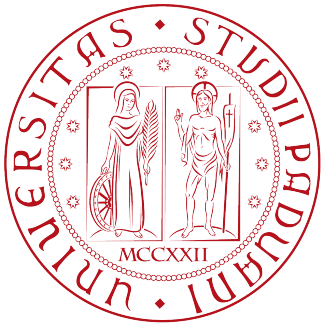

Tesi di Laurea in
Ingegneria Informatica

Implementation of Entropic Profiler for DNA Sequences by using Suffix Trees

Laureando:
Stefano Mazzocca

Relatore:
Matteo Comin

28 Marzo 2013

Contents

INTRODUCTION

Desoxyribonucleic acid (DNA) is a molecule encoding the genetic instructions used in the development and functioning of all known living organisms. It is one of the three major macromolecules that are essential for all known forms of life and its genetic information is encoded as a sequence of nucleotides (guanine, adenine, thymine, and cytosine) so its digital information can be represented by a one-dimensional character string of G's, A's, T's and C's.
The increasing availability of biological sequences requires an IT development which allows classification of a huge collection of data: in this regard can be useful the concept of Entropic Profiler (EP) which can extract and classify relevant and statistically significant segments of DNA sequence. The study of these motifs is very important because under or over-representation segments are often associated with significant biological meaning.
The Entropic Profiler's properties are well discussed in [1], [2] and [3] and the main purpose of this work is to illustrate a Java implementation of the algorithm proposed in [3] which uses suffix tree realized with Ukkonen's algorithm [4].
It is recalled that EP function is:

$$g_{L,\phi}(i) = \frac{1 + 1/n \sum_{k=1}^{L} 4^k \phi^k c[i-k+1, i]}{\sum_{k=0}^{L} \phi^k} \qquad (0.1)$$

where n is the length of the sequence, L is the length resolution chosen, φ is a smoothing parameter and $c[i-k+1, i]$ is the number of times the substring of length k that ends at position i appears in the whole sequence. Nevertheless, for ease of explanation, we redefine the above formula to evaluate the statistic of words starting at position i, instead of ending at position i so the new EP function is:

$$f_{L,\phi}(i) = \frac{1 + 1/n \sum_{k=1}^{L} 4^k \phi^k c[i, i+k-1]}{\sum_{k=0}^{L} \phi^k} \qquad (0.2)$$

In this way function (0.1) is equivalent to compute $f_{L,\phi}(n-i)$ for the reverse of the given string.

CHAPTER I

IMPLEMENTATION CODE

Below is available the complete java code implementing the Entropic Profiler and, in next section, a detailed explanation of every part is given. Italic and smaller rows indicate previous written code [5] implementing suffix tree's pure structure and won't be analyzed.

```
import java.util.Map;
import java.util.HashMap;
import java.util.Collection;
import java.lang.Math;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.util.StringTokenizer;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.util.NoSuchElementException;

 class Node{
        private SuffixTree suffixTree;

       private Node suffixNode;
       private Map<Character, Edge> edges;
       private int name;
       private int count=0;
       private int length=0;
       private double entropy=0;

        public Node(Node node, Node suffixNode) {
            this(node.suffixTree, suffixNode);
        }

        public Node(SuffixTree suffixTree, Node suffixNode) {
            this.suffixTree = suffixTree;
            name = suffixTree.getNewNodeNumber();

            this.suffixNode = suffixNode;
            edges = new HashMap<Character, Edge>();
        }

        public char charAt(int index) {
            return suffixTree.getText().charAt(index);
        }

        public void addEdge(int charIndex, Edge edge) {
            edges.put(charAt(charIndex), edge);
        }

        public void removeEdge(int charIndex) {
            edges.remove(charAt(charIndex));
        }

        public Edge findEdge(char ch) {
            return edges.get(ch);
        }

        public Node getSuffixNode() {
```

```
        return suffixNode;
    }

    public int getName(){
        return name;
    }

    public void setSuffixNode(Node suffixNode) {
        this.suffixNode = suffixNode;
    }

    public Collection<Edge> getEdges() {
        return edges.values();
    }

    public String toString() { //Override
        return ((Integer) name).toString();
    }

      public int getCount(){return count;}
      public void setCount(int i){count=i;}
      public int getLength(){return length;}
      public void setLength(int i){length=i;}
      public boolean isLeaf(){return !(name==0)&&suffixNode==null;}
      public void setEntropy(double i){entropy=i;}
      public double getEntropy(){return entropy;}
     }


class Edge{
    private int beginIndex;     // can't be changed
    private int endIndex;
    private Node startNode;
    private Node endNode;       // can't be changed, could be used as edge id

    // each time edge is created, a new end node is created
    public Edge(int beginIndex, int endIndex, Node startNode) {
        this.beginIndex = beginIndex;
        this.endIndex = endIndex;
        this.startNode = startNode;
        this.endNode = new Node(startNode, null);
    }

    public Node splitEdge(Suffix suffix) {
        remove();
        Edge newEdge = new Edge(beginIndex, beginIndex + suffix.getSpan(),
                suffix.getOriginNode());
        newEdge.insert();
        newEdge.endNode.setSuffixNode(suffix.getOriginNode());
        newEdge.getEndNode().setLength(newEdge.getEndIndex()-
          newEdge.getBeginIndex()+1+newEdge.getStartNode().getLength());
        beginIndex += suffix.getSpan() + 1;
        startNode = newEdge.getEndNode();
        insert();
        return newEdge.getEndNode();
    }

    public void insert() {
        startNode.addEdge(beginIndex, this);
    }

    public void remove() {
        startNode.removeEdge(beginIndex);
    }

    public int getSpan() {
        return endIndex - beginIndex;
    }

    public int getBeginIndex() {
        return beginIndex;
    }
```

```
    public int getEndIndex() {
        return endIndex;
    }

    public void setEndIndex(int endIndex) {
        this.endIndex = endIndex;
    }

    public Node getStartNode() {
        return startNode;
    }

    public void setStartNode(Node startNode) {
        this.startNode = startNode;
    }

    public Node getEndNode() {
        return endNode;
    }

    public String toString() {    //Override
        return endNode.toString();
        }

    public boolean isSpecial(){
          if(endIndex<beginIndex) return true;
          else return false;
      }
    }


class Suffix {
    private Node originNode;
    private int beginIndex;
    private int endIndex;

    public Suffix(Node originNode, int beginIndex, int endIndex) {
        this.originNode = originNode;
        this.beginIndex = beginIndex;
        this.endIndex = endIndex;
    }

    public boolean isExplicit() {
        return beginIndex > endIndex;
    }

    public boolean isImplicit() {
        return endIndex >= beginIndex;
    }

    public void canonize() {
        if (!isExplicit()) {
            Edge edge = originNode.findEdge(originNode.charAt(beginIndex));

            int edgeSpan = edge.getSpan();
            while (edgeSpan <= getSpan()) {
                beginIndex += edgeSpan + 1;
                originNode = edge.getEndNode();
                if (beginIndex <= endIndex) {
                    edge = edge.getEndNode().findEdge(originNode.charAt(beginIndex));
                    edgeSpan = edge.getSpan();
                }
            }
        }
    }

    public int getSpan() {
        return endIndex - beginIndex;
    }

    public Node getOriginNode() {
        return originNode;
    }
```

```
        public int getBeginIndex() {
            return beginIndex;
        }

        public void incBeginIndex() {
            beginIndex++;
        }

        public void changeOriginNode() {
            originNode = originNode.getSuffixNode();
        }

        public int getEndIndex() {
            return endIndex;
        }

        public void incEndIndex() {
            endIndex++;
        }
    }


public class SuffixTree{

     private String text;
     private Node root;
     private int nodesCount;
     private double phi;
     private double[] maxEntropy;
     private double[] keepMaxInfo;
     private int[] nodesCountPerLength;

    public SuffixTree(String text, double phi){
     nodesCount = 0;
     maxEntropy = new double[text.length()];
     keepMaxInfo = new double[2];
     nodesCountPerLength = new int[text.length()];

      this.text = text +"$";
     root = new Node(this, null);

     Suffix active = new Suffix(root, 0, -1);
      for (int i = 0; i <= text.length(); i++) {
          addPrefix(active, i);
        }
     setAllCount(root);
     root.setCount(0);
     setEntropy(phi);
     }
     }

    private void addPrefix(Suffix active, int endIndex) {
        Node lastParentNode = null;
        Node parentNode;

        while (true) {
            Edge edge;
            parentNode = active.getOriginNode();

            if (active.isExplicit()) {
                edge = active.getOriginNode().findEdge(text.charAt(endIndex));
                if (edge != null) {
                    break;}
            }

            else {
                edge = active.getOriginNode().findEdge(text.charAt(active.getBeginIndex()));
                int span = active.getSpan();

                if (text.charAt(edge.getBeginIndex() + span + 1) == text.charAt(endIndex)){
```

```
                break;}
            parentNode = edge.splitEdge(active);
        }

        Edge newEdge = new Edge(endIndex, text.length() - 1, parentNode);
        newEdge.insert();
        newEdge.getEndNode().setLength(text.substring(newEdge.getBeginIndex(),
             newEdge.getEndIndex()).length()+parentNode.getLength());
       newEdge.setEndIndex(newEdge.getEndIndex()-1);
       updateSuffixNode(lastParentNode, parentNode);
       lastParentNode = parentNode;

        if (active.getOriginNode() == root)
            active.incBeginIndex();
        else
            active.changeOriginNode();
        active.canonize();
    }
    updateSuffixNode(lastParentNode, parentNode);
    active.incEndIndex();
    active.canonize();
}

private void updateSuffixNode(Node node, Node suffixNode) {
    if ((node != null) && (node != root)) {
        node.setSuffixNode(suffixNode);
    }
}

public int getNewNodeNumber() {
    return nodesCount++;
}

public boolean contains(String str) {
    int index = indexOf(str);
    return  index >= 0;
}

public int indexOf(String str) {
    if (str.length() == 0)
        return -1;

    int index = -1;
    Node node = root;

    int i = 0;
    while (i<str.length()) {
        if ((node == null) || (i == text.length()))
            return -1;

        Edge edge = node.findEdge(str.charAt(i));
        if (edge == null)
            return -1;

        index = edge.getBeginIndex()-i;
        i++;

        for(int j=edge.getBeginIndex()+1; j<=edge.getEndIndex(); j++) {
            if (i == str.length())
                break;
            if (text.charAt(j) != str.charAt(i))
                return -1;
            i++;
        }
        node = edge.getEndNode();
    }
    return index;
}

public String getText() {
    return text;
}

public Node getRootNode() {
    return root;}
```

```
 private int setAllCount(Node v){

     if(v.isLeaf()){v.setCount(1);
     return 1;}

     else{
         Edge e[]= v.getEdges().toArray(new Edge[0]);
         int x=0;
         for (int i = 0; i < e.length; i++){
             x=x+setAllCount(e[i].getEndNode());
         }
     v.setCount(x);
     return x;
     }
 }


 private void setEntropy(double phiGiven){
   phi=phiGiven;
   setEntropyByNode(root);
   for(int i=0; i<maxEntropy.length; i++){
     maxEntropy[i]=0;
   }
   keepMaxInfo[0]=0;
   keepMaxInfo[1]=0;
}


 private void setEntropyByNode(Node v){
    Edge e[]= v.getEdges().toArray(new Edge[0]);
    for(int i=0;i<e.length;i++){

     double partial=getSummation(0, e[i].getEndNode().getLength(), 4*phi)-
               getSummation(0, e[i].getStartNode().getLength(), 4*phi);
     double entropy=partial*e[i].getEndNode().getCount();


     e[i].getEndNode().setEntropy( e[i].getStartNode().getEntropy()+entropy);


     if (!e[i].getEndNode().isLeaf()) {
          setEntropyByNode(e[i].getEndNode());
        }
    }
 }


 public double getMaxEntropy(int length){
    if(maxEntropy[length-1]==0) searchMax(length), (int)keepMaxInfo[0]+1);
    return maxEntropy[length-1];
 }


 private void searchMax(int length, int index){
    double[] a=new double[3];
    double mpm=keepMaxInfo[1];
         for(int l=index; l<=length; l++){
             a[2]=1;
             a=searchMaxPerLength(root, a, mpm, l);
             nodesCountPerLength[l-1]=(int)a[2];
```

```
            double numerator=1+(a[0]/(text.length()-1));
            double denominator= getSummation(0, l, phi);
            maxEntropy[l-1]= numerator/denominator;
            if(a[1]==0) mpm=a[0]+Math.pow(4*phi, l+1);
            else {mpm=0; a[0]=0;}
    }
    keepMaxInfo[0]=length;
    keepMaxInfo[1]=mpm;
 }


 private double[] searchMaxPerLength(Node v, double[] a, double compare,
                                     int length){

        Edge edges[]= v.getEdges().toArray(new Edge[0]);
        for(int i=0; i<edges.length; i++){


        if(v.getLength()+edges[i].getSpan()+1>=length) {
           double entropy=v.getEntropy()+
                  edges[i].getEndNode().getCount()*getSummation(
                  v.getLength()+1, length, 4*phi);

        if(entropy>a[0]){
           a[0]=entropy;

        if( (length == edges[i].getEndNode().getLength()) &&
             edges[i].getEndNode().isLeaf() ) a[1]=1;
           else a[1]=0;
                }

        else if(entropy==a[0] && a[1]==1 && !(length ==
                edges[i].getEndNode().getLength()) &&
                edges[i].getEndNode().isLeaf())
                a[1]=0;

        }

        else {
                if(edges[i].getEndNode().getEntropy()+
                  (edges[i].getEndNode().getCount()-1)*getSummation(
                  (edges[i].getEndNode().getLength()+1,length,4*phi)>=
                   compare  &&  !edges[i].isSpecial()){

                   a[2]+=1;
                   a=searchMaxPerLength(edges[i].getEndNode(), a, compare,
                                        length);}
                 }
          }
  return a;
 }


public double[] getNormalizedEntropy(int position, int lengthChosen){
    double a[]=new double[2];
    a[1]=getEntropy(position, lengthChosen);
    a[0]=a[1]/getMaxEntropy(lengthChosen);
    return a;
 }
```

```
public double getEntropy(int position, int lengthChosen){
   double main= searchEntropy(root, position, lengthChosen-1);
   double numerator=1+(main/(text.length()-1));
   double denominator= getSummation(0, lengthChosen, phi);
   return numerator/denominator;
}


private double searchEntropy(Node v, int currentPosition, int stepsLeft){

  double value;
  Edge e=v.findEdge(text.charAt(currentPosition));

  int compare=currentPosition;

  while((currentPosition-compare<e.getSpan())&&(stepsLeft!=0)){
      currentPosition++;
      stepsLeft--;}

  if(stepsLeft!=0){
     return searchEntropy(e.getEndNode(),currentPosition+1,stepsLeft-1);
  }

  else {
      if(currentPosition-compare==e.getSpan()){
          value= e.getEndNode().getEntropy();
      }

      else{
          double entropyNode=e.getStartNode().getEntropy();
          double addingPartial=getSummation(e.getStartNode().getLength()+1,
                 e.getStartNode().getLength()+1+currentPosition-compare,
                 4*phi);
          value=entropyNode+(addingPartial*e.getEndNode().getCount());
      }
      return value;
  }
}


private double getSummation(int begin, int end, double param){
    if(param!=1){
        return (Math.pow(param, begin) - Math.pow(param, end+1))/(1-param);
    }
    else return (double) end-begin+1;
}
```

```
public static void main( String args[] )throws IOException{

      try{
            long start=System.nanoTime();

            int begin=Integer.parseInt(args[0]);
            int length=Integer.parseInt(args[1]);
            double phi=Double.parseDouble(args[2]);
            int gap=Integer.parseInt(args[3]);

            String text;

            if(args[4].length()>=4 && args[4].substring
                  (args[4].length()-4).equals(".txt")){
                BufferedReader br = new BufferedReader
                  (new FileReader(args[4]));
                text=br.readLine();
            }
            else text=args[4];

            SuffixTree st = new SuffixTree(text, phi);

            InputStreamReader reader = new InputStreamReader (System.in);
            BufferedReader myInput = new BufferedReader (reader);
            int x=0;
            String input= new String();
            StringTokenizer token;

            long finish = System.nanoTime() - start;
            System.out.println("\nTime to build suffix tree: "+
                              (finish / 1000000000.0) + " seconds" );

            while(true){

            long start2=System.nanoTime();

            FileWriter fw=new FileWriter("Normalized Entropy List "+x+".txt");
            PrintWriter out=new PrintWriter(fw);
            out.println("~~ Normalized Entropy List from position "+begin+
                     " with length "+length+", gap "+gap+" and phi="+phi+" ~~");
            out.println();
            for(int i=begin; i<=begin+gap; i++){
                  out.print(st.getNormalizedEntropy(i, length)[0]+" ");
            }
            out.close();


            FileWriter fw1=new FileWriter(" Visited Nodes "+x+".txt");
            PrintWriter out1=new PrintWriter(fw1);
            out1.println("~~ Number of Visited Nodes from length 1 to
                        length "+length+" with phi="+phi+" (granularity 5) ~~");
            out1.println();
            for(int i=0; i<length; i+=5){
                  out1.println(st.nodesCountPerLength[i]);
            }
            out1.close();

            long finish2 = System.nanoTime() - start2;
            System.out.println("Time to execute the query: "+
                              (finish2 / 1000000000.0) + " seconds\n" );
```

```
            input = myInput.readLine();
            if(input.equals("q")| input.equals("Q")) break;

            x++;
            token = new StringTokenizer(input);
            begin = Integer.parseInt(token.nextToken());
            length = Integer.parseInt(token.nextToken());
            double phiNew = Double.parseDouble(token.nextToken());
              if(phiNew!=phi) {
                    long start3=System.nanoTime();
                    st.setEntropy(phiNew);
                  phi=phiNew;
                  long finish3 = System.nanoTime() - start3;
                  System.out.println("\nTime to update nodes entropies: "+
                                     (finish3 / 1000000000.0) + " seconds" );
              }
            gap = Integer.parseInt(token.nextToken());
          }
      }

      catch(ArrayIndexOutOfBoundsException e){
            System.out.println("\n!!ERROR!!\nNot enough parameters or
              invalid values. In order are required:\n\n[Beginning Position],
              [Chosen Length], [Phi Value], [Gap], [Sequence] (understood
              as \"NameSequence.txt\" or, directly, \"Sequence\")]\n\n
              Every value must be greater than 0 and only \"Beginning Position\"
              and \"Gap\" can be equal to 0.\nIf it is not the first query,
              last parameter is not required");
            }

      catch(NullPointerException e){
            System.out.println("\n!!ERROR!!\n\"Gap\" / \"Chosen Length\"
              or both parameters too high");
            }

      catch(StringIndexOutOfBoundsException e){
            System.out.println("\n!!ERROR!!\nInvalid \"Beginning Position\"
              parameter: it must be greater than 0 and lower than sequence
              length");
            }

        catch(FileNotFoundException e){
            System.out.println("\n!!ERROR!!\nFile \""+args[4]+"\" not found");
            }

        catch(NoSuchElementException e){
            System.out.println("\n!!ERROR!!\nNot enough parameters.In order are
              required:\n\n[Beginning Position], [Chosen Length], [Phi Value],
              [Gap]");
            }

        catch(NumberFormatException e){
            System.out.println("\n!!ERROR!!\nWrong digit");
            }
}
```

ANALYSIS

As the reader can see, some packet was added on the top and *Suffix* class has not been changed. *Node* class remained unchanged except for *count* and *length* variables and some easily understandable method. *entropy* variable is also added and represents the numerator's summation of formula (0.2), i.e. formula (1.1). *Edge* class was amplified in *splitEdge* method by adding an instruction which calculates nodes length: in fact, during tree construction, an edge can be split in two (if it represents a substring with length bigger than one) creating a new node within so this new node length needs to be set. Boolean *isSpecial* method is also available and it returns *true* if given edge represents only *$* character (*$* character role will be soon clear after introducing new variables).

SuffixTree class is the one that has had more changes.
Added variables are the following:
-*phi* represents smoothing parameter φ;
-*maxEntropy*, at *i* position, contains max entropy for length *L=i+1*;
-*keepMaxInfo* is used when max entropy, for a given length, is needed. At position 1 it contains last *mpm* found (see *searchMax* method) and at position 0 it contains corresponding length;
-*nodesCountPerLength*, at *i* position, contains the number of visited nodes for length *L=i+1*;

Constructor sets count and entropy variables in every node calling respective method and correcting, in the end, root count but first it adds special character *$* (which does not belong to the alphabet) to the given string *S* of length *n*: this ensures that no suffix is a prefix of another, and that there will be *n* leaf nodes, one for each of the *n* suffixes of *S*. By doing this we are sure that a suffix tree exists for every *S*.

Before analyzing individual methods, a further addition should be noted: in *addPrefix* method, after creating a new edge, *endIndex* is decreased by 1 because of special character and the end node length is set.

Here is the list of all created methods and, from now on, for ease of explanation, when not explicitly said, "entropy" will refer to formula (1.1) instead of formula (0.2):

$$f_{L,\phi}(i) = \sum_{k=1}^{L} 4^k \phi^k c[i, i+k-1] \qquad (1.1)$$

`int setAllCount(Node v)`: called by constructor (giving *v*=root) allows to traverse the entire suffix tree setting every node count. If *v* is leaf its count is set to 1, otherwise its children counts are read iterating this method itself and eventually *v*'s count will be the sum of them.

`void` `setEntropy(`**`double`** `phiGiven)`: it assigns *phi* value and calls *setEntropyByNode* method. Every time this method is called, i.e. every time φ value is changed, all *maxEntropy* and *keepMaxInfo* informations are reset.

`void` `setEntropyByNode(Node v)`: its role is to set all *v*'s children entropy. *partial* are sums like $4\varphi^k + 4\varphi^{k+1} + 4\varphi^{k+2}$... here obtained by subtracting child's *partial* from father's *partial* and *entropy* variable is calculated by multiplying *partial* by father's entropy; at last child entropy is calculated adding *v*'s entropy to *entropy* variable. If current child, whose entropy has just been calculated, is not a leaf this method is iterated.

`double` `getMaxEntropy(`**`int`** `length)`: it returns the maximum complete entropy (formula (0.2)) for given length if it is already available, otherwise it is first calculated by calling *searchMax* method.

`void` `searchMax(`**`int`** `length, `**`int`** `index)`: it finds all maximum entropies for all lengths lower than *length*. At first a double supporting vector *a* is created and a pre-existing global variable is renamed: since *keepMaxInfo[1]* represents the last minimum potential maximum found (see *Chapter II* for more details), for ease of code understanding, here it is called *mpm.*
for cycle finds maximum entropy for every length *l* from *l=index* to needed length and the first time this cycle is run *index* is necessarily equal to 1. For every iteration *a[2]*, which contains the number of visited nodes for current length, is set to 1 in order not to miss root node.
All information needed to find maximum entropy for current length are put in *a* array by *searchMaxPerLength* method and, at this point, it is possible to save the number of visited nodes in *nodesCountPerLength* global variable. Subsequently real entropy of formula (0.2) is obtained and stored in *maxEntropy* respective position.
To proceed, it is necessary to check if maximum entropy was returned by a leaf node: if so, *a[1]* will be equal to *1* and *mpm* and *a[0]*, which contains maximum entropy found, must be reset because, at next step, a higher entropy is not sure anymore; otherwise *a[1]* will be equal to *0* and *mpm* can be increased as explained in *Chapter II*.
Eventually, out of *for* cycle, *keepMaxInfo* is updated in order not to start all over again if higher lengths will be necessary.

`double``[] searchMaxPerLength(Node v, `**`double`**`[] a, `**`double`** `compare, `**`int`** `length)`: it solves a subproblem of *searchMax* method.
Initially an *Edge* array containing all *v*'s lower edges is created and every children edge is considered. If current edge reaches length *length*, corresponding entropy can be easily obtained due *v* and current child information. In this case there are three possibilities for just found entropy:

1) it is lower than that previously found
2) it is equal to that previously found
3) it is higher than that previously found

Case 1) is not interesting because it does not involve any change since that solution will be discarded.
Case 2) it is not common for long sequences but it must be considered to be sure that maximum entropy is actually returned by a leaf node, in fact, when there are more nodes giving the same value, the program must know if all of these are terminal nodes because, if not, it is sure that at length l*ength+1* an higher entropy will be found and *searchMax* method does not needs to put *mpm* to zero when called.
Case 3) is very easy to understand: if an higher entropy is found, the current maximum entropy value stored in *a[0]* is updated and a "isLeaf()" test is done.
If current edge does not reaches length *length*, "mpm test" (see *Chapter II*) is done with *mpm=compare* and if it is successful this method is iterated and therefore *a[2]* is increased by one.

`double[] getNormalizedEntropy(int position, int lengthChosen)`: it returns a double array of length 2 which contains , for the given input, normalized entropy of formula (2.2) at position 0 and non normalized entropy of formula (0.2) at position 1.

`double getEntropy(int position, int lengthChosen)`: it returns non normalized entropy of formula (0.2) for the given input; *main* variable represents simplified entropy of formula (1.1) obtained by tree traversal.

`private double searchEntropy(Node v, int currentPosition, int leftLength)`: it starts by selecting the needed edge *e* of *v* and if it is not deep enough this method is iterated subtracting *e*'s length from *leftLength*, otherwise entropy is easily recovered.

`private double getSummation(int begin, int end, double param)`: it is simply the implementation of summations in (0.2).

`public static void main(String args[])`: its arguments, in order, are:

- *begin*, representing beginning position;
- *length*, representing chosen length of substrings;
- *phi*, representing φ smooting parameter;
- *gap*, representing the range beginning at position *begin* and ending at position *begin+gap* where substrings of length *length* are examined;
- *args[4]*, representing the sequence: it can be the name of the *.txt* file containing the sequence or, directly, it can be the sequence itself.

For every query it shows on the screen required time to reach the answer and it creates two *.txt* files: the first one contains the entropic normalized values and the other contains the number of visited nodes with granularity 5 for length L' where $1 \leq L' \leq length$.
If it is not the first query *args[4]* parameter is not required because suffix tree of given sequence is already available; in this case, if *phi* parameter has not the same value of previous query, *setEntropy* method is used to update entropy values of every node.
To stop its running, *q* or *Q* character must be entered.

For example, if we want to know entropic values of *sequence1* from position 0 to position 10 using *L=5* and φ*=8* and then we want entropic values again on *sequence1* but from position 2 to position 7 using *L=4* and φ*=3* following instructions needs to be inserted:

```
java -jar EntropicProfiler.jar 0 5 8 10 sequence1.txt
2 4 3 5
q
```

CHAPTER II

MAXIMUM ENTROPY

As it was previously remembered, the main EP function is defined by (0.2) and, to allow the comparison of different parameter combinations, it can be normalized by formula:

$$EP_{L,\phi}(i) = \frac{f_{L,\phi}(i) - m_{L,\phi}}{s_{L,\phi}} \qquad (2.1)$$

where

$$m_{L,\phi} = \frac{1}{n}\sum_{i=1}^{n} f_{L,\phi}(i) \qquad s_{L,\phi} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(f_{L,\phi}(i) - m_{L,\phi}\right)^2}$$

After performing some algebraic considerations (see formulas (2.2) and (2.3) in [3]) it can be stated that this kind of normalization is still too expensive because it requires $O(n^3)$ time and $O(n^2)$ space.
These problems leads us to normalize EP function in a different way by formula

$$EP_{L,\phi}(i) = \frac{f_{L,\phi}(i)}{\mathbf{max}_{0\le j<n}[f_{L,\phi}(j)]} \qquad (2.2)$$

where $\boldsymbol{max}_{0\le j<n}\,[f_{L,\Phi}(j)]$ returns the maximum value of $f_{L,\Phi}(j)$ over all words of size *L*. Note that normalized EP function assumes values between 0 and 1.

To find maximum entropy for every length it is possible to avoid comparing each word of length *L*: as described in [3], minimum potential maximum (mpm) and maximum potential maximum (MPM) concepts are needed and, from now on, for ease of explanation, we consider simplified entropy (1.1) instead of complete entropy. Nevertheless, unlike [3], MPM will be calculated node by node and not length by length.

Definition 1:

$$\mathrm{MPM_L(v)} = \mathrm{entropy(v)} + (\mathrm{count(v)}\text{-}1)\cdot\sum_{k=h(v)+1}^{L}(4\varphi)^k \qquad (2.3)$$

where *h(v)* is the path length from root to *v* node and *v* is a node such that h(v)<L.
It defines an upper bound to the maximum entropy obtainable for the path starting from the root and passing through *v*.

Definition 2:

$$\mathrm{mpm}_L = \begin{cases} 0 & \text{if } L=1 \text{ or } MPM_{L-1} \text{ was returned by a leaf node} \\ \mathrm{MAX}_{\mathrm{MPM}_{L-1}} + (4\varphi)^L & \text{otherwise} \end{cases} \quad (2.4)$$

where $MAX_{MPM_{L-1}}$ is the maximum among all MPMs found at previous step.
It defines a lower bound to the maximum entropy for L.

To find maximum entropy for length L' we must traverse the tree for every L where $1 \leq L \leq L'$ through all promising nodes, which have MPM grater than mpm. By doing this, when a non promising node is reached, its path will not be followed pruning the search space.
Every time L is increased, the suffix tree traversal starts from the beginning and mpm_L is set to the maximum MPM_{L-1} obtained from nodes where the traversal is arrived plus $(4\varphi)^L$ by definition. However, if the node which returned MPM_{L-1} is a leaf it is not more guaranteed that, at next step, mpm_L corresponds to the minimum possible entropy therefore it is reset.
When arrived to the end of every promising path, the maximum entropy will be chosen to calculate next mpm or, eventually, to return asked value.

Before proceed, it will be useful understand with an example how entropy is recovered node by node and a simple implicit ($ character omitted) suffix tree having $\varphi=0.25$ is considered at *Figure 2.1*.

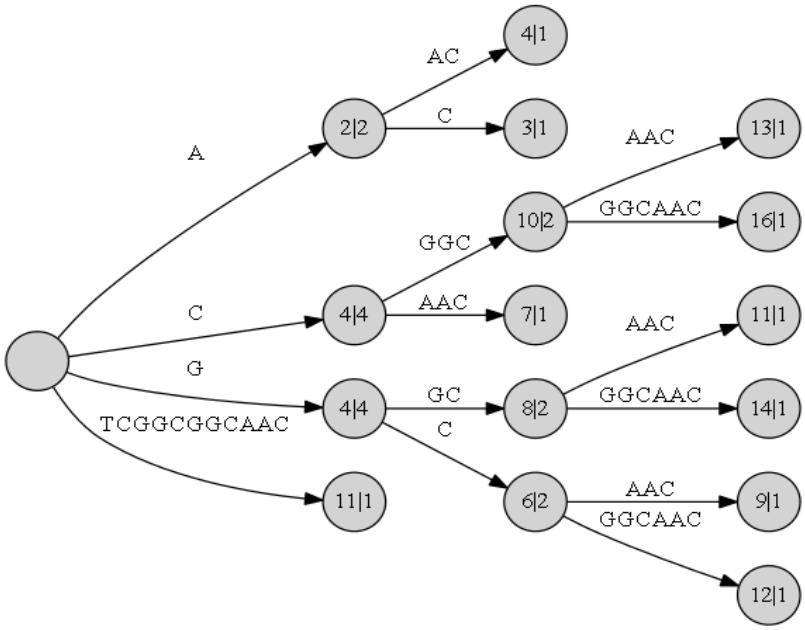

Figure 2.1: implicit suffix tree having $\varphi=0.25$ *for string* TCGGCGGCAAC.
Nodes are labeled with the corresponding values of entropy|count

To recover the entropy of a node *v*, the algorithm, as it is clear in *searchEntropy* method, does not need to read *v* and it can stop at its parent.
For example, to know the entropy of *TCGGCGGCAAC* string, it will be enough to stay on root node and add to its entropy, which is 0 by default, what is missing, namely the length of edge multiplied by child count: *0 + 11*1 = 11*. Instead, if string is *CGGC*, the algorithm will stop at *C* node, having entropy equal to 4, and calculations are *4 + 3*2 = 10*.

EXPERIMENTAL RESULTS

This approach to search maximum entropy is $O(n^2)$ in time and it is more efficient than searching through all possible nodes; to prove this some chart is now reported.

Returning to *Figure 2.1* example, at *Figure 2.2* are shown the results obtained with this strategy using different φ values.

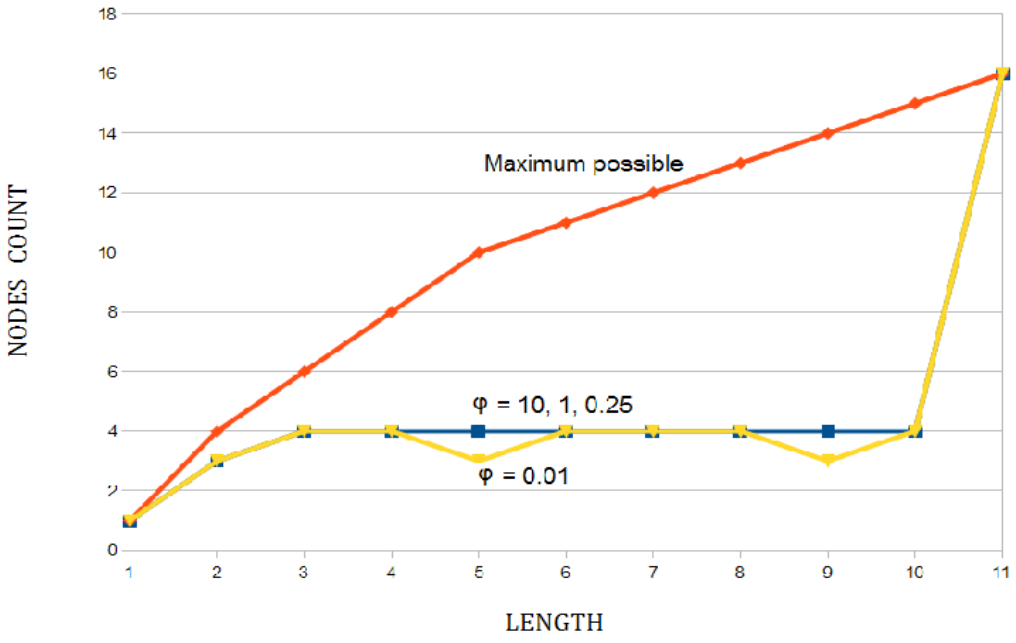

Figure 2.2: number of visited nodes for Figure 2.1 *with different* φ *values*

As you can see, the maximum number of visitable nodes for L=11 is 16 though the suffix tree has 17 nodes overall. This happens because, as previously explained, *TCGGCGGCAAC* node does not need to be read to know its entropy.
Note that at length 10 the entropy is given by *CGGCGGCAC* node, which is a leaf node, therefore, at following length, mpm_{11} is zero and every possible node will be visited until current length will not be grater or equal to eleven (in this example it happens only with *TCGGCGGCAAC* node which is not read).
The program was also tested on different random balanced strings having length 3000 and the average of obtained data are illustrated at *Figure 2.3*.

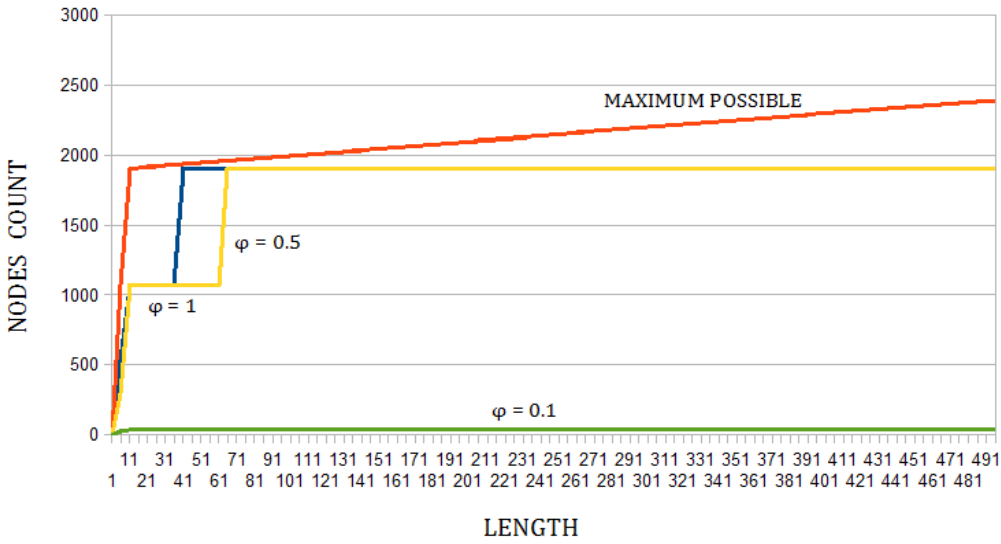

Figure 2.3: average of obtained data from random balanced strings of length 3000 with different φ values. Used granularity: 5

When strings are still random but unbalanced, i.e. when *C* and *G* characters appear twice the times of *A* and *T*, obtained average is not significantly different from the previous one.

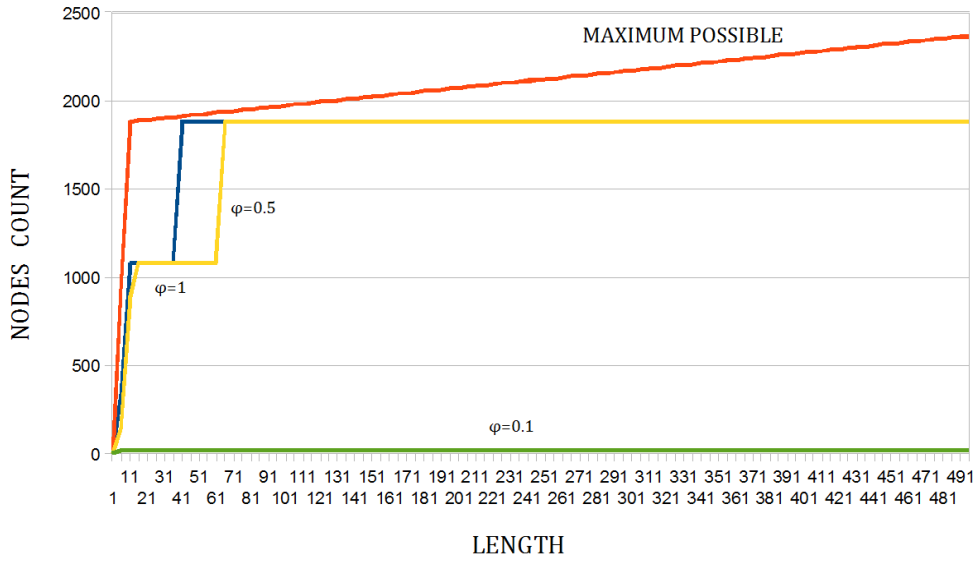

Figure 2.4: average of obtained data from random unbalanced strings of length 3000 with different φ values: C *and* G *characters appears with probability 1/3 Used granularity: 5*

It is clear that a CG-rich sequence does not make significant difference because both charts are practically indistinguishable and they are plotted with lines that often have derivative equal to zero. Each time lines become horizontal, it means that the path followed by the program is the same for every length in that range and it ends on a "long" edge. In fact, with a random sequence, increasing searched length, it is easier to find "long" edges representing an equally "long" random subsequence.
Decreasing φ, the space search is reduced and we can note that if this smoothing parameter is sufficiently low, the number of visited nodes per length is heavily reduced. This happens because of (2.3) formula: every time L is increased, MPM increases very slightly and most of the times it will be less than mpm. Unfortunately this achievement is not reached without a price, in fact a small value of parameter φ causes a not understandable entropic chart.
Anyhow, whatever the φ value, this strategy will never be more expensive than a complete research through the sub-suffix tree.

CHAPTER III

ENTROPIC PROFILER

In [2] the authors proposed a C++ implementation of the Entropic Profiler, also available at [6]: it is based on a truncate suffix tree which has a node per character and side links to connect nodes of the same length (see *Figure 3.1*).

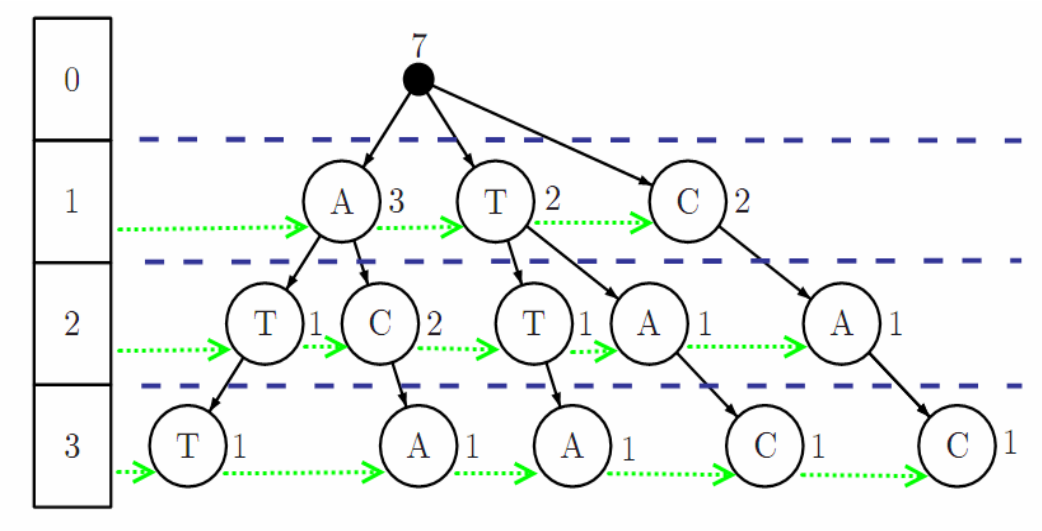

Figure 3.1: Representation of the data structure created by algorithm proposed in [2] for ATTACAC *string*

Its properties were analyzed in [3] and they are here listed:

- the number of nodes is $O(n^2)$ where *n* is the string length
- the height is equal to the length of the longest string, that is the length of the whole sequence
- word matching for a *L* long pattern takes O(L) time
- constructing the tree takes $O(m^2)$ time

One of this program limitation is that it requires a lot of space because of nodes: character aggregations are not allowed and this choice statistically causes many single child per node. This is evident in *Figure 3.2*: changing nodes creation policy it is possible to drastically reduce required resources.
Another drawback is that *L* input, which represents the length of the substrings that are going to be explored, can not be greater than 10, in fact the suffix tree created by this program can be deep at most 10. Besides φ parameter is freezed to 10 and the search window can not be greater than 1000.

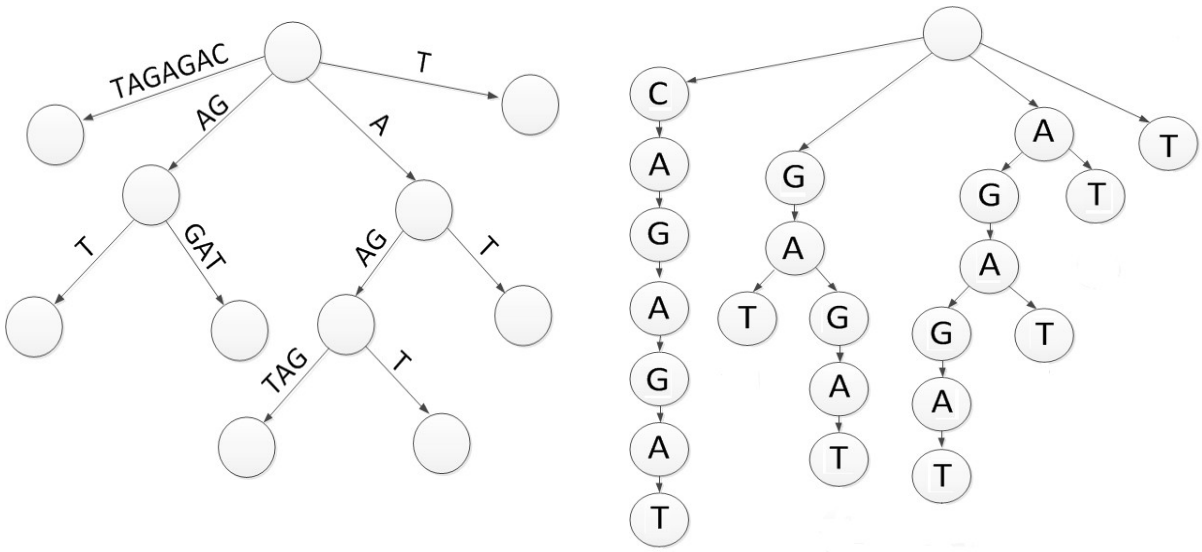

Figure 3.2: Representation of suffix tree for CAGAGAT *string. On the left the corresponding structure using program exposed in* Chapter I*; on the right the equivalent suffix tree with program exposed in [2]*

Some comparing example between program introduced in *Chapter I* and that available in [6] is now analyzed and, from now on, they will be called *EP1* and *EP2* respectively. Every chart refers to a 1000-long random string in which a k-long motif is inserted more times (at least two).

Example I: k=4

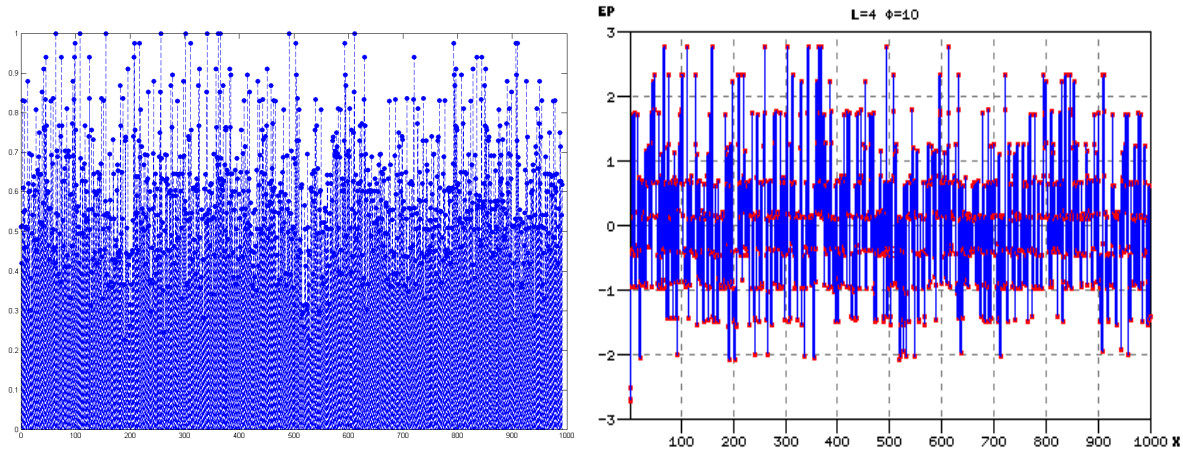

Figure 3.3a: entropic results for a k=4-long motif inserted 9 times. On the left EP1 *output with* L=4 *and* φ=2*; on the right* EP2 *output with* L=4 *and* φ=10

It is clear that k=4 is too low if the entire sequence has length 1000. Both charts are therefore very "noisy" and the functions do not improve even if parameters are changed. This happens because there are 4-long random motifs that occur more times, in fact *CCCG* string was deliberately inserted nine times but *TACA* string appears ten times, *TCGT* and *TTCG* strings appear nine times too, many others strings appear eight times and so on.

Example II: k=6

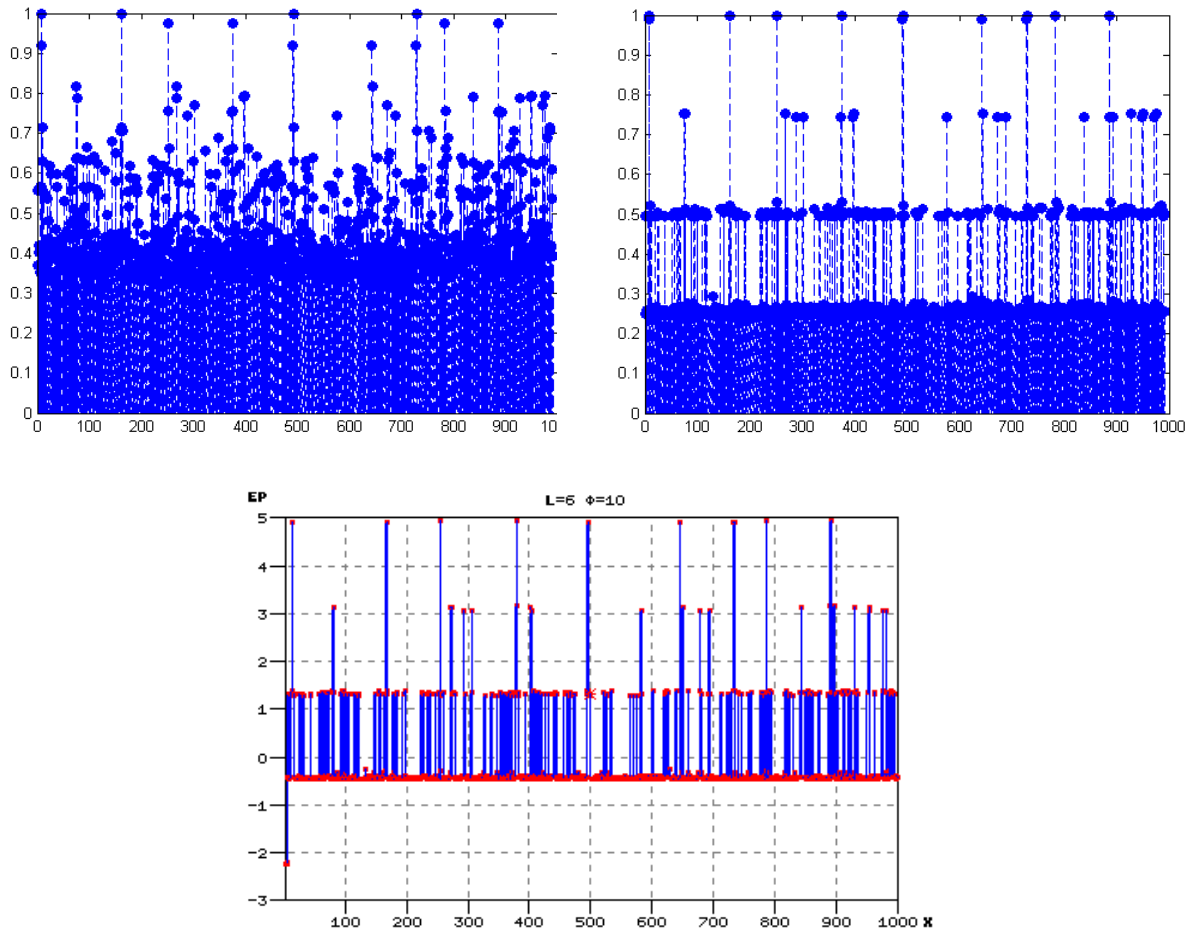

Figure 3.3b: entropic results for a k=6-long motif inserted 2 times.
On the top EP1 *output with* L=6 *and* φ=1 at left and L=6 *and* φ=7 at right*;*
on the bottom EP2 *output with* L=6 *and* φ=10

A 6-long motif still remains too short (into a 1000-long sequence) if it is deliberately inserted only two times because many 6-long random strings appear four, three and two times and this makes it impossible to locate wanted motif. Particularly, in *EP1*, the first ones have entropy equal to 1, the second ones have entropy approximately equal to 0.75, the third ones have entropy approximately equal to 0.5.
Note that a small value of parameter *φ* can make the chart "noisy".

Example III: k=7

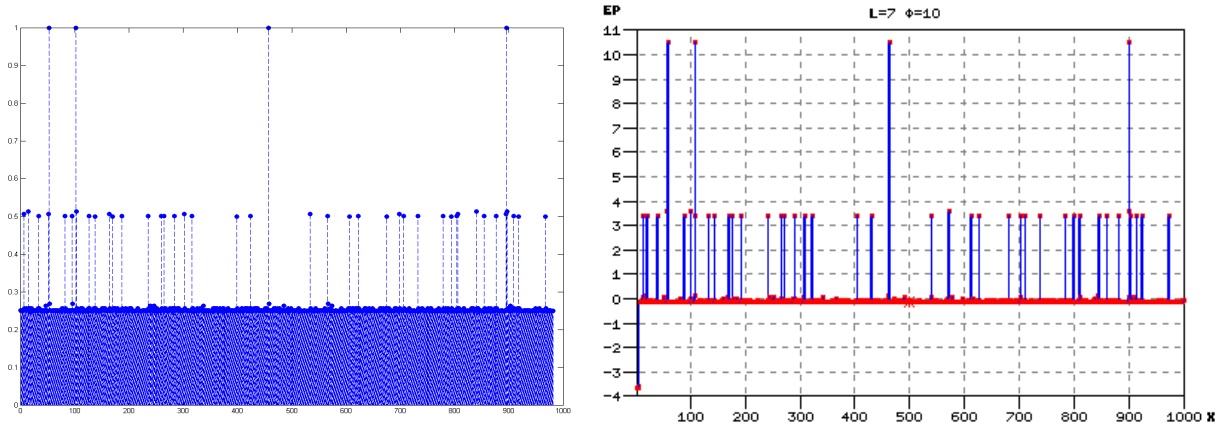

Figure 3.3c: entropic results for a k=7-long motif inserted 4 times.
On the left EP1 *output with* L=7 *and* φ=10*;*
on the right EP2 *output with* L=7 *and* φ=10

Further increasing the motif length, better results are obtained: all four inserted motif are located by both EPs and other 7-long random motifs, each of them repeated twice, are detected with a lower entropy value.
If *L* had been 8 instead of 7, with *φ*=10 the chart would have had three peaks corresponding to three out of four inserted motifs; with *φ*=1 the chart would have been negatively distorted but still attributable to that illustrated.

Example IV: k=10

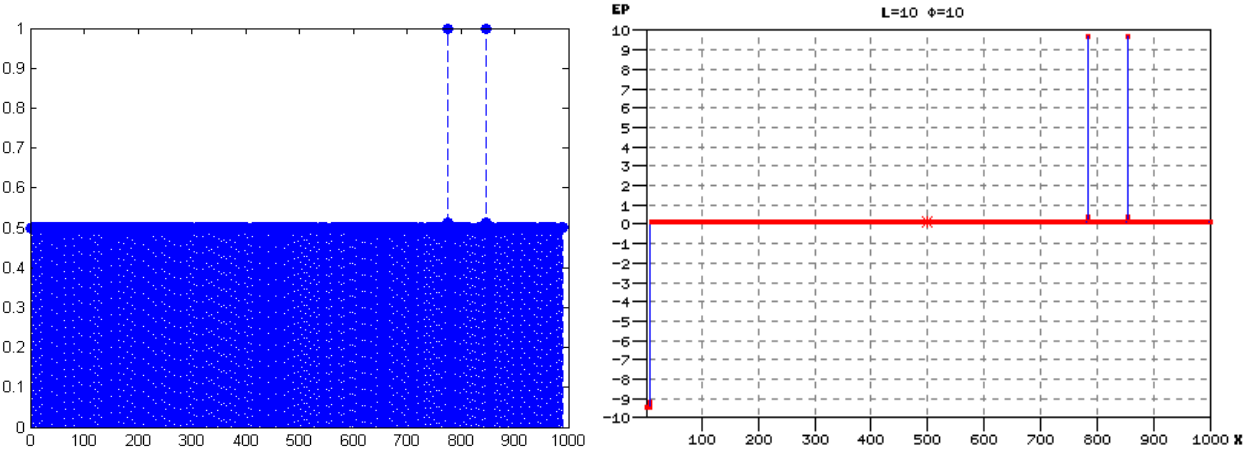

Figure 3.3d: entropic results for a k=10-long motif inserted 2 times.
On the left EP1 *output with* L=10 *and* φ=10*;*
on the right EP2 *output with* L=10 *and* φ=10

With *k*=10 is reasonably sure that there are no other 10-long string repeated, in fact both inserted motifs are found by EPs. Charts would been "clean" even with *L*=9 mantaining *φ* to 10 but setting *φ* to 1 they would been more "noisy".

Example V: k=50

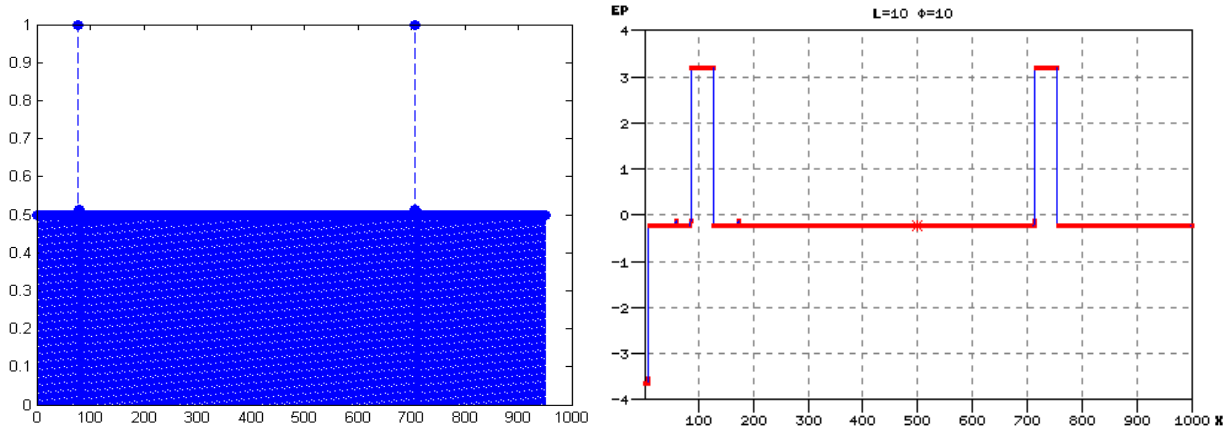

Figure 3.3e: entropic results for a k=50-long motif inserted 2 times. On the left EP1 *output with* L=50 *and* φ=10*; on the right* EP2 *output with* L=10 *and* φ=10

When *k* is greater than 10, *EP2* can not recognize the single 50-long motif and it finds only 10-long motifs in a row: this is the reason why *EP1* has only two peaks and *EP2* has two series of them. Since a k=50-long motif is pretty remarkable into a 1000-long sequence, charts are not compromised even when *L*=40 and *φ*=0.5.

Example VI: k=10; GC-rich random sequence

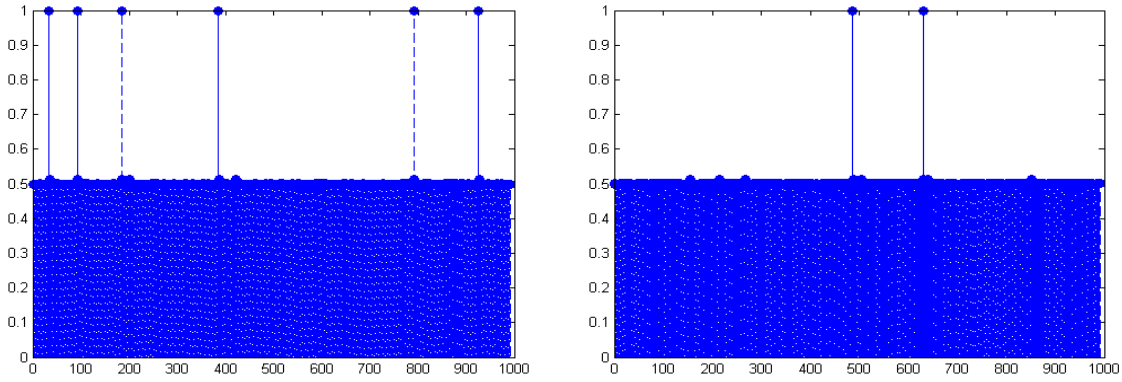

Figure 3.4: entropic results for two k=10-long motifs inserted 2 times into a GC-rich random sequence of length 1000. On the left, random sequence has more 10-substrings repeated; on the right random sequence has only one 10-substring repeated twice as expected

With this ending example supplementing the issue, a GC-rich random sequence of length 1000 is considered in order to show that outputs are not much different from a balanced random sequence and, to prove this, only *EP1* is needed.
In *Example IV* was shown that, when *k=10*, the entropic profiler can recognize inserted motif without reporting any other repeated substring. When involved string is not

balanced anymore, this is not always true, in fact this *k* value now serves as the border between expected and unexpected charts: at *Figure 3.4* are reported obtained data by inserting two 10-long motifs into two different GC-rich sequences and we can note that, depending on the case, the output can be "clean", with only two expected peaks, or not.

PERFORMANCE COMPARISON

Through all these examples it was shown that *EP1* and *EP2* have quite similar outputs, nevertheless *EP2* has all limitations exposed at the beginning of this chapter and this makes *EP1* more complete, in fact the entropic profiler exposed in *Chapter I* has no limitations in *L*, *φ* and *gap* parameters. Besides the internal structure built by *EP1* makes it more efficient as previously explained.
To concretely prove the greater efficiency of *EP1* on *EP2*, some time test on these EPs was done by a domestic machine running "Windows 7 Ultimate x64 Service Pack 1" and having CPU "Intel(R) Core(TM)2 CPU 3600 @ 1.86GHz". Four balanced random sequences of length 1000 have been used for this purpose and obtained results for every output are here reported:

EP2 L=10 / φ=10 / gap=990	Test 1 (in milliseconds)	Test 2 (in milliseconds)	Test 3 (in milliseconds)	Average (in milliseconds)
Sequence 1	359	359	434	384
Sequence 2	343	327	359	343
Sequence 3	343	328	343	338
Sequence 4	328	296	328	317
> Global average for *EP2*: 346 milliseconds <				

EP1 L=10 / φ=10 / gap=990	Test 1 (in milliseconds)	Test 2 (in milliseconds)	Test 3 (in milliseconds)	Average (in milliseconds)
Sequence 1	65	64	65	65
Sequence 2	64	65	68	66
Sequence 3	65	68	68	67
Sequence 4	64	64	63	64
> Global average for *EP1*: 66 milliseconds <				

Then, empirical data shows that *EP1* uses only 19% of time resources required by *EP2*. Nevertheless *EP1* advantages are not finished yet. Unlike *EP2*, that for every query it builds an unnecessary new suffix tree even if the sequence remains unchanged, *EP1* allows to reuse previously built suffix tree in order to optimize time resources. This entails a response time reduced by a factor 3 as shown by next table:

EP1 Second Query L=10 / φ=10 / gap=990	Test 1 (in milliseconds)	Test 2 (in milliseconds)	Test 3 (in milliseconds)	Average (in milliseconds)
Sequence 1	21	20	20	20
Sequence 2	20	26	20	22
Sequence 3	21	23	22	22
Sequence 4	22	20	20	21
> Global average for *EP1 Second Query*: 21 milliseconds <				

Finally, another last possibility is remained to be analyzed: when it is not the first query (i.e. when suffix tree is already built) and a new φ parameter value is given. In this case all nodes needs to be traversed in order to update their entropy and this operation requires a little extra time but, overall, the query can be solved in half time compared to the first case:

EP1 Third Query L=10 / φ=8 / gap=990	Test 1 (in milliseconds)	Test 2 (in milliseconds)	Test 3 (in milliseconds)	Average (in milliseconds)
Sequence 1	33	33	32	33
Sequence 2	33	33	33	33
Sequence 3	31	30	32	31
Sequence 4	33	33	31	32
> Global average for *EP1 Third Query*: 32 milliseconds <				

Bibliography

[1] Vinga S, Almeida JS: *Local Rényi entropic profiles of DNA sequences*. BMC Bioinformatics; 2007.

[2] Fernandes F, Freitas AT, Almeida JS, Vinga S: *Entropic Profiler - detection of conservation in genomes using information theory*. BMC Research Notes; 2009.

[3] Antonello M, Comin M: *Entropic Profiler of DNA Sequences Using Suffix Tree*

[4] Gusfield D: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* (page 94). Cambridge University Press; 1997.

[5] source code at http://code.google.com/p/pwaalproject/source/browse/trunk/java/?r=2

[6] http://kdbio.inesc-id.pt/software/ep/

Other useful references:

Gusfield D: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press; 1997.

Jernigan RW, Baran RH: Pervasive properties of the genomic signature. BMC Genomics 2002, 3:23.

Karlin S, Burge C: *Dinucleotide relative abundance extremes: a genomic signature*. Trends Genet 1995, 11:283-290.