Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Implementazione ed analisi di metodi alignment free con mismatch

Laureando

Daniele Bisello

Relatrice

Prof.ssa Cinzia Pizzi

Anno Accademico 2014/2015

Alla mia famiglia

Daniele Bisello

Indice

1 Introduzione **1**

2 Analisi Filogenetica **3**

2.1 Ricostruzione di Alberi Filogenetici 4

2.1.1 Metodo basato sulle distanze 5

2.1.2 Metodo della massima parsimonia 7

2.1.3 Metodo della massima verosimiglianza 7

2.2 Metodi Alignment-free 7

3 Strumenti software utilizzati **11**

3.1 Valgrind 11

3.2 OpenMP 12

3.3 Phylip 12

4 Notazione e concetti base **15**

5 Massima Correlazione **19**

5.1 MaxCor versione quadratica 19

5.1.1 Pseudocodice 20

5.2 MaxCor versione subquadratica 23

5.2.1 Preprocessing 24

5.2.2 Processing 24

6 Correlazione Media **27**

6.1 AvCor versione quadratica 27

6.1.1 Pseudocodice 27

6.2 AvCor versione subquadratica 29

6.2.1 Preprocessing 29

6.2.2 Processing 30

7 Media della Massima Correlazione **31**
7.1 Approccio ACS e sottostringhe con k mismatch 31
7.2 Approssimazione sottostringhe con k mismatch 32

8 Test e discussione dei risultati **35**
8.1 Esperimenti sull'efficacia in un dataset biologico 35
8.1.1 Descrizione del dataset usato 35
8.1.2 Test 36
8.1.3 Discussione dei risultati 38
8.2 Esperimenti sull'efficienza su sequenze random 43
8.2.1 Discussione 43

9 Conclusioni **49**

10 Ringraziamenti **51**

Bibliografia **55**

Sommario

Da sempre la ricostruzione della storia evolutiva degli esseri viventi del nostro pianeta è un argomento centrale nella ricerca scientifica. Per evidenziare le relazioni evolutive tra gli esseri viventi, si usano dei diagrammi detti alberi filogenetici. Inizialmente per costruire questo tipo di diagrammi si usavano degli approcci che si basavano sulle omologie tra le morfologie degli organismi da confrontare, mentre oggi si usano le sequenze di DNA dei vari organismi. Per la costruzione di questi alberi si analizzano le sequenze di DNA degli organismi che si vogliono confrontare. Esistono diversi modi per costruire gli alberi filogenetici partendo da sequenze di DNA, alcuni si basano sull'allineamento delle sequenze stesse, altri si basano sulla misura della similarità tra esse. Negli ultimi tempi sono stati fatti grossi progressi nel campo del sequenziamento del DNA, e quindi è disponibile una enorme quantità di materiale su cui si può basare l'analisi della storia evolutiva degli organismi. Questo enorme quantitativo di informazioni ha reso necessario l'uso di approcci più veloci per la comparazione di sequenze di DNA, rendendo i metodi di tipo alignment-based non adatti a tutti tipi di dataset che si vogliono analizzare, in quanto piuttosto lenti, sebbene molto precisi. Per questo motivo si sono introdotti dei metodi di tipo alignment-free, molto più veloci, sebbene meno accurati. In questa tesi si sono analizzati tre approcci di tipo alignment-free per trovare la similarità tra sequenze di DNA, *maxCor*, *avCor* e *kACS*. La particolarità di tali approcci è che si basano tutti su matching approssimato delle componenti. È stato inoltre realizzato un insieme di test per misurarne le performance su di un dataset di sequenze di DNA di tipo mitocondriale di 34 mammiferi. Gli esperimenti condotti su questo dataset hanno mostrato che l'introduzione di mismatch può essere efficacemente usata per effettuare analisi filogenetiche.

Capitolo 1

Introduzione

Il problema della ricostruzione della storia evolutiva degli organismi viventi è stato un argomento centrale nella ricerca scientifica sin dai tempi di Darwin. I primi approcci erano basati su informazioni riguardanti l'omologia della morfologia degli individui analizzati. Con l'avvento della biologia molecolare l'attenzione si è però spostata sull'analisi delle sequenze molecolari degli individui presi in considerazione. La comparazione di queste sequenze di caratteri che rappresentano genomi, si basa sull'allineamento delle sequenze stesse, in pratica si cerca di trovare omologie tra le sequenze prese in considerazione. Questo tipo di metodi per l'analisi delle sequenze è detto alignment-based. Esistono diversi algoritmi molto famosi e usati per trovare l'allineamento ottimo tra sequenze, due dei più famosi sono *Smith-Waterman*[5] e *Needleman-Wunsch*[6]. Purtroppo trovare l'allineamento ottimo tra sequenze si rivela essere un processo piuttosto lento e pesante computazionalmente, quindi si preferisce usare degli algorimi che fanno uso di euristiche, così da diminuire i tempi di elaborazione. Dei tool che effettuano l'allineamento tra sequenze sono ad esempio BLAST[7], FASTA[24] e ClustalW[10][23]. Negli ultimi anni sono state proposte diverse tecniche di tipo alignment-free, che invece di effettuare l'allineamento tra le sequenze, si propongono di misurarne la distanza[8]. In genere trovare l'allineamento tra due sequenze comporta l'utilizzo di una quantità di tempo proporzionale al prodotto delle lunghezze delle sequenze prese in considerazione, mentre molte delle tecniche alignment-free usano una quantità di tempo lineare rispetto alla taglia dell'input. Sebbene le tecniche alignment-free siano più veloci, queste hanno però il difetto di essere meno accurate delle tecniche alignment-based. L'obiettivo di questa tesi è il confronto delle tre misure *maxCor*, *avCor* e *kACS* che sono tre metodi alignment-free con mismatch di recente introduzione[2][3]. Per valutare la qualità di queste misure si sono misurate le similarità tra 34 sequenze di DNA di tipo mitocondriale, e da queste misure di similarità si sono poi creati

e valutati gli alberi filogenetici risultanti.

La tesi è così strutturata: nel Capitolo 2 viene introdotto il problema della ricostruzione degli alberi filogenetici, nel Capitolo 3 vengono descritti i tool usati durante i test effettuati, nel Capitolo 4 viene descritta la notazione e i concetti di base usati nella tesi. Nei Capitoli 5 e 6 vengono descritte le implementazioni delle misure *maxCor* e *avCor*. Infine nel Capitolo 7 viene descritta la misura *kACS* e nel Capitolo 8 vengono presentati e discussi i risultati del confronto sperimentale su sequenze di DNA mitocondriale e su sequenze random.

Capitolo 2

Analisi Filogenetica

Un albero filogenetico è un diagramma usato per rappresentare le relazioni evolutive che intercorrono tra diversi organismi. Un albero filogenetico è costituito da nodi, rami e foglie. Le foglie (o nodi esterni) rappresentano gli organismi di cui si vogliono rappresentare le relazioni di tipo evolutivo, mentre i nodi interni rappresentano i predecessori degli stessi ed infine i rami indicano le relazioni che intecorrono tra essi. Esistono diversi tipi di albero filogenetico, usati per rappresentare diversi tipi di relazione tra gli organismi presenti nel diagramma. In un *filogramma* si rappresentano le relazioni evolutive tra gli organismi presenti, in particolare i nodi esterni rappresentano gli organismi che si stanno prendendo in considerazione e i nodi interni invece rappresentano gli antenati comuni. In questo tipo di albero le lunghezze dei rami rappresentano la distanza evolutiva dei vari componenti del diagramma. Se al contrario la lunghezza dei rami non ha alcun significato, si ha un *cladogramma*. Gli alberi filogenetici sono suddivisi anche in altre due categorie, radicati e non radicati. In un albero radicato è presente un nodo particolare, detto radice, che rappresenta l'antenato comune a tutti gli organismi rappresentati. In un albero filogenetico radicato, i rami sono orientati in funzione del tempo. Un albero filogenetico non radicato descrive solo le relazioni evolutive tra gli oggetti, non fornendo alcuna informazione per quanto riguarda il processo evolutivo in termini del tempo. In Figura 2.1 si può osservare un esempio di un albero filogenetico non radicato.

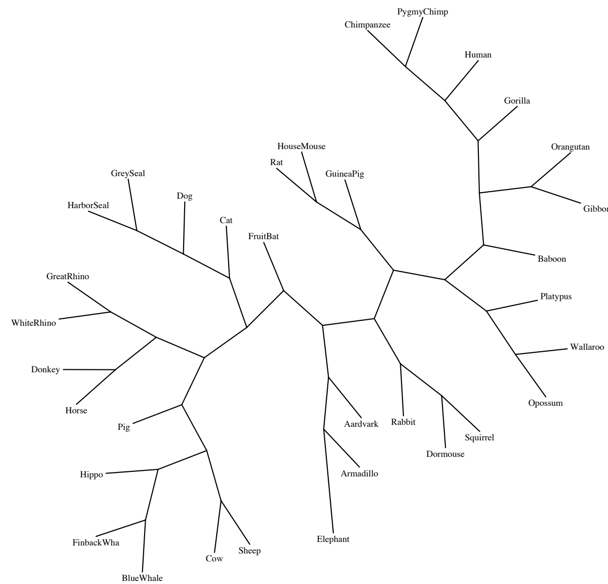


Figura 2.1: Esempio di albero filogenetico non radicato

2.1 Ricostruzione di Alberi Filogenetici

La costruzione di un albero filogenetico per un insieme di organismi può essere effettuata in base a diverse caratteristiche presenti negli individui stessi, come ad esempio alcune caratteristiche fisiche. La caratteristica presente in tutti gli organismi, e che offre il maggior numero di vantaggi è il DNA. Tra i vantaggi dell'utilizzo del DNA per la costruzione di un albero filogenetico, rispetto all'utilizzo delle omologie tra le morfologie degli organismi ci sono:

- la descrizione dei caratteri su cui si sta effettuando la costruzione dell'albero non è ambigua
- la somiglianza di diversi organismi dovuta a fattori ambientali (e quindi non di tipo genetico) non influisce nel processo di costruzione
- tutti gli organismi hanno DNA

Per questo motivo esistono diverse metodologie per la costruzione di alberi filogenetici basati sulle sequenze di DNA, i metodi basati sulla parsimonia, i metodi basati sulla verosimiglianza e i metodi basati sulla stima di distanze.

2.1.1 Metodo basato sulle distanze

Una categoria di metodi per la ricostruzione di alberi filogenetici è quella che fa uso della *distanza* tra le sequenze di DNA dei vari organismi presi in considerazione. Alla base di questi metodi c'è quindi la definizione di una quantità, la distanza $d(S_i, S_j) = d_{ij}$ tra tutte le coppie di un insieme di sequenze $(S_1, S_2, \ldots, S_N)$. In particolare la distanza d_{ij} deve avere delle caratteristiche ben precise:

- $d_{ij} \geq 0 \quad \forall i, j$
- $d_{ij} = 0 \iff i = j$
- $d_{ij} = d_{ji}$

Una volta definito il concetto di distanza, questa viene calcolata per ogni coppia di sequenze e con questi valori viene quindi costruita una matrice delle distanze, che viene usata per la costruzione dell'albero filogenetico. La matrice delle distanze ha le proprietà di essere simmetrica e quadrata di lato N, ovvero la cardinalità dell'insieme delle sequenze di DNA che si sta prendendo in considerazione. Con la matrice delle distanze si può quindi costruire un albero filogenetico mediante diverse tecniche, come ad esempio *UPGMA* e *Neighbor-Joining*[12].

UPGMA

UPGMA ovvero *Unweighted Pair Group Method with Arithmetic mean* è uno dei più semplici metodi per effettuare una clusterizzazione. Questo approccio usa un algoritmo iterativo che procede associando ad ogni iterazione le sequenze più simili. L'algoritmo funziona iterando i seguenti passi:

- identificare la distanza minima tra tutte le possibili coppie di sequenze
- formare un cluster con la coppia identificata al punto precedente
- ricalcolare la matrice delle distanze, definendo come distanza dal cluster la media delle distanze da ciascuna sequenza del cluster
- ricominciare l'iterazione cercando la nuova distanza minima tra tutte le possibili coppie di sequenze

In tutto vengono quindi effettuate $N - 1$ iterazioni, dove N è il numero delle sequenze.

Neighbor-Joining

Un altro metodo basato sulla matrice delle distanze per la costruzione di alberi filogenetici è Neighbor-Joining. Questo approccio per il clustering è di tipo bottom-up. Anche Neighbor-Joining è un algoritmo di tipo iterativo, in particolare effettua $N-3$ iterazioni, dove N è il numero delle sequenze che fanno parte del dataset preso in considerazione. L'algoritmo funziona iterando i seguenti passi:

- per ogni sequenza i calcolare $r_i = \sum_{k \neq i} d_{ik}$
- scegliere la coppia di sequenze (i,j) per cui il valore $d_{ij} - \frac{r_i + r_j}{N-2}$ è minimo
- unire i e j in un cluster (i,j) in un nodo dell'albero e calcolare la lunghezza dei rami che collegano i e j a questo nuovo nodo (i,j)

$$d_{i,(ij)} = \frac{1}{2} d_{ij} + \frac{r_i - r_j}{2(N-2)} \quad d_{j,(ij)} = \frac{1}{2} d_{ij} + \frac{r_j - r_i}{2(N-2)}$$

- calcolare la distanza del nuovo cluster da ogni altro cluster k

$$d_{(ij)k} = \frac{d_{ik} + d_{jk} - d_{ij}}{2}$$

- sostituire il i e j con il nuovo cluster (i,j)

In Figura 2.2 è riportato un diagramma generale di come *Neighbor-Joining* funziona.

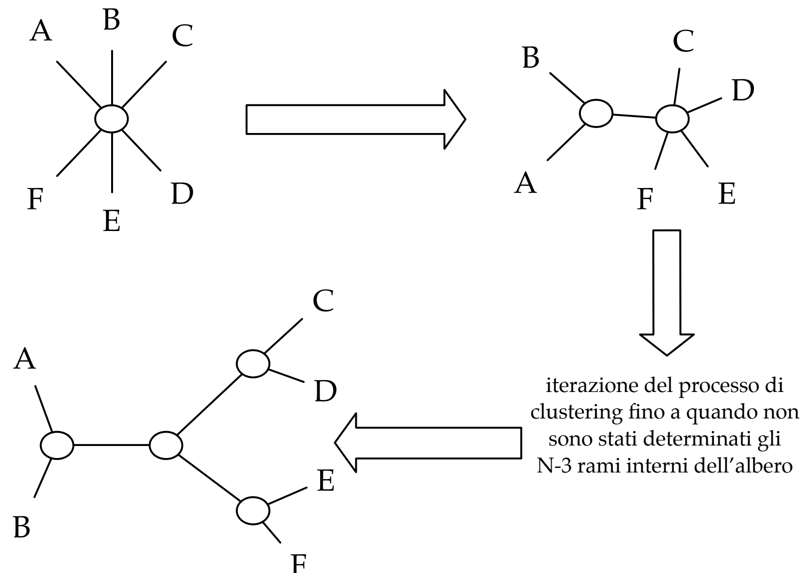


Figura 2.2: Esempio algoritmo Neighbor-Joining

2.1.2 Metodo della massima parsimonia

Uno dei più grossi difetti dei metodi di costruzione basati sulla matrice delle distanze è che l'estrazione dell'informazione riguardante la distanza tra le coppie del dataset comporta una perdita di informazione. Per evitare questa perdita di informazione si può operare sulle sequenze stesse piuttosto che sulle distanze, che è l'idea su cui si basa il metodo della massima parsimonia. Questo approccio stabilisce la lunghezza di un ramo come il numero minimo di sostituzioni occorse tra i nodi che congiunge. Questo metodo consiste nei seguenti step:

- selezionare i siti informativi. Un sito è informativo se favorisce uno o più alberi tra tutti i possibili e se contiene almeno due differenti caratteri, ciascuno dei quali è presente almeno in due sequenze

- una volta selezionati i siti informativi, si calcola il numero minimo di sostituzioni richieste da ciascuno dei possibili alberi senza radice che descrivono le relazioni filogenetiche tra le sequenze in esame

- l'albero (o gli alberi) di massima parsimonia è quello che richiede il numero minimo di sostituzioni totalizzate fra tutti i siti informativi considerati

Questo metodo ha delle limitazioni, in quanto non tiene conto delle sostituzioni parallele, convergenti o multiple, che in pratica sono molto frequenti. Inoltre tutte le sostituzioni sono considerate equivalenti, fatto che non corrisponde ad un reale processo evolutivo.

2.1.3 Metodo della massima verosimiglianza

Il metodo della massima verosimiglianza usa direttamente le sequenze per la costruzione dell'albero, così come il metodo della massima parsimonia. Questo metodo cerca di quantificare la probabilità che un albero filogenetico corrisponda ad un allineamento multiplo. Quindi l'albero che ottiene il massimo valore di probabilità rappresenta la stima di massima verosimiglianza tra le sequenze considerate. Una grossa limitazione di questo metodo è l'elevato carico computazionale.[21]

2.2 Metodi Alignment-free

Come è già stato riportato nell'introduzione, negli ultimi vent'anni sono state proposte diverse tecniche di tipo alignment-free, in quanto sono molto più

veloci sebbene meno accurate. In particolare le tecniche alignment-based hanno il grosso svantaggio di non scalare molto bene all'aumentare della taglia dell'input[17]. Inoltre sono molto sensibili ai rearrangement e alle mutazioni che le sequenze di DNA tendono naturalmente a presentare. Per questo motivo le tecniche di tipo alignment-free sono molto importanti.

Metodo basato sulla frequenza degli l-mer

Questa prima categoria di metodi di tipo alignment-free si basa sul conteggio della frequenza degli l-mer nelle stringhe che si stanno analizzando, senza tenere in considerazione la posizione degli l-mer nelle stesse. I vantaggi principali nell'utilizzo di questo tipo di metodi sono il fatto che usano tempo lineare nella taglia dell'input. L'idea che sta alla base di questo metodo è quella di creare un vettore di frequenze dei vari l-mer per le stringhe date in ingresso, e successivamente calcolare la distanza tra vettori corrispondenti alle sequenze in ingresso

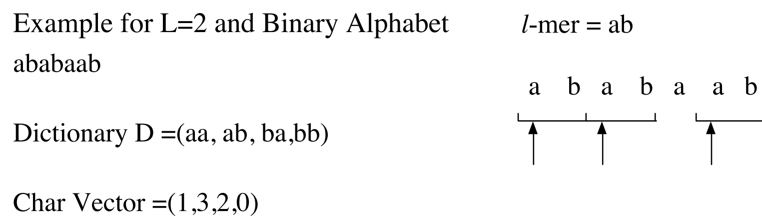


Figura 2.3: Esempio di approccio basato su l-mer con match esatto[25]

In Figura 2.3 un esempio di estrazione delle frequenze degli l-mer esatti dalla stringa in ingresso, in particolare nella figura sono evidenziate le posizioni in cui si trovano i vari 2-mer *ab*.

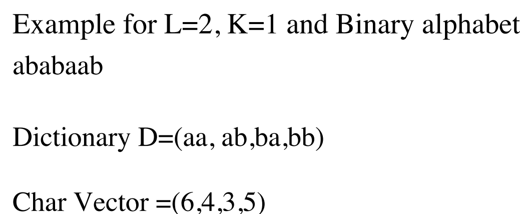


Figura 2.4: Esempio di approccio basato su l-mer con mismatch[25]

Questo tipo di approccio è stato modificato [25] trovando le frequenze non degli l-mer esatti, ma trovando gli l-mer con permessi un certo numero di mismatch, questo approccio è stato applicato nell'ambito della classificazione

di proteine. In Figura 2.4 si vede un esempio di estrazione delle frequenze di l-mer, con $l = 2$ e permettendo $k = 1$ mismatch.

ACS

Un approccio basato sulle sottostringhe è *ACS* ovvero *Average Common Substring* che è stato proposto in [4]. L'idea che sta alla base di questo approccio è quella di trovare per ogni posizione i di una sequenza, la più lunga sottostringa che parte da una posizione j nella seconda sequenza. Questo approccio si basa sul match esatto tra le sottostringhe delle due sequenze prese in esame. Nel Capitolo 7 verrà data una spiegazione molto più dettagliata di questo approccio.

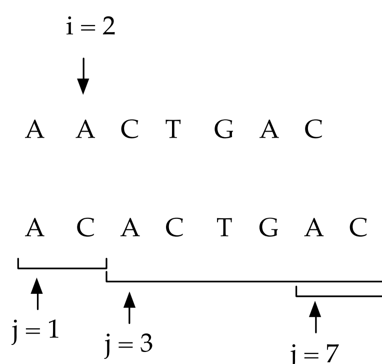


Figura 2.5: Esempio ACS

In Figura 2.5 si vede come per la posizione $i = 2$ della prima stringa vengono trovati 3 match esatti nella seconda, in $j = 1, 3, 7$. In particolare il valore di ACS per la posione $i = 2$ varrà 6 in quanto lunghezza della più lunga sottostringa con match esatto nella seconda sequenza.

MaxCor e *AvCor*

Un primo approccio basato sulle sottostringhe è *maxCor* e *AvCor* come proposto in [2]. L'idea che sta alla base di *MaxCor* è quella di trovare la massima sottostringa con k mismatch che è possibile ottenere confrontando due strighe x e y facendo partire il confronto da ogni possibile posizione di i in x e j in y. L'idea che sta alla base di *AvCor* invece si basa sul calcolo della media di ogni possibile sottostringa contenente k mismatch che si ottiene confrontando le due stringhe x e y partendo da ogni possibile posizione i in x e j in y. Entrambe queste misure verrranndo approfondite nei Capitoli 4,5 e 6.

kACS

Un altro approccio che si basa sulle sottostringhe è *kACS*, proposto in [3]. Questo approccio è una generalizzazione dell'approccio ACS presentato nella sottosezione precedente, con la differenza che per ogni posizione i della prima sequenza, si andrà a trovare la più lunga sottostringa nella seconda sequenza, questa volta permettendo al massimo k mismatch. Anche per questo approccio verrà data una descrizione più approfondita nel Capitolo 7.

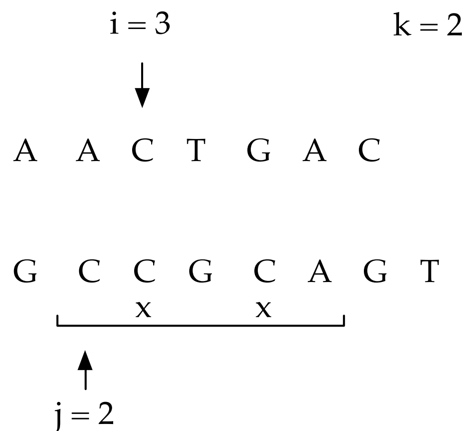


Figura 2.6: Esempio *kACS* con $k = 2$

In Figura 2.6 si vede come per la posizione $i = 3$ nella prima sequenza, viene trovata una sottostringa nella seconda a partire dalla posizione $j = 2$ con esattamente 2 mismatch.

k-LCF

Un altro approccio basato sulle sottostringhe è k-LCF ovvero *Longest Common Substring with k mismatch* [1]. Questo approccio è una generalizzazione del problema LCS. L'idea che sta alla base di questo approccio è di trovare la più lunga sottostring in comune tra due stringhe permettendo al massimo k mismatch, ovvero la distanza di Hamming tra le sue sottostringhe deve essere minore di k. Sempre in [1] viene presentato un algoritmo che impiega un tempo proporzionale al prodotto delle lunghezze delle sequenze in ingresso.

Capitolo 3

Strumenti software utilizzati

In questa sezione vengono illustrati gli strumenti software che sono stati usati per lo sviluppo della tesi.

3.1 Valgrind

Valgrind [13] è un software per l'analisi del codice sorgente di un programma. In particolare i tool messi a disposione permettono di analizzare la memoria utilizzata, evitando quindi memory leak che porterebbero ad un peggioramento delle performance. Un altro strumento molto utile è il profiler, che permette all'utente di capire quali sono le parti del sorgente che sono più pesanti computazionalmente. Questo strumento in particolare è stato utile nel capire il collo di bottiglia della versione subquadratica dell'algoritmo. Con l'analisi effettuata con il profiler infatti è emerso che il collo di bottiglia in quest'ultimo caso sia la fase di preprocesing dell'algoritmo. In Figura 3.1 e 3.2 si vedono due screenshot del tool valgrind alla fine di una analisi delle prestazioni del codice sorgente.

Function	Location	Called	Self Cost: I	Incl. Cost: Ir
SQMaxCorr::align(int, int, int)	/home/...	33121	30.539.998.311	141.163.795.484
SQMaxCorr::computeMaxCorrAlignment(int)	/home/...	33121	18.423.385.866	21.030.526.231
void __gnu_cxx::new_allocator<int>::construct<int, ...	/usr/in...	401648016	8.836.256.352	14.057.680.560
std::string::at(unsigned long)	/usr/lib...	551627360	8.826.037.766	8.826.044.373
std::_Deque_iterator<int, int&, int*>::difference_ty...	/usr/in...	229299155	8.484.068.735	12.382.154.370
SQMaxCorr::reinitTheHouds()	/home/...	33121	6.126.954.427	6.126.954.427
std::deque<int, std::allocator<int> >::push_back(int ...	/usr/in...	200824008	5.409.828.576	13.243.363.802
std::vector<int, std::allocator<int> >::push_back(int ...	/usr/in...	200824008	5.220.146.400	21.648.019.281
std::deque<int, std::allocator<int> >::pop_front()	/usr/in...	200824008	5.006.628.105	6.588.311.542
int const& std::forward<int const&>(std::remove_re...	/usr/in...	805008213	4.830.049.278	4.830.049.278

Figura 3.1: Screen-shot valgrind

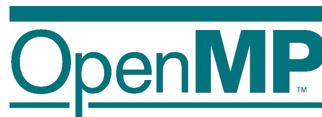
Valgrind Function Profiler — Instruction read acce... — Filter...

Callee	Calls	Cost
std::vector<int, std::allocator<int> >::push_back(int const&)	200824008	21.648.019.281
std::queue<int, std::deque<int, std::allocator<int> > >::size() const	229299155	17.426.735.780
std::queue<int, std::deque<int, std::allocator<int> > >::push(int co...	200824008	15.653.251.898
std::queue<int, std::deque<int, std::allocator<int> > >::front()	200824008	13.254.384.528
std::string::at(unsigned long)	551627360	8.826.044.373
std::queue<int, std::deque<int, std::allocator<int> > >::pop()	200824008	8.395.727.614

Figura 3.2: Screen-shot valgrind

In Figura 3.1 si vede ad esempio come il profiler segnali che il metodo *align* sia molto pesante in termini di tempo nella computazione totale.

3.2 OpenMP



OpenMP [15] è un API multipiattaforma per la creazione di applicazioni parallele a memoria condivisa. In particolare è stato usato per lanciare in parallelo diversi job in modo da ridurre il tempo totale di esecuzione dei vari test effettuati. Da sottolineare che non è stato usato per velocizzare la singola esecuzione.

3.3 Phylip

Phylip [9] è un pacchetto di trentacinque software scritti in linguaggio C forniti dall'Università di Washington. I tool che sono stati usati in questa tesi sono:

- NEIGHBOR: è un tool che implementa l'algoritmo *Neighbor Joining Method* [11] e *UPGMA*. Questo tool è stato usato per la costruzione degli alberi filogenetici data la matrice delle distanze calcolate mediante l'uso della misura *maxCor*, *avCor* e *kACS*. In Figura 3.3 uno screenshot del tool neighbor da riga di comando, in cui sono visibili i parametri usati.

```
Neighbor-Joining/UPGMA method version 3.696

Settings for this run:
  N       Neighbor-joining or UPGMA tree?  Neighbor-joining
  O                        Outgroup root?  No, use as outgroup species  1
  L         Lower-triangular data matrix?  No
  R         Upper-triangular data matrix?  No
  S                        Subreplicates?  No
  J     Randomize input order of species?  No. Use input order
  M           Analyze multiple data sets?  No
  0   Terminal type (IBM PC, ANSI, none)?  ANSI
  1    Print out the data at start of run  No
  2  Print indications of progress of run  Yes
  3                        Print out tree  Yes
  4       Write out trees onto tree file?  Yes


_ Y to accept these or type the letter for one to change
```

Figura 3.3: Screenshot del tool neighbor della libreria Phylip

- TREEDIST: è un tool che calcola la distanza tra alberi filogenetici. In particolare è stata usata la misura della distanza *Robinson-Foulds* come definita in [22]. In Figura 3.4 uno screenshot del tool treedist da riga di comando, in cui sono visibili i parametri usati.

```
Tree distance program, version 3.696

Settings for this run:
 D                         Distance Type:  Symmetric Difference
 R          Trees to be treated as Rooted:  No
 T    Terminal type (IBM PC, ANSI, none):  ANSI
 1  Print indications of progress of run:  Yes
 2                 Tree distance submenu:  Distance between adjacent pairs

Are these settings correct? (type Y or the letter for one to change)
```

Figura 3.4: Screenshot del tool treedist della libreria Phylip

Capitolo 4

Notazione e concetti base

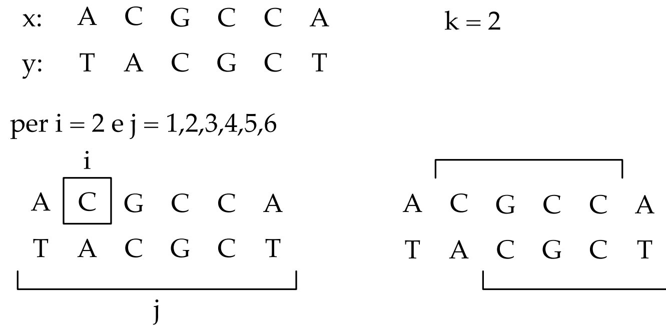
In questo capitolo viene introdotta la notazione che verrà usata nelle definizioni delle misure e nel resto della tesi. Siano x e y due sequenze di caratteri definite su di un alfabeto Σ. Sia $x[i]$ l'i-esimo elemento di x e si indichi con $|x| = n$ la lunghezza della sequenza x. Inoltre viene indicato con $x[i \ldots j]$ la sottostringa contigua di x che parte dalla posizione i e termina alla posizione j. In particolare $x[i \ldots |x|]$ è definito come l'i-esimo suffisso di x. Sia infine k un numero intero molto minore di $\log n$. Sia $COR(i, j), i = 1, 2, \ldots, n; j \geq i$ la lunghezza della più lunga sottostringa che inizia alla posizione i in x e che può essere copiata partendo dalla posizione j in y permettendo esattamente k mismatch. In [2] sono state introdotte due misure di correlazione sulla base delle quali è possibile definire delle misure di similarità tra due sequenze tenendo conto di un numero fissato di mismatch:

- $MaxCor(i)$,$i = 1, 2, \ldots, n$ è definita come il massimo valore assunto da $COR(i, j)$ per i fissato e per ogni possibile j. In Figura 4.1 si vede il valore di $maxCor(2)$. Il confronto viene effettuato fissando per la sequenza x, $i = 2$ e facendo variare per la sequenza y, $j = 1, \ldots, 6$. Si vede facilmente che COR(2,3) assume valore massimo, quindi il valore di $maxCor(2)$= 4.

- $AvCor(i)$,$i = 1, 2, \ldots, n$ è definita come il valore medio assunto da $COR(i, j)$ per ogni i fissato e per ogni possibile j

A partire dai due vettori di correlazione $MaxCor(i)$ e $AvCor(i)$ è possibile definire diverse misure di similarità:

- $MaxCor$ è definita come il massimo valore che COR assume per ogni $i \in (1, 2, \ldots, n)$ in x e $j \in (i, i+1, \ldots, n)$ in y. In Figura 4.2 si vedono tutti i $maxCor(i)$ $i = 1, \ldots, 6$, e per ogni $maxCor(i)$ tutti i COR(i, j)



maxCor(2) = max{COR(2,1),COR(2,2),COR(2,3),COR(2,4),COR(2,5),COR(2,6)} =
= max{3,3,4,2,2,1} = 4

Figura 4.1: Esempio di $maxCor(2)$

per ogni i fissato e $j = 1, \ldots, 6$. Il valore di $maxCor$ è quindi il massimo tra tutti i $maxCor(i)$ e quindi il suo valore nell'esempio riportato vale 5.

- $AvCor$ è definita come il valore medio che COR assume per ogni $i \in (1, 2, \ldots, n)$ in x e $j \in (i, i+1, \ldots, n)$ in y. In Figura 4.2 si può facilmente verificare che il valore di $AvCor$ vale 2.1 .

- $kACS$ è definita come il valore medio che $MaxCor(i)$ assume per $i \in (1, 2, \ldots, n)$ In Figura 4.2 si può facilmente verificare che il valore di $kACS$ vale 3.

x: A C G C C A   k = 2

y: T A C G C T

maxCor(1)	maxCor				
A C G C C A	A C G C C	A C G C	A C G	A C	A
T A C G C T	A C G C T	C G C T	G C T	C T	T
COR(1,1) = 3	COR(1,2) = 5	COR(1,3) = 2	COR(1,4) = 3	COR(1,5) = 2	COR(1,6) = 1
maxCor(2)					
C G C C A	C G C C A	C G C C	C G C	C G	C
T A C G C	A C G C T	C G C T	G C T	C T	T
COR(2,1) = 3	COR(2,2) = 3	COR(2,3) = 4	COR(2,4) = 2	COR(2,5) = 2	COR(2,6) = 1
maxCor(3)					
G C C A	G C C A	G C C A	G C C	G C	G
T A C G	A C G C	C G C T	G C T	C T	T
COR(3,1) = 3	COR(3,2) = 3	COR(3,3) = 3	COR(3,4) = 3	COR(3,5) = 2	COR(3,6) = 1
maxCor(4)					
C C A	C C A	C C A	C C A	C C	C
T A C	A C G	C G C	G C T	C T	T
COR(4,1) = 2	COR(4,2) = 3	COR(4,3) = 3	COR(4,4) = 3	COR(4,5) = 2	COR(4,6) = 1
maxCor(5)					
C A	C A	C A	C A	C A	C
T A	A C	C G	G C	C T	T
COR(5,1) = 2	COR(5,2) = 2	COR(5,3) = 2	COR(5,4) = 2	COR(5,5) = 2	COR(5,6) = 1
maxCor(6)					
A	A	A	A	A	C
T	A	C	G	C	T
COR(6,1) = 1	COR(6,2) = 1	COR(6,3) = 1	COR(6,4) = 1	COR(6,5) = 1	COR(6,6) = 1

Figura 4.2: Esempio di *maxCor*

Capitolo 5

Misura di similarità basata su massima correlazione

In questo capitolo verrà descritta la procedura per ricavare la misura *maxCor* come è già stata precedentemente definita nel capitolo 4. Di seguito viene inoltre descritto sia il metodo quadratico, che quello subquadratico per ricavare *maxCor*[2], e verranno fornite e commentate delle porzioni di pseudocodice.

5.1 MaxCor versione quadratica

I passi per ricavare la misura *maxCor* date le due stringhe x e y sono i seguenti.

1. produciamo i vettori binari ottenuti dalla sovrapposizione di y su x cominciando dalle posizioni $1, 2, \ldots, n$.

2. chiamiamo ora il generico vettore ottenuto al passo precedente CORR e LENGTH un altro vettore ausiliario della sua stessa lunghezza. Partendo dalla prima posizione del vettore CORR, troviamo la lunghezza della più lunga stringa w con k mismatch e la annotiamo in LENGTH(1). Sia inoltre u il primo run di uni in w.

3. osserviamo che data la scelta fatta di w possiamo dire che

   $$\text{CORR}(|w| + 1) = \text{CORR}(\text{LENGTH}(1) + 1)) = 0$$

   in quanto la stringa w contiene k mismatch e quindi CORR($|w|$+1) deve essere necessariamente un mismatch. A questo punto si può dedurre che LENGTH(2),LENGTH(3), ...,LENGTH($|u| + 1$) sono decrementi

unitari di LENGTH(1). Ora è possibile calcolare LENGTH($|u| + 2$) semplicemente spostando il puntatore sinistro dopo il primo zero che cade all'interno di w e spostando il puntatore destro al prossimo zero che viene trovato all'interno di CORR.

4. continuare la scansione in modo da calcolare tutti i rimanenti valori di LENGTH.

5. ripetere il processo appena descritto per tutti i vettori CORR e ritornare il massimo valore ottenuto.

5.1.1 Pseudocodice

Di seguito si riporta lo pseudocodice della funzione MaxCor(x, y, k)che effettua per il calcolo di *maxCor* date le due stringhe in ingresso e il numero di mismatch permessi. Si suppone che x sia la sequenza di lunghezza maggiore, y sia la sequenza di lunghezza minore ed infine k sia il numero di mismatch permessi. La funzione MaxCor non fa altro che trovare ogni possibile CORR(i) spostando prima a sinistra la stringa più lunga e mantenendo fissa la stringa più corta e successivamente facendo il contrario ovvero fissando la stringa più lunga e facendo scorrere a sinistra la stringa più corta, e salvando allo stesso tempo il valore massimo di CORR(i) ritornandolo poi alla fine della procedura. La funzione computeCORR prende in ingresso le due stringhe, gli indici da cui far partire il confronto per ciascuna di esse e la lunghezza del confronto stesso che si vuole effettuare. Invece di salvare in CORR il risultato di tutti i confronti dei singoli caratteri delle due stringhe, si è preferito salvare gli indici in cui si trovano i mismatch. In questo modo si ha un vettore mediamente di lunghezza minore, ottenendo un leggero miglioramento in termini di tempi di esecuzione. Da notare che nel caso peggiore, ovvero quando le due stringhe sono composte totalmente da caratteri diversi, questa variazione non ha effetti in termini di velocità di esecuzione, in quanto si otterrà un vettore CORR di lunghezza uguale a quella dell'allineamento che si sta effettuando. La funzione computeMaxCor infine, dato il vettore contenente gli indici dei mismatch corrispondenti all'allineamento corrente e la lunghezza dell'allineamento stesso, fornisce in output il valore massimo di CORR(i). In particolare nella funzione si controlla che esistano mismatch nell'allineamento, perchè in caso contrario si può ritornare direttamente la lunghezza massima. Successivamente si passa all'inizializzazione delle variabili necessarie e al calcolo della lunghezza del primo allineamento con k mismatch. Una volta ottenuto il primo allineamento, facendo puntare il puntatore destro e sinistro alla stessa posizione e poi spostando il destro di

k posizioni, si prosegue calcolando la lunghezza dei restanti spostando a destra il puntatore sinistro fino al primo mismatch all'interno dell'allineamento corrente e spostando a destra il puntatore destro fino al primo mismatch all'esterno dell'allineamento corrente. Questo shift dei puntatori destro e sinistro viene effettuato spostando semplicemente a destra di una posizione entrambi i puntatori e leggendo il valore contenuto nel vettore, questo è possibile perchè si è precedentemente salvata la posizione dei mismatch. Prima di effettuare il calcolo della lunghezza di ogni allineamento viene controllata la validità della posizione del puntatore destro, in quanto potrebbe trovarsi in una posizione non valida a causa della variazione della modalità con cui viene scansionato il vettore CORR, che ora contiene gli indici dei mismatch e non una serie di zeri e uni in base ai match o mismatch dei caratteri delle due sequenze date in ingresso. In Figura 5.1 un esempio di come vengono calcolati alcuni allineamenti parziali, permettendo un numero di mismatch pari a 2. In particolare si vede come ad ogni posizione venga trovata la sottostringa più lunga che parte da una certa posizione e che contiene esattamente 2 mismatch. Come si nota facilmente molti allineamenti parziali sono inutili per quanto riguarda l'obiettivo finale di trovare l'allienamento più lungo. Nella sezione 4 verrà spiegato un metodo per evitarli.

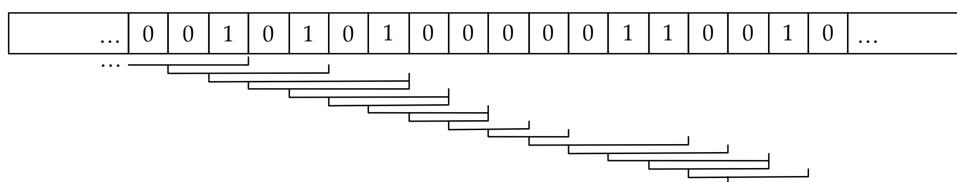


Figura 5.1: Esempio calcolo allineamenti parziali usando il metodo quadratico

COMPUTECORR($x, y, idxX, idxY, lenCORR$)

1 ▷ sia CORR un vettore vuoto
2 **for** $i \leftarrow 0$ **to** $lenCORR$
3   **do**
4   **if** $x[idxX] \neq y[idxY]$
5   **then**
6               $push(CORR, i)$
7
8         $idxX \leftarrow idxX + 1$
9         $idxY \leftarrow idxY + 1$
10
11 **return** $CORR$

```
computeMaxCor(CORR,lenCORR)
 1  if size(CORR) ≤ 0
 2     then
 3          return lenCorr
 4
 5  ▷ inizializzazione
 6  maxCorTmp ← 0
 7  idxFirstZero ← 0   firstZero ← CORR[idxFirstZero]
 8  idxLastZero ← 0   firstPtr ← 0   lastZero ← CORR[idxLastZero]
 9  nmm ← 1
10
11  ▷ trovo la lunghezza del primo allineamento
12  while nmm ≤ k and idxLastZero ≤ size(CORR)
13        do
14          idxLastZero ← idxLastZero + 1
15          lastZero ← CORR[idxLastZero]
16          nmm ← nmm + 1
17
18  if idxLastZero + 1 ≥ size(CORR)
19     then
20          lastPtr ← lenCORR
21     else
22          lastPtr ← CORR[idxLastZero + 1]
23
24  len ← lastPtr − leftPtr
25  maxCorTmp ← max(maxCorTmp, len)
26
27  ▷ trovo la lunghezza degli altri allineamenti
28  while lastPtr < lenCORR
29        do
30          firstPtr ← firstZero + 1
31          idxFirstZero ← idxFirstZero + 1
32          firstZero ← CORR[idxFirstZero]
33          idxLastZero ← idxLastZero + 1
34          if idxLastZero + 1 ≥ size(CORR)
35             then
36                  lastPtr ← lenCORR
37             else
38                  lastPtr ← CORR[idxLastZero + 1]
39
40          len ← lastPtr − firstPtr
41          maxCorTmp ← max(maxCorTmp, len)
42
43  return maxCorTmp
```

```
MaxCor(x, y, k)
 1  ▷ sia x la sequenza di lunghezza maggiore
 2  ▷ sia y la sequenza di lunghezza minore
 3  ▷ scorro a sinistra x
 4  lenX ← len(x)
 5  lenY ← len(y)
 6  maxCor ← 0
 7  for i ← 0 to lenX
 8        do
 9            lenCORR ← min(lenX − i, lenY)
10            if lenCORR ≤ maxCor
11               then
12                     break
13
14            computeCORR(x, y, i, 0, lenCORR)
15            tmpCor ← computeMaxCor(CORR, lenCORR)
16            maxCor ← max(maxCor, tmpCor)
17
18  ▷ scorro a sinistra y
19  for i ← 1 to lenY
20        do
21            lenCORR ← lenY − i
22            if lenCORR ≤ maxCor
23               then
24                     break
25
26            CORR ← computeCORR(x, y, 0, i, lenCORR)
27            tmpCor ← computeMaxCor(CORR, lenCORR)
28            maxCor ← max(maxCor, tmpCor)
29
30  return maxCor
```

5.2 MaxCor versione subquadratica

In questa sezione si descrive una versione subquadratica per il calcolo della misura *maxCor*. Come per la versione quadratica, anche in questo caso si vuole trovare il massimo valore assunto da COR su tutte le coppie (i, j) con $i(1, 2, \dots, n)$ in x e $j(i, i+1, \dots, n)$ in y. Dato il generico vettore CORR, questa volta contenente il risultato della comparazione dei singoli caratteri delle due sequenze, si vuole quindi calcolare il segmento più lungo contenen-

te esattamente k mismatch. Dato quindi il vettore CORR, il segmento che stiamo cercando può essere interamente contenuto in un blocco, oppure può cominciare in un blocco e terminare al di fuori dello stesso. Se il segmento inizia e finisce nello stesso blocco, il blocco deve contenere almeno k mismatch. Un blocco con almeno k mismatch è detto *denso*. L'approccio subquadratico si basa sull'osservazione che il numero di blocchi di taglia $\log n$ è esattamente n. Questa importante osservazione ci porta ad effettuare un preprocessing del vettore CORR, così da ottenere delle informazioni che ci permettono di velocizzare i tempi di esecuzione.

5.2.1 Preprocessing

Una volta calcolato il vettore CORR, si procede con il preprocessing accennato nella sezione precedente. In particolare, per ogni blocco si annotano i primi e gli ultimi k mismatch che si incontrano nello stesso, ed inoltre se il blocco che si sta analizzando è denso, si procede anche con il calcolo della lunghezza del segmento contenente k mismatch al suo interno. Si nota che l'analisi di ciascun blocco comporta un carico di lavoro dell'ordine di grandezza $O(\log n)$. Per come è stata implementata la parte di preprocessing, essa esegue in tempo quadratico rispetto alla taglia delle sequenze in input. Nella fase di analisi delle prestazioni quindi il tempo impiegato in questa fase non verrà preso in considerazione.

5.2.2 Processing

A questo punto il vettore CORR è scansionato da sinistra a destra in blocchi di $\log n$ bit alla volta. In corrispondenza del generico blocco, viene calcolata la lunghezza del più lungo segmento contenente k mismatch partendo all'interno del blocco alla posizione dei $k-h, k-i-1, \ldots,$ zeri posizionati più a destra. Con $h \leq k$ è il più piccolo valore per cui tali zeri sono stati trovati all'interno del blocco. A questo punto il calcolo del segmento più lungo consiste tipicamente nella scansione di un certo numero di blocchi non densi. Una volta trovata una sequenza con k mismatch si può far avanzare sia il puntatore destro che quello sinistro alla posizione successiva. Nel caso il $k-$esimo zero della sequenza facesse parte di un blocco denso, si procede con lo spostamento del puntatore sinistro alla posizione del $k-$esimo mismatch a destra del blocco e spostando il puntatore destro in accordo con questo salto. Questo salto è possibile effettuarlo in quanto la lunghezza del segmento più lungo contenuto nel blocco denso che si è raggiunto è noto in quanto calcolato nella fase di preprocessing. Per la scansione di ogni blocco si spende

$O(k)$, quindi in totale per un vettore CORR si spende $O(kn/\log n)$, da cui la complessità finale di $O(kn^2/\log n)$.

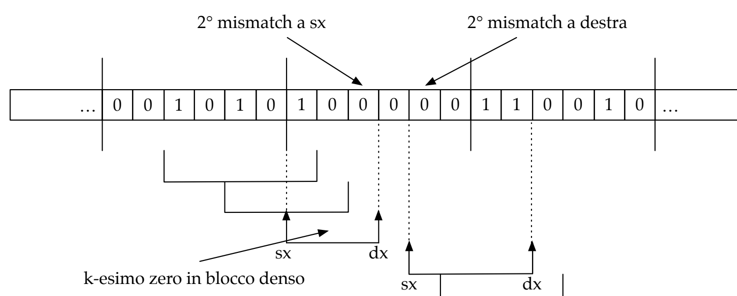


Figura 5.2: Esempio salto in blocco denso

In Figura 5.2 una esemplificazione di come avviene il salto quando il $k-$esimo zero di una sequenza cade all'interno di un blocco denso. In particolare in Figura 5.2 si permettono 2 mismatch. Si vede dalla figura come una volta che il k-esimo mismatch entra nel blocco denso, il calcolo della successiva sottostringa più lunga contenente k mismatch parta dal k-esimo zero a destra del blocco stesso.

Un accorgimento adottato per diminuire i tempi di esecuzione, sia per quanto riguarda la versione quadratica, che per quella subquadratica, è il seguente: se la lunghezza dell'allineamento residuo è minore del valore di *maxCor* attualmente trovato, si può saltare alla scansione del vettore CORR successivo.

Capitolo 6

Misura di similarità basata su correlazione media

In questa sezione viene presentata la misura *avCor* e la sua implementazione sia nella versione quadratica che subquadratica.

6.1 AvCor versione quadratica

La misura *avCor* è definita come già specificato nel capitolo 4 come la media di tutti i valori asssunti da COR per tutti i possibili valori di $i \in (1, 2, \ldots, n)$ in x e $j \in (i, i+1, \ldots, n)$ in y. I passi per calcolare la misura *avCor* di due sequenze in ingresso, sono gli stessi che si vengono effettuati per il calcolo di *maxCor*, ma con delle lievi accortezze. In particolare viene introdotta una variabile inizializzata a zero a cui verranno sommati tutti i valori di COR ottenuti durante tutta l'elaborazione. Oltre alla somma di tutti i valori di CORR, viene anche salvato un contatore per il successivo calcolo della media. Alla fine dell'elaborazione viene semplicemente restituita la media con i valori che sono stati salvati durante l'esecuzione.

6.1.1 Pseudocodice

Di seguito viene fornito un frammento di pseudocodice in cui viene sottolineata la differenza con il procedimento descritto per il calcolo di *maxCor*.

```
AvCor(x, y, k)
 1  ▷ sia x la sequenza di lunghezza maggiore
 2  ▷ sia y la sequenza di lunghezza minore
 3  ▷ scorro a sinistra x
 4  lenX ← len(x)
 5  lenY ← len(y)
 6  avCor ← 0
 7  for i ← 0 to lenX
 8        do
 9           lenCORR ← min(lenX − i, lenY)
10           computeCORR(x, y, i, 0, lenCORR)
11           computeAvCor(CORR, lenCORR)
12
13  ▷ scorro a sinistra y
14  for i ← 1 to lenY
15        do
16           lenCORR ← lenY − i
17           CORR ← computeCORR(x, y, 0, i, lenCORR)
18           computeAvCor(CORR, lenCORR)
19
20  avCor ← avCor/n
21  return avCor
```

La funzione AvCor è molto simile alla funzione MaxCor. Da sottolineare che sono stati tolti i *break* altrimenti verrebbero escluse delle sequenze dal calcolo. Inoltre n rappresenta il numero di sequenze trovate, questo valore viene calcolato incrementando una variabile ogni volta che viene trovata una sequeza con k mismatch.

Nel frammento di pseudocodice di computeAvCor è stato messo in evidenza come viene incrementato il valore di *avCor* con le lunghezze delle varie sequenze contenenti k mismatch come verrà poi spiegato nella sezione 6.2.2. La variabile *avCor* in questo caso non viene esplicitamente passata come parametro, in quanto può essere considerata una variabile globale.

Alla fine dei due cicli *for* presenti in AvCor la variabile *avCor* conterrà quindi la somma di tutte le lunghezze di tutte le sequenze contenenti esattamente k mismatch.

```
COMPUTEAVCOR(CORR,LENCORR)
 1  ...
 2  len ← lastPtr − leftPtr
 3  h ← firstZero − firstPtr
 4  if h > 0
 5     then
 6          avCor ← avCor + len(len + 1)/2 − ((len − h)(len − h + 1))/2
 7     else
 8          avCor ← avCor + len
 9
10  n ← n + 1
11
12  while lastPtr < lenCORR
13       do
14          ...
15          len ← lastPtr − firstPtr
16          h ← firstZero − firstPtr
17          if h > 0
18             then
19                  avCor ← avCor + len(len + 1)/2 − ((len − h)(len − h + 1))/2
20             else
21                  avCor ← avCor + len
22
23          n ← n + 1
```

6.2 AvCor versione subquadratica

In questa sezione viene presentata la versione subquadratica per il calcolo di *avCor*. Il procedimente per l'estrazione della misura *avCor* di due sequenze date in ingresso, fissato il numero di mismatch k si compone di due passi principali, il preprocessing e il processing. Di seguito una descrizione dettagliata di entrambi.

6.2.1 Preprocessing

Come descritto in 5.2.1 il vettore CORR viene diviso in n blocchi di taglia $\log n$ e si procede con il recupero delle informazioni necessarie nella fase di processing. Per ogni blocco quindi si tiene traccia delle posizioni dei primi e degli ultimi k mismatch presenti in ogni blocco. Se il blocco preso in considerazione è denso si procede con il calcolo della sequenza più lunga contenuta

nello stesso, ma prestando attenzione anche nel salvare la somma di tutte le lunghezze dei segmenti contenuti nel blocco denso che si sta prendendo in considerazione. Questa fase di preprocessing, similmente a come già detto richiede tempo $O(\log n)$. Anche in questo caso per come è stata implementata la parte di preprocessing, essa esegue in tempo quadratico rispetto alla taglia delle sequenze in input.

6.2.2 Processing

Anche in questo caso la procedura è simile a quanto descritto in 5.2.2. La differenza principale sta nel tener traccia delle lunghezze di tutte le sequenze con k mismatch che vengono trovati durante l'esecuzione. Consideriamo un segmento con k mismatch che comincia in una certa posizione f in cui parte un run di uni lungo h. Sia l la lunghezza del segmento che stiamo prendendo in considerazione. Dato che il run di uni del segmento che stiamo prendendo in considerazione è seguito da uno zero, si può dedurre che i più lunghi segmenti con precisamente k zeri che partono rispettivamente dalle posizioni $f+1, f+2, \dots, f+h-1$ avranno lunghezze $l, l-1, \dots, l-h$. Dalla conoscenza dovuta alla fase di preprocessing delle posizioni degli zeri, e quindi delle lunghezze dei run di uni, possiamo calcolare le somme dette in precedenza con la seguente differenza:

$$l(l+1)/2 - (l-h)(l-h+1)/2$$

Da notare che nel calcolo di *avCor* non si possono saltare calcolo di sequenze per velocizzare i tempi di esecuzione in quanto per il calcolo di questa misura servono tutte le lunghezze di tutte le sequenze con k mismatch.

Capitolo 7

Misura di similarità basata sulla media della massima correlazione

Kmacs è una implementazione di un approccio di tipo alignment-free per l'analisi delle sequenze proposto in [3]. Questo approccio è basato sul metodo ACS proposto in [4]. Il metodo *ACS* ovvero *average common substring* calcola per ogni posizione i di una sequenza, la lunghezza della più lunga sottostringa che comincia alla posizione i e che ha un match perfetto nella seconda sequenza. *Kmacs* generalizza l'approccio proposto da *ACS* considerando, per ogni posizione i di una sequenza il più lungo prefisso che comincia alla posizione i e che ha un matching con k mismatch nella seconda sequenza.

7.1 Approccio ACS e sottostringhe con k mismatch

Date due sequenze x e y l'approccio *ACS* determina per ogni posizione i in x, la lunghezza del più lungo matching perfetto tra la sequenza x partendo dalla posizione i e la sequenza y. Le lunghezze di questi matching perfetti sono mediate e normalizzate per definire la misura di similarità

$$L(x,y) = \frac{1}{|x|} * \sum_{i=1}^{|x|} maxCor(i) \quad \text{con } k = 0$$

che viene poi portata in una misura (non simmetrica) della distanza definendo

$$d(x,y) = \frac{\log |y|}{L(x,y)} - \frac{\log |x|}{L(x,x)}$$

per ottenere una distanza simmetrica, la distanza da x e y è definita come

$$d_{ACS}(x, y) = \frac{d(x, y) + d(y, x)}{2}$$

In [3] si generalizza questa misura permettendo sottostringhe con k mismatch, ovvero permettendo a k di essere diverso da zero in $maxCor(i)$.

7.2 Approssimazione sottostringhe con k mismatch

Come descritto in [3], per una coppia di sequenze, il valore esatto di *maxCor*(i) può essere calcolato in $O(k{*}n^2)$ usando strutture dati come *suffix tree*. Per arrivare a questa approssimazione, come primo passo viene calcolata per ogni posizione i in x la lunghezza l della più lunga sottostringa in comune che comincia alla posizione i e che ha un match esatto in y. Sia j la posizione in cui comincia il match in y, è da notare che il carattere $x[i + l]$ deve essere differente dal carattere $y[j + l]$. A questo punto viene esteso il match in x dalla posizione $i + l + 1$ e in y dalla posizione $j + l + 1$ finchè il prossimo mismatch non viene incontrato. Questo procedimento viene ripetuto finchè non viene incontrato il $k + 1$-mismatch o non viene raggiunta la fine di una delle due sequenze.

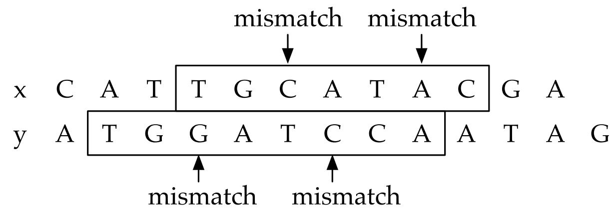


Figura 7.1: Esempio di calcolo di x(4) con permessi 2 mismatch

In Figura 7.1 si vede come in posizione $i = 4$ in x e con $k = 2$ la sottostringa ritornata è segnata con un riquadro in particolare in y comincia in $j = 2$. Per ottenere tale sottostringa, in primo luogo viene determinata la più lunga sottostringa in comune che comincia dalla posizione $i = 4$ in x che ha un match esatto in y. Questa sottostringa viene quindi trovata nella posizione $j = 2$ in y con lunghezza 2. A questo punto questo match viene esteso fino a che non viene raggiunto il successivo mismatch. La lunghezza della sottostringa con due mismatch è quindi pari a sette.

In Figura 7.2 invece si vuol far vedere come per una posizione i in x, la corrispondente posizione j in y del più lungo match esatto potrebbe non

x	C	A	T	T	G	C	A	T	A	C	G	A	
y	A	T	G	G	A	T	C	C	A	A	T	A	G

Figura 7.2: Esempio di presenza di match esatti multipli

essere unico. Infatti si vede che per $i = 2$ abbiamo ben tre match esatti in y alle posizione 1, 5 e 10. In casi come questo verranno calcolati tutti i possibili match con il numero fissato di mismatch e verrà mantenuto solo quello di lunghezza massima.

Capitolo 8

Test e discussione dei risultati

8.1 Esperimenti sull'efficacia in un dataset biologico

In questa sezione verrà descritto il dataset utilizzato, e come sono stati effettuati i test per analizzare la qualità degli alberi filogenetici ottenuti mediante l'utilizzo delle tre misure *maxCor*, *avCor* e *kACS*.

8.1.1 Descrizione del dataset usato

Il dataset usato per valutare la qualità degli alberi filogenetici ottenuti comprende trentaquattro sequenze di DNA di tipo mitocondriale di diversi mammiferi (Figura 8.1). Per questo dataset si ha anche a disposizione il *consensus tree* usato in [4][17] che poi è stato usato per valutare la qualità degli alberi filogenetici ottenuti con le misure testate in questa tesi. Il *consensus tree* usato per effettuare una analisi qualitativa degli alberi ottenuti è stato ottenuto mediante l'uso di ML applicato a 13 proteine di tipo mitocondriale provenienti dal dataset NCBI.

Classe	Super-ordine	Nome	Taglia
EUTHERIA	Euarchontoglires	GuineaPig	16801
		Rat	16313
		Housemouse	16299
		Rabbit	17245
		Squirrel	16507
		Dormouse	16602
		Baboon	16521
		Gibbon	16472
		Orangutan	16389
		Gorilla	16364
		Human	16569
		PygmyChimpanzee	16563
		Chimpanzee	16554
	Laurasiatheria	Dog	16727
		Cat	17009
		GreySeal	16797
		HarborSeal	16826
		WhiteRhino	16832
		GreatRhino	16829
		Donkey	16670
		Horse	16660
		FruitBat	16651
		BlueWhale	16402
		FinbackWhale	16398
		Hippo	16407
		Pig	16613
		Sheep	16616
		Cow	16341
	Afrotheria	Elephant	16866
		Aardvark	16816
	Xenarthra	Armadillo	17056
METATHERIA	Ameridelphia	Wallaroo	16896
		Opossum	17084
AUSTRLOSPHENIDA	Monotremata	Platypus	17019

Figura 8.1: Specie contenute nel dataset

8.1.2 Test

I test sono stati effettuati mediante il seguente procedimento (la stessa procedura vale anche per la misura *avCor*):

- calcolo della matrice dei valori ottenuti per *maxCor* tra ogni coppia di sequenze presenti nel dataset.
- calcolo della matrice delle distanze dalla matrice dei valori di *maxCor* ottenuta al punto precedente. Come in [2], sia q_{AB} il valore di *MaxCor* per gli organismi A e B, per ottenere la distanza si procede calcolando la media della distanza euclidea tra i valori q_{AB} e q_{AA} e tra q_{BA} e q_{BB}.
- mediante il tool *neighbor* costruzione dell'albero filogenetico a partire dalla matrice delle distanze ottenuta al punto precedente.
- calcolo della distanza dal *consensus tree* mediante l'uso del tool *treedist*

Tutti i passi appena elencati vengono ripetuti per vari valori di k. I risultati sono riportati in Figura 8.2.

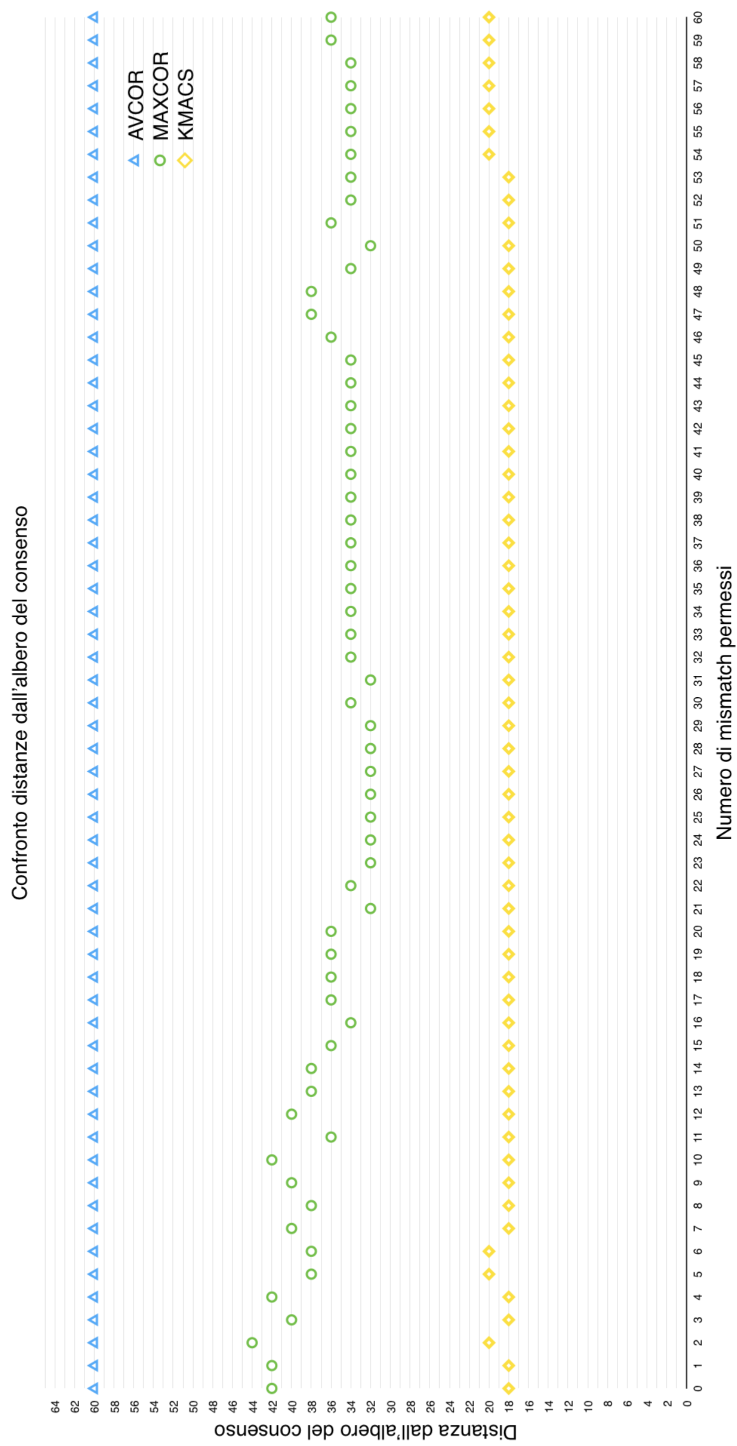


Figura 8.2: Distanza dal consensus tree

In particolare da questo grafico si vede come *kmacs* porti in generale ad avere degli alberi con una distanza minore dal *consensus tree*. Per quanto riguarda *maxCor* invece si nota che questa misura fornisce risultati migliori permettendo un numero di mismatch compreso tra 20 e 30. La misura *avCor*, come si vede dal grafico, non fornisce dei risultati promettenti. In fase di debug, su alcune coppie di DNA mitocondriale, si è notato che nel caso il vettore contenente i match e mismatch dell'allineameto sia molto denso di mismatch, il valore di $AvCor(i)$ sia in molti casi molto vicino al numero di mismatch permessi k, facendolo pesare poi nel calcolo del loro valore medio. Per questo motivo si ipotizza questo risultato.

8.1.3 Discussione dei risultati

Dal grafico presente in Figura 8.2 si vede che per k pari a 31 usando la misura *maxCor* si ottiene un albero filogenetico di distanza minore dal *consensus tree*. In Figura 8.4 e 8.5 si vedono gli alberi filogenetici ottenuti usando la misura *maxCor* permettendo un numero di mismatch pari a 2 e 31, mentre in Figura 8.11 e 8.12 una heat map ottenuta ancora con *maxCor* con permessi sempre lo stesso numero di mismatch. In Figura 8.13 la heat map ottenuta usando *kmacs* e permettendo 7 mismatch. É possibile valutare la qualità degli alberi filogenetici ottenuti osservando come sono stati clusterizzati gli animali appartenenti allo stesso superordine, come visibile in Figura 8.1.

Si può osservare che l'albero ottenuto con 2 mismatch sia piuttosto approsimativo. Come si vede in Figura 8.4 il gruppo dei *primates* sia stato clusterizzato piuttosto bene, eccetto che per *baboon* e *orangutan*. Gli altri superordini invece non sono stati clusterizzati bene, infatti gli animali appartenenti al superordine degli *afroteria* e *xenarthra* sono piuttosto lontani, lo stesso per *marsupials* e *rodents*.

L'albero in Figura 8.5 è invece molto più accurato. Si vede infatti che il superordine dei *primates* sono tutti clusterizzati molto vicini, in accordo con il *consensus tree*, e anche gli animali appartenenti al superordine dei *rodents*, eccetto che per *sqirrel* che è stato posto piuttosto lontano. Lo stesso vale per il superordine *marsupials*, infatti *opossum* e *wallaroo* sono stati clusterizzati bene. Lo stesso non si può dire della superordine degli *afrotheria*, ovvero di *elephant* e del *aardvark* che sono stati posti molto lontani. Per quanto riguarda il superordine *laurasiatheria*, possiamo identificare dei sotto-ordini:

- *carnivora* (*cat, dog, grey seal, harbor seal*)
- *perissodactyla* (*white rinho, great rhino, donkey, horse*)
- *chitophera* (*fruit bat*)

- *cetacea* (*blue whale, finback whale*)

- *artiodactyla* (*hippo, pig, sheep, cow*)

tutti questi sotto-gruppi sono stati clusterizzati piuttosto bene, eccetto che per il sotto-ordine dei *artiodactyla*, infatti questi ultimi sono stati divisi in due gruppi, e posti piuttosto lontani.

In Figura 8.6 l'albero filogenetico usando *kACS*. Dalla figura si vede come i vari gruppi di animali appartenenti ai diversi superordini e sotto-ordini definiti sopra siano clusterizzati in modo corretto, sebbene con qualche differenza.

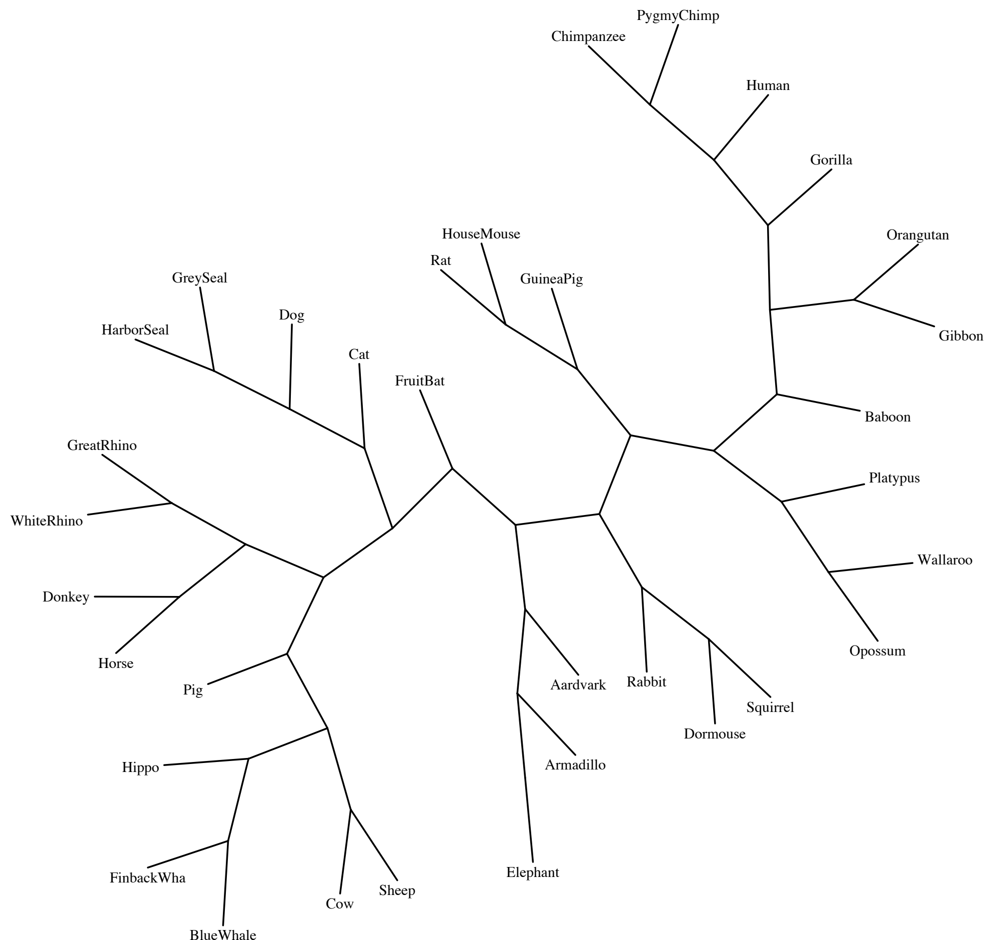


Figura 8.3: Consensus tree per un dataset di 34 genomi mitocondriali

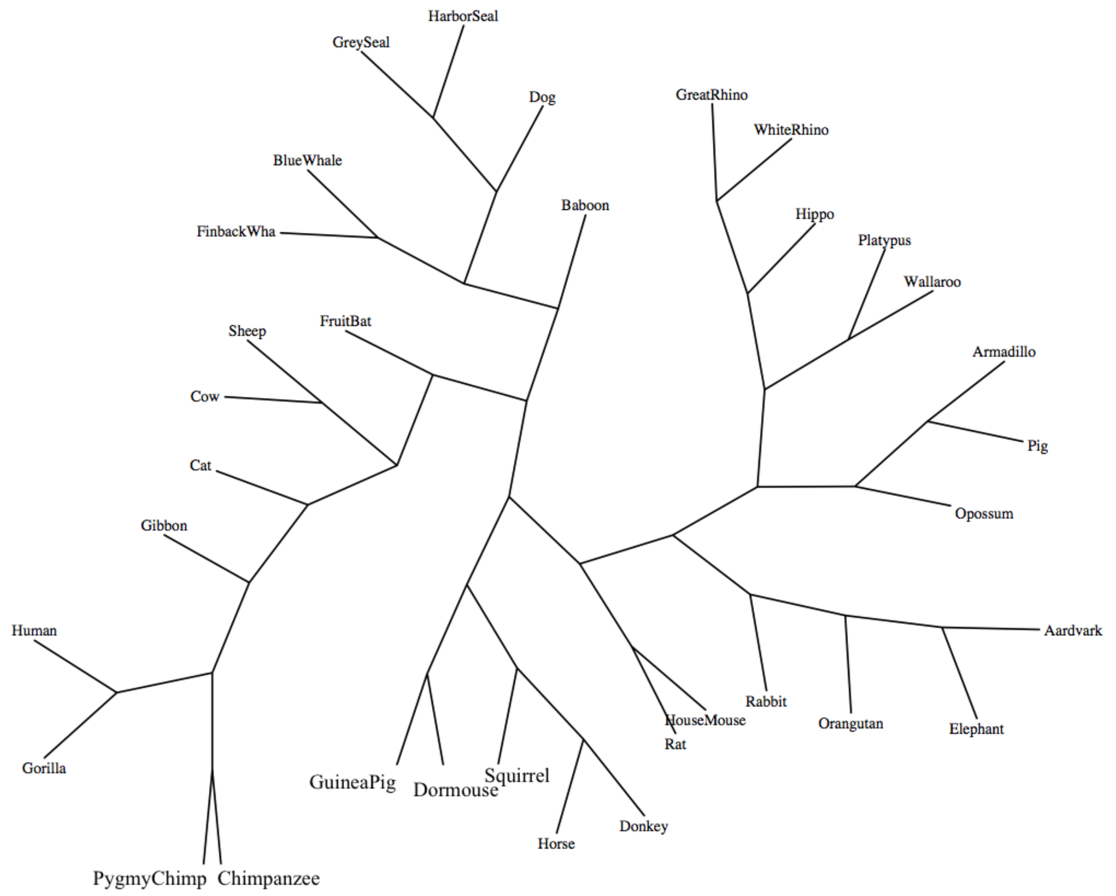


Figura 8.4: Albero ottenuto con k = 2 (*maxCor*)

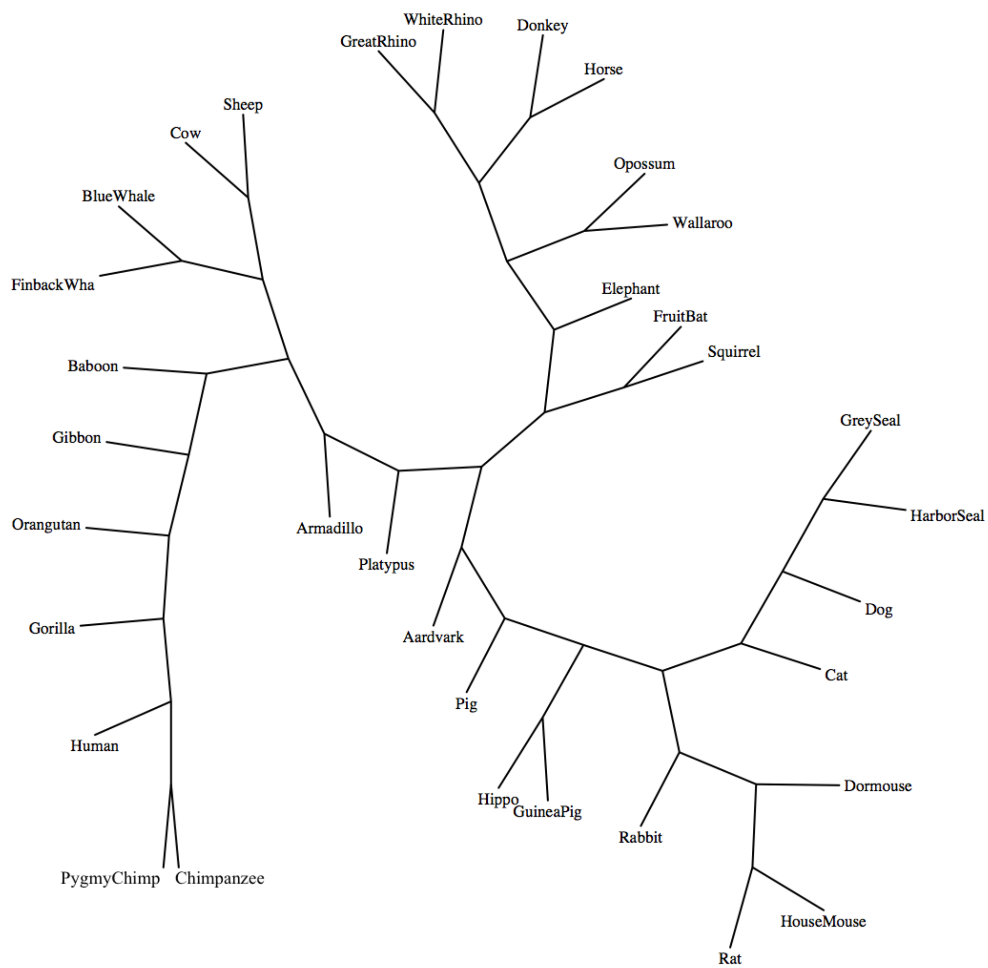


Figura 8.5: Albero ottenuto con k = 31 (*maxCor*)

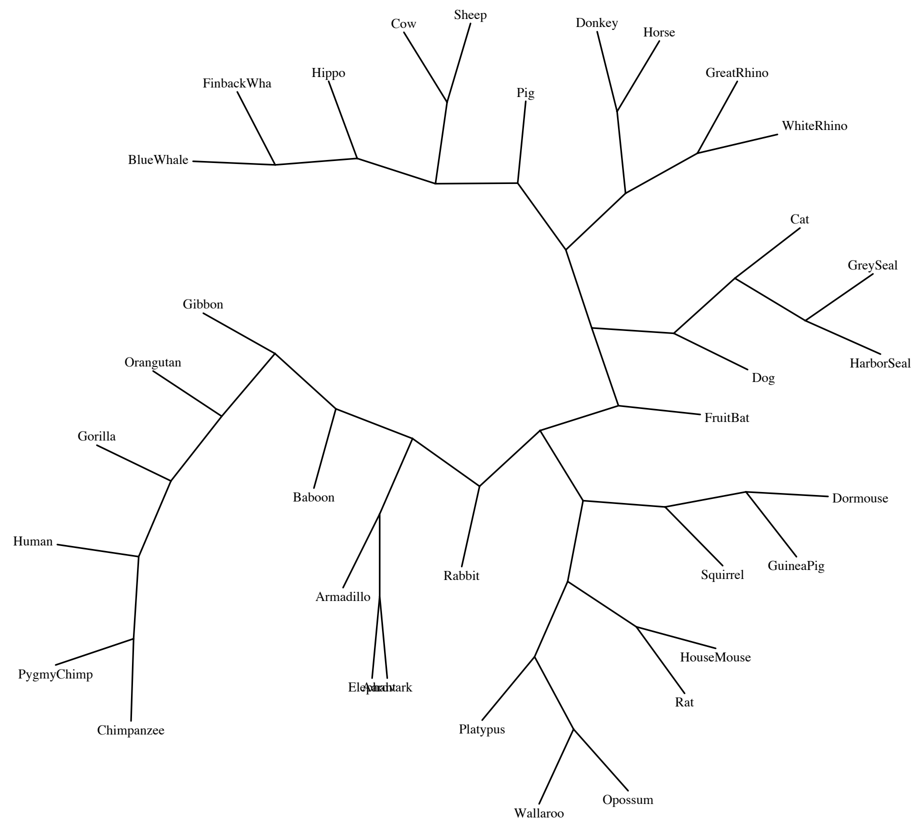


Figura 8.6: Albero ottenuto con k = 7 ($kACS$)

8.2 Esperimenti sull'efficienza su sequenze random

Oltre ai test per la valutazione delle prestazioni in termini di qualità della misura, sono stati effettuati dei test per valutare e confrontare la velocità di esecuzione della versione quadratica e subquadratica di *maxCor*. I test sono stati effettuati per $k = 2, 4, 8, 16$ e per lunghezze delle sequenze, da 10000 a 60000 caratteri con passo 10000. Le sequenze sono state generate in modo random, con simboli presi da un alfabeto di quattro caratteri. Per entrambe le versioni sono stati annotati solo i tempi di esecuzione della fase di processing.

8.2.1 Discussione

Nelle Figure 8.7, 8.8, 8.9 e 8.10 si vedono i grafici ottenuti. Dalle figure di nota che nella versione quadratica il tempo di esecuzione non dipende dal numero di mismatch permessi. Per quanto riguarda la versione subquadratica si nota come il tempo di processing aumenti all'aumentare del numero di mismatch permessi, questo è dovuto al fatto che il numero di accessi ($n/\log n$ per ciascun COR) costa $O(k)$. Dal profile sull'implementazione della versione subquadratica, si è riscontrato anche che un rallentamento è dovuto al fatto che si facciano molti accessi alle strutture dati, manipolando puntatori nei due vettori contenenti le posizioni dei primi e ultimi k mismatch dei vari blocchi, a differenza della versione quadratica in cui si scorrono due puntatori sullo stesso vettore. Nell'implementazione sono stati usati per la codifica dei blocchi dei vettori di caratteri, è possibile che cambiando la loro rappresentazione usando invece vettori di bit si ottengano prestazioni migliori [26].

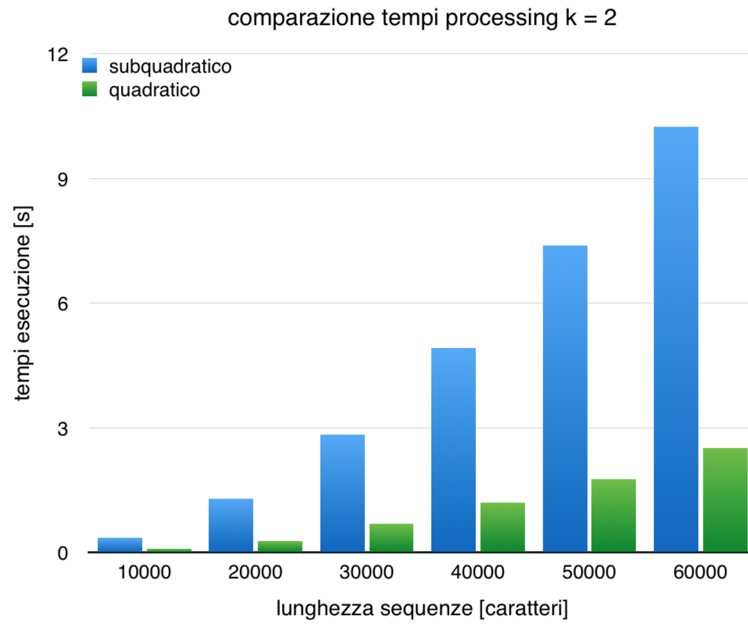


Figura 8.7: Confronto tempi processing per k = 2

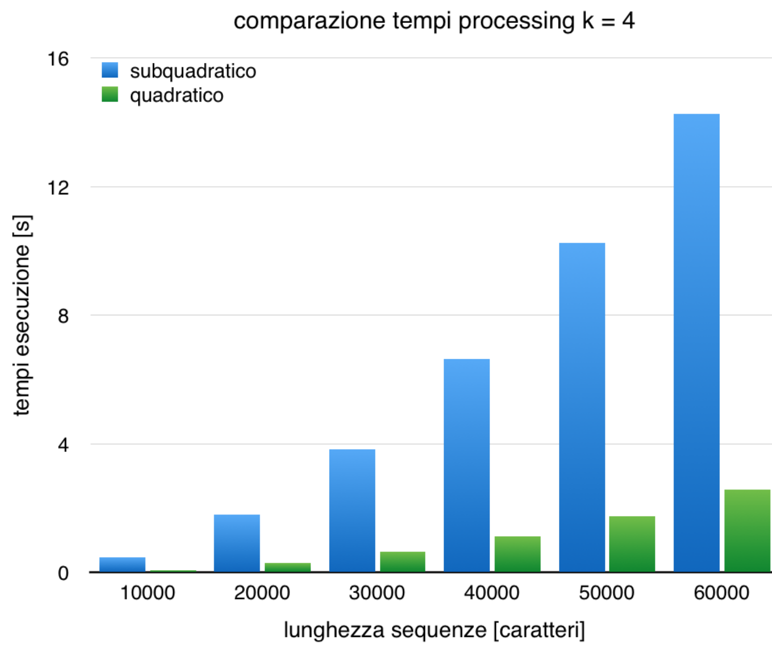


Figura 8.8: Confronto tempi processing per k = 4

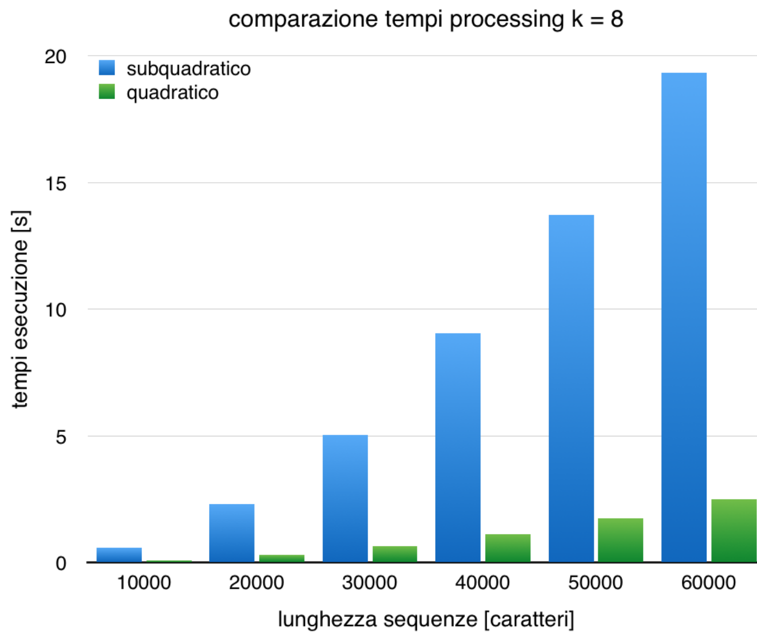


Figura 8.9: Confronto tempi processing per k = 8

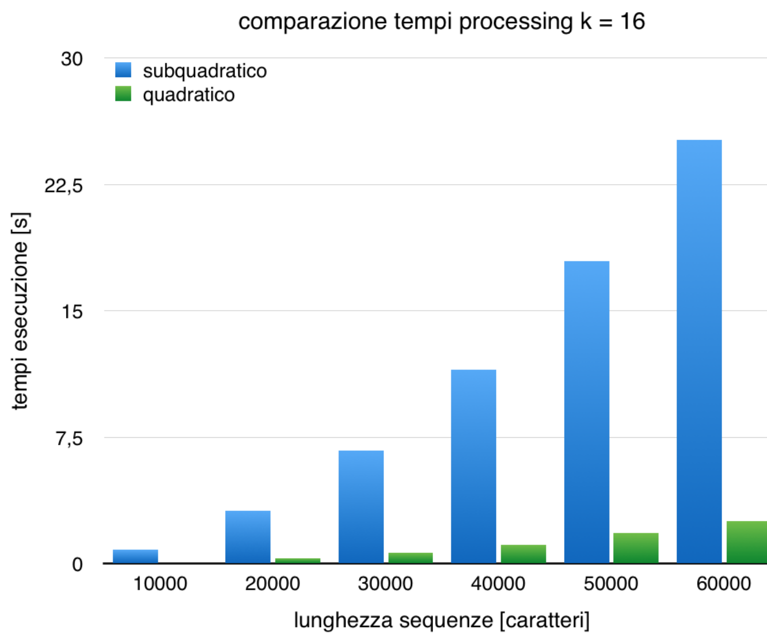


Figura 8.10: Confronto tempi processing per k = 16

Figura 8.11: Heap map ottenuta con $k = 2$ ($maxCor$)

	Chimpan	PygmyCl	Human	Gorilla	Orangut	Gibbon	Baboon	Horse	Donkey	WhiteRh	GreatRhi	Aardvark	Armadill	Elephant	Squirrel	Dormous	Rabbit
Chimpan		1243	786	793	505	412	327	218	218	288	284	274	276	264	233	272	276
PygmyCl	1243		952	857	510	397	326	214	212	283	283	276	276	263	232	270	275
Human	786	952		856	551	397	341	227	219	244	250	262	239	240	234	261	266
Gorilla	793	857	856		551	416	334	206	208	282	281	268	269	263	224	263	274
Orangut	505	510	551	551		366	329	208	208	280	282	279	274	263	227	264	277
Gibbon	412	397	397	416	366		323	230	209	290	288	275	269	264	229	256	280
Baboon	327	326	341	334	329	323		213	213	282	281	274	271	262	227	262	274
Horse	218	214	227	206	208	230	213		666	379	402	215	224	306	209	209	227
Donkey	218	212	219	208	208	209	213	666		370	424	242	245	308	215	209	212
WhiteRh	288	283	244	282	280	290	282	379	370		574	288	278	267	248	237	300
GreatRhi	284	283	250	281	282	288	281	402	424	574		294	276	272	228	206	301
Aardvark	274	276	262	268	279	275	274	215	242	288	294		269	265	249	251	275
Armadill	276	276	239	269	274	269	271	224	245	278	276	269		268	263	248	271
Elephant	264	263	240	263	263	264	262	306	308	267	272	265	268		236	251	261
Squirrel	233	232	234	224	227	229	227	209	215	248	228	249	263	236		256	256
Dormous	272	270	261	263	264	256	262	209	209	237	206	251	248	251	256		328
Rabbit	276	275	266	274	277	280	274	227	212	300	301	275	271	261	256	328	
Rat	261	258	263	256	255	258	268	263	260	268	222	262	256	230	273	332	301
HouseM	257	252	252	258	252	256	270	272	273	261	257	242	240	233	263	315	291
GuineaPi	262	260	224	263	257	221	261	201	227	248	258	278	243	206	239	288	298
Sheep	287	284	250	285	276	293	276	281	284	304	305	274	278	272	242	221	281
Cow	284	283	250	286	277	290	281	253	247	300	297	294	288	265	259	313	277
Hippo	283	281	269	277	283	278	277	273	275	312	310	328	276	266	268	329	323
Pig	280	281	262	272	282	279	278	242	227	300	307	303	273	270	275	312	311
FruitBat	220	220	215	214	215	220	212	219	225	247	245	223	200	218	233	230	228
BlueWha	285	283	248	283	283	283	287	248	241	314	311	279	277	272	276	208	277
FinbackV	284	282	247	282	283	283	291	242	238	312	311	282	277	271	243	241	281
Platypus	250	250	251	245	243	248	243	194	188	258	265	251	244	245	212	221	243
Wallaroo	213	197	208	188	186	185	181	262	262	264	240	200	199	229	221	203	205
Opossum	192	191	207	200	209	218	192	247	254	247	247	202	209	243	212	218	193
HarborS	272	271	267	263	265	257	266	244	236	273	258	304	232	236	261	339	338
GreySeal	269	268	273	257	262	254	264	245	266	271	260	300	253	238	272	335	326
Dog	277	274	270	257	265	257	266	223	253	264	261	293	260	237	275	339	309
Cat	264	262	283	246	256	253	243	289	283	271	272	258	228	231	260	309	303

	Rat	HouseM	GuineaPi	Sheep	Cow	Hippo	Pig	FruitBat	BlueWha	FinbackV	Platypus	Wallaroo	Opossum	HarborS	GreySeal	Dog	Cat
Chimpan	261	257	262	287	284	283	280	220	285	284	250	213	192	272	269	277	264
PygmyCl	258	252	260	284	283	281	281	220	283	282	250	197	191	271	268	274	262
Human	263	252	224	250	250	269	262	215	248	247	251	208	207	267	273	270	283
Gorilla	256	258	263	285	286	277	272	214	283	282	245	188	200	263	257	257	246
Orangut	255	252	257	276	277	283	282	215	283	283	243	186	209	265	262	265	256
Gibbon	258	256	221	293	290	278	279	220	283	283	248	185	218	257	254	257	253
Baboon	268	270	261	276	281	277	278	212	287	291	243	181	192	266	264	266	243
Horse	263	272	201	281	253	273	242	219	248	242	194	262	247	244	245	223	289
Donkey	260	273	227	284	247	275	227	225	241	238	188	262	254	236	266	253	283
WhiteRh	268	261	248	304	300	312	300	247	314	312	258	264	247	273	271	264	271
GreatRhi	222	257	258	305	297	310	307	245	311	311	265	240	247	258	260	261	272
Aardvark	262	242	278	274	294	328	303	223	279	282	251	200	202	304	300	293	258
Armadill	256	240	243	278	288	276	273	200	277	277	244	199	209	232	253	260	228
Elephant	230	233	206	272	265	266	270	218	272	271	245	229	243	236	238	237	231
Squirrel	273	263	239	242	259	268	275	233	276	243	212	221	212	261	272	275	260
Dormous	332	315	288	221	313	329	312	230	208	241	221	203	218	339	335	339	309
Rabbit	301	291	298	281	277	323	311	228	277	281	243	205	193	338	326	309	303
Rat		328	257	238	295	311	294	220	229	260	202	243	221	304	294	282	297
HouseM	328		250	233	266	280	259	233	257	209	224	238	225	288	281	275	286
GuineaPi	257	250		201	277	311	290	226	247	222	203	210	215	300	291	287	255
Sheep	238	233	201		442	289	300	239	317	321	253	223	226	256	270	240	267
Cow	295	266	277	442		345	330	236	310	314	253	254	237	349	348	356	327
Hippo	311	280	311	289	345		323	242	290	296	253	212	224	345	343	341	318
Pig	294	259	290	300	330	323		258	294	295	256	209	230	332	330	328	319
FruitBat	220	233	226	239	236	242	258		244	210	209	211	218	270	243	232	237
BlueWha	229	257	247	317	310	290	294	244		960	256	203	210	252	259	261	245
FinbackV	260	209	222	321	314	296	295	210	960		254	203	210	251	254	247	245
Platypus	202	224	203	253	253	253	256	209	256	254		200	198	254	256	250	243
Wallaroo	243	238	210	223	254	212	209	211	203	203	200		306	216	216	204	204
Opossum	221	225	215	226	237	224	230	218	210	210	198	306		237	233	213	213
HarborS	304	288	300	256	349	345	332	270	252	251	254	216	237		1227	362	328
GreySeal	294	281	291	270	348	343	330	243	259	254	256	216	233	1227		359	323
Dog	282	275	287	240	356	341	328	232	261	247	250	204	213	362	359		324
Cat	297	286	255	267	327	318	319	237	245	245	243	204	213	328	323	324	

Primates — (Perissodactyla) Laurasiatheria — Afrotheria and Xenarthra — (Rodents) Laurasiatheria — (Artiodactyla) Laurasiatheria — (Chitophera) Laurasiatheria — (Cetacea) — Marsupials and Monotremes — (Carnivora) Laurasiatheria

Figura 8.12: Heap map ottenuta con k = 31 ($maxCor$)

Primates

(Perissodactyla) Laurasiatheria

Afrotheria and Xenarthra

(Rodents) Laurasiatheria

(Artiodactyla) Laurasiatheria

(Chitophera) Laurasiatheria

(Cetacea) Laurasiatheria

Marsupials and Monotremes

(Carnivora) Laurasiatheria

Figura 8.13: Heap map ottenuta con k = 7 (*kACS*)

Capitolo 9

Conclusioni

In questa tesi sono state implementate le misure *maxCor* e *avCor* come presentate in [2]. Entrambe le misure sono state implementate sia nella versione quadratica che subquadratica, valutandone le differenze in termini di tempi di esecuzione. Le performance di queste due misure sono state testate calcolando le matrici delle distanze permettendo diversi valori per i mismatch. Dalle matrici sono quindi stati costruiti gli alberi filogenetici e quindi valutata la loro bontà confrontandoli con l'albero del consenso del dataset preso in considerazione usato anche in [4]. Inoltre è stata valutata anche la bontà degli alberi filogenetici ottenuti usando come misura *kACS* come proposto in [3]. Dai test ottenuti usando come dataset sequenze di DNA mitocondriale di trentaquattro mammiferi è emerso che *kACS* fornisca dei risultati più accurati in quanto le distanze degli alberi costruiti siano molto minori dall'albero del consenso. Inoltre si è notato come tra *maxCor* e *avCor*, i risultati migliori si siano ottenuti con la prima misura, infatti gli alberi ottenuti usando la misura *maxCor* hanno distanza molto più piccola di quelli ottenuti con *avCor* rispetto al *consensus tree*. É inoltre da osservare che gli esperimenti sono stati fatti usando una distanza pittosto grezza, ed è possibile che si riescano ad ottenere degli alberi migliori ad esempio considerando l'aspettazione dei valori di *maxCor*.

Capitolo 10

Ringraziamenti

I miei più sentiti ringraziamenti alla Prof.ssa Cinzia Pizzi per l'enorme pazienza e disponibilità dimostratami nello seguirmi durante lo sviluppo di questa tesi.

Un grazie ai miei genitori Armando e Edi Anna e a mio fratello Davide per avermi supportato e sopportato durante questi lunghi anni universitari.

Ringrazio gli amici che ho conosciuto all'università: Matteo L., Matteo B., Marco C., Marco B., Davide e Giacomo per avermi aiutato durante questi anni e per aver reso più leggero questo percorso di studi.

Infine un ringraziamento agli amici di Piazzola e dintorni: Mosè, Alberto, Emmanuele, Giacomo e Adele, Mattia, Sergio e Caterina, Piero, Silvio e Christoper per aver reso il più pesante, difficoltoso e scomodo possibile arrivare alla fine.

Bibliografia

[1] T. Fluori, E. Giaquinta, K. Kobert, and E. Ukkonen, "Longest common substring with k mismatches," *Information Processing Letters*, Volume 115, Issues 6–8, Pages 643–647 (2015).

[2] A. Apostolico, C. Guerra, and C. Pizzi, "Alignment free sequence similarity with bounded hamming distance," *Data Compression Conference (DCC)*, (2014).

[3] C. Leimeister and B. Morgenstern, "kmacs: the k-mismatch average common substring approach to alignment-free sequence comparison," *Bioinformatics*, (2014).

[4] I. Ulitsky, D. Burnstein, T. Tuller, and B. Chor, "The average common substring approach to phylogenetic reconstruction," *J. Comput. Biol.*, (2003).

[5] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, 147 (1981) 195–197.

[6] S. Needleman and C. J. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, 48 (1970) 443–453.

[7] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, (1990).

[8] S. Vinga and J. Almeida, "Alignment-free sequence comparison - a review," *Bioinformatics*, (2002).

[9] `http://evolution.genetics.washington.edu/phylip.html`.

[10] `http://www.clustal.org`.

[11] N. Saitou and M. Nei, "The neighbor-joining method: a new method for reconstructing phylogenetic trees.," *Molecular Biology and Evolution*, volume 4, issue 4, pp. 406-425 (1987).

[12] N. C. Jones and P. A. Pevzner, *An introducion to bioinformatics algorithms*. MIT Press, 2004.

[13] `http://valgrind.org`.

[14] S. Grabowsky, "A note on the longest common substring with k mismatches," *Information Processing Letters*, Volume 115, Issues 6–8, Pages 640–642 (2015).

[15] `http://openmp.org/wp/`.

[16] D. Graur, M. Gouy, and L. Duret, "Evolutionary affinities of the order perissodactyla and the phylogenetic status of the superordinal taxa ungulata and altungulata," *Mol. Phylogenet. Evol.*, 7, (1997) 195–200.

[17] T. Attwood, "Genomics: the babel of bioinformatics," *Science*, 290 (2000) 471–473.

[18] Y. Cao, A. Janke, P. J. Waddell, M. Westerman, O. Takenaka, S. Murata, N. Okada, S. Pabo, and M. Hasegawa, "Conflict among individual mitochondrial proteins in resolving the phylogeny of eutherian orders," *J. Mol. Evol.*, 47 (1998) 307-322.

[19] M. Li, J. Badger, X. Chen, S. Kwong, P. Kearney, and H. Zhang, "An information- based sequence distance and its application to whole mitochondrial genome phylogeny," *Bioinformatics*, 17 (2), (2001) 149-154.

[20] F. Murtagh, "Complexities of hierarchic clustering algorithms: the state of the art," *Computational Statistics Quarterly*, 1: 101–113 (1984).

[21] C. Stewart, "The powers and pitfalls of parsimony," *Nature*, 361 (6413), (1993), 603–607.

[22] D. Robinson and L. Foulds, "Comparison of phylogenetic trees," *Mathematical Biosciences*, vol. 53, no. 1–2, pp. 131 – 147, 1981.

[23] F. Sievers, A. Wilm, D. Dineen, T. Gibson, K. Karplus, W. Li, R. Lopez, H. McWilliam, M. Remmert, J. Söding, J. Thompson, and D. Higgins, "Fast, scalable generation of high-quality protein multiple sequence alignments using clustal omega," *Molecular Systems Biology*, 7:539 (2011).

[24] D. Lipman and W. Pearson, "Rapid and sensitive protein similarity searches," *Science*, (1985).

[25] C. S. Leslie, E. Eskin, A. Cohen, J. Weston, and W. S. Noble, "Mismatch string kernels for discriminative protein classification.," *Bioinformatics*, (2004).

[26] V. Pieterse, D. G. Kourie, L. Cleophas, and B. W. Watson, "Performance of c++ bit-vector implementations," *SAICSIT 2010*, (2010).