JMenu("Art");
  artMenu.setMnemonic('t');

  //Creates Flowers menu
  JMenu flowersMenu = new JMenu("Flowers");
  flowersMenu.setMnemonic('O');

  final JMenuItem colorFlower
          = new JMenuItem("ColorFlower");
  //colorFlower.setMnemonic( 'l' );
  flowersMenu.add(colorFlower);
  super.getContentPane().setBackground(colorF);
  colorFlower.addActionListener((ActionEvent event) -> {
      Color c = colorF;
      final Container contentPane = getContentPane();
```

```
        c = JColorChooser.showDialog(
                colorFlower, getName(), c
        );
        contentPane.setBackground(c);
        colorF = c;
    });

    JMenuItem nameFlower = new JMenuItem("Name");
    nameFlower.setMnemonic('e');
    flowersMenu.add(nameFlower);

    nameFlower.addActionListener((ActionEvent event) -> {
        JOptionPane.showMessageDialog(MenuShow.this,
                "Every flower deserves a name",
                "name", JOptionPane.PLAIN_MESSAGE
        );
    });

    //flowersMenu is added to the artMenu
    artMenu.add(flowersMenu);

    //A new submenu is added to the Art menu
    JMenu branchesMenu = new JMenu("Branches");
    branchesMenu.setMnemonic('C');

    // array listing string colors
    String colors[]
            = {"Black", "Yellow", "Red", "Green"};

    // creates radio button menu items for colors
    COLORITEMS
            = new JRadioButtonMenuItem[colors.length];
    COLORBUTTONGROUP = new ButtonGroup();

    //Menus are called and activated separately.
    //Radio Buttons work as disjunctive ensemble:
    //the activation of one deactivates the others.
    //Checkboxes can be activated and deactivated
    //separately.
    //Instantiation of an inner class to listen
    //at the action over radiobuttons.
    ItemHandler itemHandler = new ItemHandler();

    //Creates color radio button menu items
    for (int count = 0; count < colors.length; count++) {
        COLORITEMS[count]
```

```
                = new JRadioButtonMenuItem(colors[count]);
        branchesMenu.add(COLORITEMS[count]);
        COLORBUTTONGROUP.add(COLORITEMS[count]);

        //Calling of inner class itemhandler
        COLORITEMS[count]
                .addActionListener(itemHandler);
    }
    //Initialization
    COLORITEMS[0].setSelected(true);

    artMenu.add(branchesMenu);

    //Adds separator
    artMenu.addSeparator();

    //FONT control structure
    String fontNames[]
            = {"Serif", "MonoSpaced", "SansSerif"};
    JMenu fontMenu = new JMenu("Font of title ");
    fontMenu.setMnemonic('n');

    //Creates radio button menu items for font names
    FONTS = new JRadioButtonMenuItem[fontNames.length];
    FONTBUTTONGROUP = new ButtonGroup();

    //Creates Font radio button menu items
    for (int count = 0; count < FONTS.length; count++) {
        FONTS[count]
                = new JRadioButtonMenuItem(fontNames[count]);
        fontMenu.add(FONTS[count]);
        FONTBUTTONGROUP.add(FONTS[count]);
        //Calling of inner class itemhandler
        FONTS[count].addActionListener(itemHandler);
    }

    FONTS[0].setSelected(true);
    fontMenu.addSeparator();
    //Names of styles
    String styleNames[] = {"Bold", "Italic"};
    STYLEITEMS
            = new JCheckBoxMenuItem[styleNames.length];

    //Instantiation of an inner class to listen
    //at actions over checkboxes.
    StyleHandler styleHandler = new StyleHandler();
```

```
        //Creates style checkbox menu items
        for (int count = 0; count < styleNames.length; count++) {
            STYLEITEMS[count]
                    = new JCheckBoxMenuItem(styleNames[count]);
            fontMenu.add(STYLEITEMS[count]);
            //Calling of inner class
            STYLEITEMS[count].addItemListener(styleHandler);
        }
        //adds Font menu to Art menu
        artMenu.add(fontMenu);

        //The whole munu bar is assembled
        JMenuBar bar = new JMenuBar();
        super.setJMenuBar(bar);
        bar.add(fileMenu);
        bar.add(demoMenu);
        bar.add(evolutionMenu);
        bar.add(artMenu);

    } // end MenuFrame constructor

    //*****************************************************
    //*****************************************************
    //Inner class to handle action events:
    //buttons from branches and FONTS.
    private class ItemHandler implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent event) {
            // Branch color selection
            for (int i = 0; i < COLORITEMS.length; i++) {
                if (COLORITEMS[i].isSelected()) {
                    DISPLAYJLABEL
                            .setForeground(COLORVALUES[i]);
                    break;
                }
            }

            // Font style selection
            for (JRadioButtonMenuItem font : FONTS) {
                if (event.getSource() == font) {
                    DISPLAYJLABEL.setFont(new Font(font.getText(), style
                }
            }
```

```
            repaint();
        }
    }

    //************************************************
    // Inner class to handle item events
    //from check box menu items
    private class StyleHandler implements ItemListener {
        // process font style selections

        @Override
        public void itemStateChanged(ItemEvent e) {
            style = 0; // initialize style

            // check for bold selection
            if (STYLEITEMS[0].isSelected()) {
                style += Font.BOLD;
            }

            // check for italic selection
            if (STYLEITEMS[1].isSelected()) {
                style += Font.ITALIC;
            }

            DISPLAYJLABEL.setFont(
                    new Font(
                            DISPLAYJLABEL
                            .getFont().getName(), style, 12)
            );
            repaint();
        }
    }

    public static void main(String args[]) {
        MenuShow x = new MenuShow();
        x.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        x.setSize(800, 700);
        x.setVisible(true);
    }
}//End of Program C8, MenuShow
```

9 ***Observations***

1. One has the possibility to enable a shortcut activation of a menu item as in:

```
fileMenu.setMnemonic( 'F' );
```

In that case, the User presses Alt+ F and the corresponding menu gets open. A submenu does not obey such a call unless the parent menu is already activated.

2. Hierarchy and names of menus and submenus: fathers and sons are Jmenu, grandsons are JmenuItem: artMenu, flowersMenu and branchesMenu are Jmenu while colorFlower and nameFlower are JmenuItem.

3. Menus are called and activated separately. Radio Buttons work as disjunctive ensemble: the activation of one deactivates the others. Checkboxes can be activated and deactivated separately.

1.4 Saving a graphic

Now that we can produce nice graphics, we would like to have the possibility to save them. This is explained in this section.

10 ***Saving a screen Shot***

In the next code we save a rectangular region of the screen to a jpg file.

```
//Progam C10 saveScreenShot
//This class saves a  rectangular region
//of the screen of Eclipse as a jpg file
//that can be opened with Paint, Draw
//or ImagemMagick.

//To see the saved region, search the file in a
//File Manager.
//It can be opened with Paint, Draw or ImageMagick.

package ejvol3v2p;

import java.awt.AWTException;
import java.awt.Graphics;
import java.awt.Robot;
import java.awt.Rectangle;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class SaveScreenShot extends JPanel {


    @Override
```

```
    public void paint(Graphics g) {
        //The region to be saved is specified
        Rectangle screenRect = new Rectangle(0, 0, 425, 200);
        //the graphic is captures
        Robot robot;
        BufferedImage image = null;
        try {
            robot = new Robot();
            image
                    = robot.createScreenCapture(screenRect);
        } catch (AWTException e) {
        }

        String nameOfFile = "Drawing1234.jpg";
        // save captured image to jpg file
        try {
            ImageIO.write(image, "jpg", new File(nameOfFile));
            // give feedback
            if (image != null) {
                System.out.println("Saved screen shot "
                        + "(" + image.getWidth()
                        + " x " + image.getHeight() + " pixels) to file \""
                        + nameOfFile + "\".");
            }
        } catch (IOException | NullPointerException e) {
        }
    }

    public static void main(String args[])
            throws AWTException, IOException {
        // creates frame
        JFrame frame = new JFrame("Save screen shot");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Creates an instance  of saveScreenShot
        SaveScreenShot x = new SaveScreenShot();
        //Adds x to frame
        frame.add(x);
        frame.setSize(425, 200);
        //frame.setBackground( Color.BLUE);
        frame.setVisible(true);
    }//end main
}//end of Progam C10 saveScreenShot
```

11 ***Exercise*** *Run the program and and play with the code. To see the saved region, search the saved file in a File Manager. It can be opened with Paint, Draw or ImageMagick.*

12 ***Saving an image*** *The next code saves a rectangular region of a graphic that has been made by the User.*

```
//Program C12, SaveImage
//This class saves a   graphic that
//has been made by the User
//to a jpg file
//in the workspace.

package ejvol3v2p;


import java.awt.*;
import java.awt.geom.*;
import java.awt.image.BufferedImage;
import java.awt.image.RenderedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JFrame;

public class SaveImage extends JFrame {

    private static final long serialVersionUID = 1L;

    //Title and dimensions
    public SaveImage() {
        super("Saving a rectangular portion");
        super.setSize(425, 200);
        super.setVisible(true);
    }


    //Saves the image to jpg file
    private   void saveDrawing() {
        RenderedImage image = createRenderedDrawing();

        try {
            ImageIO.write((RenderedImage) image,
                    "jpg", new File("Drawing2.jpg"));
        } catch (IOException e) {
        }
        //Gives feedback
        System.out.println("Saved graphic "
                + "(" + image.getWidth()
```

```
                + " x " + image.getHeight()
                + " pixels) to file \""
                + "Drawing2" + "\".");
        System.out.println("Use a File Manager "
                + "to localize the file.");
        System.out.println("Open the file with Paint, "
                + "Draw or ImageMagick.");
    }

    //Creates the images to be saved
    private RenderedImage createRenderedDrawing() {
        //The region to be saved begins at the topleft.
        //It is rectangular with the following dimensions:
        int width = 200;
        int height = 200;

        //Creates a buffered image in which to draw
        BufferedImage image = new BufferedImage(width
                , height, BufferedImage.TYPE_INT_ARGB);
        //Creates a graphics contents on the buffered image
        Graphics2D g2d = image.createGraphics();
        //Paints the image
        drawer(g2d);
        return image;
    }

    //Creates the image to be seen
    private  void drawer(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        //Defines background
        Color pastel = new Color(0, 20, 100, 200);
        g2d.setColor(pastel);
        g2d.fill(new Rectangle2D.Double(0, 0, 425, 200));

        g2d.setPaint(Color.blue);
        g2d.fill(new Ellipse2D.Double(5, 40, 120, 100));

        g2d.setPaint(Color.cyan);
        g2d.setStroke(new BasicStroke(10.0f));
        g2d.draw(new Rectangle2D.Double(80, 30, 65, 100));

        g2d.setPaint(Color.GREEN);
        g2d.draw(new Line2D.Double(20, 30, 200, 180));
        g2d.draw(new Line2D.Double(20, 20, 20, 200));
        g2d.draw(new Line2D.Double(100, 20, 100, 200));
        g2d.draw(new Line2D.Double(150, 20, 150, 200));
```

```
    }

    //Drawing manager
    @Override
    public void paint(Graphics g) {
        //Paints the screen
        drawer(g);
        //Saves graphic
        saveDrawing();
    }

    public static void main(String args[])
            {
        //Creates an instance of SaveImage
        SaveImage x = new SaveImage();
        //Fires x
        x.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

}//End of Program C12, SaveImage
```

13 ***Exercise*** *Run the code and play with it changing the position and size of the region to be saved. Make a change to the drawing and save it.*

1.5 Lindenmayer systems

To see menus in action, let us implement a **Lindenmayer system**, which is a very nice idea that offers another proposal as biological form could be implemented by language theory, i.e., we will see in our program a connection of certain strings with forms.

L-systems (the shorthand for Lindenmayer systems) were originally proposed by the biologist Aristid Lindenmayer in 1968 as a theory for explaining the form as the morphology of plants is encoded in the genetic code. Say, it encodes the form as a the arms of a tree is encoded in a string. Central to L-systems, is the notion of recursiveness: complex objects are built by iterative replacement of parts of a simple object using a set of rewriting rules or productions (Ochoa, [55] 1998).

Our version below is offered as the smartest possible realization of these systems. It has a peculiarity: all mutations are viable. You can read directly from the program below how this has been achieved.

14 ***The turtle***

At the root of the encoding of L-systems resides a **turtle**, which is a pen to draw curves. The turtle has various variables and may be implemented in many diverse forms. We choose to elect the next description:

1. A point where the head of the turtle sits.

2. A boolean variable to say whether or not the pen is down (ready to write) or up (writing mode off).

3. A direction in which the turtle will move the next time. The length of movement is usually a constant.

4. A given symbol, say, F, says that the pen is down and encodes an order to write a segment in the direction in which the head of the turtle points to. A new symbol, say f, says that the pen is up and advances without writing in the direction in which the head of the turtle points to.

5. A set of instructions to change directions. Our turtle lives in a two dimensional world and has instructions to rotate the direction of movement by ± 30 degrees or by 180 degrees. The instruction to rotate 30 degrees is the plus symbol '+', the contrary rotation is '-', a giro of 180 degrees is 'g'.

Example: we can encode a square just as "F+++F+++F+++F".
There could be modifications to the magnitude of the segment that is written, to the rotation of the direction of movement, to the certainty as the operations are done, to the color of drawing, to the dimensions of the base space and so on. Compare (Drabek et al, [20] 2007).

15 *The code for a Lindenmayer system*

Apart from the things concerning GUIs, here we learn a very important item of Java that is useful for complex projects: StringBuffers. These are heavy duty strings. The next instruction encloses a declaration, an instantiation and a symbolic initialization of the StringBuffer variable workingString:

```
StringBuffer workingString = new StringBuffer("");
```

StringBuffers have their own operators that are similar to those of Strings. The Reader is invited to look at them along the program. The possible operators that are applicable to a variable are displayed automatically by Eclipse and NetBeans if one adds a period to the variable, as in 'workingString.'

The code follows:

```
//Program C15, Lindenmayer
//A Swing GUI (Graphic user interface).
//Use of menus, windows  and buttons
//We implement a Lindermayer system lab,
//which consist in the study of strings
//and recurrence to encode
//for morphological information of a path.
//The program is divided in two main parts.
//The first part, Lindenmayer, is the director.
```

```
//It contains the menu bar
// and fundamental commands.
//The second part, LGrammar, takes care of
//the Lindemayer system and of
//the drawing window.
//Evolution with mutation and
//artificial selection by the User
//is also implemented.
package ejvol3v2p;


import java.awt.AWTException;
import java.awt.Color;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.BorderLayout;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import java.awt.geom.Rectangle2D;
import java.awt.image.BufferedImage;
import java.awt.image.RenderedImage;
import java.io.File;
import java.io.IOException;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.imageio.ImageIO;

import javax.swing.JColorChooser;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.ButtonGroup;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JMenuBar;
```

```
//MenuBar and fundamental controls
public class Lindenmayer extends JFrame {

    private static final long serialVersionUID = 1L;

    //Comunication of Lindenmayer with Lgrammar
    public static boolean demo1 = true;
    public static boolean demo2 = true;
    public static boolean random = true;
    public static boolean evol = false;
    public static String Name
            = "Jose Rodriguez, 2007 Colombia";



    //General controls
    public static Color colorF = Color.BLUE;
    public static Color colorG = Color.CYAN;
    private final Color colorValues[]
            = {Color.BLACK, Color.YELLOW, Color.MAGENTA,
                Color.GREEN};

    //Menu items
    private final JRadioButtonMenuItem colorItems[];
    private final JRadioButtonMenuItem fonts[];
    //font style menu items
    private final JCheckBoxMenuItem styleItems[];
    //displays a text
    private final JLabel displayJLabel;

    private final ButtonGroup fontButtonGroup;
    private final ButtonGroup colorButtonGroup;
    //Style of font
    private int style;


    //This is the constructor of the main class
    public Lindenmayer() {
        //Title
        super("Lindenmayer grammars, order,"
                + " complexity and art.");
        //Background color
        super.getContentPane().setBackground(colorF);
```

```
//Initialization
displayJLabel = new JLabel("Lindenmayer Systems");
displayJLabel.setForeground(Color.BLACK);
displayJLabel.setFont(new Font("SanSerif", Font.PLAIN, 12));
super.add(displayJLabel, BorderLayout.CENTER);

//*:::::::::::::::::::::::::::::::::::::::::
//The first column of  the menu is worked out
//Creates file menu
JMenu fileMenu = new JMenu("File");
//File menu is enabled  with Alt+ F
fileMenu.setMnemonic('F');

//The first menu item is added to the first column
JMenuItem aboutItem = new JMenuItem("Purpose...");
//If FILE IS enabled, About is open with Alt + A
aboutItem.setMnemonic('A');
//Adds about item to first column of the menu
fileMenu.add(aboutItem);
aboutItem.addActionListener( //Strucutre to answer to the user
        (ActionEvent event) -> {
            JOptionPane.showMessageDialog(
                    Lindenmayer.this,
                    "We use a turtle to implement "
                    + "\nLindenmayer systems."
                    + "\nA turtle is a pen that has a direction"
                    + "\nand that can be moved with or"
                    + "\nwithout drawing."
                    + "\nThe internal commands of the turtle are
                    + "\nF of forward and draw."
                    + "\nf of forward but without drawing."
                    + "\n+ for a positive change of direction"
                    + "\nof 30 degrees. "
                    + "\n- is the contrary."
                    + "\ng is a giro of 180 degrees."
                    + "\nA Lindenmayer system generates a  "
                    + "\ncorrespondence among  strings and "
                    + "\nmovements of the turtle. "
                    + "\nIn our encoding "
                    + "\nevery random string is viable.",
                    "Purpose", JOptionPane.PLAIN_MESSAGE
            );
        } //Displays message dialog
);
```

```
//The second menu item is added to the first column
JMenuItem lessonsItem = new JMenuItem("Lessons");
lessonsItem.setMnemonic('s');
fileMenu.add(lessonsItem);
lessonsItem.addActionListener((ActionEvent event) -> {
    JOptionPane.showMessageDialog(
            Lindenmayer.this,
            "One great lesson is this: "
            + "\nsome apparently very disordered systems"
            + "\nare indeed very simple  because  "
            + "\nthey are supported by a complex machinery."
            + "\nOther lesson is:"
            + "\nrandomness can produce order, "
            + "\nesthetics and art.",
            "Lessons", JOptionPane.PLAIN_MESSAGE
    );
} // Displays message dialog
);

//The third menu item is added to the first column
JMenuItem exitItem = new JMenuItem("Exit");
exitItem.setMnemonic('x');
fileMenu.add(exitItem);
exitItem.addActionListener((ActionEvent event) -> {
    System.exit(0);
} // terminates application
);

/**
 * ************************************
 * A new column is added to the menu
 * ************************************
 */
JMenu demoMenu = new JMenu("Demo");
demoMenu.setMnemonic('D');

//The first menu item is added to the Demo menu
JMenuItem demo1Item = new JMenuItem("demo1");
demo1Item.setMnemonic('K');
demoMenu.add(demo1Item);
demo1Item.addActionListener((ActionEvent event) -> {
    demo1 = true;
    demo2 = false;
    random = false;
    evol = false;
```

```
            Lindenmayer x1 = new Lindenmayer();
            LGrammar y1 = x1.new LGrammar(demo1, demo2, random, evol, Na
            y1.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
        });

        //The second menu item is added to the Demo menu
        JMenuItem demo2Item = new JMenuItem("demo2");
        demo2Item.setMnemonic('z');
        demoMenu.add(demo2Item);
        demo2Item.addActionListener((ActionEvent event) -> {
            demo1 = false;
            demo2 = true;
            random = false;
            evol = false;
            Lindenmayer x1 = new Lindenmayer();
            LGrammar y1 = x1.new LGrammar(demo1, demo2, random, evol, Na
            y1.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
        });

        /**
         * *********************************
         * A new column is added to the menu
    *********************************
         */
        JMenu compoMenu = new JMenu("Composition");
        demoMenu.setMnemonic('D');

        //The first menu item is added
        JMenuItem rItem = new JMenuItem("Random");
        rItem.setMnemonic('r');
        compoMenu.add(rItem);
        rItem.addActionListener((ActionEvent event) -> {
            demo1 = false;
            demo2 = false;
            random = true;
            evol = false;
            Lindenmayer x1 = new Lindenmayer();
            LGrammar y1 = x1.new LGrammar(demo1, demo2, random, evol, Na
            y1.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
        });

        //The second menu item is added
        JMenuItem eItem = new JMenuItem("Evolution");
        eItem.setMnemonic('h');
        compoMenu.add(eItem);
        eItem.addActionListener((ActionEvent event) -> {
```

```
    demo1 = false;
    demo2 = false;
    random = false;
    evol = true;
    Lindenmayer x1 = new Lindenmayer();
    LGrammar y1 = x1.new LGrammar(demo1, demo2, random, evol, Name);
    y1.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
});

/**
 * ****************************************
 * A new column will be added to the menu,
 * ****************************************
 */
//Hierarchy and names of menus and submenus:
//fathers and sons are Jmenu
//grandsons are JmenuItem:
//artMenu, backgroundMenu and branchesMenu are Jmenu
//color and name  are JmenuItem of backgroundMenu
JMenu artMenu = new JMenu("Art");
artMenu.setMnemonic('t');

//Creates Background menu
JMenu backgroundMenu = new JMenu("Background");
backgroundMenu.setMnemonic('O');

final JMenuItem colorB
        = new JMenuItem("Color");

backgroundMenu.add(colorB);
super.getContentPane().setBackground(colorF);
colorB.addActionListener((ActionEvent event) -> {
    Color c = Color.BLUE;
    final Container contentPane = getContentPane();
    c = JColorChooser.showDialog(
            colorB, getName(), c
    );
    contentPane.setBackground(c);
    colorF = c;
});

JMenuItem nameAuthor = new JMenuItem("Author");
nameAuthor.setMnemonic('e');
backgroundMenu.add(nameAuthor);

nameAuthor.addActionListener((ActionEvent event) -> {
```

```
        Name = JOptionPane.showInputDialog("Write your name here");
        System.out.println(Name);
    });

    //flowersMenu is added to the artMenu
    artMenu.add(backgroundMenu);

    //A new submenu is added to the Art menu
    JMenu lineMenu = new JMenu("lineColor");
    lineMenu.setMnemonic('C');

    // array listing string colors
    String colors[]
            = {"Black", "Yellow", "Red", "Green"};

    // creates radio button menu items for colors
    colorItems
            = new JRadioButtonMenuItem[colors.length];
    colorButtonGroup = new ButtonGroup();

    //Menus are called and activated separately.
    //Radio Buttons work as disjunctive ensemble:
    //the activation of one deactivates the others.
    //Checkboxes can be activated and deactivated
    //separately.
    //Instantiation of an inner class to listen
    //at the action over radiobuttons.
    ItemHandler itemHandler = new ItemHandler();

    //Creates color radio button menu items
    for (int count = 0; count < colors.length; count++) {
        colorItems[count]
                = new JRadioButtonMenuItem(colors[count]);
        lineMenu.add(colorItems[count]);
        colorButtonGroup.add(colorItems[count]);

        //Calling of inner class itemhandler
        colorItems[count]
                .addActionListener(itemHandler);
    }
    //Initialization
    colorItems[0].setSelected(true);

    artMenu.add(lineMenu);

    //Adds separator
```

```
artMenu.addSeparator();

//FONT control structure
String fontNames[]
        = {"Serif", "MonoSpaced", "SansSerif"};
JMenu fontMenu = new JMenu("Font of title ");
fontMenu.setMnemonic('n');

//Creates radio button menu items for font names
fonts = new JRadioButtonMenuItem[fontNames.length];
fontButtonGroup = new ButtonGroup();

//Creates Font radio button menu items
for (int count = 0; count < fonts.length; count++) {
    fonts[count]
            = new JRadioButtonMenuItem(fontNames[count]);
    fontMenu.add(fonts[count]);
    fontButtonGroup.add(fonts[count]);
    //Calling of inner class itemhandler
    fonts[count].addActionListener(itemHandler);
}

fonts[0].setSelected(true);
fontMenu.addSeparator();
//Names of styles
String styleNames[] = {"Bold", "Italic"};
styleItems = new JCheckBoxMenuItem[styleNames.length];

//Instantiation of an inner class to listen
//at actions over checkboxes.
StyleHandler styleHandler = new StyleHandler();

//Creates style checkbox menu items
for (int count = 0; count < styleNames.length; count++) {
    styleItems[count]
            = new JCheckBoxMenuItem(styleNames[count]);
    fontMenu.add(styleItems[count]);
    //Calling of inner class
    styleItems[count].addItemListener(styleHandler);
}
//adds Font menu to Art menu
artMenu.add(fontMenu);

//The whole munu bar is assembled
JMenuBar bar = new JMenuBar();
super.setJMenuBar(bar);
```

```
        bar.add(fileMenu);
        bar.add(demoMenu);
        bar.add(compoMenu);
        bar.add(artMenu);

    } // end MenuFrame constructor

    //***********************************
    //Inner class to handle action events:
    //buttons from branches and fonts.
    private class ItemHandler implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent event) {
            // Branch color selection
            for (int i = 0; i < colorItems.length; i++) {
                if (colorItems[i].isSelected()) {
                    colorG = colorValues[i];
                    displayJLabel
                            .setForeground(colorValues[i]);
                    break;
                }
            }

            // Font style selection
            for (JRadioButtonMenuItem font : fonts) {
                if (event.getSource() == font) {
                    displayJLabel.setFont(new Font(font.getText(), style
                }
            }

            repaint();
        }
    }

    //*****************************************************
    // Inner class to handle
    // item events from check box menu items
    private class StyleHandler implements ItemListener {
        // process font style selections

        @Override
        public void itemStateChanged(ItemEvent e) {
            style = 0; // initialize style

            // check for bold selection
```

```
        if (styleItems[0].isSelected()) {
            style += Font.BOLD;
        }

        // check for italic selection
        if (styleItems[1].isSelected()) {
            style += Font.ITALIC;
        }

        displayJLabel.setFont(
                new Font(
                        displayJLabel
                        .getFont().getName(), style, 12)
        );
        repaint();
    }
}


public static void main(String args[])
throws AWTException, IOException
{
    Lindenmayer x = new Lindenmayer();
    x.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    x.setSize(700, 700);
    x.setVisible(true);

    LGrammar y = x.new LGrammar(demo1, demo2,
            random, evol, Name);
    y.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

}

/**
 * ******************************************************
 *
 * LGrammar: creates, shows and saves the image
 *
 ******************************************************
 */
public class LGrammar extends JFrame {
    //The image can be seen with Paint or Draw
    //The region to be saved begins at the topleft corner.
    //It is rectangular with the following dimensions:

    int Width = 470;
```

```
int Height = 600;
//The image is a path that begins in here:
Dimension root = new Dimension(225, 300);
//To draw the image, we use a turtle,
//which is a pen that has a direction of movement.
//At the beginning, the turtle looks upwards.
double theta = Math.PI / 2;
//The angle of inclination of the direction
//can be change by deltaTheta.
double deltaTheta = Math.PI / 6;
//Dimension is a point in the plane.
Dimension position = new Dimension(root.width, root.height);
Dimension movement = new Dimension(0, 0);
//Each trace has a length
double length = 20;
//The Lindenmayer system begins with the axiom
String axiom = "";
//The L systems replaces recurrently the axiom
//by  the rule.
String rule = "";
//We define a thread of execution to order
//a prudent elapse of time during
//which an image is seen.
String name = "";

public Thread aThread = null;
//A StringBuffer is a String for heavy duty.
StringBuffer workingString = new StringBuffer("");

//Controls of the emergent window
boolean previous = false;
boolean next = false;
boolean mutation = false;
boolean newBeginning = false;
boolean save = false;
boolean first = false;
String saveAs = "a";

//The number of iterations of the recurrence:
int nIter = 3;

//Buttons to nagivate over the L system
private final JTextField text;
private final JButton btnPrevious;
private final JButton btnNext;
private final JButton btnNew;
```

```
private final JButton btnMutation;
private final JButton btnSave;

// Turn on  the random generator
Random r = new Random();

private static final long serialVersionUID = 1L;

//Constructor: Title and dimensions.
//The boolean variables are the channel of
//communication of the main part with this.
public LGrammar(boolean demo1, boolean demo2,
        boolean random, boolean evol,
        String name) {
    //super("L Grammars");
    super(name);
    Container c = super.getContentPane();
    c.setLayout(new FlowLayout());
    super.setSize(Width, Height);
    super.setVisible(true);
    c.setBackground(colorF);

    //::::::::::::button one less iteration
    btnPrevious = new JButton("ITER--");
    c.add(btnPrevious);
    //btnPrevious.setBounds(10,40,100,20);
    btnPrevious.setVisible(evol);

    //:::::::::: button one more iteration
    btnNext = new JButton("ITER++");
    c.add(btnNext);
    //btnNext.setBounds(120,40,100,20);
    btnNext.setVisible(evol);

    //:::::::::::::::::button NEW
    btnNew = new JButton("NEW");
    c.add(btnNew);
    //btnNew.setBounds(230,40,100,20);
    btnNew.setVisible(evol);

    //:::::::::::::::::button MUTATION
    btnMutation = new JButton("MUTATION");
    c.add(btnMutation);
    //btnMutation.setBounds(340,40,100,20);
    btnMutation.setVisible(evol);
```

```
        //:::::::::::::::::button SAVE
        btnSave = new JButton("SAVE");
        c.add(btnSave);
        //btnSave.setBounds(250,10,100,20);
        btnSave.setVisible(evol);

        //:::::::::::::::::textArea
        text = new JTextField(name);
        c.add(text);
        text.setEditable(false);
        text.setVisible(false);
        //text.setBounds(10,10,200,20);

        //actionListeners are added
        ButtonHandler bh = new ButtonHandler();

        btnSave.addActionListener(bh);
        btnPrevious.addActionListener(bh);
        btnNext.addActionListener(bh);
        btnNew.addActionListener(bh);
        btnMutation.addActionListener(bh);
    }

    //Inner class to deal with the actions of the user
    private class ButtonHandler implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
            if (null != e.getActionCommand()) {
                switch (e.getActionCommand()) {
                    case "ITER--":
                        previous = true;
                        next = false;
                        newBeginning = false;
                        mutation = false;
                        save = false;
                        break;
                    case "ITER++":

                        previous = false;
                        next = true;
                        newBeginning = false;
                        mutation = false;
                        save = false;
                        break;
                    case "NEW":
```

```
                        previous = false;
                        next = false;
                        newBeginning = true;
                        mutation = false;
                        save = false;
                        break;
                    case "MUTATION":

                        previous = false;
                        next = false;
                        newBeginning = false;
                        mutation = true;
                        save = false;
                        break;
                    case "SAVE":
                    {
                        previous = false;
                        next = false;
                        newBeginning = false;
                        mutation = false;
                        save = true;
                        saveAs = JOptionPane.showInputDialog(
                        "Save as (Example: mutant)");
                    }
                    default:
                        break;
                }
            }


            repaint();
            evol = true;



        }
    }

    //Saves the image to a jpg file
    private void saveDrawing() {
        RenderedImage image = memoryDrawing();
        try {
            ImageIO.write((RenderedImage) image,
                    "jpg", new File(saveAs + ".jpg"));
            System.out.println("Saved graphic(" + image.getWidth()
```

```
                        + " x " + image.getHeight() + " pixels) to file
                        + saveAs + ".jpg\"");
                System.out.println("The file was saved to local folder i
workspace.");
                System.out.println("Open the file with Paint or Draw in
Windows."
                        + "\nIn Linux, use not Image Viewer, "
                        + "\ninstead ImageMagick or Okular");
            } catch (IOException e) {
            }


        }

        //Creates in memory the image to be saved
        private RenderedImage memoryDrawing()
                  {
            int width = Width;
            int height = Height;
            System.out.println("width = " + width + " height = " + heigh
            //Creates a buffered image in which to draw
            BufferedImage image = new BufferedImage(width, height,
BufferedImage.TYPE_INT_ARGB);
            //Creates a graphics contents on the buffered image
            Graphics2D g2d = image.createGraphics();
            //Background
            g2d.setColor(Color.WHITE);
            g2d.fill(new Rectangle2D.Double(0, 0, Width, Height));
            name = Name;
            colorG = Color.cyan;
            //Paints the image
            System.out.println("AAAAAAAAAAA = " + axiom);
            System.out.println("RRRRRRRR = " + rule + "nIter = " + nIte
            synthesier(axiom, rule, nIter, g2d);
            return image;
        }

        //Movements of the turtle
        //Forward moves and paints
        private void Forward(Graphics g) {
            //System.out.println("F");
            Graphics2D g2d = (Graphics2D) g;
            g2d.setColor(colorG);
            Dimension position2 = new Dimension(0, 0);
            movement.width
                    = (int) Math.round(length * Math.cos(theta));
```

```
        movement.height
                = -(int) Math.round(length * Math.sin(theta));

        position2.width = position.width + movement.width;
        position2.height = position.height + movement.height;

        g2d.drawLine(position.width, position.height,
                position2.width, position2.height);
        position.width = position2.width;
        position.height = position2.height;
    }

    //forward moves but does not paint
    private void forward() {
        // System.out.println("f");
        Dimension position2 = new Dimension(0, 0);
        movement.width
                = (int) Math.round(length * Math.cos(theta));
        movement.height
                = -(int) Math.round(length * Math.sin(theta));

        position2.width = position.width + movement.width;
        position2.height = position.height + movement.height;

        position.width = position2.width;
        position.height = position2.height;
    }

    //deltaTheta
    private void rotateLeft() {
        // System.out.println("+");
        theta = theta + deltaTheta;
    }

    //deltaTheta
    private void rotateRight() {
        // System.out.println("-");
        theta = theta - deltaTheta;
    }

    //half a round is rotated
    private void giro() {
        //System.out.println("g");
        theta = theta + Math.PI;
    }
```

```
//A stringBuffer is translated to drawing actions
private void decode(StringBuffer workingString,
        Graphics g) {
    //System.out.println("decode");
    position.width = root.width;
    position.height = root.height;
    int n = workingString.length();
    for (int i = 0; i < n; i++) {
        char symbol = workingString.charAt(i);

        switch (symbol) {
            case 'F':
                Forward(g);
                break;
            case 'f':
                forward();
                break;
            case '+':
                rotateLeft();
                break;
            case '-':
                rotateRight();
                break;
            case 'g':
                giro();
                break;
        }
    }
}

//The axiom is replaces by the rule.
//The overall idea is this:
// a rule tarces a path.
//To insert a section, the original position
//must be restored. Thus, the section  is  completed
//with its inverse path.
//But moreover,  upon insertion
//the original direction must be restored.
//Hence, some additional bookkeeping must be done.
private StringBuffer replaceAll(StringBuffer here,
        String what,
        String byWhat) {
    // System.out.println("Axiom = " + what);
    // System.out.println("rule = " + byWhat);
    int m = what.length();
    int n = here.indexOf(what);
```

```
        int l = byWhat.length();
        //System.out.println("m = " + m);
        //System.out.println("n = " + n);
        while (n >= 0) {
            here = here.replace(n, n + m, byWhat);
            n = here.indexOf(what, n + l);
        }
        return here;
    }

    //A string is inverted
    private String invert(String a) {
        //System.out.println("invert");
        String b = "";
        int n = a.length();

        for (int i = n - 1; i >= 0; i--) {
            b = b + a.charAt(i);
        }

        return b;
    }

    //The backward path
    private String antiString(String s) {
        System.out.println("rule= " + s);
        String a = s.replaceAll("F", "h");
        a = a.replaceAll("f", "h");
        a = a.replaceAll("\\+", "m");
        a = a.replaceAll("-", "p");
        a = a.replaceAll("h", "f");
        a = a.replaceAll("m", "-");
        a = a.replaceAll("p", "+");
        a = invert(a);

        //System.out.println("anti-rule= " + a);
        return a;
    }

    //Axiom and rule is made into a super-rule
    private void synthesier(String axiom, String rule,
            int nIterations, Graphics2D g) {
        String bigRule = rule + "g" + antiString(rule) + "gF";
        workingString = new StringBuffer(bigRule);
        for (int i = 1; i <= nIterations; i++) {
            workingString = replaceAll(workingString, axiom, bigRule);
```

```
        }

        g.setColor(colorF);
        g.fill(new Rectangle2D.Double(0, 0, Width, Height));
        decode(workingString, g);
    }

    //a rule is synthesized ab initio
    private String improvization() {
        String impro = "";
        int n = r.nextInt(35) + 5;
        String alphabet = "Ffg+-";

        for (int i = 1; i <= n; i++) {
            int j = r.nextInt(5);
            char s = alphabet.charAt(j);
            impro = impro + s;
        }
        return impro;
    }

    private void sleep() {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
        }

    }

    //Synthesis de novo
    private void impro(Graphics2D g) {
        {
            axiom = "F";
            rule = improvization();
            // name = "Improvization";
            nIter = 2;
            synthesier(axiom, rule, nIter, g);

        }
    }

    //Insertion or deletion
    private String mutate() {

        int n = rule.length();
        int i = r.nextInt(n);
```

```
        StringBuffer brule = new StringBuffer(rule);
        double p = r.nextDouble();
        if (p > 0.5) {
            String alphabet = "Ffg+-h";
            int j = r.nextInt(6);
            char s = alphabet.charAt(j);
            String insertion = "" + s;
            if (insertion.equals("h")) {
                insertion = "F+Fgf-g";
            }
            brule = brule.insert(i, insertion);
        } else {
            brule = brule.deleteCharAt(i);
        }
        rule = brule.toString();
        return rule;
    }

    //Creates the image
    private void createImage(Graphics2D g)
            throws InterruptedException {

        //Defines the color of background:
        g.setColor(colorF);
        g.fill(new Rectangle2D.Double(0, 0, Width, Height));

        if (demo1 == true) {
            cells(g);
            Thread.sleep(2000);
            bacteria(g);
            Thread.sleep(2000);
            flower(g);
            Thread.sleep(2000);
            tree(g);
            Thread.sleep(2000);
            tree2(g);
            Thread.sleep(2000);
            planctum(g);
            Thread.sleep(2000);
            regular(g);
            Thread.sleep(2000);
            coneyHome(g);
            Thread.sleep(2000);
            demo1 = false;
            name = Name;
        }
```

```
if (demo2 == true) {
    plusMinus(g);
    Thread.sleep(2000);
    together(g);
    Thread.sleep(2000);
    nice(g);
    Thread.sleep(2000);
    Escher(g);
    Thread.sleep(2000);
    cubes(g);
    Thread.sleep(2000);
    esthetics(g);
    Thread.sleep(2000);
    stars(g);
    Thread.sleep(2000);
    name = Name;
    demo2 = false;
}

//random generation
if (random == true) {
    for (int i = 1; i <= 12; i++) {
        name = "random";
        super.setTitle(name);
        impro(g);
        sleep();
    }
    random = false;
}

if (evol == true) {
    name = Name;
    if (first == false) {

        first = true;
        nIter = 2;
        impro(g);
        sleep();

        evol = false;
        previous = false;
        next = false;
        newBeginning = false;
        mutation = false;
        save = false;
```

```
}

if (previous == true)//one iteration less
{
    if (nIter >= 1) {
        nIter--;
        synthesier(axiom, rule, nIter, g);
    }
    previous = false;
    next = false;
    newBeginning = false;
    mutation = false;
    save = false;
}

if (next == true) //one iteration more
{
    nIter++;
    synthesier(axiom, rule, nIter, g);
    previous = false;
    next = false;
    newBeginning = false;
    mutation = false;
    save = false;
}

if (mutation == true)//rule is mutated
{
    nIter = 2;
    rule = mutate();
    synthesier(axiom, rule, nIter, g);
    previous = false;
    next = false;
    newBeginning = false;
    mutation = false;
    save = false;
}

if (newBeginning == true) //new rule
{
    nIter = 2;
    impro(g);
    //sleep();
    previous = false;
    next = false;
    newBeginning = false;
```

```
                mutation = false;
                save = false;
            }

            if (save = true) {
                if (saveAs.length() > 0) {
                    saveDrawing();
                }
                previous = false;
                next = false;
                newBeginning = false;
                mutation = false;
                save = false;
            }

        }

    }

    //:::::::::::::::::: Demo1
    private void cells(Graphics2D g) {
        String myAxiom = "F";
        String myRule = "FF-fgg-Ff++fF";
        root.width = 50;
        root.height = 450;
        name = "cells";
        super.setTitle(name);
        int n = 2;

        synthesier(myAxiom, myRule, n, g);
    }

    private void flower(Graphics2D g) {
        String myAxiom = "F";
        String myRule
                = "F--g+gFgf++gFgf";
        root.width = 225;
        root.height = 250;
        name = "flower";
        super.setTitle(name);
        int n = 3;
        synthesier(myAxiom, myRule, n, g);
    }

    private void tree(Graphics2D g) {
        String myAxiom = "F";
```

```
        String myRule
                = "F++g-gFFgf----------g+gFFgf+gFgf+gFFgf-gFgf+gFFgf-gFgf"
        root.width = 185;
        root.height = 350;
        name = "tree";
        super.setTitle(name);
        int n = 3;
        synthesier(myAxiom, myRule, n, g);
    }

    private void tree2(Graphics2D g) {
        String myAxiom = "F";
        String myRule
                = "FF-FF+++-gg-+Fgfg--Fggfg-g++-f";
        root.width = 185;
        root.height = 350;
        name = "tree";
        super.setTitle(name);
        int n = 3;
        synthesier(myAxiom, myRule, n, g);
    }

    private void bacteria(Graphics2D g) {
        String myAxiom = "F";
        String myRule
                = "F-----g+gFgf +gFgf+gFgf+gFgf+gFgf+gFgf+gFgf+gFgf+gFgf";
        root.width = 225;
        root.height = 250;
        name = "bacterium";
        super.setTitle(name);
        int n = 3;
        synthesier(myAxiom, myRule, n, g);
    }

    private void planctum(Graphics2D g) {
        String myAxiom = "F";
        String myRule = "F-F+F+F-gFgf+gFFgf";
        root.width = 195;
        root.height = 250;
        name = "planctum";
        super.setTitle(name);
        int n = 3;
        synthesier(myAxiom, myRule, n, g);
    }

    private void regular(Graphics2D g) {
```

```
        String myAxiom = "F";
        String myRule = "f+F+Ffg";
        root.width = 195;
        root.height = 250;
        name = "regular";
        super.setTitle(name);
        int n = 5;
        synthesier(myAxiom, myRule, n, g);
    }

    private void coneyHome(Graphics2D g) {
        String myAxiom = "F";
        String myRule = "g+++g+F+g-F-gf-ff-g-";
        root.width = 195;
        root.height = 250;
        name = "coneyHome";
        super.setTitle(name);
        int n = 5;
        synthesier(myAxiom, myRule, n, g);
    }

    //:::::::::::::::::Demo2
    private void plusMinus(Graphics2D g) {
        String myAxiom = "F";
        String myRule = "F+gfgg+FgF";
        root.width = 200;
        root.height = 250;
        name = "plusMinus";
        super.setTitle(name);
        int n = 3;
        synthesier(myAxiom, myRule, n, g);
    }

    private void together(Graphics2D g) {
        String myAxiom = "F";
        String myRule
                = ("gf-g+gF+-+fgggf-gFf++g-F+ffggg");
        root.width = 170;
        root.height = 250;
        name = "together";
        super.setTitle(name);
        int n = 3;
        synthesier(myAxiom, myRule, n, g);
    }

    private void nice(Graphics2D g) {
```

```
        String myAxiom = "F";
        String myRule = "---FffggFf-f-f++gF-";
        root.width = 175;
        root.height = 300;
        name = "nice";
        super.setTitle(name);
        int n = 3;
        synthesier(myAxiom, myRule, n, g);
    }

    private void Escher(Graphics2D g) {
        String myAxiom = "F";
        String myRule
                = "--F++g-+gF++ggfF+++";
        root.width = 225;
        root.height = 350;
        name = "Escher";
        super.setTitle(name);
        int n = 3;
        synthesier(myAxiom, myRule, n, g);
    }

    private void cubes(Graphics2D g) {
        String myAxiom = "F";
        String myRule
                = "g+gfgf+fggFgg-ff-FFf+g+FF+fgg-";
        root.width = 205;
        root.height = 250;
        name = "cubes";
        super.setTitle(name);
        int n = 3;
        synthesier(myAxiom, myRule, n, g);
    }

    private void esthetics(Graphics2D g) {
        String myAxiom = "F";
        String myRule = "--gFffgg-+Fg---f-Ff";
        root.width = 230;
        root.height = 350;
        name = "esthetics";
        super.setTitle(name);
        int n = 5;
        synthesier(myAxiom, myRule, n, g);
    }

    private void stars(Graphics2D g) {
```

```
            String myAxiom = "F";
            String myRule = "f++f--f----fgfFf+-+f--gff-F+++gg";
            root.width = 230;
            root.height = 350;
            name = "stars";
            super.setTitle(name);
            int n = 5;
            synthesier(myAxiom, myRule, n, g);
        }

        //Drawing manager
        @Override
        public void paint(Graphics g) {
            Graphics2D g2d = (Graphics2D) g;
            g2d.setColor(colorF);
            g2d.fill(new Rectangle2D.Double(0, 0, Width, Height));
            //demo1 = false;
            //demo2 = false;
            //random = false;
            //paint the screen with
            //the image created by createImage(g)
            try {
                createImage(g2d);
            } catch (InterruptedException ex) {
                Logger.getLogger(Lindenmayer.class.getName()).log(Level
null, ex);
            }
        } //end of paint()

    }// End of LGrammar
}//End of Program C15 Lindenmayer
```

16 ***Exercise*** *Run the code and play with it. Use the navigating tools to see the form as two large classes, Lindenmayer and LGrammar are posited together.*

17 ***Please, have patience.*** *Java is so rich in possibilities that it can be adapted to fill the needs of all types of developers from aficionado, our first chapters, to professional, as in this chapter, to sophisticated, as those application that cost some thousand dollars. As we see, professional termination of programs demands specific know-how whose acquisition may demand a great effort. A developer always has new ideas that render a given GUI useless, so finely terminated programs are done at a high price in effort and know-how. Now, there are all around the world millions of people that consider that evolution is a synonym of evil, to say the least. What we must say is that evolution is a delicious matter of study that is done with more fun if we use Java.*

1.6 Conclusion

To illustrate the idea that evolution has no purpose at all but it is just a game with gambling and selection, we presented two programs to generate morphological diversity, the biomorphs of Dawkins and an original implementation of an L-grammar. Nevertheless, our simulations are far away from evolutionary biology: to begin with, our simulations are systematically directed to create variability, so it reflects more a spirit of design than the achievements of purposeless evolution. In second place, our programs implement very sophisticated evolutionary environments, whose creation is due to highly trained people that live in a highly developed culture. In third place, our selection was not an emergent feature of the environment but an activity of the person that runs the program. Thus, we must confess that we fail in our purpose of presenting the creation of variability without soiling artifacts, hidden purposes or hidden intelligence. In plain words, our models are perfect for creationism. Most possibly, the only way to escape from complaints and accusations of this sort is to devise a dynamical system in which evolution could arise from nothing to see what it gives from itself. Henceforth, we cannot save forever the study of the question of the origin of evolution.

Chapter 2

Evolution as an style

`The software of the genome is a marvel with an evolvable style`

18 ***Motivation and purpose*** *We perceive by instinct that living beings are marvelous. Our purpose is to discuss the form as this instinctive perception could be translated into quantitative studies. Whatever this could mean, this implies in first instance that the software contained in their genome is also marvelous. In fact, we discover that the genome obeys an style of design that can be appropriately called evolvable.*

2.1 From feelings to measures

Let us elaborate the feeling that alive beings are marvelous to extract something interesting for science.

19 ***Feelings and more feelings***

A top ten tennis player is near the net and his opponent proposes a lobe. To answer to this challenge, he recedes without changing the orientation of his body, jumps 40 cms, hits the ball that departs at 90 m/h and wins the point. After this, he nicely lands. The public breaks in an applause while he celebrates with enthusiasm and then goes for a new service.

This description of the normal things that a trained human can do could be continued for the infinite of time. One marvels at things like that just because the star we applause can do things that are very difficult for us. So, we ask: Does this subjective qualification have any objective, evolutionary meaning?

The answer is a definitive yes: the great achievements in robotics, such as they are saw in the concourse for robots in which they play football, show that it is quite a difficult and very challenging task to have a robot to play football with as a high level as that of a two years old baby. On the other hand, one easily can imagine multitude of sceneries in which the dexterity for movement defines who lives and who dies or who reproduces and who goes to extinction in the struggle for life, say, in all coupling rituals in which movement plays a fundamental role.

We find then that our feelings of admiration can predict evolutionary important aspects of life. Why? Sociobiologists immediately might answer: feeling might enclose important evolutionary descriptors because feelings are important for behavior, which has been and

continues to be molded by evolution. For mechanicists, such a thesis represents a terrible challenge: How can we simulate in Java a feeling? Are we going to claim that feelings are not part of science and hence and for all effects they do not exist? What do we lack if we ignore feelings and consciousness? These questions prepare us to think rather prudently about the human and social meaning of our studies and conclusions. For the time being, let us return to our purpose of elaborating the feeling of admiration for life into some important scientific point that could be manageable with Java.

20 ***Theory vs practice***

The only way to turn Java into a tool for studying problems is to detect important questions (maybe using the guide of feelings) and to formulate those problems in quantitative language. So, we have detected that the software contained in the genome is marvelous and that this quality is of technical and evolutionary importance. Let us emphasize the technical importance: software has become an essential and crucial part of our modern world. That is why the production of high effective software is one of the dreams of any company or developer in the trade.

In other words, engineers, mathematicians and physicists working in computing science are very interested in the study of all questions in relation with high quality software. The sophistication that has been achieved by modern standards in the study of the theme is so high that non experts could feel overwhelmed. In this regard, it could be interesting to read a comment written not long ago:

'The onion-domed Church of the Intercession of the Virgin on the Moat in Moscow is one of Russia's most famous landmarks. It was built by Tsar Ivan IV just outside of the Kremlin walls in 1552 to commemorate Russia's victory over the Tatars at Kazan. The church is better known by it's nickname, St. Basil's. Ivan too is better known by his nickname "Ivan the Terrible". Legend has it that once the cathedral was completed, Ivan, ever true to his reputation, had the architects blinded, so that they could never build anything more beautiful. Alas, the state of software architecture today is such that few of us need fear for our eyesight' (Footer and Yoder, [25] 1999).

This comment means that experts in computing science feel that the problem of design of high quality software is so complex that there is a tremendous gap between theory and practice and so we are just at the dawn of its study.

2.2 Evolution in the world of software

Software is fundamental to modern society much as air and water and oil. Henceforth, it is important to understand the know-how of producing competitive software. The next is a quite solid picture that represents a good approach to our problematic. It departs from accepting that design does not consist in making a program to fulfill a certain function. Reality is otherwise: when somebody says that he has a problem to be solved, most of times he has no idea of what he or she wants. The client begins to understand what he or she wants when the developer brings forth a solution to what he or she asked for. At this moment, a succession of corrections to the objective of the project follows, whose termination is dictated more by budgets problems than by perfect satisfaction.

The wise solution to this problem consists in accepting that any project is an eternally changing objective. Henceforth, one must be pre-adapted to the most drastic changes, i.e., **one must be ready to evolve** when that is possible and to begin anew from scratch as many times as necessary but reusing as much as possible from previous work.

On the other hand, when a designer makes a program to fill a function, he or she does not know that to fill a certain function encloses to fill another thousands functions 'that anyone with eyes in the front shall know that they must be fulfilled'. This is pathetic in modern programs in which the User has the option to combine too many parameters and options to be run and that would required too many experiments to have them function perfectly. A commercial strategy is to release a beta version without fees that Users could test them with a facility to report found bugs or failures. These are given a priority for fixing according to its statistical significance. But there are a multitude of cases in which restrictions of time and/or clients allow too few tests and so one must go around without the help of a beta version.

Because some types of bugs seem naive after their unveiling, experts pretend that some few reasons could explain a good proportion of errors. Therefore, they invent programming platforms where committing those errors is forbidden or has a low probability. Java and Eclipse has incorporated some crucial points of this study, say, Eclipse posts a warning against any non used variable or incomplete code. By the same token, Java allows the formal declaration variables whose range shall be restricted to suitable bounds. Another more sophisticated proposal is to design an automatic tester environment (Alloy group, [2] 2008) but seemingly this solution falls also short of power to deal with very complex projects.

The implication of the pervasiveness of bugs is that one generates a regression of corrections that **define per se an evolutionary process**.

Henceforth, evolution is the very quality that defines what is good software: a given software is good when it allows evolution, easy changes, easy restructuring, easy last minute amendments (Footer and Yoder, [23], 1995). The great lesson is that the quality of evolvability tightly depends on the programming platform that one uses. That is why programming languages are carefully designed and many options are available because the complexity of the problem cannot be got rid with just one solution, but instead one can propose efficient solutions for problems in a concrete, specific domain.

It is believed that OOP (object oriented programming) is the basic platform that best fits the need of evolvability that good software must have (Footer and Yoder, l.c., 1995). The reason is that OOP tries to mimic reality: in real life, one can posit in a box too many objects of the most diverse nature, one can change them, one can modify their dependence and interconnection, and one can add and remove. An all this could be done effortlessly. So, a good artificial reality shall enjoy the very same properties. OOP was built with these objectives in mind. This is so a powerful reason that various languages, Java included, are completely based on OOP. Today, OOP is considered insufficient to cope with all the complexity of programming and new proposals are under study (Mens at al. , 2008). Please, do not worry unnecessarily about so much development: one is very limited and it is impossible to be an expert in everything and moreover that is sometimes superfluous if one has patience with own insufficiencies.

Real life permanently shows that the good property of evolvability of a programming lan-

guage could be used to produce programs that are not evolvable at all: they are too messy to be understood and hardly accept functional modifications. So, a good style of programming must be atop a good programming platform. But this dream is too naive: real programmers produce programs that resemble a big ball of mud, which 'is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle'(Footer and Yoder, l.c., 1999).

How is that possible?

In a competitive market place, the only quality required by software is just to function as required no matter what it could have in its entrails. This is the client point of view. But as we saw, the client originates an eternal regression of changes and so evolvability is necessary. For projects from low to medium size, developers know how to function with their messy style. But for very large projects, their very powerful intelligence is not enough to cope with all the complexity and projects are abandoned when the opportunity arises.

So, if you want to work in large projects, learn to work with a very clean and organized style. In this regard, let us make explicit that the programs that are offered in here classified also as belonging in the big ball of mud. The reason is that it is not enough to use structured programs but it is also necessary to diminish the interconnecting load. An innocent but efficient form as one loads interconnection is to use globally defined variables when locally defined ones would do the job. Another reason is that we have not used the full fledged power of OOP. In this approach, one encapsulates methods that work on the same variables in the same object. But apart from the initialization methods, all others belong to the common reservoir, i.e., to the highest level class. So, we lack sufficient compartmentalization.

There is still another point: software is very difficult to design. So, it is intelligent to look for reuse: why do you must bother in re-inventing the warm water? Just use the extant software if it meets your needs. But reuse is not always possible. This is achieved if your objects, methods and classes have clear functions and transparent and few connections. This is a very fine style of design that is also enabled by OOP and by Java. Experience shows that this style demands a great dose of additional effort, so much that most people produce just big balls of mud especially because everyone is under pressures of budget and time.

Reuse is just another fold of evolution: if one modifies own programs, one exercises vertical evolution. But if one goes for reuse, one is exercising horizontal transfer and recombination at large scale, and these are other mechanisms of evolution.

So, our conclusion is that evolvability is at the heart of the set of qualities of good software.

2.3 Divide and conquer

One of the main points of this chapter is that there are various styles of software design and that most developers produce a sort of software that resembles a spaghetti jungle and that is now traditionally called big ball of mud. The problem is that such a style strongly diminishes the potentialities of evolvability that good software has. By contrast, some people seem to

preach that there is a style of design that empowers evolvability. We are sure that no such style has been invented and that software continues to be produced with great suffering. This, of course, does not preclude that such style could be created somehow. But we are skeptical.

Let us present now a parable that somehow justifies our negative position. Our parable is presented in the form of a simulation in which a certain evolvable style is defined and tried to be implemented.

Everyone appreciates a style of software design in which one is given the opportunity to read and study units or methods whose complete structure could be grasped at once over the display. People that are able to work with such a style are experts in applying the doctrine that invites to 'divide, name and conquer'. For instance, ancient cultures divided the human body in natural parts and named them. I do know how important is to distinguish my left hand from my right ear and so on. Now, thousands of inconvenient divisions could be allotted but the one delivered by ancient cultures is perfect and so they conquered my respect for what they did. That is why we choose to study the complexity of implementing a very simple criterion to divide a whole into pieces.

A whole is represented by a set of elements and their interrelations. This characterization is done by intuition, by instinct and its final output is the point we take as start. The simplest mathematical structure that allows us to represent a whole is a **graph**, which is a couple (V,E) where V is a set of vertexes or nodes and E is a set of edges or links among nodes. The information of a graph is located in a boolean matrix. The vertexes are called $0,1,....,n-1$, and the matrix has a *true* else a *false* at the entry (i,j) if and only if there is a link from node i to node j. So, a link could denote a direction of the interrelation. If one does not want any direction, one might use a symmetric matrix: if (i,j) is true, then (j,i) is also true.

One can devise many diverse criteria to divide a graph in parts. The simplest one is to cut it in two equal parts. But this operation can be executed in many forms: ours is to cut naturally, i.e. causing the minimum possible trauma. The mathematical formulation of our purpose is enclosed in the so called **minimum bisection problem** for a graph (V,E) with an even number of vertexes, which consists in cutting V into two parts of equal size, such that the number of edges between them is to be minimized. This problem is easy for small graphs or for graphs with scare edges or for graphs with clearly marked divisions, but this problem could be very tedious for other graphs. We will consider in our simulation graphs that are populated with links at random. Our purpose is to study the number of operations of an algorithm that are needed to solve the problem depending on the number of vertexes and of links.

21 ***The code*** *The code that finds an approximate solution to the minimum bisection problem follows. The degree of approximation depends on the number of generations that the program is allowed to run. The code is new for this version.*

```
package ejvol3v2p;

/* Program C21, Bisection.


A good style of programming produces modules that
are maximally interconnected internally
```

```
but with a minumum of interconnections among modules.
Is that easy to achieve?
To offer a tool to partially study this question
we study the problem Bisection:

(V,E) is a graph with an even number of vertexes,
nodes or cities V,  and with a set E of links, edges or roads.
The task is to divided the graph in
two parts of equal size such that
the number of links between them is to be minimized.
So, the FITNESS function is the number of links between
the two parts. One must minimize that number.

The problem is solved by a helped genetic algorithm.
The individuals represent bisections.
They are modelled over arrays not over strings.
The program compares  initial
and final populations of bisections.
 */
import java.util.Random;

public class Bisection {

    private static final int N = 5;
    //NCITIES  must be even
    private static final int NCITIES = 2 * N;
    //A link is generated at random with this threshold
    private static final double PROBOFLINK = 0.4;

    // LINKS keeps a record of the interconnecting links
    private static final boolean LINKS[][]
            = new boolean[NCITIES][NCITIES];

    //This is the population size of the GA
    private static final int NBISECTIONS = 50;
    private static final boolean BISECTION[][]
            = new boolean[NBISECTIONS][NCITIES];
    private static final int FITNESS[] = new int[NBISECTIONS];
    private static final int ORDER[] = new int[NBISECTIONS];
    private static int generation;
    private static final int LASTGEN = 2000;
    private static final double MUTATIONRATE = 0.1;
    private static final boolean WITHRECOMBINATION = false;
    private static final double RECOMBRATE = 0.2;
    private static int minZeroGen = 0;
    private static int minLastGen = 0;
```

```
    private static final boolean BESTBS[]
            = new boolean[NCITIES];
    private static final boolean PRINT = false;


    //Turn on the random generator
    static Random r = new Random();

    //Initializes LINKS[][],
    //a boolean matrix that encodes for links among cities:
    //There is a link that connects city i to city j
    //iff LINKS[i][j] = true
    // No link of the form (i,i) .
    //To avoid stakoverflowing created by cycles
    //if a link is created, its reverse is interdicted.
    private static void initializeLinks() {
        //Background initialization in false = no links
        for (int l = 0; l < NCITIES; l++) {
            for (int m = 0; m < NCITIES; m++) {
                LINKS[l][m] = false;
            }
        }

        for (int l = 0; l < NCITIES; l++) {
            for (int m = l + 1; m < NCITIES; m++) {
                double q = r.nextDouble();
                if (q < PROBOFLINK) {
                    int k = r.nextInt(2);
                    if (k == 0) {
                        LINKS[l][m] = true;
                    } else {
                        LINKS[m][l] = true;
                    }
                }
            }
        }
    }

    //Prints interconnection matrix
    private static void printMatrix() {

        System.out.print("     ");
        for (int i = 0; i < NCITIES; i++) {
            System.out.print(i % 10 + " ");
        }
        System.out.println();
```

```
        for (int i = 0; i < NCITIES; i++) {
            System.out.print(i % 10 + "    ");
            for (int j = 0; j < NCITIES; j++) {
                if (LINKS[i][j] == true) {
                    System.out.print(1 + " ");
                } else {
                    System.out.print(0 + " ");
                }
            }
            System.out.println();
        }
    }

    private static void printMatrixB() {

        System.out.println("Bisections");
        System.out.print("     ");
        for (int i = 0; i < NCITIES; i++) {
            System.out.print(i % 10 + " ");
        }
        System.out.println();
        for (int i = 0; i < NBISECTIONS; i++) {
            System.out.print(i + "    ");
            for (int j = 0; j < NCITIES; j++) {
                if (BISECTION[i][j] == true) {
                    System.out.print(1 + " ");
                } else {
                    System.out.print(0 + " ");
                }
            }
            System.out.println();
        }
    }

    private static void printBisection(boolean[] bs) {

        for (int j = 0; j < NCITIES; j++) {
            if (bs[j] == true) {
                System.out.print(1 + " ");
            } else {
                System.out.print(0 + " ");
            }
        }
        System.out.println();
    }
```

```
//To bisect the set of 2n vertexes,
//n  different vertexes are associated at random
//to part A and the others to part B.
//The number of bisections is nBISECTIONS
//Bisections are kept in BISECTION[][]
private static void initializeBisections() {

    for (int i = 0; i < NBISECTIONS; i++) {
        for (int j = 0; j < NCITIES; j++) {
            BISECTION[i][j] = false;
        }
    }

    for (int i = 0; i < NBISECTIONS; i++) {
        // System.out.println("Nodes in BISECTION A");

        for (int j = 0; j < N; j++) {
            int l = 0;
            boolean repeated = true;
            // || means logical or.
            while (repeated) {
                l = r.nextInt(NCITIES);
                /*
               //Equivalent to
               if (BISECTION[j][l]  == true) repeated = true;
               else repeated = false;
                     */
                repeated = BISECTION[i][l] == true;
            }
            BISECTION[i][l] = true;
            //System.out.print(l + " ");

        }
        //System.out.println();

    }
    printMatrixB();
}

//The FITNESS of   Bisection k is the number
//of edges that link SECTION A with SECTION B
private static void fitness() {

    for (int k = 0; k < NBISECTIONS; k++) {

        FITNESS[k] = 0;
```

```
            for (int i = 0; i < NCITIES; i++) {
                for (int j = 0; j < NCITIES; j++) {
                    /*
                    System.out.println("\nFitness of bisection " + k + ": "
                        + FITNESS[k]
                           + " i = " + i + " j = " + j
                           +"\nBISECTION[k][i] = " + BISECTION[k][i]
                           + "\nBISECTION[k][j] = " + BISECTION[k][j]
                           + "\nLINKS[i][j] = " + LINKS[i][j]
                           + "\nLINKS[j][i] = " + LINKS[j][i] );
                     */
                    //If i and j are in contrary sections
                    if ((BISECTION[k][i] == !BISECTION[k][j])
                            //and are linked
                            && (LINKS[i][j] == true)) {
                        FITNESS[k]++;
                    }

                    /*
                    System.out.println("\nFitness of bisection "
                            + k + ": "  + FITNESS[k]);
                     */
                }
            }
            /*
            System.out.println("\nFitnesssssss of bisection "
                            + k + ": "  + FITNESS[k]);
             */
        }

        //Initial and last populations are recorded
        if ((generation == 0) || (generation == LASTGEN - 1)) {
            System.out.println("\nGeneration = " + generation);
            for (int k = 0; k < NBISECTIONS; k++) {
                System.out.println("Fitness of bisection " + k + ": "
                        + FITNESS[k]);
            }
        }

    }

    //Individuals are sorted by FITNESS
    private static void sorting() {
        int Champ;
        // Necessary default initialization
```

```
        //We sort individuals by FITNESS: the lesser the better
        for (int i = 0; i < NBISECTIONS; i++) {
            Champ = 0;
            for (int j = 0; j < NBISECTIONS; j++) {
                if (FITNESS[j] <= FITNESS[Champ]) {
                    Champ = j;
                }
            }
            //The champ is recorded
            if ((i == 0) && (generation == 0)) {
                minZeroGen = FITNESS[Champ];
            }
            if ((i == 0) && (generation == LASTGEN - 1)) {
                minLastGen = FITNESS[Champ];
                System.arraycopy(BISECTION[Champ], 0, BESTBS, 0, NCITIES

            }

            //The array ORDER orders individuals by  FITNESS:
            // the best is the first.
            ORDER[i] = Champ;

            FITNESS[Champ] = 1000;
        }

        /*
        for(int i = 0; i< NBISECTIONS;i++)
System.out.println( ORDER[i]);
         */
    }

    // The new generation is made
    private static void newGeneration() {
        boolean BISECTION2[][] = new boolean[NCITIES][NCITIES];
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 2;
        int k = 0;  //rank, 0 is the highest

        while (i < NCITIES) {
            int counter = numbCopies;
            while ((counter > 0) & (i < NCITIES)) {
                //BISECTION2[i][l] = BISECTION[ORDER[k]][l];
                System.arraycopy(BISECTION[ORDER[k]], 0, BISECTION2[i],
```

```
NCITIES);
                //System.out.println(i + " " + counter + " " + ORDER[k]);
                i++;
                counter = counter - 1;
            }
            k++;
        }

        for (int j = 0; j < NCITIES; j++) {
            //BISECTION[j][l] = BISECTION2[j][l];
            System.arraycopy(BISECTION2[j], 0, BISECTION[j], 0, NCITIES);
            /*System.out.println( j+ " length at newGen " +
            Pop1[j].length);
             */
        }
        //printMatrixP();
    }

    //Bisections are subjected to recombination
    //with a rate equal to RECOMBRATE
    private static void recombination() {
        boolean BISECTION2[][]
                = new boolean[NBISECTIONS][NCITIES];
        for (int i = 0; i < NBISECTIONS; i++) {
            int j = r.nextInt(NBISECTIONS);
            //Bisections i and j are recombined
            for (int k = 0; k < NCITIES; k++) {
                double q = r.nextDouble();
                if (q < RECOMBRATE) {
                    BISECTION2[i][k] = BISECTION[j][k];
                } else {
                    BISECTION2[i][k] = BISECTION[i][k];
                }
            }
        }

        for (int j = 0; j < NCITIES; j++) //BISECTION[j][l] = BISECTION2[j][l]
        {
            System.arraycopy(BISECTION2[j], 0, BISECTION[j], 0, NCITIES);
        }
        //printMatrixP();
    }

    //The same number of elements in both sections
    //is enforced.
    private static void restoreOrder() {
```

```
        for (int i = 0; i < NBISECTIONS; i++) {
            int numberOfTruths = 0;
            for (int k = 0; k < NCITIES; k++) {
                if (BISECTION[i][k] == true) {
                    numberOfTruths++;
                }
            }
            int error = N - numberOfTruths;
            //System.out.println("\nError1 = " + error);

            while (error > 0)//false is dominant
            {
                int l = r.nextInt(NCITIES);
                if (BISECTION[i][l] == false) {
                    BISECTION[i][l] = true;
                    error--;
                }

            }
            while (error < 0)//true is dominant
            {
                int l = r.nextInt(NCITIES);
                if (BISECTION[i][l] == true) {
                    BISECTION[i][l] = false;
                    error++;
                }
            }
            /*
            numberOfTruths = 0;
            for (int k = 0; k < NCITIES; k++)
                if (BISECTION[i][k] == true) numberOfTruths++;
            error = N-numberOfTruths;
            System.out.println("Error2 = " + error);
             */
        }
    }

    //Bisections are boolean vectors.
    //A mutation is a change of a boolean value,
    //it is always inviable because the numbers
    //of cities in each side gets different.
    //So, one must restore the order: both sections
    //must have the same number of cities.
    private static void mutation2() {
        for (int i = 0; i < NBISECTIONS; i++) {
            //System.out.println( l);
```

```
            for (int k = 0; k < NCITIES; k++) {
                double q = r.nextDouble();
                if (q < MUTATIONRATE) {
                    BISECTION[i][k] = !(BISECTION[i][k]);
                }
            }
        }

        //printMatrixP();
    }

    //Bisections are boolean vectors.
    //A mutation is a change of a boolean value,
    //it is always inviable because the numbers
    //of cities in each side gets different.
    //But a  swap of two cities is always viable:
    //Originally: city i in A, city j in B.
    //After swapping: city i in B, city j in A.
    private static void mutation() {
        for (int i = 0; i < NBISECTIONS; i++) {
            //Bisection i undergoes mutation
            boolean go = true;
            //System.out.println( l);
            for (int k = 0; k < NCITIES; k++) {
                double q = r.nextDouble();
                if (q < MUTATIONRATE) {
                    while (go) {
                        int l = r.nextInt(NCITIES);
                        if (BISECTION[i][k] == BISECTION[i][l]); else {
                            if (PRINT) {
                                System.out.println("\nIncoming bisection");
                                printBisection(BISECTION[i]);
                            }
                            BISECTION[i][k] = !(BISECTION[i][k]);
                            BISECTION[i][l] = !(BISECTION[i][l]);
                            go = false;
                            if (PRINT) {

                                System.out.println("Mutated bisection");
                                printBisection(BISECTION[i]);
                            }
                        }
                    }
                }
            }
        }
```

```
    }

    private static void evolution() {

        for (int gen = 0; gen < LASTGEN; gen++) {
            generation = gen;
            fitness();
            sorting();
            newGeneration();
            if (WITHRECOMBINATION) {
                recombination();
                //Recombination always produces disparity
                restoreOrder();
            }
            mutation();

        }
    }

    //The best Bisection is obtained
    private static void bisect() {

        evolution();
    }

    public static void main(String args[]) {
        System.out.println("Number of cities = " + NCITIES);
        System.out.println("Probability of link = "
                + PROBOFLINK);
        System.out.println("Numb of bisections  = "
                + NBISECTIONS);
        System.out.println("Number of generations = "
                + LASTGEN);
        System.out.println("MUTATIONRATE/char/gen = "
                + MUTATIONRATE);
        System.out.println("WITHRECOMBINATION = "
                + WITHRECOMBINATION);
        if (WITHRECOMBINATION) {
            System.out.println("RECOMBRATE/char/gen= "
                    + RECOMBRATE);
        }
        System.out.println();
        initializeLinks();
        System.out.println("Matrix of links ");
        printMatrix();
        initializeBisections();
```

```
        bisect();
        System.out.println("\n\nFitness "
                + "= number of interlacing links");
        System.out.println("\nFitness first gen = "
                + minZeroGen);
        System.out.println("Fitness last  gen = "
                + minLastGen);
        double efficiency = minZeroGen;
        efficiency = efficiency / minLastGen;

        System.out.println("Efficiency "
                + "= minZeroGen/minLastGen = "
                + efficiency);
        System.out.println("Best bisection: ");
        for (int j = 0; j < NCITIES; j++) {
            if (BISECTION[ORDER[0]][j] == true) {
                System.out.print(1 + " ");
            } else {
                System.out.print(0 + " ");
            }
        }
        System.out.println();
    }
}//End of Program C21, Bisection.
```

22 ***Exercise*** *Play with the code and try to understand it. What can you say about the easiness of finding an interesting bisection (one bisection with around a half of the number of crosses of a random bisection)?*

Evolution is good to solve problems, so we can make sure that the conclusion obtained in the previous exercise is valid for every possible algorithm: graphs with scare links can be efficiently bisected even by randomness but evolution is of no help for graphs with high density of links.

In relation with the style of design, our conclusion can be translated as follows: a good style of developing software, one which divides to conquer, is implementable for easy problems (one in which the interacting parts are not so strongly tied one with another) but no good style exists for problems with appropriately high density of links. Of course, one can produce small units with as few connections as desired, but the web of the interrelation of those units will have a highly interlinked topology. It is a well known fact that programs with parts that have high density of interconnections are very hard to read, to debug and to evolve. Thus, it is natural to admire a transparent style when one is found or created.

2.4 Diverse evolutionary styles

How is the style of a developer that has taken evolvability as the fundamental quality of good software? Let us highlight two styles that an evolutionary developer could have.

23 *Biogenesis style*

One evolutionary style that is very intriguing seems to obey the next dictum: life is begotten from life more easily than from no life. Actually, this is a directive that is followed by instinct: one tries to begin from very simple programs that work perfectly to add more and more complexities that are tested as soon as possible until requested demands are fulfilled. For very large projects instantaneous testing is not possible, so some people aim at a daily build for every day while other tries to get a working version by the end of each week (Foot and Yoder, l.c., [25] 1999). Warning: keep with yourself the last working version because the dealt is very hard and one does not know if by the end of the day one caught something alive.

What must we do to optimize this methodology? We need to actively compose an evolvable project. Let us discuss this more explicitly.

It is a recurrent observation of theoretics of software design that any evolutionary branch leads to stagnation. The following is a description of that fact:

'The staged process model.. .. is interesting in that it explicitly takes into account the inevitable problem of *software aging*.. .. After the initial stage of development of a first running version, the evolution stage follows for any kind of modification to the software, as long as the architectural integrity remains preserved. If this is no longer the case, there is a loss of evolvability (also referred as *decay*) and the servicing stage starts. During this stage, only small patches can be applied to keep the software up and running. If even such small patches become too costly to carry out, the phase-out stage starts, leading to ultimate close down of the stakeholders, it is called a *legacy system*. In that case, it may be wise to *migrate* to a new system that offers the similar or extended functionality, without exhibiting the poor quality of the legacy system. The planning to migrate to such a new system should be done as soon as possible, preferably during the service stage' (Mens, [47] 2008).

We see that stagnation is a reality in software engineering but it is also a common denominator along biological evolution. The reason is that a **species** is a state in which evolution has been trapped, stagnated, and they can be counted by millions, most of which belong to bacteria.

We have as yet no experimental reason to suggest that stagnation could be avoided, but we might consider some reasons to think that it could be significantly delayed. In this regard, let propose that the ideology of the daily build can be optimized if it is made to conform to the **biogenesis law**, which reads: 'embriogenesis recapitulates evolution' or 'every living thing in its individual development passes through a series of constructive stages like those in the evolutionary development of the race to which it belongs'(Patten, [57] 1958). This is also known in molecular biology as 'Haeckel's evolutionary paradigm of embryonic development, where an intermediate in the assembly of a complex represents a form that appeared in its own evolutionary history' (Levi, [43] 2008).

Now, working over daily builds always produce an evolutionary history and suitable biogenesis laws, but not necessarily evolvable software. In hindsight, the strategy to quit a stagnating trap is to return to ever more early builds to see what offers a durable evolutionary branch. This is an idea that arises from a consideration of the trees made by taxonomists and

cladists. This property defines the evolvable style of the natural genome: early builds allow for both rapid specializations as well as for long periods of evolution.

Let us propose the next implementation of the biogenesis style of software design: if the client request to design the software encoding for a human being, make a great plan, an architecture, encoding for a great god putting apart an evolutionary path from simple to complex to more complex to final product in such a way that at each step, but specially in early builds, a great span for diversity is guaranteed. Next, fill in the details, beginning from the more simple and continue filling in the intermediate stages. But, please, try not to construct things from the pure beginning, instead, use and reuse software as much as possible. More to the point, before embarking in great projects, compose a great reservoir of tested methods, programs and classes in various hierarchies. This last advise results from the modern studies of horizontal gene transfer, pointing to the fact that, say, horizontal gene transfer could be an ordinary activity in bacteria while less frequent but important events of this nature are already recognized in higher taxa (Choi and Kim, [15] 2007; Bergthorsson et al, [7] 2004). So, we can say that the bacteria world contains the basic protein domain library used in the design of higher structures.

Dear Reader: the most difficult part of the machinery needed to conform to the biogenesis style of software programming is already done and it is right now at your disposition: it consists in the whole library of Java classes of the Java Platform Standard Edition that we are used to and that are thoroughly tested, very powerful and very rich (various thousands of classes), although it is not free of criticisms. This library gives a tremendous combinatorial advantage over low level languages, such as Assembler. The complete Java SE Reference Guide is public domain. The next step is to choose an specific theme to work upon and design a domain specific library of classes. This is another invitation to get part of a team working in a specific theme.

In the lore, a library is called API(Application Programming Interface) which is a set of functions, procedures or classes that an operating system, library or service provides to support requests made by computer programs (Wikipedia). A good API is not easy to design but one can learn how to progress in the deal (Bloch, [10] 2008). A working and very important example is *BioJava*, which is an open-source project dedicated to providing a Java framework for bioinformatics (processing biological data, specially of molecular origin). There are more than 50 publications referencing BioJava (BioJava, [8] 2008).

24 *A transparent style*

The idea of transparency is that one can understand the architecture or enveloping scheme of a piece of code by goggling a bit. A transparent piece of code is really distinct from a ball of mud, which is plenty of void interconnections. So, a good object of study to see transparency in action is gene regulation:

It was discovered by Jacques and Monod, as one can read in his celebrated book *Chance and necessity* that the genome of bacteria contains instructions that regulates the expression of other portions of code. One uses to interpret this fact as a functional implementation of efficiency in resource management.

Now, when one looks at the diagrams of regulation of, say, the level of bilateral development in sea urchin, one gets astonished by the tremendous complexity of regulating, self referencing networks (Hood et al, [32] 2003). Of course, an evolutionary insight is readily

proposed that somehow diminishes the load for its understanding (Howard-Ashby et al, [34] 2006). To get an appreciation of the dynamics of regulation, please, gave a glance at an applet presented by Sneppen et al. ([66] 2008).

The task of understanding a code is very easily hardened if a good style of programming is not observed. To see this, let us consider the next portion of pseudocode that is not executed serially but in parallel (the genome has code that can be executed in parallel, say, cells can reproduce at many places at the same moment, nevertheless embryo development is marshalled by a serial commander):

```
Enable permission for task 1 if signal A is received.
Execute task1 if you have permission.

Enable permission for task 2 if signal B is received.
Execute task2 if you have permission.

Enable permission for task 3 if signal C is received.
Execute task 3 if you have permission.
```

What does this code encode for? It is readily seen that it is regulatory system for executing three tasks. Now, thanks to parallelism, we can rewrite another piece of pseudocode that does the very same work but whose understanding is blurred by a disordered style of encoding, as in the next example:

```
Execute task1 if you have permission.
Enable permission for task 2 if signal B is received.
Execute task3 if you have permission.

Enable permission for task 3 if signal C is received.
Execute task2 if you have permission.

Enable permission for task 1 if signal A is received.
```

Now, let us try to imagine what entanglement can do to a very complex program: it amounts to a sort of encryption. By contrast, it happens that the code of the genome is transparent as in the first piece of pseudocode. That is why many charts in bioinformatics and genomics about gene regulation are self-explaining.

A **transparent style** can be defined as the behavior characterized by the recurrent allocation in neighboring places of those units that have functional dependence or that together form a functional module. This is an style because it encloses the election of options that are not compelling.

It is good for everyone in our community to choose a theme to work upon. Let us say why regulation is a wonderful theme that is worth being chosen. The reason is that regulation can be considered as a prerequisite for life. Let us see why:

It is a fact that every substance gets poisonous it is administered to any alive being in large doses. Say, we can kill the cells of our body if we drink water in exaggerated quantities. This means that a prerequisite for life is to keep ingestion and production of any substance in within appropriate doses. When catalysis is involved, one must avoid noise in the regulation of enzymes at whatever cost it takes. Thus, one must make sure that all molecules are processed or immobilized as fast as they are produced. Hence, regulation must be found at the very dawn of life and in fact that is so.

To enrich your knowledge over regulation, please, give a look at the following site
http://en.wikipedia.org/wiki/Gene_regulatory_network
where one can find an introduction to the theme together with some links.

25 *The style of the genome*

The genome is software and, according to the evolutionary theory, evolution is the software developer responsible for the existence of that software. In this insight, there are no purposes in evolution and hence evolution lacks design and style of design. Nevertheless, a dog walking around us has a genome whose purpose is to enable it to be a dog. In this way, we introduce a posteriori into the evolutionary theory the concept of **purpose and function**.

Our choice is just a formalization of what every evolutionist does at every moment. Just open any book in cell physiology and verify that any enzyme is given the function of catalyzing the reaction it catalyzes and this assignation is automatically pulled back to the corresponding DNA code: it has a function, a purpose, which by the way could be very specific in spite of the fact that this code could be distributed in various exons. The overall envelop of tiny functions like these, a big function or purpose emerges: to be a dog.

Because we have functions and purposes in evolutionary biology, we have now the right to ask: what is the style of the software of the genome and of its design?

We know from the experience of human designers that good software must be functional. In some cases, especially for minor detachable projects, that is all to it. And so, we could say: the style of this code is 'functional'. But in other cases, as in large complex projects, evolvability is a fundamental part of functionality. For human designers, evolvability is tied to a good platform, to a very good programming language and to a very refined style.

The natural genome classify as a proficient, very large and complex project. Besides, it is the epitome of evolution and has a very clean, transparent and organized style. Thus, in terms of human designers, it is a marvel. Let us explain this:

- The genome is proficient, it correctly makes hard things, with ease and expertise. Example: the genome encodes for enzymes. We have developed a technology to use the immune system to synthesize enzymes. The functional products of this technology are called abzymes.So, an an abzyme is an antibody that has been selected for to act as an enzyme. Recent experiments has given some light on the dichotomy between enzymes and abzymes: in first instance we are now sure that life on earth can not be imagined without enzymes: there is an enzyme, orotidine 5' - phosphate decarboxylase, that catalyzes a fundamental reaction, the last essential step in the biosynthesis of pyrimidines, necessary for DNA assembling. The half time of that reaction is, under physiological conditions, as large as 78 million years. The enzyme enhances the rate

of reaction by a factor of 10^{17} (Radzicka et al, [60] 1995).Thus, thinking of life as an acceleration of abiotic processes is wrong: life, as traced by thermodynamics, is created by enzymes, by the genome. Moreover, let us recall that in a free reaction, a given set of reactants can give rise to many different products but enzymes choose just one output. In this way, enzymes allow the existence of a coherent body of reactions that convert a cell into an organism: life, as traced by biochemistry, is created by enzymes, by the genome. Moreover, the affinity of enzymes for their respective transition states 'can reach values as high as 10^{-24} M, far exceeding the 10^{-9} M values observed for the best abzymes' (Stewart et al, l.c., [68] 1995). A factor of 10^{13} separates enzymes from the best abzymes .

- The genome classify as a large, complex project. A short genome might encode for 1000 proteins, each one with 150 amino acids. So, it amounts to the specification of 150000 amino acids. In human terms, this could represent code with 150000 lines. Such a project would very large indeed.

- The genome is the epitome of evolution. This means that the genome is evolution-prone to such a high degree that the official belief of science is that evolution gave rise to the genome. In any case, we just need to look at us, humans, to observe how different we are in spite of the fact that we all have a common descent: this is evolution in flying colors.

- The genome fulfills naturally the law of biogenesis: functional complexity is more easily begotten from functional complexity and at the same time evolvability has been achieved.

- The style of the natural software is so clean, transparent and organized that 50 years of research after the discovery of DNA were more than enough to allow the appearing of a profitable genetic engineering that covers the entire genome from bacterium to the human being.

- It is argued that to design software with more evolvability and lesser proneness to get bugged, the code must be simplified replacing repeated or slightly different pieces of code by, say, parameterized methods (Koschker, 2008). The reason is that if one wants to modify a unit, one must modify as many repeat as there are and this is a bug-prone operation. Nevertheless, the genome contains many repeats and this has caused no problem to its functionality nor to reverse engineering operations required by genetic engineering.

- It has been observed with software produced by humans that every evolutionary branch propitiates its own stagnation: no further evolution is allowed and the product must be abandoned or extincted. This also happens with natural software: the evolutionary arisen of new species is so difficult to observe in vivo and altogether in the lab that one must infer that evolution has been driven to the limit.

Henceforth, we are concluding that the genome as software is a challenging marvel in terms of human designers. High functionality could be expected to be a byproduct of evolution if a very sophisticated and extremely powerful evolutionary environment exists. But, what about transparency and clear organization?

The very first reaction is to think that there is no one single reason as a transparent and clean style could be observed by the genome, were it the result of natural evolution. In effect: big balls of mud are produced in ordinary work of human designers because this is what happens when one only cares for functionality. And functionality is all to natural evolution (artificial evolution could include selection for a given style). Therefore, big balls of mud are what natural evolution predicts.

But on the other hand, a good style of design arises as an emergent property in human design due to the existence of clients that pay whatever it takes for very large projects, say, a windows based operational system for desktops. Successful operational systems are large project with high functionality and whose code is synthesized by developers of the best quality with a lot of experience in surpassing the big ball of mud (most of which know nothing about the possibility of using evolution to design software). So, we might say that the clean, organized and transparent style of programming arises as an emergent property where high complexity is rewarded. We feel that the extrapolation of this conclusion to the realm of the genome is a good idea: it is plainly mechanistic and defines a crude challenge for our community.

The problem with this theory of the emergence of a transparent style as an answer to selection for high complexity is that the probability of a good style amidst all possible styles is nearly zero. The reason is that in nature there are also pressures for time and budget, i.e. raw material, space and opportunities. So, big balls of mud are the natural expectancy for the natural genome. This reply makes the theory of the emergence of a clean style when there is selection for large complexity more interesting.

What do our simulations say about the emergence of clean styles?

We already have material from our simulations to discuss the theory of the emergence of clean styles when selection of large complexity operates. In fact, we studied in Vol II the possibility of using evolution to design a parallel binary adder. We notice that if the combinatorial bases correspond to low level instructions, evolution is fruitless. But, if one furnishes an evolutionary environment with a three level hierarchy, in which the combinatorial bases may include methods instead of instructions then evolution begins to produce fruits, although some tens of hours of execution allow success for very simple inputs.

We have denied in that form the possibility that a low level language could be a successful platform for evolution and by contrast we have recognized that a high level language, with super alphabets, can do it. If we recognize that a piece of code written with super alphabets looks more transparent that pieces of code written on alphabets, then we are saying that transparency is a prerequisite for evolution. Can we somehow revert this conclusion to show that transparency could be a result of evolution? We have here another challenge for our community.

Let us pay attention to another fold of the negative relation of interconnectivity and evolvability.

2.5 The NK model

This section has undergone strong modifications for the second version of this volume. Misconceptions were corrected.

Evolution always works in nature: just look around and verify how different are we one from another. Or, you can create a race of say, wheat, by just traversing a field and looking for special traits. But, the theory of the origin of species by an evolutionary process lacks, after more than 150 years of research, transparent and powerful experimental evidence. What is perfectly clear is that at present the evolutionary origin of species is halted: the most we have made in the lab is to make some initial steps into the realm of incipient speciation. From other side, we know that the genome is software and from the research in software engineering we do know that any piece of software evolves as much as it can, first rapidly, then slowly, then evolution is halted and then, most of times, the software disappears from the trade. This is the **evolutionary cycle of software**.

Stagnation is then the last state of any branch of software evolution be it in nature or in software engineering. Our explanation is that this parallelism may be explained by the same simple reason: stagnation is the result of the incapability of the computing capacity of the system (the genomes of living beings or software engineers + computers) to achieve a better local peak in a landscape dominated by a high density of uncorrelated peaks.

Let us explain this terminology:

There are many programs or classes that solve the same problem. Notorious recurrent differences among them could be classified as styles. It has always been believed that some styles are better than others (at least for medium to high level of complexity of programs). So, one could eventually assign a fitness to a given program, which gives a real number as output, the higher the fitter. In that way we get a function from the space of programs into the real numbers: the domain of the function is assimilated to a plane and the fitness function to a highness over each point of the plane. We have a mountain and in general we have a **landscape**.

There are many forms of assigning a fitness function to software programs. In software engineering, such forms are called measures. Our present problem is to find a simple measure that predicts the evolutionary cycle of software and in particular the undesirable state of stagnation (not at the optimal point). In this regard, let us pay attention to the degree of interconnectivity of its compounding units. Software engineering have the official measures of fan-in and fan-out complexities that measure the number of variables that represent non-redundant inputs and outputs respectively. The expected thesis is the following: the interconnectivity load of software determines the degree of evolvability. Simple software, with few interconnections always can be modified, enlarged, evolved (while it does not become too complex) but software with a lot of interdependence could be modified at a high cost and eventually no significant improvement can be further achieved. This incapability is not absolute by just a result of time and computing power tradeoffs.

We will justify our asseveration with a simulation of the NK model, which is by now part of the evolutionary lore. This model was proposed by Kauffman (l.c., [38], 1995), who was investigating the dependence of evolution of gene networks on different types of landscapes.

Suppose we have a library of methods that are named with upper case letters, just as we did in Volume II in regard with grammars. So, DSFVR, represents an execution in which method R goes first, followed by method V and so on. Our purpose is to find a program that fits a certain specific function. When we try diverse combinations and we test them, we can say: this program is good, that is bad, that other is an incipient solution and so on.

More formally, we can ascribe a fitness, a numerical value, to each program. But, can we say that the overall fitness of the program depends on the fitness of individual methods or of the different coalitions or interconnections among them? The NK model is an extremely simple idealization in which the fan-in and fan-out measures are changed by the clustering they cause. It is a great simplification. Nevertheless it captures the load that the interconnection represents for evolvability, for the application of evolution to the design of software.

Let us give an example of a program that is extremely simple because its methods are not interconnected at all: the program has 26 methods, each one prints an upper letter. We can demand to print a string 20 chars long. It is straightforward to see that all letters have the same fitness equal to 1/20 and that the overall fitness of any string 20 chars long adds up to 1. We have given an example of simplicity because each method helps the overall purpose with 0 interactions with other methods. Geometrically, we have here a landscape that has only one peak whose domains of attraction is all the universe, so one can begin from anywhere, gather small change and triumph as in fairy tales.

Suppose now that we must guess the number 3456738 that guards a ciphered door. The door responds all or nothing, it gets opened or not. In this case we have that the string 3456738 has perfect fitness while all other strings and substrings have fitness zero. This problem represents the highest case of complexity because you cannot get approximate solutions that eventually can be refined. Geometrically we have a landscape with a needle, it is singled peaked but the domain of attraction that allows gathering change is just the perfect solution. And what about intermediate cases? Problems with intermediate complexity have peaks with not null domains of attractions most of which are suboptimal.

Following Kauffman, his N0 model describes a simple situation: it is when the fitness of any string is the sum of the fitnesses of its chars which form no clusters with neighboring chars. Example: $f(0) = 1$ and $f(1) = 2$, then $f(1001101) = f(1) + f(0) + f(0) + f(1) + f(1) + f(0) + f(1) = 2 + 1 + 1 + 2 + 2 + 1 + 2 = 11$. The problem is to find a string with maximal fitness. It is 1111111 whose fitness is 14.

The next level is the N1 model, in which the fitness of a string is derived from the fitnesses of the substrings with 2 chars. This is a N1 model because each char is accompanied with 1-other-char. Example:

Let us consider, for the sake of simplicity, that we have only two methods, to print 0 else 1, so a program can be named by a binary string, say, 10010. For a certain task we have the following map of partial fitnesses: $f(00) = 1$, $f(01) = 2$, $f(10) = 1$, $f(11) = 3$. Then the fitness of the (N=5)-string 10101 is, by definition equal to $f(10) + f(01) + f(10) + f(01) = 1 + 2 + 1 + 2 = 6$. (If all string are circularized, the last char will be not orphan. For instance, in the present case the term $f(11)$ must also be added. That defines the **NK-model** of fitness or landscape). The question for optimizers is: find the string that optimizes the fitness function. This is clearly a problem that can be given to evolution to be solved but we will not pay attention to it here because we are interested in the subtle question of evolvability.

Now, how are we going to propose predictors of evolvability? The natural claim is that ruggedness is a predictor of evolvability: a rugged landscape generates a difficult problem while a smooth landscape will correspond with an easy problem. So, we plan to measure the fitness along a random walk to see what we get: simple problems will produce walks with a rather smooth paths while complex problems will be filled with ups and downs of short

range. Let us see.

26 ***The code*** *We enable a comparison in regard with ruggedness of the 5-0 and 5-1 Kauffman models.*

```
 /* Program C26, NKmodel
 * To understand why any branch along
 * software evolution ends in stagnation,
 * we study the NK model.
 * In this program we study the N0 model.
 *
 * 'Any branch' sounds like 'a random walk'
 * along a given space. So, we program a random walk.
 *
 * In this model, we represent a program by a string,
 * say, DSFVR, represents an execution in which method
 * R goes first, followed by method  V and so on.
 * The  problem for optimization is to find a program that fits
 * a certain specific function. That is not our problem here:
 * we pay attention to evolvability, so,
 * we study the ups and downs of the
 * fitness fucntion along a random walk along
 * the space of programs, i.e., of strings with
 * the same length.

 * This program proposes a N1-model in which subsequences
 * of two chars are assigned fitnesses from which the fitness
 * of every -chars strings can be calculated. Specifically:
 * f(00) = 1, f(01) = 2, f(10) = 1, f(11) = 0.
 * So, the fitness of 10101 is, by definition equal to
 * f(10) + f(01) + f(10)+ f(01) = 1 + 2 + 1 + 2 = 6.

 * The program reports successive   peaks
 * along a random walk.
 * The Hamming distance among successive peaks
 * is also reported.
 */
package ejvol3v2p;

import java.util.HashMap;
import java.util.Map;
import java.util.Random;
```

```
public class NKmodel {


    // Turn on   the random generator
    static Random r = new Random();

    //N = number of methods per program
    private static final int N = 5;

    //Maximum number of steps in the random walk
    private static final int NSTEPS = 500;

    private static  Map fitnessTable50;
    private static  Map fitnessTable51;



    //Hamming distance among STATEs
    private static int d;
    private static final boolean TEST = false;


    //A fitness table is initialized for the model 5-0
    private static void initializeTable50() {
        fitnessTable50 = new HashMap();
        fitnessTable50.put("0", 1. );
        fitnessTable50.put("1", 2. );
    }


    //The fitness is the sum of individual fitnesses
    private static double fitness50(String s)
    {
        double f = 0;
        String d ;
        for(int i = 0; i < N-2; i++)
        {
            d = s.substring(i,i+1);
            f = f + (double) fitnessTable50.get(d);
            if (TEST)
                System.out.println("d = " + d + " -> "
                +      (double) fitnessTable50.get(d));   }
        return f;
    }
```

```
private static void initializeTable51() {

    fitnessTable51 = new HashMap();
    fitnessTable51.put("00", 1. );
    fitnessTable51.put("01", 1.5 );
    fitnessTable51.put("10", 1.5 );
    fitnessTable51.put("11", 2. );

}



private static String newStart() {
    String a = "";
    for (int m = 0; m < N; m++) {
        int l = r.nextInt(2);
        a = a + l;
    }
    return a;
}




////The fitness is the sum of 2-cluster fitnesses
private static double fitness51(String s)
{
    double f = 0;
    String d ;
    for(int i = 0; i < N-2; i++)
    {
        d = s.substring(i,i+2);
        f = f +  (double) fitnessTable51.get(d);
        if (TEST)
            System.out.println("d = " + d + " -> "
            +      (double) fitnessTable51.get(d));   }
    return f;
}



//The boolean vector that represents a STATE
//is mutated in  a number of  places = mode.
private static String mutation(String s) {
    //System.out.println("\ns in = " + s);
    int m = r.nextInt(s.length());
```

```
        char ch = s.charAt(m);
        if (ch == '0') ch = '1';
        else ch = '0';

        s = s.substring(0,m)  + ch + s.substring(m+1);

        //System.out.println("m = " + m + " s out = " + s);
        return s;
    }



    //A random walk over 5-strings is enabled
    private static void wander() {
        System.out.println("String --> fitness50 + sign of change"
                + "\n+ fitness51 + sign of change");
        double oldFitness50;
        double newFitness50;
        double oldFitness51;
        double newFitness51;
        char sign50 ='k';
        char sign51 = 'v';
        double totalVariation50 = 0;
        double totalVariation51 = 0;

        String s = newStart();
        for (int step = 0; step < NSTEPS; step++) {
            oldFitness50 = fitness50(s);
            oldFitness51 = fitness51(s);

            s = mutation(s);
            newFitness50 = fitness50(s);
            newFitness51 = fitness51(s);
            if (newFitness50 > oldFitness50) sign50 = '+';
            if (newFitness50 == oldFitness50) sign50 = '=';
            if (newFitness50 < oldFitness50) sign50 = '-';
            if (newFitness51 > oldFitness51) sign51 = '+';
            if (newFitness51 == oldFitness51) sign51 = '=';
            if (newFitness51 < oldFitness51) sign51 = '-';
            totalVariation50 =  totalVariation50 + Math.abs(newFitness50 - old
            totalVariation51 =  totalVariation51 + Math.abs(newFitness51 - old
            System.out.println(s + " -> "
                    + fitness50(s) + sign50 + "   "
                    + fitness51(s) + sign51);
        }
```

```
        System.out.println("totalVariation50 = "  + totalVariation50);
        System.out.println("totalVariation51 = "  + totalVariation51);
    }

    public static void main(String args[]) {
        initializeTable50();
        initializeTable51();
        wander();
    }
}//End of Program C26, NKmodel
```

27 ***Exercise*** *Play with the code and try to understand it. Play enough to agree else disagree with the conclusion of the Author: where the 5-0 model sees a plane path, the 5-1 model sees a sloping one and that is why the 5-1 model produces a more rugged landscape than the 5-0 model. In evolutionary terms: the 5-0 model is more akin to neutral changes than the 5-1 model. So, there are many variants that are equi-functional in the 5-0 model as in the renown neutral theory of molecular evolution by Kimura.*

28 ***Exercise*** *Add to previous program a comparison with a 5-2 model in regard with ruggedness. Modify the fitness function in order to leave no orphan at the end of the strings: close them circularly. Example:* $f(10101) = f(10) + f(01) + f(10) + f(01) + f(11) = 1+2+1+2+3 = 9$. *Answer*

29 ***Exercise*** *Develop the code to study the behavior of fitness along a random walk for the general NK model. Answer*

30 ***Challenge*** *The idea is that when K grows, when interacting clusters grow bigger, the corresponding landscape is rugged and evolvability diminishes. Think of having two persons agreeing rather than 123. Develop a GA to test this idea. If you succeed, you have verified that the NK model is a tunable rugged fitness landscape that predicts complexity. If you fail, you have a reason to reject one of the most cherished belief in the trade.*

31 ***Stagnation is not a duty***

The problem behind the scenes is evolvability and stagnation. Now, let us highlight the fact that the registered stagnation is not due to have finding the highest peak across the landscape but to the computing incapability to make exhaustive scans: stagnation mean just having found a local maximum. Thus, stagnation is not a duty but a result of insufficient computing capabilities: true for us that work with a desktop, true for every computing system no matter how super it could be, true for every software design team. This is also true for natural evolution: natural evolution has arrived to stagnation and we are not capable of registering the contemporary arising of new species. By contrast, genetic engineering has abundantly shown that the genome admits modifications of many kinds but nature has no access to so high level of intelligence and so evolution looks like halted.

32 ***Challenge*** *Appropriately modify the previous programs and carry out suitable experiments to decide whether or not the next conclusions about the NK model forwarded by Kauffman in the aforementioned book are valid or not. In this regard, we say that a landscape is rugged when a random walk observes a large variation of peaks and walleyes over short stretches of path:*

- *For fixed N, as K gets higher, the peaks decrease in fitness, while neighborhood around the peak becomes more rugged (figure 8.6 of Kauffman's book).*
- *For fixed N, for very low K, high peaks cluster near one another, but when K increases, high peaks have no tendency to be near one another (figure 8.7 of Kauffman's book).*
- *For moderately rugged landscapes (N= 24, K = 2), the highest peaks are the ones that can be scaled from the greatest number of initial positions. The landscape looks like a plane desert with one pyramid in the middle.*
- *The waiting time of a state is defined as the number of tries or steps to find a fitter neighbor. As ruggedness increases, waiting times increase while the fraction of fitter neighbors diminishes.*

33 ***Exercise: Where is science?*** *The relation of palebiology with the NK model of intra-connectivity of software could be done along the following trend of ideas:*

Let us consider mammal evolution as it is traced by the fossil record. Decide how many genes could be responsible for the information carried by its skeleton: this is N. Assume some value for K, the number of genes whose effects somehow interferes with those of other genes. Use the program developed in exercise 32 to prove that successive peaks are connected by valleys of immense depth. Conclude that there must exist abundant fossil record pointing to every sort of monstrosities among fossils. Find in the archaeological literature a verification of your prediction else explain why it is not found. Answer

34 ***Challenge*** *A very interesting idea presented in this chapter is the following: a transparent style emerges in an evolutionary environment in which evolvability leading to very high complexity is rewarded. Please, do something to convert this wonderful idea into a laboratory. Otherwise, how can we pass from mythology to science?*

2.6 Conclusion

The genome is software and as such it can be compared with the software produced by humans. In this regard, the natural genome classifies as a proficient, very large and complex software project, it is the epitome of evolution and has a very clean, transparent and organized style. Thus, in terms of software engineering, it is a marvel. To explain why the genome is a representative of a transparent style, we were led to consider the possibility that a transparent style emerges in an evolutionary environment in which evolvability leading to very high complexity is rewarded. The challenge is to convert this nice proposal into a laboratory for science. Furthermore, evolution, apart from being a process, is also an style of software design that is very modern and with a great future. It is characterized by the next directive: *if the client requests the design of the software for a human being, make a great plan, an architecture, for a great god putting apart a path from simple to complex to more complex to final product in such a way that at each step, but specially in early builds, a great span for diversity is guaranteed. Next, fill in the details, beginning from the more simple and continue filling in the intermediate stages. But, please, try not to construct things from the pure beginning, instead, use and reuse software as much as possible. More to the point, before embarking in great projects, compose a great library of tested methods, programs and classes in various hierarchies.*

Chapter 3

Evolution as a necessity

`Not chance but necessity.`

35 ***Motivation*** *Evolution is a software developer and it is also a style or paradigm of software design. But, How did evolution appear? One immediate answer is: life with its evolutionary potentials appeared by mere chance. The problem is that to believe that theory, one needs to invoke a happy and long succession of events of extremely low probability. Henceforth, in the market place of ideologies, this is not an attractive explanation. Can science provide another more reliable explanation to the origin of evolution?*

36 ***Purpose*** *Our aim is to review some few of the ideas of Kauffman([38] 1995), according to which life, in all its splendor and evolutionary possibilities, arouse not by chance but by necessity: there are concrete mechanisms plus initial conditions in which life and evolution are expected to emerge with a great probability. That is why he claims that we are at home in the universe. We reproduce some aspects of his work and add an ingredient that makes thing more realistic.. .. i.e. more improbably.*

3.1 The Networks of life

We will study right now a model proposed by Stuart Kauffman, which appears in the first chapters of his book *At home in the universe (l.c., [38] 1995)* that offers an explanation to the origin of life and evolution. This model is interesting in that it predicts that given some initial conditions, life and evolution was expected to arise with certainty. Thus, the model enables a liberation against the tyranny of chance + religion. It is usual to say, according to this and similar models, that life arouse by **self-organization**, meaning that there is no design but only things and dynamical processes and structures that were assembled by mere chance without a purpose, just following the laws of matter. More concretely:

Life, according to the first pages of that book, is characterized by some few points:

- The first ingredient of life is catalysis: *'At its heart, a living organism is a system of chemicals that has the capacity to catalyze its own reproduction'* (p 49).
- The second one is autocatalysis: *'Two dimer molecules, AB and BA, are formed from two simple monomers, A and B. Sice AB and BA catalyze the very reactions that join*

As and Bs to make dimers, the network is ***autocatalytic****: given a supply of "food" molecules (As and Bs), it will sustain itself'* (caption to figure 3.1, p49).

- The third is an initial condition that furnishes a sufficiently diverse mix of molecules that accumulates somewhere: *'What I aim to show is that if a sufficiently diverse mix of molecules accumulates somewhere, the chance that an autocatalytic system - a self-maintaining and self-reproducing metabolism-will spring forth becomes a near certainty. If so, then the emergence of life may have been much easier than we have supposed.'* (p 50).

- The fourth is a thermodynamic flux, which results from the combustion of some input material and its reuse, in the form of free energy and building blocks, for the purpose of self-maintenance, grow and reproduction: *'Catalysis alone, however, is not sufficient for life. All living systems "eat": they take in matter and energy in order to reproduce themselves. This means that they are.. .. open thermodynamic systems.'*. When there is a flux, the system is said that it is out of equilibrium or in nonequilibrium. When a thermodynamic flux is maintained, dynamic structures may appear, say, the silhouette of the flame, or a non fading wave.

- In conclusion,*'life is the natural accomplishment of catalysis in sufficiently complex nonequilibrium chemical systems'* (p51).

Our purpose now is to show the mechanistic elaboration of these ideas. We begin with a seemingly unrelated study on the degree of connectivity of a set of cities when roads are defined at random.

37 *The giant component*

Let us consider an ensemble of n cities, some of which are connected two by two with bidirectional ways, rows or links. The component associated to city i is formed by taking all cities that are joined to i by paths of any length. Beginning from city i, the most you can visit is the component of i.

Our problem is to understand how probably is the appearing of a giant component that covers most of the cities. By design, one can solve the problem in many diverse ways, for instance: we number the cities and we build a road from city i to city $i+1$ and from the last one to first. In that way, we have a single component: one can make a round tour through all cities. So, design is certainly good to form a giant component that covers the whole world. And what about randomness?

Let us consider now a random linking (when we have no design at all): Which is the maximum number of cities that one can visit? The next program offers an opportunity to begin the study of that question.

```
/* Program C37, Cities.
A number of cities is joined by random links.
Which is the maximum number of Cities that
```

```
one can visit?
A component consists in a set of cities that
are somehow interconnected.
To answer the posed question,
we must find the largest component.
To that aim:

The random links are recorded in a boolean matrix LINKS.
LINKS[i][j] = true if there is a link, road or thread
between city i and city j.
LINKS are bidirectional, hence LINKS[j][i] = true.
One finds all the components in this system
and then picks up the largest one.
To find a component, a first link (i,j) is found
and its Cities are annotated to COMP[].
Then, all Cities connected to city i
and to city j are annotated to COMP[].
And so on until all Cities of COMP[] have been
investigated for connections.
The algorithm produces repetitions,
henceforth COMP[] must be simplified.
To avoid interference with the search of a new component,
annotated links are blanked.
A new component is searched for.
And so on until no first link can be found.
At that moment, the work is finished (false).
 */
package ejvol3v2p;

import java.util.Random;

public class Cities {

// Turn on   the random generator
    static Random r = new Random();

    private static int cities;
    private static int randLINKS;
    // p= nlinks/cities
    private static double p;
    // LINKS keeps a record of the interconnecting links
    private static final boolean LINKS[][]
            = new boolean[1000][1000];
    //COMP keeps   a list of the Cities in a COMPonent
    private static final int COMP[] = new int[1000];
    //COMPBIG keeps the  list of the biggest COMPonent
```

```
    private static final int COMPBIG[] = new int[1000];
    //length is the number of elements in the COMPonent
    private static int length;
    //length is the number of elements in the big COMPonent
    private static int lengthBig;
    //scanned is the number of scanned rows
    private static int scanned;
    //the matrix is scanned row by row but not in order.
    private static int row;
    //If work = true, there is more work to be done.
    private static boolean work;

//The interconnection matrix is generated at random
    private static void generateMatrix() {
        //Initialization in false = no links
        for (int i = 0; i < cities; i++) {
            for (int j = 0; j < cities; j++) {
                LINKS[i][j] = false;
            }
        }
        //generation of random links but without (i,i) links
        for (int counter = 0; counter < randLINKS; counter++) {
            int l = r.nextInt(cities);
            int m = r.nextInt(cities);
            while (m == l) {
                m = r.nextInt(cities);
            }
            LINKS[l][m] = true;
            LINKS[m][l] = true;

        }
    }

    //Prints interconnection matrix
    private static void printMatrix() {
        System.out.println();
        System.out.println("Interconnection matrix");
        System.out.print("     ");
        for (int i = 0; i < cities; i++) {
            System.out.print(i % 10 + " ");
        }
        System.out.println();
        for (int i = 0; i < cities; i++) {
            System.out.print(i % 10 + "    ");
            for (int j = 0; j < cities; j++) {
                if (LINKS[i][j] == true) {
```

```
                System.out.print(1 + " ");
            } else {
                System.out.print(0 + " ");
            }
        }
        System.out.println();
    }
    System.out.println();
}

//Scans the matrix to find the first link
private static void findFirstPair() {
    length = 0; //number of Cities in the COMPonent
    //remain1,2 = gates that get closed when
    //the first link is found.
    boolean remain1 = true;
    boolean remain2 = true;
    for (int i = 0; remain1; i++) {
        for (int j = 0; remain2; j++) {
            if (LINKS[i][j] == true) //first link found
            {
                remain1 = false; //exit the for loop
                remain2 = false;
                length = length + 1;
                COMP[0] = i; //connected Cities are recorded
                COMP[1] = j;
                //Once recorded, links are blanked that
                //a new COMPonent could be found.
                LINKS[i][j] = false;
                LINKS[j][i] = false;
            }
            if (j == cities - 1) {
                remain2 = false;
            }
        }
        remain2 = true;
        if (i == cities - 1) {
            remain1 = false;
        }
    }
    if (length == 0) {
        work = false;  //end of all works
    }
}

//Scans row i to find all links to city  i
```

```
    private static void scanRow(int i) {
        for (int j = 0; j < cities; j++) {
            if (LINKS[i][j] == true) {
                LINKS[i][j] = false;
                LINKS[j][i] = false;
                length = length + 1;
                COMP[length] = j;
            }
        }
        scanned = scanned + 1;
    }

    //Multiple annotations are erased
    private static void simplify() {
        //help[i] = true if city i belongs in the COMPonent
        //otherwise it is false
        boolean help[] = new boolean[1000];
        for (int i = 0; i < cities; i++) {
            help[i] = false;
        }
        for (int i = 0; i <= length; i++) {
            help[COMP[i]] = true;
        }
        //the Cities of the COMPonent are rewritten
        //in order and without repetitions.
        int counter = 0;
        for (int i = 0; i < cities; i++) {
            if (help[i] == true) {
                COMP[counter] = i;
                counter = counter + 1;
            }
        }

        length = counter - 1;

    }

    //The matrix is scanned for a COMPonent
    private static void scan() {
        scanned = 0;

        while (scanned <= length) {
            row = COMP[scanned];
            scanRow(row);
        }
        /* System.out.print("Not ");
```

```
  printComponent(); */
        simplify();
    }

    // A COMPonent is printed
    private static void printComponent() {
        // System.out.println("simplified");
        for (int i = 0; i <= length; i++) {
            System.out.print(COMP[i] + "  ");
        }
        // System.out.println();

    }

    // A COMPonent is found
    private static void findComponent() {
        findFirstPair();
        scan();
        if (scanned > 1) {
            printComponent();
        }
        //The largest COMPonent is kept in COMPBIG
        //whose length is lengthBig
        if (length >= lengthBig) {
            for (int i = 0; i <= length; i++) {
                COMPBIG[i] = COMP[i];
                lengthBig = length;
            }
        }
        System.out.println();

    }

    public static void main(String args[]) {
        cities = 10;
        randLINKS = 7;
        double c = cities;
        double j = randLINKS;
        p = j / c;
        System.out.println("Number of cities = " + cities);
        System.out.println("Numb rand links  = " + randLINKS);
        System.out.println("prop links/cities  = " + p);
        generateMatrix();
        printMatrix();
        System.out.println("Components");
        System.out.println();
```

```
        work = true;
        while (work) {
            findComponent();
        }
        System.out.println("Biggest component");
        for (int i = 0; i <= length; i++) {
            COMP[i] = COMPBIG[i];
            length = lengthBig;
        }
        printComponent();
        System.out.println();
        System.out.print("Number of connected elements = ");
        System.out.println(length + 1 + " of " + cities);
    }
}//End of Program C37, Cities.
```

38 ***Exercise*** *Run the code, play with it and try to understand it. Try to grasp a first impression on the number of cities that belong in the largest component. To that aim, keep the number of cities constant, say, 20. Next, vary the number of links or roads from 10 to 50 and record the number of cities in the largest component.* Answer

39 ***Exercise*** *Keep the number of cities constant and equal to 400. Next, vary the number of links or roads from 0 to 600 and record the number of cities in the largest component. What do you observe?* Answer

We see that design and randomness differ not significantly in terms of the size of the largest component of a set of cities. The reason is that if one can afford as many as n roads, then it is not a great effort to arrive to a total of 3n roads.

Most possibly, one could think that design and randomness must differ in many respects, so the following comment is important: when the number of cities is small, design could unite the cities by order or would try to minimize the length of the roads, while randomness will reveal itself in a spaghetti soup like topology. But when the number of cities grows, design is ever less capable of keeping a given style or optimizing a given criterion and so randomness and design are much less distinguishable (this is the reason of why most large programs look like a big ball of mud).

Let us summarize the results of the last exercise in the form of a theorem, whose mathematical proof can be found in the books of random graphs.

40 ***Fundamental theorem of self-organization*** *Let us consider a group of n cities that is joined by m roads. A component is a maximal set of cities that are somehow interconnected. When n cities are joined by design, one can form a giant component that includes all cities using n links or roads. By contrast, when links or roads are dictated by chance, the largest component embraces more than 50% when we use n links. If the number of links is 3 times the number of cities, more that 99% of the cities are united in a giant component. This result escalates very well: if one enlarges the problem by ten or hundred, the conclusion remains.*

In short, design and randomness differ not significantly in terms of the size of the largest component of a set of cities.

We need to explain now why this theorem is presented as the fundamental stone of self-organization. The reason is this: let us consider a given chemical system that thanks to its reversible reactions can be in any of n states differentiated just by the name of their chemical components. Two states are linked if there is a single one step reaction that joins them. When the number of states or nodes is great and there are also many roads or links, one can make sure that many closed pathways begin to appear and proliferate. Now, if we consider a group of reversible chemical reactions, a closed pathway or cycle defines a self-sustaining metabolism, which is the fundamental piece of an ecological system.

Since it is difficult to accept the existence of a large body of reversible chemical reactions, a fundamental feature in the previous discussion, let us comment on this.

The direction of a chemical reaction depends on various factors: the first is the temperature, which is the main source of life of a chemical reaction. The second is the energy delivered or absorbed by the reaction in vacuum. The third is the concentrations of reactants and of products. If a reaction is exo-energetic, producing free energy, it tends to go down but one can reverse the direction by augmenting the concentration of products that will react in the contrary direction propelled by the global effect of temperature. Also, one can reverse the natural direction of a reaction by coupling it to other reactions that are exo-energetic but that go in the contrary direction.

So, reversibility is not a mysterious matter but a plausible abstraction and that is why we tolerate it.

In conclusion, when we have a large body of chemical reactions, the arisen of a closed pathway or cycle defining a self-sustaining metabolism, an ecological milieu, is highly probably. Subsamples of that great world could eventually get encapsulated giving origin to a *first version of life*: in such a complex chemical milieu as one needed in here, the existence of molecules that could form vesicles is guaranteed (Deamer, 2008). Once a great catalytic machinery has been set up, it would be easier to imagine the successful incorporation of RNA or DNA with genetics and evolution. As an alternative, we have the theory of the hypercycle, proposed by Eigen and Schultz (1979), in which enzymes and RNA coevolve from the very beginning.

Our model and similar ones are proposed here as candidates to support the idea of the natural origin of life in all its splendor. It could help to assess its explanative power to consider that the modern world with its global market of goods and services is also described by this model and, moreover, the global market and the proposed self-organized chemical autocatalytic system are somehow isomorphic. The reason is that a fabric uses various items to fabricate some products, an action that is similar to a chemical reaction in which some reactants interact to produce a product (see chapter 12 of the Book of Kauffman). Now, what is the equivalent of catalysis? Its role is played by the intelligence of designers that is used to implement some technological idea to fit the market needs of the moment. Let us highlight that inventors and engineers fill in one step or link of the great system and that no one of them knows how the general market behaves and how it will be affected by their products. This shows that intelligence functions at the microscopic level and randomness, self-organization,

at the higher one.

41 *More realism*

One may consider that reversibility is too a cheap abstraction of chemistry and that the existence of cycles in a complex chemical random world, giving a background ecosystem, is plainly possible but must be more judiciously justified. So, let us refactor our previous simulation to include:

- Unidirectional reactions.
- A detector of cycles to learn how the probability of its formation behaves.

We also have the next problem: in the generating method of random links used in program 27, the same link can be generated various times, so there is the possibility of getting misled in the counting of effective links. This shall not be. So, we must fix this bug.

The solution of this problems is as follows:

To simulate unidirectional reactions is sufficient to generate only the link (i,j) but not (j,i).

To make sure that random generation of links produces different links, one must include a checking procedure. In this regard, one must take into account that when the number of links grows, a moment arrives when positing a new link is quite difficult because most possible links are already enabled. So, we use the next trick: to posit 20 links in a 5×5 matrix is equivalent to take out 5 links in a matrix that previously had been replenished in links.

And what about a detector of cycles?

We encode links in a boolean matrix, Links:

Links(i,j) = true means that the link is enabled;

Links(i,j) = false means that the link is disabled.

We depart from a matrix with all terms in the diagonal (i,i) in false, i.e., with no futile cycles. But, if we find the n-th power of the matrix Links, and if we get an entry (i,i) with a value of true, we have detected a cycle.

To fix ideas, let us consider the next situation:

Number of cities = 5 Numb random links = 5

The original interconnection matrix Links, with numbered rows and columns, is the following

```
Links
      0  1  2  3  4
0     0  0  0  1  0
1     0  0  0  0  1
2     0  0  0  0  0
3     0  0  0  0  1
4     0  1  0  1  0
```

Let us observe that we have a link from 1 to 4, and another form 4 to 1, i.e., we have a cycle of length 2, that involves cities 1 and 4. By the same token, we have a link from 4 to 3, and another form 3 to 4, i.e., we have another cycle of length 2 that involves cities 4 and 3. In conclusion, we have cycles or order 2 around cities 1,3 and 4. These cycles are automatically detected if we find the second power of the matrix Links:

```
Links ^ 2
     0 1 2 3 4
0    0 0 0 0 1
1    0 1 0 1 0
2    0 0 0 0 0
3    0 1 0 1 0
4    0 0 0 0 1
```

This matrix has one over the diagonal in the entries 1, 3 and 4. This is interpreted as having cycles of order 2 that include these cities or nodes.

If we find the third power, we find no cycles, as expected:

```
Links ^ 3
     0 1 2 3 4
0    0 1 0 1 0
1    0 0 0 0 1
2    0 0 0 0 0
3    0 0 0 0 1
4    0 1 0 1 0
```

We have found a cycle of order two connecting 1 and 4, and another connecting 4 and 3. Henceforth, we must detect cycles of order 4 around 1, 3 and 4. To verify the prediction, let us calculate the fourth power of the matrix Links: it must have ones over the diagonal at the entries 1, 3, 4 and zero elsewhere over it. Let us see:

```
Links ^ 4
     0 1 2 3 4
0    0 0 0 0 1
1    0 1 0 1 0
2    0 0 0 0 0
3    0 1 0 1 0
4    0 0 0 0 1
```

So, the prediction is verified and we have found all cycles. Of course, we can continue with more powers of Links but nothing new would be found in this case.

It could be worth realizing why the procedure of multiplication functions as a cycle detector. The reason is at the original matrix encodes direct links only. To unveil indirect links two links apart, we must join all possible pair of continuous links. And to know whether or not we have found a cycle, we must verify that the path with 2 links ends exactly where it began: that is why we must look at the diagonal. And so on.

How is this done automatically?

If we want to know whether or not there is an indirect link two links apart between nodes i and j, we must check for all intermediate possible nodes l, and next check the connections (i,l)

AND (l,j). So, we have a logical AND for every pair of couples (i,l) and (l,j). Now, to get a positive answer about a path of length two, it is enough that just one of those composite paths be enabled. So, this corresponds to a logical OR. All this is arranged into a mathematical formalism that is called boolean multiplication of boolean matrices.

42 ***The code*** *The next program says when a node may form part of a cycle:*

```
/* Program C42, Cycles.
A number of cities is joined by
unidirectional random roads.
Can one make a round trip?
To answer this question, we must study
the arising of cycles in a random graph
with unidirectional links.

To that aim:

The random links are recorded in the boolean matrix LINKS.
LINKS[i][j] = true if there is a link, road or thread
between city i and city j.
LINKS are unidirectional.
LINKS[i][i] = false, to avoid futile Cycles.
All links are guaranteed to be different.
When the number of links is too high,
the probability of picking at random a new different
link  tends to zero, henceforth
a negative refilling is adopted.

Cycles are found using the k-th power of
the boolean matrix LINKS.
If the entry [i][i] of LINKSˆk is  true,
there is a cycle of length k that includes node i.

 */
package ejvol3v2p;

import java.util.Random;

public class Cycles {

// Turn on   the random generator
    static Random r = new Random();

    private static int cities;
    private static int randLINKS;
    private static int nLINKS;
    // p= nlinks/cities
```

```
    private static double p;

    private static final int MAXCITIES = 200;
    // LINKS keeps a record of the interconnecting links
    private static final boolean LINKS[][]
            = new boolean[MAXCITIES][MAXCITIES];
    //The half of the max number of links
    private static int half;

    //Initializes a boolean matrix
    // No terms of the form (i,i)
    //No repeated links.
    private static void fill() {
        for (int counter = 0; counter < nLINKS; counter++) {
            int l = 0;
            int m = 0;
            boolean equal = true;
            boolean repeated = true;
            // || means logical or.
            while ((equal) || (repeated)) {
                l = r.nextInt(cities);
                m = r.nextInt(cities);
                /*Equivalent to
                if (m == l) {
                equal = true;
                } else {
                equal = false;
                 */
                equal = m == l;
                /*
                //Equivalent to
                if (LINKS[l][m] == true) {
                repeated = true;
                } else {
                repeated = false;
                }
                 */
                //repeated = (LINKS[l][m] == true);
                repeated = LINKS[l][m] == true;
                /*System.out.println( l + " " + m + equal
                + " " +  repeated );
                 */
            }
            LINKS[l][m] = true;
        }
    }
```

```
    // a boolean matrix is reversed
    // ! mens reverse: true -> false; false -> true
    private static void revert() {
        for (int l = 0; l < cities; l++) {
            for (int m = 0; m < cities; m++) {
                LINKS[l][m] = !LINKS[l][m];
            }
        }
        for (int i = 0; i < cities; i++) {
            LINKS[i][i] = false;
        }
    }

//The interconnection matrix is generated at random
    private static void generateMatrix() {
        //Initialization in false = no links
        for (int l = 0; l < cities; l++) {
            for (int m = 0; m < cities; m++) {
                LINKS[l][m] = false;
            }
        }
        //Generation of random links but without (i,i) links.
        //All generated links are different one from another.
        // if randLINKS > (1/2) (cities * cities )

        if (randLINKS <= half) {
            nLINKS = randLINKS;
            fill();
        } else //choose a negative refilling
        {
            nLINKS = cities * cities - randLINKS - cities;
            fill();
            revert();
        }
    }
    //Prints interconnection matrix

    private static void printMatrix() {

        System.out.print("     ");
        for (int i = 0; i < cities; i++) {
            System.out.print(i % 10 + " ");
        }
        System.out.println();
        for (int i = 0; i < cities; i++) {
```

```
            System.out.print(i % 10 + "    ");
            for (int j = 0; j < cities; j++) {
                if (LINKS[i][j] == true) {
                    System.out.print(1 + " ");
                } else {
                    System.out.print(0 + " ");
                }
            }
            System.out.println();
        }
    }

    //Finds and detects the nodes in some cycle
    private static void printNodes() {
        System.out.println("Nodes in some cycle");
        boolean thereIs = false;
        for (int i = 0; i < cities; i++) {
            if (LINKS[i][i] == true) {
                System.out.print(i + " ");
                thereIs = true;
            }
        }
        if (thereIs == false) {
            System.out.print("No nodes");
        } else;
        System.out.println();
        System.out.println();
    }

    //Detects the Cycles in the matrix LINKS
    private static void detectCycles() {
        //Power  k of LINKS is kept in Memory[k]
        boolean Memory[][][]
                = new boolean[2 * MAXCITIES][MAXCITIES][MAXCITIES];
        //Memory is initialized with LINKS
        for (int i = 0; i < cities; i++) {
            //Memory[1][i][j] = LINKS[i][j];
            System.arraycopy(LINKS[i], 0, Memory[1][i], 0, cities);
        }

        //Boolean powers of matrix LINKS are calculated
        int Max = Math.min(randLINKS + 2, 2 * cities + 1);
        for (int k = 2; k < Max; k++) {
            for (int i = 0; i < cities; i++) {
                for (int j = 0; j < cities; j++) {
                    //Initialization
```

```
                    Memory[k][i][j] = false;
                    //Boolean multiplication of the (k-1)-th power
                    // of LINKS with LINKS.
                    for (int l = 0; l < cities; l++) {
                        Memory[k][i][j] = Memory[k][i][j]
                                || (Memory[k - 1][i][l] & Memory[1][l][j]);
                    }
                    LINKS[i][j] = Memory[k][i][j];
                }
            }
            System.out.println("LINKS ^ " + k);
            printMatrix();
            printNodes();
        }
    }

    public static void main(String args[]) {
        //Number of cities
        cities = 5;
        //Number of unidirectional random links
        //shall be less than cities^2 - cities
        randLINKS = 12;
        double c = cities;
        double j = randLINKS;
        p = j / c;
        half = cities * cities / 2;
        System.out.println("Number of cities = " + cities);
        System.out.println("Numb rand links  = " + randLINKS);
        System.out.println("prop links/cities  = " + p);
        System.out.println("Half  = " + half);
        generateMatrix();
        System.out.println();
        System.out.println("LINKS = Interconnection matrix");
        printMatrix();
        detectCycles();
    }
}//End of Program C42, Cycles.
```

43 ***Exercise*** *Play with the code to understand how it functions and why.*

44 ***Exercise*** *Resolve the next complaint: the program outputs a list of nodes that form part of some cycle. But nowhere is guaranteed that these nodes forms part of the same cycle. Hence, the program is not good to study the arisen of giant cycles.* Answer

45 ***Exercise*** *Work with a constant number of cities or nodes and vary the number of unidirectional links from half the number of nodes to four times that number. Next, answer to the*

question: Which condition is sufficient to guarantee the arising of a giant cycled component? How giant is it? Answer

46 ***Exercise*** *Why do we need great cycles? Explain.* Answer

47 ***Challenge*** *To make more realistic the program of the previous exercise, make the necessary modifications ensuing the fact that all reactions in every living being go down across thermodynamic potentials. This means that somewhere you must include a source of free energy (light or redox potentials) that must be used to charge suitable molecules (AMP $\rightarrow$ ADP $\rightarrow$ ATP), which in their turn are used as fuel for reactions that otherwise are forbidden because they counteract dissipation. Your new model, in the presence of a great cycle, will look like self-sustained nonequilibrium chemical systems.*

48 *Adding catalysis*

Of course, what we call *first version of life* is only remotely related to what life is indeed. A crucial difference is that real life relies on catalysis. (Observe that we do not mention DNA as yet because if any alive being is deprived from its DNA, that being would survive more than an hour in almost perfect conditions.)

Now, our model presupposes that nodes are eternally defined, i.e., that we always have enough molecules of the same type. It is difficult to imagine that that is possible unless one can provide catalytic regeneration of the chemical pool. So, catalysis could be considered as a prerequisite for life.

So, how do we explain the incorporation of catalysis in our model?

Let us claim, following Kauffman, that catalysis could be an emergent property of complexity. The problem is that this claim was printed in 1995 with the promise of being tested in a near future some 10 years beyond. Many years have elapsed and no test has been made. So, we need to understand not only the theory but the possible cause of reluctance of the community to test it. In consequence, we need to give a technical look at the bases of enzymatic catalysis, which is the most powerful form of catalysis:

- A chemical reaction happens when the system formed by reactant molecules is reorganized into product molecules. Because products and reactants are stable molecules, they are represented by minimums of the potential energy and any path that joins them has a maximum along it (we have a saddle point). Many possible paths of reorganization exist, but some few paths or even a single one is preferred because of economy of energy, let us call it the reaction path (it can be calculated using quantum mechanics). The corresponding structure to the maximum along the reaction path is called activated complex. In reality, a reaction may have many diverse local maximums and diverse intermediate structures or activated complexes.

- Since the activated complexes are intermediate states that are absolutely required, the reaction can be accelerated if only the path through those complexes is made more expedite.

- This is easily achieved if the formation of one of the activated complex is facilitated.

- Enzyme catalysis is due to the asymmetry caused by the thermodynamically feasibility of the reaction and to the fact that the reaction center of the enzyme is dynamically complementary to the activated complexes of the chosen reaction, facilitating they formation and further evolution.

- It is important to note that enzymes don't fasten to any of the activated complexes otherwise the reaction would be stopped. In fact, powerful poisons function by just imitating an activated complex that once captured does not separate from the enzyme, inhibiting it from its usual work.

This theory of enzyme catalysis is not mythology, rather it is now a fundamental piece of science and technology: to design an enzyme for a given reaction it is sufficient to build a big molecule with a complementary craft to an specific activated complex of the reaction. The marvelous result is that we can use the immune system of a mammal as agent of design. The immune system can capture invader molecules with immunoglobulines: these proteins have terminals that have pocketlike formations. If a molecule passes in front of an immunoglobulin and has a face complementary to those formations and the electrical interaction is attractive, the passing molecule gets imprisoned by the immunoglobuline. More exactly: if the attraction is large enough, the molecule is captured, but this to be possible needs an stereo complementarity between the molecule and the immunoglobuline because the involved forces are of multipolar origin that are very weak.

Therefore, the immune system can be forced to fabricate an enzyme for a reaction if and only if a molecule with the structure of the prescribed activated complex is given to the immune system as an stimulus. This is done as follows: the activated complex is calculated by intuition or by quantum mechanics and then a molecule with the same structure as that complex is glued to a large foreign protein and injected into the bloodstream of a mammal, say, a rabbit. After two weeks, the mammal is examined for antibodies which are separated and proved as enzymes. Some antibodies work as enzymes but other not, as expected: antibodies are specialized molecules that capture other molecules, so they are in the line of poisons and not of enzymes.

Abzyme technology was invented by Tramontano and coworkers ([71] 1986) and Pollack and coworkers ([59] 1986, compare Hansen, [30] 1987) and has evolved today into a fertile discipline, antibody catalysis, with a very fast pace of development because by 1995 there were more than 60 examples of reactions that had been successfully catalyzed by monoclonal antibodies (Steward et al, l.c., [68] 1995). Instances include cases of reactions that are strongly disfavored because of the high free energy of the transition state (Shabat et al, [64] 1995).

So, what do we have? We have a completely mechanistic tested theory about enzyme catalysis that explains it in terms of synchronized spatial configurations. Nothing here prevents us to think that this could also happen by mere chance. Of course, the probability is a tiny one, so this problem is remedied with a very rich source of big molecules. How rich? Seemingly rich enough to prevent every scientist in the earth from making a test to this theory. And we have more problems, say, from our discussion about the theory about enzyme catalysis, we know that an enzyme must observe a given affinity to intermediate transition states but in such a way that no significant binding could arise. So, enzymes are much more complex than immunoglobulins that just form immobilized complexes. In fact,

immunoglobulins are calculated by the immune system, while enzymes are read from DNA and are not calculated because that is much more difficult. And reciprocally, a pocket made at random in a big molecule could serve more as an immobilizer of small molecules, if any, than as a facilitator for a chemical reaction.

It seems that a very discouraging suspicion arises: because trapping is highly more probable than catalysis, the path of self-organization to the arisen of an auto-catalytic chemical network is plagued by every kind of self-clogging events. Besides, a very common type of enzyme catalysis is directed to bond cleavage, so an auto-catalytic chemical network is is always in danger of self-digestion. Now, a system may exist by mere chance that has not such dangers. But, where is a calculation of the probability of its appearing?

Let us consider now the incorporation of DNA: that is absolutely necessary but the plausibility of its incorporation seems a rather crude supposition. This is argued as follows:

By 1995, RNA catalysis was the good news of the day, so Kauffman supposed that his complex world was populated by RNA polymers that would allow a soft transition to DNA, genetics, evolution and adaptation. Indeed, no RNA is needed at the beginning because not too much catalysis is expected from RNA, which has not enough richness in forms of folding, which is a prerequisite for efficient catalysis. Instead, an ongoing autocatalytic system could eventually acquire RNA and/or DNA and refactor all the complex machinery needed for their functional incorporation into genetics and evolution. But from other side, the machinery cannot go without auto-catalysis and this means that at the interior of the system there shall be catalyzed reactions that enable the formation of all needed catalysts. As our programs invite to think, randomness could cause the arisen of such a system but with incipient efficiency. So, genetics and evolution is wanted to be there as soon as possible to refine catalysis. This shows that a genetic system and evolution are mandatory ingredients of any form of refined life.

Our considered model, although half cooked, is simple and powerful. It is inspiring. It is like good poetry. The problem is that it collides head to head with science: on one hand, the model has mandatory predictions that are not observed, and on the other, the model is good more for creationism than for science. Let us see why. In regard with mandatory predictions, we have that:

The overall tendency of thermodynamic systems is toward dissipation that leads chemical systems toward molecules with lower and lower potential energy. If random catalysis somehow appears at the stage, there is an absolute certainty that even lower states of energy could be achieved. Those molecules would lose any capability to enter in a spontaneous reaction. So, they will appear as undesirable unrecyclable wastes. And this situation would remain for a very long period. Therefore, the model predicts that huge amounts of chemical inert organic material shall appear in the geological record. It is very easy to recognize that material: when one burns a meal, no matter of what type, it leaves a brownish-black residue at the bottom of the pot in which it was cooked. This is the sort of material predicted by the model and one should find it in ancient seas underneath trilobites and the like. Such material never has been found not by you, not by us, not by somebody. So the theory is false.

Nevertheless, the model can be understood as an idealization of the impressive technological network of goods and services of our modern world. They were build by people that

know how to tie the ends in their local trade but that have no idea of the general topology of the overall production graph: so the global market is managed in the large scale by randomness, by selforganization, although it is enabled by intelligent actions at the microscopic level. The intelligence may be found in implementing a given plan or design else in taking profit of the random discoveries and opportunities. This last aspect of science and technology is called serendipity, in memory of a Persian princess of a fairly tale that solved a problem by pure accident.

The model is also very good for projects on artificial or synthetic life and for some forms of creationism, in which the Creator is considered more as an engineer (as in the Popul Vuh, a sacred book of the Mayan literature) than as a magician (as in naive interpretation of the Genesis). The fundamental idea is that life is a property of some systems. Once life has been characterized, then one can recreate it in a computer (this produces AL or artificial life) or one can try to synthesize real systems with the distinctive properties of living beings to have synthetic life. In general, if one can provide a high degree of intelligence and a refined technology, one might consider to use the power of self-organization to create or recreate life: one can get rid of self-clogging and self-digestion by selective filtering and, more to the point, one can provide pure sceneries to guarantee results.

To see the importance of having pure sceneries, let us consider the incorporation of DNA into a self-catalytic system as that considered above. Let us notice that in all our simulations and in every mathematical model of the international literature, DNA is assumed as a pristine substance that is unable to chemically interact with the milieu. But contrast, every DNA base is chemically alive: as purines as pyrimidines, the bases of DNA, have at least 3 double bonds that are exposed to immediate chemical attack. This possibility is so threatening that the cell has the histones to pack DNA that it could be saved from undesirable chemical attack. So, the DNA of evolution has more of a fiction than of a fact of self-organization.

3.2 Conclusion

Looking for a possible natural explanation of the existence of the sophisticated evolutionary environment that life represents, we have revised one the most beautiful masterpieces of modern scientific poetry. It consists in a model proposed by Kauffman and that predicts the necessity of the arising of auto-catalytic chemical networks that function far away from equilibrium. This beautiful model has the following prediction: so abundant mix of complex compounds as the one demanded there and that moreover includes catalysis and dissipation would leave clear tracks in the geological record such as those left by a burnt meal in the pot where it was cooked. But no one has ever found such a trace. On the other hand, the model assumes the unrealizable possibility that DNA is kept untouched by the chemical milieu. So the model is plainly scientific and falsifiable, in fact, it is false.

Chapter 4

Software discontinuity and semantics

```
Most mutations are lethal
```

49 ***Motivation*** *We have learned that genetic programming must battle with the tremendous load of computations that might lead to nowhere. In trying to understand why, we have said that this is due to the discontinuity of software, which is the fact that tiny changes in software may produce terrible changes in the output else produce the dead of the program. Can we somehow study this phenomenon a bit better?*

50 ***Purpose*** *An invitation is formulated to study with more care the discontinuity or chaotic nature of software. We relate this fact with semantics, the science of meanings. In this regard, the Reader is invited to review chapter 9 of Vol I, where semantics was introduced.*

4.1 From Grammar to semantics

Let us discuss the effect of grammar and syntax on the discontinuity of software.

51 ***The nonsense sea***

One is acquainted with software mainly through the code of viable programs written in specific programming languages. At presently, any developer must obey certain tight grammatical and syntactic rules otherwise his or her program will not compile - although some compilers can be instructed to do whatever is possible in spite of errors.

To fix ideas, let us consider a phrase in Java:

```
if (randLinks <= half ) fill();
```

At presently, one cannot write *of, af, ef, of, uf* instead of *if* because if one makes that, the compiler will stop compilation and, most of times, any further execution. This sort of errors are referred as grammatical. One is also forbidden to write

```
fill() if (randLinks <= half ) ;
```

because one would commit a syntactic error.

Working over such type of errors is possibly the usual way as most people learn that tiny changes in the code produce the death of the code most of times and in any case tremendous changes. We refer this as **software discontinuity** or **chaotic nature of software**.

In order to advance, let us make explicit the meaning of the fundamental terms of linguistics, grammar and syntax (see Webster's II New Riverside University Dictionary). The word **grammar** could be understood as the system of rules of a given language, whose mechanical application generates all possible sentences in that language. Example: apart from irregular verbs, add *ed* to a verb to form the past tense in English. **Syntax** is the part of grammar that deal with the formation of phrases and sentences as combinations of words. Example: in German, put the verb in the second place of a phrase. **Semantics** is the science of meaning in language forms, the study of relationships between signs and symbols and what they represent to their interpreters. Example: if a nice lady tells you'Hasta la vista, baby' (until we meet again!), smile and kiss her but if those words come from a snow knight, hide away as fast as possible.

Any piece of software is written in a specific programming language. Now, a programming language is just another language, as much as English or French, and share with them some important characteristics. From the stand point of mutation, one property that enjoy all natural human languages is that most random strings are senseless. The effect is that a random mutation is almost always deleterious, while some few tandem mutations are lethal: any trace of meaning is lost. And nevertheless, we have the possibility to generate meaningful words from other ones. We say that words are generated by recursiveness. Example, Beijin is a compound word of north + capital while Nanjin is compound of south + capital, the name given to the city where the emperor must rest in summer to be in harmony with the laws of the Tao. Steels ([67] 1997) proposes that the positive feedback between success in use and re-use proficiently selforganizes this process. For this reason, it is always problematic to extract grammars from human languages.

Most random strings have no meaning and minor modification of significant strings render no sense. We say that any meaning is surrounded by a sea of nonsense. Why does such a sea exist? Any human language must have a sea of nonsense that must be deeper and wider the higher the level of recursion. This results from a compromise between two forces. The first deal with memory restrictions: The sea of nonsense is a necessary emergent feature in any human language because, on one hand, we have a marvelous memory that can cope with near to one million words. That is too much, but that number is too small when it is compared with all possible strings with 20 characters, which is $26^{20} = 2 \times 10^{28}$. Henceforth, most strings 20 letters long must have no meaning.

But we have a problem: if one can memorize just one million words, why must we use strings with 20 chars? Would it be not enough to use strings with 6 chars? One reason is that because of memory restrictions, the use of recursion to generate new words is profitable and so, one readily would surpass the length of 6 chars. So, long words and sentences are unavoidable. Hence, the sea of nonsense emerges from a combination of memory restrictions and the success of recursiveness. But, is this all to it?

52 *Self correcting codes*

Humans have been able to reproduce languages that are adapted to the milieu: human languages are very good with keeping the stability of the conveyed meaning is spite of noise

in the channels of communications, be it the air between the two sides of a river or the interferences from other voices in a meeting or even from memory incertitudes. How this is achieved? The most elementary solution is by interposing a distance among the words with meaning. To see the effect of this measure, let us give a glance to one virtue of modern word processors:

Most word processors for human languages already come with an inbuilt facility of self correction. Say, if one writes 'acurate', the word processor would automatically retype 'accurate'. It is not at all easy to do this for every possible error and some would think that this is impossible because humans language are context dependent, an expression that means that the meaning of a given string depends on the whole environment that might surpass the paragraph level. Anyway, simple and viable solutions have been proposed. For instance, Eclipse and NetBeans offer for objects an automatic typing assistance which presents either the most probably functions to be used or an exhaustive list. This favors both memory and error-free typing.

Self correction is a remedy against the lethality of most mutations. As we have seen, genetic programming can be studied with the help of Java and one can manage to guarantee the viability of the code of any possible mutating program. All what we need is to concatenate a viable piece of code to another one, such that the output of the first could be processed by the entering method. The programming language LISP offers others and in certain sense more fundamental facilities that enables viability after mutations.

In summary, memory restrictions and the convenience of recursiveness conspire together to give rise to a nonsense sea around any meaningful string which shall be wider the higher the level of organization, when we pass from words to phrases to paragraphs to chapters to books to cosmovisions. When the nonsense sea is wide enough, self - correction becomes possible.

4.2 Semantics and genetics

We know that human languages have a nonsense sea and we know why must this be so. What is the corresponding situation with the genome as the natural software, which is a language too?

A nonsense sea is found in genetics too: most mutations are deleterious if not lethal, a fact that is more precisely expressed by the neutral theory of molecular evolution (Kimura, 1983): a DNA function is filled by some few strings whose performance is roughly equivalent. Strings are few, because of the sea of nonsense surrounding small islands of meaning.

Is that surprising ? Yes, and indeed too much. Let us see why : we can say that a genetic string is a message transmitted by means of a biochemical machinery to nature: we say that nature understands the message when it allows the transmitting carrier cell or organism to live and reproduce further. But, why does the versatility of the universe seems so limited as to being unable of understanding most mutations? That is very surprising. Or maybe not. To begin with, let us mention a very interesting fact that is quite contrary to the view of a generalized nonsense sea:

When Kimura published his theory, the equivalent strings were imagined to be some few in number, in a literal sense. Now there is at least one studied case in which the equivalent

strings are not few: it is t-RNA. The reason of a huge of allowed neutral mutation is that the object of selection is the phenotype and not the genotype, but in the t-RNA case we have that too many genotypes correspond with one single phenotype. For t-RNA, the genotype is the sequence of nucleotides, but the phenotype is represented very well by its second structure:

"Call two sequences connected if they differ by one or at most two point mutations. A neutral network, then, is a set of sequences with identical structure so that each sequence is connected to at least one other sequence. The crucial point for our discussion comes from a recent study of the standard secondary structure prediction algorithm, which showed that such networks percolate though sequence space. For example, starting at a sequence that folds into a tRNA structure, it is possible to traverse sequence space along a connected path, thus changing every nucleotide position without ever changing the structure. Moreover, due to the high-dimensionality of sequence space, networks of frequent structures penetrate each other so that each frequent structure is almost realized within a small distance of any random sequence. These features seem to be intrinsic to RNA folding, since they are insensitive to whether the folding algorithm is thermodynamic, kinetic, or maximum matching (...) or whether one considers one minimum free energy structure or the entire Boltzmann ensemble." (Huynen et al., [36] 1996).

The unveiling of neutral networks is a clear and bright hope for evolution: "The time evolution of a population of asexually replicating molecules on a flat landscape can be described by a random walk. On more complex fitness landscapes a selective pressure drives the population towards sites of higher fitness. In the simplest case one may think of one or few structures. Then the population searches "along a neutral network" until sequences of another network corresponding to a fitter structure are found." (Reidys et al., [61] 1997).

The idea of a neutral network is attractive but its extension beyond t-RNAs is not expected. Therefore, its evolutionary role to explain complexity is rather limited. So, let us return at the meantime to our problem: how do we explain the genetic nonsense sea?

Let us discuss an explanation using the very same reasons that we used to explain the nonsense sea in the case of human languages: memory restrictions, the convenience of recursiveness and self-correction:

- Memory restrictions: How can we accept memory restrictions to self-organization and evolution when one can use atoms as memory devices and in a handful of water there are around 1×10^{24} of them? The reason is that in the world of combinatorics 1×10^{24} is really tiny, insignificant.

- The convenience of recursiveness: Is it not amazing that one can built a house using tiles? So, recursiveness pays and pays very good no matter what we deal with, houses or organisms: recursiveness might produce functional complexity quite swiftly. So, recursiveness could be part of a possible explanation of the nonsense sea in genetics.

- The convenience of self-correction: this is a verified good property of the genetic code of present organisms: it minimizes the effect of errors in transcription and translation.

So, memory restrictions, recursiveness and self-correction are in good order. Besides we have other type of restrictions: extant life is based on catalysis, and we need many catalyzed reactions of very different nature that work in ensemble to build an organism. On the other

hand, catalysis is the result of a delicate dynamical stereo-chemical matching that relies on very weak forces of short range. As a consequence, some protein sequences will do the job while others not. This is referred to as functional restrictions.

We consider that recursiveness and functional restrictions are sufficient to cause the observed nonsense sea in genetics, whose effect is somehow neutralized by self correction mechanisms.

4.3 The complexity of a semantics

A modern language might have as much as one million words in its grammar. By contrast, a normal person can have some 10 to 20 thousands words in its repertoire, while little babies suffer a lot to surpass the barrier of 7 words. This contrast shows us that complexity defines a classification among different semantics. Let us study this in more detail.

53 ***Expressiveness and graininess***

Software comes in the form of code or verbal instructions to be executed by a computer or more precisely by a CPU (Central Processing Unit). Some computers just can add, other can execute the fundamental operations, other can simulate logical operation. We say that a computer or CPU has universal computation when it can execute every recursive function. A programming language to be interesting must enable the transmission of any possible order that the CPU can execute. The relation of the verbal orders with the actions is the *semantics* of the language.

A semantics enjoys two properties: expressiveness and graininess. To understand what these terms mean, let us consider a cow in relation with a chimp: the arms of the cow can execute just the motion of a pendulum, while the arms of a chimp can execute every sort of movement: to the front, to the sides, circular motions plus the freedom that comes from having mobile shoulders. In consequence, the semantics of the language that describes the movements of the arm of the cow has a simpler expressiveness than that of the chimp. Let us consider now the hand of a little boy in comparison with that of an artisan: the little boy can do some simple imprecise movements, while the fingers of the master exhibits a dexterity, whose precise characterization demands fine terminology. The corresponding languages are differentiated by the graininess of the semantics: high for the artisan, low for the baby.

The overall function of an ordinary computer, to have universal computation, is just to do whatever comes to the mind. By the same token, the overall function of an alive being is to be adapted to the milieu. Because these operation are quite complex, we have every reason to say that the semantics of programming languages of ordinary computers and of alive beings must be very expressive and extremely fine grained.

Now, because of computing and memory restrictions, one cannot combine great expressiveness with fine graininess. The result is that one must conform with patching solutions: trying to cover a great field, being very specific here but quite coarse there. To cover a great field, we use large modules, domain or schemes, and to conserve fine graininess we must enable the possibility of mutations at any moment. We see here the grammatical necessity of evolution.

When the semantics of a problem is very expressive and fine grained and it has been fairly modeled by a code or by a language, slightly different orders can mean very different actions

with very different results. Human languages are in general well adapted to the milieu and so they can correctly convey information in spite of noise in the milieu. This is achieved by choosing appropriate combinations of sounds that give rise to self correcting code operations.

On top of all this, we have the nonlinearity of systems. A system is nonlinear when, say, a tiny change in the input produces exaggerated changes in the output. Example: some extremely expensive perfumes are just a bit far from being repugnant, so if one adds a bit more of certain compound, one kills the trade. Why? At the root of this phenomena reside finely tuned cultural and biological responses.

Summing up: the richness of behaviors of nature as measured by its non linear systems demands high expressiveness of the language dealing with them. Nonlinearity of systems causes that the perceived world is composed of quantitative patches that are represented in a fair language by different names, which in their turn call for finely grained semantics. So, human languages and software are and must be chaotic because they deal with nature, which is very complex in both, expressiveness and fine graininess.

Let us retake now the NK model, which is also good to represent the complexity of semantics, in both aspects of expressiveness and graininess.

54 ***The NK model***

Trying to understand why industrial software has a cycle similar to that of an alive being, the NK model allowed us to reinforce the idea that a fundamental complexity measure that makes software resistant to evolution is the complexity of its interconnections. We will see now that this very same model can be interpreted in terms of the complexity of a semantics. Let us see why.

To begin with, if the interconnections of a program form a very complex web, you will need a lot of terminology and long reports to describe them. Henceforth, the very same NK model that served to model interconnectivity is also good to model the effect of expressive complexity of a semantics over the form of evolution of the system. In other words, we are proposing that the expressiveness of a semantics has a principal component represented by the complexity of the interconnections of the forming parts of the system.

And, what about the relation of graininess with the NK model?

Let us consider a piece of functional software, a module of Java. Or a scheme in a protein. The properties of modules and schemes are represented by names, say, 'that model is easy to read' or 'that scheme is highly repetitive'. Properties are not elementary properties of the program, instead they can be traced down to more elementary descriptors, say, a method of Java may be easy to read because it is short and has simple lines. Or, a protein scheme is repetitive because some few substrings have high relative frequencies. With some care, one can elaborate the elementary descriptors into a quantitative ones. For instance: we can count the number of words in a method, or the proportion of amino acids that are different than the five most frequent ones. In general, one always can make a change to get an elementary descriptor such that it is a predictor of the degree to which a certain property is valid for the system.

There is variability in the space of methods (or of schemes) that fill the same function or that have the same property. The simpler supposition about the distribution of fitness is that it is normally distributed, with the form of a Gauss Bell, around certain average value.

Now, how do we represent the difference between a fine grained and a coarse grained semantics? The graininess of the semantics is represented by the aperture of the bell: a very closed bell represents a very specific range of allowed values with a relatively high fitness. A widely opened bell represents a wide span of allowed values with relatively moderate fitness. The incompatibility of high fitness and low specificity is precisely what we have in enzymatic catalysis: the turn over of very specific enzymes could be very high, while unspecific catalysis always have middle to low turnovers.

So, we have a bell with a given mean and standard deviation. Where is in here the NK model? Let us recall that in the NK model, the N parameter was held constant for a given simulation. So, the only living parameter was K. What was the effect of K in the NK model? K was the number of interconnected subunits to the subunit under consideration: we took K random values with a uniform distribution in within 0 and 1, and averaged them to produce an instance of the fitness of certain interconnected program. How is then distributed the average fitness? According to the central limit theorem, the average fitness is just the sample mean of a uniform distribution and its standard deviation diminishes with K: the more items are averaged, the less far are they from the overall mean.

Thus, in the NK model, K was a measure of the standard deviation of a Gauss Bell. So, we can change standard deviation by K (a minor discrepancy may arise because K is integer while the deviation is real): for a given K, the corresponding standard deviation could be taken as $1/\sqrt{K}$.

So, the NK model of the complexity of a semantics has a double interpretation. First, expressiveness can be modeled by interconnectivity, which is directly represented by the NK model, while the graininess of a semantics is also represented by the standard deviation $1/\sqrt{K}$ of the fitness distribution, which is also normally distributed.

In conclusion, we have recycled the NK model through various interpretations and so, we can use the results of the original simulation to predict that: the more complex is a semantics, be in expressiveness or in graininess, the more rugged is the corresponding landscape and the slower is the evolutionary process along any evolutionary branch.

The prediction of our new interpretation of the NK Model sound nice, i.e., in agreement with what we expect by common sense: evolution is slowed or halted by complexity of the semantics involved, be it in expressiveness or in graininess. Let us review now another aspect of the complexity of a semantics: context sensitiveness, which is a descriptor that is good to discuss the very fact that most mutations are lethal, be it in the genome, or in most programming languages, or in all human languages.

4.4 Context dependence

Context dependence is a reserved expression in computing science with constructive definitions based on automates, but we will take here a very pragmatic approach to it:

A terrible problem of human languages is that the so called idiomatic expressions do not have a meaning that is directly related to the meaning of its compounding words. Thus, such expressions have a meaning that is dominated by the effect of the surrounding material. We will refer to this fact as context dependence.

55 *Some examples*

We have seen that *grammar* is the system of rules of a given language, whose mechanical application generates all possible sentences in that language. So, a grammar is composed of a list of primitive words, plus some rules that create more words from extant ones. In this definition, a word is any string, so it represents in a human language words, sentences and all forms of more complex units.

Some strings have a meaning, i.e., can be translated into actions or explanations. This is the semantics of the language. Human communication is not only verbal but also include signs with the hands, with the mode of looking or speaking, but one can make an approximation and consider *the semantics as part of the grammar*, just by inserting rules that replace a string by an explanation of its meaning, i.e., by including a dictionary as part of the grammar.

All human languages have idiomatic expressions, whose meaning could be more or less clear as in *'he is the only fly in the ointment'* but there are other more conspicuous as in *'he is about as much poet as I am'* that generally means that he is no poet at all. In general, good poetry and narrative may form unities with meanings that cannot be attached to any of their parts. The first time that I realized this was when reading *The Foreigner (L'Étranger)*, a work by Albert Camus. These linguistic possibilities make of literature a very challenging art that we refer to as *context dependence*: it is the property of a grammar of having irreducible rules of arbitrary length for formation of compound words.

Why context dependence is important for us? Because it is an easy form of modeling the almost mystic concepts of holism and of emergent properties and to study their effect on the mode and tempo of evolution. Let us make a simulation. We will study the semantics generated by the prime numbers. An integer number is prime when it has only two divisors: one and itself. Example: 7 is prime; 9 is not. In our semantics, we will have a list with all prime numbers less than, say, 1000.

56 ***A list of primes*** *The code that generates a list with all primes until 1000 is the following:*

```
/* Program C56, Primes:
 * The program generate all Primes
 * until 10000.
 * An integer number is prime when
 * it has only two divisors: one and itself.
 * Example: 7 is prime; 9 is not.
 *
 */

package ejvol3v2p;


public class Primes
{
 private static final int N = 1000;
```

```
 private static final int PRIMES[]  = new int[100000];

 public static void main(String[] args)
 {
  // First two Primes are known
  System.out.println(1);
  System.out.println(2);
  //The vector of Primes
  PRIMES[0] = 2;
  //Counter of the number of Primes
  int counter = 1;
  int l;
  //All odd numbers are scanned
  for(int i = 3; i<= N; i=i+2)
  {
boolean isPrime = true;
//Primness is tested
//Division by lesser Primes is
//enough to decide the test.
    for(int j = 0; j < counter; j++)
    {
     //The result of the division is kept as
     // int, so it is truncated:
     // 2.7 becomes 2 and so on.
     l = i / PRIMES[j];
     if ( i == l * PRIMES[j]) isPrime = false;
    }
    if (isPrime)
    {
     PRIMES[counter] = i;
     counter = counter +1;
     System.out.println(i);
    }
  }
  System.out.println("Number of primes = " + counter + " + 2");
 }
}//End of Program C56, Primes
```

57 ***Exercise*** *Play with the code and verify that it does exactly what is ought to .*

58 ***Exercise*** *Primes have been investigated since ancient times: Eratosthenes invented some 2200 years ago a very efficient algorithm, which is quite different than the one dictated by the elementary common sense. The idea is this: if a number is multiple of another, it is not a prime. So, we begin with a list of all numbers and we begin to delete from the list any number that is multiple of 2, next of 3, next of 5 and so on. What does remain? The prime numbers. Example: let us find all prime numbers less than 25:*

We begin with the complete list:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Next we delete numbers that are multiple of 2:
1 2 3 5 7 9 11 13 15 17 19 21 23 25
Now, we delete the multiples of 3:
1 2 3 5 7 11 13 17 19 23 25
Now, we delete the multiples of 5:
1 2 3 5 7 11 13 17 19 23
Now, we delete the multiples of 7:
1 2 3 5 7 11 13 17 19 23
We observe that there is no multiples of 7. There is no multiples of 11 or 13 or 17 or 19 or 23. This leads us to think that we need to verify the algorithm just until the square root of 25, which is 5. After doing everything possible, we are left with primes numbers.
Implement this algorithm and write down the prime numbers from 1 to 1000. Answer

59 *The prime semantics*

Now that we have a list of primes, let us use it to run a simulation of evolution. Prime numbers generate an evolutionary semantics as follows: one has a population of binary strings that represent the organisms. These binary strings must match a predefined pattern, which is a prime. The degree of matching defines the fitness of the binary string. If the prime is fully matched, the next prime becomes the new pattern. And so on.

To compose our simulation, we could go around with what we know, but Java is very rich in tools and if one learns a bit from time to time, one can eventually become a true expert knowing many tools and mastering a fine tunned use of them. We will learn this time some aspect of the class BitSet, which allows us to manage boolean strings without wasting huge memory resources, such as we did when we used strings to encode binary strings. An instance of BitSet can be understood as a heavy duty binary string, and so this class has its own operators:

The instruction

```
BitSet set = new BitSet();
```

creates an instance of Bitset of infinite length, such that each character is false. To declare that some char is true, one can use an instruction as the next one:

```
set.set(1);
```

To retrieve the truth value of some char, we use:

```
set.get(6)
```

If some value has been set to true, and one wants it to become false, one can use:

```
set.clear(1)
```

To copy the value of the variable set into the BitSet variable set2, we use

```
set2 = (BitSet) set.clone();
```

To compare two variables set1 and set2, we use, for instance

```
if (set1.equals(set2)) set2.clear(1)
```

One can also copy a portion of a variable into another. The next instructions copy into set1 bits from set2 begining with position 3 to 5, i.e., 3 and 4.

```
set1 = set2.get(3, 5)
```

The instruction

```
System.out.println(set2);
```

prints a set consisting of all those integer numbers, whose corresponding char has been set to true. Thus, the class BinSet is very appropriate to encode sets of binary flags. Numbering begins always from zero.

Every instance of BitSet, and of any class, must be initialized even though as a thoroughly false instance.

The method of Eratosthenes to generate all primes less than certain N, which was implemented in exercise 58, is now rewritten using the BitSet class. This class is also used for other tasks.

60 ***The code for the prime semantics using BitSet follows***

```
/*Program C60, PrimeSemantics:
 *
 * A community of binary organisms is given the task
 * of guessing all primes less than, say, 1000.
 * We assign to a prime its binary expression.
 * The actual   prime serves as
 * pattern for selection:
 * the fitness of a binary string  or organism   is
 * the number of bits in which it is
 * congruent  with the pattern.
 * Example: pattern = eleven = 1011,
 * organism = 1010; fitness = 3;
 * When a prime has been fully guessed,
 * the next prime becomes the pattern
 * and so on.
 *
 * New Java material: we employ the BitSet class,
 * which can be understood as enabling
 * heavy duty binary strings or as
 * a set of membership flags.
```

```
 */
package ejvol3v2p;

import java.util.BitSet;
import java.util.Random;

public class PrimeSemantics {

    //We deal with prime numbers less than PRIMESLESSTHAN
    private static final int PRIMESLESSTHAN = 1000;
    /*PRIMES is a string of flags:
 true for a prime, false otherwise*/
 /*The next instruction allotes dynamic (modifiable) memory,
 gives a name to it, enables writting and reading,
 and initializes every bit of PRIMES in false.
     */
    private static final BitSet PRIMES = new BitSet(PRIMESLESSTHAN);
    private static int numbPRIMES;
    //A population of binary numbers is considered.
    //The number of individuals is constant
    private static int numbInd;
    private static final int MAXNUMBIND = 10000;
    private static final BitSet IND1[] = new BitSet[MAXNUMBIND];
    private static final BitSet IND2[] = new BitSet[MAXNUMBIND];
    private static double FitnessInd[]
            = new double[MAXNUMBIND];
    private static final int ORDERIND[] = new int[MAXNUMBIND];
    private static int lengthInd;
    //counter of generations per prime
    private static int gen;
    //Cumulative counter of generations
    private static int totalNumbGen = 0;
//Cumulative counter of trials or individuals
    private static int totalNumbInd = 0;
    private static boolean problem;
    private static int actualPrime;
    private static int counterPRIMES;
    //target is the actual prime in binary notation
    private static BitSet target = new BitSet();
//actualLength is the number of bits of target
    private static int actualLength;
    private static boolean recombination;
    private static double pointMutationRate;
// Turn on  the random generator
    static Random r = new Random();
```

```
    private static void setParameters() {
        //PRIMESLESSTHAN could be as large as a million
        //for 2GB in RAM;

        numbInd = 2; //less than MAXNUMBIND = 10000
        recombination = true; //true = enabled; false = disabled
        pointMutationRate = 0.3;
    }

    //PRIMES are listed according to Eratosthenes
    //The BitSet class is used to generate a string of
    //membership flags
    private static void generatePRIMES() {
        //Flags are initialized in true:
        //we have a complete list.
        //A number is primer if it is marked as true in PRIMES.
        //At the beginning, everyone is prime.
        for (int i = 1; i < PRIMESLESSTHAN; i++) {
            PRIMES.set(i);
        }
        //Multiples are deleted: primes remain.
        numbPRIMES = 1;
        for (int i = 2; i < PRIMESLESSTHAN; i++) {
            if (PRIMES.get(i))//If i is marked as true (as prime)
            {
                for (int j = 2 * i; j < PRIMESLESSTHAN; j = j + i) //!!
                {
                    PRIMES.clear(j); //multiples are marked as false
                }
            }
        }
        for (int i = 2; i < PRIMESLESSTHAN; i++) {
            if (PRIMES.get(i)) {
                numbPRIMES = numbPRIMES + 1;
            }
        }
        //The next instruction prints the set of
        //ALL PRIMES until PRIMESLESSTHAN;

        //System.out.println(PRIMES);
    }

    //IND1 is a population of binary strings
    //initialized in false everywhere.
    //The BitSet class is used to instanciate
    //binary strings as individuals
```

```
    private static void initialize() {
        BitSet s = new BitSet(); //set is false everywhere
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) s.clone();
        }
        //Reinitialization with random truth values.
        //At the beginning, all individuals have 2 bits:
        for (int i = 0; i < numbInd; i++) {
            for (int j = 0; j < lengthInd; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
        }
    }

    private static void nextPrime() {
        //System.out.println("OldPRIME = " + actualPrime);
        boolean nothing = true;
        for (int i = actualPrime + 1; nothing; i++) {
            if (PRIMES.get(i)) {
                nothing = false;
                actualPrime = i;
            }
            if (i >= PRIMESLESSTHAN) {
                nothing = false;
                problem = false;
            }
        }
        // System.out.println();
        // System.out.println("ACTUAL PRIME = " + actualPrime);
    }

    //Converts actualPrime into a BitSet value
    private static void defineTarget() {
        //actualPrime is converted to a boolean string
        String a = Integer.toBinaryString(actualPrime);
        //The boolean string is converted into BitSet
        actualLength = a.length();
        for (int j = 0; j < actualLength; j++) {
            char c = a.charAt(j);
            if (c == '1') {
                target.set(j);
            }
        }
```

```
        //If actualLength = 5,
        // target = {0,2,3,4} = 10111 = 10000 + 100 + 10 + 1
        // = 16+4+2+1 = 23
        //System.out.println("Target = " + target);
        //System.out.println("ActualLength = " + actualLength);

    }

    //Initialization
    private static void beginning() {
        generatePRIMES();
        initialize();
        //test for assessInd(s);
        //test1();
        actualPrime = 1;
        counterPRIMES = 1;

        nextPrime();
        defineTarget();
        //Individuals are sorted by fitness
        problem = sortingInd();
        //Loop of evolution?.
        gen = 1;
    }

    static private int decode(BitSet match) {
        int i = 0;
        int k;
        int n = match.length();
        for (int j = 0; j < n; j++) {
            if (match.get(j) == true) {
                k = 1;
            } else {
                k = 0;
            }
            i = k + i * 2;
        }
        return i;
    }

// Each individual is tested
// for how good it matches the target
    static private int assessInd(BitSet s) {
        //The fitness of s is the number of bits in which
        //s and traget coincide;
        int fitness = 0;
```

```
        for (int i = 0; i < actualLength; i++) {
            if (s.get(i) == target.get(i)) {
                fitness = fitness + 1;
            }
        }
        /*
 System.out.print("target = " + target);
 System.out.print(" s = " + s + "fitness = "
                                  + fitness);
 System.out.println(" actualLength = " +actualLength);
         */
        return fitness;
    }

    // test for assessInd(s);
    private static void test1() {
        BitSet t = new BitSet(5);
        target = (BitSet) t.clone();
        target.set(1);
        target.set(3);
        target.set(5);
        BitSet s = new BitSet(5);
        s.set(0);
        s.set(2);
        s.set(3);
        s.set(4);
        assessInd(s);
    }

// Individuals are sorted by fitness.
// the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        double FitnessIndc[];
        FitnessInd = new double[MAXNUMBIND];
        FitnessIndc = new double[MAXNUMBIND];
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
```

```
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < numbInd; i++) {
            Champ = 0;
            for (int j = 0; j < numbInd; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array order orders individuals by  fitness:
            //The fittest is number zero
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        /*System.out.println("Gen = " +gen);
  System.out.println("Champ = " + IND1[ORDERIND[0]]);
  for(int i = 0; i< numbInd;i++)
  {
   System.out.println(ORDERIND[i] + "     " +
            FitnessInd[ORDERIND[i]]);
  }
         */
        // if actualPrime has been guessed
        if (FitnessInd[ORDERIND[0]] == actualLength) {
            problem = false;
        }
        return problem;
    }

// The new generation is made
    private static void newGenerationInd() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest

        while (i < numbInd) {
            int counter = numbCopies;
            while ((counter > 0) & (i < numbInd)) {
                //System.out.println(i + "" + l);
                IND2[i] = (BitSet) IND1[ORDERIND[k]].clone();
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
```

```
            }
            k++;
        }

        for (int j = 0; j < numbInd; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*System.out.println( j+ " IND1 "
                                + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(numbInd);
        //b is false everywhere
        BitSet b = new BitSet();
        //B is the recombinant of l and k
        for (int j = 0; j < actualLength; j++) {
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    b.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
                    b.set(j);
                }
            }
        }
        /*
 System.out.println();
 System.out.println("Ind[l] = " + IND1[l]);
 System.out.println("Ind[k] = " + IND1[k]);
 System.out.println("Recomb = " + b);
         */
        return b;
    }

// The definitive population  is a recombinant one.
    private static void recombinationInd() {
        //IND2 is a copy of IND1
```

```
        for (int i = 0; i < numbInd; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //IND2 is a recombinant version of IND1
        for (int l = 0; l < numbInd; l++) {
            IND2[l] = recInd(l);
        }
        //IND2 is transferred to IND1
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }

// Individual  a is mutated at a random place.
    private static BitSet pointMutationInd(BitSet a) {
        int l = r.nextInt(actualLength);
        a.flip(l);
        return a;
    }

// Individuals are mutated
    private static void mutationInd() {
        for (int i = 0; i < numbInd; i++) {
            double p = r.nextDouble();
            if (p < pointMutationRate) {
                IND1[i] = pointMutationInd(IND1[i]);
            }
        }
    }

// Given a goal, it is guessed by evolution
    private static void evolution() {
        //The new generation is produced
        //as copies of the fittest
        newGenerationInd();
        if (recombination) {
            recombinationInd();
        }
        //The new population is subjected to mutation
        mutationInd();
        gen++;
        problem = sortingInd();
    }

    private static void report1() {
        System.out.println("Task ended by generation =" + gen);
```

```
        System.out.println("Number of trials= " + gen * numbInd);
        //match is a BitSet that matches actualPrime
        BitSet match = (BitSet) IND1[ORDERIND[0]].clone();
        System.out.println("Match = " + match);
        System.out.println("Found target = "
                + decode(match));
    }

    private static void report2() {
        System.out.println();
        System.out.println("PRIMES until " + PRIMESLESSTHAN);
        System.out.println("Ind/generation = " + numbInd);
        System.out.println("Recombination  = " + recombination);
        System.out.println("Number of guessed primes = " + numbPRIMES);
        System.out.println("Last guessed prime = " + actualPrime);
        System.out.println("totalNumbGen = " + totalNumbGen);
        System.out.println("totalNumbInd = " + totalNumbInd);
    }

//The population must match all primes, one by one
//beginning with 2 until PRIMESLESSTHAN.
    public static void main(String[] args) {
        setParameters();
        beginning();
        int percent = 0;
        problem = true;
        while ((problem)) {
            //System.out.println( "gen of Ind = " + gen);
            evolution();
            // if a prime has  been guessed ( problem = false)
            //then, pass to the next prime.
            if ((problem == false)) {
                //report1();
                counterPRIMES = counterPRIMES + 1;
                totalNumbGen = totalNumbGen + gen;
                totalNumbInd = totalNumbInd + gen * numbInd;
                if (counterPRIMES < numbPRIMES) {
                    int percentage = (int) 100 * counterPRIMES / numbPRIMES;
                    if (percentage > percent) {
                        percent = percentage;
                        System.out.println("Done work = " + percent + "%");
                    }
                    nextPrime();
                    // Old memory is erased
                    target.clear();
                    defineTarget();
```

```
                    gen = 0;
                    problem = true;
                }
            }
        }
        report2();
    }
}//End of Program C60, PrimeSemantics
```

61 ***Exercise*** *Run the code with various values of parameters and try to understand it. Does the prime semantics represent a challenge for evolution? Answer*

62 ***Exercise*** *Use the previous program to assess the effect of parallelism (various individuals evolve without recombination) and recombination over the performance of evolution. Explain the results. Answer*

We saw in the Weasel program in Vol II, that it is no problem for evolution to match a predefined string and we have seen that to match a set of strings, the prime numbers, is neither a challenge for evolution when the semantics is gradualist. So, we must ask ourselves: does exist a problem sufficiently discontinuous to challenge evolution? The answer is affirmative: from Vol II we do know that genetic programming is a tremendous challenge for evolution and it can be reformulated in semantic terms: to devise strings that represent viable meaningful computer programs with not obvious function. The problem with that semantics is that it is too complex for a study with one single PC. So, it would be convenient to devise a semantics of intermediate complexity that could be investigated with our scare resources. Let us make a try.

63 ***Guessing random sequences***

We generate random binary numbers with f(n) bits that serve as pattern for selection. Example: for f(n) = n, we generate random binary numbers with 1,2,...,n bits. For f(n) = 5n, we generate random binary numbers with 1,5,10 bits and so on.

A community of binary organisms is given the task of guessing all generated numbers that serve as patterns, until a give bound. The fitness of a binary string or organism is the number of bits in which it is congruent with the pattern. Example: pattern = eleven = 1011, organism = 1010; fitness = 3; When a number has been fully guessed, the next in the list becomes the pattern, and so on.

64 ***The code*** *The program for the study of families of random semantics follows:*

```
/*Program C64, randSem:
 *
 * Our purpose is to match a set of random binary strings.
 *
 * To that aim, we generate random binary numbers with
```

```
 * f(n) bits that serve as target for evolution
 * and as pattern  for selection.
 * For f(n) = n, we generate random binary numbers
 * with 1,2,...,n bits.
 * For f(n) = 5n, we generate random binary numbers
 * with 1,5,10 bits and so on.
 *
 * A community of binary organisms is given the task
 * of guessing all generated numbers that serve as patterns,
 * until a give bound.
 *
 * The fitness of a binary string  or organism   is
 * the number of bits in which it is
 * congruent  with the pattern that is also the target.
 * Example: pattern = eleven = 1011,
 * organism = 1010; fitness = 3;
 * When a target has been  guessed,
 * the next   in the list  becomes the new target,
 * and so on.
 *
 */
package ejvol3v2p;

import java.util.BitSet;
import java.util.Random;

public class RandSeman {

    private static int maxLength;
    //A population of binary numbers is considered.
    //The number of individuals is constant
    private static int numbInd;
    private static final int MAXNUMBIND = 10000;
    private static final BitSet IND1[] = new BitSet[MAXNUMBIND];
    private static final BitSet IND2[] = new BitSet[MAXNUMBIND];
    private static double FitnessInd[]
            = new double[MAXNUMBIND];
    private static final int ORDERIND[] = new int[MAXNUMBIND];
    private static int gen;
    //Cummulative counter of generations
    private static int cumNumbGen = 0;
//Cummulative counter of trials or individuals
    private static int cumNumbInd = 0;
    private static boolean problem = true;
    private static int counter;
    private static BitSet target = new BitSet();
```

```
private static int unitLength;
private static int maxNumbUnits;
private static int length;
private static boolean recombination;
//if the pop is reinitialized at each gen, reinit = true
private static boolean reinit;
private static double mutationRate;
// Turn on  the random generator
static Random r = new Random();

private static void setParameters() {
    //length of unit
    unitLength = 2;
    maxNumbUnits = 20;
    numbInd = 2; //less than MAXNUMBIND = 10000
    recombination = true; //true = recombination is enabled
    //reinit = true: the pop is initialized with a new task
    reinit = true;
    mutationRate = 0.3;
    //maxLength is the maximum length of the pattern
    maxLength = maxNumbUnits * unitLength;
}

private static int length(int n) {
    int z = n * unitLength;
    return z;
}

//Converts a BitSet value into a String
static private String decode(BitSet t) {
    String s = "";
    for (int i = 0; i < length; i++) {
        if (t.get(i) == true) {
            s = s + "1";
        } else {
            s = s + "0";
        }
    }
    return s;
}

//A random binary pattern  or target is generated
//f(n) bits long
private static BitSet defineTarget(int n) {
    BitSet pattern = new BitSet();
    length = length(n);
```

```
        for (int i = 0; i < length; i++) {
            int l = r.nextInt(2);
            if (l == 1) {
                pattern.set(i);
            }
        }

        System.out.println("New length = " + length);
        System.out.println("New  binary  target = "
                + decode(pattern));
        System.out.println("BitSet = "
                + (BitSet) pattern);

        return pattern;
    }

    //IND1 is a population of binary strings.
    //The BitSet class is used to instanciate
    //binary strings as individuals
    private static void initializePop() {
        BitSet s = new BitSet();
        //IND1 is false everywhere
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) s.clone();
        }
        //Reinitialization with random truth values.
        for (int i = 0; i < numbInd; i++) {
            for (int j = 0; j < length; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
        }
        // System.out.println("IND11 : " + IND1[1].get(1));
    }

    //Initialization
    private static void beginning() {
        counter = 1;
        target = defineTarget(counter);
        initializePop();
    }

// Each individual is tested
// for how good it matches the target
```

```
    static private int assessInd(BitSet s) {
        //The fitness of s is the number of bits in which
        //s and traget coincide;
        int fitness = 0;
        for (int i = 0; i < length; i++) {
            if (s.get(i) == target.get(i)) {
                fitness = fitness + 1;
            }
        }
        /*
 System.out.print("target = " + target);
 System.out.print(" s = " + s + "fitness = "
                                  + fitness);
 System.out.println(" unitLength = " +unitLength);
         */
        return fitness;
    }

    // test for assessInd(s);
    private static void test1() {
        BitSet t = new BitSet(5);
        target = (BitSet) t.clone();
        target.set(1);
        target.set(3);
        target.set(5);
        BitSet s = new BitSet(5);
        s.set(0);
        s.set(2);
        s.set(3);
        s.set(4);
        assessInd(s);
    }

// Individuals are sorted by fitness.
// the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        double FitnessIndc[];
        FitnessInd = new double[MAXNUMBIND];
        FitnessIndc = new double[MAXNUMBIND];
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
```

```
        }
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < numbInd; i++) {
            Champ = 0;
            for (int j = 0; j < numbInd; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array ORDERIND orders individuals by  fitness:
            //The fittest is number zero
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        /*System.out.println("Gen = " +gen);
  System.out.println("Champ = " + IND1[ORDERIND[0]]);
  for(int i = 0; i< numbInd;i++)
  {
   System.out.println(ORDERIND[i] + "     " +
            FitnessInd[ORDERIND[i]]);
  }
         */
        // if Pattern has been guessed
        if (FitnessInd[ORDERIND[0]] == length) {
            problem = false;
            //System.out.println("Champ = " + IND1[ORDERIND[0]]);
        }
        return problem;
    }

// The new generation is made
    private static void newGenerationInd() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest

        while (i < numbInd) {
            int COUNTER2 = numbCopies;
            while ((COUNTER2 > 0) & (i < numbInd)) {
```

```
                //System.out.println(i + "" + l);
                IND2[i] = (BitSet) IND1[ORDERIND[k]].clone();
                //System.out.println(i + " " + counter + " " + k);
                i++;
                COUNTER2 = COUNTER2 - 1;
            }
            k++;
        }
        for (int j = 0; j < numbInd; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*  System.out.println( j+ " IND1 "
                                + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(numbInd);
        //b is false everywhere
        BitSet b = new BitSet();
        //B is the recombinant of l and k
        for (int j = 0; j < length; j++) {
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    b.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
                    b.set(j);
                }
            }
        }
        /*
 System.out.println();
 System.out.println("Ind[l] = " + IND1[l]);
 System.out.println("Ind[k] = " + IND1[k]);
 System.out.println("Recomb = " + b);
         */
        return b;
    }
```

```
// The definitive population  is a recombinant one.
    private static void recombinationInd() {
        //IND2 is a copy of IND1
        for (int i = 0; i < numbInd; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //IND2 is a recombinant version of IND1
        for (int l = 0; l < numbInd; l++) {
            IND2[l] = recInd(l);
        }
        //IND2 is transferred to IND1
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }

// Individual  a is mutated at a random place.
    private static BitSet pointMutationInd(BitSet a) {
        int l = r.nextInt(length);
        a.flip(l);
        return a;
    }

// Individuals are mutated
    private static void mutationInd() {
        for (int i = 0; i < numbInd; i++) {
            double p = r.nextDouble();
            if (p < mutationRate) {
                IND1[i] = pointMutationInd(IND1[i]);
            }
        }
    }

// Given a goal, it is guessed by evolution
    private static void evolution() {

        //The new generation is produced
        //as copies of the fittest
        newGenerationInd();
        if (recombination) {
            recombinationInd();
        }
        //The new population is subjected to mutation
        mutationInd();
        gen++;
```

```
        problem = sortingInd();
    }

    private static void report1() {
        System.out.println("Task ended by generation =" + gen);
        System.out.println("Number of trials= " + gen * numbInd);
        BitSet match = (BitSet) IND1[ORDERIND[0]].clone();
        System.out.println("Match  = " + match);
        System.out.println("Found binary target = "
                + decode(match));
        System.out.println("cumulatedNumbGen = " + cumNumbGen);
        System.out.println("cumulatedNumbInd = " + cumNumbInd);
    }

    private static void report2() {
        System.out.println();
        System.out.println("FINAL REPORT");
        System.out.println("length until " + maxLength);
        System.out.println("UnitLength = " + unitLength);
        System.out.println("Ind/generation = " + numbInd);
        System.out.println("Mutation rate = " + mutationRate);
        System.out.println("Recombination  = " + recombination);
        System.out.println("Reinitialization  = " + reinit);
        System.out.println("Number of guessed targets = "
                + counter);
        System.out.println("totalNumbGen = " + cumNumbGen);
        System.out.println("totalNumbInd = " + cumNumbInd);
    }

//The population must match all patterns, one by one
    public static void main(String[] args) {
        setParameters();
        beginning();
        int percent = 0;
        gen = 0;
        while ((problem)) {
            //System.out.println( "gen of Ind = " + gen);
            evolution();
            //if a pattern has  been guessed ( problem = false)
            //then, pass to the next one.
            if ((problem == false)) {
                cumNumbGen = cumNumbGen + gen;
                cumNumbInd = cumNumbInd + gen * numbInd;
                report1();
                int percentage = (int) 100
                        * counter / (maxLength);
```

```
                if (percentage > percent) {
                    percent = percentage;
                    System.out.println("Done work = " + percent + "%");
                }

                if (counter < maxNumbUnits) {
                    counter = counter + 1;
                    System.out.println("");
                    target = defineTarget(counter);
                    //System.out.println("Decoded = " + decode(target));
                    if (reinit) {
                        initializePop();
                    }
                    gen = 0;
                    problem = true;
                }
            }
        }
        report2();
    }
}//End of Program C64, randSem
```

65 ***Exercise*** *Run the program and play with the code to understand its design and operation.*

66 ***Exercise*** *Add to the previous program suitable code to execute repetitions with mean and standard deviation. Answer*

67 ***Exercise*** *Study the transition of one individual in the population to two individuals with and without recombination: you will be studying the effect of parallelism by contrast with sex and recombination. Answer*

68 ***Exercise*** *Run the program 66 with maxNumbUnits = 1; and numbInd = 10; and with varying increasing number of unitLength from 2 to, say, 16384. Study the change in the number of generations that are necessary to achieve the task. Conclude that the proposed semantics offers no challenge to evolution. Answer*

4.5 A discontinuous semantics

In evolution, the semantics is determined by the selection mechanism: it determines what is allowed to get reproduced and what not. We say: if a DNA message can be reproduced, it makes sense, but not otherwise. In nature, natural selection arises as an emergent feature from the interaction of the whole DNA and its biochemical environment in relation with the surrounding exterior milieu. In our simulations, selection is posed by hand. In our two previous exercises, guessing prime numbers and guessing a sequence of random strings, selection was defined in such a way that any likeness, no matter how tiny, of an organism

with the predefined pattern was rewarded. Our next exercise considers the possibility to restrict rewarding to those individuals that match the predefined target along sufficiently long regions.

We need to consider matching of predefined targets along very long regions because of facts as the following: *S. cerevisiae* contains 6357 genes (Meyer, [49] 2003) but only around 18.7% of them proved essential for surviving in a rich glucose milieu (Giaever, [**?**] 2002; Ihmels, [37] 2007). This is equivalent to say that the present *S. cerevisiae* has a set of at least 1271 genes that cannot function if anyone of them is damaged. Let us suppose that each gene encodes for a protein with 100 amino acids, so we need to specify information for 127100 amino acids or equivalently for 3811300 DNA bases. If somebody can explain 99.9% of them by some evolutionary mechanism, we still need an explanation for 381 bases. Our purpose is to argue that this is a huge number that is more near to infinity than to us. To that aim we will study a discontinuous version of the NK model, a discontinuous semantics, as follows:

Let us suppose that we are trying to guess the string

```
1010101010
```

In the previous exercises, the individual

```
1010101010
```

was given a fitness of 10, because the target and the individual coincide after alignment in 10 sites. By contrast, the individual

```
1111111111
```

was given a fitness of 5 because target and organism have 5 sites of coincidence. If we count the number of mismatches, we get the **Hamming distance**. The fitness measures coincidences and is the complement of the distance with respect to N, the length of all strings.

In the next simulation, we change the rules toward discontinuity: if we say that the level of matching must be 3, then we must align the two sequences and take all substring of the pattern 3 bits long to see whether or not the homologous subsequences in the organism coincide.

So, if the pattern and the organism are:

```
1110111011
1011101101
```

the corresponding subsequences 3 bits long are:

```
111 110 101 011 111 110 101 011
101 011 111 110 101 011 110 101
```

Next, we compare all subsequences in the corresponding order and count the number of mismatches: Our procedure defines the **NK orphan distance** for binary strings of length N taking substring K-bits long. To avoid orphans at the upper tail of strings, these must be circularized. Instead of counting mismatches, we can count coincidences to define the fitness of each individual when the purpose is to match a given pattern. The fitness of this individual is zero. By contrast with the old rule, the fitness should have been 5. Thus, we immediately see that our new rule produces a lot of mortality, which is the most direct impact of semantics discontinuity over the population.

69 *The code*

The next code is suitable to study the effect of the degree of discontinuity of the semantics upon the number of generations and of individuals necessary to fit that string.

```
/*Program C69, DiscSem:

We introduce a discontinuous semantics
with regulated level of discontinuity.

Our aim is to study the effect of the level of
discontinuity over the number of generations
that are necessary to fit a given pattern.

Let us suppose that the level of discontinuity = 3,
and that the pattern and one organism are:


Pattern:  1110111011
Organism: 1011101101


then we form  all  the subsequences of the pattern
3 bits long and the corresponding ones of the organism:


Pattern:  111 110 101 011 111 110 101 011
Organism: 101 011 111 110 101 011 110 101


Next, we compare all subsequences in the corresponding
order and count the number of coincidences: the fitness
of this individual is zero.
 */
package ejvol3v2p;

import java.util.BitSet;
```

```
import java.util.Random;

public class DiscSem {

    //Number of repetitions
    private static final int NREP = 1;
    //level of discontinuity
    private static final int DISCLEVEL  = 34;
    //Length of pattern to match
    private static final int PATTLENGTH = 34;
    //A population of binary numbers is considered.
    //The number of individuals is constant
    private static final int NUMBIND = 10;


    private static final BitSet IND1[] = new BitSet[NUMBIND];
    private static final BitSet IND2[] = new BitSet[NUMBIND];
    private static double FitnessInd[]
            = new double[NUMBIND];
    private static final int ORDERIND[] = new int[NUMBIND];
    private static int gen;

    private static boolean problem = true;
    private static BitSet target = new BitSet();
    //RECOMBINATION = true =  is enabled
    private static final boolean RECOMBINATION = true;
    //REINIT = true: the pop is initialized with a new task
    private static final boolean REINIT = true;
    //Mutation rate per individual
    private static final double MUTATIONRATE = 0.3;


    //Number of generations for each repetition
    private static final long NGEN[] = new long[NREP];


    // Turn on  the random generator
    static Random r = new Random();


    //Converts a BitSet value into a boolean string
    static private String decode(BitSet t) {
        String s = "";
        for (int i = 0; i < PATTLENGTH; i++) {
            if (t.get(i) == true) {
                s = s + "1";
```

```
            } else {
                s = s + "0";
            }
        }
        return s;
    }

    //A random binary pattern  or target is generated
    //PATTLENGTH bits long
    private static BitSet defineTarget(int PATTLENGTH) {
        BitSet pattern = new BitSet();
        for (int i = 0; i < PATTLENGTH; i++) {
            int l = r.nextInt(2);
            if (l == 1) {
                pattern.set(i);
            }
        }

        System.out.println("BitSet = "
                           + (BitSet) pattern);
  System.out.println("Binary  target = "
                     + decode(pattern));


        return pattern;
    }

    //IND1 is a population of binary strings.
    //The BitSet class is used to instantiate
    //binary strings as individuals
    private static void initializePop() {
        BitSet s = new BitSet();
        //IND1 is false everywhere
        for (int i = 0; i < NUMBIND; i++) {
            IND1[i] = (BitSet) s.clone();
        }
        //Reinitialization with random truth values.
        for (int i = 0; i < NUMBIND; i++) {
            for (int j = 0; j < PATTLENGTH; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
        }
        // System.out.println("IND11 : " + IND1[1].get(1));
```

```
    }

    //Initialization
    private static void beginning() {
        target = defineTarget(PATTLENGTH);
        initializePop();
    }

// Each individual is tested
// for how good it matches the target
    static private int assessInd(BitSet s) {
        //The fitness of s is the number of substrings
        //DISCLEVEL bits long that s matches the target
        int fitness = 0;
        for (int i = 0; i < PATTLENGTH - DISCLEVEL + 1; i++) {
            if (s.get(i, i + DISCLEVEL)
                    .equals(target.get(i, i + DISCLEVEL))) {
                    fitness = fitness + 1;
                }
        }
        /*
    System.out.print("target = " + target);
 System.out.print(" s = " + s + "fitness = "
 + fitness);
 System.out.println();
         */
        return fitness;
    }

// Individuals are sorted by fitness.
// the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        double FitnessIndc[];
        FitnessInd = new double[NUMBIND];
        FitnessIndc = new double[NUMBIND];
        for (int i = 0; i < NUMBIND; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < NUMBIND; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
```

```
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < NUMBIND; i++) {
            Champ = 0;
            for (int j = 0; j < NUMBIND; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array ORDERIND orders individuals by  fitness:
            //The fittest is number zero
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        /*System.out.println("Gen = " +gen);
  System.out.println("Champ = " + IND1[ORDERIND[0]]);
  for(int i = 0; i< NUMBIND;i++)
  {
   System.out.println(ORDERIND[i] + "     " +
            FitnessInd[ORDERIND[i]]);
  }
         */
        // if Pattern has been guessed
        if (FitnessInd[ORDERIND[0]]
                == PATTLENGTH - DISCLEVEL + 1) {
            problem = false;
        //System.out.println("Champ = " + IND1[ORDERIND[0]]);
        }
        return problem;
    }

// The new generation is made
    private static void newGenerationIND() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest

        while (i < NUMBIND) {
            int counter = numbCopies;
            while ((counter > 0) & (i < NUMBIND)) {
                //System.out.println(i + "" + l);
                IND2[i] = (BitSet) IND1[ORDERIND[k]].clone();
            //System.out.println(i + " " + counter + " " + k);
```

```
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int j = 0; j < NUMBIND; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*  System.out.println( j+ " IND1 "
                                + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(NUMBIND);
        //b is false everywhere
        BitSet b = new BitSet();
        //B is the recombinant of l and k
        for (int j = 0; j < PATTLENGTH; j++) {
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    b.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
                    b.set(j);
                }
            }
        }
        /*
 System.out.println();
 System.out.println("Ind[l] = " + IND1[l]);
 System.out.println("Ind[k] = " + IND1[k]);
 System.out.println("Recomb = " + b);
         */
        return b;
    }

// The definitive population  is a recombinant one.
    private static void recombinationIND() {
```

```
        //Ind2 is a copy of IND1
        for (int i = 0; i < NUMBIND; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //Ind2 is a recombinant version of IND1
        for (int l = 0; l < NUMBIND; l++) {
            IND2[l] = recInd(l);
        }
        //Ind2 is transferred to IND1
        for (int i = 0; i < NUMBIND; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }

// Individual  a is mutated at a random place.
    private static BitSet pointMutationIND(BitSet a) {
        int l = r.nextInt(PATTLENGTH);
        a.flip(l);
        return a;
    }

// Individuals are mutated
    private static void mutationIND() {
        for (int i = 0; i < NUMBIND; i++) {
            double p = r.nextDouble();
            if (p < MUTATIONRATE) {
                IND1[i] = pointMutationIND(IND1[i]);
            }
        }
    }

// Given a goal, it is guessed by evolution
    private static void evolution() {

        //The new generation is produced
        //as copies of the fittest
        newGenerationIND();
        if (RECOMBINATION) {
            recombinationIND();
        }
        //The new population is subjected to mutation
        mutationIND();
        gen++;
        if (gen % 100000 == 0)
            System.out.println(" Generation = " + gen);
        problem = sortingInd();
```

```
    }


    private static void reportParameters() {
        System.out.println();
        System.out.println("FINAL REPORT");
        System.out.println("Discontinuity level = " + DISCLEVEL);
        System.out.println("Pattern length = " + PATTLENGTH);
        System.out.println("Ind/generation = " + NUMBIND);
        System.out.println("Mutation rate = " + MUTATIONRATE);
        System.out.println("Recombination  = " + RECOMBINATION);
        System.out.println("Reinitialization  = " + REINIT);
    }



//The population must match a  patterns
    private static void task() {
        beginning();
        gen = 0;
        while ((problem)) {
            //System.out.println( "gen of Ind = " + gen);
            evolution();
        }
        //report3();
    }

    public static void main(String[] args) {
        reportParameters();
        for (int i = 0; i < NREP; i++) {
            System.out.println("Repetition = " + i);
            problem = true;
            task();
            NGEN[i] = gen;

            System.out.println(" Generations = " + gen);

        }
        double sumGen = 0;

        for (int i = 0; i < NREP; i++) {
            sumGen = sumGen + NGEN[i];

        }
        double meanGEN = sumGen / NREP;
```

```
        double sumDGen = 0;

        for (int i = 0; i < NREP; i++) {
            sumDGen = sumDGen + Math.pow((NGEN[i] - meanGEN), 2);

        }

        double stanDevGen = Math.sqrt(sumDGen / NREP);

        System.out.println("Repetitions = " + NREP);
        System.out.println("MeanGEN = " + meanGEN);
        System.out.println("stanDevGen = " + stanDevGen);

    }
}//End of Program C69, DiscSem
```

70 ***Exercise*** *Run the program and play with the code.*

71 ***Exercise*** *How does evolution behave when challenged by tasks with a semantics of increasing discontinuity? Answer*

72 ***Exercise*** *Design a program using BitSet to match a predefined random boolean pattern by a random walk. Record the effect of the length of the pattern on the number of steps or generations necessary to match the pattern. Compare the performance of randomness with that of evolution under a discontinuous semantics. Answer*

73 ***Challenge*** *Prove else refute, by whatever means, that* ***stagnation is the brand of semantic discontinuity****. More accurately, prove else refute that, under semantic discontinuity, the evolution of a population consists of very long periods of ecstasy that are interrupted by very rapid changes. Answer*

74 ***Exercise*** *Devise your own semantics and a simulation in which higher peaks are ever more difficult to reach and therefore the periods of ecstasy are increasingly larger. Answer*

75 ***Challenge*** *Relate our discussion on semantic discontinuity and ecstasy with the theory of Gould and Eldridge, who propose that evolution takes place in short, sudden leaps, interspersed with long periods of stability (Cromie, 1997).*

4.6 Conclusion

In linguistics, a string makes sense when it is 'understood' by the interlocutor. In genetics, a DNA string makes sense when it encodes for a viable organism. In genetics as in any human language, the strings that make sense are surrounded by a sea of nonsense, because of which most mutations are deleterious or lethal (kill the meaning or produce inviable individuals). Memory and functional restrictions, plus recursiveness are properties that seem to be sufficient cause of the nonsense sea as in linguistics as in genetics. This sea allows noise stability

and self-correction as in genetics as in linguistics. When one makes a random walk, the sea of nonsense is seem as the sudden lost of viability, a property that is also observed by Java programs: we say that we have a semantic discontinuity. We have confronted evolution against an increasing level of semantic discontinuity and we have found that after a hard battle, evolution recedes. A high level of semantic discontinuity, as in the NK-distance, may cause evolution to behave worst than a random walk in relation with the amount of needed resources (the global number of individuals). We also have managed to cause evolution to have periods of ecstasy as long as desired.

Chapter 5

Is there any limit to evolution?

```
Natural evolution looks at presently heavily tired.
```

76 ***Motivation*** *Evolution functions as a self-regulated diffusive process, which at presently looks very tired: not too many new species are seen in the field or in the lab. So, we ask: is this appreciation real or is it an effect of our very short existence?*

77 ***Purpose*** *We study in this chapter a simple model that is appropriate to study the boundedness of the possibilities of evolution as a diffusive process. It is* ***SAT*** *or the satisfiability problem of mathematical logic. This model is so good that it is easy to forget that it is just a model and that reality is much more complex. Various errors were detected in the first version and corrected for version 2 in the programs of this chapter.*

5.1 SAT as an abstraction from language

The phrase '3 is even' is false, while '4 is even' is true. In general, any expression that can be classified as true or false given a reference set is called a **proposition**.

Let us consider the expression over integer numbers 'p(x) = x is even'. This expression is true for $x = 8$, but for $x = 7$, $p(7)$ is false, and for each integer number it is true else false. So, 'p(x) = x is even' is not a proposition but becomes a proposition if we instantiate x, i.e., if we replace 'x' by a given integer number. An expression of this type over a given universal set is called a **predicate**.

A language composed of predicates is appropriate for describing events that come classified in one of two classes. Thus, it is the more simple setting under which one can study the world and make science.

The use of predicates is usual for humans, intelligent beings. To study the possibilities of natural evolution, we prefer for the time being to restrict ourselves to propositions, which are appropriate to describe concrete events of nature.

One can generate **formulas** or compound propositions beginning from elementary propositions or variables or from other formulas by the use of logical operators:

- The NOT or negation operator: which takes one proposition p and produces the proposition denoted $NOT(p)$ or $\sim p$ that is false whenever p is true and is true whenever p is false.

- The *AND* operator takes two propositions p and q and produces a third one, noted $AND(p,q)$ or $p \wedge q$ that is true if and only both of p **and** q are true.

- The *OR* operator takes two formulas p and q and produces a third one, denoted $OR(p,q)$ or $p \vee q$ that is true if one of p **or** q is true and false if both are false.

To go further, let us clarify that the NOT operator is mono-ary, i.e., it takes a proposition and produces another, while the AND and OR operators are binary, i.e., take two propositions and produce one. Nevertheless, AND and OR can be extended very easily to more than two propositions, say $p \vee q \vee r$ can be taken as $p \vee (q \vee r)$ or $(p \vee q) \vee r$ and in both cases they compute to the same final output for every truth assignment of the variables.

One can see that with the set of operators $\{NOT, AND, OR\}$, one has more than enough to generate all possible operators that take one, two or any number of inputs to produce one compound proposition. For that reason, we restrict ourselves to these operators.

One can always find an assignment of truth values for p such that $NOT(p)$ could be true: just assign FALSE to p. Also, we can assign TRUE to both p and q to make $AND(p,q) = p \wedge q$ true. If we assign TRUE to p and FALSE to q, $OR(p,q) = p \vee q$ becomes true. But one cannot find an assignment of truth values for $p \wedge \sim p = AND(p, NOT(p))$ such that this compound formula could become true, because we deny the possibility that a proposition could be true and false at the same time.

We say that a proposition is **satisfiable** when we can show a truth assignment to its variables that makes the proposition true. We also say that the truth assignment satisfies the proposition or the formula. Example: $p \wedge (q \vee (\sim r)) = AND(p, OR(q, NOT(r)))$ is true if p is true and q is true, while r could be true or false. So, this formula is satisfiable. The proposition $OR(r, NOT(r))$ is true for any truth value of r, so it is satisfiable. $AND(r, NOT(r))$ is not.

SAT or the satisfiability problem of mathematical logic consists in studying the assignment of truth values to the variables of a proposition with the purpose of making it true. SAT proposes a question in regard with a given proposition: Does this proposition is satisfiable? To answer YES you must show a satisfying assignment otherwise you must prove that all assignments evaluate to false.

Which is the relation of satisfiability with practice? If we can guarantee that a proposition is satisfiable at least for one truth assignment, that would mean that a system with a function described by the given proposition can be implemented and run. Of course, we must take care of not given to the compound proposition truth assignments that causes false as final output.

In mathematics, to decide whether a formula is satisfiable or not, one must try out every possible truth assignment. But to consider the diffusive possibilities of evolution, we will consider just a given number of random trials, because evolution does not have a systematic mean to execute every possible truth assignment.

To be satisfiable is a restriction imposed over propositions. We feel that this restriction is a terrible one because it amounts to the requirement to speak, may be a lot, without falling into inner contradictions.

Now, let us ask ourselves a fundamental question: In which way is SAT fundamentally related to evolution as a diffusive process? In other words: Is evolution fundamentally restricted? The answer is by no means obvious, for suppose that one goes to a trial at the court. If we are ready to say the truth and only the truth, we never will commit contradictions. So, to pretend that evolution could commit contradictions seems to be equivalent to pretend that evolution could lie. That is unthinkable.

Nevertheless, there is another insight that looks at evolution as fundamentally restricted by a type of constrain that is well captured by SAT: atoms or molecules do not contradict one another, but processes can: let us consider a poisonous snake: it must produce an inhibitor of, say, the transmission of chemical signals among synapses in a potential enemy **but not** in its own body. So, the arising of a poisonous snake is an inherently constrained problem. Now, is this situation fundamentally general and inherent to evolution and life? Yes, it is: every catalyzed reaction in the cell must be inhibited, must be denied. The most obvious inhibitor is formed by the outputs of the reactions if they are accumulated in great quantities. Moreover, life is not an acceleration of natural processes; instead life is created at the biochemical level by enzymes. And many enzymes have allosteric regulation, i.e. natural mechanisms to get inhibited. So, we conclude that life, such as we know it, is fundamentally restricted by self-constrains that can be expressed verbally by expressions of the sort *do this but not that*. That is why SAT is good to study the possibilities of evolution ab initio: evolution is not a diffusive process; it is a diffusive process in logical space restricted to satisfiable propositions. A lethal mutation could be seen as a contradiction of evolution.

Our model has the immediate criticism that life is not the avoidance of self-restrictions but the wise administration of that possibility. That is why regulation is at the very heart of life. So, SAT, our model for inherent self-restrictions, could be considered primitive.

5.2 Diffusion in logical space

Evolution has no purpose at all, but it must conform to the possibilities dictated by nature and that we propose to be captured by SAT. So, we will study the diffusion in logical space, in the space of satisfiable formulas to see what limits one can observe.

78 *Generation of propositions*

To compose a simulation of diffusion in logical space, we need in first place an automatic mechanism for the random generation of compound propositions.

We begin our simulation with the definition of two parameters, the number of operators per proposition and the number of variables that are used in it. Example $AND(p, OR(q, NOT(r)))$ has 3 operators and 3 variables.

Let us generate at random a (3,3) proposition (the first 3 is for three operators, the second 3 is for three variables):

1. We generate at random a first operator, let it be NOT. We write $NOT(l)$.

2. We generate at random a second operator, let it be AND. In consequence, we substitute l in $NOT(l)$ by $AND(l,l)$. We get $NOT(AND(l,l))$.

3. We generate at random a second operator, let it be OR. We substitute one of the l's in $NOT(AND(l,l))$, which must be chosen at random, by $OR(l,l)$. Suppose we choose the second l, we get $NOT(AND(l,OR(l,l)))$.

Once we get the final l-proposition, say, $NOT(AND(l,OR(l,l)))$, we replace every instance of l by one of the allowed variables. We have three variables, let us denote them $\{p,q,r\}$, and we replace l in the formula above by a variable taken at random. We get, say, $NOT(AND(p,OR(r,q)))$.

Next, we must give a truth value to each variable to assess the truth value of the overall compound proposition. Suppose we assign the truth values:

- $p \to T$
- $q \to F$
- $r \to T$

We replace these values in the proposition. We get
$NOT(AND(p,OR(r,q))) \to NOT(AND(T,OR(T,F)))$
Thereafter, we evaluate the operator as we already know:
$NOT(AND(T,OR(T,F))) \to NOT(AND(T,T)) \to NOT(T) \to F$
We conclude that this assignment does not satisfy the generated proposition.

We can automate our work with the help of a grammar, i.e., a set of ordered rules to form new words from more primitive ones. The rules are:

Rules for the generation of the l-proposition:

- $l \to OR(l,l)$, the number of $l's$ increases in 1.
- $l \to AND(l,l)$, the number of $l's$ increases in 1.
- $l \to NOT(l)$, the number of $l's$ remains the same.

Rules for converting an l-proposition into a proposition:

- $l \to p$
- $l \to q$
- $l \to r$

Rules for the evaluation of truth values of AND:

- $AND(T,T) \to T$

- $AND(T,F) \rightarrow F$
- $AND(F,T) \rightarrow F$
- $AND(F,F) \rightarrow F$

Rules for the evaluation of truth values of OR:

- $OR(T,T) \rightarrow T$
- $OR(T,F) \rightarrow T$
- $OR(F,T) \rightarrow T$
- $OR(F,F) \rightarrow F$

Rules for the evaluation of truth values of NOT:

- $NOT(T) \rightarrow F$
- $NOT(F) \rightarrow T$

79 *The code for a proposition* *The next program shows how a random proposition is formed with m operators and n variables, with a random truth assignment of the variables and a complete evaluation to false else true.*

```
/* Program C79, Prop.
 * We generate a logical proposition
 * with m operators and n variables
 * chosen at random.

 * Number of variables shall be less than 10 so,
 * they can be named by numbers 0, 1, ... 9
 * that are encoded as chars.

 * Random  equiprobable  truth values are assigned
 * to the chosen variables and
 * the Prop is evaluated to true else false.
 *
 */
package ejvol3v2p;

import java.util.Random;

public class Prop {

    // Turn on  the random generator
    static Random r = new Random();
```

```
    //Number of operators
    private static int nOperators;
    //Number of variables, could be as large as MAXV.
    //Variables will be represented by numbers.
    private static int nVar;
    private static final int MAXV = 1000;
    //Proposition in formation

    //Relative proportions  of different operators
    private static double probA, probO;
    private static int numbL;
    private static final boolean TRUTH[]
            = new boolean[MAXV];




    private static void setParameters() {
        //Number of operators
        nOperators = 10;
        //Number of variables ( < 10)
        nVar = 4;
        //Relative proportions of operators
        probA = 0.4;
        probO = 0.4;
        //probN = 1-probA-probO;
    }


    private static String substitution(String a) {
        //Type of operator is chosen
        int type = -1;
        double h = r.nextDouble();
        if (h < probA) {
            type = 0;
        }
        if ((h < probA + probO) & (h > probA)) {
            type = 1;
        }
        if (h > probA + probO) {
            type = 2;
        }
        //number of 1's is updated.
        if (type < 2) {
            numbL = numbL + 1;
        }
```

```
        //'f'  is replaced by the operator;
        switch (type) {
            case 0:
                a = a.replace("f", "AND(l,l)");
                break;
            case 2:
                a = a.replace("f", "OR(l,l)");
                break;
            case 1:
                a = a.replace("f", "NOT(l)");
                break;
        }
        return a;
    }

    //Composes an l-proposition    with m operators
    //and various l's.
    private static String lProposition() {
        System.out.println("Construction of "
                + "the l-proposition");
        //Proposition is initialized.
        String a = "l";
        //Number of l's is recorded.
        numbL = 1;
        //m operators are generated
        for (int i = 0; i < nOperators; i++)
        {
            //A place of insertion is chosen:
            //a given l must be replaced by the operator.
            int givenl = r.nextInt(numbL);
            int counter = 0;
            for (int j = 0; j < a.length(); j++)
            {
                if (a.charAt(j) == 'l') {
                    if (counter == givenl) {
                        //System.out.println("Old a = " + a);
                        //the place of replacement is marked with 'f'
                        a = a.substring(0, j) + "f"
                                + a.substring(j + 1);
                        //System.out.println("New a = " + a);
                    }
                    counter = counter + 1;
                }
            }
            a = substitution(a);
            System.out.println("l-prop      = " + a);
```

```
        }
        return a;
    }

    //Each l of the l-proposition is replaced
    //by one of the variables.
    //Variables are not letters;
    //they are numbers from 0 to n-1;
    private static String proposition(String a) {
        System.out.println();
        System.out.println("l-prop --> proposition");
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                Integer vari = r.nextInt(nVar);
                String s = vari.toString();
                a = a.substring(0, j) + s
                        + a.substring(j + 1);
            }
        }
        System.out.println("prop        = " + a);
        return a;
    }

    //Variables are given truth values
    private static void defineValuation() {
        for (int i = 0; i < nVar; i++) {
            int j = r.nextInt(2);
            //true else false
            TRUTH[i] = j == 0;
        }
    }

    //The proposition is evaluated
    private static String evaluation(String b) {
        System.out.println("\n\nAssignment = " );
        for (int i = 0; i < nVar; i++)
            System.out.println("Var " + i + "   "
                    +  TRUTH[i] );
        System.out.println();
        System.out.println("Truth evaluation begins");

        for (int j = 0; j < b.length(); j++) {
            Character myChar = b.charAt(j);
            if (Character.isDigit(myChar))
            {
                int k = Character.getNumericValue(myChar);
```

```
                //System.out.println(myChar  + "   " + k);
                if (TRUTH[k] == true) {

                    b = b.substring(0, j) + "T"
                            + b.substring(j + 1);
                } else {
                    b = b.substring(0, j) + "F"
                            + b.substring(j + 1);
                }
            }
        }
        System.out.println("Truth value = " + b);
        return b;
    }


    //The proposition is evaluated until the end
    private static String simplification(String b) {
        while (b.length() > 1) {
            b = b.replace("AND(T,T)", "T");
            b = b.replace("AND(T,F)", "F");
            b = b.replace("AND(F,T)", "F");
            b = b.replace("AND(F,F)", "F");

            b = b.replace("OR(T,T)", "T");
            b = b.replace("OR(T,F)", "T");
            b = b.replace("OR(F,T)", "T");
            b = b.replace("OR(F,F)", "F");

            b = b.replace("NOT(T)", "F");
            b = b.replace("NOT(F)", "T");
            System.out.println("Truth value = " + b);
        }
        return b;
    }

    public static void main(String[] args) {
        setParameters();

        String b = lProposition();
        b = proposition(b);
        defineValuation();
        b = evaluation(b);
        b = simplification(b);
        System.out.println("Final evaluation = " + b);
    }
```

```
}//End of Program C79, Prop
```

80 ***Exercise*** *Run the program and play with the code.*

81 ***Exercise*** *The diversity of variables is restricted by the previous code to a number less than 10. This does not make justice to the overwhelming importance of SAT, which is related to the concept of function. So, make the needed generalization to any number of variables.* *Answer*

82 ***Exercise*** *Add to the previous code some facilities for statistics and study the question of the dependence of the probability to get satisfied on the number of operators and variables.* *Answer*

83 *What do you mean?*

Our logical propositions represent possible sentences of human languages. Say, the proposition $(p \wedge r) \wedge (\sim q \vee p)$ corresponds to the sentence *John wins and smiles and (he is not loved by his girlfriend or he wins)*. As we see, one can get confused by the meaning of a given sentence. Fortunately, a clarifying algorithm exists that reformulates a sentence in clear terms. Let us learn how that algorithm functions over a concrete example, precisely over the aforementioned sentence.

The semantic value of a proposition is given by the form as the proposition gets true else false as a reaction to a given evaluation with suitable truth assignments. So, we pose in a table every possible form of evaluation and we get a truth table. If the proposition is always false for each truth assignment, it is equivalent to the proposition $p \wedge \sim p$ that is a contradiction. The problem becomes more interesting if the proposition or formula is satisfiable, i.e., if there exists at least a truth assignment that evaluates the proposition to true. What do we have in our case? The truth table follows:

Truth table			
p	q	r	$(p \wedge r) \wedge (\sim q \vee p)$
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

Next, we take those lines of the truth table that output true, the first and third ones. The truth values for p,q,r are in the first line TTT, and TFT in the third. The value TTT generates the proposition $(p \wedge q \wedge r)$ while TFT generates $(p \wedge \sim q \wedge r)$. Because we have two channels that produce true as output, we join them with and OR operator: $(p \wedge q \wedge r) \vee (p \wedge \sim q \wedge r)$. This means that the proposition or formula $(p \wedge r) \wedge (\sim q \vee p)$ has exactly the same truth table as $(p \wedge q \wedge r) \vee (p \wedge \sim q \wedge r)$, a fact that one can verify over the corresponding truth table:

Truth table			
p	q	r	$(p \wedge q \wedge r) \vee (p \wedge \sim q \wedge r)$
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

We say that $(p \wedge r) \wedge (\sim q \vee p)$ and $(p \wedge q \wedge r) \vee (p \wedge \sim q \wedge r)$ are equivalent or semantically equivalent because their truth tables coincide.

Let us verbalize the proposition $(p \wedge q \wedge r) \vee (p \wedge \sim q \wedge r)$ to see how much we have gained in clarity: John wins and his girlfriend loves him and he smiles, or, he wins and his girl friend does not love him and he smiles. The new sentences is so clear that we can see at once that the girlfriend of John causes no effect over the truth table and that, therefore, we can simplify the original sentence to the final expression $p \wedge r$: John wins and smiles. Please, over the truth table for the three variables, p, q, r, with eight rows, calculate the truth values of $p \wedge r$ to verify that we have a new table that is equivalent to the very original one.

A proposition of the form $(p \wedge q \wedge r) \vee (p \wedge \sim q \wedge r)$ is, by definition, in **normal disjunctive full form**: it is an OR package of propositions, every one of which is string of variables, negated or not, joined by the AND operator. Each AND string is called a clause. The normal disjunctive form is full and normal when every clause contains all variables of the discourse without repetitions. The formula $(p \wedge q \wedge r) \vee (p \wedge \sim q \wedge r)$ has two clauses over three variables.

The above procedure shows that every satisfiable proposition is equivalent to a given disjunctive normal full form. The converse is also true: any disjunctive normal full form is always satisfiable: just take the first clause, and compose a truth assignment by instinct: true for a variable when it is not negated and false when it is negated. In that way the first clause gets true and with it the whole proposition. For instance, if the first clause is $p \wedge \sim q \wedge r \sim s$ then we can give the assignment $TFTF$ and the proposition becomes automatically true. Since this procedure can be repeated over any formula that has at least one satisfying assignment, we can conclude: the set of satisfiable propositions and the set of propositions in disjunctive normal full form coincide.

5.3 Solving SAT

SAT is a decision problem for a given proposition: one must decide whether or not there exists at least one satisfying assignment. To decide YES, one must show one satisfying truth assignment. To decide NO, one must show that all possible assignments cause the proposition to get false.

How difficult is to solve SAT?

SAT is related with the problem of expressing oneself without falling into contradictions while trying to say something interesting. Everyone knows by personal experience that this is not at all an easy task, so we could expect that SAT demands huge amounts of computing resources to be solved.

Let us try to solve SAT over the set of normal disjunctive full forms. Let us keep in mind that this family is satisfiable and that one can find a satisfying truth assignment by mere inspection. We are testing our machinery.

84 *SAT vs. randomness*

Our setting corresponds to the situation of an experimentalist that does not know that given propositions are satisfiable. We make two definitions. First: α as the ratio of clauses to variables. Second, a proposition is unsatisfiable when no satisfying assignment can be found after a certain number of random assignments. This definition is appropriate for the study of evolution.

What is expected under this setting?

Since we restrict ourselves to disjunctive normal forms, studied propositions are all of the same architecture as $(p \wedge q \wedge r) \vee (p \wedge \sim q \wedge r)$. The more clauses are added, the more probably is to get a true value, because we add more ORs, i.e., more possibilities to get true. So for large values of α, we have a very high probability of getting satisfiable or a very low probability of getting unsatisfiable. For low values of α, we have few channels of getting true, so the probability of being satisfiable is nearly zero or the probability of being unsatisfiable is nearly one.

Let us test our expectations for disjunctive normal forms, when a new clause adds one more channel of getting true: Is it true that α, the ratio of the number of clauses to that of variables, and the probability of getting unsatisfiable are inversely related for disjunctive normal forms?

85 *The code*

The next program composes propositions in disjunctive normal full form (that are satisfiable). We study the incidence of α, the ratio of clauses to variables, over the fraction of assignments that evaluate to false.

```
/* Program C85, DisjuncForm.
 * We study the dependence of the probability
 * of getting true for disjunctive normal forms
 * depending on alpha, the ratio of
 * clauses to variables.
 * The prob of negation of a variable is 0.5.
 * Random   equiprobable truth values are assigned
 * to the  variables and
 * the prop is evaluated to true else false.
 *
 * In mode single, one can see the construction
 * and evaluation of each  proposition.
```

```
 * In mode multiple,
 * we generate NPROPS propositions at random and
 * inquire over the fraction of
 * unsatisfiable propositions.
 * To decide whether or not a prop is satisfiable,
 * we make   nAssign random truth assignments.
 * The  dissatisfaction ratio (the proportion of
 * false values) is reported for each prop.
 *
 */
package ejvol3v2p;

import java.util.Random;

public class DisjuncForm {

// Turn on  the random generator
    static Random r = new Random();

    //Number of clauses
    private static int nClauses;
    //Number of variables, could be as large as MAXV.
    //Variables will be represented by numbers.
    private static int nVar;
    private static final int MAXV = 1000;
    //Proposition in formation
    private static String a;
    private static String b;
    private static String c;

    private static int NPROPS = 1000;
    private static final boolean TRUTH[]
            = new boolean[MAXV * MAXV];
    //The assignment of variables is recorded in VAR[]
    private static final int VAR[]
            = new int[MAXV * MAXV];
    //Number of random trials
    private static int nAssign;
    private static double disSatFraction;
    private static boolean single = false;
    private static double alpha, nn;

    private static void setParameters() {
        //Number of clauses
        nClauses = 3;
        //Number of variables
```

```
        nVar = 5;
        //Number of times that each prop is tested,
        //valid in  mode 'single'.
        nAssign = 7* (int) Math.pow(2, nVar);
        //Number of propositions
        NPROPS = 100;
        alpha = nClauses;
        nn = nVar;
        alpha = nClauses / nn;
    }


    private static void printParameters()
    {
        System.out.println("nClauses = " + nClauses);
        System.out.println("nVar = " + nVar);
        System.out.print("alpha = " + alpha);
        System.out.println();
    }

    //Not operators are inserted with prob 0.5
    private static String negations() {
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                int d = r.nextInt(2);
                if (d == 0) {
                    a = a.substring(0, j) + "f"
                            + a.substring(j + 1);
                }
            }
        }
        a = a.replace("f", "Nl");
        return a;
    }

    //Composes a random normal disjunctive full l-proposition
    // with nClauses clauses.
    //The NOT operator is included for each l
    //with probability 0.5.
    private static String lProposition() {
        //System.out.println("Construction of the l-proposition");
        //Proposition is initialized.
        a = "(l";
        for (int i = 0; i < nClauses - 1; i++) {
            for (int j = 0; j < nVar - 1; j++) {
                a = a.concat("Al");
```

```
            }
            a = a.concat(") OR (l");
        }
        for (int j = 0; j < nVar - 1; j++) {
            a = a.concat("Al");
        }
        a = a.concat(")");
        if (single) {
            System.out.println("l-prop without NOT = " + a);
        }
        negations();
        if (single) {
            System.out.println("l-prop with NOTs   = " + a);
        }
        return a;
    }


    //Each l of the l-proposition is replaced by one of
    //the variables. All variables appear in each clause.
    //Variables are not letters;
    //they are numbers from 0 to nVar-1;
    private static String proposition(String a) {
        //System.out.println();
        //System.out.println("l-prop --> proposition");
        int u = 0;
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                Integer vari = u;
                String s = vari.toString();
                a = a.substring(0, j) + s
                        + a.substring(j + 1);
                VAR[u] = vari;
                u = u + 1;
                if (u == nVar) {
                    u = 0;
                }
            }
        }
        if (single) {
            System.out.println("prop  = " + a);
        }
        return a;
    }

    //Variables are given truth values
```

```
    private static void defineValuation() {
        //System.out.println();
        for (int i = 0; i < nVar; i++) {
            int j = r.nextInt(2);
            /*
            //Equivalent to
            if (j == 0) {
            TRUTH[i] = true;
            } else {
            TRUTH[i] = false;
            }
             */
            //TRUTH[i] = (j == 0);
            TRUTH[i] = j == 0;
            //System.out.println(TRUTH[i]);
        }
    }

//The proposition is evaluated
    private static String evaluation() {
        System.out.println("\n\nTruth assignment "
                + "to each of " + nVar + " variables = " );
        for (int i = 0; i < nVar; i++)
            System.out.println(i + "   " + TRUTH[i] );
        if (single) {
            System.out.println();
            System.out.println("Truth evaluation begins");
        }

        //Variables observe the same order in all clauses
        int u = 0;
        for (int j = 0; j < b.length(); j++)
        {
            if (b.charAt(j) == 'l')
            {
                if (TRUTH[VAR[u]] == true)
                {
                    b = b.substring(0, j) + "T"
                            + b.substring(j + 1);
                } else {
                    b = b.substring(0, j) + "F"
                            + b.substring(j + 1);
                }
                u = u + 1;
                //Restart counting for the next clause
                u = u% nVar;
```

```
            }
        }


        if (single) {
            System.out.println("Truth valuer = " + b);
        }
        return b;
    }

//The proposition is evaluated until the end
    //Not operators are evaluated at the
    //very beginning for they apply over the next
    //to the right variable.
    private static String simplification() {
        while (b.length() > 1) {
            b = b.replace("NT", "F");
            b = b.replace("NF", "T");
            if (single) {
                System.out.println("Truth value = " + b);
            }
            b = b.replace("TAT", "T");
            b = b.replace("TAF", "F");
            b = b.replace("FAT", "F");
            b = b.replace("FAF", "F");

            if (single) {
                System.out.println("Truth value = " + b);
            }

            b = b.replace("(T)", "T");
            b = b.replace("(F)", "F");

            if (single) {
                System.out.println("Truth value = " + b);
            }

            b = b.replace("T OR T", "T");
            b = b.replace("T OR F", "T");
            b = b.replace("F OR T", "T");
            b = b.replace("F OR F", "F");

            if (single) {
                System.out.println("Truth value = " + b);
            }
        }
```

```
        return b;
    }

    //A prop is constructed and nAssign truth
    //assignments are done. The fraction of
    //false outputs is reported.
    private static void planSingle() {
        b = lProposition();
        c = b;
        proposition(b);
        double counter = 0;
        for (int i = 0; i < nAssign; i++) {
            defineValuation();
            evaluation();
            b = simplification();
            if (b.equals("F")) {
                counter = counter + 1;
            }
            b = c;
        }
        disSatFraction = counter / nAssign;
        System.out.println("disSat fraction of 1 prop = "
                + disSatFraction);
    }


    /* We generate at random NPROPS props, and report the
   * fraction of them that is not satisfiable.
   * A prop is not satisfiable when there exists no
   * truth assignment that satisfies it.
   * If after assign random trials, no satisfying
   * assignment has been found, the prop is declared
   * as not satisfiable */
    private static void planMultiple() {
        double counter = 0;
        for (int i = 0; i < NPROPS; i++) {
            planSingle();
            if (disSatFraction == 1) {
                counter = counter + 1;
            }
        }
        double d = counter / NPROPS;
        System.out.println("UnSat fraction of prop= " + d);
    }

    public static void main(String[] args) {
```

```
        //The random generator is commanded
        //to produce the same sequence.
        //r.setSeed(23);
        setParameters();
        printParameters();
        single = true;
        planSingle(); //one prop is studied
        //single = false;
        //planMultiple(); //many props are studied

    }
}// End of  Program C85, DisjuncForm
```

86 ***Exercise*** *Play with the code and try to understand it. Plan and carry out a series of experiments to test the expected relation between α, the ratio of clauses to variables, and the probability of getting unsatisfiable. Explain the results.* Answer

87 ***Exercise*** *Refurnish the previous code to answer to the following question: How many random trials are necessary to find a satisfying assignment for a disjunctive normal full form, which is always satisfiable?* Answer

We find that the mean number of random trials to get a satisfying assignment to a normal disjunctive form is exponential in the number of variables while the number of clauses plays a minor role. So, we have a reason to think that for a random complex proposition, the few satisfying assignments are perfectly hidden amidst a multitude of assignments that are not satisfying. In other words, we have a challenging problem for evolution.

5.4 SAT dares evolution

When we use evolution to solve a problem, we must encode the problem in evolutionary language and use a population of individuals whose fitness is determined by the degree in which they represent solutions to the problem.

For SAT, our individuals are valuations, which are composed of strings of true-else-false values that are assigned to a given set of variables, say TFTTF are assigned in the listed order to variables 0 to 4 in a formula like (0A1AN2A3AN4) OR (N0A1AN2AN3A4) which is valuated to (TAFANTATANF) OR (NTAFANTANTAF), where A is a shorthand for AND and N for NOT. Now, the first clause evaluates to F and the second to F, so the proposition is false. But we have a problem: a proposition in normal disjunctive form is satisfiable if at least one clause is. So, this problem produces an all else nothing fitness function: as we already know, this is precisely the type of problems for which evolution is not guaranteed to function, because one does not have the opportunity of enriching a tentative solution through the accumulation of small change. Nevertheless, let us concede us an opportunity.

88 ***The code*** *The next program solves SAT using a GA (genetic algorithm = a simulation of evolution). The mean number of individuals to solve a given formula is reported.*

```
/* Program C88, HowManyEvol.
 * We solve SAT by means of evolution.
 *
 * We take disjunctive normal forms,
 * which are always satisfiable, and inquire
 * over the number of   individuals
 * that are necessary
 * to get a satisfying assignment.
 *

 * The prob of negation of a variable is 0.5.
 * Random   equiprobable truth values are assigned
 * to the  variables and
 * the prop is evaluated to true else false.
 *
 * An statistical facility is added.
 */
package ejvol3v2p;

import java.util.BitSet;
import java.util.Random;

public class HowManyEvol {

    // Turn on  the random generator
    static Random r = new Random();

    //Number of clauses
    private static int nClauses;
    //Number of variables, could be as large as MAXV.
    //Variables will be represented by numbers.
    private static int nVar;
    private static final int MAXV = 1000;
    //Proposition in formation
    private static String a, b, c;

    private static int nProps = 1000;

    //The variables assignment is recorded in VAR []
    private static boolean print = false;
    private static double alpha, nn;
    private static boolean problem = true;

    //A population of binary numbers is considered.
    //The number of individuals is constant
```

```
    private static int numbInd;
    private static final int MAXNUMBIND = 10000;
    private static final BitSet IND1[]
                    = new BitSet[MAXNUMBIND];
    private static final BitSet IND2[]
                    = new BitSet[MAXNUMBIND];
    private static int FitnessInd[]
                    = new int[MAXNUMBIND];
    private static final int ORDERIND[]
                    = new int[MAXNUMBIND];
    private static int gen;
    private static boolean recombination;

    private static double mutationRate;
    private static final int MAXNREP = 100;
    //memory for statistics
    private static final long NGEN[] = new long[MAXNREP];
    private static final long NIND[] = new long[MAXNREP];

    private static void setParameters() {
        //Number of clauses
        nClauses = 5;
        //Number of variables
        nVar = 3;
        //Number of propositions
        nProps = 10;
        numbInd = 10;
        recombination = true; //true = recombination is enabled
        mutationRate = 0.8;
        alpha = nClauses;
        nn = nVar;
        alpha = nClauses / nn;
    }

    //Not operators are inserted with prob 0.5
    private static String negations() {
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                int d = r.nextInt(2);
                if (d == 0) {
                    a = a.substring(0, j) + "f"
                            + a.substring(j + 1);
                }
            }
        }
        a = a.replace("f", "Nl");
```

```
        return a;
    }

    //Composes a random normal disjunctive full l-proposition
    // with nClauses clauses. The NOT operator is included for each l
    //with probability 0.5.
    private static String lProposition() {
        //System.out.println("Construction of the l-proposition");
        //Proposition is initialized.
        a = "(l";
        for (int i = 0; i < nClauses - 1; i++) {
            for (int j = 0; j < nVar - 1; j++) {
                a = a.concat("Al");
            }
            a = a.concat(") OR (l");
        }
        for (int j = 0; j < nVar - 1; j++) {
            a = a.concat("Al");
        }
        a = a.concat(")");
        if (print) {
            System.out.println("l-prop without NOT = " + a);
        }
        negations();
        if (print) {
            System.out.println("l-prop with NOTs   = " + a);
        }
        return a;
    }

    //Each l of the l-proposition is replaced by one of
    //the variables. All variables appear in each clause.
    //Variables are not letters;
    //they are numbers from 0 to nVar-1;
    private static String proposition(String a) {
        //System.out.println();
        //System.out.println("l-prop --> proposition");
        int u = 0;
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                Integer vari = u;
                String s = vari.toString();
                a = a.substring(0, j) + s
                        + a.substring(j + 1);
                u = u + 1;
                if (u == nVar) {
```

```
                    u = 0;
                }
            }
        }
        if (print) {
            System.out.println("prop                = " + a);
        }
        return a;
    }


//The proposition is evaluated
    private static String evaluation(BitSet s) {
        b = c;
        if (print)
        {
            System.out.println();
            System.out.println("Truth evaluation begins");
            System.out.println("prop = " + b);
        }
        int u = 0;
        for (int j = 0; j < b.length(); j++) {
            if (b.charAt(j) == 'l') {
                if (s.get(u) == true) {
                    b = b.substring(0, j) + "T"
                            + b.substring(j + 1);
                } else {
                    b = b.substring(0, j) + "F"
                            + b.substring(j + 1);
                }
                u = u + 1;
                if (u == nVar) {
                    u = 0;
                }
            }

        }
        if (print)
            System.out.println("Truth value8 = " + b);

        return b;
    }


    /*
    int u = 0;
```

```
        for (int j = 0; j < b.length(); j++)
        {
            if (b.charAt(j) == 'l')
            {
                if (TRUTH[VAR[u]] == true)
                {
                    b = b.substring(0, j) + "T" + b.substring(j + 1);
                } else {
                    b = b.substring(0, j) + "F" + b.substring(j + 1);
                }
                u = u + 1;
                //Restart counting for the next clause
                u = u% nVar;
            }
        }
    */


    //The proposition is evaluated until the end
    //Not operators are evaluated at the
    //very beginning for they apply over the next
    //to the right variable.
    private static String simplification(String b) {
        while (b.length() > 1) {
            b = b.replace("NT", "F");
            b = b.replace("NF", "T");
            if (print) {
                System.out.println("Truth value1 = " + b);
            }
            b = b.replace("TAT", "T");
            b = b.replace("TAF", "F");
            b = b.replace("FAT", "F");
            b = b.replace("FAF", "F");

            if (print) {
                System.out.println("Truth value2 = " + b);
            }

            b = b.replace("(T)", "T");
            b = b.replace("(F)", "F");

            if (print) {
                System.out.println("Truth value3 = " + b);
            }

            b = b.replace("T OR T", "T");
```

```
            b = b.replace("T OR F", "T");
            b = b.replace("F OR T", "T");
            b = b.replace("F OR F", "F");

            if (print) {
                System.out.println("Truth value4 = " + b);
            }
        }
        return b;
    }

//Converts a BitSet value into a String
    static private String decode(BitSet t) {
        String s = "";
        for (int i = 0; i < nVar; i++) {
            if (t.get(i) == true) {
                s = s + "1";
            } else {
                s = s + "0";
            }
        }
        return s;
    }

    //Ind1 is a population of binary strings.
    //The BitSet class is used to instantiate
    //binary strings as individuals
    private static void initializePop() {
        BitSet s = new BitSet();
        //Ind1 is false everywhere
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) s.clone();
        }
        //Reinitialization with random truth values.
        for (int i = 0; i < numbInd; i++) {
            for (int j = 0; j < nVar; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
            //System.out.println("Ind1 : " + decode(IND1[i]));
        }
    }

// Does an individual   satisfy  the TARGET?
```

```
    static private int assessInd(BitSet s) {
        String bb = evaluation(s);
        bb = simplification(bb);
        int fitness = 0;
        if (bb.equals("T")) {
            fitness = 1;
        }
        /*
  System.out.print( "s      = " + decode(s) + "fitness = "
                                + fitness);
  System.out.println();
         */
        return fitness;

    }

// Individuals are sorted by fitness.
// the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        int FitnessIndc[];
        FitnessInd = new int[MAXNUMBIND];
        FitnessIndc = new int[MAXNUMBIND];
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < numbInd; i++) {
            Champ = 0;
            for (int j = 0; j < numbInd; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array OrderInd orders individuals by  fitness:
            //The fittest is number zero
            ORDERIND[i] = Champ;
```

```
            FitnessIndc[Champ] = 0;
        }
        // if Pattern has been guessed
        if (FitnessInd[ORDERIND[0]] == 1) {
            problem = false;
            //System.out.println("Champ = " + IND1[ORDERIND[0]]);
            b = c;
        }
        return problem;
    }

// The new generation is made
    private static void newGenerationIND() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest
        while (i < numbInd) {
            int counter = numbCopies;
            while ((counter > 0) & (i < numbInd)) {
                //System.out.println(i + "" + l);
                IND2[i]
                    = (BitSet) IND1[ORDERIND[k]].clone();
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int j = 0; j < numbInd; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*  System.out.println( j+ " Ind1 "
                                + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(numbInd);
        //b is false everywhere
        BitSet bb = new BitSet();
        //B is the recombinant of l and k
        for (int j = 0; j < nVar; j++) {
```

```
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    bb.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
                    bb.set(j);
                }
            }
        }
        return bb;
    }

// The definitive population  is a recombinant one.
    private static void recombinatioNIND() {
        //Ind2 is a copy of Ind1
        for (int i = 0; i < numbInd; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //Ind2 is a recombinant version of Ind1
        for (int l = 0; l < numbInd; l++) {
            IND2[l] = recInd(l);
        }
        //Ind2 is transferred to Ind1
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }

// Individual  a is mutated at a random place.
    private static BitSet pointMutatioNIND(BitSet a) {
        int l = r.nextInt(nVar);
        a.flip(l);
        return a;
    }

// Individuals are mutated
    private static void mutatioNIND() {
        for (int i = 0; i < numbInd; i++) {
            double p = r.nextDouble();
            if (p < mutationRate) {
                IND1[i] = pointMutatioNIND(IND1[i]);
```

```
            }
        }
    }

    // Given a goal, it is guessed by evolution
    private static void evolution() {

        //The new generation is produced
        //as copies of the fittest
        newGenerationIND();
        if (recombination) {
            recombinatioNIND();
        }
        //The new population is subjected to mutation
        mutatioNIND();
        gen++;
        problem = sortingInd();
    }

    public static void report1() {
        System.out.println("\nnClauses = " + nClauses);
        System.out.println("nVar = " + nVar);
        System.out.print("alpha = " + alpha);
        System.out.println();
    }

    /* We generate at random nProps satisfiable props,
   * and report the mean and deviation of the
   * necessary number of individuals to get
   * a satisfying assignment.
     */
    public static void main(String[] args) {

        setParameters();
        for (int i = 0; i < nProps; i++) {
            System.out.println("Prop numb = " + i);
            gen = 0;
            problem = true;
            print = false; //printing instruction
            b = lProposition();
            c = b;
            initializePop();
            while ((problem)) {
                evolution();
            }
            NGEN[i] = gen;
```

```
            NIND[i] = numbInd * gen;
        }
        double sumGen = 0;
        double sumInd = 0;
        for (int i = 0; i < nProps; i++) {
            sumGen = sumGen + NGEN[i];
            sumInd = sumInd + NIND[i];
        }
        double meanGEN = sumGen / nProps;
        double meanIND = sumInd / nProps;
        double sumDGen = 0;
        double sumDInd = 0;
        for (int i = 0; i < nProps; i++) {
            sumDGen = sumDGen
                    + Math.pow((NGEN[i] - meanGEN), 2);
            sumDInd = sumDInd
                    + Math.pow((NIND[i] - meanIND), 2);
        }

        double stanDevGen = Math.sqrt(sumDGen / nProps);
        double stanDevInd = Math.sqrt(sumDInd / nProps);
        report1();
        System.out.println("Repetitions = " + nProps);
        System.out.println("MeanGEN = " + meanGEN);
        System.out.println("stanDevGen = " + stanDevGen);
        System.out.println("MeanIND = " + meanIND);
        System.out.println("stanDevInd = " + stanDevInd);
        double l = Math.log(meanIND);
        System.out.println("Log mean numb Ind = " + l);
    }
}//Program C88, HowManyEvol
```

89 ***Exercise*** *Run the program and play with the code. Use this program and the code of program 87* `DisjuncForm` *to decide whether or not evolution does it better than randomness when faced with the task of solving SAT. Answer*

90 ***Is really evolution worst than randomness?***

We have found two instances in which the performance of evolution is worse than that of randomness when faced with a given problem: a semantics that produces ever larger periods of ecstasy and SAT. Both cases are related with discontinuous semantics, in which significant levels of information must be gathered while selection is dictated by an all or nothing fitness function.

To correctly analyze this result, one must recall that there are infinitely many ways of programming evolution, many of which are useless to solve complex problems, so when one says that evolution does it worst than randomness, we are generally cheating. The most one can claim is that a given simulation of evolution has done it worst than randomness. In other

words, evolution with difficulty would be challenged by a given problem. But the developer is. It is a good idea to keep this in mind.

Let us see now how parallelism can help us.

91 ***Exercise*** *Parallelism exists in computing science because it attacks problems at the same time at different places, so it accelerates the work. Parallelism is naturally included in evolution through the different members of the population. Please, enlarge the population in the previous simulation to see if the exponential dependence of the size of required resources is somehow diminished.* Answer

92 ***Challenge*** *As the previous exercise shows, parallelism, designed to be for help in difficult time, is converted by our simulation of evolution into a fiasco. Is this result a property of our simulation or of evolution in general? Please, study our methods to fabricate the new generation from the extant one, and try various reforms to see what happens. For instance, you might include some of the worst fitted as fathers for the next generation or you could divide the whole population in small demes or subpopulations, augmenting in that way the parallelism without imposing the stagnation caused by large populations.*

93 *Relation with natural evolution*

Very appealing is the fact that there are problems, say, SAT, that are solved by our simulations of evolution only after an exponential number of trials in the given parameter, say, the number of boolean variables. One wishes to extrapolate this difficulty to the realm of natural evolution, but it is desirable to be prudent because the verified computing capabilities of natural evolution are admirable: just consider our incapability to deal with the evolutionary based resistance of microbes to drugs even under our best brilliant efforts (Michel et al., [50] 2008).

Nevertheless, we found at the introduction to this chapter that SAT is at the very root of the essence of life, because it reflects the inherent constrain of enzymatic random networks of avoiding self-clogging, self-digesting, self-poisoning. Nevertheless, the planet is full in too many diverse flavors of auto-catalytic functional networks: each living being together with its milieu is one such system. Henceforth, one feels the temptation to reread the the **theory of evolution** in the next form: natural evolution must be capable of solving SAT very easily, in less than exponential time. This reading of the evolutionary theory might be hasty because another variant must be considered:

We have as yet not excluded that life is not a representative of the whole richness of SAT but just of certain subfamily, whose complexity is sensibly lower than that of the most terrible families of SAT. This option was very clear to Darwin: he was aware of the fact that there are problems that are lethal for evolution, but at the same time, his proposal consisted precisely in that gradual changes may build wonders and that biology is the perfect field to prove this. Eyes are now and forever good objects to illustrate his point of view. Let us analyze them in its essential details to see to which degree is vision an evolvable capability:

- Ordinary light is composed of electromagnetic radiation with a spectrum of very diverse frequencies, with a peak somewhere in the visible range. Ordinary light looks white. The electromagnetic radiation conveys energy.

- When an atom or molecule can interact with light, it usually captures some frequencies and reflects the rest. Reflected light is not white because it lacks the frequencies captured by the molecule, so it looks otherwise: it has acquired a color. We arrive to the conclusion that colored molecules interact with light capturing its energy. Those colored molecules that are useful for biology are called pigments and are special in that the original electronic excitation is transformed into a cis-trans conformational change.

- The conformational change of the pigment is perceived by a molecule, rhodopsin, to which the pigment is attached. This is a very complex protein that is inserted in the membrane and forms a globular structure with a channel in its interior that is also a channel of the membrane. Rhodopsin takes the energy of the pigment and converts it in useful form for the cell. For instance, rhodopsin can suffer a conformational change that opens its interior channel plus a separation of charges that converts the channel into a specific one-directional way for the transit of ions. The specificity of the allowed ion is regulated by the inner residues of the amino acids and by the diameter of the channel. The direction of the allowed flux goes against the gradient vector of the electric field. So, we have a mean to accumulate electric energy thanks to the incoming radiation: we have a photocell. We also have a dissipative Maxwell's demon. It is dissipative because when the correct ion enters the channel, it recombines with the local charge and can traverse the channel as a neutral form that acquires its usual charge at the exit of the channel. So, the channel must be recharged with each new transported ion.

- The potential electric energy accumulated across the membrane can be used for whatever purpose, for instance to synthesize ATP. This happens in the *Halobacterium*, in the chloroplast and in the mithochondria.

- There is an algae, *Nitella syncarpa*, that looks like a small and delicate aquatic tree. It is spectacular in that this tree is segmented with segments some five centimeters long and each one of them is composed by one single cell. The protoplasmatic membrane of each cell is perfectly covered from the inside by a regular tapestry of chloroplasts. In ordinary situations, light is converted by these chloroplasts into chemical fuel. But if instead of ordinary light, one uses a laser beam during some ten minutes, at last the cell responds with an electric spike across the membrane of the cell and that one can register with a system of electrodes. The effect of the spike is also visible through a simple magnifying glass: the cytoplasm is animated with a periodic motion and when the spike appears, the movement automatically gets frozen to be recovered some few minutes later. We know that the effect of the laser is through the photo-receptors of the chloroplast because if one uses an inhibitor for the corresponding photo-transducers, the appearing of the spike is retarded. So, we have that the *Nitella* is an eye for a laser beam.

- In effect, an eye is basically a system that converts incoming light into an electric spike that causes the release of certain chemical signals that can be captured by other nervous cells. All what is needed to pass from an eye for a laser beam to an eye for ordinary light is to augment the sensitivity. One can use various magnifying effects, say, forming a conglomerate of those organelles that serve as transducers: in the eye of man these organelles are discs allocated at the outer segments of the rods.

- Our eyes contain, moreover, an optical system to direct light toward the retina, a curved wall where images are projected. The retina is composed of cellular transducers: when a cell is hit by light, the cell produces a nervous signal.

- The eyes of a cat are more complex: they come armed with reflection dispositives to avail as much light as possible and with other cells whose specific mission is to amplify weakly signals.

- Vision is enabled by a joint work of wonderful eyes plus a system for the processing and interpretation of nervous signals, a task that is done by amazing neuronal circuits. Some stories with persons that fall asleep while they drive their cars indicate that vision, from perceiving visual forms to react to them, is not necessarily mediated by the consciousness. Nevertheless, the synthesis of perception could follow directives in real time given by conscious or unconscious parameters (Rodríguez, [**?**] 2007).

We see here that the very complex function of human vision can be schematically connected with the properties of simple matter through a path consisting of living beings. Can a gradualist path like this be found connecting beings that are moreover related by descent? Experts do not doubt that that is indeed the case (Gehring, [26] 2005), a conclusion that is backed by theorists Nilsson and Pelger ([54] 1994), who say in the abstract of their paper:

Theoretical considerations of eye design allow us to find routes along which the optical structures of eyes may have evolved. If selection constantly favors an increase in the amount of detectable spatial information, a light-sensitive patch will gradually turn into a focused lens eye through continuous small improvements of design. An upper limit for the number of generations required for the complete transformation can be calculated with a minimum of assumptions. Even with a consistently pessimistic approach the time required becomes amazingly short: only a few hundred thousand years.

One sees that gradualism is defined as small change in the genotype that produces small change in fitness. After reading this abstract, one can immediately conclude that for gradualism everything is simple, that anything can be solved by gradualism and that a claim to the contrary would mean just a lack of imagination. As a consequence, our former preoccupation about the partial sampling of SAT by nature would had no interest: evolution under gradualism is strong enough to deal with any problem no matter how complex it might look.

Can we reproduce such victorious good news from our stand? Which characteristics of our stand could defy the conclusions of gradualism?

Our stand is that of genetic programming: the genome is software and evolution is a software developer. The first implication, that is very good for evolution, is that gradualism in not a law in biology. The reason is that changes that involve large portions of meaningful code might occur in one single generation. But on the other hand, a tiny change in the initial conditions or in the code can produce disproportionate results as in an if-then condition: if $(x < 3)$ go to the North Pole else wait during 3 years and then go to the South Pole.

Nevertheless, we can translate gradualism to our field if we reformulate it as follows: it is a way of attacking problems that makes them easy to be solved, i.e. that enables a

rapid convergence of algorithms to the desired solution. A problem arises at once: our definition makes sense for design but not for natural evolution that is in first instance a diffusive process without purpose. Nevertheless, our definition is good for applied evolution: if natural evolution can devise highly functional software in less than a blink, a developer must be able of using applied evolution to design software in real time to be very competitive in the trade. Is this reasonable?

We have already found two points that somehow favor the postulate of gradualism that there is always a point of view that converts a problem, no matter how complex it could be, into a realizable task. They are:

1. Evolution is a style of software design that is very appropriate to deal with very large projects and that can be optimized if one follows the directives of the *biogenesis* methodology.

2. The problem of optimization of genetic programming is a matter of current investigation: we verified in VOL II that a three level hierarchy is good enough to deal with a problem that was for us unsolvable with one and two level hierarchies. So, we converted an impossible problem into a more realizable task. Now, our methodology with hierarchies is elementary because in plain words it says that a house must not be built with atoms but with tiles and that to gain efficiency various types of tiles must be allowed. So, if an elementary strategy can help a lot, what can we expect from sophistication?

Do these evolutionary methodologies convert software design into a task for babies?

No, to the best of our understanding, these and all good methodologies are suitable to convert a terrible problem into one that can be attacked by very smart and finely trained experts. To be sure, the generality of industrial developers do not know that evolution could be used to design software but all of them use an incipient evolutionary style that is tried to be optimized by some schools such as that of *extreme programming*.

In general, we do not doubt that everything is possible for evolution in our field or in any other very complex field if one is provided with no restrictions of time, memory, computing power and intelligence to design the combinatorial basis and hierarchies to work upon. Problems and real life begin if any one of those aforementioned factors suffers some kind of limitation. Which is the effect of those limitations? That we burn time and space and do not arrive to a solution. Why? It is because the event that could solve the problem does not appear of if it appears it disappears. So, we at last see that gradualism looks very simple: **an evolutionary environment for applied evolution is gradualistic** when the events leading to the solution of problems are forced to appear with sufficiently high probability to overcome sampling effects given some tight constrains of time and memory.

How does look gradualism in natural evolution from the perspective of genetic programming?

Our proposal is the same that we use for applied evolution: a tiny step is a change that is highly probably and that therefore defines the front of diffusion. Such a change could be a change of a basis in a DNA string or an inversion of a segment with millions of bases. Or such a change could be, eventually, any one of a multitude of modifications with equivalent final function: each change is not probable but a member of the whole family is. Or such a

change could be anyone among a multitude of different functions whose only in-hindsight-purpose should be to improve reproductive potential of the carrier.

Given such a generality, one must ask: Is there a change that is crucial for further diffusion but with too low a probability of appearing in a process of diffusion across the world of programs? As we have seen, that depends on the relation of the semantics of the fitness function with the size of computing resources: given an evolutionary environment, a discontinuous semantics is one whose events have exceedingly low probability. What do we have in nature? Our point is that some folds of natural evolution as a diffusive process can be depicted by SAT and happens through satisfiable propositions. SAT is per se not gradualistic:

In SAT, a proposition is false else true. The classification is without grays. So, we have an all else nothing fitness function that generates a discontinuous semantics. But in a sufficiently powerful evolutionary environment, a discontinuous semantics offers no problem. Is SAT hard enough to challenge any powerful but finite evolutionary environment? According to our simulations and to experience of the Academic community, the answer is affirmative: the number of trials to solve SAT IN THE WORST CASE is exponential in the number of boolean variables, so our perception is that SAT is a perfect synonym of complexity.

Why do we care about the worst case? What does it mean?

It means that SAT contains many subfamilies some of which can be indeed solved by evolution at once, say, tautologies. For instance $p \vee \sim p$ evaluates to true no matter which truth assignment one might use. Therefore, the very first attempt produces a success. Now, it is not easy to decide the degree of evolutionary importance of easy evolutionary problems. Our interpretation of Darwin seems to imply that easy evolutionary problems are enough to explain the complexity of actual life. But the experience of the whole academic community and of any one person that tries to express himself or herself is that ordinary problems may correspond to the hard cases of SAT.

In operation language, we are reluctant to believe that many ordinary problems that are themselves subproblems of SAT could be solved in less than exponential time. Can our prudence be transformed into a dogma?

94 *Too many ifs*

The scientific community has been unable to find up to date an algorithm, parallel or not, evolutionary or not, that could solve SAT using less than exponential number of trials or of memory allocations. But, on the other hand, a proof that such an algorithm does not exists has not been found although it is popular to think that such an algorithm does not exist. Nevertheless, we cannot assure that we will found the searched algorithm or that we will prove that it does not exist. Why? The reason is that the future contains too many ifs, too many unexplored possibilities. We can gain some insight into what a creative insight could be by exposing two ideas:

1. In our simulations, any valuation to be tested in SAT was as valid and probable as any other. Since they are counted by exponential functions in the number of boolean variables while those valuations that fit SAT are very few in number, we conclude that the probability of finding a satisfying valuation is exceedingly low and so the computer waste huge amounts of time to solve SAT. We can say that SAT is complex for our solving algorithms. We see that the theory of probability is connected with complexity theory. For a rigorous treatment see Chaitin ([14] 1987). So, if one can break the

exponential regime, it is because one caused a bias in the generation mechanism of valuations. Henceforth, we must study the effect of various biases over the complexity of SAT.

2. SAT, genetic programming and similar problems are challenging because of their all or nothing fitness function that prevents gradualism of picking up small change. The remedy is then obvious: we must insert these problems in an artificial gradualistic world, where evolution could get in charge.

We will study two implementations of the second idea. The first one uses a proposal made by Zadeh ([76] 1965): the fuzzy logic. The second deals with an extension of SAT known as MAX-K-SAT, that will appear below. The first implementation goes as follows:

Instead of considered that a proposition is true else false, we must concede some degree of truth to each proposition. The degree of truth is quantified by a number in the interval [0,1], zero for a perfect false, like a contradiction, true for a perfect truth, say a tautology, but 0.5 for a proposition that more or less fits given requirements. The AND, OR, and NOT operators are defined in this **fuzzy logic** as follows (Wikipedia, [74] 2008):

NOT x = (1 - truth(x))
x AND y = minimum(truth(x), truth(y))
x OR y = maximum(truth(x), truth(y))

95 ***The code*** *The next program implements an evolutionary round of the idea of fuzzy logic to solve SAT.*

```
/* Program C95, Zadeh.
 * We solve SAT by means of evolution
 * with the help of fuzzy logic.
 *
 * We take disjunctive normal forms,
 * which are always satisfiable, and inquire
 * over the number of  individuals
 * that are necessary
 * to get a satisfying assignment.
 *
 * An assignment consists in the assignation to each
 * variable of a real number in the interval [0,1],
 * which is interpreted as its degree of truth.
 * Logical operations are calculated according to the rules
 * given by Zadeh (1965):
 * True = 1; false = 0;
 * NOT x = (1 - truth(x))
 * x AND y = minimum(truth(x), truth(y))
 * x OR y = maximum(truth(x), truth(y))
 *
 * The fitness function rewards the degree of truth,
```

```
 * so for a satisfiable proposition,
 * a fully satisfying truth assignment is expected to be
 * recovered. The problem is whether or not this is done in
 * less than exponential time.
 */
package ejvol3v2p;

import java.util.Random;

public class Zadeh {

    // Turn on  the random generator
    static Random r = new Random();

    //Number of clauses
    private static int nClauses;
    //Number of VARiables, could be as large as MAXV.
    //Variables will be represented by numbers.
    private static int nVar;
    private static final int MAXV = 300;
    //Proposition in formation
    private static String a, b, c;

    private static int nProps = 1000;
    //NEGations in the prop are recorded
    private static final boolean NEG[] = new boolean[1000];
    //The VARiables assignment is recorded in VAR[]
    private static final int VAR[] = new int[MAXV * MAXV];
    private static boolean print = false;
    private static double alpha, nn;
    private static boolean problem = true;

    //A population of binary numbers is considered.
    //The number of individuals is constant
    private static int numbInd;
    private static final int MAXNUMBIND = 10000;
    //Truth value of valuation i variable j
    private static final double IND1[][]
            = new double[MAXNUMBIND][MAXV];
    private static final double IND2[][]
            = new double[MAXNUMBIND][MAXV];
    private static double FitnessInd[]
            = new double[MAXNUMBIND];
    private static final int ORDERIND[] = new int[MAXNUMBIND];
    private static int gen;
    private static boolean recombination;
```

```
    private static double mutationRate;
    private static final int MAXNREP = 100;
    //Memory for statistics
    private static final long NGEN[] = new long[MAXNREP];
    private static final long NIND[] = new long[MAXNREP];

    private static void setParameters() {
        //Number of clauses
        nClauses = 5;
        //Number of VARiables
        nVar = 2;
        //Number of propositions
        nProps = 10;
        numbInd = 10;
        recombination = true; //true = recombination is enabled
        mutationRate = 0.8;
        alpha = nClauses;
        nn = nVar;
        alpha = nClauses / nn;
    }

    //NOT operators are inserted with prob 0.5
    private static String negations() {
        int negCounter = 0;
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                int d = r.nextInt(2);
                if (d == 1) {
                    a = a.substring(0, j) + "f" + a.substring(j + 1);
                    NEG[negCounter] = true;
                } else {
                    NEG[negCounter] = false;
                }
                negCounter++;
            }
        }
        a = a.replace("f", "Nl");
        return a;
    }

    //Composes a random normal disjunctive full l-proposition
    // with nClauses clauses. The NOT operator is included for each l
    //with probability 0.5.
    private static String lProposition() {
        //System.out.println("Construction of the l-proposition");
```

```
        //Proposition is initialized.
        a = "(l";
        for (int i = 0; i < nClauses - 1; i++) {
            for (int j = 0; j < nVar - 1; j++) {
                a = a.concat("Al");
            }
            a = a.concat(") OR (l");
        }
        for (int j = 0; j < nVar - 1; j++) {
            a = a.concat("Al");
        }
        a = a.concat(")");
        if (print) {
            System.out.println("l-prop without NOT = " + a);
        }
        negations();
        if (print) {
            System.out.println("l-prop with NOTs   = " + a);
        }
        return a;
    }

    private static void report2(int i) {
        System.out.println();
        System.out.println("prop = " + b);
        System.out.println("Truth evaluation begins:");
        System.out.println("This is IND1[i]:");
        for (int j = 0; j < nVar; j++) {
            System.out.print(IND1[i][j] + " ");
        }
        System.out.println();
        System.out.println("NEGations in prop:");
// NEGations are reported
        for (int u = 0; u < nClauses * nVar; u++) {
            if (NEG[u] == true) {
                System.out.print("N");
            } else {
                System.out.print("-");
            }
        }
        System.out.println();

    }

    //The proposition is evaluated.
    //Becuase it is in normal disjuntive format
```

```
    //all clauses are the same, parallel ands,
    //except for  the negative operators.
    //Truth values are in IND1.
    private static double assessInd(int i) {
        b = c;
        if (print) {
            report2(i);
        }
        if (print) {
            System.out.println("This is the valuation:");
        }
        int negCounter = 0;
        //The normal disjunctive form is evaluated
        double truth = 0;
        double aa;
        //Evaluate every clause
        for (int k = 0; k < nClauses; k++)
        {
            double truth1 = 1;

            //Evaluate all variables, one by one
            for (int j = 0; j < nVar; j++) {
                aa = IND1[i][j];
                //Take care of negations
                if (NEG[negCounter] == true) {
                    aa = 1 - aa;
                }
                if (print) {
                    System.out.print(aa + " : ");
                }
                //Evaluate AND for variables
                truth1 = Math.min(truth1, aa);
                negCounter = negCounter + 1;
            }

            if (print) {
                System.out.print(" OR  : ");
            }
            //Evaluate OR for clauses
            truth = Math.max(truth, truth1);
        }
        if (print) {
            System.out.println();
            System.out.println("Truth value of prop = " + truth);
        }
        return truth;
```

```
    }

    private static void initializePop() {
        //Reinitialization with random truth values.
        for (int i = 0; i < numbInd; i++) {
            for (int j = 0; j < nVar; j++) {
                double l = r.nextDouble();
                IND1[i][j] = l;
            }
        }
    }

    // Individuals are sorted by fitness.
    // the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        double FitnessIndc[];
        FitnessInd = new double[MAXNUMBIND];
        FitnessIndc = new double[MAXNUMBIND];
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = assessInd(i);
            FitnessIndc[i] = FitnessInd[i];
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < numbInd; i++) {
            Champ = 0;
            for (int j = 0; j < numbInd; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array ORDERIND orders individuals by  fitness:
            //The fittest is number zero
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        // if SAT has been solved
        if (FitnessInd[ORDERIND[0]] == 1) {
```

```
            problem = false;
            //System.out.println("Champ = " + IND1[ORDERIND[0]]);
            b = c;
        }
        return problem;
    }

// The new generation is made
    private static void newGenerationIND() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest
        while (i < numbInd) {
            int counter = numbCopies;
            while ((counter > 0) & (i < numbInd)) {
                //System.out.println(i + "" + l);
                //IND1[ORDERIND[k]][j] is copied into IND2[i][j]
                System.arraycopy(IND1[ORDERIND[k]], 0, IND2[i], 0, nVar);

                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int l = 0; l < numbInd; l++) {
            //IND2 is copied into IND1
            System.arraycopy(IND2[l], 0, IND1[l], 0, nVar);
        }
    }

// Recombination is enabled
    private static void recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(numbInd);

        //IND2[l] is the recombinant of IND1[l] and IND1[k]
        for (int j = 0; j < nVar; j++) {
            int m = r.nextInt(2);
            if (m == 1) //take the truth value from  l
            {
                IND2[l][j] = IND1[l][j];
            }
            if (m == 0)//take the truth from k
```

```
            {
                IND2[l][j] = IND1[k][j];
            }
        }

    }

// The definitive population  is a recombinant one.
    private static void recombinatioNIND() {
        //IND2 is a copy of IND1
        for (int l = 0; l < numbInd; l++) {
            //IND1 is copied into IND2
            System.arraycopy(IND1[l], 0, IND2[l], 0, nVar);
        }
        //IND2 is a recombinant version of IND1
        for (int l = 0; l < numbInd; l++) {
            recInd(l);
        }
        //IND2 is transferred to IND1
        for (int l = 0; l < numbInd; l++) {
            System.arraycopy(IND2[l], 0, IND1[l], 0, nVar);
        }
    }

    // Individual  i is mutated at a random place.
    private static void pointMutatioNIND(int i) {
        int l = r.nextInt(nVar);
        int k = r.nextInt(3);
        if (k == 0) {
            IND1[i][l] = r.nextDouble();
        }
        if (k == 1) {
            IND1[i][l] = Math.min(IND1[i][l] + 0.1, 1);
        }
        if (k == 2) {
            IND1[i][l] = Math.max(IND1[i][l] - 0.1, 0);
        }
    }

    // Individuals are mutated
    private static void mutatioNIND() {
        for (int i = 0; i < numbInd; i++) {
            double p = r.nextDouble();
            if (p < mutationRate) {
                pointMutatioNIND(i);
            }
```

```
        }
    }

    // Given a goal, it is guessed by evolution
    private static void evolution() {
        //The new generation is produced
        //as copies of the fittest
        newGenerationIND();
        if (recombination) {
            recombinatioNIND();
        }
        //The new population is subjected to mutation
        mutatioNIND();
        gen++;
        problem = sortingInd();
    }

    public static void report1() {
        System.out.println();
        System.out.println("Final report:");
        System.out.println("nClauses = " + nClauses);
        System.out.println("nVar = " + nVar);
        System.out.println("alpha = " + alpha);

    }

    /* We generate at random nProps satisfiable props,
   * and report the mean and deviation of the
   * necessary number of individuals to get
   * a satisfying assignment.
     */
    public static void main(String[] args) {
        setParameters();
        for (int i = 0; i < nProps; i++) {
            System.out.println("Prop numb = " + i);
            problem = true;
            print = true; //printing instruction
            problem = true;
            b = lProposition();
            c = b;
            initializePop();
            while ((problem)) {
                evolution();
            }
            NGEN[i] = gen;
            NIND[i] = numbInd * gen;
```

```
        }
        double sumGen = 0;
        double sumInd = 0;
        for (int i = 0; i < nProps; i++) {
            sumGen = sumGen + NGEN[i];
            sumInd = sumInd + NIND[i];
        }
        double meanGen = sumGen / nProps;
        double meanInd = sumInd / nProps;
        double sumDGen = 0;
        double sumDInd = 0;
        for (int i = 0; i < nProps; i++) {
            sumDGen = sumDGen + Math.pow((NGEN[i] - meanGen), 2);
            sumDInd = sumDInd + Math.pow((NIND[i] - meanInd), 2);
        }
        double stanDevGen = Math.sqrt(sumDGen / nProps);
        double stanDevInd = Math.sqrt(sumDInd / nProps);
        report1();
        System.out.println("Repetitions = " + nProps);
        System.out.println("MeanGen = " + meanGen);
        System.out.println("stanDevGen = " + stanDevGen);
        System.out.println("MeanInd = " + meanInd);
        System.out.println("stanDevInd = " + stanDevInd);
        double l = Math.log(meanInd);
        System.out.println("Log mean numb Trials = " + l);
    }
}//End of Program C95 Zadeh.
```

96 ***Exercise*** *Run and play with the code. How does the present implementation of fuzzy logic behave? Does it solve SAT in less than exponential time?* Answer

97 ***Challenge*** *Try out your own modifications to the previous code to see the effect over the exponential regime.*

5.5 The arising of functions

SAT has been introduced in this chapter as an abstraction of the intrinsic problem that every auto-catalytic system has of avoiding self clogging and self digestion. But SAT can be given thousands of other interpretations. Let us see another one: SAT allows us to study the arising of functions in evolutionary biology.

In strict sense, the notion of function is alien to evolutionary biology, because function implies a purpose, a project. Nevertheless, following the standard habit, we also have allowed ourselves to speak of function, a concept than seems to be unavoidable: what is the purpose or function of a nuclease? It is to cleave a DNA string into its constituent bases. What is the fundamental purpose of limbs? To allow movement.

Let us make an effort to correctly accommodate the concept of function in the realm of evolutionary biology, in which everything is thought to be the result of accident and selection, which is itself an emergent property of a given ecosystem.

In first instance, we can see that, say, a protease has as function the cleavage of proteins only if we have a well prepared milieu in which the protease and the proteins can meet together and have the appropriate conditions to interact. The easiest way to imagine that is in the cell. So, a cell is one background which allows us to accept in hindsight the notion of function. The discussed above model of Kauffman about life as an auto-catalytic system that functions far from equilibrium is also good for that purpose. We see that we cannot define life in abstracto but only as a property of an ecosystem.

98 *Properties and functions*

Let us contrast now the next two concepts: **property** and **function**. A property is a measure of the reaction of a system to a given stimulus. For instance, mass is the constant of proportionality that relates force and acceleration of a given body. The stimulus is a force, the reaction is acceleration. The reaction is proportional to the stimulus and mass is the constant of proportionality. Thus, the property of interest is the mass.

The concept of property has meaning with and without purposes, life or consciousness. On the other hand, a function in evolutionary biology is a property that helps the sustaining of an auto-catalytic system and very especially of those systems that can reproduce. In short, function is anything that can sustain life and evolution. Thus life and evolution are the semantic organizing actors of the discourse about functions and properties.

In regard with complexity, functions and properties are very different one than the other: at the beginning of an evolutionary process, anything can be used for thousands of functions, but when sophistication is increased and the concept of function is allowed, each thing can be used for just one or some few purposes. In fact, a property that constitutes an arising function can be described by a short satisfiable proposition, one with some few variables and connectives. It can always be extended to other longer satisfiable propositions. Such a high degree of freedom implies that the word of properties is unbounded. But if we impose the burden of fitting a function, then we know by intuition that this can be achieved in very restricted ways. Specifically:

The world of functions is the world of restrictions that can be represented by a clause with just AND connectives: do this and that and not that other and not that of that other... The programs that we have developed can be readily reused for the study of this setting if we pose the number of clauses equal to one in a disjunctive normal form. So, the game is over: ensuing SAT problems need 2^n random trials in average to be satisfied, where n is the number of variables.

In real situations, we do not pursue pristine functions but implementations that include the asked function plus a possible complement that cause tolerable interference. In this regard, let us get acquainted with conjunctive normal forms to see their expressive power and complexity.

99 *Conjunctive normal forms*

As we know, disjunctive normal forms are very easy to understand and allow us a very transparent experiment. We can get another setting if we negate a normal disjunctive form. Negations can be calculated thanks to the **laws of Morgan**:

$\sim (p \wedge q) = \sim p \vee \sim q$
$\sim (p \vee q) = \sim p \wedge \sim q$
So, the proposition in normal disjunctive form
$(p \wedge q \wedge r) \vee (p \wedge \sim q \wedge r)$
is equal to the negation of its negation:
$\sim [\sim [(p \wedge q \wedge r) \vee (p \wedge \sim q \wedge r)]]$
and it is equal, by the laws of Morgan, to
$\sim [(\sim p \vee \sim q \vee \sim r) \wedge (\sim p \vee q \vee \sim r)]$
We see here an expression which is the conjunction of various clauses, each of which is an OR evaluation of fundamental variables. An expression like

$$(\sim p \vee \sim q \vee \sim r) \wedge (\sim p \vee q \vee \sim r)$$

is called a **conjunctive normal form**. Conjunctive normal forms have satisfiable exponents, such as $p \wedge q$ which is true for the TT evaluation. And also have not satisfiable members such as $p \wedge \sim p$, which is a contradiction. The overall structure of these forms, the chain of ANDS, represents restrictions: do this and that. But the clauses contain diverse options. This means that these forms represent as restrictions as complementary optional properties. Let us pass now to making experiments.

If we define, as before, α as the ratio of clauses to variables, then we have a very transparent experiment:

When α is low, with less clauses than variables, one is making a discourse of unrelated things and so one hardly can fall into a contradiction. Thus, for low values of α most propositions are satisfiable. But, when we begin to tie the ends, then we use more clauses on the same number of variables, and the probability of falling into contradiction grows. Hence, for high values of α we shall expect that most random propositions in conjunctive normal form are unsatisfiable, i.e. that almost all truth assignments end in a final false evaluation.

100 ***The code*** *The next code fits the study of conjunctive normal full forms. We use the code to verify else reject the next expectation: α, the ratio of clauses to variables, and satisfiability are inversely related one to another for conjunctive normal forms.*

```
/* Program C100, ConjForm.

 * We take conjunctive normal full forms,
 * and inquire over the relation of
 * alpha, the ratio of clauses to variables,
 * and satisfiability.
 * full forms: all clauses have all variables.
 *
 * The prob of negation of a variable is 0.5.
 * Random   equiprobable truth values are assigned
 * to the  variables and
 * the prop is evaluated to true else false.
 *
```

```
 * A proposition is declared to be unsatisfiable if
 * after a certain number of generations no
 * satisfiable assignment has  been found.
 *
 */
package ejvol3v2p;

import java.util.BitSet;
import java.util.Random;

public class ConjForm {

    // Turn on  the random generator
    static Random r = new Random();

    //Number of clauses
    private static int nClauses;
    //Number of variables, could be as large as MAXV.
    //Variables will be represented by numbers.
    private static int nVar;

    //Proposition in formation
    private static String a, b, c;

    private static int nProps = 1000;

    //The variables assignment is recorded in VAR[]
    private static final boolean PRINT = false;
    private static double alpha, nn;
    private static boolean problem = true;
    //if the prop is satisfiable, sat = true else false.
    private static boolean sat = false;

    //A population of binary numbers is considered.
    //The number of individuals is constant
    private static int numbInd;
    private static final int MAXNUMBIND = 10000;
    private static final BitSet IND1[] = new BitSet[MAXNUMBIND];
    private static final BitSet IND2[] = new BitSet[MAXNUMBIND];
    private static int FitnessInd[]
            = new int[MAXNUMBIND];
    private static final int ORDERIND[] = new int[MAXNUMBIND];
    private static int gen;
    private static int maxGen;
    private static boolean recombination;
```

```
    private static double mutationRate;

    private static void setParameters() {
        //Number of clauses
        nClauses = 7;
        //Number of variables
        nVar = 2;
        //Number of propositions
        nProps = 10;
        numbInd = 16;
        recombination = true; //true = recombination is enabled
        mutationRate = 0.8;
        alpha = nClauses;
        nn = nVar;
        alpha = nClauses / nn;
        maxGen = (int) Math.ceil(Math.pow(2, nVar));
        System.out.println("maxGen = " + maxGen);
    }


    //NOT operators are inserted with prob 0.5
    private static String negations() {
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                int d = r.nextInt(2);
                if (d == 0) {
                    a = a.substring(0, j) + "f" + a.substring(j + 1);
                }
            }
        }
        a = a.replace("f", "Nl");
        return a;
    }


    //Composes a random normal disjunctive full l-proposition
    // with nClauses clauses. The NOT operator is included
    //for each l with probability 0.5.
    private static String lProposition() {
        //System.out.println("Construction of the l-proposition");
        //Proposition is initialized.
        a = "(l";
        for (int i = 0; i < nClauses - 1; i++) {
            for (int j = 0; j < nVar - 1; j++) {
                a = a.concat("ORl");
            }
```

```
            a = a.concat(") A (l");
        }
        for (int j = 0; j < nVar - 1; j++) {
            a = a.concat("ORl");
        }
        a = a.concat(")");
        if (PRINT) {
            System.out.println("l-prop without NOT = " + a);
        }
        negations();
        if (PRINT) {
            System.out.println("l-prop with NOTs   = " + a);
        }
        return a;
    }

    //Each l of the l-proposition is replaced by one of
    //the variables. All variables appear in each clause.
    //Variables are not letters;
    //they are numbers from 0 to nVar-1;
    private static String proposition(String a) {
        //System.out.println();
        //System.out.println("l-prop --> proposition");
        int u = 0;
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                Integer vari = u;
                String s = vari.toString();
                a = a.substring(0, j) + s + a.substring(j + 1);
                u = u + 1;
                if (u == nVar) {
                    u = 0;
                }
            }
        }
        if (PRINT) {
            System.out.println("prop               = " + a);
        }
        return a;
    }

//The proposition is evaluated
    private static String evaluation(BitSet s) {
        b = c;
        if (PRINT) {
            System.out.println();
```

```
            System.out.println("Truth evaluation begins");
            System.out.println("prop = " + b);
            System.out.println("Ind : " + decode(s));
        }
        int u = 0;
        for (int j = 0; j < b.length(); j++) {
            if (b.charAt(j) == 'l') {
                if (s.get(u) == true) {
                    b = b.substring(0, j) + "T" + b.substring(j + 1);
                } else {
                    b = b.substring(0, j) + "F" + b.substring(j + 1);
                }
                u = u + 1;
                if (u == nVar) {
                    u = 0;
                }
            }

        }
        if (PRINT) {
            System.out.println("Truth value8 = " + b);
        }
        return b;
    }


//The proposition is evaluated until the end
    //Not operators are evaluated at the
    //very beginning for they apply over the next
    //to the right variable.
    private static String simplification(String b) {
        while (b.length() > 1) {
            b = b.replace("NT", "F");
            b = b.replace("NF", "T");
            if (PRINT) {
                System.out.println("Truth value1 = " + b);
            }
            b = b.replace("TORT", "T");
            b = b.replace("TORF", "T");
            b = b.replace("FORT", "T");
            b = b.replace("FORF", "F");

            if (PRINT) {
                System.out.println("Truth value2 = " + b);
            }
```

```
            b = b.replace("(T)", "T");
            b = b.replace("(F)", "F");

            if (PRINT) {
                System.out.println("Truth value3 = " + b);
            }
            b = b.replace("T A T", "T");
            b = b.replace("T A F", "F");
            b = b.replace("F A T", "F");
            b = b.replace("F A F", "F");

            if (PRINT) {
                System.out.println("Truth value4 = " + b);
            }

        }
        return b;
    }

//Converts a BitSet value into a String
    static private String decode(BitSet t) {
        String s = "";
        for (int i = 0; i < nVar; i++) {
            if (t.get(i) == true) {
                s = s + "1";
            } else {
                s = s + "0";
            }
        }
        return s;
    }

    //IND1 is a population of binary strings.
    //The BitSet class is used to instantiate
    //binary strings as individuals
    private static void initializePop() {
        BitSet s = new BitSet();
        //IND1 is false everywhere
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) s.clone();
        }
        if (PRINT)
                System.out.println("Initial population "
                        + "of truth assignments");
        //Reinitialization with random truth values.
        for (int i = 0; i < numbInd; i++) {
```

```
            for (int j = 0; j < nVar; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
            if (PRINT)
                System.out.println("IND1 : "
                        + decode(IND1[i]));
        }
    }

    //Does an individual   satisfy  the target?
    static private int assessInd(BitSet s) {
        String bb = evaluation(s);
        bb = simplification(bb);
        int fitness = 0;
        if (bb.equals("T")) {
            fitness = 1;
        }
        /*
  System.out.print( "s      = " + decode(s) + "fitness = "
                                 + fitness);
  System.out.println();
         */
        return fitness;

    }

// Individuals are sorted by fitness.
// the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        int FitnessIndc[];
        FitnessInd = new int[MAXNUMBIND];
        FitnessIndc = new int[MAXNUMBIND];
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
```

```
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < numbInd; i++) {
            Champ = 0;
            for (int j = 0; j < numbInd; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array ORDERIND orders individuals by  fitness:
            //The fittest is number zero
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        // if Pattern has been guessed
        if (FitnessInd[ORDERIND[0]] == 1) {
            problem = false;
            sat = true;
            //System.out.println("Champ = " + IND1[ORDERIND[0]]);
            b = c;
        }
        return problem;
    }

// The new generation is made
    private static void newGenerationInd() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest
        while (i < numbInd) {
            int counter = numbCopies;
            while ((counter > 0) & (i < numbInd)) {
                //System.out.println(i + "" + l);
                IND2[i] = (BitSet) IND1[ORDERIND[k]].clone();
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int j = 0; j < numbInd; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*  System.out.println( j+ " IND1 "
```

```
                                    + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(numbInd);
        //b is false everywhere
        BitSet bb = new BitSet();
        //B is the recombinant of l and k
        for (int j = 0; j < nVar; j++) {
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    bb.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
                    bb.set(j);
                }
            }
        }
        return bb;
    }

// The definitive population  is a recombinant one.
    private static void recombinationInd() {
        //IND2 is a copy of IND1
        for (int i = 0; i < numbInd; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //IND2 is a recombinant version of IND1
        for (int l = 0; l < numbInd; l++) {
            IND2[l] = recInd(l);
        }
        //IND2 is transferred to IND1
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }
```

```
// Individual  a is mutated at a random place.
    private static BitSet pointMutationInd(BitSet a) {
        int l = r.nextInt(nVar);
        a.flip(l);
        return a;
    }

// Individuals are mutated
    private static void mutationInd() {
        for (int i = 0; i < numbInd; i++) {
            double p = r.nextDouble();
            if (p < mutationRate) {
                IND1[i] = pointMutationInd(IND1[i]);
            }
        }
    }

    // Given a goal, it is guessed by evolution
    private static void evolution() {

        //The new generation is produced
        //as copies of the fittest
        newGenerationInd();
        if (recombination) {
            recombinationInd();
        }
        //The new population is subjected to mutation
        mutationInd();
        gen++;
        problem = sortingInd();
        System.out.println("Gen = " + gen);
        if (gen == maxGen) {
            problem = false;
            sat = false;
        }
    }

    public static void report1() {
        System.out.println();
        System.out.println("nClauses = " + nClauses);
        System.out.println("nVar = " + nVar);
        System.out.println("alpha = " + alpha);
        System.out.println("Repetitions = " + nProps);
        System.out.println();
    }
```

```
    /* We generate at random nProps satisfiable props,
    * and report the mean and deviation of the
    * necessary number of individuals to get
    * a satisfying assignment.
    */
    public static void main(String[] args) {

        setParameters();
        double nSat = 0;
        for (int i = 0; i < nProps; i++) {
            System.out.println("**********");
            System.out.println("Prop numb = " + i);
            problem = true;

            problem = true;
            sat = false;
            b = lProposition();
            c = b;
            proposition(b);
            initializePop();
            gen = 0;
            while ((problem)) {
                evolution();
            }
            if (sat == true) {
                nSat = nSat + 1;
            }
        }
        report1();
        double p = nSat / nProps;
        System.out.println("Prob of Sat = " + p);

    }
}//End of Program C100, ConjForm.
```

101 ***Exercise*** *Run the code and verify else refute the expected relation between* α *and satisfiability.* Answer

102 ***The MAX-K-SAT gradualistic attack over SAT***

We have defined in the all-else-nothing semantics the fitness of a valuation as one if the valuation satisfies the proposition and zero otherwise. This discontinuous semantics can be weakened for conjunctive forms to get a gradualistic version as follows: we redefine the fitness of a valuation as the number of clauses in the proposition that are satisfied by the valuation. For instance, if we have the proposition $(p \vee q) \wedge (\sim p \vee q)$, the fitness of the valuation TF, true for p and false for q, is 1 because with that valuation, the clause $(p \vee q)$

gets true while $(\sim p \vee q)$ is false. In the old semantics, the fitness should had been 0 because the whole proposition gets false with the given valuation.

In this regard, we have two definitions. The first is a generalization of SAT that is the usual test bed for research and the second is a gradualistic version of SAT:

K-SAT is the decision problem (answers yes else not) whose goal is to find a satisfying assignment for a given conjunctive normal formula over N variables, where each clause contains at most K variables taken from a set of N variables. The old SAT problem is now the N-SAT problem. Very explicitly: K-SAT receives as input a logical proposition built over a set of N boolean variables, each of which can be assigned 'true' else 'false', and a set of M clauses. Each clause is a logical OR of K variables (randomly sampled among the given N original variables), where each chosen variable can be negated. The goal is to find an assignment to the variables such that all clauses are satisfied, i.e. a satisfying assignment for the conjunctive global proposition.

MAX-K-SAT=Maximum K-satisfiability is the problem whose goal is to find an assignment that maximizes the number of satisfied clauses for a given conjunctive normal formula over N variables, where each clause contains at most K variables and each chosen variable can be negated. We see that a proposition is satisfiable if and only if MAX-K-SAT ends with a valuation that satisfies all clauses, i.e. if the maximal fitness in the population is equal to the number of clauses in the proposition. So, MAX-K-SAT extends K-SAT and SAT. But we have a gradualistic extension because the fitness can effectively take on intermediate values that lie in within the extreme ones.

Let us solve K-SAT using the gradualistic semantics induced by by MAX-K-SAT.

103 ***The code*** *The next code uses the gradualistic semantics induced by MAX-K-SAT to solve K-SAT. The fitness of a valuation is given by the number of satisfied clauses in the proposition. The code reports the relation of α (the ratio of clauses to variables) and the probability of unsatisfiability.*

```
/* Program C103, KSAT.
 *
 * K-SAT is  the decision problem (answers yes else not)
 * whose goal is to find a satisfying assignment for a
 * given   conjunctive normal  formula over N variables,
 * where each clause contains at most K variables.
 * The prob of negation of a variable is 0.5.
 * Random   equiprobable truth values are assigned
 * to the  variables and each clause is evaluated to true
 * else false. The fitness of a valuation is given by
 * the number of satisfied clauses.
 *
 * Since the fitness can take on intermediate values
 * in within extreme ones, this new setting defines
```

```
 * a gradualistic embedding of SAT.
 *
 * We study the relation of unsatisfiability vs
 * the ratio of clauses to variables, alpha.
 * Since we know the qualitative  answer,
 * we are also testing the software for correctness.
 *
 * A proposition is declared to be unsatisfiable if
 * after 2^(number of Variables) generations no
 * satisfiable assignment has  been found.
 *
 */

package ejvol3v2p;

import java.util.BitSet;
import java.util.Random;

public class KSAT {

     // Turn on  the random generator
    static Random r = new Random();

    //Number of clauses
    private static int nClauses;
    //Number of variables that may appear in the formula,
    // it could be as large as MAXV.
    //Variables will be represented by numbers.
    private static int nVar;
    //Each clause is an OR disjunction of   a sample of
    //KVar variables that could be possibly negated.
    private static int KVar;
    private static final int MAXV = 1000;
    private static final int MAXCLAUSES = 1000;
    private static final boolean CHOSENVAR[][] = new boolean[MAXCLAUSES]
    private static final int LISTCHOSEN[][] = new int[MAXCLAUSES][MAXV]
    //Proposition in formation
    private static String a, b, c;

    private static int nProps;

    private static boolean print;
    private static double alpha, nn;
    private static boolean problem = true;
    //if the prop is satisfiable, sat = true else false.
    private static boolean sat = false;
```

```
//A population of binary numbers is considered.
//The number of individuals is constant
private static int numbInd;
private static final int MAXNUMBIND = 10000;
private static final BitSet IND1[] = new BitSet[MAXNUMBIND];
private static final BitSet IND2[] = new BitSet[MAXNUMBIND];
private static int FitnessInd[]
        = new int[MAXNUMBIND];
private static final int ORDERIND[] = new int[MAXNUMBIND];
private static int gen;
private static int maxGen;
private static boolean recombination;

private static double mutationRate;

private static void setParameters() {
    print = true;
    //Number of clauses
    nClauses = 3;
    //Number of variables
    nVar = 4;
    //Number of variables in a clause
    KVar = 2; //must be less or equal than nVar
    //Number of propositions
    nProps = 1;
    numbInd = 5;
    recombination = true; //true = recombination is enabled
    mutationRate = 0.8;
    nn = nVar;
    alpha = nClauses / nn;
    maxGen = (int) Math.ceil(Math.pow(2, nVar));
    System.out.println("maxGen = " + maxGen);
}

//Not operators are inserted with prob 0.5
private static String negations() {
    for (int j = 0; j < a.length(); j++) {
        if (a.charAt(j) == 'l') {
            int d = r.nextInt(2);
            if (d == 0) {
                a = a.substring(0, j) + "f" + a.substring(j + 1);
            }
        }
    }
    a = a.replace("f", "Nl");
```

```
        return a;
    }

    //Composes a random normal disjunctive full l-proposition
    //with nClauses clauses. Each clause has   KVar variables
    //that are sampled from a set with nVar variables.
    //The NOT operator is included for each l
    //with probability 0.5.
    private static String lProposition() {
        //System.out.println("Construction of the l-proposition");
        //Proposition is initialized.
        a = "(l";
        for (int i = 0; i < nClauses - 1; i++) {
            for (int j = 0; j < KVar - 1; j++) {
                a = a.concat("ORl");
            }
            a = a.concat(") A (l");
        }
        for (int j = 0; j < KVar - 1; j++) {
            a = a.concat("ORl");
        }
        a = a.concat(")");
        if (print) {
            System.out.println("l-prop without NOT = " + a);
        }
        negations();
        if (print) {
            System.out.println("l-prop with NOTs   = " + a);
        }
        return a;
    }

    //For each clause , a sample of kVar variables is made
    //among a set of nvar variables
    private static void chooseKVars() {

        // System.out.println("chooseKVars ");
        for (int k = 0; k < nClauses; k++) {
            for (int i = 0; i < nVar; i++) {
                CHOSENVAR[k][i] = false;
            }
        }
        for (int k = 0; k < nClauses; k++) {
            int n = 0;
            //System.out.println();
            //System.out.println("k = "  + k);
```

```
            while (n < KVar) {
                int j = r.nextInt(nVar);
                if (CHOSENVAR[k][j] == false) {
                    CHOSENVAR[k][j] = true;
                    //System.out.print(j + " ");
                    n = n + 1;
                }
            }
        }
        //Chosen variables are listed
        for (int k = 0; k < nClauses; k++) {
            int n = 0;
            for (int i = 0; i < nVar; i++) {
                if (CHOSENVAR[k][i] == true) {
                    LISTCHOSEN[k][n] = i;
                    n = n + 1;
                    // System.out.print(i + " ");
                }
            }
            //System.out.println();
        }
    }

    //Each l of the l-proposition is replaced by one of
    //the variables. All variables appear in each clause.
    //Variables are not letters;
    //they are numbers from 0 to nVar-1;
    private static String proposition(String a) {
        //System.out.println();
        //System.out.println("l-prop --> proposition");
        int u = 0;
        //The sample of k vars is made for each clause;
        chooseKVars();
        int k = 0;
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {

                Integer vari = LISTCHOSEN[k][u];
                String s = vari.toString();
                a = a.substring(0, j) + s + a.substring(j + 1);

                u = u + 1;
                if (u == KVar) {
                    u = 0;
                    k = k + 1;
                }
```

```
            }

            //System.out.println("k = " + k + "u = " + u);
        }
        if (print) {
            System.out.println("prop               = " + a);
        }
        return a;
    }

    private static void printingJob(BitSet s) {
        System.out.println();
        System.out.println("Truth evaluation begins");
        System.out.println("prop = " + b);
        System.out.println("Tested valuation = " + decode(s));
        System.out.print("Each clause has " + KVar);
        System.out.println(" variables chosen among " + nVar);
        System.out.println("Chosen variable = 1, unchosen 0:");
        for (int k = 0; k < nClauses; k++) {
            System.out.print("Clause " + k + ": ");
            for (int i = 0; i < nVar; i++) {
                if (CHOSENVAR[k][i] == true) {
                    System.out.print("1");
                } else {
                    System.out.print("0");
                }
            }

            System.out.println();
        }
        System.out.println("List of chosen variables");
        for (int k = 0; k < nClauses; k++) {
            System.out.print("Clause " + k + ": ");
            for (int i = 0; i < KVar; i++) {
                System.out.print(LISTCHOSEN[k][i] + " ");
            }
            System.out.print("Valuation: ");
            for (int i = 0; i < KVar; i++) {
                System.out.print(s.get(LISTCHOSEN[k][i]));
            }
            System.out.println();
        }
    }

//The proposition is evaluated
    private static String evaluation(BitSet s) {
```

```
        b = c;
        if (print) {
            printingJob(s);
        }
        int u = 0;
        int k = 0;
        for (int j = 0; j < b.length(); j++) {
            if (b.charAt(j) == 'l') {
                if (s.get(LISTCHOSEN[k][u]) == true) {
                    b = b.substring(0, j) + "T" + b.substring(j + 1);
                } else {
                    b = b.substring(0, j) + "F" + b.substring(j + 1);
                }
                u = u + 1;
                if (u == KVar) {
                    u = 0;
                    k = k + 1;
                }
            }
        }
        if (print) {
            System.out.println("Truth value8 = " + b);
        }
        return b;
    }

//The proposition is evaluated until the end
    //Not operators are evaluated at the
    //very beginning for they apply over the next
    //to the right variable.
    private static String simplification(String b) {
        // while ( b.length()>1 )
        boolean progress = true;

        while (progress) {
            String f = "" + b;
            b = b.replace("NT", "F");
            b = b.replace("NF", "T");
            b = b.replace("TORT", "T");
            b = b.replace("TORF", "T");
            b = b.replace("FORT", "T");
            b = b.replace("FORF", "F");

            b = b.replace("(T)", "T");
            b = b.replace("(F)", "F");
```

```
            if (print) {
                System.out.println("Truth value1 = " + b);
            }
            if (b.equals(f)) {
                progress = false;
            }
        }
        return b;
    }

//Converts a BitSet value into a String
    static private String decode(BitSet t) {
        String s = "";
        for (int i = 0; i < nVar; i++) {
            if (t.get(i) == true) {
                s = s + "T";
            } else {
                s = s + "F";
            }
        }
        return s;
    }

    //Our pop is formed by valuations.
    //IND1 is a population of binary strings.
    //The BitSet class is used to instantiate
    //binary strings as individuals
    private static void initializePop() {
        BitSet s = new BitSet();
        //IND1 is false everywhere
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) s.clone();
        }
        //Reinitialization with random truth values.
        for (int i = 0; i < numbInd; i++) {
            for (int j = 0; j < nVar; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
            //System.out.println("IND1 : " + decode(IND1[i]));
        }
    }

// Does an individual   satisfy  the target?
```

```
    static private int assessInd(BitSet s) {
        String bb = evaluation(s);
        bb = simplification(bb);
        int fitness = 0;
        for (int i = 0; i < bb.length(); i++) {
            if (bb.charAt(i) == 'T') {
                fitness = fitness + 1;
            }
        }
        if (print) {
            System.out.println("bb      = " + bb);
            System.out.println("fitness = " + fitness
                    + " of " + nClauses);
        }
        return fitness;
    }

// Individuals are sorted by fitness.
// the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        int FitnessIndc[];
        FitnessInd = new int[MAXNUMBIND];
        FitnessIndc = new int[MAXNUMBIND];
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < numbInd; i++) {
            Champ = 0;
            for (int j = 0; j < numbInd; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array ORDERIND orders individuals by  fitness:
            //The fittest is number zero
```

```
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        // if Pattern has been guessed
        if (FitnessInd[ORDERIND[0]] == nClauses) {
            problem = false;
            sat = true;
            //System.out.println("Champ = " + IND1[ORDERIND[0]]);
            b = c;
        }
        return problem;
    }

// The new generation is made
    private static void newGenerationInd() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest
        while (i < numbInd) {
            int counter = numbCopies;
            while ((counter > 0) & (i < numbInd)) {
                //System.out.println(i + "" + l);
                IND2[i] = (BitSet) IND1[ORDERIND[k]].clone();
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int j = 0; j < numbInd; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*  System.out.println( j+ " IND1 "
                                    + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(numbInd);
        //b is false everywhere
        BitSet bb = new BitSet();
        //B is the recombinant of l and k
```

```
        for (int j = 0; j < nVar; j++) {
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    bb.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
                    bb.set(j);
                }
            }
        }
        return bb;
    }

// The definitive population  is a recombinant one.
    private static void recombinationInd() {
        //IND2 is a copy of IND1
        for (int i = 0; i < numbInd; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //IND2 is a recombinant version of IND1
        for (int l = 0; l < numbInd; l++) {
            IND2[l] = recInd(l);
        }
        //IND2 is transferred to IND1
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }

// Individual  a is mutated at a random place.
    private static BitSet pointMutationInd(BitSet a) {
        int l = r.nextInt(nVar);
        a.flip(l);
        return a;
    }

// Individuals are mutated
    private static void mutationInd() {
        for (int i = 0; i < numbInd; i++) {
            double p = r.nextDouble();
            if (p < mutationRate) {
```

```
                IND1[i] = pointMutationInd(IND1[i]);
            }
        }
    }

    // Given a goal, it is guessed by evolution
    private static void evolution() {

        //The new generation is produced
        //as copies of the fittest
        newGenerationInd();
        if (recombination) {
            recombinationInd();
        }
        //The new population is subjected to mutation
        mutationInd();
        gen++;
        problem = sortingInd();
        System.out.println("Gen = " + gen);
        if (gen == maxGen) {
            problem = false;
            sat = false;
        }
    }

    public static void report1() {
        System.out.println();
        System.out.println("nClauses = " + nClauses);
        System.out.println("Total N of variables  = " + nVar);
        System.out.println("Variables per clause = " + KVar);
        System.out.println("alpha = " + alpha);
        System.out.println("Repetitions = " + nProps);
        System.out.println();
    }

    /* We generate at random nProps satisfiable props,
   * and report the mean and deviation of the
   * necessary number of individuals to get
   * a satisfying assignment.
     */
    public static void main(String[] args) {

        setParameters();
        if (KVar > nVar)
        {
            System.out.println("Error: KVar > nVar");
```

```
            System.exit(0);
        }
        double nSat = 0;
        for (int i = 0; i < nProps; i++) {
            System.out.println("***********");
            System.out.println("Prop numb = " + i);
            problem = true;

            problem = true;
            sat = false;
            b = lProposition();
            c = b;
            proposition(b);
            initializePop();
            gen = 0;
            while ((problem)) {
                evolution();

            }
            if (sat == true) {
                nSat = nSat + 1;
                System.out.println("Satisfiable");
            } else {
                System.out.println("Unsatisfiable");
            }
        }
        report1();
        double p = 1 - nSat / nProps;
        System.out.println("Prob of Un-Satisfiability = " + p);

    }
}//End of Program C103, KSAT.
```

104 ***Exercise:*** *Run the program and play with the code.*

105 ***Exercise:*** *Run the original program for diverse values of α ranging from 1 to 60 and record the probability of unsatisfiability. To that aim, in the method setParameters(), at the beginning of the body, change appropriately the value of nClauses keeping everything else unchanged. That method looks so:*

```
 private static void setParameters( )
 {
  //Number of clauses
  nClauses = 5;
  //Number of variables
  nVar = 3;
  //Number of variables in a clasue
```

```
  KVar = nVar; //must be less or equal than nVar
  //Number of propositions
  nProps = 100;
  numbInd = 50;
  recombination = true; //true = recombination is enabled
  mutationRate = 0.8;
  nn = nVar;
  alpha = nClauses / nn;
  maxGen = (int) Math.ceil( Math.pow(2, nVar));
  System.out.println("maxGen = " + maxGen);
 }
```

You must reproduce what we already know: the higher is α, the more entangled is a proposition and so it is more probable to get unsatisfied: the probability of unsatisfiability is directly related to α. Comment: you can make contact with the international literature if you compare your results with those of Monasson et al ([51] 1999), where instead of a probabilistic definition of satisfiability, as the one used by us, an exhaustive and systematic scan over all possible assignments is done. According to the cited authors, the most interesting region to study the relation between α and the probability of unsatisfiability is when the number of variables passes from 2 to 3. Intrigue: how can we define a number of variables such as 2.3? Answer

106 ***Exercise:*** *According to what we have say, we expect that the probability of unsatisfiability shall be a good function of α, the ratio of clauses to N, the total number of variables. Test this prediction. In other words, show else refute that the probability of unsatisfiability is almost the same when the number of clauses is α times that of variables, for any number of variables. Take a number of variables greater than 3. Answer*

107 *Does the gradualism induced by MAX-K-SAT break down the exponential regime?*

Since we have solved SAT by means of the gradualistic semantics induced by MAX-K-SAT, we can investigate now if our new setting breaks the exponential regime down.

Now, what is a good object for experimentation? As before, we might solve SAT restricted to disjunctive normal forms that are always satisfiable. In that way, we make sure that we must arrive to an end. The good news are that we need no experiment at all: in regard with disjunctive normal forms, the fitness induced by MAX-K-SAT coincides over disjunctive forms with ONE clause with that given by the full discontinuous semantics of SAT.

Therefore, we have no reason to think that this new setting breaks down the exponential regime demanded by SAT.

Anyway, we must emphasize that the SAT contains a very great number of subfamilies, some of which might be solved at once by evolution. Example: consider the family of tautologies, say, $p \vee \sim p$ and all its duplications. Each member of this family is satisfied by all possible valuations, so evolution solves this family at once.

Since SAT has many subproblem of varying complexity, let us get used to the jargon: an exponential number of operations is demanded in the worst case of SAT but many subfamilies of SAT might be easier for evolution even with immediate solutions.

108 ***Diffusion toward higher functional complexity***

The inverse relation between α, the ratio of clauses to variables, and satisfiability that is observed in our study of the conjunctive family allows us to predict that the longer a satisfiable proposition is, the harder is for it to diffuse further into higher complexity. This would imply that a nascent function can be plugged to many complex functions but that well established functions rarely would admit extensions and changes. This is a property of any function that comes into existence be it by self-organization, by evolution, by design or by whatever possible cause.

According to Kuhn, this property is also applicable to ideologies in their period of formation: overdeveloped ideologies, and the brains that sustain them, are bound to mummification, but new and ill-terminated ideologies, and the young brains that sustain them, are plenty of future, modifications and surprises. In sociological terms, this implies that: 'a new scientific truth does not triumph by convincing its opponents and making them see the light, but rather because its opponents eventually die, and a new generation grows up that is familiar with it' (Kuhn, [42] 1970). In terms of biology, we can infer that a species is more probable extincted than reformed into a new species: this is **the fatigue of evolution** be it natural or artificial or simulated.

109 ***Exercise*** *Compose a program to exhibit a direct verification of the principle of the fatigue of evolution and of computing systems in general: the longer a satisfiable proposition is, the harder is for it to diffuse further into higher complexity (if the number of variables is kept constant).* *Answer*

110 ***The mystery of the pink strip***

One of the most terrible frustrations of modern science is that the arising of new species is more a matter of theory than of laboratory results or of field observations. In that regard, we presented with pleasure the results of the previous exercise, in which it is predicted under a given setting the fatigue of any process that picks up complexity, be it evolutionary or not. Nevertheless, the complete reality is much more complex than that, as we see in the next exercise.

111 ***Exercise*** *Use the code developed for the exercise 109 to prove that the fatigue of evolution has nothing to do with evolution in general: one can extend a satisfiable proposition in infinitely many diverse forms and with infinitely many informative clauses if only one allows the introduction of sufficiently abundant new variables.* *Answer*

We see that one can extend a satisfiable proposition in infinitely many varied forms and lengths if one adds sufficiently many new variables. This is a strong result that is not obvious: one could think that it is highly improbably to introduce new variables, new genes and functions, without disturbing the actual functions that enable life. Nevertheless, that can be done and can be done with absolute certainty.

So, evolution would be similar to a female cat, whose neck has been adorned with a *pink strip*, and, since the strip is not annoying the cat in any sense, the cat goes around without taking care of the strip. The genome is predicted in this way to be overloaded with any sort of nascent genes, whose function is void.

So, a mystery appears: if complexity has no limit whatever, why is natural evolution so terribly tired? Otherwise we should see at every pad or valley the arisen of new species. Can we explain this mystery?

In ancient times, the prediction of an overloaded genome would be considered as verified with a simple and direct proof: in eukaryotes there is a code that is not transcribed, which is highly repetitive and that can be 50000 times as much as that which is transcribed into RNA. But today, it is anymore secure that the genome contains junk code: a given pseudogene (a modified copy of a functional gene that does not serve as template for producing a protein), in other time considered junk DNA, is now thought to be important part of the gene regulative machinery (Lee, [45] 2003). Additionally, according to Arneodo et al., ([4] 1996) that 'junk' DNA has a clear function: the folding of eukaryotic DNA into the cell nucleus involves a hierarchy of structures on different length scales that is determined by self-correlation in DNA, mainly of the so called junk type.

So, a lot of work that is not done as yet is necessary to show that the genome contains pink strips. Anyway, recombination is a natural and permanent operation in nature, and it can lead to gene duplication. Thus, we think that expansive forces are always at work upon the genome, trying to enlarge it. But a contrary simplifying force is also under operation, as a study of parasites shows. A parasite is an organism that takes some elaborated resources from others living beings, and therefore the genetic programs of parasites, coding for restricted metabolic pathways, are shorter than those in wild ones. It has been found that the shorter programs of parasites are more competitive than entire and large ones of wild varieties:

"The mechanisms of feedback inhibition and/or gene repression which would help the parental strain to economize are not universal and/or not complete enough to offset the advantage that the "defective" strain has in not being involved at all in this particular biosynthetic process" (Zamenhof et al., [77] 1962).

That parasites have shorter genomes was known since 1887, when Bovery discovered that Ascaris has in its somatic cells only a part of the chromatin which is present in the germ line cell nuclei (see Ammermann, [3] 1985).

Viruses is another "easy case" for selection in favor of simplified genomes: their code has overlapping genes using different reading frames, overlapping genes using alternative promoters in the same reading frame and differential splicing of a single premessenger to produce a variety of polypeptides in adenovirus. Such a proficiency in information packing leads one to think that in a virus everything is fundamental. In this regard, the following comments that appear in Cavalier-Smith ([13] 1985), and that he attributes to Joklik, are very illustrative:

"Plasmid DNA and virus chromosomes are always much smaller than cellular chromosomes. As they lack significant amounts of non-coding DNA reasonably strong selective forces must exist to prevent its accumulation. For viruses the nature of this force is fairly obvious. The closed geometry of a viral capsid cannot contain more than a limited amount of DNA. Moreover capsid size, unlike cell size, cannot increase gradually but would have to evolve in major discontinuous steps that are inherently improbable". By the same token, he proposes that "the uniformly low chloroplast genome size is a sign of strong selection against wasteful noncoding DNA".

So, we have forces that enlarge the genome and forces that simplify it. What kind of forces do prevail?

It seems that, in general, amplification is stronger because large deletions have a greater tendency to be lethal than large duplications (Cavalier-Smith l.c. , [13] 1985). Moreover, gene amplification is one strategy of defense of cells against poisons because, say, their enzymatic destruction is made faster. So, redundancy is kept in within strict bounds. Nonetheless, one cannot think that redundancy are all to nature. Just think of the so called C-paradox: why is the DNA content usually constant in any one species while it can vary in different species by over a million-fold? (Cavalier-Smith l.c., [13] 1985).

Anyway, experimental deletions has shown today that in *S. cerevisiae* that are cultivated in rich media less than 20% of genes are essential, and the large majority of deletions in the remaining 80% has little or no detectable effect on growth. Moreover, similar observations have been made for other eukaryotes and prokaryotes. Is this discovery a direct solution of the mystery of the pink strip? Not at all because a different explanation also must be studied: the genome is functionally redundant and moreover it has many genes that exist to be used under poor environments and/or extreme conditions (Ihmels et al, [37] 2007).

We have now a contrast: on one hand, we have evolution that is in literal sense open ended, but on the other, the very existence of species shows that evolution has got tired. How do we explain the mystery of the fatigue of evolution when the possibilities of evolution are unbounded at any one moment?

Our answer is this: the cell is computing machinery that is always trying new ways but the problem is that it has not enough computing power to devise at once the highly functional code that is demanded to synthesize new functions that are acceptable for selection.

You can disagree with our conclusion by arguing that natural evolution has programmed itself to get tired. So, you can carry out a crucial experiment: dismantle the mechanisms that program evolution to get tired and prove that evolution is by idea and by fact unlimited. In particular, you must succeed in protecting the elongation capability of DNA from negative selection due to competition with shorter and less expensive genomes.

It is now healthy to pass from very abstract discussions to the consideration of some concrete facts of nature.

112 *The race for perfection*

Darwin in his Origin was the first to propose that a wonderful function, such as the vision enjoyed by complex eyes, can arise by accumulation of small change. Given the perfection of so many wonders of so diverse nature that living beings contain, we conclude that, according to the evolutionary theory, the race for perfection is a successful routine for natural evolution. In the light of SAT this is obviously false: the perfection of extant functions is by far more difficult than the invention of low quality functions. Just consider that you can learn the basics of a new language in some 10 months but with difficulty you will reach in ten years the level of a native speaker. Why? it is because perfection is described by two dual flavors of SAT. The first deals with diffusion over satisfiable propositions: do things if just they function. The second is related with SAT as a restrictive factor: refurnish but be aware

of the fact that reforms may cause strong damage to a sufficiently well defined function. Both aspects of SAT are loaded with the problem of stagnation, i.e. of increasing periods of ecstasy.

Let us recall that a function is a property that helps surviving. Our experience with SAT invites us to believe that nascent functions can be synthesized quite easily. Notice now that **evolution has a purpose: to perfect nascent functions.** The race for perfection locates evolution at the same level of engineers: both have as common purpose to improve a function that is already operating. Therefore, all the theory of complexity than can be applied to engineers also can be applied to evolution.

113 *Duplications and SAT*

We have seen that evolution has two channels to diffuse across complexity: to work over the perfection of extant functions and second to tinker with new, momentarily useless properties. Our description of evolution contrasts with that of the scholastics: gene duplication (Meyer, [49] 2003) is the first step of evolution toward complexity, which eventually could be followed by mutation. How are related these two points of view?

In the light of SAT, two aspects must be considered. The first is that when one has a satisfiable proposition p, then $p \wedge p$ and $p \vee p$ are both satisfiable. So, duplications are tautological methods to diffuse into complexity. But duplications represent a cost for the carrying individual. So, to be favored by selection, they must offer some advantage. Therefore we have here also the dual role of SAT as imposing restrictions to diffusion into higher complexity.

114 *Diffusion in folding space*

Our general purpose is to study the power of evolution as a filtered diffusive process. In this regard, it seems to us that the world of protein folding is a very good field to contrast theoretical and observational facts. That field slowly develops but it nevertheless shows interesting results:

A protein is a string of amino acids (first structure) that suffers some structural conformations among nearly amino acids (secondary structure), and in the large 3D scale (tertiary structure) and among diverse strings (quaternary structure). How can we assess the similarity among diverse protein structures? A usual methodology is to classify the diverse structures by some descriptors, thank to which proteins are mapped to a given space. Similar structures would appear close one to another. This is the cluster approach to protein structure. A very different approach based on ideal archetypes also has been attempted (Taylor, [70] 2002).

Some 10 structures might cope for as much as the 30% of protein structures (Orengo, [56] 1994). This fact might be explained by physical constrains, by common descent and by sampling effects. On the other hand, it seems that the number of different modes as a protein can fold (in the range of thousands) is larger than the number of folds observed in nature (Chothia, [16] 1992; Taylor l.c., [70] 2002).

How can we explain that deficit?

In the light of the mystery of the pink strip, given the expected power of mutation plus the easiness of diffusing in new areas, folding space should be completely occupied. But this is not so: Does folding space contain bewitched places that somehow could be forbidden to diffusion? This question has been answered by Kuhlman et al ([41] 2003), who succeeded

in designing a globular protein with a structure that was absent from the data bases. So, we observe a deficit of occupation by diffusion that shall not happen. What do we have to say?

Let us stress that easiness of diffusion is defined in terms of the relation of available computing power with the complexity of tasks. The deficiency in nature of occupancy of folding space shows again that there are problems that are too heavy for natural evolution. So, we have every reason to belief that there are subfamilies of SAT that cannot be solved by evolution if not in the exponential regime and that these already happen in nature.

115 *Snake venom*

The life we know is full in excellence, variety and complexity. A very vivid example of an excel function is the next one related with snakes.

One is used to imagine that a snake injects into the prey a specific substance that inhibits, say, transmission of nervous signals. Such action of poisons of snakes is really important but not the only one. Snake venom is indeed a cocktail (Wikipedia: [75] 2008b; Montgomery et al, [52] 1983) that may contain:

- Inhibitors of cholinesterase to make the prey lose muscle control (inhibitors operate not over the nervous cell but over muscles).
- Phosphodiesterases that are used to interfere with the cardiac system of the prey, mainly to lower the blood pressure. Phosphodiesterase could also interfere with the formation of RNA that is used in transcription thus inhibiting the synthesis of proteins and of any repair system.
- Phospholipases that act on membrane lipids producing free fatty acids that attack the mitochondria destroying the proton gradient and killing in that way the synthesis of ATP.
- Hyaluronidase that increases tissue permeability to increase the rate that other digesting enzymes are absorbed into the tissues of the prey.
- Amino acid oxidases and proteases that are used for digestion. Amino acid oxidase also triggers some other enzymes and it is responsible for the yellow color of the venom of some species.
- Snake venom often contains ATPases which are used for breaking down ATP, the molecular source of energy of the prey.

How did such extremely complex functions, as this, appear? Let us make a try with evolution:

To begin with, it is clear that there are non poisonous snakes too and that their anatomy and physiology is simpler than those of poisonous ones.

So, we must retro-predict that poisonous snakes arouse by evolution from non poisonous ones and that this should be clearly written in the fossil record. So, one could suppose, for instance, that at the beginning, a snake arouse that was able of exuding a slightly dangerous substance. It was exuded precisely because it represented a danger for the internal milieu of the snake. But once it has been exuded, we have there the SAT problem of making a

defense of the skin of the snake against any possible damage of that substance. Next we must imagine the arising of a system for injecting that substance in the prey together with the ensuing improving of the whole cocktail. Or we can begin from the end: an injecting system was formed at the beginning that then was used to inject a poison.

Explanations of this type were sufficient for science not long ago. But we fear that the time of plausible, fuzzy explanations is gone and is gone forever. Now, we must provide specific mechanisms to be tested under experiment. We have a report of one experiment in relation with the reaction of natural selection against poisonous snakes if they already have appeared somehow:

It has been usual to imagine that the poisonous cocktail adds adaptive value to the snakes, say, by helping assimilation of the pray. It is pleasure to communicate that such a belief has been tested now by direct experiment:

'A complete randomized block design allowed repeated measures of specific dynamic action and gut passage time to be measured in eight snakes ingesting four feeding treatments (i.e., artificially envenomated live mice, artificially envenomated prekilled mice, saline injected live mice, and saline injected prekilled mice). A second experiment measured ASSIM [apparent assimilation efficiency] in eight snakes ingesting a series of six artificially envenomated or six saline injected mice meals over an 8-week period. Contrary to expectations, the results of both these experiments revealed that envenomation had no significant influence on any of the measured digestive performance variables.. .. The results of this study reinforce the need to systematically examine the potential adaptive advantages that venoms confer on the snakes that produce them' (McCue, [46] 2007).

So, the problem of the poisonous snake is open and so are thousands of other ones. Let us highlight that they are not only open but also extremely challenging. Let us explain why.

As we have seen, the diffusion of evolution toward higher complexity can be made along two channels. The first is toward perfection of the existing functions. This looks like diffusion under a constant number of variables. This is possible but highly improbable. The second channel is that of the pink strip and happens through the synthesis of useless items that expect the appearing of appropriate partners to form a valuable function. This channel is highly probably if one can provide sufficiently numerous new variables. Because evolution is pray of sampling effects, the last channel is the winner. But this would imply that perfection would be rarely achieved.

Actually, the use of evolution in intentional design has proved that these two channels can cooperate one with another: a period of expansion of superfluous properties is ensued by filtering of functions, which could be followed by selection against redundancy. This mode of operation has shown to be productive in technology, see for example, Murukawa et al., ([53] 1997). So, it is not excluded that it could also be productive in natural evolution. But this to be acceptable needs the affording of the next issues:

1. A mechanism to effectively synthesize new arrangements. Gene amplification, a natural phenomenon, is not suitable to fill this option because it happens by tens or hundreds but not by billions.

2. A synchronizing mechanism to enable an initial dominance of the synthesis of nascent functions, which must give entrance to selection for functions and simplification.

In our opinion, the best stage to consider that this is a real possibility in biology is the microbial world with its horizontal gene transfer: microbes conform a bank of genes that are

reflushed by horizontal gene transfer with the eventual formation of powerful gene aggregates that are useful to combat, say, antibiotics.

5.6 Algorithmic complexity theory

The incapability of dealing with all the potentialities of evolution in concrete simulations causes a great incertitude, which must be attacked with mathematical methods that study the generality of algorithms. The corresponding theory is known as **algorithmic complexity theory** and forms now a body that is well defined, well developed and is plenty of future. The next is an introduction that is sufficient to connect our work with that discipline:

116 ***Deterministic and non-deterministic machines***

A string of symbols that can be interpreted by a code and executed is called a **verbal instruction**. A computer is a machine that can follow verbal instructions. Technically: a computer can calculate recursive functions, i.e., functions that use recursive calls to predefined methods, modules, subroutines or functions.

An algorithm is a recursive set of instructions for a given computer that solves a specific problem, say, to order n numbers (Caicedo, [12] 1990; Ebbinghaus et al. [21] 1984). The parameter n measures the input complexity of the task, but because of counting problems we prefer to take its logarithm in binary base. Say, if $n = 8 = 2^3$, we take 3 as the standard measure of the input complexity. Its official name is the **length of the input**.

We will consider various types of computers: the first one corresponds to ordinary computers that execute instructions serially, one after other. Next, we have parallel computers that have possibly infinitely many interconnected processors and can execute many serial programs at the same time. In computing science, we have formal definitions related with these two types of computers:

A **deterministic machine** is a computer than can follow simple instructions one after another. The instruction 'add 5 and 3' is simple. The instruction 'if ($x > 3$), add 7 to x else add 4 to x' is not simple because it has a branching: one branch goes for $x = 9$ and another for $x = 1$. Branching is not necessarily tied to if-then conditions. Say, a program can function like a mother that prepares food without salt for her husband and other food with salt and pepper for her son at the same time. So, branching really means adopting parallelism when it is needed.

A **non-deterministic machine** is a computer than can follow branched instructions.

We have noticed that some programs may run during too long periods of time without producing an output. Others cannot be run because of a deficit in memory resources. Our finding is a universal nightmare: no matter how developed will be the technological resources, one always can find algorithms that get frustrated because of time and/or memory limitations. The study of this thematic is known as **algorithmic complexity theory**.

Although there are many ways of measuring the complexity of algorithms (Balcázar et al. [6] 1995), we choose the most elementary election: for serial computers, **the complexity of an algorithm** is the number of fundamental steps needed to execute the program completely in the worst case. For nondeterministic computers, the complexity of an algorithm is defined

as the number of elementary steps taken by the largest branch in the worst case. For instance, if a deterministic algorithm needs at least 125 simple steps to order in the worst case 8 numbers, we say that this algorithm has complexity 125. The complexity of an algorithm depends crucially on the length of the input: since ordering is related to permutations, whose number is related to n!, which is greater than an exponential, we expect that the complexity of the ordering problem could be bounded below by an exponential function in the length of the input. Of course, this must be verified because the complexity of a problem is not the complexity of the first algorithm that comes to the head but that of the best one.

In order to compare the complexity of different algorithms, it is usual to restrict oneself to a very concrete type of problems (no generality is lost):

A **decision problem** is a problem that formulates a question in relation with a universal set that must be answered as Not else Yes. Example: SAT questions about a given boolean proposition: is this proposition satisfiable?

We also need to take account of different operational systems, programming languages and tricks of intelligent programmers. It has been shown that the following definitions are good enough to deal with this variability.

P is the class of decision problems solvable in polynomial time by a deterministic computer (Turing machine, for specialists). To add together n numbers is clearly in P (add together n numbers can be recasted as a family of decision problems by asking: Is $a+b+....+n=m$?

NP is the class of decision problems solvable in polynomial time by a non-deterministic computers. SAT is in NP (see below).

117 *SAT is in NP*

To understand why SAT is in the class of problems that can be solved in polynomial time by a non deterministic machine, we need the next observation: SAT is associated with a tree whose complexity along any branch is bounded above by a polynomial function in the length of the input, i.e., in the number of boolean variables.

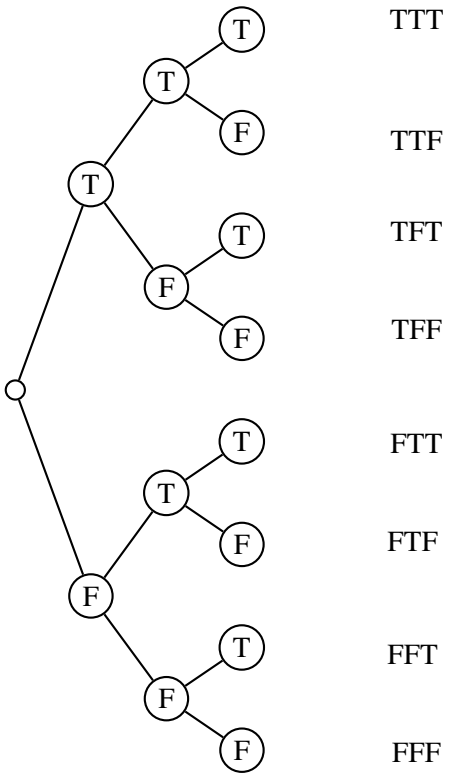


Figure 5.0. All 2^3 valuations of a set of 3 boolean variables are associated with a tree that have 2^3 leaves. The valuation TFT associates True to the first variable, False to the second and True to the third one. Each valuation can be built in 3 steps.

That tree is built just by posing at the root the initialization symbol, from which two branches leave, one for False, one for True, the two possible values of the first variable. Next, from each new node, two branches leave for the two possible values of truth of the second variable. And so on. We have a tree, whose complexity along any branch is precisely the number of variables, so it is linear, therefore it is polynomial in the length of the input. This argument shows that one can generate all valuations using a non deterministic machine and running it in polynomial complexity.

Let us recall now that in our programs with SAT, apart from generating valuations, one must evaluate the given formula or proposition to see its final value, true else false. To that purpose, we used suitable grammars, i.e., sets of rules for substitutions. Can be this be done for any one formula by a non deterministic machine in polynomial time?

The good news is that any formula can also be represented by a tree. This result is at the root of LISP and of many powerful simulations about evolution and genetic programming, so let us illustrate it with a very simple example:

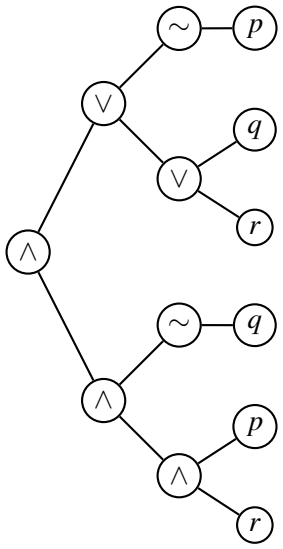


Figure 5.1. This tree represents the formula $(\sim p \vee (p \vee r) \wedge (\sim q \wedge (p \wedge r))$.

Let m be the number of operators of the formula. Any leaf in the corresponding tree can be reached from the root in less than m+1 steps. So, the tree can be built by a non deterministic machine in a number of steps that is less than m +1. This proves that a non deterministic machine can decide whether or not a formula with n variables and m operators is satisfiable in less than $n+m+2$ steps, or in linear time. We see that SAT is really simple for a non-deterministic machine in relation with time as a resource.

What does happen when one tries to solve SAT with a serial machine? The usual solution is to visit all the leaves of the tree that generates the valuations, which are 2^n, where n is the number of variables. This is why a serial computer seems to require an exponential number of trials to solve SAT.

Let us notice now that in complexity theory an exponential number of trials look like an antonym of a polynomial number of trials. This is indeed so and the reason is the following:

A polynomial is a function of the form:

$p(x) = a_o + a_1 x + ... + a_n x^n$

The exponential function e^x can be approximated as much as desired by polynomials of the form

$p(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} .... + \frac{x^n}{n!}$

To get a better approximation, one needs just to increase the degree of the expansion. By this reason, we say that the class of functions that is immediately below the exponential class is the polynomial class. So, if one wants to break an exponential regime, the natural alternative is the polynomial one.

118 *NP >?P*

It is clear that P is contained in NP: if a deterministic machine can solved a problem in polynomial time, then a non deterministic machine can do the same going after the steps of the deterministic one. According to experience, when a serial computer or a deterministic machine attempts to solve SAT or a problem of similar complexity, the serial computers requires an exponential number of operations. So, an ordinary problem for non deterministic machine is seen by a highly complex problem by a serial one.

Nevertheless, it is not allowed to generalize current experience to the status of truth: such a graduation must be done on theoretical backgrounds alone. Why? Because experience is based on propositional logic as in the proposition '6 is even', but mathematical thinking is based on predicates, on abstractions, as in '8n is even for any integer n'. So, mathematics is more powerful than experience.

All in all, the relation between P and NP is theoretically an open problem. Most people think that $NP > P$ strictly but some few persons think that P=NP. This disproportion among thinkers can now be rationalized, because in a given and precisely defined sense, $NP > P$ with probability 1 (Bennett and Gill, [9] 1981). Anyway, the crude reality is that the P=?NP question is unsettled: the elucidation of the exact relation between P and NP is recurrently one of the most famous open problems of mathematics (Smale, [65] 1998).

119 ***Exercise*** *Analyze the following reasoning: One can see that the tree representation to generate valuations and to evaluate a formula are optimal, i.e., they cannot in general be simplified. We say that they are incompressible. Henceforth, it is obvious that a serial machine must generate, in general, 2^n valuations, one after another. In another terms, it is obvious that a serial machine needs an exponential number of steps or more to decide whether or not a formula is satisfiable. This immediately proves that SAT cannot be solved anyhow in polynomial time: $NP > P$.* *Answer*

120 ***SAT is NP-complete***

One could try to elucidate which are the most powerful algorithms in a given class. Informally, we say that a problem is **NP-hard** when it is universal to NP in the sense that the know-how to solve this problem can be reused at low cost (polynomial cost) to solve any one problem in NP. If, moreover, a NP-hard problem belongs to NP, we say that the problem is **NP-complete**. Thus, a NP-complete problem is a maximal element in NP, when this class is ordered by reusability. To be fully rigorous one shall replace reusability by reducibility (see Balcázar et al. l.c. [6] 1995).

SAT was the first problem to be shown to be NP-hard and NP-complete. Our work allows us to agree at once with the announcement of hardness: SAT is very difficult, i.e., we know how to solve SAT by randomness and by evolution, but both solutions fall in the exponential regime.

We have now more than 3000 NP-complete problems. For an introduction to this thematic, see

http://en.wikipedia.org/wiki/List_of_NP-complete_problems

Each one of these problems is equivalent to SAT. So, they can be reinterpreted as alternate versions of SAT. Such problems may look quite different than SAT, but that is only at the surface. We can illustrate this idea with the problem on **protein design**, which has as aim to find a sequence that fits, say, nice thermodynamic characteristics.

Let us try to see why the problem of protein design is another version of SAT: when we deal with a two valued logic, False else True, we have boolean logic. But instead of two values, we can consider 5, 10 or 25. This extension is called modal logic. Now, we can consider the problem of protein design as a version of SAT for a modal logic: each position in the protein is equivalent to a variable that can assume one among 25 values, because proteins are strings with an alphabet of some 25 letters, the amino acids.

We see at once that protein design is apparently a problem that is not less difficult than SAT. But again, we must leave plausible explanations and go for exact proofs. So, what do we have? A formulation of protein design as a decision problem has been proved to be NP-hard (Pierce and Winfree, [58] 2002), i.e. no less difficult than SAT. The formulated problem is rather academic, but it is a model of what the complexity of protein design could mean for biotechnology. Since SAT is NP-complete, protein design looks harder than all problems in NP. What does this mean? This mean that if someone devises a protein, by whatever mean, with evolution or without it, he or she has solved more than 3000 extremely difficult problems from all fields of science and technology. Hence, he or she is invited to publish his or her results in the most famous journals of the world, as it actually happens. But if moreover the procedure runs in polynomial time, the glory is expected for sure.

When studying the complexity of SAT, we considered the possibility to embed SAT into a gradualistic world and to that aim we used, with negative results, the fuzzy logic of Zadeh. A similar consideration can be made here. Let us see: the proteins of livings beings contain no more than 25 amino acids. These molecules are grouped into various families because they share some chemical properties. Anyway, when one replaces one amino acid by another one, a high probability exists that a deleterious or lethal mutation could appear. This looks like a discontinuous semantics. Can this be remedied somehow?

The fact is that an amino acid is distinguished by its capability to form a peptide bond, but apart from the ensuing restrictions, an amino acid might have a side chain of whatever nature. So, the possible number of amino acids is literally infinite. Actually, plants produce more than 150 different amino acids. Can we extend the present genetic discontinuous genetic code into another more gradualistic, more evolvable, thanks to all those possible amino acids? Can that world be used to solve SAT more easily than we can today?

We have a reason to expect not too much: pattern recognizing operations of proteins and enzymes are mainly due to short range interactions that are very weak, so a close spatial complementarity is needed to produce effects. Therefore, it looks rather unavoidable a discontinuous semantics in the chemical world. Anyway, if a complete relief cannot be found by proposed gradualism, maybe a partial remedy could be important.

121 *Complexity highways*

NP is a great family. Maybe it is too large in comparison with the real problems of biology and evolution. More mechanistically, it is convenient to retake at this moment a question that was formulated above but that was dismissed by gradualism: Does life represent the full complexity of SAT, which is our epitome of complexity?

It might happen that a rather simple recurrent constructive process could form the fundamental raw material which was then given to evolve, to diffuse into the realm of complexity. It is much similar to a highway that does not carry you to the door of your house but that with the help of locally appropriate rows can satisfy all needs of communications in an expedite way.

We can give now a rather secure answer to this important preoccupation: there are more than 3000 NP-complete problems. Each one of these problems is an interpretation of SAT. Most probably, such interpretations are also good to model some aspects of life. So, it is naive to expect that the full complexity of SAT is not represented in biology. Let us contrast now this point of view of complexity theory with that of modern biology that teaches that the origin of species is explained by the evolutionary process.

Taking into account the extreme diversity and perfection of the forms of life plus the cosmological bounds of time, space, energy and matter, to say that the diverse species appeared by evolution is the same as to say that natural evolution can solve the NP class, SAT included, definitively better than chance, say, in polynomial time or less. In short: evolution claims to be almighty in the space of problems. In terms of theory of complexity this implies that a positive, highly productive and obvious answer to the NP$>$?P question must exist.

We find a contradiction among these two points of view, one coming from complexity practice, the other from modern biology. So, we must ask: can we expect something extraordinary from natural evolution? Not. Let us see why: the computing power of natural evolution has already been handed to technology by the very modern DNA - computers, a technological trend begun by Adleman ([1] 1994). It is clear that these machines cannot use any magic and that they are burdened by the same complexity loads that hinder all other computing system. Moreover, the real branching capabilities of these machines are bounded. So, the natural parallelism of DNA computers can only be useful for small problems, because for large ones, a computer with finite parallelism is indistinguishable from a serial one. So, for the problems that worry us, the origin of life and of species, both technology and nature work as serial machines, i.e. in the exponential regime.

122 ***Exercise*** *There is a great difference between electronic computers and DNA computers: electronic computers work with microscopic entities that are giant in comparison with the atoms and molecules of a DNA computer. This could mean that memory resources could eventually be multiplied by orders of* 10^{15} *and that transfer time by 100 or 1000. Who dares to think that this is not enough to cope with all the computing needs for the appearing of life and of species? Answer*

123 ***Exercise*** *Analyze the next reasoning: Evolution is obviously almighty in the space of problems. To see this, just consider that the terrible SAT can be solved by a non deterministic machine in linear time. But if a non deterministic machine can do something, evolution also can do the same because evolution is per se a parallel processing system that can achieve solutions by mere randomness and selection. Henceforth, evolution also can solve SAT in linear time. Henceforth, evolution is almighty in the space of problems. Answer*

124 ***SAT is NP-harsh***

The great majority of young people marvel at news about suicides. But as one gets old, ones becomes to agree with Freud that teaches that every person has an instinct for life and also an instinct for dead. According to our own observations, at the root of the instinct of dead resides the recurrent feeling of having battled for a long time without any sensible advance in the solution of main problems. In other words, complexity overwhelms the instinct for life. But how can this happen when it is very clear that solving difficult problems is one of the main fuels of a life to the full? The reason is that life is like the experience suffered by one friend of us:

I was used to swing in the sea when it was brave but one day I suddenly found myself caught in a too strong surf, whose possible appearance was anyway predicted by the experts of the beach. I was recycled by the waves, which pushed me toward the beach to pull me next apart from it not before having smashed me against the bottom of the sea. The little stones at the bottom with frequency hurt my backbone. Through all this process my mind was not void. At the time that I was suffering there, I was able to remember most of my life, as if I were attending my final judgment. Most probably I was at the very hell because breathing problems were terrible. When I arrived to the verified conclusion that God abandoned me, I remembered that someone told me that waves come in a series of nine, from a very small one to a very giant. So, the strategy to leave from the surf is to wait for the giant one and to accompany its crest until the beach. So, I prepared to execute the strategy but the first time I failed. So I suffered for other 8 rounds and made a new try. I succeeded, thanks the Lord, and the sea threw me far away from its returning point. And so, I remained alive.

We all know that storms are good when they are few in number but too many are a burden for life to such a degree that one might prefer to die. That is why some prisoners kill themselves because recurrent tortures are not an optional way of life. We see that complexity + recursiveness form a lethal combination. This is real life. So, we ask: Do mathematicians and computer scientists have included at least one year of real life into their models of complexity?

The next paragraphs to answer that question have mainly been taken from Rodríguez ([62] 2003):

The complexity of a problem could be triggered by an addendum of a simple condition: anyone person can juggle 2 balls in the air but to juggle 3 balls is simply impossible for most people. Or equivalently, a very difficult problem that is weakened somehow may give rise to a simple problem: think of ethical problems. This phenomenon is well known in technical and industrial affairs: a high quality product could be very expensive but a similar product with moderate quality can cost five, ten, one thousand or ten millions times less than the former. The same fact is witnessed in artworks and legal documents with imitations and falsifications. This is why, in applied mathematics the classification of a problem as NP-hard is in general worthless: it is not excluded that a minor and possibly irrelevant change to a NP-hard problem could produce a P problem.

Optimization problems are especially suitable for this discussion: while the exact optimum of a given function under certain restrictions could be not computable, one can devise, in general, heuristics that find a rough approximation almost instantaneously (Hall, [29] 1995). Recently, it has been investigated the hardness of finding good approximations to NP-hard optimization problems in polynomial time. It has been found that there exists such wonderful cases in which a polynomial time algorithm can deliver a solution to a NP-hard problem that is within one unit off the optimum. Such is the case of the NP-hard problem of coloring the edges of a simple graph with the minimum of colors. So, if the minimum number of colors is m, then the algorithm produces in polynomial time a solution with no more than m + 1 colors (Hochbaum, [33] 1995).

But in other cases to deliver a good approximate solution to a NP-hard optimization problem is still a NP-hard problem: these are the NP-hard optimization problems with bad properties of approximation, which can be thought of as the really complex problems (our universal reference: Arora, [5] 1995). We refer to these problems as **NP- harsh** problems. They have been tentatively separated into four different classes according to the degree of approximation that is hard to achieve. More accurately, we define that an algorithm achieves an approximation ratio α for a maximization problem if, for every instance, it produces a solution of value at least OPT/α , where OPT is the value of the optimal solution. For class I, the approximation ratio that is hard is of the form $1+\varepsilon$. For class II, it behaves like $logn$, i.e. of the same order of magnitude as the length of the input. For class III it is of the form $2^{log^{1-\varepsilon}n}$ and for class IV, of the form n^{ε}.

We have seen that SAT is very complex, so much that it defines an all else nothing semantics, which is enough to overshadow evolution. Anyway, let us remind that SAT with one or two variables can be solved efficiently, in polynomial time, but as the number of clauses in a conjunctive normal form (i.e. of restrictions) grows, the problem may get overconstrained and unsatisfiability will be more probably. A phase transition in the fraction of unsatisfiable formulae has been observed at a critical ratio of clauses to variables equal to 1 (Monasson et al., l.c. [51] ,1999). Nevertheless, SAT is NP-complete for a number of variables larger than 2.

A very important class of NP-harsh problems is formed by optimization problems, whose aim is to find a maximum of certain function given a series of restrictions. Optimization problems are similar to climbing a mountain, whose complexity depends on the type and degree of roughness of the landscape. Our first example of optimization problems that are NP-harsh is related with SAT:

Let us recall that we define **K-SAT=Maximum K-satisfiability** as follows: given a conjunctive normal formula, where each clause contains at most K variables, find an assignment that maximizes the number of satisfied clauses. MAX-3-SAT is NP-harsh and belongs to all IV classes: it is at the very center of badness. It has bad properties of approximation even in the case in which each variable occurs in at most 5 clauses and in this case it belongs to classes I, II and III.

With some practice, one can see that all classes of NP-harsh problems have instances that are relevant to any field, be it in science, or in technology or in culture (Rodríguez, l.c., [62] 2003). So, we have proved that our models of complexity predict that complexity in it is steepest version is in the air we breathe. This is specifically certain for design, biology and evolution:

- Design is NP-harsh: one needs to satisfy a series of requirements, i.e. to satisfy SAT. That is why brilliant inventions gain a honorable place in the history of mankind. But to be competitive in modern markets, one must solve problems in the best possible way given some restrictions of time, computer power and budget. So, design generates naturally optimization problems that include SAT and that are therefore NP-harsh.

- The origin of life is NP-harsh: the formation of an auto-catalytic system that functions far from equilibrium encloses the solution of SAT given in the form of restrictions of the form 'do this but not that'. Moreover, the origin of life as we know it must include the enabling of evolution, of genetic programming and of universal computation. So, to propose a theory for the origin of life includes a magical solution of SAT.

- Evolution is NP-harsh: in hindsight, the natural purpose of evolution is to perfect nascent functions that were assembled by randomness or by self-organization or by design (as in genetic engineering) by means of genetic programming. As we have seen from our experience in Vol II, genetic programming seems to be by far more complex than SAT. So, if you doubt that evolution is the correct explanation of your eyes, it is because you have correctly understood what an NP-harsh problem is.

125 *Evolution as a paradigm*

We have seen that one can use evolution to solve the most varied type of problems and one gets the impression that a good proportion of usual problem can be solved with the help of evolution. Which is the exact relation of evolution with the other computing paradigms?

A first result in computing science is that all known paradigms of computing science are roughly equivalent in regard with performance. This conclusion is also valid for evolution through the **Markov algorithms** and **Post Systems** of computer science (Tourlakis, [69] 1984), which can be considered as abstractions of evolution because they also deal with copy-paste grammars over strings.

So, evolution is, for practical effects, as good or as bad as the generality of known solving problem strategies in computing science. In plain words, complexity is transcultural: problems are difficult to solve and there are problems that are very difficult for everybody and whose cheapest solution can be found by scanning all allowed states. Our example is SAT.

126 *Evolution and bugs*

The word **bug** finds no place in evolutionary biology. But we can make a concession and admit the use of that terrible word as a consideration in hindsight. Actually, the word *selection* is also allowed in evolution in the same spirit: there is in nature no agent that picks up individuals, such as it indeed happens in artificial selection of, say, races of horses. The only feature that is present in nature is differential reproduction. By the same token, one can allow the use of the word *bug* in evolutionary biology as follows:

On one hand, the genome is software and evolution is a natural software developer. On the other, some changes in the genome correlate with more ability to survive and reproduce. From above, that looks as if some functions were getting more perfect, more fitted to the environment. Because reproduction awards fitness, the average fitness of the population is

expected to grow over time. This is not an aim, this is not a goal, this is instead an emergent characteristic of the overall diffusive system: in hindsight, evolution has as purpose the perfection of the species, i.e., the increasing of the average fitness which can be traced to the perfection of some functions. Can evolution commit a transgression against its emergent goal of perfecting nascent or running functions? Yes, but the answer depends on the scale of time: at a short scale of time any deleterious mutation is a bug. But at a longer scale of time, even death could be beneficial for the species: death could be very good, say, when a very big population is threatened by extinction because of starvation due to exhaustion of resources. Actually, we cannot imagine complex life without apoptosis, which is cell suicide committed for the sake of the whole organism.

In relation with a short scale of time, natural evolution commits bugs at every moment: from the very fact that mutation cannot be halted and that we see no arising of new species at each pot or valley, we immediately deduce that most mutations are nearly irrelevant else lethal, i.e. bugs against further evolution. Of course, this is well documented in molecular evolutionary biology. But if we flip our view to a longer scale of time, we must consider a rather different vision, which is motivated by a geometric point of view: a geometric landscape can be associated to every optimization problem. An optimization problem is simple when the associated landscape has one or some few peaks and in this case the immediate success leads to the optimum. But a problem is complex when the density of local optima in the associated landscape overburdens the computational power at hand. In a case like that, an immediate success leads to a local optimum that might represent stagnation. Therefore, the final solution for a complex problem always has an ancestor that was unfitted with respect to the leaders of the population. But it is not excluded that what looks as a bug in the short scale of time may give rise the smart solution in a larger scale.

So, **complexity** is what converts immediate success into an evolutionary bug and local bugs into solutions. Gradualism is the not as yet verified postulate that every complex problem always admits an evolutionary reformulation in which immediate success always leads to a global optimum and therefore the problem can be solved step by step. Most possibly, gradualism is false, but, beware, no one knows the covering of those difficult problems that can be converted into tractable ones by a change of insight, by an Ansatz. Henceforth, gradualism never will cease of producing amazing results. Please, trust us: you can be the compositor of various important simplifying insights.

We have witnessed a tension between the short scale of time and a longer one, but we can give a place to both types of considerations if we simple report the distribution of fitness for a given generation: the fittest individuals, those that lead to stagnation, appear at one extreme of the distribution while unfit ones, leading to higher peaks of adaptation, will appear near to the other extreme. In this way, we are in position of giving an answer to the most elementary question that a software designer could give to us: if evolution is a software developer, where are the bugs of evolution?

Our stand about bugs could be summarized as follows:

At its face value, any imperfection is a bug. So, all individuals that are necessary to arrive to the optimal solution are bugged. So, for a given generation, a wide distribution of fitness is a sign of bugginess. That is why we report in the next program the distribution of fitness along an evolutionary branch. But the less fitter individuals are not necessarily the more

bugged: for complex problems an evolutionary branch could begin with them to lead to the optimal solution.

The problem to be solved is MAX-K-SAT, whose goal is to find an assignment that maximizes the number of satisfied clauses for a given conjunctive normal formula, where each clause contains at most K variables. The prob of negation of a variable is 0.5. Random equiprobable truth values are assigned to the variables and each clause is evaluated to true else false. The fitness of a valuation is given by the number of satisfied clauses.

127 *The next code uses evolution to solve MAX-K-SAT and reports the distribution of fitness. Given a proposition, the fitness of a valuation is defined as the number of clauses satisfied by it.*

```
/* Program C127, Bugs.
 *
 * Where are the Bugs of evolution?
 *
 * Given that evolution is a diffusive process in the
 * space of software programs, a bug is in last term
 * a waste of resources that is, in hindsight,
 * of no help to arrive to the front of diffusion.
 *
 * A direct investigation of this problem is proposed
 * as a challenge for our community. For the time being,
 * we try out two abusive simplifying measures:
 *
 * 1) The problem belongs in the realm of diffusive
 * genetic programming, the purposelessly synthesis
 * of software by means of evolution. But it is very
 * expensive. So,  we change diffusion by design and
 * step down from genetic programming to
 * genetic algorithms, the solution of problems
 * by simulated directed evolution.
 *
 * 2) Hindsight criteria must be investigated by
 * back-tracking. To avoid such a daunting task,
 * we contend that   a wide DISTRIbution of fitness
 * is a sign of bugginess. That is why
 * we report the DISTRIbution of fitness along
 * the evolutionary branch.
 *
 * The problem to be solved is  MAX-K-SAT, whose goal is to
 * find an assignment that maximizes the number of
 * satisfied clauses for a  given a conjunctive normal
 * formula, where each clause contains at most K variables
 * chosen among a set with nVar variables.
```

```
 * The prob of negation of a variable is 0.5.
 * Random   equiprobable truth values are assigned
 * to the  variables and each clause is evaluated to true
 * else false. The fitness of a valuation is given by
 * the number of satisfied clauses.
 *
 * Since the fitness can take on intermediate values
 * in within extreme ones, this   setting defines
 * a gradualistic embedding of SAT.
 *
 * A proposition is declared to be unsatisfiable if
 * after 2^(number of Variables) generations no
 * satisfiable assignment has  been found.
 *
 */

package ejvol3v2p;

import java.util.BitSet;
import java.util.Random;

public class Bugs {


    // Turn on  the random generator
    static Random r = new Random();

    //Number of clauses
    private static int nClauses;
    //Number of variables that may appear in the formula,
    // it could be as large as MAXV.
    //Variables will be represented by numbers.
    private static int nVar;
    //Each clause is an OR disjunction of   a sample of
    //KVar variables that could be possibly negated.
    private static int KVar;
    private static final int MAXV = 1000;
    private static final int MAXCLAUSES = 10000;
    private static final boolean CHOSENVAR[][] = new boolean[MAXCLAUSES][MAXV]
    private static final int LISTCHOSEN[][] = new int[MAXCLAUSES][MAXV];
    //Proposition in formation
    private static String a, b, c;

    private static int nProps;

    private static boolean print = false;
```

```
    private static double alpha, nn;
    private static boolean problem = true;
    //if the prop is satisfiable, sat = true else false.
    private static boolean sat = false;

    //A population of binary numbers is considered.
    //The number of individuals is constant
    private static int numbInd;
    private static final int MAXNUMBIND = 10000;
    private static final BitSet IND1[] = new BitSet[MAXNUMBIND];
    private static final BitSet IND2[] = new BitSet[MAXNUMBIND];
    private static int FitnessInd[]
            = new int[MAXNUMBIND];
    private static final int ORDERIND[] = new int[MAXNUMBIND];
    private static int minFitness, maxFitness;
    private static int gen;
    private static int maxGen;
    private static boolean recombination;

    private static double mutationRate;

    private static final int DISTRI[] = new int[8];

    private static void setParameters() {
        //Number of clauses: must be a multiple of 7
        nClauses = 6 * 7;
        //Number of variables
        nVar = 3;
        //Number of variables in a clause
        KVar = 3; //must be less or equal than nVar
        //Number of propositions
        nProps = 1;
        numbInd = 21;
        recombination = true; //true = recombination is enabled
        mutationRate = 0.8;
        nn = nVar;
        alpha = nClauses / nn;
        maxGen = (int) Math.ceil(Math.pow(2, nVar));
        System.out.println("maxGen = " + maxGen);
    }

    //Not operators are inserted with prob 0.5
    private static String negations() {
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                int d = r.nextInt(2);
```

```
                if (d == 0) {
                    a = a.substring(0, j) + "f" + a.substring(j + 1);
                }
            }
        }
        a = a.replace("f", "Nl");
        return a;
    }

    //Composes a random normal disjunctive full l-proposition
    //with nClauses clauses. Each clause has   KVar variables
    //that are sampled from a set with nVar variables.
    //The NOT operator is included for each l
    //with probability 0.5.
    private static String lProposition() {
        //System.out.println("Construction of the l-proposition");
        //Proposition is initialized.
        a = "(l";
        for (int i = 0; i < nClauses - 1; i++) {
            for (int j = 0; j < KVar - 1; j++) {
                a = a.concat("ORl");
            }
            a = a.concat(") A (l");
        }
        for (int j = 0; j < KVar - 1; j++) {
            a = a.concat("ORl");
        }
        a = a.concat(")");
        if (print) {
            System.out.println("l-prop without NOT = " + a);
        }
        negations();
        if (print) {
            System.out.println("l-prop with NOTs   = " + a);
        }
        return a;
    }

    //For each clause , a sample of kVar variables is made
    //among a set of nvar variables
    private static void chooseKVars() {

        // System.out.println("chooseKVars ");
        for (int k = 0; k < nClauses; k++) {
            for (int i = 0; i < nVar; i++) {
                CHOSENVAR[k][i] = false;
```

```
            }
        }
        for (int k = 0; k < nClauses; k++) {
            int n = 0;
            //System.out.println();
            //System.out.println("k = "  + k);
            while (n < KVar) {
                int j = r.nextInt(nVar);
                if (CHOSENVAR[k][j] == false) {
                    CHOSENVAR[k][j] = true;
                    //System.out.print(j + " ");
                    n = n + 1;
                }
            }
        }
        //Chosen variables are listed
        for (int k = 0; k < nClauses; k++) {
            int n = 0;
            for (int i = 0; i < nVar; i++) {
                if (CHOSENVAR[k][i] == true) {
                    LISTCHOSEN[k][n] = i;
                    n = n + 1;
                    // System.out.print(i + " ");
                }
            }
            //System.out.println();
        }
    }

    //Each l of the l-proposition is replaced by one of
    //the variables. All variables appear in each clause.
    //Variables are not letters;
    //they are numbers from 0 to nVar-1;
    private static String proposition(String a) {
        //System.out.println();
        //System.out.println("l-prop --> proposition");
        int u = 0;
        //The sample of k vars is made for each clause;
        chooseKVars();
        int k = 0;
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {

                Integer vari = LISTCHOSEN[k][u];
                String s = vari.toString();
                a = a.substring(0, j) + s + a.substring(j + 1);
```

```
                u = u + 1;
                if (u == KVar) {
                    u = 0;
                    k = k + 1;
                }
            }

            //System.out.println("k = " + k + "u = " + u);
        }
        if (print) {
            System.out.println("prop               = " + a);
        }
        return a;
    }

    private static void printingJob(BitSet s) {
        System.out.println();
        System.out.println("Truth evaluation begins");
        System.out.println("prop = " + b);
        System.out.println("Tested valuation = " + decode(s));
        System.out.print("Each clause has " + KVar);
        System.out.println(" variables chosen among " + nVar);
        System.out.println("Chosen variable = 1, unchosen 0:");
        for (int k = 0; k < nClauses; k++) {
            System.out.print("Clause " + k + ": ");
            for (int i = 0; i < nVar; i++) {
                if (CHOSENVAR[k][i] == true) {
                    System.out.print("1");
                } else {
                    System.out.print("0");
                }
            }

            System.out.println();
        }
        System.out.println("List of chosen variables");
        for (int k = 0; k < nClauses; k++) {
            System.out.print("Clause " + k + ": ");
            for (int i = 0; i < KVar; i++) {
                System.out.print(LISTCHOSEN[k][i] + " ");
            }
            System.out.print("Valuation: ");
            for (int i = 0; i < KVar; i++) {
                System.out.print(s.get(LISTCHOSEN[k][i]));
            }
```

```
            System.out.println();
        }
    }

//The proposition is evaluated
    private static String evaluation(BitSet s) {
        b = c;
        if (print) {
            printingJob(s);
        }
        int u = 0;
        int k = 0;
        for (int j = 0; j < b.length(); j++) {
            if (b.charAt(j) == 'l') {
                if (s.get(LISTCHOSEN[k][u]) == true) {
                    b = b.substring(0, j) + "T" + b.substring(j + 1);
                } else {
                    b = b.substring(0, j) + "F" + b.substring(j + 1);
                }
                u = u + 1;
                if (u == KVar) {
                    u = 0;
                    k = k + 1;
                }
            }
        }
        if (print) {
            System.out.println("Truth value8 = " + b);
        }
        return b;
    }

//The proposition is evaluated until the end
    //Not operators are evaluated at the
    //very beginning for they apply over the next
    //to the right variable.
    private static String simplification(String b) {
        // while ( b.length()>1 )
        boolean progress = true;

        while (progress) {
            String f = "" + b;
            b = b.replace("NT", "F");
            b = b.replace("NF", "T");
            b = b.replace("TORT", "T");
            b = b.replace("TORF", "T");
```

```
            b = b.replace("FORT", "T");
            b = b.replace("FORF", "F");

            b = b.replace("(T)", "T");
            b = b.replace("(F)", "F");

            if (print) {
                System.out.println("Truth value1 = " + b);
            }
            if (b.equals(f)) {
                progress = false;
            }
        }
        return b;
    }

//Converts a BitSet value into a String
    static private String decode(BitSet t) {
        String s = "";
        for (int i = 0; i < nVar; i++) {
            if (t.get(i) == true) {
                s = s + "T";
            } else {
                s = s + "F";
            }
        }
        return s;
    }

    //The DISTRIbution is set to constant zero.
    private static void resetDistri() {
        for (int i = 0; i <= 7; i++) {
            DISTRI[i] = 0;
        }
    }

    //Our pop is formed by valuations.
    //IND1 is a population of binary strings.
    //The BitSet class is used to instantiate
    //binary strings as individuals
    private static void initializePop() {
        BitSet s = new BitSet();
        //IND1 is false everywhere
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) s.clone();
        }
```

```
        //Reinitialization with random truth values.
        for (int i = 0; i < numbInd; i++) {
            for (int j = 0; j < nVar; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
            //System.out.println("IND1 : " + decode(IND1[i]));
        }
    }

// Does an individual   satisfy  the target?
    static private int assessInd(BitSet s) {
        String bb = evaluation(s);
        bb = simplification(bb);
        int fitness = 0;
        for (int i = 0; i < bb.length(); i++) {
            if (bb.charAt(i) == 'T') {
                fitness = fitness + 1;
            }
        }
        if (print) {
            System.out.println("bb      = " + bb);
            System.out.println("fitness = " + fitness
                    + " of " + nClauses);
        }
        return fitness;
    }

    private static void updateDistri() {
        resetDistri();
        double nC = nClauses;
        double den = nC / 7;
        int k;
        for (int i = 0; i < numbInd; i++) {
            double f = FitnessInd[i];
            k = (int) Math.floor(f / den);
            DISTRI[k] = DISTRI[k] + 1;
        }
        for (int i = 0; i <= 7; i++) {
            System.out.print(DISTRI[i] + " ");
        }
        double inf = minFitness;
        inf = inf / nClauses;
        double sup = maxFitness;
```

```
        sup = sup / nClauses;
        System.out.print("Min = " + minFitness + "(" + inf + ")");
        System.out.print(" Max= " + maxFitness + "(" + sup + ")");

        System.out.println();
    }

// Individuals are sorted by fitness.
// the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        int FitnessIndc[];
        FitnessInd = new int[MAXNUMBIND];
        FitnessIndc = new int[MAXNUMBIND];
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
            //System.out.println("Fitness i " + FitnessInd[i]);
        }

        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < numbInd; i++) {
            Champ = 0;
            for (int j = 0; j < numbInd; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array ORDERIND orders ind by fitness.
            //The fittest is number zero
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        //if Pattern has been guessed
        maxFitness = FitnessInd[ORDERIND[0]];
        minFitness = FitnessInd[ORDERIND[numbInd - 1]];
        updateDistri();
        /*
```

```
            for(int j = 0; j< numbInd;j++)
            System.out.println("Order = " +  ORDERIND[j] + " "
                 + FitnessInd[ORDERIND[j]]);
         */
        if (maxFitness == nClauses) {
            problem = false;
            sat = true;
            // System.out.println("Champ = " + IND1[ORDERIND[0]]);
            b = c;
        }

        return problem;
    }

    //The new generation is made
    private static void newGenerationInd() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest
        while (i < numbInd) {
            int counter = numbCopies;
            while ((counter > 0) & (i < numbInd)) {
                //System.out.println(i + "" + l);
                IND2[i] = (BitSet) IND1[ORDERIND[k]].clone();
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int j = 0; j < numbInd; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*  System.out.println( j+ " IND1 "
                                + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(numbInd);
        //b is false everywhere
        BitSet bb = new BitSet();
```

```
        //B is the recombinant of l and k
        for (int j = 0; j < nVar; j++) {
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    bb.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
                    bb.set(j);
                }
            }
        }
        return bb;
    }

// The definitive population is a recombinant one.
    private static void recombinationInd() {
        //IND2 is a copy of IND1
        for (int i = 0; i < numbInd; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //IND2 is a recombinant version of IND1
        for (int l = 0; l < numbInd; l++) {
            IND2[l] = recInd(l);
        }
        //IND2 is transferred to IND1
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }

// Individual  a is mutated at a random place.
    private static BitSet pointMutationInd(BitSet a) {
        int l = r.nextInt(nVar);
        a.flip(l);
        return a;
    }

// Individuals are mutated
    private static void mutationInd() {
        for (int i = 0; i < numbInd; i++) {
            double p = r.nextDouble();
```

```
            if (p < mutationRate) {
                IND1[i] = pointMutationInd(IND1[i]);
            }
        }
    }

    // Given a goal, it is guessed by evolution
    private static void evolution() {

        //The new generation is produced
        //as copies of the fittest
        newGenerationInd();
        if (recombination) {
            recombinationInd();
        }
        //The new population is subjected to mutation
        mutationInd();
        gen++;
        System.out.print("Gen = " + gen + ": ");
        problem = sortingInd();

        if (gen == maxGen) {
            problem = false;
            sat = false;
        }
    }

    public static void report1() {
        System.out.println();
        System.out.println("nClauses = " + nClauses);
        System.out.println("Total N of variables  = " + nVar);
        System.out.println("Variables per clause = " + KVar);
        System.out.println("alpha = " + alpha);
        System.out.println("Repetitions = " + nProps);
        System.out.println();
    }

    /* We generate at random nProps satisfiable props,
   * and report the mean and deviation of the
   * necessary number of individuals to get
   * a satisfying assignment.
     */
    public static void main(String[] args) {
        print = false; //printing instruction
        setParameters();
        double nSat = 0;
```

```
        for (int i = 0; i < nProps; i++) {
            System.out.println("***********");
            System.out.println("Prop numb = " + i);
            System.out.println("Distribution of fitness in 7 "
                    + "classes, " + numbInd + " individuals");
            problem = true;

            problem = true;
            sat = false;
            b = lProposition();
            c = b;
            proposition(b);
            initializePop();
            System.out.print("Gen = " + gen + ": ");
            problem = sortingInd();
            gen = 0;
            while ((problem)) {
                evolution();

            }
            if (sat == true) {
                nSat = nSat + 1;
                System.out.print("Satisfiable ");
            } else {
                System.out.print("Unsatisfiable ");
            }
            System.out.print("MaxFitness = " + maxFitness);
        }
        report1();
        double p = 1 - nSat / nProps;
        System.out.println("Prob of Un-Satisfiability = " + p);

    }
}//End of Program C127, Bugs.
```

128 ***Exercise*** *Run the program and play with the code.*

129 ***Exercise*** *Devise and run some few formal experiments. Write down your conclusions. Answer to the question: where are the bugs of evolution?* *Answer*

130 ***From almost perfection to perfection***

According to the last exercise and in reference to applied evolution, we see that one can at once produce results, whose best exponents differ from perfection in around 1/7 of demanded

requirements. Additionally, some few generations of evolution are sufficient to almost touch perfection. But on the other hand, to produce perfect results, to find a satisfiable valuation, seems to demand an exponential number of trials in the number of variables per clause. Therefore, the SAT version of evolution looks like this: applied evolution wastes resources not to evolve from nothingness to almost perfection but to pass from almost perfection to perfection.

Our results explain why one can differentiate at once the quality of the local industry of developing countries from the usual standards of developed ones (industry also follow an evolutionary process). It is also clear why high technological goods are absent from the production of developing countries. Nevertheless, as the know how diffuses over the world and as appropriate combinatorial bases get less expensive, more countries can compete: this is the Eastern Asian phenomenon.

We can also understand why creationist people are very religious, showing so a great enthusiasm for their Creator: their daily experience with SAT prepares them to belief that the Creator is very smart and powerful and with an exceedingly refined sense of honor because the life we know is the standard and transcultural paradigm for perfection. Nevertheless, we must be aware of the fact that perfection is a concept with a subjective load. So, most people think that life is excel while some few researches have produced arguments pointing to defects of design or to evolution in progress. Let us illustrate this point:

- The human form of communication by means of sound is very primitive as compared with modern multichannel wide band electromagnetic communications. So, some people might conclude that we are under evolution in progress.

- Some forms of rheumatism have an autoimmune origin: a system intended to serve in the very defense of the organism turns into its most acute enemy. So, some people might claim that this illness must be classified as an elementary error of design, a fact that would rule out the belief in a Creator, or at least as a bug of evolution, in accordance with our predictions.

One can see that this type of complains are fair and rest upon the fact that very complex optimization projects can have a lot of dissimilar formulations and solutions. So, we dismiss any further discussion.

131 ***Almost perfection in biology?***

We have no transcultural examples of what imperfect life could be or look like. So, examples must be created with the imagination. The importance of this exercise is the following:

Natural evolution is the synthesis of genetic software as a diffusive process and is committed to the perfection of nascent functions. Given that applied genetic programming is very difficult and that natural evolution involves no magic, we infer that perfection is not less expensive for natural evolution than it is for the applied one. So, a spectrum of diverse degrees of quality must be observed. And this is a mandatory prediction. To fix ideas, let us consider some examples:

1. We have bladder. This is a recipient that allows us to pour out urine at will. Actually, bladder is not necessary: many persons are operated of the kidneys and must temporarily use a plastic tube that is directly connected to the kidneys to evacuate urine.

So, evacuation of urine is necessary but this can be done without bladder by using a conduct that in mammals might be along their tail, when they have one. So, bladder is the brand of perfection. Where is the record fossil indicating that there is evolution from no bladder to a high quality bladder as the one you have?

2. Our eyes provide perfect vision thanks to many concurrent features. One of them is the following: the sensitive tissue of the eye is the retina. But this tissue is highly dissipative and must be fed at every instant. So, it must be irrigated by blood. The problem is that we have a four chamber heart that pushes blood by tremendous impulses. So, one must provide a mechanism to damp the impact of the strikes of the heart over the retina. The solution that we know is to place the retina over a bone with appropriate geometry and to put the retina right down over that bone while vessels navigate over the retina. In that way, the impact created by the heart is not felt by the retina. A problem remains in relation with the optical interference caused by the moving vessels, but this can be neutralized if the eye keeps moving around changing the place of formation of the image. Notice that the octopus has not these anatomic facilities: its retina goes over the vessels, so it must have a good vision that by instants gets very blurry. (A similar problem is faced in bull's eyes practice with guns: the secret consists in detecting the strike of the heart that causes the hand to tremble and to shoot just after a strike). Where is the evolution of the geometry of the eye showing square eyes and the evolution of relative position of the retina and vessels?

3. Slight imperfection in the duplication of the genetic information may cause acute and chronic disease. Example: Glycine is the simplest non-essential amino acid, which is used as a major inhibitory neurotransmitter, whose accumulation causes severe poisoning. Therefore, inborn enzymatic defect in the cleavage system will lead to accumulation of glycine in body fluids, especially those that nurture the nervous system. The effects of the intoxication produce a disease known as non-ketotic hyperglycinaemia. Symptoms typically appear in the first few days of life, with stupor, hypotonia, seizures, multifocal myoclonus, hiccups, and finally episodes of apnoea and coma. Most patients die in the neonatal period or within the first year. The few survivors have hyperreflexia, frequent hiccupping, myoclonic seizures, and severe psychomotor retardation (Hui et al, [35] 2004). Illnesses of this type, due to simple mutations, now are listed by tens in the books of medical interest (Zschocke and Hoffmann, [78] 1999) and with the genome project they will be extended to include some few thousands. That is why we ask: Where are the human populations that have not evolved enough to be as healthy and perfect as we are?

Considered examples show that we can make very easily a point in biology from the existence of perfection that has no left tracks of its evolution. But on the other hand, the evolution of perfection is at the very heart of the evolutionary theory. Let us consider for instance the next observations: the human has muscles in his or her hand that enable writing, the chimp not; the human has vision in three fundamental colors, the chimp in two; the human can speak, the chimp not; the human has cultural evolution, the chimp, not; the human has vessels that irrigate its face, which warm him or her in time of cold, the chimp not: as a consequence, the human can migrate to cold regions of the planet, the chimp not. The interpretation of these facts at their face value that is made by the evolutionary theory

is clear: there is an evolutionary history from the almost perfection of the chimp to the perfection in the human being.

The problem is that the chimp is also very perfect: its bladder is as good as that of a human. And its brain, eyes and hand allow it to grasp and throw a stone to threaten lions. Where is the evolution of the perfection of the chimp? In general, the evolution of primates is traced back to lemurs, which are also very perfect: just give a look at their bladder, it is elastic and perfectly impermeable, it is contractile, it is bounded with regulated sphincters, it is ergonomic to the changing size of the animal, it can be evacuated at will and its maintenance is automatic.

We have now two points of view that are contrary one to another. The first says that the evolution of perfection exists and is the proof of evolution, the second argues that the evolution of perfection is mandatory and half cooked products must be found along every evolutionary branch, a prediction that is clearly false: everything we know about biology is that living beings are the epitome of perfection.

To correctly understand these two positions, let us depart from a fundamental premise: natural evolution contains no magic. Therefore, with the due care, one can infer how natural evolution behaves knowing our experience with every kind of artificial or simulated evolutions.

132 *Bugs and laws of software design*

Let us summarize our experience with bugs in the next descriptive **laws of software design**, which express that the most direct experience of every designer is his or her battle against bugs:

1. We do not know how to produce software without bugs. We postulate that committing bugs is the cheapest way of producing software.

2. Because bugs are unavoidable, they must be corrected. But in general, when one tries to correct a bug, other bugs appear while there are bugs that are very difficult to correct. That is why the industry releases beta versions of their products to identify the more frequent bugs to next work hard in their correction.

3. Because bugs are immanent to design and correcting bugs generate more bugs, any developing path is very long and is always plagued with half-cooked, dysfunctional products. Nevertheless, every developer likes to do his or her best to offer clean, terminated products. So, a developer possibly advances everyday but releases products rather from time to time.

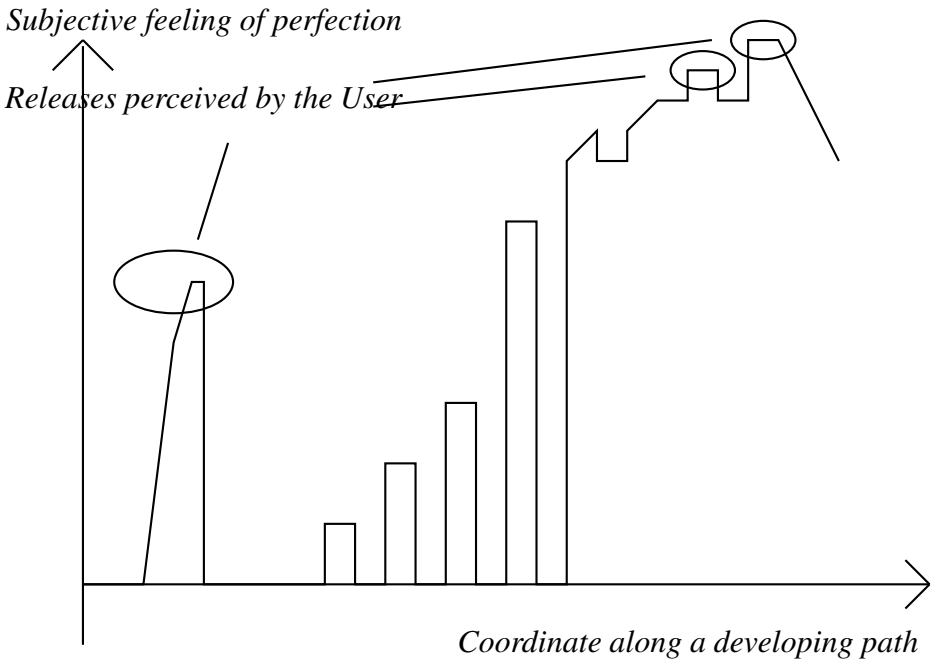


Figure 5.3. Evolution of software is mandatory but the perception of this process is different for the Developer Team (broken line) than for the User (Elliptical enclosers). For the Developer team, perfection is only statistical and begins to appear after a very tortuous path that includes the correction of many bugs with varying degrees of fatality. On the other side, the User sees an evolution that goes from not a bad product to a good project to a better one to excellent quality. When projects are very complex, with every kind of multifunctionality, the User also shares the feeling of dissatisfaction caused by half cooked products.

4. The whole phenomenon includes two perspectives of the developing process: one by the User, the other by the developer. The User sees an evolution from a success to a greater success. The developer sees a developing path that is filled in every kind of pitfalls but with an increasing envelope. We reported only 3 intermediate states in the developing process of our painting application, but a complete report may include more than 100 viable programs.

5. This declaration forms our basic laws of software design and we postulate them to be valid for both rational and irrational developers.

For instance, Windows is updated on a daily schedule and every updating produces a better functional version. But some changes are so drastic that updating becomes impossible and one must migrate to a new platform. But and because of complexity, migration is not guaranteed to be accompanied with a gain in satisfaction. This happened with the migration from from Windows XP to Windows Vista as well as with some versions of Linux and happens everywhere at every time.

133 ***Evolution and bugs***

Our laws that link the fundamental laws of software design with bugs are valid for human developers. But not for evolution because a bug is a fault against the purpose of a project and evolution has no purpose at all, so evolution cannot commit bugs.

Nevertheless, evolution has an emergent purpose, which is to convert properties into functions and functions into more excel functions that could help in the struggle for life. From a geometrical stand point, evolution has as purpose to explore the landscape of functions and to climb the highest peaks, those that represent the highest fitness. So, a bug is simply a step that diminishes the fitness of the population.

Now, the description of the natural evolutionary process that is made by the international literature corresponds to the description of industrial software as it is made by the Users. We need just one correspondence: release $\leftrightarrow$ species.

Nevertheless, our comparison is not completely fair. The quality of industrial software, even of that produced by the best companies, is very low as compared with that of the natural one. To get a feel of the difference, imagine all the people that take care of the industrial plants of a city of regular size. But by contrast, just a single mother is enough to take care of her son, in whose cells go reactions to fabricate more than 3000 products. And some effort by the mother is most of times more than enough: according to the ancestral Russian tradition, a healthy baby to sleep happily must be clean, full and dry. There are two things to celebrate. The first is the capacity of synthesis of that culture and the second is that this simple receipt functions: this culture has more than 1300 years of existence and has all signs of permanence for a long future.

So, the elementary question that we must formulate is the following one: where is the description of the natural evolutionary process in which we go from errors to corrections to more errors to less errors to relative perfection to excellence?

We do not see such a description anywhere in the literature. Neither in nature.

How can we explain this deficit of registered imperfection that is mandatory?

Mankind has been trying out since long ago mainly two solutions:

1. To believe that life was made by God the Creator. In fact, we claim that we have found a rationalization to the usual and transcultural belief that the extreme complexity and perfection of critters is a symbol of a Creator. A mandatory prediction easily follows: creationist religions will remain forever, a conclusion that now looks completely normal in the field of the anthropology of religions (Boyer, [11] 2008).

2. To adjudicate to evolution magical powers such as it is doing by all the international literature up to this very day.

5.7 Summary on complexity vs evolution

The relation of evolution vs complexity can be summarized as follows:

- In relation with performance, evolution lacks any magics: it is equivalent to all known paradigms of computing science.

- In spite of terrible efforts, we do not know how to solve anyone of the generality of NP-complete problems in less than exponential number of trials (in the length of the input). Example: SAT.
- NP>P with probability 1, a result that explains the fact that the experience of the Academic Community is that NP-complete problems demand an exponential number of trials and that predicts that so will be your own experience.
- Evolution has as purpose the perfection of nascent functions, which is an optimization problem. The complexity of those problems is studied by the theory of approximation of NP optimization problems and it is clear that there are some problems of this type that, to the best of current knowledge, demand an exponential number of trials to be solved while approximate, imperfect solutions can be found sometimes immediately. Example: MAX-K-SAT.
- We have as yet not a reason to think that the complexity of the origin of life or of species is not represented by NP-complete or more complex problems.
- Evolution is pray of sampling effects that are magnified by selection, so if one needs to catch events of very low probability, evolution in large populations is not a suitable tool to be used. Instead, one can use a random walk and get results in less time and with less programming effort although wasting an exponential number of trials.
- Evolution is by essence open ended but the very existence of species is a clear proof that the computing power of biological evolution is in its present state not sufficiently high to overcome stagnation.
- A mandatory prediction of evolution is that there must exist an evolution of perfection that must left behind visible and perdurable traces through populations and across the fossil record. This prediction fails.
- The natural translation of the gradualism of Darwin to the realm of computing science is that there must exists a simple and universal method to convert complex problems into simple ones. Without gradualism evolution loses its strength against falsification. The most simple attempt of putting gradualism into action is that of proving that NP =P. But we have failed. Given that the quest for gradualism has not provided fruits to solve complexity, must it be abandoned? By no means: we as yet do not know what very important problems that seem to be very complex can be converted into simple ones through a change of insight. So, it is good for everybody to train him or herself to create new and revealing points of view. Therefore, we have a command for everyone: keep creating!

134 *Complexity in a nutshell*

One may learn form experts in the field of the origin of life that this is a very complex problem. But our study of SAT reveals that that is not necessarily so: logical propositions generated at random can be easily satisfied in a good proportion of its clauses. This seems to say that the properties of random systems might afford to incredible wonders. By the same token, one can consider that a dead organism is in within an ace of being alive. These

occurrences show that SAT is an important object of study but as a model of complexity, it is probably too simple. Given this warning, let us consider now to try to rationalize why complexity defeats evolution and to such a degree that evolution ends behaving worst than randomness.

In fact, we have seen various simulations in which evolution solves quite swiftly a problem that is very complex for randomness. But we also have seen cases in which a simulation of evolution solves a problem wasting more resources than randomness. These experiences do not allow us to claim as yet that this implies that evolution is worst than randomness because a certain number of simulations of evolution do not represent the potentialities of evolution to the full. If such a claim can be done, that must be concluded from theoretical considerations.

We lack such a proof. And if such a proof appears some day, please, remember that there are still thousands of very important problems that can be converted into simple ones by a change of insight. Anyway, by looking at the generalized difficulty with which we solve NP-complete problems (Hochbaum, l.c., [33] 1995) or NP-harsh optimization problems, we can ask ourselves: what is the ultimate reason of this generalized experimental failure?

We propose the next explanation that compares a solution to simple problems vs the solution to complex ones.

A simple optimization problem looks like climbing a mountain with one single peak:

You can begin from whatever place, test some 6 values in the close vicinity and choose the better. Next, repeat this procedure but sample preferentially the established growing direction. Stop when any move diminishes the value of the assessed function. To gain more precision, begin a new search from the place you are but using very tiny steps. Once you are done, you could notice that your entire walk was executed as if you were following a compass that guided you to the cusp.

By contrast, solving a complex problem looks like climbing an artificial landscape that is very vast and entirely composed of needles of random height. To conquer one needle is of no help to find the global optimum. And randomness, as in a random walk, could be more appropriate than an evolutionary attack. Why?

Suppose we have an individual that classify as the fittest around a local optimum. If it does not disappear by mere chance, it rather swiftly will leave too many representatives in the population. By contrast, an individual with a low fitness is weak against drift, the possibility to disappear by mere chance, and has no selection to defend it. The net consequence is that in large populations selection works in favor of stagnation and henceforth the evolutionary branches that might be begun with those individuals of low fitness is interdicted: stagnation is produced at mediocre solutions.

By the way, given that randomness is faster than evolution to solve a very complex problem, is randomness the explanation of complexity? Not, it is not: randomness can do it better than evolution but anyway wasting an exponential number of trials, a possibility that is interdicted by cosmological constrains of space and time, i.e. by relativity. So, randomness could come as a solution for simple problems in the form of good luck, but complex problems on the contrary sweep away good luck to give place to the law of large numbers, i.e. to an

exponential demanding trend of determinism.

5.8 Conclusion

A property is a measure of the reaction of a system to a given stimulus. A function is a property that helps the sustaining of life and evolution. Functions can be described by logical propositions while evolution relies on diffusion over satisfiable propositions that can be evaluated to true. That is why we studied the **SAT-isfiablity** problem of mathematical logic. Evolution as well as randomness can solve SAT but evolution does it worst. Nevertheless both work in practice under an exponential regime. We found two channels that enable two distinct types of evolution: the first is working in the perfection of a given extant function and the second is tinkering with new properties. Both options are filled in complexity, but the second channel is highly more probably than the first and so the excel perfection of actual life looks like a mystery. So, our study of SAT predicts that the evolution of perfection is mandatory with abundant tracks across populations and in the fossil record. This prediction is not corroborated by nature.

Chapter 6

Conclusions

`Java for the study of evolution is powerful to make science but evolution without rigor is a fairy tale as it is in the International Literature:`

1. We studied in Vol II of this series a proposal that the genome is **software** and evolution is a software **developer**. In that regard, we were able to tame evolution to produce some pieces of software but our work allowed us to appreciate at once the burden of complexity: a lot of time is necessary to achieve rather humble results. So, we were unable to see the marvels expected from evolution. Anyway, **genetic programming**, the production of software with the help of evolution, is a flowering discipline that already has produced interesting and important results (genetic-programming.org, [27] 2). In spite of our frustration, we found ourselves involved in delicate questions about the origin of the genetic code: our work predicted that we must observe a rich variability of genetic codes, but this prediction is not backed by nature: we have a **genetic code** with minor variations that can be explained by descend and that do not correspond to the expected high level of diversity. This discrepancy is matter of current investigation (Vetsigian et al, [72] 2006), and it is enough to consider our new insight of evolution as software developer as a precious tool to generate science, a fact that we promised to illustrate in other forms in the present volume. Our first step was to discuss the next objection: If evolution is a software developer, where are the *bugs* of evolution?

2. The answer to the question about the bugs of evolution is immediate: *evolution is not teleological*, i.e., evolution has no purpose at all, henceforth evolution cannot commit errors of design. The lack of goals is apparent if we present evolution as a diffusive process no more mysterious than the process of diffusion observed by ink through a paper. Evolution has two means to diffuse: mutation and recombination. To try to illustrate this insight, we viewed two programs to generate morphological diversity, the biomorphs of Dawkins and an original implementation of an L-grammar. Nevertheless, our illustrations suffer some deficiencies that put them far away from evolutionary biology: to begin with, our simulations are systematically directed to create variability, so it reflects more a spirit of design than the achievements of purposeless evolution. In second place, our programs implement very sophisticated evolutionary environments, whose creation is due to highly trained people that live in a highly developed culture. In third place, our selection was not an emergent feature of the environment but an

activity of the person that runs the programs. Thus, our models are perfect for **creationism**. Most possibly, the only way to escape from complaints and accusations of this sort is to devise a dynamical system in which evolution could arise from nothing to see what it gives from itself. Henceforth, we cannot save forever the study of the question of the origin of life, whose complexity and perfection overwhelms us.

3. We are now in position of making a strong warning:

   The world is filled in applications of evolution in which it is given a purpose from above, say, to improve an extant design with a prescribed function. This realm has nothing to do with natural evolution, which has no purpose at all. Nevertheless and as it was illustrated in chapter I, evolution can achieve wonderful results by mere diffusion. Now, that process can be filtered, be by a human being in artificial selection or by the environment when a change favors in nature the reproductive power if its carrier. An important point arises: the incidence of that change over reproduction assigns in both cases, natural and artificial, a function to the mutant structure. Say, some ideas are continuously cited and explored further. We say that they generate science. In hindsight we can conclude that ideas have the function of enliven science because of their insight, clarity and elegance. We observe that evolution acquires as an emergent purpose, as a purpose ascribed in hindsight, the quest for perfecting the successful function, the function tied to reproductive success. And it may happen that evolution needs many turns, many generations, to achieve a result that merits to be appreciated. This means that we can say in hindsight that there were failures, blind alleys, or, given that the genome is software, bugs. We see that, in the light of complexity, in the light of the extreme complexity of designing software, a mandatory prediction of the Evolutionary Theory of the origin of species is that all living beings must be filled in malfunctions and that the fossil record must be very rich in every kind of malformations. This is obviously not so and therefore the Evolutionary Theory is false.

4. It is a transcultural fact that we perceive by instinct that living beings are marvelous. This implies that the software contained in their genome is also marvelous. But, what does this mean? This means that the genome is software and as such can be compared with the software produced by humans. In this regard, the natural genome classifies as a proficient, very large and complex software project, it is the epitome of evolution and has a very clean, transparent and organized style. Thus, in terms of software engineering, it is a marvel. To explain why the genome is a representative of a transparent style, we were led to consider the nice speculation that a transparent style emerges in an evolutionary environment in which evolvability leading to very high complexity is rewarded. On the other hand, evolution, apart from being a process, is also a style of software design that is very modern and with a great future. So, **the biogenesis style of software design** is characterized by the next directives:

   (a) If the client request the design of the software encoding for a human being, make a great plan, an architecture, for encoding a great god putting apart a path from simple to complex to more complex to final product in such a way that at each step, but specially in early builds, a great span for diversity is guaranteed.

(b) Next, fill in the details, beginning from the simpler and continue filling in the intermediate stages.

(c) But, please, try not to construct things from the pure beginning, instead, use and reuse software as much as possible.

(d) More to the point, before embarking in great projects, compose a great library of tested methods, programs and classes in various hierarchies.

5. When facing the terrible question of the origin of evolution, one immediate answer is: life with its evolutionary potentials appeared by mere chance. More elaborated answers has been proposed and in this regard, we review some few of the ideas of Kauffman ([38], 1995): life, in all its splendor and evolutionary possibilities, arouse not by chance but by necessity. The reason is that there are concrete mechanisms plus initial conditions in which life and evolution are expected to emerge with a great probability. That is why he claims that we are *at home in the universe*. His model predicts the necessity of the arising of auto-catalytic chemical network that function far away from equilibrium. This beautiful model has two problems, first, it is self-clogging and second, it has the following prediction: so abundant mix of complex compounds as the one demanded there and that moreover includes catalysis and dissipation should had leaf clear tracks in the geological record such as those left by a burnt meal in the pot where it was cooked. But no one has ever found such a trace. On the other hand, the model assumes the unrealizable possibility that DNA is kept untouched by the chemical milieu. So the model is plainly scientific, falsifiable. In fact, it is false. In general, we consider that the problem of the origin of life is unsolved by modern science.

6. We have found that a direct investigation of very important problems surpasses the capabilities of a modern desktop. So, we can help ourselves with the study of suitable models. That is why we present linguistics as a very good setting to study every question about evolution. In linguistics, a string makes sense when it is 'understood' by the interlocutor. In genetics, a DNA string makes sense when it encodes for a viable organism. In genetics as in any human language, the strings that make sense are surrounded by a sea of nonsense, because of which most mutations are deleterious and some are lethal. Memory and functional restrictions, plus recursiveness and self-correction are properties that seem to be sufficient cause of the nonsense sea as in linguistics as in genetics. When one makes a random walk, the sea of nonsense is seem as the sudden lost of viability, a property that is also observed by Java programs: we say that we have a semantic discontinuity. We have confronted evolution against an increasing level of semantic discontinuity and we have found that after a hard battle, evolution recedes. We have also found that a high level of semantic discontinuity may cause evolution to behave worst than a random walk in comparison with the amount of needed resources (the global number of individuals). We also have managed to get periods of ecstasy as long as desired.

7. Our study of SAT allows to classify complexity in two types:

(a) All-else-nothing tasks, as matching a proposition like $p_1 \wedge p_2 \wedge p_3 \wedge .... \wedge p_n$ and that represents a craft that fills a function with many constraints and that does not

function if someone is not fulfilled. For such tasks evolution is outperformed by chance, which in its turn needs an exponential number of trials to solve the task. The fossil record predicted by evolution must look like sudden perfection that is preceded by myriads of false starts.

(b) Tasks related to robust functions. These functions are represented by weakening of the all-else-nothing propositions, say when other channels are allowed as in $(p_1 \wedge p_2 \wedge p_3 \wedge .... \wedge p_n) \vee (\sim p_1 \wedge p_2 \wedge p_3 \wedge .... \wedge p_n)$ or when singular conditions are given a relief as in $((p_1 \vee q_1) \wedge p_2 \wedge p_3 \wedge .... \wedge p_n)$. Randomness is very good to make an almost perfect matching to these functions and evolutions adds very little. So, if one sees in the fossil record that some organism was near to perfection and in some upper geological sheet that it evolved into perfection, then one can be certain that we have here a case of genuine evolution.

8. Linguistics lends itself to a partial rigorous formalization while observing at the same time a direct relation with the mechanisms of life. To show this, we begin with a distinction between two concepts: property and function. A property is a measure of the reaction of a system to a given stimulus. A function is a property that helps the sustaining of life and evolution. Properties can be described by logical propositions while evolution relies on diffusion over satisfiable propositions that can be evaluated to true. That is why we studied the **SAT-isfiablity** problem of mathematical logic. Evolution as well as randomness can solve SAT but evolution does it worst. Nevertheless both work in practice under an exponential regime. We found two channels to solve SAT that enable two distinct types of evolution: the first is working in the perfection of a given extant function and the second is tinkering with new properties. Both options are filled in complexity, but the second channel is highly more probably than the first and so the excel perfection of actual life looks like a mystery. Moreover, our study of SAT together with our knowledge about software design predicts that the evolution of perfection is mandatory with abundant tracks across populations and in the fossil record. This prediction is not corroborated by nature.

9. We have gained some practice in converting feelings and fuzzy ideas into concrete questions that generate simulations and specific discussions. This could be the most important pedagogical fruit of our work.

Chapter 7

Glossary

Algorithm An ordered set of instructions for solving a problem.

Alphabet A set of letters or sounds (phonemes). In nature, the fundamental alphabet is $\{A,T,C,G\}$. In genetic programming, the letters correspond to Java methods. In most applications, alphabets are useless because of combinatorial explosions, so one prefers super-alphabets, whose letters are composed of words built with letters of the alphabet.

Application program A computer program to fulfill a set of functions.

Autocatalytic system A self-maintaining and self-reproducing metabolism. By physical reasons, it is more easily imagined as consisting of an autocatalytic network that functions inside a compartment.

AWT (Abstract Window Toolkit) The initial facility of Java to make GUIs (Graphic User Interfaces). It has been superseded by Swing, which is completely based on OOP.

Binary The quality of having only two possibilities.

Binary code A message written in an alphabet consistent of 0 and 1. The rules to decipher such a message.

Biocomputation Computation done by biological processes.

Bioinformatics The study of how information is represented and transmitted in biological systems.

Bit A quantity and measure of information that corresponds to a yes else not answer.

Boolean Variables that can take value true else false. Boole is the mathematician that studied these variables and corresponding functions.

Boolean operators The mathematical operators AND, OR, and NOT.

Bug Error of programming committed by a software designer against his or her determined goal. Misbehavior of a program. Wasted resources that in hindsight were useless to find the solution to a problem.

Catalysis The selective acceleration of the velocity of a chemical reaction by substances that after the reaction are left intact.

Code A portion of a program.

Combinatorial basis Is an alphabet for an evolutionary process.

Combinatorial explosion The generation of so many possibilities to test that a life would not be enough. Such an explosion is evidenced because if one augments the length of the input, the number of possibilities augments in an exponential form.

Complex The quality of an object that cannot be described with few words.

Compiler A program that translates a portion of code written in Java to a machine-language program, which can be understood by the CPU.

Computer program A set of instructions that tells a computer which operations to perform.

Computer A machine that with the appropriate instructions can run programs.

Context dependence The property of a grammar of having rules of irreducible arbitrary length for formation of new words.

Creationism The belief claiming that life was created by a supernatural spirit, God. Some variants change God by aliens. For Judaism, the creation is important because *we were created by G-d and for His Glory, hence He has the effective right to assess, weight and judge us*. Leading scientists have a tendency to be completely atheistic. Since mutation, the core of evolution, is an undeniable reality, believers must combine evolution and faith to explain their very existence, but people largely differ among themselves by the form as they do that. The personal directive of the Author is: if evolution explains me, no god must be invoked.

CPU (Central processing unit) The brain of the computer.

Datum The record of an observation.

Debugger A program that helps the developer to find bugs.

Debugging The process of cleaning bugs from a program.

Design To conceive in the mind. To have as a goal or purpose. To plan by making a preliminary sketch, outline, or drawing. To create or execute in an artistic or highly skilled manner(Webster's II Dictionary). We restrict the application of design to actors with free will.

Designer A rational entity that is committed to design.

Developer Any entity, rational or not, that is committed to the fabrication of something. Evolution is a software developer.

Diffusion Penetration of one substance into another, say, ink into a paper. In evolution, the diverse recombinations of bases diffuse into the space of viable genes.

Digital Class of systems that are based on all or nothing outputs.

Distributed The quality of being partitioned in pieces.

Environment The world outside the studied system.

Epistemology The division of philosophy that studies the nature and origin of knowledge.

eJGrid The Grid of the evolJava community.

Evolution In epistemology, it is the scientific project whose aim is to prove that variations along living beings can be explained by a natural mechanism from just some few origins or maybe from one or maybe from no origin at all. In evolutionary biology, it is the change along lineages that add together to give rise to the origin of new species and phyla. In the genome project, evolution is the union of all sort of factors that affect genomes of individuals or of populations, be it mutations from parents to offspring or changes from a germ cell to a somatic one or transfer of genetic information from an organism to another. In sociobiology, it is the fundamental force that programs instincts.

Evolutionary environment It is an ambient of work composed of an alphabet plus a genetic code for interpretation plus a form to try out diverse super-alphabets is called an . If

one has an evolutionary environment plus infinite computing power and time, evolution can be run automatically.

EvolJava community The community of all persons that currently use Java as a precious tool to study own and foreign ideas about evolution. Since our task surpasses the possibilities of one single person, our community will soon get organized. Please, visit http://www.evoljava.com or write to the Author: jose@evoljava.com

Evolutionary cycle of software The process suffered by any piece of software: it evolves as much as it can, first rapidly, then slowly, then evolution is halted and then, most of times, the software disappears from the trade. This happens in software engineering as in artificial evolution of microorganisms.

Execute To perform instructions.

Failure An inability to perform the assigned function.

Genetic algorithm A simulation of evolution in which the encoding and rewarding selection is used to solve a given problem.

Genetic code The correspondence among codons and aminoacids. In genetic programming, it is a corresponding among strings generated at random or mutation or recombination and actions to be taken and that appear in suitable methods, programs or libraries of programs.

Genetic programming The use of evolution to design software.

Grammar The system of rules of a given language, whose mechanical application generates all possible sentences in that language.

Graph Couple (V,E) where V is a set of vertexes or nodes and E is a set of edges or links among nodes. The information of a graph is located in a boolean matrix. The vertexes are called $0,1,....,n-1$, and the matrix has a *true* else a *false* at the entry (i,j) if and only if there is a link from node i to node j. So, a link could denote a direction of the interrelation. If one does not want any direction, one might use a symmetric matrix: if (i,j) is *true*, then (j,i) is also *true*.

Grid The union of many desktop computers in a single structure to exploit the potentialities of parallel computation.

GUIs Graphic User interfaces that allow mouse operation.

Information theory The science of communication.

Landscape A function that to each possible structure that eventually could participate in an evolutionary race assigns a real number interpreted as fitness and that determines the rate of differential reproduction of the structures. If the set of structures could be embedded into a plane, the resulting image is that of a mountain or landscape.

Logical AND Boolean function that answer true when both inputs are true and false otherwise.

Logical OR Boolean function that answer true when at least one of the inputs is true and false when none is true.

Machine language The set of instructions in binary code that are understood and executed by a computer processor.

Model An abstraction of some part of the world.

Modeling The art of making models.

OOP Object Oriented Programming, an style of programming in which objects and not functions are the focus of the programming. In that way, programming is converted in a

simulation of reality. Its importance resides in that one can extrapolate the common sense acquired in reality to easily walk over that artificial world created in each program.

Purpose The object toward which one strives or for which something exists: Goal. A desired or intended result or effect. Determination: resolution (Webster's II Dictionary).

Recursion The use of building blocks to build bigger building blocks.

Recombination The act of posing together some pieces to possibly fulfill a function.

SAT or the satisfiability problem of mathematical logic consists in studying the assignment of truth values to the variables of a proposition with the purpose of making it true. SAT proposed a question in regard with a given proposition: Does this proposition is satisfiable? To answer YES you must show a satisfying assignment (one that makes the proposition true) otherwise you must prove that no assignment evaluates to true.

Scheme A DNA pattern. In general, it is word of a grammar.

Selection The connection of the changes of the environment with those in the genome or in the frequencies of their representatives.

Semantics The study or science of meaning in language forms, esp. with regard with historical changes. The study of the relation between sign and symbols and what they represent to the interpreters (Webster's II Dictionary). In evolution and software developing, semantics corresponds to viability: a program that produces an output, say, an alive being, is viable and acquires a meaning.

SOFT The problem of developing software. The developer could be a conscious being, like a human, or an unconscious process, like evolution. If the conscious being has a purpose, we speak of design. The evolutionary theory reads: evolution is the developer that is responsible for the existence of the genome, the natural software. Evolution has no purposes, so it is not a designer, but acquires an emergent purpose: to gain more and more surviving capability, so evolution has a purpose and its developed software must comply with a function: augment differential surviving and reproduction. Therefore, all what human designers know about software design is also valid for natural evolution.

Software discontinuity or **chaotic nature of software**: the property of software of producing great changes in the output after tiny changes in the code or, sometimes, in the input data.

Style Behavior characterized by the recurrence of the same set of not compelling options taken from a great ensemble. A possible style of programming could be to pose together those units that have functional dependence while the contrary style is to ignore any relation between functionality and order in the code.

Syntax is the part of grammar that deals with the formation of phrases and sentences as combination of words.

Systematics The identification of unique characteristics that are shared by certain organisms and not others. From this, it is found who is most closely related to whom.

Swing The package that represents the modern way of making graphic User interfaces in Java.

Teleologic Adjectivized form of teleology.

Teleology The philosophical study of design or purpose in natural phenomena. The use of ultimate purpose or design as a means of explaining natural phenomena (Webster's II Dictionary).

Thread A specific sequencing of execution. Java can run many threads at the same time.

Universal computation The property enjoyed by those computers that can compute whatever is computable.

Chapter 8

Answers to exercises

Problems of Chapter 2

28, page 109. The model 5-2 is sometimes more rugged than the others two previous models. This was appreciated with a graphic made in LibreOffice after pasting output data.

```
/* Program C28, NKmodel2
*
* We study the NK model proposed by Kauffman(1998).
* In this program we propose a comparison of
* the N0 and N1 models in regard with ruggedness.
* So, we report ups and downs along  a random walk.
*
*
* We interpret each string as a program,
* say, DSFVR, represents an execution in which method
* R goes first, followed by method  V and so on.
* The  problem for optimization is to find a program that fits
* a certain specific function. That is not our problem here:
* we pay attention to evolvability, so,
* we study the ups and downs of the
* fitness function along a random walk along
* the space of programs, i.e., of strings with
* the same length.
* The N0 models proposes a fitness for each char
* and the fitness of a string is the sum of the
* fitnesses of its chars.
* In the N1-model the subsequences
* of two chars are assigned a fitness from which the fitness
* of every  string can be calculated. Example:
* f(00) = 1, f(01) = 2, f(10) = 1, f(11) = 0.
* So, the fitness of 10101 is, by definition equal to
```

```
 * f(10) + f(01) + f(10)+ f(01) = 1 + 2 + 1 + 2 = 6.
 *

 * We add here a 5-2 model.
 *

 */
package ejvol3v2p;

import java.util.HashMap;
import java.util.Map;
import java.util.Random;

public class NKmodel2 {


    // Turn on   the random generator
    static Random r = new Random();

    //N = number of methods per program
    private static final int N = 5;

    //Maximum number of steps in the random walk
    private static final int NSTEPS = 500;

    private static  Map fitnessTable50;
    private static  Map fitnessTable51;
    private static  Map fitnessTable52;

    private static final boolean TEST = false;


    //A fitness table is initialized for the model 5-0
    private static void initializeTable50() {
        fitnessTable50 = new HashMap();
        fitnessTable50.put("0", 1. );
        fitnessTable50.put("1", 2. );
    }


    //The fitness is the sum of individual fitnesses.
    //Strings are closed
    private static double fitness50(String s)
    {
        double f = 0;
        String d ;
```

```
    for(int i = 0; i < N-2; i++)
    {
        d = s.substring(i,i+1);
        f = f + (double) fitnessTable50.get(d);
        if (TEST)
            System.out.println("d = " + d + " -> "
            +      (double) fitnessTable50.get(d));   }
    return f;
}

private static void initializeTable51() {

    fitnessTable51 = new HashMap();
    fitnessTable51.put("00", 1. );
    fitnessTable51.put("01", 1.5 );
    fitnessTable51.put("10", 1.5 );
    fitnessTable51.put("11", 2. );

}


private static void initializeTable511() {

    fitnessTable51 = new HashMap();
    fitnessTable51.put("00", r.nextInt(16) );
    fitnessTable51.put("01", r.nextInt(16) );
    fitnessTable51.put("10", r.nextInt(16));
    fitnessTable51.put("11", r.nextInt(16));

}

private static String newStart() {
    String a = "";
    for (int m = 0; m < N; m++) {
        int l = r.nextInt(2);
        a = a + l;
    }
    return a;
}




//The fitness of string s
//is the sum of 2-cluster fitnesses
//Strings are closed.
```

```
private static double fitness51(String s)
{
    int n = s.length();
    //System.out.println("\ns = " + s  );
    double f = 0;
    //Circularizing
    s = s + s.substring(0,2);
    String d ;
    for(int i = 0; i < n; i++)
    {
        d = s.substring(i,i+2);
        //System.out.print("i = " + i + " d = " + d + " -> "  );
        f = f +  (int) fitnessTable51.get(d);
        //System.out.println( (double) fitnessTable51.get(d));
        if (TEST)
            System.out.println("d = " + d + " -> "
            +      (double) fitnessTable51.get(d));

    }
    //f = f/2;
    return f;
}


private static void initializeTable52() {

    fitnessTable52 = new HashMap();
    fitnessTable52.put("000", fitness51("000") );
    fitnessTable52.put("001", fitness51("001") );
    fitnessTable52.put("010", fitness51("010") );
    fitnessTable52.put("011", fitness51("011") );
    fitnessTable52.put("100", fitness51("100") );
    fitnessTable52.put("101", fitness51("101") );
    fitnessTable52.put("110", fitness51("110") );
    fitnessTable52.put("111", fitness51("111") );

}

private static void initializeTable522() {

    fitnessTable52 = new HashMap();
    fitnessTable52.put("000", r.nextInt(16) );
    fitnessTable52.put("001", r.nextInt(16) );
    fitnessTable52.put("010", r.nextInt(16) );
    fitnessTable52.put("011", r.nextInt(16) );
    fitnessTable52.put("100", r.nextInt(16) );
```

```
    fitnessTable52.put("101", r.nextInt(16) );
    fitnessTable52.put("110", r.nextInt(16) );
    fitnessTable52.put("111", r.nextInt(16) );

}

//The fitness is the sum of 3-cluster fitnesses.
//The fitness of a 3-substring is the sum
//of the fitness of 2-substrings of the N-1 model.
//Strings are closed.
private static double fitness52(String s)
{
    double f = 0;
    int n = s.length();
    s = s + s.substring(0,3);
    String d ;
    for(int i = 0; i < n; i++)
    {
        d = s.substring(i,i+3);
        f = f +  (int) fitnessTable52.get(d);

        if (TEST)
            System.out.println("d = " + d + " -> "
            +      (double) fitnessTable52.get(d));
    }

    return f;
}


//String s is mutated at one place chosen at random
private static String mutation(String s) {
    //System.out.println("\ns in = " + s);
    int m = r.nextInt(s.length());
    char ch = s.charAt(m);
    if (ch == '0') ch = '1';
    else ch = '0';

    s = s.substring(0,m)  + ch + s.substring(m+1);

    //System.out.println("m = " + m + " s out = " + s);
    return s;
}
```

```
//A random walk over 5-strings is enabled
private static void wander() {
    /*
    System.out.println("String --> fitness50 + sign of change"
            + "\n+ fitness51 + sign of change");
    */
    System.out.println( "RANDOM WALK "
            + "IN THE SPACE OF STRINGS");
    System.out.println( "fitness50 , "
            + "fitness51 ,fitness52");
    System.out.println( "CopyPaste to Excel "
            + "or LibreOffice to make a graphic");
    double oldFitness50;
    double newFitness50;
    double oldFitness51;
    double newFitness51;
    double oldFitness52;
    double newFitness52;
    char sign50 ='=';
    char sign51 = '=';
    char sign52 = '=';
    double totalVariation50 = 0;
    double totalVariation51 = 0;
    double totalVariation52 = 0;

    String s = newStart();
    for (int step = 0; step < NSTEPS; step++) {
        oldFitness50 = fitness50(s);
        oldFitness51 = fitness51(s);
        oldFitness52 = fitness52(s);

        s = mutation(s);
        newFitness50 = fitness50(s);
        newFitness51 = fitness51(s);
        newFitness52 = fitness52(s);
        if (newFitness50 > oldFitness50) sign50 = '+';
        if (newFitness50 == oldFitness50) sign50 = '=';
        if (newFitness50 < oldFitness50) sign50 = '-';
        if (newFitness51 > oldFitness51) sign51 = '+';
        if (newFitness51 == oldFitness51) sign51 = '=';
        if (newFitness51 < oldFitness51) sign51 = '-';
        if (newFitness52 > oldFitness52) sign52 = '+';
        if (newFitness52 == oldFitness52) sign52 = '=';
        if (newFitness52 < oldFitness52) sign52 = '-';
        totalVariation50 =  totalVariation50
```

```
                    + Math.abs(newFitness50 - oldFitness50);
            totalVariation51 =  totalVariation51
                    + Math.abs(newFitness51 - oldFitness51);
            totalVariation52 =  totalVariation52
                    + Math.abs(newFitness52 - oldFitness52);
            /*
            System.out.println(s + " -> "
                    + newFitness50 + sign50 + "   "
                    + newFitness51 + sign51 + "   "
                    + newFitness52 + sign52);*/
            System.out.println(
                      newFitness50 +   "\t"
                    + newFitness51 + "\t"
                    + newFitness52 );
        }
        System.out.println("totalVariation50 = "
                + totalVariation50);
        System.out.println("totalVariation51 = "
                + totalVariation51);
        System.out.println("totalVariation52 = "
                + totalVariation52);
    }

    public static void main(String args[]) {
        initializeTable50();
        /*
        initializeTable51();
        initializeTable52();
*/
        initializeTable511();
        initializeTable522();
        wander();
    }
}//End of Program C28, NKmodel2
```

29, page 109. The following code reports the behavior of the fitness function along a random walk for the general NK model.

```
/* Program C29, NKmodel3
*
* We study the NK model proposed by Kauffman(1998).
* We report ups and downs of the fitness function
* along  a random walk.
```

```
 *
 *
 * We interpret each string as a program,
 * say, DSFVR, represents an execution in which method
 * R goes first, followed by method  V and so on.
 * The  problem for optimization is to find a program that fits
 * a certain specific function. That is not our problem here:
 * we pay attention to evolvability, so,
 * we study the ups and downs of the
 * fitness function along a random walk along
 * the space of programs, i.e., of strings with
 * the same length.
 * The N0 models proposes a fitness for each char
 * and the fitness of a string is the sum of the
 * fitnesses of its chars.
 * In the N1-model the subsequences
 * of two chars are assigned a fitness from which the fitness
 * of every  string can be calculated. Example:
 * f(00) = 1, f(01) = 2, f(10) = 1, f(11) = 3.
 * So, the fitness of 10101 is, by definition equal to
 * f(10) + f(01) + f(10)+ f(01) = 1 + 2 + 1 + 2 = 6.
 *

 * In the general NK model, the present purpose,
 * the fitness of each char is
 * the fitness of a substring that begins at it
 * and contains the following  k chars.
 * To avoid orphans at the upper tail,
 * strings are circularized:
 * f(10101) = f(10) + f(01) + f(10)+ f(01) + f(11)
 * = 1 + 2 + 1 + 2 + 3 = 9.

 */
package ejvol3v2p;

import java.util.HashMap;
import java.util.Map;
import java.util.Random;

public class NKmodel3 {


    // Turn on   the random generator
    static Random r = new Random();

    //Length of all strings
```

```
private static final int N = 10;

//Length of all clustering substrings
private static final int K1 = 2;
private static final int K2 = 8;

//Maximum number of steps in the random walk
private static final int NSTEPS = 100;

private static  Map fitnessTableNK1;
private static  Map fitnessTableNK2;


private static final boolean TEST = false;



private static String randomBinaryString() {
    String a = "";
    for (int m = 0; m < N; m++) {
        int l = r.nextInt(2);
        a = a + l;
    }
    return a;
}


//A binary string N chars long is output
private static String randomBinaryString(int N) {
    String a = "";
    for (int m = 0; m < N; m++) {
        int l = r.nextInt(2);
        a = a + l;
    }
    return a;
}


//Int i is converted to binary string
//Model NK
private static String iToString(int K, int i)
{
    //int i in base 10 to base 2
    String s = Integer.toString(i, 2);
    //System.out.println("s = " + s);
    int l = s.length();
```

```
    for(int j = 0; j < K-l+1; j++)
        s = "0" + s;
    //System.out.println("s = " + s);
    return s;
}


//The hash table of the K substrings, NK model.
//Challenge: this table is used to read
//the value, fitness, associated
//to a key which is a string.
//Because strings are ordered and isomorphic to
//natural numbers,   this table can be changed by
//a function of the type
//binary string -> fitness.
//Necessary if one falls short of memory.
private static  Map initializeTableNK(int K)
{
    Map fitnessTable = new HashMap();
    for(int i = 0; i < Math.pow(2, K+1); i++)
    {
        String s =  iToString(K,i);
        //System.out.println("initializeTableNK:  s = " + s);
        fitnessTable.put(s, r.nextDouble());
    }
    return fitnessTable;
}



//The fitness is the sum of k-cluster-fitnesses.
//Strings are closed.
private static double fitnessNK(int K,
        Map fitnessTable, String s)
{

    double f = 0;
    int n = s.length();
    s = s + s.substring(0,K);
    String d ;
    for(int i = 0; i < n; i++)
    {
        d = s.substring(i,i+K+1);

        f = f +  (double) fitnessTable.get(d);
```

```
        if (TEST)
            System.out.println("d = " + d + " -> "
            +      (double) fitnessTable.get(d));
    }
    return f;
}

//String s is mutated at one place chosen at random
private static String mutation(String s) {
    //System.out.println("\ns in = " + s);
    int m = r.nextInt(s.length());
    char ch = s.charAt(m);
    if (ch == '0') ch = '1';
    else ch = '0';

    s = s.substring(0,m)  + ch + s.substring(m+1);

    //System.out.println("m = " + m + " s out = " + s);
    return s;
}



//A random walk over N-strings is enabled
private static void wander1() {
    System.out.println( "RANDOM WALK "
            + "IN THE SPACE OF STRINGS");
    System.out.println( "fitness");
    System.out.println( "CopyPaste to "
            + "Excel or LibreOffice to make a graphic");
    double oldFitnessNK;
    double newFitnessNK;
    double totalVariationNK = 0;


    String s = randomBinaryString(N);
    for (int step = 0; step < NSTEPS; step++) {
        oldFitnessNK = fitnessNK(K1,fitnessTableNK1, s);
        s = mutation(s);
        newFitnessNK = fitnessNK(K1, fitnessTableNK2, s);
        totalVariationNK =  totalVariationNK
                + Math.abs(newFitnessNK - oldFitnessNK);
        System.out.println(newFitnessNK);
    }
    System.out.println("totalVariation50 = "
```

```
            + totalVariationNK);
    }


    //Comparison of two values of K, same N
    private static void wander2() {

        System.out.println( "RANDOM WALK "
                + "IN THE SPACE OF STRINGS");
        System.out.println( "fitnessK1 "
                + ",  fitnessK2");
        System.out.println( "CopyPaste to Excel "
                + "or LibreOffice to make a graphic");
        double oldFitnessK1;
        double newFitnessK1;
        double oldFitnessK2;
        double newFitnessK2;

        double totalVariationK1 = 0;
        double totalVariationK2 = 0;

        String s = randomBinaryString();
        for (int step = 0; step < NSTEPS; step++) {

            oldFitnessK1 = fitnessNK(K1,fitnessTableNK1, s);
            oldFitnessK2 = fitnessNK(K2,fitnessTableNK2, s);

            s = mutation(s);
            newFitnessK1 = fitnessNK(K1, fitnessTableNK1, s);
            newFitnessK2 = fitnessNK(K2, fitnessTableNK2, s);

            totalVariationK1 =  totalVariationK1
                    + Math.abs(newFitnessK1 - oldFitnessK1);
            totalVariationK2 =  totalVariationK2
                    + Math.abs(newFitnessK2 - oldFitnessK2);

            System.out.println(
                     newFitnessK1 + "\t"
                    + newFitnessK2 );
        }
        System.out.println("totalVariationK1 = "
                + totalVariationK1);
        System.out.println("totalVariationK2 = "
                + totalVariationK2);
    }
```

```
    public static void main(String args[]) {
        fitnessTableNK1 =   initializeTableNK(K1);
        fitnessTableNK2 =   initializeTableNK(K2);
        //wander1();
        wander2();
    }
}//End of Program C29, NKmodel3
```

33, page 110. Mutations in the genome of actual living beings produce morphological abnormalities that we call monstrosities. The fitness of a being like that is in general significantly lowered. There are no conglomerates of monstrosities in the fossil record. Thus, the NK model is ruled as a general model of biological evolution. The reason of this failure is that the NK model predicts that any pair of peaks, which represent a triumphant species, is joined by viable paths that go down across valleys to rise again. But, by contrast, some few mutations in the genome of living beings are enough to kill viability and so the local peaks that represent species are effectively isolated from the rest. Nevertheless, the NK model could eventually represent the evolution of the immunological system, whose purpose is precisely to generate innumerable viable variants of antigens. Also, this model could represent evolution along artificial worlds such as that enabled by LISP, a programming language in which one can do mutations to a code of a program that always produce viable programs. That is why it is intelligent to consider LISP as a powerful allied of genetic programming. Natural evolution is, in general, more similar to evolution of Java code: most random mutations halt the compiler. Nevertheless, an intelligent large change always can be made, both in Java and in nature with genetic engineering.

Problems of Chapter 3

38, page 118. For number of cities = 20, if the number of random links is less than 20, the number of cities in the largest component could be as few as 10. But when the number of random links grows passing 20, the number of cities in the largest component steadily approaches 20. When the number of links is 60, it is almost certain that all cities form the one and only component.

39, page 118. The connectivity grows from zero to 60% when the number of links is 300. When the number of roads approached 400, 80% of the cities belongs in the giant component. For 800 links, 97% or more of the cities enter to form part of the giant component. For 1200 links, more than the 99% of the cities enters into the giant component.

44, page 126. Answer: one looks at the matrix as a whole, if a giant cycle arises, most probably it is accompanied with a giant interlaced component. This means that cycles are no miracles in any sense. A giant interconnected component has giant cycles and a giant cycle is accompanied with a giant interconnected component. Hence, when we have a giant cycle, a good proportion of various continuous entries of high powers of the matrix Links must be filled in ones.

45, page 126. Unidirectional reactions are quite powerful to generate cycles that, when

there is sufficient number of links, cover most of the population of nodes or cities. Thus, a great cycled component is a necessity in unidirectional random graphs for density of links sufficiently high (four times more links than nodes is a proportion that enables that the great cycled component encloses more than 90% of the population).

46, page 127. Great cycles serve as background ecosystems, a sample subsystem of which could get encapsulated to form a first model of life. This life must coupled to the milieu and so it needs to acquire and digest free energy and resources to maintain its consistency, to grow and to reproduce. All this entails complexity that cannot be posited at the interior of a small cycle. Besides, adaptation to the milieu could best be imagined in the presence of evolution, so the incorporation of RNA or DNA is mandatory.

Problems of Chapter 4

58, page 139. The code for primes following Eratosthenes follows:

```
/*Program C58, Eratosthenes:
 * The program generates all primes
 * until 1000 following the method
 * that was proposed by Eratosthenes, which
 * consists in deleting from a complete list of numbers
 * all multiples of already found primes, knowing that
 * 1 and 2 are primes.
 *
 */

package ejvol3v2p;


public class Eratosthenes {


    private static final int N = 1000;

    //A vector of flags: true for a prime, false otherwise
    private static final boolean PRIMES[] = new boolean[N];

    public static void main(String[] args) {
        //Flags are initialized in true:
        //we have a complete list.
        for (int i = 1; i < N; i++) {
            PRIMES[i] = true;
        }
        //Multiples are deleted:
        int limit = (int) Math.sqrt(N);
```

```
        for (int i = 2; i < limit; i++) {
            if (PRIMES[i]) {
                for (int j = 2 * i; j < N; j = j + i) {
                    PRIMES[j] = false;
                }
            }
        }
        //PRIMES are reported
        int counter = 0;
        for (int i = 1; i < N; i++) {
            if (PRIMES[i]) {
                counter = counter + 1;
                System.out.println(i);
            }
        }
        System.out.println("Number of primes = " + counter);
    }
}//End of Program C58 Eratosthenes
```

61, page 151. Evolution readily guesses all primes one after another without much effort. Let us underscore that this is possible because of two reasons. First: it is a retro-engineering problem: the list of primes is produced by the Eratosthenes algorithm. Second, the semantics is coarse because it is gradualist: string 101010 is near to 111010. Things would be otherwise if the semantics were all-or-nothing: you have a matching else you have nothing.

62, page 151. The more striking results happen in the transition from a population with one individual to a population with two individuals. For the first primes until 1000, we have: if one runs the program with one individual and with or without recombination, the average number of generations to achieve the goal is 375000 and the total number of individuals is the same. But if one runs the program with 2 individuals and without recombination, the number of generations lowers to the sixth part, and the number of individuals to a third. If additionally one turns on recombination, the efficiency of evolution in time is augmented just around 1%, while the efficiency in resources is still lower. These results are good for our honor: we had been unable to give a single little reason to explain why sex and recombination have not been banned from nature, but this exercise shows us that sex and recombination might offer a selective advantage. Nevertheless, the advantage is not too blasting, a result that is backed by the fact that some natural populations, say, of lizards, can flip flop forth and back from sexual reproduction to asexual one but with some preference for the sexual one. Research: study the so called **Muller's ratchet** that says that asexual populations are pray of chronic parasitism by deleterious mutations.

One might marvel at the fact that the effect of parallelism is orders of magnitude more powerful than the marginal effect of recombination given that parallelism has been already turned on. This is astonishing because one tends to imagine that interacting parallelism is just a synonym of recombination. So, which is the origin of such a telepathic effect between diverse lines that have no contact at all? The answer is that diverse lines do have contact

through preferential reproduction of the fittest: this is much effective than isolated reproduction of separated lines. We are lead to conclude that a mixed strategy of sexual and asexual reproductions is the best option. So, why hermaphroditism is not the rule?

66, page 160. Repetitions and statistics are included:

```
/*Program C66, RandSemStat:
 *
 * Our purpose is to match a set of random binary strings.
 * We add some statistics.
 *
 * To that aim, we generate random binary numbers with
 * f(n) bits that serve as target for evolution
 * and as pattern  for selection.
 * For f(n) = n, we generate random binary numbers
 * with 1,2,...,n bits.
 * For f(n) = 5n, we generate random binary numbers
 * with 1,5,10 bits and so on.
 *
 * A community of binary organisms is given the task
 * of guessing all generated numbers that serve as patterns,
 * until a give bound.
 *
 * The fitness of a binary string  or organism   is
 * the number of bits in which it is
 * congruent  with the pattern that is also the target.
 * Example: pattern = eleven = 1011,
 * organism = 1010; fitness = 3;
 * When a target has been  guessed,
 * the next   in the list  becomes the new target,
 * and so on.
 *
 * New Java material: we use the class long,
 * which is designed for keeping integer values
 * that could be very large.
 */
package ejvol3v2p;

import java.util.BitSet;
import java.util.Random;

public class RandSemStat {

    private static final long serialVersionUID = 1L;
```

```
    private static int maxLength;
    //A population of binary numbers is considered.
    //The number of individuals is constant
    private static int numbInd;
    private static final int NUMBIND = 10000;
    private static final BitSet IND1[] = new BitSet[NUMBIND];
    private static final BitSet IND2[] = new BitSet[NUMBIND];
    private static double FitnessInd[]
            = new double[NUMBIND];
    private static final int ORDERIND[] = new int[NUMBIND];
    private static int gen;
    //Cummulative counter of generations
    private static int cumNumbGen = 0;
//Cummulative counter of trials or individuals
    private static int cumNumbInd = 0;
    private static boolean problem = true;
    private static int counter;
    private static BitSet target = new BitSet();
    private static int unitLength;
    private static int maxNumbUnits;
    private static int length;
    private static boolean recombination;
    //if the pop is reinitialized at each gen, reinit = true
    private static boolean reinit;
    private static double mutationRate;

    private static final int NREP = 100;
    //memory for statistics
    private static final long NGEN[] = new long[NREP];
    private static final long NIND[] = new long[NREP];
    //Number of repetitions
    private static final int NREPETITIONS = 10;

    // Turn on  the random generator
    static Random r = new Random();

    private static void setParameters() {
        //length of unit
        unitLength = 1;
        maxNumbUnits = 8;
        numbInd = 10; //less than MaxNumbInd = 10000
        recombination = false; //true = recombination is enabled
        //reinit = true: the pop is initialized with a new task
        reinit = true;
        mutationRate = 0.3;
        //maxLength is the maximum length of the pattern
```

```
        maxLength = maxNumbUnits * unitLength;
    }

    private static int length(int n) {
        int z  = n * unitLength;
        return z;
    }

    //Converts a BitSet value into a String
    static private String decode(BitSet t) {
        String s = "";
        for (int i = 0; i < length; i++) {
            if (t.get(i) == true) {
                s = s + "1";
            } else {
                s = s + "0";
            }
        }
        return s;
    }

    //A random binary pattern  or target is generated
    //f(n) bits long
    private static BitSet defineTarget(int n) {
        BitSet pattern = new BitSet();
        length = length(n);
        for (int i = 0; i < length; i++) {
            int l = r.nextInt(2);
            if (l == 1) {
                pattern.set(i);
            }
        }
        /*
  System.out.println("New length = " + length);
  System.out.println("New  binary  target = "
                     + decode(pattern));
  System.out.println("BitSet = "
                           + (BitSet) pattern);
         */
        return pattern;
    }

    //IND1 is a population of binary strings.
    //The BitSet class is used to instantiate
    //binary strings as individuals
    private static void initializePop() {
```

```
        BitSet s = new BitSet();
        //IND1 is false everywhere
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) s.clone();
        }
        //Reinitialization with random truth values.
        for (int i = 0; i < numbInd; i++) {
            for (int j = 0; j < length; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
        }
        // System.out.println("IND11 : " + IND1[1].get(1));
    }

    //Initialization
    private static void beginning() {
        counter = 1;
        target = defineTarget(counter);
        initializePop();
    }

// Each individual is tested
// for how good it matches the target
    static private int assessInd(BitSet s) {
        //The fitness of s is the number of bits in which
        //s and target coincide;
        int fitness = 0;
        for (int i = 0; i < length; i++) {
            if (s.get(i) == target.get(i)) {
                fitness = fitness + 1;
            }
        }
        /*
 System.out.print("target = " + target);
 System.out.print(" s = " + s + "fitness = "
                                 + fitness);
 System.out.println(" unitLength = " +unitLength);
         */
        return fitness;
    }

    // test for assessInd(s);
    private static void test1() {
```

```
        BitSet t = new BitSet(5);
        target = (BitSet) t.clone();
        target.set(1);
        target.set(3);
        target.set(5);
        BitSet s = new BitSet(5);
        s.set(0);
        s.set(2);
        s.set(3);
        s.set(4);
        assessInd(s);
    }

// Individuals are sorted by fitness.
// the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        double FitnessIndc[];
        FitnessInd = new double[NUMBIND];
        FitnessIndc = new double[NUMBIND];
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < numbInd; i++) {
            Champ = 0;
            for (int j = 0; j < numbInd; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array Order keeps a record of fitness.
            //The fittest is number zero
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        /*System.out.println("Gen = " +gen);
```

```
  System.out.println("Champ = " + IND1[ORDERIND[0]]);
  for(int i = 0; i< numbInd;i++)
  {
   System.out.println(ORDERIND[i] + "     " +
            FitnessInd[ORDERIND[i]]);
  }
         */
        // if Pattern has been guessed
        if (FitnessInd[ORDERIND[0]] == length) {
            problem = false;
            //System.out.println("Champ = " + IND1[ORDERIND[0]]);
        }
        return problem;
    }

// The new generation is made
    private static void newGeneratioNIND() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest

        while (i < numbInd) {
            int myCounter = numbCopies;
            while ((myCounter > 0) & (i < numbInd)) {
                //System.out.println(i + "" + l);
                IND2[i] = (BitSet) IND1[ORDERIND[k]].clone();
                //System.out.println(i + " " + counter + " " + k);
                i++;
                myCounter = myCounter - 1;
            }
            k++;
        }
        for (int j = 0; j < numbInd; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*  System.out.println( j+ " IND1 "
                                + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(numbInd);
```

```
        //b is false everywhere
        BitSet b = new BitSet();
        //B is the recombinant of l and k
        for (int j = 0; j < length; j++) {
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    b.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
                    b.set(j);
                }
            }
        }
        /*
 System.out.println();
 System.out.println("Ind[l] = " + IND1[l]);
 System.out.println("Ind[k] = " + IND1[k]);
 System.out.println("Recomb = " + b);
         */
        return b;
    }

// The definitive population  is a recombinant one.
    private static void recombinatioNIND() {
        //IND2 is a copy of IND1
        for (int i = 0; i < numbInd; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //IND2 is a recombinant version of IND1
        for (int l = 0; l < numbInd; l++) {
            IND2[l] = recInd(l);
        }
        //IND2 is transferred to IND1
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }

// Individual  a is mutated at a random place.
    private static BitSet pointMutatioNIND(BitSet a) {
        int l = r.nextInt(length);
```

```
        a.flip(l);
        return a;
    }

// Individuals are mutated
    private static void mutatioNIND() {
        for (int i = 0; i < numbInd; i++) {
            double p = r.nextDouble();
            if (p < mutationRate) {
                IND1[i] = pointMutatioNIND(IND1[i]);
            }
        }
    }

// Given a goal, it is guessed by evolution
    private static void evolution() {

        //The new generation is produced
        //as copies of the fittest
        newGenerationNIND();
        if (recombination) {
            recombinatioNIND();
        }
        //The new population is subjected to mutation
        mutatioNIND();
        gen++;
        problem = sortingInd();
    }

    private static void report1() {
        System.out.println("Task ended by generation =" + gen);
        System.out.println("Number of trials= " + gen * numbInd);
        BitSet match = (BitSet) IND1[ORDERIND[0]].clone();
        System.out.println("Match  = " + match);
        System.out.println("Found binary target = "
                + decode(match));
        System.out.println("cumulatedNumbGen = " + cumNumbGen);
        System.out.println("cumulatedNumbInd = " + cumNumbInd);
    }

    private static void report3() {
        System.out.println();
        System.out.println("FINAL REPORT");
        System.out.println("length until " + maxLength);
        System.out.println("UnitLength = " + unitLength);
        System.out.println("Ind/generation = " + numbInd);
```

```
        System.out.println("Mutation rate = " + mutationRate);
        System.out.println("Recombination  = " + recombination);
        System.out.println("Reinitialization  = " + reinit);
    }

    private static void report2() {
        report3();
        System.out.println("Number of guessed targets = "
                + counter);
        System.out.println("totalNumbGen = " + cumNumbGen);
        System.out.println("totalNumbInd = " + cumNumbInd);
        System.out.println();
        System.out.println("FINAL REPORT");
        System.out.println(maxLength);
        System.out.println(unitLength);
        System.out.println(numbInd);
        System.out.println(mutationRate);
        System.out.println(recombination);
        System.out.println(reinit);
        System.out.println(counter);
        System.out.println(cumNumbGen);
        System.out.println(cumNumbInd);
    }

//The population must match all patterns, one by one
    private static void task() {
        setParameters();
        beginning();
        int percent = 0;
        gen = 0;
        cumNumbGen = 0;
        cumNumbInd = 0;
        while ((problem)) {
            //System.out.println( "gen of Ind = " + gen);
            evolution();
            //if a pattern has  been guessed ( problem = false)
            //then, pass to the next prime.
            if ((problem == false)) {
                cumNumbGen = cumNumbGen + gen;
                cumNumbInd = cumNumbInd + gen * numbInd;
                //report1();
                int percentage = (int) 100
                        * counter / (maxLength);
                if (percentage > percent) {
                    percent = percentage;
                    System.out.println("Done work = " + percent + "%");
```

```
            }

            if (counter < maxNumbUnits) {
                counter = counter + 1;
                System.out.println("");
                target = defineTarget(counter);
                //System.out.println("Decoded = " + decode(target));
                if (reinit) {
                    initializePop();
                }
                gen = 0;
                problem = true;
            }
        }
    }
    //report3();
    counter = 0;
}

public static void main(String[] args) {
    for (int i = 0; i < NREPETITIONS; i++) {
        System.out.println("Repetition = " + i);
        problem = true;
        task();
        NGEN[i] = cumNumbGen;
        NIND[i] = cumNumbInd;
    }
    double sumGen = 0;
    double sumInd = 0;
    for (int i = 0; i < NREPETITIONS; i++) {
        sumGen = sumGen + NGEN[i];
        sumInd = sumInd + NIND[i];
    }

    double meaNGEN = sumGen / NREPETITIONS;
    double meaNIND = sumInd / NREPETITIONS;
    double sumDGen = 0;
    double sumDInd = 0;
    for (int i = 0; i < NREPETITIONS; i++) {
        sumDGen = sumDGen + Math.pow((NGEN[i] - meaNGEN), 2);
        sumDInd = sumDInd + Math.pow((NIND[i] - meaNIND), 2);
    }
    report3();
    double stanDevGen = Math.sqrt(sumDGen / NREPETITIONS);
    double stanDevInd = Math.sqrt(sumDInd / NREPETITIONS);
    System.out.println("Repetitions = " + NREPETITIONS);
```

```
        System.out.println("MeaNGEN = " + meaNGEN);
        System.out.println("stanDevGen = " + stanDevGen);
        System.out.println("MeaNIND = " + meaNIND);
        System.out.println("stanDevInd = " + stanDevInd);
    }
}//Program C66, RandSemStat
```

67, page 160. The great invention is parallelism + selection. If recombination is added, its effect is slightly negative. So, why do sex and recombination have not been banned from nature? An extended belief is that recombination is very good for reparation of damaged information and in this regard it could do it very good. Dear Reader, let me to dare you: compose the corresponding code that would allow you to prove else refute this belief that would explain why sex has not been banned from life.

68, page 160. The number of generations has a linear dependence on the length of the unit to be matched. Thus, this semantics offers no challenge to evolution at all.

71, page 170. Evolution fiercely resists the attack of semantic discontinuity but at last it recedes and begins to consume exaggerated quantities of time. To augment the degree of parallelism causes a tiny delay but represents no remedy at all. The next table shows the number of generations needed to solve a given discontinuity level for a population of 10 and 100 individuals, with recombination, a pattern length of 128 and without repetitions:

```
Generations needed to match a predefined random target
Disc level   N=10        N=100
1             234          41
2             295          48
4             357          73
8            1053         215
16           4306        5337
25         498070      613585
32     >116000000   >12200000
```

72, page 170. The code for a random walk that matches a pattern follows:

```
/*Program C72, RandWalk:

 We generate a random walk, which stops when matching
 a predefined boolean pattern.

 To reuse program C69, DiscSem, we notice that
 a random walk is equal to evolution
 without recombination and selection.
 */
package ejvol3v2p;

import java.util.BitSet;
```

```
import java.util.Random;

public class RandWalk {

    private static int pattLength;
    private static long gen;
    private static boolean problem = true;
    private static BitSet target = new BitSet();
    private static BitSet individual = new BitSet();
    //number of bits where the individual is mutated
    private static int mode;
    private static final int MAXNREP = 100;
    //memory for statistics
    private static final long NGEN[] = new long[MAXNREP];
    //Number of repetitions
    private static final int NREP = 10;

    static Random r = new Random();

    //The   length of the pattern is defined
    private static void setParameters() {
        //pattLength is the   length of the pattern
        pattLength = 32;
        System.out.println("length of pattern = " + pattLength);
    }

    //A random binary pattern  or target is generated
    private static BitSet defineTarget(int pattLength) {
        BitSet pattern = new BitSet();
        for (int i = 0; i < pattLength; i++) {
            int l = r.nextInt(2);
            if (l == 1) {
                pattern.set(i);
            }
        }
        return pattern;
    }

    //Initialization
    private static void beginning() {
        target = defineTarget(pattLength);
        individual = defineTarget(pattLength);
    }

//individual is mutated in  a number of  places = mode.
    private static void mutation(int mode) {
```

```
        for (int i = 0; i < mode; i++) {
            int m = r.nextInt(pattLength);
            if (individual.get(m) == true) {
                individual.clear(m);
            } else {
                individual.set(m);
            }
        }
    }

//Does the pattern has been matched?
    private static boolean assessInd() {
        /*
        //Equivalent to
        if (individual.equals(target)) {
            return false;
        } else {
            return true;
        }
        */
        return !individual.equals(target);



    }

//Given a pattern, it is guessed by a random walk
    private static void evolution() {
        mode = 1;
        mutation(mode);
        gen++;
        if (gen % 100000000 == 0)
            System.out.println(" Generation = " + gen);

        problem = assessInd();
    }

//The rand walk must match a pattern
    private static void task() {
        setParameters();
        beginning();
        gen = 0;
        while ((problem)) {
            //System.out.println( "gen of Ind = " + gen);
            evolution();
```

```
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < NREP; i++) {
            System.out.print("Repetition = " + i);
            problem = true;
            task();
            NGEN[i] = gen;
            System.out.println(" Generations = " + gen);
        }
        double sumGen = 0;
        for (int i = 0; i < NREP; i++) {
            sumGen = sumGen + NGEN[i];
        }
        double meaNGEN = sumGen / NREP;
        double sumDGen = 0;
        for (int i = 0; i < NREP; i++) {
            sumDGen = sumDGen + Math.pow((NGEN[i] - meaNGEN), 2);
        }
        double stanDevGen = Math.sqrt(sumDGen / NREP);
        System.out.println("Repetitions = " + NREP);
        System.out.println("MeanGen = " + meaNGEN);
        System.out.println("stanDevGen = " + stanDevGen);
    }
}//End of Program C72, RandWalk
```

To compare the performance of randomness with that of evolution under a discontinuous semantics, we studied a task consisting in matching a string B bits long. In a random walk we always work with one individual, so the number of generated individuals, a measure of needed resources, is equal to the number of generations. For $B \leq 25$ evolution was found to be by far better than randomness. But for $B = 30$ we got:

```
For Randomness
Repetitions = 10
MeanGen = 7.683521293E8 = number of individuals
stanDevGen = 8.052556333737423E8
Distribution with a long upper tail.

For evolution:
Discontinuity level = 29
Pattern length = 30
Ind/generation = 10
Mutation rate = 0.3
Recombination  = true
Repetitions = 1
MeanGEN = 7.9321303E7
```

```
stanDevGen = 0.0
Total number of individuals = 7.9321303E8

Discontinuity level = 29
Pattern length = 30
Ind/generation = 100
Mutation rate = 0.3
Recombination  = true
Total number of individuals > 1180000000 = 1.1E9
```

For a target 32 bits long we got:

```
For Randomness
Repetitions = 10
MeanGen = 5.8602937084E9 = number of individuals
stanDevGen = 4.604046228197776E9
Distribution with a long upper tail.

For evolution:
Discontinuity level = 31
Pattern length = 32
Ind/generation = 10
Mutation rate = 0.3
Recombination  = true
Repetitions = 1
MeanGEN = 1.72465642E8
stanDevGen = 0.0
Total number of individuals = 1.72465642E9
```

For a target 34 bits long we got:

```
For Randomness
Repetitions = 10
MeanGen = 1.95526296193E10
stanDevGen = 2.4080285835150 25E10
Distribution with a long upper tail.

For evolution:
Discontinuity level = 34
Pattern length = 34
Ind/generation = 10
Mutation rate = 0.3
Recombination  = true
Repetitions = 1
MeanGEN  > 2147400000


Total number of individuals >  2.1E10
```

We see that evolution under the pressure of heavy semantic discontinuity sometimes behaves worst than gambling and that the inefficiency is greater as the population size grows. Why? We see three possible causes. First: natural selection turns from being an agent of optimization into an agent of stagnation around local maxims. The mechanism might be as follows: selection replaces unfitted individuals by the fittest ones so, unfitted individuals do not have the opportunity to explore new alleys possibly leading to a higher peak. As a result, fittest individuals that belong in a suboptimal hill get dominant and are the cause of no further evolution. So, natural selection is the very cause that makes evolution to perform worst than randomness when semantics observes a high degree of discontinuity. Second: a discontinuous semantics creates a landscape, in which the optimal hill have an extremely low probability with respect to the set of suboptimal ones. Third: the proposed discontinuous semantics just do one thing: it effectively enlarges the input by adding chains to chains and henceforth the problem gets more complex. It is very intriguing to disentangle the truth that each mentioned reason could have. In any case, evolution under discontinuous semantics is not an alternative to randomness as an acceptable explanation of complexity: randomness is much more simple to program and runs more than 10 times faster.

73, page 170. The next program reports the periods of ecstasy, in which no change occurs.

```
/*Program C73, Ecstasy:

Our aim is to study the thesis that under
semantics discontinuity, long periods of Ecstasy are
interrupted by sporadic changes.

We modify  program C69 in order to register the periods
of Ecstasy, when no change occurs.
 */
package ejvol3v2p;

import java.util.BitSet;
import java.util.Random;

public class Ecstasy {

    //Number of repetitions
    private static final int NREP = 1;
    //level of discontinuity
    private static final int DISCLEVEL  = 17;
    //Length of pattern to match
    private static final int PATTLENGTH = 70;
    //A population of binary numbers is considered.
    //The number of individuals is constant
    private static final int NUMBIND = 100;
```

```
private static final BitSet IND1[] = new BitSet[NUMBIND];
private static final BitSet IND2[] = new BitSet[NUMBIND];
private static double FitnessInd[]
        = new double[NUMBIND];
private static final int ORDERIND[] = new int[NUMBIND];
private static int gen;

private static boolean problem = true;
private static BitSet target = new BitSet();
//RECOMBINATION = true =  is enabled
private static final boolean RECOMBINATION = true;
//REINIT = true: the pop is initialized with a new task
private static final boolean REINIT = true;
//Mutation rate per individual
private static final double MUTATIONRATE = 0.3;


//Number of generations for each repetition
private static final long NGEN[] = new long[NREP];
private static double bestFitness;
//Number of generations in which no change is observed
private static long ecstasy;
private static long oldGen;

// Turn on  the random generator
static Random r = new Random();


//Converts a BitSet value into a boolean string
static private String decode(BitSet t) {
    String s = "";
    for (int i = 0; i < PATTLENGTH; i++) {
        if (t.get(i) == true) {
            s = s + "1";
        } else {
            s = s + "0";
        }
    }
    return s;
}

//A random binary pattern  or target is generated
//PATTLENGTH bits long
private static BitSet defineTarget(int PATTLENGTH) {
```

```
        BitSet pattern = new BitSet();
        for (int i = 0; i < PATTLENGTH; i++) {
            int l = r.nextInt(2);
            if (l == 1) {
                pattern.set(i);
            }
        }

        System.out.println("BitSet = "
                           + (BitSet) pattern);
  System.out.println("Binary  target = "
                     + decode(pattern));


        return pattern;
    }

    //IND1 is a population of binary strings.
    //The BitSet class is used to instantiate
    //binary strings as individuals
    private static void initializePop() {
        BitSet s = new BitSet();
        //IND1 is false everywhere
        for (int i = 0; i < NUMBIND; i++) {
            IND1[i] = (BitSet) s.clone();
        }
        //Reinitialization with random truth values.
        for (int i = 0; i < NUMBIND; i++) {
            for (int j = 0; j < PATTLENGTH; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
        }
        // System.out.println("IND11 : " + IND1[1].get(1));
    }

    //Initialization
    private static void beginning() {
        target = defineTarget(PATTLENGTH);
        initializePop();
    }

// Each individual is tested
// for how good it matches the target
```

```
    static private int assessInd(BitSet s) {
        //The fitness of s is the number of substrings
        //DISCLEVEL bits long that s matches the target
        int fitness = 0;
        for (int i = 0; i < PATTLENGTH - DISCLEVEL + 1; i++) {
            if (s.get(i, i + DISCLEVEL)
                    .equals(target.get(i, i + DISCLEVEL))) {
                    fitness = fitness + 1;
                }
        }
    /*
    System.out.println("target = " + decode(target));
 System.out.println("     s = " + decode(s) + "\nfitness = "
 + fitness);
*/

        return fitness;
    }

// Individuals are sorted by fitness.
// the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        double FitnessIndc[];
        FitnessInd = new double[NUMBIND];
        FitnessIndc = new double[NUMBIND];
        for (int i = 0; i < NUMBIND; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < NUMBIND; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < NUMBIND; i++) {
            Champ = 0;
            for (int j = 0; j < NUMBIND; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
```

```
            //The array ORDERIND orders individuals by  fitness:
            //The fittest is number zero
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        /*System.out.println("Gen = " +gen);
  System.out.println("Champ = " + IND1[ORDERIND[0]]);
  for(int i = 0; i< NUMBIND;i++)
  {
   System.out.println(ORDERIND[i] + "     " +
            FitnessInd[ORDERIND[i]]);
  }
         */
        // if Pattern has been guessed
        if (FitnessInd[ORDERIND[0]]
                == PATTLENGTH - DISCLEVEL + 1) {
            problem = false;
        //System.out.println("Champ = " + IND1[ORDERIND[0]]);
        }

        if (FitnessInd[ORDERIND[0]] > bestFitness) {
            //d = distance();
            ecstasy = gen - oldGen;
            System.out.print("Ecstasy = " + ecstasy + "\t");
            oldGen = gen;
            bestFitness = FitnessInd[ORDERIND[0]];
            System.out.print(" Fitness winner = " + bestFitness + "\t");
            System.out.print(" Gen = " + gen);
            System.out.println();
        }

        return problem;
    }

// The new generation is made
    private static void newGenerationInd() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest

        while (i < NUMBIND) {
            int counter = numbCopies;
            while ((counter > 0) & (i < NUMBIND)) {
                //System.out.println(i + "" + l);
```

```
                IND2[i] = (BitSet) IND1[ORDERIND[k]].clone();
            //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int j = 0; j < NUMBIND; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*  System.out.println( j+ " IND1 "
                                    + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(NUMBIND);
        //b is false everywhere
        BitSet b = new BitSet();
        //B is the recombinant of l and k
        for (int j = 0; j < PATTLENGTH; j++) {
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    b.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
                    b.set(j);
                }
            }
        }
        /*
 System.out.println();
 System.out.println("Ind[l] = " + IND1[l]);
 System.out.println("Ind[k] = " + IND1[k]);
 System.out.println("Recomb = " + b);
         */
        return b;
    }
```

```
// The definitive population  is a recombinant one.
    private static void recombinationInd() {
        //Ind2 is a copy of IND1
        for (int i = 0; i < NUMBIND; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //Ind2 is a recombinant version of IND1
        for (int l = 0; l < NUMBIND; l++) {
            IND2[l] = recInd(l);
        }
        //Ind2 is transferred to IND1
        for (int i = 0; i < NUMBIND; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }

// Individual  a is mutated at a random place.
    private static BitSet pointMutationInd(BitSet a) {
        int l = r.nextInt(PATTLENGTH);
        a.flip(l);
        return a;
    }

// Individuals are mutated
    private static void mutationInd() {
        for (int i = 0; i < NUMBIND; i++) {
            double p = r.nextDouble();
            if (p < MUTATIONRATE) {
                IND1[i] = pointMutationInd(IND1[i]);
            }
        }
    }

// Given a goal, it is guessed by evolution
    private static void evolution() {

        //The new generation is produced
        //as copies of the fittest
        newGenerationInd();
        if (RECOMBINATION) {
            recombinationInd();
        }
        //The new population is subjected to mutation
        mutationInd();
        gen++;
        if (gen % 100000 == 0)
```

```
            System.out.println(" Generation = " + gen);
        problem = sortingInd();
    }


    private static void reportParameters() {
        System.out.println();
        System.out.println("FINAL REPORT");
        System.out.println("Discontinuity level = " + DISCLEVEL);
        System.out.println("Pattern length = " + PATTLENGTH);
        System.out.println("Ind/generation = " + NUMBIND);
        System.out.println("Mutation rate = " + MUTATIONRATE);
        System.out.println("Recombination  = " + RECOMBINATION);
        System.out.println("Reinitialization  = " + REINIT);
    }



//The population must match a  patterns
    private static void task() {
        beginning();
        gen = 0;
        while ((problem)) {
            //System.out.println( "gen of Ind = " + gen);
            evolution();
        }
    }

    public static void main(String[] args) {
        reportParameters();
        for (int i = 0; i < NREP; i++) {
            System.out.println("Repetition = " + i);
            problem = true;
            task();
            NGEN[i] = gen;

            System.out.println(" Generations = " + gen);

        }
        double sumGen = 0;

        for (int i = 0; i < NREP; i++) {
            sumGen = sumGen + NGEN[i];

        }
        double meanGEN = sumGen / NREP;
```

```
        double sumDGen = 0;

        for (int i = 0; i < NREP; i++) {
            sumDGen = sumDGen + Math.pow((NGEN[i] - meanGEN), 2);

        }

        double stanDevGen = Math.sqrt(sumDGen / NREP);

        System.out.println("Repetitions = " + NREP);
        System.out.println("MeanGEN = " + meanGEN);
        System.out.println("stanDevGen = " + stanDevGen);

    }

}//End of Program C73, Ecstasy
```

This program shows that our semantics generates two types of evolution: for low discontinuity, 1 to 10, more discontinuity means larger periods of ecstasy but the marginal fitness (difference between the maximum fitness of successive generations) could be larger at the beginning. For high level of discontinuity, the first success if very difficult to reach but the subsequent ones are not so hard. We conclude that our semantics represents the process of enrichment in which it is difficult to pick up the initial capital but once it is ready, then further richness are not difficult to get.

74, page 170. The next code produces a tempo of evolution characterized, almost perfectly, by ever larger periods of ecstasy:

```
/*Program C74, KSeries:

We try to answer to the challenge of creating
a semantics whose evolutionary unfolding
could be characterized by increasingly longer periods
of ecstasy.

The target of evolution is a given random string.
This string is subdivided in substrings that are
concatenated one after another. The first string
has 1 bit, the second 2, .. ..,
the k-th substring has k bits.
Each substring defines an all or nothing semantics:
if you have guessed it in toto, you have one point,
else you have nothing.
```

```
Example:
target     = 1-01-100-1110
individual = 1-01-010-1101
Fitness 1+1+0+0 = 2
 */
package ejvol3v2p;

import java.util.BitSet;
import java.util.Random;

public class KSeries {

// private static int  discLevel;
    //Number of Substrings
    private static int numbSubs;
    private static int pattLength;
    //A population of binary numbers is considered.
    //The number of individuals is constant
    private static int numbInd;
    private static final int MAXNUMBIND = 10000;
    private static final BitSet IND1[] = new BitSet[MAXNUMBIND];
    private static final BitSet IND2[] = new BitSet[MAXNUMBIND];
    private static int FitnessInd[]
            = new int[MAXNUMBIND];
    private static final int ORDERIND[] = new int[MAXNUMBIND];
    private static int gen;

    private static boolean problem = true;
    private static BitSet target = new BitSet();

    private static boolean recombination;
    //if the pop is reinitialized at each gen, reinit = true
    private static boolean reinit;
    private static double mutationRate;
    private static final int MAXNREP = 100;
    //memory for statistics
    private static final long NGEN[] = new long[MAXNREP];
    private static final long NIND[] = new long[MAXNREP];
    //Number of repetitions
    private static final int NREP = 1;
    private static int bestFitness;
    //Number of generations in which no change is observed
    private static long ecstasy;
    private static long oldGen;

    // Turn on  the random generator
```

```
  static Random r = new Random();

  private static void setParameters() {
      //number of substrings
      numbSubs = 6;
      numbInd = 1; //less than MaxNumbInd = 10000
      recombination = true; //true = recombination is enabled
      //reinit = true: the pop is initialized with a new task
      reinit = true;
      mutationRate = 0.3;
  }

  //Converts a BitSet value into a String
  static private String decode(BitSet t) {
      String s = "";
      for (int i = 0; i < pattLength; i++) {
          if (t.get(i) == true) {
              s = s + "1";
          } else {
              s = s + "0";
          }
      }
      return s;
  }

  //A random binary pattern  or target is generated
  //pattLength bits long
  private static BitSet defineTarget(int pattLength) {
      BitSet pattern = new BitSet();
      for (int i = 0; i < pattLength; i++) {
          int l = r.nextInt(2);
          if (l == 1) {
              pattern.set(i);
          }
      }
      /*
System.out.println("Binary  target = "
                   + decode(pattern));
System.out.println("BitSet = "
                         + (BitSet) pattern);
       */
      return pattern;
  }

  //IND1 is a population of binary strings.
  //The BitSet class is used to instantiate
```

```
    //binary strings as individuals
    private static void initializePop() {
        BitSet s = new BitSet();
        //IND1 is false everywhere
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) s.clone();
        }
        //Reinitialization with random truth values.
        for (int i = 0; i < numbInd; i++) {
            for (int j = 0; j < pattLength; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
        }
        // System.out.println("IND11 : " + IND1[1].get(1));
    }

    //Initialization
    private static void beginning() {
        pattLength = numbSubs * (numbSubs + 1) / 2;
        target = defineTarget(pattLength);
        initializePop();
    }

// Each individual is tested
// for how good it matches the target.
    static private int assessInd(BitSet s) {
        int fitness = 0;
        int l = 0;
        for (int i = 1; i <= numbSubs; i++) {
            if (s.get(l, l + i).equals(target.get(l, l + i))) {
                fitness = fitness + 1;
            }
            l = l + i;
        }
        /*
    System.out.println("target = " + decode(target));
 System.out.print( "s      = " + decode(s) + "fitness = "
                                + fitness);
 System.out.println();
         */
        return fitness;
    }
```

```
// Individuals are sorted by fitness.
// the fittest is the first, with minimum error
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        int FitnessIndc[];
        FitnessInd = new int[MAXNUMBIND];
        FitnessIndc = new int[MAXNUMBIND];
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < numbInd; i++) {
            Champ = 0;
            for (int j = 0; j < numbInd; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array ORDERIND orders individuals by  fitness:
            //The fittest is number zero
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        /*System.out.println("Gen = " +gen);
  System.out.println("Champ = " + IND1[ORDERIND[0]]);
  for(int i = 0; i< numbInd;i++)
  {
   System.out.println(ORDERIND[i] + "     " +
            FitnessInd[ORDERIND[i]]);
  }
         */
        // if Pattern has been guessed
        if (FitnessInd[ORDERIND[0]] == numbSubs) {
            problem = false;
            //System.out.println("Champ = " + IND1[ORDERIND[0]]);
        }
        if (FitnessInd[ORDERIND[0]] > bestFitness) {
```

```
            //d = distance();
            ecstasy = gen - oldGen;
            System.out.print("Ecstasy = " + ecstasy + "\t");
            oldGen = gen;
            bestFitness = FitnessInd[ORDERIND[0]];
            System.out.print(" best fitness  = " + bestFitness + "\t");
            System.out.print(" Gen = " + gen);
            System.out.println();
        }
        return problem;
    }

// The new generation is made
    private static void newGenerationInd() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest
        while (i < numbInd) {
            int counter = numbCopies;
            while ((counter > 0) & (i < numbInd)) {
                //System.out.println(i + "" + l);
                IND2[i] = (BitSet) IND1[ORDERIND[k]].clone();
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int j = 0; j < numbInd; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*  System.out.println( j+ " IND1 "
                                + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(numbInd);
        //b is false everywhere
        BitSet b = new BitSet();
        //B is the recombinant of l and k
        for (int j = 0; j < pattLength; j++) {
```

```
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    b.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
                    b.set(j);
                }
            }
        }
        /*
 System.out.println();
 System.out.println("Ind[l] = " + IND1[l]);
 System.out.println("Ind[k] = " + IND1[k]);
 System.out.println("Recomb = " + b);
         */
        return b;
    }

// The definitive population  is a recombinant one.
    private static void recombinationInd() {
        //IND2 is a copy of IND1
        for (int i = 0; i < numbInd; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //IND2 is a recombinant version of IND1
        for (int l = 0; l < numbInd; l++) {
            IND2[l] = recInd(l);
        }
        //IND2 is transferred to IND1
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }

// Individual  a is mutated at a random place.
    private static BitSet pointMutationInd(BitSet a) {
        int l = r.nextInt(pattLength);
        a.flip(l);
        return a;
    }
```

```
// Individuals are mutated
    private static void mutationInd() {
        for (int i = 0; i < numbInd; i++) {
            double p = r.nextDouble();
            if (p < mutationRate) {
                IND1[i] = pointMutationInd(IND1[i]);
            }
        }
    }

    // Given a goal, it is guessed by evolution
    private static void evolution() {

        //The new generation is produced
        //as copies of the fittest
        newGenerationInd();
        if (recombination) {
            recombinationInd();
        }
        //The new population is subjected to mutation
        mutationInd();
        gen++;
        problem = sortingInd();
    }


    private static void report3() {
        System.out.println();
        System.out.println("FINAL REPORT");
        System.out.println("Pattern length " + pattLength);
        System.out.println("Ind/generation = " + numbInd);
        System.out.println("Mutation rate = " + mutationRate);
        System.out.println("Recombination  = " + recombination);
        System.out.println("Reinitialization  = " + reinit);
    }


//The population must match all patterns, one by one
    private static void task() {
        setParameters();
        beginning();
        gen = 0;
        ecstasy = 0;
        bestFitness = 0;
        oldGen = 0;
        while ((problem)) {
```

```
            //System.out.println( "gen of Ind = " + gen);
            evolution();
        }
        //report3();
    }

    public static void main(String[] args) {
        for (int i = 0; i < NREP; i++) {
            System.out.println("Repetition = " + i);
            problem = true;
            task();
            NGEN[i] = gen;
            NIND[i] = numbInd * gen;
        }
        double sumGen = 0;
        double sumInd = 0;
        for (int i = 0; i < NREP; i++) {
            sumGen = sumGen + NGEN[i];
            sumInd = sumInd + NIND[i];
        }
        double meanGen = sumGen / NREP;
        double meanInd = sumInd / NREP;
        double sumDGen = 0;
        double sumDInd = 0;
        for (int i = 0; i < NREP; i++) {
            sumDGen = sumDGen + Math.pow((NGEN[i] - meanGen), 2);
            sumDInd = sumDInd + Math.pow((NIND[i] - meanInd), 2);
        }
        report3();
        double stanDevGen = Math.sqrt(sumDGen / NREP);
        double stanDevInd = Math.sqrt(sumDInd / NREP);
        System.out.println("Repetitions = " + NREP);
        System.out.println("MeanGen = " + meanGen);
        System.out.println("stanDevGen = " + stanDevGen);
        System.out.println("MeanInd = " + meanInd);
        System.out.println("stanDevInd = " + stanDevInd);
    }
}//End of Program C74, KSeries
```

Problems of Chapter 5

81, page 182. A proposition with any number of variables is built and evaluated:

```
/* Program C81, Prop2.
```

```
 * We generate a logical proposition
 * with m operators and n variables
 * chosen at random.

 * Number of variables shall be less than 10 so,
 * they can be named by numbers 0, 1, ... 9
 * that are encoded as chars.

 * Number of variables is restricted to
 * positive int numbers.


 * Random  equiprobable  truth values are assigned
 * to the chosen variables and
 * the Prop is evaluated to true else false.
 *
 */
package ejvol3v2p;

import java.util.Random;

public class Prop2 {

    // Turn on  the random generator
    static Random r = new Random();

    //Number of operators
    private static final int NOPERATORS = 10;
    //Number of variables, could be as large as MAXV.
    //Variables will be represented by numbers.
    private static final int NVAR = 1800;

    //Proposition in formation

    //Relative proportions  of different operators.
    //The proportion of NEGATION is the complement to 1.
    private static final double PROBAND = 0.4;
    private static final double  PROBOR = 0.4;
    //Counters of occurrences of pseudo-variable l:
    //It counts the number of  occurrences of variables
    private static int numbL;
    //A truth assignment is kept here.
    private static final boolean TRUTH[]
            = new boolean[NVAR];
    private static final boolean PRINTDEFINEVALUATION = false;
```

```
private static String substitution(String a) {
    //Type of operator is chosen
    int type = -1;
    double h = r.nextDouble();
    if (h < PROBAND) {
        type = 0;
    }
    if ((h < PROBAND + PROBOR) & (h > PROBAND)) {
        type = 1;
    }
    if (h > PROBAND + PROBOR) {
        type = 2;
    }
    //number of l's is updated.
    if (type < 2) {
        numbL = numbL + 1;
    }
    //'f'  is replaced by an operator;
    switch (type) {
        case 0:
            a = a.replace("f", "AND(l,l)");
            break;
        case 1:
            a = a.replace("f", "OR(l,l)");
            break;
        case 2:
            a = a.replace("f", "NOT(l)");
            break;
    }
    return a;
}

//Composes an l-proposition    with m operators
//and various l's.
private static String lProposition() {
    System.out.println("Construction of "
            + "the l-proposition");
    //Proposition is initialized.
    String a = "l";
    //Number of l's is recorded.
    numbL = 1;
```

```
        //m operators are generated
        for (int i = 0; i < NOPERATORS; i++)
        {
            //A place of insertion is chosen:
            //a given l must be replaced by the operator.
            int givenl = r.nextInt(numbL);
            int counter = 0;
            for (int j = 0; j < a.length(); j++)
            {
                if (a.charAt(j) == 'l') {
                    if (counter == givenl) {
                        //System.out.println("Old a = " + a);
                        //the place of replacement is marked with 'f'
                        a = a.substring(0, j) + "f"
                                + a.substring(j + 1);
                        //System.out.println("New a = " + a);
                    }
                    counter = counter + 1;
                }
            }
            a = substitution(a);
            System.out.println("l-prop      = " + a);
        }
        return a;
    }

    //Each l of the l-proposition is replaced
    //by one of the variables.
    private static String proposition(String a) {
        System.out.println();
        System.out.println("l-prop --> proposition");
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                Integer vari = r.nextInt(NVAR);
                String s = vari.toString();
                a = a.substring(0, j) + s
                        + a.substring(j + 1);
            }
        }
        System.out.println("prop        = " + a);
        return a;
    }


    //Variables are given truth values
    private static void defineValuation() {
```

```
    if (PRINTDEFINEVALUATION)
        System.out.println("\n\nTruth assignment = " );
    for (int i = 0; i < NVAR; i++) {
        int j = r.nextInt(2);
        //true else false
        //TRUTH[i] = (j == 0);
        TRUTH[i] = j == 0;
        char truthValue = 'F';
        if (TRUTH[i]) truthValue = 'T';
        if (PRINTDEFINEVALUATION)
            System.out.println("Var " + i + "   "
                +  truthValue );
    }
    System.out.println();
}



//The proposition is evaluated
private static String evaluation(String prop) {
    //The length  of prop varies with each operation so,
    //we work with the original prop
    String prop0 = prop;
    System.out.println("Truth evaluation begins");
    System.out.println("Prop = " + prop);
    int j = 0;

    while (j < prop0.length())
    {
        //Substrings that encode for an int are parsed
        String myNumber = "";
        Character myChar = prop0.charAt(j);
        while (Character.isDigit(myChar))
        {
            myNumber = myNumber + myChar  ;
            j++;
            myChar = prop0.charAt(j);
        }

        if (myNumber.contentEquals(""))j++;
        else
        {
            //Var's names are replaced by truth values
            System.out.println("myNumber = " + myNumber );
            int k = Integer.parseInt(myNumber );
             System.out.println("k  = " + k + " -> " + TRUTH[k] );
```

```
                if (TRUTH[k] == true) {
                    prop = prop.replaceFirst(myNumber, "T");
                } else {
                    prop = prop.replaceFirst(myNumber, "F");
                }
                System.out.println("Truth value = " + prop);
            }
        }
        System.out.println("Truth value = " + prop);
        return prop;
    }



    //The proposition is evaluated until the end
    private static String simplification(String b) {
        while (b.length() > 1) {
            b = b.replace("AND(T,T)", "T");
            b = b.replace("AND(T,F)", "F");
            b = b.replace("AND(F,T)", "F");
            b = b.replace("AND(F,F)", "F");

            b = b.replace("OR(T,T)", "T");
            b = b.replace("OR(T,F)", "T");
            b = b.replace("OR(F,T)", "T");
            b = b.replace("OR(F,F)", "F");

            b = b.replace("NOT(T)", "F");
            b = b.replace("NOT(F)", "T");
            System.out.println("Truth value = " + b);
        }
        return b;
    }

    public static void main(String[] args) {
        String number = "1012123";
int result = Integer.parseInt(number) +5;
System.out.println(result);

        String b = lProposition();
        b = proposition(b);
        defineValuation();
        b = evaluation(b);
        b = simplification(b);
        System.out.println("Final evaluation = " + b);
```

```
    }
}//End of Program C81, Prop2
```

82, page 182. The following code takes constant numbers of operators and variables, generates, say, 1000 propositions at random and inquire over the fraction of unsatisfiable propositions. To decide whether or not a prop is satisfiable, we make a given number of random truth assignments. The dissatisfaction ratio for each proposition (the proportion of final false values) is also reported. The general tendency is that for a given number of operators, the more variables are included the less probably becomes to be unsatisfiable. Or: with more variables, there are more chances to get true.

```
/* Program C82, PropStat.
 * We begin with m operators and n variables
 * chosen at random.
 * Random   equiprobable truth values are assigned
 * to the chosen variables and
 * the prop is evaluated to true else false.
 *
 * A facility for statistics, especially suited
 * to study the SAT-isfaction problem, has been added.
 * Given   constant numbers of operators and variables,
 * we generate some propositions at random and
 * inquire over the fraction of
 * unsatisfiable propositions.
 * To decide whether or not a prop is satisfiable,
 * we make a given number of random truth assignments.
 * The  dissatisfaction ratio (the proportion of
 *  final false values) is reported for each prop.
 *
 */
package ejvol3v2p;

import java.util.Random;

public class PropStat {
// Turn on  the random generator

    static Random r = new Random();

    //Number of operators
    private static int nOperators;
    //Number of variables, could be as large as MAXV.
    //Variables will be represented by numbers.
    private static int nVar;
    private static final int NPROPOSITIONS = 10;
    //Proposition in formation
    private static String c;
```

```
//Relative proportions  of different operators
private static double probA, probO;
private static int numbL;
private static  boolean TRUTH[]   ;

//Number of random assignments for each proposition
private static int nAssignments;
private static double disSatFraction;
private static final boolean PRINTDEFINEVALUATION = false;
private static final boolean PRINTEVALUATION = false;



private static void setParameters() {
    //Number of operators
    nOperators = 10;
    //Number of variables
    nVar = 2;
    TRUTH   = new boolean[nVar];
    //Relative proportions of operators
    probA = 0.8; //And
    probO = 0.1; //Or
    //nAssignments = 7* (int)  Math.pow(2, nVar); //7*2^n
    nAssignments = 1000;
}


private static void printParameters()
{
    double probN = 1 - probA - probO;
    System.out.println("Number of operators = "
            + nOperators
    + "\nNumber of variables = " + nVar
    + "\nProbability of AND = " + probA
    + "\nProbability of OR = " + probO
    + "\nProbability of negation = " +  probN
    + "\nNumber of propositions = " + NPROPOSITIONS
    + "\nNumber of assignments per prop = "
            + nAssignments);
}


private static String substitution(String a) {
    //Type of operator is chosen
    int type = -1;
    double h = r.nextDouble();
```

```
    if (h < probA) {
        type = 0;
    }
    if ((h < probA + probO) & (h > probA)) {
        type = 1;
    }
    if (h > probA + probO) {
        type = 2;
    }
    //number of l's is updated.
    if (type < 2) {
        numbL = numbL + 1;
    }
    //'f'  is replaced by the operator;
    switch (type) {
        case 0:
            a = a.replace("f", "AND(l,l)");
            break;
        case 1:
            a = a.replace("f", "OR(l,l)");
            break;
        case 2:
            a = a.replace("f", "NOT(l)");
            break;
    }
    return a;
}

//Composes an l-proposition    with m operators
//and various l's.
private static String lProposition() {
    //System.out.println("Construction of the l-proposition");
    //Proposition is initialized.
    String a = "l";
    //Number of l's is recorded.
    numbL = 1;
    //m operators are generated
    for (int i = 0; i < nOperators; i++) {
        //A place of insertion is chosen:
        //a given l must be replaced by the operator.
        int givenl = r.nextInt(numbL);
        int counter = 0;
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                if (counter == givenl) {
                    //System.out.println("Old a = " + a);
```

```
                    //the place of replacement is marked with 'f'
                    a = a.substring(0, j)
                            + "f" + a.substring(j + 1);
                    //System.out.println("New a = " + a);
                }
                counter = counter + 1;
            }
        }
        a = substitution(a);
        //System.out.println("l-prop      = " + a);
    }
    return a;
}

//Each l of the l-proposition is replaced
//by one of the variables.
//Variables are not letters;
//they are numbers from 0 to n-1;
private static String proposition(String a) {
    //System.out.println();
    //System.out.println("l-prop --> proposition");
    for (int j = 0; j < a.length(); j++) {
        if (a.charAt(j) == 'l') {
            Integer vari = r.nextInt(nVar);
            String s = vari.toString();
            a = a.substring(0, j) + s
                    + a.substring(j + 1);
        }
    }
    //System.out.println("\nProposition = " + a);
    return a;
}

//Variables are given truth values
private static void defineValuation() {
    if (PRINTDEFINEVALUATION)
        System.out.println("\n\nTruth assignment = " );
    for (int i = 0; i < nVar; i++) {
        int j = r.nextInt(2);
        //true else false
        //TRUTH[i] = (j == 0);
        TRUTH[i] = j == 0;
        char truthValue = 'F';
        if (TRUTH[i]) truthValue = 'T';
        if (PRINTDEFINEVALUATION)
            System.out.println("Var " + i + "   "
```

```
                +  truthValue );
        }
        if (PRINTDEFINEVALUATION)
            System.out.println();
    }

    //The proposition is evaluated
    private static String evaluation(String prop) {
        //The length  of prop varies with each operation so,
        //we work with the original prop
        String prop0 = prop;
        if (PRINTEVALUATION)
        {
            System.out.println("Truth evaluation begins");
            System.out.println("Prop = " + prop);
        }
        int j = 0;

        while (j < prop0.length())
        {
            //Substrings that encode for an int are parsed
            String myNumber = "";
            Character myChar = prop0.charAt(j);
            while (Character.isDigit(myChar))
            {
                myNumber = myNumber + myChar  ;
                j++;
                myChar = prop0.charAt(j);
            }

            if (myNumber.contentEquals(""))j++;
            else
            {
                //Var's names are replaced by truth values
                if (PRINTEVALUATION)
                    System.out.println("myNumber = " + myNumber );
                int k = Integer.parseInt(myNumber );
                if (PRINTEVALUATION)
                    System.out.println("k  = " + k + " -> "
                            + TRUTH[k]);

                if (TRUTH[k] == true) {
                    prop = prop.replaceFirst(myNumber, "T");
                } else {
                    prop = prop.replaceFirst(myNumber, "F");
                }
```

```
                if (PRINTEVALUATION)
                    System.out.println("Truth value = " + prop);
            }
        }
        if (PRINTEVALUATION)
            System.out.println("FINAL EVALUATION = " + prop);
        return prop;
    }

//The proposition is evaluated until the end
    private static String simplification(String b) {
        while (b.length() > 1) {
            b = b.replace("AND(T,T)", "T");
            b = b.replace("AND(T,F)", "F");
            b = b.replace("AND(F,T)", "F");
            b = b.replace("AND(F,F)", "F");

            b = b.replace("OR(T,T)", "T");
            b = b.replace("OR(T,F)", "T");
            b = b.replace("OR(F,T)", "T");
            b = b.replace("OR(F,F)", "F");

            b = b.replace("NOT(T)", "F");
            b = b.replace("NOT(F)", "T");
            // System.out.println("Truth value = " + b);
        }
        return b;
    }

    //A prop is constructed and nAssignments truth
    //assignments are done. The fraction of
    //false outputs is reported.
    private static void planSingle() {
        String b = lProposition();
        b = proposition(b);
        c = b;
        double counter = 0;
        for (int i = 0; i < nAssignments ; i++) {
            defineValuation();
            b = evaluation(b);
            b = simplification(b);
            if (b.equals("F")) {
                counter = counter + 1;
            }
            b = c;
        }
```

```
        disSatFraction = counter / nAssignments ;
        System.out.println("Fraction of unsatisfaction  = "
                + disSatFraction);
    }


    /* We generate at random NPROPOSITIONS props, and report the
     * fraction of them that is not satisfiable.
     * A prop is not satisfiable when there exists no
     * truth assignment that satisfies it.
     * If after nAssignments random trials, no satisfying
     * assignment has been found, the prop is declared
     * as not satisfiable */
    private static void planMultiple() {

        double counter = 0;
        for (int i = 0; i < NPROPOSITIONS; i++) {
            planSingle();
            if (disSatFraction == 1) {
                counter = counter + 1;
            }
        }
        double d = counter / NPROPOSITIONS;
        System.out.println("\nFraction "
                + "of unsatisfiable propositions  = " + d);

    }

    public static void main(String[] args) {
        setParameters();
        printParameters();
        planMultiple();
    }
}//End of Program C82, PropStat
```

86, page 191. The expected relation is fulfilled but with one complication: as the number of variables grows, the algorithm slows down. This is due to the fact that with more variables, there are more possibilities of embarrassing a conjunction with a false value. This experiment shows that satisfying assignments are indeed few amidst the multitude of non satisfying ones.

87, page 191. The next code answers to the question: How many random trials are necessary to find a satisfying assignment for a disjunctive normal full form, which is always satisfiable? The answer is that after an initial segment in which the parallelism of the various clauses tries to combat SAT, we have that the number of random trials to get a satisfying

assignment is exponential $e^{0.6n}$ with a tendency to increase the exponent. Verified for 5 clauses and a number of variables ranging from 1 to 16. This coincides with what is expected in average, 2^n.

```
/* Program C87, HowMany.
 * We take disjunctive normal forms,
 * which are always satisfiable, and inquire
 * over the number of necessary random trials
 * to get a satisfying assignment.
 *

 * The prob of negation of a variable is 0.5.
 * Random   equiprobable truth values are assigned
 * to the  variables and
 * the prop is evaluated to true else false.
 *
 * An statistical facility is added.
 */
package ejvol3v2p;

import java.util.Random;

public class HowMany {

    // Turn on  the random generator
    static Random r = new Random();

    //Number of clauses
    private static int nClauses;
    //Number of variables, could be as large as MAXV.
    //Variables will be represented by numbers.
    private static int nVar;
    private static final int MAXV = 1000;
    //Proposition in formation
    private static String a;
    private static String b;
    private static String c;

    private static int nProps = 1000;
    private static final boolean TRUTH[]
            = new boolean[MAXV * MAXV];
    //The variables assignment in recorded in VAR[]
    private static final int VAR[]
            = new int[MAXV * MAXV];
    private static boolean print = false;
    private static double alpha, nn;
```

```
private static int nTrials;
private static final int NTRIAL[]
        = new int[2 ^ MAXV];
private static boolean problem = true;

private static void setParameters() {
    //Number of clauses
    nClauses = 5;
    //Number of variables
    nVar = 8;
    //Number of propositions
    nProps = 10;
    alpha = nClauses;
    nn = nVar;
    alpha = nClauses / nn;
}

//Not operators are inserted with prob 0.5
private static String negations() {
    for (int j = 0; j < a.length(); j++) {
        if (a.charAt(j) == 'l') {
            int d = r.nextInt(2);
            if (d == 0) {
                a = a.substring(0, j) + "f"
                        + a.substring(j + 1);
            }
        }
    }
    a = a.replace("f", "Nl");
    return a;
}

//Composes a random normal disjunctive full l-proposition
// with nClauses clauses. The NOT operator is included for each l
//with probability 0.5.
private static String lProposition() {
    //System.out.println("Construction of the l-proposition");
    //Proposition is initialized.
    a = "(l";
    for (int i = 0; i < nClauses - 1; i++) {
        for (int j = 0; j < nVar - 1; j++) {
            a = a.concat("Al");
        }
        a = a.concat(") OR (l");
    }
    for (int j = 0; j < nVar - 1; j++) {
```

```
            a = a.concat("Al");
        }
        a = a.concat(")");
        if (print) {
            System.out.println("l-prop without NOT = "
                    + a);
        }
        negations();
        if (print) {
            System.out.println("l-prop with NOTs   = "
                    + a);
        }
        return a;
    }

    //Each l of the l-proposition is replaced by one of
    //the variables. All variables appear in each clause.
    //Variables are not letters;
    //they are numbers from 0 to nVar-1;
    private static String proposition(String a) {
        //System.out.println();
        //System.out.println("l-prop --> proposition");
        int u = 0;
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                Integer vari = u;
                String s = vari.toString();
                a = a.substring(0, j) + s + a.substring(j + 1);
                VAR[u] = vari;
                u = u + 1;
                if (u == nVar) {
                    u = 0;
                }
            }
        }
        if (print)
        {
            System.out.println("prop  = " + a);
        }
        return a;
    }

    //Variables are given truth values
    private static void defineValuation() {
        for (int i = 0; i < nVar; i++) {
            int j = r.nextInt(2);
```

```
            /*
   //Equivalent to
   if (j==0) TRUTH[i] = true;
    else TRUTH[i] = false;
             */
            //TRUTH[i] = (j==0);
            TRUTH[i] = j == 0;
        }
    }

//The proposition is evaluated
    private static String evaluation() {
        if (print) {
            System.out.println();
            System.out.println("Truth evaluation begins");
        }
        int u = 0;
        for (int j = 0; j < b.length(); j++) {
            if (b.charAt(j) == 'l') {
                if (TRUTH[VAR[u]] == true) {
                    b = b.substring(0, j) + "T"
                            + b.substring(j + 1);
                } else {
                    b = b.substring(0, j) + "F"
                            + b.substring(j + 1);
                }
                u = u + 1;
                if (u == nVar) {
                    u = 0;
                }
                //Restart counting for the next clause
                u = u % nVar;
            }
        }
        if (print) {
            System.out.println("Truth value = " + b);
        }
        return b;
    }

//The proposition is evaluated until the end
    //Not operators are evaluated at the
    //very beginning for they apply over the next
    //to the right variable.
    private static String simplification() {
        while (b.length() > 1) {
```

```
            b = b.replace("NT", "F");
            b = b.replace("NF", "T");
            if (print) {
                System.out.println("Truth value = " + b);
            }
            b = b.replace("TAT", "T");
            b = b.replace("TAF", "F");
            b = b.replace("FAT", "F");
            b = b.replace("FAF", "F");

            if (print) {
                System.out.println("Truth value = " + b);
            }

            b = b.replace("(T)", "T");
            b = b.replace("(F)", "F");

            if (print) {
                System.out.println("Truth value = " + b);
            }

            b = b.replace("T OR T", "T");
            b = b.replace("T OR F", "T");
            b = b.replace("F OR T", "T");
            b = b.replace("F OR F", "F");

            if (print) {
                System.out.println("Truth value = " + b);
            }
        }
        return b;
    }

    //A satisfiable prop is constructed and
    // a satiafiabe assignment is found.
    //the number of required trials is reported.
    private static void task() {
        print = false; //printing instruction
        nTrials = 0;
        problem = true;
        b = lProposition();
        c = b;
        proposition(b);
        while ((problem)) {
            defineValuation();
            evaluation();
```

```
            b = simplification();
            nTrials = nTrials + 1;
            if (b.equals("T")) {
                problem = false;
            } else {
                b = c;
            }
        }
    }

    public static void report1() {
        System.out.println("nClauses = " + nClauses);
        System.out.println("nVar = " + nVar);
        System.out.print("alpha = " + alpha);
        System.out.println();
    }

    /* We generate at random nProps satisfiable props,
    * and report the mean and deviation of the
    * necessary number of random trials to get
    * a satisfying assignment.
    */


    public static void main(String[] args) {
        //The random generator is commanded
        //to produce the same sequence.
        //r.setSeed(23);
        setParameters();

        for (int i = 0; i < nProps; i++) {
            //System.out.println("prop = " + i);
            problem = true;
            task();
            NTRIAL[i] = nTrials;
        }
        double sumT = 0;
        for (int i = 0; i < nProps; i++) {
            sumT = sumT + NTRIAL[i];
        }
        double meanT = sumT / nProps;
        double sumDT = 0;

        for (int i = 0; i < nProps; i++) {
            sumDT = sumDT + Math.pow((NTRIAL[i] - meanT), 2);
        }
```

```
        double stanDevT = Math.sqrt(sumDT / nProps);
        report1();
        System.out.println("tested props = " + nProps);
        System.out.println("Mean numb Trials = " + meanT);
        System.out.println("stanDev numb trials = "
                + stanDevT);
        double l = Math.log(meanT);
        System.out.println("Log mean numb Trials = " + l);
    }
}//Program C87, HowMany
```

89, page 202. Our simulation of evolution does it worst than randomness. This was the conclusion allowed by programs `C89a HowMany2` and `C89b HowManyEvol2` whose outputs were pasted to LibreOffice. The following table shows the effect of the number of variables over the log of the number of random trials and over the log of the total number of individuals employed by the simulation of evolution:

Chance vs Evolution		
nVar	Rand	Evol
1	0	2,3
2	0,1	2,3
3	1,3	2,3
4	1,03	2,5
5	2	3
6	2,5	3,8
7	3,7	4,5
8	4	4,2
9	5,2	4,8
10	5,6	6,3
11	6,4	6,6
12	7,1	8
13	7,2	8,9
14	8	9,1
15	8,4	10,6
16	9,3	12,38

This data were graphicated to LibreOffice and it became visible that after an initial dissimilitude, both functions get nearly linear an parallel one to another, but that of evolution goes upwards nearly half a unit. Thus, the present simulation of evolution is $e^{0.5} = 1.6$ times more expensive than randomness but both run in the exponential regime.

91, page 203. What does happen if the number of individuals of the population is augmented? In general, the algorithm gets sensibly slower, needs more generations and

more individuals, because i big populations selection works in favor of stagnation and not of innovation.

96, page 217. The next table presents the log of the number of individuals required to solve SAT with the help of fuzzy logic for the specified number of variables (the number of individuals is proportional to the number of needed generations). We have taken a constant number of clauses = 5, first column with 10 individuals in the population per generation, second with 100. Recombination is allowed.

Fussy logic and log(cost) for solving SAT		
nVar	10-ind	100-ind
1	4,82	6,53
2	5,9	7,25
3	6,53	7,93
4	7,05	8,27
5	7,41	8,59
6	7,67	8,91
7	7,89	9,23
8	8,1	9,45
9	8,61	9,69
10	9,67	9,77
11	9,18	10,02
15	11,95	10,45
20	14,66	10,79
25	17,99	14,44
30	Infty	18,4

We observe that fuzzy logic is a burden for evolution. Passing a certain critical number of variables and given a constant number of clauses, the present implementation of evolution + fuzzy logic falls in a terrible exponential regime. Nevertheless, parallelism is given its expected helping value precisely for a large number of variables. We conclude that the present simulation does not solves SAT in less than exponential time.

101, page 229. Verified: the more you speak about something, the more likely you get contradicted (unless you always say the truth). For some few variables, 2 or 3, this happens rather soon, but as the number of variables increases, contradictions can be somehow delayed by invoking too many diverse facts.

105, page 242. Taking the number of total variables as 3, as α crosses from zero to 60, the probability of unsatisfiability grows along a sigmoidal curve from almost zero, before α = 4, to 1, near to α =60. So, we verified what we already know: when the number of clauses grows in relation to the number of variables, the discourse becomes more entangled and the probability of getting inconsistent also grows. A similar behavior is observed for a total number of variables equal to 5, but a difference exists: the curve of unsatisfiability rises up more slowly in the last case. It seems that with more pieces in the game, one can manage to have a better move.

106, page 243. The dependence of the probability of unsatisfiability on α is parametric: for a constant number of total variables, N, the higher is α the higher is the unsatisfiability. But the exact form of this dependence hangs on the number of variables: when N grows, unsatisfiability growth gets retarded.

109, page 244. The next program finds a satisfiable proposition and extends it. Satisfiability of the extended proposition is checked. It is readily verified that if the number of variables is kept constant then longer propositions are harder to extend.

```
/* Program C109, Extension.

 * We fabricate a  satisfiable conjunctive normal form,
 * extend it and check the extended proposition
 * for satisfiability.
 *
 * The prob of negation of a variable is 0.5.
 * Random   equiprobable truth values are assigned
 * to the  variables and
 * the prop is evaluated to true else false.
 *
 * A proposition is declared to be unsatisfiable if
 * after 2^(number of Variables) generations no
 * satisfiable assignment has  been found.
 *
 */
package ejvol3v2p;

import java.util.BitSet;
import java.util.Random;

public class Extension {

    // Turn on  the random generator
    static Random r = new Random();

    //Number of clauses
    private static int nClauses;
    //Number of clauses added in the Extension
    private static int nAddedClauses;
    //Number of variables of the original prop,
    //This number could be as large as MAXV.
    //Variables will be represented by numbers.
    private static int nVar;
    //Variables used in the Extension.
```

```
//By default, nNewVar = nVar;
private static int nNewVar;
//Proposition in formation
private static String a, b, c, d;

private static int nProps;

private static boolean print = false;
private static double alpha, nn;
private static double newAlpha;
private static boolean problem = true;
//if the prop is satisfiable, sat = true else false.
private static boolean sat = false;

//A population of binary numbers is considered.
//The number of individuals is constant
private static int numbInd;
private static final int MAXNUMBIND = 10000;
private static final BitSet IND1[] = new BitSet[MAXNUMBIND];
private static final BitSet IND2[] = new BitSet[MAXNUMBIND];
private static int FitnessInd[]
        = new int[MAXNUMBIND];
private static final int ORDERIND[] = new int[MAXNUMBIND];
private static int gen;
private static int maxGen, newMaxGen;
private static boolean recombination;

private static double mutationRate;

private static void setParameters() {
    //Number of clauses of the original prop
    nClauses = 7;
    //Number of variables of the original prop
    nVar = 3;
    //Number of added clauses
    nAddedClauses = 10;
    //Number of variables of the added clauses.
    //nNewVar >=nVar because
    //the first nvar variables used in the Extension are
    //the same variables of the original proposition.
    nNewVar = nVar;
    //Number of tested propositions
    nProps = 10;
    //Number of individuals per generation
    numbInd = 16;
    recombination = true; //true = recombination is enabled
```

```
        mutationRate = 0.8;
        alpha = nClauses;
        nn = nVar;
        alpha = nClauses / nn;
        nn = nNewVar;
        newAlpha = (nClauses + nAddedClauses) / nn;
        maxGen = (int) Math.ceil(Math.pow(2, nVar));
        newMaxGen = (int) Math.ceil(Math.pow(2, nNewVar));
        System.out.println("maxGen = " + maxGen);
    }

    //Not operators are inserted with prob 0.5
    private static String negations() {
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                int dd = r.nextInt(2);
                if (dd == 0) {
                    a = a.substring(0, j) + "f" + a.substring(j + 1);
                }
            }
        }
        a = a.replace("f", "Nl");
        return a;
    }

    //Composes a random normal disjunctive full l-proposition
    // with nClauses clauses. The NOT operator is included for each l
    //with probability 0.5.
    private static String lProposition(int nClauses, int nVar) {
        //System.out.println("Construction of the l-proposition");
        //Proposition is initialized.
        a = "(l";
        for (int i = 0; i < nClauses - 1; i++) {
            for (int j = 0; j < nVar - 1; j++) {
                a = a.concat("ORl");
            }
            a = a.concat(") A (l");
        }
        for (int j = 0; j < nVar - 1; j++) {
            a = a.concat("ORl");
        }
        a = a.concat(")");
        if (print) {
            System.out.println("l-prop without NOT = " + a);
        }
        negations();
```

```
        if (print) {
            System.out.println("l-prop with NOTs   = " + a);
        }
        return a;
    }

    //Each l of the l-proposition is replaced by one of
    //the variables. All variables appear in each clause.
    //Variables are not letters;
    //they are numbers from 0 to nVar-1;
    private static String proposition(String a) {
        //System.out.println();
        //System.out.println("l-prop --> proposition");
        int u = 0;
        for (int j = 0; j < a.length(); j++) {
            if (a.charAt(j) == 'l') {
                Integer vari = u;
                String s = vari.toString();
                a = a.substring(0, j) + s + a.substring(j + 1);
                u = u + 1;
                if (u == nVar) {
                    u = 0;
                }
            }
        }
        if (print) {
            System.out.println("prop               = " + a);
        }
        return a;
    }

//The proposition is evaluated
    private static String evaluation(BitSet s) {
        b = c;
        if (print) {
            System.out.println();
            System.out.println("Truth evaluation begins");
            System.out.println("prop = " + b);
        }
        int u = 0;
        for (int j = 0; j < b.length(); j++) {
            if (b.charAt(j) == 'l') {
                if (s.get(u) == true) {
                    b = b.substring(0, j) + "T" + b.substring(j + 1);
                } else {
                    b = b.substring(0, j) + "F" + b.substring(j + 1);
```

```
                }
                u = u + 1;
                if (u == nVar) {
                    u = 0;
                }
            }

        }
        if (print) {
            System.out.println("Truth value8 = " + b);
        }
        return b;
    }

//The proposition is evaluated until the end.
    //Not operators are evaluated at the
    //very beginning for they apply over the next
    //to the right variable.
    private static String simplification(String b) {
        while (b.length() > 1) {
            b = b.replace("NT", "F");
            b = b.replace("NF", "T");
            if (print) {
                System.out.println("Truth value1 = " + b);
            }
            b = b.replace("TORT", "T");
            b = b.replace("TORF", "T");
            b = b.replace("FORT", "T");
            b = b.replace("FORF", "F");

            if (print) {
                System.out.println("Truth value2 = " + b);
            }

            b = b.replace("(T)", "T");
            b = b.replace("(F)", "F");

            if (print) {
                System.out.println("Truth value3 = " + b);
            }
            b = b.replace("T A T", "T");
            b = b.replace("T A F", "F");
            b = b.replace("F A T", "F");
            b = b.replace("F A F", "F");

            if (print) {
```

```
                System.out.println("Truth value4 = " + b);
            }

        }
        return b;
    }

//Converts a BitSet value into a String
    static private String decode(BitSet t) {
        String s = "";
        for (int i = 0; i < nVar; i++) {
            if (t.get(i) == true) {
                s = s + "1";
            } else {
                s = s + "0";
            }
        }
        return s;
    }

    //IND1 is a population of binary strings.
    //The BitSet class is used to instantiate
    //binary strings as individuals
    private static void initializePop(int nVar) {
        BitSet s = new BitSet();
        //IND1 is false everywhere
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) s.clone();
        }
        //Reinitialization with random truth values.
        for (int i = 0; i < numbInd; i++) {
            for (int j = 0; j < nVar; j++) {
                int l = r.nextInt(2);
                if (l == 1) {
                    IND1[i].set(j); //the j-th bit is true
                }
            }
            //System.out.println("IND1 : " + decode(IND1[i]));
        }
    }

// Does an individual   satisfy  the target?
    static private int assessInd(BitSet s) {
        String bb = evaluation(s);
        bb = simplification(bb);
        int fitness = 0;
```

```
        if (bb.equals("T")) {
            fitness = 1;
        }
        return fitness;

    }

// Individuals are sorted by fitness.
    private static boolean sortingInd() {
        int Champ;

        //FitnessIndc[] is a copy of FitnessInd[]
        //used as workbench
        int FitnessIndc[];
        FitnessInd = new int[MAXNUMBIND];
        FitnessIndc = new int[MAXNUMBIND];
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = 0;
            FitnessIndc[i] = 0;
        }
        for (int i = 0; i < numbInd; i++) {
            FitnessInd[i] = assessInd(IND1[i]);
            FitnessIndc[i] = FitnessInd[i];
        }
        //We sort individuals by fitness
        // 0 is minimum fitness
        for (int i = 0; i < numbInd; i++) {
            Champ = 0;
            for (int j = 0; j < numbInd; j++) {
                if (FitnessIndc[j] > FitnessIndc[Champ]) {
                    Champ = j;
                }
            }
            //The array Order keeps a record of fitness.
            //The fittest is number zero
            ORDERIND[i] = Champ;
            FitnessIndc[Champ] = 0;
        }
        // if Pattern has been guessed
        if (FitnessInd[ORDERIND[0]] == 1) {
            problem = false;
            sat = true;
            //System.out.println("Champ = " + IND1[ORDERIND[0]]);
            b = c;
        }
        return problem;
```

```
    }

// The new generation is made
    private static void newGenerationInd() {
        int i = 0;
        //The fittest individuals are reproduced
        // numbCopies times
        int numbCopies = 10;
        int k = 0;  //rank, 0 is the highest
        while (i < numbInd) {
            int counter = numbCopies;
            while ((counter > 0) & (i < numbInd)) {
                //System.out.println(i + "" + l);
                IND2[i] = (BitSet) IND1[ORDERIND[k]].clone();
                //System.out.println(i + " " + counter + " " + k);
                i++;
                counter = counter - 1;
            }
            k++;
        }
        for (int j = 0; j < numbInd; j++) {
            IND1[j] = (BitSet) IND2[j].clone();
            /*  System.out.println( j+ " IND1 "
                                    + (BitSet) IND1[j]);
             */
        }
    }

// Recombination is enabled
    private static BitSet recInd(int l) {
        //We choose a partner at random
        int k = r.nextInt(numbInd);
        //b is false everywhere
        BitSet bb = new BitSet();
        //B is the recombinant of l and k
        for (int j = 0; j < nVar; j++) {
            int m = r.nextInt(2);
            if (m == 1) //take the bit from l
            {
                if (IND1[l].get(j) == true) {
                    bb.set(j);
                }
            }
            if (m == 0)//take the bit from k
            {
                if (IND1[k].get(j) == true) {
```

```
                    bb.set(j);
                }
            }
        }
        return bb;
    }

// The definitive population  is a recombinant one.
    private static void recombinationInd() {
        //IND2 is a copy of IND1
        for (int i = 0; i < numbInd; i++) {
            IND2[i] = (BitSet) IND1[i].clone();
        }
        //IND2 is a recombinant version of IND1
        for (int l = 0; l < numbInd; l++) {
            IND2[l] = recInd(l);
        }
        //IND2 is transferred to IND1
        for (int i = 0; i < numbInd; i++) {
            IND1[i] = (BitSet) IND2[i].clone();
        }
    }

// Individual  a is mutated at a random place.
    private static BitSet pointMutationInd(BitSet a) {
        int l = r.nextInt(nVar);
        a.flip(l);
        return a;
    }

// Individuals are mutated
    private static void mutationInd() {
        for (int i = 0; i < numbInd; i++) {
            double p = r.nextDouble();
            if (p < mutationRate) {
                IND1[i] = pointMutationInd(IND1[i]);
            }
        }
    }

    // Given a goal, it is guessed by evolution
    private static void evolution() {
        //The new generation is produced
        //as copies of the fittest
        newGenerationInd();
        if (recombination) {
```

```
            recombinationInd();
        }
        //The new population is subjected to mutation
        mutationInd();
        gen++;
        problem = sortingInd();
        if (gen % 1000 == 0) {
            System.out.println(d + "Gen = " + gen);
        }
        if (gen == maxGen) {
            problem = false;
            sat = false;
        }
    }

// A satisfiable prop is tried out
    static private void makeATry() {
        problem = true;
        b = lProposition(nClauses, nVar);
        c = b;
        proposition(b);
        initializePop(nVar);
        gen = 0;
        sat = false;
        // A satisfying assignment is found
        while ((problem)) {
            evolution();
        }
    }

    //A satisfiable prop is invented
    static private void inventProp() {
        d = "Invent ";
        makeATry();
        if (sat == false) {
            inventProp();
        }
    }

    //The original prop is extended
    static private void extendProp() {
        System.out.println("Old prop = " + b);
        String e;
        if (nAddedClauses > 0) {
            e = lProposition(nAddedClauses, nNewVar);
            b = b + " A " + e;
```

```
        }
        System.out.println("Extended prop = " + b);
    }

    //The extended prop is checked for satisfiability
    static private void checkExtension() {
        d = "Check ext ";
        maxGen = Math.max(maxGen, newMaxGen);
        System.out.println("maxGen = " + maxGen);
        c = b;
        proposition(b);
        initializePop(nNewVar);
        gen = 0;
        sat = false;
        problem = true;
        while ((problem)) {
            evolution();
        }
    }

    public static void report1() {
        System.out.println();
        System.out.println("Original proposition");
        System.out.println("nClauses = " + nClauses);
        System.out.println("nVar = " + nVar);
        System.out.println("alpha = " + alpha);
        System.out.println("Extended proposition");
        System.out.println("nAddedClauses = " + nAddedClauses);
        System.out.println("nNewVar = " + nNewVar);
        System.out.println("Newalpha = " + newAlpha);
        System.out.println("Repetitions = " + nProps);
        System.out.println();
    }

    public static void main(String[] args) {
        print = false; //printing instruction
        setParameters();
        double nSat = 0;
        for (int i = 0; i < nProps; i++) {
            System.out.println("**********");
            System.out.println("Prop numb = " + i);
            problem = true;
            sat = false;
            inventProp();
            System.out.println("A SAT prop has been fabricated");
            extendProp();
```

```
            checkExtension();
            if (sat == true) {
                nSat = nSat + 1;
                System.out.println("Extended  prop is satisfiable");
            } else {
                System.out.println("Extended  prop is not satisfiable");
            }
            System.out.println("nSat = " + nSat);
        }

        report1();

        double rr = nProps;
        double p = nSat / rr;
        System.out.println("Proportion of Sat of extended prop = " + p);

    }
}//End of Program C109, Extension
```

111, page 244. With some patience, one can show that any satisfiable proposition can be extended in as many forms and lengths as desired. To that aim, add sufficiently many new variables.

??, page ??. The $NP >? P$ conundrum can be appreciated very easily from the evolutionary perspective: for a person that must be restricted to a boolean logic, the tree depiction of SAT looks simple, incompressible and necessary. But, what could happen if we immerse SAT in a gradualistic world where evolution can make marvels? We already have tried the idea of Zadeh on fuzzy logic and the K-MAX-SAT extension but that was of no help for us. Are they the only forms of embedding SAT into gradualism? Not, of course not, and nobody knows what will the future show us. But our problem is that no new idea with improving results has been brought forth up to date.

122, page 256. The first point is that the space of combinations is enormous. For instance, in terms of DNA, $4^{180} > 10^{100}$. In second place, unless you do something really important, DNA computers never will compete with electronic ones. One complication is that DNA computers are fueled by diffusion, which must be kept very slow to be reliable. And then comes the problem of reading computing solutions. This could be as daunting as the genome project. Else, you must be very smart and reformulate your problems as decision ones, which output the solution in 1 bit (Is that possible?).

123, page 256. Suppose we must solve SAT for a formula with n variables. Then, the tree to generate all valuations has branches n units long but has 2^n leaves. The length of branches corresponds to the number of operations that must be done one after another, to the time complexity. The number of leaves corresponds to the material resources that must be wasted to generate all possible valuations without augmenting the waste of time. Thus,

the state of the art to solve SAT is as follows: use a minimum of memory resources but an exponential number of serial operations else a minimal number of serial operations but an exponential number of memory allocations. One can also use intermediate solutions, say, a number of memory allocations that is polynomial in n but a less strong exponential regime in the time variable. Technically: P-space $>$ NP (Balcázar et al, l.c., [6] 1995). None of these solutions is useful to propel the almightiness of evolution: do not forget the restriction given by the dominant cosmological model, according to which the whole cosmos has existed for a finite number of years and henceforth, the reachable universe during that time is also finite. So, we say that our universe has less than 10^{100} atoms and an age of around 1015 years.

129, page 274. We have been stroke by the following results: 1) The distribution of fitness is closed around almost perfection from the very outset that was produced by randomness. So, chance might produce almost perfection and evolution adds very little. 2) Anyway, evolution produces valuations that satisfy as much as 6/7 of the number of clauses almost immediately. 3) Unsatisfiable propositions usually have as much as 6/7 of the number of clauses satisfied. With these facts on hand, we have a criterion to decide whether or not something has been the fruit of evolution: wherever you can find almost perfection with some few bugs that is followed by perfection, you have the right to claim that the studied system arouse by evolution. Our considerations lead us to a warning: we have preached that SAT is solved by evolution in the exponential regime but we have made too few trials to guarantee this for all subfamilies: what we know is that some instances of SAT run in the exponential regime as the proposition of the form $p_1 AND p_2 AND p_3 AND .... p_n$. This is the worst case. But many subfamilies might exist that run below, say tautologies $pORNOT\, p$, are solved inmediately. This is worth further investigation.

Bibliography

[1] ADLEMAN L (1994) Molecular computation of solution to combinatorial problems. *Science* 266: 1021–1024(Nov 11).
https://www.usc.edu/dept/molecular-science/papers/adleman-science.p
Verified 256

[2] ALLOY GROUP (2008) The Alloy Analyzer
http://alloy.mit.edu/
Verified 9/XI/2016 83

[3] AMMERMANN D (1985) Chromatin diminution and chromosome elimination : mechanism and adaptive significance. In The Evolution of Genome Size, edited by Cavalier -Smith, J. Wiley & Sons Ltd., pp 427–442. 245

[4] ARNEODO A, AUBENTON Y, BACRY E, GRAVES P, MUZY J, THERMES C, (1996), Wavelet based fractal analysis of DNA sequences, *Physica D* 96 291–320. 245

[5] ARORA S, LUND C (1995) Hardness of approximations. *Approximation algorithms for NP-hard problems*. Edited by Dorit S. Hochbaum, PWS Publishing Company, Boston, 399–440. 258

[6] BALCÁZAR J, DÍAZ J, GABARRÓ J (1995), Structural complexity I, Springer Verlag, Berlin. 250, 254, 372

[7] BERGTHORSSON U, RICHARDSON A, YOUNG G, GOERTZEN L, PALMER J (2004), Massive horizontal transfer of mitochondrial genes from diverse land plant donors to the basal angiosperm Amborella. *PNAS 2004* 101:17747–17752;
https://www.ncbi.nlm.nih.gov/pubmed/15598737
Verified 11/XI/2016 98

[8] BIOJAVA: http://biojava.org/
Cited 28 August 2008. 98

[9] BENNET C, GILL J (1981), Relative to a Random Oracle A, $P^A \neq NP^A \neq Co-NP^A$ with Probability 1 , *SIAM j. comput.* Vol. 10, 1, pp. 96-113.
http://epubs.siam.org/doi/abs/10.1137/0210008
Verified 254

[10] BLOCH J (2008), How to Design a Good API and Why it Matters
http://landawn.com/How%20to%20Design%20a%20Good%20API%20and%20Why%2
Verified 11/XI/2016 98

[11] BOYER P (2008), Religion: Bound to believe? Nature 455: 1038–1039.
http://www.nature.com/nature/journal/v455/n7216/full/4551038a.html
Verified 23/XI/2016 279

[12] CAICEDO X (1990), *Elementos de lógica y calculabilidad,* una empresa docente, Bogotá. 250

[13] CAVALIER-SMITH T,(1985), Introduction : the Evolutionary Significance of Genome Size. In *The Evolution of Genome Size*, edited by Cavalier -Smith, J. Wiley & Sons Ltd., 1–36. 245, 246

[14] ,CHAITIN G (1987), Algorithmic information theory. Cambridge University Press.
https://www.cs.auckland.ac.nz/~chaitin/cup.pdf
Verified 18/XI/2016 207

[15] CHOI I, KIM S (2007) Global extent of horizontal gene transfer. *PNAS* 2007 104:4489–4494;
http://www.pnas.org/content/104/11/4489
Verified 11/XI/2016 98

[16] CHOTHIA C (1992), One thousand families for the molecular biologists. *Nature* 337, 543–544.
http://www.nature.com/nature/journal/v357/n6379/abs/357543a0.html
Verified 247

[17] CROMIE W (1997), Snails Caught in Act of Evolution, The Harvard University Gazette:
http://www.hno.harvard.edu/gazette/1997/01.09/SnailsCaughtinA.html
Verified Cited 26 August 2008.

[18] DAWKINS R (1986) , *The blind watchmaker* W.W.Norton & Company, N.Y, London).
http://terebess.hu/keletkultinfo/The_Blind_Watchmaker.pdf
Revised 9/XI/2016.
2

[19] DEAMER D (2008), How leaky were primitive cells? Nature 454, 37–38.

Verified

[20] DRABEK CHRISTINE AND BRAUNL THOMAS(1998), L-systems.
ftp://ftp.informatik.uni-stuttgart.de/pub/lgrammar">http://www.informatik.
Cited Nov 2007. Link broken by 2016. Our code is more than enough to see how L-systems are implemented although these can also be embellished and this is not shown by us but was done by the cited link. 52

[21] EBBINGHAUS H, FLUM J, THOMAS W (1984), *Mathematical logic*, Springer, N.Y. 250

[22] EIGEN M, SCHUSTER P (1979), *The hypercycle: A principle of natural self-organization.* Springer. See abstract at:
http://pespmc1.vub.ac.be/ORIGIN.html
Verified

[23] FOOT B, YODER J (1995), Architecture, evolution and metamorphosis
http://www.laputan.org/metamorphosis/metamorphosis.html
Verified 9/XI/2016 83

[24] FOOT B, YODER J (1997), The selfish class
http://www.laputan.org/selfish/selfish.html
Verified 9/XI/2016

[25] FOOT B, YODER J (1999) Big Ball of Mud.
http://www.laputan.org/mud/mud.html
Verified 10/XI/2016 82, 97

[26] GEHRING W (2005), New Perspectives on Eye Development and the Evolution of Eyes and Photoreceptors. *Journal of Heredity* 2005 96(3):171–184; doi:10.1093/jhered/esi027
http://jhered.oxfordjournals.org/content/96/3/171.abstract
Verified 205

[27] GENETIC PROGRAMMING ORG (2008),
http://www.genetic-programming.org/
Verified 24/XI/2016 283

[28] GOGNIAT A (1998), Biomorphs
http://www.cs.bgu.ac.il/~sipper/biomorphs/
Verified Cited 8 july 2007. 3

[29] HALL L (1995), Approximation algorithms for scheduling. *Approximation algorithms for NP-hard problems*, edited by Dorit S. Hochbaum, PWS Publishing Company, Boston. 1–45.
258

[30] HANSEN D (1987), Enzyme Catalysis. Antibodies with some Bite. *Nature* 325: 304.
http://www.nature.com/nature/journal/v325/n6102/abs/325304a0.html
Verified 16/XI/2016 128

[31] HARMON R, HÔTE C (2006) Automatic Engine Control Code Generation with Integrated Automatic Static Code Verification.
http://www.mathworks.fr/industries/auto/iac06/presentations/10-Harm
Verified Cited 1 july 2008.

[32] HOOD L, GALAS D (2003), The digital code. *Nature* 421: 44–448.
http://www.nature.com/nature/journal/v421/n6921/full/nature01410.ht
Verified 98

[33] HOCHBAUM D (1995), Various notions of approximations: Good, better, Best, and More. *Approximation Algorithms for NP-hard Problems*, edited by Dorit S. Hochbaum, PWS Publishing Company, Boston, 346–394. 258, 281

[34] HOWARD-ASHBY M, MATERNA S, BROWN C., TU Q, OLIVERI P, CAMERON R., AND DAVIDSON E (2006), High regulatory gene use in sea urchin embryogenesis: Implications for bilaterian development and evolution *Dev Biol. December 1; 300(1): 27-34*.
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1790870/
Verified 11/XI/2016 99

[35] HUI HNT, JC HO, HB CHAN, SCF TAM, NLS TANG, PLS IP (2004), A baby with nonketotic hyperglycinaemia, HK J Paediatry (new series), 9: 243-247.
http://www.hkjpaed.org/pdf/2004;9;243-247.pdf
Verified 23/XI/2016 276

[36] HUYNEN M, STADLER P, FONTANA W (1996) , Smoothness within ruggedness: The role of neutrality in adaptation, *Proc. Natl. Acad. Sci.USA*, Vol 93, pp. 397-401.
http://www.pnas.org/content/93/1/397.full.pdf
Verified 16/XI/2016 134

[37] IHMELS J, COLLINS S, SCHULDINER M, KROGAN N,WEISSMAN J (2007), Backup without redundancy: genetic interactions reveal the cost of duplicate gene loss. *Molecular Systems Biology* 3: 86 doi:10.1038/msb4100127.
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1847942/
Verified 17/XI/2016 161, 246

[38] KAUFFMAN S (1995), At home in the universe. Oxford University Press, NY. vii, 103, 111, 285

[39] KIMURA M (1993) *The Neutral Theory of Molecular Evolution*, 1993, Cambridge University Press.

Verified

[40] KOSCHKE R (2008), Identifying and Removing Software Clones. In: Software Evolution. Mens T, Demeyer S (Eds.), Springer, pp 15-36.

Verified

[41] KUHLMAN B, DANTAS G, IRETON G, VARANI G, STODDARD B, BAKE D (2003), Design of a Novel Globular Protein Fold with Atomic-Level Accuracy. Science 302, 1364–1368.
http://science.sciencemag.org/content/302/5649/1364
Verified 247

[42] KUHN T (1970), The Structure of Scientific Revolutions, 2nd. ed., Chicago: Univ. of Chicago.
http://projektintegracija.pravo.hr/_download/repository/Kuhn_Structure_of_
Verified 244

[43] LEVY E, BOERI E, ROBINSON C, TEICHMANN S (2008), Assembly reflects evolution of protein complexes. *Nature* 453, 1262-1265.
http://www.nature.com/nature/journal/v453/n7199/full/nature06942.ht
Verified 97

[44] LOY M, ECKSTEIN R, WOOD D, ELLIOT J, COLE B (2003), *Java Swing*. O'Reilly, Cambridge. 3

[45] LEE J (2003), Molecular biology: Complicity of gene and pseudogene. *Nature* 423, 26–28 — doi:10.1038/423026a
http://www.nature.com/nature/journal/v423/n6935/full/423026a.html
Verified 245

[46] MCCUE M (2007), Prey envenomation does not improve digestive performance in western diamondback rattlesnakes (Crotalus atrox). *J. Exp. Zool*. 307A:568-577, 2007.
http://onlinelibrary.wiley.com/wol1/doi/10.1002/jez.411/abstract
Verified 23/XI/2016 249

[47] MENS T(2008), Introduction and Roadmap: History and Challenges of Software Evolution. In: Software Evolution. Mens T, Demeyer S (Eds.), Springer, pp 1–14. 97

[48] MENS K, TOURWE T (2008), Evolution Issues in Aspect-Oriented Programming. In: Software Evolution. Mens T, Demeyer S (Eds.), Springer, pp 203–232.

Verified

[49] MEYER A (2003), Duplication, duplication. *Nature* 421: 31–32.
http://www.nature.com/nature/journal/v421/n6918/full/421031a.html
Verified 17/XI/2016 161, 247

[50] MICHEL J, YEH P, CHAIT R, MOELLERING R, KISHONY R (2008), Drug interactions modulate the potential for evolution of resistance. *PNAS*, 105, 14918-14923.
http://www.pnas.org/content/105/39/14918.full
Verified 203

[51] MONASSON R, ZECCHINA R, KIRKPATRICK S, SELMAN B, TROYANSKY L (1999), Determining computational complexity from characteristic phase transitions', Nature 400: 133–137.
http://www.nature.com/nature/journal/v400/n6740/full/400133a0.html
Verified 22/XI/2016 243, 258

[52] MONTGOMERY R, DRYER R, CONWAY T, SPECTOR A (1983) , Biochemistry, a case study. Fourth edition. The CV Mosby Company, Toronto. 248

[53] MURUKAWA M, YOSHIZAWA S, HIGUCHU T, (1997), "Adaptive Equalization of Digital Communication Channels Using Evolvable Hardware", in : *Evolvable Systems: From Biology to Hardware*, edited by Higuchi, Iwata and Liu, Springer, pp 379-389.
https://pdfs.semanticscholar.org/0aba/2037fca109cdc4a45f6903022c032
Verified 249

[54] NILSSON D, PELGER S (1994), A Pessimistic Estimate of the Time Required for an Eye to Evolve. *Proc R Soc Lond* B 256: 53-58. doi:10.1098/rspb.1994.0048.
http://rspb.royalsocietypublishing.org/content/256/1345/53
Verified 205

[55] OCHOA GABRIELA (1998), L-systems.
http://www.biologie.uni-hamburg.de/b-online/e28_3/lsys.html
Link was broken by 2016. Please, give a look at
http://mathworld.wolfram.com/LindenmayerSystem.html
Revised 9/XI/2016 51

[56] ORENGO C, JONES D, THORNTON J (1994), Protein superfamilies and domain superfolds (1994) Nature 372, 631–634.
http://www.nature.com/nature/journal/v372/n6507/abs/372631a0.html
Verified 247

[57] PATTEN B (1958), Foundations of embryology. McGraw-Hill, N.Y. 97

[58] PIERCE N, WINFREE E (2002), Protein Design is NP-hard. *Protein Engineering* vol.15 no.10 pp.779-782.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.132.8486&rep=rep1
Verified 22/XI/2016 255

[59] POLLACK S, JACOBS J, SCHULTZ P(1986), Selective Chemical Catalysis by an Antibody , *Science* 234, 4783, 1570–1573.
http://science.sciencemag.org/content/234/4783/1570
Verified 16/XI/2016 128

[60] RADZICKA A (1995), A Proficient Enzyme. *Science*, 267, pp. 90–93.
https://www.ncbi.nlm.nih.gov/pubmed/7809611
Verified 101

[61] REIDYS C, KOPP S, SCHUSTER P (1997), Evolutionary Optimization of Biopolymers and Sequence Structure Maps. In : *Artificial Life V*, edited by Langton and Shimohara, The MIT Press. 134

[62] RODRÍGUEZ J (2003), NP-harsh problems in education, *International Journal of Mathematical Education in Science and Technology*, v34, n4, p537-547. EJ770423.
http://www.tandfonline.com/doi/abs/10.1080/0020739031000108547
Verified 23/XI/2016 257, 258

[63] RODRÍGUEZ J (2007), On peacebuilding and individual psychology. *Antípoda* 5, 209–237. In Spanish.
https://antipoda.uniandes.edu.co/view.php/72/index.php?id=72
Verified

[64] SHABAT D, ITZHAKY H, REYMOND J, KEINAN E (1995), Antibody Catalysis of a Reaction Otherwise Strongly Disfavoured in Water. *Nature* 374: 143–146.
http://www.nature.com/nature/journal/v374/n6518/abs/374143a0.html
Verified 16(XI/2016 128

[65] SMALE S (1998), Mathematical Problems for the Next Century, *The Mathematical Intelligencer*, Vol. 20, 2, 7-15.
http://www6.cityu.edu.hk/ma/doc/people/smales/pap104.pdf
Verified 254

[66] SNEPPEN KIM, MICHEELSEN M AND DODD I (2008), *Nucleosome Mediated Transcription Regulation*
http://cmol.nbi.dk/models/epitrans/transcript.html
Verified 11/XI/2016 99

[67] STEELS L (1997), Self-Organizing Vocabularies, In: *Artificial Life V*, Edited by Langton and Shimohara, The MIT Press, 179-184. 132

[68] STEWARD J , BENKOVIC S (1995), Transition-state stabilization as a measure of the efficiency of antibody catalysis. *Nature*, vol. 375: 388–391.
http://www.nature.com/nature/journal/v375/n6530/abs/375388a0.html
Verified 101, 128

[69] TOURLAKIS G (1984), Computability. Reston Publishing Company Inc, Reston, Virginia. 259

[70] TAYLOR W (2002), A 'periodic table' for protein structures. Nature 416, 657–660.
http://www.nature.com/nature/journal/v416/n6881/full/416657a.html
Verified 247

[71] TRAMONTANO A, JANDA K, LERNER R (1986), Catalytic Antibodies. *Science* 234, 4783, pp. 1566-1570.
http://science.sciencemag.org/content/234/4783/1566
Verified 16/XI/2016 128

[72] VETSIGIAN K, WOESE C, AND GOLDENFELD N (2006) Collective evolution and the genetic code. *Proc. Nat. Acad. Sci. USA* Vol.103, No 28: 10696–10701.
http://www.pnas.org/content/103/28/10696.full
Verified 24/XI/2016 283

[73] WIENER LAUREN (1993) Digital Woes. Addison-Wesley, Massachusetts.

Verified

[74] WIKIPEDIA CONTRIBUTORS, "FUZZY LOGIC," WIKIPEDIA, THE FREE ENCYCLOPEDIA
https://en.wikipedia.org/w/index.php?title=Fuzzy_logic&oldid=745662
Verified 19/XI/2016. 208

[75] WIKIPEDIA CONTRIBUTORS. SNAKE VENOM [INTERNET]. WIKIPEDIA, THE FREE ENCYCLOPEDIA;
https://en.wikipedia.org/w/index.php?title=Snake_venom&oldid=749695
Verified 22/XI/2016. 248

[76] ZADEH L (1965), Fuzzy sets, Information and Control 8, 338–353.
https://people.eecs.berkeley.edu/~zadeh/papers/Fuzzy%20Sets-Information%20
Verified 19/XI/2016 208

[77] ZAMENHOF S, EICHHORN H, (1962), Study of Microbial Evolution through Loss of Biosynthetic Functions. Establishment of "Defective" Mutants. *Nature*. 216: 456–458.
http://www.nature.com/nature/journal/v216/n5114/abs/216456a0.html
Verified 245

[78] ZSCHOCKE J, HOFFMANN G (1999), Vademecum Metabolicum, Manual of Metabolic Paediatrics. Milupa, Schattauer, Stuttgard, NY. 276

Index

MAX-K-SAT=Maximum K-satisfiability, 230, 258

algorithmic complexity theory, 250
autocatalytic, 112
AWT (Abstract Window Toolkit), 3

biogenesis law, 97
biogenesis style of software design, 284
bug, 2, 259

chaotic nature of software, 132
complexity, 260
conjunctive normal form, 219
context dependence, 138
creationism, 284

developer, vii, 283
distance
    Hamming, 161
    NK orphan , 162

evolution
    the fatigue of , 244
evolutionary cycle of software, 103
evolutionary environment, 288
    gradualistic , 206

formula, 173
function, 100, 218
fuzzy logic, 208

genetic code, vii, 283
genetic programming, 283
grammar, 132, 138
graph, 85
GUI (Graphical User Interface), 1

K-SAT, 230

landscape, 103
laws of Morgan, 218
laws of software design, 277
length of the input, 250
life
    first version of , 119, 127
Lindenmayer system , 51

machine
    deterministic, 250
    non-deterministic, 250
Markov algorithms, 259
menu, 38
menus, 38
minimum bisection problem, 85
morphs of Dawkins, 2
Muller's ratchet, 307

NK-model, 104
NP, 251
NP- harsh, 258

P, 251
Post Systems, 259
problem
    decision, 251
property, 218
proposition, 173
protein design, 255
purpose, 2, 100

SAT, viii, 173, 174, 282, 286, 290
satisfiable, 174
Saving a graphic, 47
self-organization, 111
Semantics, 132
semantics, 135
software, vii, 283
software discontinuity, 132
species, 97
Syntax, 132

the complexity of an algorithm, 250

theory of evolution, 203
transparent style, 99
turtle, 51

verbal instruction, 250