JAVA FOR THE STUDY OF EVOLUTION

VOLUME XI

More trees

José del Carmen Rodríguez Santamaría
The EvolJava Community

Contents

1 Evolution of tree topologies **1**

1.1 Trees come in species 2
1.2 Implementing mutation over tree topologies 5
1.3 Recombination of tree topologies. 26
1.4 Evolution to the full 64
1.5 Conclusion 220

2 The Lamarck recursion **221**

2.1 Step one: selfexamination 221
2.2 Step two: examination of traps 245
2.3 Step three: examination of new traps... 260
2.4 Step four: examination of new traps... 267
2.5 Conclusion 269

3 Why is the tree problem so difficult? **271**

3.1 Our proposal 272
3.2 Relation with genetics 274
3.3 Conclusion 275

4 Evolution vs. pleitropy **277**

4.1 Introduction and purpose 277
4.2 Darwinian evolution vs. Lamarckian evolution 278
4.3 The Lamarck recursion 280
4.4 Our problem is very difficult 281
4.5 Why is our problem so difficult? 282
4.6 Relation with genetics 283
4.7 Relation with evolution 286
4.8 Conclusion 288

Answers to exercises **291**
4.9 nothing 566

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.

This is the eleventh volume: More trees. We arm evolution with varied hints to attack the problem of fitting a tree to a distance matrix. Its performance is assessed.

Bogotá, Colombia, *José Rodríguez*
December 2013

Introduction

We challenged evolution in Volume X with an important problem: to find the perfect matching of a tree to a distance matrix. Our efforts were insufficient to get an algorithm with acceptable performance. This is no surprise because our prior experience have clearly showed us that evolution can be programmed as bad as desired and that most surely any current program competes for being the worst.

This terrible truth has the following implication: we have plenty of space to search for powerful hints to help evolution to become a marvelous solution to the problem that worries us. We are departing, of course, from Darwinian evolution and coming into the realm of Lamarckian or helped evolution.

Our first task to hint evolution is to solve the problem of how to mutate and recombine tree topologies, a task that will allow us to implement evolution to the full. This is done in chapter I. Since we notice a very small improvement in performance, we generate varied hints in chapter II to help evolution. So long as we perceived no important improvement, we examine in chapter III the cause of the difficulty of the studied problem. We find that it is an extreme degree of **pleitropy** that results when every gene affects every observable character. As a direct result of our failure and in the light of the extreme complexity of life, the idea that evolution could explain the origin of the generality of species appears to be ridiculous. Nevertheless, a better understanding shows that it is actually sane to suggest that evolution explains the origin of species: pleitropy in diverse degrees predicts a core of invariant characters plus a variable part, whose components may evolve with diverse degrees of velocity. This is compatible with the life we see. Nevertheless, we state in chapter IV, our summary, that the Evolutionary Theory of the origin of species is obviously false: pleitropy predicts abundant and long standing cases of mediocrity as in the fossil record as in modern populations but nothing like that is found in nature.

Chapter 1

Evolution of tree topologies

`Evolution to the full`

1 ***Introduction.*** *We learned in Volume X how to use various versions of Java trees to simulate evolution. We defined a* ***tree topology*** *as a string that contains the information of how nodes are linked. If we add lengths to the links of a tree topology, we get a* ***tree****.*

Our main aim in Volume X was to challenge evolution with an important problem: to find the tree that best represents a distance matrix. We use as test the problem in which the distance matrix was derived from a tree and we demanded from the program to output the tree with which we began.

We tried out a solution by scanning all possible tree topologies, one by one, to find the best tree in within each chosen topology. These bundle of optimization problems were solved by evolution. After certain number of generations the algorithm was stopped and the best topology was declared to be that with the optimal tree. The problem of scanning all possible topologies encountered a combinatorial explosion because of which that problem was classified as intractable and for a desktop this happens rather early, when the tree have just 6 to 10 leaves. Nevertheless, it was found that the best topology could be singled out by comparing performances during, say, the first 150 generations. So, to improve the algorithm we included two phases. The first was to scan all possible topologies during some 100 generations to single out the best one. The second was to run the best topology to find the best tree.

Conclusion: Our improved algorithm was good to solve the posed problem with 3, 4 and 5 leaves. Running time was so long beyond this that the algorithm was declared to be inappropriate for a desktop.

2 ***Objective.*** *The immediate explanation of our failure is that evolution was not used to its full power. In fact, mutation and recombination were implemented inside tree topologies and nothing else. So, our purpose in this chapter is to test the power of mutation and of recombination of topologies: instead of a bunch of optimization problems that must be solved separately, we have just one problem that explores alternatives and recombines the best from partial solutions. In that way we expect to test the power of evolution to the full.*

Our first step is to propose a frame to discuss our task.

1.1 Trees come in species

Although we have no universally accepted definition of species, most people consider that that concept must be tied to reproduction and hence to evolution. One also might add the specification of sexual reproduction. So, one immediately would say: the cat and the dog belong in different species because they can not produce hybridize. Now, sexual reproduction means recombination of genetic material. If we accept this directive, we must be ready to recognize that trees come in species. Let us see that in detail.

3 ***Viability of recombinants.***

Our genetic or verbal representation of trees is by means of Newick's and link notation:

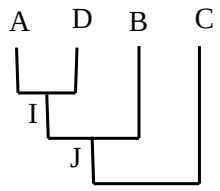


Figure 1.0. This tree can be represented as (((A,D),B),C) in Newick's notation, or as ((AI,DI)J,BJ),CJ) in link notation.

Let us consider now the recombinant products of the pair of trees with the same set of leaves ((A,D),B),C) and ((A,B), (C,D)). To that aim, we align them and take one part from the first tree and glue it to a second part from the other tree:

```
((A,D),B),C)
((A,B),(C,D))
```

Recombinants:

```
((A,B),(C,D)) : viable, clone of the second tree.
((A,D),(C,D)) : not viable for it has a repeated leaf.
((A,D),B),D)) : not viable for it has a repeated leaf.
((A,D),B),C)) : not a tree for the number of right
                parentheses do not match that of left ones.
```

This simple example shows us that blind and random recombination among tree topologies, i.e., recombination in the sense of Darwinian evolution, is in general a lost of time because of so much inviability. But on the other hand, let us remember that we had no trouble to implement recombination of lengths of links in within the same topology. Which is the lesson to be drawn? That each tree topology is a species on its own.

4 ***Challenge.*** *Develop a program to decide when a recombinant is viable, i.e., produces a string that can be interpreted as a binary tree and without lost or repetition of leaves. Use it to prove that recombination among different topologies is in general very rich in inviability.*

Let us pay attention to a possible way to implement recombination that always produce viable recombinants.

5 ***The Tot distance***

We saw in Vol X a recursive method to build all possible topologies. The output is a tree in which each node represents a topology tree. Let us refer to this super-tree as **Tot: the tree of trees**.

6 ***Definition.*** *The* ***Tot distance in within tree topologies S and T*** *is the the number of steps from S to T that one must walk along the shortest path in Tot.*

7 ***Example.*** *The Tot distance between*
(((A,F),((B,E),D)),C) and
((A,((B,E),D)),(C,F)) is 2 because one can transform the first topology to the second in two steps. First, take away leaf F from the first topology to get
((A,((B,E),D)),C) and second, reinsert it anew in the correct place.

8 ***Example.*** *The Tot distance between*
(((((A,F),E),D),B),C) and
((A,B),(((C,D),E),F)) is 6 because to go along the tree of trees from the first topology to the second, one needs to take away leaves D, E and F to reinsert them anew appropriately. The transforming sequence looks like this: begin with
(((((A,F),E),D),B),C), take away leaf F
((((A,E),D),B),C), take away leaf E
(((A,D),B),C), take away leaf D
((A,B),C), reinsert leaf D
((A,B),(C,D)), reinsert leaf E
((A,B),((C,D),E)), reinsert leaf F
((A,B),(((C,D),E),F)).

9 ***Exercise.*** *Run program J158 TreeOfTrees7 of Vol X to check whether or not the assertions of the previous example make sense over the tree of trees.*

Why we expect this distance to be important? We knew that each tree topology is a species and now we recognize that species grade by similarity. We expect that the probability of successful recombination diminishes as the Tot distance increases.

10 ***Challenge*** *The Tot distance is based on transformations. It is an algorithmic distance. Such distances make sense only when the distance refers to the shortest series of transformations that connects the two given objects. Does our Tot distance comply with this specification? We will assume a positive answer.*

Aforementioned discussion serves us to make explicit some points:

11 ***Darwinian evolution vs. Lamarckian evolution***

The basic idea of Darwinism such as we see it today in relation with mutation and recombination is that they happen at random. On the other hand, the basic idea of Lamarckism is that evolution can be helped by twisting randomness to make more probable those changes that lead to full adaptation of the evolving population to their environment or selective procedure. This contrast seems to imply that randomness in Darwinism must be defined as randomness according to the distribution given by the direct encoding of (nude) evolution.

To fix ideas, let us consider a Java simulation. The encoding of evolution allows here to interpret randomness as randomness according to a uniform distribution: it is readily implemented and produces no twisting, no preference for certain mutations. So, in regard with our Java simulation, **Darwinism** means that mutation

happens at randomness according to a uniform distribution. Every sort of twisting of randomness will be referred to as applied **Lamarckism**. But if we turn to the genome, we must take into account that mutation is driven by biophysics and biochemistry in rather complex operations and there is no need to suppose that, say, all possible base substitutions happen with equal probability. The experimentally observed distribution is what must be taken as the distribution that rules randomness in Darwinian evolution when the genome is referred to. We already know how to simulate in Java those cases.

And, what about recombination? In many cases it happens that the encoding of evolution naturally leads to recombination of homologous sequences, i.e., of sequences that encode for the same functional subunits. This is also the case of the genome. For those cases, Darwinism means that the site of recombination is chosen at random according to a uniform distribution. Any twisting is Lamarckian.

Let us discuss these ideas in regard with our problem of fitting a tree to a distance matrix. If mutation can change a char by another, then it is perfectly allowed that leaf C may be substituted by leaf D and so the mutant string has too leaves D and no leaf C: the string does not encode for a tree on the original sets of leaves. It is not viable and this is a generalized illness. By the same token, recombination is also plagued with generalized inviability. Thus:

1. It is useless to think that Darwinian mutation and recombination may rescue us from our troubles with found combinatorial explosion.

2. To implement Lamarckian evolution we must twist randomness to our favor. Our choice is to refer mutation to the tree of trees. So, the tree of trees is our natural frame for implementing evolution among tree topologies.

3. Mutation will be understood as a suitable walk along the tree of trees.

4. Recombination will be executed as the substitution of one sub-topology (a substring that is a tree topology on its own) by another only if both are subtopologies over the same subset of leaves.

1.2 Implementing mutation over tree topologies

We know how to recursively generate all possibles tree topologies using the tree of trees. Now, this tree engenders a combinatorial explosion. So, we cannot keep its representation in memory. Therefore, we can not use that tree to implement mutation of tree topologies. Rather we will implement mutation as a random walk over that tree but guided by rules not by the tree itself.

12 ***The two fundamental operations***

We need to know how to delete and how to reinsert a leaf in a given place to generate a walk over Tot but without actually using of seeing it. The rules to insert a leaf into a tree topology were worked out in Vol X. They resulted to be simple in our link notation. We might expect that the rules to delete a leaf are inverse to those of inserting. So, let us look at rule 1 for inserting a leaf F into link AI:

AI $->$ (AN,FN)I

for leaf A, new internal node N and old internal node I. But if we need to insert a leaf into a link that glue a subtree S, we apply rule 2

(S)I $->$ ((S)N, FN)I

where (S) is a subtree, N and I= internal nodes, F new leaf, FN new grafted branch. Rule 2 generalizes rule 1.

Antirule 1, the inverse of rule 1, is:

(AN,FN)I $->$ AI

Antirule 2, the inverse of rule 2 reads:

((S)N,FN)I $->$ (S)I

To implement these rules, we need to learn how to recognize a subtree even if it consists in just one leaf. Moreover, we cannot delete an inner node because the tree would get split. Other problem refers to parsing. To fix ideas, let us consider the tree:

((Aa,Ba),(((Cb,Db)c,Ec)d,Fd)a)

Since the order of insertion is alphabetic, the last inserted leaf was F. Its insertion generated the link Fd, so d was the last generated inner node. This says that F was glued to the subtree ((Cb,Db)c,Ec). Therefore, the parent tree must be

((Aa,Ba),(((Cb,Db)c,Ec))a)

But if we look at the tree of trees, we see that the direct ancestor of

((Aa,Ba),(((Cb,Db)c,Ec)d,Fd)a) is

((Aa,Ba),((Cb,Db)c,Ec)a)

We have an extra pair of parentheses. If we are not careful in this regard, we may deceive the tree parser. With all this in mind, let us compose the corresponding code:

13 ***The code for deleting a leaf follows.*** *Only antiRule1 is implemented: (AN,FN)I* $->$ *AI*

```
//Program K13 DeleteLeaf
//This program generates a random tree and
//deletes the last inserted leaf.

//We apply two deleting rules:
//antiRule 1:
//(An,Fn)i  -> Ai
//antiRule 2:
//((S)n,Fn)i -> (S)i
//where A and F are leaves, n and i are  inner nodes,
//(S) is a complex structure.
//These rules are inverse to inserting rules.

//Only antiRule 1 is implemented.



import java.util.Random;


public class DeleteLeaf
{

 //Link's notation for trees
 private static   String   parent =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPRSTUVWXYZ";
```

```
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;

 private static  Random r = new Random();

 private static boolean print = false;


 //Detect internal nodes, different than leaves
 private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
  else return false;
 }

//Simplification from Link notation to Newick's notation:
 // (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTree)
{
 String tree = "";
//length of string newTree
int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
  }
 return tree;
}

//Rule 2 is used to generate a new tree:
```

```
//(S)I -> ((S)N,FN)I.
//S = subtree, I = internal node, F = new leaf,
//N = new internal node.
//Rule 2 grafts new branch FN to link (S)I
//i is the position of char I in string parentTree
private static String useRule2(String parentTree, int i,
        char F, char N)
{
 String newTree = "";
 char I = parentTree.charAt(i);
 if (print)
 System.out.println("i = " + i + " I = " + I);
 //Subtree S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = i;
 boolean go = true;
 //All right parentheses to the near left of I are captured
 while (go)
  {
   position = position -1;
   String a = "" + parentTree.charAt(position);

   if (print)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
   else go = false;
  }//End of while
 if (print)
 System.out.println("counter = " + counter);
 //The complete subtree is captured:
 //Parentheses must balanced
```

```
 while (counter > 0)
  {
   String a = "" + parentTree.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print)
   System.out.println("Subtree = " + S);
   position = position -1;
   }
 //Replacement rule 2:
 //(S)I -> ((S)N,FN)I.  N= nextIntNode;
 //F = nextLeaf
 //Target to be replaced
 String target  =   S  + I;
 String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
 //Rule 1 is executed
 newTree = parentTree.replace(target, replacement);
 //If I links to a complex structure
 return newTree;
}

//From parentTree, a tree in Newick's notation,
//a Child  tree is generated by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTree)
{

 nextLeafIndex++;
 char nextLeaf= leaves.charAt(nextLeafIndex);
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = internalNodes.charAt(nextIntNodeIndex);


 //link to be replaced
 String link = "";
 //String link is replaced by what:
 String replacement = "";
```

```
 //new tree generated by replacement
 String newTree = "";
 //length of parent tree
 int l = parentTree.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
  {
   //parentTree is scanned char by char
   char I = parentTree.charAt(i);
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
  }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
  {
    //parentTree is scanned char by char
char I = parentTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = parentTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(parentTree, i,
               nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
```

```
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTree = parentTree.replace(link, replacement);
 }
System.out.println(newTree + " -> "
                       + simplify(newTree));


 i = l;
   }
  counter++;
}
  }
 return newTree;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply the antirule1:
//(An,Fn)i  -> Ai
private static String  antiRule1(String tree,
int positionLeaf, int positionNode)
{
 int beginIndex = positionLeaf - 4;
 int endIndex = positionNode + 3;
 String AnFni = tree.substring(beginIndex, endIndex);
 System.out.println("Anfni = " + AnFni);
 String Ai = "" + tree.charAt(beginIndex+1)
                + tree.charAt(endIndex-1);
 System.out.println("Ai = " + Ai);
 tree = tree.replace(AnFni, Ai);
return tree;
}

//Leaf belongs in a complex structure and is deleted
//We apply the antirule2:
//((S)n,Fn)i -> (S)i
private static String  antiRule2(String tree,
int positionLeaf, int positionNode)
```

```
{
 return "";
}


//The last inserted  leaf is deleted from tree.

private static String deleteLeaf(String tree)
{
  //Leaf to be deleted
  String leaf = "" + leaves.charAt(nextLeafIndex);
  System.out.println("\nLeaf to be deleted = " + leaf);
  //Accompanying inner node
  String innerNode = ""
            + internalNodes.charAt(nextIntNodeIndex);
  System.out.println("Accompanying inner  node = "
                    + innerNode);
  //new tree = output
  String newTree = "";
  //length of  tree as string
  int l = tree.length();
  //Last positions are found
  int positionLeaf = -1;
  int positionNode = -1;
  //tree is scanned char by char
  for(int i = 0; i < l; i++ )
   {
 String c = "" + tree.charAt(i);
 if (c.equals(leaf)) positionLeaf = i;
 if (c.equals(innerNode)) positionNode = i;
  }
  System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
  System.out.println("Position of " + innerNode +
          " = " + positionNode);
  int firstPosition = tree.indexOf(innerNode);
  //Check for the char before firstPosition
  char a = tree.charAt(firstPosition - 1);
  //If I links to a complex structure
  if (a == ')')
```

```
  newTree = antiRule2(tree,positionLeaf,positionNode);
  else  //Apply antiRule1 for simple cases
  newTree = antiRule1(tree,positionLeaf,positionNode);
  return newTree;
}


public static void main(String[] args)
{
 //A tree is generated
 System.out.println("Parent tree = \n" + parent
 + " -> "    + simplify(parent));
 //Number of leaves
 int nLeaves = 2;
 for(int i = 0; i < nLeaves; i++ )
  parent = generateChild(parent);
 //If antiRule 1 is applicable to last inserted leaf,
 //it is deleted.
  String tree = deleteLeaf(parent);
  System.out.println("Output tree = \n" + tree);
}

}//End of main class K13 DeleteLeaf
```

14 ***Exercise.*** *Run the program and play with the code. Verify that when antiRule 1 is applicable, it deletes the last inserted leaf.*

15 ***Exercise.*** *Implement antiRule2: ((S)N,FN)I* $->$ *(S)I. Answer*

We know now how to insert and how to delete a leaf. Since this is all one needs to walk over the tree of trees, we are ready to implement mutation in the set of tree topologies. One constraint must be observed: we must end with a tree topology that must have the same set of leaves with which we began.

The mutation we speak about is not Darwinian: it is not at random over the strings that encode for trees. It is Lamarckian: it implements a change that is viable by construction. In that way, we twist randomness to our favor, i.e., there is no lost of time due to producing and trying out so huge amounts of changes that do not encode for a tree. So, let us see how Lamarckian mutation is implemented.

16 ***The code that implements mutation of tree topologies follows.***

```
//Program K16 TopoMutation.
//This program generates a random binary tree topology
//and produces a mutation, whose output must be
//a binary tree topology with the same set of leaves.

//A mutation results from deleting
//a given number of leaves
//to reinsert them anew in
//other places chosen at random.



import java.util.Random;


public class TopoMutation
{

 //Link's notation for topologies
 private static   String   parent =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;

 private static  Random r = new Random();

 private static boolean print1 = false;
 private static boolean print2 = false;

 //This is the original topology:
 private static  String original = "";
```

```
 //============Size of topology ================
 //Number of leaves to be inserted to parent
 private static  int nAdditionalLeaves = 10;
 //=========================================

 //Number of leaves to be deleted and reinserted
 //to create a mutation
 private static int nUndoRedo = 0;


 //Detect internal nodes, different than leaves
 private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
  else return false;
 }

//Simplification from Link notation to Newick's notation:
 // (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTopo)
{
 String topo = "";
//length of string newTopo
int l = newTopo.length();
 for(int i = 0; i < l; i++ )
  {
   //newTopo is scanned char by char
  char c = newTopo.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    topo = topo + c;
  }
 return topo;
}

//Rule 2 is used to generate a new topology:
//(S)I -> ((S)N,FN)I.
```

```
//S = subtopology, I = internal node, F = new leaf,
//N = new internal node.
//Rule 2 grafts new branch FN to link (S)I
//i is the position of char I in string parentTopo
private static String useRule2(String parentTopo, int i,
        char F, char N)
{
 String newTopo = "";
 char I = parentTopo.charAt(i);
 if (print1)
 System.out.println("i = " + i + " I = " + I);
 //Subtopology S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = i;
 boolean go = true;
 //All right parentheses to the near left of I are captured
 while (go)
  {
   position = position -1;
   String a = "" + parentTopo.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
   else go = false;
  }//End of while
 if (print1)
 System.out.println("counter = " + counter);
 //The complete subtopology is captured:
 //Parentheses must balanced
 while (counter > 0)
```

```
  {
   String a = "" + parentTopo.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
   System.out.println("Subtopo = " + S);
   position = position -1;
   }
 //Replacement rule 2:
 //((S)n,Fn)i -> (S)i
//where   F are leaves, n and i are  inner nodes,
//(S) is a complex structure.
 String SnFni  =   S  + I;
 String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
 //Rule 1 is executed
 newTopo = parentTopo.replace(SnFni, replacement);
 //If I links to a complex structure
 return newTopo;
}

//From parentTopo, a topology in Newick's notation,
//a Child  topology is generated by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTopo)
{

 nextLeafIndex++;
 char nextLeaf= leaves.charAt(nextLeafIndex);
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = internalNodes.charAt(nextIntNodeIndex);


 //link to be replaced
 String link = "";
 //String link is replaced by what:
 String replacement = "";
 //new topology generated by replacement
```

```
 String newTopo = "";
 //length of parent topology
 int l = parentTopo.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
  {
   //parentTopo is scanned char by char
   char I = parentTopo.charAt(i);
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
  }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
  {
    //parentTopo is scanned char by char
char I = parentTopo.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = parentTopo.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTopo = useRule2(parentTopo, i,
               nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
```

```
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTopo = parentTopo.replace(link, replacement);
 }
if (print2)
System.out.println(newTopo + " -> "
                       + simplify(newTopo));


 i = l;
   }
  counter++;
}
  }
 return newTopo;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply the antirule1:
//(An,Fn)i  -> Ai
private static String  antiRule1(String topo,
int positionLeaf, int positionNode)
{
 if (print2)
  System.out.println("AntiRule1");
 int beginIndex = positionLeaf - 4;
 int endIndex = positionNode + 3;
 String AnFni = topo.substring(beginIndex, endIndex);
 if (print2)
  System.out.println("Anfni = " + AnFni);
 String Ai = "" + topo.charAt(beginIndex+1)
                + topo.charAt(endIndex-1);
 if (print2)
  System.out.println("Ai = " + Ai);
 topo = topo.replace(AnFni, Ai);
return topo;
}
```

```
//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply the antirule2:
//((S)n,Fn)i -> (S)i
//positionNode refers to the last appearing of node n
private static String  antiRule2(String topo,
int positionLeaf, int positionNode)
{
 if (print2)
System.out.println("AntiRule2");
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 if (print2)
  System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
           + internalNodes.charAt(nextIntNodeIndex);
 int firstPosition = topo.indexOf(innerNode);
 String newTopo = "";
 char i = topo.charAt(positionNode+2);
 if (print2)
 System.out.println("FirstPosition = " +
                     firstPosition + " i = " + i);
 //Subtopology S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = firstPosition;
 boolean go = true;
 //All right parentheses to the near left of I are captured
 while (go)
  {
   position = position -1;
    String as = "" + topo.charAt(position);

   if (print2)
```

```
   System.out.println("a = " + as + " p = " + pr);
   if (as.equals(pr) )
   {
counter++;
S = as + S;
   }
   else go = false;
  }//End of while
 if (print2)
 System.out.println("counter = " + counter);
 //The complete subtopology is captured:
 //Parentheses must balanced
 while (counter > 0)
  {
   String as = "" + topo.charAt(position);
   if (as.equals(pr))   counter++;
   if (as.equals(pl))  counter--;
   S = as + S;
   if (print2)
   System.out.println("Subtopo = " + S);
   position = position -1;
   }
  String n = innerNode;
 //Replacement antiRule 2:
 //((S)n,Fn)i -> (S)i
 //where A and F are leaves, n and i are  inner nodes,
 //(S) is a complex structure.
 String SnFni  =  "("+ S  + n + "," + leaf + n + ")" + i ;
 String Si = S + i;
 if (print2)
  System.out.println("\nSnFni = " + SnFni + "\n Si = " + Si);
 //Rule 1 is executed
 newTopo = topo.replace(SnFni, Si);
 return newTopo;
}



//The last inserted  leaf is deleted from topology.
private static String deleteLeaf(String topology)
```

```
{
  //Leaf to be deleted
  String leaf = "" + leaves.charAt(nextLeafIndex);
  if (print2)
  System.out.println("\nLeaf to be deleted = " + leaf);
  //Accompanying inner node
  String innerNode = ""
            + internalNodes.charAt(nextIntNodeIndex);
  if (print2)
  System.out.println("Accompanying inner  node = "
                    + innerNode);
  //new topology = output
  String newtopology = "";
  //length of  topology as string
  int l = topology.length();
  //Last positions are found
  int positionLeaf = -1;
  int positionNode = -1;
  //topology is scanned char by char
  for(int i = 0; i < l; i++ )
   {
 String c = "" + topology.charAt(i);
 if (c.equals(leaf)) positionLeaf = i;
 if (c.equals(innerNode)) positionNode = i;
   }
  if (print2)
   {
    System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
    System.out.println("Position of " + innerNode +
          " = " + positionNode);
   }
  int firstPosition = topology.indexOf(innerNode);
  //Check for the char before firstPosition
  char a = topology.charAt(firstPosition - 1);
  //If I links to a complex structure
  if (a == ')')
  newtopology = antiRule2(topology,positionLeaf,positionNode);
  else  //Apply antiRule1 for simple cases
  newtopology = antiRule1(topology,positionLeaf,positionNode);
```

```
  nextLeafIndex--;
  nextIntNodeIndex--;
  return newtopology;
}


/**
 * @param args
 */
public static void main(String[] args)
{
 //A topology is generated
 if (print2)
  System.out.println("Parent topology = \n" + parent
 + " -> "    + simplify(parent));
 String topology = parent;

 for(int i = 0; i < nAdditionalLeaves; i++ )
   topology = generateChild(topology);
 //This original topology is next subject to mutation
 original = topology;
 if (print2)
  System.out.println("\nOriginal   topology = \n"
                + original  );
 //Number of leaves to be deleted and then reinserted
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
  else
   nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;
 //nUndoRedo = 13;

 //Mutation has two stages:
 //deletion of some leaves that undoes generating work +
 //re-insertion in places chosen at random.

 if (print2)
  System.out.println("Number of leaves to" +
   " be deleted and then reinserted = " +
   nUndoRedo);
 //Some leaves are deleted:
 if (print2)
```

```
 System.out.println("Some leaves are deleted");
 for(int i = 0; i < nUndoRedo; i++ )
 {
  //Last inserted leaf is deleted.
  topology = deleteLeaf(topology);
  if (print2)
   System.out.println(topology);
 }
//Some leaves are reinserted
 if (print2)
  System.out.println("Deleted leaves are reinserted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
   topology = generateChild(topology);
  }
 //Original and mutant topologies are compared.
 String mutant = topology;
 System.out.println("\nOriginal and mutant topologies = \n"
  + original + "\n" + mutant);
 if (original.equals(mutant))
  System.out.println(
  "Original and mutant topologies coincide");
 else
  System.out.println(
  "\nOriginal and mutant topologies are different");
}

}//End of main class K16 TopoMutation
```

17 ***Exercise.*** *Run the program and play with the code.*

18 ***Exercise.*** *Previous program implements mutation over topologies. Add lengths of links to get trees and implement mutation over trees to include the possibility of changing topology and/or lengths of links.* *Answer*

We can now undertake the task of enabling recombination, an operation that will enable evolution to the full.

1.3 Recombination of tree topologies.

Recombination is defined over pair of strings: it takes two strings and outputs at least one recombinant product that contains a part from the first string and another part from the second.

19 ***Darwinian recombination.*** *Darwinian recombination is distinguished because everything happens at random with a uniform distribution. Say, all strings have equal probabilities to be chosen for recombination and the place of recombination is chosen at random with a uniform distribution over all possible places.*

Darwinian recombination of trees is a failure:

20 ***Exercise.*** *Give examples that show that recombination of strings that represent trees produces in general not viable products.*

21 ***Lamarckian recombination.*** *When Darwinian evolution leads to nothing, Lamarckian evolution is the salvation table. Our Lamarckian recombination must take two binary tree topologies with the same set of leaves and produce a recombinant that must be a well formed binary tree topology on the same set of leaves. To implement this type of recombination, one must look for all subtopologies in both input tree topologies and if there is a pair of subtopologies on exactly the same set of leaves, one in the first topology and the other in the second, then one can interchange them.*

22 ***Challenge.*** *Imagine a better way of implementing Lamarckian recombination else prove that this is the best.*

To go ahead in our purpose of implementing recombination, we must be able of listing all possible subtopologies of a given topology. Let us recall that a topology is just a tree whose links lack associated lengths. So, we are looking after all subtrees of a tree without worrying about lengths of links.

23 ***Example.*** *Let us consider the tree topology*
((((Ad,Fd)a,(Bc,Ec)a),(Cb,Db)a)
The complete set of subtopologies is conformed by:
(Ad,Fd)
(Bc,Ec)
(Cb,Db)
((Ad,Fd)a,(Bc,Ec)a)

24 ***Exercise.*** *List all subtopologies of*
((Aa,Ba),((Cb,Db)c,((Ee,Ge)d,Fd)c)a)
Answer

25 ***Exercise.*** *We have chosen our subtopologies to be unrooted. Decide whether or not this is correct and whether or not this question is irrelevant. Answer*

26 ***The next code lists all subtopologies of a topology on a set of n leaves.*** *We restrict to subtopologies with a number of leaves in within 2 and n-1. Subtopologies are substrings with a balanced set of parentheses, as many right parentheses as left ones.*

```
//Program K26 SubTopol
/*
 * This program list all possible subtopologies
 * of a given topology.
 *
 * We begin with a left parenthesis "(" and cumulate
 * symbols until the number of
 * right and left parentheses get balanced.
 *
 * All left parenthesis are investigated.
 *
 */

import java.util.Random;

public class SubTopol
{

//Link's notation for trees
 private static   String   parent =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
```

```
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;

 //Leaves that remain in the tree
 private static  String presentLeaves =   "ABC";
 private static String presentNodes =  "a";


 private static  Random r = new Random();

 private static boolean print1 = false;
 private static boolean print2 = true;



//====== A random topology is generated =============

//Detect internal nodes, different than leaves
private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
   else return false;
 }


//Simplification from Link notation to Newick's notation:
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTree)
 {
  String tree = "";
  //length of string newTree
  int l = newTree.length();
  for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
```

```
    tree = tree + c;
  }
 return tree;
 }

 //Rule 2 is used to generate a new tree:
 //(S)I -> ((S)N,FN)I.
 //S = subtree, I = internal node, F = new leaf,
 //N = new internal node.
 //Rule 2 grafts new branch FN to link (S)I
 //i is the position of char I in string parentTree
 private static String useRule2(String parentTree, int i,
        char F, char N)
 {
  String newTree = "";
  char I = parentTree.charAt(i);
  if (print1)
  System.out.println("i = " + i + " I = " + I);
  //Subtree S of link (S)I must be read
  String S = "";
  //Counter of left parenthesis
  int counter = 0;
  String pl = "" + '(';
  String pr = "" + ')';
  //Position to be scanned
  int position = i;
  boolean go = true;
  //All right parentheses to the near left of I are captured
  while (go)
  {
   position = position -1;
   String a = "" + parentTree.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
```

```
   else go = false;
  }//End of while
  if (print1)
 System.out.println("counter = " + counter);
  //The complete subtree is captured:
  //Parentheses must balanced
  while (counter > 0)
  {
   String a = "" + parentTree.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
   System.out.println("Subtree = " + S);
   position = position -1;
   }
  //Replacement rule 2:
  //((S)n,Fn)i -> (S)i
  //where   F are leaves, n and i are  inner nodes,
  //(S) is a complex structure.
  String SnFni  =   S  + I;
  String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
  //Rule 1 is executed
  newTree = parentTree.replace(SnFni, replacement);
  //If I links to a complex structure
  return newTree;
 }


//Input: parentTree, a tree in Newick's notation.
//Output : newTree =  parent tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTree)
 {
  nextLeafIndex++;
  char nextLeaf = leaves.charAt(nextLeafIndex);
  presentLeaves = presentLeaves + nextLeaf;
  //Next internal node to be inserted
```

```
  nextIntNodeIndex++;
  char nextIntNode = internalNodes.charAt(nextIntNodeIndex);
  presentNodes = presentNodes + nextIntNode;

  //link to be replaced
  String link = "";
  //String link is replaced by what:
  String replacement = "";
  //new tree generated by replacement
  String newTree = "";
  //length of parent tree
  int l = parentTree.length();
  //counter of links = 0;
  int counter = 0;
  //Links are counted
  for(int i = 0; i < l; i++ )
  {
   //parentTree is scanned char by char
   char I = parentTree.charAt(i);
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
  }
  //A link is chosen
  int k = r.nextInt(counter);
  //System.out.println("Random number = " + k);
  //new branch is grafted at k
  counter = 0;
  for(int i = 0; i < l; i++ )
   {
//parentTree is scanned char by char
char I = parentTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
  {
//Check for the char before I
char a = parentTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(parentTree, i,
```

```
              nextLeaf, nextIntNode);
 else //if I links to a leaf, use rule 1
  {
   //Link to be replaced
   link = "" + a + I;
   //Replacement rule 1:
   //aI -> (aN,FN)I;   N= nextIntNode;
   //F = nextLeaf
  replacement = "(" + a + nextIntNode + ","
       + nextLeaf + nextIntNode + ")" + I;
   //Rule 1 is executed
   newTree = parentTree.replace(link, replacement);
  }
 System.out.println(newTree + " -> "
                      + simplify(newTree));

 i = l;
}
   counter++;
  }
 }
    return newTree;
   }


//========== Subtopologies are listed ============


//Subtopologies of  topology are found
public static void subtopologies(String topology)
{
 System.out.println("Subtopologies are found");
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Length of String   topology
 int m = topology.length();
 //String topology is scanned char by char
 for(int i = 0; i < m; i++ )
```

```
  {
   String Sub = "";
   String a = "" + topology.charAt(i);
   int position = i+1;
   if (a.equals(pl))
   {
counter++;
 Sub =  Sub + a;
//The complete subtopology is captured:
//Parentheses must balanced
while (counter > 0)
 {
  String b = "" + topology.charAt(position);
  if (b.equals(pl))   counter++;
  if (b.equals(pr))  counter--;
  Sub =  Sub + b;

  position++;
 }
 if (print2)
   System.out.println(Sub);
   }
 }
}


 public static void main(String[] args)
 {
   System.out.println("A random topology is generated");
   //Seed
   System.out.println("\n" + parent
 + " -> "    + simplify(parent));
   //A tree topology is generated
   String topology = parent;
   //Number of leaves to be inserted
   int nLeavesToBeInserted = 6;
   for(int i = 0; i < nLeavesToBeInserted; i++ )
   topology = generateChild(topology);
   //Test
   //topology = " ((Aa,Ba),((Cb,Db)c,((Ee,Ge)d,Fd)c)a)";
```

```
   //Subtopologies are found
   subtopologies(topology);
 }
}//End of main class K26  SubTopol
```

27 ***Exercise.*** *Run the program and play with the code.*

28 ***Exercise.*** *The previous program just prints a list of all subtopologies. Pack the list in some structure to be kept in memory, say an array or something. In that way, that list will be reusable to implement recombination.* *Answer*

29 ***Recombination of topologies***

Our task is to compose the code for the recombination of two trees. Our plan is to find the two topologies associated to the two trees and to calculate all their subtopologies. If there are two subtopologies that have the same set of leaves, we authorize recombination but not of topologies but of trees. Example:

From the topology ((((Ac,Ec)b,Db)a,Ba),Ca) we select the subtopology ((Ac,Ec)b,Db). And from ((((Ab,Db)c,Ec)a,Ba),Ca), we select ((Ab,Db)c,Ec). The two subtopologies have the same set of leaves: A,D,E. We notice that, happily, the sets of inner nodes of subtopologies is also equal: c, d. If we swap subtopologies we end with the following strings that indeed correspond with two viable, correctly described topologies:

((((Ac,Ec)b,Db)a,Ba),Ca) $->$ ((((Ab,Db)c,Ec)a,Ba),Ca)
((((Ab,Db)c,Ec)a,Ba),Ca) $->$ ((((Ac,Ec)b,Db)a,Ba),Ca)

Now, we have a preoccupation in regard with subtopologies: Is the equality of the sets of leaves always transported to the set of inner nodes? Actually, is this question important? Does the inequality of the sets of inner nodes implies inviability of recombinants? To enlighten these questions let us add to the previous program the capability to show the set of leaves and the set of inner nodes of all subtopologies.

```
//Program K29 SubTopol3
/*
 * This program generates two topologies  on the same
 * set of leaves.
 * All their subtopologies are calculated
 * and packed in arrays.
```

```
 * Sets of leaves and inner nodes of each subtopology
 * are found.
 *
 */

import java.util.Random;

public class SubTopol3
{

//Link's notation for trees
 private static   String   parent =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;



 //Leaves that remain in the tree
 private static  String presentLeaves =   "ABC";
 private static String presentNodes =  "a";
 //number of chars of master topology
 private static  int m1 = 0;
 private static  int m2 = 0;

 private static  Random r = new Random();

 private static boolean print1 = false;
 private static boolean print2 = false;



//====== A random topology is generated =============
```

```
//Detect internal nodes, different than leaves
private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
   else return false;
 }


//Simplification from Link notation to Newick's notation:
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTree)
 {
  String tree = "";
  //length of string newTree
  int l = newTree.length();
  for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
  }
 return tree;
 }

 //Rule 2 is used to generate a new tree:
 //(S)I -> ((S)N,FN)I.
 //S = subtree, I = internal node, F = new leaf,
 //N = new internal node.
 //Rule 2 grafts new branch FN to link (S)I
 //i is the position of char I in string parentTree
 private static String useRule2(String parentTree,
 int i,   char F, char N)
 {
  String newTree = "";
  char I = parentTree.charAt(i);
  if (print1)
```

```
  System.out.println("i = " + i + " I = " + I);
  //Subtree S of link (S)I must be read
  String S = "";
  //Counter of left parenthesis
  int counter = 0;
  String pl = "" + '(';
  String pr = "" + ')';
  //Position to be scanned
  int position = i;
  boolean go = true;
  //All right parentheses to the near left
  //of I are captured
  while (go)
  {
   position = position -1;
   String a = "" + parentTree.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
   else go = false;
  }//End of while
  if (print1)
 System.out.println("counter = " + counter);
  //The complete subtree is captured:
  //Parentheses must balanced
  while (counter > 0)
  {
   String a = "" + parentTree.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
   System.out.println("Subtree = " + S);
   position = position -1;
   }
```

```
  //Replacement rule 2:
  //((S)n,Fn)i -> (S)i
  //where   F are leaves, n and i are  inner nodes,
  //(S) is a complex structure.
  String SnFni  =   S  + I;
  String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
  //Rule 1 is executed
  newTree = parentTree.replace(SnFni, replacement);
  //If I links to a complex structure
  return newTree;
 }


//Input: parentTree, a tree in Newick's notation.
//Output : newTree =  parent tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTree)
 {
  nextLeafIndex++;
  char nextLeaf = leaves.charAt(nextLeafIndex);
  presentLeaves = presentLeaves + nextLeaf;
  //Next internal node to be inserted
  nextIntNodeIndex++;
  char nextIntNode =
  internalNodes.charAt(nextIntNodeIndex);
  presentNodes = presentNodes + nextIntNode;

  //link to be replaced
  String link = "";
  //String link is replaced by what:
  String replacement = "";
  //new tree generated by replacement
  String newTree = "";
  //length of parent tree
  int l = parentTree.length();
  //counter of links = 0;
  int counter = 0;
  //Links are counted
```

```
  for(int i = 0; i < l; i++ )
  {
   //parentTree is scanned char by char
   char I = parentTree.charAt(i);
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
  }
  //A link is chosen
  int k = r.nextInt(counter);
  //System.out.println("Random number = " + k);
  //new branch is grafted at k
  counter = 0;
  for(int i = 0; i < l; i++ )
   {
//parentTree is scanned char by char
char I = parentTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
  {
//Check for the char before I
char a = parentTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(parentTree, i,
         nextLeaf, nextIntNode);
 else //if I links to a leaf, use rule 1
  {
   //Link to be replaced
   link = "" + a + I;
   //Replacement rule 1:
   //aI -> (aN,FN)I;   N= nextIntNode;
   //F = nextLeaf
  replacement = "(" + a + nextIntNode + ","
       + nextLeaf + nextIntNode + ")" + I;
   //Rule 1 is executed
   newTree = parentTree.replace(link, replacement);
  }
if (print1)
 System.out.println(newTree + " -> "
```

```
                    + simplify(newTree));

 i = l;
}
   counter++;
  }
 }
    return newTree;
   }


//========== Subtopologies are listed ============


//Subtopologies of  topology are found
public static String[] subtopologies(String topology)
{
 System.out.println("\nSubtopologies are found");
 //Counter of left parenthesis
 int counter = 0;
 //Counter of subtopologies
 int counterSubtop = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Length of String   topology
 int m = topology.length();
 String[] subtopologies = new String[m];
 //String topology is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   String Sub = "";
   String a = "" + topology.charAt(i);
   int position = i+1;
   if (a.equals(pl))
   {
counter++;
 Sub =  Sub + a;
//The complete subtopology is captured:
//Parentheses must balanced
while (counter > 0)
```

```
 {
  String b = "" + topology.charAt(position);
  if (b.equals(pl))   counter++;
  if (b.equals(pr))  counter--;
  Sub =  Sub + b;
  position++;
 }
 if (print2)
   System.out.println(Sub);
   subtopologies[counterSubtop] = Sub;
   counterSubtop++;
  }
 }
 return subtopologies;
}



 public static String generateTopology()
 {
 System.out.println("A random topology is generated");
   //Seed
   if (print1)
   System.out.println("\n" + parent
 + " -> "    + simplify(parent));
 //Next leaf to be grafted
 nextLeafIndex = 2;
 //Next internal node to be inserted
 nextIntNodeIndex = 0;
   //A tree topology is generated
   String topology = parent;
   //Number of leaves to be inserted
   int nLeavesToBeInserted = 5;
   for(int i = 0; i < nLeavesToBeInserted; i++ )
   topology = generateChild(topology);
   return topology;
 }
```

```
//Leaves of subtopologies  are found
 public static String[]leaves(int m, String[] subtop)
{
 if (print1)
  System.out.println("Leaves are found");
 String[] myLeaves = new String[m];
 for(int i = 0; i < m; i++)
  {
String s = subtop[i];
if (s == null);
  else
  {
   myLeaves[i] = "";
   int n = s.length();
   for(int j = 0; j < n; j++)
    {
     String c = "" + subtop[i].charAt(j);
     if (leaves.contains(c))
      myLeaves[i] = myLeaves[i] + c;
    }
   if (print1)
   System.out.println(subtop[i] + " "
                           + myLeaves[i] );
  }
   }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
 System.out.println("Leaves are listed orderly" );
 for(int i = 0; i < m; i++)
 {
String s = subtop[i];
if (s == null);
  else
  {
    s =   myLeaves[i];
    myLeaves[i] = "";
   int n = leaves.length();
   for(int j = 0; j < n; j++)
    {
```

```
     String c = "" + leaves.charAt(j);
     if (s.contains(c))
      myLeaves[i] = myLeaves[i] + c;
    }
   if (print1)
   System.out.println(subtop[i] + " " + myLeaves[i] );
  }
  }
 return myLeaves;
  }



 //Inner nodes of subtopologies  are found
  public static String[] inner(int m, String[] subtop)
 {
  if (print1)
  System.out.println("\nInternal nodes are found");
  String[] myNodes = new String[m];
  for(int i = 0; i < m; i++)
   {
  String s = subtop[i];
  if (s == null);
    else
    {
   myNodes[i] = "";
     int n = s.length();
     for(int j = 0; j < n; j++)
      {
       String c = "" + subtop[i].charAt(j);
       if (internalNodes.contains(c))
       myNodes[i] = myNodes[i] + c;
      }
    if (print1)
 System.out.println(subtop[i] + " " + myNodes[i] );
    }
    }
  //Chars of Leaves are rewritten in alphabet order
  if (print1)
  System.out.println("Inner nodes are listed orderly" );
```

```
  for(int i = 0; i < m; i++)
  {
  String s = subtop[i];
  if (s == null);
    else
    {
      s =   myNodes[i];
     myNodes[i] = "";
     int n = internalNodes.length();
     for(int j = 0; j < n; j++)
      {
       String c = "" + internalNodes.charAt(j);
       if (s.contains(c))
       myNodes[i] = myNodes[i] + c;
      }
    if (print1)
     System.out.println(subtop[i] + " " + myNodes[i] );
    }
   }
  return myNodes;
   }



public static void recombineTrees()
{
 System.out.println("Two trees are recombined");

}


public static void main(String[] args)
{
  //A topology is generated
  String topo1 = generateTopology();
  System.out.println(topo1 + " -> "
         + simplify(topo1));
  //Length of String   topology
  m1 = topo1.length();
  //Subtopologies are found
```

```
   String[] subtopologies1 = new String[m1];
   subtopologies1 = subtopologies(topo1);

   /*
   String[] Nodes1 = new String[m1];
   innerNodes(subtopologies1); */

   System.out.println();
   String[] Leaves1 = new String[m1];
   Leaves1 = leaves(m1,subtopologies1);
   String[] Inner1 = new String[m1];
   Inner1 = inner(m1,subtopologies1);
   //Chars of Leaves are rewritten in alphabet order
   System.out.println("\nSubtopologies, Leaves, " +
                "Inner nodes" );
   for(int i = 0; i < m1; i++)
   {
 if (subtopologies1[i] == null);
  else
  {

   System.out.println(i + "   " + subtopologies1[i]
   + "   " +  Leaves1[i] + "   " + Inner1[i]);
  }
   }

   System.out.println();

   //A second topology is generated

   String topo2 = generateTopology();
   System.out.println(topo2 + " -> "
          + simplify(topo2));
   //Length of String   topology
   m2 = topo2.length();
   //Subtopologies are found
   String[] subtopologies2 = new String[m2];
   subtopologies2 = subtopologies(topo2);

   /*
```

```
   String[] Nodes1 = new String[m1];
   innerNodes(subtopologies1); */

   System.out.println();
   String[] Leaves2 = new String[m2];
   Leaves2 = leaves(m2,subtopologies2);
   String[] Inner2 = new String[m2];
   Inner2 = inner(m2,subtopologies2);
   //Chars of Leaves are rewritten in alphabet order
   System.out.println("\nSubtopologies, Leaves, " +
                 "Inner nodes" );
   for(int i = 0; i < m2; i++)
   {
 if (subtopologies1[i] == null);
  else
  {

   System.out.println(i + "   " + subtopologies2[i]
   + "   " +  Leaves2[i] + "   " + Inner2[i]);
  }
   }
 }
}//End of main class K29  SubTopol3
```

30 ***Exercise.*** *Run the program and play with the code.*

31 ***Exercise.*** *Add to the previous program the capability to automatically decide when there are subtopologies whose set of leaves coincide. The same decision must be done for inner nodes. Use the program to decide whether or not the equality of the sets of leaves is transported over the set of inner nodes. Make an estimation of the frequency with which possible recombinant subtopologies appear.* *Answer*

32 ***Recombination of trees.*** *The following code generates two topologies on the same set of leaves and adds lengths to links to get two trees. Next, all subtopologies are found and they are compared by pairs, one from the first topology and the other from the second. If a pair of subtopologies happen to have the same set of leaves and of inner nodes, recombination is authorized: the corresponding subtrees are swapped. The original topologies always have the same set of leaves and of inner nodes but they do not count. If there are more than one pair of subtopologies that allow recombination, the first found pair is selected.*

```
//Program K32 SubTopol5
/*
 * Code for the
 * recombination of tree topologies:
 *
 * This program generates two topologies  on the same
 * set of leaves and adds  lengths to links
 * to get two trees.
 * All their subtopologies are calculated
 * and packed in arrays.
 * The sets of leaves and of inner nodes
 * of each subtopology are found.
 * Subtopologies are compared by pairs,
 * one from each topology.
 * If they happen to have the same set of leaves
 * and of inner nodes, recombination is authorized:
 * the  subtrees that correspond with
 * the subtopologies are swapped.
 *
 * If more than one pair of subtopologies authorize
 * recombination, the last found pair is chosen.
 * Original topologies serves as subtopologies
 * that authorize recombination but they do not count.
 *
 */

import java.util.Random;

public class SubTopol5
{

//Link's notation for trees
private static   String   parent =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
```

```
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;

 //===========Size of the tree===========
 //Number of leaves to be inserted
 private static  int nLeavesToBeInserted = 3;
 //=====================================
//Number of leaves that must go in all trees > 3
 private static int numberOfLeaves =
             3 + nLeavesToBeInserted;

   //Length of the link that glues a leaf to the tree
  private static int lengthsLeaves[]  =
               new int[numberOfLeaves];
  //Length of the link that glues
  //an inner node to another one
  private static int lengthsInnerLinks[]  =
              new int[numberOfLeaves];


 //Counter of subtopologies
 private static  int counterSubtop;
 //Particular number of subtopologies
 private static  int m1 = 0;


 //Leaves that remain in the tree
 private static  String presentLeaves =   "ABC";
 private static String presentNodes =  "a";


 private static  Random r = new Random();
 //The control parameter of the random generator for int
 private static int maxLength = 10;

 private static  String[] subtrees1, subtrees2;
 private static  String tree1, tree2,
                        subTree1, subTree2,
                        treeRec1, treeRec2;
```

```
 private static boolean print1 = false;
 private static boolean print2 = false;
 private static boolean print3 = true;



//====== A random topology is generated =============

//Detect internal nodes, different than leaves
private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
   else return false;
 }


//Simplification from Link notation to Newick's notation:
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String linkToNewick(String newTree)
 {
  String tree = "";
  //length of string newTree
  int l = newTree.length();
  for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
  }
 return tree;
 }


//Complete simplification of subtrees
//(L:5.0m:3.0,O:6.0m:3.0) ->  (L,O)
```

```
//Output = subtopology
private static String simplify(String newTree)
{
String topology = "";
//The alphabet of topologies
String allowedChars = leaves + "(),";
//length of string newTree
int l = newTree.length();
for(int i = 0; i < l; i++ )
 {
  //newTree is scanned char by char
  String c = "" + newTree.charAt(i);
  //Is c is an allowed char, add it  to output
  if (allowedChars.contains(c))
   topology = topology + c;
 }
//System.out.println( newTree + "  simplify: " + topology);
return topology;
}
```

```
 //Rule 2 is used to generate a new tree:
 //(S)I -> ((S)N,FN)I.
 //S = subtree, I = internal node, F = new leaf,
 //N = new internal node.
 //Rule 2 grafts new branch FN to link (S)I
 //i is the position of char I in string parentTree
 private static String useRule2(String parentTree,
 int i,   char F, char N)
 {
  String newTree = "";
  char I = parentTree.charAt(i);
  if (print1)
  System.out.println("i = " + i + " I = " + I);
  //Subtree S of link (S)I must be read
  String S = "";
  //Counter of left parenthesis
  int counter = 0;
```

```
  String pl = "" + '(';
  String pr = "" + ')';
  //Position to be scanned
  int position = i;
  boolean go = true;
  //All right parentheses to the near left
  //of I are captured
  while (go)
  {
   position = position -1;
   String a = "" + parentTree.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
   else go = false;
  }//End of while
  if (print1)
 System.out.println("counter = " + counter);
  //The complete subtree is captured:
  //Parentheses must balanced
  while (counter > 0)
  {
   String a = "" + parentTree.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
   System.out.println("Subtree = " + S);
   position = position -1;
   }
  //Replacement rule 2:
  //((S)n,Fn)i -> (S)i
  //where   F are leaves, n and i are  inner nodes,
  //(S) is a complex structure.
  String SnFni  =   S  + I;
```

```
  String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
  //Rule 1 is executed
  newTree = parentTree.replace(SnFni, replacement);
  //If I links to a complex structure
  return newTree;
 }


//Input: parentTree, a tree in Newick's notation.
//Output : newTree =  parent tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTree)
 {
  nextLeafIndex++;
  char nextLeaf = leaves.charAt(nextLeafIndex);
  presentLeaves = presentLeaves + nextLeaf;
  //Next internal node to be inserted
  nextIntNodeIndex++;
  char nextIntNode =
  internalNodes.charAt(nextIntNodeIndex);
  presentNodes = presentNodes + nextIntNode;

  //link to be replaced
  String link = "";
  //String link is replaced by what:
  String replacement = "";
  //new tree generated by replacement
  String newTree = "";
  //length of parent tree
  int l = parentTree.length();
  //counter of links = 0;
  int counter = 0;
  //Links are counted
  for(int i = 0; i < l; i++ )
  {
   //parentTree is scanned char by char
   char I = parentTree.charAt(i);
//Is I an internal node that denotes a link?
```

```
if (charIsInternalNode(I)) counter++;
  }
  //A link is chosen
  int k = r.nextInt(counter);
  //System.out.println("Random number = " + k);
  //new branch is grafted at k
  counter = 0;
  for(int i = 0; i < l; i++ )
   {
//parentTree is scanned char by char
char I = parentTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
  {
//Check for the char before I
char a = parentTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(parentTree, i,
         nextLeaf, nextIntNode);
 else //if I links to a leaf, use rule 1
  {
   //Link to be replaced
   link = "" + a + I;
   //Replacement rule 1:
   //aI -> (aN,FN)I;   N= nextIntNode;
   //F = nextLeaf
  replacement = "(" + a + nextIntNode + ","
       + nextLeaf + nextIntNode + ")" + I;
   //Rule 1 is executed
   newTree = parentTree.replace(link, replacement);
  }
if (print1)
 System.out.println(newTree + " -> "
                    + linkToNewick(newTree));

 i = l;
}
   counter++;
```

```
  }
 }
    return newTree;
   }

//===========Trees are synthesized==============


//Random lengths are added to a tree in Link notation
private static void makeLengths()
{
//Lengths with digit values are generated
for(int i = 0; i < numberOfLeaves; i++)
 {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
 }
}

//Random lengths are added to a tree in Link notation
private static String insertLengths(String treeLinkN)
{
String thisChar = "";
String newTree = "";
//length of String   tree
int m = treeLinkN.length();
//Tree is scanned char by char
for(int i = 0; i < m; i++ )
 {
  thisChar = "" + treeLinkN.charAt(i);
  newTree = newTree + thisChar;
  if (print1)
  System.out.println(newTree.length() + " " + newTree);
  //Lengths are appended
  if (leaves.contains(thisChar))
   {
    int k = leaves.indexOf(thisChar);
    double j = lengthsLeaves[k];
   newTree = newTree + ":" + j;
```

```
   }
  if (internalNodes.contains(thisChar))
  {
   int k = internalNodes.indexOf(thisChar);
   double j = lengthsInnerLinks[k];
   newTree = newTree + ":" + j;
  }
 }
 return newTree;
}


//========== Subtrees are listed ============


//Subtrees of  topology are found
private static String[] subtopologies(String topology)
{
 if (print1)
System.out.println("\nSubtrees are found");
 //Counter of left parenthesis
 int counter = 0;
 //Counter of subtopologies
 counterSubtop = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Length of String   topology
 int m = topology.length();
 String[] subtopologies = new String[m];
 //String topology is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   String Sub = "";
   String a = "" + topology.charAt(i);
   int position = i+1;
   if (a.equals(pl))
   {
counter++;
 Sub =  Sub + a;
//The complete subtopology is captured:
```

```
//Parentheses must balanced
while (counter > 0)
 {
  String b = "" + topology.charAt(position);
  if (b.equals(pl))   counter++;
  if (b.equals(pr))  counter--;
  Sub =  Sub + b;
  position++;
 }
 if (print2)
   System.out.println(Sub);
   subtopologies[counterSubtop] = Sub;
   counterSubtop++;
  }
 }
 return subtopologies;
}



private static String generateTopology()
 {
 System.out.println("A random topology is generated");
   //Seed
   if (print1)
   System.out.println("\n" + parent
 + " -> "    + linkToNewick(parent));
 //Next leaf to be grafted
 nextLeafIndex = 2;
 //Next internal node to be inserted
 nextIntNodeIndex = 0;
   //A tree topology is generated
   String topology = parent;
   for(int i = 0; i < nLeavesToBeInserted; i++ )
   topology = generateChild(topology);
   return topology;
 }
```

```
//Leaves of subtopologies  are found
private static String[]leaves(int m, String[] subtop)
{
 if (print1)
  System.out.println("Leaves are found");
 String[] myLeaves = new String[m];
 for(int i = 0; i < m; i++)
  {
String s = subtop[i];
if (s == null);
  else
  {
   myLeaves[i] = "";
   int n = s.length();
   for(int j = 0; j < n; j++)
    {
     String c = "" + subtop[i].charAt(j);
     if (leaves.contains(c))
      myLeaves[i] = myLeaves[i] + c;
    }
   if (print1)
   System.out.println(subtop[i] + " "
                          + myLeaves[i] );
  }
   }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
 System.out.println("Leaves are listed orderly" );
 for(int i = 0; i < m; i++)
 {
String s = subtop[i];
if (s == null);
  else
  {
    s =   myLeaves[i];
    myLeaves[i] = "";
   int n = leaves.length();
   for(int j = 0; j < n; j++)
    {
```

```
     String c = "" + leaves.charAt(j);
     if (s.contains(c))
      myLeaves[i] = myLeaves[i] + c;
    }
   if (print1)
   System.out.println(subtop[i] + " " + myLeaves[i] );
  }
  }
 return myLeaves;
  }



 //Inner nodes of subtopologies  are found
 private static String[] inner(int m, String[] subtop)
 {
  if (print1)
  System.out.println("\nInternal nodes are found");
  String[] myNodes = new String[m];
  for(int i = 0; i < m; i++)
   {
  String s = subtop[i];
  if (s == null);
    else
    {
   myNodes[i] = "";
     int n = s.length();
     for(int j = 0; j < n; j++)
      {
       String c = "" + subtop[i].charAt(j);
       if (internalNodes.contains(c))
       myNodes[i] = myNodes[i] + c;
      }
    if (print1)
 System.out.println(subtop[i] + " " + myNodes[i] );
    }
    }
  //Chars of Leaves are rewritten in alphabet order
  if (print1)
  System.out.println("Inner nodes are listed orderly" );
```

```
   for(int i = 0; i < m; i++)
   {
   String s = subtop[i];
   if (s == null);
     else
     {
       s =   myNodes[i];
      myNodes[i] = "";
      int n = internalNodes.length();
      for(int j = 0; j < n; j++)
       {
        String c = "" + internalNodes.charAt(j);
        if (s.contains(c))
        myNodes[i] = myNodes[i] + c;
       }
     if (print1)
      System.out.println(subtop[i] + " " + myNodes[i] );
     }
    }
   return myNodes;
    }


 //Subtopologies are compared to decide whether
 //or not they have the same set of leaves
 //and the same set of inner nodes.



private static boolean comparison(String Leaves1[],
                      String Leaves2[],
                      String Inner1[],
                      String Inner2[])
 {
  System.out.println("\nComparison of leaves and nodes");
  //Print everything
  Boolean printAll = false;
  Boolean recAllowed = false;
  int m = Leaves1.length;
  int n = Leaves2.length;
```

```
  for(int i = 1; i < m; i++)
  {
   for(int j = 1; j < n; j++)
   {
if ( Leaves1[i].equals(Leaves2[j]))
{
 subTree1 = subtrees1[i];
 subTree2 = subtrees2[j];
 recAllowed = true;
 System.out.println(
 "Sets of leaves are EQUAL " + i +
 "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
 if ( Inner1[i].equals(Inner2[j]))
System.out.println(
        "Sets of inner nodes are EQUAL " + i +
    "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
  else
System.out.println(
        "Sets of inner nodes are DIFFERENT " + i +
"  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
}
else
if (printAll)
System.out.println(
        "Sets of leaves are DIFFERENT " + i +
   "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
   }
  }
  return recAllowed;
 }


private static  void recombination()
{
 treeRec1 = tree1.replace(subTree1, subTree2);
 treeRec2 = tree2.replace(subTree2, subTree1);
 System.out.println("\nRecombination is allowed:"
 + "\nOriginal trees \n"
 + tree1 + "\n" + tree2
 + "\nSwapping subtrees \n"
```

```
 +  subTree1 + "\n" + subTree2
 + "\nRecombinant trees \n"
 + treeRec1 + "\n" + treeRec2 );

}


 public static void main(String[] args)
 {
   //A topology is generated
   String topo1 = generateTopology();
   System.out.println(topo1 + " -> "
          + linkToNewick(topo1));
   tree1 = insertLengths(topo1);
   makeLengths();
 //Lengths of links are added to tree topology
   tree1 = insertLengths(topo1);
   if (print3)
   System.out.println("\n Tree1 = " + tree1);
   //Length of String   topology
   m1 = topo1.length();

    subtrees1 = new String[m1];
   subtrees1 = subtopologies(tree1);
   m1 = counterSubtop;
   //Subtopologies are found
   String[] subtopologies1 = new String[m1];
   for(int i = 0; i < m1; i++)
   subtopologies1[i] =  simplify(subtrees1[i]);



   System.out.println();
   String[] Leaves1 = new String[m1];
   Leaves1 = leaves(m1,subtopologies1);
   String[] Inner1 = new String[m1];
   Inner1 = inner(m1,subtrees1);
   //Chars of Leaves are rewritten in alphabet order
   System.out.println("\nSubtrees, subtopologies, Leaves, " +
                "Inner nodes" );
```

```
   for(int i = 0; i < m1; i++)
   {
 if (subtopologies1[i] == null);
  else
  {

   System.out.println(i
   + "   " + subtrees1[i]
   + "   " + subtopologies1[i]
   + "   " +  Leaves1[i] + "   " + Inner1[i]);
  }
   }

   System.out.println();

   //A second topology is generated
   String topo2 = generateTopology();
   System.out.println(topo1 + " -> "
          + linkToNewick(topo2));
   tree2 = insertLengths(topo2);
   makeLengths();
 //Lengths of links are added to tree topology
   tree2 = insertLengths(topo2);
   if (print3)
   System.out.println("\n Tree2 = " + tree1);
   //Length of String   topology
   m1 = topo2.length();

     //Subtopologies are found
   subtrees2 = new String[m1];
   subtrees2 = subtopologies(tree2);
   m1 = counterSubtop;

   String[] subtopologies2 = new String[m1];
   for(int i = 0; i < m1; i++)
   subtopologies2[i] =  simplify(subtrees2[i]);



   System.out.println();
```

```
   String[] Leaves2 = new String[m1];
   Leaves2 = leaves(m1,subtopologies2);
   String[] Inner2 = new String[m1];
   Inner2 = inner(m1,subtrees2);
   //Chars of Leaves are rewritten in alphabet order
   System.out.println("\nSubtopologies, Leaves, " +
                 "Inner nodes" );
   for(int i = 0; i < m1; i++)
   {
 if (subtopologies2[i] == null);
  else
  {

   System.out.println(i
   + "   " + subtrees2[i]
   + "   " + subtopologies2[i]
   + "   " +  Leaves2[i] + "   " + Inner2[i]);
  }
   }
   boolean recAllowed =
   comparison(Leaves1,Leaves2, Inner1,Inner2);
   if (recAllowed)
   recombination();
   else
   System.out.println("Recombination is not allowed");
 }
}//End of main class K32  SubTopol5
```

33 ***Exercise.*** *Run the program and play with the code. Make your best to make sure that the program is correctly composed: it does what it must.*

34 ***Exercise.*** *When recombination is allowed by the previous program and there are various pairs of subtrees that allow recombination, the last found pair is chosen. Modify the program in order that the selection of allowed topology be made at random.* *Answer*

35 ***Challenge.*** *Add to the previous program the capability to see a depiction of the trees that represent the products of recombination. This will help to test that recombination is correctly implemented.*

1.4 Evolution to the full

We learned in the previous volume how to implement mutation and recombination in within a tree topology. We have learned here how to implement mutation and recombination of topologies and trees in general that are built upon the same set of leaves. This means that we can now implement evolution of trees to the full to try to solve the problem we battle with in Vol X:

36 ***Fitting a tree to a distance matrix***

We are given a set of species L that live together with us and whose genomes have been sequenced or whose ethologies or anatomies or physiologies have been described and appear in appropriate texts. Using those texts we compute a matrix of pairwise distances between species. Our suggested method is the n-gram technology, which abstracts the mode of operation of the immune system. So, the n-grams distances in within pairs of texts is consigned in matrix D_L. Our problem is to find the best fitting tree T to D_L. That tree is the solution to the following optimization problem: *find the tree T with associated distance matrix D such that $d(D,D_L)$, the absolute distance between matrix D and the target matrix D_L, is a minimum.*

To make sure that our procedures are correct, we can run an immediate test: we must end with the same tree if we go forth from a tree to a distance matrix and back to the best fitting tree. Formally:

37 ***Requirement:*** *The best fitting to D_L, the associated distance matrix to a tree T with set of leaves L, must be T itself. By abuse of language, we will say the best fitting to D_T must be T itself.*

Now, evolutionary trees are iconic to evolution. Therefore and by our honor, this problem must be solved by evolution, which is a general method for solving optimization problems. We made a try in Vol X: a bundle of topologies was generated and after some initial phase, the best was chosen to next pursue the best tree with chosen topology which was found by means of evolution. This attack was found to be highly inefficient. For instance, a problem with 6 leaves was proposed. The 104 topologies were calculated and the best was picked after some initial phase, 30 generations with 100 individuals. That topology hosted the best fitting tree thus far with a distance to target matrix of 27. The fine tunning of lengths was a task given to evolution (same parameters as before). By generation 100000 the distance was downed to 4, a situation that remained invariant until generation 360000.

We are ready now to try evolution, mutation and recombination, over trees with the possibility to evolve also their topologies.

38 ***The next code includes evolution of topologies for proposing a solution to the problem of fitting a tree to a distance matrix.***

```
//Program K38 EvoFull
/*
 * We implement here
 * a full fledged evolutionary process
 * to propose a solution to the problem of
 * fitting a tree to a distance matrix.
 *
 *
 * We generate a random tree topology
 * and attach random integer lengths to links.
 * In that way we get the target tree, T.
 * Next, we find the associated distance matrix D  to T.
 * Our problem is:
 *
 * Find tree t such that the absolute distance between
 * matrix D and the associated matrix to t is a minimum.
 * Absolute distance between M and N = sum |Mij-Nij|.
 * This distance defines the FITNESS of each tree.
 * It decreases as good performance augments.
 * The best possible fitness is zero.
 *
 * We run the next test:
 * If we begin with a tree T, compute D, and then t,
 * we expect to return with the initial tree: t = T.
 *
 *
 * To associate a distance matrix to a tree,
 * we calculate the distance in within
 * each pair of nodes (leaves + inner nodes).
 * To that aim, we begin with the first node and
 * walk over the tree until the second node
 * and add the lengths of all traversed links.
 * The resulting number is the  distance.
```

```
*
*
* Formats = Link and Newick's notations. And variants.
*
* An automatic halting option has been added to be applied
* if perfect matching is found.
* The program accepts for input
* a distance matrix with integer values.
* A test for being a distance matrix is run.
*
* To find the tree that best fit D,
* we  use evolution as follows:
*
* 1) Initialization:
*  We generate N topologies  on the same
*  set of leaves. By adding   lengths to links,
*  every topology gets represented by a tree.
* The fitness of each tree
* is calculated and the best is selected.
*  If the fitness of the best is zero,
*  the program halts and the tree is reported.
*
* 2) Evolution is composed of rounds of
*     cloning according to fitness,
*     random mutation and recombination.
*       After each generation, fitness is checked
*     for accomplishment.
*
*
* Mutation is Lamarckian:
* it always produces viable mutants.
* Topologies as well as links can mutate.
*
* A mutation results from deleting
* a given number of leaves
* to reinsert them anew in
* other places chosen at random.
* This creates a mutant topology.
* To mutate length of links:
* Lengths are read from tree and
```

```
 * mutated separately.
 * Then, they are reinserted.
 *
 *
 * Recombination is lamarckian: it is authorized only
 * when it produces viable recombinants.
 * All their subtopologies are calculated
 * and packed in arrays.
 * The sets of leaves and of inner nodes
 * of each subtopology are found.
 * Subtopologies are compared by pairs,
 * one from each topology.
 * If they happen to have the same set of leaves
 * and of inner nodes, recombination is authorized:
 * the  subtrees that correspond with
 * the subtopologies are swapped.
 *
 * If more than one pair of subtopologies authorize
 * recombination, one is chosen at random.
 * A hashtable is used to memorize those pairs that do it.
 * Original topologies serves as subtopologies
 * that authorize recombination but they do not count.
 *
 *
 */



import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Random;

import javax.swing.JButton;
import javax.swing.JFrame;
```

```
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.event.TreeModelEvent;
import javax.swing.event.TreeModelListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreePath;
import javax.swing.tree.TreeSelectionModel;

import java.awt.*;



import javax.swing.JComponent;


public class EvoFull extends JPanel
                          implements ActionListener
{

 private static final long serialVersionUID = 1L;

 //The program accepts a distance matrix as input
 //else
 //it generates a random tree, calculates the distance
 //matrix associated to leaves and posits it
 //as target matrix.
 private static boolean inputMatrix = false;


 //Input distance matrix:
 //Example 5.4, pag 99, Isaev (2001)
 //Introduction to Mathematical Methods in Bioinformatics
 //Springer
 private static double targetMatrix[][]
 = {{0,8,3,14,10,12},
   {8,0,9,10,6,8},
   {3,9,0,15,11,13},
```

```
  {14,10,15,0,10,8},
  {10,6,11,10,0,8},
  {12,8,13,8,8,0}};

/*
private static double targetMatrix5[][]
= {{0,8,3,14,10},
  {8,0,9,10,6},
  {3,9,0,15,11},
  {14,10,15,0,10},
  {10,6,11,10,0}};

private static double targetMatrix4[][]
= {{0,8,3,14},
  {8,0,9,10},
  {3,9,0,15},
  {14,10,15,0}};

  */

//Number of leaves that must go in all trees > 3
private static int numberOfLeaves;

//Everything is built by recursion over this tree
private static   String   seed =
   "((Aa,Ba),Ca)";

//The best distance found in initialization
private static  int  distance0;


private static  String leaves =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
private static String internalNodes =
"abcdefghijklmnopqrstuvwxyz";

//Next leaf to be grafted
private static  int nextLeafIndex = 2;
//Next internal node to be inserted
private static  int nextIntNodeIndex = 0;
```

```
private static int maxNumberOfLeaves = leaves.length();



//Number of leaves to be deleted and reinserted
//to create a mutation
private static int nUndoRedo = 0;


private static boolean print1 = false;


//This is the original topology:
private static  String original = "";



//===========Most important parameters ===========

//Number of leaves to be inserted to seed:
//to get a tree with 4 leaves, add one leaf to seed
private static  int nAdditionalLeaves = 3;



//Number of topologies = number of trees that
//conform the evolving population
private static  int N = 50;

//Max number of generations to run evolution
static private int numberOfGenerations = 50000001;
static private long generation = 0;

//Mutation rate per site   per generation
//in the lengths of links
```

```
 static private double    m = 0.5;



//Report Best tree every lapse generations
static private int lapse = 1000;

 //================================================


 //Data for binary tree
 private static String[] Record =  new String[20*N];


 private static  int indexOfTopologies;
 private static String topologies[] = new String[N];
 private static String trees[] = new String[N];
 private static String newTrees[] = new String[N];

 //The distance to target of the best fitting tree
 //of the given topology
 private static double[] bestFitting =  new double[N];
 private static String[] bestTreeOfTopology
                             = new String[N];

 //Last filled position  in Record[]
 private static int lastUsedPosition;
 private static String selectedTree = "";


  //DynamicTree is an inner service class that comes
  //at the end of this file.
  private static DynamicTree15 treePanel;
  private static String VISUALIZE_COMMAND = "Visualize";

   //Length of the link that glues a leaf to the tree
  private static int lengthsLeaves[]  =
                new int[maxNumberOfLeaves];
  //Length of the link that glues
  //an inner node to another one
```

```
 private static int lengthsInnerLinks[]  =
             new int[maxNumberOfLeaves];


 private static Random r = new Random();
 //The control parameter of the random generator for int
 private static int maxLength = 10;

 //Paths from each leaf to node a
 private static String[] Paths =
             new String[maxNumberOfLeaves];

 private static String targetTree;



 //=======Evolution parameters============



static private double distance[] = new double[N];
static private double fitness[] = new double[N];



//Index of the best
static private int best;
//The best tree for a given topology
static private String bestTree = "";


//=========Debugging help ===============

private static boolean print  = false;

private static boolean print2 = false;
private static boolean print3 = false;
private static boolean print4 = false;
//Turn on print5 firstly
```

```
private static boolean print5 = false;
private static boolean print6 = false;
private static boolean print7 = true;


//==============

private static String titleFrame = "";
private static String title1 = "";
private static String title2 = "";
private static EvoFull newContentPane = new EvoFull();



 //=======Constructor of main class: GUI basics=====

 public EvoFull()
 {
  super(new BorderLayout());
  JPanel instructionPanel = new JPanel(new GridLayout(0,1));
  JLabel instructions1 = new JLabel(
     "Select  a tree and click the Visualize button.");


  instructionPanel.setFocusable(true);
  instructionPanel.add(instructions1);

  add(instructionPanel, BorderLayout.PAGE_END);

  //A new node is added
  JButton visualizeButton = new JButton("Visualize");
  visualizeButton.setActionCommand(VISUALIZE_COMMAND);
  visualizeButton.addActionListener(this);

  // Create the components.
  treePanel = new DynamicTree15();
  // Lay everything out.
  treePanel.setPreferredSize(new Dimension(700, 800));
  instructionPanel.add(visualizeButton);
  add(treePanel, BorderLayout.CENTER);
```

```
  }//End of constructor

  //Response to selecting visualization
  public void actionPerformed(ActionEvent e)
   {
String command = e.getActionCommand();
    if (VISUALIZE_COMMAND.equals(command))
 {
      //Selected topology must be read
      selectedTree = treePanel.unveilTree();
      System.out.println("\nSelected topology " +
       "in Link notation  after unveilTree() = \n"
       + selectedTree);
      /*Random lengths are added to topology
      generateLengths();
      //Lengths are added
      selectedTree = insertLengths(selectedTree);
      */
      int i =  findTheBest(selectedTree);
      selectedTree =  bestTreeOfTopology[i];
      double d = bestFitting[i];
      System.out.println( " Best fitting = \n " +
                                  selectedTree);
      System.out.println( " Best distance = \n "
       + d);


      decoderLengths(selectedTree);
  //Translation from one format into another
      //output = a vector with binary addresses and contents
      double[][] M =  associatedMatrix(selectedTree);
      printMatrix(M, "distance matrix associated to the " +
      "best tree for this topology");
      //Selected tree is drawn
      title1 = "Best tree for the topology = "
       + simplify(selectedTree);
      title2 = "Distance to target = " + d;
      DrawBiTree12 a = newContentPane.new DrawBiTree12();
      a.createAndShowGUI();
```

```
 }
  }

//======================
```

```
public static void main(String[] args)
{
   // Schedule a job for the event-dispatching thread:
   // creating and showing this application's GUI.
   javax.swing.SwingUtilities.invokeLater(new Runnable()
    {
     public void run()
     {
      //If input is a distance matrix:
      if (inputMatrix)
      {
       printMatrix(targetMatrix, "Input Matrix");
       boolean isD = isMatrixDistanceMatrix(targetMatrix);
         if (isD) System.out.println("\nMatrix " +
        "is a distance matrix");
         else
          {
       System.out.println("Matrix " +
            "is NOT a distance matrix");
       System.exit(0);
          }
        numberOfLeaves = targetMatrix.length;
        nAdditionalLeaves = numberOfLeaves-3;
    }
    else//if a random tree is generated -> Matrix
    {
     numberOfLeaves =    3 + nAdditionalLeaves;
     String tree = randomTopology();
     generateLengths();
     targetTree = insertLengths(tree);
```

```
System.out.println("Target tree: each link " +
"has its own length = \n" + targetTree);
targetMatrix  =
    new double[numberOfLeaves][numberOfLeaves];
targetMatrix = associatedMatrix(targetTree);
printMatrix(targetMatrix, "Target Matrix");

double d = absDistance(numberOfLeaves,
              targetMatrix, targetMatrix);


System.out.println( "\nDistance from target matrix " +
 "to target = " + d);




 //Draw Target tree: this malfunctions.
   /*
    {
     titleFrame = "Target tree = " + tree;
      selectedTree = targetTree;
      DrawBiTree12 b = newContentPane.new DrawBiTree12();
      b.createAndShowGUI();
      }
     */
}
createAndShowGUI();

System.out.println("\nDefinition: \nDistance to target = " +
      " Absolute distance between" +
     " target matrix  and \n" +
     "the distance matrix associated to " +
     "the tree with lengths of links that \n" +
     "is the best fitting  of the topology.\n" +
     "Perfect fitting: distance = 0 \n");
System.out.println("\nNumber of Generations = "
       + numberOfGenerations);
if (print7)
```

```
     System.out.println("Number of leaves = "
                         + numberOfLeaves +
                          "\nMutation rate = "   + m +
                  "\nPopulation size = " + N);

     initialization();
     evolution();

     }
    });
  }



//Test for being a distance matrix.
//MaxLenght = max value among entries is calculated.
private static boolean
    isMatrixDistanceMatrix(double[][] M)
{
 int l = M.length;
 boolean isD = true;
 //Is M(u,u) = 0?
 for(int u = 0; u < l; u++)

if (M[u][u] ==  0) ;
 else
  {
isD = false;
return isD;
  }

 //Is M(u,v) = M(v,u)?
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
if (M[u][v] == M[v][u] ) ;
 else
{
 isD = false;
 return isD;
```

```
}

 //Is M(u,v)>0 for u not v?
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
if (M[u][v] >  0) ;
 else
{
 isD = false;
 return isD;
}



 //Is M(u,v) < M(u,w) + M(w,v)?
 for(int u = 0; u < l; u++)
 for(int v = 0; v < l; v++)
 for(int w = 0; w < l; w++)

if (M[u][v] <= M[u][w]+M[w][v]  ) ;
 else
{
 isD = false;
 return isD;
}
 double m = 0;
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
 if (M[u][v] > m ) m = M[u][v];
 maxLength = (int) m + 1;
 return true;
}



//Simplification from Link notation to Newick's notation:
//(((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String linkToNewick(String newTree)
{
 String tree = "";
```

```
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
  }
 return tree;
}



private static String generateTopology()
 {
  //Next leaf to be grafted
  nextLeafIndex = 2;
  //Next internal node to be inserted
  nextIntNodeIndex = 0;
  //A tree topology is generated
  String topology = seed;
  for(int i = 0; i < nAdditionalLeaves; i++ )
 topology = generateChild(topology);
  return topology;
 }


//Random lengths are added to a tree in Link notation
private static void makeLengths()
{
//Lengths with digit values are generated
for(int i = 0; i < numberOfLeaves; i++)
 {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
 }
```

```
}

//The population is initialized with nTop topologies.
//Lengths are inserted to get nTop trees
//Fitness is assessed:
//Has the solution been guessed?
private static void initialization()
{
 System.out.println("\n====" +"INITIALIZATION : ");
 if (print5)
  System.out.println("\nSeed = " + seed
 + " -> "    + linkToNewick(seed));

 if (print7)
  System.out.println( N
   + " random topologies " +
   "and their trees are generated \n");

 //nTop topologies and trees are generated
 for(int i = 0; i < N; i++)
 {
  String topo1 = generateTopology();
  topologies[i] = topo1;
  String tree = insertLengths(topo1);
  makeLengths();
  //Lengths of links are added to tree topology
  tree = insertLengths(topo1);
  trees[i] = tree;
  if (print6)
  {
System.out.println("\n" +  i + "   " + topo1 + " -> "
          + linkToNewick(topo1));
System.out.println("Tree1 = " + tree);
  }

 }
 assessFitnesses();
 scaleFitnesses();
}
```

```
//=====================================
//============= EVOLUTION =============
//=====================================


//Fitness is calculated for each tree
//Fitness  =
//Distance between target matrix and
//the associated distance matrix to tree is calculated.
//The minimal distance is found:
//this renders the fittest tree.
//Has the solution been guessed?
private static double assessFitnesses()
{
 if (print6)
  System.out.println(  "\n=====Fitnesses are assessed");

 String tree = "";
 double d = 0;
for(int i = 0; i < N; i++)
{

 tree = trees[i];
 decoderLengths(tree);
 double M[][] = distanceMatrix(tree);
 //Absolute distance from the associated distance matrix
 //of the tree
 //to target matrix is evaluated
 d = absDistance(numberOfLeaves,targetMatrix, M);
 distance[i] = d;


 if (print6)
  {
   System.out.println( "\nTree = \n" + trees[i]);
   System.out.println( "\nDistance to target = " + d);
  }
```

```
if (print5)
 {
  printMatrix(M, "Distance matrix associated to tree");
  printMatrix(targetMatrix, "Target matrix");
}

//If solution has been found, halt.
if (d == 0)
 {
  System.out.println( "\n \nSOLUTION HAS BEEN FOUND: \n" +
"at generation " + generation);
  System.out.println( " Distance to target = " + d
   + " \nSolution tree = " + tree
   + " \nTarget tree   = " + targetTree
   + " \nInitial distance = " + distance0);
printMatrix(M, "Distance matrix associated to tree");
printMatrix(targetMatrix, "Target matrix");
if (d == 0) System.exit(0);
if (print5) print5 = false;
 }
}//End of for(int i...



//The fittest individual minimizes distance function
double min = distance[0];
best = 0;
for(int i = 1; i < N; i++)
{
 if (distance[i] < min )
  {
   min = distance[i];
   best = i;
  }
}

 //Best tree
 bestTree = trees[best];
 if (generation == 0 ) distance0 = (int) min;
```

```
 if (print7)
 System.out.println("\n BestTree  = \n" +  bestTree
           + "\nDistance = " + min
           + "\ntarget tree = \n" + targetTree
           + "\nInitial distance = " + distance0);
 return min;
}



//Fitness is rescaled to take values
//in within 0(the worst) and 1(the best)
private static void scaleFitnesses()
{
if (print6)
 System.out.println(  "\n====Fitnesses are rescaled: " +
  "0(the worst) and 1(the best)");
double maxD = distance[0];
double minD = distance[0];
double b = 1 ;
//Min and max distances are found
for(int i = 1; i < N; i++)
 {
if ( distance[i] > maxD ) maxD =  distance[i];
if ( distance[i] < minD ) minD =  distance[i];
 }

if (minD == maxD) //All   equally fit
 {
  for(int i = 0; i < N; i++)
 fitness[i] = 1;
 }
else
 {
 //Distance to target determine fitness:
//fitness[maxD] = 0;    fitness[minD] = 1;
 for(int i = 0; i < N; i++)
 {
  b = 1;
  b = b/(minD - maxD);
```

```
  double a = -b*maxD;
  fitness[i] = a + b * distance[i];
}
 }
if (print6)
 for(int i = 0; i < N; i++)
 {
 System.out.println("Individual = "
            + i + " has distance " +
          distance[i] + " and fitness " + fitness[i]);
 }
}

//Probability of cloning is proportional to fitness.
//N new individuals are formed.
private static void selectiveCloning()
{
 if (print6)
  System.out.println(  "\n====Selective cloning");
//The best is cloned
 newTrees[0] = trees[best];
 if (print6)
 System.out.println("Individual = "
                     + best + " with fitness " +
 fitness[best] + "-> new individual " + 0
 + " = "  + newTrees[0]  );
//N-1 children are sampled
for(int i = 1; i < N; i++)
{
 //flag
 boolean go = true;
 while (go)
  {
 //An individual is chosen at random
 int k = r.nextInt(N);
 //Probability of cloning is proportional to fitness.
 double q = r.nextDouble();
 if ( q < fitness[k]) //make a copy of k
  {
 newTrees[i] = trees[k];
```

```
 if (print6)
 System.out.println("Individual = "
                      + k + " with fitness " +
 fitness[k] + "-> new individual " + i
 + " = "  + newTrees[i]  );

 go = false;//end task
  }
  }

}//End of for //N-1 children are sampled
}

//============Mutation============




//Detect internal nodes, different than leaves
private static boolean charIsInternalNode(char c)
{
 String s = "" + c;
 if (internalNodes.contains(s)) return true;
 else return false;
}



//Simplification from Link notation to Newick's notation:
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTopo)
{
String topo = "";
//length of string newTopo
int l = newTopo.length();
for(int i = 0; i < l; i++ )
 {
  //newTopo is scanned char by char
 char c = newTopo.charAt(i);
```

```
 //Is c an internal node to be deleted?
 if (charIsInternalNode(c));
  else //add char c to output
   topo = topo + c;
 }
return topo;
}

//Rule 2 is used to generate a new topology:
//(S)I -> ((S)N,FN)I.
//S = subtopology, I = internal node, F = new leaf,
//N = new internal node.
//Rule 2 grafts new branch FN to link (S)I
//i is the position of char I in string parentTopo
private static String useRule2(String parentTopo, int i,
       char F, char N)
{
 String newTopo = "";
 char I = parentTopo.charAt(i);
 if (print1)
  System.out.println("i = " + i + " I = " + I);
 //Subtopology S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = i;
 boolean go = true;
 //All right parentheses to the near left of I are captured
 while (go)
  {
   position = position -1;
   String a = "" + parentTopo.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
    {
```

```
 counter++;
 S = a + S;
    }
   else go = false;
  }//End of while
 if (print1)
 System.out.println("counter = " + counter);
 //The complete subtopology is captured:
 //Parentheses must balanced
 while (counter > 0)
  {
   String a = "" + parentTopo.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
    System.out.println("Subtopo = " + S);
   position = position -1;
   }
 //Replacement rule 2:
 //((S)n,Fn)i -> (S)i
 //where   F are leaves, n and i are  inner nodes,
 //(S) is a complex structure.
 String SnFni  =   S  + I;
 String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
 //Rule 1 is executed
 newTopo = parentTopo.replace(SnFni, replacement);
 //If I links to a complex structure
 return newTopo;
}

//From parentTopo, a topology in Newick's notation,
//a Child  topology is generated by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTopo)
{

 nextLeafIndex++;
 char nextLeaf= leaves.charAt(nextLeafIndex);
```

```
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = internalNodes.charAt(nextIntNodeIndex);


 //link to be replaced
 String link = "";
 //String link is replaced by what:
 String replacement = "";
 //new topology generated by replacement
 String newTopo = "";
 //length of parent topology
 int l = parentTopo.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
  {
   //parentTopo is scanned char by char
   char I = parentTopo.charAt(i);
   //Is I an internal node that denotes a link?
   if (charIsInternalNode(I)) counter++;
 }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
 {
   //parentTopo is scanned char by char
char I = parentTopo.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = parentTopo.charAt(i-1);
//If I links to a complex structure
```

```
if (a == ')') newTopo = useRule2(parentTopo, i,
               nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTopo = parentTopo.replace(link, replacement);
 }
if (print2)
System.out.println(newTopo + " -> "
                      + simplify(newTopo));


 i = l;
   }
  counter++;
}
 }
return newTopo;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply the antirule1:
//(An,Fn)i  -> Ai
private static String  antiRule1(String topo,
int positionLeaf, int positionNode)
{
 if (print2)
  System.out.println("AntiRule1");
 int beginIndex = positionLeaf - 4;
 int endIndex = positionNode + 3;
 String AnFni = topo.substring(beginIndex, endIndex);
 if (print2)
```

```
  System.out.println("Anfni = " + AnFni);
 String Ai = "" + topo.charAt(beginIndex+1)
               + topo.charAt(endIndex-1);
 if (print2)
  System.out.println("Ai = " + Ai);
 topo = topo.replace(AnFni, Ai);
return topo;
}



//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply the antirule2:
//((S)n,Fn)i -> (S)i
//positionNode refers to the last appearing of node n
private static String  antiRule2(String topo,
int positionLeaf, int positionNode)
{
 if (print2)
System.out.println("AntiRule2");
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 if (print2)
  System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
            + internalNodes.charAt(nextIntNodeIndex);
 int firstPosition = topo.indexOf(innerNode);
 String newTopo = "";
 char i = topo.charAt(positionNode+2);
 if (print2)
  System.out.println("FirstPosition = " +
                     firstPosition + " i = " + i);
 //Subtopology S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
```

```
//Position to be scanned
int position = firstPosition;
boolean go = true;
//All right parentheses to the near left of I are captured
while (go)
 {
  position = position -1;
  String as = "" + topo.charAt(position);

  if (print2)
   System.out.println("a = " + as + " p = " + pr);
  if (as.equals(pr) )
   {
counter++;
S = as + S;
   }
  else go = false;
 }//End of while
if (print2)
System.out.println("counter = " + counter);
//The complete subtopology is captured:
//Parentheses must balanced
while (counter > 0)
 {
  String as = "" + topo.charAt(position);
  if (as.equals(pr))   counter++;
  if (as.equals(pl))  counter--;
  S = as + S;
  if (print2)
   System.out.println("Subtopo = " + S);
  position = position -1;
 }
String n = innerNode;
//Replacement antiRule 2:
//((S)n,Fn)i -> (S)i
//where A and F are leaves, n and i are  inner nodes,
//(S) is a complex structure.
String SnFni  =  "("+ S  + n + "," + leaf + n + ")" + i ;
String Si = S + i;
if (print2)
```

```
  System.out.println("\nSnFni = " + SnFni + "\n Si = " + Si);
 //Rule 1 is executed
 newTopo = topo.replace(SnFni, Si);
 return newTopo;
}



//The last inserted  leaf is deleted from topology.
private static String deleteLastLeaf(String topology)
{
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 if (print2)
 System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
           + internalNodes.charAt(nextIntNodeIndex);
 if (print2)
 System.out.println("Accompanying inner  node = "
                   + innerNode);
 //new topology = output
 String newtopology = "";
 //length of  topology as string
 int l = topology.length();
 //Last positions are found
 int positionLeaf = -1;
 int positionNode = -1;
 //topology is scanned char by char
 for(int i = 0; i < l; i++ )
  {
 String c = "" + topology.charAt(i);
 if (c.equals(leaf)) positionLeaf = i;
 if (c.equals(innerNode)) positionNode = i;
  }
 if (print2)
  {
   System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
   System.out.println("Position of " + innerNode +
```

```
         " = " + positionNode);
  }
 int firstPosition = topology.indexOf(innerNode);
 //Check for the char before firstPosition
 char a = topology.charAt(firstPosition - 1);
 //If I links to a complex structure
 if (a == ')')
 newtopology = antiRule2(topology,positionLeaf,positionNode);
 else  //Apply antiRule1 for simple cases
  newtopology = antiRule1(topology,positionLeaf,positionNode);
 nextLeafIndex--;
 nextIntNodeIndex--;
 return newtopology;
}



//==========================

//Lengths are deleted from newTree
private static String deleteLengths(String newTree)
{
 String tree = "";
 String numeric = ".:0123456789";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
   char c = newTree.charAt(i);
   String s = "" + c;
   //Is c numeric: delete it!
   if (numeric.contains(s)); //add char c to output
  else tree = tree + c;
  }
 return tree;
}
```

```
//Mutation:
//some leaves are deleted and then reinserted
//in random order.
//Deleting can happen in either of two ways:
//Reversing the order of synthesis else
//in random order.
public static String mutation(String topology,
                      int nAdditionalLeaves)
{
 //Mutation has two stages:
 //deletion of some leaves that undoes generating work +
 //re-insertion in places chosen at random.

 //Number of leaves to be deleted and then reinserted
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
  else
   nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;


 if (print2)
  System.out.println("Number of leaves to" +
" be deleted and then reinserted = " +
nUndoRedo);

 //Some leaves are deleted:
 //reversed order to that of synthesis.
 if (print1)
  System.out.println("Some leaves are deleted");
 for(int i = 0; i < nUndoRedo; i++ )
 {
   //Last inserted leaf is deleted.
   topology = deleteLastLeaf(topology);
   if (print1)
    System.out.println(topology);
 }
 //deleted leaves are reinserted
 if (print1)
  System.out.println("Deleted leaves are reinserted");
```

```
 for(int i = 0; i < nUndoRedo; i++ )
  {
  topology = generateChild(topology);
  }



 //Output
 String mutant = topology;
 if (print1)
  System.out.println("\nOriginal and mutant topologies = \n"
  + original + "\n" + mutant);
 if (print1)
 if (original.equals(mutant))
  System.out.println("Mutation made nothing");
   else
  System.out.println("\nMutation worked");
 return mutant;
}


public static String mutation(String topology)
{
 //This original topology is next subject to mutation
 original = topology;
 String mutantTopology = topology;
 if (print2)
System.out.println("\nOriginal   topology = \n"
                + original  );
 //Number of leaves to be deleted and then reinserted
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
else
 nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;
 //nUndoRedo = 13;

 //Mutation has two stages:
 //deletion of some leaves that undoes generating work +
 //re-insertion in places chosen at random.

 if (print2)
```

```
  System.out.println("Number of leaves to" +
   " be deleted and then reinserted = " +
   nUndoRedo);
 //Some leaves are deleted:
 if (print2)
 System.out.println("Some leaves are deleted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
//Last inserted leaf is deleted.
mutantTopology = deleteLastLeaf(mutantTopology);
if (print2)
   System.out.println(mutantTopology);
  }
 //Some leaves are reinserted
 if (print2)
 System.out.println("Deleted leaves are reinserted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
   topology = generateChild(topology);
  }
 return mutantTopology;
}



//Random lengths are added to a tree in Link notation
public static String insertLengths(String treeLinkN)
{
 String thisChar = "";
 String newTree = "";
 //length of String   treeLinkN
 int m = treeLinkN.length();
 //TreeLinkN is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   thisChar = "" + treeLinkN.charAt(i);
   newTree = newTree + thisChar;
   if (print1)
    System.out.println(newTree.length() + " " + newTree);
   //Lengths are appended
```

```
  if (leaves.contains(thisChar))
   {
    int k = leaves.indexOf(thisChar);
    double j = lengthsLeaves[k];
    newTree = newTree + ":" + j;
   }
 if (internalNodes.contains(thisChar))
  {
   int k = internalNodes.indexOf(thisChar);
   double j = lengthsInnerLinks[k];
   newTree = newTree + ":" + j;
  }
 }
 return newTree;
}


//Input: string tree with complete information.
//Work: lengthsLeaves[] and lengthsInnerLinks[]
//are read from tree.
private static void decoderLengths(String tree)
{
 if(print2)
 System.out.println(
"Input tree at decoverLengths = \n "   + tree  );
 //Delete :
 tree = tree.replaceAll(":", "");
 if(print2)
  System.out.println(
"Input tree without : = \n "   + tree  );
 int n = tree.length();
 String thisChar = "";
 String nextChar = "";
 String numbers = ".0123456789";
 boolean leaf = false;
 boolean node = false;
 String Leaf = "";
 String numberLeaf = "";
 String numberNode = "";
 String Node = "";
```

```
 int j ;
 double f;

 //All chars in tree are scanned
 for(int i=0; i < n; i++)
 {
  thisChar = "" + tree.charAt(i);
  if (i < n-1) nextChar  = "" + tree.charAt(i+1);
  //Do we have a leaf?
  if (leaves.contains(thisChar))
   {
 leaf = true;
 Leaf = "" + thisChar;
   }

  //Do we have an internal node?
  if (internalNodes.contains(thisChar))
   {
 node = true;
 Node = "" + thisChar;
   }

  //Do we have a number that belongs in a leaf?
  if ( (leaf) & (numbers.contains(thisChar)))
{
 numberLeaf = numberLeaf + thisChar;
}

  //Do we have a number that belongs in a leaf?
  if ( (node) & (numbers.contains(thisChar)))
    {
  numberNode = numberNode + thisChar;
 }

  //Must we end accumulation of numbers of a leaf?
  if ( (leaf) & (internalNodes.contains(nextChar)))
 {
  leaf = false;
  j = leaves.indexOf(Leaf);
  if (print1)
```

```
  System.out.println("Leaf = " + Leaf +
  " numberLeaf = " + numberLeaf);
  f = new Double(numberLeaf);
  lengthsLeaves[j] = (int) Math.floor(f);
  numberLeaf = "";
 }

  //Must we end accumulation of numbers of a node?
  if ( (node) & (!(numbers.contains(nextChar))))
 {
  node = false;
  j = internalNodes.indexOf(Node);
  if (print1)
  System.out.println("Node = " + Node);
  f = new Double(numberNode);
  lengthsInnerLinks[j] = (int) Math.floor(f);
  numberNode = "";
 }

 }
}

//Random lengths are added to a tree in Link notation
public static void generateLengths()
{
 //Lengths with digit values are generated
 for(int i = 0; i < numberOfLeaves; i++)
  {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
  }
}


//Lengths undergo mutation
private static  void mutateLengths()
{

 //Mutation in links that connect leaves
```

```
 for(int k = 0; k < numberOfLeaves; k++)
  {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsLeaves[k] = r.nextInt(maxLength);

  }
 //Mutation in links that connect inner nodes
 for(int k = 0; k < numberOfLeaves-2; k++)
  {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsInnerLinks[k] = r.nextInt(maxLength);

  }



}



//Type of  mutation is decided and executed:
//mutation of lengths of links
//else full mutation (topology and lengths of links).
private static String mutateTree(String tree)
{
 //Topology is read from tree
 String topology = deleteLengths(tree);
 //Lengths are read from tree and recorded in vectors
 //lengthsLeaves[] and lengthsInnerLinks[]
 decoderLengths(tree);
 mutateLengths();
 String  mutantTree = "";
 //Mutate lengths else topology ( + new lengths)?
 double s = r.nextDouble();
 if ( s < 0)
{
 if (print1)
 System.out.println("\n====Mutation of lengths");
```

```
  mutantTree =  insertLengths(topology);

  if (print1)
  {
  System.out.println("Input tree =  " + tree);
  System.out.println("Topology =  " + topology);
  System.out.println("Output tree= " + mutantTree);
  }
 }
else
 {
 if (print1)
  System.out.println("\n====Mutation of topology");


//Mutation of topology
  String mutantTopology = mutation(topology,
                           nAdditionalLeaves);
  //Mutant lengths are reinserted
  mutantTree =  insertLengths(mutantTopology);
 }

 return mutantTree;
}



//Individuals are mutated
private static void mutation()
{
 if (print6)
  {
System.out.println("\n====Mutation \n");

System.out.println("Trees to be mutated");
  }
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
```

```
  if (print6)
  System.out.println(trees[i]);
 }
 //The old best, i = 0, is conserved intact
 newTrees[0] = trees[0];
 //Other N-1 individuals suffer mutation
 for(int i = 1; i < N; i++)
  newTrees[i] = mutateTree(trees[i]);


if (print6)
System.out.println("Mutant trees");
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print6)
  System.out.println(trees[i]);
 }
}

//==========Recombination============


 //Counter of subtopologies
 private static  int counterSubtop;
 //Particular number of subtopologies
 private static  int m1 = 0;



 private static  String[] subtrees1, subtrees2;
 private static  String   subTree1, subTree2,
                      treeRec1, treeRec2;

 //Structure of hashtable : integer + {string1, string2}
 private static Hashtable<Integer, String[]> hashtable =
 new  Hashtable<Integer, String[]>();
```

```
//========== Subtrees are listed ============


//Subtrees of  topology are found
private static String[] subtopologies(String topology)
{
 if (print1)
  System.out.println("\nSubtrees are found");
 //Counter of left parenthesis
 int counter = 0;
 //Counter of subtopologies
 counterSubtop = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Length of String   topology
 int m = topology.length();
 String[] subtopologies = new String[m];
 //String topology is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   String Sub = "";
   String a = "" + topology.charAt(i);
   int position = i+1;
   if (a.equals(pl))
   {
    counter++;
    Sub =  Sub + a;
    //The complete subtopology is captured:
    //Parentheses must balanced
   while (counter > 0)
    {
     String b = "" + topology.charAt(position);
     if (b.equals(pl))   counter++;
     if (b.equals(pr))  counter--;
     Sub =  Sub + b;
     position++;
  }
 if (print2)
```

```
   System.out.println(Sub);
 subtopologies[counterSubtop] = Sub;
 counterSubtop++;
 }
}
return subtopologies;
}


//Leaves of subtopologies  are found
private static String[]leaves(int m, String[] subtop)
{
 if (print1)
  System.out.println("Leaves are found");
 String[] myLeaves = new String[m];
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
   else
    {
     myLeaves[i] = "";
     int n = s.length();
     for(int j = 0; j < n; j++)
      {
       String c = "" + subtop[i].charAt(j);
       if (leaves.contains(c))
       myLeaves[i] = myLeaves[i] + c;
     }
   if (print1)
     System.out.println(subtop[i] + " "
                        + myLeaves[i] );
   }
  }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
  System.out.println("Leaves are listed orderly" );
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
```

```
   if (s == null);
    else
     {
      s =   myLeaves[i];
      myLeaves[i] = "";
      int n = leaves.length();
   for(int j = 0; j < n; j++)
    {
     String c = "" + leaves.charAt(j);
     if (s.contains(c))
     myLeaves[i] = myLeaves[i] + c;
    }
   if (print1)
   System.out.println(subtop[i] + " " + myLeaves[i] );
  }
 }
 return myLeaves;
}



//Inner nodes of subtopologies  are found
private static String[] inner(int m, String[] subtop)
{
 if (print1)
  System.out.println("\nInternal nodes are found");
 String[] myNodes = new String[m];
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
    else
     {
      myNodes[i] = "";
      int n = s.length();
      for(int j = 0; j < n; j++)
       {
        String c = "" + subtop[i].charAt(j);
        if (internalNodes.contains(c))
        myNodes[i] = myNodes[i] + c;
```

```
       }
    if (print1)
System.out.println(subtop[i] + " " + myNodes[i] );
  }
 }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
 System.out.println("Inner nodes are listed orderly" );
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
    else
     {
      s =   myNodes[i];
      myNodes[i] = "";
      int n = internalNodes.length();
      for(int j = 0; j < n; j++)
       {
        String c = "" + internalNodes.charAt(j);
        if (s.contains(c))
      myNodes[i] = myNodes[i] + c;
       }
  if (print1)
   System.out.println(subtop[i] + " " + myNodes[i] );
 }
}
return myNodes;
}


//Subtopologies are compared to decide whether
//or not they have the same set of leaves
//and the same set of inner nodes.
//All pairs that authorize recombination
// are kept in a hastable (=  a table for pairing)
private static boolean comparison(String Leaves1[],
                      String Leaves2[],
                      String Inner1[],
                      String Inner2[])
```

```
{
 if (print6)
 System.out.println("\nComparison of leaves and nodes");
 Boolean recAllowed = false;
 int m = Leaves1.length;
 int n = Leaves2.length;
 Integer counter = 0;
 for(int i = 1; i < m; i++)
  {
   for(int j = 1; j < n; j++)
    {
     if ( Leaves1[i].equals(Leaves2[j]))
      {
       subTree1 = subtrees1[i];
       subTree2 = subtrees2[j];
       recAllowed = true;
       if (print6)
        System.out.println(
          "Sets of leaves are EQUAL " + i +
        "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
        String val[] = {subTree1, subTree2};

        //Recombinant subtrees are kept in a hashtable
        hashtable.put(counter,val);
        counter++;
        if ( Inner1[i].equals(Inner2[j]))
         {
      if (print6)
      System.out.println(
        "Sets of inner nodes are EQUAL " + i +
          "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
         }
        else
         {
       if (print6)
  System.out.println(
      "Sets of inner nodes are DIFFERENT " + i +
      "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
         }
     }
```

```
    else
  if (print6)
   System.out.println(
       "Sets of leaves are DIFFERENT " + i +
        "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
   }
 }
 return recAllowed;
}


//A pair of subtrees that authorize recombination
//is chosen at random from the hashtable and
//recombination is executed.
private static  String recombinationTrees(String tree1,
                                          String tree2)
{
 //Possible pairs of recombinant subtrees are recovered
 //from the hashtable and put into an enumeration<>
 Enumeration<Integer> enumKey = hashtable.keys();
 int counter = 0;
 while(enumKey.hasMoreElements())
  {
   Integer key = enumKey.nextElement();
   String[] val = hashtable.get(key);
   if (print3)
    System.out.println("Hasthtable " + key    + "\n"
 + val[0]  + "\n"+  val[1]);
   counter++;
   }

 //A pair of recombinant subtrees is chosen at random
 Integer election = r.nextInt(counter);
 if (print6)
  System.out.println("Selected pair = " + election);
 String[] val = hashtable.get(election);
 subTree1 = val[0];
 subTree2 = val[1];
 if (print6)
   System.out.println("Recombinant subtrees \n" +
```

```
      subTree1 + "\n" +  subTree2);

 //Recombination is executed
 treeRec1 = tree1.replace(subTree1, subTree2);
 treeRec2 = tree2.replace(subTree2, subTree1);
 if (print6)
  System.out.println("\nRecombination is allowed:"
 + "\nOriginal trees \n"
 + tree1 + "\n" + tree2
 + "\nSwapping subtrees \n"
 +  subTree1 + "\n" + subTree2
 + "\nRecombinant trees \n"
 + treeRec1 + "\n" + treeRec2 );
 return treeRec1;
}


//Recombination of tree1 and tree2 is tried out
public static String recombinationMaster(String tree1,
                              String tree2)
{
 //First topology
 String topo1 = deleteLengths(tree1);
 if (print6)
  {
   System.out.println(topo1 + " -> "
       + linkToNewick(topo1));
   System.out.println("\n Tree1 = " + tree1 +
 "\n Topo1 = " + topo1);
  }


 //Length of String   topology
 m1 = topo1.length();

 subtrees1 = new String[m1];
 subtrees1 = subtopologies(topo1);
 m1 = counterSubtop;
 //Subtopologies are found
 String[] subtopologies1 = new String[m1];
```

```
for(int i = 0; i < m1; i++)
 subtopologies1[i] =  simplify(subtrees1[i]);
if (print6)
System.out.println();
String[] Leaves1 = new String[m1];
Leaves1 = leaves(m1,subtopologies1);
String[] Inner1 = new String[m1];
Inner1 = inner(m1,subtrees1);
//Chars of Leaves are rewritten in alphabet order
if (print5)
 System.out.println("\nSubtrees, subtopologies, Leaves, " +
            "Inner nodes" );
for(int i = 0; i < m1; i++)
 {
  if (subtopologies1[i] == null);
  else
   {
if (print5)
       System.out.println(i
  + "   " + subtrees1[i]
  + "   " + subtopologies1[i]
  + "   " +  Leaves1[i] + "   " + Inner1[i]);
   }
 }

if (print6)
 System.out.println();

//Second topology
String topo2 = deleteLengths(tree2);
if (print6)
 {
  System.out.println(topo1 + " -> "
  + linkToNewick(topo2));

System.out.println("\n Tree2 = " + tree2 +
"\n Topo2 = " + topo2);
 }
//Length of String   topology
m1 = topo2.length();
```

```
subtrees2 = new String[m1];
subtrees2 = subtopologies(topo2);
m1 = counterSubtop;
//Subtopologies are found
String[] subtopologies2 = new String[m1];
for(int i = 0; i < m1; i++)
 subtopologies2[i] =  simplify(subtrees2[i]);


if (print6)
 System.out.println();
String[] Leaves2 = new String[m1];
Leaves2 = leaves(m1,subtopologies2);
String[] Inner2 = new String[m1];
Inner2 = inner(m1,subtrees2);
//Chars of Leaves are rewritten in alphabet order
if (print5)
 System.out.println("\nSubtopologies, Leaves, " +
            "Inner nodes" );
for(int i = 0; i < m1; i++)
 {
  if (subtopologies2[i] == null);
  else
   {
if (print5)
     System.out.println(i
  + "   " + subtrees2[i]
  + "   " + subtopologies2[i]
  + "   " +  Leaves2[i] + "   " + Inner2[i]);
   }
 }
boolean recAllowed =
  comparison(Leaves1,Leaves2, Inner1,Inner2);
String recombinant = "";
if (recAllowed)
  recombinant = recombinationTrees(tree1,tree2);
else
 {
  recombinant = tree1;
```

```
   if (print6)
  System.out.println("Recombination is not allowed");
  }
 return recombinant;
}


//Recombination methods may produce an output
//with various and different assignment of lengths to
//inner node a. One is chosen.
public static void correction()
{
String topo;
 for(int i = 1; i < N; i++)
  {
decoderLengths(newTrees[i]);
topo = deleteLengths(newTrees[i]);
newTrees[i] = insertLengths(topo);
  }
}

//Each one of the N individuals  are tried out
//for recombination: a partner is chosen at random
//and if recombination is allowed, it is executed.
//Otherwise, it is cloned.
public static void recombination()
{
 if (print6)
 System.out.println( "\n====RECOMBINATION \n");
 //Best individual is excluded from recombination
  newTrees[0] = trees[0];

 for(int i = 1; i < N; i++)
  {
   //A partner for i is chosen
   int j = i;
   String t1 = "";
   String t2 = "";
   while (i == j)
```

```
 j =   r.nextInt(N);
   if (print6)
System.out.println( "\n==RECOMBINATION OF INDIVIDUALS "
+ i + " and " + j);
   t1 = trees[i];
   t2 = trees[j];
   newTrees[i] = recombinationMaster(t1,t2);
   correction();
  }
 if (print6)
 System.out.println("Recombination output");
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print6)
  System.out.println(trees[i]);
 }
}



//Evolution of trees involving
//recombination and mutation of topologies,
//in within and between.
private static void evolution()
{
 for(int i = 1; i < numberOfGenerations; i++)
 {
generation = i;
  if ( (i % lapse) == 0)
  print7 = true;
  else print7 = false;

  if (print7)
  System.out.println( "\n===GENERATION  = " + i);
  selectiveCloning();
  mutation();
  recombination();
  assessFitnesses();
  scaleFitnesses();
```

```
 }
}
```

```
//A tree topology is generated at random.
public static String randomTopology()
 {
  if (print2)
  System.out.println("Parent tree = \n" + seed
   + " -> "    + simplify(seed));
  String tree = seed;
  int limit = numberOfLeaves-3; //less than 11.
  for(int i = 0; i < limit; i++ )
  tree = generateChild(tree);
  System.out.println("Target topology = \n" + tree);
  return tree;
 }


//======================
```

```
//The best fitting tree for chosen topology is found
public static int findTheBest(String tree)
{
 String selectedTopology = simplify(tree);
 String thisTopology = "";
 String thisTree = "";
 int bestPosition = -1;

for(int i = 0; i < indexOfTopologies; i++)
 {
  thisTree = bestTreeOfTopology[i];
  thisTopology = simplify(thisTree);
```

```
  if (selectedTopology.equals(thisTopology))
  {
   bestPosition = i;
   return bestPosition;
  }
 }
return  bestPosition;
}


  /*
  //Lengths with real values are generated
  double lengthsLeaves[]  = new double[l];
  //Last two entries are redundant
  double lengthsInnerLinks[]  = new double[l];
  for(int i = 0; i < numberOfLeaves; i++)
   {
 lengthsLeaves[i] = r.nextDouble();
 lengthsInnerLinks[i] =  r.nextDouble();
   }
  */



  //Data about the tree is printed to the console
  public void report()
  {
System.out.println("The root has " +
      treePanel.rootNode.getChildCount() + " children");
System.out.println("The tree's depth is " +
      treePanel.rootNode.getDepth());
System.out.println("The tree has " +
      treePanel.rootNode.getLeafCount() + " leaves");
 System.out.println("'Root' is really a root? " +
      treePanel.rootNode.isRoot());
 System.out.println("Root's userObject: " +
      treePanel.rootNode.toString());
  }
```

```
  /**
   * Create the GUI and show it. For thread safety,
   * this method should be
   * invoked from the event-dispatching thread.
   */
private static void createAndShowGUI()
 {
  // Create and set up the window.
  JFrame frame =
new JFrame("Tree of tree topologies.");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  // Create and set up the content pane.

  // content panes must be opaque
  newContentPane.setOpaque(true);
  frame.setContentPane(newContentPane);

  // Display the window.
  frame.pack();
  frame.setVisible(true);
 }
```

```
//========= Translation in within formats ==========


//bTree in (modified) Newick's notation
//is used to generate Record[]
//with addresses and contents.
public static void makeInput(String bTree)
{
 if (print)
```

```
 System.out.println("bTree at makeInput ////// " + bTree);
 int counter = 0;
 //length of String   tree
 int l = bTree.length();
 String thisChar = "";
 String address = "";
 String binNumbers = "01";
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + bTree.charAt(i);
if (binNumbers.contains(thisChar))
   address = address + thisChar;
if (leaves.contains(thisChar))
{
 Record[counter] = address;
     counter++;
 Record[counter] = thisChar;

counter++;
address = "";
   }
  }

 if (print)
for(int i = 0; i < counter; i = i+2 )
  System.out.println(Record[i] +
 " bbbb " + Record[i+1]);

lastUsedPosition = counter;
if (print)
System.out.println("lastUsedPosition ==== "
        + lastUsedPosition);
}



//Outputs a word in reversed order.
private static String reverser(String word)
{
```

```
 return new StringBuilder(word).reverse().toString();
}


private static String cleanTree(String tree)
{
 if (print2)
 System.out.println("Input tree at cleanTree = \n "
        + tree);
 String treeClean = "";
 String numeric = ".:0123456789";
 String plusMinus = "+-";
 String thisChar = "";
  boolean leaf = false;
 //Numerics are washed
 //length of String   tree
 int l = tree.length();
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + tree.charAt(i);
   if (numeric.contains(thisChar));
    else treeClean = treeClean + thisChar;
  }
 if (print2)
 System.out.println("Tree without numerics = \n"
+ treeClean);
 //+/- are deleted from inner nodes that accompany leaves
 l = treeClean.length();
 String treeOut = "";
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
 {
  thisChar = "" + treeClean.charAt(i);
  if (leaves.contains(thisChar))
  leaf  = true;
  if ((leaf) &  (plusMinus.contains(thisChar)));
else treeOut = treeOut + thisChar;
  if (internalNodes.contains(thisChar))
  leaf  = false;
```

```
}

 if (print2)
 System.out.println("Tree at output of cleanTree = \n"
+ treeOut);
return treeOut;

}




//The path from each leaf to node a is found.
//The input contains branch's indicator, say:
//((++Aa,+-Ba),(-+Cb,((--++Dd,--+-Fd)--c,---Ec)-b)+a)
//Output for leaf A: Aa+.
//Output for leaf C: Cba-
//Output for leaf D: Ddcba-
private static void findPaths(String tree)
{
 if (print2)
 System.out.println("Input tree at findPaths = \n" + tree);
 //length of String   tree
 int l = tree.length();
 String plusMinus = "+-";
 String thisChar;
 String nextChar;
 String path = "";
 //Branch indicative with respect to root, + else -
 String branch;
 String address ;
 //The number of links of a path to root
 int level = 0;
 String treeReversed = reverser(tree);
 if (print2)
 System.out.println("Reversed tree = " + treeReversed);

//Each leaf has a path to a
for(int k= 0; k < numberOfLeaves; k++)
{
```

```
 String leaf = "" + leaves.charAt(k);
 //Candidate nodes to be part of the path to node a
 //must have  sign +/- as prefix.
 //Or as suffix in the reversed tree.
 int position = treeReversed.indexOf(leaf);
 if (position > -1) // if leaf is in tree
 {
  if (print2)
  System.out.println("\nLeaf = " + leaf );
  address = "";
  boolean go = true;
  //branchIndicator and level are found
  for(int i = position+1 ; ((i < l) & (go)); i++ )
  {
   thisChar = "" + treeReversed.charAt(i);
   if   (plusMinus.contains(thisChar)  )
address = thisChar + address;
   else go = false;
  }
  level = address.length();
  if (print2)
  System.out.println("LeafAddressLevel  = " + leaf
                     + address + level);
  branch = "" + address.charAt(0);


  //Path from leaf to node a is found in (not reversed) tree
  position = tree.indexOf(leaf);
  thisChar = "" + tree.charAt(position);
  nextChar = "" + tree.charAt(position + 1);
  //Output
  path = "" + leaf + nextChar;
  go = true;
  if  (nextChar.contentEquals("a") ) //leaf is neighbor of a
   {
  path = path + branch;
  if (print)
  System.out.println("Path short = " + path);
   }
  else  //level > 1
```

```
  {

  //Tree is scanned char by char to get path
  String iChar = "";
  for(int i = position + 2; ((i < l) & (go)); i++ )
  {
   iChar = "" + tree.charAt(i);


   int newLevel = 0;
   if (internalNodes.contains(iChar))//InternalNodes
{
     if   (iChar.contentEquals("a")  )
      {
   path  = path +  iChar;
   go = false; //end
      }
     else //Internal node different than a
     {
      if (print)
      System.out.println("Internal node not a) = " + iChar );
      int rP = l -i-1; //index of iChar in treeReversed
      String  nextNewChar = "" + treeReversed.charAt(rP+1);

      if ( plusMinus.contains(nextNewChar )) //candidate node
       {
       if (print)
     System.out.println("next candidate = " + iChar);
        String addressNew = "";

        //level of candidate node iChar  is inquired
        String thatChar = "";
        boolean go2 = true;
        for(int j = rP+1 ; ((j < l) & (go2)); j++ )
         {
       thatChar = "" + treeReversed.charAt(j);
       if   (plusMinus.contains(thatChar)  )
          addressNew = thatChar + addressNew;
       else go2 = false;
         }
```

```
        if (print)
        System.out.println("AddressNew " +
                                    iChar + addressNew);
       newLevel = addressNew.length();
       //A path links a node with another in a lower level
       if  ( (newLevel < level) &
        (!(nextChar.contentEquals(iChar))))
        {
       path = path + iChar;
       level = newLevel;
       if (print)
       System.out.println("Path newLevel= " + path);
        }
      } //End of if candidate node
     }//End of else Internal node different than a
}//End of if InternalNodes
   }//End of Tree is scanned char by char

  path = path + branch;

 }//End of  else level > 1
 }//End of if leaf is in tree
 if (print2)
 System.out.println("Path completed  = " + path);
 Paths[k] = path;
}//End of each leaf
}



//Paths from leaves to node a are found
//Reverse paths are also calculated.
//Input tree in quasibinary notation
//with addresses and lengths
private static void pathFinder(String tree)
{
 //Useless information is deleted
 String tree2 = cleanTree(tree);
 //Find the path to node a from each leaf.
 //Add an indicator of the branch
```

```
 //with respect to the root: up(+) else down(-)
  findPaths(tree2);
}

//Input: a long raw string with path from leaf to node a
//plus path from node a to leaf.
//Editing is done to get true paths.
private static String grammar (String path)
{
 int n = path.length();
 String thisChar = "";
 String nextChar = "";
 String oldPath = path;
 String plusMinus = "a+-a";
 String minusPlus = "a-+a";
 String two = "";
 int N = path.length();
//Rule 1: two equal symbols produce nothing
 for(int j=1; j < N; j++)
  {
   for(int i=1; i < n-1;)
   {
    thisChar = "" + path.charAt(i);
    nextChar = "" + path.charAt(i+1);
    two = "" + thisChar + nextChar;
    if (thisChar.contentEquals(nextChar))
  {
   i = i + 2;
   path = path.replace(two, "");
  }
    else
  {
   i = i+1;
  }
  n = path.length();
   }
  }

 //Rule 2: a+-a -> a
```

```
 if  (path.contains(plusMinus))
      path = path.replace(plusMinus, "a");
 if   (path.contains(minusPlus))
     path = path.replace(minusPlus, "a");
 if (print2)
 System.out.println("input = " + oldPath +
                    " output = " + path);
 return path;
}

//The distance matrix among leaves is calculated
private static double[][] matrixBuilder()
{
double M[][] =
new double[numberOfLeaves][numberOfLeaves];
 //Path to node a from leaf
 String pathLeafToa = "";
 //Path to leaf from node a
 String pathaToLeaf = "";
 //Path from leaf to lead
 String pathLeafiToLeafj = "";
 String innerNode = "";
 double distance;
 for(int i = 0; i < numberOfLeaves; i++)
  for(int j = 0; j < numberOfLeaves; j++)
  {
   pathLeafToa = Paths[i];
   pathaToLeaf = reverser( Paths[j] );
   //Raw paths
   pathLeafiToLeafj = pathLeafToa + pathaToLeaf;
   //An editing grammar to get true paths
   pathLeafiToLeafj = grammar( pathLeafiToLeafj );
   if (i == j) M[i][j] = 0;
   else
    {
 String leaf = "" + pathLeafiToLeafj.charAt(0);
 int p = leaves.indexOf(leaf);
 distance = lengthsLeaves[p];
 int n = pathLeafiToLeafj.length();
 for(int k = 1; k < n-1; k++)
```

```
 {
  innerNode =  "" + pathLeafiToLeafj.charAt(k);
  p = internalNodes.indexOf(innerNode);
  distance = distance + lengthsInnerLinks[p];
 }
 leaf =  "" + pathLeafiToLeafj.charAt(n-1);
 p = leaves.indexOf(leaf);
 distance = distance + lengthsLeaves[p];
  M[i][j] = distance;
    }
  }
 return  M;
}



//Translation from Newick's to quasi-binary notation:
//addresses are attached to tree in Newick's notation
//Inner nodes are conserved. Lengths are conserved.
private static String toQuasiBinaryInner(String tree)
{
 if (print2)
  System.out.println("Input tree at toQuasiBinaryInner " +
                          "= \n" + tree);
 //Output
 String treeToBi = "";
 String numeric = ".:0123456789";
 //The binary address of leaf
 String address = "";
 //The name or content of the tree
 String content = "";
 String thisChar = "";
 String nextChar = "";
 String previousChar = "";
 //length of String   tree
 int l = tree.length();
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + tree.charAt(i);
```

```
   if ( i < l-1) nextChar = "" + tree.charAt(i+1);
    else nextChar = "end";
   if ( i > 0) previousChar = "" + tree.charAt(i-1);
    else previousChar = "start";


   //This char = (
   if (thisChar.contentEquals("("))
    {
     content = content + thisChar;
     address = address + '+';
     if (print)
 System.out.println("Address = " + address);
    }


 //If this char is a leaf
 if (leaves.contains(thisChar))
  {
   // System.out.println("leaf");
   treeToBi =  treeToBi + content + address + thisChar;
   content = "";
  }

 if (numeric.contains(thisChar))
 treeToBi =  treeToBi + thisChar;

 //If this char is an inner node
 if (internalNodes.contains(thisChar))
  {
   int m = address.length();
//delete last bit from the right
   if (m>0) address = address.substring(0, m-1) ;
   if (address.contentEquals("")) address = "+";
   treeToBi =  treeToBi + address + thisChar;

  }


 //This char = , and goes after an inner node
```

```
 if  (thisChar.contentEquals(","))
  {
   //System.out.println
 //("comma : "  + previousChar + thisChar );
   treeToBi =  treeToBi + thisChar ;
   if (print)
 System.out.println("Address at ,1   = " + address);

   if    (numeric.contains(previousChar))
  address = address + '-';//Add a - to the right
   if (print)
 System.out.println("Address at ,2   = " + address);
  }


 if (thisChar.contentEquals(")"))
  {
treeToBi =  treeToBi + thisChar ;
    //Diphthonge   ),
if   (nextChar.contentEquals(","))
{
     int m = address.length();
 //delete last bit from the right
 if (m >0)   address = address.substring(0, m-1) + "-";
   }
  }

}//End of for(int i...
return treeToBi;
}


//bTree  (in links with lengths notation)
//is used to generate Record[]
//with addresses and contents.
//Names of inner nodes are visible
public static void makeInputInner(String bTree)
{
 int counter = 0;
 //length of String   tree
```

```
 int l = bTree.length();
 String thisChar = "";
 String address = "";
 String binNumbers = "+-";
 String numeric = ".:0123456789";
 //String tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + bTree.charAt(i);
if (binNumbers.contains(thisChar))
   address = address + thisChar;
if ( (leaves.contains(thisChar)) ||
(internalNodes.contains(thisChar)))
{
 Record[counter] = address;
     counter++;
 Record[counter] = thisChar;

 counter++;
 address = "";
    }

if (numeric.contains(thisChar))
 {
Record[counter-1] = Record[counter-1] + thisChar;
 }

  }


 if (print)
  for(int i = 0; i < counter; i = i+2 )
  System.out.println(Record[i] +
 " bbbb " + Record[i+1]);

 lastUsedPosition = counter;
 if (print)
System.out.println("lastUsedPosition ==== "
      + lastUsedPosition);
}//End of method
```

```
//Prints  a matrix and its title
private static void printMatrix(double Matrix[][],
                        String title)
{
 System.out.println( "\n" + title );
 int m = Matrix.length;
 System.out.print(  "\t");
 for ( int j = 0; j < m; j++ )
  System.out.print( leaves.charAt(j)+ "\t");
 System.out.println();
 for ( int i = 0; i < m; i++ )
  {
   System.out.print(leaves.charAt(i ) + "\t");
   for ( int j = 0; j < m; j++ )
 System.out.print( Matrix[ i ][j]+ "\t");
   System.out.println();
  }
}


//Tree is translated
//from link notation to binary.
//Output in Record[]: even entries= addresses, odd = names.
//Associated matrix is also calculated
public static double [][] associatedMatrix(String tree)
{
 /*
 System.out.println("Tree  at translate = " + tree);
 //String NwTree = simplify(tree);
 String bTree = toBinary(tree);
 //A vector with addresses and names is composed
 makeInput(bTree);
 */
//String NwTree = simplify(tree);
 String bTree = toQuasiBinaryInner(tree);

 if (print2)
 {
```

```
 System.out.println("Input tree  at translate = "    + tree);
  System.out.println("Selected tree in " +
  "pseudo-Newick's notation = \n" + simplify(tree));
  System.out.println("Tree in quasi-binary notation: \n"
              + bTree);
 }

 //A vector with addresses and other with names are composed
 makeInputInner(bTree);
 //Paths from leaves to node a are found.
 //Reverse paths are also listed.
 pathFinder(bTree);
 //The distance matrix among leaves is calculated
 double M[][] = matrixBuilder();
 return M;
}



//The absolute distance among two matrices is calculated.
//Only triangular part of matrices are compared.
private static double absDistance(int n,
               double[][] M1, double[][] M2)
{
double abs = 0;
for(int i = 0; i < n; i++)
for(int j = i+1; j < n; j++)
 abs = abs + Math.abs(M1[i][j]-M2[i][j]);
return abs;
}


//Input : tree in link notation.
//Output:
//distance matrix associated to the leaves of the tree.
private static double[][] distanceMatrix(String tree)
{
String s = toQuasiBinaryInner(tree) ;
```

```
if (print4)
 System.out.println("\n Tree = " + s);
//Paths from leaves to node a are found.
//Reverse paths are also listed.
pathFinder(s);
//The distance matrix among leaves is calculated
double M[][] = matrixBuilder();
return M;
}
```

```
//=========================================================
//==================Inner service class====================
//=========================================================


class DynamicTree15 extends JPanel
{

 private static final long serialVersionUID = 1L;
```

```
protected DefaultMutableTreeNode rootNode;
protected DefaultTreeModel treeModel;
protected JTree tree;
//Next leaf to be grafted
char nextLeaf = leaves.charAt(3);
//Next internal node to be inserted
char nextIntNode = internalNodes.charAt(1);
//counter of generations
int counterGen = 0;


//Constructor
public DynamicTree15()
{
   super(new GridLayout(1, 0));

   rootNode = new DefaultMutableTreeNode(seed);
   treeModel = new DefaultTreeModel(rootNode);

   tree = new JTree(treeModel);
   tree.setEditable(true);
   tree.getSelectionModel().setSelectionMode(
       TreeSelectionModel.SINGLE_TREE_SELECTION);
   tree.setShowsRootHandles(true);

   JScrollPane scrollPane = new JScrollPane(tree);
   add(scrollPane);
 }


 /** Selected tree is unveiled */
 public String unveilTree()
 {
   DefaultMutableTreeNode selectedNode = null;
   TreePath selectedPath = tree.getSelectionPath();

   if (selectedPath == null) {
    selectedNode = rootNode;
   } else {
```

```
    selectedNode = (DefaultMutableTreeNode)
            (selectedPath.getLastPathComponent());
    }

    String tree =  selectedNode.toString();
    if (print)
    System.out.println("\nSelected tree at unveilTree() = "
                        + tree);
    return tree;
  }


  //Add a node
  public DefaultMutableTreeNode
              addObject(DefaultMutableTreeNode seed,
      Object child, boolean shouldBeVisible) {
    DefaultMutableTreeNode childNode =
               new DefaultMutableTreeNode(child);

    if (seed == null) {
      seed = rootNode;
    }

    // It is key to invoke this on the TreeModel,
    //and NOT DefaultMutableTreeNode
    treeModel.insertNodeInto
             (childNode, seed, seed.getChildCount());

    // Make sure the user can see the lovely new node.
    if (shouldBeVisible) {
      tree.scrollPathToVisible
                   (new TreePath(childNode.getPath()));
    }
    return childNode;
  }
```

```
//Listener
class MyTreeModelListener implements TreeModelListener
{
  public void treeNodesChanged(TreeModelEvent e)
  {
    DefaultMutableTreeNode node;
    node = (DefaultMutableTreeNode)
       (e.getTreePath().getLastPathComponent());

    /*
     * If the event lists children,
     * then the changed node is the child of the
     * node we've already gotten.
     * Otherwise, the changed node and the
     * specified node are the same.
     */

    int index = e.getChildIndices()[0];
    node = (DefaultMutableTreeNode)
               (node.getChildAt(index));

    System.out.println(
      "The user has finished editing the node.");
    System.out.println("New value: " +
                node.getUserObject());
  }

  public void treeNodesInserted(TreeModelEvent e)
  {
  }

  public void treeNodesRemoved(TreeModelEvent e)
  {
  }

  public void treeStructureChanged(TreeModelEvent e)
  {
  }
}
```

```
}//End of DynamicTree15



//==========================================
//==Inner class  for drawing selected tree ==
//==========================================


//This program translates a string that
//denotes a tree in Newick's notation
//to our binary notation
//(assign a zero to the left, a one to the right).

//A generator of random trees in Newick's notation
//is included.

//Resultant trees are drawn.




class DrawBiTree12 extends JPanel

{
 private static final long serialVersionUID = 1L;
 private JPanel drawingPane32;
 JScrollPane scroller;



 //=====================================



 //Vertical  size of the panel
 int verticalSize;
 int horizontalSize;
```

```
//Scale must be a power of 2
 int scale = 1;

 //Symbolic horizontal length of each stroke
 int horizontalLength = 30;

 int verticalLength = 10;



 String  NwTree = "";



 boolean print = false;



 //This is our encoding for a tree in our binary notation:
 //We use a vector of strings:
 //Even entries are the paths in the tree  of the nodes,
 //odd ones are the strings, names or contents.



 //Y-coordinates of points where strings are drawn
 int[] yCoordinate = new int[1000];



 //Set up the instructions.
 JLabel instruct1;
 JLabel instruct2;

public DrawBiTree12()
{
 super(new BorderLayout());
```

```
 JPanel instructionPanel = new JPanel(new GridLayout(0,1));


 //Set up the instructions.
 instruct1 = new JLabel(title1);
 instruct2 = new JLabel(title2);

 instructionPanel.setFocusable(true);
 instructionPanel.add(instruct1);
 instructionPanel.add(instruct2);

 //Set up the drawing area.
 drawingPane32 = new drawingPane32();
 drawingPane32.setBackground(Color.white);

 //Put the drawing area in a scroll pane.
 JScrollPane scroller = new JScrollPane(drawingPane32);
 scroller.setPreferredSize(new Dimension(500,900));
 scroller.revalidate();

 //Lay out this demo.
 add(instructionPanel, BorderLayout.PAGE_END);
 add(scroller, BorderLayout.CENTER);
}


//=========================================
//============= Painting class ============
//=========================================


/** The component inside the scroll pane. */
 class drawingPane32 extends JPanel
{
 private static final long serialVersionUID = 1L;

 //Initialization of coordinates
 int centerX = 1;
 int centerY = 1;
 //Coordinates of the origin
```

```
 int x00;
 int y00;
 //Target coordinates
 int x11 ;
 int y11;
 //Horizontal and basic vertical strokes
 int H;
 int V;
 //Addresses are interpreted as paths
 String path = "";
 char direction;



//Calculate the size of a box to hold the tree
//Scrolling controls are managed
private void sizeOfTree(Graphics2D g2d)
{
 //Calculate the dimension of the strings
 //All strings have equal size.

 //Get   font in use
 Font font = g2d.getFont();
 //get metrics from the graphics
 FontMetrics metrics = g2d.getFontMetrics(font);
 //get the advance of my text in this font
 //and render context
 int lengthText = metrics.stringWidth("Aa") + 2;
 //get the height of a line of text in this
 //font and render context (some padding is included)
 int hgt = metrics.getHeight()  + 8;
 if (print)
  System.out.println("height of letters = " + hgt);

 //Horizontal and basic vertical strokes
 H = scale * horizontalLength;
 V = scale * verticalLength;

 horizontalSize = lengthText +
```

```
              (int) (H *  numberOfLeaves);
 verticalSize =
 (int) (hgt *   numberOfLeaves);


 //area = size of the graphic (width, height)
 Dimension area =
 new Dimension(horizontalSize,12*verticalSize);
 //area is set as the preferred size of the graphic
 drawingPane32.setPreferredSize(area);
 //The graphic demands a space for itself:
 //if the alloted space is not enough,
 //turn on scrolling bars.
 drawingPane32.revalidate();
 verticalLength = 2*verticalSize;
 //Coordinates of the origin
 x00 =    scale*centerX;
 y00 =   scale* centerY + verticalSize  - 600;
 //Target coordinates
 x11 =  x00 + (int) ( scale*horizontalLength);
 y11 =  y00;

}



//Tree is drawn
private void DrawTree3(Graphics2D g2d)
{
//The tree is drawn:
//A path from the origin of the tree to
//each leaf is drawn
 if (print)
 System.out.println("LastUsedPositoon at DrawTree3=  "
                               + lastUsedPosition);
 for(int i = 0; i < lastUsedPosition; i = i+2)
  {
//Addresses will be used as paths to  be drawn
path = Record[i];
if (print)
```

```
System.out.println( i + " Path = " + path);
//Reinitialization
Dimension center = new Dimension(x11,y11);
V = scale * verticalLength;
//Path is interpreted
for(int j = 0; j < path.length(); j++)
 {
  direction = path.charAt(j);
  int x0 =  center.width;
  int y0 =  center.height;
  int x1 =  x0;
  int y1;
  if (direction == '+') //go up
   {
    y1 =  y0 + V;
    center.height = center.height + V;
   }
   else // if direction == '-'  =  go down
   {
     y1 =  y0 - V;
     center.height = center.height - V;
    } //End of if + else
    //Draw vertical stroke
    g2d.drawLine(x0, verticalSize -y0, x1,
     verticalSize- y1);
    //Draw horizontal stroke
    g2d.drawLine(x1, verticalSize - y1,
        x1 + H,  verticalSize - y1);
    //Move center
    center.width = center.width + H ;
    //Vertical stroke is divided in two
    V = V/2;
    }//End of for j
 }  //End of for: numberOfLeaves
}



//Strings are drawn
private void drawStrings(Graphics2D g2d)
```

```
{
 int counter = 0;
 //Seed is not  drawn: we lack one
 //g2d.drawString(seed,x00,verticalSize - y00);
 yCoordinate[counter] = verticalSize - y00;
 //counter++;
 //Strings are drawn:
 for(int i = 0; i < lastUsedPosition; i = i+2)
 {
  //Addresses will be used as paths to the places
  //where strings must be drawn
  path = Record[i];
  String s = Record[i+1];
  //Reinitialization
  Dimension center = new Dimension(x00,y00);
  Dimension target = center;
  V = scale * verticalLength;
  //Path is interpreted
  for(int j = 0; j < path.length(); j++)
{
 direction = path.charAt(j);
 if (direction == '+') //go up
 target.height = target.height + V;
 else // if direction == '-'  =  go down
 target.height = target.height - V;
 target.width = target.width + H ;
 V = V/2;
}//End of for j
//String is drawn
g2d.drawString(s, target.width + 1,
                      verticalSize - target.height-1);
//y-coordinates are recorded
yCoordinate[counter] = verticalSize - target.height-4;
 counter++;
  }
}


//Basic routine of the drawing procedure
private void drawBasic(Graphics2D g2d)
```

```
{
 g2d.setColor(Color.red);
 //Stem is drawn: our trees is artificially rooted.
 g2d.drawLine(x00, verticalSize - y00,
      x11,  verticalSize - y11);
 sizeOfTree(g2d);
 DrawTree3(g2d);
 drawStrings(g2d);
}

//Graphical  master : it is called automatically
//at starting and by repaint()
protected void paintComponent(Graphics g)
{
 super.paintComponent(g);
 Graphics2D g2d = (Graphics2D)g;
 //Font to draw strings
 Font font = new Font("Arial", Font.PLAIN, 12);
 g2d.setFont(font);
 g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
  RenderingHints.VALUE_ANTIALIAS_ON);
 drawBasic(g2d);
}


}//End of double inner class drawingPane32



/**
* Create the GUI and show it.  For thread safety,
* this method should be invoked from the
* event-dispatching thread.
*/
 void createAndShowGUI()
{
 //Create and set up the window.
 JFrame frame =
   new JFrame(titleFrame);
 frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

```
 //Create and set up the content pane.
 //Constructor of the main application is used.
 JComponent newContentPane = new DrawBiTree12();
 //content panes must be opaque
 newContentPane.setOpaque(true);
 frame.setContentPane(newContentPane);

 //Display the window.
 frame.pack();
 frame.setVisible(true);
}

}//End of  inner class DrawBiTree12

}//End of main class K38 EvoFull
```

39 ***Exercise.*** *Run the program and play with the code. Choose a population of 50 individuals. Verify that it can solve a tree with 4 leaves (nLeavesToBeInserted = 1;) and most problems with 5 leaves (nLeavesToBeInserted = 2;) in less than 50000 generations and some instances of 6 leaves in less than 100000 generations. But to solve a problem with 7 or 8 leaves, one needs to wait for an eternity.*

40 ***Exercise.*** *Experiment enough with the previous program to agree else disagree with the conclusions of the Author: The program is correct. In regard with finding the exact solution, the program is pretty useless because of its very slow convergence. Nevertheless, performance is much better when the population size is 50 than with 5. Anyway, the program is good to somehow improve an initial solution proposed by randomness.*

41 ***Exercise.*** *Add to the previous program the capability of reporting the generation at which the target topology is recovered. Decide whether or not its efficiency is sufficient to find the correct topology.* *Answer*

Answer

41, page 143. The main modification that allowed the report of the generation at which the target topology was recovered was the insertion of the following lines of code in the method assessFitnesses(). The Author found that with a population of 50 individuals 1000 generations were enough to recover the target topology in the cases with 7 leaves or less. Problems with 8, 9 or 10 leaves need more than 50000 generations.

```
//Program K41 EvoFull2
...

private static double assessFitnesses()
{
 ...
 //Target topology has been found?
 topo = deleteLengths(bestTree);
 topo = simplify(topo);
 if (targetTopology.contentEquals(topo))
  {
   System.out.println( "\n \nTOPOLOGY HAS BEEN FOUND: \n" +
"at generation " + generation
+ "\nFound topology  = \n" + topo
+ "\nTarget Topology = \n" + targetTopology);
  System.exit(0);
  if (print5) print5 = false;
 }
   ...
}
```

42 ***Challenge.*** *Prove that the previous program is much better than randomness. This means that if randomness is provided with the same amount of memory and time as those given to our program to solve the problem, randomness would end with a worst solution. So, a reason exists for many people to reject randomness as an explanation of complexity of life and our existence and rather resort to evolution to try to appease their minds.*

43 ***Challenge.*** *In Darwinism if something happened at random, it was favored with a probability of the same value as that of those cases that were not chosen. Prove that the previous program is much better than a program that relies on pure Darwinism. This means that if such a simulation were provided with the same amount of memory and time as those given to our Lamarckian program to solve the problem, Darwinian evolution would end with a worst solution: most mutations would produce no viable strings and most recombinants would be inviable. Our program is more successful because it twists randomness: viable changes are more probably and in some cases they turn into a necessity.*

We consider that we are making decisive judgments which are footed on the supposition that our code is correct, i.e., that it does what it promises: to simulate

evolution making viable mutations and recombinations highly probable even a necessity. Thus, it is our responsibility to help ourselves to make sure that the code is correct:

44 ***Exercise.*** *Provide the previous program with the possibility to see a depiction as of the best tree provided by the simulation together with the distance matrices associated as to the best approximation as to the target tree. This facility will ease the task of checking output for correctness. The previous program was developed beginning from program J173 EvoTree4 of Vol X. The desire of the Author was to rely on reuse. The result was a program that has a lot of not used things, say, the machinery to deal with the tree of trees. That not used machinery was deleted. This prompts a comment: One tends to imagine that evolution proceeds by a process that begins with simplicity and adds complexity. Nevertheless, in the software business it is also normal to reuse very complex programs to end with simpler ones. So, evolution also can proceed from complexity to simplicity. The implications of this idea in regard with the origin of life has been beautifully but unrealistically in important points developed by Kauffman in his renown book* `At Home in the Universe`*. Also, a game in which strings are simplified and then enlarged seems to be a real evolutionary phenomenon in the evolution of languages. Strings are simplified when they are frequently used and they may be enlarged to diminish recognizing errors.* *Answer*

Experimentation with the program composed in the previous exercise leads us to appreciate graphics very much as a tool for the overall verification of the correctness of programs. We can add a further detail:

45 ***The following code proposes an approximated solution to the problem of fitting a tree to a distance matrix by use of evolution that includes mutation and recombination of tree topologies. We test the code by beginning with a target tree that must be recovered were the correct solution found. This is possible for trees with 4 and 5 leaves. Otherwise, the program runs during certain number of generations and outputs an approximated solution. Depiction of target tree and of best approximation tree after the termination of the simulation is also provided.***

```
//Program K45 EvoFull4
/*
 * We implement here
 * a full fledged evolutionary process
 * to propose an approximate solution to the problem of
```

```
 * fitting a tree to a distance matrix.
 *
 * The program runs a certain number of generations,
 * reports the tree that best approximates the  solution,
 * draws the  best approximation tree together with
 * the target tree  and halts.
 *
 * We generate a random tree topology
 * and attach random integer lengths to links.
 * In that way we get the target tree, T.
 * Next, we find the associated distance matrix D  to T.
 * Our problem is:
 *
 * Find tree t such that the absolute distance between
 * matrix D and the associated matrix to t is a minimum.
 * Absolute distance between M and N = sum |Mij-Nij|.
 * This distance defines the FITNESS of each tree.
 * It decreases as good performance augments.
 * The best possible fitness is zero.
 *
 * We run the next test:
 * If we begin with a tree T, compute D, and then t,
 * we expect to return with the initial tree: t = T.
 *
 *
 * To associate a distance matrix to a tree,
 * we calculate the distance in within
 * each pair of nodes (leaves + inner nodes).
 * To that aim, we begin with the first node and
 * walk over the tree until the second node
 * and add the lengths of all traversed links.
 * The resulting number is the  distance.
 *
 *
 * Formats = Link and Newick's notations. And variants.
 *
 * An automatic halting option has been added to be applied
 * if perfect matching is found.
 * The program accepts for input
 * a distance matrix with integer values.
```

```
 * A test for being a distance matrix is run.
 *
 * To find the tree that best fit D,
 * we  use evolution as follows:
 *
 * 1) Initialization:
 *  We generate N topologies  on the same
 *  set of leaves. By adding   lengths to links,
 *  every topology gets represented by a tree.
 * The fitness of each tree
 * is calculated and the best is selected.
 *  If the fitness of the best is zero,
 *  the program halts and the tree is reported.
 *
 * 2) Evolution is composed of rounds of
 *     cloning according to fitness,
 *     random mutation and recombination.
 *       After each generation, fitness is checked
 *     for accomplishment.
 *
 *
 * Mutation is Lamarckian:
 * it always produces viable mutants.
 * Topologies as well as links can mutate.
 *
 * A mutation results from deleting
 * a given number of leaves
 * to reinsert them anew in
 * other places chosen at random.
 * This creates a mutant topology.
 * To mutate length of links:
 * Lengths are read from tree and
 * mutated separately.
 * Then, they are reinserted.
 *
 *
 * Recombination is lamarckian: it is authorized only
 * when it produces viable recombinants.
 * All their subtopologies are calculated
 * and packed in arrays.
```

```
 * The sets of leaves and of inner nodes
 * of each subtopology are found.
 * Subtopologies are compared by pairs,
 * one from each topology.
 * If they happen to have the same set of leaves
 * and of inner nodes, recombination is authorized:
 * the  subtrees that correspond with
 * the subtopologies are swapped.
 *
 * If more than one pair of subtopologies authorize
 * recombination, one is chosen at random.
 * A hashtable is used to memorize those pairs that do it.
 * Original topologies serves as subtopologies
 * that authorize recombination but they do not count.
 *
 *
 */



import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;

import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Random;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

import java.awt.*;

import javax.swing.JComponent;


public class EvoFull4 extends JPanel
```

```
{

 private static final long serialVersionUID = 1L;

 //The program accepts a distance matrix as input
 //else
 //it generates a random tree, calculates the distance
 //matrix associated to leaves and posits it
 //as target matrix.
 private static boolean inputMatrix = false;



 //Input distance matrix:
 //Example 5.4, pag 99, Isaev (2001)
 //Introduction to Mathematical Methods in Bioinformatics
 //Springer
 private static double targetMatrix[][]
 = {{0,8,3,14,10,12},
   {8,0,9,10,6,8},
   {3,9,0,15,11,13},
   {14,10,15,0,10,8},
   {10,6,11,10,0,8},
   {12,8,13,8,8,0}};

 /*
 private static double targetMatrix5[][]
 = {{0,8,3,14,10},
   {8,0,9,10,6},
   {3,9,0,15,11},
   {14,10,15,0,10},
   {10,6,11,10,0}};

 private static double targetMatrix4[][]
 = {{0,8,3,14},
   {8,0,9,10},
   {3,9,0,15},
   {14,10,15,0}};

   */
```

```
//Number of leaves that must go in all trees > 3
private static int numberOfLeaves;

//Everything is built by recursion over this tree
private static   String   seed =
   "((Aa,Ba),Ca)";

//The best distance found in initialization
private static  int  distance0;


private static  String leaves =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
private static String internalNodes =
"abcdefghijklmnopqrstuvwxyz";

//Next leaf to be grafted
private static  int nextLeafIndex = 2;
//Next internal node to be inserted
private static  int nextIntNodeIndex = 0;


private static int maxNumberOfLeaves = leaves.length();




//Number of leaves to be deleted and reinserted
//to create a mutation
private static int nUndoRedo = 0;


private static boolean print1 = false;


//This is the original topology:
private static  String original = "";
```

```
 //===========Most important parameters ===========

 //Number of leaves to be inserted to seed:
 //to get a tree with 4 leaves, add one leaf to seed
 private static  int nAdditionalLeaves = 1;



 //Number of topologies = number of trees that
 //conform the evolving population
 private static  int N = 50;

 //Max number of generations to run evolution
 static private int numberOfGenerations = 101;
 static private long generation = 0;

 //Mutation rate per site   per generation
 //in the lengths of links
 static private double    m = 0.5;



//Report Best tree every lapse generations
static private int lapse = 100;


 //================================================


 //Data for binary tree
 private static String[] Record =  new String[20*N];


 private static  int indexOfTopologies;
 private static String topologies[] = new String[N];
 private static String trees[] = new String[N];
 private static String newTrees[] = new String[N];
```

```
private static String[] bestTreeOfTopology
                              = new String[N];

//Last filled position  in Record[]
private static int lastUsedPosition;



 //Length of the link that glues a leaf to the tree
private static int lengthsLeaves[]  =
             new int[maxNumberOfLeaves];
//Length of the link that glues
//an inner node to another one
private static int lengthsInnerLinks[]  =
            new int[maxNumberOfLeaves];


private static Random r = new Random();
//The control parameter of the random generator for int
private static int maxLength = 10;

//Paths from each leaf to node a
private static String[] Paths =
            new String[maxNumberOfLeaves];

private static String targetTree;
private static String targetTopology;
private static String bestTopology;


//=======Evolution parameters============



static private double distance[] = new double[N];
static private double fitness[] = new double[N];
```

```
//Index of the best
static private int best;
//The best tree for a given topology
static private String bestTree = "";
//Distance from bestTree to target
static private double bestDistance ;
//Boolean flag for termination of job
static private boolean workDone = true;



//=========Debugging help ===============

private static boolean print  = false;

private static boolean print2 = false;
private static boolean print3 = false;
private static boolean print4 = false;
//Turn on print5 firstly
private static boolean print5 = false;
private static boolean print6 = false;
private static boolean print7 = true;


//==============

private static String titleFrame = "";
private static String title1 = "";
private static String title2 = "";
private static String title3 = "";


private static EvoFull4 newContentPane = new EvoFull4();

private static DrawBiTree15 a;


 //=======Constructor of main class
```

```
  public EvoFull4()
  {

  }//End of constructor




public static void main(String[] args)
{
   // Schedule a job for the event-dispatching thread:
   // creating and showing this application's GUI.
   javax.swing.SwingUtilities.invokeLater(new Runnable()
    {
     public void run()
     {
      //If input is a distance matrix:
      if (inputMatrix)
      {
       printMatrix(targetMatrix, "Input Matrix");
       boolean isD = isMatrixDistanceMatrix(targetMatrix);
         if (isD) System.out.println("\nMatrix " +
        "is a distance matrix");
         else
          {
        System.out.println("Matrix " +
             "is NOT a distance matrix");
        System.exit(0);
           }
         numberOfLeaves = targetMatrix.length;
         nAdditionalLeaves = numberOfLeaves-3;
    }
    else//if a random tree is generated -> Matrix
    {
     numberOfLeaves =    3 + nAdditionalLeaves;
     String topology = randomTopology();
     targetTopology = simplify(topology);
     generateLengths();
     targetTree = insertLengths(topology);
```

```
    System.out.println("Target tree: each link " +
    "has its own length = \n" + targetTree);
    targetMatrix  =
        new double[numberOfLeaves][numberOfLeaves];
    targetMatrix = associatedMatrix(targetTree);
    printMatrix(targetMatrix, "Target Matrix");

    double d = absDistance(numberOfLeaves,
                  targetMatrix, targetMatrix);


    System.out.println( "\nDistance from target matrix " +
     "to target = " + d);
    }


   System.out.println("\nDefinition: \nDistance to target = " +
         " Absolute distance between" +
        " target matrix  and \n" +
        "the distance matrix associated to " +
        "the tree with lengths of links that \n" +
        "is the best fitting  of the topology.\n" +
        "Perfect fitting: distance = 0 \n");
    System.out.println("\nNumber of Generations = "
           + numberOfGenerations);
    if (print7)

    System.out.println("Number of leaves = "
                        + numberOfLeaves +
                         "\nMutation rate = "   + m +
                 "\nPopulation size = " + N);

    initialization();
    evolution();

    }
   });
}
```

```
//Test for being a distance matrix.
//MaxLenght = max value among entries is calculated.
private static boolean
    isMatrixDistanceMatrix(double[][] M)
{
 int l = M.length;
 boolean isD = true;
 //Is M(u,u) = 0?
 for(int u = 0; u < l; u++)

if (M[u][u] ==  0) ;
 else
  {
isD = false;
return isD;
  }

 //Is M(u,v) = M(v,u)?
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
if (M[u][v] == M[v][u] ) ;
 else
{
 isD = false;
 return isD;
}

 //Is M(u,v)>0 for u not v?
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
if (M[u][v] >  0) ;
 else
{
 isD = false;
 return isD;
}
```

```
 //Is M(u,v) < M(u,w) + M(w,v)?
 for(int u = 0; u < l; u++)
 for(int v = 0; v < l; v++)
 for(int w = 0; w < l; w++)

if (M[u][v] <= M[u][w]+M[w][v]  ) ;
 else
{
 isD = false;
 return isD;
}
 double m = 0;
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
 if (M[u][v] > m ) m = M[u][v];
 maxLength = (int) m + 1;
 return true;
}



//Simplification from Link notation to Newick's notation:
//(((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String linkToNewick(String newTree)
{
 String tree = "";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
  }
 return tree;
}
```

```
private static String generateTopology()
 {
  //Next leaf to be grafted
  nextLeafIndex = 2;
  //Next internal node to be inserted
  nextIntNodeIndex = 0;
  //A tree topology is generated
  String topology = seed;
  for(int i = 0; i < nAdditionalLeaves; i++ )
 topology = generateChild(topology);
  return topology;
 }


//Random lengths are added to a tree in Link notation
private static void makeLengths()
{
//Lengths with digit values are generated
for(int i = 0; i < numberOfLeaves; i++)
 {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
 }
}

//The population is initialized with nTop topologies.
//Lengths are inserted to get nTop trees
//Fitness is assessed:
//Has the solution been guessed?
private static void initialization()
{
 System.out.println("\n====" +"INITIALIZATION : ");
 if (print5)
  System.out.println("\nSeed = " + seed
 + " -> "    + linkToNewick(seed));
```

```
 if (print7)
  System.out.println( N
   + " random topologies " +
   "and their trees are generated \n");

 //nTop topologies and trees are generated
 for(int i = 0; i < N; i++)
 {
  String topo1 = generateTopology();
  topologies[i] = topo1;
  String tree = insertLengths(topo1);
  makeLengths();
  //Lengths of links are added to tree topology
  tree = insertLengths(topo1);
  trees[i] = tree;
  if (print6)
  {
System.out.println("\n" +  i + "   " + topo1 + " -> "
          + linkToNewick(topo1));
System.out.println("Tree1 = " + tree);
  }

 }
 assessFitnesses();
 scaleFitnesses();
}


//====================================
//============= EVOLUTION =============
//====================================


//Fitness is calculated for each tree
//Fitness  =
//Distance between target matrix and
//the associated distance matrix to tree is calculated.
//The minimal distance is found:
//this renders the fittest tree.
//Has the solution been guessed?
```

```
private static double assessFitnesses()
{
 if (print6)
  System.out.println(  "\n=====Fitnesses are assessed");

 String tree = "";
 //String topo = "";
 double d = 0;
for(int i = 0; i < N; i++)
{

 tree = trees[i];
 decoderLengths(tree);
 double M[][] = distanceMatrix(tree);
 //Absolute distance from the associated distance matrix
 //of the tree
 //to target matrix is evaluated
 d = absDistance(numberOfLeaves,targetMatrix, M);
 distance[i] = d;


 if (print6)
  {
   System.out.println( "\nTree = \n" + trees[i]);
   System.out.println( "\nDistance to target = " + d);
  }


if (print5)
 {
  printMatrix(M, "Distance matrix associated to tree");
  printMatrix(targetMatrix, "Target matrix");
}

/*

//Target topology has been found?
topo = deleteLengths(bestTree);
topo = simplify(topo);
if (targetTopology.contentEquals(topo))
```

```
 {
  System.out.println( "\n \nTOPOLOGY HAS BEEN FOUND: \n" +
"at generation " + generation
+ "\nFound topology  = \n" + topo
+ "\nTarget Topology = \n" + targetTopology);
 //System.exit(0);
 if (print5) print5 = false;
}

*/

//End work if solution has been found.
if (d == 0)
{
System.out.println("\n BestTree  = \n" +  bestTree
           + "\nDistance = " + bestDistance
           + "\ntarget tree = \n" + targetTree
           + "\nInitial distance = " + distance0);

workDone = false;
}
}//End of for(int i...



 //The fittest individual minimizes distance function
 bestDistance = distance[0];
 best = 0;
 for(int i = 1; i < N; i++)
 {
  if (distance[i] < bestDistance )
   {
    bestDistance = distance[i];
    best = i;
   }
 }

 //Best tree
 bestTree = trees[best];
```

```
 if (generation == 0 ) distance0 = (int) bestDistance;
 if (print7)
 {
 System.out.println("\n BestTree  = \n" +  bestTree
           + "\nDistance = " + bestDistance
           + "\ntarget tree = \n" + targetTree
           + "\nInitial distance = " + distance0);


 }
 return bestDistance;
}



//Fitness is rescaled to take values
//in within 0(the worst) and 1(the best)
private static void scaleFitnesses()
{
 if (print6)
  System.out.println(  "\n====Fitnesses are rescaled: " +
  "0(the worst) and 1(the best)");
 double maxD = distance[0];
 double minD = distance[0];
 double b = 1 ;
 //Min and max distances are found
 for(int i = 1; i < N; i++)
  {
if ( distance[i] > maxD ) maxD =  distance[i];
if ( distance[i] < minD ) minD =  distance[i];
  }

 if (minD == maxD) //All   equally fit
  {
   for(int i = 0; i < N; i++)
```

```
 fitness[i] = 1;
  }
 else
  {
 //Distance to target determine fitness:
//fitness[maxD] = 0;    fitness[minD] = 1;
 for(int i = 0; i < N; i++)
 {
  b = 1;
  b = b/(minD - maxD);
  double a = -b*maxD;
  fitness[i] = a + b * distance[i];
}
  }
 if (print6)
  for(int i = 0; i < N; i++)
   {
 System.out.println("Individual = "
             + i + " has distance " +
          distance[i] + " and fitness " + fitness[i]);
   }
}

//Probability of cloning is proportional to fitness.
//N new individuals are formed.
private static void selectiveCloning()
{
 if (print6)
  System.out.println(  "\n====Selective cloning");
//The best is cloned
 newTrees[0] = trees[best];
 if (print6)
 System.out.println("Individual = "
                      + best + " with fitness " +
 fitness[best] + "-> new individual " + 0
 + " = "  + newTrees[0]  );
//N-1 children are sampled
for(int i = 1; i < N; i++)
{
 //flag
```

```
 boolean go = true;
 while (go)
  {
 //An individual is chosen at random
 int k = r.nextInt(N);
 //Probability of cloning is proportional to fitness.
 double q = r.nextDouble();
 if ( q < fitness[k]) //make a copy of k
  {
 newTrees[i] = trees[k];
 if (print6)
 System.out.println("Individual = "
                      + k + " with fitness " +
 fitness[k] + "-> new individual " + i
 + " = "  + newTrees[i]  );

 go = false;//end task
  }
  }

}//End of for //N-1 children are sampled
}

//============Mutation============
```

```
//Detect internal nodes, different than leaves
private static boolean charIsInternalNode(char c)
{
 String s = "" + c;
 if (internalNodes.contains(s)) return true;
 else return false;
}
```

```
//Simplification from Link notation to Newick's notation:
```

```
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTopo)
{
 String topo = "";
 //length of string newTopo
 int l = newTopo.length();
 for(int i = 0; i < l; i++ )
  {
   //newTopo is scanned char by char
  char c = newTopo.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    topo = topo + c;
  }
 return topo;
}

//Rule 2 is used to generate a new topology:
//(S)I -> ((S)N,FN)I.
//S = subtopology, I = internal node, F = new leaf,
//N = new internal node.
//Rule 2 grafts new branch FN to link (S)I
//i is the position of char I in string parentTopo
private static String useRule2(String parentTopo, int i,
       char F, char N)
{
 String newTopo = "";
 char I = parentTopo.charAt(i);
 if (print1)
  System.out.println("i = " + i + " I = " + I);
 //Subtopology S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = i;
 boolean go = true;
```

```
//All right parentheses to the near left of I are captured
while (go)
 {
  position = position -1;
  String a = "" + parentTopo.charAt(position);

  if (print1)
  System.out.println("a = " + a + " p = " + pr);
  if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
  else go = false;
 }//End of while
if (print1)
System.out.println("counter = " + counter);
//The complete subtopology is captured:
//Parentheses must balanced
while (counter > 0)
 {
  String a = "" + parentTopo.charAt(position);
  if (a.equals(pr))   counter++;
  if (a.equals(pl))  counter--;
  S = a + S;
  if (print1)
   System.out.println("Subtopo = " + S);
  position = position -1;
  }
//Replacement rule 2:
//((S)n,Fn)i -> (S)i
//where   F are leaves, n and i are  inner nodes,
//(S) is a complex structure.
String SnFni  =   S  + I;
String replacement =  "("+ S +   N + ","
         + F + N+ ")" + I;
//Rule 1 is executed
newTopo = parentTopo.replace(SnFni, replacement);
//If I links to a complex structure
return newTopo;
```

```
}

//From parentTopo, a topology in Newick's notation,
//a Child  topology is generated by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTopo)
{

 nextLeafIndex++;
 char nextLeaf= leaves.charAt(nextLeafIndex);
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = internalNodes.charAt(nextIntNodeIndex);


 //link to be replaced
 String link = "";
 //String link is replaced by what:
 String replacement = "";
 //new topology generated by replacement
 String newTopo = "";
 //length of parent topology
 int l = parentTopo.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
  {
   //parentTopo is scanned char by char
   char I = parentTopo.charAt(i);
   //Is I an internal node that denotes a link?
   if (charIsInternalNode(I)) counter++;
 }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
 {
```

```
   //parentTopo is scanned char by char
char I = parentTopo.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = parentTopo.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTopo = useRule2(parentTopo, i,
               nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTopo = parentTopo.replace(link, replacement);
 }
if (print2)
System.out.println(newTopo + " -> "
                      + simplify(newTopo));


 i = l;
   }
  counter++;
}
 }
return newTopo;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply the antirule1:
```

```
//(An,Fn)i  -> Ai
private static String  antiRule1(String topo,
int positionLeaf, int positionNode)
{
 if (print2)
  System.out.println("AntiRule1");
 int beginIndex = positionLeaf - 4;
 int endIndex = positionNode + 3;
 String AnFni = topo.substring(beginIndex, endIndex);
 if (print2)
  System.out.println("Anfni = " + AnFni);
 String Ai = "" + topo.charAt(beginIndex+1)
               + topo.charAt(endIndex-1);
 if (print2)
  System.out.println("Ai = " + Ai);
 topo = topo.replace(AnFni, Ai);
return topo;
}



//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply the antirule2:
//((S)n,Fn)i -> (S)i
//positionNode refers to the last appearing of node n
private static String  antiRule2(String topo,
int positionLeaf, int positionNode)
{
 if (print2)
System.out.println("AntiRule2");
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 if (print2)
  System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
            + internalNodes.charAt(nextIntNodeIndex);
 int firstPosition = topo.indexOf(innerNode);
 String newTopo = "";
```

```
char i = topo.charAt(positionNode+2);
if (print2)
 System.out.println("FirstPosition = " +
                    firstPosition + " i = " + i);
//Subtopology S of link (S)I must be read
String S = "";
//Counter of left parenthesis
int counter = 0;
String pl = "" + '(';
String pr = "" + ')';
//Position to be scanned
int position = firstPosition;
boolean go = true;
//All right parentheses to the near left of I are captured
while (go)
 {
  position = position -1;
  String as = "" + topo.charAt(position);

  if (print2)
   System.out.println("a = " + as + " p = " + pr);
  if (as.equals(pr) )
   {
counter++;
S = as + S;
   }
  else go = false;
 }//End of while
if (print2)
System.out.println("counter = " + counter);
//The complete subtopology is captured:
//Parentheses must balanced
while (counter > 0)
 {
  String as = "" + topo.charAt(position);
  if (as.equals(pr))   counter++;
  if (as.equals(pl))  counter--;
  S = as + S;
  if (print2)
   System.out.println("Subtopo = " + S);
```

```
   position = position -1;
  }
 String n = innerNode;
 //Replacement antiRule 2:
 //((S)n,Fn)i -> (S)i
 //where A and F are leaves, n and i are  inner nodes,
 //(S) is a complex structure.
 String SnFni  =  "("+ S  + n + "," + leaf + n + ")" + i ;
 String Si = S + i;
 if (print2)
  System.out.println("\nSnFni = " + SnFni + "\n Si = " + Si);
 //Rule 1 is executed
 newTopo = topo.replace(SnFni, Si);
 return newTopo;
}



//The last inserted  leaf is deleted from topology.
private static String deleteLastLeaf(String topology)
{
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 if (print2)
 System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
           + internalNodes.charAt(nextIntNodeIndex);
 if (print2)
 System.out.println("Accompanying inner  node = "
                  + innerNode);
 //new topology = output
 String newtopology = "";
 //length of  topology as string
 int l = topology.length();
 //Last positions are found
 int positionLeaf = -1;
 int positionNode = -1;
 //topology is scanned char by char
 for(int i = 0; i < l; i++ )
```

```
  {
 String c = "" + topology.charAt(i);
 if (c.equals(leaf)) positionLeaf = i;
 if (c.equals(innerNode)) positionNode = i;
  }
 if (print2)
  {
   System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
   System.out.println("Position of " + innerNode +
         " = " + positionNode);
  }
 int firstPosition = topology.indexOf(innerNode);
 //Check for the char before firstPosition
 char a = topology.charAt(firstPosition - 1);
 //If I links to a complex structure
 if (a == ')')
 newtopology = antiRule2(topology,positionLeaf,positionNode);
 else  //Apply antiRule1 for simple cases
  newtopology = antiRule1(topology,positionLeaf,positionNode);
 nextLeafIndex--;
 nextIntNodeIndex--;
 return newtopology;
}



//=========================

//Lengths are deleted from newTree
private static String deleteLengths(String newTree)
{
 String tree = "";
 String numeric = ".:0123456789";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
   char c = newTree.charAt(i);
```

```
   String s = "" + c;
   //Is c numeric: delete it!
   if (numeric.contains(s)); //add char c to output
  else tree = tree + c;
  }
 return tree;
}




//Mutation:
//some leaves are deleted and then reinserted
//in random order.
//Deleting can happen in either of two ways:
//Reversing the order of synthesis else
//in random order.
public static String mutation(String topology,
                      int nAdditionalLeaves)
{
 //Mutation has two stages:
 //deletion of some leaves that undoes generating work +
 //re-insertion in places chosen at random.

 //Number of leaves to be deleted and then reinserted
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
  else
   nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;


 if (print2)
  System.out.println("Number of leaves to" +
" be deleted and then reinserted = " +
nUndoRedo);

 //Some leaves are deleted:
 //reversed order to that of synthesis.
 if (print1)
  System.out.println("Some leaves are deleted");
```

```
 for(int i = 0; i < nUndoRedo; i++ )
 {
   //Last inserted leaf is deleted.
   topology = deleteLastLeaf(topology);
   if (print1)
    System.out.println(topology);
 }
 //deleted leaves are reinserted
 if (print1)
  System.out.println("Deleted leaves are reinserted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
  topology = generateChild(topology);
  }



 //Output
 String mutant = topology;
 if (print1)
  System.out.println("\nOriginal and mutant topologies = \n"
  + original + "\n" + mutant);
 if (print1)
 if (original.equals(mutant))
  System.out.println("Mutation made nothing");
   else
  System.out.println("\nMutation worked");
 return mutant;
}


public static String mutation(String topology)
{
 //This original topology is next subject to mutation
 original = topology;
 String mutantTopology = topology;
 if (print2)
System.out.println("\nOriginal   topology = \n"
                + original  );
 //Number of leaves to be deleted and then reinserted
```

```
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
else
 nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;
 //nUndoRedo = 13;

 //Mutation has two stages:
 //deletion of some leaves that undoes generating work +
 //re-insertion in places chosen at random.

 if (print2)
  System.out.println("Number of leaves to" +
   " be deleted and then reinserted = " +
   nUndoRedo);
 //Some leaves are deleted:
 if (print2)
 System.out.println("Some leaves are deleted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
//Last inserted leaf is deleted.
mutantTopology = deleteLastLeaf(mutantTopology);
if (print2)
   System.out.println(mutantTopology);
  }
 //Some leaves are reinserted
 if (print2)
 System.out.println("Deleted leaves are reinserted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
   topology = generateChild(topology);
  }
 return mutantTopology;
}



//Random lengths are added to a tree in Link notation
public static String insertLengths(String treeLinkN)
{
 String thisChar = "";
 String newTree = "";
```

```
 //length of String   treeLinkN
 int m = treeLinkN.length();
 //TreeLinkN is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   thisChar = "" + treeLinkN.charAt(i);
   newTree = newTree + thisChar;
   if (print1)
    System.out.println(newTree.length() + " " + newTree);
   //Lengths are appended
  if (leaves.contains(thisChar))
   {
    int k = leaves.indexOf(thisChar);
    double j = lengthsLeaves[k];
    newTree = newTree + ":" + j;
   }
 if (internalNodes.contains(thisChar))
  {
   int k = internalNodes.indexOf(thisChar);
   double j = lengthsInnerLinks[k];
   newTree = newTree + ":" + j;
  }
 }
 return newTree;
}


//Input: string tree with complete information.
//Work: lengthsLeaves[] and lengthsInnerLinks[]
//are read from tree.
private static void decoderLengths(String tree)
{
 if(print2)
 System.out.println(
"Input tree at decoverLengths = \n "   + tree  );
 //Delete :
 tree = tree.replaceAll(":", "");
 if(print2)
  System.out.println(
"Input tree without : = \n "   + tree  );
```

```
 int n = tree.length();
 String thisChar = "";
 String nextChar = "";
 String numbers = ".0123456789";
 boolean leaf = false;
 boolean node = false;
 String Leaf = "";
 String numberLeaf = "";
 String numberNode = "";
 String Node = "";
 int j ;
 double f;

 //All chars in tree are scanned
 for(int i=0; i < n; i++)
 {
  thisChar = "" + tree.charAt(i);
  if (i < n-1) nextChar  = "" + tree.charAt(i+1);
  //Do we have a leaf?
  if (leaves.contains(thisChar))
   {
 leaf = true;
 Leaf = "" + thisChar;
   }

  //Do we have an internal node?
  if (internalNodes.contains(thisChar))
   {
 node = true;
 Node = "" + thisChar;
   }

  //Do we have a number that belongs in a leaf?
  if ( (leaf) & (numbers.contains(thisChar)))
{
 numberLeaf = numberLeaf + thisChar;
}

  //Do we have a number that belongs in a leaf?
  if ( (node) & (numbers.contains(thisChar)))
```

```
    {
  numberNode = numberNode + thisChar;
 }

  //Must we end accumulation of numbers of a leaf?
  if ( (leaf) & (internalNodes.contains(nextChar)))
 {
  leaf = false;
  j = leaves.indexOf(Leaf);
  if (print1)
  System.out.println("Leaf = " + Leaf +
  " numberLeaf = " + numberLeaf);
  f = new Double(numberLeaf);
  lengthsLeaves[j] = (int) Math.floor(f);
  numberLeaf = "";
 }

  //Must we end accumulation of numbers of a node?
  if ( (node) & (!(numbers.contains(nextChar))))
 {
  node = false;
  j = internalNodes.indexOf(Node);
  if (print1)
  System.out.println("Node = " + Node);
  f = new Double(numberNode);
  lengthsInnerLinks[j] = (int) Math.floor(f);
  numberNode = "";
 }

 }
}

//Random lengths are added to a tree in Link notation
public static void generateLengths()
{
 //Lengths with digit values are generated
 for(int i = 0; i < numberOfLeaves; i++)
  {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
```

```
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
  }
}


//Lengths undergo mutation
private static  void mutateLengths()
{

 //Mutation in links that connect leaves
 for(int k = 0; k < numberOfLeaves; k++)
  {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsLeaves[k] = r.nextInt(maxLength);

  }
 //Mutation in links that connect inner nodes
 for(int k = 0; k < numberOfLeaves-2; k++)
  {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsInnerLinks[k] = r.nextInt(maxLength);

  }



}



//Type of  mutation is decided and executed:
//mutation of lengths of links
//else full mutation (topology and lengths of links).
private static String mutateTree(String tree)
{
 //Topology is read from tree
 String topology = deleteLengths(tree);
 //Lengths are read from tree and recorded in vectors
```

```
 //lengthsLeaves[] and lengthsInnerLinks[]
 decoderLengths(tree);
 mutateLengths();
 String  mutantTree = "";
 //Mutate lengths else topology ( + new lengths)?
 double s = r.nextDouble();
 if ( s < 0)
{
 if (print1)
 System.out.println("\n====Mutation of lengths");

  mutantTree =  insertLengths(topology);

  if (print1)
  {
  System.out.println("Input tree =  " + tree);
  System.out.println("Topology =  " + topology);
  System.out.println("Output tree= " + mutantTree);
  }
 }
else
 {
 if (print1)
  System.out.println("\n====Mutation of topology");


//Mutation of topology
  String mutantTopology = mutation(topology,
                           nAdditionalLeaves);
  //Mutant lengths are reinserted
  mutantTree =  insertLengths(mutantTopology);
 }

 return mutantTree;
}



//Individuals are mutated
private static void mutation()
```

```
{
 if (print6)
  {
System.out.println("\n====Mutation \n");

System.out.println("Trees to be mutated");
  }
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print6)
  System.out.println(trees[i]);
 }
 //The old best, i = 0, is conserved intact
 newTrees[0] = trees[0];
 //Other N-1 individuals suffer mutation
 for(int i = 1; i < N; i++)
  newTrees[i] = mutateTree(trees[i]);


if (print6)
System.out.println("Mutant trees");
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print6)
  System.out.println(trees[i]);
 }
}

//==========Recombination============


 //Counter of subtopologies
 private static  int counterSubtop;
 //Particular number of subtopologies
 private static  int m1 = 0;
```

```
 private static  String[] subtrees1, subtrees2;
 private static  String   subTree1, subTree2,
                      treeRec1, treeRec2;

 //Structure of hashtable : integer + {string1, string2}
 private static Hashtable<Integer, String[]> hashtable =
 new  Hashtable<Integer, String[]>();




//========== Subtrees are listed ============


//Subtrees of  topology are found
private static String[] subtopologies(String topology)
{
 if (print1)
  System.out.println("\nSubtrees are found");
 //Counter of left parenthesis
 int counter = 0;
 //Counter of subtopologies
 counterSubtop = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Length of String   topology
 int m = topology.length();
 String[] subtopologies = new String[m];
 //String topology is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   String Sub = "";
   String a = "" + topology.charAt(i);
   int position = i+1;
   if (a.equals(pl))
   {
    counter++;
    Sub =  Sub + a;
    //The complete subtopology is captured:
```

```
    //Parentheses must balanced
   while (counter > 0)
    {
     String b = "" + topology.charAt(position);
     if (b.equals(pl))   counter++;
     if (b.equals(pr))  counter--;
     Sub =  Sub + b;
     position++;
  }
 if (print2)
   System.out.println(Sub);
 subtopologies[counterSubtop] = Sub;
 counterSubtop++;
 }
}
return subtopologies;
}


//Leaves of subtopologies  are found
private static String[]leaves(int m, String[] subtop)
{
 if (print1)
  System.out.println("Leaves are found");
 String[] myLeaves = new String[m];
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
   else
    {
     myLeaves[i] = "";
     int n = s.length();
     for(int j = 0; j < n; j++)
      {
       String c = "" + subtop[i].charAt(j);
       if (leaves.contains(c))
       myLeaves[i] = myLeaves[i] + c;
     }
   if (print1)
```

```
     System.out.println(subtop[i] + " "
                        + myLeaves[i] );
   }
  }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
  System.out.println("Leaves are listed orderly" );
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
    else
     {
      s =   myLeaves[i];
      myLeaves[i] = "";
      int n = leaves.length();
   for(int j = 0; j < n; j++)
    {
     String c = "" + leaves.charAt(j);
     if (s.contains(c))
     myLeaves[i] = myLeaves[i] + c;
    }
   if (print1)
   System.out.println(subtop[i] + " " + myLeaves[i] );
  }
 }
 return myLeaves;
}



//Inner nodes of subtopologies  are found
private static String[] inner(int m, String[] subtop)
{
 if (print1)
  System.out.println("\nInternal nodes are found");
 String[] myNodes = new String[m];
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
```

```
   if (s == null);
    else
     {
      myNodes[i] = "";
      int n = s.length();
      for(int j = 0; j < n; j++)
       {
        String c = "" + subtop[i].charAt(j);
        if (internalNodes.contains(c))
        myNodes[i] = myNodes[i] + c;
       }
    if (print1)
System.out.println(subtop[i] + " " + myNodes[i] );
  }
 }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
 System.out.println("Inner nodes are listed orderly" );
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
    else
     {
      s =   myNodes[i];
      myNodes[i] = "";
      int n = internalNodes.length();
      for(int j = 0; j < n; j++)
       {
        String c = "" + internalNodes.charAt(j);
        if (s.contains(c))
      myNodes[i] = myNodes[i] + c;
       }
  if (print1)
   System.out.println(subtop[i] + " " + myNodes[i] );
 }
}
return myNodes;
}
```

```
//Subtopologies are compared to decide whether
//or not they have the same set of leaves
//and the same set of inner nodes.
//All pairs that authorize recombination
// are kept in a hastable (=  a table for pairing)
private static boolean comparison(String Leaves1[],
                      String Leaves2[],
                      String Inner1[],
                      String Inner2[])
{
 if (print6)
 System.out.println("\nComparison of leaves and nodes");
 Boolean recAllowed = false;
 int m = Leaves1.length;
 int n = Leaves2.length;
 Integer counter = 0;
 for(int i = 1; i < m; i++)
  {
   for(int j = 1; j < n; j++)
    {
     if ( Leaves1[i].equals(Leaves2[j]))
      {
       subTree1 = subtrees1[i];
       subTree2 = subtrees2[j];
       recAllowed = true;
       if (print6)
        System.out.println(
          "Sets of leaves are EQUAL " + i +
        "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
        String val[] = {subTree1, subTree2};

        //Recombinant subtrees are kept in a hashtable
        hashtable.put(counter,val);
        counter++;
        if ( Inner1[i].equals(Inner2[j]))
         {
      if (print6)
      System.out.println(
        "Sets of inner nodes are EQUAL " + i +
```

```
          "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
         }
        else
         {
       if (print6)
   System.out.println(
       "Sets of inner nodes are DIFFERENT " + i +
       "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
          }
      }
    else
  if (print6)
   System.out.println(
       "Sets of leaves are DIFFERENT " + i +
        "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
  }
 }
 return recAllowed;
}


//A pair of subtrees that authorize recombination
//is chosen at random from the hashtable and
//recombination is executed.
private static  String recombinationTrees(String tree1,
                                          String tree2)
{
 //Possible pairs of recombinant subtrees are recovered
 //from the hashtable and put into an enumeration<>
 Enumeration<Integer> enumKey = hashtable.keys();
 int counter = 0;
 while(enumKey.hasMoreElements())
  {
   Integer key = enumKey.nextElement();
   String[] val = hashtable.get(key);
   if (print3)
    System.out.println("Hasthtable " + key    + "\n"
 + val[0]  + "\n"+  val[1]);
   counter++;
   }
```

```
 //A pair of recombinant subtrees is chosen at random
 Integer election = r.nextInt(counter);
 if (print6)
  System.out.println("Selected pair = " + election);
 String[] val = hashtable.get(election);
 subTree1 = val[0];
 subTree2 = val[1];
 if (print6)
   System.out.println("Recombinant subtrees \n" +
      subTree1 + "\n" +  subTree2);

 //Recombination is executed
 treeRec1 = tree1.replace(subTree1, subTree2);
 treeRec2 = tree2.replace(subTree2, subTree1);
 if (print6)
  System.out.println("\nRecombination is allowed:"
 + "\nOriginal trees \n"
 + tree1 + "\n" + tree2
 + "\nSwapping subtrees \n"
 +  subTree1 + "\n" + subTree2
 + "\nRecombinant trees \n"
 + treeRec1 + "\n" + treeRec2 );
 return treeRec1;
}


//Recombination of tree1 and tree2 is tried out
public static String recombinationMaster(String tree1,
                                String tree2)
{
 //First topology
 String topo1 = deleteLengths(tree1);
 if (print6)
  {
   System.out.println(topo1 + " -> "
       + linkToNewick(topo1));
   System.out.println("\n Tree1 = " + tree1 +
 "\n Topo1 = " + topo1);
  }
```

```
//Length of String   topology
m1 = topo1.length();

subtrees1 = new String[m1];
subtrees1 = subtopologies(topo1);
m1 = counterSubtop;
//Subtopologies are found
String[] subtopologies1 = new String[m1];
for(int i = 0; i < m1; i++)
 subtopologies1[i] =  simplify(subtrees1[i]);
if (print6)
System.out.println();
String[] Leaves1 = new String[m1];
Leaves1 = leaves(m1,subtopologies1);
String[] Inner1 = new String[m1];
Inner1 = inner(m1,subtrees1);
//Chars of Leaves are rewritten in alphabet order
if (print5)
 System.out.println("\nSubtrees, subtopologies, Leaves, " +
            "Inner nodes" );
for(int i = 0; i < m1; i++)
 {
  if (subtopologies1[i] == null);
  else
   {
if (print5)
       System.out.println(i
  + "   " + subtrees1[i]
  + "   " + subtopologies1[i]
  + "   " +  Leaves1[i] + "   " + Inner1[i]);
   }
 }

if (print6)
 System.out.println();

//Second topology
String topo2 = deleteLengths(tree2);
```

```
if (print6)
 {
  System.out.println(topo1 + " -> "
  + linkToNewick(topo2));

System.out.println("\n Tree2 = " + tree2 +
"\n Topo2 = " + topo2);
 }
//Length of String   topology
m1 = topo2.length();

subtrees2 = new String[m1];
subtrees2 = subtopologies(topo2);
m1 = counterSubtop;
//Subtopologies are found
String[] subtopologies2 = new String[m1];
for(int i = 0; i < m1; i++)
 subtopologies2[i] =  simplify(subtrees2[i]);


if (print6)
 System.out.println();
String[] Leaves2 = new String[m1];
Leaves2 = leaves(m1,subtopologies2);
String[] Inner2 = new String[m1];
Inner2 = inner(m1,subtrees2);
//Chars of Leaves are rewritten in alphabet order
if (print5)
 System.out.println("\nSubtopologies, Leaves, " +
            "Inner nodes" );
for(int i = 0; i < m1; i++)
 {
  if (subtopologies2[i] == null);
  else
   {
if (print5)
     System.out.println(i
  + "   " + subtrees2[i]
  + "   " + subtopologies2[i]
  + "   " +  Leaves2[i] + "   " + Inner2[i]);
```

```
    }
  }
 boolean recAllowed =
   comparison(Leaves1,Leaves2, Inner1,Inner2);
 String recombinant = "";
 if (recAllowed)
   recombinant = recombinationTrees(tree1,tree2);
 else
  {
   recombinant = tree1;

   if (print6)
  System.out.println("Recombination is not allowed");
  }
 return recombinant;
}


//Recombination methods may produce an output
//with various and different assignment of lengths to
//inner node a. One is chosen.
public static void correction()
{
String topo;
 for(int i = 1; i < N; i++)
  {
decoderLengths(newTrees[i]);
topo = deleteLengths(newTrees[i]);
newTrees[i] = insertLengths(topo);
  }
}

//Each one of the N individuals  are tried out
//for recombination: a partner is chosen at random
//and if recombination is allowed, it is executed.
//Otherwise, it is cloned.
public static void recombination()
{
 if (print6)
 System.out.println( "\n====RECOMBINATION \n");
```

```
 //Best individual is excluded from recombination
  newTrees[0] = trees[0];

 for(int i = 1; i < N; i++)
  {
   //A partner for i is chosen
   int j = i;
   String t1 = "";
   String t2 = "";
   while (i == j)
 j =   r.nextInt(N);
   if (print6)
System.out.println( "\n==RECOMBINATION OF INDIVIDUALS "
+ i + " and " + j);
   t1 = trees[i];
   t2 = trees[j];
   newTrees[i] = recombinationMaster(t1,t2);
   correction();
  }
 if (print6)
 System.out.println("Recombination output");
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print6)
  System.out.println(trees[i]);
 }
}



//Evolution of trees involving
//recombination and mutation of topologies,
//in within and between.
private static void evolution()
{
 for(int i = 1;
 ((i < numberOfGenerations) & (workDone)); i++)

 {
```

```
generation = i;
  if ( (i % lapse) == 0)
  {
              print7 = true;
          //  a.setEnabled(false);
  }
  else print7 = false;

  if (print7)
  System.out.println( "\n===GENERATION  = " + i);
  selectiveCloning();
  mutation();
  recombination();
  assessFitnesses();
  scaleFitnesses();

 }



 bestTopology = simplify(deleteLengths(bestTree));

 title1 = "Target tree = "  + targetTree;
 title2 = "Best     tree = "   + bestTree;
 title3 = "Distance to target = " + bestDistance;

 decoderLengths(bestTree);

 //Absolute distance from the associated distance matrix
 //of the tree
 //to target matrix is evaluated
 //d = absDistance(numberOfLeaves,targetMatrix, M);




 double[][] M = associatedMatrix(bestTree);

 a = newContentPane.new DrawBiTree15();
 a.createAndShowGUI();
```

```
 a.setLocation(500,  100);


 if (!(workDone))
  System.out.println( "\n \nSOLUTION HAS BEEN FOUND: \n" +
"at generation " + generation);

 System.out.println("\nSummary at  generation " + generation +
 "\nTarget tree   = \n" + targetTree
   + " \nBest tree = \n" + bestTree
   + " \nDistance to target = " + bestDistance
   + " \nBest distance at initialization = "
   + distance0
   + "\nTarget topology = \n" + targetTopology
   + "\nBest topology = \n" + bestTopology);
 printMatrix(M, "Distance matrix associated to best tree");
 printMatrix(targetMatrix,
      "Distance matrix associated to target tree");

}




//A tree topology is generated at random.
public static String randomTopology()
 {
  if (print2)
  System.out.println("Parent tree = \n" + seed
   + " -> "    + simplify(seed));
  String tree = seed;
  int limit = numberOfLeaves-3; //less than 11.
  for(int i = 0; i < limit; i++ )
  tree = generateChild(tree);
  System.out.println("Target topology = \n" + tree);
  return tree;
 }
```

```
//The best fitting tree for chosen topology is found
public static int findTheBest(String tree)
{
 String selectedTopology = simplify(tree);
 String thisTopology = "";
 String thisTree = "";
 int bestPosition = -1;

for(int i = 0; i < indexOfTopologies; i++)
 {
  thisTree = bestTreeOfTopology[i];
  thisTopology = simplify(thisTree);
  if (selectedTopology.equals(thisTopology))
  {
   bestPosition = i;
   return bestPosition;
  }
 }
return  bestPosition;
}


  /*
  //Lengths with real values are generated
  double lengthsLeaves[]  = new double[l];
  //Last two entries are redundant
  double lengthsInnerLinks[]  = new double[l];
  for(int i = 0; i < numberOfLeaves; i++)
   {
 lengthsLeaves[i] = r.nextDouble();
 lengthsInnerLinks[i] =  r.nextDouble();
   }
  */
```

```
//========= Translation in within formats ==========



//bTree in (modified) Newick's notation
//is used to generate Record[]
//with addresses and contents.
public static void makeInput(String bTree)
{
 if (print)
 System.out.println("bTree at makeInput ////// " + bTree);
 int counter = 0;
 //length of String   tree
 int l = bTree.length();
 String thisChar = "";
 String address = "";
 String binNumbers = "01";
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + bTree.charAt(i);
if (binNumbers.contains(thisChar))
   address = address + thisChar;
if (leaves.contains(thisChar))
{
 Record[counter] = address;
     counter++;
 Record[counter] = thisChar;

counter++;
address = "";
   }
  }

 if (print)
```

```
  for(int i = 0; i < counter; i = i+2 )
  System.out.println(Record[i] +
 " bbbb " + Record[i+1]);

 lastUsedPosition = counter;
 if (print)
System.out.println("lastUsedPosition ==== "
        + lastUsedPosition);
}



//Outputs a word in reversed order.
private static String reverser(String word)
{
 return new StringBuilder(word).reverse().toString();
}


private static String cleanTree(String tree)
{
 if (print2)
 System.out.println("Input tree at cleanTree = \n "
        + tree);
 String treeClean = "";
 String numeric = ".:0123456789";
 String plusMinus = "+-";
 String thisChar = "";
  boolean leaf = false;
 //Numerics are washed
 //length of String   tree
 int l = tree.length();
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + tree.charAt(i);
   if (numeric.contains(thisChar));
    else treeClean = treeClean + thisChar;
  }
 if (print2)
```

```
 System.out.println("Tree without numerics = \n"
+ treeClean);
 //+/- are deleted from inner nodes that accompany leaves
 l = treeClean.length();
 String treeOut = "";
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
 {
  thisChar = "" + treeClean.charAt(i);
  if (leaves.contains(thisChar))
  leaf  = true;
  if ((leaf) &  (plusMinus.contains(thisChar)));
else treeOut = treeOut + thisChar;
  if (internalNodes.contains(thisChar))
  leaf  = false;
}

 if (print2)
 System.out.println("Tree at output of cleanTree = \n"
+ treeOut);
return treeOut;

}



//The path from each leaf to node a is found.
//The input contains branch's indicator, say:
//((++Aa,+-Ba),(-+Cb,((--++Dd,--+-Fd)--c,---Ec)-b)+a)
//Output for leaf A: Aa+.
//Output for leaf C: Cba-
//Output for leaf D: Ddcba-
private static void findPaths(String tree)
{
 if (print2)
 System.out.println("Input tree at findPaths = \n" + tree);
 //length of String   tree
 int l = tree.length();
 String plusMinus = "+-";
```

```
 String thisChar;
 String nextChar;
 String path = "";
 //Branch indicative with respect to root, + else -
 String branch;
 String address ;
 //The number of links of a path to root
 int level = 0;
 String treeReversed = reverser(tree);
 if (print2)
 System.out.println("Reversed tree = " + treeReversed);

//Each leaf has a path to a
for(int k= 0; k < numberOfLeaves; k++)
{
 String leaf = "" + leaves.charAt(k);
 //Candidate nodes to be part of the path to node a
 //must have  sign +/- as prefix.
 //Or as suffix in the reversed tree.
 int position = treeReversed.indexOf(leaf);
 if (position > -1) // if leaf is in tree
 {
  if (print2)
  System.out.println("\nLeaf = " + leaf );
  address = "";
  boolean go = true;
  //branchIndicator and level are found
  for(int i = position+1 ; ((i < l) & (go)); i++ )
  {
   thisChar = "" + treeReversed.charAt(i);
   if   (plusMinus.contains(thisChar)  )
address = thisChar + address;
   else go = false;
  }
  level = address.length();
  if (print2)
  System.out.println("LeafAddressLevel  = " + leaf
                     + address + level);
  branch = "" + address.charAt(0);
```

```
  //Path from leaf to node a is found in (not reversed) tree
  position = tree.indexOf(leaf);
  thisChar = "" + tree.charAt(position);
  nextChar = "" + tree.charAt(position + 1);
  //Output
  path = "" + leaf + nextChar;
  go = true;
  if  (nextChar.contentEquals("a") ) //leaf is neighbor of a
   {
  path = path + branch;
  if (print)
  System.out.println("Path short = " + path);
   }
  else  //level > 1
  {

  //Tree is scanned char by char to get path
  String iChar = "";
  for(int i = position + 2; ((i < l) & (go)); i++ )
  {
   iChar = "" + tree.charAt(i);


   int newLevel = 0;
   if (internalNodes.contains(iChar))//InternalNodes
{
     if   (iChar.contentEquals("a")  )
      {
   path  = path +  iChar;
   go = false; //end
      }
     else //Internal node different than a
     {
      if (print)
      System.out.println("Internal node not a) = " + iChar );
      int rP = l -i-1; //index of iChar in treeReversed
      String  nextNewChar = "" + treeReversed.charAt(rP+1);

      if ( plusMinus.contains(nextNewChar )) //candidate node
```

```
       {
       if (print)
     System.out.println("next candidate = " + iChar);
        String addressNew = "";

        //level of candidate node iChar  is inquired
        String thatChar = "";
        boolean go2 = true;
        for(int j = rP+1 ; ((j < l) & (go2)); j++ )
         {
       thatChar = "" + treeReversed.charAt(j);
       if   (plusMinus.contains(thatChar)  )
          addressNew = thatChar + addressNew;
       else go2 = false;
         }
        if (print)
        System.out.println("AddressNew " +
                                    iChar + addressNew);
       newLevel = addressNew.length();
       //A path links a node with another in a lower level
       if  ( (newLevel < level) &
        (!(nextChar.contentEquals(iChar))))
        {
       path = path + iChar;
       level = newLevel;
       if (print)
       System.out.println("Path newLevel= " + path);
        }
      } //End of if candidate node
     }//End of else Internal node different than a
}//End of if InternalNodes
   }//End of Tree is scanned char by char

  path = path + branch;

 }//End of  else level > 1
 }//End of if leaf is in tree
 if (print2)
 System.out.println("Path completed  = " + path);
 Paths[k] = path;
```

```
}//End of each leaf
}



//Paths from leaves to node a are found
//Reverse paths are also calculated.
//Input tree in quasibinary notation
//with addresses and lengths
private static void pathFinder(String tree)
{
 //Useless information is deleted
 String tree2 = cleanTree(tree);
 //Find the path to node a from each leaf.
 //Add an indicator of the branch
 //with respect to the root: up(+) else down(-)
  findPaths(tree2);
}

//Input: a long raw string with path from leaf to node a
//plus path from node a to leaf.
//Editing is done to get true paths.
private static String grammar (String path)
{
 int n = path.length();
 String thisChar = "";
 String nextChar = "";
 String oldPath = path;
 String plusMinus = "a+-a";
 String minusPlus = "a-+a";
 String two = "";
 int N = path.length();
//Rule 1: two equal symbols produce nothing
 for(int j=1; j < N; j++)
  {
   for(int i=1; i < n-1;)
   {
    thisChar = "" + path.charAt(i);
    nextChar = "" + path.charAt(i+1);
    two = "" + thisChar + nextChar;
```

```
    if (thisChar.contentEquals(nextChar))
  {
   i = i + 2;
   path = path.replace(two, "");
  }
    else
  {
   i = i+1;
  }
  n = path.length();
   }
  }

 //Rule 2: a+-a -> a

 if  (path.contains(plusMinus))
      path = path.replace(plusMinus, "a");
 if   (path.contains(minusPlus))
     path = path.replace(minusPlus, "a");
 if (print2)
 System.out.println("input = " + oldPath +
                    " output = " + path);
 return path;
}

//The distance matrix among leaves is calculated
private static double[][] matrixBuilder()
{
double M[][] =
new double[numberOfLeaves][numberOfLeaves];
 //Path to node a from leaf
 String pathLeafToa = "";
 //Path to leaf from node a
 String pathaToLeaf = "";
 //Path from leaf to lead
 String pathLeafiToLeafj = "";
 String innerNode = "";
 double distance;
 for(int i = 0; i < numberOfLeaves; i++)
  for(int j = 0; j < numberOfLeaves; j++)
```

```
  {
   pathLeafToa = Paths[i];
   pathaToLeaf = reverser( Paths[j] );
   //Raw paths
   pathLeafiToLeafj = pathLeafToa + pathaToLeaf;
   //An editing grammar to get true paths
   pathLeafiToLeafj = grammar( pathLeafiToLeafj );
   if (i == j) M[i][j] = 0;
   else
    {
 String leaf = "" + pathLeafiToLeafj.charAt(0);
 int p = leaves.indexOf(leaf);
 distance = lengthsLeaves[p];
 int n = pathLeafiToLeafj.length();
 for(int k = 1; k < n-1; k++)
 {
  innerNode =  "" + pathLeafiToLeafj.charAt(k);
  p = internalNodes.indexOf(innerNode);
  distance = distance + lengthsInnerLinks[p];
 }
 leaf =  "" + pathLeafiToLeafj.charAt(n-1);
 p = leaves.indexOf(leaf);
 distance = distance + lengthsLeaves[p];
  M[i][j] = distance;
    }
  }
 return  M;
}



//Translation from Newick's to quasi-binary notation:
//addresses are attached to tree in Newick's notation
//Inner nodes are conserved. Lengths are conserved.
private static String toQuasiBinaryInner(String tree)
{
 if (print2)
  System.out.println("Input tree at toQuasiBinaryInner " +
                          "= \n" + tree);
 //Output
```

```
String treeToBi = "";
String numeric = ".:0123456789";
//The binary address of leaf
String address = "";
//The name or content of the tree
String content = "";
String thisChar = "";
String nextChar = "";
String previousChar = "";
//length of String   tree
int l = tree.length();
//Tree is scanned char by char
for(int i = 0; i < l; i++ )
 {
  thisChar = "" + tree.charAt(i);
  if ( i < l-1) nextChar = "" + tree.charAt(i+1);
   else nextChar = "end";
  if ( i > 0) previousChar = "" + tree.charAt(i-1);
   else previousChar = "start";


  //This char = (
  if (thisChar.contentEquals("("))
   {
    content = content + thisChar;
    address = address + '+';
    if (print)
System.out.println("Address = " + address);
   }


//If this char is a leaf
if (leaves.contains(thisChar))
 {
  // System.out.println("leaf");
  treeToBi =  treeToBi + content + address + thisChar;
  content = "";
 }

if (numeric.contains(thisChar))
```

```
 treeToBi =  treeToBi + thisChar;

 //If this char is an inner node
 if (internalNodes.contains(thisChar))
  {
   int m = address.length();
//delete last bit from the right
   if (m>0) address = address.substring(0, m-1) ;
   if (address.contentEquals("")) address = "+";
   treeToBi =  treeToBi + address + thisChar;

  }


 //This char = , and goes after an inner node
 if  (thisChar.contentEquals(","))
  {
   //System.out.println
 //("comma : "  + previousChar + thisChar );
   treeToBi =  treeToBi + thisChar ;
   if (print)
 System.out.println("Address at ,1   = " + address);

   if    (numeric.contains(previousChar))
  address = address + '-';//Add a - to the right
   if (print)
 System.out.println("Address at ,2   = " + address);
  }


 if (thisChar.contentEquals(")"))
  {
treeToBi =  treeToBi + thisChar ;
    //Diphthonge   ),
if   (nextChar.contentEquals(","))
{
     int m = address.length();
 //delete last bit from the right
 if (m >0)   address = address.substring(0, m-1) + "-";
   }
```

```
  }

}//End of for(int i...
return treeToBi;
}


//bTree  (in links with lengths notation)
//is used to generate Record[]
//with addresses and contents.
//Names of inner nodes are visible
public static void makeInputInner(String bTree)
{
 int counter = 0;
 //length of String   tree
 int l = bTree.length();
 String thisChar = "";
 String address = "";
 String binNumbers = "+-";
 String numeric = ".:0123456789";
 //String tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + bTree.charAt(i);
if (binNumbers.contains(thisChar))
   address = address + thisChar;
if ( (leaves.contains(thisChar)) ||
(internalNodes.contains(thisChar)))
{
 Record[counter] = address;
     counter++;
 Record[counter] = thisChar;

 counter++;
 address = "";
    }

if (numeric.contains(thisChar))
 {
Record[counter-1] = Record[counter-1] + thisChar;
```

```
 }

  }


 if (print)
  for(int i = 0; i < counter; i = i+2 )
  System.out.println(Record[i] +
 " bbbb " + Record[i+1]);

 lastUsedPosition = counter;
 if (print)
System.out.println("lastUsedPosition ==== "
      + lastUsedPosition);
}//End of method


//Prints  a matrix and its title
private static void printMatrix(double Matrix[][],
                        String title)
{
 System.out.println( "\n" + title );
 int m = Matrix.length;
 System.out.print(  "\t");
 for ( int j = 0; j < m; j++ )
  System.out.print( leaves.charAt(j)+ "\t");
 System.out.println();
 for ( int i = 0; i < m; i++ )
  {
   System.out.print(leaves.charAt(i ) + "\t");
   for ( int j = 0; j < m; j++ )
 System.out.print( Matrix[ i ][j]+ "\t");
   System.out.println();
  }
}


//Tree is translated
//from link notation to binary.
//Output in Record[]: even entries= addresses, odd = names.
```

```
//Associated matrix is also calculated
public static double [][] associatedMatrix(String tree)
{
 /*
 System.out.println("Tree  at translate = " + tree);
 //String NwTree = simplify(tree);
 String bTree = toBinary(tree);
 //A vector with addresses and names is composed
 makeInput(bTree);
 */
//String NwTree = simplify(tree);
 String bTree = toQuasiBinaryInner(tree);

 if (print2)
 {
 System.out.println("Input tree  at translate = "    + tree);
  System.out.println("Selected tree in " +
  "pseudo-Newick's notation = \n" + simplify(tree));
  System.out.println("Tree in quasi-binary notation: \n"
              + bTree);
 }

 //A vector with addresses and other with names are composed
 makeInputInner(bTree);
 //Paths from leaves to node a are found.
 //Reverse paths are also listed.
 pathFinder(bTree);
 //The distance matrix among leaves is calculated
 double M[][] = matrixBuilder();
 return M;
}




//The absolute distance among two matrices is calculated.
//Only triangular part of matrices are compared.
private static double absDistance(int n,
               double[][] M1, double[][] M2)
{
```

```
 double abs = 0;
 for(int i = 0; i < n; i++)
 for(int j = i+1; j < n; j++)
   abs = abs + Math.abs(M1[i][j]-M2[i][j]);
 return abs;
}


//Input : tree in link notation.
//Output:
//distance matrix associated to the leaves of the tree.
private static double[][] distanceMatrix(String tree)
{
String s = toQuasiBinaryInner(tree) ;
if (print4)
 System.out.println("\n Tree = " + s);
//Paths from leaves to node a are found.
//Reverse paths are also listed.
pathFinder(s);
//The distance matrix among leaves is calculated
double M[][] = matrixBuilder();
return M;
}



//=========================================
//==Inner class  for drawing selected tree ==
//=========================================


//This program translates a string that
//denotes a tree in Newick's notation
//to our binary notation
//(assign a zero to the left, a one to the right).

//A generator of random trees in Newick's notation
//is included.

//Resultant trees are drawn.
```

```
class DrawBiTree15 extends JPanel

{
 private static final long serialVersionUID = 1L;
 private JPanel drawingPane35;
 JScrollPane scroller;



 //=====================================



 //Vertical  size of the panel
 int verticalSize;
 int horizontalSize;
//Scale must be a power of 2
 int scale = 1;

 //Symbolic horizontal length of each stroke
 int horizontalLength = 30;

 int verticalLength = 10;



 String  NwTree = "";



 boolean print = false;
```

```
 //This is our encoding for a tree in our binary notation:
 //We use a vector of strings:
 //Even entries are the paths in the tree  of the nodes,
 //odd ones are the strings, names or contents.



 //Y-coordinates of points where strings are drawn
 int[] yCoordinate = new int[1000];



 //Set up the instructions.
 JLabel instruct1;
 JLabel instruct2;
 JLabel instruct3;

public DrawBiTree15()
{
 super(new BorderLayout());


 JPanel instructionPanel = new JPanel(new GridLayout(0,1));


 //Set up the instructions.
 instruct1 = new JLabel(title1);
 instruct2 = new JLabel(title2);
 instruct3 = new JLabel(title3);

 instructionPanel.setFocusable(true);
 instructionPanel.add(instruct1);
 instructionPanel.add(instruct2);
 instructionPanel.add(instruct3);

 //Set up the drawing area.
 drawingPane35 = new drawingPane35();
 drawingPane35.setBackground(Color.white);
```

```
 //Put the drawing area in a scroll pane.
 JScrollPane scroller = new JScrollPane(drawingPane35);
 scroller.setPreferredSize(new Dimension(500,900));
 scroller.revalidate();

 //Lay out this demo.
 add(instructionPanel, BorderLayout.PAGE_END);
 add(scroller, BorderLayout.CENTER);
}


//========================================
//============= Painting class ===========
//========================================


/** The component inside the scroll pane. */
 class drawingPane35 extends JPanel
{
 private static final long serialVersionUID = 1L;

 //Initialization of coordinates
 int centerX = 1;
 int centerY = 1;
 Dimension center = new Dimension(centerX,centerY);
 //Coordinates of the origin
 int x00;
 int y00;
 //Target coordinates
 int x11 ;
 int y11;
 //Horizontal and basic vertical strokes
 int H;
 int V;
 //Addresses are interpreted as paths
 String path = "";
 char direction;
 int xMax = 0;
 //Repainting cause some coordinates to change:
 //work with the first ones
```

```
 boolean first = true;


//Calculate the size of a box to hold the tree
//Scrolling controls are managed
private void sizeOfTree(Graphics2D g2d)
{
 //Calculate the dimension of the strings
 //All strings have equal size.

 //Get   font in use
 Font font = g2d.getFont();
 //get metrics from the graphics
 FontMetrics metrics = g2d.getFontMetrics(font);
 //get the advance of my text in this font
 //and render context
 int lengthText = metrics.stringWidth("Aa") + 2;
 //get the height of a line of text in this
 //font and render context (some padding is included)
 int hgt = metrics.getHeight()  + 8;
 if (print)
  System.out.println("height of letters = " + hgt);

 //Horizontal and basic vertical strokes
 H = scale * horizontalLength;
 V = scale * verticalLength;

 horizontalSize = lengthText +
              (int) (H *  numberOfLeaves);
 verticalSize =
 (int) (hgt *   numberOfLeaves);


 //area = size of the graphic (width, height)
 Dimension area =
 new Dimension(horizontalSize,13*verticalSize);
 //area is set as the preferred size of the graphic
 drawingPane35.setPreferredSize(area);
 //The graphic demands a space for itself:
 //if the alloted space is not enough,
```

```
 //turn on scrolling bars.
 drawingPane35.revalidate();
 verticalLength = 2*verticalSize;
 //Coordinates of the origin
 x00 =    scale*centerX;
 y00 =   scale* centerY + verticalSize  - 600;
 //Target coordinates
 x11 =  x00 + (int) ( scale*horizontalLength);
 y11 =  y00;

}



//Tree is drawn
private void DrawTree3(Graphics2D g2d, Dimension center)
{
//The tree is drawn:
//A path from the origin of the tree to
//each leaf is drawn
 if (print)
 System.out.println("LastUsedPosition at DrawTree3=  "
                              + lastUsedPosition);
 for(int i = 0; i < lastUsedPosition; i = i+2)
  {
//Addresses will be used as paths to  be drawn
path = Record[i];
if (print)
System.out.println( i + " Path = " + path);
//Reinitialization
Dimension newCenter =
new Dimension(center.width,center.height);
V = scale * verticalLength;
//Path is interpreted
for(int j = 0; j < path.length(); j++)
 {
  direction = path.charAt(j);
  int x0 =  newCenter .width;
  int y0 =  newCenter .height;
  int x1 =  x0;
```

```
  int y1;
  if (direction == '+') //go up
   {
    y1 =  y0 + V;
    newCenter.height = newCenter.height + V;
   }
   else // if direction == '-'  =  go down
   {
     y1 =  y0 - V;
     newCenter.height = newCenter.height - V;
   } //End of if + else
  //Draw vertical stroke
  g2d.drawLine(x0, verticalSize -y0, x1,
     verticalSize- y1);
  //Draw horizontal stroke
  g2d.drawLine(x1, verticalSize - y1,
        x1 + H,  verticalSize - y1);
  //Move center
  newCenter.width = newCenter.width + H ;
  //Vertical stroke is divided in two
  V = V/2;
  //xMax = left extreme of the graphic
  if (first)
     if (xMax < x1 + H) xMax = x1 + H;

 }//End of for j
   }  //End of for: numberOfLeaves

 //System.out.println("xMax = " + xMax);
}



//Strings are drawn
private void drawStrings(Graphics2D g2d, Dimension center)
{
 int counter = 0;
 //Seed is not  drawn: we lack one
 //g2d.drawString(seed,x00,verticalSize - y00);
 yCoordinate[counter] = verticalSize - y00;
```

```
 //counter++;
 //Strings are drawn:
 for(int i = 0; i < lastUsedPosition; i = i+2)
 {
  //Addresses will be used as paths to the places
  //where strings must be drawn
  path = Record[i];
  String s = Record[i+1];
  //Reinitialization
  Dimension target =
  new Dimension(center.width,center.height);
  V = scale * verticalLength;
  //Path is interpreted
  for(int j = 0; j < path.length(); j++)
{
 direction = path.charAt(j);
 if (direction == '+') //go up
 target.height = target.height + V;
 else // if direction == '-'  =  go down
 target.height = target.height - V;
 target.width = target.width + H ;
 V = V/2;
}//End of for j
//String is drawn
g2d.drawString(s, target.width + 1,
                      verticalSize - target.height-1);
//y-coordinates are recorded
yCoordinate[counter] = verticalSize - target.height-4;
 counter++;
  }
}


//Basic routine of the drawing procedure
private void drawBasic(Graphics2D g2d)
{
 g2d.setColor(Color.red);
 /*
 //Stem is drawn: our trees is artificially rooted.
 g2d.drawLine(x00, verticalSize - y00,
```

```
      x11,  verticalSize - y11);
*/
 sizeOfTree(g2d);

 //targetTree is drawn
 decoderLengths(targetTree);

 //Absolute distance from the associated distance matrix
 //of the tree
 //to target matrix is evaluated
 //d = absDistance(numberOfLeaves,targetMatrix, M);



 associatedMatrix(targetTree);

 center = new Dimension(x11,y11);
 DrawTree3(g2d, center);
 center = new Dimension(x00,y00);
 drawStrings(g2d,center);
 first = false;



 //bestTree is drawn


 decoderLengths(targetTree);

 //Absolute distance from the associated distance matrix
 //of the tree
 //to target matrix is evaluated
 //d = absDistance(numberOfLeaves,targetMatrix, M);



 associatedMatrix(bestTree);
```

```
 center = new Dimension(xMax + 30,y11);
 DrawTree3(g2d, center);
 center = new Dimension(
 xMax - (int) ( scale*horizontalLength) + 30,y00);
 drawStrings(g2d,center);

}

//Graphical  master : it is called automatically
//at starting and by repaint()
protected void paintComponent(Graphics g)
{
 super.paintComponent(g);
 Graphics2D g2d = (Graphics2D)g;
 //Font to draw strings
 Font font = new Font("Arial", Font.PLAIN, 12);
 g2d.setFont(font);
 g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
  RenderingHints.VALUE_ANTIALIAS_ON);
  drawBasic(g2d);
}


}//End of double inner class drawingPane35



/**
* Create the GUI and show it.  For thread safety,
* this method should be invoked from the
* event-dispatching thread.
*/
 void createAndShowGUI()
{
 //Create and set up the window.
 JFrame frame =
   new JFrame(titleFrame);
 frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

```
 //Create and set up the content pane.
 //Constructor of the main application is used.
 JComponent newContentPane = new DrawBiTree15();
 //content panes must be opaque
 newContentPane.setOpaque(true);
 frame.setContentPane(newContentPane);

 //Display the window.
 frame.pack();
 frame.setVisible(true);
}

}//End of  inner class DrawBiTree15

}//End of main class K45 EvoFull4
```

46 ***Exercise.*** *Run the program and play with the code.*

47 ***Exercise.*** *Carefully verify the consistence of the output of the previous program. If it is not right, everything said here is garbage. But if it is right, our claims gain confidence. To that aim take into account how to calculate the distance to target, which is the measure of error: perfect fitting implies zero distance to target, and as the distance increases, the error increases. The distance to target of a tree is calculated as follows: each tree generates a distance matrix from leaf to leaf by adding up the lengths of links across the (shortest) path from one leaf to the other. So, we get two matrices:* $T(i,j) =$ *the distance matrix associated to the target tree, and* $B(i,j)$*, the matrix associated to the best tree produced by the simulation. The distance from the best tree to target is the absolute distance between these two matrices =* $\Sigma|T(i,j) - B(i,j)|$*. Your task is to take the drawings of the trees to reproduce* T *and* B *and the distance to target.*

1.5 Conclusion

We are very proud of presenting at last a program that simulates evolution to the full to propose an approximate solution to the problem of fitting a tree to a distance matrix. Implemented evolution classifies as Lamarckian because it twists randomness to accelerate evolution: it artificially fabricates mutation and recombination methods whose products are viable with certainty.

Chapter 2

The Lamarck recursion

`Systematically twisting randomness to our profit`

48 ***Introduction and purpose.*** *We have composed in the previous chapter a program that simulates evolution to the full to propose an approximate solution to the problem of fitting a tree to a distance matrix. The implemented type of evolution is by its very conception Lamarckian: it twists randomness to make sure the viability of mutants and of recombinants. In that way it increases the rate of evolution in comparison with Darwinian simulation of evolution that would be plagued with inviability of the products of evolution (in a Darwinian simulation of evolution mutation and recombination happen directly over encoding strings and they are at random according to a uniform distribution and so the great majority of products would not encode for a tree). The program is nevertheless too inefficient to be of interest. The causes of this are the following: one one hand, we face to a combinatorial explosion and on the other the methods to generate Lamarckian mutations and recombinations together with those to assess fitness are cumbersome and lengthy. Now, we ask: is that all to evolution and concretely to Lamarckian evolution? To give a good battle against this question is our purpose in this chapter.*

2.1 Step one: selfexamination

When one finds that a Lamarckain simulation of evolution is inefficient and wants to improve its performance, the first step is to look in hindsight:

Since Lamarckism is all about twisting of randomness, one must be aware that any twisting favors some specific line of future and disfavors all the others: is the overall output positive or is it a hindrance? In this regard we present and resolve an important complaint against the form as we implemented Lamarckism in the previous program.

49 ***Redundancies vs. vacuums***

The idea of mutation is to explore the world. In our case, the world is the set of topologies. When one faces to a combinatorial explosion, it is mandatory to visit as many states as possible in the least possible time. This implies that all states must have the same probability of being visited. What happens if the visiting distribution is not uniform? A bias is created that favor some problems and disfavor other ones. More to the point, some problems may not be solved in any finite lapse of time. Because we want to create a tool to deal with every kind of problems, no bias is allowed.

Which is our case?

The mutation we have implemented causes faster mutation rates of subtrees with recently added leaves. To see that, let us consider a process of generating a tree: we begin with the tree ((A,B),C) and insert in order leaves D,E, and F. Once the tree is formed, we generate a number between 1 and 3 corresponding to the number of leaves that must be deleted. The corresponding distribution is uniform. But it is wrong to imagine that this means that leaves D,E and F will be deleted with equal probabilities. Instead, if 1 is generated, leaf F will be deleted. If 2 is generated, leaves F and E will be deleted. If 3 is generated, leaves F,E and D will be deleted. So, leaf F will be deleted with certainty but leaf E will be deleted with probability 2/3 and D with probability 1/3. Leaves A, B and C never are removed.

We have proved that our procedure to mutate trees is loaded with a bias due to the order of listing leaves. Therefore, the algorithm preferentially explores the zone around the last added leaf. This is clearly a bonus when one wants to explore states around that leaf. But it is an ache if we want to make wider explorations.

At these considerations a possible solution springs: to list leaves in reverse order or to permute them. While this is indeed a solution to the bias problem another trouble is created dealing with the redundant representation of trees. Redundancy refers to the following fact: the string ((A,B),C) represents the very same tree than (A,(B,C)) and ((B,A),C) and ((B,C),A). And the more leaves are added the more possible representations exist. Our procedure to generate mutation is free from redundancies but creates a bias. Now, we suspect that every remedy against aforementioned bias gets prey of redundancies. Hence it would be good if we have various remedies to test. The next proposal looks interesting.

50 ***Deleting a leaf at random.***

To fix ideas, let us consider the tree

(((Ab,Db)a,(Bc,Ec)a),Ca)

This tree was created by adding to tree ((Aa,Ba), Ca) leaves D and E in that order. We know how to delete leaf E because it is the last we added. The same procedure would work fine to directly delete leaf D. But our rules are not addressed to delete leaf B. The reason is that our rules insert a new link after the place of insertion and their inverse also delete links after that place. But leaf B has no other leaf to its left. This means that we must devise new deleting rules. If we delete leaf B, we must end with (((Ab,Db)a,Ea),Ca).

Our new rules are:
AntiRule 3 to delete leaf F
(FN,AN)I $->$ AI
AntiRule 4 to delete leaf F
(FN,(S)N)I $->$ (S)I.

We can include these new AntiRules for deleting random leaves in the new programs.

51 ***The next code improves mutation: it allows to work with given alphabetic order or with a reversed one, or with a permuted one. It also allows to delete leaves in reversed order to that of insertion at the time of synthesis else to delete leaves in random order. Deleted leaves are then reinserted at random.***

```
//Program K51 TopoMutation3.
//This program generates a random binary tree topology
//and produces a mutation, whose output must be
//a binary tree topology over the same set of leaves.

//At the beginning it is decided with which order to work:
//normal, reversed or permuted.
//Next, a tree is synthesized
//and a mutation is made.
//
//A mutation results from deleting
//a given number of leaves
//to reinsert them anew in
//other places chosen at random.
//Leaves can be deleted in reversed order of insertion
//or at random.
```

```
//Seed tree, with three leaves, is conserved.
//Example in ordinary order ((Aa,Ba),Ca).



import java.util.Random;


public class TopoMutation3
{

 //Link's notation for topologies
 private static   String   seed =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;



 //Number of leaves to be inserted to seed:
 //to get a tree with 4 leaves, add one leaf to seed
 private static  int nAdditionalLeaves = 3;


 //Number of leaves to be deleted and reinserted
 //to create a mutation.
 //mustbe less than nAdditionalLeaves
 private static int nUndoRedo = 0;


 //Leaves that remain in the topology
 private static  String presentLeaves =   "ABC";
 private static String presentNodes =  "a";
```

```
 //Founder leaves ABC or equivalent cannot be removed
 private static String deletableLeaves = "";
 //Deleted leaves are then reinserted
 private static String reinsertingLeaves = "";
 private static String reinsertingNodes = "";

 private static  Random r = new Random();

 private static boolean print1 = false;
 private static boolean antiRule4 = false;

 //This is the original topology:
 private static  String original = "";




//String s ( a char) is deleted from mother
public static String delete(String s, String mother)
{
String output = "";
int l = mother.indexOf(s);
output = mother.substring(0, l)
 + mother.substring(l+1) ;
return output;
}



//String leaves is reversed
public static String reverse(String leaves)
{
String reversed = "";
int l = leaves.length();
for(int i = l-1; i >= 0; i--)
 {
String c = "" + leaves.charAt(i);
reversed = reversed + c;
 }
if (print1)
```

```
System.out.println("Reverse order = " + reversed);
seed =  "((Za,Ya),Xa)";
if (print1)
System.out.println("seed = " + seed);
presentLeaves =   "ZYX";
return reversed;
}


//String leaves is permuted
public static String permuteOrder(String leaves)
{
 if (print1)
System.out.println("original and permuted order = \n"
                         + leaves);
String permuted = "";
int l = leaves.length();
int m = l;
for(int i = 0; i < l; i++)
 {
int d = r.nextInt(m) ;
String c = "" + leaves.charAt(d);
leaves = delete(c,leaves);
m = m-1;
permuted = permuted + c;
 }
if (print1)
System.out.println(permuted);
String first = "" + permuted.charAt(0);
String second = "" + permuted.charAt(1);
String third = "" + permuted.charAt(2);
seed = "((" +first + "a," +second + "a)," +third + "a)";
if (print1)
System.out.println("seed = " + seed);
presentLeaves =   first + second + third;
return permuted;
}


//Detect internal nodes, different than leaves
```

```
private static boolean charIsInternalNode(char c)
{
 String s = "" + c;
 if (internalNodes.contains(s)) return true;
 else return false;
}

//Simplification from Link notation to Newick's notation:
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTree)
{
String tree = "";
//length of string newTree
int l = newTree.length();
for(int i = 0; i < l; i++ )
 {
  //newTree is scanned char by char
 char c = newTree.charAt(i);
 //Is c an internal node to be deleted?
 if (charIsInternalNode(c));
  else //add char c to output
   tree = tree + c;
 }
return tree;
}

//Rule 2 is used to generate a new tree:
//(S)I -> ((S)N,FN)I.
//S = subtree, I = internal node, F = new leaf,
//N = new internal node.
//Rule 2 grafts new branch FN to link (S)I
//i is the position of char I in string seedTree
private static String useRule2(String seedTree, int i,
       char F, char N)
{
String newTree = "";
char I = seedTree.charAt(i);
if (print1)
System.out.println("i = " + i + " I = " + I);
//Subtree S of link (S)I must be read
```

```
String S = "";
//Counter of left seedhesis
int counter = 0;
String pl = "" + '(';
String pr = "" + ')';
//Position to be scanned
int position = i;
boolean go = true;
//All right seedheses to the near left of I are captured
while (go)
 {
  position = position -1;
  String a = "" + seedTree.charAt(position);

  if (print1)
  System.out.println("a = " + a + " p = " + pr);
  if (a.equals(pr) )
  {
counter++;
S = a + S;
  }
  else go = false;
 }//End of while
if (print1)
System.out.println("counter = " + counter);
//The complete subtree is captured:
//seedheses must balanced
while (counter > 0)
 {
  String a = "" + seedTree.charAt(position);
  if (a.equals(pr))   counter++;
  if (a.equals(pl))  counter--;
  S = a + S;
  if (print1)
  System.out.println("Subtree = " + S);
  position = position -1;
  }
//Replacement rule 2:
//((S)n,Fn)i -> (S)i
//where   F are leaves, n and i are  inner nodes,
```

```
//(S) is a complex structure.
String SnFni  =   S  + I;
String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
//Rule 1 is executed
newTree = seedTree.replace(SnFni, replacement);
//If I links to a complex structure
return newTree;
}

//Input: seedTree, a tree in Newick's notation.
//Output : newTree =  seed tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String seedTree)
{

nextLeafIndex++;
char nextLeaf = leaves.charAt(nextLeafIndex);
presentLeaves = presentLeaves + nextLeaf;
//Next internal node to be inserted
nextIntNodeIndex++;
char nextIntNode = internalNodes.charAt(nextIntNodeIndex);
presentNodes = presentNodes + nextIntNode;

//link to be replaced
String link = "";
//String link is replaced by what:
String replacement = "";
//new tree generated by replacement
String newTree = "";
//length of seed tree
int l = seedTree.length();
//counter of links = 0;
int counter = 0;
//Links are counted
for(int i = 0; i < l; i++ )
 {
  //seedTree is scanned char by char
  char I = seedTree.charAt(i);
```

```
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
 }
//A link is chosen
int k = r.nextInt(counter);
//System.out.println("Random number = " + k);
//new branch is grafted at k
counter = 0;
for(int i = 0; i < l; i++ )
 {
   //seedTree is scanned char by char
char I = seedTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = seedTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(seedTree, i,
               nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTree = seedTree.replace(link, replacement);
 }
 if (print1)
System.out.println(newTree + " -> "
                      + simplify(newTree));


 i = l;
```

```
   }
  counter++;
}
 }
return newTree;
}


//Input: seedTree, a tree in Newick's notation.
//Output : newTree =  seed tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String reinsertLeaf(String seedTree)
{
nextLeafIndex++;
char nextLeaf = reinsertingLeaves.charAt(nextLeafIndex);
presentLeaves = presentLeaves + nextLeaf;
//Next internal node to be inserted
nextIntNodeIndex++;
char nextIntNode = reinsertingNodes.charAt(nextIntNodeIndex);
presentNodes = presentNodes + nextIntNode;

//link to be replaced
String link = "";
//String link is replaced by what:
String replacement = "";
//new tree generated by replacement
String newTree = "";
//length of seed tree
int l = seedTree.length();
//counter of links = 0;
int counter = 0;
//Links are counted
for(int i = 0; i < l; i++ )
 {
  //seedTree is scanned char by char
  char I = seedTree.charAt(i);
  //Is I an internal node that denotes a link?
  if (charIsInternalNode(I)) counter++;
 }
```

```
//A link is chosen
int k = r.nextInt(counter);
//System.out.println("Random number = " + k);
//new branch is grafted at k
counter = 0;
for(int i = 0; i < l; i++ )
 {
   //seedTree is scanned char by char
char I = seedTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = seedTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(seedTree, i,
               nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTree = seedTree.replace(link, replacement);
 }
 if (print1)
System.out.println(newTree + " -> "
                      + simplify(newTree));


 i = l;
   }
  counter++;
}
```

```
 }
return newTree;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply the antirule1:
//(An,Fn)i  -> Ai
private static String  antiRule1(String tree,
int positionLeaf, int positionNode)
{
 if (print1)
System.out.println("AntiRule1");
int beginIndex = positionLeaf - 4;
int endIndex = positionNode + 3;
String AnFni = tree.substring(beginIndex, endIndex);
if (print1)
System.out.println("Anfni = " + AnFni);
String Ai = "" + tree.charAt(beginIndex+1)
               + tree.charAt(endIndex-1);
if (print1)
System.out.println("Ai = " + Ai);
tree = tree.replace(AnFni, Ai);
if (print1)
System.out.println("Output tree = \n" + tree);
return tree;
}



//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply the antirule2:
//((S)n,Fn)i -> (S)i
//positionNode refers to the last appearing of node n
private static String  antiRule2(String tree,
int positionLeaf, int positionNode)
{

 if (print1)
```

```
 System.out.println("AntiRule2");
//Leaf to be deleted
String leaf = "" + tree.charAt(positionLeaf);
if (print1)
System.out.println("\nDeletion = " + leaf);
//Accompanying inner node
String innerNode = ""
           + tree.charAt(positionNode);
int firstPosition = tree.indexOf(innerNode);
String newTree = "";
String i = ""; //BUG?????
if (positionNode +2 < tree.length())
        i = ""+ tree.charAt(positionNode+2);
if (print1)
System.out.println("FirstPosition = " +
                    firstPosition + " i = " + i);
//Subtree S of link (S)I must be read
String S = "";
//Counter of left seedhesis
int counter = 0;
String pl = "" + '(';
String pr = "" + ')';
//Position to be scanned
int position = firstPosition;
boolean go = true;
//All right seedheses to the near left of I are captured
while (go)
 {
  position = position -1;
   String as = "" + tree.charAt(position);

  if (print1)
  System.out.println("a = " + as + " p = " + pr);
  if (as.equals(pr) )
  {
counter++;
S = as + S;
  }
  else go = false;
 }//End of while
```

```
if (print1)
System.out.println("counter = " + counter);
//The complete subtree is captured:
//seedheses must balanced
while (counter > 0)
 {
  String as = "" + tree.charAt(position);
  if (as.equals(pr))   counter++;
  if (as.equals(pl))  counter--;
  S = as + S;
  if (print1)
  System.out.println("Subtree = " + S);
  position = position -1;
  }
 String n = innerNode;
//Replacement antiRule 2:
//((S)n,Fn)i -> (S)i
//where A and F are leaves, n and i are  inner nodes,
//(S) is a complex structure.
String SnFni  =  "("+ S  + n + "," + leaf + n + ")" + i ;
String Si = S + i;
if (print1)
 System.out.println("\nSnFni = " + SnFni + "\n Si = " + Si);
//Rule 1 is executed
newTree = tree.replace(SnFni, Si);
if (print1)
System.out.println("Output tree = \n" + newTree);
return newTree;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply   antirule3:
//(Fn,An)i  -> Ai
private static String  antiRule3(String topo,
int positionLeaf, int positionNode)
{
 if (print1)
System.out.println("AntiRule3"
+ "\nInput topology = \n" + topo);
```

```
int beginIndex = positionLeaf - 1;
int endIndex = positionNode + 6;
String FnAni = topo.substring(beginIndex, endIndex);
if (print1)
System.out.println("FnAni = " + FnAni);
String Ai = "" + topo.charAt(beginIndex+4)
         + topo.charAt(endIndex-1);
if (print1)
System.out.println("Ai = " + Ai);
topo = topo.replace(FnAni, Ai);
if (print1)
System.out.println("Output topology = \n" + topo);
return topo;
}



//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply antirule4:
//(Fn,(S)n)i -> (S)i
private static String  antiRule4(String tree,
int positionLeaf, int positionNode)
{

//Leaf to be deleted
String leaf = "" + tree.charAt(positionLeaf);

//Accompanying inner node
String innerNode = ""
            + tree.charAt(positionNode);
if (print1)
System.out.println("\nAntiRule4"
+ "\nLeaf to be deleted = " + leaf
 + " at position " + positionLeaf
+  "\nAccompanying node = " + innerNode
 + " at first position " + positionNode);
//Second position of accompanying node
int positionNode2 = 0;
```

```
int l = tree.length();
boolean go = true;
//tree is scanned char by char
for(int i = positionNode + 1; i < l; i++ )
{
 String c = "" + tree.charAt(i);

 if (go)
 {
  if (c.equals(innerNode))
 {
  positionNode2 = i;

        go = false;
        if (print1)
        System.out.println(
       "Second position of accompanying node = " +
          positionNode2 + " i = " + i);
 }
 }

}


int beginIndex = positionLeaf - 1;
int endIndex = positionNode2+2;
String i = "";
if (endIndex < tree.length())
         i = ""+ tree.charAt(endIndex);

if (print1)
System.out.println("EndIndex = " +
                     endIndex + " i = " + i);

String FnSni  = tree.substring(beginIndex, endIndex) + i;
if (print1)
System.out.println("FnSni = " + FnSni);
String S = tree.substring(beginIndex+4, endIndex-2);
if (print1)
System.out.println("S = " + S);
```

```
String Si = S + i;
if (print1)
 System.out.println("Si = " + Si);

String newTree = tree.replace(FnSni, Si);
if (print1)
 System.out.println("Output tree = \n" + newTree);
//System.exit(0);
return newTree;
}



//The last inserted  leaf is deleted from tree.
private static String deleteLastLeaf(String tree)
{
//Leaf to be deleted
String leaf = "" + leaves.charAt(nextLeafIndex);
if (print1)
System.out.println("\nLeaf to be deleted = " + leaf);
//Accompanying inner node
String innerNode = ""
       + internalNodes.charAt(nextIntNodeIndex);
if (print1)
System.out.println("Accompanying inner  node = "
              + innerNode);
//new tree = output
String newTree = "";
//length of  tree as string
int l = tree.length();
//Last positions are found
int positionLeaf = -1;
int positionNode = -1;
//tree is scanned char by char
for(int i = 0; i < l; i++ )
  {
 String c = "" + tree.charAt(i);
 if (c.equals(leaf)) positionLeaf = i;
 if (c.equals(innerNode)) positionNode = i;
  }
```

```
if (print1)
System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
if (print1)
System.out.println("Position of " + innerNode +
     " = " + positionNode);
int firstPosition = tree.indexOf(innerNode);
//Check for the char before firstPosition
char a = tree.charAt(firstPosition - 1);
//If I links to a complex structure
if (a == ')')
  newTree = antiRule2(tree,positionLeaf,positionNode);
 else  //Apply antiRule1 for simple cases
  newTree = antiRule1(tree,positionLeaf,positionNode);
nextLeafIndex--;
nextIntNodeIndex--;
return newTree;
}



//A random leaf is deleted from tree.
private static String deleteRandomLeaf(String tree)
{
 if (print1)
 {
System.out.println("\nMethod deleteRandomLeaf"
 + "\ntree = " + tree
+ "\ndeletableLeaves = " + deletableLeaves);
 }
int l = deletableLeaves.length();
int position = r.nextInt(l);
//Leaf to be deleted
String leaf = "" + deletableLeaves.charAt(position);
int nodeIndex = tree.indexOf(leaf) + 1;
//Accompanying inner node
String innerNode = ""
       + tree.charAt(nodeIndex);

deletableLeaves = delete(leaf,deletableLeaves);
```

```
presentLeaves = delete(leaf,presentLeaves);
reinsertingLeaves = reinsertingLeaves + leaf;
reinsertingNodes = reinsertingNodes + innerNode;


if (print1)
System.out.println("\nLeaf to be deleted = " + leaf
+ "\nReinserting leaves = " + reinsertingLeaves
    + "\nAccompanying inner  node = "   + innerNode
    + "\nReinserting nodes = " + reinsertingNodes);
//new tree = output
String newTree = "";
//length of  tree as string
l = tree.length();
//Last positions are found
int positionLeaf = tree.indexOf(leaf);
int positionNode = positionLeaf + 1;


if (print1)
System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
if (print1)
System.out.println("Position of " + innerNode +
     " = " + positionNode);
//Which set of deleting rules to apply:
//1 and 2 else 3 and 4?
String previousChar = "" + tree.charAt(positionLeaf-1);
if (previousChar.equals(",")) //Apply antiRules 1 and 2
{
if (print1)
System.out.println("Apply antiRules 1 and 2");
int firstPosition = tree.indexOf(innerNode);
//Check for the char before firstPosition
char a = tree.charAt(firstPosition - 1);
//If I links to a complex structure
if (a == ')')
newTree = antiRule2(tree,positionLeaf,positionNode);
else  //Apply antiRule1 for simple cases
  newTree = antiRule1(tree,positionLeaf,positionNode);
```

```
nextLeafIndex--;
nextIntNodeIndex--;
}
else //Apply antiRules 3 and 4
{
if (print1)
System.out.println("Apply antiRules 3 and 4");
int firstPosition = tree.indexOf(innerNode);
//Check for the char before firstPosition
char a = tree.charAt(firstPosition + 2);
//If I links to a complex structure
if (a == '(')
 newTree = antiRule4(tree,positionLeaf,positionNode);
    else  //Apply antiRule3 for simple cases
newTree = antiRule3(tree,positionLeaf,positionNode);
nextLeafIndex--;
nextIntNodeIndex--;
}

if (print1)
System.out.println("Tree after deletion" + newTree);
return newTree;
}




//Mutation =
//Random deletions are reinserted.
public static String  mutationRandomDeletion(
           String tree, int nAdditionalLeaves)
{
//Some leaves are deleted: random order
 if (print1)
System.out.println("Some leaves are deleted AT RANDOM");
for(int i = 0; i < nUndoRedo; i++ )
 {
  //Last inserted leaf is deleted.
  tree = deleteRandomLeaf(tree);
 }
```

```
//Deleted leaves are reinserted
nextLeafIndex = -1;
nextIntNodeIndex = -1;
if (print1)
System.out.println("Deleted leaves are reinserted");
for(int i = 0; i < nUndoRedo; i++ )
   {

     tree = reinsertLeaf(tree);
   }
return tree;
}



//Mutation:
//some leaves are deleted and then reinserted
//in random order.
//Deleting can happen in either of two ways:
//Reversing the order of synthesis else
//in random order.
public static void mutation(String topology,
                      int nAdditionalLeaves)
{
 //seedal 3 leaves are not removed
 deletableLeaves = presentLeaves.substring(3);
 //Number of leaves to be deleted and then reinserted
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
  else
   nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;
 System.out.println("\nNumber of leaves to delete =  "
                    + nUndoRedo);

 //choose: random order else reverse to that of synthesis
 double pReverseSynth = 0;
 double s = r.nextDouble();
 if ( s < pReverseSynth)//Choose deletion in reversed order
  {
   //Some leaves are deleted:
   //reversed order to that of synthesis.
```

```
   System.out.println("Leaves are deleted " +
                              "in reverse order");
    for(int i = 0; i < nUndoRedo; i++ )
 {
  //Last inserted leaf is deleted.
  topology = deleteLastLeaf(topology);
  System.out.println(topology);
 }
   //deleted leaves are reinserted
   System.out.println("\nDeleted leaves are reinserted");
   for(int i = 0; i < nUndoRedo; i++ )
    {
  topology = generateChild(topology);
    }
  }
 else
  topology = mutationRandomDeletion(topology, nUndoRedo);
 //Output
 String mutant = topology;
 System.out.println("\nOriginal and mutant topologies = \n"
  + original + "\n" + mutant);
 if (original.equals(mutant))
  System.out.println("Mutation made nothing");
   else
  System.out.println("\nMutation worked");
}


public static void main(String[] args)
{
 //Are you going to work with the given order,
 //a reverse one, or with a permutation?
 double pGivenOrder = 0;
 double pReverseOrder = 0;
 double a = pGivenOrder + pReverseOrder;
 //Implicit: probability of permutation =
 //1 - (pGivenOrder+pReverseOrder).
 //Election is made
 double s = r.nextDouble();
 if ( s < pGivenOrder); //Do nothing = order is not changed
```

```
  else if ( s < a )    leaves = reverse(leaves);
 else leaves = permuteOrder(leaves);//permute if  s > a
 //Change of format
 System.out.println("seed topology = \n" + seed
 + " -> "    + simplify(seed));
 //A topology topology is generated

 String topology = seed;

 for(int i = 0; i < nAdditionalLeaves; i++ )
   topology = generateChild(topology);
 //This original topology topology is next subject to mutation
 original = topology;
 mutation(topology, nAdditionalLeaves);
 if (antiRule4)
 System.out.println("ANTIRULE 4");
}
}//End of main class K51 TopoMutation3
```

52 ***Exercise.*** *Run the program and play with the code. Check the program for correctness.*

53 ***Challenge.*** *Simplify the previous program. This is possible because, say, two methods are almost identical:* `generateChild(String seedTree)` *and* `reinsertLeaf(String seedTree)`.

54 ***Exercise.*** *The previous program generates a mutation in within the set of topologies. Convert it into a generator of mutations in within the set of trees.* *Answer*

55 ***Exercise.*** *The previous program generates a mutation by deleting leaves at random to next reinserting them at a random place. Insert this program into a new version of EvoFull. Try your new program out to see whether or not it has improved performance in relation with the program in which deleting of leaves was in reversed order to that of construction of trees. The Author expects that the new program results as useless as the previous version. Nevertheless, an expectant doubt stings his mind.* *Answer*

We have corrected a bias created by our procedure to generate mutant topologies. We feel that our soul rests in peace and moreover we saw an improvement in performance. But the new program is as useless as its predecessor. What can we do?

2.2 Step two: examination of traps

Making our best, our program uses to get stagnated in an approximate solution and cannot leave this trap to produce something better. Because we are dealing with optimization problems, we say that the program has been trapped in a local optimum.

Now, every chaser examines what his trap captured. By the same token, the second and recurrent step of a Lamarckian program for solving optimization problems is to study local optima to catch a regularity in order to look for the ensuing solution. Let us illustrate with our problem how this functions:

56 ***Example.***

The next is a situation of a trapping in a local optimum from generation 10000 to generation 60000, with 7 leaves and population size = 50:

BestTree =
((A:3.0a:5.0,((((B:1.0c:3.0,E:4.0c:3.0)e:6.0,G:6.0e:6.0)d:7.0,F:3.0d:7.0)b:9.0, D:6.0b:9.0)a:5.0),C:8.0a:5.0)

target tree =
((A:3.0a:9.0,((((B:3.0c:1.0,E:6.0c:1.0)e:6.0,G:6.0e:6.0)d:8.0,F:3.0d:8.0)b:8.0, D:5.0b:8.0)a:9.0),C:4.0a:9.0)

One immediately sees that the difference can be somehow explained by coupling errors. To convince oneself of this, let us consider the following pairs of lengths of links, the first pair from the best tree and the second from the target:

B:1.0c:3.0 and B:3.0c:1.0
E:4.0c:3.0 and E:6.0c:1.0
C:8.0a:5.0 and C:4.0a:9.0

In all cases we observe a very simple pattern: a leaf and its closest inner node are accompanied with lengths of links whose errors have opposite sign:

B:-c:+ and B:3.0c:1.0
E:-c:+ and E:6.0c:1.0
C:+a:- and C:4.0a:9.0

What is then the natural solution to get away from trapping?

We fear that there is no natural solution. Instead, many procedures may come to the mind and only experimentation will say how good is each one of them. Our immediate proposal is to generate mutations in which a length is augmented or diminished by one (if only that is possible: our lengths take on digit values, i.e. 0, 1,...,9). These mutations are called tiny. By mere randomness correlated mutations might appear and solve the problem. So, we expect from this procedure to be of great profit.

57 ***Implementation.*** *The most important change to the code to implement tiny mutations, which adds or subtracts one to the length of a link, follows.*

```
 //Program K57 EvoFull6
/*
 * We implement here
 * a full fledged evolutionary process
 * to propose an approximate solution to the problem of
 * fitting a tree to a distance matrix.
 *
 * As a remedy against getting trapped in local optima
 * in program EvoFull5,
 * we implement tiny mutations,
 * which consist in  adding  or subtracting
 * one to the length of a link.
 *
 ... */


//The length of a link can decrease or increase
//by one
private static  void tinyMutations()
{
 //Mutation in length of links that connect leaves
 for(int k = 0; k < numberOfLeaves; k++)
   {
double q = r.nextDouble();
if (q < m) //mutate
   {
   double f = r.nextDouble();
   if (f < m) //mutate
```

```
lengthsLeaves[k] = lengthsLeaves[k] + 1 ;
   else
   lengthsLeaves[k] = lengthsLeaves[k] - 1 ;

   if (lengthsLeaves[k] > 9)  lengthsLeaves[k] = 9;
   if (lengthsLeaves[k] < 0)  lengthsLeaves[k] = 0;
   }
   }
  //Mutation in length of links that connect inner nodes
  for(int k = 0; k < numberOfLeaves-2; k++)
   {
double q = r.nextDouble();
if (q < m) //mutate
 {
  double f = r.nextDouble();
  if (f < m) //mutate
lengthsInnerLinks[k] = lengthsInnerLinks[k] + 1 ;
   else
lengthsInnerLinks[k] = lengthsInnerLinks[k] - 1 ;

  if (lengthsInnerLinks[k] > 9) lengthsInnerLinks[k] = 9;
  if (lengthsInnerLinks[k] < 0) lengthsInnerLinks[k] = 0;
}
  }
}
```

```
//Lengths undergo mutation:
//Choose tiny mutation (+/-1) else
//the old  procedure.
private static  void mutateLengths()
{
 double f = r.nextDouble();
 if ( f < pTinyMutations ) //new procedure
  {
tinyMutations();
  }
  else //Old procedure
```

```
   {

     //Mutation in length of links that connect leaves
     for(int k = 0; k < numberOfLeaves; k++)
      {
   double q = r.nextDouble();
   if (q < m) //mutate
lengthsLeaves[k] = r.nextInt(maxLength);

      }
    //Mutation in length of links that connect inner nodes
    for(int k = 0; k < numberOfLeaves-2; k++)
     {
  double q = r.nextDouble();
  if (q < m) //mutate
lengthsInnerLinks[k] = r.nextInt(maxLength);

     }
   }
}
```

58 ***Exercise.*** *Run the previous program and play with the code. Play enough to verify else reject the appreciation of the Author: the addition of tiny mutations causes an amazing improvement of performance: after 10000 generations under 50 individuals the original program diminishes the initial error 5 times while the program that includes tiny mutations diminishes it 10 times. Anyway, for a problem with 20 leaves the program behaves rather humbly: it can recurrently diminish the error but needs too much time to advance.*

59 ***Exercise.*** *Study the next spontaneous claim: The beautiful result that tiny mutations remedy the problem of the previous program of getting trapped into local optima shows that the gradualism of Darwin is the very reason of our existence.* *Answer*

60 ***Testing pure Darwinism plus tiny mutations*** *We found that tiny mutations accelerate our Lamarckian evolutionary simulation in its quest for the optimum. It seems now that it is a must to investigate what pure Darwinism with tiny mutations can give of itself.*

The next program simulates pure Darwinian evolution to solve the problem of fitting a tree to a distance matrix. The program relies on tiny mutations which must be understood as changes of one char by another. The program includes a method for testing whether or not a string correctly encodes for a tree. A string that fails is understood as an abort. The cost in aborts of each viable string is reported. The program halts when the first viable mutant is found. The main modifications are included in the following exert:

```
//Program K60 EvoFull7
/*
 * We implement here
 * a purely Darwinian mutational process
 * to propose an approximate solution to the problem of
 * fitting a tree to a distance matrix.
 *
 * Mutations are tiny and by this we understand
 * random mutations of a single char.
 *
 * It is found that
 * the usual output of mutation is an inviable string.
 * The program finds the first viable string and halts.
 * The price in inviability is reported.
 *
 * The method that check for the viability of strings+
 * was developed by trial and error.
 * So, it may lack some rules to be discovered.
 *
 */
...


public class EvoFull7 extends JPanel

{

 ...
```

```
//Input = string s.
//Output = an alphabetically ordered list of the chars of s
//according to the given alphabet. Repetitions are forgotten.
//Example : BACDA -> ABCD
private static String reorder(String s, String alphabet)
{
 String myLeaves = "";
 int n = alphabet.length();
 for(int j = 0; j < n; j++)
  {
   String c = "" + alphabet.charAt(j);
   if (s.contains(c))
   myLeaves = myLeaves + c;
  }
 if (print5)
 System.out.println("Reorder " + s + " " + myLeaves);
 return myLeaves;
}


//Does string s encode for a tree?
private static boolean isTree(String s)
{
 if (print5)
System.out.println("String s = \n"+ s);
 boolean isTree = true;
 String numbers = "0123456789";


 String actualLeaves = "";
 String actualNodes = "";
 String leftParenthesis = "(";
 String rightParenthesis = ")";
 int counterParLeft = 0;
 int counterParRight = 0;
 int l = s.length();
 String c = "";
 //String s is scanned char by char
 //to get data
 for(int i = 0; i < l; i++)
```

```
 {
  c = "" + s.charAt(i);
  if (leaves.contains(c))
  actualLeaves = actualLeaves + c;
  if (internalNodes.contains(c))
  actualNodes = actualNodes + c;
  if (leftParenthesis.contains(c) ) counterParLeft++;
  if (rightParenthesis.contains(c) ) counterParRight++;

 }



if (print5)
{
System.out.println("Before reordering");
System.out.println("presentLeaves length = "
+ presentLeaves.length());
 System.out.println("actualLeaves  length = "
+ actualLeaves.length());
 System.out.println("presentNodes  length  = "
+ presentNodes.length());
 System.out.println("actualNodes  length  = "
+ actualNodes.length());
 System.out.println("presentLeaves         = "
                                      + presentLeaves);
 System.out.println("actualLeaves with Rep = "
                                        + actualLeaves);
 System.out.println("presentNodes         = " +
                                  ""+ presentNodes);
 System.out.println("actualNodes with Rep = "
                                           + actualNodes);
 System.out.println("counterParLeft    = "+ counterParLeft);
 System.out.println("counterParRight   = "
                                         + counterParRight);
 System.out.println("nRightParentheses = "
                                  + nRightParentheses);
}
```

```
 actualLeaves = reorder(actualLeaves, leaves);
 actualNodes= reorder(actualNodes, internalNodes);

 if (presentLeaves.length() == actualLeaves.length());
 else isTree = false;
if (print5)
System.out.println("isTree a = "+ isTree);
 if (presentNodes.length() == actualNodes.length());
else isTree = false;
 if (print5)
System.out.println("isTree b = "+ isTree);


 if (print5)
 {
 System.out.println("After reordering");
 System.out.println("presentLeaves length = "
 + presentLeaves.length());
  System.out.println("actualLeaves  length = "
 + actualLeaves.length());
  System.out.println("presentNodes  length  = "
 + presentNodes.length());
  System.out.println("actualNodes  length  = "
 + actualNodes.length());
  System.out.println("presentLeaves = "+ presentLeaves);
  System.out.println("actualLeaves = "+ actualLeaves);
  System.out.println("presentNodes = "+ presentNodes);
  System.out.println("actualNodes = "+ actualNodes);
  System.out.println("counterParLeft = "+ counterParLeft);
  System.out.println("counterParRight = "+ counterParRight);
  System.out.println("nRightParentheses = "
                                    + nRightParentheses);
 }


 //Check for leaves
 if  (actualLeaves.contentEquals(presentLeaves));
  else isTree = false;
```

```
 if (print5)
System.out.println("isTree 1 = "+ isTree);

 //Check for nodes
 if (actualNodes.contains(presentNodes));
  else isTree = false;
 if (print5)
System.out.println("isTree 2 = "+ isTree);


 //Check for  parentheses
 if (counterParRight == nRightParentheses);
   else isTree = false;
 if (print5)
System.out.println("isTree 3 = "+ isTree);

 if (counterParLeft == nRightParentheses);
 else isTree = false;

 if (print5)
System.out.println("isTree 4 = "+ isTree);

 //Check for patterns

//String s is scanned char by char

 String colon = ":";
 String dot = ".";
 String beforeLeaf = "(,";

//A leaf cannot be the initial char
if  (presentLeaves.contains("" + s.charAt(0)))
isTree = false;


 //Test for subpattern A:6.0a
 for(int i = 1; i < l-6; i++)
  {
   if  (presentLeaves.contains("" + s.charAt(i)))
   {
```

```
 if ( beforeLeaf.contains("" + s.charAt(i-1)));
     else isTree = false;
 if ( colon.contentEquals("" + s.charAt(i+1)));
       else isTree = false;

    if (numbers.contains("" + s.charAt(i+2)));
    else isTree = false;

    if ((dot).contentEquals("" + s.charAt(i+3)));
     else isTree = false;
    if (("0").contentEquals("" + s.charAt(i+4)));
     else isTree = false;
    if  (presentNodes.contains("" + s.charAt(i+5)));
     else isTree = false;

    String node = "" + s.charAt(i+5);
    if (print5)
System.out.println("leaf = "+ s.charAt(i)
+ "  " + s.substring(i, i+6)  +
 " node = " + node
 +  "\n isTree 5 = "+ isTree);
  }

 }



 //A node cannot be the initial char
 if  (presentNodes.contains("" + s.charAt(0)))
  isTree = false;

 String beforeNode = "0)";
 //Test for subpattern a:7.0
 for(int i = 1; i < l-5; i++)
  {
   if  (presentNodes.contains("" + s.charAt(i)))
    {
     if ( beforeNode.contains("" + s.charAt(i-1)));
      else isTree = false;
     if ( colon.contentEquals("" + s.charAt(i+1)));
```

```
      else isTree = false;
     if (numbers.contains("" + s.charAt(i+2)));
      else isTree = false;
     if ((dot).contentEquals("" + s.charAt(i+3)));
      else isTree = false;
     if (("0").contentEquals("" + s.charAt(i+4)));
      else isTree = false;
     if ((",)").contains("" + s.charAt(i+5)));
      else isTree = false;
 }
}



 if (print5)
System.out.println("isTree 6 = "+ isTree);

 //if pattern  )v(  appears, a must be equal to ,
 String target  = "),(";
 for(int i = 0; i < l-3; i++)
  {
   String thisChar = "" + s.charAt(i);
   String thatChar = "" + s.charAt(i+2);
   String u = s.substring(i, i+3);
   if (  (thisChar.contentEquals(")") )
   &  (thatChar.contentEquals("(") ) )
    {

 if ( target.contentEquals(u));
  else isTree = false;
 if (print5)
System.out.println("u = "+ u);
    }
  }

 if (print5)
System.out.println("isTree 7 = "+ isTree);
```

```
 //Node a must appear tree times but others only twice
 for(int j = 0; j < presentNodes.length(); j++)
 {
  String node = "" + presentNodes.charAt(j);
  int counter = 0;
  for(int i = 0; i < l; i++)
   if ( node.contentEquals("" + s.charAt(i))) counter++;
  int mustHave;
  if   (node.contentEquals("a") ) mustHave = 3;
   else mustHave = 2;
  if (counter == mustHave );
   else isTree = false;

  if (print5)
System.out.println(" node = " + node
 +  "\n counter = " + counter
                +  "\n mustHave = "+ mustHave);
 }

 if (print5)
System.out.println("isTree 8 = "+ isTree);

 return isTree;
}




//Char newChar  replaces the symbol of tree
//in the position given by place.
private static String replace(String tree,
int place, String newChar)
{
 String mutant = tree.substring(0, place)
 + newChar + tree.substring(place+1);
 return mutant;
}

//Tiny mutation = changes of a single char.
```

```
//Each place of tree can undergo mutation
//Mutants are checked for correct encoding of a tree.
//Inviable strings = aborts. The number of aborts is
//reported before a viable string appears.
private static String mutateTree(String tree)
{
 String  mutantTree = tree;

 //test
 /*
 if (isTree(targetTree))
{
 System.out.println("targetTree is tree");
 System.exit(0);
}
 else
{
 System.out.println("targetTree = \n" + targetTree +
                       " is not tree");
 System.exit(0);
}
  */

 int numberOfAborts = 0;
 boolean inviable = true;
 int l = tree.length();
 String completeAlphabet = presentLeaves + presentNodes +
 "0123456789" + "():,";
 int n = completeAlphabet.length();
 int i = 0;

 //This loop  finds a viable mutant string,
 //one that encodes for a tree.

 while ( inviable)
  {
   mutantTree = tree;
   //Each place in s may undergo mutation
   for(int place = 0; place < l; place++)
{
```

```
   //Replacement char from completeAlphabet
 i = r.nextInt(n);
 String newChar = "" + completeAlphabet.charAt(i);
 double t = r.nextDouble();
 if ( t < m)
  mutantTree = replace(mutantTree, place, newChar);
 }


   if (print6)
 System.out.println( " Original tree = \n" + tree
    + "\nMutantTree = \n" + mutantTree);

//mutantTree = targetTree; //test

 // If a mutant string is found that correctly
 //represents a tree, the program
 if ( (isTree(mutantTree)) & (!(mutantTree.contentEquals(tree)) ) )
  {
inviable = false;
if (print6)
 System.out.println("A VIABLE MUTANT HAS BEEN FOUND \n"
+ " Original tree = \n" + tree
    + "\nMutantTree = \n" + mutantTree);
System.exit(0);

   }
  else
 {
  numberOfAborts++;
 }


  if (print6)
System.out.println("Number of aborts = "
                                    + numberOfAborts);

  }//End of while
 return mutantTree;
}
```

61 ***Exercise.*** *Run the program and play with the code. Play enough to agree else disagree with the appreciation of the Author: Darwinian tiny mutations can produce a change in lengths of links and nothing else, in particular no change of topology.*

62 ***Exercise.*** *Add to the previous program the appropriate methods to enable Darwinian recombination, i.e., recombination of strings without worries about meanings. Use the resulting program to study the proposition: Darwinism is a reliable approach for attacking the program of fitting a tree to a distance matrix.* *Answer*

63 ***Exercise.*** *The previous program simulates almost pure Darwinism to attack the problem of fitting a tree to a distance matrix. Much as early Darwinism, this program assumes that evolution goes on top of a highly developed form of pre-existing life. So, the program initializes with the synthesizes of a population of strings that perfectly encodes for a tree. How this is done was implemented by the Author using a grammar and with no recursion to Darwinian evolution. By contrast, every scientist in modern days believes that the assumption of pre-existent life can be dispensed because protolife can spontaneously appear because of the tendency of matter to get organized when it is allowed to unfold itself under appropriate conditions. This includes in every case the synthesis of original strings with a lot of information. To check the spontaneous appearing of that information by mere randomness, add to previous previous program the possibility to synthesize the string that encodes for a tree by mere randomness. Report the cost of that operation as a function of the number of leaves.* *Answer*

64 ***Challenge.*** *The idea of evolution is that a complex object might be the result of gathering (possibly small) random modifications of a basic object that appeared by mere randomness. Adapt the previous program to test that idea: take as basic object a tree with 3 leaves, say, ((Aa,Ba),Ca) and then add random modifications to next check whether or not the resultant string is a tree with 4 leaves. And so on. One can consider various setting for selection. For instance, one can reward only perfect outputs else one can define tree-ness as a graded quality of a string. For instance, one may define that string ((Aa,Ba),)Ca) is almost a tree. And so on for ever and ever.*

2.3 Step three: examination of new traps...

Our problem is to fit a tree to a distance matrix. We have witnessed that as Darwinian as Lamarckian evolutions fall short of power to solve our problem over a desktop. In regard with Lamarckism, this is not a verdict but a challenge: find suitable hints that could help evolution to faster produce individuals better fitted to the environment or target function. Curiously enough, the idea of gathering small change, fundamental in Darwinism, functioned as a hint that enlivened our research. Nevertheless, our efforts also were insufficient to battle the complexity of the problem and the program falls in stagnation.

Concretely, we have found two types of regularities of local optima at which program K57 EvoFull6 that incorporates tiny mutations gets stagnated. The first is evidenced here:

```
Best topology  =
(((A,G),(((B,H),F),E)),(C,D))
Target Topology =
(((A,G),(((B,F),H),E)),(C,D))
```

The remedy is just to incorporate random swapping of leaves. Another case is represented by the following output:

```
Best topology  =
(((A,F),B),((C,H),((D,E),G)))
Target Topology =
(((A,F),B),(C,(((D,E),G),H)))
```

Let us notice that the subtopology ((A,F),B) is correct but that subtopology (C,(((D,E),G),H)) is wrong. Swapping leaves helps nothing. Instead, two types of mutations might help: re-structuring of randomly chosen subtopologies and reconstruction of a complete topology around a conserved subtopology. We restructure small subtopologies but reconstruct the complement of large subtopologies.

Now, in regard with lengths the next case shows a local optimum:

```
 BestTree  =
((B:5.0b:7.0,D:0.0b:7.0)...)
target tree =
((B:6.0b:6.0,D:1.0b:6.0)...)
```

Comparing the substrings B:5.0b:7.0 of the best tree and B:6.0b:6.0 of the target tree we decided to incorporate in one previous program tiny mutations, changes

by ± 1. Target tree would be recovered if by mere randomness two fortunate mutations compensate one another. This is plainly possible and so we found that tiny mutations accelerated evolution. Nevertheless, we can incorporate as a Lamarckian hint the possibility to make correlated mutations in a pair of lengths of the form $(\pm 1, \pm 1)$.

65 ***Code with various Lamarckian hints.*** *The program incorporates into program K57 EvoFull6 the following operations: swapping of leaves, restructuring of the complement to subtopologies, and correlated tiny mutations of the form* $(\pm 1, \pm 1)$*. The next exert shows how the restructuring of the complement of subtopologies was composed:*

```
//Program K65 EvolFull10
/*
 * We implement here
 * a full fledged evolutionary process
 * to propose an approximate solution to the problem of
 * fitting a tree to a distance matrix.
 *
 * This code incorporates into
 * program K57 EvoFull6 the following operations:
 * swapping of leaves, restructuring of subtopologies
 *  and correlated tiny double mutations of the form
 * (+/-1, +/-1).
 *
...
*/


//The complement of a  chosen at random subtopology
//is restructured
public static String
       topComplementReestructuring(String topology)
{
 if (print7)
System.out.println( "\nMMethod subtopReestructuring"
+ "\nTopology = \n" + topology);
 //Data about topology
 //Leaves and nodes in topology
 int lTop = topology.length();
```

```
String leavesOfTopol = "";
String nodesOfTopol = "";
String c = "";
//Number of right parenthesis
nRightParentheses = 0;
String rightPar = ")";
for(int j = 0; j < lTop; j++)
 {
c = "" + topology.charAt(j);
if (leaves.contains(c))
   leavesOfTopol = leavesOfTopol + c;
if (internalNodes.contains(c))
    nodesOfTopol = nodesOfTopol + c;
if ( c.contentEquals(rightPar)) nRightParentheses++;
 }

//A subtopology is chosen:
//it will remain untouched while its
//complement is restructured
String newTopology = "";
String[] subtop = new String[counterSubtop];
subtop = subtopologies(topology);
int i;
if (counterSubtop > 1)
 i = r.nextInt(counterSubtop-1) + 1;
 i=1;
String subTop = subtop[i];
if (print7)
 System.out.println("\nChosen subtopology = \n" + subTop);

int l = subTop.length();

String leavesOfSubTop = "";
String nodesOfSubTop = "";
boolean hasA= false;
boolean hasB = false;
boolean hasC = false;

//String subTop is scanned char by char
//to get information
```

```
 for(int j = 0; j < l; j++)
  {
   c = "" + subTop.charAt(j);
   if (leaves.contains(c))
   leavesOfSubTop = leavesOfSubTop + c;
   if (internalNodes.contains(c))
   nodesOfSubTop = nodesOfSubTop + c;
   if (c.contentEquals("A")) hasA = true;
   if (c.contentEquals("B")) hasB = true;
   if (c.contentEquals("C")) hasC = true;
  }
 //How many times does a appear in nodesOfSubTop?
 int howManya = 0;
 for(int j = 0; j < nodesOfSubTop.length(); j++)
 {
  c = "" + nodesOfSubTop.charAt(j);
  if (c.contentEquals("a"))
  howManya++;
 }

 int nleavesOfSubTop = leavesOfSubTop.length();
 //All large subtopologies are marked with 4
 if (nleavesOfSubTop > 3) nleavesOfSubTop = 4;

 if (print7)
 System.out.println("\nnleavesOfSubTop = "
                    + nleavesOfSubTop);
 String newSubTop = "";
  //Restruc. of complement of large subtopologies

   //Lacking leaves and nodes of subTop
   String lackingLeaves = leavesOfTopol;
   l = leavesOfTopol.length();
   for(int j = 0; j < l; j++)
{
 c = "" + leavesOfTopol.charAt(j);
 if (subTop.contains(c))
     lackingLeaves = lackingLeaves.replace(c, "");
}
```

```
   String lackingNodes = nodesOfTopol;
   l = nodesOfTopol.length();
   for(int j = 0; j < l; j++)
    {
 c = "" + nodesOfTopol.charAt(j);
 if (subTop.contains(c))
     lackingNodes = lackingNodes.replace(c, "");
}
 lackingLeaves = reorder(lackingLeaves, leaves);
 lackingNodes = reorder(lackingNodes, internalNodes);
   if (print7)
   System.out.println("Case large subTop"
        + "\nleavesOfSubTop  = " + leavesOfSubTop
  + "\nnodesOfSubTop = " + nodesOfSubTop
  + "\nleavesOfTopol = " + leavesOfTopol
  + "\nlacking leaves in order = " + lackingLeaves
  + "\nlacking nodes in order= " + lackingNodes);


   //Our restructuring method needs more than
   //2 lacking leaves
   if ((lackingLeaves.length() < 2)
   || (lackingNodes.length() < 1))
   {
 newSubTop = subTop; //Do nothing
 newTopology = topology;
   }
   else //Rearrange the complement of subTop
    {
 //Compose a topology with
 //lackingLeaves and subTop
     //but touch not subtop. To that aim,
 //replace it by the  symbol ?
 //and later restore it back to subtop:



 //  Step 1: a basic topology is built.
 //Begin with ((A,B),C) and
 //replace A or B or C by ? appropriately.
```

```
 String basicTopology = "";

 //If subTop has A
 if (hasA)
 {
  if (( (hasA) & (!(hasB))) & (!(hasC))) //Has only A
  {
   if (howManya == 0) //Has no a
basicTopology = "((?a,Ba),Ca)";
   if (howManya == 1) //Has one a
basicTopology = "((?, Ba),Ca)";
  }
  if ((hasB) & (!(hasC))) //has A and B
  {
   basicTopology = "(?,Ca)";
  }
 }
//If subTop has only B
 if (( (hasB) & (!(hasA))) & (!(hasC)))
   {
   if (howManya == 0) //Has no a
basicTopology = "((Aa,?a),Ca)";
  if (howManya == 1) //has A and B
basicTopology = "((Aa,?),Ca)";
   }


 //If subTop has only C
 if (( (hasC) & (!(hasA))) & (!(hasB)))
   {
   if (howManya == 0) //Has no a
basicTopology = "((Aa,Ba),?a)";
  if (howManya == 1) //has a
basicTopology = "((Aa,Ba),?)";
   }

 //If subtop has no A, no B, no C
 if (( (!(hasC)) & (!(hasA))) & (!(hasB)))
   {
//Add ? as a new "leaf"
```

```
 lackingLeaves = lackingLeaves + "?";
   }


   lackingLeaves = lackingLeaves.replace("A", "");
   lackingLeaves = lackingLeaves.replace("B", "");
   lackingLeaves = lackingLeaves.replace("C", "");
   lackingNodes = lackingNodes.replace("a","");


 //Basic topology is done. Now:
 //Insert lacking leaves (but do not touch subTop)
 newSubTop = basicTopology;
 while(! (lackingLeaves.contentEquals("")))
  {
   if (print7)
System.out.println("\nnewSubTop e = " + newSubTop
+ "\nlacking leaves  = " + lackingLeaves
+ "\nlacking nodes = " + lackingNodes);

 String leaf = "" + lackingLeaves.charAt(0);
   String node = "" + lackingNodes.charAt(0);
   newSubTop = insertLink(newSubTop, leaf, node );
   lackingLeaves = lackingLeaves.substring(1);
   lackingNodes = lackingNodes.substring(1);
  }

 //Restore subTop
 newTopology = newSubTop.replace("?", subTop);


 if (print7)
  System.out.println("\nnewSubTop f = " + newSubTop);
    }




 if (print7)
```

```
  {
 System.out.println("\n Topology = " + topology
 + " \nCounter of SubTops = " + counterSubtop
+ " \ni = " + i
+ "\nChosen subtopology  =  "   + subTop
+ "\nleavesOfSubTop = " + leavesOfSubTop
+ "\nnodesOfSubTop = " + nodesOfSubTop
+ "\n NewSubTopology = " + newSubTop
+ "\n New topology = " + newTopology);
  }



 //System.exit(0);
 return newTopology;
}



}//End of main class K65 EvolFull10
```

66 ***Exercise.*** *Run the program and play with the code. Play enough to accept else refute the appreciation of the Author: this program represents no improvement over its predecessors. On the other hand, code is heaping up over and over and this causes that the capability to judge about its correctness is weakened if not lost. So, carefully revise final drawings with 5 leaves to test that it actually solves the problem.*

2.4 Step four: examination of new traps...

We have battled with complexity and we have shown that one always can gain some terrain. But no one knows how hard is that battle unless he or she comes to the battlefield:

67 ***Challenge.*** *Give a new round to the the Lamarckian recursion to hint evolution. You must look directly into the structure of local optima to propose a method to exit them. Additionally, you might investigate the effect of introducing restructuring of subtopologies. We already have incorporated restructuring of whole topologies and restructuring of the complement of topologies but now we wonder at the possible benefits of restructuring of subtopologies. Try it out.*

68 ***Challenge.*** *The three rounds just given by the Author to the Lamarckian recursion caused him to suffer a lot. Why? That was because to make new modifications became harder and harder. This is a generalized illness of software development for complex problems and it is the blind alley of non evolvable software. Experts consider that that alley is the final state of inexperts and only of inexperts. If you accept their dictum as a challenge, reorganize the previous code to make it more evolvable, modifiable. You might, say, notice that there is a lot of repeated code in various methods. So, large pieces of code might be fractionated in such a way as to minimize repetitions and to gain combinatorial power, which is a key point of evolution.*

69 ***Challenge.*** *Show else refute the next suspicion of the Author in regard with the problem of fitting a tree to a distance matrix: there is a chain of hints of evolution that produces an increasing improvement in performance but such that any finite subset is useless to attacking a problem with a number of leaves sufficiently large. An example of that chain is the ordered set of correlated tiny random mutations of the lengths of links given by*

± 1
$(\pm 1, \pm 1)$
$(\pm 1, \pm 1, \pm 1,)$
...
$(\pm 1, \pm 1, ..., \pm 1, ...)$.

We have seen that the single mutation (± 1) *together with double correlated mutation* $(\pm 1, \pm 1)$ *are very good accelerators. But the problem is that if one augments the number of leaves, these hints are of no help.*

70 ***Challenge.*** *Refine program K6 EvolFull10 that it could accept trees with thousand of leaves and improve the drawing facility that one can visualize so large trees. The program must run with trees with 30 leaves or species. Leaves were denoted in our previous programs with upper case letters while inner nodes with lower case letters. Think of the corresponding notation under this new setting. On the other hand, one letter notation is good for simulations, but for the study of data of natural populations, leaves correspond with species that come with their own names. How would you deal with that? In regard with simulation, the next program reminds us that Java have letters by thousands.*

71 ***The chars of Java***

```
//Program K71 alphabets
//This program prints the chars of Java
//corresponding to many alphabets
public class alphabets
{
 public static void main(String[] args)
  {
   for(int i=0;i<=5000;i++)
     System.out.println(i + " " + (char)i);

  }
}
```

72 ***Challenge.*** *We have too many parameters: population size, recombination rate, and we have various mechanisms for mutation that can be controlled by specific mutation rates, not all of which have been explicitly declared. By giving different assignations to these parameters, prove that the performance is variable as a function of their values. This means that we have another optimization problem: to finely tune the dynamic setting of parameters. This new problem might also be solved by evolution. Would someone give a try? A question like this suggest that supercomputing is the ordinary world for the study of evolution. This is one more reason to think of the EvolJava grid. Nevertheless, supercomputing may refer not to machines but to brains: because developing problems are so harsh, we need great expertise in programming. So, it is not excluded that you, following your intuition, could render assignations of parameters in such a way as to produce marvelous results.*

2.5 Conclusion

By working over the problem of fitting a tree to a distance matrix, we have illustrated a methodology to improve the performance of genetic algorithms which are simulations of evolution to solve optimization problems. The method consists in recursive rounds of studying the structure of local optima to propose hints to pass to a better local optimum. In general, hints twist randomness by making more probable those changes that enable and/or accelerate evolution. This is our vision of Lamarckism. The next is an example of a hint that is specific to our problem: correlated mutations of the form (+/-1, +/-1). This means that we choose at random a link, add one to its lengths and subtract one from the length of the link of the

accompanying node. According to our experience, the task of hinting evolution is extremely cumbersome while improvements might be poor.

Chapter 3

Why is the tree problem so difficult?

`So much work to get so feeble results`

73 ***Introduction.*** *We have been using evolution to propose an approximate solution to the problem of fitting a tree to a distance matrix. Darwinian evolution seems very ineffective to deal with this problem. Nevertheless, we have armed evolution with what we thought were powerful hints and we ended with program K65 EvolFull10 that exemplifies Lamarckian evolution. The glory of our work and the misery of evolution are revealed by a typical experiment: the program was launched 4 times with an input matrix with six entries to be fitted by a tree with 6 leaves. After five hours of running in a desktop at 4 Gigahertz and 150000 generations of evolution, two launches ended with an approximate solution with an error of two and with the correct topology, another with an error of 8 and another with an error of 12. The last two launches were not able of finding the correct topology. The error of the initial population that was synthesized at random was in all cases around 180. If one rises the number of leaves to 20, the program can lower the initial error to a half after running an hour but the best found topology just has tiny coincidences with the target one.*

This experience has been highly traumatic to me: I expected more from evolution. The trauma is the following: A year ago I imagined Evolution as an automatic procedure to perfect an incipient good idea. So, I was open minded to the possibility that evolution could build thousands of species beginning from, say, a protobird that was created by God. But now, I know that evolution can perfect good ideas that solve complex problems to the level of mediocrity or maybe up to acceptable solutions but not to the perfection I see in living beings. So, I am now completely

reticent against cheap evolutionism: if someone wants me to respect his or her ideas in regard with the evolution of species, he or she must exhibit as field observations as laboratory work that shows the nature of sexual barriers together with the probability of retro-mutation that erases those barriers. Example in that direction: Grand and Grant (2014). Information of this sort will show that the notion of species based on sexual barriers is futile, just enough to make believe that the Evolutionary Theory has scientific confirmation.

We see that the Author has converted a personal experience into a world vision that allows him to deprecate the work of thousands of scientists during 160 years. Is that right?

74 ***Objective.*** *We will propose a characterization of the type of complexity possessed by the problem of fitting a tree to a distance matrix to next argue that extant life is pervaded in that type of complexity. Hence, we feel authorized to claim that as evolution is powerless to attack our fitting problem, so it is powerless to cause deep changes in the genetics of life. Nevertheless, evolution can modify superficial characters, such as those that are responsible for the arising of some types of sexual barriers and speciation.*

3.1 Our proposal

Why is the problem of fitting a tree to a distance matrix so difficult for evolution even when this has been powerfully hinted? The explanation we offer is very simple:

75 ***Every change has many side effects, some good other bad.***

Let us consider that we have found the correct topology and that we are busy trying to finely tune lengths of links and that we got trapped in a local minimum. The great frustration is that seemingly there is nothing to be done because every modification of the length of anyone link causes that every distance between every pair of leaves is automatically changed and the error also increases.

76 ***Example.*** *Let us consider that we are trying to find the best fitting tree to the target matrix given below and that we already have found the correct topology (((A,D),B),(C,E)). Our present situation is given by:*

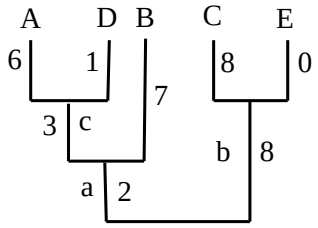


Figure 3.0. Our tree:
(((A:6.0c:3.0,D:1.0c:3.0)a:2.0,B:7.0a:2.0),(C:8.0b:8.0,E:0.0b:8.0)a:2.0).

It happens that this tree differs from the optimal solution by just one change: the link associated to C has length 8 but it must be 10. Our program describes the situation as follows:

```
Target tree   =
(((A:6.0b:3.0,D:1.0b:3.0)a:7.0,B:7.0a:7.0),(C:9.0c:2.0,E:1.0c:2.0)a:7.0)
Best tree =
(((A:6.0c:3.0,D:1.0c:3.0)a:2.0,B:7.0a:2.0),(C:8.0b:8.0,E:0.0b:8.0)a:2.0)
Distance to target = 2.0
Best distance at initialization = 36
)



Distance matrix associated to best tree
    A    B    C    D    E
A  0.0 16.0 27.0  7.0 19.0
B 16.0  0.0 25.0 11.0 17.0
C 27.0 25.0  0.0 22.0  8.0
D  7.0 11.0 22.0  0.0 14.0
E 19.0 17.0  8.0 14.0  0.0

Target distance matrix
   A    B    C    D    E
A  0.0 16.0 27.0 7.0 19.0
B 16.0  0.0 25.0 11.0 17.0
C 27.0 25.0  0.0 22.0 10.0
D  7.0 11.0 22.0  0.0 14.0
E 19.0 17.0 10.0 14.0  0.0
```

One might naively think that the obvious solution is to change the length of link C from 8 to 10. This will indeed remedy the problem with the distance between C and E. But the price we must pay is that all other distances will change. Say, the distance between A and C will change from 27 to 29. And so on: every distance will suffer a change. This implies that every single change, tiny as it could be, will cause the error to increase.

3.2 Relation with genetics

The observed phenomenon in which anyone change has many side effect is also present in genetics where it has been well studied:

Pleiotropy = pleio (more) + tropos (turn) is the effect of a single gene on more than one characteristic. We see that, in the problem of fitting a tree to a distance matrix, lengths of links function as genes and distances in within pair of leaves as characters: a change in any gene produces a change in every character. An example in genetics is the ”frizzle-trait” in chickens: that gene is directly responsible for the observed curled feathers that give fouls a very nice appearance. But besides, this gene also caused the fowl to have abnormal body temperatures, higher metabolic and blood flow rates, and greater digestive capacity. So, we see that a change in this gene causes both good and bad side effects on diverse characters (Lobo, 2008).

77 ***Example in medical genetics: PKU = phenylketonuria. This example and others below were documented mainly from Wikipedia.***

If phenylalanine is found in the urine, one suffers from PKU. This is a terrible disease for children, whose mechanisms is as follows. In a healthy organism, phenylalanine is converted into tyrosine by the enzyme phenylalanine hydroxylase. If a mutation in the corresponding gene causes a defective enzyme, phenylalanine accumulates in the bloodstream and attacks directly the nervous system. So, a child that suffers from PKU has mental retardation. Moreover, the nervous systems marshals part of the ontogeny, the development of the creature. So, if the developing nervous system is defective, most surely the corporal development will be also defective. This might explain why the ill child suffers abnormalities in the articulations, and so gait and posture are not the normal ones. On the other hand, tyrosine is used by the body to make melanin, the pigment that gives a obscure brown color to the skin. If tyrosine is lacked, skin and hair can lack pigmentation. Tyrosine has various roles in the organism: it is used as a precursor of thyroxine, dopamine and norepinephrine. Thus, a child with PKU also has endocrine troubles.

If a child with PKU survives, the mutation in the enzyme phenylalanine hydroxylase is not that severe and so it produces some level of tyrosine. The problem is then how to avoid an excess of phenylalanine. It happens that this amino acid comes in the food and one can observe specialized diet and get healthy children. The ill child cannot take milk, cheese or peanuts.

78 ***Exercise.*** *Make your best to prove that life is pervaded in extreme pleiotropy and that therefore evolution is impotent to cause severe changes to the structure of extant life. Relate pleitropy with the would-be evolutionary appearing of species and with their ecstasy. You can compare your answer with ours that appears in the next chapter.*

3.3 Conclusion

We have armed evolution with a varied series of hints but anyway it was incapable of dealing with the problem of fitting a tree to a distance matrix for a number of leaves beyond 5. Upon examination, our problem resulted to be difficult for evolution because it is an extreme case of pleitropy that results when every gene affects every observable character. The lengths of links in a tree functions as genes and the distances between pair of leaves as characters: pleitropy means that if one changes the length of whatever link, all distances change. Pleitropy has been abundantly documented in genetics, so the failure of evolution to attack pleitropy has anti-evolutionary implications.

Chapter 4

Evolution vs. pleitropy

We have witnessed in this volume that as a child that plays with a plastic shovel to attack a basalt mountain, so is evolution trying to solve the problem of fitting a tree to a distance matrix. It is argued that this fitting problem and that of the evolutionary origin of species must solve the very same difficulty: massive and pervasive pleitropy that happens when products use to have many roles. In spite of our failure, we found that it is sound to suggest that the origin of species could be linked to evolution. Nevertheless, the Evolutionary Theory of the origin of species is obviously false: evolution under pleiotropy predicts a huge amount of mediocre solutions as in the fossil record as in extant populations but nothing like that is found in nature. In fact, extant life is filled in perfection from the Precambrian to our days.

4.1 Introduction and purpose

Our aim in these last two volumes has been to challenge evolution with the problem of fitting a tree to a distance matrix. Of all possible problems this is the one and only that for our honor we must solve with the exclusive use of evolution: this is the problem that best incarnates evolution. In fact, evolution and the tree of life associated to a matrix of some sort of genetic distances are inseparable.

We used as test a distance matrix that was derived from a tree and we demanded from the program to output the tree with which we began.

We tried out in Volume X a solution by scanning all possible tree topologies (just the graph of the tree), one by one, to find the best tree (graph + lengths of links) in within each chosen topology. These bundle of optimization problems were solved by evolution. After certain number of generations the algorithm was stopped and the best topology was declared to be that with the optimal tree. The

inefficiency of this approach awoke the need to look for a new one. So, it was found that for 6 leaves or less the best topology could be singled out by comparing performances rather early, after say, 150 generations over a population of 50. So, to improve the algorithm we included two phases. The first was to scan all possible topologies during some 100 generations to single out the best one. The second was to find by evolution the best tree associated to the chosen topology.

We found that our improved algorithm was good to solve the posed problem with 5 leaves. Beyond 5 leaves the running time was so long that the algorithm was declared to be inappropriate for a desktop.

The immediate explanation of our failure is that evolution was not used to its full power. In fact, mutation and recombination were implemented inside tree topologies and nothing else. So, our first purpose for this volume was to test the power of mutation and of recombination of topologies: instead of a bunch of optimization problems to be solved separately, we have just one problem that explores alternatives and recombines them to find the best. Found troubles with the implementation are worth a comment in regard with two types of evolution.

4.2 Darwinian evolution vs. Lamarckian evolution

The basic idea of Darwinism such as we see it today in relation with mutation and recombination is that they happen at random. On the other hand, the basic idea of Lamarckism, such as we understand it, is that evolution can be helped by twisting randomness to make more probable those changes that lead to full adaptation of the evolving population to their environment or selective procedure. This contrast seems to imply that randomness in Darwinism must be defined as randomness according to the distribution given by the direct encoding of (nude) evolution. What does this mean?

To fix ideas, let us consider a Java simulation. The encoding of evolution allows here to interpret randomness as randomness according to a uniform distribution: it is readily implemented and produces no twisting, no preference for certain mutations. So, in regard with our Java simulation, **Darwinism** means that mutation happens at random according to a uniform distribution. Every sort of twisting of randomness will be referred to as applied **Lamarckism**. But if we turn to the genome, we must take into account that mutation is driven by biophysics and biochemistry in rather complex operations and there is no need to suppose that, say, all possible base substitutions happen with equal probability. The experimentally observed distribution is what must be taken as the distribution that rules random-

ness in Darwinian evolution when the genome is referred to. We already know how to simulate in Java those cases.

We always have been in the real of Lamarckian evolution:

Darwinian evolution is prey of sampling effects and so one can observe that the fittest individual of a given generation may be less fitter than the best individual of the previous one. One finds this feature very disappointing. Nevertheless, a Lamarckian solution to this problem is to clone the best individual of a given generation and in this form the best solution never will be worst than those of the past. We did this rather early but anyway results were very poor to be of interest: the program gets stagnated at mediocre solutions and cannot leave them anyhow. Because we are dealing with optimization problems, we say that the program has been trapped in a local optimum.

For complex problems, Darwinian evolution is prey of general inviability. To fix ideas, let us think of our problem of fitting a tree to a distance matrix. If mutation can change a char by another, then it is perfectly allowed that leaf C may be substituted by leaf D and so the mutant string has too leaves D and no leaf C: the string does not encode for a tree on the original sets of leaves. It is not viable and this is a generalized illness. And, what about recombination? In many cases it happens that the encoding of evolution naturally leads to recombination of homologous sequences, i.e., of sequences that encode for the same functional subunits. This is also the case of the genome. For those cases, Darwinism means that the site of recombination is chosen at random according to a uniform distribution. Now, it is readily seen that Darwinian recombination of strings encoding for trees lead to general inviability.

So, our situation can be described as follows:

1. It is useless to think that Darwinian mutation and recombination of topologies may rescue us from our troubles with found combinatorial explosion.

2. To implement Lamarckian evolution we must twist randomness to our favor. Since our problem is general inviability, we must find a method to fabricate viable mutations and recombinations. Our choice was to refer mutation to the tree of trees. Let us recall that the tree of trees is a tree that results in the recursive generation of all possible trees. It is initialized with a tree with 3 leaves and all possible trees are built by successive insertion of other leaves. So, the tree of trees is our natural frame for implementing evolution among

tree topologies. Mutation was and will be understood as a suitable walk along the tree of trees.

3. Recombination will be executed as the substitution of one sub-tree (a substring that is a tree on its own) by another only if both derived subtopologies (a substring that is a tree topology on its own) have the same subset of leaves.

In that way we implemented the power of evolution to the full. An, what did we get? We make an improvement but without leaving the class of useless programs because of its exaggerated waste of time.

Our next purpose was to find others ways to somehow help evolution.

4.3 The Lamarck recursion

When one finds that a Lamarckian simulation of evolution is inefficient and wants to improve its performance, the first step is to look in hindsight:

Since Lamarckism is all about twisting of randomness, one must be aware that any twisting favors some specific line of future and disfavors all the others: is the overall output positive or is it a hindrance? In this regard we present and resolve an important complaint against the form as we implemented Lamarckism.

The idea of mutation is to explore the world. In our case, the world is the set of topologies. When one faces to a combinatorial explosion, it is mandatory to visit as many states as possible in the least possible time. This implies that all states must have the same probability of being visited. What happens if the visiting distribution is not uniform? A bias is created that favor the exploration of some parts of state space and disfavor others. More to the point, some problems may not be solved in any finite lapse of time. Because we want to create a tool to deal with every kind of problems, no bias is allowed.

Which is our case?

The mutation we implemented at the beginning caused faster mutation rates of subtrees with recently added leaves. To see that, let us consider a process of generating a tree: we begin with the tree ((A,B),C) and insert in order leaves D,E, and F. Once the tree is formed, we generate a number between 1 and 3 corresponding to the number of leaves that must be deleted. The corresponding distribution is uniform. But it is wrong to imagine that this means that leaves D,E and F will be deleted with equal probabilities. Instead, if 1 is generated, leaf F will be deleted. If 2 is generated, leaves F and E will be deleted. If 3 is generated, leaves F,E

and D will be deleted. So, leaf F will be deleted with certainty but leaf E will be deleted with probability 2/3 and D with probability 1/3. Leaves A, B and C never are removed.

We have proved that our procedure to mutate trees is loaded with a bias due to the recursion we use to built trees. Therefore, the algorithm preferentially explores the zone around the last added leaf. This is clearly a bonus when one wants to explore states around that leaf. But it is an ache if we want to make wider explorations.

So, the next task was to implement deletion of a leaf at random. After implementation an improvement was found but again without leaving the class of useless programs: we irremediably fell trapped into local optima representing mediocre solutions.

Now, every chaser examines what his trap has captured. By the same token, a recurrent step of a Lamarckian methodology for solving optimization problems is to study local optima to catch a regularity in order to look for a hint that causes the program to scape out of them. We tried out various hints. For instance, we discovered that tiny mutations, in agreement with gradualism, accelerated evolution and so they functioned as a hint. This positive result triggered our curiosity: what can be the role of tiny mutations alone? We found that they have no power at all. Another example of a hint that is specific to our problem is to cause correlated mutations of the form (+/-1, +/-1). This means that we choose at random a leave, add one to the length of its link and subtract one from the length of the link of the accompanying node. This hint has a not bounded chain of improvements denoted as (+/-1,..., +/-1).

4.4 Our problem is very difficult

After so much work, we declare the problem of fitting a tree to a distance matrix to be intractable for evolution: it is not good to find the perfect solution for a problem with a number of leaves higher than 6, although it can propose approximated rough solutions for a number of leaves as high as 20. But, moreover, this project has been extremely difficult for the developer.

Developing software is difficult in general but to work in this task has been particularly challenging while improvements resulted to be poor or nil. In fact, pattern recognition of strings that indicate a character becomes too tedious because of too many iff's that cumulate over and over. Besides, one swings among tiny details that call for reuse of extant code. Our solution has been to duplicate the code to modify it apart. It seems that duplication is the first step towards intelligence driven evolution of code development. Many of these duplications were made to

fix bugs. Because one corrects bugs as they appear, the final output is a mountain of code whose existence cannot be explained anyhow because one has no intelligence to see the explaining pattern, if any. Moreover, some bugs cause malfunction far in the future after their action and backtracking is always cumbersome. The remedy is always the same: to divide great troubles in small ones and to print the output of any suspicious line of code to compare it with what one expects. Automatic detection of wrong outputs also was included. This has been very helpful.

From all this suffering, goods news emerge: since to hint evolution is extremely difficult for very difficult problems, there is plenty of work for developers that love to cultivate intelligence and endurance.

4.5 Why is our problem so difficult?

Let us explain why this problem is so difficult for evolution.

To fix ideas, let us consider that we are trying to find the best fitting tree to the target matrix given below and that we already have found the correct topology (((A,D),B),(C,E)). Our best approximated solution is depicted by:

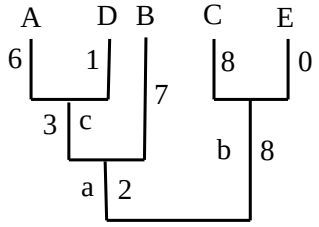


Figure 4.0. Our tree:
(((A:6.0c:3.0,D:1.0c:3.0)a:2.0,B:7.0a:2.0),(C:8.0b:8.0,E:0.0b:8.0)a:2.0).

It happens that this tree differs from the optimal solution by just one change: the link associated to C has length 8 but it must be 10. Our program describes the situation as follows:

```
Target tree   =
(((A:6.0b:3.0,D:1.0b:3.0)a:7.0,B:7.0a:7.0),(C:9.0c:2.0,E:1.0c:2.0)a:7.
Best tree =
(((A:6.0c:3.0,D:1.0c:3.0)a:2.0,B:7.0a:2.0),(C:8.0b:8.0,E:0.0b:8.0)a:2.
Distance to target = 2.0
```

```
Best distance at initialization = 36
)



Distance matrix associated to best tree
    A    B    C    D    E
A  0.0 16.0 27.0  7.0 19.0
B 16.0  0.0 25.0 11.0 17.0
C 27.0 25.0  0.0 22.0  8.0
D  7.0 11.0 22.0  0.0 14.0
E 19.0 17.0  8.0 14.0  0.0

Target distance matrix
   A    B    C    D    E
A  0.0 16.0 27.0 7.0 19.0
B 16.0  0.0 25.0 11.0 17.0
C 27.0 25.0  0.0 22.0 10.0
D  7.0 11.0 22.0  0.0 14.0
E 19.0 17.0 10.0 14.0  0.0
```

One might naively think that the obvious solution is to change the length of link C from 8 to 10. This will indeed remedy the problem with the distance between C and E. But the price we must pay is that all other distances will change. Say, the distance between A and C will change from 27 to 29. And so on: every distance will suffer a change.

In short, the difficulty of the problem resides in that every single change in a basic parameter will cause changes in every observed character.

4.6 Relation with genetics

We see that, in the problem of fitting a tree to a distance matrix, lengths of links function as genes and distance in within pair of leaves as characters: a change in any gene produces a change in every character. This is situation is known as **pleitropy** and our problem gives rise to its maximum degree: a punctual change directed to solve a local problem always causes troubles everywhere around.

Pleitropy was discovered in genetics because some inborn illnesses with many symptoms were found to be caused by just one single mutation. The classical

example is PKU = phenylketonuria. This example and others below were documented mainly from Wikipedia.

If phenylalanine is found in the urine, one suffers from PKU. This is a terrible disease for children, whose mechanisms is as follows. In a healthy organism, phenylalanine is converted into tyrosine by the enzyme phenylalanine hydroxylase. If a mutation in the corresponding gene causes a defective enzyme, phenylalanine accumulates in the bloodstream and attacks directly the nervous system. So, a child that suffers from PKU has mental retardation. Moreover, the nervous systems marshals part of the ontogeny, the development of the creature. So, if the developing nervous system is defective, most surely the corporal development will be also defective. This might explain why the ill child suffers abnormalities in the articulations, and so gait and posture are not the normal ones. On the other hand, tyrosine is used by the body to make melanin, the pigment that gives a obscure brown color to the skin. If tyrosine is lacked, skin and hair can lack pigmentation. Tyrosine has various roles in the organism: it is used as a precursor of thyroxine, dopamine and norepinephrine. Thus, a child with PKU also has endocrine troubles.

If a child with PKU survives, the mutation in the enzyme phenylalanine hydroxylase is not that severe and so it produces some level of tyrosine. The problem is then how to avoid an excess of phenylalanine. It happens that this amino acid comes in the food and the child can observe specialized diet and recover health. The child cannot take milk, cheese or peanuts.

Now, we content that extreme pleiotropy pervades extant life. Why?

Enzymes work in ensemble lines processing the products of other enzymes. Because the expression of the genes encoding for enzymes varies from organ to organ, the same input molecules may be involved in different organs to built different products.

Example: tyrosine is processed by the thyroid gland into thyroxine, by the brain in dopamine, by the medulla into norepinephrine, also called noradrenaline, by the adrenal gland into epinephrine, also known as adrenaline, and by the skin into melanin.

Or consider glutamate. It is an amino acid that appears in the genetic code and so may form part of a great majority of proteins. So, it has, say, one thousand of different roles. Besides, it on its own is used as neurotransmitter, it is the precursor for the synthesis of the inhibitory GABA as in the cerebellum as in the pancreas.

Think of cholesterol. This molecule is feared because it causes cardiovascular disease when its blood levels are high. But, behold, cholesterol is a fundamen-

tal component of cell membranes, it is necessary both to build and to maintain them. Moreover, it also serves in cell signaling and nerve conduction. In the liver, cholesterol is converted to bile, whose accompanying salts are necessary to solubilize fats and fat-soluble vitamin A to enable its absorption. Cholesterol is converted by the skin into vitamin D. By the adrenal gland into hormones cortisol and aldosterone. In sexual glands cholesterol is used to synthesize progesterone, estrogens, and testosterone.

Adenosine is also multipurpose. It is used in every cell to sythesize adenosine triphosphate (ATP), the molecular recargable battery of the cell. In the form of cyclic adenosine mononophosphate, it echoes to the plasma cell the signal given by glucagon that cannot pass though the cell membrane. It is also an inhibitory neurotransmitter. It also causes vasodilation in various organs.

Riboflavin or vitamin B2 is a pleitropy champion, as all vitamins do, because it is the central component of the cofactors FAD and FMN that are required by all flavoproteins. These proteins exhibit a wide range of redox potential and therefore can play a wide variety of roles in intermediary metabolism. Some of these roles are:

1. Flavoproteins play important roles in the electron transport chain.
2. Decarboxylation of pyruvate and α-ketoglutarate requires FAD.
3. Fatty acyl CoA dehydrogenase requires FAD in fatty acid oxidation.
4. FAD is required to the production of pyridoxic acid from pyridoxal (vitamin B6).
5. The primary coenzyme form of vitamin B6 (pyridoxal phosphate) is FMN dependent.
6. FAD is required to convert retinol (vitamin A) to retinoic acid.
7. Synthesis of an active form of folate (5-methyl THF) is FADH2 dependent.
8. FAD is required to convert tryptophan to niacin (vitamin B3).
9. Reduction of the oxidized form of glutathione (GSSG) to its reduced form (GSH) is also FAD dependent.

Genetic combinatorial bases are also sources of rich examples of pleiotropy. A combinatorial basis is a set of genes whose products are combined to create varied full functional units. Example: the immune system can recognize thousands of antigens thanks to immunoglobulins that result from the combinatorics of the products of a number of exons. The same happens with the receptors of light that can distinguish among hundreds of colors by registering the intensity of the stimulation of tree receptors, yellow, red and blue. That is also the case with the perception of odors: a human nose has around 400 types of scent receptors that report type and quantity of excitation. Additionally, we have some 50 types of gustatory receptor cells. The richness of these last two systems helps to explain why some trained persons can distinguish so great variety of wines. Now, these cases of pleitropy have different selective importance: while some humans that lack the possibility to perceive odors can go around having a happy life, special care is needed by those with defective immune system. We have here an example of pleitropy of maximal importance and another of low one.

We can conclude that pleitropy is a politics of design of the biochemistry network of life. The reason is that it saves code. On the other hand and according to our work in this volume, pleiotropy is the anti-evolutionary agent per excellence.

So, what is the meaning of pleiotropy for the average Joe?

It means that everything is intertwined with everything and so if he has been born healthy, his health will be highly stable. But if his dynamical status quo is shifted by whatever reason, and that always will happen, the new situation could be harmful and he gets ill. To recover health many correlated and finely tunned changes would be necessary: the illness is chronic. Pleitropy means at this moment that troubles multiply and he irremediably worsens. Recovery might turn to be so complex a task that he dies.

4.7 Relation with evolution

Immediately one would exclaim: if a species is modified, the new thing will be doomed to die, to extinction because the most probable outcome of a modification of a complex pleitropy unit would be a genetic failure. While this is expected to be a general truth, exceptions are not excluded. To motivate a discussion about their existence, let us bring forth the case of the genetic code. This is a table that associates genetic codons with the amino acids that will be used to synthesize proteins and enzymes. If you make a change in the genetic code, almost all proteins

will suffer a change and, given the extreme selectivity of most enzymes, inviability would be the most probably output. Nevertheless, it has been experimentally proved that the genetic code is modifiable and many industrial examples exist, say, an artificial genetic system has been tailored that works with 4 bases per codon instead of 3. Apart from this, there are more than 10 different genetic codes in nature and plausible evolutionary explanations that have been accepted by science have been put forward.

The variability of the genetic code is spectacular because it is by its very nature at the deepest level of pleitropic influence. This contrast gives wings to the idea that exceptions against pleitropy might add through hundreds of millions of years to an explanation of the existence of species. Is that really possible?

Were life conformed by a whole pleitropic unit, there were no evolution at all. But reality is more complex than that and the stagnating force of pleitropy in nature is graded from zero to infinite. In fact, very small pleitropic units are fully modifiable, say, to fit a tree to a distance matrix for two species is a problem that shall offer to evolution no trouble. Our suffering began with 6 leaves. And the complexity increases with the number of additional leaves. Now, life has characters that conform whole, indivisible pleitropic units, say, the system responsible for the basal metabolism of the cell. It is reused in circa 280 different types of cells in our organism. This does not mean that all involved genes, more than 400, are responsible for products that conform pleitropic units but that more than 20 are. This core is, of course, evolution-proof invariant. Other subsystems may experience weaker pleiotropic interactions, like the system responsible for the synthesis of the nails: some people lack a nail but this causes no great affection. Yet others characters may be almost completely free, as those related with some tonalities of the color of the hair.

Now, you can define species in terms of pleitropic-free or almost pleiotropic-free characters and your definition will count with many natural confirmations. By the way, the evolutionary definition of species is of that nature: the spermatozoon needs to recognize a receptor in the ovule to fecundate it. If this recognition task is interdicted, by a mutation, infertility results. But another compatible change in the ovule may restore pattern recognition and we have the birth of a new evolutionary branch: we predict speciation according to the evolutionary definition whose field confirmation has been reported in sea urchin. This definition of speciation is very cheap because speciation and hybridization are not antagonistic terms: compare (Grant and Grant, 2014).

We get a picture of evolution in which a core remains invariant, maybe included as part of a more complex structure, while other subsystems with shallow pleiotropy may suffer strong changes. Is that the type of life that we see in this

planet? Yes, it seems to me that life is just like that. So, it is sound to suggest that life is linked to evolution. But, does this mean that evolution is the cause of our existence? The answer is: it is obvious that evolution is not the cause of our existence. If life or species had arisen by evolution, the Earth would be filled in long standing mediocre solutions that arouse by evolution towards complexity when pleitropy was weak. It would also exist some excursions into acceptable quality systems and some exceptional cases of high quality products. But, how is the world I live in?

1. It is plenty of pervading and overwhelming perfection from bacteria to whales. To understand what is perfection, just consider your bladder: it is elastic (you can keep there more than you ever have thought), extremely strong (no fear at all that it will break and pour out urine inside your body) and it allows to be perfectly voided.

2. There is no trace of an evolution from imperfection to perfection: there is no registered inelastic bladders.

3. We lack evidence of the many troubles that one encounters when trying to modify a non trivial pleitropic unit: bladders are part of a system that causes too much trouble to old people. But nobody is thinking of genetic engineering to change things. Why? It is because we lack both computing power and knowledge to rule out the myriads of false but convincing solutions.

4.8 Conclusion

We have armed evolution with a varied series of hints but anyway it has been incapable of dealing with the problem of fitting a tree to a distance matrix for a number of leaves beyond 5. Upon examination, our problem resulted to be so difficult for evolution because it is an extreme case of pleitropy that results when every gene affects every observable character. In fact, the lengths of links in a tree functions as genes and the distances between pair of leaves as characters: pleitropy means that if one changes the length of whatever link, all distances change. It has been argued that pleitropy pervades life. Thus, the idea that evolution could explain the origin of the generality of species appears to be ridiculous. Nevertheless, a better understanding shows that it is actually sane to suggest that evolution explains the origin of species: pleitropy in diverse degrees predicts a core of invariant characters, such as the basal metabolism that is shared by circa 300 types of different cells in our organism, plus a variable, evolvable part, such as the tonality of the color of the hair. So, it is sound to suggest that life is linked to evolution and no doubt the appearing of some species will readily be explained by it. Nevertheless, the Evolutionary Theory of the origin of species is obviously false in general: pleitropy

predicts abundant and long standing cases of mediocrity as in the fossil record as in modern populations but nothing like that is found in nature.

Answers to exercises

Problems of Chapter 1

15, page 14. The code that implements antiRule2 follows:

```
//Program K15 DeleteLeaf2
//This program generates a random tree and
//deletes the last inserted leaf.

//We apply two deleting rules:
//antiRule 1:
//(An,Fn)i  -> Ai
//antiRule 2:
//((S)n,Fn)i -> (S)i
//where A and F are leaves, n and i are  inner nodes,
//(S) is a complex structure.
//These rules are inverse to inserting rules.

//Both rules are implemented.



import java.util.Random;


public class DeleteLeaf2
{

 //Link's notation for trees
```

```
 private static   String   parent =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;

 private static  Random r = new Random();

 private static boolean print1 = false;
 private static boolean print2 = true;

 //After deleting the last inserted leaf,
 //this must result:
 private static  String expected = "";

 //Detect internal nodes, different than leaves
 private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
  else return false;
 }

//Simplification from Link notation to Newick's notation:
 // (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTree)
{
 String tree = "";
//length of string newTree
int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
```

```
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
  }
 return tree;
}

//Rule 2 is used to generate a new tree:
//(S)I -> ((S)N,FN)I.
//S = subtree, I = internal node, F = new leaf,
//N = new internal node.
//Rule 2 grafts new branch FN to link (S)I
//i is the position of char I in string parentTree
private static String useRule2(String parentTree, int i,
        char F, char N)
{
 String newTree = "";
 char I = parentTree.charAt(i);
 if (print1)
 System.out.println("i = " + i + " I = " + I);
 //Subtree S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = i;
 boolean go = true;
 //All right parentheses to the near left of I are captured
 while (go)
  {
   position = position -1;
   String a = "" + parentTree.charAt(position);

   if (print1)
```

```
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
   else go = false;
  }//End of while
 if (print1)
 System.out.println("counter = " + counter);
 //The complete subtree is captured:
 //Parentheses must balanced
 while (counter > 0)
  {
   String a = "" + parentTree.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
   System.out.println("Subtree = " + S);
   position = position -1;
   }
 //Replacement rule 2:
 //((S)n,Fn)i -> (S)i
//where   F are leaves, n and i are  inner nodes,
//(S) is a complex structure.
 String SnFni  =   S  + I;
 String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
 //Rule 1 is executed
 newTree = parentTree.replace(SnFni, replacement);
 //If I links to a complex structure
 return newTree;
}

//From parentTree, a tree in Newick's notation,
//a Child  tree is generated by a generative grammar.
//A random election is made among all possibilities.
```

```
private static String generateChild(String parentTree)
{

 nextLeafIndex++;
 char nextLeaf= leaves.charAt(nextLeafIndex);
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = internalNodes.charAt(nextIntNodeIndex);


 //link to be replaced
 String link = "";
 //String link is replaced by what:
 String replacement = "";
 //new tree generated by replacement
 String newTree = "";
 //length of parent tree
 int l = parentTree.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
  {
   //parentTree is scanned char by char
   char I = parentTree.charAt(i);
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
  }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
  {
    //parentTree is scanned char by char
char I = parentTree.charAt(i);
//Is I an internal node that denotes a replacement?
```

```
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = parentTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(parentTree, i,
               nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTree = parentTree.replace(link, replacement);
 }
System.out.println(newTree + " -> "
                      + simplify(newTree));


 i = l;
   }
  counter++;
}
  }
 return newTree;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply the antirule1:
//(An,Fn)i  -> Ai
private static String  antiRule1(String tree,
```

```
int positionLeaf, int positionNode)
{
 System.out.println("AntiRule1");
 int beginIndex = positionLeaf - 4;
 int endIndex = positionNode + 3;
 String AnFni = tree.substring(beginIndex, endIndex);
 System.out.println("Anfni = " + AnFni);
 String Ai = "" + tree.charAt(beginIndex+1)
                + tree.charAt(endIndex-1);
 System.out.println("Ai = " + Ai);
 tree = tree.replace(AnFni, Ai);
return tree;
}



//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply the antirule2:
//((S)n,Fn)i -> (S)i
//positionNode refers to the last appearing of node n
private static String  antiRule2(String tree,
int positionLeaf, int positionNode)
{
System.out.println("AntiRule2");
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
            + internalNodes.charAt(nextIntNodeIndex);
 int firstPosition = tree.indexOf(innerNode);
 String newTree = "";
 char i = tree.charAt(positionNode+2);
 if (print2)
 System.out.println("FirstPosition = " +
                      firstPosition + " i = " + i);
 //Subtree S of link (S)I must be read
```

```
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = firstPosition;
 boolean go = true;
 //All right parentheses to the near left of I are captured
 while (go)
  {
   position = position -1;
    String as = "" + tree.charAt(position);

   if (print2)
   System.out.println("a = " + as + " p = " + pr);
   if (as.equals(pr) )
   {
counter++;
S = as + S;
   }
   else go = false;
  }//End of while
 if (print2)
 System.out.println("counter = " + counter);
 //The complete subtree is captured:
 //Parentheses must balanced
 while (counter > 0)
  {
   String as = "" + tree.charAt(position);
   if (as.equals(pr))   counter++;
   if (as.equals(pl))  counter--;
   S = as + S;
   if (print2)
   System.out.println("Subtree = " + S);
   position = position -1;
   }
  String n = innerNode;
```

```
 //Replacement antiRule 2:
 //((S)n,Fn)i -> (S)i
 //where A and F are leaves, n and i are  inner nodes,
 //(S) is a complex structure.
 String SnFni  =  "("+ S  + n + "," + leaf + n + ")" + i ;
 String Si = S + i;
 if (print2)
  System.out.println("\nSnFni = " + SnFni + "\n Si = " + Si);
 //Rule 1 is executed
 newTree = tree.replace(SnFni, Si);




 return newTree;
}



//The last inserted  leaf is deleted from tree.

private static String deleteLeaf(String tree)
{
  //Leaf to be deleted
  String leaf = "" + leaves.charAt(nextLeafIndex);
  System.out.println("\nLeaf to be deleted = " + leaf);
  //Accompanying inner node
  String innerNode = ""
            + internalNodes.charAt(nextIntNodeIndex);
  System.out.println("Accompanying inner  node = "
                    + innerNode);
  //new tree = output
  String newTree = "";
  //length of  tree as string
  int l = tree.length();
  //Last positions are found
  int positionLeaf = -1;
  int positionNode = -1;
```

```
  //tree is scanned char by char
  for(int i = 0; i < l; i++ )
   {
 String c = "" + tree.charAt(i);
 if (c.equals(leaf)) positionLeaf = i;
 if (c.equals(innerNode)) positionNode = i;
   }
  System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
  System.out.println("Position of " + innerNode +
          " = " + positionNode);
  int firstPosition = tree.indexOf(innerNode);
  //Check for the char before firstPosition
  char a = tree.charAt(firstPosition - 1);
  //If I links to a complex structure
  if (a == ')')
  newTree = antiRule2(tree,positionLeaf,positionNode);
  else  //Apply antiRule1 for simple cases
  newTree = antiRule1(tree,positionLeaf,positionNode);
  return newTree;
}


public static void main(String[] args)
{
 //A tree is generated
 System.out.println("Parent tree = \n" + parent
 + " -> "    + simplify(parent));
 //Number of leaves
 int nLeaves = 13;
 for(int i = 0; i < nLeaves; i++ )
  {
   parent = generateChild(parent);
   if (i == nLeaves -2) expected = parent;
  }
 //Last inserted leaf is deleted.
 String answer = deleteLeaf(parent);
 System.out.println("\nExpected and actual answer = \n"
```

```
    + expected + "\n" + answer);
 if (answer.equals(expected))
  System.out.println("Answer is correct.");
  else
  System.out.println("\nAnswer is wrong!");
}

}//End of main class K15 DeleteLeaf2
```

18, page 25. The next code implements mutation over trees with the possibility of changing topology and/or lengths of links.

```
//Program K18 TopoMutation2.
//This program generates a random binary tree
//and produces a mutation, whose output must be
//a binary tree with the same set of leaves.

//Tree = topology (graph) + lengths of links

//A mutation results from deleting
//a given number of leaves
//to reinsert them anew in
//other places chosen at random.
//This creates a mutant topology that
//includes mutation of lengths of links.
//Lengths of links can also be mutated separately.


import java.util.Random;


public class TopoMutation2
{

 //Link's notation for topologies
 private static   String   parent =
 "((Aa,Ba),Ca)";
```

```
 private static  String leaves =
 "ABCDEFGHIJKLMNOPRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;


 //Number of leaves to be deleted and reinserted
 //to create a mutation
 private static int nUndoRedo = 0;


 private static  Random r = new Random();
 //The control parameter of the random generator for int
 private static int maxLength = 10;

//Mutation rate in length of link per link
//per generation
static private double    m = 0.5;


 private static boolean print1 = false;
 private static boolean print2 = false;

 //This is the original topology:
 private static  String original = "";

 //============Size of topology ================
 //Number of leaves to be inserted to parent
 private static  int nAdditionalLeaves = 1;
 //=========================================

 //Total number of leaves in all trees
 private static  int numberOfLeaves =
                       3 + nAdditionalLeaves;
```

```
 //Detect internal nodes, different than leaves
 private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
  else return false;
 }

 //Length of the link that glues a leaf to the tree
 private static int lengthsLeaves[]  =
                new int[10];
 //Length of the link that glues
 //an inner node to another one
 private static int lengthsInnerLinks[]  =
               new int[10];

//Simplification from Link notation to Newick's notation:
 // (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTopo)
{
 String topo = "";
//length of string newTopo
int l = newTopo.length();
 for(int i = 0; i < l; i++ )
  {
   //newTopo is scanned char by char
  char c = newTopo.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    topo = topo + c;
  }
 return topo;
}
```

```
//Rule 2 is used to generate a new topology:
//(S)I -> ((S)N,FN)I.
//S = subtopology, I = internal node, F = new leaf,
//N = new internal node.
//Rule 2 grafts new branch FN to link (S)I
//i is the position of char I in string parentTopo
private static String useRule2(String parentTopo, int i,
        char F, char N)
{
 String newTopo = "";
 char I = parentTopo.charAt(i);
 if (print1)
 System.out.println("i = " + i + " I = " + I);
 //Subtopology S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = i;
 boolean go = true;
 //All right parentheses to the near left of I are captured
 while (go)
  {
   position = position -1;
   String a = "" + parentTopo.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
   else go = false;
  }//End of while
```

```
 if (print1)
 System.out.println("counter = " + counter);
 //The complete subtopology is captured:
 //Parentheses must balanced
 while (counter > 0)
  {
   String a = "" + parentTopo.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
   System.out.println("Subtopo = " + S);
   position = position -1;
   }
 //Replacement rule 2:
 //((S)n,Fn)i -> (S)i
//where   F are leaves, n and i are  inner nodes,
//(S) is a complex structure.
 String SnFni  =   S  + I;
 String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
 //Rule 1 is executed
 newTopo = parentTopo.replace(SnFni, replacement);
 //If I links to a complex structure
 return newTopo;
}

//From parentTopo, a topology in Newick's notation,
//a Child  topology is generated by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTopo)
{

 nextLeafIndex++;
 char nextLeaf= leaves.charAt(nextLeafIndex);
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = internalNodes.charAt(nextIntNodeIndex);
```

```
 //link to be replaced
 String link = "";
 //String link is replaced by what:
 String replacement = "";
 //new topology generated by replacement
 String newTopo = "";
 //length of parent topology
 int l = parentTopo.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
  {
   //parentTopo is scanned char by char
   char I = parentTopo.charAt(i);
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
  }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
  {
    //parentTopo is scanned char by char
char I = parentTopo.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = parentTopo.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTopo = useRule2(parentTopo, i,
```

```
              nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTopo = parentTopo.replace(link, replacement);
 }
if (print2)
System.out.println(newTopo + " -> "
                      + simplify(newTopo));


 i = l;
   }
  counter++;
}
  }
 return newTopo;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply the antirule1:
//(An,Fn)i  -> Ai
private static String  antiRule1(String topo,
int positionLeaf, int positionNode)
{
 if (print2)
  System.out.println("AntiRule1");
 int beginIndex = positionLeaf - 4;
 int endIndex = positionNode + 3;
 String AnFni = topo.substring(beginIndex, endIndex);
```

```
 if (print2)
  System.out.println("Anfni = " + AnFni);
 String Ai = "" + topo.charAt(beginIndex+1)
                + topo.charAt(endIndex-1);
 if (print2)
  System.out.println("Ai = " + Ai);
 topo = topo.replace(AnFni, Ai);
return topo;
}



//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply the antirule2:
//((S)n,Fn)i -> (S)i
//positionNode refers to the last appearing of node n
private static String  antiRule2(String topo,
int positionLeaf, int positionNode)
{
 if (print2)
System.out.println("AntiRule2");
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 if (print2)
  System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
           + internalNodes.charAt(nextIntNodeIndex);
 int firstPosition = topo.indexOf(innerNode);
 String newTopo = "";
 char i = topo.charAt(positionNode+2);
 if (print2)
 System.out.println("FirstPosition = " +
                     firstPosition + " i = " + i);
 //Subtopology S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
```

```
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = firstPosition;
 boolean go = true;
 //All right parentheses to the near left of I are captured
 while (go)
  {
   position = position -1;
    String as = "" + topo.charAt(position);

   if (print2)
   System.out.println("a = " + as + " p = " + pr);
   if (as.equals(pr) )
   {
counter++;
S = as + S;
   }
   else go = false;
  }//End of while
 if (print2)
 System.out.println("counter = " + counter);
 //The complete subtopology is captured:
 //Parentheses must balanced
 while (counter > 0)
  {
   String as = "" + topo.charAt(position);
   if (as.equals(pr))   counter++;
   if (as.equals(pl))  counter--;
   S = as + S;
   if (print2)
   System.out.println("Subtopo = " + S);
   position = position -1;
   }
  String n = innerNode;
 //Replacement antiRule 2:
 //((S)n,Fn)i -> (S)i
```

```
 //where A and F are leaves, n and i are  inner nodes,
 //(S) is a complex structure.
 String SnFni  =  "("+ S  + n + "," + leaf + n + ")" + i ;
 String Si = S + i;
 if (print2)
  System.out.println("\nSnFni = " + SnFni + "\n Si = " + Si);
 //Rule 1 is executed
 newTopo = topo.replace(SnFni, Si);
 return newTopo;
}



//The last inserted  leaf is deleted from topology.
private static String deleteLastLeaf(String topology)
{
  //Leaf to be deleted
  String leaf = "" + leaves.charAt(nextLeafIndex);
  if (print2)
  System.out.println("\nLeaf to be deleted = " + leaf);
  //Accompanying inner node
  String innerNode = ""
            + internalNodes.charAt(nextIntNodeIndex);
  if (print2)
  System.out.println("Accompanying inner  node = "
                    + innerNode);
  //new topology = output
  String newtopology = "";
  //length of  topology as string
  int l = topology.length();
  //Last positions are found
  int positionLeaf = -1;
  int positionNode = -1;
  //topology is scanned char by char
  for(int i = 0; i < l; i++ )
   {
 String c = "" + topology.charAt(i);
 if (c.equals(leaf)) positionLeaf = i;
```

```
 if (c.equals(innerNode)) positionNode = i;
   }
  if (print2)
   {
    System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
    System.out.println("Position of " + innerNode +
          " = " + positionNode);
   }
  int firstPosition = topology.indexOf(innerNode);
  //Check for the char before firstPosition
  char a = topology.charAt(firstPosition - 1);
  //If I links to a complex structure
  if (a == ')')
  newtopology = antiRule2(topology,positionLeaf,positionNode);
  else  //Apply antiRule1 for simple cases
  newtopology = antiRule1(topology,positionLeaf,positionNode);
  nextLeafIndex--;
  nextIntNodeIndex--;
  return newtopology;
}



//==========================

//Lengths are deleted from newTree
private static String deleteLengths(String newTree)
{
 String tree = "";
 String numeric = ".:0123456789";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
 {
  //newTree is scanned char by char
  char c = newTree.charAt(i);
  String s = "" + c;
```

```
  //Is c numeric: delete it!
  if (numeric.contains(s)); //add char c to output
  else tree = tree + c;
 }
 return tree;
}
```

```
//Mutation:
//some leaves are deleted and then reinserted
//in random order.
//Deleting can happen in either of two ways:
//Reversing the order of synthesis else
//in random order.
public static String mutation(String topology,
                        int nLeavesToBeInserted)
{
 //Mutation has two stages:
 //deletion of some leaves that undoes generating work +
 //re-insertion in places chosen at random.

 //Number of leaves to be deleted and then reinserted
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
  else
   nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;
 //nUndoRedo = 13;


 if (print2)
  System.out.println("Number of leaves to" +
  " be deleted and then reinserted = " +
  nUndoRedo);

 //Some leaves are deleted:
 //reversed order to that of synthesis.
```

```
 if (print1)
  System.out.println("Some leaves are deleted");
  for(int i = 0; i < nUndoRedo; i++ )
 {
  //Last inserted leaf is deleted.
  topology = deleteLastLeaf(topology);
  if (print1)
   System.out.println(topology);
 }
 //deleted leaves are reinserted
 if (print1)
  System.out.println("Deleted leaves are reinserted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
  topology = generateChild(topology);
  }



 //Output
 String mutant = topology;
 if (print1)
  System.out.println("\nOriginal and mutant topologies = \n"
  + original + "\n" + mutant);
 if (print1)
 if (original.equals(mutant))
  System.out.println("Mutation made nothing");
   else
  System.out.println("\nMutation worked");
 return mutant;
}


public static String mutation(String topology)
{
 //This original topology is next subject to mutation
 original = topology;
 String mutantTopology = topology;
```

```
 if (print2)
System.out.println("\nOriginal   topology = \n"
                + original  );
 //Number of leaves to be deleted and then reinserted
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
else
 nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;
 //nUndoRedo = 13;

 //Mutation has two stages:
 //deletion of some leaves that undoes generating work +
 //re-insertion in places chosen at random.

 if (print2)
  System.out.println("Number of leaves to" +
   " be deleted and then reinserted = " +
   nUndoRedo);
 //Some leaves are deleted:
 if (print2)
 System.out.println("Some leaves are deleted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
//Last inserted leaf is deleted.
mutantTopology = deleteLastLeaf(mutantTopology);
if (print2)
   System.out.println(mutantTopology);
  }
 //Some leaves are reinserted
 if (print2)
 System.out.println("Deleted leaves are reinserted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
   topology = generateChild(topology);
  }
 return mutantTopology;
}
```

```
//Random lengths are added to a tree in Link notation
public static String insertLengths(String treeLinkN)
{
 String thisChar = "";
 String newTree = "";
 //length of String   tree
 int m = treeLinkN.length();
 //TreeLinkN is scanned char by char
 for(int i = 0; i < m; i++ )
 {
  thisChar = "" + treeLinkN.charAt(i);
  newTree = newTree + thisChar;
  if (print1)
   System.out.println(newTree.length() + " " + newTree);
  //Lengths are appended
  if (leaves.contains(thisChar))
   {
    int k = leaves.indexOf(thisChar);
    double j = lengthsLeaves[k];
    newTree = newTree + ":" + j;
   }
 if (internalNodes.contains(thisChar))
  {
   int k = internalNodes.indexOf(thisChar);
   double j = lengthsInnerLinks[k];
   newTree = newTree + ":" + j;
 }
}
return newTree;
}


//Input: string tree with complete information.
//Work: lengthsLeaves[] and lengthsInnerLinks[]
//are read from tree.
private static void decoderLengths(String tree)
{
```

```
 if(print2)
 System.out.println(
"Input tree at decoverLengths = \n "   + tree  );
 //Delete :
 tree = tree.replaceAll(":", "");
 if(print2)
 System.out.println(
"Input tree without : = \n "   + tree  );
 int n = tree.length();
 String thisChar = "";
 String nextChar = "";
 String numbers = ".0123456789";
 boolean leaf = false;
 boolean node = false;
 String Leaf = "";
 String numberLeaf = "";
 String numberNode = "";
 String Node = "";
 int j ;
 double f;

 //All chars in tree are scanned
 for(int i=0; i < n; i++)
  {
   thisChar = "" + tree.charAt(i);
   if (i < n-1) nextChar  = "" + tree.charAt(i+1);
   //Do we have a leaf?
   if (leaves.contains(thisChar))
    {
 leaf = true;
 Leaf = "" + thisChar;
    }

   //Do we have an internal node?
   if (internalNodes.contains(thisChar))
    {
 node = true;
 Node = "" + thisChar;
```

```
    }

   //Do we have a number that belongs in a leaf?
   if ( (leaf) & (numbers.contains(thisChar)))
{
 numberLeaf = numberLeaf + thisChar;
}

   //Do we have a number that belongs in a leaf?
   if ( (node) & (numbers.contains(thisChar)))
     {
  numberNode = numberNode + thisChar;
 }

   //Must we end accumulation of numbers of a leaf?
   if ( (leaf) & (internalNodes.contains(nextChar)))
 {
  leaf = false;
  j = leaves.indexOf(Leaf);
  if (print1)
  System.out.println("Leaf = " + Leaf +
  " numberLeaf = " + numberLeaf);
  f = new Double(numberLeaf);
  lengthsLeaves[j] = (int) Math.floor(f);
  numberLeaf = "";
 }

   //Must we end accumulation of numbers of a node?
   if ( (node) & (!(numbers.contains(nextChar))))
 {
  node = false;
  j = internalNodes.indexOf(Node);
  if (print1)
  System.out.println("Node = " + Node);
  f = new Double(numberNode);
  lengthsInnerLinks[j] = (int) Math.floor(f);
  numberNode = "";
 }
```

```
  }
}

//Random lengths are added to a tree in Link notation
public static void generateLengths()
{
 //Lengths with digit values are generated
 for(int i = 0; i < numberOfLeaves; i++)
  {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
  }
}


//Lengths undergo mutation
private static  void mutateLengths()
{

  //Mutation in links that connect leaves
  for(int k = 0; k < numberOfLeaves; k++)
   {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsLeaves[k] = r.nextInt(maxLength);

   }
  //Mutation in links that connect inner nodes
  for(int k = 0; k < numberOfLeaves-2; k++)
   {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsInnerLinks[k] = r.nextInt(maxLength);

   }
```

```
}


//Type of  mutation is decided and executed:
//mutation of lengths of links
//else full mutation (topology and lengths of links).
private static String mutateTree(String tree)
{
 //Topology is read from tree
 String topology = deleteLengths(tree);
 //Lengths are read from tree and recorded in vectors
 //lengthsLeaves[] and lengthsInnerLinks[]
 decoderLengths(tree);
 mutateLengths();
 String  mutantTree = "";
 //Mutate lengths else topology ( + new lengths)?
 double s = r.nextDouble();
 if ( s < 0)
{
 if (print1)
 System.out.println("\n====Mutation of lengths");

  mutantTree =  insertLengths(topology);

  if (print1)
  {
  System.out.println("Input tree =  " + tree);
  System.out.println("Topology =  " + topology);
  System.out.println("Output tree= " + mutantTree);
  }
 }
else
 {
 if (print1)
  System.out.println("\n====Mutation of topology");
```

```
//Mutation of topology
  String mutantTopology = mutation(topology,
                             nAdditionalLeaves);
  //Mutant lengths are reinserted
  mutantTree =  insertLengths(mutantTopology);
 }

 return mutantTree;
}



/**
 * @param args
 */
public static void main(String[] args)
{
 //A topology is generated
 if (print2)
  System.out.println("Parent topology = \n" + parent
 + " -> "    + simplify(parent));
 String topology = parent;

 for(int i = 0; i < nAdditionalLeaves; i++ )
   topology = generateChild(topology);
 //This original topology is next subject to mutation
 original = topology;

 //Lengths are added to links.
 //They are  recorded in vectors
 //lengthsLeaves[] and lengthsInnerLinks[]
 generateLengths();
 String originalTree = insertLengths(original);
 if (print2)
  System.out.println("\nOriginal   topology = \n"
                + original  +
```

```
                "\nOriginal  tree = \n"
                + originalTree  );
 //Mutation is executed
 String mutantTree = mutateTree(originalTree);

 //Original and mutant trees are compared.
 System.out.println("\nOriginal and mutant trees = \n"
  + originalTree + "\n" + mutantTree);
 if (originalTree.equals(mutantTree))
  System.out.println(
  "Original and mutant trees coincide");
 else
  System.out.println(
  "\nOriginal and mutant trees are different");
}

}//End of main class K18 TopoMutation2
```

24, page 27. List all subtopologies of
((Aa,Ba),((Cb,Db)c,((Ee,Ge)d,Fd)c)a):

(Aa,Ba)
(Cb,Db)
(Ee,Ge)
((Ee,Ge)d,Fd)
((Cb,Db)c,((Ee,Ge)d,Fd)c)

25, page 27. The Author does not know. We will proceed by instinct to next see whether or not one can go further without troubles.

28, page 34. The next code generates all subtopologies of a given topology and pack them in an array.

```
//Program K28 SubTopol2
/*
 * This program list all possible subtopologies
 * of a given topology.
```

```
 *
 * We begin with a left parenthesis "(" and cumulate
 * symbols until the number of
 * right and left parentheses get balanced.
 *
 * All left parenthesis are investigated.
 *
 * Subtopologies are packed for future use.
 *
 */

import java.util.Random;

public class SubTopol2
{

//Link's notation for trees
 private static   String   parent =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;

 //Leaves that remain in the tree
 private static  String presentLeaves =   "ABC";
 private static String presentNodes =  "a";


 private static  Random r = new Random();

 private static boolean print1 = false;
 private static boolean print2 = false;
```

```
//====== A random topology is generated =============

//Detect internal nodes, different than leaves
private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
   else return false;
 }


//Simplification from Link notation to Newick's notation:
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTree)
 {
  String tree = "";
  //length of string newTree
  int l = newTree.length();
  for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
  }
 return tree;
 }

 //Rule 2 is used to generate a new tree:
 //(S)I -> ((S)N,FN)I.
 //S = subtree, I = internal node, F = new leaf,
 //N = new internal node.
 //Rule 2 grafts new branch FN to link (S)I
 //i is the position of char I in string parentTree
```

```
 private static String useRule2(String parentTree, int i,
        char F, char N)
 {
  String newTree = "";
  char I = parentTree.charAt(i);
  if (print1)
  System.out.println("i = " + i + " I = " + I);
  //Subtree S of link (S)I must be read
  String S = "";
  //Counter of left parenthesis
  int counter = 0;
  String pl = "" + '(';
  String pr = "" + ')';
  //Position to be scanned
  int position = i;
  boolean go = true;
  //All right parentheses to the near left of I are captured
  while (go)
  {
   position = position -1;
   String a = "" + parentTree.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
   else go = false;
  }//End of while
  if (print1)
 System.out.println("counter = " + counter);
  //The complete subtree is captured:
  //Parentheses must balanced
  while (counter > 0)
  {
   String a = "" + parentTree.charAt(position);
```

```
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
   System.out.println("Subtree = " + S);
   position = position -1;
   }
  //Replacement rule 2:
  //((S)n,Fn)i -> (S)i
  //where   F are leaves, n and i are  inner nodes,
  //(S) is a complex structure.
  String SnFni  =   S  + I;
  String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
  //Rule 1 is executed
  newTree = parentTree.replace(SnFni, replacement);
  //If I links to a complex structure
  return newTree;
 }


//Input: parentTree, a tree in Newick's notation.
//Output : newTree =  parent tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTree)
 {
  nextLeafIndex++;
  char nextLeaf = leaves.charAt(nextLeafIndex);
  presentLeaves = presentLeaves + nextLeaf;
  //Next internal node to be inserted
  nextIntNodeIndex++;
  char nextIntNode = internalNodes.charAt(nextIntNodeIndex);
  presentNodes = presentNodes + nextIntNode;

  //link to be replaced
  String link = "";
  //String link is replaced by what:
```

```
  String replacement = "";
  //new tree generated by replacement
  String newTree = "";
  //length of parent tree
  int l = parentTree.length();
  //counter of links = 0;
  int counter = 0;
  //Links are counted
  for(int i = 0; i < l; i++ )
  {
   //parentTree is scanned char by char
   char I = parentTree.charAt(i);
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
  }
  //A link is chosen
  int k = r.nextInt(counter);
  //System.out.println("Random number = " + k);
  //new branch is grafted at k
  counter = 0;
  for(int i = 0; i < l; i++ )
   {
//parentTree is scanned char by char
char I = parentTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
  {
//Check for the char before I
char a = parentTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(parentTree, i,
               nextLeaf, nextIntNode);
 else //if I links to a leaf, use rule 1
  {
   //Link to be replaced
   link = "" + a + I;
```

```
   //Replacement rule 1:
   //aI -> (aN,FN)I;   N= nextIntNode;
   //F = nextLeaf
  replacement = "(" + a + nextIntNode + ","
      + nextLeaf + nextIntNode + ")" + I;
   //Rule 1 is executed
   newTree = parentTree.replace(link, replacement);
  }
if (print1)
 System.out.println(newTree + " -> "
                      + simplify(newTree));

 i = l;
}
   counter++;
  }
 }
    return newTree;
   }


//========== Subtopologies are listed ============


//Subtopologies of  topology are found
public static String[] subtopologies(String topology)
{
 System.out.println("Subtopologies are found");
 //Counter of left parenthesis
 int counter = 0;
 //Counter of subtopologies
 int counterSubtop = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Length of String   topology
 int m = topology.length();
 String[] subtopologies = new String[m];
 //String topology is scanned char by char
```

```
 for(int i = 0; i < m; i++ )
  {
   String Sub = "";
   String a = "" + topology.charAt(i);
   int position = i+1;
   if (a.equals(pl))
   {
counter++;
 Sub =  Sub + a;
//The complete subtopology is captured:
//Parentheses must balanced
while (counter > 0)
 {
  String b = "" + topology.charAt(position);
  if (b.equals(pl))   counter++;
  if (b.equals(pr))  counter--;
  Sub =  Sub + b;

  position++;
 }
 if (print2)
   System.out.println(Sub);
   subtopologies[counterSubtop] = Sub;
   counterSubtop++;
  }
 }
 return subtopologies;
}



 public static void main(String[] args)
 {
   System.out.println("A random topology is generated");
   //Seed
   if (print1)
   System.out.println("\n" + parent
 + " -> "    + simplify(parent));
```

```
   //A tree topology is generated
   String topology = parent;
   //Number of leaves to be inserted
   int nLeavesToBeInserted = 6;
   for(int i = 0; i < nLeavesToBeInserted; i++ )
   topology = generateChild(topology);
   System.out.println(topology + " -> "
          + simplify(topology));
   //Length of String   topology
   int m = topology.length();
   String[] subtopologies = new String[m];
   //Subtopologies are found
   subtopologies = subtopologies(topology);
   for(int i = 0; i < m; i++)
    {
   if (subtopologies[i] == null);
   else
   System.out.println(i + " " + subtopologies[i]);
    }
 }
}//End of main class K28  SubTopol2
```

31, page 46. The next code generates two topologies and automatically decides when there are subtopologies whose set of leaves coincide. The same decision is taken for the corresponding sets of inner nodes. Playing with the program allowed us to conclude that apart from the original, complete topologies, rarely other subtopologies have the same set of leaves but when that happened the set of inner nodes were also equal. Besides, the probability of finding possible recombinant subtopologies diminishes with the number of leaves of the topology. Therefore, we expect not too much from recombination: topologies come in species that in general are not recombination-prone. Nevertheless, only experiment can assess how important is the role of recombination of topologies to accelerate the rate of evolution.

```
//Program K31 SubTopol4
```

```
/*
 * This program generates two topologies  on the same
 * set of leaves.
 * All their subtopologies are calculated
 * and packed in arrays.
 * Sets of leaves and inner nodes of each subtopology
 * are found.
 *
 *
 */

import java.util.Random;

public class SubTopol4
{

//Link's notation for trees
 private static   String   parent =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;

 //===========Size of the tree===========
 //Number of leaves to be inserted
 private static  int nLeavesToBeInserted = 20;
 //=====================================


 //Counter of subtopologies
 private static  int counterSubtop;
 //Particular number of subtopologies
 private static  int m1 = 0;
```

```
 private static  int m2 = 0;

 //Leaves that remain in the tree
 private static  String presentLeaves =   "ABC";
 private static String presentNodes =  "a";


 private static  Random r = new Random();

 private static boolean print1 = false;
 private static boolean print2 = false;



//====== A random topology is generated =============

//Detect internal nodes, different than leaves
private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
   else return false;
 }


//Simplification from Link notation to Newick's notation:
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTree)
 {
  String tree = "";
  //length of string newTree
  int l = newTree.length();
  for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
```

```
   else //add char c to output
    tree = tree + c;
  }
 return tree;
 }

 //Rule 2 is used to generate a new tree:
 //(S)I -> ((S)N,FN)I.
 //S = subtree, I = internal node, F = new leaf,
 //N = new internal node.
 //Rule 2 grafts new branch FN to link (S)I
 //i is the position of char I in string parentTree
 private static String useRule2(String parentTree,
 int i,   char F, char N)
 {
  String newTree = "";
  char I = parentTree.charAt(i);
  if (print1)
  System.out.println("i = " + i + " I = " + I);
  //Subtree S of link (S)I must be read
  String S = "";
  //Counter of left parenthesis
  int counter = 0;
  String pl = "" + '(';
  String pr = "" + ')';
  //Position to be scanned
  int position = i;
  boolean go = true;
  //All right parentheses to the near left
  //of I are captured
  while (go)
  {
   position = position -1;
   String a = "" + parentTree.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
```

```
   {
counter++;
S = a + S;
   }
   else go = false;
  }//End of while
  if (print1)
 System.out.println("counter = " + counter);
  //The complete subtree is captured:
  //Parentheses must balanced
  while (counter > 0)
  {
   String a = "" + parentTree.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
   System.out.println("Subtree = " + S);
   position = position -1;
   }
  //Replacement rule 2:
  //((S)n,Fn)i -> (S)i
  //where   F are leaves, n and i are  inner nodes,
  //(S) is a complex structure.
  String SnFni  =   S  + I;
  String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
  //Rule 1 is executed
  newTree = parentTree.replace(SnFni, replacement);
  //If I links to a complex structure
  return newTree;
 }


//Input: parentTree, a tree in Newick's notation.
//Output : newTree =  parent tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
```

```
private static String generateChild(String parentTree)
 {
  nextLeafIndex++;
  char nextLeaf = leaves.charAt(nextLeafIndex);
  presentLeaves = presentLeaves + nextLeaf;
  //Next internal node to be inserted
  nextIntNodeIndex++;
  char nextIntNode =
  internalNodes.charAt(nextIntNodeIndex);
  presentNodes = presentNodes + nextIntNode;

  //link to be replaced
  String link = "";
  //String link is replaced by what:
  String replacement = "";
  //new tree generated by replacement
  String newTree = "";
  //length of parent tree
  int l = parentTree.length();
  //counter of links = 0;
  int counter = 0;
  //Links are counted
  for(int i = 0; i < l; i++ )
  {
   //parentTree is scanned char by char
   char I = parentTree.charAt(i);
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
  }
  //A link is chosen
  int k = r.nextInt(counter);
  //System.out.println("Random number = " + k);
  //new branch is grafted at k
  counter = 0;
  for(int i = 0; i < l; i++ )
   {
//parentTree is scanned char by char
char I = parentTree.charAt(i);
```

```
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
  {
//Check for the char before I
char a = parentTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(parentTree, i,
         nextLeaf, nextIntNode);
 else //if I links to a leaf, use rule 1
  {
   //Link to be replaced
   link = "" + a + I;
   //Replacement rule 1:
   //aI -> (aN,FN)I;   N= nextIntNode;
   //F = nextLeaf
  replacement = "(" + a + nextIntNode + ","
       + nextLeaf + nextIntNode + ")" + I;
   //Rule 1 is executed
   newTree = parentTree.replace(link, replacement);
  }
if (print1)
 System.out.println(newTree + " -> "
                     + simplify(newTree));

 i = l;
}
   counter++;
  }
 }
    return newTree;
   }


//========== Subtopologies are listed ============
```

```
//Subtopologies of  topology are found
public static String[] subtopologies(String topology)
{
 if (print1)
System.out.println("\nSubtopologies are found");
 //Counter of left parenthesis
 int counter = 0;
 //Counter of subtopologies
 counterSubtop = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Length of String   topology
 int m = topology.length();
 String[] subtopologies = new String[m];
 //String topology is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   String Sub = "";
   String a = "" + topology.charAt(i);
   int position = i+1;
   if (a.equals(pl))
   {
counter++;
 Sub =  Sub + a;
//The complete subtopology is captured:
//Parentheses must balanced
while (counter > 0)
 {
  String b = "" + topology.charAt(position);
  if (b.equals(pl))   counter++;
  if (b.equals(pr))  counter--;
  Sub =  Sub + b;
  position++;
 }
 if (print2)
   System.out.println(Sub);
   subtopologies[counterSubtop] = Sub;
   counterSubtop++;
```

```
  }
 }
 return subtopologies;
}



 public static String generateTopology()
 {
 System.out.println("A random topology is generated");
   //Seed
   if (print1)
   System.out.println("\n" + parent
 + " -> "    + simplify(parent));
 //Next leaf to be grafted
 nextLeafIndex = 2;
 //Next internal node to be inserted
 nextIntNodeIndex = 0;
   //A tree topology is generated
   String topology = parent;
   for(int i = 0; i < nLeavesToBeInserted; i++ )
   topology = generateChild(topology);
   return topology;
 }



 //Leaves of subtopologies  are found
  public static String[]leaves(int m, String[] subtop)
 {
  if (print1)
   System.out.println("Leaves are found");
  String[] myLeaves = new String[m];
  for(int i = 0; i < m; i++)
   {
 String s = subtop[i];
 if (s == null);
```

```
   else
   {
    myLeaves[i] = "";
    int n = s.length();
    for(int j = 0; j < n; j++)
     {
      String c = "" + subtop[i].charAt(j);
      if (leaves.contains(c))
       myLeaves[i] = myLeaves[i] + c;
     }
    if (print1)
    System.out.println(subtop[i] + " "
                           + myLeaves[i] );
   }
    }
  //Chars of Leaves are rewritten in alphabet order
  if (print1)
  System.out.println("Leaves are listed orderly" );
  for(int i = 0; i < m; i++)
  {
 String s = subtop[i];
 if (s == null);
   else
   {
     s =   myLeaves[i];
     myLeaves[i] = "";
    int n = leaves.length();
    for(int j = 0; j < n; j++)
     {
      String c = "" + leaves.charAt(j);
      if (s.contains(c))
       myLeaves[i] = myLeaves[i] + c;
     }
    if (print1)
    System.out.println(subtop[i] + " " + myLeaves[i] );
   }
   }
  return myLeaves;
```

```
 }


//Inner nodes of subtopologies  are found
 public static String[] inner(int m, String[] subtop)
{
 if (print1)
 System.out.println("\nInternal nodes are found");
 String[] myNodes = new String[m];
 for(int i = 0; i < m; i++)
  {
 String s = subtop[i];
 if (s == null);
   else
   {
  myNodes[i] = "";
    int n = s.length();
    for(int j = 0; j < n; j++)
     {
      String c = "" + subtop[i].charAt(j);
      if (internalNodes.contains(c))
      myNodes[i] = myNodes[i] + c;
     }
   if (print1)
System.out.println(subtop[i] + " " + myNodes[i] );
   }
   }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
 System.out.println("Inner nodes are listed orderly" );
 for(int i = 0; i < m; i++)
 {
 String s = subtop[i];
 if (s == null);
   else
   {
     s =   myNodes[i];
```

```
      myNodes[i] = "";
      int n = internalNodes.length();
      for(int j = 0; j < n; j++)
       {
        String c = "" + internalNodes.charAt(j);
        if (s.contains(c))
        myNodes[i] = myNodes[i] + c;
       }
     if (print1)
      System.out.println(subtop[i] + " " + myNodes[i] );
     }
    }
   return myNodes;
    }


 //Subtopologies are compared to decide whether
 //or not they have the same set of leaves
 //and the same set of inner nodes.

public static void comparison(String[] Leaves1,
                      String[] Leaves2,
                      String[] Inner1,
                      String[] Inner2)
 {
  System.out.println("\nComparison of leaves and nodes");
  //Print everything
  Boolean printAll = false;
  int m = Leaves1.length;
  int n = Leaves2.length;
  for(int i = 0; i < m; i++)
  {
   for(int j = 0; j < n; j++)
   {
if ( Leaves1[i].equals(Leaves2[j]))
{
 System.out.println(
 "Sets of leaves are EQUAL " + i +
```

```
 "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
 if ( Inner1[i].equals(Inner2[j]))
System.out.println(
        "Sets of inner nodes are EQUAL " + i +
    "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
  else
System.out.println(
        "Sets of inner nodes are DIFFERENT " + i +
"  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
}
else
if (printAll)
System.out.println(
        "Sets of leaves are DIFFERENT " + i +
   "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
   }
  }
 }


 public static void main(String[] args)
 {
   //A topology is generated
   String topo1 = generateTopology();
   System.out.println(topo1 + " -> "
          + simplify(topo1));
   //Length of String   topology
   m1 = topo1.length();
   //Subtopologies are found
   String[] subtopologies1 = new String[m1];
   subtopologies1 = subtopologies(topo1);
   m1 = counterSubtop;
   /*
   String[] Nodes1 = new String[m1];
   innerNodes(subtopologies1); */

   System.out.println();
   String[] Leaves1 = new String[m1];
```

```
   Leaves1 = leaves(m1,subtopologies1);
   String[] Inner1 = new String[m1];
   Inner1 = inner(m1,subtopologies1);
   //Chars of Leaves are rewritten in alphabet order
   System.out.println("\nSubtopologies, Leaves, " +
                 "Inner nodes" );
   for(int i = 0; i < m1; i++)
   {
 if (subtopologies1[i] == null);
  else
  {

   System.out.println(i + "   " + subtopologies1[i]
   + "   " +  Leaves1[i] + "   " + Inner1[i]);
  }
   }

   System.out.println();

   //A second topology is generated

   String topo2 = generateTopology();
   System.out.println(topo2 + " -> "
          + simplify(topo2));
   //Length of String   topology
   m2 = topo2.length();
   //Subtopologies are found
   String[] subtopologies2 = new String[m2];
   subtopologies2 = subtopologies(topo2);
   m2 = counterSubtop;
   System.out.println();
   String[] Leaves2 = new String[m2];
   Leaves2 = leaves(m2,subtopologies2);
   String[] Inner2 = new String[m2];
   Inner2 = inner(m2,subtopologies2);
   //Chars of Leaves are rewritten in alphabet order
   System.out.println("\nSubtopologies, Leaves, " +
                 "Inner nodes" );
```

```
   for(int i = 0; i < m2; i++)
   {
 if (subtopologies1[i] == null);
  else
  {

   System.out.println(i + "   " + subtopologies2[i]
   + "   " +  Leaves2[i] + "   " + Inner2[i]);
  }
   }
  comparison(Leaves1,Leaves2, Inner1,Inner2);
 }
}//End of main class K31  SubTopol4
```

34, page 63. The next code selects at random a pair of subtrees that allow recombination and implements it.

```
//Program K34 SubTopol6
/*
 * Code for the
 * recombination of tree topologies:
 *
 * This program generates two topologies  on the same
 * set of leaves and adds  lengths to links
 * to get two trees.
 * All their subtopologies are calculated
 * and packed in arrays.
 * The sets of leaves and of inner nodes
 * of each subtopology are found.
 * Subtopologies are compared by pairs,
 * one from each topology.
 * If they happen to have the same set of leaves
 * and of inner nodes, recombination is authorized:
 * the  subtrees that correspond with
 * the subtopologies are swapped.
 *
 * If more than one pair of subtopologies authorize
```

```
 * recombination, one is chosen at random.
 * A hashtable is used to memorize those pairs that do it.
 * Original topologies serves as subtopologies
 * that authorize recombination but they do not count.
 *
 */


import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Random;

public class SubTopol6
{

 //Link's notation for trees
 private static   String   parent =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;

 //===========Size of the tree==========
 //Number of leaves to be inserted
 private static  int nLeavesToBeInserted = 3;
 //=====================================
 //Number of leaves that must go in all trees > 3
 private static int numberOfLeaves =
             3 + nLeavesToBeInserted;

 //Length of the link that glues a leaf to the tree
 private static int lengthsLeaves[]  =
                new int[numberOfLeaves];
```

```
//Length of the link that glues
//an inner node to another one
private static int lengthsInnerLinks[]  =
              new int[numberOfLeaves];

//Counter of subtopologies
private static  int counterSubtop;
//Particular number of subtopologies
private static  int m1 = 0;


//Leaves that remain in the tree
private static  String presentLeaves =   "ABC";
private static String presentNodes =  "a";


private static  Random r = new Random();
//The control parameter of the random generator for int
private static int maxLength = 10;

private static  String[] subtrees1, subtrees2;
private static  String tree1, tree2,
                       subTree1, subTree2,
                       treeRec1, treeRec2;

//Structure of hashtable : integer + {string1, string2}
private static Hashtable<Integer, String[]> hashtable =
new  Hashtable<Integer, String[]>();


private static boolean print1 = false;
private static boolean print2 = false;
private static boolean print3 = true;



//====== A random topology is generated =============
```

```
//Detect internal nodes, different than leaves
private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
   else return false;
 }


//Simplification from Link notation to Newick's notation:
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String linkToNewick(String newTree)
 {
  String tree = "";
  //length of string newTree
  int l = newTree.length();
  for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
  }
 return tree;
 }


//Complete simplification of subtrees
//(L:5.0m:3.0,O:6.0m:3.0) ->  (L,O)
//Output = subtopology
private static String simplify(String newTree)
{
String topology = "";
//The alphabet of topologies
String allowedChars = leaves + "(),";
//length of string newTree
```

```
int l = newTree.length();
for(int i = 0; i < l; i++ )
 {
  //newTree is scanned char by char
  String c = "" + newTree.charAt(i);
  //Is c is an allowed char, add it  to output
  if (allowedChars.contains(c))
   topology = topology + c;
 }
//System.out.println( newTree + "  simplify: " + topology);
return topology;
}



 //Rule 2 is used to generate a new tree:
 //(S)I -> ((S)N,FN)I.
 //S = subtree, I = internal node, F = new leaf,
 //N = new internal node.
 //Rule 2 grafts new branch FN to link (S)I
 //i is the position of char I in string parentTree
 private static String useRule2(String parentTree,
 int i,   char F, char N)
 {
  String newTree = "";
  char I = parentTree.charAt(i);
  if (print1)
  System.out.println("i = " + i + " I = " + I);
  //Subtree S of link (S)I must be read
  String S = "";
  //Counter of left parenthesis
  int counter = 0;
  String pl = "" + '(';
  String pr = "" + ')';
  //Position to be scanned
  int position = i;
  boolean go = true;
```

```
  //All right parentheses to the near left
  //of I are captured
  while (go)
  {
   position = position -1;
   String a = "" + parentTree.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
   else go = false;
  }//End of while
  if (print1)
 System.out.println("counter = " + counter);
  //The complete subtree is captured:
  //Parentheses must balanced
  while (counter > 0)
  {
   String a = "" + parentTree.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
   System.out.println("Subtree = " + S);
   position = position -1;
   }
  //Replacement rule 2:
  //((S)n,Fn)i -> (S)i
  //where   F are leaves, n and i are  inner nodes,
  //(S) is a complex structure.
  String SnFni  =   S  + I;
  String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
  //Rule 1 is executed
```

```
  newTree = parentTree.replace(SnFni, replacement);
  //If I links to a complex structure
  return newTree;
 }


//Input: parentTree, a tree in Newick's notation.
//Output : newTree =  parent tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTree)
 {
  nextLeafIndex++;
  char nextLeaf = leaves.charAt(nextLeafIndex);
  presentLeaves = presentLeaves + nextLeaf;
  //Next internal node to be inserted
  nextIntNodeIndex++;
  char nextIntNode =
  internalNodes.charAt(nextIntNodeIndex);
  presentNodes = presentNodes + nextIntNode;

  //link to be replaced
  String link = "";
  //String link is replaced by what:
  String replacement = "";
  //new tree generated by replacement
  String newTree = "";
  //length of parent tree
  int l = parentTree.length();
  //counter of links = 0;
  int counter = 0;
  //Links are counted
  for(int i = 0; i < l; i++ )
  {
   //parentTree is scanned char by char
   char I = parentTree.charAt(i);
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
```

```
  }
  //A link is chosen
  int k = r.nextInt(counter);
  //System.out.println("Random number = " + k);
  //new branch is grafted at k
  counter = 0;
  for(int i = 0; i < l; i++ )
   {
//parentTree is scanned char by char
char I = parentTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
  {
//Check for the char before I
char a = parentTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(parentTree, i,
         nextLeaf, nextIntNode);
 else //if I links to a leaf, use rule 1
  {
   //Link to be replaced
   link = "" + a + I;
   //Replacement rule 1:
   //aI -> (aN,FN)I;   N= nextIntNode;
   //F = nextLeaf
  replacement = "(" + a + nextIntNode + ","
       + nextLeaf + nextIntNode + ")" + I;
   //Rule 1 is executed
   newTree = parentTree.replace(link, replacement);
  }
if (print1)
 System.out.println(newTree + " -> "
                     + linkToNewick(newTree));

 i = l;
}
```

```
   counter++;
  }
 }
    return newTree;
   }

//===========Trees are synthesized==============


//Random lengths are added to a tree in Link notation
private static void makeLengths()
{
//Lengths with digit values are generated
for(int i = 0; i < numberOfLeaves; i++)
 {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
 }
}

//Random lengths are added to a tree in Link notation
private static String insertLengths(String treeLinkN)
{
String thisChar = "";
String newTree = "";
//length of String   tree
int m = treeLinkN.length();
//Tree is scanned char by char
for(int i = 0; i < m; i++ )
 {
  thisChar = "" + treeLinkN.charAt(i);
  newTree = newTree + thisChar;
  if (print1)
  System.out.println(newTree.length() + " " + newTree);
  //Lengths are appended
  if (leaves.contains(thisChar))
   {
```

```
    int k = leaves.indexOf(thisChar);
    double j = lengthsLeaves[k];
   newTree = newTree + ":" + j;
   }
  if (internalNodes.contains(thisChar))
  {
   int k = internalNodes.indexOf(thisChar);
   double j = lengthsInnerLinks[k];
   newTree = newTree + ":" + j;
  }
 }
 return newTree;
}


//========== Subtrees are listed ============


//Subtrees of  topology are found
private static String[] subtopologies(String topology)
{
 if (print1)
System.out.println("\nSubtrees are found");
 //Counter of left parenthesis
 int counter = 0;
 //Counter of subtopologies
 counterSubtop = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Length of String   topology
 int m = topology.length();
 String[] subtopologies = new String[m];
 //String topology is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   String Sub = "";
   String a = "" + topology.charAt(i);
   int position = i+1;
```

```
   if (a.equals(pl))
   {
counter++;
 Sub =  Sub + a;
//The complete subtopology is captured:
//Parentheses must balanced
while (counter > 0)
 {
  String b = "" + topology.charAt(position);
  if (b.equals(pl))   counter++;
  if (b.equals(pr))  counter--;
  Sub =  Sub + b;
  position++;
 }
 if (print2)
   System.out.println(Sub);
   subtopologies[counterSubtop] = Sub;
   counterSubtop++;
  }
 }
 return subtopologies;
}



private static String generateTopology()
 {
 System.out.println("A random topology is generated");
   //Seed
   if (print1)
   System.out.println("\n" + parent
 + " -> "    + linkToNewick(parent));
 //Next leaf to be grafted
 nextLeafIndex = 2;
 //Next internal node to be inserted
 nextIntNodeIndex = 0;
   //A tree topology is generated
```

```
  String topology = parent;
  for(int i = 0; i < nLeavesToBeInserted; i++ )
  topology = generateChild(topology);
  return topology;
}



//Leaves of subtopologies  are found
private static String[]leaves(int m, String[] subtop)
{
 if (print1)
  System.out.println("Leaves are found");
 String[] myLeaves = new String[m];
 for(int i = 0; i < m; i++)
  {
String s = subtop[i];
if (s == null);
  else
  {
   myLeaves[i] = "";
   int n = s.length();
   for(int j = 0; j < n; j++)
    {
     String c = "" + subtop[i].charAt(j);
     if (leaves.contains(c))
      myLeaves[i] = myLeaves[i] + c;
    }
   if (print1)
   System.out.println(subtop[i] + " "
                          + myLeaves[i] );
  }
   }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
 System.out.println("Leaves are listed orderly" );
 for(int i = 0; i < m; i++)
 {
```

```
String s = subtop[i];
if (s == null);
  else
  {
    s =   myLeaves[i];
    myLeaves[i] = "";
   int n = leaves.length();
   for(int j = 0; j < n; j++)
    {
     String c = "" + leaves.charAt(j);
     if (s.contains(c))
      myLeaves[i] = myLeaves[i] + c;
    }
   if (print1)
   System.out.println(subtop[i] + " " + myLeaves[i] );
  }
  }
 return myLeaves;
  }



 //Inner nodes of subtopologies  are found
 private static String[] inner(int m, String[] subtop)
 {
  if (print1)
  System.out.println("\nInternal nodes are found");
  String[] myNodes = new String[m];
  for(int i = 0; i < m; i++)
   {
  String s = subtop[i];
  if (s == null);
    else
    {
   myNodes[i] = "";
     int n = s.length();
     for(int j = 0; j < n; j++)
      {
```

```
        String c = "" + subtop[i].charAt(j);
        if (internalNodes.contains(c))
        myNodes[i] = myNodes[i] + c;
       }
     if (print1)
 System.out.println(subtop[i] + " " + myNodes[i] );
     }
     }
  //Chars of Leaves are rewritten in alphabet order
  if (print1)
  System.out.println("Inner nodes are listed orderly" );
  for(int i = 0; i < m; i++)
  {
  String s = subtop[i];
  if (s == null);
    else
    {
      s =   myNodes[i];
     myNodes[i] = "";
     int n = internalNodes.length();
     for(int j = 0; j < n; j++)
      {
       String c = "" + internalNodes.charAt(j);
       if (s.contains(c))
       myNodes[i] = myNodes[i] + c;
      }
    if (print1)
     System.out.println(subtop[i] + " " + myNodes[i] );
    }
   }
  return myNodes;
   }


//Subtopologies are compared to decide whether
//or not they have the same set of leaves
//and the same set of inner nodes.
 //All pairs that authorize recombination
```

```
  // are kept in a hastable (=  a table for pairing)
private static boolean comparison(String Leaves1[],
                      String Leaves2[],
                      String Inner1[],
                      String Inner2[])
 {
  System.out.println("\nComparison of leaves and nodes");
  //Print everything
  Boolean printAll = false;
  Boolean recAllowed = false;
  int m = Leaves1.length;
  int n = Leaves2.length;
  Integer counter = 0;
  for(int i = 1; i < m; i++)
  {
   for(int j = 1; j < n; j++)
   {
if ( Leaves1[i].equals(Leaves2[j]))
{
 subTree1 = subtrees1[i];
 subTree2 = subtrees2[j];
 recAllowed = true;
 System.out.println(
 "Sets of leaves are EQUAL " + i +
 "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
 String val[] = {subTree1, subTree2};

 //Recombinant subtrees are kept in a hashtable
 hashtable.put(counter,val);
 counter++;
 if ( Inner1[i].equals(Inner2[j]))
System.out.println(
        "Sets of inner nodes are EQUAL " + i +
    "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
  else
System.out.println(
        "Sets of inner nodes are DIFFERENT " + i +
"  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
```

```
}
else
if (printAll)
System.out.println(
        "Sets of leaves are DIFFERENT " + i +
   "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
   }
  }
  return recAllowed;
 }

//Lengths are deleted from newTree
private static String deleteLengths(String newTree)
{
 String tree = "";
 String numeric = ".:0123456789";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
   char c = newTree.charAt(i);
   String s = "" + c;
   //Is c numeric: delete it!
   if (numeric.contains(s)); //add char c to output
  else tree = tree + c;
  }
 return tree;
}


//Input: string tree with complete information.
//Work: lengthsLeaves[] and lengthsInnerLinks[]
//are read from tree.
private static void decoderLengths(String tree)
{
if(print2)
System.out.println(
```

```
"Input tree at decoverLengths = \n "   + tree  );
//Delete :
tree = tree.replaceAll(":", "");
if(print2)
System.out.println(
"Input tree without : = \n "   + tree  );
int n = tree.length();
String thisChar = "";
String nextChar = "";
String numbers = ".0123456789";
boolean leaf = false;
boolean node = false;
String Leaf = "";
String numberLeaf = "";
String numberNode = "";
String Node = "";
int j ;
double f;

//All chars in tree are scanned
for(int i=0; i < n; i++)
{
thisChar = "" + tree.charAt(i);
if (i < n-1) nextChar  = "" + tree.charAt(i+1);
//Do we have a leaf?
if (leaves.contains(thisChar))
 {
 leaf = true;
 Leaf = "" + thisChar;
 }

//Do we have an internal node?
if (internalNodes.contains(thisChar))
 {
 node = true;
 Node = "" + thisChar;
 }
```

```
//Do we have a number that belongs in a leaf?
if ( (leaf) & (numbers.contains(thisChar)))
{
 numberLeaf = numberLeaf + thisChar;
}

//Do we have a number that belongs in a leaf?
if ( (node) & (numbers.contains(thisChar)))
  {
  numberNode = numberNode + thisChar;
 }

//Must we end accumulation of numbers of a leaf?
if ( (leaf) & (internalNodes.contains(nextChar)))
 {
  leaf = false;
  j = leaves.indexOf(Leaf);
  if (print1)
  System.out.println("Leaf = " + Leaf +
  " numberLeaf = " + numberLeaf);
  f = new Double(numberLeaf);
  lengthsLeaves[j] = (int) Math.floor(f);
  numberLeaf = "";
 }

//Must we end accumulation of numbers of a node?
if ( (node) & (!(numbers.contains(nextChar))))
 {
  node = false;
  j = internalNodes.indexOf(Node);
  if (print1)
  System.out.println("Node = " + Node);
  f = new Double(numberNode);
  lengthsInnerLinks[j] = (int) Math.floor(f);
  numberNode = "";
 }

}
```

```
}



//Recombination methods may produce an output
//with various and different assignment of lengths to
//inner node a. One is chosen.
public static String correction(String tree)
{
 String topo;
 String newTree = "";
 decoderLengths(tree);
 topo = deleteLengths(tree);
 newTree = insertLengths(topo);
 return newTree;
}


//A pair of subtrees that authorize recombination
//is chosen at random from the hashtable and
//recombination is executed.
private static  void recombination()
{
 //Possible pairs of recombinant subtrees are recovered
 //from the hashtable and put into an enumeration<>
 Enumeration<Integer> enumKey = hashtable.keys();
 int counter = 0;
 while(enumKey.hasMoreElements())
{
 Integer key = enumKey.nextElement();
 String[] val = hashtable.get(key);
 if (print3)
 System.out.println("Hasthtable " + key    + "\n"
 + val[0]  + "\n"+  val[1]);
 counter++;
    }

 //A pair of recombinant subtrees is chosen at random
```

```
 Integer election = r.nextInt(counter);
 System.out.println("Selected pair = " + election);
 String[] val = hashtable.get(election);
 subTree1 = val[0];
 subTree2 = val[1];
 System.out.println("Recombinant subtrees \n" +
             subTree1 + "\n" +  subTree2);

 //Recombination is executed and cleaned
 treeRec1 = tree1.replace(subTree1, subTree2);
 //Housekeeping
 treeRec1 = correction(treeRec1);
 treeRec2 = tree2.replace(subTree2, subTree1);
 treeRec2 = correction(treeRec2);
 System.out.println("\nRecombination is allowed:"
 + "\nOriginal trees \n"
 + tree1 + "\n" + tree2
 + "\nSwapping subtrees \n"
 +  subTree1 + "\n" + subTree2
 + "\nRecombinant trees \n"
 + treeRec1 + "\n" + treeRec2 );

}


 public static void main(String[] args)
 {
   //A topology is generated
   String topo1 = generateTopology();
   System.out.println(topo1 + " -> "
          + linkToNewick(topo1));
   tree1 = insertLengths(topo1);
   makeLengths();
   //Lengths of links are added to tree topology
   tree1 = insertLengths(topo1);
   if (print3)
   System.out.println("\n Tree1 = " + tree1);
   //Length of String   topology
```

```
  m1 = topo1.length();

   subtrees1 = new String[m1];
  subtrees1 = subtopologies(tree1);
  m1 = counterSubtop;
  //Subtopologies are found
  String[] subtopologies1 = new String[m1];
  for(int i = 0; i < m1; i++)
  subtopologies1[i] =  simplify(subtrees1[i]);



  System.out.println();
  String[] Leaves1 = new String[m1];
  Leaves1 = leaves(m1,subtopologies1);
  String[] Inner1 = new String[m1];
  Inner1 = inner(m1,subtrees1);
  //Chars of Leaves are rewritten in alphabet order
  System.out.println("\nSubtrees, subtopologies, Leaves, " +
                "Inner nodes" );
  for(int i = 0; i < m1; i++)
  {
if (subtopologies1[i] == null);
 else
 {

  System.out.println(i
  + "   " + subtrees1[i]
  + "   " + subtopologies1[i]
  + "   " +  Leaves1[i] + "   " + Inner1[i]);
 }
  }

  System.out.println();

  //A second topology is generated
  String topo2 = generateTopology();
  System.out.println(topo1 + " -> "
```

```
          + linkToNewick(topo2));
   tree2 = insertLengths(topo2);
   makeLengths();
 //Lengths of links are added to tree topology
   tree2 = insertLengths(topo2);
   if (print3)
   System.out.println("\n Tree2 = " + tree1);
   //Length of String   topology
   m1 = topo2.length();

   subtrees2 = new String[m1];
   subtrees2 = subtopologies(tree2);
   m1 = counterSubtop;
   //Subtopologies are found
   String[] subtopologies2 = new String[m1];
   for(int i = 0; i < m1; i++)
   subtopologies2[i] =  simplify(subtrees2[i]);



   System.out.println();
   String[] Leaves2 = new String[m1];
   Leaves2 = leaves(m1,subtopologies2);
   String[] Inner2 = new String[m1];
   Inner2 = inner(m1,subtrees2);
   //Chars of Leaves are rewritten in alphabet order
   System.out.println("\nSubtopologies, Leaves, " +
                 "Inner nodes" );
   for(int i = 0; i < m1; i++)
   {
 if (subtopologies2[i] == null);
  else
  {

   System.out.println(i
   + "   " + subtrees2[i]
   + "   " + subtopologies2[i]
   + "   " +  Leaves2[i] + "   " + Inner2[i]);
```

```
  }
   }
   boolean recAllowed =
   comparison(Leaves1,Leaves2, Inner1,Inner2);
   if (recAllowed)
   recombination();
   else
   System.out.println("Recombination is not allowed");
 }
}//End of main class K34  SubTopol6
```

44, page 145. The following code allows to see drawings as of the target tree as of the best tree by current generation.

```
//Program K44 EvoFull3
/*
 * We implement here
 * a full fledged evolutionary process
 * to propose an approximate solution to the problem of
 * fitting a tree to a distance matrix.
 *
 * The program runs a certain number of generations,
 * reports the tree that best approximates the  solution,
 * draws the  best approximation tree
 * and halts.
 *
 * We generate a random tree topology
 * and attach random integer lengths to links.
 * In that way we get the target tree, T.
 * Next, we find the associated distance matrix D  to T.
 * Our problem is:
 *
 * Find tree t such that the absolute distance between
 * matrix D and the associated matrix to t is a minimum.
 * Absolute distance between M and N = sum |Mij-Nij|.
 * This distance defines the FITNESS of each tree.
 * It decreases as good performance augments.
 * The best possible fitness is zero.
```

```
*
* We run the next test:
* If we begin with a tree T, compute D, and then t,
* we expect to return with the initial tree: t = T.
*
*
* To associate a distance matrix to a tree,
* we calculate the distance in within
* each pair of nodes (leaves + inner nodes).
* To that aim, we begin with the first node and
* walk over the tree until the second node
* and add the lengths of all traversed links.
* The resulting number is the  distance.
*
*
* Formats = Link and Newick's notations. And variants.
*
* An automatic halting option has been added to be applied
* if perfect matching is found.
* The program accepts for input
* a distance matrix with integer values.
* A test for being a distance matrix is run.
*
* To find the tree that best fit D,
* we  use evolution as follows:
*
* 1) Initialization:
*  We generate N topologies  on the same
*  set of leaves. By adding   lengths to links,
*  every topology gets represented by a tree.
* The fitness of each tree
* is calculated and the best is selected.
*  If the fitness of the best is zero,
*  the program halts and the tree is reported.
*
* 2) Evolution is composed of rounds of
*     cloning according to fitness,
*     random mutation and recombination.
```

```
*       After each generation, fitness is checked
*     for accomplishment.
*
*
* Mutation is Lamarckian:
* it always produces viable mutants.
* Topologies as well as links can mutate.
*
* A mutation results from deleting
* a given number of leaves
* to reinsert them anew in
* other places chosen at random.
* This creates a mutant topology.
* To mutate length of links:
* Lengths are read from tree and
* mutated separately.
* Then, they are reinserted.
*
*
* Recombination is lamarckian: it is authorized only
* when it produces viable recombinants.
* All their subtopologies are calculated
* and packed in arrays.
* The sets of leaves and of inner nodes
* of each subtopology are found.
* Subtopologies are compared by pairs,
* one from each topology.
* If they happen to have the same set of leaves
* and of inner nodes, recombination is authorized:
* the  subtrees that correspond with
* the subtopologies are swapped.
*
* If more than one pair of subtopologies authorize
* recombination, one is chosen at random.
* A hashtable is used to memorize those pairs that do it.
* Original topologies serves as subtopologies
* that authorize recombination but they do not count.
*
```

```
 *
 */



import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;

import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Random;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

import java.awt.*;

import javax.swing.JComponent;


public class EvoFull3 extends JPanel

{

 private static final long serialVersionUID = 1L;

 //The program accepts a distance matrix as input
 //else
 //it generates a random tree, calculates the distance
 //matrix associated to leaves and posits it
 //as target matrix.
 private static boolean inputMatrix = false;
```

```
//Input distance matrix:
//Example 5.4, pag 99, Isaev (2001)
//Introduction to Mathematical Methods in Bioinformatics
//Springer
private static double targetMatrix[][]
= {{0,8,3,14,10,12},
  {8,0,9,10,6,8},
  {3,9,0,15,11,13},
  {14,10,15,0,10,8},
  {10,6,11,10,0,8},
  {12,8,13,8,8,0}};

/*
private static double targetMatrix5[][]
= {{0,8,3,14,10},
  {8,0,9,10,6},
  {3,9,0,15,11},
  {14,10,15,0,10},
  {10,6,11,10,0}};

private static double targetMatrix4[][]
= {{0,8,3,14},
  {8,0,9,10},
  {3,9,0,15},
  {14,10,15,0}};

  */

//Number of leaves that must go in all trees > 3
private static int numberOfLeaves;

//Everything is built by recursion over this tree
private static   String   seed =
   "((Aa,Ba),Ca)";

//The best distance found in initialization
private static  int  distance0;
```

```
private static  String leaves =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
private static String internalNodes =
"abcdefghijklmnopqrstuvwxyz";

//Next leaf to be grafted
private static  int nextLeafIndex = 2;
//Next internal node to be inserted
private static  int nextIntNodeIndex = 0;


private static int maxNumberOfLeaves = leaves.length();




//Number of leaves to be deleted and reinserted
//to create a mutation
private static int nUndoRedo = 0;


private static boolean print1 = false;


//This is the original topology:
private static  String original = "";




//===========Most important parameters ===========

//Number of leaves to be inserted to seed:
//to get a tree with 4 leaves, add one leaf to seed
private static  int nAdditionalLeaves = 1;
```

```
 //Number of topologies = number of trees that
 //conform the evolving population
 private static  int N = 50;

 //Max number of generations to run evolution
 static private int numberOfGenerations = 2001;
 static private long generation = 0;

 //Mutation rate per site   per generation
 //in the lengths of links
 static private double    m = 0.5;



//Report Best tree every lapse generations
static private int lapse = 1000;


 //================================================


 //Data for binary tree
 private static String[] Record =  new String[20*N];


 private static  int indexOfTopologies;
 private static String topologies[] = new String[N];
 private static String trees[] = new String[N];
 private static String newTrees[] = new String[N];


 private static String[] bestTreeOfTopology
                              = new String[N];

 //Last filled position  in Record[]
 private static int lastUsedPosition;
```

```
private static String selectedTree = "";


  //Length of the link that glues a leaf to the tree
 private static int lengthsLeaves[]  =
              new int[maxNumberOfLeaves];
 //Length of the link that glues
 //an inner node to another one
 private static int lengthsInnerLinks[]  =
             new int[maxNumberOfLeaves];


 private static Random r = new Random();
 //The control parameter of the random generator for int
 private static int maxLength = 10;

 //Paths from each leaf to node a
 private static String[] Paths =
             new String[maxNumberOfLeaves];

 private static String targetTree;
 private static String targetTopology;
 private static String bestTopology;


 //=======Evolution parameters============




static private double distance[] = new double[N];
static private double fitness[] = new double[N];



//Index of the best
static private int best;
```

```
//The best tree for a given topology
static private String bestTree = "";
//Distance from bestTree to target
static private double bestDistance ;
//Boolean flag for termination of job
static private boolean continueWork = true;


//=========Debugging help ===============

private static boolean print  = false;

private static boolean print2 = false;
private static boolean print3 = false;
private static boolean print4 = false;
//Turn on print5 firstly
private static boolean print5 = false;
private static boolean print6 = false;
private static boolean print7 = true;


//==============

private static String titleFrame = "";
private static String title1 = "";
private static String title2 = "";
private static EvoFull3 newContentPane = new EvoFull3();

private static DrawBiTree14 a;


 //=======Constructor of main class: GUI basics=====

 public EvoFull3()
 {

 }//End of constructor
```

```
public static void main(String[] args)
{
   // Schedule a job for the event-dispatching thread:
   // creating and showing this application's GUI.
   javax.swing.SwingUtilities.invokeLater(new Runnable()
    {
     public void run()
     {
      //If input is a distance matrix:
      if (inputMatrix)
      {
       printMatrix(targetMatrix, "Input Matrix");
       boolean isD = isMatrixDistanceMatrix(targetMatrix);
         if (isD) System.out.println("\nMatrix " +
        "is a distance matrix");
         else
          {
        System.out.println("Matrix " +
             "is NOT a distance matrix");
        System.exit(0);
           }
         numberOfLeaves = targetMatrix.length;
         nAdditionalLeaves = numberOfLeaves-3;
    }
    else//if a random tree is generated -> Matrix
    {
     numberOfLeaves =    3 + nAdditionalLeaves;
     String topology = randomTopology();
     targetTopology = simplify(topology);
     generateLengths();
     targetTree = insertLengths(topology);
     System.out.println("Target tree: each link " +
     "has its own length = \n" + targetTree);
     targetMatrix  =
```

```
        new double[numberOfLeaves][numberOfLeaves];
    targetMatrix = associatedMatrix(targetTree);
    printMatrix(targetMatrix, "Target Matrix");

    double d = absDistance(numberOfLeaves,
                  targetMatrix, targetMatrix);


    System.out.println( "\nDistance from target matrix " +
     "to target = " + d);
    }


  System.out.println("\nDefinition: \nDistance to target = " +
        " Absolute distance between" +
       " target matrix  and \n" +
       "the distance matrix associated to " +
       "the tree with lengths of links that \n" +
       "is the best fitting  of the topology.\n" +
       "Perfect fitting: distance = 0 \n");
    System.out.println("\nNumber of Generations = "
          + numberOfGenerations);
    if (print7)

    System.out.println("Number of leaves = "
                       + numberOfLeaves +
                        "\nMutation rate = "   + m +
                "\nPopulation size = " + N);

    initialization();
    evolution();

    }
  });
}
```

```
//Test for being a distance matrix.
//MaxLenght = max value among entries is calculated.
private static boolean
    isMatrixDistanceMatrix(double[][] M)
{
 int l = M.length;
 boolean isD = true;
 //Is M(u,u) = 0?
 for(int u = 0; u < l; u++)

if (M[u][u] ==  0) ;
 else
  {
isD = false;
return isD;
  }

 //Is M(u,v) = M(v,u)?
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
if (M[u][v] == M[v][u] ) ;
 else
{
 isD = false;
 return isD;
}

 //Is M(u,v)>0 for u not v?
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
if (M[u][v] >  0) ;
 else
{
 isD = false;
 return isD;
}
```

```
 //Is M(u,v) < M(u,w) + M(w,v)?
 for(int u = 0; u < l; u++)
 for(int v = 0; v < l; v++)
 for(int w = 0; w < l; w++)

if (M[u][v] <= M[u][w]+M[w][v]  ) ;
 else
{
 isD = false;
 return isD;
}
 double m = 0;
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
 if (M[u][v] > m ) m = M[u][v];
 maxLength = (int) m + 1;
 return true;
}



//Simplification from Link notation to Newick's notation:
//(((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String linkToNewick(String newTree)
{
 String tree = "";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
```

```
  }
 return tree;
}



private static String generateTopology()
 {
  //Next leaf to be grafted
  nextLeafIndex = 2;
  //Next internal node to be inserted
  nextIntNodeIndex = 0;
  //A tree topology is generated
  String topology = seed;
  for(int i = 0; i < nAdditionalLeaves; i++ )
 topology = generateChild(topology);
  return topology;
 }


//Random lengths are added to a tree in Link notation
private static void makeLengths()
{
//Lengths with digit values are generated
for(int i = 0; i < numberOfLeaves; i++)
 {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
 }
}

//The population is initialized with nTop topologies.
//Lengths are inserted to get nTop trees
//Fitness is assessed:
//Has the solution been guessed?
private static void initialization()
{
```

```
 System.out.println("\n====" +"INITIALIZATION : ");
 if (print5)
  System.out.println("\nSeed = " + seed
 + " -> "    + linkToNewick(seed));

 if (print7)
  System.out.println( N
   + " random topologies " +
   "and their trees are generated \n");

 //nTop topologies and trees are generated
 for(int i = 0; i < N; i++)
 {
  String topo1 = generateTopology();
  topologies[i] = topo1;
  String tree = insertLengths(topo1);
  makeLengths();
  //Lengths of links are added to tree topology
  tree = insertLengths(topo1);
  trees[i] = tree;
  if (print6)
  {
System.out.println("\n" +  i + "   " + topo1 + " -> "
          + linkToNewick(topo1));
System.out.println("Tree1 = " + tree);
  }

 }
 assessFitnesses();
 scaleFitnesses();
}


//=====================================
//============= EVOLUTION =============
//=====================================
```

```
//Fitness is calculated for each tree
//Fitness  =
//Distance between target matrix and
//the associated distance matrix to tree is calculated.
//The minimal distance is found:
//this renders the fittest tree.
//Has the solution been guessed?
private static double assessFitnesses()
{
 if (print6)
  System.out.println(  "\n=====Fitnesses are assessed");

 String tree = "";
 //String topo = "";
 double d = 0;
for(int i = 0; i < N; i++)
{

 tree = trees[i];
 decoderLengths(tree);
 double M[][] = distanceMatrix(tree);
 //Absolute distance from the associated distance matrix
 //of the tree
 //to target matrix is evaluated
 d = absDistance(numberOfLeaves,targetMatrix, M);
 distance[i] = d;


 if (print6)
  {
   System.out.println( "\nTree = \n" + trees[i]);
   System.out.println( "\nDistance to target = " + d);
  }


if (print5)
 {
  printMatrix(M, "Distance matrix associated to tree");
```

```
  printMatrix(targetMatrix, "Target matrix");
}

/*

//Target topology has been found?
topo = deleteLengths(bestTree);
topo = simplify(topo);
if (targetTopology.contentEquals(topo))
 {
  System.out.println( "\n \nTOPOLOGY HAS BEEN FOUND: \n" +
"at generation " + generation
+ "\nFound topology  = \n" + topo
+ "\nTarget Topology = \n" + targetTopology);
 //System.exit(0);
 if (print5) print5 = false;
}

*/

//End work if solution has been found.
if (d == 0)
{
System.out.println("\n BestTree  = \n" +  bestTree
           + "\nDistance = " + bestDistance
           + "\ntarget tree = \n" + targetTree
           + "\nInitial distance = " + distance0);

continueWork = false;
}
}//End of for(int i...



 //The fittest individual minimizes distance function
 bestDistance = distance[0];
 best = 0;
```

```
 for(int i = 1; i < N; i++)
 {
  if (distance[i] < bestDistance )
   {
    bestDistance = distance[i];
    best = i;
   }
 }

 //Best tree
 bestTree = trees[best];




 if (generation == 0 ) distance0 = (int) bestDistance;
 if (print7)
 {
 System.out.println("\n BestTree  = \n" +  bestTree
           + "\nDistance = " + bestDistance
           + "\ntarget tree = \n" + targetTree
           + "\nInitial distance = " + distance0);


 }
 return bestDistance;
}



//Fitness is rescaled to take values
//in within 0(the worst) and 1(the best)
private static void scaleFitnesses()
{
 if (print6)
  System.out.println(  "\n====Fitnesses are rescaled: " +
  "0(the worst) and 1(the best)");
```

```
 double maxD = distance[0];
 double minD = distance[0];
 double b = 1 ;
 //Min and max distances are found
 for(int i = 1; i < N; i++)
  {
if ( distance[i] > maxD ) maxD =  distance[i];
if ( distance[i] < minD ) minD =  distance[i];
  }

 if (minD == maxD) //All   equally fit
  {
   for(int i = 0; i < N; i++)
 fitness[i] = 1;
  }
 else
  {
 //Distance to target determine fitness:
//fitness[maxD] = 0;    fitness[minD] = 1;
 for(int i = 0; i < N; i++)
 {
  b = 1;
  b = b/(minD - maxD);
  double a = -b*maxD;
  fitness[i] = a + b * distance[i];
}
  }
 if (print6)
  for(int i = 0; i < N; i++)
   {
 System.out.println("Individual = "
            + i + " has distance " +
          distance[i] + " and fitness " + fitness[i]);
   }
}

//Probability of cloning is proportional to fitness.
//N new individuals are formed.
```

```
private static void selectiveCloning()
{
 if (print6)
  System.out.println(  "\n====Selective cloning");
//The best is cloned
 newTrees[0] = trees[best];
 if (print6)
 System.out.println("Individual = "
                      + best + " with fitness " +
 fitness[best] + "-> new individual " + 0
 + " = "  + newTrees[0]  );
//N-1 children are sampled
for(int i = 1; i < N; i++)
{
 //flag
 boolean go = true;
 while (go)
  {
 //An individual is chosen at random
 int k = r.nextInt(N);
 //Probability of cloning is proportional to fitness.
 double q = r.nextDouble();
 if ( q < fitness[k]) //make a copy of k
  {
 newTrees[i] = trees[k];
 if (print6)
 System.out.println("Individual = "
                      + k + " with fitness " +
 fitness[k] + "-> new individual " + i
 + " = "  + newTrees[i]  );

 go = false;//end task
  }
  }

}//End of for //N-1 children are sampled
}
```

```
//============Mutation============




//Detect internal nodes, different than leaves
private static boolean charIsInternalNode(char c)
{
 String s = "" + c;
 if (internalNodes.contains(s)) return true;
 else return false;
}



//Simplification from Link notation to Newick's notation:
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTopo)
{
 String topo = "";
 //length of string newTopo
 int l = newTopo.length();
 for(int i = 0; i < l; i++ )
  {
   //newTopo is scanned char by char
  char c = newTopo.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    topo = topo + c;
  }
 return topo;
}

//Rule 2 is used to generate a new topology:
//(S)I -> ((S)N,FN)I.
//S = subtopology, I = internal node, F = new leaf,
```

```
//N = new internal node.
//Rule 2 grafts new branch FN to link (S)I
//i is the position of char I in string parentTopo
private static String useRule2(String parentTopo, int i,
       char F, char N)
{
 String newTopo = "";
 char I = parentTopo.charAt(i);
 if (print1)
  System.out.println("i = " + i + " I = " + I);
 //Subtopology S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = i;
 boolean go = true;
 //All right parentheses to the near left of I are captured
 while (go)
  {
   position = position -1;
   String a = "" + parentTopo.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
    {
 counter++;
 S = a + S;
    }
   else go = false;
  }//End of while
 if (print1)
 System.out.println("counter = " + counter);
 //The complete subtopology is captured:
 //Parentheses must balanced
```

```
 while (counter > 0)
  {
   String a = "" + parentTopo.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
    System.out.println("Subtopo = " + S);
   position = position -1;
   }
 //Replacement rule 2:
 //((S)n,Fn)i -> (S)i
 //where   F are leaves, n and i are  inner nodes,
 //(S) is a complex structure.
 String SnFni  =   S  + I;
 String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
 //Rule 1 is executed
 newTopo = parentTopo.replace(SnFni, replacement);
 //If I links to a complex structure
 return newTopo;
}

//From parentTopo, a topology in Newick's notation,
//a Child  topology is generated by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String parentTopo)
{

 nextLeafIndex++;
 char nextLeaf= leaves.charAt(nextLeafIndex);
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = internalNodes.charAt(nextIntNodeIndex);


 //link to be replaced
 String link = "";
```

```
 //String link is replaced by what:
 String replacement = "";
 //new topology generated by replacement
 String newTopo = "";
 //length of parent topology
 int l = parentTopo.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
  {
   //parentTopo is scanned char by char
   char I = parentTopo.charAt(i);
   //Is I an internal node that denotes a link?
   if (charIsInternalNode(I)) counter++;
 }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
 {
   //parentTopo is scanned char by char
char I = parentTopo.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = parentTopo.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTopo = useRule2(parentTopo, i,
               nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
```

```
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTopo = parentTopo.replace(link, replacement);
 }
if (print2)
System.out.println(newTopo + " -> "
                      + simplify(newTopo));


 i = l;
   }
  counter++;
}
 }
return newTopo;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply the antirule1:
//(An,Fn)i  -> Ai
private static String  antiRule1(String topo,
int positionLeaf, int positionNode)
{
 if (print2)
  System.out.println("AntiRule1");
 int beginIndex = positionLeaf - 4;
 int endIndex = positionNode + 3;
 String AnFni = topo.substring(beginIndex, endIndex);
 if (print2)
  System.out.println("Anfni = " + AnFni);
 String Ai = "" + topo.charAt(beginIndex+1)
              + topo.charAt(endIndex-1);
```

```
 if (print2)
  System.out.println("Ai = " + Ai);
 topo = topo.replace(AnFni, Ai);
return topo;
}



//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply the antirule2:
//((S)n,Fn)i -> (S)i
//positionNode refers to the last appearing of node n
private static String  antiRule2(String topo,
int positionLeaf, int positionNode)
{
 if (print2)
System.out.println("AntiRule2");
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 if (print2)
  System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
            + internalNodes.charAt(nextIntNodeIndex);
 int firstPosition = topo.indexOf(innerNode);
 String newTopo = "";
 char i = topo.charAt(positionNode+2);
 if (print2)
  System.out.println("FirstPosition = " +
                     firstPosition + " i = " + i);
 //Subtopology S of link (S)I must be read
 String S = "";
 //Counter of left parenthesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
```

```
int position = firstPosition;
boolean go = true;
//All right parentheses to the near left of I are captured
while (go)
 {
  position = position -1;
  String as = "" + topo.charAt(position);

  if (print2)
   System.out.println("a = " + as + " p = " + pr);
  if (as.equals(pr) )
   {
counter++;
S = as + S;
   }
  else go = false;
 }//End of while
if (print2)
System.out.println("counter = " + counter);
//The complete subtopology is captured:
//Parentheses must balanced
while (counter > 0)
 {
  String as = "" + topo.charAt(position);
  if (as.equals(pr))   counter++;
  if (as.equals(pl))  counter--;
  S = as + S;
  if (print2)
   System.out.println("Subtopo = " + S);
  position = position -1;
 }
String n = innerNode;
//Replacement antiRule 2:
//((S)n,Fn)i -> (S)i
//where A and F are leaves, n and i are  inner nodes,
//(S) is a complex structure.
String SnFni  =  "("+ S  + n + "," + leaf + n + ")" + i ;
String Si = S + i;
```

```
 if (print2)
  System.out.println("\nSnFni = " + SnFni + "\n Si = " + Si);
 //Rule 1 is executed
 newTopo = topo.replace(SnFni, Si);
 return newTopo;
}



//The last inserted  leaf is deleted from topology.
private static String deleteLastLeaf(String topology)
{
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 if (print2)
 System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
           + internalNodes.charAt(nextIntNodeIndex);
 if (print2)
 System.out.println("Accompanying inner  node = "
                   + innerNode);
 //new topology = output
 String newtopology = "";
 //length of  topology as string
 int l = topology.length();
 //Last positions are found
 int positionLeaf = -1;
 int positionNode = -1;
 //topology is scanned char by char
 for(int i = 0; i < l; i++ )
  {
 String c = "" + topology.charAt(i);
 if (c.equals(leaf)) positionLeaf = i;
 if (c.equals(innerNode)) positionNode = i;
  }
 if (print2)
  {
```

```
   System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
   System.out.println("Position of " + innerNode +
         " = " + positionNode);
  }
 int firstPosition = topology.indexOf(innerNode);
 //Check for the char before firstPosition
 char a = topology.charAt(firstPosition - 1);
 //If I links to a complex structure
 if (a == ')')
 newtopology = antiRule2(topology,positionLeaf,positionNode);
 else  //Apply antiRule1 for simple cases
  newtopology = antiRule1(topology,positionLeaf,positionNode);
 nextLeafIndex--;
 nextIntNodeIndex--;
 return newtopology;
}



//=========================

//Lengths are deleted from newTree
private static String deleteLengths(String newTree)
{
 String tree = "";
 String numeric = ".:0123456789";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
   char c = newTree.charAt(i);
   String s = "" + c;
   //Is c numeric: delete it!
   if (numeric.contains(s)); //add char c to output
  else tree = tree + c;
  }
```

```
 return tree;
}




//Mutation:
//some leaves are deleted and then reinserted
//in random order.
//Deleting can happen in either of two ways:
//Reversing the order of synthesis else
//in random order.
public static String mutation(String topology,
                       int nAdditionalLeaves)
{
 //Mutation has two stages:
 //deletion of some leaves that undoes generating work +
 //re-insertion in places chosen at random.

 //Number of leaves to be deleted and then reinserted
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
  else
   nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;


 if (print2)
  System.out.println("Number of leaves to" +
" be deleted and then reinserted = " +
nUndoRedo);

 //Some leaves are deleted:
 //reversed order to that of synthesis.
 if (print1)
  System.out.println("Some leaves are deleted");
 for(int i = 0; i < nUndoRedo; i++ )
 {
   //Last inserted leaf is deleted.
```

```
   topology = deleteLastLeaf(topology);
   if (print1)
    System.out.println(topology);
 }
 //deleted leaves are reinserted
 if (print1)
  System.out.println("Deleted leaves are reinserted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
  topology = generateChild(topology);
  }



 //Output
 String mutant = topology;
 if (print1)
  System.out.println("\nOriginal and mutant topologies = \n"
  + original + "\n" + mutant);
 if (print1)
 if (original.equals(mutant))
  System.out.println("Mutation made nothing");
   else
  System.out.println("\nMutation worked");
 return mutant;
}


public static String mutation(String topology)
{
 //This original topology is next subject to mutation
 original = topology;
 String mutantTopology = topology;
 if (print2)
System.out.println("\nOriginal   topology = \n"
                + original  );
 //Number of leaves to be deleted and then reinserted
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
```

```
else
 nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;
 //nUndoRedo = 13;

 //Mutation has two stages:
 //deletion of some leaves that undoes generating work +
 //re-insertion in places chosen at random.

 if (print2)
  System.out.println("Number of leaves to" +
   " be deleted and then reinserted = " +
   nUndoRedo);
 //Some leaves are deleted:
 if (print2)
 System.out.println("Some leaves are deleted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
//Last inserted leaf is deleted.
mutantTopology = deleteLastLeaf(mutantTopology);
if (print2)
   System.out.println(mutantTopology);
  }
 //Some leaves are reinserted
 if (print2)
 System.out.println("Deleted leaves are reinserted");
 for(int i = 0; i < nUndoRedo; i++ )
  {
   topology = generateChild(topology);
  }
 return mutantTopology;
}



//Random lengths are added to a tree in Link notation
public static String insertLengths(String treeLinkN)
{
 String thisChar = "";
```

```
 String newTree = "";
 //length of String   treeLinkN
 int m = treeLinkN.length();
 //TreeLinkN is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   thisChar = "" + treeLinkN.charAt(i);
   newTree = newTree + thisChar;
   if (print1)
    System.out.println(newTree.length() + " " + newTree);
   //Lengths are appended
  if (leaves.contains(thisChar))
   {
    int k = leaves.indexOf(thisChar);
    double j = lengthsLeaves[k];
    newTree = newTree + ":" + j;
   }
 if (internalNodes.contains(thisChar))
  {
   int k = internalNodes.indexOf(thisChar);
   double j = lengthsInnerLinks[k];
   newTree = newTree + ":" + j;
  }
 }
 return newTree;
}


//Input: string tree with complete information.
//Work: lengthsLeaves[] and lengthsInnerLinks[]
//are read from tree.
private static void decoderLengths(String tree)
{
 if(print2)
 System.out.println(
"Input tree at decoverLengths = \n "   + tree  );
 //Delete :
 tree = tree.replaceAll(":", "");
```

```
 if(print2)
  System.out.println(
"Input tree without : = \n "   + tree  );
 int n = tree.length();
 String thisChar = "";
 String nextChar = "";
 String numbers = ".0123456789";
 boolean leaf = false;
 boolean node = false;
 String Leaf = "";
 String numberLeaf = "";
 String numberNode = "";
 String Node = "";
 int j ;
 double f;

 //All chars in tree are scanned
 for(int i=0; i < n; i++)
 {
  thisChar = "" + tree.charAt(i);
  if (i < n-1) nextChar  = "" + tree.charAt(i+1);
  //Do we have a leaf?
  if (leaves.contains(thisChar))
   {
 leaf = true;
 Leaf = "" + thisChar;
   }

  //Do we have an internal node?
  if (internalNodes.contains(thisChar))
   {
 node = true;
 Node = "" + thisChar;
   }

  //Do we have a number that belongs in a leaf?
  if ( (leaf) & (numbers.contains(thisChar)))
{
```

```
 numberLeaf = numberLeaf + thisChar;
}

  //Do we have a number that belongs in a leaf?
  if ( (node) & (numbers.contains(thisChar)))
    {
  numberNode = numberNode + thisChar;
 }

  //Must we end accumulation of numbers of a leaf?
  if ( (leaf) & (internalNodes.contains(nextChar)))
 {
  leaf = false;
  j = leaves.indexOf(Leaf);
  if (print1)
  System.out.println("Leaf = " + Leaf +
  " numberLeaf = " + numberLeaf);
  f = new Double(numberLeaf);
  lengthsLeaves[j] = (int) Math.floor(f);
  numberLeaf = "";
 }

  //Must we end accumulation of numbers of a node?
  if ( (node) & (!(numbers.contains(nextChar))))
 {
  node = false;
  j = internalNodes.indexOf(Node);
  if (print1)
  System.out.println("Node = " + Node);
  f = new Double(numberNode);
  lengthsInnerLinks[j] = (int) Math.floor(f);
  numberNode = "";
 }

 }
}

//Random lengths are added to a tree in Link notation
```

```
public static void generateLengths()
{
 //Lengths with digit values are generated
 for(int i = 0; i < numberOfLeaves; i++)
  {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
  }
}


//Lengths undergo mutation
private static  void mutateLengths()
{

 //Mutation in links that connect leaves
 for(int k = 0; k < numberOfLeaves; k++)
  {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsLeaves[k] = r.nextInt(maxLength);

  }
 //Mutation in links that connect inner nodes
 for(int k = 0; k < numberOfLeaves-2; k++)
  {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsInnerLinks[k] = r.nextInt(maxLength);

  }



}
```

```
//Type of  mutation is decided and executed:
//mutation of lengths of links
//else full mutation (topology and lengths of links).
private static String mutateTree(String tree)
{
 //Topology is read from tree
 String topology = deleteLengths(tree);
 //Lengths are read from tree and recorded in vectors
 //lengthsLeaves[] and lengthsInnerLinks[]
 decoderLengths(tree);
 mutateLengths();
 String  mutantTree = "";
 //Mutate lengths else topology ( + new lengths)?
 double s = r.nextDouble();
 if ( s < 0)
{
 if (print1)
 System.out.println("\n====Mutation of lengths");

  mutantTree =  insertLengths(topology);

  if (print1)
  {
  System.out.println("Input tree =  " + tree);
  System.out.println("Topology =  " + topology);
  System.out.println("Output tree= " + mutantTree);
  }
 }
else
 {
 if (print1)
  System.out.println("\n====Mutation of topology");


//Mutation of topology
  String mutantTopology = mutation(topology,
                           nAdditionalLeaves);
```

```
  //Mutant lengths are reinserted
  mutantTree =  insertLengths(mutantTopology);
 }

 return mutantTree;
}



//Individuals are mutated
private static void mutation()
{
 if (print6)
  {
System.out.println("\n====Mutation \n");

System.out.println("Trees to be mutated");
  }
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print6)
  System.out.println(trees[i]);
 }
 //The old best, i = 0, is conserved intact
 newTrees[0] = trees[0];
 //Other N-1 individuals suffer mutation
 for(int i = 1; i < N; i++)
  newTrees[i] = mutateTree(trees[i]);


if (print6)
System.out.println("Mutant trees");
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print6)
  System.out.println(trees[i]);
```

```
 }
}

//==========Recombination============


 //Counter of subtopologies
 private static  int counterSubtop;
 //Particular number of subtopologies
 private static  int m1 = 0;



 private static  String[] subtrees1, subtrees2;
 private static  String   subTree1, subTree2,
                     treeRec1, treeRec2;

 //Structure of hashtable : integer + {string1, string2}
 private static Hashtable<Integer, String[]> hashtable =
 new  Hashtable<Integer, String[]>();




//========== Subtrees are listed ============


//Subtrees of  topology are found
private static String[] subtopologies(String topology)
{
 if (print1)
  System.out.println("\nSubtrees are found");
 //Counter of left parenthesis
 int counter = 0;
 //Counter of subtopologies
 counterSubtop = 0;
 String pl = "" + '(';
```

```
 String pr = "" + ')';
 //Length of String   topology
 int m = topology.length();
 String[] subtopologies = new String[m];
 //String topology is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   String Sub = "";
   String a = "" + topology.charAt(i);
   int position = i+1;
   if (a.equals(pl))
   {
    counter++;
    Sub =  Sub + a;
    //The complete subtopology is captured:
    //Parentheses must balanced
   while (counter > 0)
    {
     String b = "" + topology.charAt(position);
     if (b.equals(pl))   counter++;
     if (b.equals(pr))  counter--;
     Sub =  Sub + b;
     position++;
  }
 if (print2)
   System.out.println(Sub);
 subtopologies[counterSubtop] = Sub;
 counterSubtop++;
 }
}
return subtopologies;
}


//Leaves of subtopologies  are found
private static String[]leaves(int m, String[] subtop)
{
 if (print1)
```

```
 System.out.println("Leaves are found");
String[] myLeaves = new String[m];
for(int i = 0; i < m; i++)
 {
  String s = subtop[i];
  if (s == null);
  else
   {
    myLeaves[i] = "";
    int n = s.length();
    for(int j = 0; j < n; j++)
     {
      String c = "" + subtop[i].charAt(j);
      if (leaves.contains(c))
      myLeaves[i] = myLeaves[i] + c;
    }
  if (print1)
    System.out.println(subtop[i] + " "
                       + myLeaves[i] );
  }
 }
//Chars of Leaves are rewritten in alphabet order
if (print1)
 System.out.println("Leaves are listed orderly" );
for(int i = 0; i < m; i++)
 {
  String s = subtop[i];
  if (s == null);
   else
    {
     s =   myLeaves[i];
     myLeaves[i] = "";
     int n = leaves.length();
  for(int j = 0; j < n; j++)
   {
    String c = "" + leaves.charAt(j);
    if (s.contains(c))
    myLeaves[i] = myLeaves[i] + c;
```

```
    }
   if (print1)
   System.out.println(subtop[i] + " " + myLeaves[i] );
  }
 }
 return myLeaves;
}



//Inner nodes of subtopologies  are found
private static String[] inner(int m, String[] subtop)
{
 if (print1)
  System.out.println("\nInternal nodes are found");
 String[] myNodes = new String[m];
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
    else
     {
      myNodes[i] = "";
      int n = s.length();
      for(int j = 0; j < n; j++)
       {
        String c = "" + subtop[i].charAt(j);
        if (internalNodes.contains(c))
        myNodes[i] = myNodes[i] + c;
       }
    if (print1)
System.out.println(subtop[i] + " " + myNodes[i] );
  }
 }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
 System.out.println("Inner nodes are listed orderly" );
 for(int i = 0; i < m; i++)
```

```
  {
   String s = subtop[i];
   if (s == null);
    else
     {
      s =   myNodes[i];
      myNodes[i] = "";
      int n = internalNodes.length();
      for(int j = 0; j < n; j++)
       {
        String c = "" + internalNodes.charAt(j);
        if (s.contains(c))
      myNodes[i] = myNodes[i] + c;
       }
  if (print1)
   System.out.println(subtop[i] + " " + myNodes[i] );
 }
}
return myNodes;
}


//Subtopologies are compared to decide whether
//or not they have the same set of leaves
//and the same set of inner nodes.
//All pairs that authorize recombination
// are kept in a hastable (=  a table for pairing)
private static boolean comparison(String Leaves1[],
                      String Leaves2[],
                      String Inner1[],
                      String Inner2[])
{
 if (print6)
 System.out.println("\nComparison of leaves and nodes");
 Boolean recAllowed = false;
 int m = Leaves1.length;
 int n = Leaves2.length;
 Integer counter = 0;
```

```
for(int i = 1; i < m; i++)
 {
  for(int j = 1; j < n; j++)
   {
    if ( Leaves1[i].equals(Leaves2[j]))
     {
      subTree1 = subtrees1[i];
      subTree2 = subtrees2[j];
      recAllowed = true;
      if (print6)
       System.out.println(
         "Sets of leaves are EQUAL " + i +
       "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
       String val[] = {subTree1, subTree2};

       //Recombinant subtrees are kept in a hashtable
       hashtable.put(counter,val);
       counter++;
       if ( Inner1[i].equals(Inner2[j]))
        {
     if (print6)
     System.out.println(
       "Sets of inner nodes are EQUAL " + i +
         "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
        }
       else
        {
      if (print6)
  System.out.println(
      "Sets of inner nodes are DIFFERENT " + i +
      "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
         }
     }
   else
 if (print6)
  System.out.println(
      "Sets of leaves are DIFFERENT " + i +
       "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
```

```
   }
 }
 return recAllowed;
}


//A pair of subtrees that authorize recombination
//is chosen at random from the hashtable and
//recombination is executed.
private static  String recombinationTrees(String tree1,
                                          String tree2)
{
 //Possible pairs of recombinant subtrees are recovered
 //from the hashtable and put into an enumeration<>
 Enumeration<Integer> enumKey = hashtable.keys();
 int counter = 0;
 while(enumKey.hasMoreElements())
  {
   Integer key = enumKey.nextElement();
   String[] val = hashtable.get(key);
   if (print3)
    System.out.println("Hasthtable " + key    + "\n"
 + val[0]  + "\n"+  val[1]);
   counter++;
   }

 //A pair of recombinant subtrees is chosen at random
 Integer election = r.nextInt(counter);
 if (print6)
  System.out.println("Selected pair = " + election);
 String[] val = hashtable.get(election);
 subTree1 = val[0];
 subTree2 = val[1];
 if (print6)
   System.out.println("Recombinant subtrees \n" +
      subTree1 + "\n" +  subTree2);

 //Recombination is executed
```

```
 treeRec1 = tree1.replace(subTree1, subTree2);
 treeRec2 = tree2.replace(subTree2, subTree1);
 if (print6)
  System.out.println("\nRecombination is allowed:"
 + "\nOriginal trees \n"
 + tree1 + "\n" + tree2
 + "\nSwapping subtrees \n"
 +  subTree1 + "\n" + subTree2
 + "\nRecombinant trees \n"
 + treeRec1 + "\n" + treeRec2 );
 return treeRec1;
}


//Recombination of tree1 and tree2 is tried out
public static String recombinationMaster(String tree1,
                              String tree2)
{
 //First topology
 String topo1 = deleteLengths(tree1);
 if (print6)
  {
   System.out.println(topo1 + " -> "
       + linkToNewick(topo1));
   System.out.println("\n Tree1 = " + tree1 +
 "\n Topo1 = " + topo1);
  }


 //Length of String   topology
 m1 = topo1.length();

 subtrees1 = new String[m1];
 subtrees1 = subtopologies(topo1);
 m1 = counterSubtop;
 //Subtopologies are found
 String[] subtopologies1 = new String[m1];
 for(int i = 0; i < m1; i++)
```

```
 subtopologies1[i] =  simplify(subtrees1[i]);
if (print6)
System.out.println();
String[] Leaves1 = new String[m1];
Leaves1 = leaves(m1,subtopologies1);
String[] Inner1 = new String[m1];
Inner1 = inner(m1,subtrees1);
//Chars of Leaves are rewritten in alphabet order
if (print5)
 System.out.println("\nSubtrees, subtopologies, Leaves, " +
            "Inner nodes" );
for(int i = 0; i < m1; i++)
 {
  if (subtopologies1[i] == null);
  else
   {
if (print5)
       System.out.println(i
  + "   " + subtrees1[i]
  + "   " + subtopologies1[i]
  + "   " +  Leaves1[i] + "   " + Inner1[i]);
   }
 }

if (print6)
 System.out.println();

//Second topology
String topo2 = deleteLengths(topo2);
if (print6)
 {
  System.out.println(topo1 + " -> "
  + linkToNewick(topo2));

System.out.println("\n Tree2 = " + tree2 +
"\n Topo2 = " + topo2);
 }
//Length of String   topology
```

```
m1 = topo2.length();

subtrees2 = new String[m1];
subtrees2 = subtopologies(tree2);
m1 = counterSubtop;
//Subtopologies are found
String[] subtopologies2 = new String[m1];
for(int i = 0; i < m1; i++)
 subtopologies2[i] =  simplify(subtrees2[i]);


if (print6)
 System.out.println();
String[] Leaves2 = new String[m1];
Leaves2 = leaves(m1,subtopologies2);
String[] Inner2 = new String[m1];
Inner2 = inner(m1,subtrees2);
//Chars of Leaves are rewritten in alphabet order
if (print5)
 System.out.println("\nSubtopologies, Leaves, " +
            "Inner nodes" );
for(int i = 0; i < m1; i++)
 {
  if (subtopologies2[i] == null);
  else
   {
if (print5)
     System.out.println(i
  + "   " + subtrees2[i]
  + "   " + subtopologies2[i]
  + "   " +  Leaves2[i] + "   " + Inner2[i]);
   }
 }
boolean recAllowed =
  comparison(Leaves1,Leaves2, Inner1,Inner2);
String recombinant = "";
if (recAllowed)
  recombinant = recombinationTrees(tree1,tree2);
```

```
 else
  {
   recombinant = tree1;

   if (print6)
  System.out.println("Recombination is not allowed");
  }
 return recombinant;
}


//Recombination methods may produce an output
//with various and different assignment of lengths to
//inner node a. One is chosen.
public static void correction()
{
String topo;
 for(int i = 1; i < N; i++)
  {
decoderLengths(newTrees[i]);
topo = deleteLengths(newTrees[i]);
newTrees[i] = insertLengths(topo);
  }
}

//Each one of the N individuals  are tried out
//for recombination: a partner is chosen at random
//and if recombination is allowed, it is executed.
//Otherwise, it is cloned.
public static void recombination()
{
 if (print6)
 System.out.println( "\n====RECOMBINATION \n");
 //Best individual is excluded from recombination
  newTrees[0] = trees[0];

 for(int i = 1; i < N; i++)
  {
```

```
   //A partner for i is chosen
   int j = i;
   String t1 = "";
   String t2 = "";
   while (i == j)
 j =   r.nextInt(N);
   if (print6)
System.out.println( "\n==RECOMBINATION OF INDIVIDUALS "
+ i + " and " + j);
   t1 = trees[i];
   t2 = trees[j];
   newTrees[i] = recombinationMaster(t1,t2);
   correction();
  }
 if (print6)
 System.out.println("Recombination output");
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print6)
  System.out.println(trees[i]);
 }
}



//Evolution of trees involving
//recombination and mutation of topologies,
//in within and between.
private static void evolution()
{
 for(int i = 1;
 ((i < numberOfGenerations) & (continueWork)); i++)

 {
generation = i;
  if ( (i % lapse) == 0)
  {
```

```
            print7 = true;
         //  a.setEnabled(false);
 }
 else print7 = false;

 if (print7)
 System.out.println( "\n===GENERATION  = " + i);
 selectiveCloning();
 mutation();
 recombination();
 assessFitnesses();
 scaleFitnesses();

}

selectedTree = bestTree;
bestTopology = simplify(deleteLengths(bestTree));
//Selected tree is drawn
title1 = "Best tree = "
   + selectedTree;

decoderLengths(selectedTree);

//Absolute distance from the associated distance matrix
//of the tree
//to target matrix is evaluated
//d = absDistance(numberOfLeaves,targetMatrix, M);



double[][] M = associatedMatrix(selectedTree);
title2 = "Distance to target = " + bestDistance;
a = newContentPane.new DrawBiTree14();
a.createAndShowGUI();
a.setLocation(500,  100);
```

```
 if (!(continueWork))
  System.out.println( "\n \nSOLUTION HAS BEEN FOUND: \n" +
"at generation " + generation);
 System.out.println("\nSummary at  generation " + generation +
 "\nTarget tree   = \n" + targetTree
   + " \nBest tree = \n" + bestTree
   + " \nDistance to target = " + bestDistance
   + " \nBest distance at initialization = "
   + distance0
   + "\nTarget topology = \n" + targetTopology
   + "\nBest topology = \n" + bestTopology);
 printMatrix(M, "Distance matrix associated to best tree");
 printMatrix(targetMatrix,
      "Distance matrix associated to target tree");

}




//A tree topology is generated at random.
public static String randomTopology()
 {
  if (print2)
  System.out.println("Parent tree = \n" + seed
   + " -> "    + simplify(seed));
  String tree = seed;
  int limit = numberOfLeaves-3; //less than 11.
  for(int i = 0; i < limit; i++ )
  tree = generateChild(tree);
  System.out.println("Target topology = \n" + tree);
  return tree;
 }


//======================
```

```
//The best fitting tree for chosen topology is found
public static int findTheBest(String tree)
{
 String selectedTopology = simplify(tree);
 String thisTopology = "";
 String thisTree = "";
 int bestPosition = -1;

for(int i = 0; i < indexOfTopologies; i++)
 {
  thisTree = bestTreeOfTopology[i];
  thisTopology = simplify(thisTree);
  if (selectedTopology.equals(thisTopology))
  {
   bestPosition = i;
   return bestPosition;
  }
 }
return  bestPosition;
}


  /*
  //Lengths with real values are generated
  double lengthsLeaves[]  = new double[l];
  //Last two entries are redundant
  double lengthsInnerLinks[]  = new double[l];
  for(int i = 0; i < numberOfLeaves; i++)
   {
 lengthsLeaves[i] = r.nextDouble();
 lengthsInnerLinks[i] =  r.nextDouble();
   }
  */
```

```
//========= Translation in within formats ==========



//bTree in (modified) Newick’s notation
//is used to generate Record[]
//with addresses and contents.
public static void makeInput(String bTree)
{
 if (print)
 System.out.println("bTree at makeInput ////// " + bTree);
 int counter = 0;
 //length of String   tree
 int l = bTree.length();
 String thisChar = "";
 String address = "";
 String binNumbers = "01";
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + bTree.charAt(i);
if (binNumbers.contains(thisChar))
   address = address + thisChar;
if (leaves.contains(thisChar))
{
 Record[counter] = address;
     counter++;
 Record[counter] = thisChar;

counter++;
address = "";
```

```
   }
  }

 if (print)
for(int i = 0; i < counter; i = i+2 )
  System.out.println(Record[i] +
 " bbbb " + Record[i+1]);

lastUsedPosition = counter;
if (print)
System.out.println("lastUsedPosition ==== "
        + lastUsedPosition);
}



//Outputs a word in reversed order.
private static String reverser(String word)
{
 return new StringBuilder(word).reverse().toString();
}


private static String cleanTree(String tree)
{
 if (print2)
 System.out.println("Input tree at cleanTree = \n "
        + tree);
 String treeClean = "";
 String numeric = ".:0123456789";
 String plusMinus = "+-";
 String thisChar = "";
  boolean leaf = false;
 //Numerics are washed
 //length of String   tree
 int l = tree.length();
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
```

```
  {
   thisChar = "" + tree.charAt(i);
   if (numeric.contains(thisChar));
    else treeClean = treeClean + thisChar;
  }
 if (print2)
 System.out.println("Tree without numerics = \n"
+ treeClean);
 //+/- are deleted from inner nodes that accompany leaves
 l = treeClean.length();
 String treeOut = "";
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
 {
  thisChar = "" + treeClean.charAt(i);
  if (leaves.contains(thisChar))
  leaf  = true;
  if ((leaf) &  (plusMinus.contains(thisChar)));
else treeOut = treeOut + thisChar;
  if (internalNodes.contains(thisChar))
  leaf  = false;
}

 if (print2)
 System.out.println("Tree at output of cleanTree = \n"
+ treeOut);
return treeOut;

}



//The path from each leaf to node a is found.
//The input contains branch's indicator, say:
//((++Aa,+-Ba),(-+Cb,((--++Dd,--+-Fd)--c,---Ec)-b)+a)
//Output for leaf A: Aa+.
//Output for leaf C: Cba-
```

```
//Output for leaf D: Ddcba-
private static void findPaths(String tree)
{
 if (print2)
 System.out.println("Input tree at findPaths = \n" + tree);
 //length of String   tree
 int l = tree.length();
 String plusMinus = "+-";
 String thisChar;
 String nextChar;
 String path = "";
 //Branch indicative with respect to root, + else -
 String branch;
 String address ;
 //The number of links of a path to root
 int level = 0;
 String treeReversed = reverser(tree);
 if (print2)
 System.out.println("Reversed tree = " + treeReversed);

//Each leaf has a path to a
for(int k= 0; k < numberOfLeaves; k++)
{
 String leaf = "" + leaves.charAt(k);
 //Candidate nodes to be part of the path to node a
 //must have  sign +/- as prefix.
 //Or as suffix in the reversed tree.
 int position = treeReversed.indexOf(leaf);
 if (position > -1) // if leaf is in tree
 {
  if (print2)
  System.out.println("\nLeaf = " + leaf );
  address = "";
  boolean go = true;
  //branchIndicator and level are found
  for(int i = position+1 ; ((i < l) & (go)); i++ )
  {
   thisChar = "" + treeReversed.charAt(i);
```

```
   if   (plusMinus.contains(thisChar)  )
address = thisChar + address;
   else go = false;
  }
  level = address.length();
  if (print2)
  System.out.println("LeafAddressLevel  = " + leaf
                     + address + level);
  branch = "" + address.charAt(0);


  //Path from leaf to node a is found in (not reversed) tree
  position = tree.indexOf(leaf);
  thisChar = "" + tree.charAt(position);
  nextChar = "" + tree.charAt(position + 1);
  //Output
  path = "" + leaf + nextChar;
  go = true;
  if  (nextChar.contentEquals("a") ) //leaf is neighbor of a
   {
  path = path + branch;
  if (print)
  System.out.println("Path short = " + path);
   }
  else  //level > 1
  {

  //Tree is scanned char by char to get path
  String iChar = "";
  for(int i = position + 2; ((i < l) & (go)); i++ )
  {
   iChar = "" + tree.charAt(i);


   int newLevel = 0;
   if (internalNodes.contains(iChar))//InternalNodes
{
     if   (iChar.contentEquals("a")  )
```

```
   {
path  = path +  iChar;
go = false; //end
   }
  else //Internal node different than a
  {
   if (print)
   System.out.println("Internal node not a) = " + iChar );
   int rP = l -i-1; //index of iChar in treeReversed
   String  nextNewChar = "" + treeReversed.charAt(rP+1);

   if ( plusMinus.contains(nextNewChar )) //candidate node
    {
    if (print)
  System.out.println("next candidate = " + iChar);
     String addressNew = "";

     //level of candidate node iChar  is inquired
     String thatChar = "";
     boolean go2 = true;
     for(int j = rP+1 ; ((j < l) & (go2)); j++ )
      {
    thatChar = "" + treeReversed.charAt(j);
    if   (plusMinus.contains(thatChar)  )
       addressNew = thatChar + addressNew;
    else go2 = false;
      }
     if (print)
     System.out.println("AddressNew " +
                                  iChar + addressNew);
    newLevel = addressNew.length();
    //A path links a node with another in a lower level
    if  ( (newLevel < level) &
     (!(nextChar.contentEquals(iChar))))
     {
    path = path + iChar;
    level = newLevel;
    if (print)
```

```
       System.out.println("Path newLevel= " + path);
        }
      } //End of if candidate node
     }//End of else Internal node different than a
}//End of if InternalNodes
   }//End of Tree is scanned char by char

  path = path + branch;

 }//End of  else level > 1
 }//End of if leaf is in tree
 if (print2)
 System.out.println("Path completed  = " + path);
 Paths[k] = path;
}//End of each leaf
}



//Paths from leaves to node a are found
//Reverse paths are also calculated.
//Input tree in quasibinary notation
//with addresses and lengths
private static void pathFinder(String tree)
{
 //Useless information is deleted
 String tree2 = cleanTree(tree);
 //Find the path to node a from each leaf.
 //Add an indicator of the branch
 //with respect to the root: up(+) else down(-)
  findPaths(tree2);
}

//Input: a long raw string with path from leaf to node a
//plus path from node a to leaf.
//Editing is done to get true paths.
private static String grammar (String path)
{
```

```
 int n = path.length();
 String thisChar = "";
 String nextChar = "";
 String oldPath = path;
 String plusMinus = "a+-a";
 String minusPlus = "a-+a";
 String two = "";
 int N = path.length();
//Rule 1: two equal symbols produce nothing
 for(int j=1; j < N; j++)
  {
   for(int i=1; i < n-1;)
   {
    thisChar = "" + path.charAt(i);
    nextChar = "" + path.charAt(i+1);
    two = "" + thisChar + nextChar;
    if (thisChar.contentEquals(nextChar))
  {
   i = i + 2;
   path = path.replace(two, "");
  }
    else
  {
   i = i+1;
  }
  n = path.length();
   }
  }

 //Rule 2: a+-a -> a

 if  (path.contains(plusMinus))
      path = path.replace(plusMinus, "a");
 if   (path.contains(minusPlus))
     path = path.replace(minusPlus, "a");
 if (print2)
 System.out.println("input = " + oldPath +
                    " output = " + path);
```

```
 return path;
}

//The distance matrix among leaves is calculated
private static double[][] matrixBuilder()
{
double M[][] =
new double[numberOfLeaves][numberOfLeaves];
 //Path to node a from leaf
 String pathLeafToa = "";
 //Path to leaf from node a
 String pathaToLeaf = "";
 //Path from leaf to lead
 String pathLeafiToLeafj = "";
 String innerNode = "";
 double distance;
 for(int i = 0; i < numberOfLeaves; i++)
  for(int j = 0; j < numberOfLeaves; j++)
  {
   pathLeafToa = Paths[i];
   pathaToLeaf = reverser( Paths[j] );
   //Raw paths
   pathLeafiToLeafj = pathLeafToa + pathaToLeaf;
   //An editing grammar to get true paths
   pathLeafiToLeafj = grammar( pathLeafiToLeafj );
   if (i == j) M[i][j] = 0;
   else
    {
 String leaf = "" + pathLeafiToLeafj.charAt(0);
 int p = leaves.indexOf(leaf);
 distance = lengthsLeaves[p];
 int n = pathLeafiToLeafj.length();
 for(int k = 1; k < n-1; k++)
 {
  innerNode =  "" + pathLeafiToLeafj.charAt(k);
  p = internalNodes.indexOf(innerNode);
  distance = distance + lengthsInnerLinks[p];
 }
```

```
 leaf =  "" + pathLeafiToLeafj.charAt(n-1);
 p = leaves.indexOf(leaf);
 distance = distance + lengthsLeaves[p];
  M[i][j] = distance;
    }
  }
 return  M;
}



//Translation from Newick's to quasi-binary notation:
//addresses are attached to tree in Newick's notation
//Inner nodes are conserved. Lengths are conserved.
private static String toQuasiBinaryInner(String tree)
{
 if (print2)
  System.out.println("Input tree at toQuasiBinaryInner " +
                          "= \n" + tree);
 //Output
 String treeToBi = "";
 String numeric = ".:0123456789";
 //The binary address of leaf
 String address = "";
 //The name or content of the tree
 String content = "";
 String thisChar = "";
 String nextChar = "";
 String previousChar = "";
 //length of String   tree
 int l = tree.length();
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + tree.charAt(i);
   if ( i < l-1) nextChar = "" + tree.charAt(i+1);
    else nextChar = "end";
   if ( i > 0) previousChar = "" + tree.charAt(i-1);
```

```
    else previousChar = "start";


   //This char = (
   if (thisChar.contentEquals("("))
    {
     content = content + thisChar;
     address = address + '+';
     if (print)
 System.out.println("Address = " + address);
    }


 //If this char is a leaf
 if (leaves.contains(thisChar))
  {
   // System.out.println("leaf");
   treeToBi =  treeToBi + content + address + thisChar;
   content = "";
  }

 if (numeric.contains(thisChar))
 treeToBi =  treeToBi + thisChar;

 //If this char is an inner node
 if (internalNodes.contains(thisChar))
  {
   int m = address.length();
//delete last bit from the right
   if (m>0) address = address.substring(0, m-1) ;
   if (address.contentEquals("")) address = "+";
   treeToBi =  treeToBi + address + thisChar;

  }


 //This char = , and goes after an inner node
 if  (thisChar.contentEquals(","))
```

```
  {
   //System.out.println
 //("comma : "  + previousChar + thisChar );
   treeToBi =  treeToBi + thisChar ;
   if (print)
 System.out.println("Address at ,1   = " + address);

   if    (numeric.contains(previousChar))
  address = address + '-';//Add a - to the right
   if (print)
 System.out.println("Address at ,2   = " + address);
  }


 if (thisChar.contentEquals(")"))
  {
treeToBi =  treeToBi + thisChar ;
    //Diphthonge   ),
if   (nextChar.contentEquals(","))
{
     int m = address.length();
 //delete last bit from the right
 if (m >0)   address = address.substring(0, m-1) + "-";
   }
  }

}//End of for(int i...
return treeToBi;
}


//bTree  (in links with lengths notation)
//is used to generate Record[]
//with addresses and contents.
//Names of inner nodes are visible
public static void makeInputInner(String bTree)
{
 int counter = 0;
```

```
 //length of String   tree
 int l = bTree.length();
 String thisChar = "";
 String address = "";
 String binNumbers = "+-";
 String numeric = ".:0123456789";
 //String tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + bTree.charAt(i);
if (binNumbers.contains(thisChar))
   address = address + thisChar;
if ( (leaves.contains(thisChar)) ||
(internalNodes.contains(thisChar)))
{
 Record[counter] = address;
     counter++;
 Record[counter] = thisChar;

 counter++;
 address = "";
    }

if (numeric.contains(thisChar))
 {
Record[counter-1] = Record[counter-1] + thisChar;
 }

  }


 if (print)
  for(int i = 0; i < counter; i = i+2 )
  System.out.println(Record[i] +
 " bbbb " + Record[i+1]);

 lastUsedPosition = counter;
 if (print)
```

```
System.out.println("lastUsedPosition ==== "
      + lastUsedPosition);
}//End of method


//Prints  a matrix and its title
private static void printMatrix(double Matrix[][],
                        String title)
{
 System.out.println( "\n" + title );
 int m = Matrix.length;
 System.out.print(  "\t");
 for ( int j = 0; j < m; j++ )
  System.out.print( leaves.charAt(j)+ "\t");
 System.out.println();
 for ( int i = 0; i < m; i++ )
  {
   System.out.print(leaves.charAt(i ) + "\t");
   for ( int j = 0; j < m; j++ )
 System.out.print( Matrix[ i ][j]+ "\t");
   System.out.println();
  }
}


//Tree is translated
//from link notation to binary.
//Output in Record[]: even entries= addresses, odd = names.
//Associated matrix is also calculated
public static double [][] associatedMatrix(String tree)
{
 /*
 System.out.println("Tree  at translate = " + tree);
 //String NwTree = simplify(tree);
 String bTree = toBinary(tree);
 //A vector with addresses and names is composed
 makeInput(bTree);
 */
```

```
//String NwTree = simplify(tree);
 String bTree = toQuasiBinaryInner(tree);

 if (print2)
 {
 System.out.println("Input tree  at translate = "    + tree);
  System.out.println("Selected tree in " +
  "pseudo-Newick's notation = \n" + simplify(tree));
  System.out.println("Tree in quasi-binary notation: \n"
              + bTree);
 }

 //A vector with addresses and other with names are composed
 makeInputInner(bTree);
 //Paths from leaves to node a are found.
 //Reverse paths are also listed.
 pathFinder(bTree);
 //The distance matrix among leaves is calculated
 double M[][] = matrixBuilder();
 return M;
}



//The absolute distance among two matrices is calculated.
//Only triangular part of matrices are compared.
private static double absDistance(int n,
               double[][] M1, double[][] M2)
{
double abs = 0;
for(int i = 0; i < n; i++)
for(int j = i+1; j < n; j++)
 abs = abs + Math.abs(M1[i][j]-M2[i][j]);
return abs;
}
```

```
//Input : tree in link notation.
//Output:
//distance matrix associated to the leaves of the tree.
private static double[][] distanceMatrix(String tree)
{
String s = toQuasiBinaryInner(tree) ;
if (print4)
 System.out.println("\n Tree = " + s);
//Paths from leaves to node a are found.
//Reverse paths are also listed.
pathFinder(s);
//The distance matrix among leaves is calculated
double M[][] = matrixBuilder();
return M;
}



//==========================================
//==Inner class  for drawing selected tree ==
//==========================================


//This program translates a string that
//denotes a tree in Newick's notation
//to our binary notation
//(assign a zero to the left, a one to the right).

//A generator of random trees in Newick's notation
//is included.

//Resultant trees are drawn.




class DrawBiTree14 extends JPanel
```

```
{
 private static final long serialVersionUID = 1L;
 private JPanel drawingPane34;
 JScrollPane scroller;



 //=======================================



 //Vertical  size of the panel
 int verticalSize;
 int horizontalSize;
//Scale must be a power of 2
 int scale = 1;

 //Symbolic horizontal length of each stroke
 int horizontalLength = 30;

 int verticalLength = 10;



 String  NwTree = "";



 boolean print = false;



 //This is our encoding for a tree in our binary notation:
 //We use a vector of strings:
 //Even entries are the paths in the tree  of the nodes,
```

```
 //odd ones are the strings, names or contents.



 //Y-coordinates of points where strings are drawn
 int[] yCoordinate = new int[1000];



 //Set up the instructions.
 JLabel instruct1;
 JLabel instruct2;

public DrawBiTree14()
{
 super(new BorderLayout());


 JPanel instructionPanel = new JPanel(new GridLayout(0,1));


 //Set up the instructions.
 instruct1 = new JLabel(title1);
 instruct2 = new JLabel(title2);

 instructionPanel.setFocusable(true);
 instructionPanel.add(instruct1);
 instructionPanel.add(instruct2);

 //Set up the drawing area.
 drawingPane34 = new drawingPane34();
 drawingPane34.setBackground(Color.white);

 //Put the drawing area in a scroll pane.
 JScrollPane scroller = new JScrollPane(drawingPane34);
 scroller.setPreferredSize(new Dimension(500,900));
 scroller.revalidate();
```

```
 //Lay out this demo.
 add(instructionPanel, BorderLayout.PAGE_END);
 add(scroller, BorderLayout.CENTER);
}


//========================================
//============= Painting class ===========
//========================================


/** The component inside the scroll pane. */
 class drawingPane34 extends JPanel
{
 private static final long serialVersionUID = 1L;

 //Initialization of coordinates
 int centerX = 1;
 int centerY = 1;
 //Coordinates of the origin
 int x00;
 int y00;
 //Target coordinates
 int x11 ;
 int y11;
 //Horizontal and basic vertical strokes
 int H;
 int V;
 //Addresses are interpreted as paths
 String path = "";
 char direction;



//Calculate the size of a box to hold the tree
//Scrolling controls are managed
private void sizeOfTree(Graphics2D g2d)
```

```
{
 //Calculate the dimension of the strings
 //All strings have equal size.

 //Get   font in use
 Font font = g2d.getFont();
 //get metrics from the graphics
 FontMetrics metrics = g2d.getFontMetrics(font);
 //get the advance of my text in this font
 //and render context
 int lengthText = metrics.stringWidth("Aa") + 2;
 //get the height of a line of text in this
 //font and render context (some padding is included)
 int hgt = metrics.getHeight()  + 8;
 if (print)
  System.out.println("height of letters = " + hgt);

 //Horizontal and basic vertical strokes
 H = scale * horizontalLength;
 V = scale * verticalLength;

 horizontalSize = lengthText +
              (int) (H *  numberOfLeaves);
 verticalSize =
 (int) (hgt *   numberOfLeaves);


 //area = size of the graphic (width, height)
 Dimension area =
 new Dimension(horizontalSize,12*verticalSize);
 //area is set as the preferred size of the graphic
 drawingPane34.setPreferredSize(area);
 //The graphic demands a space for itself:
 //if the alloted space is not enough,
 //turn on scrolling bars.
 drawingPane34.revalidate();
 verticalLength = 2*verticalSize;
 //Coordinates of the origin
```

```
 x00 =    scale*centerX;
 y00 =   scale* centerY + verticalSize  - 600;
 //Target coordinates
 x11 =  x00 + (int) ( scale*horizontalLength);
 y11 =  y00;

}



//Tree is drawn
private void DrawTree3(Graphics2D g2d)
{
//The tree is drawn:
//A path from the origin of the tree to
//each leaf is drawn
 if (print)
 System.out.println("LastUsedPositoon at DrawTree3=  "
                                 + lastUsedPosition);
 for(int i = 0; i < lastUsedPosition; i = i+2)
  {
//Addresses will be used as paths to  be drawn
path = Record[i];
if (print)
System.out.println( i + " Path = " + path);
//Reinitialization
Dimension center = new Dimension(x11,y11);
V = scale * verticalLength;
//Path is interpreted
for(int j = 0; j < path.length(); j++)
 {
  direction = path.charAt(j);
  int x0 =  center.width;
  int y0 =  center.height;
  int x1 =  x0;
  int y1;
  if (direction == '+') //go up
   {
```

```
    y1 =  y0 + V;
    center.height = center.height + V;
   }
   else // if direction == '-'  =  go down
   {
     y1 =  y0 - V;
     center.height = center.height - V;
    } //End of if + else
    //Draw vertical stroke
    g2d.drawLine(x0, verticalSize -y0, x1,
     verticalSize- y1);
    //Draw horizontal stroke
    g2d.drawLine(x1, verticalSize - y1,
        x1 + H,  verticalSize - y1);
    //Move center
    center.width = center.width + H ;
    //Vertical stroke is divided in two
    V = V/2;
    }//End of for j
 }  //End of for: numberOfLeaves
}



//Strings are drawn
private void drawStrings(Graphics2D g2d)
{
 int counter = 0;
 //Seed is not  drawn: we lack one
 //g2d.drawString(seed,x00,verticalSize - y00);
 yCoordinate[counter] = verticalSize - y00;
 //counter++;
 //Strings are drawn:
 for(int i = 0; i < lastUsedPosition; i = i+2)
 {
  //Addresses will be used as paths to the places
  //where strings must be drawn
  path = Record[i];
```

```
  String s = Record[i+1];
  //Reinitialization
  Dimension center = new Dimension(x00,y00);
  Dimension target = center;
  V = scale * verticalLength;
  //Path is interpreted
  for(int j = 0; j < path.length(); j++)
{
 direction = path.charAt(j);
 if (direction == '+') //go up
 target.height = target.height + V;
 else // if direction == '-'  =  go down
 target.height = target.height - V;
 target.width = target.width + H ;
 V = V/2;
}//End of for j
//String is drawn
g2d.drawString(s, target.width + 1,
                      verticalSize - target.height-1);
//y-coordinates are recorded
yCoordinate[counter] = verticalSize - target.height-4;
 counter++;
  }
}


//Basic routine of the drawing procedure
private void drawBasic(Graphics2D g2d)
{
 g2d.setColor(Color.red);
 //Stem is drawn: our trees is artificially rooted.
 g2d.drawLine(x00, verticalSize - y00,
      x11,  verticalSize - y11);
 sizeOfTree(g2d);
 DrawTree3(g2d);
 drawStrings(g2d);
}
```

```
//Graphical  master : it is called automatically
//at starting and by repaint()
protected void paintComponent(Graphics g)
{
 super.paintComponent(g);
 Graphics2D g2d = (Graphics2D)g;
 //Font to draw strings
 Font font = new Font("Arial", Font.PLAIN, 12);
 g2d.setFont(font);
 g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
  RenderingHints.VALUE_ANTIALIAS_ON);
 drawBasic(g2d);
}


}//End of double inner class drawingPane34



/**
* Create the GUI and show it.  For thread safety,
* this method should be invoked from the
* event-dispatching thread.
*/
 void createAndShowGUI()
{
 //Create and set up the window.
 JFrame frame =
   new JFrame(titleFrame);
 frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

 //Create and set up the content pane.
 //Constructor of the main application is used.
 JComponent newContentPane = new DrawBiTree14();
 //content panes must be opaque
 newContentPane.setOpaque(true);
 frame.setContentPane(newContentPane);
```

```
 //Display the window.
 frame.pack();
 frame.setVisible(true);
}

}//End of  inner class DrawBiTree14

}//End of main class K44 EvoFull3
```

54, page 244. The next code generates a mutation in a tree by deleting leaves at random and by reinserted them also at a random place. It also cause mutations in the lengths of links.

```
//Program K54 TopoMutation4.
//This program generates a random binary tree
//and produces a mutation, whose output must be
//a binary tree  over the same set of leaves.

//At the beginning it is decided with which order
//of insertion of leaves to work:
//normal, reversed or permuted.
//Next, a tree is synthesized
//and a mutation is made.
//
//A mutation results from deleting
//a given number of leaves
//to reinsert them anew in
//other places chosen at random.
//Leaves can be deleted in reversed order of insertion
//or at random.
//Lengths of links are also mutated.

//Seed tree, with three leaves, is conserved.
//Example in ordinary order ((Aa,Ba),Ca).
```

```
import java.util.Random;


public class TopoMutation4
{

 //Link’s notation for topologies
 private static   String   seed =
 "((Aa,Ba),Ca)";
 private static  String leaves =
 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
 private static String internalNodes =
 "abcdefghijklmnopqrstuvwxyz";
 //Next leaf to be grafted
 private static  int nextLeafIndex = 2;
 //Next internal node to be inserted
 private static  int nextIntNodeIndex = 0;

 //Number of leaves to be inserted to seed:
 //to get a tree with 4 leaves, add one leaf to seed
 private static  int nAdditionalLeaves = 8;
 //Total number of leaves
 private static int numberOfLeaves;

 //Number of leaves to be deleted and reinserted
 //to create a mutation.
 //must be less than nAdditionalLeaves
 private static int nUndoRedo = 0;

 //Leaves that remain in the topology
 private static  String presentLeaves =   "ABC";
 private static String presentNodes =  "a";
 //Founder leaves ABC or equivalent cannot be removed
 private static String deletableLeaves = "";
 //Deleted leaves are then reinserted
 private static String reinsertingLeaves = "";
 private static String reinsertingNodes = "";
```

```
 private static int maxNumberOfLeaves = leaves.length();

 //Length of the link that glues a leaf to the tree
 private static int lengthsLeaves[]  =
                 new int[maxNumberOfLeaves];
 //Length of the link that glues
 //an inner node to another one
 private static int lengthsInnerLinks[]  =
                new int[maxNumberOfLeaves];



 private static  Random r = new Random();
 //The control parameter of the random generator for int
 private static int maxLength = 10;

 private static boolean print1 = false;
 private static boolean print2 = true;


 //This is the original topology:
 private static  String original = "";

//MUTATION PARAMETERS===========


 //Are you going to work with the given order
 //of the alphabet of leaves
 //a reverse one, or with a permutation?
 private static double pGivenOrder = 1;
 private static double pReverseOrder = 0;
 //a must be less than 1
 //Implicit: probability of permutation = a =
 //1 - (pGivenOrder+pReverseOrder).
 private static double a = pGivenOrder + pReverseOrder;

 //Mutation of topology=
 //           deleting leaves + reinsertion at random.
```

```
 //Choose one of two forms of deleting:
 //reverse order to that of synthesis else random.
 //One is given, the other is the complement to 1
 private static  double pReverseSynth = 0;

 //You can mutate only lengths else
 //topology + lengths
 private static double pMutateOnlyLengths = 0.5;

 //If lengths of links are mutated
 //Mutation rate per link per generation
 static private double    m = 0.5;
 //End of mutation parameters


//String s ( a char) is deleted from mother
public static String delete(String s, String mother)
{
 String output = "";
 int l = mother.indexOf(s);
 output = mother.substring(0, l)
 + mother.substring(l+1) ;
 return output;
}



//String leaves is reversed
public static String reverse(String leaves)
{
 String reversed = "";
 int l = leaves.length();
 for(int i = l-1; i >= 0; i--)
  {
String c = "" + leaves.charAt(i);
reversed = reversed + c;
  }
 if (print1)
```

```
 System.out.println("Reverse order = " + reversed);
 seed =  "((Za,Ya),Xa)";
 if (print1)
 System.out.println("seed = " + seed);
 presentLeaves =   "ZYX";
return reversed;
}


//String leaves is permuted
public static String permuteOrder(String leaves)
{
 if (print1)
 System.out.println("original and permuted order = \n"
                          + leaves);
 String permuted = "";
 int l = leaves.length();
 int m = l;
 for(int i = 0; i < l; i++)
  {
int d = r.nextInt(m) ;
String c = "" + leaves.charAt(d);
leaves = delete(c,leaves);
m = m-1;
permuted = permuted + c;
  }
 if (print1)
 System.out.println(permuted);
 String first = "" + permuted.charAt(0);
 String second = "" + permuted.charAt(1);
 String third = "" + permuted.charAt(2);
 seed = "((" +first + "a," +second + "a)," +third + "a)";
 if (print1)
 System.out.println("seed = " + seed);
 presentLeaves =   first + second + third;
 return permuted;
}
```

```
 //Detect internal nodes, different than leaves
 private static boolean charIsInternalNode(char c)
 {
  String s = "" + c;
  if (internalNodes.contains(s)) return true;
  else return false;
 }

//Simplification from Link notation to Newick's notation:
 // (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTree)
{
 String tree = "";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
  }
 return tree;
}

//Rule 2 is used to generate a new tree:
//(S)I -> ((S)N,FN)I.
//S = subtree, I = internal node, F = new leaf,
//N = new internal node.
//Rule 2 grafts new branch FN to link (S)I
//i is the position of char I in string seedTree
private static String useRule2(String seedTree, int i,
        char F, char N)
{
 String newTree = "";
```

```
 char I = seedTree.charAt(i);
 if (print1)
 System.out.println("i = " + i + " I = " + I);
 //Subtree S of link (S)I must be read
 String S = "";
 //Counter of left seedhesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = i;
 boolean go = true;
 //All right seedheses to the near left of I are captured
 while (go)
  {
   position = position -1;
   String a = "" + seedTree.charAt(position);

   if (print1)
   System.out.println("a = " + a + " p = " + pr);
   if (a.equals(pr) )
   {
counter++;
S = a + S;
   }
   else go = false;
  }//End of while
 if (print1)
 System.out.println("counter = " + counter);
 //The complete subtree is captured:
 //seedheses must balanced
 while (counter > 0)
  {
   String a = "" + seedTree.charAt(position);
   if (a.equals(pr))   counter++;
   if (a.equals(pl))  counter--;
   S = a + S;
   if (print1)
```

```
   System.out.println("Subtree = " + S);
   position = position -1;
   }
 //Replacement rule 2:
 //((S)n,Fn)i -> (S)i
//where   F are leaves, n and i are  inner nodes,
//(S) is a complex structure.
 String SnFni  =   S  + I;
 String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
 //Rule 1 is executed
 newTree = seedTree.replace(SnFni, replacement);
 //If I links to a complex structure
 return newTree;
}

//Input: seedTree, a tree in Newick's notation.
//Output : newTree =  seed tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String seedTree)
{

 nextLeafIndex++;
 char nextLeaf = leaves.charAt(nextLeafIndex);
 presentLeaves = presentLeaves + nextLeaf;
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = internalNodes.charAt(nextIntNodeIndex);
 presentNodes = presentNodes + nextIntNode;

 //link to be replaced
 String link = "";
 //String link is replaced by what:
 String replacement = "";
 //new tree generated by replacement
 String newTree = "";
 //length of seed tree
```

```
 int l = seedTree.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
  {
   //seedTree is scanned char by char
   char I = seedTree.charAt(i);
//Is I an internal node that denotes a link?
if (charIsInternalNode(I)) counter++;
  }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
  {
    //seedTree is scanned char by char
char I = seedTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = seedTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(seedTree, i,
               nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
```

```
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTree = seedTree.replace(link, replacement);
 }
 if (print1)
System.out.println(newTree + " -> "
                       + simplify(newTree));


 i = l;
   }
  counter++;
}
  }
 return newTree;
}


//Input: seedTree, a tree in Newick's notation.
//Output : newTree =  seed tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String reinsertLeaf(String seedTree)
{
 nextLeafIndex++;
 char nextLeaf = reinsertingLeaves.charAt(nextLeafIndex);
 presentLeaves = presentLeaves + nextLeaf;
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = reinsertingNodes.charAt(nextIntNodeIndex);
 presentNodes = presentNodes + nextIntNode;

 //link to be replaced
 String link = "";
 //String link is replaced by what:
 String replacement = "";
 //new tree generated by replacement
 String newTree = "";
```

```
 //length of seed tree
 int l = seedTree.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
  {
   //seedTree is scanned char by char
   char I = seedTree.charAt(i);
   //Is I an internal node that denotes a link?
   if (charIsInternalNode(I)) counter++;
  }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
  {
    //seedTree is scanned char by char
char I = seedTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = seedTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(seedTree, i,
                nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
```

```
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTree = seedTree.replace(link, replacement);
 }
 if (print1)
System.out.println(newTree + " -> "
                       + simplify(newTree));


 i = l;
   }
  counter++;
}
  }
 return newTree;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply the antirule1:
//(An,Fn)i  -> Ai
private static String  antiRule1(String tree,
int positionLeaf, int positionNode)
{
 if (print1)
 System.out.println("AntiRule1");
 int beginIndex = positionLeaf - 4;
 int endIndex = positionNode + 3;
 String AnFni = tree.substring(beginIndex, endIndex);
 if (print1)
 System.out.println("Anfni = " + AnFni);
 String Ai = "" + tree.charAt(beginIndex+1)
                + tree.charAt(endIndex-1);
 if (print1)
 System.out.println("Ai = " + Ai);
 tree = tree.replace(AnFni, Ai);
 if (print1)
```

```
 System.out.println("Output tree = \n" + tree);
 return tree;
}



//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply the antirule2:
//((S)n,Fn)i -> (S)i
//positionNode refers to the last appearing of node n
private static String  antiRule2(String tree,
int positionLeaf, int positionNode)
{

 if (print1)
 System.out.println("AntiRule2");
 //Leaf to be deleted
 String leaf = "" + tree.charAt(positionLeaf);
 if (print1)
 System.out.println("\nDeletion = " + leaf);
 //Accompanying inner node
 String innerNode = ""
           + tree.charAt(positionNode);
 int firstPosition = tree.indexOf(innerNode);
 String newTree = "";
 String i = ""; //BUG?????
 if (positionNode +2 < tree.length())
        i = ""+ tree.charAt(positionNode+2);
 if (print1)
 System.out.println("FirstPosition = " +
                     firstPosition + " i = " + i);
 //Subtree S of link (S)I must be read
 String S = "";
 //Counter of left seedhesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
```

```
 //Position to be scanned
 int position = firstPosition;
 boolean go = true;
 //All right seedheses to the near left of I are captured
 while (go)
  {
   position = position -1;
    String as = "" + tree.charAt(position);

   if (print1)
   System.out.println("a = " + as + " p = " + pr);
   if (as.equals(pr) )
   {
counter++;
S = as + S;
   }
   else go = false;
  }//End of while
 if (print1)
 System.out.println("counter = " + counter);
 //The complete subtree is captured:
 //seedheses must balanced
 while (counter > 0)
  {
   String as = "" + tree.charAt(position);
   if (as.equals(pr))   counter++;
   if (as.equals(pl))  counter--;
   S = as + S;
   if (print1)
   System.out.println("Subtree = " + S);
   position = position -1;
   }
  String n = innerNode;
 //Replacement antiRule 2:
 //((S)n,Fn)i -> (S)i
 //where A and F are leaves, n and i are  inner nodes,
 //(S) is a complex structure.
 String SnFni  =  "("+ S  + n + "," + leaf + n + ")" + i ;
```

```
 String Si = S + i;
 if (print1)
  System.out.println("\nSnFni = " + SnFni + "\n Si = " + Si);
 //Rule 1 is executed
 newTree = tree.replace(SnFni, Si);
 if (print1)
 System.out.println("Output tree = \n" + newTree);
 return newTree;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply   antirule3:
//(Fn,An)i  -> Ai
private static String  antiRule3(String topo,
int positionLeaf, int positionNode)
{
 if (print1)
System.out.println("AntiRule3"
+ "\nInput topology = \n" + topo);
int beginIndex = positionLeaf - 1;
int endIndex = positionNode + 6;
String FnAni = topo.substring(beginIndex, endIndex);
if (print1)
System.out.println("FnAni = " + FnAni);
String Ai = "" + topo.charAt(beginIndex+4)
          + topo.charAt(endIndex-1);
if (print1)
System.out.println("Ai = " + Ai);
topo = topo.replace(FnAni, Ai);
if (print1)
System.out.println("Output topology = \n" + topo);
return topo;
}
```

```
//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply antirule4:
//(Fn,(S)n)i -> (S)i
private static String  antiRule4(String tree,
int positionLeaf, int positionNode)
{

//Leaf to be deleted
String leaf = "" + tree.charAt(positionLeaf);

//Accompanying inner node
String innerNode = ""
            + tree.charAt(positionNode);
if (print1)
System.out.println("\nAntiRule4"
+ "\nLeaf to be deleted = " + leaf
 + " at position " + positionLeaf
+  "\nAccompanying node = " + innerNode
 + " at first position " + positionNode);
//Second position of accompanying node
int positionNode2 = 0;
int l = tree.length();
boolean go = true;
//tree is scanned char by char
for(int i = positionNode + 1; i < l; i++ )
{
 String c = "" + tree.charAt(i);

 if (go)
 {
  if (c.equals(innerNode))
 {
  positionNode2 = i;

        go = false;
        if (print1)
        System.out.println(
```

```
       "Second position of accompanying node = " +
          positionNode2 + " i = " + i);
 }
 }

}


int beginIndex = positionLeaf - 1;
int endIndex = positionNode2+2;
String i = "";
if (endIndex < tree.length())
         i = ""+ tree.charAt(endIndex);

if (print1)
System.out.println("EndIndex = " +
                     endIndex + " i = " + i);

String FnSni  = tree.substring(beginIndex, endIndex) + i;
if (print1)
System.out.println("FnSni = " + FnSni);
String S = tree.substring(beginIndex+4, endIndex-2);
if (print1)
System.out.println("S = " + S);
String Si = S + i;
if (print1)
 System.out.println("Si = " + Si);

String newTree = tree.replace(FnSni, Si);
if (print1)
 System.out.println("Output tree = \n" + newTree);
//System.exit(0);
return newTree;
}



//The last inserted  leaf is deleted from tree.
```

```
private static String deleteLastLeaf(String tree)
{
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 if (print1)
 System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
        + internalNodes.charAt(nextIntNodeIndex);
 if (print1)
 System.out.println("Accompanying inner  node = "
                + innerNode);
 //new tree = output
 String newTree = "";
 //length of  tree as string
 int l = tree.length();
 //Last positions are found
 int positionLeaf = -1;
 int positionNode = -1;
 //tree is scanned char by char
 for(int i = 0; i < l; i++ )
   {
 String c = "" + tree.charAt(i);
 if (c.equals(leaf)) positionLeaf = i;
 if (c.equals(innerNode)) positionNode = i;
   }
 if (print1)
 System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
 if (print1)
 System.out.println("Position of " + innerNode +
      " = " + positionNode);
 int firstPosition = tree.indexOf(innerNode);
 //Check for the char before firstPosition
 char a = tree.charAt(firstPosition - 1);
 //If I links to a complex structure
 if (a == ')')
   newTree = antiRule2(tree,positionLeaf,positionNode);
```

```
  else  //Apply antiRule1 for simple cases
  newTree = antiRule1(tree,positionLeaf,positionNode);
 nextLeafIndex--;
 nextIntNodeIndex--;
 return newTree;
}



//A random leaf is deleted from tree.
private static String deleteRandomLeaf(String tree)
{
 if (print1)
 {
System.out.println("\nMethod deleteRandomLeaf"
 + "\ntree = " + tree
+ "\ndeletableLeaves = " + deletableLeaves);
 }
int l = deletableLeaves.length();
int position = r.nextInt(l);
//Leaf to be deleted
String leaf = "" + deletableLeaves.charAt(position);
int nodeIndex = tree.indexOf(leaf) + 1;
//Accompanying inner node
String innerNode = ""
        + tree.charAt(nodeIndex);

deletableLeaves = delete(leaf,deletableLeaves);
presentLeaves = delete(leaf,presentLeaves);
reinsertingLeaves = reinsertingLeaves + leaf;
reinsertingNodes = reinsertingNodes + innerNode;


if (print1)
System.out.println("\nLeaf to be deleted = " + leaf
+ "\nReinserting leaves = " + reinsertingLeaves
     + "\nAccompanying inner  node = "   + innerNode
     + "\nReinserting nodes = " + reinsertingNodes);
```

```
//new tree = output
String newTree = "";
//length of  tree as string
l = tree.length();
//Last positions are found
int positionLeaf = tree.indexOf(leaf);
int positionNode = positionLeaf + 1;


if (print1)
System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
if (print1)
System.out.println("Position of " + innerNode +
      " = " + positionNode);
//Which set of deleting rules to apply:
//1 and 2 else 3 and 4?
String previousChar = "" + tree.charAt(positionLeaf-1);
if (previousChar.equals(",")) //Apply antiRules 1 and 2
{
if (print1)
System.out.println("Apply antiRules 1 and 2");
int firstPosition = tree.indexOf(innerNode);
//Check for the char before firstPosition
char a = tree.charAt(firstPosition - 1);
//If I links to a complex structure
if (a == ')')
newTree = antiRule2(tree,positionLeaf,positionNode);
else  //Apply antiRule1 for simple cases
  newTree = antiRule1(tree,positionLeaf,positionNode);
nextLeafIndex--;
nextIntNodeIndex--;
}
else //Apply antiRules 3 and 4
{
if (print1)
System.out.println("Apply antiRules 3 and 4");
int firstPosition = tree.indexOf(innerNode);
```

```
//Check for the char before firstPosition
char a = tree.charAt(firstPosition + 2);
//If I links to a complex structure
if (a == '(')
 newTree = antiRule4(tree,positionLeaf,positionNode);
    else  //Apply antiRule3 for simple cases
newTree = antiRule3(tree,positionLeaf,positionNode);
nextLeafIndex--;
nextIntNodeIndex--;
}

if (print1)
System.out.println("Tree after deletion" + newTree);
return newTree;
}



//Mutation =
//Random deletions are reinserted.
public static String  mutationRandomDeletion(
           String tree, int nAdditionalLeaves)
{
 //Some leaves are deleted: random order
 if (print1)
 System.out.println("Some leaves are deleted AT RANDOM");
 for(int i = 0; i < nUndoRedo; i++ )
 {
  //Last inserted leaf is deleted.
  tree = deleteRandomLeaf(tree);
 }
 //Deleted leaves are reinserted
 nextLeafIndex = -1;
 nextIntNodeIndex = -1;
 if (print1)
 System.out.println("Deleted leaves are reinserted");
 for(int i = 0; i < nUndoRedo; i++ )
```

```
   {

     tree = reinsertLeaf(tree);
   }
 return tree;
}


//Mutation:
//some leaves are deleted and then reinserted
//in random order.
//Deleting can happen in either of two ways:
//Reversing the order of synthesis else
//in random order.
public static String mutationOfTop(String topology,
                       int nAdditionalLeaves)
{
original = topology;
if (print1)
{
System.out.println("\nMethod mutationOfTop" +
"\nPresentLeaves =  "
                   + presentLeaves);
}

//Seed leaves are not removed
deletableLeaves = presentLeaves.substring(3);
//Number of leaves to be deleted and then reinserted
if (nAdditionalLeaves == 1) nUndoRedo = 1;
else
nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;
if (print1)
System.out.println("\nNumber of leaves to delete =  "
                 + nUndoRedo);

double s = r.nextDouble();
if ( s < pReverseSynth)//Choose deletion in reversed order
{
```

```
//Some leaves are deleted:
//reversed order to that of synthesis.
 if (print1)
System.out.println("Leaves are deleted " +
                        "in reverse order");
for(int i = 0; i < nUndoRedo; i++ )
 {
  //Last inserted leaf is deleted.
  topology = deleteLastLeaf(topology);
  System.out.println(topology);
 }
//deleted leaves are reinserted
if (print1)
System.out.println("\nDeleted leaves are reinserted");

for(int i = 0; i < nUndoRedo; i++ )
{
  topology = generateChild(topology);
}
}
else //Choose  at random leaves to be deleted
topology = mutationRandomDeletion(topology, nUndoRedo);
//Output
String mutant = topology;
if (print1)
System.out.println("\nOriginal and mutant topologies = \n"
  + original + "\n" + mutant);
if (print1)
{
if (original.equals(mutant))

  System.out.println("Mutation made nothing");
   else
  System.out.println("\nMutation worked");
}
return mutant;
}
```

```
//Random lengths are added to a tree in Link notation
public static void generateLengths()
{
 //Lengths with digit values are generated
 for(int i = 0; i < numberOfLeaves; i++)
  {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
  }
}



//Random lengths are added to a tree in Link notation
public static String insertLengths(String treeLinkN)
{
 String thisChar = "";
 String newTree = "";
 //length of String   treeLinkN
 int m = treeLinkN.length();
 //TreeLinkN is scanned char by char
 for(int i = 0; i < m; i++ )
 {
  thisChar = "" + treeLinkN.charAt(i);
  newTree = newTree + thisChar;
  if (print1)
   System.out.println(newTree.length() + " " + newTree);
  //Lengths are appended
  if (leaves.contains(thisChar))
   {
    int k = leaves.indexOf(thisChar);
    double j = lengthsLeaves[k];
    newTree = newTree + ":" + j;
   }
  if (internalNodes.contains(thisChar))
  {
```

```
   int k = internalNodes.indexOf(thisChar);
   double j = lengthsInnerLinks[k];
   newTree = newTree + ":" + j;
  }
 }
 return newTree;
}


//Lengths are deleted from newTree
private static String deleteLengths(String newTree)
{
 String tree = "";
 String numeric = ".:0123456789";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
   char c = newTree.charAt(i);
   String s = "" + c;
   //Is c numeric: delete it!
   if (numeric.contains(s)); //add char c to output
  else tree = tree + c;
  }
 return tree;
}


//Input: string tree with complete information.
//Work: lengthsLeaves[] and lengthsInnerLinks[]
//are read from tree.
private static void decoderLengths(String tree)
{
 if(print1)
  System.out.println(
"Input tree at decoderLengths = \n "   + tree  );
 //Delete :
```

```
 tree = tree.replaceAll(":", "");
 if(print1)
  System.out.println(
"Input tree without : = \n "   + tree  );
 int n = tree.length();
 String thisChar = "";
 String nextChar = "";
 String numbers = ".0123456789";
 boolean leaf = false;
 boolean node = false;
 String Leaf = "";
 String numberLeaf = "";
 String numberNode = "";
 String Node = "";
 int j ;
 double f;

 //All chars in tree are scanned
 for(int i=0; i < n; i++)
  {
   thisChar = "" + tree.charAt(i);
   if (i < n-1) nextChar  = "" + tree.charAt(i+1);
  //Do we have a leaf?
  if (leaves.contains(thisChar))
   {
 leaf = true;
 Leaf = "" + thisChar;
   }

  //Do we have an internal node?
  if (internalNodes.contains(thisChar))
   {
 node = true;
 Node = "" + thisChar;
   }

  //Do we have a number that belongs in a leaf?
  if ( (leaf) & (numbers.contains(thisChar)))
```

```
{
 numberLeaf = numberLeaf + thisChar;
}

  //Do we have a number that belongs in a leaf?
  if ( (node) & (numbers.contains(thisChar)))
{
 numberNode = numberNode + thisChar;
}

  //Must we end accumulation of numbers of a leaf?
  if ( (leaf) & (internalNodes.contains(nextChar)))
{
  leaf = false;
  j = leaves.indexOf(Leaf);
  if (print1)
  System.out.println("Leaf = " + Leaf +
  " numberLeaf = " + numberLeaf);
  f = new Double(numberLeaf);
  lengthsLeaves[j] = (int) Math.floor(f);
  numberLeaf = "";
}

  //Must we end accumulation of numbers of a node?
  if ( (node) & (!(numbers.contains(nextChar))))
{
  node = false;
  j = internalNodes.indexOf(Node);
  if (print1)
  System.out.println("Node = " + Node);
  f = new Double(numberNode);
  lengthsInnerLinks[j] = (int) Math.floor(f);
  numberNode = "";
}

  }
}
```

```
//Lengths undergo mutation
private static  void mutateLengths()
{
 //Mutation in links that connect leaves
 for(int k = 0; k < numberOfLeaves; k++)
  {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsLeaves[k] = r.nextInt(maxLength);

  }
 //Mutation in links that connect inner nodes
 for(int k = 0; k < numberOfLeaves-2; k++)
  {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsInnerLinks[k] = r.nextInt(maxLength);

  }
}



//Type of  mutation is decided and executed:
//mutation of lengths of links
//else full mutation (topology and lengths of links).
private static String mutateTree(String tree)
{
 //Topology is read from tree
 String topology = deleteLengths(tree);
 //Lengths are read from tree and recorded in vectors
 //lengthsLeaves[] and lengthsInnerLinks[]
 decoderLengths(tree);
 mutateLengths();
 String  mutantTree = "";
 //Mutate lengths else topology ( + mutated lengths)?
 double s = r.nextDouble();
```

```
 if ( s < pMutateOnlyLengths) //only lengths are mutated
{
 if (print1)
 System.out.println("\n====Mutation of lengths");

  mutantTree =  insertLengths(topology);

  if (print1)
  {
  System.out.println("Input tree =  " + tree);
  System.out.println("Topology =  " + topology);
  System.out.println("Output tree= " + mutantTree);
  }
 }
  else //topology + lengths are mutated
{
 if (print1)
  System.out.println("\n====Mutation of topology");


     //Mutation of topology
 String mutantTopology = mutationOfTop(topology,
                           nAdditionalLeaves);
 //Mutant lengths are reinserted
 mutantTree =  insertLengths(mutantTopology);
}

 return mutantTree;
}

public static void main(String[] args)
{

 //Election is made
 double s = r.nextDouble();
 if ( s < pGivenOrder); //Do nothing = order is not changed
  else if ( s < a )    leaves = reverse(leaves);
 else leaves = permuteOrder(leaves);//permute if  s > a
```

```
 //Change of format
 if (print1)
 System.out.println("seed topology = \n" + seed
 + " -> "    + simplify(seed));

 //A topology   is generated
 String topology = seed;
 for(int i = 0; i < nAdditionalLeaves; i++ )
   topology = generateChild(topology);
 //This original topology  is next subject to mutation
 original = topology;

 //Lengths are added to topology to get a tree
 numberOfLeaves =    3 + nAdditionalLeaves;
 generateLengths();
 String originalTree = insertLengths(original);
 //Tree undergoes mutation
 String mutantTree  =  mutateTree(originalTree);
 String mutantTopology = simplify(deleteLengths(mutantTree));
 if (print2)
 {
 System.out.println( "\nOriginal topology = \n" +
              simplify(original)
 + "\nMutant topology = \n" + mutantTopology);
 System.out.println( "\nOriginal tree = \n" + originalTree
 + "\nMutant tree = \n" + mutantTree);
 }
}
}//End of main class K54 TopoMutation4
```

55, page 244. The next code uses evolution to propose an approximated solution to the problem of fitting a tree to a distance matrix. It is Lamarckian because mutation and recombination are programmed to produce viable outputs. All possible outputs have equal probability of occurrence. In the realm of useless programs, this is an improvement over its predecessor.

```
//Program K55 EvoFull5
/*
 * We implement here
```

```
 * a full fledged evolutionary process
 * to propose an approximate solution to the problem of
 * fitting a tree to a distance matrix.
 *
 * The program runs a certain number of generations,
 * reports the tree that best approximates the  solution,
 * draws the  best approximation tree together with
 * the target tree  and halts.
 *
 * We generate a random tree topology
 * and attach random integer lengths to links.
 * In that way we get the target tree, T.
 * Next, we find the associated distance matrix D  to T.
 * Our problem is:
 *
 * Find tree t such that the absolute distance between
 * matrix D and the associated matrix to t is a minimum.
 * Absolute distance between M and N = sum |Mij-Nij|.
 * This distance defines the FITNESS of each tree.
 * It decreases as good performance augments.
 * The best possible fitness is zero.
 *
 * We run the next test:
 * If we begin with a tree T, compute D, and then t,
 * we expect to return with the initial tree: t = T.
 *
 *
 * To associate a distance matrix to a tree,
 * we calculate the distance in within
 * each pair of nodes (leaves + inner nodes).
 * To that aim, we begin with the first node and
 * walk over the tree until the second node
 * and add the lengths of all traversed links.
 * The resulting number is the  distance.
 *
 *
 * Formats = Link and Newick's notations. And variants.
 *
```

```
* An automatic halting option has been added to be applied
* if perfect matching is found.
* The program accepts for input
* a distance matrix with integer values.
* A test for being a distance matrix is run.
*
* To find the tree that best fit D,
* we  use evolution as follows:
*
* 1) Initialization:
*  We generate N topologies  on the same
*  set of leaves. By adding   lengths to links,
*  every topology gets represented by a tree.
* The fitness of each tree
* is calculated and the best is selected.
*  If the fitness of the best is zero,
*  the program halts and the tree is reported.
*
* 2) Evolution is composed of rounds of
*     cloning according to fitness,
*     random mutation and recombination.
*       After each generation, fitness is checked
*     for accomplishment.
*
*
* Mutation is Lamarckian:
* it always produces viable mutants.
* Topologies as well as links can mutate.
*
* A mutation results from deleting
* a given number of leaves
* chosen at random
* to reinsert them anew in
* other places chosen at random.
* This creates a mutant topology.
* To mutate length of links:
* Lengths are read from tree and
* mutated separately.
```

```
 * Then, they are reinserted.
 *
 *
 * Recombination is Lamarckian: it is authorized only
 * when it produces viable recombinants.
 * All their subtopologies are calculated
 * and packed in arrays.
 * The sets of leaves and of inner nodes
 * of each subtopology are found.
 * Subtopologies are compared by pairs,
 * one from each topology.
 * If they happen to have the same set of leaves
 * and of inner nodes, recombination is authorized:
 * the  subtrees that correspond with
 * the subtopologies are swapped.
 *
 * If more than one pair of subtopologies authorize
 * recombination, one is chosen at random.
 * A hashtable is used to memorize those pairs that do it.
 * Original topologies serves as subtopologies
 * that authorize recombination but they do not count.
 *
 *
 */



import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;

import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Random;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
```

```
import javax.swing.JScrollPane;

import java.awt.*;

import javax.swing.JComponent;


public class EvoFull5 extends JPanel

{

 private static final long serialVersionUID = 1L;



 //===========Most important parameters ===========

 //Max number of generations to run evolution
 static private int numberOfGenerations = 1000000;
//Report Best tree every lapse generations
static private int lapse = 1000;


 //Number of leaves to be inserted to seed:
 //to get a tree with 4 leaves, add one leaf to seed
 private static  int nAdditionalLeaves = 5;



 //Number of topologies = number of trees that
 //conform the evolving population
 private static  int N = 50;


 static private long generation = 0;

 //=============================================
```

```
//MUTATION PARAMETERS===========



//Mutation of topology=
//          deleting leaves + reinsertion at random.
//Choose one of two forms of deleting:
//reverse order to that of synthesis else random.
//One is given, the other is the complement to 1
private static  double pReverseSynth = 0;

//You can mutate only lengths else
//topology + lengths
private static double pMutateOnlyLengths = 0.5;

//If lengths of links are mutated
//Mutation rate per link per generation
static private double    m = 0.5;
//End of mutation parameters




//================================================


 //The program accepts a distance matrix as input
 //else
 //it generates a random tree, calculates the distance
 //matrix associated to leaves and posits it
 //as target matrix.
 private static boolean inputMatrix = false;
```

```
//Input distance matrix:
//Example 5.4, pag 99, Isaev (2001)
//Introduction to Mathematical Methods in Bioinformatics
//Springer
private static double targetMatrix[][]
= {{0,8,3,14,10,12},
  {8,0,9,10,6,8},
  {3,9,0,15,11,13},
  {14,10,15,0,10,8},
  {10,6,11,10,0,8},
  {12,8,13,8,8,0}};

/*
private static double targetMatrix5[][]
= {{0,8,3,14,10},
  {8,0,9,10,6},
  {3,9,0,15,11},
  {14,10,15,0,10},
  {10,6,11,10,0}};

private static double targetMatrix4[][]
= {{0,8,3,14},
  {8,0,9,10},
  {3,9,0,15},
  {14,10,15,0}};

  */

//Number of leaves that must go in all trees > 3
private static int numberOfLeaves;

//Everything is built by recursion over this tree
private static   String   seed =
   "((Aa,Ba),Ca)";

//The best distance found in initialization
private static  int  distance0;
```

```
private static  String leaves =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
private static String internalNodes =
"abcdefghijklmnopqrstuvwxyz";

//Next leaf to be grafted
private static  int nextLeafIndex = 2;
//Next internal node to be inserted
private static  int nextIntNodeIndex = 0;


private static int maxNumberOfLeaves = leaves.length();




//Number of leaves to be deleted and reinserted
//to create a mutation
private static int nUndoRedo = 0;
//Basic set of non removable leaves
private static  String basicLeaves =   "ABC";
//Leaves   already present in the topology
private static  String presentLeaves =   basicLeaves;
//Full set of leaves that conform the topology
private static  String allLeaves;

private static String presentNodes =  "a";
//Founder leaves ABC or equivalent cannot be removed
private static String deletableLeaves = "";
//Deleted leaves are then reinserted
private static String reinsertingLeaves = "";
private static String reinsertingNodes = "";
```

```
//This is the original topology:
private static  String original = "";




//Data for binary tree
private static String[] Record =  new String[20*N];


private static  int indexOfTopologies;
private static String topologies[] = new String[N];
private static String trees[] = new String[N];
private static String newTrees[] = new String[N];


private static String[] bestTreeOfTopology
                            = new String[N];

//Last filled position  in Record[]
private static int lastUsedPosition;



  //Length of the link that glues a leaf to the tree
 private static int lengthsLeaves[]  =
               new int[maxNumberOfLeaves];
 //Length of the link that glues
 //an inner node to another one
 private static int lengthsInnerLinks[]  =
              new int[maxNumberOfLeaves];


 private static Random r = new Random();
 //The control parameter of the random generator for int
 private static int maxLength = 10;
```

```
 //Paths from each leaf to node a
 private static String[] Paths =
             new String[maxNumberOfLeaves];

 private static String targetTree;
 private static String targetTopology;
 private static String bestTopology;


 //=======Evolution parameters============



static private double distance[] = new double[N];
static private double fitness[] = new double[N];



//Index of the best
static private int best;
//The best tree for a given topology
static private String bestTree = "";
//Distance from bestTree to target
static private double bestDistance ;
//Boolean flag for termination of job
static private boolean workDone = true;



//=========Debugging help ===============

private static boolean print  = false;
private static boolean print1 = false;
private static boolean print2 = false;
private static boolean print3 = false;
private static boolean print4 = false;
//Turn on print5 firstly
private static boolean print5 = false;
```

```
 private static boolean print6 = true;
 private static boolean print7 = true;


 //==============

 private static String titleFrame = "";
 private static String title1 = "";
 private static String title2 = "";
 private static String title3 = "";


 private static EvoFull5 newContentPane = new EvoFull5();

 private static DrawBiTree16 draw;


  //=======Constructor of main class

  public EvoFull5()
  {

  }//End of constructor




public static void main(String[] args)
{
   // Schedule a job for the event-dispatching thread:
   // creating and showing this application's GUI.
   javax.swing.SwingUtilities.invokeLater(new Runnable()
    {
     public void run()
     {
      //If input is a distance matrix:
      if (inputMatrix)
```

```
      {
       printMatrix(targetMatrix, "Input Matrix");
       boolean isD = isMatrixDistanceMatrix(targetMatrix);
         if (isD) System.out.println("\nMatrix " +
        "is a distance matrix");
         else
          {
        System.out.println("Matrix " +
             "is NOT a distance matrix");
        System.exit(0);
           }
         numberOfLeaves = targetMatrix.length;
         nAdditionalLeaves = numberOfLeaves-3;
    }
    else//if a random tree is generated -> Matrix
    {
     numberOfLeaves =    3 + nAdditionalLeaves;
     String topology = generateTopology();
     allLeaves =
     leaves.substring(0, 3 + nAdditionalLeaves);
     if (print5)
     System.out.println( "\nAll leaves = " + allLeaves);

     targetTopology = simplify(topology);
     generateLengths();
     targetTree = insertLengths(topology);
     System.out.println("Target tree: each link " +
     "has its own length = \n" + targetTree);
     targetMatrix  =
         new double[numberOfLeaves][numberOfLeaves];
     targetMatrix = associatedMatrix(targetTree);
     printMatrix(targetMatrix, "Target Matrix");

     double d = absDistance(numberOfLeaves,
                  targetMatrix, targetMatrix);


     System.out.println( "\nDistance from target matrix " +
```

```
      "to target = " + d);
     }


    System.out.println("\nDefinition: \nDistance to target = " +
          " Absolute distance between" +
         " target matrix  and \n" +
         "the distance matrix associated to " +
         "the tree with lengths of links that \n" +
         "is the best fitting  of the topology.\n" +
         "Perfect fitting: distance = 0 \n");
     System.out.println("\nNumber of Generations = "
            + numberOfGenerations);
     if (print7)

     System.out.println("Number of leaves = "
                        + numberOfLeaves +
                         "\nMutation rate = "   + m +
                 "\nPopulation size = " + N);

     initialization();
     evolution();

     }
    });
  }



//Test for being a distance matrix.
//MaxLenght = max value among entries is calculated.
private static boolean
    isMatrixDistanceMatrix(double[][] M)
{
 int l = M.length;
 boolean isD = true;
 //Is M(u,u) = 0?
```

```
 for(int u = 0; u < l; u++)

if (M[u][u] ==  0) ;
 else
  {
isD = false;
return isD;
  }

 //Is M(u,v) = M(v,u)?
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
if (M[u][v] == M[v][u] ) ;
 else
{
 isD = false;
 return isD;
}

 //Is M(u,v)>0 for u not v?
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
if (M[u][v] >  0) ;
 else
{
 isD = false;
 return isD;
}



 //Is M(u,v) < M(u,w) + M(w,v)?
 for(int u = 0; u < l; u++)
 for(int v = 0; v < l; v++)
 for(int w = 0; w < l; w++)

if (M[u][v] <= M[u][w]+M[w][v]  ) ;
 else
```

```
{
 isD = false;
 return isD;
}
 double m = 0;
 for(int u = 0; u < l; u++)
 for(int v = u+1; v < l; v++)
 if (M[u][v] > m ) m = M[u][v];
 maxLength = (int) m + 1;
 return true;
}



//Simplification from Link notation to Newick's notation:
//(((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String linkToNewick(String newTree)
{
 String tree = "";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
  char c = newTree.charAt(i);
  //Is c an internal node to be deleted?
  if (charIsInternalNode(c));
   else //add char c to output
    tree = tree + c;
  }
 return tree;
}


//Reinitialize and generate a random topology:
//Leaves are added recursively to
//seed = ((Aa,Ba),Ca)
private static String generateTopology()
```

```
 {
  //Next leaf to be grafted
  nextLeafIndex = 2;
  //Next internal node to be inserted
  nextIntNodeIndex = 0;
  //A tree topology is generated
  String topology = seed;
  presentNodes = "a";
  for(int i = 0; i < nAdditionalLeaves; i++ )
 topology = generateChild(topology);
  return topology;
 }


//Random lengths are added to a tree in Link notation
private static void makeLengths()
{
//Lengths with digit values are generated
for(int i = 0; i < numberOfLeaves; i++)
 {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
 }
}

//The population is initialized with nTop topologies.
//Lengths are inserted to get nTop trees
//Fitness is assessed:
//Has the solution been guessed?
private static void initialization()
{
 System.out.println("\n====" +"INITIALIZATION : ");
 if (print5)
  System.out.println("\nSeed = " + seed
 + " -> "    + linkToNewick(seed));

 if (print7)
```

```
  System.out.println( N
   + " random topologies " +
   "and their trees are generated \n");

 //nTop topologies and trees are generated
 for(int i = 0; i < N; i++)
 {
  presentLeaves = basicLeaves;
  String topo1 = generateTopology();
  topologies[i] = topo1;
  String tree = insertLengths(topo1);
  makeLengths();
  //Lengths of links are added to tree topology
  tree = insertLengths(topo1);
  trees[i] = tree;
  if (print5)
  {
System.out.println("\n" +  i + "   " + topo1 + " -> "
          + linkToNewick(topo1));
System.out.println("Tree1 = " + tree);
  }

 }
 assessFitnesses();
 scaleFitnesses();
}


//=====================================
//============= EVOLUTION =============
//=====================================


//Fitness is calculated for each tree
//Fitness  =
//Distance between target matrix and
//the associated distance matrix to tree is calculated.
//The minimal distance is found:
```

```
//this renders the fittest tree.
//Has the solution been guessed?
private static double assessFitnesses()
{
 if (print5)
  System.out.println(  "\n=====Fitnesses are assessed");

 String tree = "";
 //String topo = "";
 double d = 0;
for(int i = 0; i < N; i++)
{

 tree = trees[i];
 decoderLengths(tree);
 double M[][] = distanceMatrix(tree);
 //Absolute distance from the associated distance matrix
 //of the tree
 //to target matrix is evaluated
 d = absDistance(numberOfLeaves,targetMatrix, M);
 distance[i] = d;


 if (print5)
  {
   System.out.println( "\nTree = \n" + trees[i]);
   System.out.println( "\nDistance to target = " + d);
  }


if (print5)
 {
  printMatrix(M, "Distance matrix associated to tree");
  printMatrix(targetMatrix, "Target matrix");
}

/*
```

```
//Target topology has been found?
topo = deleteLengths(bestTree);
topo = simplify(topo);
if (targetTopology.contentEquals(topo))
 {
  System.out.println( "\n \nTOPOLOGY HAS BEEN FOUND: \n" +
"at generation " + generation
+ "\nFound topology  = \n" + topo
+ "\nTarget Topology = \n" + targetTopology);
 //System.exit(0);
 if (print5) print5 = false;
}

*/

//End work if solution has been found.
if (d == 0)
{
System.out.println("\n BestTree  = \n" +  bestTree
           + "\nDistance = " + bestDistance
           + "\ntarget tree = \n" + targetTree
           + "\nInitial distance = " + distance0);

workDone = false;
}
}//End of for(int i...




 //The fittest individual minimizes distance function
 bestDistance = distance[0];
 best = 0;
 for(int i = 1; i < N; i++)
 {
  if (distance[i] < bestDistance )
   {
    bestDistance = distance[i];
```

```
    best = i;
   }
 }

 //Best tree
 bestTree = trees[best];




 if (generation == 0 ) distance0 = (int) bestDistance;
 if (print7)
 {
 System.out.println("\n BestTree  = \n" +  bestTree
           + "\nDistance = " + bestDistance
           + "\ntarget tree = \n" + targetTree
           + "\nInitial distance = " + distance0);
 //Target topology has been found?
 String topo = deleteLengths(bestTree);
 topo = simplify(topo);

 System.out.println( "\nBest topology  = \n" + topo
+ "\nTarget Topology = \n" + targetTopology);
 if (targetTopology.contentEquals(topo))
 {
  System.out.println( "\n \nTOPOLOGY HAS BEEN FOUND! \n");
 //System.exit(0);
 if (print5) print5 = false;
}

 }
 return bestDistance;
}



//Fitness is rescaled to take values
```

```
//in within 0(the worst) and 1(the best)
private static void scaleFitnesses()
{
 if (print5)
  System.out.println(  "\n====Fitnesses are rescaled: " +
  "0(the worst) and 1(the best)");
 double maxD = distance[0];
 double minD = distance[0];
 double b = 1 ;
 //Min and max distances are found
 for(int i = 1; i < N; i++)
  {
if ( distance[i] > maxD ) maxD =  distance[i];
if ( distance[i] < minD ) minD =  distance[i];
  }

 if (minD == maxD) //All   equally fit
  {
   for(int i = 0; i < N; i++)
 fitness[i] = 1;
  }
 else
  {
 //Distance to target determine fitness:
//fitness[maxD] = 0;    fitness[minD] = 1;
 for(int i = 0; i < N; i++)
 {
  b = 1;
  b = b/(minD - maxD);
  double a = -b*maxD;
  fitness[i] = a + b * distance[i];
}
  }
 if (print5)
  for(int i = 0; i < N; i++)
   {
 System.out.println("Individual = "
             + i + " has distance " +
```

```
          distance[i] + " and fitness " + fitness[i]);
   }
}

//Probability of cloning is proportional to fitness.
//N new individuals are formed.
private static void selectiveCloning()
{
 if (print5)
  System.out.println(  "\n====Selective cloning");
//The best is cloned
 newTrees[0] = trees[best];
 if (print5)
 System.out.println("Individual = "
                      + best + " with fitness " +
 fitness[best] + "-> new individual " + 0
 + " = "  + newTrees[0]  );
//N-1 children are sampled
for(int i = 1; i < N; i++)
{
 //flag
 boolean go = true;
 while (go)
  {
 //An individual is chosen at random
 int k = r.nextInt(N);
 //Probability of cloning is proportional to fitness.
 double q = r.nextDouble();
 if ( q < fitness[k]) //make a copy of k
  {
 newTrees[i] = trees[k];
 if (print5)
 System.out.println("Individual = "
                      + k + " with fitness " +
 fitness[k] + "-> new individual " + i
 + " = "  + newTrees[i]  );

 go = false;//end task
```

```
  }
  }

}//End of for //N-1 children are sampled
}

//============Mutation============


//String s ( a char) is deleted from mother
public static String delete(String s, String mother)
{
 String output = "";
 int l = mother.indexOf(s);
 output = mother.substring(0, l)
 + mother.substring(l+1) ;
 return output;
}



//String leaves is reversed
public static String reverse(String leaves)
{
 String reversed = "";
 int l = leaves.length();
 for(int i = l-1; i >= 0; i--)
  {
String c = "" + leaves.charAt(i);
reversed = reversed + c;
  }
 if (print1)
 System.out.println("Reverse order = " + reversed);
 seed =  "((Za,Ya),Xa)";
 if (print1)
 System.out.println("seed = " + seed);
 basicLeaves = "ZYX";
 presentLeaves =   basicLeaves;
```

```
 return reversed;
}


//String leaves is permuted
public static String permuteOrder(String leaves)
{
 if (print1)
  System.out.println("original and permuted order = \n"
                          + leaves);
 String permuted = "";
 int l = leaves.length();
 int m = l;
 for(int i = 0; i < l; i++)
  {
int d = r.nextInt(m) ;
String c = "" + leaves.charAt(d);
leaves = delete(c,leaves);
m = m-1;
permuted = permuted + c;
  }
 if (print1)
 System.out.println(permuted);
 String first = "" + permuted.charAt(0);
 String second = "" + permuted.charAt(1);
 String third = "" + permuted.charAt(2);
 seed = "((" +first + "a," +second + "a)," +third + "a)";
 if (print1)
  System.out.println("seed = " + seed);

  basicLeaves =  first + second + third;
 presentLeaves =   basicLeaves;

 return permuted;
}


//Detect internal nodes, different than leaves
```

```
private static boolean charIsInternalNode(char c)
{
 String s = "" + c;
 if (internalNodes.contains(s)) return true;
  else return false;
}

//Simplification from Link notation to Newick's notation:
// (((AJ,DJ)I,BI),CI) -> (((A,D),B),C)
private static String simplify(String newTree)
{
 String tree = "";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
  {
   //newTree is scanned char by char
   char c = newTree.charAt(i);
   //Is c an internal node to be deleted?
   if (charIsInternalNode(c));
    else //add char c to output
   tree = tree + c;
  }
 return tree;
}

//Rule 2 is used to generate a new tree:
//(S)I -> ((S)N,FN)I.
//S = subtree, I = internal node, F = new leaf,
//N = new internal node.
//Rule 2 grafts new branch FN to link (S)I
//i is the position of char I in string seedTree
private static String useRule2(String seedTree, int i,
      char F, char N)
{
 String newTree = "";
 char I = seedTree.charAt(i);
 if (print1)
```

```
 System.out.println("i = " + i + " I = " + I);
//Subtree S of link (S)I must be read
String S = "";
//Counter of left seedhesis
int counter = 0;
String pl = "" + '(';
String pr = "" + ')';
//Position to be scanned
int position = i;
boolean go = true;
//All right seedheses to the near left of I are captured
while (go)
 {
  position = position -1;
  String a = "" + seedTree.charAt(position);

  if (print1)
   System.out.println("a = " + a + " p = " + pr);
  if (a.equals(pr) )
   {
    counter++;
S = a + S;
   }
  else go = false;
 }//End of while
if (print1)
System.out.println("counter = " + counter);
//The complete subtree is captured:
//seedheses must balanced
while (counter > 0)
 {
  String a = "" + seedTree.charAt(position);
  if (a.equals(pr))   counter++;
  if (a.equals(pl))  counter--;
  S = a + S;
  if (print1)
  System.out.println("Subtree = " + S);
  position = position -1;
```

```
  }
 //Replacement rule 2:
 //((S)n,Fn)i -> (S)i
 //where   F are leaves, n and i are  inner nodes,
 //(S) is a complex structure.
 String SnFni  =   S  + I;
 String replacement =  "("+ S +   N + ","
          + F + N+ ")" + I;
 //Rule 1 is executed
 newTree = seedTree.replace(SnFni, replacement);
 //If I links to a complex structure
 return newTree;
}

//Input: seedTree, a tree in Newick's notation.
//Output : newTree =  seed tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String generateChild(String seedTree)
{

 nextLeafIndex++;
 char nextLeaf = leaves.charAt(nextLeafIndex);

 presentLeaves = presentLeaves + nextLeaf;
 if (print5)
  System.out.println("Method generateChild \n " +
"Present leaves = " + presentLeaves);
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = internalNodes.charAt(nextIntNodeIndex);
 presentNodes = presentNodes + nextIntNode;

 //link to be replaced
 String link = "";
 //String link is replaced by what:
 String replacement = "";
 //new tree generated by replacement
```

```
 String newTree = "";
 //length of seed tree
 int l = seedTree.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
  {
   //seedTree is scanned char by char
   char I = seedTree.charAt(i);
   //Is I an internal node that denotes a link?
   if (charIsInternalNode(I)) counter++;
  }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
 {
  //seedTree is scanned char by char
char I = seedTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = seedTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(seedTree, i,
               nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
```

```
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTree = seedTree.replace(link, replacement);
 }
 if (print1)
System.out.println(newTree + " -> "
                       + simplify(newTree));


 i = l;
   }
  counter++;
}
 }
 return newTree;
}


//Input: seedTree, a tree in Newick's notation.
//Output : newTree =  seed tree + a new link.
//The new link is added by a generative grammar.
//A random election is made among all possibilities.
private static String reinsertLeaf(String seedTree)
{
 nextLeafIndex++;
 char nextLeaf = reinsertingLeaves.charAt(nextLeafIndex);
 presentLeaves = presentLeaves + nextLeaf;
 if (print5)
 System.out.println("Method reinsertLeaf \n " +
"Present leaves = " + presentLeaves);
 //Next internal node to be inserted
 nextIntNodeIndex++;
 char nextIntNode = reinsertingNodes.charAt(nextIntNodeIndex);
 presentNodes = presentNodes + nextIntNode;

 //link to be replaced
```

```
 String link = "";
 //String link is replaced by what:
 String replacement = "";
 //new tree generated by replacement
 String newTree = "";
 //length of seed tree
 int l = seedTree.length();
 //counter of links = 0;
 int counter = 0;
 //Links are counted
 for(int i = 0; i < l; i++ )
 {
  //seedTree is scanned char by char
  char I = seedTree.charAt(i);
  //Is I an internal node that denotes a link?
  if (charIsInternalNode(I)) counter++;
 }
 //A link is chosen
 int k = r.nextInt(counter);
 //System.out.println("Random number = " + k);
 //new branch is grafted at k
 counter = 0;
 for(int i = 0; i < l; i++ )
 {
  //seedTree is scanned char by char
char I = seedTree.charAt(i);
//Is I an internal node that denotes a replacement?
if (charIsInternalNode(I))
 {
  if (counter == k) //Is I the chosen link?
   {
//Check for the char before I
char a = seedTree.charAt(i-1);
//If I links to a complex structure
if (a == ')') newTree = useRule2(seedTree, i,
                nextLeaf, nextIntNode);
else //if I links to a leaf, use rule 1
 {
```

```
  //Link to be replaced
  link = "" + a + I;
  //Replacement rule 1:
  //aI -> (aN,FN)I;   N= nextIntNode;
  //F = nextLeaf
 replacement = "(" + a + nextIntNode + ","
          + nextLeaf + nextIntNode + ")" + I;
 //Rule 1 is executed
 newTree = seedTree.replace(link, replacement);
 }
 if (print1)
System.out.println(newTree + " -> "
                       + simplify(newTree));


 i = l;
   }
  counter++;
}
 }
return newTree;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply the antirule1:
//(An,Fn)i  -> Ai
private static String  antiRule1(String tree,
int positionLeaf, int positionNode)
{
 if (print1)
  System.out.println("AntiRule1");
 int beginIndex = positionLeaf - 4;
 int endIndex = positionNode + 3;
 String AnFni = tree.substring(beginIndex, endIndex);
 if (print1)
  System.out.println("Anfni = " + AnFni);
 String Ai = "" + tree.charAt(beginIndex+1)
```

```
              + tree.charAt(endIndex-1);
 if (print1)
 System.out.println("Ai = " + Ai);
 tree = tree.replace(AnFni, Ai);
 if (print1)
  System.out.println("Output tree = \n" + tree);
 return tree;
}



//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply the antirule2:
//((S)n,Fn)i -> (S)i
//positionNode refers to the last appearing of node n
private static String  antiRule2(String tree,
int positionLeaf, int positionNode)
{
 if (print1)
 System.out.println("AntiRule2");
 //Leaf to be deleted
 String leaf = "" + tree.charAt(positionLeaf);
 if (print1)
  System.out.println("\nDeletion = " + leaf);
 //Accompanying inner node
 String innerNode = ""
            + tree.charAt(positionNode);
 int firstPosition = tree.indexOf(innerNode);
 String newTree = "";
 String i = ""; //BUG?????
 if (positionNode +2 < tree.length())
       i = ""+ tree.charAt(positionNode+2);
 if (print1)
  System.out.println("FirstPosition = " +
                    firstPosition + " i = " + i);
 //Subtree S of link (S)I must be read
 String S = "";
```

```
 //Counter of left seedhesis
 int counter = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Position to be scanned
 int position = firstPosition;
 boolean go = true;
 //All right seedheses to the near left of I are captured
 while (go)
 {
  position = position -1;
  String as = "" + tree.charAt(position);

  if (print1)
  System.out.println("a = " + as + " p = " + pr);
  if (as.equals(pr) )
  {
counter++;
S = as + S;
  }
  else go = false;
 }//End of while
 if (print1)
  System.out.println("counter = " + counter);
 //The complete subtree is captured:
 //seedheses must balanced
 while (counter > 0)
  {
   String as = "" + tree.charAt(position);
   if (as.equals(pr))   counter++;
   if (as.equals(pl))  counter--;
   S = as + S;
   if (print1)
    System.out.println("Subtree = " + S);
   position = position -1;
  }
 String n = innerNode;
 //Replacement antiRule 2:
```

```
 //((S)n,Fn)i -> (S)i
 //where A and F are leaves, n and i are  inner nodes,
 //(S) is a complex structure.
 String SnFni  =  "("+ S  + n + "," + leaf + n + ")" + i ;
 String Si = S + i;
 if (print1)
  System.out.println("\nSnFni = " + SnFni + "\n Si = " + Si);
 //Rule 1 is executed
 newTree = tree.replace(SnFni, Si);
 if (print1)
  System.out.println("Output tree = \n" + newTree);
 return newTree;
}


//Leaf F and node n conform  link Fn  and are deleted.
//We apply   antirule3:
//(Fn,An)i  -> Ai
private static String  antiRule3(String topo,
int positionLeaf, int positionNode)
{
 if (print1)
  System.out.println("AntiRule3"
+ "\nInput topology = \n" + topo);
 int beginIndex = positionLeaf - 1;
 int endIndex = positionNode + 6;
 String FnAni = topo.substring(beginIndex, endIndex);
 if (print1)
 System.out.println("FnAni = " + FnAni);
 String Ai = "" + topo.charAt(beginIndex+4)
           + topo.charAt(endIndex-1);
 if (print1)
  System.out.println("Ai = " + Ai);
 topo = topo.replace(FnAni, Ai);
 if (print1)
  System.out.println("Output topology = \n" + topo);
 return topo;
}
```

```
//Leaf F appeared as an insertion into a complex structure S.
//it is deleted.
//We apply antirule4:
//(Fn,(S)n)i -> (S)i
private static String  antiRule4(String tree,
int positionLeaf, int positionNode)
{

 //Leaf to be deleted
 String leaf = "" + tree.charAt(positionLeaf);

 //Accompanying inner node
 String innerNode = ""
            + tree.charAt(positionNode);
 if (print5)
  System.out.println("\nAntiRule4"
+ "\nLeaf to be deleted = " + leaf
 + " at position " + positionLeaf
+  "\nAccompanying node = " + innerNode
 + " at first position " + positionNode);
 //Second position of accompanying node
 int positionNode2 = 0;
 int l = tree.length();
 boolean go = true;
 //tree is scanned char by char
 for(int i = positionNode + 1; i < l; i++ )
  {
 String c = "" + tree.charAt(i);

 if (go)
 {
  if (c.equals(innerNode))
 {
  positionNode2 = i;
```

```
          go = false;
          if (print5)
          System.out.println(
         "Second position of accompanying node = " +
            positionNode2 + " i = " + i);
 }
 }
  }


 int beginIndex = positionLeaf - 1;
 int endIndex = positionNode2+2;
 String i = "";
 if (endIndex < tree.length())
         i = ""+ tree.charAt(endIndex);

 if (print5)
  System.out.println("EndIndex = " +
                      endIndex + " i = " + i);

 String FnSni  = tree.substring(beginIndex, endIndex) + i;
 if (print5)
  System.out.println("FnSni = " + FnSni);
 String S = tree.substring(beginIndex+4, endIndex-2);
 if (print5)
System.out.println("S = " + S);
 String Si = S + i;
 if (print5)
 System.out.println("Si = " + Si);

 String newTree = tree.replace(FnSni, Si);
 if (print5)
 System.out.println("Output tree = \n" + newTree);
 //System.exit(0);
 return newTree;
}
```

```
//The last inserted  leaf is deleted from tree.
private static String deleteLastLeaf(String tree)
{
 //Leaf to be deleted
 String leaf = "" + leaves.charAt(nextLeafIndex);
 if (print1)
 System.out.println("\nLeaf to be deleted = " + leaf);
 //Accompanying inner node
 String innerNode = ""
      + internalNodes.charAt(nextIntNodeIndex);
 if (print1)
  System.out.println("Accompanying inner  node = "
              + innerNode);
 //new tree = output
 String newTree = "";
 //length of  tree as string
 int l = tree.length();
 //Last positions are found
 int positionLeaf = -1;
 int positionNode = -1;
 //tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
 String c = "" + tree.charAt(i);
 if (c.equals(leaf)) positionLeaf = i;
 if (c.equals(innerNode)) positionNode = i;
  }
 if (print1)
 System.out.println("Position of " + leaf +
                   " = " + positionLeaf);
 if (print1)
  System.out.println("Position of " + innerNode +
    " = " + positionNode);
 int firstPosition = tree.indexOf(innerNode);
 //Check for the char before firstPosition
 char a = tree.charAt(firstPosition - 1);
```

```
 //If I links to a complex structure
 if (a == ')')
  newTree = antiRule2(tree,positionLeaf,positionNode);
 else  //Apply antiRule1 for simple cases
  newTree = antiRule1(tree,positionLeaf,positionNode);
 nextLeafIndex--;
 nextIntNodeIndex--;
 return newTree;
}


//A random leaf is deleted from tree.
private static String deleteRandomLeaf(String tree)
{
  if (print5)
   {
     System.out.println("\nMethod deleteRandomLeaf"
 + "\ntree = " + tree
+ "\ndeletableLeaves = " + deletableLeaves);
   }
 int l = deletableLeaves.length();
 int position = r.nextInt(l);
 //Leaf to be deleted
 String leaf = "" + deletableLeaves.charAt(position);
 int nodeIndex = tree.indexOf(leaf) + 1;
 //Accompanying inner node
 String innerNode = ""
          + tree.charAt(nodeIndex);

 deletableLeaves = delete(leaf,deletableLeaves);
 presentLeaves = delete(leaf,presentLeaves);
 reinsertingLeaves = reinsertingLeaves + leaf;
 reinsertingNodes = reinsertingNodes + innerNode;


if (print5)
 System.out.println("\nLeaf to be deleted = " + leaf
+ "\nReinserting leaves = " + reinsertingLeaves
```

```
      + "\nAccompanying inner  node = "   + innerNode
      + "\nReinserting nodes = " + reinsertingNodes);
 //new tree = output
 String newTree = "";
 //length of  tree as string
 l = tree.length();
 //Last positions are found
 int positionLeaf = tree.indexOf(leaf);
 int positionNode = positionLeaf + 1;


 if (print5)
  System.out.println("Position of " + leaf
        +  " = " + positionLeaf
        + "\nPosition of " + innerNode
                + " = " + positionNode);
 //Which set of deleting rules to apply:
 // 1 and 2 else 3 and 4?
 String previousChar = "" + tree.charAt(positionLeaf-1);
 if (previousChar.equals(",")) //Apply antiRules 1 and 2
  {
   if (print5)
System.out.println("Apply antiRules 1 and 2");
   int firstPosition = tree.indexOf(innerNode);
   //Check for the char before firstPosition
   char a = tree.charAt(firstPosition - 1);
   //If I links to a complex structure
   if (a == ')')
    newTree = antiRule2(tree,positionLeaf,positionNode);
   else  //Apply antiRule1 for simple cases
  newTree = antiRule1(tree,positionLeaf,positionNode);
   nextLeafIndex--;
   nextIntNodeIndex--;
  }
 else //Apply antiRules 3 and 4
  {
if (print5)
System.out.println("Apply antiRules 3 and 4");
```

```
   int firstPosition = tree.indexOf(innerNode);
   //Check for the char before firstPosition
   char a = tree.charAt(firstPosition + 2);
   //If I links to a complex structure
   if (a == '(')
 newTree = antiRule4(tree,positionLeaf,positionNode);
    else  //Apply antiRule3 for simple cases
   newTree = antiRule3(tree,positionLeaf,positionNode);
   nextLeafIndex--;
   nextIntNodeIndex--;
  }

 if (print5)
  System.out.println("Tree after deletion" + newTree);
 return newTree;
}



//Mutation =
//Random deletions are reinserted.
public static String  mutationRandomDeletion(
           String topo, int nAdditionalLeaves)
{
//Reinitialization
 reinsertingLeaves = "";
 reinsertingNodes = "";
 //Some leaves are deleted: random order
 if (print1)
 System.out.println("Some leaves are deleted AT RANDOM");
 for(int i = 0; i < nUndoRedo; i++ )
{
 //Random leaf is deleted.
 topo = deleteRandomLeaf(topo);
 }
 //Deleted leaves are reinserted
 nextLeafIndex = -1;
 nextIntNodeIndex = -1;
```

```
 if (print1)
  System.out.println("Deleted leaves are reinserted");
 for(int i = 0; i < nUndoRedo; i++ )
{
 topo = reinsertLeaf(topo);
}
 return topo;
}


//Mutation:
//some leaves are deleted and then reinserted
//in random order.
//Deleting can happen in either of two ways:
//Reversing the order of synthesis else
//in random order.
public static String mutationOfTop(String topology,
                       int nAdditionalLeaves)
{
 original = topology;
 if (print5)
{
System.out.println("\nMethod mutationOfTop" +
"\nPresentLeaves =  "
                  + presentLeaves);
}

 //Seed leaves are not removed
 deletableLeaves = presentLeaves.substring(3);
 //Number of leaves to be deleted and then reinserted
 if (nAdditionalLeaves == 1) nUndoRedo = 1;
 else
  nUndoRedo = r.nextInt(nAdditionalLeaves-1) + 1;
 if (print1)
  System.out.println("\nNumber of leaves to delete =  "
                  + nUndoRedo);

 double s = r.nextDouble();
```

```
if ( s < pReverseSynth)//Choose deletion in reversed order
{
 //Some leaves are deleted:
 //reversed order to that of synthesis.
 if (print1)
   System.out.println("Leaves are deleted " +
                           "in reverse order");
 for(int i = 0; i < nUndoRedo; i++ )
{
 //Last inserted leaf is deleted.
 topology = deleteLastLeaf(topology);
 System.out.println(topology);
}
//deleted leaves are reinserted
 if (print1)
System.out.println("\nDeleted leaves are reinserted");

for(int i = 0; i < nUndoRedo; i++ )
 {
 topology = generateChild(topology);
 }
}
else //Choose  at random leaves to be deleted
topology = mutationRandomDeletion(topology, nUndoRedo);
//Output
String mutant = topology;
if (print5)
 System.out.println("\nOriginal and mutant topologies = \n"
 + original + "\n" + mutant);
if (print1)
 {
  if (original.equals(mutant))

 System.out.println("Mutation made nothing");
  else
 System.out.println("\nMutation worked");
 }
return mutant;
```

```
}


//Random lengths are added to a tree in Link notation
public static void generateLengths()
{
 //Lengths with digit values are generated
 for(int i = 0; i < numberOfLeaves; i++)
  {
 //Zero lengths are not allowed
 lengthsLeaves[i] = 1 + r.nextInt(maxLength - 1);
 lengthsInnerLinks[i] = 1 +  r.nextInt(maxLength - 1);
  }
}



//Random lengths are added to a tree in Link notation
public static String insertLengths(String treeLinkN)
{
 String thisChar = "";
 String newTree = "";
 //length of String   treeLinkN
 int m = treeLinkN.length();
 //TreeLinkN is scanned char by char
 for(int i = 0; i < m; i++ )
  {
   thisChar = "" + treeLinkN.charAt(i);
   newTree = newTree + thisChar;
   if (print1)
   System.out.println(newTree.length() + " " + newTree);
   //Lengths are appended
   if (leaves.contains(thisChar))
    {
     int k = leaves.indexOf(thisChar);
     double j = lengthsLeaves[k];
     newTree = newTree + ":" + j;
    }
```

```
 if (internalNodes.contains(thisChar))
  {
   int k = internalNodes.indexOf(thisChar);
   double j = lengthsInnerLinks[k];
   newTree = newTree + ":" + j;
  }
 }
 return newTree;
}


//Lengths are deleted from newTree
private static String deleteLengths(String newTree)
{
 String tree = "";
 String numeric = ".:0123456789";
 //length of string newTree
 int l = newTree.length();
 for(int i = 0; i < l; i++ )
 {
  //newTree is scanned char by char
  char c = newTree.charAt(i);
  String s = "" + c;
  //Is c numeric: delete it!
  if (numeric.contains(s)); //add char c to output
  else tree = tree + c;
 }
 return tree;
}


//Input: string tree with complete information.
//Work: lengthsLeaves[] and lengthsInnerLinks[]
//are read from tree.
private static void decoderLengths(String tree)
{
 if(print1)
  System.out.println(
```

```
"Input tree at decoderLengths = \n "   + tree  );
 //Delete :
 tree = tree.replaceAll(":", "");
 if(print1)
 System.out.println(
"Input tree without : = \n "   + tree  );
 int n = tree.length();
 String thisChar = "";
 String nextChar = "";
 String numbers = ".0123456789";
 boolean leaf = false;
 boolean node = false;
 String Leaf = "";
 String numberLeaf = "";
 String numberNode = "";
 String Node = "";
 int j ;
 double f;

//All chars in tree are scanned
for(int i=0; i < n; i++)
{
 thisChar = "" + tree.charAt(i);
 if (i < n-1) nextChar  = "" + tree.charAt(i+1);
 //Do we have a leaf?
 if (leaves.contains(thisChar))
   {
 leaf = true;
 Leaf = "" + thisChar;
   }

 //Do we have an internal node?
 if (internalNodes.contains(thisChar))
  {
 node = true;
 Node = "" + thisChar;
  }
```

```
 //Do we have a number that belongs in a leaf?
 if ( (leaf) & (numbers.contains(thisChar)))
{
 numberLeaf = numberLeaf + thisChar;
}

 //Do we have a number that belongs in a leaf?
 if ( (node) & (numbers.contains(thisChar)))
{
 numberNode = numberNode + thisChar;
}

 //Must we end accumulation of numbers of a leaf?
 if ( (leaf) & (internalNodes.contains(nextChar)))
{
  leaf = false;
  j = leaves.indexOf(Leaf);
  if (print1)
  System.out.println("Leaf = " + Leaf +
  " numberLeaf = " + numberLeaf);
  f = new Double(numberLeaf);
  lengthsLeaves[j] = (int) Math.floor(f);
  numberLeaf = "";
}

 //Must we end accumulation of numbers of a node?
 if ( (node) & (!(numbers.contains(nextChar))))
{
  node = false;
  j = internalNodes.indexOf(Node);
  if (print1)
  System.out.println("Node = " + Node);
  f = new Double(numberNode);
  lengthsInnerLinks[j] = (int) Math.floor(f);
  numberNode = "";
}

 }
```

```
}


//Lengths undergo mutation
private static  void mutateLengths()
{
 //Mutation in links that connect leaves
 for(int k = 0; k < numberOfLeaves; k++)
  {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsLeaves[k] = r.nextInt(maxLength);

  }
 //Mutation in links that connect inner nodes
 for(int k = 0; k < numberOfLeaves-2; k++)
  {
 double q = r.nextDouble();
 if (q < m) //mutate
lengthsInnerLinks[k] = r.nextInt(maxLength);

  }
}



//Type of  mutation is decided and executed:
//mutation of lengths of links
//else full mutation (topology and lengths of links).
private static String mutateTree(String tree)
{
 presentLeaves = allLeaves;
 //Topology is read from tree
 String topology = deleteLengths(tree);
 //Lengths are read from tree and recorded in vectors
 //lengthsLeaves[] and lengthsInnerLinks[]
 decoderLengths(tree);
 mutateLengths();
```

```
 String  mutantTree = "";
 //Mutate lengths else topology ( + mutated lengths)?
 double s = r.nextDouble();
 if ( s < pMutateOnlyLengths) //only lengths are mutated
{
 if (print1)
 System.out.println("\n====Mutation of lengths");

  mutantTree =  insertLengths(topology);

  if (print1)
  {
  System.out.println("Input tree =  " + tree);
  System.out.println("Topology =  " + topology);
  System.out.println("Output tree= " + mutantTree);
  }
 }
 else //topology + lengths are mutated
{
 if (print1)
  System.out.println("\n====Mutation of topology");


   //Mutation of topology
 String mutantTopology = mutationOfTop(topology,
                           nAdditionalLeaves);

 //Mutant lengths are reinserted
 mutantTree =  insertLengths(mutantTopology);
}

 return mutantTree;
}
```

```
//Individuals are mutated
private static void mutation()
{
 if (print5)
  {
System.out.println("\n====Mutation \n");

System.out.println("Trees to be mutated");
  }
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print5)
  System.out.println(trees[i]);
 }
 //The old best, i = 0, is conserved intact
 newTrees[0] = trees[0];
 //Other N-1 individuals suffer mutation
 for(int i = 1; i < N; i++)
  newTrees[i] = mutateTree(trees[i]);


if (print5)
System.out.println("Mutant trees");
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print5)
  System.out.println(trees[i]);
 }
}

//==========Recombination============


 //Counter of subtopologies
 private static  int counterSubtop;
 //Particular number of subtopologies
```

```
 private static  int m1 = 0;



 private static  String[] subtrees1, subtrees2;
 private static  String   subTree1, subTree2,
                      treeRec1, treeRec2;

 //Structure of hashtable : integer + {string1, string2}
 private static Hashtable<Integer, String[]> hashtable =
 new  Hashtable<Integer, String[]>();




//========== Subtrees are listed ============


//Subtrees of  topology are found
private static String[] subtopologies(String topology)
{
 if (print1)
  System.out.println("\nSubtopologies are found");
 //Counter of left parenthesis
 int counter = 0;
 //Counter of subtopologies
 counterSubtop = 0;
 String pl = "" + '(';
 String pr = "" + ')';
 //Length of String   topology
 int l = topology.length();
 String[] subtopologies = new String[l];
 //String topology is scanned char by char
 for(int i = 0; i < l-1; i++ )
  {
   String Sub = "";
   String a = "" + topology.charAt(i);
```

```
   int position = i+1;
   if (a.equals(pl))
   {
    counter++;
    Sub =  Sub + a;
    //The complete subtopology is captured:
    //Parentheses must balanced
   while (counter > 0)
    {
     String b = "" + topology.charAt(position);
     if (b.equals(pl))   counter++;
     if (b.equals(pr))  counter--;
     Sub =  Sub + b;
     position++;
    }
   if (print2)
   System.out.println(Sub);
   subtopologies[counterSubtop] = Sub;
   counterSubtop++;
   }
  }
 return subtopologies;
}


//Leaves of subtopologies  are found
private static String[]leaves(int m, String[] subtop)
{
 if (print1)
  System.out.println("Leaves are found");
 String[] myLeaves = new String[m];
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
   else
    {
     myLeaves[i] = "";
```

```
     int n = s.length();
     for(int j = 0; j < n; j++)
      {
       String c = "" + subtop[i].charAt(j);
       if (leaves.contains(c))
       myLeaves[i] = myLeaves[i] + c;
     }
   if (print1)
     System.out.println(subtop[i] + " "
                        + myLeaves[i] );
   }
  }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
  System.out.println("Leaves are listed orderly" );
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
    else
     {
      s =   myLeaves[i];
      myLeaves[i] = "";
      int n = leaves.length();
   for(int j = 0; j < n; j++)
    {
     String c = "" + leaves.charAt(j);
     if (s.contains(c))
     myLeaves[i] = myLeaves[i] + c;
    }
   if (print1)
   System.out.println(subtop[i] + " " + myLeaves[i] );
  }
 }
 return myLeaves;
}
```

```
//Inner nodes of subtopologies  are found
private static String[] inner(int m, String[] subtop)
{
 if (print1)
  System.out.println("\nInternal nodes are found");
 String[] myNodes = new String[m];
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
    else
     {
      myNodes[i] = "";
      int n = s.length();
      for(int j = 0; j < n; j++)
       {
        String c = "" + subtop[i].charAt(j);
        if (internalNodes.contains(c))
        myNodes[i] = myNodes[i] + c;
       }
    if (print1)
System.out.println(subtop[i] + " " + myNodes[i] );
  }
 }
 //Chars of Leaves are rewritten in alphabet order
 if (print1)
 System.out.println("Inner nodes are listed orderly" );
 for(int i = 0; i < m; i++)
  {
   String s = subtop[i];
   if (s == null);
    else
     {
      s =   myNodes[i];
      myNodes[i] = "";
      int n = internalNodes.length();
      for(int j = 0; j < n; j++)
```

```
       {
        String c = "" + internalNodes.charAt(j);
        if (s.contains(c))
      myNodes[i] = myNodes[i] + c;
       }
  if (print1)
   System.out.println(subtop[i] + " " + myNodes[i] );
 }
}
return myNodes;
}


//Subtopologies are compared to decide whether
//or not they have the same set of leaves
//and the same set of inner nodes.
//All pairs that authorize recombination
// are kept in a hastable (=  a table for pairing)
private static boolean comparison(String Leaves1[],
                      String Leaves2[],
                      String Inner1[],
                      String Inner2[])
{
 if (print5)
 System.out.println("\nComparison of leaves and nodes");
 Boolean recAllowed = false;
 int m = Leaves1.length;
 int n = Leaves2.length;
 Integer counter = 0;
 for(int i = 1; i < m; i++)
  {
   for(int j = 1; j < n; j++)
    {
     if ( Leaves1[i].equals(Leaves2[j]))
      {
       subTree1 = subtrees1[i];
       subTree2 = subtrees2[j];
       recAllowed = true;
```

```
       if (print5)
        System.out.println(
          "Sets of leaves are EQUAL " + i +
        "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
        String val[] = {subTree1, subTree2};

        //Recombinant subtrees are kept in a hashtable
        hashtable.put(counter,val);
        counter++;
        if ( Inner1[i].equals(Inner2[j]))
         {
      if (print5)
      System.out.println(
        "Sets of inner nodes are EQUAL " + i +
          "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
         }
        else
         {
       if (print5)
   System.out.println(
       "Sets of inner nodes are DIFFERENT " + i +
       "  " + Inner1[i] + "  " + j + "  " + Inner2[j] );
          }
      }
    else
  if (print5)
   System.out.println(
       "Sets of leaves are DIFFERENT " + i +
        "  " + Leaves1[i] + "  " + j + "  " + Leaves2[j] );
   }
 }
 return recAllowed;
}


//A pair of subtrees that authorize recombination
//is chosen at random from the hashtable and
//recombination is executed.
```

```
private static  String recombinationTrees(String tree1,
                                          String tree2)
{
 //Possible pairs of recombinant subtrees are recovered
 //from the hashtable and put into an enumeration<>
 Enumeration<Integer> enumKey = hashtable.keys();
 int counter = 0;
 while(enumKey.hasMoreElements())
  {
   Integer key = enumKey.nextElement();
   String[] val = hashtable.get(key);
   if (print3)
    System.out.println("Hasthtable " + key    + "\n"
 + val[0]  + "\n"+  val[1]);
   counter++;
   }

 //A pair of recombinant subtrees is chosen at random
 Integer election = r.nextInt(counter);
 if (print5)
  System.out.println("Selected pair = " + election);
 String[] val = hashtable.get(election);
 subTree1 = val[0];
 subTree2 = val[1];
 if (print5)
   System.out.println("Recombinant subtrees \n" +
      subTree1 + "\n" +  subTree2);

 //Recombination is executed
 treeRec1 = tree1.replace(subTree1, subTree2);
 treeRec2 = tree2.replace(subTree2, subTree1);
 if (print5)
  System.out.println("\nRecombination is allowed:"
 + "\nOriginal trees \n"
 + tree1 + "\n" + tree2
 + "\nSwapping subtrees \n"
 +  subTree1 + "\n" + subTree2
 + "\nRecombinant trees \n"
```

```
 + treeRec1 + "\n" + treeRec2 );
 return treeRec1;
}


//Recombination of tree1 and tree2 is tried out
public static String recombinationMaster(String tree1,
                                String tree2)
{
 //First topology
 String topo1 = deleteLengths(tree1);
 if (print5)
  {
   System.out.println(topo1 + " -> "
       + linkToNewick(topo1));
   System.out.println("\n Tree1 = " + tree1 +
 "\n Topo1 = " + topo1);
  }


 //Length of String   topology
 m1 = topo1.length();

 //Subtopologies are found
 subtrees1 = new String[m1];
 subtrees1 = subtopologies(topo1);
 m1 = counterSubtop;
 String[] subtopologies1 = new String[m1];
 for(int i = 0; i < m1; i++)
  subtopologies1[i] =  simplify(subtrees1[i]);
 if (print5)
 System.out.println();
 String[] Leaves1 = new String[m1];
 Leaves1 = leaves(m1,subtopologies1);
 String[] Inner1 = new String[m1];
 Inner1 = inner(m1,subtrees1);
 //Chars of Leaves are rewritten in alphabet order
 if (print5)
```

```
 System.out.println("\nSubtrees, subtopologies, Leaves, " +
            "Inner nodes" );
for(int i = 0; i < m1; i++)
 {
  if (subtopologies1[i] == null);
  else
   {
if (print5)
       System.out.println(i
  + "   " + subtrees1[i]
  + "   " + subtopologies1[i]
  + "   " +  Leaves1[i] + "   " + Inner1[i]);
   }
 }

if (print5)
 System.out.println();

//Second topology
String topo2 = deleteLengths(tree2);
if (print5)
 {
  System.out.println(topo2 + " -> "
  + linkToNewick(topo2));

System.out.println("\n Tree2 = " + tree2 +
"\n Topo2 = " + topo2);
 }
//Length of String   topology
int  m2 = topo2.length();

subtrees2 = new String[m2];
subtrees2 = subtopologies(topo2);
m2 = counterSubtop;
//Subtopologies are found
String[] subtopologies2 = new String[m2];
for(int i = 0; i < m2; i++)
 subtopologies2[i] =  simplify(subtrees2[i]);
```

```
if (print5)
 System.out.println();
String[] Leaves2 = new String[m2];
Leaves2 = leaves(m2,subtopologies2);
String[] Inner2 = new String[m2];
Inner2 = inner(m2,subtrees2);
//Chars of Leaves are rewritten in alphabet order
if (print5)
 System.out.println("\nSubtopologies, Leaves, " +
            "Inner nodes" );
for(int i = 0; i < m2; i++)
 {
  if (subtopologies2[i] == null);
  else
   {
if (print5)
     System.out.println(i
  + "   " + subtrees2[i]
  + "   " + subtopologies2[i]
  + "   " +  Leaves2[i] + "   " + Inner2[i]);
   }
 }
boolean recAllowed =
  comparison(Leaves1,Leaves2, Inner1,Inner2);
String recombinant = "";
if (recAllowed)
{
  recombinant = recombinationTrees(tree1,tree2);
  if (print5)
 System.out.println("Recombinant tree = "
 + recombinant );
}

else
 {
  recombinant = tree1;
```

```
   if (print5)
  System.out.println("Recombination is not allowed");
  }
 return recombinant;
}


//Recombination methods may produce an output
//with various and different assignment of lengths to
//inner node a. One is chosen.
public static void correction()
{
String topo;
 for(int i = 1; i < N; i++)
  {
decoderLengths(newTrees[i]);
topo = deleteLengths(newTrees[i]);
newTrees[i] = insertLengths(topo);
  }
}

//Each one of the N individuals  are tried out
//for recombination: a partner is chosen at random
//and if recombination is allowed, it is executed.
//Otherwise, it is cloned.
public static void recombination()
{
 if (print5)
 System.out.println( "\n====RECOMBINATION \n");
 //Best individual is excluded from recombination
  newTrees[0] = trees[0];

 for(int i = 1; i < N; i++)
  {
   //A partner for i is chosen
   int j = i;
   String t1 = "";
```

```
   String t2 = "";
   while (i == j)
 j =   r.nextInt(N);
   if (print5)
System.out.println( "\n==RECOMBINATION OF INDIVIDUALS "
+ i + " and " + j);
   t1 = trees[i];
   t2 = trees[j];
   newTrees[i] = recombinationMaster(t1,t2);
   correction();
  }
 if (print5)
 System.out.println("Recombination output");
 for(int i = 0; i < N; i++)
 {
  trees[i] = newTrees[i];
  if (print5)
  System.out.println(trees[i]);
 }
}



//Evolution of trees involving
//recombination and mutation of topologies,
//in within and between.
private static void evolution()
{
 for(int i = 1;
 ((i < numberOfGenerations) & (workDone)); i++)

 {
generation = i;
  if ( (i % lapse) == 0)
  {
              print7 = true;
          //  a.setEnabled(false);
  }
```

```
 else print7 = false;

 if (print7)
 System.out.println( "\n===GENERATION  = " + i);
 selectiveCloning();
 mutation();
 recombination();
 assessFitnesses();
 scaleFitnesses();

}



bestTopology = simplify(deleteLengths(bestTree));

title1 = "Target tree = "  + targetTree;
title2 = "Best     tree = "   + bestTree;
title3 = "Distance to target = " + bestDistance;

decoderLengths(bestTree);

//Absolute distance from the associated distance matrix
//of the tree
//to target matrix is evaluated
//d = absDistance(numberOfLeaves,targetMatrix, M);




double[][] M = associatedMatrix(bestTree);

draw = newContentPane.new DrawBiTree16();
draw.createAndShowGUI();
draw.setLocation(500,  100);


if (!(workDone))
```

```
  System.out.println( "\n \nSOLUTION HAS BEEN FOUND: \n" +
"at generation " + generation);

 System.out.println("\nSummary at  generation " + generation +
 "\nTarget tree   = \n" + targetTree
   + " \nBest tree = \n" + bestTree
   + " \nDistance to target = " + bestDistance
   + " \nBest distance at initialization = "
   + distance0
   + "\nTarget topology = \n" + targetTopology
   + "\nBest topology = \n" + bestTopology);
 printMatrix(M, "Distance matrix associated to best tree");
 printMatrix(targetMatrix,
      "Distance matrix associated to target tree");

}
```

```
//The best fitting tree for chosen topology is found
public static int findTheBest(String tree)
{
 String selectedTopology = simplify(tree);
 String thisTopology = "";
 String thisTree = "";
 int bestPosition = -1;

for(int i = 0; i < indexOfTopologies; i++)
 {
  thisTree = bestTreeOfTopology[i];
  thisTopology = simplify(thisTree);
```

```
  if (selectedTopology.equals(thisTopology))
  {
   bestPosition = i;
   return bestPosition;
  }
 }
return  bestPosition;
}


  /*
  //Lengths with real values are generated
  double lengthsLeaves[]  = new double[l];
  //Last two entries are redundant
  double lengthsInnerLinks[]  = new double[l];
  for(int i = 0; i < numberOfLeaves; i++)
   {
 lengthsLeaves[i] = r.nextDouble();
 lengthsInnerLinks[i] =  r.nextDouble();
   }
  */
```

```
//========= Translation in within formats ==========


//bTree in (modified) Newick's notation
//is used to generate Record[]
//with addresses and contents.
public static void makeInput(String bTree)
```

```
{
 if (print)
 System.out.println("bTree at makeInput ////// " + bTree);
 int counter = 0;
 //length of String   tree
 int l = bTree.length();
 String thisChar = "";
 String address = "";
 String binNumbers = "01";
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + bTree.charAt(i);
if (binNumbers.contains(thisChar))
   address = address + thisChar;
if (leaves.contains(thisChar))
{
 Record[counter] = address;
     counter++;
 Record[counter] = thisChar;

counter++;
address = "";
   }
  }

 if (print)
  for(int i = 0; i < counter; i = i+2 )
  System.out.println(Record[i] +
 " bbbb " + Record[i+1]);

 lastUsedPosition = counter;
 if (print)
System.out.println("lastUsedPosition ==== "
        + lastUsedPosition);
}
```

```
//Outputs a word in reversed order.
private static String reverser(String word)
{
 return new StringBuilder(word).reverse().toString();
}


private static String cleanTree(String tree)
{
 if (print2)
 System.out.println("Input tree at cleanTree = \n "
        + tree);
 String treeClean = "";
 String numeric = ".:0123456789";
 String plusMinus = "+-";
 String thisChar = "";
  boolean leaf = false;
 //Numerics are washed
 //length of String   tree
 int l = tree.length();
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + tree.charAt(i);
   if (numeric.contains(thisChar));
    else treeClean = treeClean + thisChar;
  }
 if (print2)
 System.out.println("Tree without numerics = \n"
+ treeClean);
 //+/- are deleted from inner nodes that accompany leaves
 l = treeClean.length();
 String treeOut = "";
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
 {
  thisChar = "" + treeClean.charAt(i);
```

```
  if (leaves.contains(thisChar))
  leaf  = true;
  if ((leaf) &  (plusMinus.contains(thisChar)));
else treeOut = treeOut + thisChar;
  if (internalNodes.contains(thisChar))
  leaf  = false;
}

 if (print2)
 System.out.println("Tree at output of cleanTree = \n"
+ treeOut);
return treeOut;

}



//The path from each leaf to node a is found.
//The input contains branch's indicator, say:
//((++Aa,+-Ba),(-+Cb,((--++Dd,--+-Fd)--c,---Ec)-b)+a)
//Output for leaf A: Aa+.
//Output for leaf C: Cba-
//Output for leaf D: Ddcba-
private static void findPaths(String tree)
{
 if (print2)
 System.out.println("Input tree at findPaths = \n" + tree);
 //length of String   tree
 int l = tree.length();
 String plusMinus = "+-";
 String thisChar;
 String nextChar;
 String path = "";
 //Branch indicative with respect to root, + else -
 String branch;
 String address ;
 //The number of links of a path to root
```

```
 int level = 0;
 String treeReversed = reverser(tree);
 if (print2)
 System.out.println("Reversed tree = " + treeReversed);

//Each leaf has a path to a
for(int k= 0; k < numberOfLeaves; k++)
{
 String leaf = "" + leaves.charAt(k);
 //Candidate nodes to be part of the path to node a
 //must have  sign +/- as prefix.
 //Or as suffix in the reversed tree.
 int position = treeReversed.indexOf(leaf);
 if (position > -1) // if leaf is in tree
 {
  if (print2)
  System.out.println("\nLeaf = " + leaf );
  address = "";
  boolean go = true;
  //branchIndicator and level are found
  for(int i = position+1 ; ((i < l) & (go)); i++ )
  {
   thisChar = "" + treeReversed.charAt(i);
   if   (plusMinus.contains(thisChar)  )
address = thisChar + address;
   else go = false;
  }
  level = address.length();
  if (print2)
  System.out.println("LeafAddressLevel  = " + leaf
                     + address + level);
  branch = "" + address.charAt(0);


  //Path from leaf to node a is found in (not reversed) tree
  position = tree.indexOf(leaf);
  thisChar = "" + tree.charAt(position);
  nextChar = "" + tree.charAt(position + 1);
```

```
  //Output
  path = "" + leaf + nextChar;
  go = true;
  if  (nextChar.contentEquals("a") ) //leaf is neighbor of a
   {
  path = path + branch;
  if (print)
  System.out.println("Path short = " + path);
   }
  else  //level > 1
  {

  //Tree is scanned char by char to get path
  String iChar = "";
  for(int i = position + 2; ((i < l) & (go)); i++ )
  {
   iChar = "" + tree.charAt(i);


   int newLevel = 0;
   if (internalNodes.contains(iChar))//InternalNodes
{
     if   (iChar.contentEquals("a")  )
      {
   path  = path +  iChar;
   go = false; //end
      }
     else //Internal node different than a
     {
      if (print)
      System.out.println("Internal node not a) = " + iChar );
      int rP = l -i-1; //index of iChar in treeReversed
      String  nextNewChar = "" + treeReversed.charAt(rP+1);

      if ( plusMinus.contains(nextNewChar )) //candidate node
       {
       if (print)
     System.out.println("next candidate = " + iChar);
```

```
        String addressNew = "";

        //level of candidate node iChar  is inquired
        String thatChar = "";
        boolean go2 = true;
        for(int j = rP+1 ; ((j < l) & (go2)); j++ )
         {
       thatChar = "" + treeReversed.charAt(j);
       if   (plusMinus.contains(thatChar)  )
          addressNew = thatChar + addressNew;
       else go2 = false;
         }
        if (print)
        System.out.println("AddressNew " +
                                    iChar + addressNew);
       newLevel = addressNew.length();
       //A path links a node with another in a lower level
       if  ( (newLevel < level) &
        (!(nextChar.contentEquals(iChar))))
        {
       path = path + iChar;
       level = newLevel;
       if (print)
       System.out.println("Path newLevel= " + path);
        }
      } //End of if candidate node
     }//End of else Internal node different than a
}//End of if InternalNodes
   }//End of Tree is scanned char by char

  path = path + branch;

 }//End of  else level > 1
 }//End of if leaf is in tree
 if (print2)
 System.out.println("Path completed  = " + path);
 Paths[k] = path;
}//End of each leaf
```

```
}



//Paths from leaves to node a are found
//Reverse paths are also calculated.
//Input tree in quasibinary notation
//with addresses and lengths
private static void pathFinder(String tree)
{
 //Useless information is deleted
 String tree2 = cleanTree(tree);
 //Find the path to node a from each leaf.
 //Add an indicator of the branch
 //with respect to the root: up(+) else down(-)
  findPaths(tree2);
}

//Input: a long raw string with path from leaf to node a
//plus path from node a to leaf.
//Editing is done to get true paths.
private static String grammar (String path)
{
 int n = path.length();
 String thisChar = "";
 String nextChar = "";
 String oldPath = path;
 String plusMinus = "a+-a";
 String minusPlus = "a-+a";
 String two = "";
 int N = path.length();
//Rule 1: two equal symbols produce nothing
 for(int j=1; j < N; j++)
  {
   for(int i=1; i < n-1;)
   {
    thisChar = "" + path.charAt(i);
    nextChar = "" + path.charAt(i+1);
```

```
    two = "" + thisChar + nextChar;
    if (thisChar.contentEquals(nextChar))
  {
   i = i + 2;
   path = path.replace(two, "");
  }
    else
  {
   i = i+1;
  }
  n = path.length();
   }
  }

 //Rule 2: a+-a -> a

 if  (path.contains(plusMinus))
      path = path.replace(plusMinus, "a");
 if   (path.contains(minusPlus))
     path = path.replace(minusPlus, "a");
 if (print2)
 System.out.println("input = " + oldPath +
                    " output = " + path);
 return path;
}

//The distance matrix among leaves is calculated
private static double[][] matrixBuilder()
{
double M[][] =
new double[numberOfLeaves][numberOfLeaves];
 //Path to node a from leaf
 String pathLeafToa = "";
 //Path to leaf from node a
 String pathaToLeaf = "";
 //Path from leaf to lead
 String pathLeafiToLeafj = "";
 String innerNode = "";
```

```
 double distance;
 for(int i = 0; i < numberOfLeaves; i++)
  for(int j = 0; j < numberOfLeaves; j++)
  {
   pathLeafToa = Paths[i];
   pathaToLeaf = reverser( Paths[j] );
   //Raw paths
   pathLeafiToLeafj = pathLeafToa + pathaToLeaf;
   //An editing grammar to get true paths
   pathLeafiToLeafj = grammar( pathLeafiToLeafj );
   if (i == j) M[i][j] = 0;
   else
    {
 String leaf = "" + pathLeafiToLeafj.charAt(0);
 int p = leaves.indexOf(leaf);
 distance = lengthsLeaves[p];
 int n = pathLeafiToLeafj.length();
 for(int k = 1; k < n-1; k++)
 {
  innerNode =  "" + pathLeafiToLeafj.charAt(k);
  p = internalNodes.indexOf(innerNode);
  distance = distance + lengthsInnerLinks[p];
 }
 leaf =  "" + pathLeafiToLeafj.charAt(n-1);
 p = leaves.indexOf(leaf);
 distance = distance + lengthsLeaves[p];
  M[i][j] = distance;
    }
  }
 return  M;
}



//Translation from Newick's to quasi-binary notation:
//addresses are attached to tree in Newick's notation
//Inner nodes are conserved. Lengths are conserved.
private static String toQuasiBinaryInner(String tree)
```

```
{
 if (print2)
  System.out.println("Input tree at toQuasiBinaryInner " +
                          "= \n" + tree);
 //Output
 String treeToBi = "";
 String numeric = ".:0123456789";
 //The binary address of leaf
 String address = "";
 //The name or content of the tree
 String content = "";
 String thisChar = "";
 String nextChar = "";
 String previousChar = "";
 //length of String   tree
 int l = tree.length();
 //Tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + tree.charAt(i);
   if ( i < l-1) nextChar = "" + tree.charAt(i+1);
    else nextChar = "end";
   if ( i > 0) previousChar = "" + tree.charAt(i-1);
    else previousChar = "start";


   //This char = (
   if (thisChar.contentEquals("("))
    {
     content = content + thisChar;
     address = address + '+';
     if (print)
 System.out.println("Address = " + address);
    }


 //If this char is a leaf
 if (leaves.contains(thisChar))
```

```
  {
   // System.out.println("leaf");
   treeToBi =  treeToBi + content + address + thisChar;
   content = "";
  }

 if (numeric.contains(thisChar))
 treeToBi =  treeToBi + thisChar;

 //If this char is an inner node
 if (internalNodes.contains(thisChar))
  {
   int m = address.length();
//delete last bit from the right
   if (m>0) address = address.substring(0, m-1) ;
   if (address.contentEquals("")) address = "+";
   treeToBi =  treeToBi + address + thisChar;

  }


 //This char = , and goes after an inner node
 if  (thisChar.contentEquals(","))
  {
   //System.out.println
 //("comma : "  + previousChar + thisChar );
   treeToBi =  treeToBi + thisChar ;
   if (print)
 System.out.println("Address at ,1   = " + address);

   if    (numeric.contains(previousChar))
  address = address + '-';//Add a - to the right
   if (print)
 System.out.println("Address at ,2   = " + address);
  }


 if (thisChar.contentEquals(")"))
```

```
  {
treeToBi =  treeToBi + thisChar ;
    //Diphthonge   ),
if   (nextChar.contentEquals(","))
{
     int m = address.length();
 //delete last bit from the right
 if (m >0)   address = address.substring(0, m-1) + "-";
   }
  }

}//End of for(int i...
return treeToBi;
}


//bTree  (in links with lengths notation)
//is used to generate Record[]
//with addresses and contents.
//Names of inner nodes are visible
public static void makeInputInner(String bTree)
{
 int counter = 0;
 //length of String   tree
 int l = bTree.length();
 String thisChar = "";
 String address = "";
 String binNumbers = "+-";
 String numeric = ".:0123456789";
 //String tree is scanned char by char
 for(int i = 0; i < l; i++ )
  {
   thisChar = "" + bTree.charAt(i);
if (binNumbers.contains(thisChar))
   address = address + thisChar;
if ( (leaves.contains(thisChar)) ||
(internalNodes.contains(thisChar)))
{
```

```
 Record[counter] = address;
     counter++;
 Record[counter] = thisChar;

 counter++;
 address = "";
    }

if (numeric.contains(thisChar))
 {
Record[counter-1] = Record[counter-1] + thisChar;
 }

  }


 if (print)
  for(int i = 0; i < counter; i = i+2 )
  System.out.println(Record[i] +
 " bbbb " + Record[i+1]);

 lastUsedPosition = counter;
 if (print)
System.out.println("lastUsedPosition ==== "
      + lastUsedPosition);
}//End of method


//Prints  a matrix and its title
private static void printMatrix(double Matrix[][],
                        String title)
{
 System.out.println( "\n" + title );
 int m = Matrix.length;
 System.out.print(  "\t");
 for ( int j = 0; j < m; j++ )
  System.out.print( leaves.charAt(j)+ "\t");
 System.out.println();
```

```
 for ( int i = 0; i < m; i++ )
  {
   System.out.print(leaves.charAt(i ) + "\t");
   for ( int j = 0; j < m; j++ )
 System.out.print( Matrix[ i ][j]+ "\t");
   System.out.println();
  }
}


//Tree is translated
//from link notation to binary.
//Output in Record[]: even entries= addresses, odd = names.
//Associated matrix is also calculated
public static double [][] associatedMatrix(String tree)
{
 /*
 System.out.println("Tree  at translate = " + tree);
 //String NwTree = simplify(tree);
 String bTree = toBinary(tree);
 //A vector with addresses and names is composed
 makeInput(bTree);
 */
//String NwTree = simplify(tree);
 String bTree = toQuasiBinaryInner(tree);

 if (print2)
 {
 System.out.println("Input tree  at translate = "    + tree);
  System.out.println("Selected tree in " +
  "pseudo-Newick's notation = \n" + simplify(tree));
  System.out.println("Tree in quasi-binary notation: \n"
              + bTree);
 }

 //A vector with addresses and other with names are composed
 makeInputInner(bTree);
 //Paths from leaves to node a are found.
```

```
 //Reverse paths are also listed.
 pathFinder(bTree);
 //The distance matrix among leaves is calculated
 double M[][] = matrixBuilder();
 return M;
}



//The absolute distance among two matrices is calculated.
//Only triangular part of matrices are compared.
private static double absDistance(int n,
               double[][] M1, double[][] M2)
{
 double abs = 0;
 for(int i = 0; i < n; i++)
 for(int j = i+1; j < n; j++)
   abs = abs + Math.abs(M1[i][j]-M2[i][j]);
 return abs;
}


//Input : tree in link notation.
//Output:
//distance matrix associated to the leaves of the tree.
private static double[][] distanceMatrix(String tree)
{
String s = toQuasiBinaryInner(tree) ;
if (print4)
 System.out.println("\n Tree = " + s);
//Paths from leaves to node a are found.
//Reverse paths are also listed.
pathFinder(s);
//The distance matrix among leaves is calculated
double M[][] = matrixBuilder();
return M;
}
```

```
//==========================================
//==Inner class  for drawing selected tree ==
//==========================================


//This program translates a string that
//denotes a tree in Newick's notation
//to our binary notation
//(assign a zero to the left, a one to the right).

//A generator of random trees in Newick's notation
//is included.

//Resultant trees are drawn.




class DrawBiTree16 extends JPanel

{
 private static final long serialVersionUID = 1L;
 private JPanel drawingPane36;
 JScrollPane scroller;



 //=====================================



 //Vertical  size of the panel
 int verticalSize;
```

```
 int horizontalSize;
//Scale must be a power of 2
 int scale = 1;

 //Symbolic horizontal length of each stroke
 int horizontalLength = 30;

 int verticalLength = 10;



 String  NwTree = "";



 boolean print = false;



 //This is our encoding for a tree in our binary notation:
 //We use a vector of strings:
 //Even entries are the paths in the tree  of the nodes,
 //odd ones are the strings, names or contents.



 //Y-coordinates of points where strings are drawn
 int[] yCoordinate = new int[1000];
 int maxValueOfY = 0;
  int minValueOfY = 100000;


 //Set up the instructions.
 JLabel instruct1;
 JLabel instruct2;
 JLabel instruct3;
```

```
public DrawBiTree16()
{
 super(new BorderLayout());


 JPanel instructionPanel = new JPanel(new GridLayout(0,1));


 //Set up the instructions.
 instruct1 = new JLabel(title1);
 instruct2 = new JLabel(title2);
 instruct3 = new JLabel(title3);

 instructionPanel.setFocusable(true);
 instructionPanel.add(instruct1);
 instructionPanel.add(instruct2);
 instructionPanel.add(instruct3);

 //Set up the drawing area.
 drawingPane36 = new drawingPane36();
 drawingPane36.setBackground(Color.white);

 //Put the drawing area in a scroll pane.
 JScrollPane scroller = new JScrollPane(drawingPane36);
 scroller.setPreferredSize(new Dimension(500,900));
 scroller.revalidate();

 //Lay out this demo.
 add(instructionPanel, BorderLayout.PAGE_END);
 add(scroller, BorderLayout.CENTER);
}


//=========================================
//============= Painting class ============
//=========================================
```

```
/** The component inside the scroll pane. */
 class drawingPane36 extends JPanel
{
 private static final long serialVersionUID = 1L;

 //Initialization of coordinates
 int centerX = 1;
 int centerY = 1;
 Dimension center = new Dimension(centerX,centerY);
 //Coordinates of the origin
 int x00;
 int y00;
 //Target coordinates
 int x11 ;
 int y11;
 //Horizontal and basic vertical strokes
 int H;
 int V;
 //Addresses are interpreted as paths
 String path = "";
 char direction;
 int xMax = 0;
 //Repainting cause some coordinates to change:
 //work with the first ones
 boolean first = true;


//Calculate the size of a box to hold the tree
//Scrolling controls are managed
private void sizeOfTree(Graphics2D g2d)
{
 //Calculate the dimension of the strings
 //All strings have equal size.

 //Get   font in use
 Font font = g2d.getFont();
 //get metrics from the graphics
 FontMetrics metrics = g2d.getFontMetrics(font);
```

```
 //get the advance of my text in this font
 //and render context
 int lengthText = metrics.stringWidth("Aa") + 2;
 //get the height of a line of text in this
 //font and render context (some padding is included)
 int hgt = metrics.getHeight()  + 8;
 if (print)
  System.out.println("height of letters = " + hgt);

 //Horizontal and basic vertical strokes
 H = scale * horizontalLength;
 V = scale * verticalLength;

 horizontalSize = lengthText +
              (int) (H *  numberOfLeaves);
 verticalSize =
 (int) (hgt *   numberOfLeaves);


 //area = size of the graphic (width, height)
 Dimension area =
 new Dimension(horizontalSize,20*verticalSize);
 //area is set as the preferred size of the graphic
 drawingPane36.setPreferredSize(area);
 //The graphic demands a space for itself:
 //if the alloted space is not enough,
 //turn on scrolling bars.
 drawingPane36.revalidate();
 verticalLength = 2*verticalSize;
 //Coordinates of the origin
 x00 =    scale*centerX;
 y00 =   scale* centerY + verticalSize  - 600;
 //Target coordinates
 x11 =  x00 + (int) ( scale*horizontalLength);
 y11 =  y00;

}
```

```
//Tree is drawn
private void DrawTree3(Graphics2D g2d, Dimension center)
{
//The tree is drawn:
//A path from the origin of the tree to
//each leaf is drawn
 if (print)
 System.out.println("LastUsedPosition at DrawTree3=  "
                              + lastUsedPosition);
 for(int i = 0; i < lastUsedPosition; i = i+2)
  {
//Addresses will be used as paths to  be drawn
path = Record[i];
if (print)
System.out.println( i + " Path = " + path);
//Reinitialization
Dimension newCenter =
new Dimension(center.width,center.height);
V = scale * verticalLength;
//Path is interpreted
for(int j = 0; j < path.length(); j++)
 {
  direction = path.charAt(j);
  int x0 =  newCenter .width;
  int y0 =  newCenter .height;
  int x1 =  x0;
  int y1;
  if (direction == '+') //go up
   {
    y1 =  y0 + V;
    newCenter.height = newCenter.height + V;
   }
   else // if direction == '-'  =  go down
   {
     y1 =  y0 - V;
     newCenter.height = newCenter.height - V;
```

```
   } //End of if + else
  //Draw vertical stroke
  g2d.drawLine(x0, verticalSize -y0, x1,
     verticalSize- y1);
  //Draw horizontal stroke
  g2d.drawLine(x1, verticalSize - y1,
        x1 + H,  verticalSize - y1);
  if (verticalSize - y1 > maxValueOfY)
    maxValueOfY = verticalSize - y1;
  if (verticalSize - y1 < minValueOfY)
    minValueOfY = verticalSize - y1;
  //Move center
  newCenter.width = newCenter.width + H ;
  //Vertical stroke is divided in two
  V = V/2;
  //xMax = left extreme of the graphic
  if (first)
     if (xMax < x1 + H) xMax = x1 + H;

 }//End of for j
   }  //End of for: numberOfLeaves

 //System.out.println("xMax = " + xMax);
}



//Strings are drawn
private void drawStrings(Graphics2D g2d, Dimension center)
{
 int counter = 0;
 //Seed is not  drawn: we lack one
 //g2d.drawString(seed,x00,verticalSize - y00);
 yCoordinate[counter] = verticalSize - y00;
 //counter++;
 //Strings are drawn:
 for(int i = 0; i < lastUsedPosition; i = i+2)
 {
```

```
  //Addresses will be used as paths to the places
  //where strings must be drawn
  path = Record[i];
  String s = Record[i+1];
  //Reinitialization
  Dimension target =
  new Dimension(center.width,center.height);
  V = scale * verticalLength;
  //Path is interpreted
  for(int j = 0; j < path.length(); j++)
{
 direction = path.charAt(j);
 if (direction == '+') //go up
 target.height = target.height + V;
 else // if direction == '-'  =  go down
 target.height = target.height - V;
 target.width = target.width + H ;
 V = V/2;
}//End of for j
//String is drawn
g2d.drawString(s, target.width + 1,
                      verticalSize - target.height-1);
//y-coordinates are recorded
yCoordinate[counter] = verticalSize - target.height-4;
 counter++;
  }
}


//Basic routine of the drawing procedure
private void drawBasic(Graphics2D g2d)
{
 g2d.setColor(Color.red);
 /*
 //Stem is drawn: our trees is artificially rooted.
 g2d.drawLine(x00, verticalSize - y00,
      x11,  verticalSize - y11);
*/
```

```
sizeOfTree(g2d);

//targetTree is drawn
decoderLengths(targetTree);

//Absolute distance from the associated distance matrix
//of the tree
//to target matrix is evaluated
//d = absDistance(numberOfLeaves,targetMatrix, M);



associatedMatrix(targetTree);

center = new Dimension(x11,y11);
DrawTree3(g2d, center);
center = new Dimension(x00,y00);
drawStrings(g2d,center);
first = false;



//bestTree is drawn


decoderLengths(targetTree);

//Absolute distance from the associated distance matrix
//of the tree
//to target matrix is evaluated
//d = absDistance(numberOfLeaves,targetMatrix, M);



associatedMatrix(bestTree);
```

```
 center = new Dimension(xMax + 30,y11);
 DrawTree3(g2d, center);
 center = new Dimension(
 xMax - (int) ( scale*horizontalLength) + 30,y00);
 drawStrings(g2d,center);

}

//Graphical  master : it is called automatically
//at starting and by repaint()
protected void paintComponent(Graphics g)
{
 super.paintComponent(g);
 Graphics2D g2d = (Graphics2D)g;
 //Font to draw strings
 Font font = new Font("Arial", Font.PLAIN, 12);
 g2d.setFont(font);
 g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
  RenderingHints.VALUE_ANTIALIAS_ON);
  drawBasic(g2d);
}


}//End of double inner class drawingPane36



/**
* Create the GUI and show it.  For thread safety,
* this method should be invoked from the
* event-dispatching thread.
*/
 void createAndShowGUI()
{
 //Create and set up the window.
 JFrame frame =
   new JFrame(titleFrame);
```

```
 frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

 //Create and set up the content pane.
 //Constructor of the main application is used.
 JComponent newContentPane = new DrawBiTree16();
 //content panes must be opaque
 newContentPane.setOpaque(true);
 frame.setContentPane(newContentPane);

 //Display the window.
 frame.pack();
 frame.setVisible(true);
}

}//End of  inner class DrawBiTree16

}//End of main class K55 EvoFull5
```

59, page 248. Tiny mutations were shown to tremendously accelerate the approaching to zero error for the case with 8 leaves. Why?

Gradualism helps near the optimum because before everything else the concept of function is in this case graded: the error is small near the optimum and the error is proportional to the error in lengths. Therefore, changes must be tiny. By contrast, large changes near the optimum would case a falling into regions dominated by extremes less optimal that the achieved one.

Now, can we say that this is the gradualism of Darwin? The problem is that Darwin did not have a mechanistic theory but only a phenomenological one. The possibility of a mechanistic science was provided by Mendel and molecular biology. In this context gradualism is softly defined when tiny changes in the genome cause tiny changes in function and in fitness. But to be accurate, we must demand from changes to be at random with respect to encoding. But it happens that our tiny mutations were accommodated on top of a machinery that causes a twisting of randomness by enabling mutations that will be viable with certainty. So, we are as far from Darwinism as the day is from night. Nevertheless, an intrigue arises: what will happen if we deprive our program from all that Lamarckian machinery and rely on tiny mutation along pure Darwinism?

62, page 259. Program K62 EvoFull 8 simulates almost pure Darwinian evolution to solve the problem of fitting a tree to a distance matrix. The program relies on tiny mutations which must be understood as punctual mutations, changes of one char by another. The program includes a method for testing whether or not a string correctly encodes for a tree. A string that fails is understood as an abort. The cost in aborts of each viable string is reported. If viable mutants are produced, they can recombine. If recombination produces an inviable mutant, it is not allowed otherwise the fist recombining string is cloned. This measure is not Darwinian, it is Lamarckian. Whether or not recombination was possible is reported. This program shows that almost pure Darwinism that rely on tiny mutations is a no-go methodology for attacking the problem of fitting a tree to a distance matrix for a number of leaves pass 5. The piece of code responsible for recombination follows:

```
//Program K62 EvoFull8
/*
 * We implement here
 * a purely Darwinian mutational process
 * to propose an approximate solution to the problem of
 * fitting a tree to a distance matrix.
 *
 * Mutations are tiny and by this we understand
 * random mutations of a single char.
 *
 *
 * It is found that
 * the usual output of mutation over a viable string
 * is an inviable string.
 * Nevertheless, a viable mutant always can be found
 * if only a price in aborts is paid.
 * That price is reported.
 *
 * The method that checks for the viability of strings
 * was developed by intuition but
 * refined by trial and error.
 * So, some rules might remain to be discovered.
 *
```

```
 * Recombination is enabled and allowed when
 * the recombinant product is viable.
 * Otherwise, the first recombining individual is cloned.
 ...
 */

 public class EvoFull8 extends JPanel
{
...


//Strings s and t recombine if the output is viable.
//Otherwise, string s is cloned.
private  static String rec(String s, String t)
{

  String  t2 = "";
  String  s1= s;

  String recombinant = "";
int m= s.length();
  if (m>2)
   {
    int m1 = r.nextInt(m-1)+1 ;
  s1= s.substring(0,m1);
  t2= t.substring( m1);
 }
  //Recombination
  recombinant = s1.concat(t2);
  //return recombinant;

  if (isTree(recombinant))
   {
    if (print6)
      System.out.println(
      "\nRECOMBINATION was possible \n");
    return recombinant;
   }
```

```
  else
  {
   if (print6)
      System.out.println(
      "\nRECOMBINATION was NOT possible \n");
   return s;
  }

}


...

}
```

63, page 259. Program K63 EvoFull9 studies the cost of synthesizing by mere randomness a string that encodes for a tree. One can choose the number of leaves beginning from 4. The responsible part for the synthesis of strings is the following:

```
//Program K63 EvoFull9
/*
 * This program studies the cost in trials
 * of the synthesis by mere randomness of
 * a string that encodes for a tree.
 *
 *
 * The method that checks a string
 * to decide whether or not it encodes for a tree
 * was developed by intuition but
 * refined by trial and error.
 * So, some rules might remain to be discovered.
 *
 * If eventually an initial population of strings
 * that encode for trees is synthesized by mere
 * randomness, it is subject to
 * almost Darwinian evolution.
```

```
 *
 *
 */

 ...

public class EvoFull9 extends JPanel
{

...

// Return a character chosen randomly from   alphabet.
private static char randomChar(String alphabet)
{
 int k = r.nextInt(alphabet.length());

return alphabet.charAt(k);
}



//Return a randomly generated string of the given length.
private static String findTree(int len,
                               String alphabet)
 {
  String s = "";
  boolean go = true;
  int counter = 0;
  while (go)
  {
   s = "";
   for (int i = 0; i < len; ++i)
 s = s + randomChar(alphabet);
    if (print7)
  System.out.println("\nTrial number = " + counter
 + "\n"    + s );
   if (isTree(s))
```

```
   {
    go = false;
    counter = 0;
    if (print7)
  System.out.println("\nTrial number = " + counter
 + " Tree found");
    System.exit(0);
   }
   else
{
 counter++;

}
   // if (counter == 10) go = false;
  }
  return s;
 }


//The population is initialized with n trees.
//Fitness is assessed:
//Has the solution been guessed?
private static void initialization()
{
 System.out.println("\n====" +"INITIALIZATION : ");
//Number of right parenthesis
String rightPar = ")";
for(int i = 0; i < targetTree.length(); i++)
{
String c = "" + targetTree.charAt(i);
if ( c.contentEquals(rightPar)) nRightParentheses++;
}
 if (print7)
  System.out.println("N random trees are generated \n");
 String numbers = "0123456789";
 String alphabet = presentLeaves + presentNodes + numbers
                     + "(),:";
 //N trees   are generated by mere randomness
```

```
 for(int i = 0; i < N; i++)
 {
  int length = targetTree.length();
  trees[i] = findTree(length,  alphabet);

  if (print7)
System.out.println("\n" +  i + "   " + trees[i] );
 }


assessFitnesses();
scaleFitnesses();
}

...

}
```

4.9 nothing

Bibliography

[1] GRANT P, B GRANT (2014) Nature 507, 178–179 doi:10.1038/507178b
http://www.nature.com/nature/journal/v507/n7491/full/507178b.html
Verified 29/III/2014

[2] LOBO I Pleiotropy: One gene can affect multiple traits. Nature Education 1(1):10
http://www.nature.com/scitable/topicpage/pleiotropy-one-gene-can-affect-multiple-traits-569
Verified 29/III/2014