UNIVERSITÀ DEGLI STUDI DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA SPECIALISTICA IN INFORMATICA

Tesi di Laurea Specialistica

Sulla distanza di mutazione tra sequenze biologiche

autore : Claudio Felicioli

relatore : Fabrizio Luccio

co-relatore : Roberto Marangoni

ANNO ACCADEMICO 2003-2004

Indice

Sommario

Nella presente tesi intendo descrivere ed analizzare nel dettaglio un algoritmo ideato per calcolare una particolare misura di distanza tra sequenze di caratteri. Più specificatamente questo algoritmo, chiamato BpMatch, ha lo scopo di determinare quanto una certa sequenza genica possa essere ritenuta imparentata con una seconda, tenendo conto dei principali fenomeni di mutazione.

La motivazione iniziale di questo lavoro è stata la grave inefficienza degli algoritmi noti: la soluzione proposta appare estremamente più efficiente, come sarà chiarito nel corso del testo.

Per poter comprendere perché un algoritmo come quello in analisi può fornire utili informazioni sulle dinamiche del processo evolutivo è necessario avere chiare alcune nozioni di biologia e di genetica, tra cui le principali tipologie di mutazione che insistono sul DNA genomico, il concetto di similitudine e quello di omologia. Una breve e sicuramente incompleta trattazione di queste nozioni (priva di ogni ambizione diversa dal giustificare determinate analisi delle sequenze geniche, tra cui quella effettuata da BpMatch) si può trovare in appendice (appendice B).

La prima sezione introduttiva parlerà di alcuni approcci allo studio ed al confronto di sequenze geniche e spiegherà in risposta a quali bisogni è stato ideato l'algoritmo BpMatch, seguirà poi una sua descrizione dettagliata, lo studio della sua complessità in spazio ed in tempo, poi infine l'analisi degli elementi dell'algoritmo che ho ritenuto più interessanti.

Per l'algoritmo BpMatch è stato sviluppato un programma completa-

mente funzionante, il cui comportamento a livello statistico su sequenze geniche reali, ed il confronto con quello previsto per le sequenze pseudocasuali nella complessità nel caso medio, sono discussi nel corso della tesi.

Nella prima appendice sono proposte le definizioni formali e la dimostrazione del corretto funzionamento delle strutture dati su cui BpMatch si basa. Nella seconda appendice si accennerà all'evoluzione ma soprattutto verranno elencati alcuni dei possibili fenomeni di mutazione. Nella terza e conclusiva appendice vi è parte del codice sviluppato per implementare BpMatch, per testarne il funzionamento e per controllarne il comportamento.

1 Introduzione

La ragione per cui, nel campo della bioinformatica, si effettuano confronti tra sequenze geniche cercando di calcolare una misura della loro similitudine, sta nell'ipotesi che una forte similitudine implica l'elevata probabilità che le sequenze siano anche omologhe[1], cioè che posseggano un antenato comune[4].

La ricerca ha quindi sviluppato diversi algoritmi in grado di calcolare una distanza di mutazione in grado di individuare le parentele, sia tra singoli geni che tra genomi.

Gli algoritmi basati sull'allineamento delle due (o più) sequenze che si vogliono confrontare cercano di fornire una misura di similitudine in funzione del numero e del tipo di mutazioni puntiformi (aggiunta, rimozione o modifica di un singolo carattere della sequenza) che possono trasformare una sequenza nell'altra.

Oggi però, grazie allo sviluppo della biologia molecolare, si sa che la mutazione puntiforme non è l'unica ad agire sul materiale genetico: lo studio dell'informazione genetica ha messo in evidenza il fondamentale ruolo coperto dalla mutazione di segmenti[3][6][7].

Un paradigma biologico emergente è che la comparsa di nuovi geni con nuove funzioni sia principalmente dovuta al fenomeno di duplicazione (tramite mutazione di segmenti) di un gene in svariate copie identiche, seguito dalla successiva divergenza funzionale di una copia causata dall'accumulo di mutazioni in essa[10][11]. Da qui l'importanza dello studio del modo in

[1] La relazione tra omologia e similitudine si trova nel fatto che le mutazioni casuali mantenute dalle sequenze si accumulano nel tempo, quindi coppie di sequenze con un comune antenato molto antico tendono ad avere una minore similitudine rispetto a quella che si può riscontrare tra coppie di sequenze che si sono separate in tempi più recenti.

cui le famiglie di geni paraloghi[2] nascono e si sviluppano e quindi la necessità di avere strumenti in grado di rilevare l'azione della mutazione di segmenti.

Gli algoritmi di allineamento però, anche se adatti per misurare la similitudine di singoli geni omologhi che si sono tra di loro allontanati per effetto della sola mutazione puntiforme, non riescono a tener conto di fenomeni come la duplicazione o la moltiplicazione di una sottosequenza, la sua traslazione all'interno della stringa originaria, le inversioni complementate ecc.[8][4].

Il fenomeno della mutazione di segmenti non è quindi analizzabile tramite gli algoritmi di allineamento, per questo motivo sono stati seguiti svariati approcci alternativi.

Uno di questi approcci, il calcolo della distanza di riarrangiamento del genoma, cerca di valutare il minimo numero di mutazioni di sequenza che possono trasformare l'ordinamento di alcune sequenze geniche da una lista ordinata ad un'altra[12][13]. Limitare l'analisi dell'azione della mutazione di segmenti al riarrangiamento dell'ordine in cui alcune sequenze compaiono in un genoma non si è però rivelato un approccio in grado di fornire informazioni utili. Lo stesso insuccesso è stato constatato per molti degli altri approcci d'analisi del materiale genetico alternativi all'allineamento.

E proprio con l'intento di rispondere alla necessità di avere una misura di distanza tra sequenze geniche che tenga conto di tutti i principali modi in cui il genoma muta, J.S. Varré, J.P. Delahaye e E. Rivals in [1] hanno inizialmente definito la *transformation distance* (TD).

La TD considera non solo le mutazioni genetiche di singole basi, ma le

[2]Dei geni si dicono tra di loro paraloghi quando sono omologhi, presenti nello stesso organismo e lievemente diversi tra di loro[9]. Una famiglia di geni paraloghi si considera originata a seguito di fenomeni di duplicazione genica, cioè di mutazione di segmenti.

mutazioni genetiche di segmenti, come la copia traslata, la copia traslata invertita e complementata e la rimozione di segmenti. Un esempio è mostrato nella figura 1.

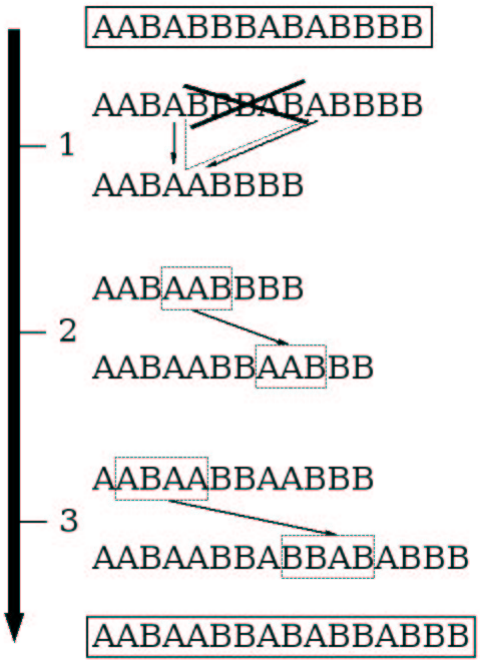

Figura 1: un esempio di mutazione di segmenti: la prima mutazione è una rimozione, la seconda è una copia, la terza è una capia invertita traslata (ponendo che il complemento di A è B e viceversa).

Nel confronto tra due sequenze, poniamo S e T, il procedimento per il calcolo di TD si basa sulla costruzione di T a partire da S applicando una sequenza di operazioni come la rimozione, la duplicazione, la traslazione o la copia traslata invertita complementata di segmenti, oppure l'inserzione di nuove sottosequenze che non compaiono già in S. Ci saranno in genere più serie di operazioni in grado di trasformare S in T. Ogni operazione

ha un costo, definito dalla quantità di informazione che la descrive, così, ad esempio, la copia traslata di una sequenza abbastanza lunga avrà un costo inferiore ad un'operazione equivalente che specifica invece ex novo la sequenza di caratteri da inserire (dato che per descrivere la copia traslata basta specificare il punto di partenza, la lunghezza della sequenza da copiare e l'offset). La misura di distanza TD è la minima somma dei costi delle operazioni di un insieme che trasforma S in T.

L'algoritmo per calcolare la TD presentato in [1], che non considera la sovrapposizione delle sottosequenze, risulta avere l'enorme complessità in tempo nel caso pessimo di $O(n^6)$ ed una elevata (seppur non analizzata in [1]) complessità anche in spazio. È proprio il costo in spazio che rende l'algoritmo generalmente inapplicabile su sequenze di lunghezza maggiore di 100kb. Un secondo algoritmo (*compact script graph*) è menzionato sempre in [1] e viene mostrato un grafico in cui il calcolo della TD sembrerebbe essere quadratico in tempo rispetto alla lunghezza delle sequenze. Una precisa analisi di questa seconda versione non viene però fornita e la complessità spaziale non è stata calcolata.

La TD è stata utilizzata in alcuni lavori di analisi del materiale genetico (ad esempio in [2] per studiare le famiglie di geni tramite la costruzione di alberi di paralogia), però l'altissimo costo computazionale dell'algoritmo che la calcola ne ha frenato l'utilizzo ed ha praticamente limitato le sue grandi potenzialità di applicazione.

Per rimediare ai notevoli problemi di costo computazionale del calcolo della TD, un nuovo algoritmo, chiamato *BpMatch*, verrà adesso proposto ed analizzato nel dettaglio. BpMatch permette il calcolo di una misura di

distanza ispirata a TD: data una sequenza di caratteri di partenza S, una sequenza di caratteri obiettivo T e una minima lunghezza l, esso determina la massima copertura di T utilizzando solo sottosequenze e sottosequenze inverse complementate di S, di lunghezza minima l, eventualmente sovrapposte parzialmente, e, in una tale copertura massima, riesce a minimizzare il numero di sottosequenze di S utilizzate.

Come sarà dimostrato nel testo BpMatch richiede una preelaborazione della sequenza di partenza S, eseguita in tempo $\mathrm{O}(n \log_d(n) \log(\log_d(n)))$ nel caso medio, con $n = |S|$ e d la cardinalità dell'alfabeto, generando due strutture dati di dimensione $\Theta(n)$. Poi, per ogni sequenza obiettivo T di lunghezza m, la distanza può essere computata in tempo e spazio $\mathrm{O}(m \log(\log(n)))$ nel caso medio se si ha che $l \simeq 2 \log_d(n)$ o maggiore (l comunque non deve essere minore se si vuole ottenere una misura di distanza attendibile, come verrà spiegato nel seguito).

Per generalizzare, BpMatch è presentato come un algoritmo indipendente dall'alfabeto delle sequenze, anche se è stato concepito per essere utilizzato principalmente sul codice genetico.

Le diverse applicazioni di BpMatch sono praticamente le stesse di TD, in questa tesi non viene quindi mostrato alcun utilizzo pratico di BpMatch. L'analisi che seguirà la descrizione dell'algoritmo sarà invece incentrata sui suoi costi computazionali e sulla variazione delle sue proprietà statistiche quando viene impiegato su sequenze genetiche reali piuttosto che su sequenze casuali.

Alcune definizioni formali e dimostrazioni sono riportate nell'appendice A.

2 L'algoritmo

L'algoritmo è diviso in due fasi.

Nella prima fase avviene una preelaborazione della sequenza di partenza S e vengono generati due oggetti che permettono un rapido riconoscimento delle sottosequenze e delle sottosequenze invertite e complementate di S.

Nella seconda fase, data una sequenza obiettivo T, i due oggetti generati durante la prima fase (G per le sottosequenze dirette di S, G' per le sottosequenze invertite e complementate di S) ed una lunghezza minima l, viene calcolata la massima copertura di T utilizzando solo sequenze riconosciute o da G o da G', di lunghezza minima l, e il numero di sequenze impiegate viene minimizzato.

Dividendo il problema in questo modo, la preelaborazione di S, computazionalmente più pesante rispetto alla seconda fase, viene eseguita una sola volta, così un calcolo della massima copertura può essere poi eseguito rapidamente per ogni possibile sequenza obiettivo T.

2.1 Prima fase : la generazione dei grafi

2.1.1 I grafi

Un BpGraph è una struttura dati che rappresenta l'intero insieme delle sottosequenze di S come percorsi di un grafo aciclico.

L'idea intuitiva di quest'oggetto, chiamato BpGraph per un rapido riferimento, è basata sul predisporre percorsi forzati tra archi di un grafo (un percorso per ogni sottosequenza di S e, se possibile, riutilizzando pezzi di

percorsi già predisposti) in cui ogni nodo viene etichettato con un carattere dell'alfabeto in uso.

Una sequenza u viene riconosciuta se un oggetto posizionato sui nodi, chiamato cursore, riesce con successo a seguire un percorso attraverso i nodi e gli archi di BpGraph tale da raggiungere nel corretto ordine i nodi le cui etichette, concatenate, formino la stringa u.

I percorsi sono detti forzati quando, dato un certo cursore posizionato su un certo nodo ed un carattere dell'alfabeto da elaborare, vi è un unico arco, tra quelli uscenti dal nodo su cui è piazzato il cursore, che possa essere percorso, se tale arco esiste.

Per ottenere questa proprietà senza dover generare un grafo enorme, i cursori sono dotati di uno stato interno, il cui valore limita gli archi che possono essere considerati come validi da percorrere. Lo stato può variare ogni volta che il cursore, percorrendo un arco, si muove da un nodo ad un altro.

Data la sequenza u ed un cursore nello stato di parenza, non ancora posizionato su di un nodo del grafo BpGraph, u è una sottosequenza di S se e solo se il cursore riesce a trovare la strada forzata (l'unica, se esiste) che gli permette di attraversare dei nodi le cui etichette, concatenate, formino la sequenza u. Ovviamente c'è un modo per inserire un cursore nel BpGraph quando il primo carattere di u viene analizzato e, dato che ogni singolo passo del percorso deve essere determinato solo dallo stato interno del cursore e dal carattere di u in analisi, per ogni carattere dell'alfabeto in uso esiste al più un unico punto di ingresso nel grafo.

Grazie alla presenza dello stato interno del cursore, possono essere per-

corse delle strade comuni da più cursori con stati diversi, senza però perdere la possibilità di distinguerli ed eventualmente, successivamente, di dividerli su diversi percorsi distinti.

Il fatto che più cursori possano seguire pezzi comuni di percorsi durante il riconoscimento di diverse sequenze di caratteri, permette ad un BpGraph di essere lineare in spazio rispetto alla lunghezza di S.

L'unicità dei percorsi è la proprietà che permette ai cursori di percorrere nel BpGraph una strada di lunghezza m in tempo $O(m \log(maxlink))$, con $maxlink$ il numero massimo di archi uscenti da un singolo nodo del grafo e, generando il BpGraph utilizzando il seguente algoritmo, questo numero massimo di archi uscenti risulta essere al peggio lineare rispetto alla lunghezza di S ($n = |S|$), quindi la complessità in tempo del determinare il percorso è, nel caso pessimo, $O(m \log(n/m))$. Nel caso medio invece la complessità è $O(m \log(\log(n)))$, sia per sequenze generate in modo pseudocasuale, che, come verrà approfondito in seguito, per sequenze geniche.

Gli archi che possono essere percorsi da qualsiasi cursore, indipendentemente dal suo stato interno, sono chiamati *diretti*, gli archi che invece possono venire percorsi solo da cursori con uno specifico stato interno sono chiamati *laterali*.

Nel seguente algoritmo tutti gli archi diretti vengono creati in modo da formare un singolo percorso, monodirezionale aciclico, che inizia da un nodo etichettato con il primo carattere di S, collegato ad un nodo contenente il secondo carattere e così via, uno per ogni carattere si S. Questi sono tutti i nodi e tutti gli archi diretti di cui è composto il BpGraph.

2.1.2 La generazione di BpGraph

I cursori, olte che per riconoscere una stringa in un BpGraph, sono utilizzati anche durante la sua creazione del grafo, ma in questo caso hanno un diverso comportamento.

d è la cardinalità dell'alfabeto in uso

$S = c_0\ c_1\ c_2\ ...\ c_{n-1}\ EOF$

L'oggetto *cursore* ha i seguenti campi :

- $nodeId$: l'identificatore del nodo su cui è posto
- $from$: il suo stato interno

L'oggetto *nodo* ha i seguenti campi :

- id : il suo identificatore
- $label$: il carattere dell'alfabeto in uso con cui è etichettato
- $outgoingArcs[\ ]$: la lista degli archi uscenti

L'oggetto *arco*, che è monodirezionale, ha i seguenti campi:

- id : il suo identificatore
- $label$: il carattere con cui è etichettato il nodo a cui punta (non è necessario ma rende più rapido l'utilizzo del grafo)
- $from$: 0 se l'arco è *diretto*, altrimenti, se è *laterale*, specifica il valore che un cursore deve avere nel suo campo $from$ per poterlo percorrere
- $toNode$: l'identificatore del nodo a cui punta

L'array $starts[\ ]$, di dimensione d, ha per elementi gli identificatori dei nodi he costituiscono i punti d'ingresso del grafo per i cursori, uno per ogni differente carattere dell'alfabeto

```
makeBpGraph(S) {
[INIT]  L'array starts[] viene inizializzato a dimensione d e tutti i suoi elementi
         settati a -1;
        i = 0;
[READ]  c = l'i-esimo carattere di input;
        If (c != EOF) {
[ADD]       newNodeId = nextNodeId();
```

```
            Un nuovo nodo con id=newNodeId e label=c viene aggiunto al grafo;
            If (i != 0) {
                Un nuovo arco diretto con id=nextLinkId() e label=c viene aggiunto
                 all'array outgoingArcs[] del nodo generato nella precedente
                 iterazione; l'arco punta al nodo generato in questa iterazione
                 (quello con id==newNodeId);
                }
[WALK]      Per ogni cursore {
                Se l'array outgoingArcs[] del nodo in cui il cursore corrente è posto
                 contiene un arco diretto con label=c, o se, altrimenti, con label=c
                 ma laterale e con il campo from uguale al campo from del cursore {
                    Il campo nodeId del cursore viene aggiornato al valore toNode
                     di tale arco (cioè, concettualmente, il cursore viene posizionato
                     sul nodo con id==toNode);
                    Se l'arco individuato è laterale (cioè se from!=0) {
                        Il valore del campo from del cursore viene aggiornato al campo
                         id dell'arco;
                        }
                    }
                Else {
                    Un nuovo arco laterale con id=nextLinkId(), label=c e from uguale
                     al campo from del cursore viene aggiunto all'array outgoingArcs[]
                     del nodo in cui il cursore è posto, questo nuovo arco punta
                     all'ultimo nodo creato (quello con id==newNodeId);
                    Il cursore viene rimosso;
                    }
                }
[CURSOR]    If (starts[c] == -1) then STARTS[c]=newNodeId;
            Else {
                Viene creato un nuovo cursore con nodeId=start[c] e from=c;
                }
            i++;
            goto [READ];
            }
        END;
```

```
}
```

2.1.3 Le proprietà di BpGraph

Il BpGraph G (costruito da S) riconosce u se e solo se u è una sottosequenza di S. La dimostrazione si trova nell'appendice A.

2.2 Seconda fase : il calcolo della massima copertura di T

Il calcolo della massima copertura di T, minimizzando il numero di sottosequenze e di sottosequenze invertite e complementate, di lunghezza minima l, eventualmente sovrapposte, richiede l'utilizzo dei BpGraph G e G', calcolati a partire da S durante la prima fase e che riconoscono rispettivamente le sue sottosequenze dirette e le sue sottosequenze invertite complementate.

L'algoritmo analizza la sequenza T ed inizia considerando il suo primo carattere come un possibile inizio per una sottosequenza, procedendo iterativamente, leggendo ad ogni iterazione il carattere successivo di T, finché non viene raggiunta la fine della sequenza.

In ogni iterazione si verifica uno dei seguenti due casi: se è verificata l'ipotesi che nella sequenza T il carattere appena precedente a quello in analisi non può essere coperto da alcuna sottosequenza, allora ci si trova nel caso 1, se invece è stata individuata una sottosequenza che termina esattamente prima del carattere in analisi, allora ci si trova nel caso 2.

Definiamo c_0 come il carattere in analisi e c_i come il carattere che si trova (in T) i posizioni più avanti di c_0. Definiamo poi il carattere $\overline{c}$ come c complementato.

2.2.1 Caso 1

È verificata l'ipotesi che nella stringa T il carattere c_{-1}, quello appena precedente a quello correntemente in analisi, non può essere coperto da alcuna sottosequenza, quindi c_0 può essere coperto se e solo se costituisce il primo carattere di una sequenza della copertura di T.

Si parte cercando una sequenza diretta di lunghezza almeno l, che inizi da c_0 e che copri T fino almeno al carattere c_{l-1}. Tale sequenza esiste se e solo se un cursore del BpGraph G', nello stato di partenza, analizzando in ordine i caratteri complementati $\overline{c_{l-1}}$ $\overline{c_{l-2}}$ ... $\overline{c_0}$, riesce a trovare un percorso e quindi c_{l-1} c_{l-2} ... c_0 è una sottosequenza invertita di S.

Se l'operazione di riconoscimento utilizzando G' ha successo, allora c_0 è il primo carattere di una sequenza diretta di lunghezza almeno l, quindi si parte a cercare tale sequenza, facendo analizzare in ordine c_0 c_1 ... ad un cursore di G nello stato di partenza. Quando, durante l'analisi di un c_i ($i \geq l$), il cursore fallisce nell'individuare un percorso, allora la sottosequenza diretta di S c_0 c_1 ... c_{i-1} è stata individuata.

Altrimenti, se la ricerca della sottosequenza invertita iniziando da c_{l-1} fallisce durante la lettura di un certo carattere c_i, si può dedurre che c_0 non può essere coperto da alcuna sequenza diretta di lunghezza minima l, perché altrimenti una tale sequenza dovrebbe contenere anche la sottosequenza c_i c_{i+1} ... c_{l-1} che, non essendo una sottosequenza diretta di S (dato che il suo complemento invertito non è stato riconosciuto da S'), non può essere contenuta in nessuna sottosequenza diretta. Ma questo significa che anche c_1 cade nell'ipotesi del caso 1 (almeno per quanto riguarda le sequenze dirette) e quindi non può essere coperto da sottosequenze dirette di S. Quindi tutti

i caratteri fino a c_i incluso non possono venire coperti da sequenze dirette.

Per cercare una sottosequenza invertita e complementata di S, che inizi da c_0, bisogna usare la stessa procedura, utilizzando G al posto di G' e vice versa.

Alla fine, tre distinti casi possono verificarsi :

1 - Se viene individuata solo una sottosequenza, diretta o invertita complementata, che parta da c_0, allora deve venire aggiunta alla copertura ed il carattere immediatamente successivo al termine della sequenza sarà analizzato seguendo la procedura descritta per il caso 2.

2 - Se sono state individuate sia la sottosequenza diretta che quella invertita complementata, allora la più lunga delle due deve venire aggiunta alla copertura ed il carattere immediatamente successivo al termine di tale sequenza sarà analizzato seguendo la procedura descritta per il caso 2.

3 - Se né la sottosequenza diretta, né quella invertita complementata sono state individuate, allora si escludono tutti i caratteri che non possono essere coperti né da alcuna sottosequenza diretta, né da alcuna sottosequenza invertita complementata, ed il carattere che subito segue quelli che sono stati esclusi sarà analizzato seguendo la procedura descritta per il caso 1.

2.2.2 Caso 2

È verificata l'ipotesi che il carattere c_{-1}, quello appena precedente a quello correntemente in analisi, è l'ultimo carattere di una sottosequenza della copertura di T, quindi tutti i caratteri precedenti fino a c_{-l} incluso sono coperti e c_0 può dunque non solo essere il primo carattere di una nuova sequenza, ma può trovarsi anche nella posizione centrale o finale di una sequenza,

di lunghezza almeno l, parzialmente sovrapposta a quella che termina nel carattere c_{-1}.

Per trovare la migliore sequenza diretta che copra c_0, se esiste, vengono analizzati dal cursore C di G, nello stato iniziale, i caratteri $c_0\ c_1\ ...\ c_i$ fino a che, durante la lettura di c_i, C fallisce nell'individuare un percorso.

Se $i \geq l$ allora si conclude. La sequenza $c_0\ c_1\ ...\ c_{i-1}$ potrebbe anche essere soltanto la parte finale di una più lunga sequenza parzialmente sovrapposta, ma è ovvio che non esiste nessuna sequenza parzialmente sovrapposta che includa i caratteri da c_0 a c_i in quanto $c_0\ c_1\ ...\ c_i$ non è riconosciuta come una sottosequenza diretta di S e quindi neanche qualsiasi sequenza che abbia $c_0\ c_1\ ...\ c_i$ come una sua sottosequenza potrebbe venire riconosciuta da G.

Se invece $i < l$, allora bisogna cercare una sequenza diretta parzialmente sovrapposta, che copra il massimo numero di caratteri non ancora coperti si sa già che non sarà in grado di coprire caratteri fino a c_i (non essendo $c_0\ c_1\ ...\ c_i$ una sequenza diretta).

I caratteri $\overline{c_0}\ \overline{c_{-1}}\ ...\ \overline{c_{-k}}$ devono venire analizzati da un cursore di G' fintanto che $k < l$ o finché il cursore fallisce nel trovare un percorso, così che $c_{-1}\ c_{-2}\ ...\ c_{-k}$ è una sottosequenza invertita di S ed è anche il più lungo, totalmente sovrapposto, prefisso (di lunghezza k) possibile della sequenza diretta parzialmente sovrapposta che si sta cercando.

Il più lungo suffisso $(c_0\ c_1\ ...\ c_{i-1})$ è di lunghezza i, quindi se $k + i < l$ si ha fallito e la sequenza diretta non esiste, altrimenti la sequenza $c_{i-l}\ c_{i-l+1}\ ...\ c_{i-1}$ deve venire analizzata da un cursore di G.

Se il cursore ha successo nel seguire un percorso fino a c_{i-1}, allora si

è trovata una sottosequenza diretta che copre tutto il massimo suffisso (e quindi la migliore che si poteva trovare), altrimenti, se si fallisce durante la lettura di c_j, significa che $c_{i-l}\ c_{i-l+1}\ ...\ c_j$ non è una sequenza diretta di S e che quindi, dato che non si può includerla in alcuna sottosequenza diretta di lunghezza l, dato che c_{i-l+1} non può essere il carattere iniziale di una sequenza diretta che arrivi fino a c_{i-1} (sarebbe al massimo di lunghezza $l-1$), la lunghezza del massimo suffisso deve essere ridotta da i a j.

Se $k+j < l$ si ha fallito e la sequenza diretta non esiste, altrimenti la stessa procedura deve essere ripetuta rimpiazzando le i con delle j.

Nel peggiore ed estremamente improbabile caso, è necessario ripetere questa procedura per un massimo di $l-1$ volte per riuscire a trovare, se esiste, la sequenza diretta parzialmente sovrapposta che si sta cercando.

Per cercare una sottosequenza invertita e complementata di S, che inizi da c_0, bisogna usare la stessa procedura, utilizzando G al posto di G' e vice versa.

Alla fine, tre distinti casi possono verificarsi :

1 - Se viene individuata solo una sottosequenza, diretta o invertita complementata, allora verrà aggiunta alla copertura ed il carattere immediatamente successivo al termine della sequenza sarà analizzato seguendo la procedura descritta per il caso 2.

2 - Se sono state individuate sia la sottosequenza diretta che quella invertita complementata, allora quella che copre più caratteri ancora scoperti verrà aggiunta alla copertura ed il carattere immediatamente successivo al termine di tale sequenza sarà analizzato seguendo la procedura descritta per il caso 2.

3 - Se né la sottosequenza diretta, né quella invertita complementata sono state individuate, allora c_0 non può essere in nessun modo coperto, e si prosegue analizzando il carattere c_1 seguendo la procedura descritta per il caso 1.

3 Minima lunghezza l : significato e quantificazione

Come detto in precedenza, per ottenere un confronto S-T significativo, cioè per individuare delle sequenze di copertura non prese a caso ma che siano statisticamente significative, bisogna utilizzare un $l > \log_d(n)$ $(n = |S|)$, con d la cardinalità dell'alfabeto.

Maggiore è l, maggiore è la probabilità che la copertura dia degli indizi su come T può essere derivato da una mutazione di S.

Di contro però, maggiore è l, minore è la sensibilità della copertura, questo perché non vengono considerate tutte le sequenze di lunghezza minore di l e, tra queste, anche quelle che potrebbero aver avuto il ruolo che si sta cercando.

Il valore di l deve quindi essere scelto tenendo chiaro in mente il rapporto attendibilità/sensibilità che si vuole ottenere

È stato osservato che, anche se il sottociclo del caso 2 può causare un notevole incremento del costo computazionale dell'algoritmo, tale effetto dannoso è presente solo quando il valore di l è prossimo a $\log_d(n)$.

Dei lunghi test, condotti in parallelo, utilizzando sia sequenze geniche generate in modo pseudocasuale che sequenze geniche reali, sia come S che come T, variando l tra diversi valori, hanno sperimentalmente confermato

che per ottenere una buona attendibilità dei dati della copertura, il valore di l deve in ogni caso essere maggiore della zona computazionalmente critica $\log_d(n)$.

Le immagini 2 e 3 (il grafo a destra è riscalato per una più chiara lettura) mostrano due esempi di questi test: l'area verde indica le iterazioni del *caso 1*, l'area blu quelle del *caso 2* e l'area rossa indica le iterazioni del sottociclo del *caso 2*, precedentemente descritto nella seconda parte dell'algoritmo BpMatch nella sezione 2.2.

Utilizzando $l = 20 \simeq 2\log_4(1000000) = 2\log_d(n)$ la copertura risulta essere non vuota solo confrontando pezzi di sequenze geniche reali.

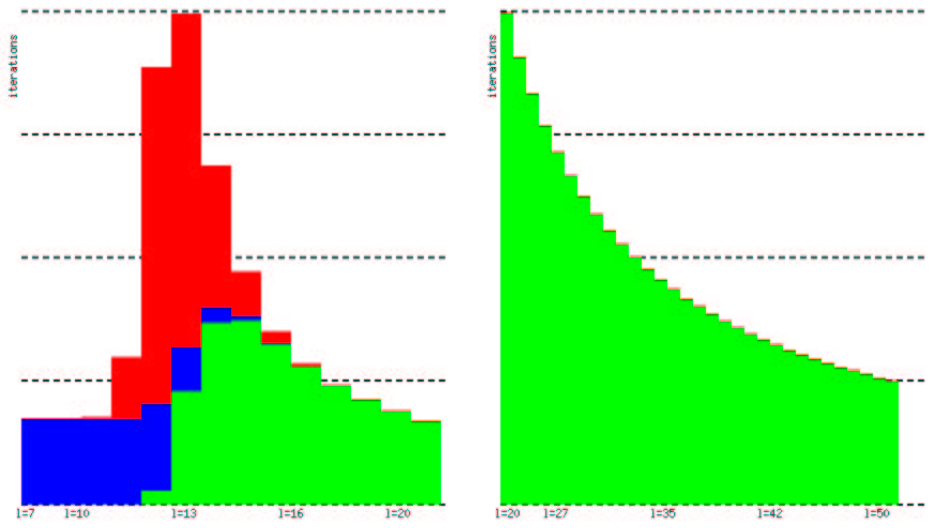

Figura 2: variazioni di l, S è una sequenza genica generata in modo pseudocasuale e T è un pezzo del primo cromosoma umano.

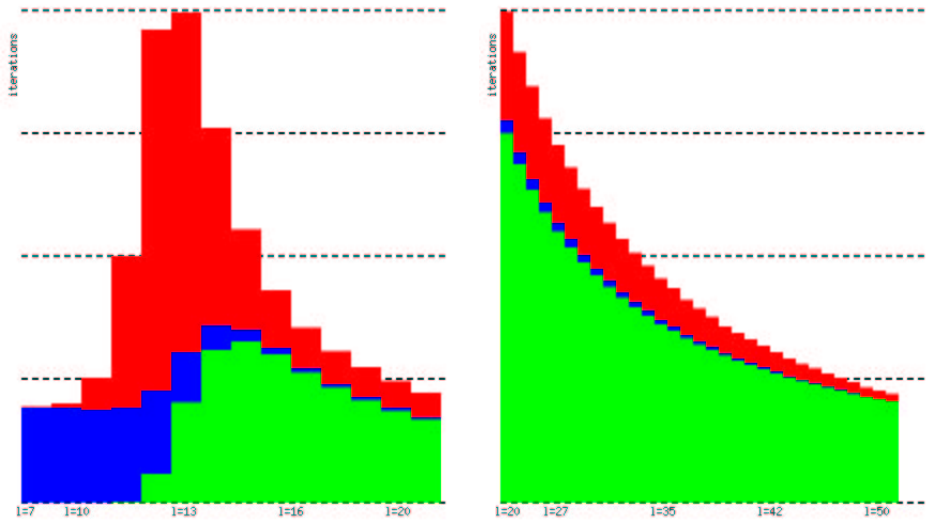

Figura 3: variazioni di l, sia S che T sono dei pezzi (non sovrapposti e lontani) del primo cromosoma umano.

4 Complessità in spazio ed in tempo

4.1 BpGraph

Generare un BpGraph da una sequenza S è il passo computazionalmente più pesante dell'algoritmo ma, d'altro canto, c'è il grosso vantaggio di poterlo eseguire una sola volta e poi utilizzare il grafo per coprire ogni possibile T.

Durante l'esecuzione dell'algoritmo di generazione del BpGraph, descritto a pagina 15, il ciclo principale [READ][ADD][WALK][CURSOR] viene ripetuto esattamente n volte ($n = |S|$) ed il sottociclo [WALK] viene ripetuto una volta per ogni cursore in quel momento presente sul grafo.

4.1.1 Dimensione del grafo e complessità in spazio della sua generazione

Un singolo nodo viene aggiunto al grafo solo durante [ADD] quindi BpGraph ha esattamente n nodi.

Un singolo arco diretto viene aggiunto al grafo solo durante [ADD] quindi BpGraph ha esattamente $n - 1$ archi diretti.

Gli archi laterali possono essere aggiunti al grafo solo durante il sottociclo [WALK] ma, dato che a seguito della creazione dell'arco un cursore viene rimosso, e che un singolo cursore viene aggiunto solo durante [CURSOR], un BpGraph ha al massimo $n - d$ archi laterali.

In aggiunta ai nodi e agli archi, diretti e laterali, un BpGraph ha un'array $starts[\,]$ di dimensione d (i punti di ingresso per i cursori).

Sia la dimensione di un BpGraph che la complessità in spazio della sua generazione sono dunque $\mathrm{O}(n\log(n) + d\log(n))$ e, con l'assuzione che l'arco di un grafo possa essere rappresentato in uno spazio costante fissato e che la cardinalità d dell'alfabeto è minore della lunghezza della stringa S, si possono considerare $\mathrm{O}(n)$.

4.1.2 Complessità in tempo della generazione di BpGraph

Ne caso pessimo durante l'i-esima iterazione possono esserci $i - 1$ cursori (un cursore può essere aggiunto durante ogni [CURSOR]) ma, nel caso medio, se S è una sequenza casuale di caratteri, ce ne sono solo $\log_d(i)$, in quanto questo è il valore stimato della massima lunghezza di una sottostringa di S, che termini nell'i-esimo carattere di S e che sia già presente in S prima del suo i-esimo carattere, condizione necessaria affinché vi siano $\log_d(i)$ cursori.

Il numero medio di cursori durante l'iterazione i-esima risulta essere $\log_4(i)$ anche per le reali sequenze geniche, ma sono state riscontrate brevi (seppure di forte entità) fluttuazioni mentre l'algoritmo si trovava ad analizzare sequenze geniche ripetitive (come ad esempio i mini ed i microsatelliti, SINES e LINES, geni in tandem, ecc.).

Durante ognuna delle n iterazioni principali, il sottociclo [WALK] deve trovare (se esiste) un arco da percorrere per ogni cursore. Il tempo richiesto per individuare l'arco è logaritmico rispetto al numero di archi uscenti dal nodo su cui il cursore è posto (dato che gli archi laterali possono venire ordinati secondo il valore del loro campo from).

C'è al più un solo arco diretto uscente per ogni nodo. Il numero di archi laterali uscenti da un nodo durante l'iterazione i-esima è nel caso pessimo meno di i, ma, nel caso medio, se S è una sequenza casuale di caratteri, è osservato essere, per l'm-esimo nodo, prossimo a $\log_d(i/m)$.

Il numero medio di archi laterali uscenti durante l'i-esima iterazione dal nodo m-esimo si osserva essere $\log_4(i/m)$ anche per le reali sequenze geniche, ma pure in questo caso, come già osservato prima per il numero dei cursori, vi sono delle fluttuazioni quando l'algoritmo si trovava ad analizzare sequenze geniche ripetitive.

Nel caso pessimo, la generazione del grafo richiede n iterazioni principali e, durante l'i-esima, i iterazioni del sottociclo [WALK], ognuna delle quali richiede un tempo $O(\log(i))$. Quindi la complessità in tempo nel caso pessimo è $O(n^2 \log(n))$.

Nel caso medio, la generazione del grafo richiede n iterazioni principali e, durante l'i-esima, $\log_d(i)$ iterazioni del sottociclo [WALK], ognuna delle quali

richiede un tempo $O(\log(\log_d(i)))$. Quindi la complessità in tempo nel caso medio è $O(n \log_d(n) \log(\log_d(n)))$.

4.1.3 Complessità in tempo dell'uso di BpGraph

La lunghezza del percorso che un cursore deve coprire su un BpGraph per poter riconoscere o rifiutare una sequenza di lunghezza n è proprio n (di meno se la rifiuta prima di analizzarne tutti i caratteri).

Nel primo passo deve essere individuato quale punto d'ingresso, tra quelli dell'array $starts[\ ]$ di dimensione d, usare per posizionare il cursore. Ogni iterazione successiva richiede invece che venga individuato un arco tra quelli uscenti dal nodo in cui è posto il cursore (la quantità degli archi uscenti è già stata analizzata in 4.1.1).

Nel caso pessimo, il tempo necessario per riconoscere una sequenza di lunghezza u è $O(u \log(n)+log(d))$. Nel caso medio invece è $O(u \log(\log_d(n))+ log(d))$ e, se si considera come costante la dimensione dell'alfabeto, $O(u \log(\log(n)))$.

4.2 Calcolo della massima copertura di T

Il calcolo della massima copertura di T richiede di analizzare u ($u = |T|$) caratteri, alternando tra il $caso1$ ed il $caso2$ descritti in precedenza. Sia il tempo richiesto per esaminare un caso che il numero di caratteri di cui si può procedere lungo T al termine dell'iterazione dipendono da quale delle seguenti condizioni si viene a verificare:

Nel caso 1:

- in caso di fallimento: il tempo è quello richiesto per riconoscere una se-

quenza lunga m ($m <= l$), si avanza di $l - m + 1$ caratteri.
- se si individua una sottosequenza: il tempo è quello richiesto per riconoscere una sequenza di lunghezza l ed una seconda di lunghezza m ($m > l$), si avanza di $m - 1$ caratteri.

Nel caso 2:
- in caso di fallimento: nel caso pessimo si deve riconoscere $l + 1$ sequenze di lunghezza l, si avanza di un solo carattere.
- se si individua una sottosequenza: il tempo è quello richiesto per riconoscere una sequenza di lunghezza m ($m > l$), si avanza di $m-1$ caratteri.
- se si individua una sottosequenza sovrapposta a quella precedente: la situazione nel caso pessimo può essere come quella nel caso di fallimento.

Nel caso pessimo (se l'alfabeto è di dimensione costante) il tempo richiesto per calcolare la copertura è $O(l^2 u \log(n))$. Nel caso medio (se l'alfabeto è di dimensione costante), utilizzando $l = 2\log_d(n)$, il tempo richiesto per calcolare la copertura è $O(u \log(\log(n)))$.

5 Esempi

I due esempi seguenti mostrano graficamente le computazioni.

L'alfabeto utilizzato, per semplificare, è composto di tre soli caratteri (A, B e C) ed ogni carattere ha sé stesso come complemento.

L'utilizzo di porzioni di codice genetico reale è finalizzato unicamente a testare se l'algoritmo varia la sua complessità in tempo del caso medio,

precedentemente stimata per le sequenze casuali di caratteri.

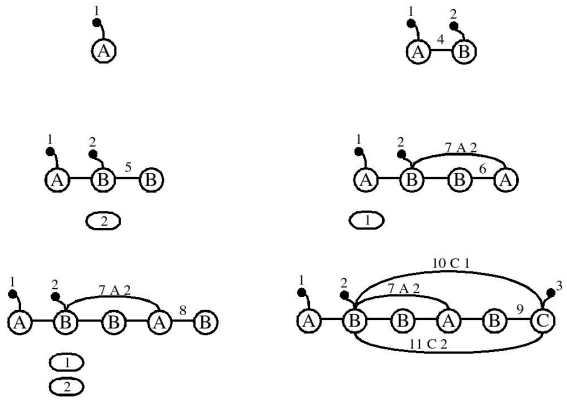

Figura 4: esempio di generazione di un BpGraph.

5.1 Esempio di generazione di un BpGraph

Le immagini 4, 5 e 6 mostrano graficamente l'esecuzione dell'algoritmo di generazione di un BpGraph descritto a pagina 15, con ABBABCACBBAC come sequenza di input (S).

I cerchi sono nodi di BpGraph, etichettati con il loro carattere, i punti neri, etichettati 1, 2 e 3, sono i tre punti d'ingresso del grafo. Ogni arco ha un'identificatore, ma, per la chiarezza dello schema, sugli archi *diretti* è indicato solo nel passo in cui viene aggiunto. Le etichette degli archi *laterali* sono: identificatore, etichetta del nodo a cui l'arco punta ed il valore del campo $from$. Gli ovali posizionati sotto i nodi rappresentano dei cursori e sono etichettati con il valore del loro campo $from$.

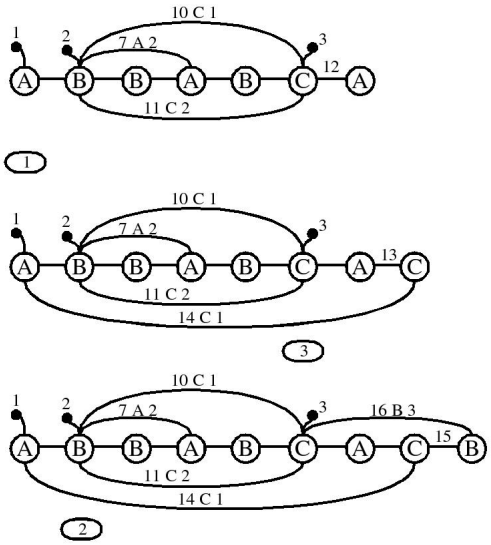

Figura 5: esempio di generazione di un BpGraph.

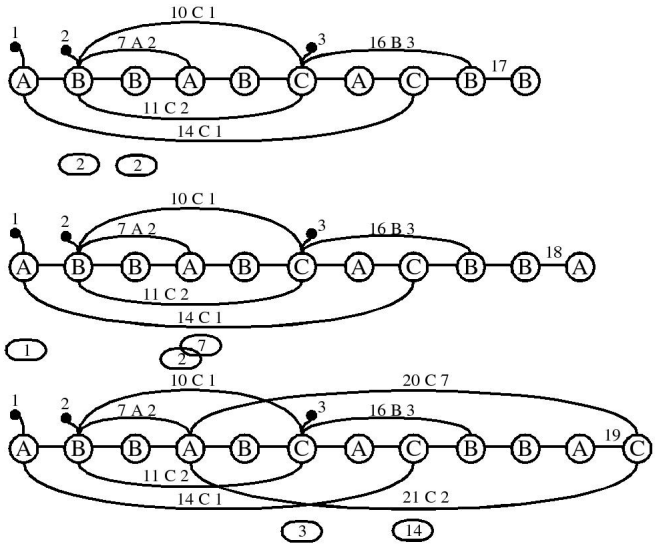

Figura 6: esempio di generazione di un BpGraph.

Si ricorda che un cursore può percorrere un arco *laterale* solo se il suo campo $from$ è uguale al campo $from$ dell'arco.

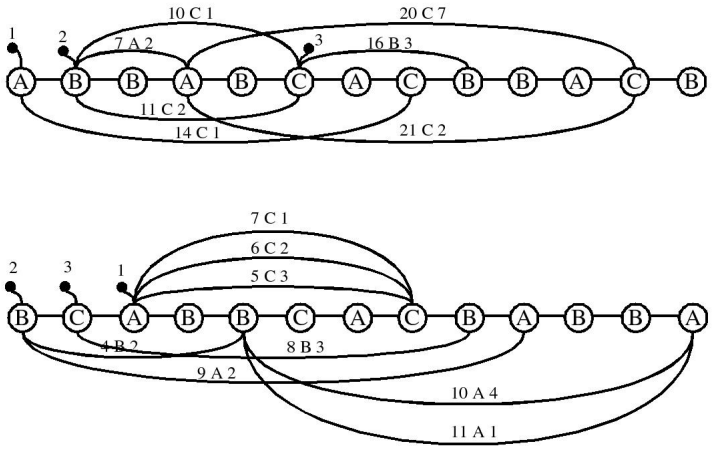

Figura 7: i BpGraph G e G', S= ABBABCACBBACB .

5.2 Esempio di calcolo della copertura massima

L'immagine 8 mostra graficamente un'esecuzione dell'algoritmo BpMatch per il calcolo della copertura massima, descritto in precedenza nella sezione 2.2.

La sequenza di partenza è S= ABBABCACBBACB e quella obiettivo da coprire è T= BAABCAAABBCBACBBACBAABCBCBCACBABCAC

.

L'immagine 7 mostra i due BpGraph: il primo, G, riconosce le sottosequenze dirette di S, mente il secondo, G', riconosce quelle invertite e com-

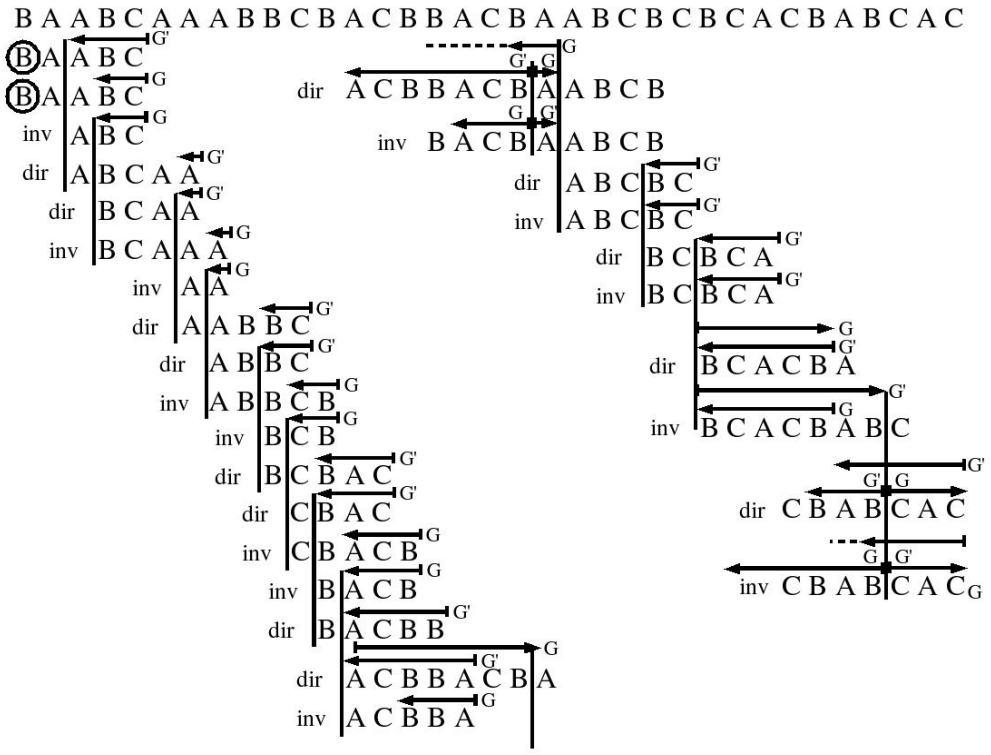

Figura 8: esempio semplificato di calcolo della copertura massima.

plementate. In questo esempio semplificato l'operazione di complemento è l'identità, quindi G' riconosce anche le sottosequenze invertite di S.

La lunghezza minima per una sottosequenza della copertura è $l = 5$.

Le etichette *dir* e *inv* significano ricerca delle sottosequenze dirette e ricerca delle sottosequenze invertite complementate, le frecce sopra i caratteri rappresentano i percorsi che sono stati individuati con successo nei grafi BpMatch.

6 Numero di cursori durante la generazione di un BpGraph

L'algoritmo della generazione dei BpGraph descritto a pagina 15 utilizza i cursori per generare gli archi laterali e ad ogni passo computazionale un nuovo cursore può essere aggiunto mentre anche più di uno possono venire rimossi.

Il numero di cursori è un parametro fondamentale per l'analisi di complessità dell'algoritmo di generazione, come descritto nella sezione 4.

Le immagini dalla 9 alla 18 mostrano il numero di cursori durante l'esecuzione dell'algoritmo, utilizzando come sequenze di partenza S delle stringhe di caratteri generate in modo pseudocasuale. La linea rossa è il numero di cursori stimato dall'analisi dell'algoritmo: $y = \log_4 x$.

Le immagini dalla 19 alla 27 mostrano il numero di cursori durante l'esecuzione dell'algoritmo, utilizzando questa volta come sequenze di partenza S spezzoni del primo cromosoma umano, della lunghezza di 2^{20} coppie di basi. La linea rossa è il numero di cursori stimato dall'analisi dell'algoritmo: $y = \log_4 x$.

Le osservazioni riguardanti questi grafici possono essere trovate nella sezione 4.

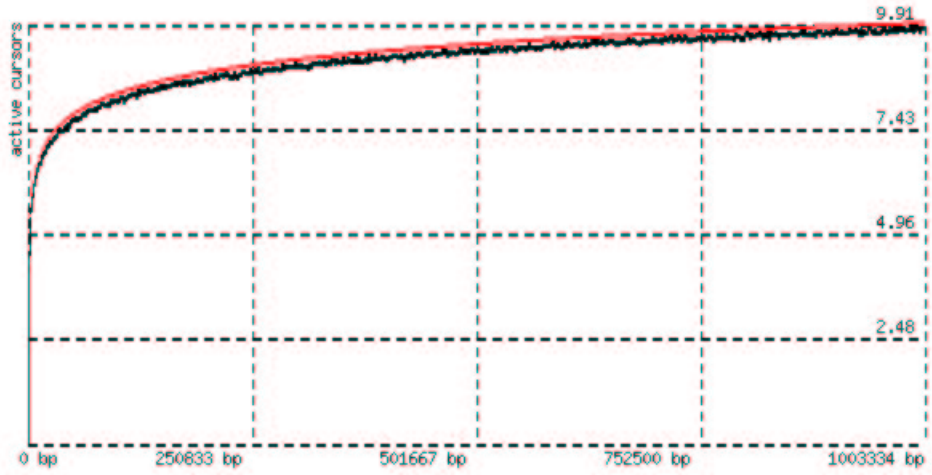

Figura 9: numero di cursori durante la generazione di un BpGraph, utilizzando sequenze di caratteri generate in modo pseudocasuale.

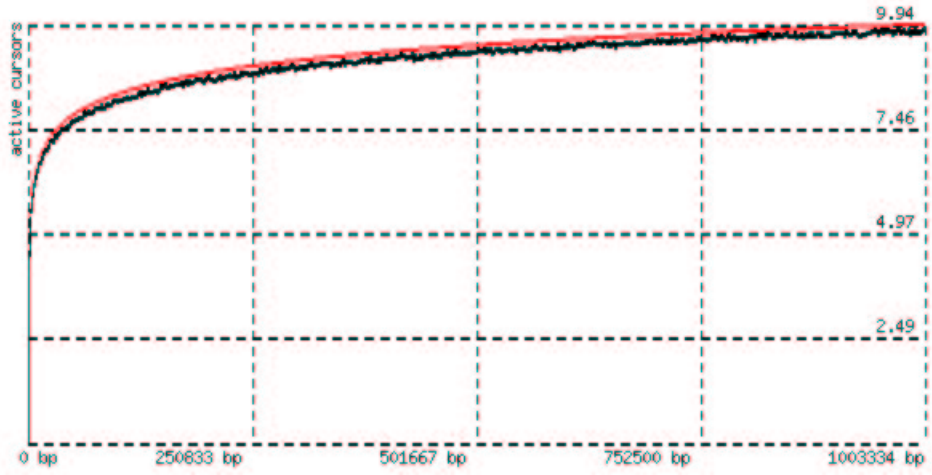

Figura 10: numero di cursori durante la generazione di un BpGraph, utilizzando sequenze di caratteri generate in modo pseudocasuale.

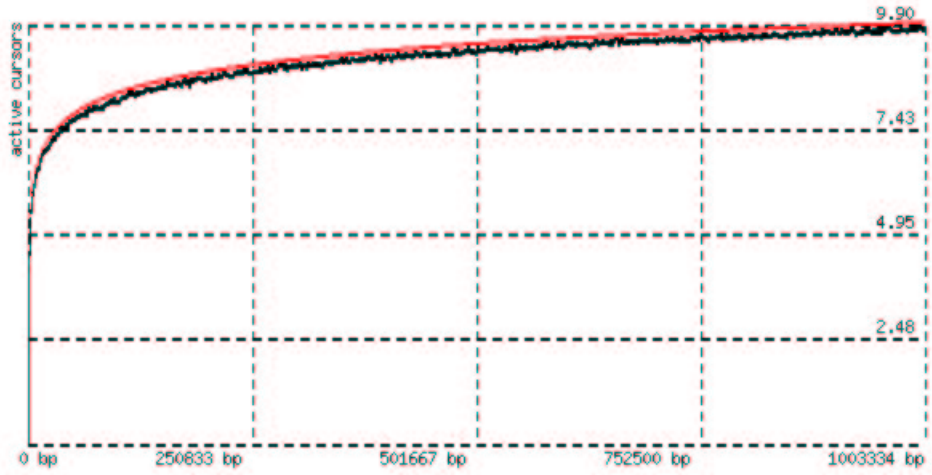

Figura 11: numero di cursori durante la generazione di un BpGraph, utilizzando sequenze di caratteri generate in modo pseudocasuale.

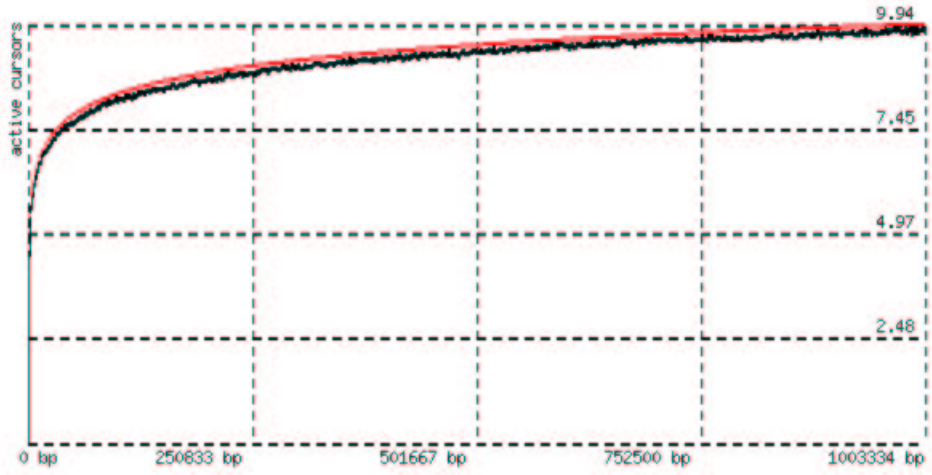

Figura 12: numero di cursori durante la generazione di un BpGraph, utilizzando sequenze di caratteri generate in modo pseudocasuale.

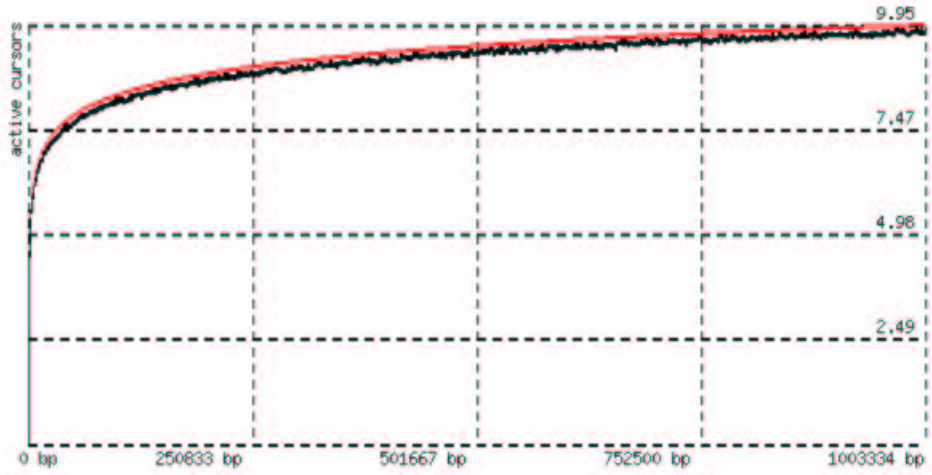

Figura 13: numero di cursori durante la generazione di un BpGraph, utilizzando sequenze di caratteri generate in modo pseudocasuale.

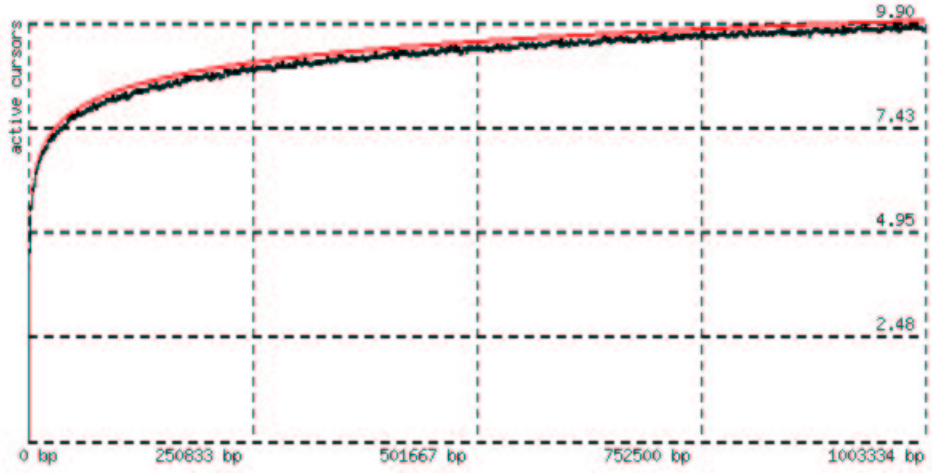

Figura 14: numero di cursori durante la generazione di un BpGraph, utilizzando sequenze di caratteri generate in modo pseudocasuale.

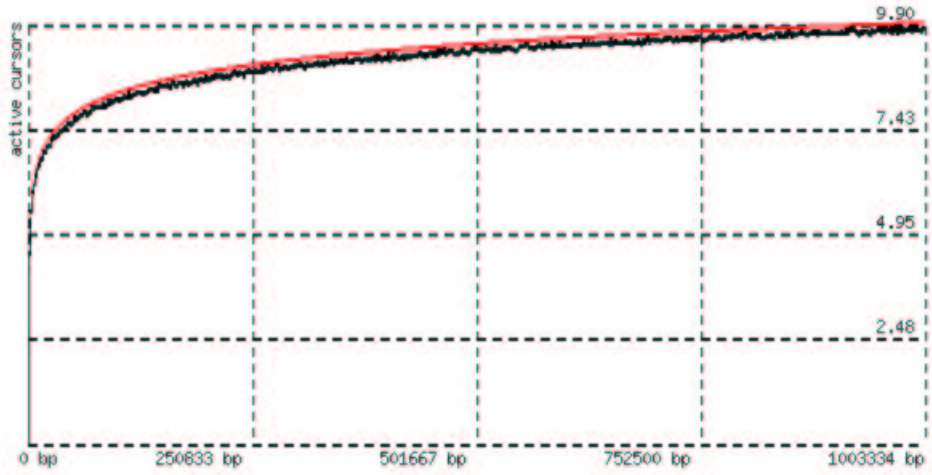

Figura 15: numero di cursori durante la generazione di un BpGraph, utilizzando sequenze di caratteri generate in modo pseudocasuale.

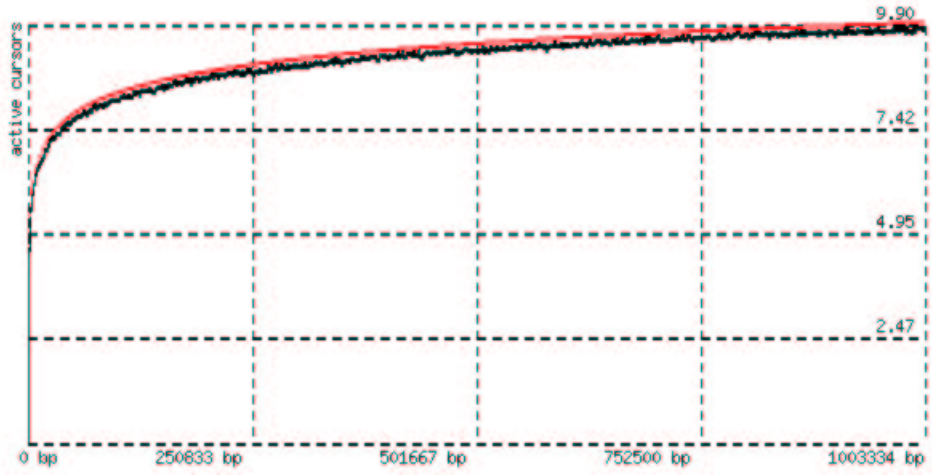

Figura 16: numero di cursori durante la generazione di un BpGraph, utilizzando sequenze di caratteri generate in modo pseudocasuale.

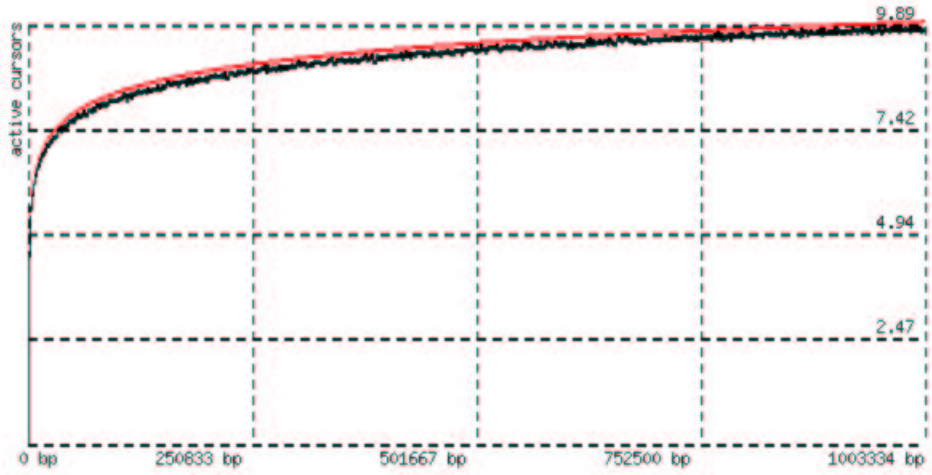

Figura 17: numero di cursori durante la generazione di un BpGraph, utilizzando sequenze di caratteri generate in modo pseudocasuale.

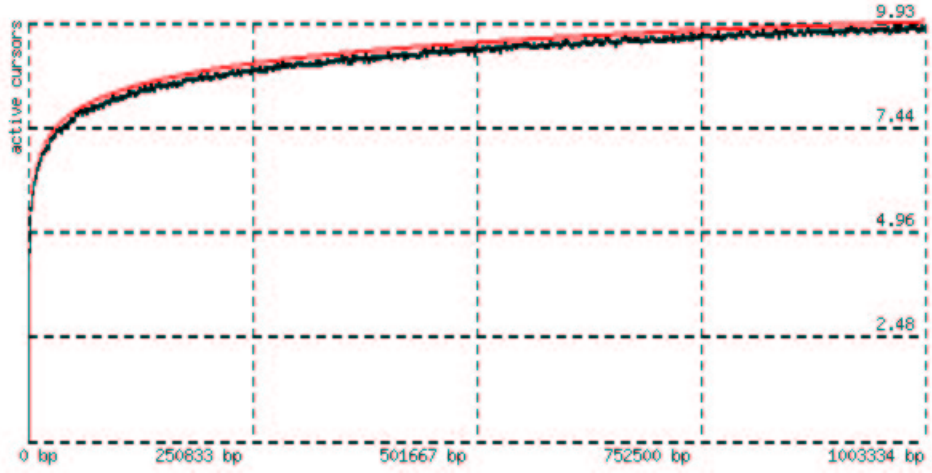

Figura 18: numero di cursori durante la generazione di un BpGraph, utilizzando sequenze di caratteri generate in modo pseudocasuale.

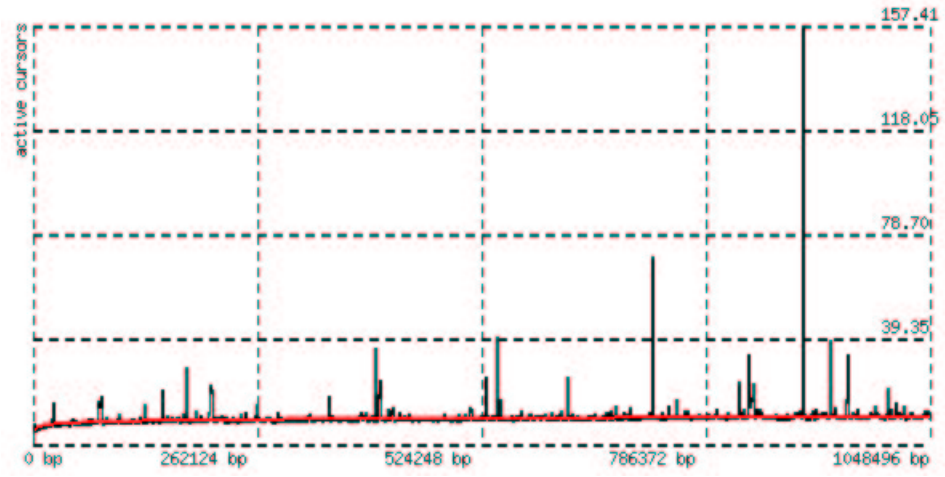

Figura 19: numero di cursori durante la generazione di un BpGraph, utilizzando spezzoni del primo cromosoma umano.

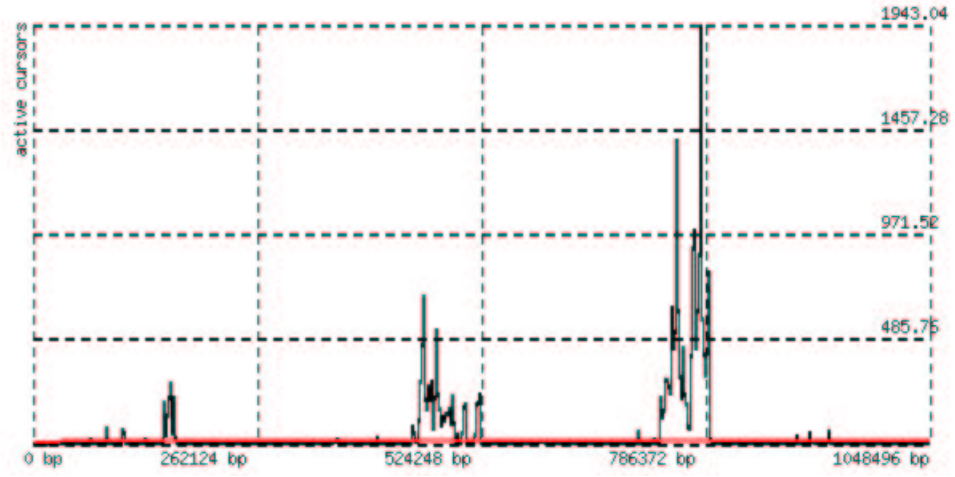

Figura 20: numero di cursori durante la generazione di un BpGraph, utilizzando spezzoni del primo cromosoma umano.

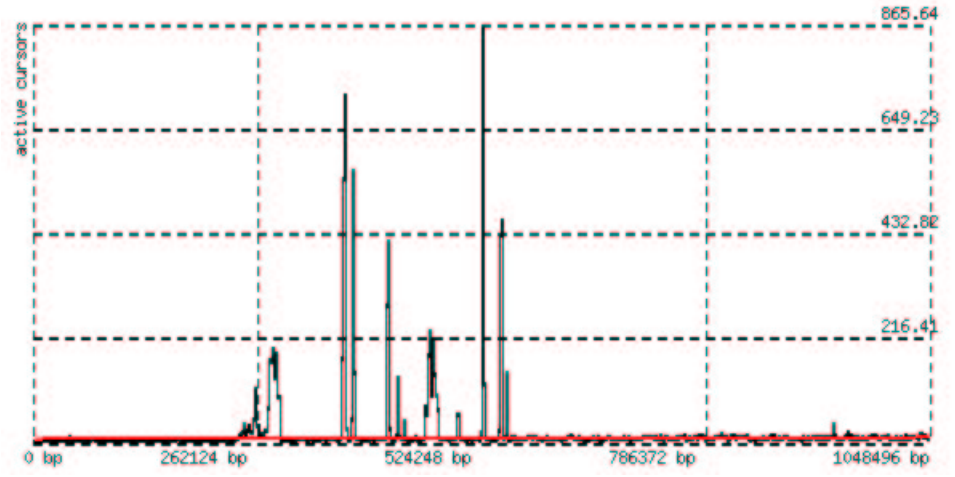

Figura 21: numero di cursori durante la generazione di un BpGraph, utilizzando spezzoni del primo cromosoma umano.

Figura 22: numero di cursori durante la generazione di un BpGraph, utilizzando spezzoni del primo cromosoma umano.

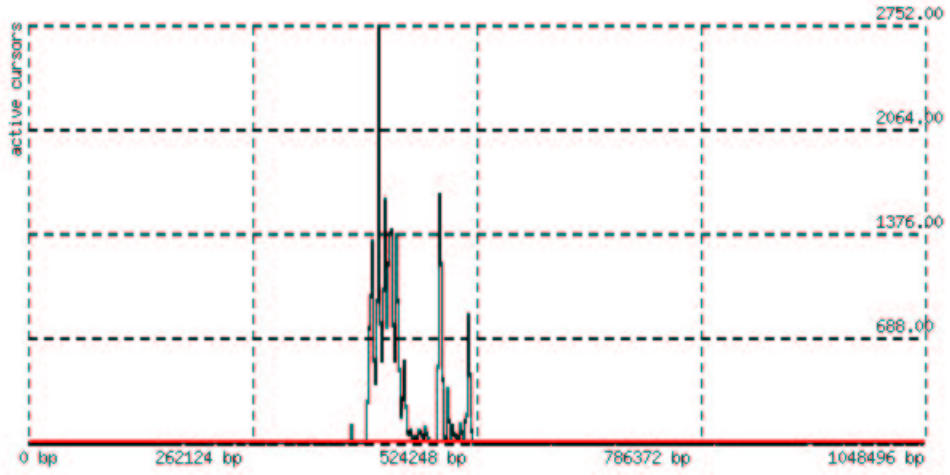

Figura 23: numero di cursori durante la generazione di un BpGraph, utilizzando spezzoni del primo cromosoma umano.

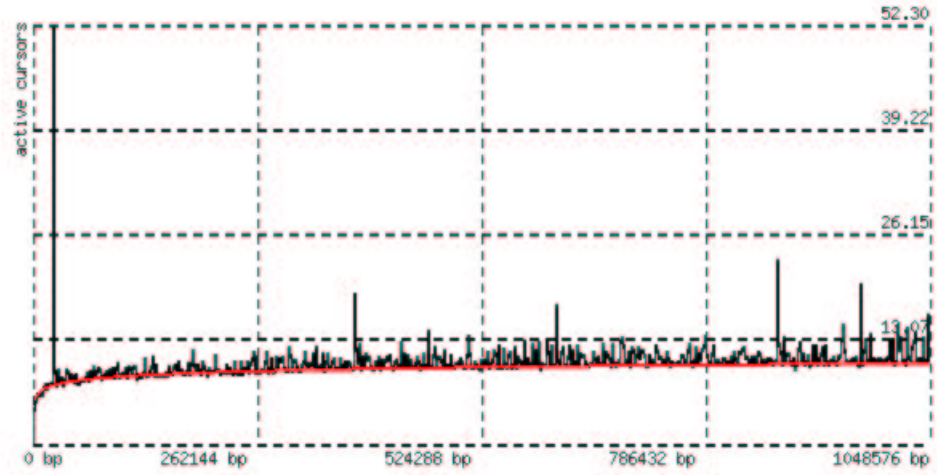

Figura 24: numero di cursori durante la generazione di un BpGraph, utilizzando spezzoni del primo cromosoma umano.

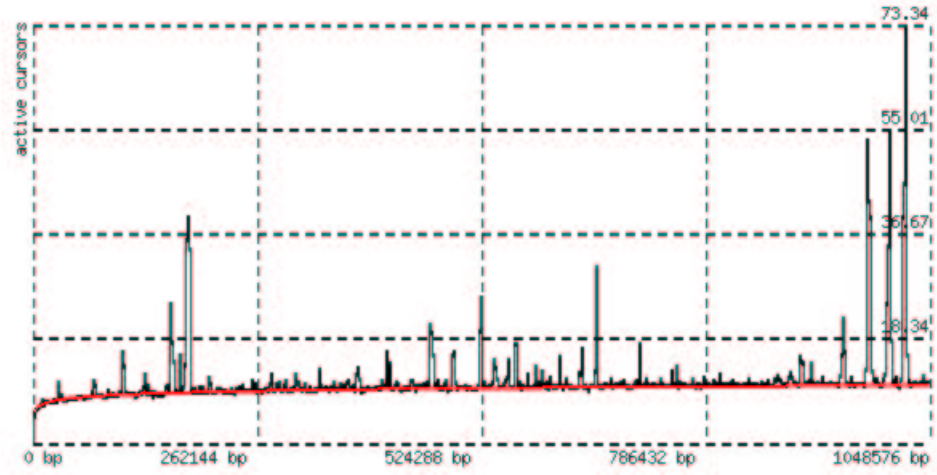

Figura 25: numero di cursori durante la generazione di un BpGraph, utilizzando spezzoni del primo cromosoma umano.

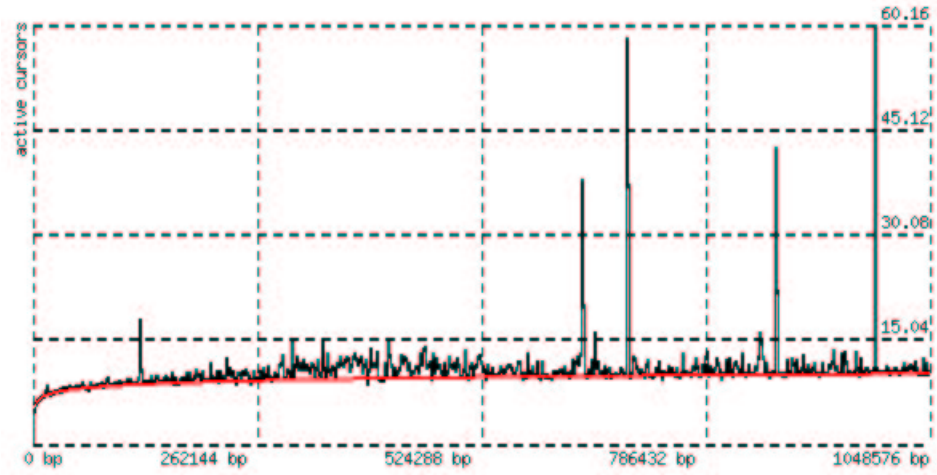

Figura 26: numero di cursori durante la generazione di un BpGraph, utilizzando spezzoni del primo cromosoma umano.

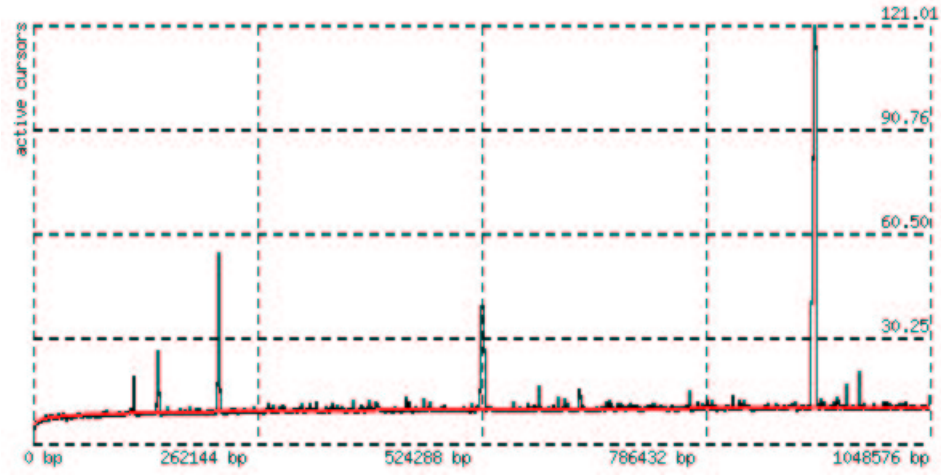

Figura 27: numero di cursori durante la generazione di un BpGraph, utilizzando spezzoni del primo cromosoma umano.

7 Distribuzione degli archi laterali tra i nodi in un BpGraph

Dall'algoritmo di generazione dei BpGraph descritto a pagina 15 deriva immediatamente che il numero di archi laterali è limitato superiormente dalla lunghezza della sequenza di partenza (n). Questo è vero in quanto un arco laterale può essere generato solo da un cursore, tale cursore viene rimosso immediatamente dopo la creazione dell'arco laterale e durante la generazione dell'albero al più un solo cursore può venire generato ad ognuna delle n iterazioni principali.

Il numero esatto di archi laterali è uguale a n, meno il numero di caratteri dell'alfabeto (4 nel caso genetico), meno il numero di cursori ancora presenti al termine dell'ultima iterazione dell'algoritmo di generazione.

Le immagini dalla 28 alla 37 mostrano la distribuzione degli archi laterali tra i nodi di un BpGraph generato utilizzando come S delle sequenze di caratteri generate in modo pseudocasuale. I nodi sono ordinati, da sinistra a destra, dal primo creato fino all'ultimo. I colori visualizzano *quando* gli archi laterali sono stati generati: archi creati nello stesso momento hanno lo stesso colore.

Le immagini dalla 38 alla 47 mostrano la distribuzione degli archi laterali tra i nodi di un BpGraph generato utilizzando questa volta come S spezzoni del primo cromosoma umano, della lunghezza di 2^{20} coppie di basi. I nodi sono ordinati, da sinistra a destra, dal primo creato fino all'ultimo. I colori visualizzano *quando* gli archi laterali sono stati generati: archi creati nello stesso momento hanno lo stesso colore.

Le osservazioni riguardanti questi grafici possono essere trovate nella sezione 4.

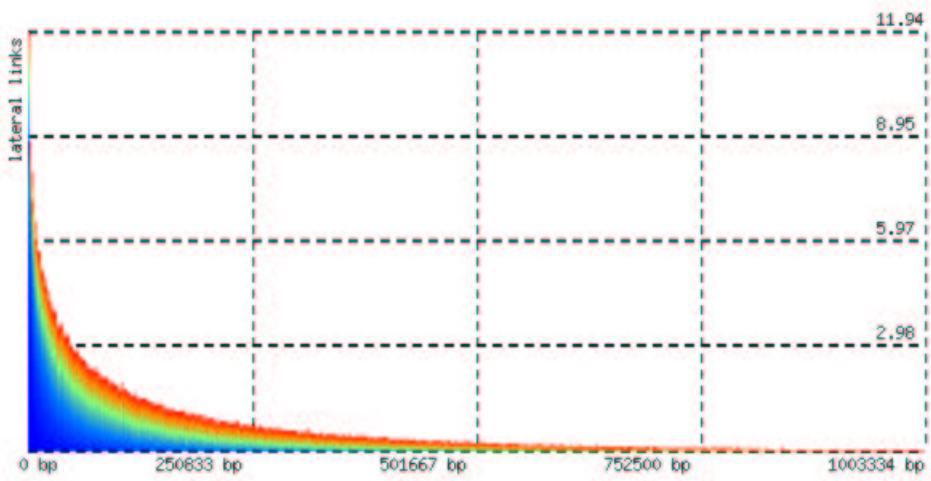

Figura 28: distribuzione degli archi laterali tra i nodi in un BpGraph, usando sequenze di caratteri generate in modo pseudocasuale.

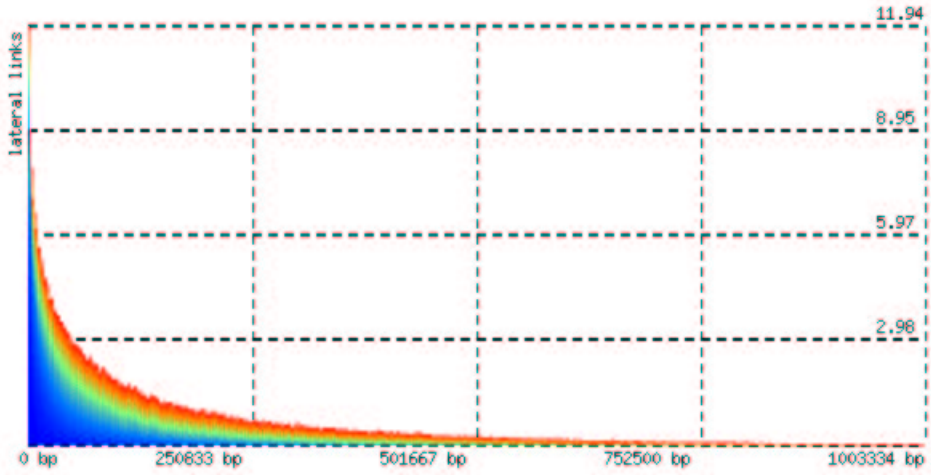

Figura 29: distribuzione degli archi laterali tra i nodi in un BpGraph, usando sequenze di caratteri generate in modo pseudocasuale.

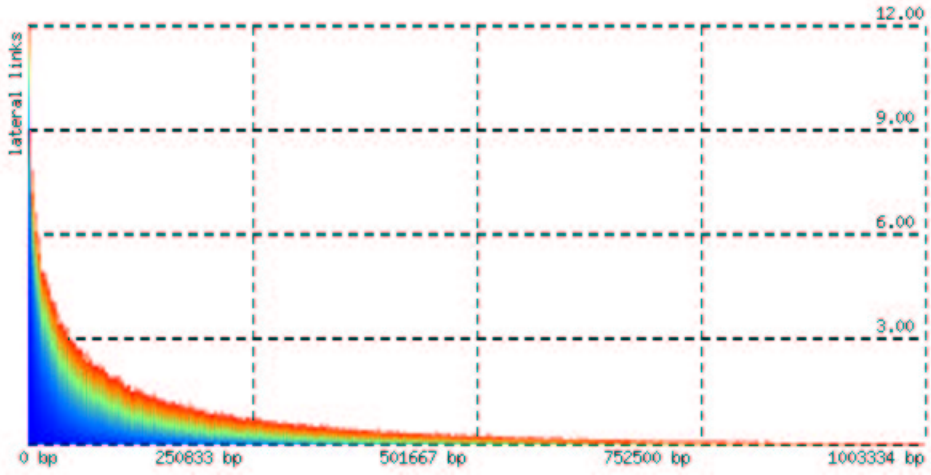

Figura 30: distribuzione degli archi laterali tra i nodi in un BpGraph, usando sequenze di caratteri generate in modo pseudocasuale.

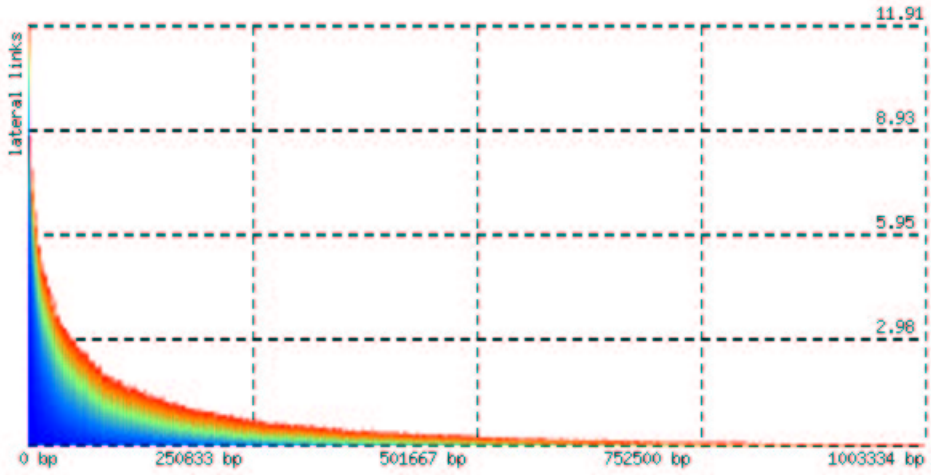

Figura 31: distribuzione degli archi laterali tra i nodi in un BpGraph, usando sequenze di caratteri generate in modo pseudocasuale.

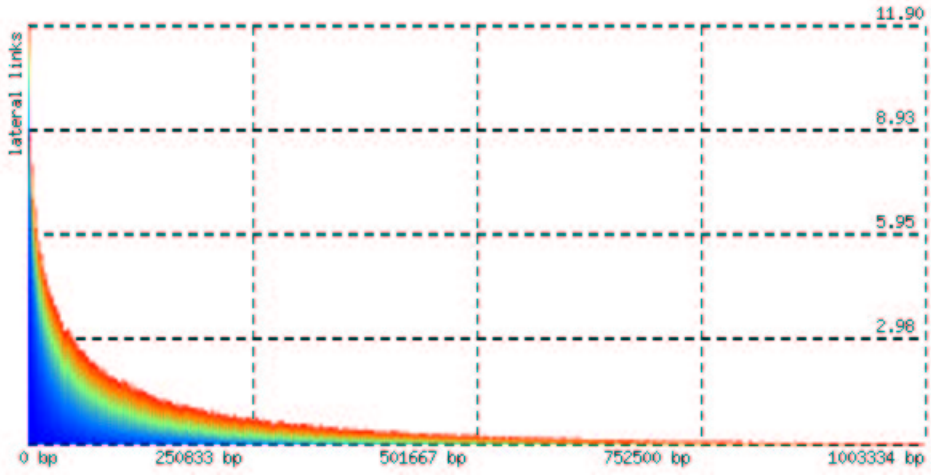

Figura 32: distribuzione degli archi laterali tra i nodi in un BpGraph, usando sequenze di caratteri generate in modo pseudocasuale.

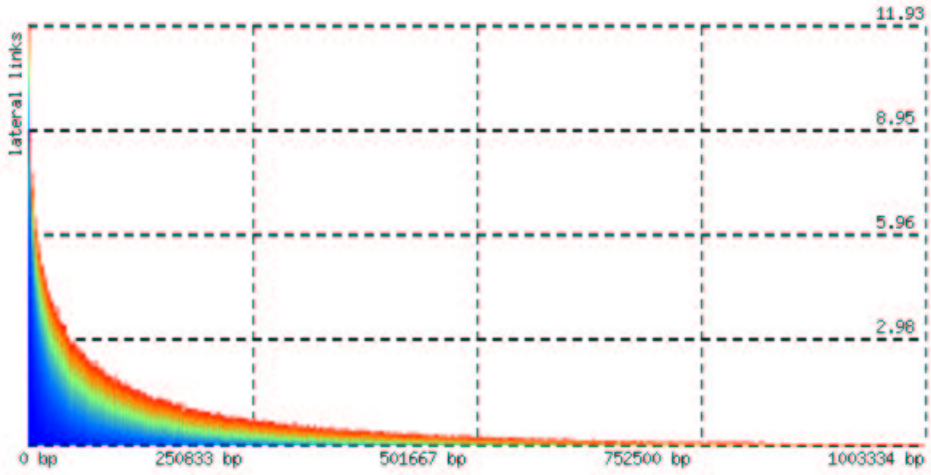

Figura 33: distribuzione degli archi laterali tra i nodi in un BpGraph, usando sequenze di caratteri generate in modo pseudocasuale.

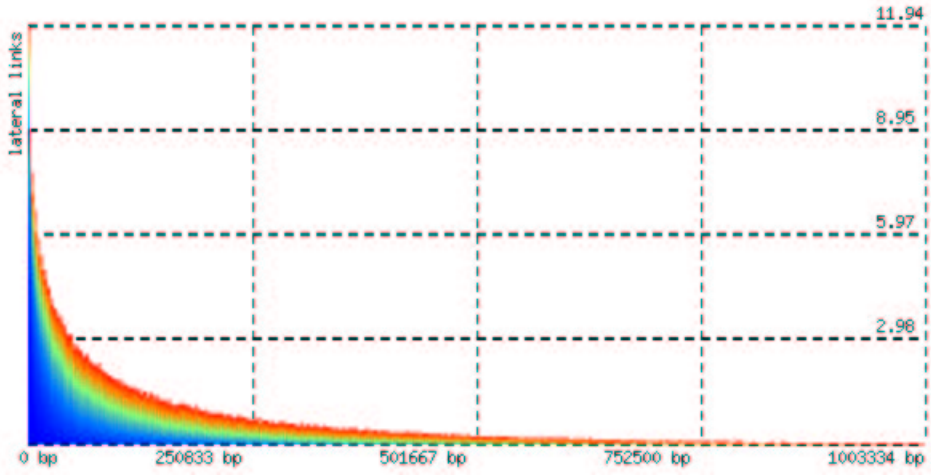

Figura 34: distribuzione degli archi laterali tra i nodi in un BpGraph, usando sequenze di caratteri generate in modo pseudocasuale.

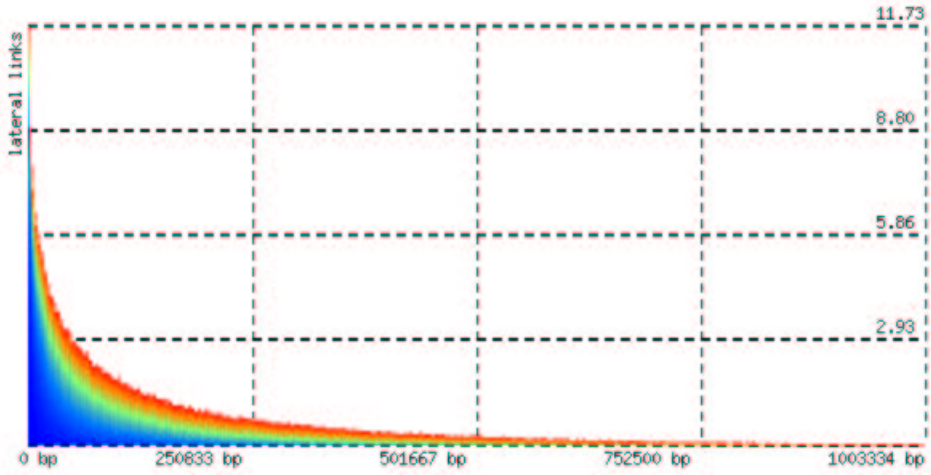

Figura 35: distribuzione degli archi laterali tra i nodi in un BpGraph, usando sequenze di caratteri generate in modo pseudocasuale.

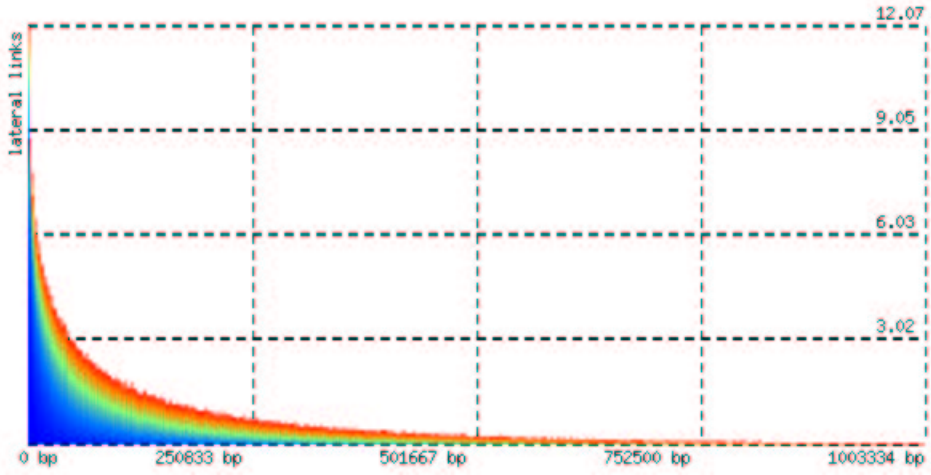

Figura 36: distribuzione degli archi laterali tra i nodi in un BpGraph, usando sequenze di caratteri generate in modo pseudocasuale.

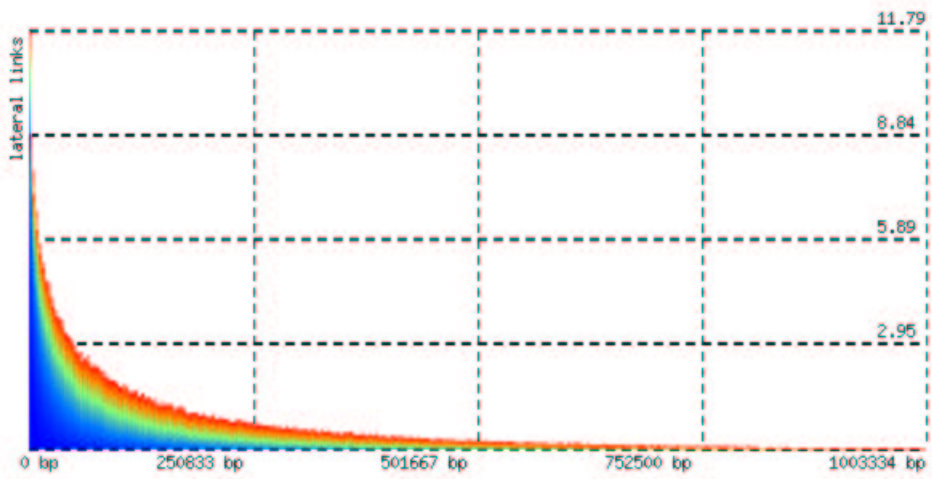

Figura 37: distribuzione degli archi laterali tra i nodi in un BpGraph, usando sequenze di caratteri generate in modo pseudocasuale.

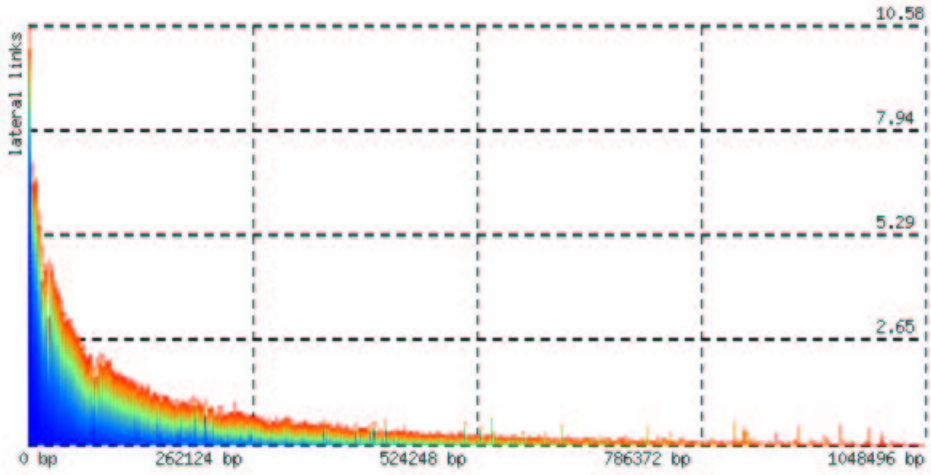

Figura 38: distribuzione degli archi laterali tra i nodi in un BpGraph, utilizzando spezzoni del primo cromosoma umano.

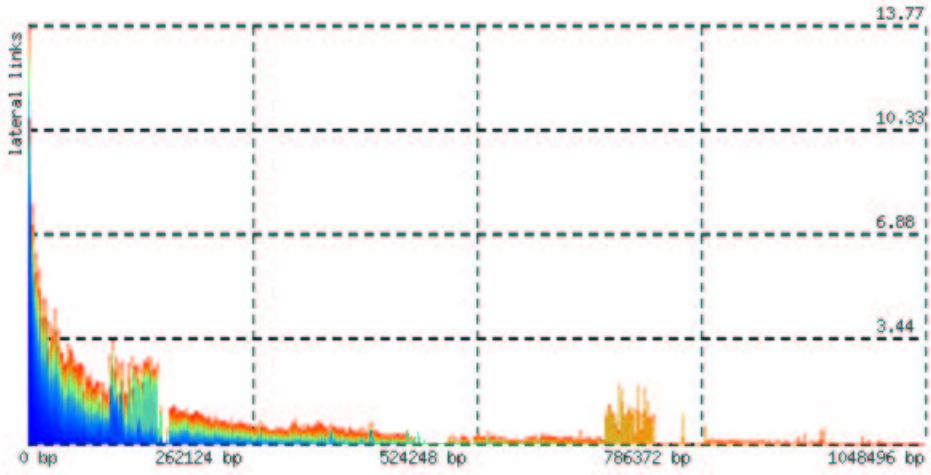

Figura 39: distribuzione degli archi laterali tra i nodi in un BpGraph, utilizzando spezzoni del primo cromosoma umano.

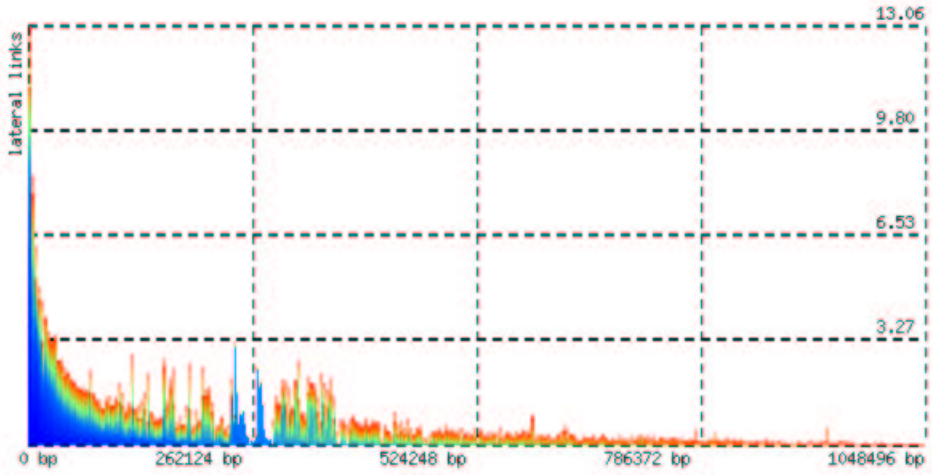

Figura 40: distribuzione degli archi laterali tra i nodi in un BpGraph, utilizzando spezzoni del primo cromosoma umano.

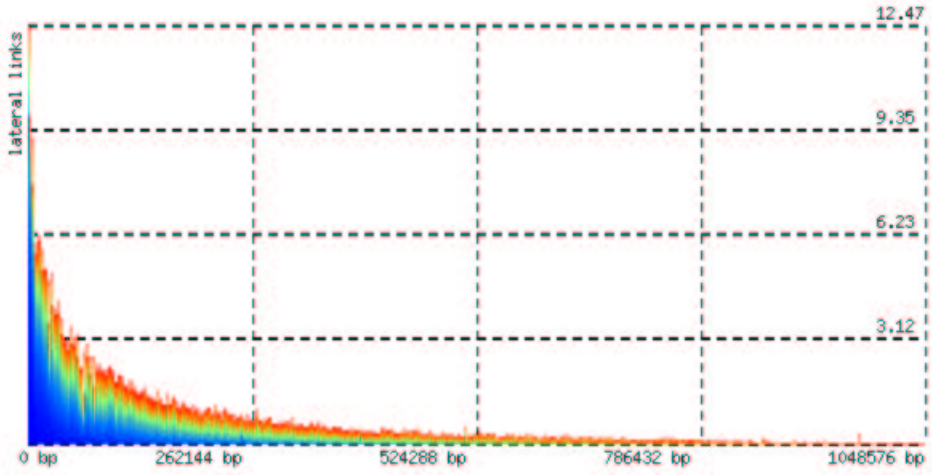

Figura 41: distribuzione degli archi laterali tra i nodi in un BpGraph, utilizzando spezzoni del primo cromosoma umano.

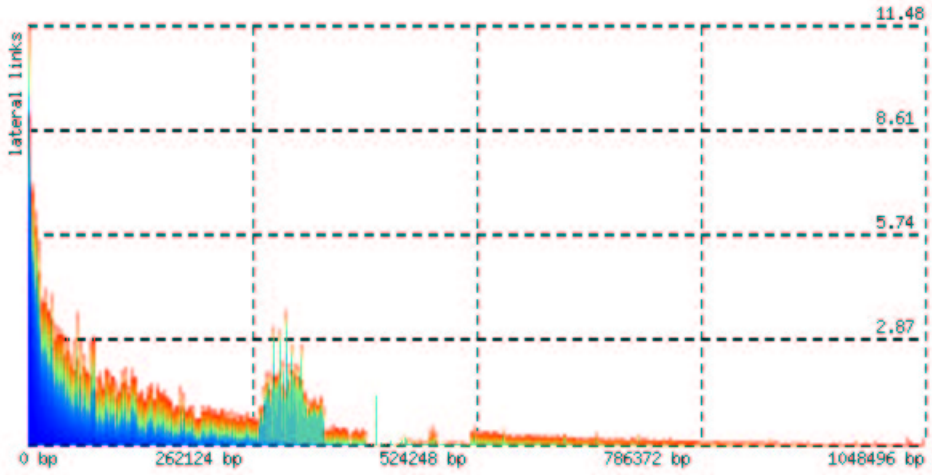

Figura 42: distribuzione degli archi laterali tra i nodi in un BpGraph, utilizzando spezzoni del primo cromosoma umano.

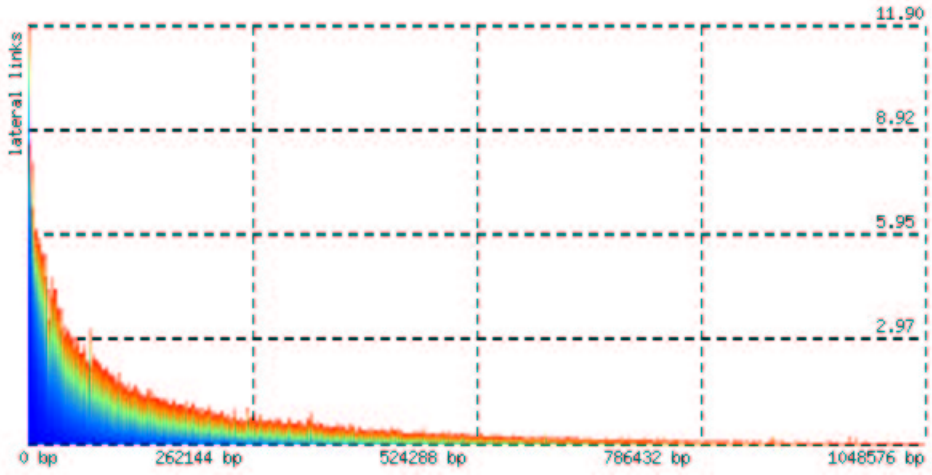

Figura 43: distribuzione degli archi laterali tra i nodi in un BpGraph, utilizzando spezzoni del primo cromosoma umano.

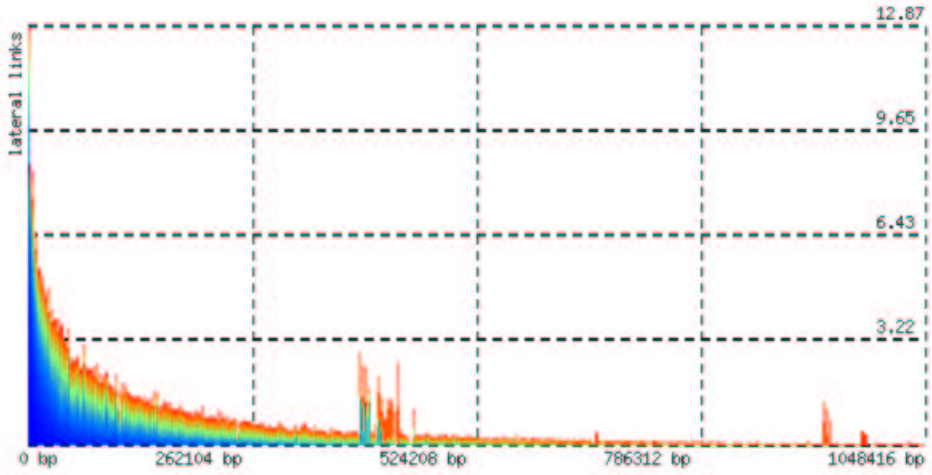

Figura 44: distribuzione degli archi laterali tra i nodi in un BpGraph, utilizzando spezzoni del primo cromosoma umano.

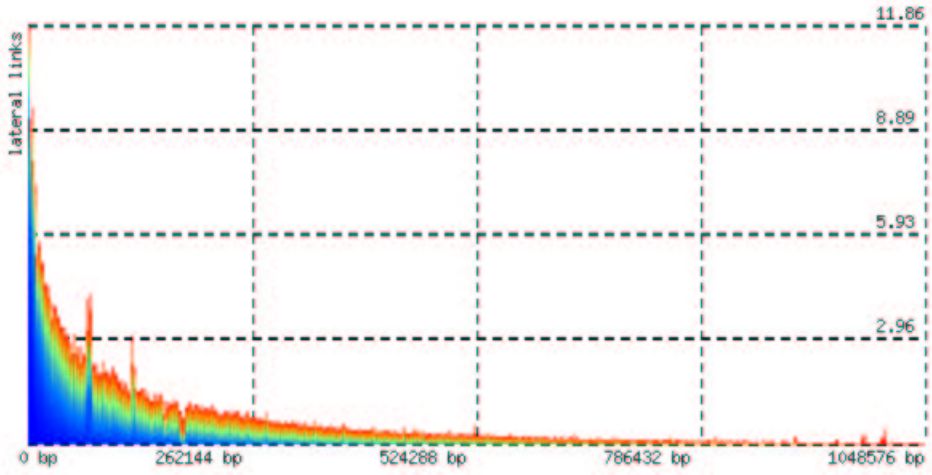

Figura 45: distribuzione degli archi laterali tra i nodi in un BpGraph, utilizzando spezzoni del primo cromosoma umano.

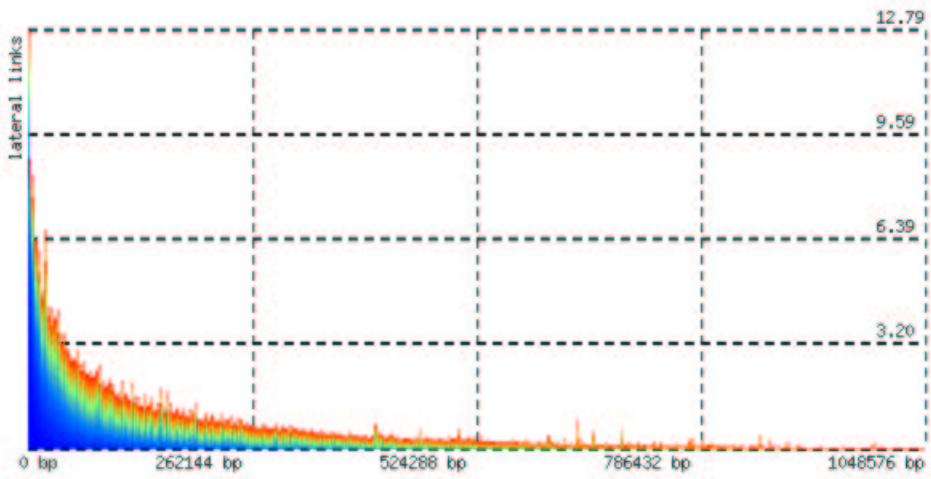

Figura 46: distribuzione degli archi laterali tra i nodi in un BpGraph, utilizzando spezzoni del primo cromosoma umano.

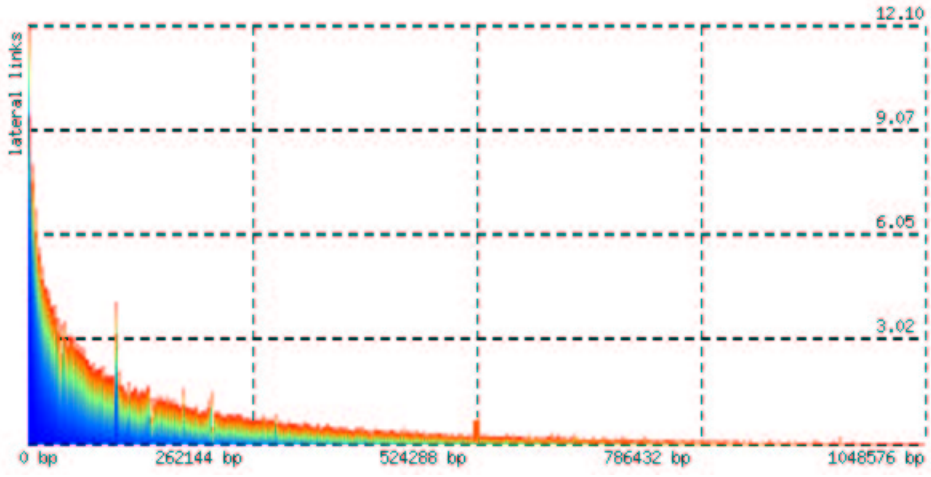

Figura 47: distribuzione degli archi laterali tra i nodi in un BpGraph, utilizzando spezzoni del primo cromosoma umano.

8 Lavori futuri

Una versione dell'algoritmo di BpMatch utilizzando dei prefix trees modificati al posto dei BpGraphs è in analisi.

Riferimenti bibliografici

[1] J.S. Varré, J.P. Delahaye and E. Rivals. *Transformation distances: A family of dissimilarity measures based on movements of segments.* Bioinformatics 15, 1999, Pp. 194-202.

[2] N. Pisanti, R. Marangoni, P. Ferragina, A. Frangioni, A. Savona, C. Pisanelli and F. Luccio. *PaTre: A Method for Parology Trees Construction.* Journal of Computational Biology Volume 10, Number 5, 2003, Pp. 791-802.

[3] F.H. Ruddle. *Vertebrate genome evolution – the decade ahead.* Genomics 46, 1997, Pp. 171-173.

[4] A.M. Lesk. *Introduction to Bioinformatics.* Oxford University Press Inc., 2002.

[5] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, J.D. Watson. *Biologia molecolare della cellula.* Zanichelli, 1995.

[6] R.B. Russel. *Domain insertion.* Protein Eng. 7, 1994, Pp. 1407-1410.

[7] W.H. Li, D. Graur. *Fundamentals of Molecular Evolution.* Sinauer Associates Inc., 1991.

[8] M.S. Waterman. *Introduction to Computational Biology.* Chapman and Hall, 1995.

[9] J.C.D. Hinton. *The* Escherichia coli *genome sequence: The end of an era or the start of the fun?* Mol. Microbiol. 26(3), 1997, Pp. 417-422.

[10] S. Ohno. *Gene Duplication*. Springer-Verlag, Berlin, 1970.

[11] T. Ohta. *Simulating evolution by gene duplication*. Genetics 115, 1987, Pp. 207-213.

[12] M. Blanchette, T. Kunisawa and D. Sankoff. *Parametric genome rearrangement*. Gene-Combis 172, 1996, Pp. 11-17.

[13] D.A. Christie. *Genome Rearrangement Problems*. Ph.D. Thesis, University of Glasgow, 1998.

[14] S.K. Holland, C.C. Blake. *Proteins, exons, and molecular evolution*. Biosystems 20, 1987, Pp. 181-206.

[15] R.E. Dickerson, I. Geis. *Hemoglobin: Structure, Function, Evolution, and Pathology*. 1983.

Appendice

A Dimostrazioni delle proprietà dei BpGraph

Segue la dimostrazione che se e solo se u è una sottosequenza di S allora il BpGraph G (costruito da S) la riconosce, ma prima alcune definizioni:

Nodi sono entità del grafo con un identificatore *id*, un'etichetta *label* indicante il carattere che rappresentano ed un array di archi uscenti.

Archi sono entità del grafo che connettono due nodi, hanno un identificatore *id*, un'etichetta *label* indicante il carattere rappresentato dal nodo a cui puntano, sono assegnati al nodo da cui partono ed hanno un campo $from$ che limita il loro utilizzo da parte dei cursori. Archi con il campo $from == 0$ sono chiamati *diretti* e possono sempre essere percorsi da ogni cursore, gli altri sono chiamati *laterali*.

Starts[ ] è un array di indici di dimensione uguale alla cardinalità dell'alfabeto. Se c è un carattere dell'alfabeto, $starts[c]$ contiene allora l'*id* del nodo che è punto di ingresso per un cursore, non ancora inserito nel grafo, che inizi il suo percorso con il carattere c.

Cursori sono oggetti che possono muoversi da un nodo del grafo ad un altro, utilizzando un arco che li collega. Sono necessari sia per la costruzione del grafo che per il suo utilizzo. Hanno il campo $nodeId$ (l'*id* del nodo su cui sono concettualmente posti) ed il campo $from$, che mantiene memoria dell'*id* dell'ultimo arco *laterale* percorso o, se ancora non ne

hanno percorsi, contiene il carattere dell'alfabeto attraverso il quale il cursore è stato inserito nel grafo.

Definizione 1 *I nodi N_a e N_b sono connessi (dall'arco L) in relazione al cursore C posizionato su N_a, se e solo se in N_a vi è un arco uscente L, che connetta N_a a N_b ed L è diretto oppure è laterale ma con $L.from = C.from$.*

Definizione 2 *I nodi N_0 N_1 ... N_n sono un percorso per il cursore C_0 posizionato sul nodo N_0 se e solo se ogni coppia di nodi N_i N_{i+1} è connessa (da un arco L_i) in relazione ad un cursore C_i posizionato su N_i definito (per ogni $i > 0$) come $C_i.nodeId = N_i.id$ e $C_i.from = C_{i-1}.from$ se L_{i-1} è diretto, $C_i.from = L_{i-1}.id$ se L_{i-1} è laterale.*

Definizione 3 *Il grafo G riconosce la stringa u (u=u_0 u_1 ... u_n) se e solo se un percorso N_0 N_1 ... N_n con $N_0.id = G.starts[u_0]$ esiste per un cursore posizionato su N_0 con $from = s_0$ e tale che per ogni indice i, $0 \leq i \leq n$, $N_i.label = u_i$.*

Lemma 1 *Se i nodi N_a e N_b sono connessi (dall'arco L_0) in relazione ad un cursore C posizionato su N_a, allora non esiste un terzo nodo N_c tale che $N_c.label = N_b.label$ e che N_a e N_c siano connessi (dall'arco L_1) in relazione allo stesso cursore C.*

dim :

Dall'algoritmo di costruzione di BpGraph descritto a pagina 15 risulta che un arco L viene aggiunto a N_a in due soli casi :

1 - quando viene aggiunto un arco *diretto* tra N_a e N_{a+1}, ed in questo caso N_a non ha ancora nessun altro arco uscente ([DIRECT]).

2 - quando non c'è ancora un arco che connetta N_a ad un nodo con una certa etichetta in relazione ad un cursore con un determinato campo $from$ ([LATERAL]).

La tesi deriva banalmente. □

Lemma 2 *Se il grafo G riconosce u (u=u_0 u_1 ... u_n) allora un percorso N_0 N_1 ... N_n, come precedentemente definito in Definizione 3, esiste unico.*

dim :

L'esistenza viene dalla *Definizione* 3, l'unicità deriva banalmente dal *Lemma* 1. □

Osservazione 1 *Se un cursore C, entrato nel grafo tramite il punto d'ingresso stabilito dall'array starts[ ], è piazzato su un certo nodo N_i e $C.from = f$, allora l'ultimo arco che ha percorso (se N_i non è il suo punto d'ingresso) è l'arco laterale con $id = f$ se quell'arco punta a N_i, altrimenti C arriva dall'arco diretto uscente dal nodo precedente N_{i+1}.*

Osservazione 2 *Un cursore, con un determinato campo from, entrato nel grafo tramite il punto d'ingresso stabilito dall'array starts[ ], piazzato su di un certo nodo, può aver seguito un unico possibile percorso.*

Osservazione 3 *Data una sequenza di caratteri u ed un grafo G, se il grafo G' è stato costruito a partire dalla stessa identica sequenza utilizzata per*

costruire G, ma letta al contrario, allora G riconosce u se e solo se u, letta al contrario, viene riconosciuta da G'.

Teorema 1 *Se u (u=u_0 u_1 ... u_n) è una sottostringa di S e se G è il grafo costruito da S utilizzando l'algoritmo di generazione di BpGraph descritto a pagina 15, allora G riconosce u.*

dim :

Durante la costruzione di G, ci devono essere stati $n+1$ cicli [READ] [ADD] [WALK] [CURSOR] in cui [READ] analizza, in sequenza, u_0, u_1 ... u_n; durante il primo di questi cicli, nel passo [CURSOR], o un nuovo cursore è stato aggiunto, oppure il nodo appena creato è stato messo come starts[u_0] (nel secondo caso la tesi deriva banalmente: u sarà riconosciuta tramite il semplice percorso di archi *diretti* che parte dal punto di ingresso di u_0).

Se durante l'iterazione 0 un nuovo cursore è stato aggiunto (piazzato su di un nodo chiamato N_0), durante ogni iterazione successiva i (se il cursore non è stato ancora eliminato) per quel cursore la condizione dell'IF contenuta nel ciclo [WALK] deve essere verificata.

Se la condizione dell'IF risulta essere vera, allora N_{i-1} e N_i sono connessi (da un arco L_i) in relazione al cursore C piazzato su N_{i-1} e se $L_i.from! = 0$ allora $C.from$ deve essere modificato a $L_i.from$.

Altrimenti, se la condizione dell'IF risulta essere falsa, un nuovo arco *laterale* viene aggiunto, rendendo N_{i-1} e N_i connessi (da

questo nuovo arco) in relazione al cursore C posto su N_{i-1}, poi C viene rimosso.

Se, iterando in questo modo, risultando sempre vera la condizione dell'IF, l'ultimo carattere di u viene raggiunto, allora il percorso seguito da C è quello che permette a G di riconoscere la sequenza u, altrimenti, se un nuovo arco *laterale* è stato aggiunto rimuovendo C da un certo nodo N_j, fino a quel nodo un percorso esiste e poi, da N_j, avendo creato un arco che lo connetteva all'ultimo nodo che era stato aggiunto durante la creazione del grafo, un percorso di archi *diretti*, percorribile da qualsiasi cursore, conduce attraverso i caratteri che ancora mancano; tale percorso è quello che permette a G di riconoscere la sequenza u.
□

Teorema 2 *Se u (u=u_0 u_1 ... u_n)* ***non è*** *una sottostringa di S e se G è il grafo generato da S tramite l'algoritmo di generazione di BpGraph descritto a pagina 15, allora G* ***non*** *riconosce u.*

dim :

Se u non è una sottostringa di S allora o u_0 non appare in S (in questo caso G banalmente non riconosce u, fallendo nel trovare il punto di ingresso quando ne legge il primo carattere) oppure vi è un j tale che :

u_0 u_1 ... u_j formano una sottostringa di S

ma u_0 u_1 ... u_j u_{j+1} non è una sottostringa di S

Dal *Teorema* 1 deriva che u_0 u_1 ... u_j è riconosciuta da G.

Dal *Lemma* 2 deriva che esiste unico il percorso $N_0\ N_1\ ...\ N_j$ che riconosce $u_0\ u_1\ ...\ u_j$.

Tutti i cursori che, partendo da N_0 $(= starts[u_0])$ con $from = u_0$, coprano $j-1$ nodi con le etichette $u_1\ u_2\ ...\ u_j$, percorrono gli stessi archi, quindi quando raggiungono N_j avranno tutti lo stesso campo $from = f$.

$N_0\ N_1\ ...\ N_j\ N_{j+1}$, con $N_{j+1}.label = u_{j+1}$, sono un percorso solo se N_j e N_{j+1} sono connessi da un arco L (essendo $N_0\ N_1\ ...\ N_j$ un percorso).

Un tale arco L può essere sia *laterale* che *diretto*, si inizia considerando l'ipotesi che esso sia *laterale*.

Per poter connettere N_j e N_{j+1} in relazione ad un cursore che abbia il campo $from$ uguale ad f, l'arco *laterale* L deve avere $L.from = f$, ma un tale arco esiste solo se, durante la generazione di G, un cursore con $from = f$ è stato posizionato su N_j ed il carattere che ha dovuto seguire successivamente è stato u_{j+1}.

Ma dall'*Osservazione* 2 un cursore C posto su N_j caratterizza univocamente un unico percorso, nel nostro caso $(C.from = f)$ formato dai nodi con le etichette $u_0\ u_1\ ...\ u_j$.

Dato che $u_0\ u_1\ ...\ u_j\ u_{j+1}$ non è una sottostringa di S, questa sequenza di caratteri non è mai apparsa durante la costruzione di G e quindi un arco *laterale* L che connette N_j a N_{j+1} con $N_{j+1}.label = s_{j+1}$ in relazione ad un cursore con $from = f$, non può esistere.

Quindi L non può essere *laterale*, se esiste deve essere *diretto*.

Chiamando L_i l'arco che connette, per ogni i, N_i e N_{i+1} nel percorso $N_0\ N_1\ ...\ N_j$ (questo percorso esiste grazie al fatto che dal *Teorema* 1 si sa che la sequenza $u_0\ u_1\ ...\ u_j$ è riconosciuta da G), si consideri l'arco L_k, $0 \leq k < j$, tale che L_k sia *laterale* e che non esista un altro arco *laterale* $L_{k'}$ con $k' > k$, $k' < j$ ($k' < j$ perché L_j è L).

L_k deve esistere altrimenti significherebbe che c'è un percorso $N_0\ N_1\ ...\ N_j\ N_{j+1}$ formato solo da archi *diretti*, ma ciò non può accadere perché, dall'algoritmo di generazione di BpGraph, un percorso formato da soli archi *diretti* esiste solo se la sequenza $u_0\ u_1\ ...\ u_j\ u_{j+1}$ appare in S, e ciò è falso per ipotesi.

Quindi il percorso $N_{k+1}\ N_{k+2}\ ...\ N_j\ N_{j+1}$ è formato da soli archi *diretti*, il che significa, come deduzione dall'algoritmo di generazione di BpGraph, che la stringa $u_{k+1}\ u_{k+2}\ ...\ u_j\ u_{j+1}$ appare in S.

Un cursore che raggiunge N_{k+1} dopo aver seguito il percorso $N_0\ N_1\ ...\ N_k$ partendo da N_0 con il campo $from = u_0$, deve avere un unico possibile campo $from$, uguale a f' (partendo con $from = s_0$ e percorrendo, forzato, la sequenza di archi $L_0\ L_1\ ...\ L_{k-1}$).

Ma l'arco *laterale* L_k, percorribile solo da cursori con un campo $from$ uguale a f', piazzato sul nodo N_k, che conduca al nodo N_{k+1}, può esistere solo se durante la costruzione del grafo un cur-

sore con $from = f'$ si è trovato posizionato su N_k ed il carattere u_{k+1} è stato analizzato.

Dall'*Osservazione* 2 un cursore posizionato su N_k determina univocamente un percorso, nel presente caso composto da nodi etichettati $u_0\ u_1\ ...\ u_k$.

Quindi, per generare l'arco *laterale* L_k, il carattere u_{k+1} deve essere stato analizzato, quindi la stringa $u_0\ u_1\ ...\ u_k\ u_{k+1}$ deve essere una sottosequenza di S.

L'esistenza di un percorso $N_{k+1}\ N_{k+2}\ ...\ N_j\ N_{j+1}$ formato solo da archi *diretti* fa però dedurre che la sottostringa $u_0\ u_1\ ...\ u_k$ in S era seguita non solo dal carattere u_{k+1}, ma anche da $u_{k+2}\ ...\ u_j\ u_{j+1}$ e quindi anche $u_0\ u_1\ ...\ u_j\ u_{j+1}$ deve essere una sottostringa di S, ma ciò è falso per ipotesi e quindi L non può essere né *laterale*, né *diretto*, e quindi non esiste.

Quindi un percorso $N_0\ N_1\ ...\ N_j\ N_{j+1}$, $N_i.label = u_i\ \ \forall i, 0 \leq i \leq j+1$, $N_0 = starts[u_0]$, non esiste per un cursore C posizionato su N_0 con $C.from = u_0$.

E quindi, dalla $Definizione$ 3, G non riconosce la sequenza u.
□

B Un accenno all'evoluzone

Per evoluzione si intende in modo semplificato l'azione combinata di mutazione e di selezione. Una delle carateristiche che accomuna gli esseri viventi (sebbene vi sia sempre un vivo dibattito epistemologico che non concede una definizione di vivente universalmente accettata) è la presenza di un patrimonio genetico in grado di dirigere la costruzione delle proteine, processo noto come sintesi proteica[5].

Il patrimonio genetico è sottoposto a fenomeni di mutazione e di ricombinazione che possono di conseguenza causare mutazioni e ricombinazioni delle proteine[3], nonché alterare la frequenza e la possibilità di sintesi delle stesse[4][14].

Dato che il *funzionamento* di un vivente è fortemente, quando non assolutamente, determinato dalle sue proteine e dato che le risorse di cui ha bisogno sono sempre limitate ed in genere ardue da ottenere, statisticamente sono i viventi dotati delle proteine più adatte (quelle in grado di dare i migliori vantaggi nella lotta competitiva in una determinata nicchia naturale) che riescono a sopravvivere e a riprodursi, tramandando alla prole il loro genoma.

Questo meccanismo abbastanza semplice può far comprendere, seppur in modo superficiale, come l'evoluzione solitamente procede: il genoma è sottoposto a mutazione casuale, le modifiche in genere alterano la sintesi proteica,

[3]Si ricorda che si sta enormemente semplificando, con l'intento di fornire solo un'idea di base: trattare ogni casistica è al di fuori dello scopo di questa tesi.

[4]Non tutto il materiale genetico codifica proteine, ma è noto che non tutto il codice non codificante è privo di funzionalità: per esempio, appartengono al DNA non codificante anche diverse sequenze di regolazione dell'attività genica.

rivelandosi miglioramenti o peggioramenti a volte fatali, ed è a questo punto che interviene la selezione. La selezione è la prova sul campo: riuscire a non soccombere contro le avversità dell'ambiente e vincere nella competizione per ottenere le risorse di cui si ha bisogno per vivere e per riprodursi. Se un vivente, con un genoma mutato, sopravvive alla selezione, allora le sue mutazioni (i cui effetti non si sono rivelati dannosi) vengono tramandati alle future generazioni; se invece non sopravvive fino alla riproduzione, allora la sua informazione genetica è stata perduta.

L'effetto dell'azione combinata di mutazione e di selezione è dunque, statisticamente, lo spostamento dei genomi dei viventi verso una delle direzioni che ne migliorano l'adattamento all'ambiente in cui vivono.

B.1 Le mutazioni del genoma

La mutazione delle sequenze geniche agisce in svariati modi. Tra vivente e vivente i meccanismi molecolari che causano la mutazione possono essere diversi e agiscono con diverse frequenze.

La tendenza all'ottimizzazione delle varie funzioni molecolari, caratteristica dell'evoluzione, ha reso il processo di mutazione e la struttura del DNA complessi, ma ben predisposti a rendere efficace il processo d'ottimizzazione stesso. Nei viventi si osservano due principali tipologie di mutazione: la mutazione puntiforme e la mutazione di segmenti.

La mutazione puntiforme consite nell'inserzione, nell'eliminazione o nella modifica di una singola base di una sequenza genica; questa tipologia di mutazione può essere considerata come un meccanismo di *regolazione fine*[5]

del genoma, che ha preso il compito di far evolvere le sequenze verso i loro massimi locali.

La mutazione di segmenti consiste invece nella moltiplicazione in tandem, duplicazione traslata, inversione, duplicazione invertita complementata traslata, delezione ed inserzione di intere sottosequenze di basi presenti in una sequenza genica; questa tipologia di mutazione può essere considerata come il meccanismo che permette di effettuare anche grossi salti evolutivi, dando quindi la possibilità di sfuggire da dei massimi locali.

Assieme, mutazione puntiforme e mutazione di segmenti, grazie alle caratteristiche di modularità e di riusabilità dell'informazione genetica[14], rendono l'evoluzione un meccanismo difficilmente prevedibile, creativo, ma soprattutto efficace nel suo migliorare l'adattamento dei viventi.

B.2 Similitudini ed omologie

Per *similitudine* si intende il livello si somiglianza o di differenza che possiamo misurare o comunque osservare.

Per *omologia* si intende invece la comune discendenza. Parlando di omologia, nel caso di due sequenze geniche, si intende che le due sequenze sono imparentate, cioè che sono derivare dalla stessa sequenza ancestrale comune la quale, in un momento del processo evolutivo, si è sdoppiata in due copie, che hanno poi potuto evolversi in modo più o meno indipendente, differenziandosi.

Mentre la similitudine tra sequenze è direttamente osservabile dal genoma, le omologie tra geni non sono direttamente osservabili, ma vengono

dedotte sulla base delle loro similitudini (quando possibile tenendo conto anche di altre osservazioni)[4].

Può un elevato livello di similitudine tra due geni di due diversi organismi giustificare la conclusione che essi sono omologhi e che i viventi che li hanno nel loro genoma sono strettamente imparentati?

Potrebbe essere che quel determinato gene che si sta osservando richiede, per funzionare, una ferrea conservazione di quasi tutte le sue basi, così che molte specie di viventi solo lontanamente imparentate si ritrovano tutte un gene quasi identico. Per questa osservazione due specie in cui individuiamo un gene molto simile possono essere anche lontanissime: riusciamo però ad accorgere andando a vedere se troviamo il gene in altre specie, studiando quindi la sua frequenza di variazione tra vivente e vivente, e poi possiamo comunque valutare altre coppie di geni per vedere se l'elevato livello di similitudine viene sempre osservato.

Potrebbe essere che i due geni abbiano ancestrali diversi, ma la pressione selettiva potrebbe averli spinti verso la stessa strada evolutiva, verso quella che probabilmente risulta essere la sequenza genica migliore, rendendoli quindi molto simili, seppur non omologhi.

In un caso in cui si hanno 3 geni omologhi, A, B e C, di cui A e B più simili che A e C, non si può poi dire con certezza che A e B si siano separati più tardivamente nell'evoluzione, in quanto potrebbe anche essere semplicemente che C, seppur più vicino ad A di quanto lo sia B, si sia trovato sottoposto ad una mutazione più rapida, sia per ragioni ambientali che molecolari.

Non è impensabile poi che due specie con sequenze ad alta similitudine

non siano affatto strettamente imparentate, ma che la sequenza genica analizzata è stata trasportata dall'una all'altra tramite un fenomeno di *lateral gene transfer* (fenomeno importante, anche se raro).

Insomma, bisogna tener conto che le strutture di parentela che l'evoluzione nel passato ha creato e che noi adesso, nel presente, andiamo ad indagare sulla base delle similitudini e della paleontologia, possono essere comprese in modo corretto solo statisticamente.

B.3 Un esempio : l'emoglobina

Molecole trasportatrici di ossigeno del tipo dell'emoglobina si trovano in tutti i vertebrati ed in molti invertebrati. La molecola più primitiva è lunga circa 150 amminoacidi ed è presente in molti vermi marini, insetti e pesci primitivi.

Sembra che circa 500 milioni di anni fa, durante l'evoluzione dei pesci superiori, ci sia stata una duplicazione genetica che ha superato il test della selezione naturale: questo evento diede origine a due geni di globina che poi sono evoluti lungo strade diverse[4][15].

La molecola di emoglobina nei vertebrati superiori è composta da due tipi di globine, leggermente diversi, così simili da poterne riconoscere la comune origine: si tratta dell'alfa e della beta globina. Le due globine sono in grado di legarsi in un complesso proteico a 4, cooperando nel loro compito di legare l'ossigeno.

Più tardi, durante l'evoluzione dei mammiferi, il gene della catena beta si è nuovamente duplicato e ha dato origine ad una variazione che viene sintetizzata specificatamente nel feto.

Altre modifiche avvenute successivamente hanno generato altri due tipi di globine, che vengono prodotte in periodi diversi dello sviluppo.

La modifica più tardiva, per quanto riguarda il ramo dei primati, è la delta globina, presente unicamente nei primati adulti.

Poiché i gruppi di diversi geni delle alfa e delle beta globine si trovano su cromosomi separati negli uccelli e nei mammiferi ma sono insieme negli anfibi, si pensa che un evento di traslocazione abbia separato i due geni circa 300 milioni di anni fa.

Il probabile albero evolutivo dell'emoglobina umana è schematizzato in figura 48.

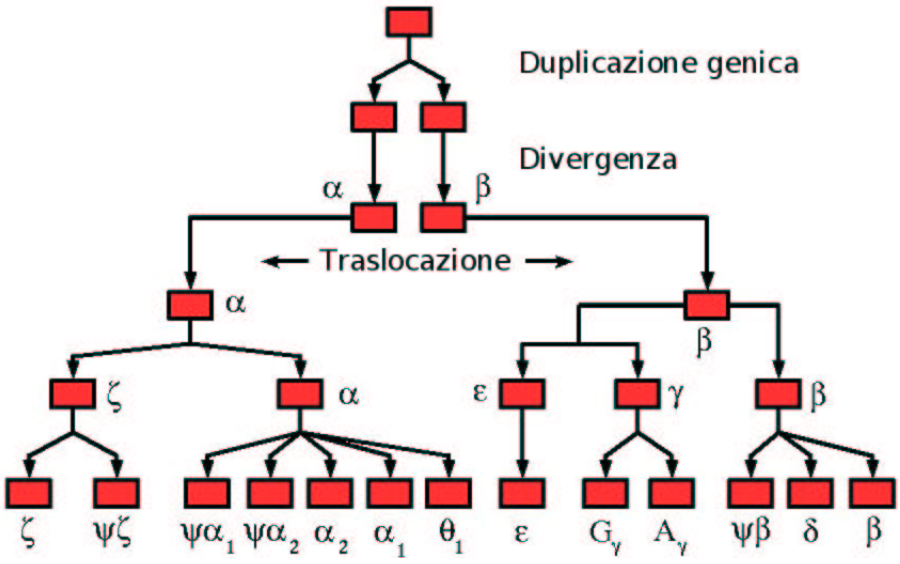

Figura 48: l'albero evolutivo dell'emoglobina umana[4].

C Codice del software sviluppato

C.1 BpGraph

C.1.1 bpgraph.h

```
1  /*
2  // copyright 2003 Claudio Felicioli (mail: c.felicioli@1d20.net)
3  //
4  // This program is free software; you can redistribute it and/or modify
5  // it under the terms of the GNU General Public License as published by
6  // the Free Software Foundation; either version 2 of the License, or
7  // (at your option) any later version.
8  */
9
10 #include "../bpmatchutils_src/bpmatchutils.h"
11
12
13 struct bpLink {
14   int id;
15   int from;
16   int link;
17   int next;
18   };
19
20 struct bpNode {
21   base label;
22   int link[4];
23   };
24
25 struct bpCursor {
26   int at;
27   int from;
28   };
29
30 class BpGraph {
31   public:
32     BpGraph(const int length);
33     BpGraph(FILE* file);
34     int length;
35     int cursorLength;
36     int maxCursorLength;
37     int linkCreated;
38     void feedNode(const base bp);
39     void viewGraph();
40     void serialize(FILE* file);
41     void test(const int n);
42     int isFinalized();
43     int match(bpCursor* cursor, const base bp);
```

```
44      int linksCount(const int nodeId, int until);
45    private:
46      int nodeListHead;
47      bpNode* nodeList;
48      bpCursor* cursors;
49      bpLink* links;
50      int starts[4];
51      int nextLinkId;
52      void addCursor(const int at, const int from);
53      void cursorsWalk(const base bp);
54      int getNextLinkId();
55      void addLink(int* linkListNode, const int to, const int from);
56      int finalized;
57    };
```

C.1.2 bpgraph.cpp

```
1  /*
2  // copyright 2003 Claudio Felicioli (mail: c.felicioli@1d20.net)
3  //
4  // This program is free software; you can redistribute it and/or modify
5  // it under the terms of the GNU General Public License as published by
6  // the Free Software Foundation; either version 2 of the License, or
7  // (at your option) any later version.
8  */
9
10 #include "bpgraph.h"
11 #define DEBUG 0
12
13
14 BpGraph::BpGraph(const int length) {
15   this->length=length;
16   nodeList=new bpNode[this->length];
17   nodeListHead=0;
18   cursors=new bpCursor[this->length];
19   cursorLength=0;
20   maxCursorLength=cursorLength;
21   starts[0]=-1;
22   starts[1]=starts[0];
23   starts[2]=starts[0];
24   starts[3]=starts[0];
25   links=new bpLink[(2*this->length)+5];
26   linkCreated=0;
27   // 1234 are assigned to base starts
28   nextLinkId=5;
29   finalized=0;
30   }
31
32 BpGraph::BpGraph(FILE* file) {
33   length=-999;
34   cursorLength=-999;
35   fread(&maxCursorLength, sizeof(int), 1, file);
36   fread(&linkCreated, sizeof(int), 1, file);
37   fread(&nodeListHead, sizeof(int), 1, file);
38   nextLinkId=-999;
39   fread(starts, sizeof(int), 4, file);
40   nodeList=new bpNode[nodeListHead];
41   length=nodeListHead;
42   fread(nodeList, sizeof(bpNode), nodeListHead, file);
43   links=new bpLink[nodeListHead-1+5+linkCreated];
44   fread(links, sizeof(bpLink), nodeListHead-1+5+linkCreated, file);
45   finalized=1;
46   }
47
```

```
48 void BpGraph::serialize(FILE* file) {
49   fwrite(&maxCursorLength, sizeof(int), 1, file);
50   fwrite(&linkCreated, sizeof(int), 1, file);
51   fwrite(&nodeListHead, sizeof(int), 1, file);
52   fwrite(starts, sizeof(int), 4, file);
53   fwrite(nodeList, sizeof(bpNode), nodeListHead, file);
54   fwrite(links, sizeof(bpLink), (nodeListHead-1+5+linkCreated), file);
55   }
56
57 void BpGraph::addLink(int* linkListNode, const int to, const int from) {
58   if(isFinalized()==1) {
59     printf("error: adding a link in finalized state\n");
60     return;
61     }
62   bpLink listNode;
63   listNode.id=getNextLinkId();
64   listNode.from=from;
65   listNode.link=to;
66   listNode.next=-1;
67   links[listNode.id]=listNode;
68   if(*linkListNode==-1) *linkListNode=listNode.id;
69   else {
70     int linkId=*linkListNode;
71     while(links[linkId].next!=-1) linkId=links[linkId].next;
72     links[linkId].next=listNode.id;
73     }
74   }
75
76 void BpGraph::feedNode(const base bp) {
77   if(isFinalized()==1) {
78     printf("error: adding a node in finalized state\n");
79     return;
80     }
81   if(DEBUG) printf("adding %c\n", DNAbases[bp]);
82   nodeList[nodeListHead].label=bp;
83   nodeList[nodeListHead].link[0]=-1;
84   nodeList[nodeListHead].link[1]=-1;
85   nodeList[nodeListHead].link[2]=-1;
86   nodeList[nodeListHead].link[3]=-1;
87   if(nodeListHead>0 && bp!=N) addLink(&nodeList[nodeListHead-1].link[bp],
       nodeListHead, 0);
88   cursorsWalk(bp);
89   if(bp!=N) {
90     if(starts[bp]==-1) starts[bp]=nodeListHead;
91     else {
92       if(DEBUG) printf("adding cursors at %c start\n", DNAbases[bp]);
93       //bp+1 because from=0 is special for the link of consecutive nodes
94       addCursor(starts[bp], bp+1);
95       }
```

```
96        }
97      nodeListHead++;
98      }
99
100   void BpGraph::viewGraph() {
101     printf("***\n");
102     printf("STARTS:\n");
103     printf("\tA=%d\n", starts[A]);
104     printf("\tG=%d\n", starts[G]);
105     printf("\tT=%d\n", starts[T]);
106     printf("\tC=%d\n", starts[C]);
107     printf("CURSORS:\n\tmax %d\n", maxCursorLength);
108     printf("LINKCREATED:\n\t%d\n", linkCreated);
109     printf("***\n");
110     }
111
112   void BpGraph::addCursor(const int at, const int from) {
113     if(isFinalized()==1) {
114       printf("error: adding a cursor in finalized state\n");
115       return;
116       }
117     cursors[cursorLength].at=at;
118     cursors[cursorLength++].from=from;
119     if(maxCursorLength<cursorLength) maxCursorLength=cursorLength;
120     }
121
122   void BpGraph::cursorsWalk(const base bp) {
123     if(isFinalized()==1) {
124       printf("error: walking cursors in finalized state\n");
125       return;
126       }
127     if(bp==N) {
128       cursorLength=0;
129       return;
130       }
131     int from, at, linkId;
132     for(int i=0;i<cursorLength;i++) {
133       from=cursors[i].from;
134       at=cursors[i].at;
135       linkId=nodeList[at].link[bp];
136       while(linkId!=-1 && links[linkId].from!=from && links[linkId].from!=0) linkId
              =links[linkId].next;
137       if(linkId==-1) {
138         if(DEBUG) printf("adding link from %d to %d\n", from, nodeListHead);
139         addLink(&nodeList[cursors[i].at].link[bp], nodeListHead, from);
140         linkCreated++;
141         cursors[i].at=cursors[cursorLength-1].at;
142         cursors[i--].from=cursors[(cursorLength--)-1].from;
143         }
```

```
144       else {
145         cursors[i].at=links[linkId].link;
146         if(links[linkId].from!=0) cursors[i].from=links[linkId].id;
147         }
148       }
149     }
150
151   int BpGraph::getNextLinkId() { return(nextLinkId++); }
152
153   void BpGraph::test(const int n) {
154     for(int i=0;i<n;i++) printf("(%2d) ", i);
155     printf("\n");
156     for(int i=0;i<n;i++) printf("%c——", DNAbases[nodeList[i].label]);
157     printf("\n\n");
158     int linkId,depth,sum;
159     for(int bp=0;bp<4;bp++) {
160     printf("link to %c:\n", DNAbases[bp]);
161       sum=1;
162       depth=0;
163       while(sum>0) {
164         sum=0;
165         for(int i=0;i<n;i++) {
166           linkId=nodeList[i].link[bp];
167           int j=0;
168           //skip direct links to next node
169           if(linkId!=-1 && links[linkId].link==i+1) linkId=links[linkId].next;
170           while(j<depth) {
171             if(linkId!=-1) linkId=links[linkId].next;
172             j++;
173             }
174           if(linkId==-1) printf("      ");
175           else {
176             printf("%2d-%2d", links[linkId].id, links[linkId].link);
177             sum++;
178             }
179           }
180         depth++;
181         printf("\n");
182         }
183       }
184     printf("\n\n");
185     }
186
187   int BpGraph::isFinalized() { return(finalized); }
188
189   int BpGraph::linksCount(const int nodeId, int until=0) {
190     if(until==0) until=length;
191     bpNode node=nodeList[nodeId];
192     int linkId, out;
```

```
193     out=0;
194     for(int i=0;i<4;i++) {
195       linkId=node.link[i];
196       while(linkId!=-1 && links[linkId].link<until) {
197         linkId=links[linkId].next;
198         out++;
199         }
200       }
201     return(out);
202     }
203
204   int BpGraph::match(bpCursor* cursor, const base bp) {
205     if(isFinalized()==0) {
206       printf("error: you can't match if it isn't finalized\n");
207       return(-1);
208       }
209     //bp==X && bp==N return mismatch ever
210     if(bp==X || bp==N) return(0);
211     if(cursor->at==-1) {
212       if(starts[bp]==-1) return(0);
213       cursor->at=starts[bp];
214       //bp+1 because from=0 is special for the link of consecutive nodes
215       cursor->from=bp+1;
216       return(1);
217       }
218     int linkId=nodeList[cursor->at].link[bp];
219     while(linkId!=-1 && links[linkId].from!=cursor->from && links[linkId].from!=0)
          linkId=links[linkId].next;
220     if(linkId!=-1) {
221       //match
222       if(DEBUG) printf("da %d a %d using %d(needtobefrom:%d) -- cursor.from=%d\n",
              cursor->at, links[linkId].link, links[linkId].id, links[linkId].from,
              cursor->from);
223       cursor->at=links[linkId].link;
224       if(links[linkId].from!=0) cursor->from=links[linkId].id;
225       return(1);
226       }
227     //mismatch
228     return(0);
229     }
```

C.2 Creazione dei BpGraph

C.2.1 bpmakegraph.cpp

```
1  /*
2  // bpmakegraph create a BpGraph graph object from a nuclotide sequence.
3  // In this version it's able to generate some statistical graphic
4  // information in png format.
5  // version 1.1
6  //
7  // copyright 2003 Claudio Felicioli (mail: c.felicioli@1d20.net)
8  //
9  // bpmakegraph is free software; you can redistribute it and/or modify
10 // it under the terms of the GNU General Public License as published by
11 // the Free Software Foundation; either version 2 of the License, or
12 // (at your option) any later version.
13 */
14
15 #include <stdlib.h>
16 #include <unistd.h>
17 #include <stdio.h>
18 #include <string.h>
19 #include <math.h>
20 #include <sys/types.h>
21 #include <sys/stat.h>
22 #include "../bpmatchutils_src/bpmatchutils.h"
23 #include "../bpgraph_src/bpgraph.h"
24
25 // graphic
26 #include "gd.h"
27 #include "gdfonts.h"
28
29
30 FILE* gene;
31 BpGraph* bpGraph;
32 int geneSize;
33 int reverse, reversel;
34 char geneName[50];
35 char graphName[50];
36 void intestation();
37 void prologo();
38 int scanBp(FILE* file, base* bp);
39 void serialize();
40 void epilogo();
41
42 // grapic
43 int GRAPIC=1;
44 FILE* outputImage;
45 char outputImageName[50];
```

```
46 gdImagePtr image;
47 int width, heigth;
48 int colors;
49 int bgColor, fgColor, tratti, expectedColor;
50 int* rainbow;
51 int step;
52 int* activeCursors;
53 void setNewImage();
54 void drawGrid();
55 void lLinks();
56 void aCursors();
57
58
59 int main(int argc, char *argv[]) {
60   if(argc!=3 && (argc!=4 || argv[3][0]!='R' || argv[3][1]!='\0')) {
61     printf("usage: %s geneSourceFile graphTargetFile [R]\nIf R the graph
         rapresent the reverse subsequences.\n", argv[0]);
62     exit(-1);
63     }
64   strncpy(geneName, argv[1], 48);
65   geneName[49]='\0';
66   strncpy(graphName, argv[2], 48);
67   graphName[49]='\0';
68   if(argc==4 && argv[3][0]=='R') reverse=2;
69   else reverse=0;
70   intestation();
71   prologo();
72   bpGraph=new BpGraph(geneSize);
73   base bp;
74   int counter=0;
75   if(GRAPIC) activeCursors=new int[geneSize];
76   while(scanBp(gene, &bp)!=EOF) {
77     if(GRAPIC) activeCursors[counter]=bpGraph->cursorLength;
78     if(counter%50000==0) printf("%d bases processed:\n\tlink created: %d\n\t%d
         cursors active (max %d)\n", counter, bpGraph->linkCreated, bpGraph->
         cursorLength, bpGraph->maxCursorLength);
79     bpGraph->feedNode(bp);
80     counter++;
81     }
82   bpGraph->viewGraph();
83   printf("counter: %d\n", counter);
84   serialize();
85   if(GRAPIC) step=(int)floor(bpGraph->length/(width-20));
86   if(GRAPIC) aCursors();
87   if(GRAPIC) lLinks();
88   epilogo();
89   }
90
91 void setNewImage() {
```

```
92    image=gdImageCreate(width, heigth);
93    bgColor=gdImageColorAllocate(image, 255, 255, 255);
94    fgColor=gdImageColorAllocate(image, 0, 0, 0);
95    expectedColor=gdImageColorAllocate(image, 255, 0, 0);
96    rainbow=new int[colors];
97    double red, green, blue;
98    int mezzi=(int)(colors/2);
99    int quarti=(int)(colors/4);
100   for(int i=0;i<colors;i++) {
101     if(i<quarti) {
102       blue=255;
103       green=(int)((255*i)/mezzi);
104       red=0;
105       }
106     else if(i<mezzi) {
107       blue=255-(int)(255*(i-quarti)/mezzi);
108       green=(int)((255*i)/mezzi);
109       red=255-blue;
110       }
111     else if(i<mezzi+quarti) {
112       blue=255-(int)(255*(i-quarti)/mezzi);
113       green=255-(int)((255*(i-mezzi))/mezzi);
114       red=255-blue;
115       }
116     else {
117       blue=0;
118       green=255-(int)((255*(i-mezzi))/mezzi);
119       red=255;
120       }
121     rainbow[i]=gdImageColorAllocate(image, (int)round(red), (int)round(green), (
          int)round(blue));
122     }
123   tratti=gdImageColorAllocate(image, 0, 0, 0);
124   }
125
126 void drawGrid() {
127   for(int i=0;i<width-40;i+=10) {
128     gdImageLine(image, i+20, 20, i+25, 20, tratti);
129     gdImageLine(image, i+20, 19, i+25, 19, tratti);
130     gdImageLine(image, i+20, heigth-20, i+25, heigth-20, tratti);
131     gdImageLine(image, i+20, heigth-19, i+25, heigth-19, tratti);
132     gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)*3/4), i+25, heigth
          -20-(int)ceil((heigth-40)*3/4), tratti);
133     gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)*3/4)-1, i+25, heigth
          -20-(int)ceil((heigth-40)*3/4)-1, tratti);
134     gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)/2), i+25, heigth
          -20-(int)ceil((heigth-40)/2), tratti);
135     gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)/2)-1, i+25, heigth
          -20-(int)ceil((heigth-40)/2)-1, tratti);
```

```
136       gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)/4), i+25, heigth
              -20-(int)ceil((heigth-40)/4), tratti);
137       gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)/4)-1, i+25, heigth
              -20-(int)ceil((heigth-40)/4)-1, tratti);
138       }
139     for(int i=0;i<heigth-40;i+=10) {
140       gdImageLine(image, 20+(int)ceil((width-40)/4), i+20, 20+(int)ceil((width-40)
              /4), i+25, tratti);
141       gdImageLine(image, 20+(int)ceil((width-40)/2), i+20, 20+(int)ceil((width-40)
              /2), i+25, tratti);
142       gdImageLine(image, 20+(int)ceil((width-40)*3/4), i+20, 20+(int)ceil((width
              -40)*3/4), i+25, tratti);
143       gdImageLine(image, 20+width-40, i+20, 20+width-40, i+25, tratti);
144       gdImageLine(image, 20, i+20, 20, i+25, tratti);
145       }
146     }
147
148   void lLinks() {
149     snprintf(outputImageName, 50, "%s.llink.png", geneName);
150     outputImageName[49]='\0';
151     printf("drawing %s...\n", outputImageName);
152     outputImage=fopen(outputImageName, "wb");
153     if(!outputImage) ERROR("failed open outputImage file");
154     setNewImage();
155     int hs[width-40][colors];
156     int links;
157     int ccc;
158     int h=0;
159     for(int i=0;i<width-40;i++) {
160       for(int j=0;j<colors;j++) {
161         links=0;
162         for(int y=0;y<step;y++) {
163           ccc=bpGraph->linksCount(i*step+y, (int)((bpGraph->length*(1+j))/colors))
                  -1;
164           if(ccc>0) links+=ccc-1;
165           }
166         hs[i][j]=links;
167         if(links>h) h=links;
168         }
169       }
170     drawGrid();
171     for(int i=0;i<width-40;i++) {
172       for(int j=colors-1;j>=0;j--) {
173         if(hs[i][j]>0) gdImageLine(image, i+20, heigth-20, i+20, heigth-20-(int)
                ceil(((heigth-40)*hs[i][j])/h), rainbow[j]);
174         }
175       }
176     char str[21];
177     for(int i=1;i<=4;i++) {
```

```
178       snprintf(str, 19, "%d bp", (int)bpGraph->length*i/4);
179       str[20]='\0';
180       gdImageString(image, gdFontSmall, 20+(int)((width-40)*i/4)-60, heigth-18, (
              unsigned char*)str, fgColor);
181       }
182     snprintf(str, 14, "lateral links");
183     str[20]='\0';
184     gdImageStringUp(image, gdFontSmall, 5, 100, (unsigned char*)str, fgColor);
185     snprintf(str, 14, "0 bp");
186     str[20]='\0';
187     gdImageString(image, gdFontSmall, 15, heigth-18, (unsigned char*)str, fgColor);
188     snprintf(str, 10, "%.2f", (double)h/step);
189     str[20]='\0';
190     gdImageString(image, gdFontSmall, width-50, 5, (unsigned char*)str, fgColor);
191     snprintf(str, 10, "%.2f", (double)h*3/(4*step));
192     str[20]='\0';
193     gdImageString(image, gdFontSmall, width-50, heigth-20-(int)ceil((heigth-40)
            *3/4)-15, (unsigned char*)str, fgColor);
194     snprintf(str, 10, "%.2f", (double)h/(2*step));
195     str[20]='\0';
196     gdImageString(image, gdFontSmall, width-50, heigth-20-(int)ceil((heigth-40)/2)
            -15, (unsigned char*)str, fgColor);
197     snprintf(str, 10, "%.2f", (double)h/(4*step));
198     str[20]='\0';
199     gdImageString(image, gdFontSmall, width-50, heigth-20-(int)ceil((heigth-40)/4)
            -15, (unsigned char*)str, fgColor);
200     gdImagePng(image, outputImage);
201     gdImageDestroy(image);
202     int outputImagefd;
203     if((outputImagefd=fileno(outputImage))==-1) ERROR("failed getting outputImage
            file descriptor");
204     if(fsync(outputImagefd)!=0) ERROR("failed fsync of outputImage");
205     if(fclose(outputImage)!=0) ERROR("failed fclose of outputImage");
206     printf("done\n");
207     }
208
209   void aCursors() {
210     snprintf(outputImageName, 50, "%s.acursors.png", geneName);
211     outputImageName[49]='\0';
212     printf("drawing %s...\n", outputImageName);
213     outputImage=fopen(outputImageName, "wb");
214     if(!outputImage) ERROR("failed open outputImage file");
215     setNewImage();
216     int hs[width-40];
217     int cursors;
218     int h=0;
219     for(int i=0;i<width-40;i++) {
220       cursors=0;
221       for(int y=0;y<step;y++) cursors+=activeCursors[i*step+y];
```

```
222       hs[i]=cursors;
223       if(cursors>h) h=cursors;
224       }
225     drawGrid();
226     double expected;
227     int lasth=0;
228     for(int i=0;i<width-40;i++) {
229       expected=log10(i*bpGraph->length/(width-40))*1.661;
230       expected*=step;
231       if(hs[i]>lasth) gdImageLine(image, i+20, heigth-20-(int)ceil(((heigth-40)*hs[
              i])/h)-1, i+20, heigth-20-(int)ceil(((heigth-40)*lasth)/h)+1, fgColor);
232       else gdImageLine(image, i+20, heigth-20-(int)ceil(((heigth-40)*lasth)/h)-1, i
              +20, heigth-20-(int)ceil(((heigth-40)*hs[i])/h)+1, fgColor);
233       lasth=hs[i];
234       gdImageLine(image, i+20, heigth-20-(int)ceil(((heigth-40)*expected)/h)-1, i
              +20, heigth-20-(int)ceil(((heigth-40)*expected)/h), expectedColor);
235       }
236     char str[21];
237     for(int i=1;i<=4;i++) {
238       snprintf(str, 19, "%d bp", (int)bpGraph->length*i/4);
239       str[20]='\0';
240       gdImageString(image, gdFontSmall, 20+(int)((width-40)*i/4)-60, heigth-18, (
              unsigned char*)str, fgColor);
241       }
242     snprintf(str, 15, "active cursors");
243     str[20]='\0';
244     gdImageStringUp(image, gdFontSmall, 5, 100, (unsigned char*)str, fgColor);
245     snprintf(str, 14, "0 bp");
246     str[20]='\0';
247     gdImageString(image, gdFontSmall, 15, heigth-18, (unsigned char*)str, fgColor);
248     snprintf(str, 10, "%.2f", (double)h/step);
249     str[20]='\0';
250     gdImageString(image, gdFontSmall, width-50, 5, (unsigned char*)str, fgColor);
251     snprintf(str, 10, "%.2f", (double)h*3/(4*step));
252     str[20]='\0';
253     gdImageString(image, gdFontSmall, width-50, heigth-20-(int)ceil((heigth-40)
          *3/4)-15, (unsigned char*)str, fgColor);
254     snprintf(str, 10, "%.2f", (double)h/(2*step));
255     str[20]='\0';
256     gdImageString(image, gdFontSmall, width-50, heigth-20-(int)ceil((heigth-40)/2)
          -15, (unsigned char*)str, fgColor);
257     snprintf(str, 10, "%.2f", (double)h/(4*step));
258     str[20]='\0';
259     gdImageString(image, gdFontSmall, width-50, heigth-20-(int)ceil((heigth-40)/4)
          -15, (unsigned char*)str, fgColor);
260     gdImagePng(image, outputImage);
261     gdImageDestroy(image);
262     int outputImagefd;
263     if((outputImagefd=fileno(outputImage))==-1) ERROR("failed getting outputImage
```

```
        file descriptor");
264   if(fsync(outputImagefd)!=0) ERROR("failed fsync of outputImage");
265   if(fclose(outputImage)!=0) ERROR("failed fclose of outputImage");
266   printf("done\n");
267   }
268
269 void serialize() {
270   printf("serializing graph...\n");
271   FILE* graph=fopen(graphName, "w");
272   bpGraph->serialize(graph);
273   int graphfd;
274   if((graphfd=fileno(graph))==-1) ERROR("failed getting graph file descriptor");
275   if(fsync(graphfd)!=0) ERROR("failed fsync of graph");
276   if(fclose(graph)!=0) ERROR("failed fclose of graph");
277   printf("done\n");
278   }
279
280 void epilogo() {
281   int genefd;
282   if((genefd=fileno(gene))==-1) ERROR("failed getting gene file descriptor");
283   if(fsync(genefd)!=0) ERROR("failed fsync of gene");
284   if(fclose(gene)!=0) ERROR("failed fclose of gene");
285   printf("termination reached without errors\n\n\n");
286   exit(EXIT_SUCCESS);
287   }
288
289 void intestation() {
290   system("clear");
291   printf("/*\n");
292   printf("// bpmakegraph create a BpGraph graph object from a nuclotide sequence\
        n");
293   printf("// In this version it's able to generate some statistical graphic\n");
294   printf("// information in png format.\n");
295   printf("// version 1.1\n");
296   printf("//\n");
297   printf("// copyright 2003 Claudio Felicioli (mail: c.felicioli@1d20.net)\n");
298   printf("//\n");
299   printf("// bpmakegraph is free software; you can redistribute it and/or modify\
        n");
300   printf("// it under the terms of the GNU General Public License as published by
        \n");
301   printf("// the Free Software Foundation; either version 2 of the License, or\n"
        );
302   printf("// (at your option) any later version.\n");
303   printf("*/\n\n");
304   }
305
306 void prologo() {
307   bpmatchutilsinit();
```

```
308     if((gene=fopen(geneName, "r"))==NULL) {
309       printf("error: %s opening fail\n", geneName);
310       exit(-1);
311       }
312     if(reverse==2 && fseek(gene, 0, SEEK_END)!=0) {
313       printf("error: fseek fail\n");
314       exit(-1);
315       }
316     struct stat geneInfo;
317     stat(geneName, &geneInfo);
318     geneSize=geneInfo.st_size;
319     if(GRAPIC) width=600;
320     if(GRAPIC) heigth=300;
321     if(GRAPIC) colors=30;
322     }
323
324   int scanBp(FILE* file, base* bp) {
325     char c;
326     if(reverse==1 && fseek(file, -2, SEEK_CUR)!=0) return EOF;
327     if(reverse==2) {
328       if(fseek(file, -1, SEEK_CUR)!=0) return EOF;
329       reverse--;
330       }
331     if(fscanf(file, "%c", &c)==EOF) return EOF;
332     if(c=='\n' || c==' ' || c=='\t') return(scanBp(file, bp));
333     switch(c) {
334       case 'A':
335         *bp=A;
336         break;
337       case 'G':
338         *bp=G;
339         break;
340       case 'T':
341         *bp=T;
342         break;
343       case 'C':
344         *bp=C;
345         break;
346       case 'U':
347         *bp=U;
348         break;
349       case 'N':
350         *bp=N;
351         break;
352       default:
353         printf("failed traducing char %c to base.\n", c);
354         exit(-1);
355       }
356     return(1);
```

```
357         }
```

C.3 Generazione di sequenze genetiche pseudocasuali

C.3.1 creationism.cpp

```
1  /*
2  // creationism generate a randomized nuclotide sequence, DNA or RNA
3  // useful for testing purposes.
4  // version 1.0
5  //
6  // copyright 2003 Claudio Felicioli (mail: c.felicioli@1d20.net)
7  //
8  // creationism is free software; you can redistribute it and/or modify
9  // it under the terms of the GNU General Public License as published by
10 // the Free Software Foundation; either version 2 of the License, or
11 // (at your option) any later version.
12 */
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <string.h>
16 #include <unistd.h>
17 #include <time.h>
18 #include <math.h>
19 #include "creationism.h"
20
21
22 int main(int argc, char *argv[]) {
23   if(argc<3 || argc>4) {
24     printf("usage: %s bplength targetfile [DNA/RNA]\ndefault coding is DNA\n",
         argv[0]);
25     exit(-1);
26     }
27   int coding=0;
28   if(argc==4) {
29     if(strcmp(argv[3], "DNA")!=0 && strcmp(argv[3], "RNA")!=0) {
30       printf("error: only DNA and RNA coding are aviable\n");
31       exit(-1);
32       }
33     if(strcmp(argv[3], "RNA")==0) coding=1;
34     }
35   char temp[13];
36   snprintf(temp, 13, "%d", atoi(argv[1]));
37   if(strcmp(temp, argv[1])!=0) {
38     printf("error: bplength %s is not a number\n", argv[1]);
39     exit(-1);
40     }
41   int bplength=atoi(argv[1]);
42   if(bplength<=0) {
43     printf("error: bplength must be greater than 0\n");
44     exit(-1);
```

```
45      }
46    char temp2[50];
47    snprintf(temp2, 50, "%s.bp", argv[2]);
48    FILE* output;
49    if((output=fopen(temp2, "w"))==NULL) {
50      printf("error: impossible to create/modify %s.bp\n", argv[2]);
51      exit(-1);
52      }
53    srandom(time(NULL));
54    for(int i=0;i<bplength;i+=3*100) {
55      int nbp=3*100;
56      if(i+nbp>bplength) nbp=bplength-i;
57      char bpseq[nbp+1];
58      for(int j=0;j<nbp;j++) bpseq[j]=bases[coding][xDy(1, 4)-1];
59      bpseq[nbp]='\0';
60      fprintf(output, "%s", bpseq);
61      fprintf(output, "\n");
62      }
63    int outputfd;
64    if((outputfd=fileno(output))==-1) {
65      printf("error: failed getting output file descriptor\n");
66      exit(-1);
67      }
68    if(fsync(outputfd)!=0) {
69      printf("error: failed fsync of output file\n");
70      exit(-1);
71      }
72    exit(EXIT_SUCCESS);
73    }
74
75  int xDy(const int x, const int y) {
76    int dice=0;
77    for(int i=0;i<x;i++) dice+=(int)ceil((((double)y)*((double)random()))/((double)
          RAND_MAX));
78    return(dice);
79    }
```

C.4 Analisi della distribuzione degli archi laterali in un BpGraph

C.4.1 creationism.cpp

```
1  /*
2  // laterallinks analize BpGraph graphs to make statistics
3  // about the distribution of lateral links
4  // version 1.0
5  //
6  // copyright 2003 Claudio Felicioli (mail: c.felicioli@1d20.net)
7  //
8  // laterallinks is free software; you can redistribute it and/or modify
9  // it under the terms of the GNU General Public License as published by
10 // the Free Software Foundation; either version 2 of the License, or
11 // (at your option) any later version.
12 */
13
14 #include <stdlib.h>
15 #include <unistd.h>
16 #include <stdio.h>
17 #include <string.h>
18 #include <sys/types.h>
19 #include <sys/stat.h>
20 #include <math.h>
21 #include "gd.h"
22 #include "gdfonts.h"
23 #include "../bpgraph_src/bpgraph.h"
24 #define DEBUG 0
25
26
27 FILE* graph;
28 BpGraph* bpGraph;
29 char graphName[50];
30 char outputImageName[50];
31 gdImagePtr image;
32 int bgColor;
33 int fgColor;
34 int tratti;
35 int colors;
36 int* rainbow;
37 int width, heigth;
38 int step, h;
39 void intestation();
40 void prologo();
41 void epilogo();
42
43
44 int main(int argc, char *argv[]) {
```

```
45    if(argc!=5) {
46      printf("usage: %s graph outputImage width heigth\n", argv[0]);
47      exit(-1);
48      }
49    strncpy(graphName, argv[1], 48);
50    graphName[49]='\0';
51    snprintf(outputImageName, 50, "%s.png", argv[2]);
52    outputImageName[49]='\0';
53    width=atoi(argv[3]);
54    heigth=atoi(argv[4]);
55    if(width<50 || heigth<50) {
56      printf("width and heigth have to be at least 50\n");
57      exit(-1);
58      }
59    intestation();
60    prologo();
61    printf("Loading %s.\n", graphName);
62    bpGraph=new BpGraph(graph);
63    bpGraph->viewGraph();
64    printf("Graph unserialized.\n");
65    printf("\n\n");
66    printf("************************************************\n");
67    printf("analizing graph\n");
68    printf("************************************************\n");
69    FILE* outputImage=fopen(outputImageName, "wb");
70    if(!outputImage) ERROR("failed open outputImage file");
71    image=gdImageCreate(width, heigth);
72    bgColor=gdImageColorAllocate(image, 255, 255, 255);
73    fgColor=gdImageColorAllocate(image, 0, 0, 0);
74    colors=20;
75    rainbow=new int[colors];
76    double red, green, blue;
77    int mezzi=(int)(colors/2);
78    int quarti=(int)(colors/4);
79    for(int i=0;i<colors;i++) {
80      if(i<quarti) {
81        blue=255;
82        green=(int)((255*i)/mezzi);
83        red=0;
84        }
85      else if(i<mezzi) {
86        blue=255-(int)(255*(i-quarti)/mezzi);
87        green=(int)((255*i)/mezzi);
88        red=255-blue;
89        }
90      else if(i<mezzi+quarti) {
91        blue=255-(int)(255*(i-quarti)/mezzi);
92        green=255-(int)((255*(i-mezzi))/mezzi);
93        red=255-blue;
```

```
94          }
95        else {
96          blue=0;
97          green=255-(int)((255*(i-mezzi))/mezzi);
98          red=255;
99          }
100       rainbow[i]=gdImageColorAllocate(image, red, green, blue);
101       }
102     tratti=gdImageColorAllocate(image, 0, 0, 0);
103     step=floor(bpGraph->length/(width-20));
104     int hs[width-40][colors];
105     int links;
106     int ccc;
107     h=0;
108     for(int i=0;i<width-40;i++) {
109       for(int j=0;j<colors;j++) {
110         links=0;
111         for(int y=0;y<step;y++) {
112           ccc=bpGraph->linksCount(i*step+y, (int)((bpGraph->length*(1+j))/colors))
                -1;
113           if(ccc>0) links+=ccc-1;
114           }
115         hs[i][j]=links;
116         if(links>h) h=links;
117         }
118       }
119     for(int i=0;i<width-40;i+=10) {
120       gdImageLine(image, i+20, 20, i+25, 20, tratti);
121       gdImageLine(image, i+20, 19, i+25, 19, tratti);
122       gdImageLine(image, i+20, heigth-20-ceil((heigth-40)*3/4), i+25, heigth-20-
            ceil((heigth-40)*3/4), tratti);
123       gdImageLine(image, i+20, heigth-20-ceil((heigth-40)*3/4)-1, i+25, heigth-20-
            ceil((heigth-40)*3/4)-1, tratti);
124       gdImageLine(image, i+20, heigth-20-ceil((heigth-40)/2), i+25, heigth-20-ceil
            ((heigth-40)/2), tratti);
125       gdImageLine(image, i+20, heigth-20-ceil((heigth-40)/2)-1, i+25, heigth-20-
            ceil((heigth-40)/2)-1, tratti);
126       gdImageLine(image, i+20, heigth-20-ceil((heigth-40)/4), i+25, heigth-20-ceil
            ((heigth-40)/4), tratti);
127       gdImageLine(image, i+20, heigth-20-ceil((heigth-40)/4)-1, i+25, heigth-20-
            ceil((heigth-40)/4)-1, tratti);
128       }
129     for(int i=0;i<heigth-40;i+=10) {
130       gdImageLine(image, 20+ceil((width-40)/4), i+20, 20+ceil((width-40)/4), i+25,
            tratti);
131       gdImageLine(image, 20+ceil((width-40)/2), i+20, 20+ceil((width-40)/2), i+25,
            tratti);
132       gdImageLine(image, 20+ceil((width-40)*3/4), i+20, 20+ceil((width-40)*3/4), i
            +25, tratti);
```

```
133       gdImageLine(image, 20+width-40, i+20, 20+width-40, i+25, tratti);
134       }
135     double expected;
136     for(int i=0;i<width-40;i++) {
137       expected=log10(bpGraph->length)*1.661-log10(i*bpGraph->length/(width-40))
              *1.661;
138       expected*=step;
139       for(int j=colors-1;j>=0;j--) {
140         if(hs[i][j]>0) gdImageLine(image, i+20, heigth-20, i+20, heigth-20-ceil(((
              heigth-40)*hs[i][j])/h), rainbow[j]);
141         }
142       gdImageLine(image, i+20, heigth-20-ceil(((heigth-40)*expected)/h)-1, i+20,
              heigth-20-ceil(((heigth-40)*expected)/h), tratti);
143       }
144     char str[21];
145     for(int i=1;i<=4;i++) {
146       snprintf(str, 19, "%d bp", (int)bpGraph->length*i/4);
147       str[20]='\0';
148       gdImageString(image, gdFontSmall, 20+(int)((width-40)*i/4)-60, heigth-18, (
              unsigned char*)str, fgColor);
149       }
150     snprintf(str, 14, "lateral links");
151     str[20]='\0';
152     gdImageStringUp(image, gdFontSmall, 5, 100, (unsigned char*)str, fgColor);
153     snprintf(str, 14, "0 bp");
154     str[20]='\0';
155     gdImageString(image, gdFontSmall, 15, heigth-18, (unsigned char*)str, fgColor);
156     snprintf(str, 10, "%.2f", (double)h/step);
157     str[20]='\0';
158     gdImageString(image, gdFontSmall, width-50, 5, (unsigned char*)str, fgColor);
159     snprintf(str, 10, "%.2f", (double)h*3/(4*step));
160     str[20]='\0';
161     gdImageString(image, gdFontSmall, width-50, heigth-20-ceil((heigth-40)*3/4)
          -15, (unsigned char*)str, fgColor);
162     snprintf(str, 10, "%.2f", (double)h/(2*step));
163     str[20]='\0';
164     gdImageString(image, gdFontSmall, width-50, heigth-20-ceil((heigth-40)/2)-15, (
          unsigned char*)str, fgColor);
165     snprintf(str, 10, "%.2f", (double)h/(4*step));
166     str[20]='\0';
167     gdImageString(image, gdFontSmall, width-50, heigth-20-ceil((heigth-40)/4)-15, (
          unsigned char*)str, fgColor);
168     gdImagePng(image, outputImage);
169     gdImageDestroy(image);
170     int outputImagefd;
171     if((outputImagefd=fileno(outputImage))==-1) ERROR("failed getting outputImage
          file descriptor");
172     if(fsync(outputImagefd)!=0) ERROR("failed fsync of outputImage");
173     if(fclose(outputImage)!=0) ERROR("failed fclose of outputImage");
```

```
174     epilogo();
175     }
176
177   void intestation() {
178     system("clear");
179     printf("/*\n");
180     printf("// laterallinks analize BpGraph graphs to make statistics\n");
181     printf("// about the distribution of lateral links\n");
182     printf("// version 1.0\n");
183     printf("//\n");
184     printf("// copyright 2003 Claudio Felicioli (mail: c.felicioli@1d20.net)\n");
185     printf("//\n");
186     printf("// laterallinks is free software; you can redistribute it and/or modify
            \n");
187     printf("// it under the terms of the GNU General Public License as published by
            \n");
188     printf("// the Free Software Foundation; either version 2 of the License, or\n"
            );
189     printf("// (at your option) any later version.\n");
190     printf("*/\n\n");
191     }
192
193   void prologo() {
194     if((graph=fopen(graphName, "r"))==NULL) {
195       printf("error: %s opening fail\n", graphName);
196       exit(-1);
197       }
198     }
199
200   void epilogo() {
201     int graphfd;
202     if((graphfd=fileno(graph))==-1) ERROR("failed getting graph file descriptor");
203     if(fsync(graphfd)!=0) ERROR("failed fsync of graph");
204     if(fclose(graph)!=0) ERROR("failed fclose of graph");
205     printf("termination reached without errors\n\n\n");
206     exit(EXIT_SUCCESS);
207     }
```

C.5 BpMatch

C.5.1 bpmatch.cpp

```
1  /*
2  // bpmatch calculate, from sequences S and T, the maximum coverage of T
3  // using only subsequences and complemented reversed subsequences of S,
4  // with minimum length l, possibly overlapped, and, in such a maximum
5  // coverage, minimize the number of subsequences used.
6  // In this version it's able to generate some statistical graphic
7  // information in png format.
8  // version 1.1
9  //
10 // copyright 2003 Claudio Felicioli (mail: c.felicioli@1d20.net)
11 //
12 // bpmatch is free software; you can redistribute it and/or modify
13 // it under the terms of the GNU General Public License as published by
14 // the Free Software Foundation; either version 2 of the License, or
15 // (at your option) any later version.
16 */
17
18 #include <stdlib.h>
19 #include <unistd.h>
20 #include <stdio.h>
21 #include <string.h>
22 #include <math.h>
23 #include <sys/types.h>
24 #include <sys/stat.h>
25 #include "../bpgraph_src/bpgraph.h"
26 #define DEBUG 0
27
28 // graphic
29 #include "gd.h"
30 #include "gdfonts.h"
31
32
33 FILE* graph;
34 FILE* graphR;
35 FILE* sequence;
36 int minimum;
37 BpGraph* bpGraph;
38 BpGraph* bpGraphR;
39 char graphName[50];
40 char graphRName[50];
41 char sequenceName[50];
42 base* frame;
43 int at;
44 base* frameR;
45 int atR;
```

```
46 int notabase;
47 int state;
48 int countCaseZero;
49 int countCaseOne;
50 int countCaseOneSub;
51 int iterations;
52 void intestation();
53 void prologo();
54 int scanBp(FILE* file, base* bp, const int position);
55 base bpAt(const int pos);
56 base bpAtR(const int pos);
57 int shiftFramesTo(const int pos);
58 void epilogo();
59
60 // grapic
61 int TEST=0;
62 int testFROM, testTO;
63 FILE* outputImage;
64 char outputImageName[100];
65 gdImagePtr image;
66 int width, heigth, colWidth;
67 int bgColor, fgColor, tratti, case0Color, case1Color, case1bColor;
68 int** statistics;
69 int h;
70 void draw();
71
72
73 int main(int argc, char *argv[]) {
74   if(argc!=5 && (argc!=8 || argv[5][0]!='T' || argv[5][1]!='\0')) {
75     printf("usage: %s graph reverseGraph geneSourceFile minimumLength [T testFROM
         testTO]\n", argv[0]);
76     exit(-1);
77     }
78   strncpy(graphName, argv[1], 48);
79   graphName[49]='\0';
80   strncpy(graphRName, argv[2], 48);
81   graphRName[49]='\0';
82   strncpy(sequenceName, argv[3], 48);
83   sequenceName[49]='\0';
84   minimum=atoi(argv[4]);
85   if(argc==8) {
86     TEST=1;
87     testFROM=atoi(argv[6]);
88     testTO=atoi(argv[7]);
89     if(testTO<testFROM) {
90       printf("to<from !?!\n");
91       exit(-1);
92       }
93     iterations=testTO-testFROM+1;
```

```
94        if(iterations>200) {
95          printf("range to-from can be at max 200\n");
96          exit(-1);
97          }
98        width=440;
99        heigth=500;
100       colWidth=(int)floor(400/iterations);
101       statistics=new int*[iterations];
102       for(int i=0;i<iterations;i++) statistics[i]=new int[3];
103       h=0;
104       }
105     else iterations=1;
106     intestation();
107     prologo();
108     printf("Loading %s.\n", graphName);
109     bpGraph=new BpGraph(graph);
110     bpGraph->viewGraph();
111     printf("Graph unserialized.\n");
112     printf("Loading %s.\n", graphRName);
113     bpGraphR=new BpGraph(graphR);
114     bpGraphR->viewGraph();
115     printf("Reverse graph unserialized.\n");
116     base bp;
117     int end;
118     int actual;
119     int errorAt, errorAtR;
120     int found, foundR;
121     int foundAt, foundAtR;
122     int dontdo, dontdoR;
123     int i;
124     int goOn;
125     int maxSuffix;
126     bpCursor cursor;
127     for(int ite=0;ite<iterations;ite++) {
128       if(TEST) minimum=ite+testFROM;
129       delete [] frame;
130       delete [] frameR;
131       frame=new base[minimum-1+minimum];
132       frameR=new base[minimum-1+minimum];
133       notabase=0;
134       at=0;
135       atR=0;
136       printf("\n\n");
137       printf("************************************************\n");
138       printf("matching - l=%d\n", minimum);
139       printf("************************************************\n");
140       for(int i=0;i<minimum-1+minimum;i++) {
141         frame[i]=X;
142         frameR[i]=X;
```

```
143         }
144       for(int i=0;i<minimum;i++) {
145         if(scanBp(sequence, &bp, i)==1) {
146           frame[minimum-1+i]=bp;
147           frameR[minimum-1+i]=frame[minimum-1+i];
148           }
149         }
150       i=0;
151       state=0;
152       end=0;
153       actual=0;
154       dontdo=0;
155       dontdoR=0;
156       countCaseZero=0;
157       countCaseOne=0;
158       countCaseOneSub=0;
159       while(!end) {
160         found=0;
161         foundR=0;
162         if(state==0) {
163           //case 0
164           if(DEBUG) printf("\nCASE 0:\n");
165           if(dontdo) dontdo=0;
166           else {
167             countCaseZero++;
168             //searching for direct seq
169             if(DEBUG) printf("searching for direct seq\n");
170             cursor.at=-1;
171             i=minimum-1;
172             //using reverse from actual+minimum to actual (backward parsing)
173             while(i>=0 && bpGraphR->match(&cursor, bpAt(actual+i))==1) {
174               if(DEBUG) printf("(%d)%c", actual+i, DNAbases[bpAt(actual+i)]);
175               i--;
176               }
177             if(i>=0) {
178               //from actual to errorAt included no match direct seq of length
                      minimum is possible
179               if(DEBUG) printf("(%d)%c(!) mismatch before %d\n", actual+i, DNAbases
                      [bpAt(actual+i)], minimum);
180               errorAt=actual+i;
181               }
182             else {
183               if(DEBUG) printf(" OK\n");
184               cursor.at=-1;
185               i=0;
186               //using direct from actual (forward parsing)
187               while(bpGraph->match(&cursor, bpAt(actual+i))==1) {
188                 if(DEBUG) printf("(%d)%c", actual+i, DNAbases[bpAt(actual+i)]);
189                 i++;
```

```
190                 }
191               if(DEBUG) printf(" direct sequence found!\n");
192               found=1;
193               foundAt=actual+i; //base immediatly successive the matching seq
194               }
195             }
196           if(dontdoR) dontdoR=0;
197           else {
198             countCaseZero++;
199             //searching for reverse seq
200             if(DEBUG) printf("searching for reverse seq\n");
201             cursor.at=-1;
202             i=minimum-1;
203             //using direct from actual+minimum to actual (backward parsing)
204             while(i>=0 && bpGraph->match(&cursor, bpAtR(actual+i))==1) {
205               if(DEBUG) printf("(%d)%c", actual+i, DNAbases[bpAtR(actual+i)]);
206               i--;
207               }
208             if(i>=0) {
209               //from actual to errorAtR included no match reverse seq of length
                      minimum is possible
210               if(DEBUG) printf("(%d)%c(!) mismatch before %d\n", actual+i, DNAbases
                      [bpAtR(actual+i)], minimum);
211               errorAtR=actual+i;
212               }
213             else {
214               if(DEBUG) printf(" OK\n");
215               cursor.at=-1;
216               i=0;
217               //using reverse from actual (forward parsing)
218               while(bpGraphR->match(&cursor, bpAtR(actual+i))==1) {
219                 if(DEBUG) printf("(%d)%c", actual+i, DNAbases[bpAtR(actual+i)]);
220                 i++;
221                 }
222               if(DEBUG) printf(" reverse sequence found!\n");
223               foundR=1;
224               foundAtR=actual+i; //base immediatly successive the matching seq
225               }
226             }
227           }
228         else {
229           int l;
230           //case 1
231           if(DEBUG) printf("\nCASE 1:\n");
232           countCaseOne++;
233           //searching for direct (eventually ovelapping) seq
234           if(DEBUG) printf("searching for direct (eventually ovelapping) seq\n");
235           cursor.at=-1;
236           maxSuffix=0;
```

```
237             //using direct from actual (forward parsing)
238             while(bpGraphR->match(&cursor, bpAt(actual+maxSuffix))==1) {
239               if(DEBUG) printf("(%d)%c", actual+maxSuffix, DNAbases[bpAt(actual+
                      maxSuffix)]);
240               maxSuffix++;
241               }
242             if(maxSuffix>=minimum) {
243               if(DEBUG) printf(" direct sequence found!\n");
244               found=1;
245               foundAt=actual+maxSuffix; //base immediatly successive the matching seq
246               }
247             else {
248               if(DEBUG) printf(" maxSuffix=%d\n", maxSuffix);
249               for(int potentialstart=actual-minimum+maxSuffix;found==0 &&
                      potentialstart>actual-minimum;potentialstart--) {
250                 countCaseOneSub++;
251                 cursor.at=-1;
252                 l=0;
253                 //using direct from potentialstart (forward parsing)
254                 while(bpGraph->match(&cursor, bpAt(potentialstart+l))==1) {
255                   if(DEBUG) printf("(%d)%c", potentialstart+l, DNAbases[bpAt(
                          potentialstart+l)]);
256                   l++;
257                   }
258                 if(l>=minimum) {
259                   if(DEBUG) printf(" direct sequence found!\n");
260                   found=1;
261                   foundAt=potentialstart+l; //base immediatly successive the matching
                          seq
262                   }
263                 else if(DEBUG) printf(" no...\n");
264                 }
265               if(found==0) {
266                 //from actual to errorAt included no match direct seq of length
                        minimum is possible
267                 if(DEBUG) printf("no possible start for overlapping direct sequence\n
                        ");
268                 errorAt=actual+maxSuffix-1;
269                 }
270               }
271             countCaseOne++;
272             //searching for reverse (eventually ovelapping) seq
273             if(DEBUG) printf("searching for reverse (eventually ovelapping) seq\n");
274             cursor.at=-1;
275             maxSuffix=0;
276             //using revers from actual (forward parsing)
277             while(bpGraphR->match(&cursor, bpAtR(actual+maxSuffix))==1) {
278               if(DEBUG) printf("(%d)%c", actual+maxSuffix, DNAbases[bpAtR(actual+
                      maxSuffix)]);
```

```
279             maxSuffix++;
280             }
281           if(maxSuffix>=minimum) {
282             if(DEBUG) printf(" reverse sequence found!\n");
283             foundR=1;
284             foundAtR=actual+maxSuffix; //base immediatly successive the matching
                    seq
285             }
286           else {
287             if(DEBUG) printf(" maxSuffix=%d\n", maxSuffix);
288             for(int potentialstart=actual-minimum+maxSuffix;foundR==0 &&
                    potentialstart>actual-minimum;potentialstart--) {
289               countCaseOneSub++;
290               cursor.at=-1;
291               l=0;
292               //using reverse from potentialstart (forward parsing)
293               while(bpGraphR->match(&cursor, bpAtR(potentialstart+l))==1) {
294                 if(DEBUG) printf("(%d)%c", potentialstart+l, DNAbases[bpAtR(
                        potentialstart+l)]);
295                 l++;
296                 }
297               if(l>=minimum) {
298                 if(DEBUG) printf(" reverse sequence found!\n");
299                 foundR=1;
300                 foundAtR=potentialstart+l; //base immediatly successive the
                        matching seq
301                 }
302               else if(DEBUG) printf(" no...\n");
303               }
304             if(foundR==0) {
305               //from actual to errorAtR included no match reverse seq of length
                      minimum is possible
306               if(DEBUG) printf("no possible start for overlapping reverse sequence\
                      n");
307               errorAtR=actual+maxSuffix-1;
308               }
309             }
310           }
311         if(found+foundR==0) {
312           if(DEBUG) printf("neither direct nor reverse seq found\n");
313           if(DEBUG) printf("error at %d and %d\n", errorAt, errorAtR);
314           goOn=errorAt+1;
315           if(errorAtR+1<goOn) {
316             goOn=errorAtR+1;
317             dontdo=1;
318             }
319           else dontdoR=1;
320           if(errorAt==errorAtR) {
321             dontdo=0;
```

```
322             dontdoR=0;
323             }
324           end=!shiftFramesTo(goOn);
325           actual=goOn;
326           state=0;
327           }
328         if(found+foundR==2) {
329           if(DEBUG) printf("both direct and reverse seq found\n");
330           if(DEBUG) printf("cover seq to %d and %d\n", foundAt-1, foundAtR-1);
331           goOn=foundAt;
332           char di='D';
333           if(foundAtR>goOn) {
334             goOn=foundAtR;
335             di='R';
336             }
337           end=!shiftFramesTo(goOn);
338           if(goOn-actual<minimum) printf("+%d(%c overlapping)", goOn-actual, di);
339           else printf("\n%c seq found starting %d - length=%d", di, actual, goOn-
                actual);
340           actual=goOn;
341           state=1;
342           }
343         if(found==1 && foundR==0) {
344           if(DEBUG) printf("only direct seq found\n");
345           if(DEBUG) printf("cover seq to %d\n", foundAt-1);
346           goOn=foundAt;
347           end=!shiftFramesTo(goOn);
348           if(goOn-actual<minimum) printf("+%d(D overlapping)", goOn-actual);
349           else printf("\nD seq found starting %d - length=%d", actual, goOn-actual)
                ;
350           actual=goOn;
351           state=1;
352           }
353         if(found==0 && foundR==1) {
354           if(DEBUG) printf("only reverse seq found\n");
355           if(DEBUG) printf("cover seq to %d\n", foundAtR-1);
356           goOn=foundAtR;
357           end=!shiftFramesTo(goOn);
358           if(goOn-actual<minimum) printf("+%d(R overlapping)", goOn-actual);
359           else printf("\nR seq found starting %d - length=%d", actual, goOn-actual)
                ;
360           actual=goOn;
361           state=1;
362           }
363         }
364       printf("\ncase0=%d\ncase1=%d\ncase1sub=%d\n\n", countCaseZero, countCaseOne,
            countCaseOneSub);
365       if(TEST) {
366         statistics[ite][0]=countCaseZero;
```

```
367         statistics[ite][1]=countCaseOne;
368         statistics[ite][2]=countCaseOneSub;
369         if(h<countCaseZero+countCaseOne+countCaseOneSub) h=countCaseZero+
                countCaseOne+countCaseOneSub;
370         }
371       }
372     if(TEST) draw();
373     epilogo();
374     }
375
376   void draw() {
377     snprintf(outputImageName, 100, "%s.%s.%d.test.png", graphRName, sequenceName,
          testFROM);
378     outputImageName[99]='\0';
379     printf("drawing %s...\n", outputImageName);
380     outputImage=fopen(outputImageName, "wb");
381     if(!outputImage) ERROR("failed open outputImage file");
382     image=gdImageCreate(width, heigth);
383     bgColor=gdImageColorAllocate(image, 255, 255, 255);
384     fgColor=gdImageColorAllocate(image, 0, 0, 0);
385     case0Color=gdImageColorAllocate(image, 0, 255, 0);
386     case1Color=gdImageColorAllocate(image, 0, 0, 255);
387     case1bColor=gdImageColorAllocate(image, 255, 0, 0);
388     tratti=gdImageColorAllocate(image, 0, 0, 0);
389     for(int i=0;i<width-40;i+=10) {
390       gdImageLine(image, i+20, 20, i+25, 20, tratti);
391       gdImageLine(image, i+20, 19, i+25, 19, tratti);
392       gdImageLine(image, i+20, heigth-20, i+25, heigth-20, tratti);
393       gdImageLine(image, i+20, heigth-19, i+25, heigth-19, tratti);
394       gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)*3/4), i+25, heigth
            -20-(int)ceil((heigth-40)*3/4), tratti);
395       gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)*3/4)-1, i+25, heigth
            -20-(int)ceil((heigth-40)*3/4)-1, tratti);
396       gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)/2), i+25, heigth
            -20-(int)ceil((heigth-40)/2), tratti);
397       gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)/2)-1, i+25, heigth
            -20-(int)ceil((heigth-40)/2)-1, tratti);
398       gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)/4), i+25, heigth
            -20-(int)ceil((heigth-40)/4), tratti);
399       gdImageLine(image, i+20, heigth-20-(int)ceil((heigth-40)/4)-1, i+25, heigth
            -20-(int)ceil((heigth-40)/4)-1, tratti);
400       }
401     for(int i=0;i<iterations;i++) {
402       for(int j=0;j<colWidth;j++) {
403         gdImageLine(image, i*colWidth+j+20, heigth-20, i*colWidth+j+20, heigth-20-(
              int)ceil(((heigth-40)*statistics[i][0])/h), case0Color);
404         gdImageLine(image, i*colWidth+j+20, heigth-20-(int)ceil(((heigth-40)*
              statistics[i][0])/h), i*colWidth+j+20, heigth-20-(int)ceil(((heigth-40)
              *statistics[i][1])/h)-(int)ceil(((heigth-40)*statistics[i][0])/h),
```

```
            case1Color);
405         gdImageLine(image, i*colWidth+j+20, heigth-20-(int)ceil(((heigth-40)*
                statistics[i][1])/h)-(int)ceil(((heigth-40)*statistics[i][0])/h), i*
                colWidth+j+20, heigth-20-(int)ceil(((heigth-40)*statistics[i][2])/h)-(
                int)ceil(((heigth-40)*statistics[i][1])/h)-(int)ceil(((heigth-40)*
                statistics[i][0])/h), case1bColor);
406         }
407       }
408     char str[21];
409     for(int i=1;i<=4;i++) {
410       snprintf(str, 10, "l=%d", testFROM+(int)((iterations-1)*i/4));
411       str[20]='\0';
412       gdImageString(image, gdFontSmall, 20+(int)((width-40)*i/4)-60, heigth-18, (
              unsigned char*)str, fgColor);
413       }
414     snprintf(str, 11, "iterations");
415     str[20]='\0';
416     gdImageStringUp(image, gdFontSmall, 5, 100, (unsigned char*)str, fgColor);
417     snprintf(str, 10, "l=%d", testFROM);
418     str[20]='\0';
419     gdImageString(image, gdFontSmall, 15, heigth-18, (unsigned char*)str, fgColor);
420     gdImagePng(image, outputImage);
421     gdImageDestroy(image);
422     int outputImagefd;
423     if((outputImagefd=fileno(outputImage))==-1) ERROR("failed getting outputImage
            file descriptor");
424     if(fsync(outputImagefd)!=0) ERROR("failed fsync of outputImage");
425     if(fclose(outputImage)!=0) ERROR("failed fclose of outputImage");
426     printf("done\n");
427     }
428
429   base bpAt(const int pos) {
430     base out;
431     if(pos-at>minimum || pos-at<=-minimum) {
432       printf("\nfatal error: invalid frame access (%d)\n", pos-at);
433       exit(-1);
434       }
435     if(pos-at==minimum) {
436       at++;
437       for(int i=0;i<minimum-1+minimum-1;i++) frame[i]=frame[i+1];
438       frame[minimum-1+minimum-1]=X;
439       }
440     out=frame[minimum-1+pos-at];
441     if(out==X) {
442       base bp;
443       if(scanBp(sequence, &bp, pos)==1) frame[minimum-1+pos-at]=bp;
444       else return(X);
445       out=frame[minimum-1+pos-at];
446       }
```

```
447   return(out);
448   }
449
450 base bpAtR(const int pos) {
451   base out;
452   if(pos-atR>minimum || pos-atR<=-minimum) {
453     printf("\nfatal error: invalid frame access (%d)\n", pos-atR);
454     exit(-1);
455     }
456   if(pos-atR==minimum) {
457     atR++;
458     for(int i=0;i<minimum-1+minimum-1;i++) frameR[i]=frameR[i+1];
459     frameR[minimum-1+minimum-1]=X;
460     }
461   out=frameR[minimum-1+pos-atR];
462   if(out==X) {
463     base bp;
464     if(scanBp(sequence, &bp, pos)==1) frameR[minimum-1+pos-atR]=bp;
465     else return(X);
466     out=frameR[minimum-1+pos-atR];
467     }
468   return(out);
469   }
470
471 int shiftFramesTo(const int pos) {
472   int delta1=pos-at;
473   at+=delta1;
474   for(int i=0;i<minimum-1+minimum-delta1;i++) frame[i]=frame[i+delta1];
475   int delta2=pos-atR;
476   atR+=delta2;
477   for(int i=0;i<minimum-1+minimum-delta2;i++) frameR[i]=frameR[i+delta2];
478   base bp;
479   if(delta1>delta2) {
480     for(int i=1;i<=delta2;i++) {
481       //is needed to read bases in order, so spaces and newlines are catched
482       if(scanBp(sequence, &bp, atR+minimum-1-delta2+i)==1) frameR[minimum-1+
              minimum-1-delta2+i]=bp;
483       else frameR[minimum-1+minimum-1-delta2+i]=X;
484       }
485     for(int i=0;i<delta1 && i<minimum-1+minimum;i++) frame[minimum-1+minimum-1-i
            ]=frameR[minimum-1+minimum-1-i];
486     }
487   else {
488     for(int i=1;i<=delta1;i++) {
489       //is needed to read bases in order, so spaces and newlines are catched
490       if(scanBp(sequence, &bp, at+minimum-1-delta1+i)==1) frame[minimum-1+minimum
              -1-delta1+i]=bp;
491       else frame[minimum-1+minimum-1-delta1+i]=X;
492       }
```

```
493       for(int i=0;i<delta2 && i<minimum-1+minimum;i++) frameR[minimum-1+minimum-1-i
              ]=frame[minimum-1+minimum-1-i];
494       }
495     if(frame[minimum-1]==X) return(0);
496     return(1);
497     }
498
499   void intestation() {
500     system("clear");
501     printf("/*\n");
502     printf("// bpmatch calculate, from sequences S and T, the maximum coverage of T
            \n");
503     printf("// using only subsequences and complemented reversed subsequences of S
            ,\n");
504     printf("// with minimum length l, possibly overlapped, and, in such a maximum\n
            ");
505     printf("// coverage, minimize the number of subsequences used.\n");
506     printf("// In this version it's able to generate some statistical graphic\n");
507     printf("// information in png format.\n");
508     printf("// version 1.1\n");
509     printf("//\n");
510     printf("// copyright 2003 Claudio Felicioli (mail: c.felicioli@1d20.net)\n");
511     printf("//\n");
512     printf("// bpmatch is free software; you can redistribute it and/or modify\n");
513     printf("// it under the terms of the GNU General Public License as published by
            \n");
514     printf("// the Free Software Foundation; either version 2 of the License, or\n"
            );
515     printf("// (at your option) any later version.\n");
516     printf("*/\n\n");
517     }
518
519   void prologo() {
520     if((graph=fopen(graphName, "r"))==NULL) {
521       printf("error: %s opening fail\n", graphName);
522       exit(-1);
523       }
524     if((graphR=fopen(graphRName, "r"))==NULL) {
525       printf("error: %s opening fail\n", graphRName);
526       exit(-1);
527       }
528     if((sequence=fopen(sequenceName, "r"))==NULL) {
529       printf("error: %s opening fail\n", sequenceName);
530       exit(-1);
531       }
532     }
533
534   int scanBp(FILE* file, base* bp, const int position) {
535     char c;
```

```
536     if(fseek(file, position+notabase, SEEK_SET)==-1 || fscanf(file, "%c", &c)==EOF)
            return(-1);
537     else {
538       if(c=='\n' || c==' ' || c=='\t') {
539         notabase++;
540         return(scanBp(file, bp, position));
541         }
542       switch(c) {
543         case 'A':
544           *bp=A;
545           break;
546         case 'G':
547           *bp=G;
548           break;
549         case 'T':
550           *bp=T;
551           break;
552         case 'C':
553           *bp=C;
554           break;
555         case 'U':
556           *bp=U;
557           break;
558         case 'N':
559           *bp=N;
560           break;
561         default:
562           printf("\nfailed traducing char %c to base, at position %d.\n", c,
                  position);
563           exit(-1);
564         }
565       }
566     return(1);
567     }
568
569   void epilogo() {
570     int graphfd, rgraphfd;
571     if((graphfd=fileno(graph))==-1) ERROR("failed getting graph file descriptor");
572     if(fsync(graphfd)!=0) ERROR("failed fsync of graph");
573     if(fclose(graph)!=0) ERROR("failed fclose of graph");
574     if((rgraphfd=fileno(graphR))==-1) ERROR("failed getting reverse graph file
          descriptor");
575     if(fsync(rgraphfd)!=0) ERROR("failed fsync of graphR");
576     if(fclose(graphR)!=0) ERROR("failed fclose of graphR");
577     printf("termination reached without errors\n\n\n");
578     exit(EXIT_SUCCESS);
579     }
```

Ringraziamenti

Dopo tutto questo tempo passato a Pisa mi rendo conto che la lista di persone che mi hanno aiutato, dandomi sostegno e motivazioni, accompagnandomi fino a questo simbolico punto di rotta, è troppo lunga per essere interamente elencata (e quasi tutti sanno che anche volendo non sarei comunque in grado di fare molti nomi), spero di non fare troppi torti. I pochi nomi che metto sono ordinati casualmente, tramite d10, ovviamente non sto scherzando.

Tra tutti ringrazio con sempre vivo affetto gli amici vicini fin dai primi tempi: Davide, Giacomo, Pippo, Cerra, Nico, Ela e Lisa. Un affettuoso sorriso a tutti i giocolieri del Parcheggione con cui ho trascorso (e pretendo continuare a trascorrere) innumerevoli momenti di infantile spensieratezza e grazie a cui ho ripreso contatto con il mio corpo. Dato che la mia attività preferita è restata fin dall'infanzia il gioco, non posso non ringraziare i tantissimi compagni dei molti (troppi) giochi vissuti negli ultimi 5 anni a Pisa (in particolare Maia e Lucilla Plinia) e in tutti gli anni vissuti a Brescia, tra questi ultimi in particolare William, Alberto, Emanuele, Federico, Teo e Luca. Un'enorme influenza umana e culturale la riconosco a tutti gli eccezionali artisti e agli amici del San Bernardo. Ringrazio poi alcuni compagni di studi della Normale che, oltre all'amicizia, mi hanno donato anche le loro passioni. Un grazie lo devo a diversi amici del Silvestre e dintorni e alla loro intensa fiducia nelle persone.

Chi più tra tutti devo ringraziare però sono coloro che mi hanno cresciuto e che mi hanno lasciato, senza ostacoli, volare dal nido, mamma e papà, che ho troppe volte fatto sentire esclusi ma che sempre mi hanno amato e che sempre ho amato. Con particolare piacere ringrazio il nonno ed il dono che mi ha fatto, quando da piccolo (non che poi abbia smesso) mi ha fatto vivere i giochi ed i divertimenti non elettronici con cui è cresciuto.

Infine, ringrazio Alessandra, a cui devo gran parte della grammatica di questa tesi (e di svariate relazioni precedenti), ma ovviamente non è solo per questo che sento di doverle più che agli altri. Un bacio.