Università degli studi di Modena e Reggio Emilia

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

All-Against-All Sequence Matching:
Implementazione mediante Suffix Array e Analisi Prestazionale Comparata

Relatore:
Prof. Paolo Tiberio

Tesi di Laurea di:
Dario Gelmini

Correlatori:
Dott. Federica Mandreoli
Ing. Riccardo Martoglia

Anno Accademico 2002-2003

Parole chiave:

Sequence Matching,
Suffix Tree,
Suffix Array,
All-Against-All,
Edit Distance.

Indice

Introduzione

In questo lavoro ci apprestiamo a trattare le tematiche relative all'*All-Against-All Sequence Matching*, intendendo con ciò l'operazione di ricerca di tutte le possibili occorrenze all'interno di una sequenza, per le quali esistono occorrenze all'interno di una seconda sequenza, la cui distanza è inferiore ad una soglia prefissata. Come vedremo tali sequenze potranno essere costituite da simboli di ogni tipo, siano essi le parole di un documento o i caratteri rappresentanti un segmento di DNA. Il problema delle occorrenze "Tutti contro Tutti" non cambia infatti la sua natura e rimane insensibile al contesto da cui derivano i dati.

Vedremo comunque che la soluzione di tale problema attraverso un approccio piuttosto che un'altro può far variare sensibilmente le prestazioni in fase di esecuzione. Nostro scopo sarà dunque quello di analizzare due algoritmi (*All-Against-All con Suffix Array*[1] e *Sub*^{2}*Matching*[2]) al fine di determinare quali siano gli ambiti applicativi ad essi più indicati. Daremo quindi un'implementazione del primo e ci appoggeremo ad un software già esistente per quanto riguarda il secondo. Quest'ultimo programma risulta essere particolarmente performante, il chè lo rende un ottimo metro di paragone per l'analisi delle prestazioni che ci prefiggiamo di eseguire.

Ciò che vogliamo infine ottenere è l'analisi completa dei campi di variabilità dei dati entro i quali risulti conveniente l'utilizzo di un determinato algoritmo.

La trattazione proseguirà dunque nel seguente modo: Nel *Capitolo 1* vengono presentate una serie di nozioni basilari che il lettore esperto può tranquillamente saltare; Nel *Capitolo 2* vengono introdotti gli algoritmi proposti per l'individuazione delle occorrenze descritte; Nel *Capitolo 3* si riportano le

modifiche implementative e le soluzioni adottate mirate all'ottimizzazione del codice; Nel *Capitolo 4* viene esposto lo studio dei test necessari e i loro risultati; Nel *Capitolo 5* infine vengono tratte le conclusioni sul lavoro svolto.

Capitolo 1

Prerequisiti

Presenteremo in questo capitolo i concetti fondamentali per una corretta comprensione degli argomenti trattati. Assumeremo che alcuni concetti basilari quali l'analisi e la progettazione di algoritmi e la conoscenza delle strutture dati fondamentali siano noti. Iniziamo dunque fornendo alcune definizioni formali inerenti al problema e quindi illustreremo alcune strutture dati di particolare interesse.

1.1 Match Approssimato fra Stringhe

Nella trattazione seguente indicheremo con *s,x,y,z,v,w* le generiche stringhe e con *a,b,c,...* le lettere di cui sono composte. Scrivere una sequenza di lettere e/o stringhe equivarrà ad effettuarne un concatenamento. Considereremo inoltre come noti i concetti di *prefisso*, *suffisso* e *sottostringa*. Per ogni stringa $s \in \sum^*$ indicheremo con $|s|$ la sua lunghezza e con s_i il suo i_esimo carattere, ove $i \in \{1, ..., |s|\}$. Indichiamo inoltre una porzione di sequenza come $s_{i...j} = s_i s_{i+1} ... s_j$, la quale, nel caso sia $i > j$, coinciderà con la stringa nulla la quale verrà indicata con ε.

Nell'introduzione abbiamo accennato al problema del match approssimato fra stringhe, indicando come esso evidenzi la necessità di riconoscere delle occorrenze, fra il testo dato ed il pattern di ricerca, a meno di un numero k di errori ammissibili. Vediamo ora di darne una definizione più rigorosa [3].

Definizione: Match Approssimato fra Stringhe

Sia $\sum$ un alfabeto finito di dimensione $|\sum| = \sigma$

Sia $T \in \sum^*$ un *testo* di lunghezza $n = |T|$

Sia $P \in \sum^*$ un *pattern* di lunghezza $m = |P|$

Sia $k \in \mathbb{R}$ il massimo errore ammissibile

Sia $d : \sum^* \times \sum^* \to \mathbb{R}$ una *distanza*

Il problema può allora essere riformulato come segue:

Dati T, P, k ed una funzione di distanza $d()$, si ricerchino tutte le posizioni del testo j per le quali esista un i tale che $d(P, T_{i...j}) \leq k$.

Detto ciò si noti come la funzione *distanza* non viene ulteriormente specificata. Essa può dunque essere scelta arbitrariamente senza modificare le definizioni di cui sopra.

1.2 Distanza tra Stringhe

All'interno della nostra trattazione, scegliamo di accettare le sole funzioni di distanza conformi alla seguente definizione [3]:

Definizione: Distanza tra Stringhe

> La distanza $d(x, y)$ fra due stringhe x ed y è data dal costo minimo di una sequenza di operazioni atte a trasformare la stringa x nella stringa y. Il costo di una sequenza di operazioni è dato dalla somma dei costi delle singole operazioni. Le operazioni sono un insieme finito di regole esprimibili nella forma $\delta(z, w) = t$, ove z e w sono due diverse stringhe e t un numero reale non negativo. Una volta che la stringa z è stata convertita nella stringa w nessun'altra operazione può essere eseguita su w.

Si noti come quest'ultima condizione impedisca di agire più e più volte sulla stessa stringa. Eliminare tale restrizione dalla definizione di distanza significherebbe permettere un livello di elaborazione della stringa tale da renderne impossibile il calcolo della distanza. Si noti inoltre come la distanza in tal modo definita definisca a sua volta una metrica, infatti, se per ogni operazione $\delta(z, w)$ esiste la rispettiva operazione $\delta(w, z)$ di ugual costo, allora la distanza è simmetrica (es: $d(x, y) = d(y, x)$), inoltre date le strighe x, y, z $d(x, y) \geq 0$, $d(x, x) = 0$ e $d(x, z) \leq d(x, y) + d(y, z)$.

Generalmente, nel maggior numero delle applicazioni, le possibili operazioni utilizzate al fine di valutare la distanza fra stringhe risultano essere:

Inserimento:	$\delta(\varepsilon, a)$, es: inserire la lettera a.
Cancellazione:	$\delta(a, \varepsilon)$, es: cancellare la lettera a.
Sostituzione:	$\delta(a, b)$ con $a \neq b$, es: sostituzione di a con b.
Trasposizione:	$\delta(ab, ba)$ con $a \neq b$, es: scambiare le lettere a e b adiacenti.

1.2.1 Funzioni di Distanza

A fronte delle definizioni precedenti possiamo dunque definizione delle funzioni di distanza maggiormente utilizzate:

Edit Distance [4]**:** consente inserimenti, sostituzioni e cancellazioni. Nella sua definizione semplificata essa prevede un costo unitario per tutte le operazioni, il che equivale a dire che la distanza così calcolata è di fatto il numero minimo di operazioni necessarie a rendere uguali le due stringhe in esame. Tale distanza è simmetrica e finita in quanto $0 \leq d(x, y) \leq max(|x|, |y|)$.

Hamming Distance [5]**:** le sole operazioni consentite sono le sostituzioni, le quali, nella versione più semplice della definizione, sono valutate a costo unitario. In letteratura è nota come funzione preposta al "Matching di stringhe con k errori". Tale distanza è simmetrica e finita in quanto $|x| = |y|$ e $0 \leq d(x, y) \leq |x|$.

Episode Distance [6]**:** ammette i soli inserimenti, i quali assumono costo unitario. Questa distanza non è simmetrica e non sempre è possibile la conversione della striga x nella stringa y. In generale $d(x, y)$ vale $|y| - |x|$ oppure ∞.

Longest Common Subsequence Distance [7]**:** permette l'uso di inserimenti e cancellazioni, tutti a costo unitario. Il nome di questa funzione di distanza deriva dal fatto che essa misura la lunghezza della più lunga sequenza di caratteri accopiabili fra le due stringhe in esame. In tal modo viene anche rispettato l'ordine delle lettere. La distanza è data dal numero di caratteri non accoppiati. Tale funzione risulta essere simmetrica e ha valori in $0 \leq d(x, y) \leq |x| + |y|$.

Nel nostro caso utilizzeremo la funzione di *Edit Distance* in quanto rende possibile una precisa valutazione della distanza fra due stringhe date, qualunque sia la loro configurazione. Inoltre, con la distanza presa in considerazione (così come con quella di *Hamming*), il problema risulta ben posto se e solo se il tasso di errore k rispetta il vincolo $0 \leq k \leq m$, ove m è la lunghezza del

Pattern da individuare all'interno del testo.

1.2.2 Calcolo dell'*Edit Distance*

L'algorimo che utilizzeremo per il calcolo dell'*Edit Distance* è di tipo dinamico, esso è stato più volte usato in passato in ambiti anche molto diversi e, nonostante non sia il più efficente, rimane comunque il più flessibile e quindi il più adatto ai nostri scopi.

Supponiamo dunque di avere due stringhe x ed y e di volerne calcolare la distanza $d(x, y)$. Prendiamo allora una matrice $C_{0\ldots|x|,0\ldots|y|}$ ed andiamo a riempirla con i criteri seguenti:

$$\begin{array}{lcll} C_{i,0} & = & i & \\ C_{0,j} & = & j & \\ C_{i,j} & = & C_{i-1,j-1} & \text{se} \quad x_i = y_i \\ & & 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}) & \text{altrimenti} \end{array}$$

In tal modo l'elemento $C_{|x|,|y|}$ rappresenta il numero di operazioni necessarie a trasformare la stringa x nella stringa y e quindi la loro *Edit Distance*. Analogamente il generico elemento $C_{i,j}$ rappresenta il numero di operazioni necessarie a trasformare la sottostringa $x_{1\ldots i}$ nella sottostringa $y_{1\ldots j}$ e quindi la loro distanza.

Analizziamo dunque il caso in cui le stringhe x ed y siano date e ne si voglia calcolare la distanza. Supponiamo inoltre che la distanza fra le sottostringhe $x_{1\ldots i-1}$ e $y_{1\ldots j-1}$ sia già stata calcolata. Consideriamo quindi i caratteri x_i e y_j. Se sono uguali non influenzeranno il calcolo e dovremo quindi passare ai prossimi. In caso contrario dovremo modificare una delle due stringhe secondo una delle tre operazioni permesse. Potremo infatti *cancellare* il carattere x_i convertendo $x_{1\ldots i-1}$ in $y_{1\ldots j}$ oppure *inserire* y_j al termine di $x_{1\ldots i}$ convertendo $x_{1\ldots i}$ in $y_{1\ldots j-1}$ o ancora *sostituire* x_i con y_j convertendo così $x_{1\ldots i-1}$ in $y_{1\ldots j-1}$. In tutti questi casi il costo totale ottenuto dopo tale operazione sarà di 1 + il costo precedentemente calcolato. Dalla definizione di *Edit Distance* si ricava infatti che il costo di ogni singola operazione è da considerarsi unitario, qualunque sia l'operazione in questione.

L'algorimo di calcolo potrà dunque riempire dinamicamente la matrice dei costi $C_{|x|,|y|}$, procedendo dall'angolo superiore sinistro verso l'angolo inferiore destro. In tal modo gli elementi sinistro, superiore e superiore sinistro saranno sempre a disposizione in quanto calcolati nei passi precedenti.

1.2.3 Ottimizzazione del calcolo dell'*Edit Distance*

Il confronto fra due stringhe mediante un tale algoritmo richiede dunque un tempo $O(|x||y|)$. Tali prestazioni possono essere migliorate con una opportuna politica di ricerca, la quale mirati al confronto in sequenza di stringhe aventi lo stesso suffisso. In tal modo è possibile sfruttare le parti comuni già calcolate all'interno della matrice stessa.

Riportiamo in Tab. 1.1 la matrice dei costi riguardante il confronto fra la stringa "MANTO" e la stringa "MANO" ed in Tab. 1.2 la matrice dei costi riguardante il confronto della stringa "MANTO" con sè stessa.

		M	**A**	**N**	**T**	**O**
	0	1	2	3	4	5
M	1	0	1	2	3	4
A	2	1	0	1	2	3
N	3	2	1	0	1	2
O	4	3	2	1	1	1

Tabella 1.1: Edit Distance fra "MANTO" e "MANO"

		M	**A**	**N**	**T**	**O**
	0	1	2	3	4	5
M	1	0	1	2	3	4
A	2	1	0	1	2	3
N	3	2	1	0	1	2
T	4	3	2	1	0	1
O	5	4	3	2	1	0

Tabella 1.2: Edit Distance fra "MANTO" e "MANTO"

Dal confronto delle due matrici dei costi risulta evidente come, al variare dei suffissi delle parole in esame, vari solo la parte terminale della tabella. Si può notare infatti che solo l'ultima riga della prima tabella e le ultime due della seconda risultano interessate dalla sostituzione della parola "MANO" con la parola "MANTO". Ciò risulta perfettamente normale in quanto tali parole hanno il medesimo suffisso "MAN", il quale, rimanendo invariato, conserva il medesimo costo.

Supponendo quindi di dover confrontare tutte le parole di un testo con una parola data, risulterebbe particolarmente interessante poterne eseguire prima un ordinamento e quindi calcolare le matrici dei costi sfruttando le operazioni effettuate sui prefissi comuni. In tal modo non solo si risparmierebbe tempo ma anche memoria in quanto per il calcolo delle parole sucessive non è necessario mantenere la conoscenza di tutta la sottomatrice bensì la sola ultima colonna unitamente all'ultima riga. Tale coppia prende normalmente il nome di *Corner*. Con riferimento agli esempi precedenti sarebbe infatti sufficente mantenere il corner composto dalla terza riga e dalla terza colonna per poter calcolare i costi di modifica delle stringhe "MANTO"-"MANO" e "MANI"-"MANO" (Vedere Tab. 1.1). Ovviamente tale ottimizzazione può essere perseguita solamente tramite lo studio di opportuni algoritmi di ricerca ed avvalendosi di strutture dati particolarmente evolute, quali ad esempio i *Suffix Tree* o i *Suffix Array*.

		M	**A**	**N**	**T**	**O**
				3	4	5
M				2	3	4
A				1	2	3
N	3	2	1	0	1	2
O	4	3	2	1	1	1

		M	**A**	**N**	**I**
				3	4
M				2	3
A				1	2
N	3	2	1	0	1
O	4	3	2	1	1

Figura 1.1: Edit Distance (con Corner) fra "MANTO" e "MANO" e fra "MANI" e "MANO"

1.3 Suffix Tree

I *Suffix Tree*[8][9][10] sono strutture dati largamente utilizzate per l'analisi e l'elaborazione del testo. Grazie ad essi è infatti possibile realizzare un indice col quale muoversi velocemente all'interno di una stringa o di un generico documento.

1.3.1 Struttura di un *Suffix Tree*

Un *Suffix Tree* è costituito da una struttura ad albero, i cui nodi contengono le informazioni necessarie per ricostruire i suffissi della stringa S sulla quale è stato esso stesso costruito. Data una qualunque posizione i, con $i \in \{0..|S|\}$, essa definisce automaticamente un *suffisso* all'interno della stringa S, il quale verrà indicato in seguito come $S_{i...|s|}$. In particolare, all'interno dei nodi intermedi, vengono memorizzate le formazioni relative ai segmenti iniziali dei singoli suffissi, sopprimendo i caratteri inutili ai fini della distinzione fra i singoli suffissi. In tal modo si ottiene un albero in cui vengono compattati i nodi unari. Nei nodi terminali (detti anche *Foglie*) vengono memorizzati i puntatori ai suffissi all'interno della stringa. Ogni foglia rappresenta un suffisso ed ogni nodo interno rappresenta un'unica sottostringa di S. Ogni sottostringa può essere raggiunta tramite un percorso che, avendo origine nella radice dell'albero, attraversi ogni nodo rappresentante una sottostringa ax. Nell'ultimo di questi nodi è quindi contenuto un collegamento al nodo rappresentante la sottostringa x.

Prendiamo in esame una stringa di esempio quale "abracadabra" ed andiamone a realizzare il *Suffix Tree* corrispondente (Fig. 1.2).

In figura Fig. 1.2 si può notare la struttura del Suffix Tree composto sui suffissi della parola "abracadabra". I quadrati indicano le posizioni all'interno del testo (*indici*, vedere Tab. 1.3) mentre il simbolo \$ rappresenta la fine del suffisso. I cerchi rappresentano invece i nodi dell'albero e racchiudono al loro interno i salti da effettuare per individuare il suffisso cercato.

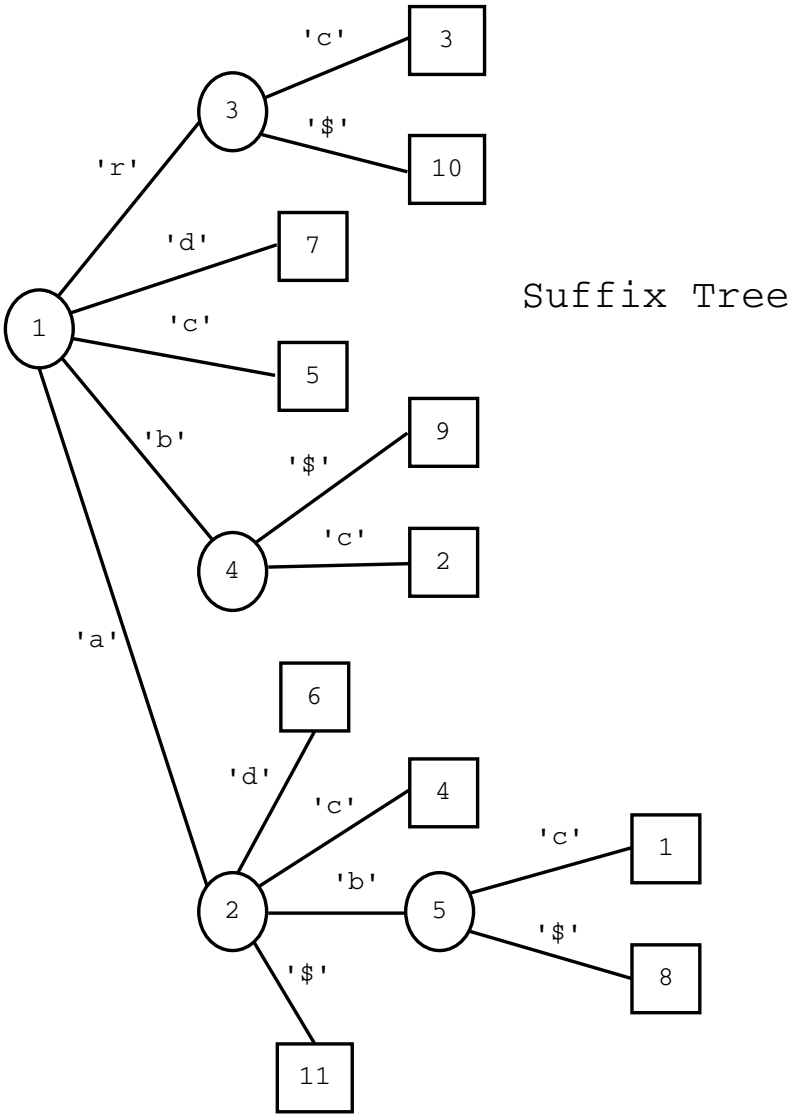

Figura 1.2: Suffix Tree della parola “abracadabra”

Indice	1	2	3	4	5	6	7	8	9	10	11
Stringa	a	b	r	a	c	a	d	a	b	r	a

Tabella 1.3: Stringa “abracadabra” con indici

1.3.2 Esplorazione di un *Suffix Tree*

Supponiamo ad esempio di iniziare la ricerca del suffisso “ra”. In tal caso ci sposteremo dal nodo radice (1) al nodo (3) attraverso l’arco indicato dalla prima lettera del suffisso, che nel nostro caso è la lettera “r”. All’interno del nodo (3) è memorizzata l’informazione del numero di caratteri ininfluenti ai fini decisionali per la scelta del prossimo percorso. Il carattere “a” infatti risulta essere presente in entrambi i suffissi “ra” presenti nel testo. Dovrà dunque essere saltato in favore del carattere ad esso sucessivo. I due archi uscenti dal nodo (3) sono infatti contraddistinti dai simboli “c” e “$”, i quali ci porteranno rispettivamente al terzo ed al decimo elemento della stringa sulla quale è stato costruito l’indice. In tal modo è quindi possibile individuare l’esatta posizione di un segmento della stringa all’interno della stringa stessa. Ovviamente nulla vieta di estendere tale approccio, con i dovuti accorgimenti, ad un intero testo, creando così un indice globale per l’insieme di tutte le parole in esso contenute.

1.4 Suffix Array

Come abbiamo visto i *Suffix Tree* sono particolarmente indicati per la ricerca di tutte le occorrenze di una stringa (e/o sottostringa) all'interno di un testo, per l'analisi delle sequenze ripetute e per il matching approssimato. Inoltre essi sono particolarmente versatili e possono essere utilizzati anche per problemi di confronto fra testi (es: analisi dei duplicati) e di analisi di sequenze di tipo genetico. Nasce quindi naturalmente l'idea di realizzarne un'implementazione migliorata, non tanto in termini di efficacia quanto in termini di efficenza, riducendone l'occupazione di memoria a parità di prestazioni. Viene così ideata una nuova struttura dati denominata *Suffix Array*[11].

1.4.1 Struttura di un *Suffix Array*

Un *Suffix Array* è costituito da una lista, alfabeticamente ordinata, contenente i suffissi del testo da indicizzare. Dato un testo $A = a_0 a_1 \cdots a_{N-1}$ di lunghezza N, indichiamo con $A_i = a_i a_{i+1} \cdots a_{N-1}$ i suffissi di A che iniziano alla i_esima posizione in A. Indichiamo inoltre con Pos l'array lessicograficamente ordinato dei suffissi di A e con $Pos[k]$ la posizione iniziale del k_esimo suffisso di A. Possiamo dunque affermare che, all'interno del *Suffix Array* così generato, vale la seguente condizione: $A_{Pos[0]} \leq A_{Pos[1]} \leq \cdots \leq A_{Pos[N-1]}$, ove con il simbolo $\leq$ si vuole indicare una diseguaglianza di tipo lessicografico.

Definiamo quindi il simbolo u^p come il prefisso della stringa u composto dai primi p caratteri della stringa stessa o come la stringa nel caso in cui $|u| \leq p$. Definiamo inoltre la relazione $<_p$ come l'ordinamento lessicografico ottenuto sui primi p caratteri di ogni stringa. In altri termini, date due stringhe u e v diremo che $u <_p v$ se e solo se $u^p < v^p$. In modo analogo definiamo le relazioni $\leq_p$, $=_p$, $\geq_p$ e $>_p$. Si noti come, indipendentemente dalla scelta di p, il *Suffix Array* "*Pos*" risulta sempre ordinato secondo la relazione $\leq_p$ in quanto $u < v$ implica $u \leq_p v$. Tutti i suffissi aventi lo stesso prefisso di p elementi, con $p < N$, avranno delle posizioni adiacenti all'interno dell'array Pos.

In merito all'esempio "abracadabra", utilizzato per dare una rappresentazione della struttura di un *Suffix Tree*, riportiamo di seguito (Fig. 1.3) la struttura del *Suffix Array* equivalente.

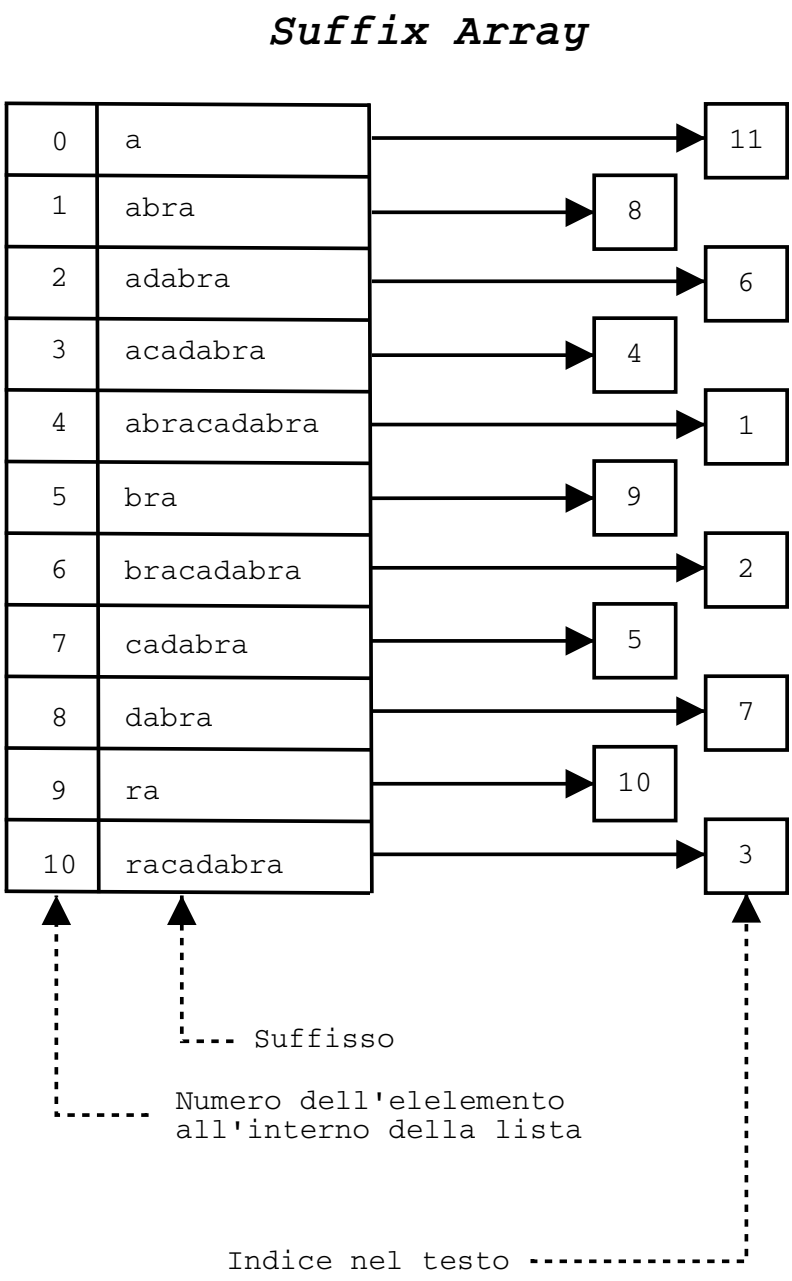

Figura 1.3: Suffix Array della parola “abracadabra”

Come si può notare ad ogni suffisso è associata la propria posizione iniziale all'interno del testo. In tal modo è possibile ricercare tutte le occorrenze di un suffisso all'interno del testo stesso. Si noti come l'impostazione di un ordinamento lessicografico, all'interno della lista dei suffissi, consenta di ottimizzare le ricerche in quanto consente di mantenere compatti fra loro i suffissi simili.

Per quanto rigurada lo spazio occupato in memoria, i *Suffix Array* risultano essere da tre a cinque volte più compatti dei *Suffix Tree*, il ché li rende particolarmente indicati per indicizzare testi di notevole entità.

1.4.2 Esplorazione di un *Suffix Array*

Grazie ad opportuni algoritmi di ricerca i *Suffix Array*, possono essere esplorati in modo analogo a quanto accade per i *Suffix Tree* e con le medesime prestazioni. Dato infatti un *Testo* A ed un *Pattern* W, rispettivamente di lunghezza N e P, entrambi costruiti su di un comune alfabeto $\sum$, supponiamo di voler cercare tutte le istanze della stringa $W = w_0 w_1 \cdots w_{p-1}$ all'interno del testo A. Sia $L_w = \min(k : W \leq_p A_{Pos[k]} \text{ o } k = N)$ e $R_w = \max(k : A_{Pos[k]} \leq_p W \text{ o } k = -1)$. Poiché Pos è ordinato secondo la relazione $\leq_p$ ne consegue che gli elementi di W combaciano con gli elementi $a_i a_{i+1} \cdots a_{i+P-1}$ se e solo se $i = Pos[k]$ per $k \in [L_w, R_w]$. Il numero di suffissi così individuati sarà $R_w - L_w + 1$ e le loro posizioni iniziali saranno $Pos[L_w], Pos[L_w + 1], ..., Pos[R_w]$. Ora, poiché l'array Pos è ordinato secondo la relazione $\leq_p$, la ricerca di L_w e R_w può essere effettuata con un semplice algoritmo di ricerca binaria, il quale richiederà al massimo $O(\log N)$ confronti fra stringhe di al massimo p caratteri, ognuno dei quali richiederà $O(P)$ confronti fra caratteri. In tal modo ci è possibile individuare tutte le istanze di una stringa W, all'interno di un testo A, in un tempo $O(P \log N)$[11].

Il tempo necessario alla ricerca delle occorrenze di W in A sarà generalmente $O(P \log |\sum|)$[8] per un *Suffix Tree* e $O(P + \log N)$[11] per un *Suffix Array*, mentre i rispettivi tempi di costruzione saranno $O(N \log |\sum|)$[8] e $O(N \log N)$[11].

Risulta quindi evidente come le prestazioni ottenute dipendano fortemente dal tipo di alfabeto utilizzato e dalla lunghezza dei testi.

Riportiamo di seguito un semplice algoritmo capace di svolgere il calcolo di L_w.

```
if W ≤_p A_Pos[0] then
        L_w ← 0
else if W >_p A_Pos[N−1] then
        L_w ← N
else
{
        (L, R) ← (0, N − 1)
        while R − L > 1 do
        {
                M ← (R + L)/2
                if W ≤_p A_Pos[M] then
                        R ← M
                else
                        L ← M
        }
        L_w ← M
}
```

Figura 1.4: Algoritmo per il calcolo di L_w

Supponiamo, ad esempio, di voler ricercare la sottostringa “ra”. Analizziamo dunque la lista dei suffissi ed estraiamo i soli elementi che inizino per “ra”. Gli indici associati a tali elementi ci daranno tutte e sole le posizioni di tale sottostringa all’interno del testo.

1.4.3 Creazione di un *Suffix Array*

La creazione di un *Suffix Array* prevede innanzitutto la capacità di scomporre il testo in ognuno dei suoi suffissi. In generale si può procedere, partendo dalla fine (o dall'inizio) del testo, ad estrarre i singoli suffissi e a memorizzarli in una lista. Una volta esequita tale operazione rimane semplicemente da ordinare la lista in modo lessicografico. Nel nostro caso tale operazione può essere eseguita con l'ausilio di un qualunque algoritmo di ordinamento in quanto la creazione degli indici rimane un'operazione preliminare, la quale quindi non inficia sull'analisi sperimentale compiuta.

Per completezza indichiamo che esiste un appropriato algoritmo di ordinamento, basato sul principio di similarità dei prefissi, il quale consente di ottenere un ordinamento della lista in tempi dell'ordine $O(N \log N)$. Per una descrizione dettagliata di tale algoritmo rimandiamo a [11].

Capitolo 2

Algoritmi per All-Against-All Sequence Matching

Il problema del match approssimato fra stringhe (o sequenze) può essere espresso nei seguenti termini[2]:

Definizione: All-Against-All Sequence Matching

> Dato un insieme di sequenze D ed un insieme di sequenze Q, non necessariamente distinti, siano d una soglia di minima distanza e $minL$ una lunghezza minima. Determinare tutte le coppie di sequenze $(S_1[i_1...j_1], S_2[i_2...j_2])$ tali che $S_1 \in D$, $S_2 \in Q$, $(j_i - i_1 + 1) \geq minL$, $(j_2 - i_2 + 1) \geq minL$ e la funzione *Edit Distance* calcolata su $(S_1[i_1...j_1], S_2[i_2...j_2])$ sia $\leq d$.

In questo capitolo andremo ad esporre gli algoritmi da noi utilizzati per il calcolo del confronto di tutte le sequenze (e sottosequenze) del testo dato con tutte le sequenze (e sottosequenze) del pattern di ricerca (*All-Against-All*).

2.1 Sub^{2}Matching Approssimato

Tale tecnica si basa sull'individuazione di coppie di *q-grammi posizionali*[2][14][15][16], contenuti all'interno di due stringhe, le quali soddisfino i vincoli imposti sia sulla lunghezza minima che sulla massima distanza consentita. Il riconoscimento di tali elementi viene effettuato con l'ausilio di più filtri posti

in cascata, i quali hanno il compito di eliminare dal set dei possibili risultati tutte le coppie le quali non soddisfano i requisiti richiesti. L'algoritmo ottenuto in questo modo risulta essere particolarmente performante, ragion per cui è stato utilizzato come campione di riferimento all'interno dei nostri test.

Prima di proseguire con la trattazione del suddetto algoritmo, riportiamo la definizione di *q-gramma posizionale*.

2.1.1 *q-grammi* Posizionali

Definizione: q-gramma Posizionale

> Un *q-gramma* posizionale di una sequenza s è costituito dalla coppia $(i; [i, ..., i+q-1])$, ove $[i, ..., i+q-1]$ è il *q-gramma* di s avente i come posizione di partenza all'interno di s. Indicheremo con G_s l'insieme di tutti i q-grammi posizionali di s costituito da tutte le $|s| + q - 1$ coppie costituite dai *q-grammi* di s.

Per quanto riguarda il match eseguito sulle stringhe intere valgono le seguenti proprietà:

Proposizione 1 (Count Filtering):

> Date le sequenze s_1 e s_2, se la loro distanza è minore di d allora la cardinalità di $G_{s_1} \cap G_{s_2} \leq \max(|s_1|, |s_2|) - 1 - (d-1) * q$, indipendentemente dalla posizione delle sequenze stesse.

Proposizione 2 (Position Filtering):

> Se le sequenze s_1 e s_2 distano al massimo d allora il *q-gramma* della prima non può differire dal *q-gramma* della seconda per più di d.

Come vedremo di seguito, gli stessi principi possono essere sfruttati per il calcolo di match fra sottostringhe.

2.1.2 Sub^{2}Count Filtering

Il Concetto alla base del *Sub^{2}Count Filtering* [2] prevede che entrambe le stringhe abbiano un determinato numero minimo di *q-grammi*, in modo che un *q-gramma* della prima stringa possa contenere una parte comune ad un *q-gramma* della seconda. In particolare il numero di *q-grammi* comuni deve essere determinato in funzione dei soli caratteri (simboli) delle sequenze stesse, senza l'uso di estensioni delle sequenze.

Proposizione:

Siano S_1 e S_2 due sequenze aventi una coppia $(S_1[i_1, ..., j_1, S_2[i_2...j_2])$ tale che $(j_i - i_1 + 1) \geq minL$, $(j_2 - i_2 + 1) \geq minL$ e la funzione *Edit Distance* calcolata su $(S_1[i_1...j_1], S_2[i_2...j_2])$ sia $\leq d$, allora la cardinalità di $G_{S_1} \cap G_{S_2}$ $\leq minL + 1 - (d + 1) * q$.

Ciò detto, partendo dalla soglia imposta nella *Proposizione 1*, $\max(|S_1|, |S_2|) - 11(d-1)*q$, sostituendo $\max(|S_1|, |S_2|)$ con la soglia di lunghezza minima $minL$ e sottraendo il numero di *q-grammi* ottenuti espandendo la stringa con l'uso di caratteri giolly, otteniamo la formula $minL - 1 - (d-1) * q - (q-1) * 2$, la quale può anche essere espressa come $minL + 1 - (d + 1) * q$. In tal modo è dunque possibile imporre un primo vincolo alla cardinalità dell'insieme composto da tutte le parti comuni dei *q-grammi* di S_1 e S_2.

2.1.3 Sub^{2}Position Filtering

Come abbiamo visto il filtro *Sub^{2}Count Filtering* non tiene conto delle posizioni delle parti comuni all'interno delle sequenze analizzate. Tale approccio, pur essendo particolarmente performante, non si rivela adatto nel caso in cui, ad esempio, si voglia trattare un testo scritto o, più in generale, un insieme di sequenze in cui la posizione relativa dei simboli sia significativa. Nasce quindi la necessità di realizzare un algoritmo capace di tener traccia di tali informazioni.

La tecnica del *Sub^{2}Position Filtering* [2] prevede quindi di effettuare un'analisi dinamica delle posizioni relative e, in parte, dell'ordine delle parole uguali all'interno delle stringhe da confrontare.

A seguire riportiamo lo schema di funzionamento dell'algoritmo utilizzato per realizzare tale tecnica.

Come si può vedere il filtro riceve in ingresso le due sequenze S_1, S_2, la soglia di lunghezza minima $minL$ e la distanza massima d calcolata con l'algoritmo dell'*Edit Distance*. L'algoritmo implementa quindi due cicli, uno esterno per le parole della sequenza S_2 ed uno interno per le parole di S_1. Ad ogni posizione p_1 di S_1 viene associato un contatore $S_1c[p_1]$ quindi, ogni volta che una parola $S_2[p_2]$, appartenenete alla stringa S_2, risulta uguale ad una parola $S_1[p_1]$, appartentente alla stringa S_1, vengono incrementati tutti i contatori da $S_1c[p_1]$ a $S_1c[p_1+w-1]$, con $w = minL$ indicante l'ampiezza della finestra del filtro. Il filtro termina positivamente qualora un generico contatore $S_1c[p_1]$ raggiunge una soglia di conteggio c fissata e contemporaneamente la parola da esso associata $S_1[p_1]$ risulta uguale alla parola $S_2[p_2]$ del ciclo esterno. In ogni altro caso il filtro darà esito negativo. Le coppie di sequenze così individuate vanno infine sottoposte ad un ultimo passagio all'interno del quale ne viene calcolata, e valutata, la distanza, la quale deve essere minore della soglia di massima distanza d.

In tal modo è vengono dunque individuate tutte le coppie di sequenze S_1 e S_2 distanti al massimo d (secondo l'*Edit Distance*) e lunghe almeno $minL$ simboli.

Si noti come la soglia c utilizzata in tale filtro sia la stessa del $Sub^2 Count$ *Filtering* con lunghezza del *q-gramma* pari a 1 (es. $q = 1$).

Algoritmo per il calcolo del *Sub^{2}Position Filtering:* [2]

```
sub²PosFilter (String S1, String S2, int minL, int d)
{
    int w ← minL;
    int c ← minL − d;
    int[] S1c;
    int P1, P2;
    boolean S1lim[];

    for (p2 = 1, ..., |S2|)
    {
        if (p2 − w > 0)
        {
            S1lim ← false;
            for (p1 = 1, ..., |S1|)
            {
                if (S1[p1] = S2[p2 − w])
                {
                    for (i = 0, ..., w − 1)
                    {
                        if (!S1lim[p1 + i])
                        {   S1c[p1 + i] − −;   S1lim[p1 + i] ← true;   }
                    }
                }
            }
        }
        S1lim ← false;
        for (p1 = 1, ..., |S1|
        {
            if (S1[p1] = S2[p2])
            {
                for (i = 0, ..., w − 1)
                {
                    if (!S1lim[p1 + i])
                    {   S1c[p1 + i] + +;   S1lim[p1 + 1] ← true;   }
                }
                if (S1c[p1] ≥ c) return true;
            }
        }
    }
    return false;
}
```

2.2 All-Against-All con *Suffix Tree*

In [1] viene proposto il seguente algoritmo per il calcolo dell'*All-Against-All Matching*. Esso si basa sul classico algoritmo "*Dynamic Programming Similarity (DPS)*[12]". Date due sequenze x e y composte con un alfabeto di dimensione s, una matrice dei costi di similarità M ed un costo di cancellazione D, tutti a valori interi, l'algoritmo *DPS* calcolerà una matrice $|x| \times |y|$ utilizzando la seguente regola[13]:

$$C(i,j) = max(C(i-1,j) + D, C(i,j-1) + D, C(i-1,j-1) + M[x_i, y_j])$$

ove $C(0,i) = iD$ e $C(j,0) = jD$. La similarità è basata sulla matrice dei costi utilizzata per il calcolo, con la quale è possibile determinare l'allineamento che massimizza le corrispondenze. Quindi possiamo dire che due sequenze x e y risultano simili (ovvero fanno un match approssimato) se esiste un valore della matrice maggiore o uguale ad una determinata soglia G, intera positiva. Il calcolo della matrice C termina una volta raggiunta tale soglia.

Per poter operare in modo efficente, l'algoritmo si avvale dell'utilizzo di un indice il quale può essere implementato mediante un *Suffix Tree*. L'algoritmo in questione simula il *DPS* percorrendo l'indice secondo la politica "Depth-First", confrontando così fra loro direttamente i sottoalberi e riducendo in tal modo il numero di confronti da effettuare. Ogni sottoalbero contiene infatti la parte iniziale, comune, delle stringhe da esso derivanti il ché consente di eseguire il calcolo della matrice dei costi una sola volta per la stessa coppia di prefissi. In tal modo è quindi possibile scartare quei sottoinsiemi di sequenze i cui prefissi comuni siano già tali da determinarne l'imposibilità di soddisfare i requisiti richiesti. Tale considerazione permette quindi di migliorare sensibilmente i tempi di esecuzione dell'algoritmo stesso, il quale può essere sintetizzato nel modo seguente.

Algoritmo di *All-Against-All con Suffix Tree* [1]

```
All-Against-All matching(Trie Root, int G)
{
    Stack ActiveSet ← Empty;
    Set Sols ← Empty;
    Trie T1, T2, t1, t2;
    int C, c;

    Push( ActiveSet, [Root, Root, ZeroCorner];
    while ActiveSet ≠ Empty do
    {           [T1, T2, C] ← Pop(ActiveSet);
        for (ogni sottoalbero non nullo t1 ∈ T1) do
        {
            for (ogni sottoalbero non nullo t2 ∈ T2) do
                {
                    \* Valutazione del grado di match fra le parti *\

                    c ← ExpandCorner(C, Symbol(t1), Symbol(t2) );
                    if (Cost(c) ≥ G) then
                    { \* Filtro *\
                        if ( (Size(t1) > 1) or (t1 ≠ t2) ) then
                        Sols ← Sols ∪ {[t1, t2, c]}
                    }
                    elseif (Cost(c) > 0) then
                    {
                        Push(ActiveSet, [t1, t2, c]);
                    }

                }
        }
    }
    return( Sols );
}
```

Figura 2.1: Algoritmo "*All-Against-All con Suffix Tree*"[1]

I punti chiave dell'algoritmo di Fig. 2.1 possono essere così evidenziati:

1. Un insieme di coppie di sottoalberi viene mantenuto in memoria e per ognuno di essi viene conservato un "*corner*" della matrice dei costi C, il quale consiste nell'ultima colonna a destra e nell'ultima riga in basso.

2. Gli indici (alberi) vengono attraversati secondo una politica "Depth-First", la quale consente di mantenere mediamente in memoria solo $2s \log n$ coppie di sottoalberi, con s il numero di simboli dell'alfabeto utilizzato.

3. Prima che ogni coppia mantenuta attiva (nello stack in memoria) venga eliminata ne vengono esplorate tutte le possibili combinazioni di sottoalberi. Per ognuna di esse viene calcolato un nuovo "*corner*" sulla base di quello corrente e sui sottoalberi da cui tale coppia ha avuto origine. Se il "*corner*" così calcolato soddisfa i requisiti viene a sua volta memorizzato insieme alla coppia di sottoalberi ad esso associata, in caso contrario viene semplicemente scartato.

Al termine dell'esecuzione, le coppie di sottoalberi indicate come "attive" costituiscono i match parziali cercati ed il corner ad esse associate fornisce le informazioni necessarie per decretarne il grado di corrispondenza. Essendo inoltre tali coppie all'interno di un indice sarà altresì possibile determinare la posizione esatta delle sequenze all'interno del testo indicizzato.

L'approccio proposto da Baeza-Yates e Gonnet[1] è stato prima implementato e quindi confrontato con quello proposto in [2]. Come vedremo nel *Capitolo 3* la funzione di misura di confronto fra le parti quì proposta è stata sostituita con quella dell'*Edit Distance*, mentre il *Suffix Tree* è stato implementato con un *Suffix Arary*. Si può notare come tale modifica non infici sulla logica dell'algoritmo, il quale mantiene la sua forma indipendentemente dal tipo di funzione di filtro utilizzata.

Capitolo 3

Implementazione

Il problema che abbiamo affrontato consiste dunque nel confronto di tutte le sequenze e sottosequenze di un primo testo A con tutte le sequenze e sottosequenze di un secondo testo B. Tale operazione deve poter essere svolta sia su sequenze di tipo genetico quali parti del DNA o sequenze di amminoacidi, sia su testi veri e propri, come ad esempio un manuale di istruzioni o un romanzo. Quest'ultima peculiarità ci ha consentito di comparare in modo approfondito le prestazioni del nuovo codice appena sviluppato, con quelle del codice basato sull'algoritmo del *Sub*^{2}*Matching*, il quale era già stato implementato in precedenza. Un ulteriore accorgimento è stato necessario per la gestione delle sequenze.

Esiste infatti la necessità di riferire direttamente ognuna di esse e per far ciò abbiamo quindi modificato la struttura dell'indice il quale, rimanendo comunque basato sul modello di un *Suffix Array*, consente ora di memorizzare anche il numero di sequenza relativo ad ogni suffisso in esso contenuto.

3.1 Testo e Genetica

Il duplice aspetto di dover operare sia con sequenze genetiche che con del testo ci ha costretto a pensare una struttura la quale, non solo potesse facilmente adattarsi ad entrambe le situazioni ma lo facesse anche nel modo più trasparente possibile. Nasce così l'idea di implementare un rudimentale dizionario il cui compito sarà quello di definire quali siano i simboli ammesi all'interno del

testo. Grazie ad essa il testo verrà poi riscritto come una sequenza di interi la quale risulterà così decontestualizzata e potrà essere facilmente elaborata dagli strati sucessivi del software. Si noti come un qualunque simbolo presente all'interno del testo ma assente dalla tabella dei simboli venga di fatto eliminato. Tale politica consente, qualora lo si desideri, di operare un filtraggio preliminare delle parti del testo che risultino di scarso interesse ai fini dell'analisi delle occorrenze (es. congiunzioni, separatori superflui, ecc...).

3.1.1 Tabella dei Simboli

Viene quindi realizzata una tabella dei simboli la quale assume la forma di un array ordinato avente il compito di instaurare un'associazione biunivoca fra un simbolo dato ed un intero non negativo (indice dell'array). I simboli utilizzabili non sono soggetti ad alcun vincolo di lunghezza e possono essere quindi semplici lettere o intere parole. Non vi sono caratteri proibiti eccezion fatta per i caratteri utilizzati dal programma come separatori di simboli e/o di sequenze ([Blank], [CR], [,] e [.]). La tabella dei simboli può essere generata dinamicamente, a partire da un file di testo, o caricata durante l'esecuzione del programma. In entrambi i casi l'algoritmo esegue le medesime operazioni. Per prima cosa separa i simboli fra loro e li inserisce all'interno di un array, in secondo luogo elimina i simboli duplicati, mantenendo una sola istanza di ogni simbolo e quindi termina eseguendo un ordinamento lessicografico sui simboli stessi. Tale accorgimento consente di risalire velocemente al testo iniziale dal testo codificato ma soprattutto consente di velocizzare la fase di codifica in quanto gli interi associati, costituiti dagli indici dell'array, possono essere discriminati con una semplice ricerca binaria sui valori contenuti nell'array stesso.

3.1.2 Conversione del Testo

Volendo inserire un nuovo testo nel programma è dunque necessario innanzitutto caricare una tabella dei simboli. Si noti come tale tabella può essere generata a partire dal testo stesso. In secondo luogo verrà eseguita la conversione del testo. In tale operazione risulta di notevole utilità una specifica funzione implementata nella tabella dei simboli, la quale indica se i simboli in

essa contenuti sono tutti di tipo “mono-carattere” o composti da più caratteri. Questa distinzione risulta di particolare importanza per una corretta codifica del testo, basti infatti pensare alla differenza fra la parola “AGATA” e la sequenza genetica “AGATA” (o “A-G-T-A”). Utilizzando una tabella dei simboli di testo, come “AGATA, BEATRICE, ...”, la codifica della prima potrebbe essere semplicemente “00”. Quella della seconda, con una tabella genetica quale “A C T G”, potrebbe essere “0002000300” (o “00-02-00-03-00”). A fronte di tale informazione il testo inserito viene quindi convertito simbolo per simbolo in funzione della tabella ed inserendo un intero negativo (solitamente “-10”) come separatore di sequenza. Il riultato finale consiste in un nuovo file (generalmente marcato con l’estensione “.sym”) composto da sequenze di interi non negativi, le quali risultano inframezzate da interi negativi (“-10”). Come abbiamo detto i primi rappresentano i simboli del testo originale mentre i secondi i separatori di sequenze.

3.2 Indicizzazione del Testo

Una volta convertito il testo in una sequenza, il flusso del programma (Fig. 3.1) passa al layer inferiore il quale è preposto alla creazione dell’indice. Come già accennato in precedenza esso è costituito da un *Suffix Array* opportunamente modificato per poter gestire le sequenze. La struttura prevede infatti un array a due colonne. La prima contiene il la posizione del primo simbolo del suffisso all’interno del testo codificato, la seconda il numero della sequenza alla quale appartiene. La nostra implementazione fa sì che, all’interno del *Suffix Array*, che rimane comunque generato su tutto il testo, possano essere presenti due suffissi *identici* i quali differiscano per il solo codice di sequenza. In realtà codesta eventualità non interferisce nè con il funzionamento del *Suffix Array* nè con gli algoritmi di esplorazione dello stesso, attraverso i quali esso continua ad essere riconducibile ad un *Suffix Tree*.

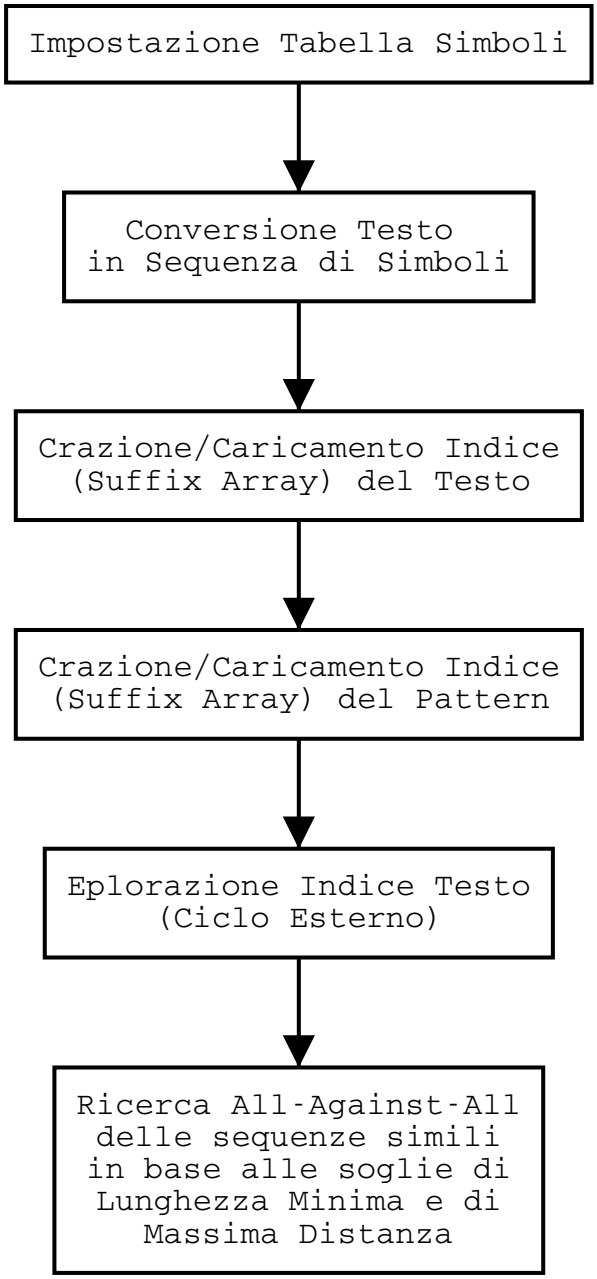

Figura 3.1: Schema di Flusso dell'Esecuzione

3.2.1 Creazione dell'Indice

In fase di creazione il testo codificato viene suddiviso in sequenze e per ognuna di esse vengono memorizzati tutti i punti di inizio dei suoi suffissi, oltre ovviamente a quello della stringa stessa. Avere in memoria i punti di inizio equivale infatti a conoscere l'intero suffisso, il quale termina con il terminatore di sequenza ("−10"). Terminata l'estrapolazione di tutti i suffissi di tutte le sequenze, il *Suffix Array* viene ordinato. Un semplice algoritmo di sort opera sugli elementi dell'array facendo riferimento non ai valori contenuti nelle sue celle, ma a quelli da esse riferiti. I puntatori al testo individuano infatti i suffissi da ordinare mantenendo così le informazioni utili senza far spreco di memoria. Terminato l'ordinamento il *Suffix Array* è pronto a svolgere la sua funzione di indice.

3.2.2 Esplorazione dell'Indice

Nel simulare un *Suffix Tree* con un *Suffix Array* la parte sicuramente più delicata, e interessantte, risulta essere quella dell'esplorazione. Mentre infatti nel primo esistono i nodi, con le loro biforcazioni, nel secondo tali nodi vanno individuati in modo dinamico, estrapolandoli a partire dai suffissi memorizzati e dal loro ordinamento. Si simula quindi un nodo radice dal quale far partire le esplorazioni iniziali. In esso si analizza il primo simbolo di ogni suffisso, effettuando una partizione dell'insieme dei suffissi in base a tale informazione. Si ottiene così il primo livello dell'albero. Ovviamente, se il primo simbolo risulta essere uguale in tutti i suffissi, si ripete l'analisi sul secondo e così via fino al raggiungimento della condizione cercata. In tal modo è quindi possibile determinare quelli che, nella teoria dei *Suffix Tree*, vengono indicati con il nome di *Prefissi Comuni*. Determinati i nodi del primo livello si reitera il procedimento per ognuno di essi e così via per quelli a seguire fino al raggiungimento dei nodi *foglia*, i quali vengono determinati dalla presenza del simbolo di fine sequenza ("−10") fra quelli da analizzare per la ricerca del livello sucessivo. In tal modo è quindi possibile risalire alla struttura di un *Suffix Tree* e l'esplorazione può dunque essere analogamente compiuta in modalità "Depth-First", così come richiesto dall'algoritmo per il match "All-Against-All".

In Fig. 3.2 vediamo come sia possibile evidenziare la corrispondenza fra

i nodi del *Suffix Tree* costruito sulla sequenza genetica “AGATA” con i suffissi contenuti nel *Suffix Array* costruito sulla medesima sequenza. Le linee tratteggiate indicano tali corrispondenze.

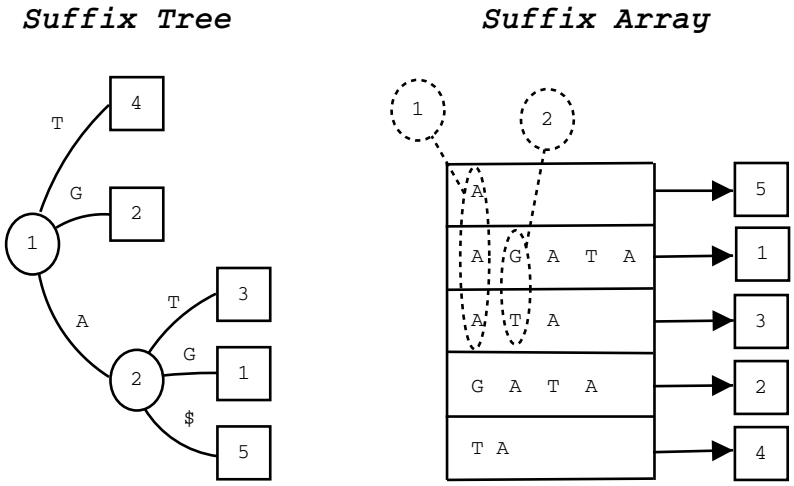

Figura 3.2: Correlazione fra Suffix Tree e Suffix Array

Indice	1	2	3	4	5
Stringa	A	G	A	T	A

Tabella 3.1: Sequenza Genetica “AGATA” con indici

3.3 All-Against-All

Come abbiamo già detto tale algoritmo si basa sul concetto di esplorare, con due cicli innestati, i *Suffix Tree* rappresentanti gli indici delle due sequenze da confrontare. Vediamo ora come sia possibile fare la stessa cosa utilizzando come indici due *Suffix Array*.

3.3.1 All-Against-All con *Suffix Array*

In prima analisi possiamo notare, facendo riferimento all'algoritmo di "All-Against-All" riportato in Fig. 2.1, come sia necessaria una struttura dati per il salvataggio dei nodi durante l'esplorazione. Essa viene indicata come un generico stack nel quale salvare la coppia di nodi ed il *corner* ad essa associato. Di basilare importanza risulta essere inoltre la funzione capace di calcolare il *corner* sucessivo, il quale non solo fornisce il grado di match ma risulta anche fondamentale per il proseguo dell'esplorazione nei sucessivi livelli dell'indice.

La nostra implementazione affronta la prima di queste due questioni implementando due stack, uno per l'indice del testo (ciclo esterno) ed uno per quello del pattern di ricerca (ciclo interno). In entrambi vengono salvati elementi contenenti le informazioni necessarie ad identificare il nodo equivalente del *Suffix Tree* all'interno del *Suffix Array*. Inoltre, per i soli nodi del ciclo interno, vengono salvate anche le informazioni del *corner*. In questo modo, ad ogni ciclo, è possibile effettuare una *Pop* delle informazioni del nodo da esaminare, calcolare quale sia il prossimo nodo mediante la funzione *FindNextNode(...)*, eseguire gli eventuali *Push* del vecchio e del nuovo nodo ed infine, se si è nel ciclo interno, valutare il grado di match dei nodi calcolando il nuovo *corner* con la funzione *NewCorner(...)*, la quale implementa il calcolo dei costi secondo l'algoritmo dell'*Edit Distance*.

Per quanto riguarda la funzione *FindNextNode(...)* essa non fa altro che implementare i concetti svilupati al punto 3.2.2 eseguendo la duplice funzionalità di individuazione dei nuovi nodi e di riconoscimento dei nodi terminali.

Di maggiore interesse risulta essere invece la scelta di mantenere le informazioni del *corner* all'interno dello stack del solo ciclo interno. Essa è corretta in quanto l'esplorazione dell'indice del pattern di ricerca viene effettuata confrontando tutti i possibili sottoalberi di tale indice (ciclo interno) con lo stesso

nodo dell'indice del testo (ciclo esterno). La medesima osservazione consente inoltre di semplificare la struttura stessa del corner. Ipotizziamo infatti di mettere lungo le colonne della matrice per il calcolo dell'*Edit Distance* i suffissi del testo (indice del ciclo esterno) e sulle righe quelli del pattern (indice del ciclo interno). Essendo fissato il suffisso del ciclo esterno, anche le colonne della matrice ne risulteranno fissate. Lo sviluppo di tale matrice avverrà quindi solo in verticale, il ché rende inutile il mantenimento dell'ultima colonna a destra della matrice all'interno del corner. In Fig. 3.3 diamo una rappresentazione visiva di tale considerazione. In tal modo ci è possibile ridurre sia l'occupazione in memoria del *corner* che il numero di operazioni necessarie a calcolarlo.

		M	**A**	**N**	**T**	**O**
	0	1	2	3	4	5
M	1	0	1	2	3	4
A	2	1	0	1	2	3
N	*3*	*2*	*1*	*0*	*1*	*2*
O	**4**	**3**	**2**	**1**	**1**	**1**

		M	**A**	**N**	**T**	**O**
	0	1	2	3	4	5
M	1	0	1	2	3	4
A	2	1	0	1	2	3
N	3	2	1	0	1	2
O	4	3	2	1	1	1
V	5	4	3	2	2	2
R	*6*	*5*	*4*	*3*	*2*	*2*
A	**7**	**6**	**5**	**4**	**3**	**3**

Figura 3.3: Sviluppo verticale del *corner*

In Fig. 3.3 abbiamo evidenziato in corsivo il corner precedente ed in grassetto il corner appena calcolato. Si può notare come per ricavare quest'ultimo sia sufficente la conoscenza della riga immediatamente precedente (corner al passo $n-1$) mentre la conoscenza della colonna risulterebbe superflua in quanto, come già detto, lo sviluppo avviene solo in modo verticale (dall'alto verso il basso). Qualora infatti il suffisso "MANTO", ottenuto dal ciclo esterno, dovesse cambiare, tutta la matrice dovrebbe essere ricalcolata. Ciò avviene ovviamente solo allo scorrere dell'indice più esterno o al variare del sottoalbero dell'indice del ciclo interno.

Operando come descritto è possibile ottenere un confronto di ogni stringa,

e sottostringa, del testo A con ogni stringa, e sottostringa, del pattern B. Resta da vedere come operare i dovuti filtraggi sui risultati così ottenuti.

3.3.2 Filtro sui risultati

Il filtro sui risultati ottenuti risulta essere, nel nostro caso, un'operazione piuttosto delicata. Essa infatti non solo consente di mantenere le sole coppie di suffissi le quali soddisfino i requisiti richiesti ma ha anche il compito di snellire il volume dei dati da elaborare. Posizionando opportunamente i controlli sui vincoli di minima lunghezza ($minL$) e di massima distanza (d), già trattati in precedenza, è infatti possibile eliminare a priori interi sottoalberi. Vediamo come.

Posizionando un controllo sulla lunghezza minima del suffisso all'interno del ciclo esterno e, più precisamente, nel controllo preposto all'avvio della ricerca sul secondo indice, è possibile evitare di effettuare inutili esplorazioni dell'indice del pattern. I suffissi del testo che non fossero abbastanza lunghi darebbero comunque luogo a match che andrebbero inevitabilmente scartati a causa del mancato rispetto del vincolo di minima lunghezza. In tal modo è quindi possibile evitare di percorrere inutilmente l'indice del pattern (ciclo interno) riducendo così i tempi di esecuzione. Rimane quindi da vedere come sfruttare il vincolo di massima distanza.

In base al ragionamento di cui al punto 3.3.1, possiamo osservare come la funzione distanza, calcolata come indicato nell'algoritmo dell'*Edit Distance*, risulti essere, una volta fissato il suffisso del testo ed il prefisso del pattern, una funzione monotona non decrescente. Con riferimento alla Figura 3.3 si noti come la distanza fra le stringhe "MANTO" e "MANOVRA" sia maggiore di quella fra le stringhe "MANTO" e "MANO" e come i costi dei corner intermedi risultino essere sempre non decrescenti. Ovviamente, visto che il suffisso posto sulle colonne (ciclo esterno) risulta essere fisso, il valore indicativo della distanza fra i suffissi in esame sarà riportato nell'ultimo elemento in basso a destra della matrice. Alla luce di tali considerazioni possiamo quindi imporre un controllo sull'esplorazione del secondo albero in modo da scartare a priori quei sottoalberi i quali darebbero luogo a match fra suffissi aventi distanza maggiore di quella consentita.

Eseguendo quindi queste modifiche all'algoritmo abbiamo ottenuto esattamente il risultato cercato, ovvero un software in grado di effettuare una ricerca di occorrenze fra tutte le possibili sequenze, e sottosequenze, di due testi dati, il quale imponga al tempo stesso un vincolo di minima lunghezza, ed uno di massima distanza, sul set dei risultati così ottenuti. L'implementazione da noi realizzata sfrutta gli accorgimenti sopra descritti e risulta essere quindi particolarmente ottimizzata.

N.B.: Il codice descritto fin'ora è presente sul CD-ROM allegato.

Capitolo 4

Analisi Sperimentale

Come già detto uno degli scopi principali del lavoro svolto è quello di verificare sperimentalmente le prestazioni che si possono ottenere utilizzando l'algoritmo di All-Against-All basato sui *Suffix Array*. In particolare si vuole confrontare l'efficenza di tale implementazione con quella realizzata basandosi sull'algoritmo del Sub^2 *Matching*. Quest'ultima infatti risulta essere particolarmente performante e si dimostra quindi un buon riferimento per eseguire un confronto.

4.1 Scelta del Data-Set

Come in ogni analisi delle prestazioni la fase di scelta dell'insieme dei dati sul quale eseguire i test ricopre una fase fondamentale. Esso deve essere infatti caratteristico del tipo di dati per i quali il software è stato pensato e allo stesso tempo deve anche essere il più variegato possibilie, racchiudendo in sè un ampio ventaglio di casi in modo da simulare al meglio una situazione reale. Esso deve inoltre non essere tanto grande da richiedere ore ed ore di calcoli ma neppure tanto piccolo da minimizzare le differenze interessanti. Infine, poichè la natura dei software che andremo a testare è fondamentalmente diversa, esso dovrà non solo rispondere ai suddetti requisiti per entrambi i programmi ma verrà anche adattato per poter essere elaborato correttamente da ognuno dei due. Infatti, il programma utilizzato come riferimento non è stato sviluppato per operare su sequenze di tipo genetico ma solo su testi, il che, oltre a costringerci ad

effettuare su di esso le dovute modifiche, ci ha imposto anche di adattare la struttura dei dati in ingresso in modo da renderla perfettamente compatibile.

Fatte tali doverose premesse possiamo descrivere il set dei dati utilizzati. Innanzitutto distinguiamo due insiemi, uno dei dati di tipo "testo" ed uno dei dati di tipo "genetico". Per il primo insieme abbiamo utilizzato parti di due differenti versioni di un manuale d'uso, il quale costituisce fondamentalmente un ottimo esempio di testo scritto. Per il secondo sono state invece utilizzate delle sequenze di DNA e di amminoacidi reperite da diverse fonti. Tutti i file utilizzati per i test sono riportati nel CD-ROM allegato.

Riportiamo a seguire, in Tab. 4.1 le principali caratteristiche dei set di sequenze utilizzati.

Set	**Tipo**	**Numero di Elementi**
DP-3	Testuale	19471
DP-5	Testuale	3575
DNA-testo-3	Genetico	360
DNA-Pattern-3	Genetico	360
DNA-testo-4	Genetico	3324
DNA-Pattern-4	Genetico	579
AA-Testo-1 (60)	Genetico	3190
AA-Pattern-1 (60)	Genetico	554
AA-Testo-1 (120)	Genetico	3190
AA-Pattern-1(120)	Genetico	554

Tabella 4.1: Data-Set

Si noti come i primi set di dati siano di tipo testuale, con un testo di riferimento (*DP-3*) molto maggiore del pattern da ricercare (*DP-5*). Essi sono derivati dai manuali di due sucessive versioni dello stesso software. *DP-3* rappresenta l'intero manuale della versione 3 mentre *DP-5* è una parte del manuale della versione 5 (Vedere test in [2]). Lo scopo di tale set è quello di simulare la ricerca di frasi, o parole, all'interno di un testo dato. Gli insiemi sucessivi di sequenze sono tutti di tipo genetico, in modo da avvicinarsi il più possibile all'ambito applicativo per il quale il software è stato realizzato. La

prima coppia (*DNA-testo-3*,*DNA-Pattern-3*) si prefigge lo scopo di simulare una vera e propria ricerca di tipo "All-Against-All". I due set contengono un ugual numero di sequenze della medesima lunghezza. Inutile dire che le sequenze in questione sono ovviamente differenti. La coppia sucessiva (*DNA-testo-4*,*DNA-Pattern-4*) vuole invece replicare il caso preso in esame con le sequenze di tipo testuale, in modo da verificare il funzionamento in caso di carico sbilanciato.

I test sucessivi sono realizzati con un alfabeto basato sugli amminoacidi anzichè sui simboli del DNA. La differenza sostanziale consiste nella dimensione dell'alfabeto, il quale consta ora di venti simboli anzichè quattro. Tali test hanno tutti un carico sbilanciato in quanto uno dei due insiemi è composto da 3190 simboli mentre l'altro da soli 554. La differenza fra tali set consiste nella diversa lunghezza, e quindi nel numero, delle sequenze ottenute, (la prima di 60 e la seconda di 120 simboli). In tal modo ci prefiggiamo di poter valutare la variazione del comportamento del software al variare della lunghezza e del numero delle sequenze.

4.2 Scelta dei Parametri

La scelta dei parametri di esecuzione della ricerca, quali la soglia di minima lunghezza $minL$ e la soglia di massima distanza d, viene effettuata in funzione del set di sequenze sul quale si vuole operare. Da essi dipende infatti non solo il numero di risultati ottenuti ma anche i tempi necessari per il singolo test. In generale abbiamo sempre cercato di impostare dei valori che consentissero di avere un discreto numero di risultati (migliaia o più) ma che al tempo stesso non implicassero tempi di attesa superiori ad alcuni minuti. In alcuni test abbiamo poi settato i parametri in modo da poter effettuare dei confronti incrociati al fine di verificare quali siano i punti di forza e quali quelli di debolezza del codice sviluppato.

4.3 Analisi dei Risultati

Riportiamo da prima le tabelle con i soli test effettuati sul software da noi realizzato, tentando di dare una valutazione generale dei risultati ottenuti. In secondo luogo andiamo ad affiancare i risultati ottenuti in alcuni test fondamentali eseguiti con entrambi i programmi, in modo da poterne effettuare un'analisi comparata.

4.3.1 Analisi Prestazionale

Iniziamo dunque con l'effetuare una serie di test mirati allo studio dell'andamento delle prestazioni al variare sia dei dati che dei parametri utilizzati. Riportiamo a seguire i risultati ottenuti con i test svolti.

Min. Lung.	**Max Dist.**	**Num. Ris.**	**Tempo [msec]**
3	1	28195	420281
6	1	2847	233032
4	2	44974	678625
4	1	8260	327406

Tabella 4.2: Test: (DP-3 , DP-5)

Min. Lung.	**Max Dist.**	**Num. Ris.**	**Tempo [msec]**
10	4	25583	204813
10	3	3910	85985
11	3	1544	82484
10	1	62	15547
6	1	2342	29734
4	1	18614	49422
3	1	53685	65468

Tabella 4.3: Test: (DNA-Testo-3 , DNA-Pattern-3)

Min. Lung.	**Max Dist.**	**Num. Ris.**	**Tempo [msec]**
10	1	205	212672

Tabella 4.4: Test: (DNA-Testo-4 , DNA-Pattern-4)

Min. Lung.	**Max Dist.**	**Num. Ris.**	**Tempo [msec]**
10	1	23735	131843
15	1	16483	114094
25	2	17746	167656

Tabella 4.5: Test: (AA-Testo-1(60) , AA-Pattern-1(60))

Min. Lung.	**Max Dist.**	**Num. Ris.**	**Tempo [msec]**
10	1	51482	393594
15	1	42367	419953
25	2	73013	810344

Tabella 4.6: Test: (AA-Testo-1(120) , AA-Pattern-1(120))

Dal confronto fra i risultati ottenuti, ed in particolare quelli riportati in Tab. 4.3, possiamo notare come, fissata la soglia di massima distanza, sia il numero di risultati ottenuti, sia i tempi impiegati calino proporzionalmente al crescere della soglia di lunghezza minima. Tale risultato non stupisce in quanto aumentando la lunghezza minima consentita vengono scartati sempre più sottorami dell'indice del ciclo esterno e quindi l'esecuzione ne risulta velocizzata. Riportiamo in Fig. 4.1 l'andamento dei tempi al crescere di $minL$ per $d = 1$.

Proseguendo con la nostra analisi possiamo notare come il test effettuato con i parametri $Min.Lung. = 6$ e $Max.Dist. = 1$ nelle tabelle 4.2 e 4.3 sia

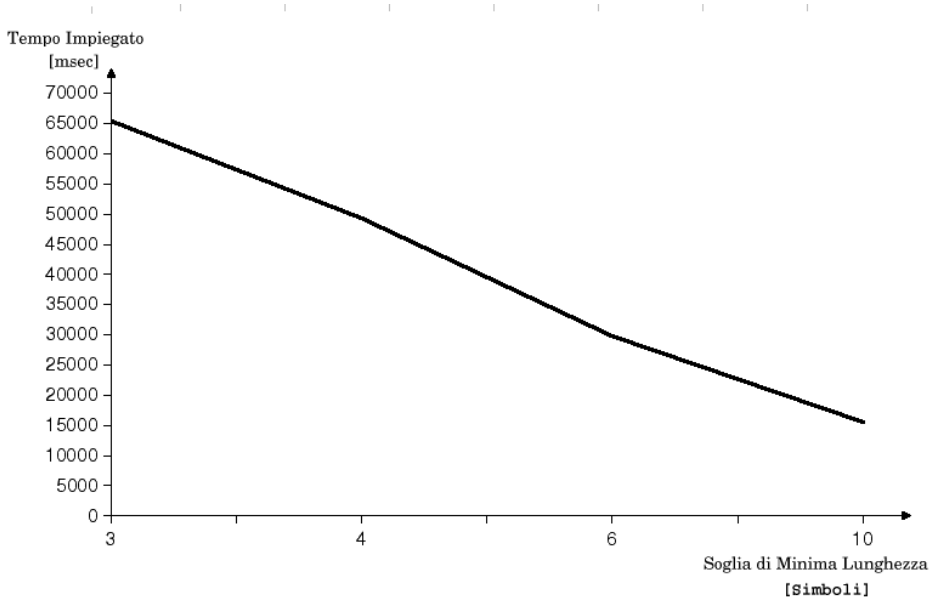

Figura 4.1: Andamento dei Tempi al crescere della soglia di Minima Lunghezza

indicativo di quanto, pur operando su basi di dati aventi un numero di elementi cento volte l'una più grande dell'altra, i tempi rilevati risultino differire per un fattore dieci. Tale osservazione ci permette di constatare fin d'ora come tale algoritmo risulti maggiormente performante in un utilizzo con sequenze di tipo genetico piuttosto che con sequenze di tipo testuale.

Le tabelle 4.5 e 4.6 mettono infine in evidenza come, in un contesto di tipo genetico, sia il numero di risultati ottenuti, sia il tempo impiegato, risultino essere maggiormente vincolati alla distanza massima consentita piuttosto che alla soglia di lunghezza minima.

Da questa prima parte dell'analisi possiamo quindi estrapolare che l'algoritmo da noi implementato sembra poter avere buone prestazioni su sequenze genetiche anche piuttosto lunghe.

Per valutare meglio tali considerazioni effettuiamo un'analisi comparativa con il software di riferimento ciato in precedenza.

4.3.2 Analisi Comparata

Come già accennato passiamo ad effettuare il confronto delle prestazioni ottenute con i due diversi programmi. Riportiamo a seguire le tabelle dei risultati ottenuti.

Min. Lung.	**Max Dist.**	**Num. Ris.**	**Tempo Sub^{2}Matching [msec]**	**Tempo A-A-A con Suffix Array [msec]**
3	1	28195	11360	420281
6	1	2847	312	233032
4	2	44974	13156	678625
4	1	8260	1547	327406

Tabella 4.7: Test: (DP-3 , DP-5)

Min. Lung.	**Max Dist.**	**Num. Ris.**	**Tempo Sub^{2}Matching [msec]**	**Tempo A-A-A con Suffix Array [msec]**
10	1	62	22297	15547
6	1	2342	22735	29734

Tabella 4.8: Test: (DNA-Testo-3 , DNA-Pattern-3)

Min. Lung.	**Max Dist.**	**Num. Ris.**	**Tempo Sub^{2}Matching [msec]**	**Tempo A-A-A con Suffix Array [msec]**
10	1	205	324563	212672

Tabella 4.9: Test: (DNA-Testo-4 , DNA-Pattern-4)

Sin dai primi test abbiamo potuto notare come il numero di match ottenuto con entrambi i programmi fosse esattamente identico. Ciò prova senza ombra di dubbio l'efficacia del software da noi sviluppato. A fronte di tale considerazione abbiamo deciso di riportare una sola volta il numero di risultati

Min. Lung.	**Max Dist.**	**Num. Ris.**	**Tempo Sub^{2}Matching [msec]**	**Tempo A-A-A con Suffix Array [msec]**
10	1	23735	3141	131843

Tabella 4.10: Test: (AA-Testo-1(60) , AA-Pattern-1(60))

Min. Lung.	**Max Dist.**	**Num. Ris.**	**Tempo Sub^{2}Matching [msec]**	**Tempo A-A-A con Suffix Array [msec]**
10	1	51482	83938	393594

Tabella 4.11: Test: (AA-Testo-1(120) , AA-Pattern-1(120))

all'interno delle tabelle dei risultati. Restano quindi da valutare le prestazioni ottenute.

Confrontando i risultati riportati in Tab. 4.7 si può notare come l'algoritmo del *Sub^{2}Matching* risulti essere, su sequenze di testo, molto più performante di quello da noi implementato. Una possibile spiegazione di tale comportamento può essere cercata nelle caratteristiche delle sequenze stesse. Un generico testo contiene infatti un elevato numero di elementi diversi. L'alfabeto con il quale viene costruita una sequenza di testo (frase) comprende infatti tutte le possibili parole contenute all'interno di un comune dizionario. Questa caratteristica è ben lontana sia dai venti simboli diversi utilizzati per codificare una sequenza di amminoacidi, sia dai quattro simboli presenti all'interno di una sequenza di DNA.

Passando ora ad analizzare le sequenze di tipo genetico (amminoacidi e DNA) possiamo notare come il comportamento generale del nostro software subisca un deciso miglioramento. In Tab. 4.8 e 4.9 possiamo notare come i tempi impiegati si avvicinino sensibilmente a quelli ottenuti con il programma di riferimento, ed arrivando, in alcuni test, ad esserne migliori.

Le ultime tabelle 4.10 e 4.11 mostrano invece chiaramente come il nostro algoritmo risulti essere meno sensibile all'allungarsi delle sequenze da analizzare. Al raddoppiare della lunghezza delle sequenze, il tempo impiegato

risulta essere circa triplo, mentre per il software di riferimento il tempo viene moltiplicato quasi di un fattore trenta.

In conclusione possiamo dunque affermare che l'algoritmo del $Sub^2 Matching$ risulta essere particolarmente adatto allo studio di sequenze di moderara lunghezza, costruite su di un alfabeto di considerevoli dimensioni. L'algoritmo da noi implementato risulta invece più idoneo per l'elaborazione di sequenze particolarmente lunghe costruite su di un alfabeto avente un numero limitato di simboli.

Capitolo 5

Conclusioni

Il lavoro svolto ci ha portato infine al conseguimento dei seguenti risultati:

- è stata implementata una struttura indice basata sui *Suffix Array* e completa di tutte le funzioni di costruzione ed esplorazione che ne rendono possibile l'utilizzo esattamente come se si operasse con un *Suffix Tree*.

- è stata implementata una tabella di conversione dinamica la quale, tramite opportune funzioni, consente di codificare un insieme di righe di testo, o una serie di sequenze di generici simboli, in un file fruibile e decontestualizzato, implementando di fatto un livello di astrazione dal tipo di dato trattato.

- è stato modificato l'algoritmo preposto al calcolo dell'*Edit Distance*, adattandolo alle particolarità del problema affrontato e dandone una implementazione ottimizzata.

- è stata realizzata un'implementazione dell'algoritmo proposto per eseguire l'*All-Against-All Sequence Matching* sfruttando i *Suffix Array*. Tale algoritmo è stato inoltre ottimizzato introducendo, oltre alla soglia di massima distanza, anche la soglia sulla minima lunghezza delle sequenze interessate al match.

- è stata quindi implementata un'interfaccia utente attraverso la quale sia possibile accedere comodamente alle funzionalità del programma e grazie

alla quale sia possibile avere una chiara rappresentazione dei risultati raggiunti. Al suo interno vengono infatti date la possiblità di:

1. Visualizzare la struttura del *Suffix Array*;
2. Visualizzare direttamente i risultati ottenuti, indicando le esatte sequenze interessate;
3. Accedere al menu delle opzioni attraverso le quali impostare i parametri di riferimento per la ricerca;
4. Generare/Caricare dinamicamente la Tabella dei Simboli.
5. Salvare/Caricare i dati elaborati e da elaborare (comprese le strutture indice);

- infine è stato possibile valutare, tramite il confronto diretto, quale sia, in termini prestazionali, il grado di efficenza ottenibile con tale implementazione. L'analisi finale rivela come essa sia indicata per operare su sequenze di tipo genetico e, allo stesso tempo, ne sia sconsigliabile l'utilizzo su documenti di testo.

Concludendo i risultati ottenuti rivelano ciò che parzialmente ci attendavamo di scoprire. Il deficit prestazionale riscontrato nell'elaborazione di documenti di testo rivela quali siano i potenziali limiti dell'approccio seguito dall'algoritmo che è stato implementato. D'altra parte i risultati positivi ottenuti sulle sequenze genetiche rendono comunque fruibile tale implementazione in una vasta gamma di interessanti applicazioni.

Bibliografia

[1] Ricardo A.Baeza-Yates, Gaston H.Gonnet, *A Fast Algorithm on Average for All-Against-All Sequence Matching* Proc. of the Int'l Workshop and Symposium on String Processing and Information Retrieval (SPIRE), pagine 16-23, anno 1999.

[2] Federica Mandreoli, Riccardo Martoglia, Paolo Tiberio *A Syntatic Approach for Searching Similarities whithin Sentences*, Proc. of the 11th ACM Conference of Information and Knowledge Management (ACM CIKM), anno 2002.

[3] Gonzalo Navarro, *A Guided Tour to Aproximate String Matching*, ACM Computing Surveys, numero 1, volume 33, pagine 31-88, anno 2001.

[4] V.Lavenshtein, *Binary Codes Capable of Correcting Spurious Insertions and Deletions of Ones*, Problem of Information Transmission, volume 1, pagine 8-17, anno 1965.

[5] D.Sankoff, J.Kruskal, *Time Warps, String Edits, and Macromulecoles: The Theory and Practice of Sequence Comparison*, Addison-Wesley, anno 1983.

[6] G.Das, R.Fleisher, L.Gasieniek, D.Gunopulos, J.Kärämen, *Episode Matching*, In Proc. CPM'97, LNCS 1264, pagine 12-27, Springer-Verlag, anno 1997.

[7] A.Apostolico, C.Guerra *The Longest Common Subsequence Problem Revisited*, Algorithmica, volume 2, pagine 315-336, anno 1987.

[8] P.Weiner, *Linear Pattern Matching Algorithms*, In Proc. IEEE Symp. on Switching and Automata Theory, pagine 1-11, anno 1973.

[9] D.E.Knuth, *The Art of Computer Programming: Sorting and Searching*, volume 3, anno 1973.

[10] A.Apostolico, Z.Galil, *Combinatorial Algorithms on Words*

[11] U.Manber, G.Myers, *Suffix Arrays: A New Method for On-Line String Searches*, In 1st ACM-SIAM Symposium on Discrete Algorithms, pagine 319-327, anno 1990.

[12] S.B.Needleman, C.D.Wunsch, *A General Method Applicable to the Search for Similarities in the Amino Acid Sequences fo Two Proteins*, J. Molec. Biol., volume 48, pagine 443-453, anno 1970.

[13] M.O.Dayhoff, R.M.Schwartz, B.C.Orcutt, *A Model of Evolutionary Change in Proteins*, volume 5, suppl. 3, pagine 345-352

[14] E.Ukkonen, *Approximate String Matching with q-grams and Maximal Matches*, Theoretical Computer Science, numero 1, volume 92, pagine 191-211, anno 1992.

[15] E.Sutien, J.Tarhio, *On Using q-gram Locations in Approximate String Matching*, Proc. of 3rd Annual European Symposium, anno 1995.

[16] E.Sutien, J.Tarhio, *Filtration with q-samples in Approximate String Matching*, Proc. of the 7th annual Symposium on Combinatorial Pattern Matching, anno 1996.