Tesi di Laurea in
Ingegneria Elettronica

Implementation of Entropic Profiler for DNA Sequences by using Truncated Suffix Trees

Laureando:
Le Ngoc Minh

Relatore:
Matteo Comin

Anno Accademico 2013 - 2014

Contenuto

Introduzione

Lo studio di DNA (Deoxyribonucleic acid) diventa sempre più importante nella medicina, nel campo biomedico, e nella bioingegneria. La necessità di studiare, analizzare, classificare e memorizzare queste sequenze codificata di DNA in modo più efficace porta a sviluppare tecniche e concetto come entropic profiler.
Che cosa è Entropic Profiler nello studio di DNA? Entropic Profiler è un concetto sviluppato per studiare, analizzare sequenze di DNA (ossia sequenze di genoma). Data una sequenza di DNA, Entropic Profiler analizza ed estrae parametri statisticamente importanti per lo studio di DNA stessa. Ad Entropic Profiler è associata una funzione chiamato EP:

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

Dove n è la lunghezza della sequenza, L è la lunghezza di risoluzione, φ è chiamato parametro di smoothing e c[i-k+1, i] è il numero di volte la sottostringa di lunghezza k che termina in posizione i appare in intera sequenza.
Per l'analisi e una descrizione più dettagliata su Entropic Profiler è rimandata all'articolo di
Fernandes F, Freitas AT, Almeida JS, Vinga S: *Entropic Profiler - detection of conservation in genomes using information theory*. BMC Research Notes; 2009 [2].
Vinga S, Almeida JS: *Local Rényi entropic profiles of DNA sequences*. BMC Bioinformatics; 2007 [1].

Nel [2] gli autori hanno presentato un algoritmo in C per Entropic Profiler. Tale algoritmo usa un truncated suffix trie che in ciascun suo nodo viene memorizzato un carattere della sequenza, e usa i così detti side link per collegare tutti nodi che stanno alla stessa profondità.

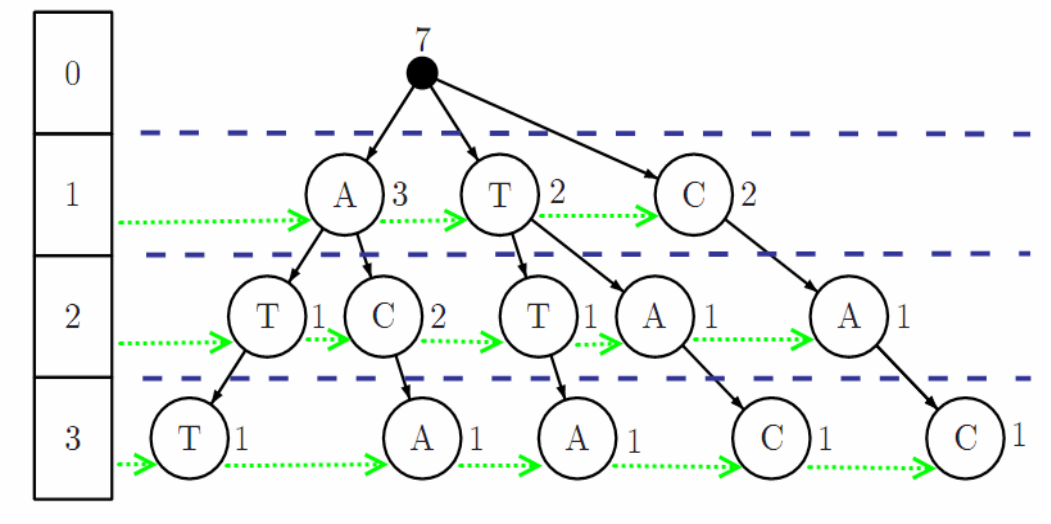


Figura1 Rappresentazione della struttura dati creata dall'algoritmo proposto in[2] per la stringa ATTACAC

Questo algoritmo analizzato nella lavoro di Antonello Morris e Matteo Comin [3] risulta non molto efficiente, sia in termine di spazio che di tempo. Infatti per una sequenza di input di lunghezza n:

- Il numero di nodi richiesti è $O(n^2)$
- Il tempo richiesto per costruire l'intero albero è $O(n^2)$.

Così Antonello Morris e Matteo Comin nel [3] hanno proposto un algoritmo più efficienti, usando una struttura di suffix tree differente e modificando anche il modo di calcolare il valore EP. Nel capitolo III verrà data una breve descrizione.
In questo lavoro verrà presentata una implementazione in Java del algoritmo proposto da
Antonello Morris, Matteo Comin: Entropic Profiler of DNA Sequences Using Suffix Tree [3].
Questa implementazione è realizzata partendo da un altra implementazione in Java, basato sempre sul algoritmo proposto da Antonello Morris e Matteo Comin, e si trova in questo lavoro di
Stefano Mazzocca, Matteo Comin: Implementation of Entropic Profiler for DNA Sequences by using Suffix Trees [4].
La differenza tra due implementazioni è nella realizzazione della struttura dati Suffix Tree. In questo lavoro viene usato il così detto k-factor tree oppure truncated suffix tree, ossia l'altezza/profondità dell'albero viene limitato da un fattore k.
Tale fattore k verrà chiesto all'utente come l'argomento in input del programma insieme ad altri parametri (quali posizione d'inizio, la lunghezza del sottostringa, phi, il gap tra posizione d'inizio e posizione fine e la sequenza che si vuole analizzare).

L’utilizzo di questa struttura truncated suffix tree è giustificato dal fatto che il suffix tree realizzato per l’intera sequenza di DNA richiede molto spazio di memoria (una intera sequenza di DNA può contenere miliardi di basi). Inoltre non sempre si ha la necessità di analizzare l’intera sequenza. Infatti più spesso si ha la necessità di analizzare solo una porzione di essa.

Capitolo I

Albero dei suffissi troncato

Come detto in precedenza l'albero dei suffissi troncato (truncated suffix tree o k-factor tree) è un albero la cui altezza viene limitata da un fattore k. Tale fattore k ci aiuterà a risparmiare lo spazio di memoria necessaria per contenere tutti i nodi dell'albero. Infatti a causa del troncamento non tutti i nodi dell'albero saranno creati e memorizzati in memoria.
Durante la costruzione dell'albero dei suffissi troncato con fattore di troncamento pari a k l'altezza delle foglie verrà controllata affinché non superi il valore di k. Se tutte le foglie di un ramo raggiungono l'altezza k, allora tale ramo non crescerà più.
Per capire come è fatto un albero dei suffissi troncato viene di seguito mostrato graficamente un albero dei suffissi prima e dopo il troncamento.

Qui sotto è l'immagine dell'albero dei suffissi della sequenza TCGGCGGCAAC prima del troncamento.

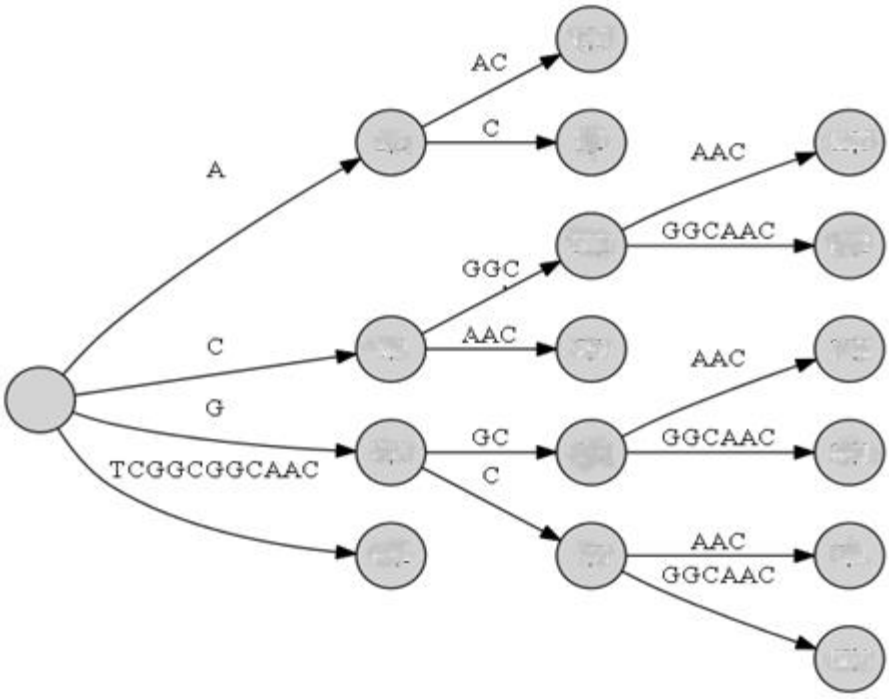


Figura 1.1

Qui sotto è l'immagine dell'albero dei suffissi dopo il troncamento, con il fattore di troncamento k = 2.

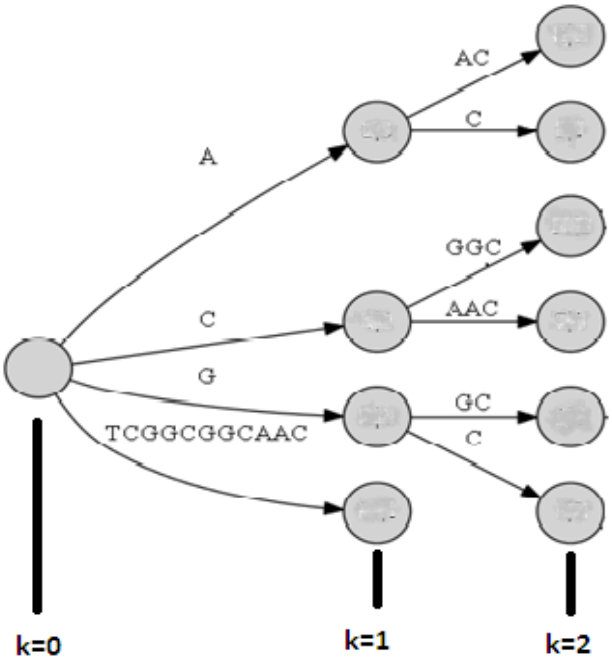


Figura 1.2

Come si nota da questa figura [*fig. 1.2*] il numero dei nodi è diminuito.

Capitolo II

Codice Java dell'implementazione

Sotto è riportato il codice Java dell'implementazione, dove le modifiche al codice originale nel lavoro di Stefano Mazzocca e Matteo Comin [4] vengono commentate con parola "MODIFICA".
Le modifiche coinvolgono la classe Node, la classe Edge, la classe SuffixTree e il metodo main.
Per realizzare questa struttura di suffix tree troncata alla classe Node viene aggiunta la variabile intera depth, in cui sarà memorizzata la profondità di tale nodo nel suffix tree, e due metodi, rispettivamente per settare la variabile depth (setDepth(i)) e per ottenere il valore memorizzato in depth (getDepth()). Viene utilizzato anche un parametro intero k Factor per il fattore di troncamento. L'utente potrà scegliere a quale profondità troncare il suffix tree fornendo il valore scelto per questo fattore al momento di esecuzione del programma EntropicProfiler.
Durante la fase di costruzione del suffix tree, la parte principale viene eseguita dal metodo addPrefix. addPrefix creerà nuovi archi e nodi da aggiungere al suffix tree, quando certe condizioni sono verificate. Ogni volta si aggiunge un nuovo arco viene creato ed aggiunto anche un nuovo nodo e questo nodo si troverà ad una profondità maggiore di uno rispetto al suo genitore. Per questo ogni volta si deve creare ed aggiungere un nuovo arco e il suo nodo finale, prima viene controllata se la profondità del nodo genitore è minore di k. Se tale condizione è soddisfatta l'operazione sarà eseguita, altrimenti si effettua addPrefix su prossimo suffisso minore e si ripete fino a che non si verificano le condizioni per uscire dal ciclo while del metodo addPrefix.

```
import java.util.Map;
import java.util.HashMap;
import java.util.Collection;
import java.util.Scanner;
import java.lang.Math;
import java.io.File;
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;
import java.util.Scanner;

 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;
        private int depth=0; //MODIFICA
        private intintLeaf=0; //MODIFICA

        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(intcharIndex, Edge edge) {
edges.put(charAt(charIndex), edge);
        }

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

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

        public Node getSuffixNode() {
            return suffixNode;
        }

        public intgetName(){
            return name;
        }
```

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

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

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

        public intgetCount(){return count;}
        public void setCount(int i){count=i;}
        public intgetLength(){return length;}
        public void setLength(int i){length=i;}
        public booleanisLeaf(){return (!(name==0))&&(suffixNode==null);}
        public void setEntropy(double i){entropy=i;}
        public double getEntropy(){return entropy;}
        public int getDepth(){return depth;} //MODIFCA
        public void setDepth(int i){depth=i;} //MODIFCA
        public booleanisIntLeaf(){return intLeaf==1;} //MODIFICA
        public void setIntLeaf(int t){intLeaf=t;} //MODICA

       }



 class Edge{
        private intbeginIndex;     // can't be changed
        private intendIndex;
        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(intbeginIndex, intendIndex, Node startNode) {
this.beginIndex = beginIndex;
this.endIndex = endIndex;
this.startNode = startNode;
this.endNode = new Node(startNode, null);
this.endNode.setDepth(startNode.getDepth() + 1); //MODIFICA
        }

        public Node splitEdge(Suffix suffix) {
            remove();
            Edge newEdge = new Edge(beginIndex, beginIndex + suffix.getSpan(),
suffix.getOriginNode());
newEdge.insert();
newEdge.endNode.setSuffixNode(suffix.getOriginNode());

//setto la length del nodo che si trova nel punto di spezzamento (il vecchio ramo
viene spezzato in due)
newEdge.getEndNode().setLength(newEdge.getEndIndex()-
newEdge.getBeginIndex()+1+newEdge.getStartNode().getLength());
beginIndex += suffix.getSpan() + 1;
startNode = newEdge.getEndNode();
            insert();
endNode.setDepth(startNode.getDepth() + 1); //MODIFICA
            return newEdge.getEndNode();
        }


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

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

        public intgetSpan() {
            return endIndex - beginIndex;
        }
```

```
        public intgetBeginIndex() {
            return beginIndex;
        }

        public intgetEndIndex() {
            return endIndex;
        }

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

        public Node getStartNode() {
            return startNode;
        }

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

        public Node getEndNode() {
            return endNode;
        }

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

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

    }



 class Suffix {
        private Node originNode;
        private intbeginIndex;
        private intendIndex;

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

        public booleanisExplicit() {
            return beginIndex>endIndex;
        }

        public booleanisImplicit() {
            return endIndex>= beginIndex;
        }

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

intedgeSpan = 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 intgetSpan() {
            return (endIndex - beginIndex);
        }
```

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

        public intgetBeginIndex() {
            return beginIndex;
        }

        public void incBeginIndex() {
beginIndex++;
        }

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

        public intgetEndIndex() {
            return endIndex;
        }

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



 public class SuffixTree{

    //Suffix Tree
    private String text;
private Node root;
private intnodesCount;  //variabile pre-esistente, conta il numero totale di nodi
che ha l'albero
    private double phi;
    private double[] maxEntropy; //alla posizione 0 c'e' il max per la lunghezza 1
    //in [0] ha la lunghezza dove il calcolo del max si è fermato; in [1] ha l'mpm
relativo a tale lunghezza...
    private double[] keepMaxInfo; //...per tenere la situazione sui massimi
calcolati non fino alla fine
    private int[] nodesCountPerLength; //alla posizione i c'è il numero di nodi
visitati per la lunghezza L=i+1
private int kFactor; //MODIFICA

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

this.text = text+"$";
        root = new Node(this, null);
root.setDepth(0); //MODIFICA

        Suffix active = new Suffix(root, 0, -1);
for (int i = 0; i <= text.length(); i++) { //ogni ciclo corrisponde a un sotto-
suffix tree
addPrefix(active, i, kFactor); //MODIFICA
}
//attraversa e assegna i count a tutti i nodi, correggendo anche le Length delle
foglie:
setAllCount(root);
        //la radicedeveavere count=0
root.setCount(0);
setEntropy(phi);
    }

    private void addPrefix(Suffix active, intendIndex, int kFactor) { //MODIFICA
Node lastParentNode = null;
        Node parentNode;
int k = kFactor;

        while (true) {  //questo ciclo viene iterato se ci sono più etichette uguali
da seminare lungo l'albero
Edge edge;
```

```
parentNode = active.getOriginNode();

            // Step 1 is to try and find a matching edge for the given node.
            // If a matching edge exists, we are done adding edges, so we break out
of this big loop.
            if (active.isExplicit()) {
                edge = active.getOriginNode().findEdge(text.charAt(endIndex));
                if (edge != null) {
                    break;}
            }

            else {
                //implicit node, a little more complicated
                edge =
active.getOriginNode().findEdge(text.charAt(active.getBeginIndex()));
int span = active.getSpan();

                if (text.charAt(edge.getBeginIndex() + span + 1) ==
text.charAt(endIndex)){
   //MODIFICA
   if ((edge.getEndNode().getDepth() == k)&&(edge.getEndNode().getLength()
<edge.getStartNode().getLength() + span +1)&&(!edge.getEndNode().isIntLeaf())){

          edge.getEndNode().setIntLeaf(1);
          edge.getEndNode().setCount(1);

   }
                    break;
                }
                //MODIFICA
                if (edge.getEndNode().getDepth() == k){
   edge.getEndNode().setIntLeaf(1);
   edge.getEndNode().setCount(edge.getEndNode().getCount() + 1);
   break;
                }
                else{
parentNode = edge.splitEdge(active);
                }
            }


            // We didn't find a matching edge, so we create a new one, add it to the
tree at the parent node position,
            // and insert it into the hash table.  When we create a new node, it
also means we need to create
            // a suffix link to the new node from the last node we visited.

            Edge newEdge = new Edge(endIndex, text.length() - 1, parentNode);
newEdge.insert();
            //setto la Length
            newEdge.getEndNode().setLength(text.substring(newEdge.getBeginIndex(),
newEdge.getEndIndex()).length()+parentNode.getLength());
newEdge.setEndIndex(newEdge.getEndIndex()-1);//così l'indice finale del ramo non
cade sull'ultimo carattere della stringa ($) ma sul...
                                                        //...penultimo carattere
(cioè l'ultimo carattere della stringa prima dell'insermento di $)


updateSuffixNode(lastParentNode, parentNode);
lastParentNode = parentNode;

// This final step is where we move to the next smaller suffix
            if (active.getOriginNode() == root)
active.incBeginIndex();
            else
active.changeOriginNode();
active.canonize();
        }//end while
updateSuffixNode(lastParentNode, parentNode);
active.incEndIndex();   //Now the endpoint is the next active point
active.canonize();
    }

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

```
    }

    public intgetNewNodeNumber() {
        return nodesCount++;
    }

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

    public intindexOf(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 intsetAllCount(Node v){

//foglia
if(v.isLeaf()){
   //MODIFICA:foglia interna
if(v.isIntLeaf()){
   return (v.getCount());
            }
            else{
   v.setCount(1);
                 return 1;
            }

        }

        //nodointerno
        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 voidsetEntropy(double phiGiven){ //dato un nodo setta l'entropia dei
figli
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){        //metodod'appoggio
Edge e[]= v.getEdges().toArray(new Edge[0]);
       for(int i=0;i<e.length;i++){

//partial sono le somme tipo (4 phi)^k + (4 phi)^(k+1) + ..., ottenute facendo
PartialEndNode - PartialStartNode
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();

//ora devo calcolare il valore dell'entropia servendomi di quella del nodo padre:
           e[i].getEndNode().setEntropy( e[i].getStartNode().getEntropy() + entropy
);


          //se il nodo appena settato non e' una foglia ripeti la procedura per
tutti i suoi figli
if (!e[i].getEndNode().isLeaf()) { //se il nodo in questione non e' una foglia c'e'
altro lavoro da fare
setEntropyByNode(e[i].getEndNode());
                     /*cosi' facendo viene settata anche l'entropia dei nodi che
hanno la sola etichetta $
                      * ma siccome non dovrebbe essere mai cercata non e' un
problema e si evitano inutili
                      * confronti del tipo "se v e' il nodo speciale non fare
nulla"; la funzione entropia,
                      * siccome i nodi col solo $ sono relativamente pochi, non
dovrebbe essere troppo onerosa
*/
           }
       }
   }


   public double getMaxEntropy(int length){
if(maxEntropy[length-1]==0) searchMax(length, (int)keepMaxInfo[0]+1); //se
l'entropia massima per quella lunghezza non e' stata mai cercata la cerco
return maxEntropy[length-1];
   }

   private void searchMax(int length, int index){
       double[] a=new double[3];
       double mpm=keepMaxInfo[1];     //per facilità di
letturarinominokeepMaxInfo[1]
            for(int l=index; l<=length; l++){

//a: array che contiene in a[0] il max attuale, in a[1] c'è 1 se sono finito in una
foglia (altrimenti c'è 0), e in a[2] il numero di nodi visitati
               a[2]=1; //controllo per ogni l il numero di nodi visitati quindi al
metodo deve essere passato 1 in a[2] (conteggio della radice)
a=searchMaxPerLength(root, a, mpm, l);

//System.out.println("-----per L="+l+" sono stati visitati "+visitedNodes+" nodi");
nodesCountPerLength[l-1]=(int)a[2];

               //ora memorizzo in maxEntropy[length-1] l'effettiva entropia massima
(calcolata con la formula completa)
               double numerator=1+(a[0]/(text.length()-1));  //-1 per via del
carattere $
               double denominator=getSummation(0, l, phi);
maxEntropy[l-1]= numerator/denominator;

if(a[1]==0) mpm=a[0]+Math.pow(4*phi, l+1); //se il massimo non è caduto in una
foglia incremento l'mpm (mpm=max(L-1) + (4phi)^k)
```

```
                else {mpm=0; a[0]=0;} //altrimenti ammetto di poter trovare
un'entropia più bassa
       }
keepMaxInfo[0]=length; //daranno indicazioni alla prossima chiamata di searchMax per
andare più in profondità con la ricerca del max
keepMaxInfo[1]=mpm;
   }


   private double[] searchMaxPerLength(Node v, double[] a, double compare, int
length){
//NB: trova soltanto la sommatoria da 1 a length di ((4phi)^k)*count
       //v = nodo di partenza  |  a = array da ritornare |
       //compare = valore minimo su cui basare i confronti  |  length = lunghezza
cercata
Edge edges[]= v.getEdges().toArray(new Edge[0]);
           for(int i=0; i<edges.length; i++){

//se col ramo riesco a coprire la length posso ricavare l'entropia
if(v.getLength()+edges[i].getSpan()+1>=length) {

                    double
entropy=v.getEntropy()+edges[i].getEndNode().getCount()*getSummation(v.getLength()+1
, length, 4*phi);
//se tale entropia è maggiore
if(entropy>a[0]){
                     a[0]=entropy;

                     //se sono caduto esattamente in una foglia lo segnalo con un 1
nella seconda cella
                     //le guardie servono per essere sicuro che effettivamente sono
arrivato all'estremità di un ramo terminale
if( (length == edges[i].getEndNode().getLength()) && edges[i].getEndNode().isLeaf())
a[1]=1;
else a[1]=0;
                   }

                    //se ho trovato un'entropia uguale a quella precedente (guardia
1)e se quella precedente è stata data da una foglia (guardia 2)
                    //verifico che cioè non avvenga anche per l'entropia corrente:
se l'entropia corrente non è stata data
                    //da una foglia impongo a[1]=0, altrimenti a[1] vale già 1 e non
serve fare niente
else if(entropy==a[0] && a[1]==1 && !(length == edges[i].getEndNode().getLength())
&& edges[i].getEndNode().isLeaf())
a[1]=0;

           }

                else { //itero solo se viene superato il test dell'MPM controllando
eventualmente che il ramo non sia speciale
                    //con una sequenza random, se si vuole L=1000, oltre phi=0.5 i
valori saturano a infinito
                    //con una sequenza random, se si vuole L=500, oltre phi=1 i
valori saturano a infinito
                    //PER IL NUMERO DI NODI MASSIMI SOSTITUIRE NEL CONFRONTO compare
CON 0 E  !!!FARE ATTENZIONE CHE I VALORI NON SATURINO!!!

                    //per verificare la saturazione si può, per esempio, attivare la
seguente stampa a video:
//if(length==500)System.out.println((edges[i].getEndNode().getEntropy()+(edges[i].ge
tEndNode().getCount()-1)*getSummation
                    //        (edges[i].getEndNode().getLength()+1,length,4*phi))+"
>= "+compare+" ?");

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;
   }
```

```
   //alla posizione 0 c'è l'entropia normalizzata e alla posizione 1 quella non
normalizzata
public double[] getNormalizedEntropy(int position, intlengthChosen){
       double a[]=new double[2];
       a[1]=getEntropy(position, lengthChosen);
       a[0]=a[1]/getMaxEntropy(lengthChosen);
       return a;
   }

   public double getEntropy(int position, intlengthChosen){
       double main= searchEntropy(root, position, lengthChosen); //position e' la
posizione Xi nell'interastringa (text)
       double numerator=1+(main/(text.length()-1));              //-1 per via del
carattere $
       double denominator= getSummation(0, lengthChosen, phi);
return numerator/denominator;
   }


   private double searchEntropy(Node v, intcurrentPosition, intleftLength){ //metodo
d'appoggio per ottenere l'entropia semplificata

      Edge e=v.findEdge(text.charAt(currentPosition)); //trovo il ramo che mi serve

if(e.getSpan()+1>=leftLength) { //se con tale ramo riesco a coprire la lunghezza
voluta
          double entropyNode=e.getStartNode().getEntropy(); //l'entropia memorizzata
nel nodo superiore
double addingPartial=getSummation(e.getStartNode().getLength()+1,
e.getStartNode().getLength()+leftLength, 4*phi);
          return entropyNode+(addingPartial*e.getEndNode().getCount());
}

      else { //altrimenti itero scalando la lunghezza del ramo dalla lunghezza
cercata
leftLength+=-e.getSpan()-1;
          return
searchEntropy(e.getEndNode(),currentPosition+e.getSpan()+1,leftLength);
      }
   }


   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 voidlistaStruttura(Node v){
        Edge e[]= v.getEdges().toArray(new Edge[0]); //crea un array avente un
elemento della collezione in ogni posizione
        for (int i = 0; i <e.length; i++){ //scorre tutta la collezione

System.out.println("Numero Nodo: "+v.getName());
System.out.println(v.getEdges());

if(v.getSuffixNode()!=null) {System.out.println("Numero del suo SuffixNode:
"+v.getSuffixNode().getName());}
else {System.out.println("NessunSuffixNode");}

System.out.println(
                      "Numero Ramo: "+e[i]+
                    "   BeginIndex: "+e[i].getBeginIndex()+
                    "   EndIndex: "+e[i].getEndIndex()+
                    "   StartNode: " +e[i].getStartNode().getName()+
                    "   EndNode: " +e[i].getEndNode().getName()+
                    "   CountStartNode: "+e[i].getStartNode().getCount() +
                    "   CountEndNode: "+e[i].getEndNode().getCount()+
                    "   Etichetta: "+text.substring(e[i].getBeginIndex(),
e[i].getEndIndex()+1)+
```

```
                    "   isSpecial?  "+e[i].isSpecial()+
                    " \nil nodo "+v+" e' foglia? "+v.isLeaf()+"       il nodo
"+e[i].getEndNode()+" e' foglia? "+e[i].getEndNode().isLeaf()+
                    " \nLength del nodo "+v+": "+v.getLength()+"      Length del
nodo "+e[i].getEndNode()+": "+e[i].getEndNode().getLength()+
" \nNumeroRamo col char in A: "+ v.findEdge('A')+ "   in C: "+v.findEdge('C')+"   in
G: "+v.findEdge('G')+"   in T: "+v.findEdge('T')+
" \nEntropy del nodo "+v+": "+v.getEntropy()+"      Entropy del nodo
"+e[i].getEndNode()+": "+e[i].getEndNode().getEntropy()+
                    "\n\n");


if(e[i].getEndNode()!=null){listaStruttura(e[i].getEndNode());} //se i rami della
collezione hanno figli itera il metodo
}
    }



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



try{
            long start=System.nanoTime(); //per registrare il tempo di costruzione
dell'albero

int begin=Integer.parseInt(args[0]);
int length=Integer.parseInt(args[1]);
            double phi=Double.parseDouble(args[2]);
int gap=Integer.parseInt(args[3]);
int kFactor=Integer.parseInt(args[4]); //MODIFCA


            String text,tmp;
            text="";
            Scanner in = new Scanner(new File(args[5]));


tmp=in.nextLine();
            while(!tmp.isEmpty())
            {

               if( tmp.charAt(0) == '>')
              {
   System.out.println("*Reading sequence \n");
     text="";
              } else
              {
     text+= tmp;

              }
tmp=in.nextLine();
            }
in.close();

            //System.out.println("text "+text);
System.out.println("Sequence of length: "+ text.length() + " \n");


/*
if(args[5].length()>=4 &&args[5].substring(args[5].length()-4).equals(".txt")){ //se
la stringa ha almeno 4 caratteri e finisce con ".txt" leggo il file
BufferedReaderbr = new BufferedReader(new FileReader(args[5]));
                text=br.readLine();
System.out.println("Sequence of length: "+ text.length() + " \n");
}
            else text=args[5]; //altrimenti la sequenza è proprio il quarto
parametro
*/

            SuffixTree st = new SuffixTree(text, phi, kFactor); //MODIFICA

InputStreamReader reader = new InputStreamReader (System.in);
BufferedReadermyInput = new BufferedReader (reader);
int x=0; //per diversificare i .txt uscenti
```

```
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();  //per registrare il tempo di esecuzione
della query

            //creo un file dove memorizzare i valori entropici per posizioni
successive (a parità di lunghezza e gap)
FileWriterfw=new FileWriter("Entropy List "+x+".txt");
PrintWriter out=new PrintWriter(fw);
out.println("~~ Input file "+ args[5] +" : Entropy List from position "+begin+" with
length "+length+", window "+gap+", k factor "+kFactor+" and phi="+phi+" ~~");
out.println();
            for(int i=begin; i<=begin+gap; i++){
out.print(st.getNormalizedEntropy(i, length)[0]+" ");
}
out.close();


            //creo un file dove memorizzare il numero di nodi visitati ad ogni
lunghezza
/*
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" );

System.out.println("\nNew Query or q/Q to exit: \n"
          + "[Starting Position] [Pattern Length] [Phi Value] [Window]");


input = myInput.readLine(); //per eventuali altri parametri
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); //se è cambiato il phi devo risistemare le entropie
phi=phiNew;
                  long finish3 = System.nanoTime() - start3;
System.out.println("\nTime to update nodes entropies: "+ (finish3 / 1000000000.0) +
" seconds" );
              }
            gap = Integer.parseInt(token.nextToken());


          }

        }//try

        catch(ArrayIndexOutOfBoundsException e){
System.out.println("\n!!ERROR!!\nNot enough parameters or invalid values. In order
are required:\n[Starting Position] " +
            "[Pattern Length] [Phi Value] [Window] [k Factor] [FastaSequence] \n" +
            "Every value must be greater than 0 and only \"Starting Position\" and
\"Window\" can be equal to 0.\n"
            + "java -jar FastEP.jar 1 7 10 100 100 ecoli.fasta");
```

```
            }

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

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

        catch(FileNotFoundException e){
System.out.println("\n!!ERROR!!\nFile \""+args[5]+"\" not found");}

        catch(NoSuchElementException e){
System.out.println("\n!!ERROR!!\nNot enough parameters. In order are required:\n\n"+
            "[Starting Position] [Pattern Length] [Phi Value] [Window]");
            }

        catch(NumberFormatException e){
   System.out.println("\n!!ERROR!!\nNot enough parameters or invalid values. In
order are required:\n[Starting Position] " +
               "[Pattern Length] [Phi Value] [Window] [k Factor] [FastaSequence] \n"
+
               "Every value must be greater than 0 and only \"Starting Position\"
and \"Window\" can be equal to 0.\n"
               + "java -jar FastEP.jar 1 7 10 100 100 ecoli.fasta");
               }
    }


     }//SuffixTree
```

Analisi

Inseguito descriverò e spiegherò in dettaglio tutte le modifiche apportate al programma originale di Stefano Mazzocca [4], seguendo il codice dall'inizio verso la fine.

1) Alla classe *Node*:
- viene aggiunta la variabile intera *depth* per memorizzare la profondità del nodo cheservirà a determinare la profondità dell'albero dei suffissi, durante la costruzione dell'albero stesso;
- viene aggiunta la variabile intera *intLeaf* per distinguere se un nodo esterno è una foglia normale oppure è diventato tale a causa del troncamento dell'albero. Questo tipo di foglia è chiamata foglia interna;
- vengono aggiunti i metodi *setDepth( )* e *getDepth( )*per settare il valore *depth* e ottenere il valore di *depth.*
- viene aggiunto il metodo *isIntLeaf()* che restituirà un valore booleano per informare se un nodo esterno è una foglia interna;
- viene aggiunto il metodo *setIntLeaf()* per settare il valore di *intLeaf*;

2) Alla classe *Edge :*
- viene aggiunta una riga di codice per settare la profondità del nodo *endNode* dopo che esso è stato creato. La sua profondità è uguale alla profondità del suo genitore (*startNode*) più uno;
- viene aggiunta una riga di codice al metodo *splitEdge* per aggiornare la profondità di *endNode* dopo che l'arco è splittato e un nuovo arco e nodo sono stati aggiunti.

3) Alla classe *SuffixTree*:
- viene aggiunta la variabile intera *kFactor* per memorizzare il fattore di troncamento
- viene aggiunto il parametro intero *k* al costruttore di SuffixTree perché il *main* possa passare il fattore di troncamento;
- viene aggiunta una riga di codice per settare il fattore di troncamento, usando il parametro ricevuto dal *main*;
- viene aggiunta una riga codice per settare il valore della profondità del nodo *root* dopo che esso è stato creato;
- al metodo *addPrefix* viene aggiunto anche il fattore di troncamento *kFactor* come parametro. Di conseguenza quando nella classe *SuffixTree* si effettua la chiamata ad *addPrefix* bisogna passare anche *kFactor*;
- all'interno del metodo *addPrefix* quando il prefix *active* è implicito e bisogna chiamare il metodo *splitEdge*, prima della chiamata a *splitEdge* vengono aggiunte delle righe di codice per controllare se la profondità di quel ramo è uguale al fattore di troncamento. In caso affermativo

non si effettua la chiamata di *splitEdge*. Si limita a settare quel nodo esterno come foglia interna e al valore di *count* si aggiunge uno;

4) Al metodo *main*:
- viene aggiunto una riga di codice per leggere in input il valore del fattore di troncamento e per effettuare la conversione da stringa in intero;

Come mostrato sopra, non è stato aggiunto nessuno nuovo metodo all'algoritmo originale. L'ho analizzato e ho modificato in parte le classi ed alcuni metodi affinché l'algoritmo funzioni con la struttura di truncated suffix tree (chiamato anche k-factor tree).
Per una approfondita ed dettagliata analisi dell'implementazione rimandiamo al lavoro di Stefano Mazzocca [4] e al lavoro di Mark Nelson [8].

Come usare il programma

Per l'esecuzione del programma EntropicProfiler verrà richiesto all'utente di fornire la posizione d'inizio (Starting Position), la lunghezza della sottostringa che si vuole analizzare (Pattern Length) , il parametro phi (Phi Value), il gap tra posizione d'inizio e posizione finale (Window), la profondità alla quale si vuole troncare il suffix tree (k Factor), e alla fine un file di testo di tipo .fasta contenente sequenza di genoma. Ad esempio, il comando per eseguire l'EntropicProfiler su un file di testo sequence1.fasta con

- Posizione d'inizio = 0;
- Lunghezza del sottostringa = 5;
- Phi = 8;
- Gap = 10;
- Fattore di troncamento = 100;

Si digitail seguente comando:
java -jar EntropicProfiler.jar 0 5 8 10 100 sequence1.fasta

Bisogna prestare attenzione al fattore di troncamento scelto. Infatti, se questo fattore è piccolo (quindi l'albero non sarà sufficientemente alto) rispetto ai parametri length e gap, il programma segnalerà l'errore con il seguente messaggio:

"Gap" / "Chosen Length"or both parameters too high

Oppure può produrre valori non corretti di Entropy.

Il comando seguente provocherà l'errore sopra detto:
java -jar EntropicProfiler.jar 0 5 8 8 10 sequence1.fasta

Questo invece no:
java -jar EntropicProfiler.jar 0 5 8 8 20 sequence1.txt

Inoltre solo Starting Position e Window possono assumere valore zero. Se non si forniscono in input valori corretti il programma segnalerà l'errore con un messaggio.
Una volta il comando è stato eseguito, e quindi è stato costruito il suffix tree troncato ad un certo fattore k, le successive analisi sullo stesso file di testo sequence1.fasta con suffix tree troncato allo stesso fattore k sarà fatte immettendo i primi quattro parametri numerici (Starting Position, Pattern Length, Phi Value e Window). Ad esempio, se si vuole analizzare la sequenza di prima (contenuta nel file sequence1.fasta)

con una sottostringa che inizia dalla posizione 2, lunghezza scelta 4, phi uguale a 3, gap uguale a 5, allora il comando è:

2 4 3 5

Per uscire dal programma EntropicProfiler si digita q oppure Q.
Se si vuole analizzare la stessa sequenza di dati però con un fattore di troncamento diverso, allora bisogna uscire dal programma e digitando di nuovo il comando con il precedente file di testo, questa volta con il nuovo fattore di troncamento.
Ogni volta si effettua un'analisi con EntropicProfiler verrà prodotto un file di testo contenente il valore normalizzato di entropic, il file Entropy List#.txt.

Capitolo III

Breve descrizione del Massimo Entropy

Il fatto che l'originale normalizzazione della funzione EP richiede molto tempo e spazio, rispettivamente $O(n^3)$e $O(n^2)$, ha portato a un altro modo di normalizzazione usando la seguente formula

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

Dove ***max***$0\leq j<n$ $[fL,\Phi(j)]$ ritorna il massimo valore di $fL,\Phi(j)$ su un insieme di parole di lunghezza L. Questa funzione normalizzata di EP può assumere valore compreso tra 0 e 1.
Per trovare massimo entropy per ogni lunghezza L senza dover comparare tutte le parole di lunghezza L si ricorre alla funzione mpm (minimum potential maximum) e alla funzione MPM (maximum potential maximum). Sono definite come segue:

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

Dove $h(v)$ è la lunghezza del percorso dalla radice fino al nodo v e v è il nodo tale che h(v)<L.
MPM definisce un limite superiore al massimo entropy ottenibile per il percorso dalla radice che passa attraverso il nodo v.

$$\mathrm{mpm_L} = \begin{cases} 0 & \text{if } L=1 \text{ or } MPM_{L-1} \text{ was returned by a leaf node} \\ \mathrm{MAX_{MPM_{L-1}}} + (4\varphi)^L & \text{otherwise} \end{cases}$$

dove $MAX_{MPM_{L-1}}$ è il massimo tra tutti i MPM. mpm definisce il limite inferiore al massimo entropy per un dato valore di L.
Per trovare massimo entropy per lunghezza L' bisogna attraversare l'albero per ogni L, dove $1\leq L\leq L'$, attraversando tutti nodi (chiamati nodi promettenti) che hanno MPM maggiore di mpm. Per i nodi (non promettenti) che non hanno MPM maggiore di mpm il percorso non

procede. In questo modo si diminuisce il tempo di ricerca.
Ogni volta il valore di L aumenta, la traversata del suffix tree comincia dall'inizio e mpm_L è settato con il massimo valore di MPM_{L-1} ottenuto dai nodi raggiunti dalla traversata più $(4\varphi)^L$ per definizione. Tuttavia, se il nodo che ritorna MPM_{L-1} è una foglia allora non è più garantito che al prossimo passo mpm_L corrisponde al minimo valore possibile di entropy, perciò esso viene resettato.
Quando si arriva alla fine di tutti percorsi promettenti, il massimo entropy sarà scelto per calcolare il prossimo mpm, o eventualmente per ritornare il valore richiesto.

Sotto è riportato un esempio per mostrare come si ottiene il valore di entropy per ogni nodo dell' implicit suffix tree (il carattere *$* è omesso) in figura, con *φ=0.25.*

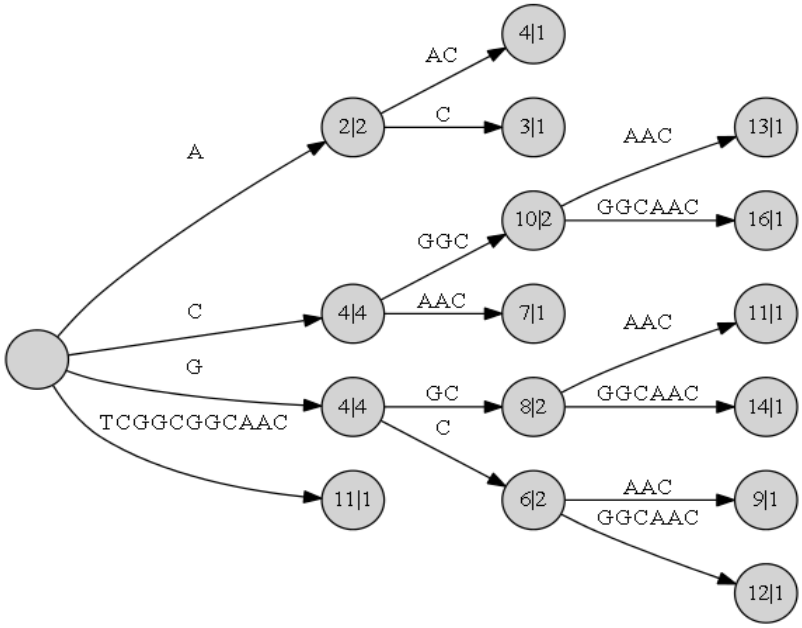
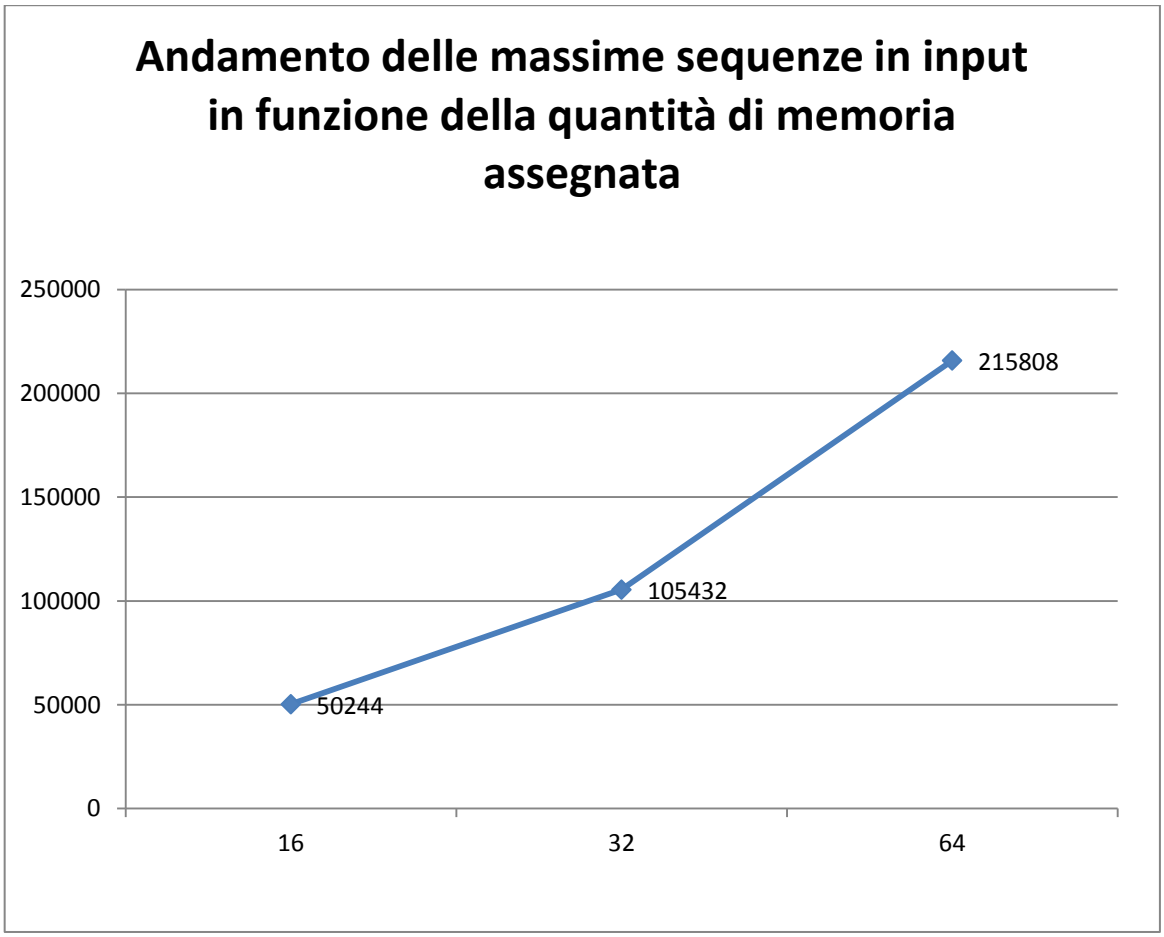


Figure3.1: implicit suffix tree having φ=0.25 for string TCGGCGGCAAC. *Nodes are labeled with the corresponding values of* entropy|count

Per ottenere entropy di un nodo *v*, l'algoritmo, implementato con metodo *searchEntropy*, non ha bisogno di visitare *v,* può fermarsi al genitore di *v*.
Ad esempio, per conoscere entropy della stringa *TCGGCGGCAAC*, è sufficiente sta alla radice e sommare al suo entropy, il quale ha valore 0 per default, la lunghezza dell'arco moltiplicata per il numero di figli: *0+11*1=11*.
Invece, se si vuole conoscere l'entropy della stringa *CGGC*, l'algoritmo si fermerà al nodo *C*, che ha entropy pari a 4, e il calcolo è:
*4+3*2=10*.

Risultati delle prove

Il grafico seguente mostra i risultati delle prove, fatte con il programma basato sul suffix tree normale, per trovare la più lunga sequenza di DNA in input per una prefissata quantità di memoria, che non causa l'errore di memoria insufficiente.



- Sull'asse delle ascisse i valori della memoria in MegaByte
- Sull'asse delle ordinate i valori della lunghezza delle sequenze di DNA in input

Di seguito sono invece i risultati delle prove eseguite su un file di testo contenente una sequenza DNA di lunghezza 278880, con la memoria riservata per l'albero dei suffissi limitata rispettivamente

A 16MB:

- Usando il programma di Stefano Mazzocca con la normale struttura di suffix tree abbiamo avuto un errore di memoria insufficiente. Il programma mostra un messaggio di tipo
  java.lang.OutOfMemoryError: Java Heap Space

- Usando il programma con la struttura di truncated suffix tree per il valore del fattore di troncamento fino a 11 (k-Factor=11) il programma non ha avuto problema di memoria insufficiente. Per k-Factor maggiore di 11 il programma ha segnalato l'errore di memoria insufficiente.

A 32MB:

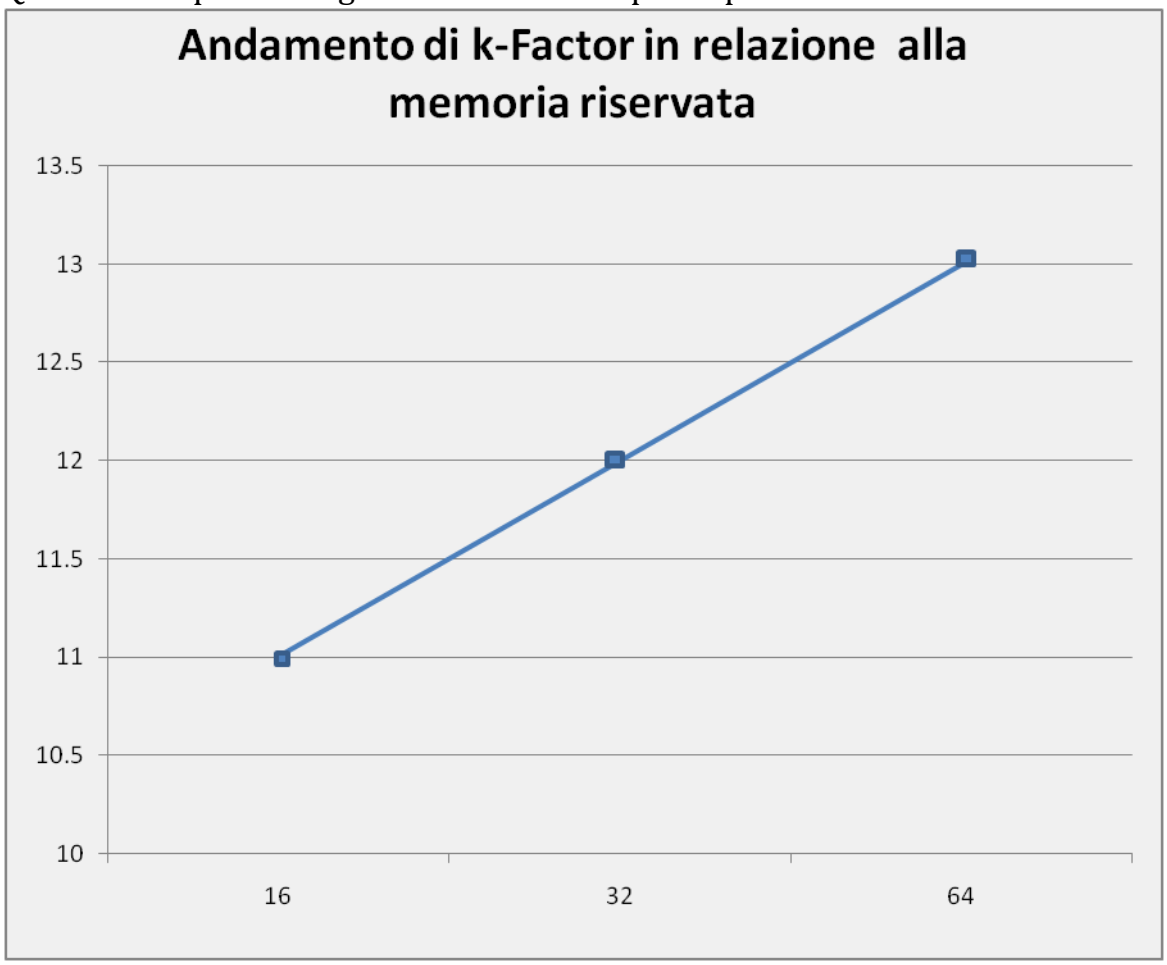
- Usando il programma con la normale struttura di suffix tree otteniamo ancora l'errore di memoria insufficiente.
- Il programma basato sulla struttura di truncated suffix tree ha avuto il problema con la memoria quando il k-Factor supera il valore 12.

A 64MB:

- Con il programma basato sul normale suffix tree abbiamo ancora problema di memoria insufficiente.
- Il programma basato sul truncated suffix tree funziona per k-Factor minore di 14.

Solo a 128MB il programma basato sul normale suffix tree non presenta più problema di memoria insufficiente.

Qui sotto è riportato il grafico relativo alle prove precedenti.



- Sull'asse delle ascisse i valori della memoria in MegaByte
- Sull'asse delle ordinate i valori del fattore di troncamento k-Factor

Bibliografia

[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] Stefano Mazzoca, Matteo Comin: *Implementation of Entropic Profiler for DNA Sequences by using Suffix Trees.*

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

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

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

[8]http://marknelson.us/1996/08/01/suffix-trees/

[9] http://gsuffix.sourceforge.net/

[10] http://illya-keeplearning.blogspot.it/2009/04/suffix-trees-java-ukkonens-algorithm.html