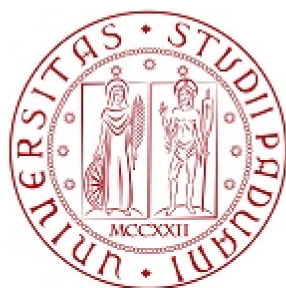

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di laurea in Ingegneria dell'Informazione

Costruzione space efficient di un suffix tree troncato con codifica TruST

LAUREANDO

Gabriele Pelizzari

RELATORE

Dott.ssa Cinzia Pizzi

ANNO ACCADEMICO 2012/2013

Sommario

La scelta di strutture dati efficienti rappresenta uno degli aspetti fondamentali nella gestione di grandi moli di dati. Questo problema investe ambiti tra loro molto diversi, come, ad esempio, il web, la bioinformatica e le scienze sociali. La progettazione di strutture dati quindi riveste un ruolo di primaria importanza in questi contesti. Storicamente, tra le strutture dati efficienti per problemi di pattern matching e discovery, figurano il suffix tree e, più recentemente, il suffix array.

L'argomento di studio di questa tesi è il suffix tree troncato, che ha il vantaggio di mantenere la struttura ad albero ma di occupare in casi di applicazione reali meno spazio del suffix tree. In particolare si è fatto riferimento alla codifica TruST.

Il procedimento originariamente utilizzato nella costruzione del suffix tree troncato (TST) con codifica TruST è basato su un TST con codifica derivata da illi e la sua efficienza, in termini di spazio occupato, dipende fortemente dal fattore di troncamento. Per questo motivo l'obiettivo di questa tesi è di ottenere un nuovo procedimento di costruzione che limiti l'occupazione spaziale di elementi non usati direttamente nella codifica TruST. L'algoritmo proposto in questa tesi basa la costruzione del TST con codifica TruST su un enhanced suffix array.

Indice

Introduzione **1**

1 Suffix tree e suffix array **2**
1.1 Suffix tree 2
1.1.1 Algoritmo di Ukkonen 4
1.1.2 Codifica illi 4
1.2 Suffix array 6
1.2.1 Enhanced suffix array 6
1.2.2 Algoritmi di costruzione 8

2 Suffix tree troncati e TruST **11**
2.1 Suffix tree troncato 11
2.1.1 Algoritmi di costruzione 13
2.1.2 Codifica derivata da illi 13
2.2 Codifica TruST 14

3 Codifica TruST da ESA **21**
3.1 Costruzione del k-factor tree da ESA 21
3.1.1 Suffix Array 22
3.1.2 Longest Common Prefix 22
3.1.3 Creazione di nodi e foglie come *lcp-interval* 23

4 Analisi delle prestazioni **27**
4.1 Analisi dell'occupazione spaziale 28
4.1.1 Algoritmo basato su illi-TST 28
4.1.2 Algoritmo basato su ESA 30
4.2 Analisi temporale 34

Conclusioni **36**

Bibliografia **37**

Introduzione

Nell'ambito della ricerca di sottostringhe, pattern matching, in sequenze di grandi dimensioni (genomi sequenziati, DNA), le strutture dati basate sui suffissi sono, attualmente, un campo di ricerca importante e in continua evoluzione.

Le strutture dati di questo tipo più usate sono i suffix tree e i suffix array. Dei primi esiste una versione ridotta: il suffix tree troncato (TST) di cui troviamo una descrizione in [14]. Il suffix tree troncato mette insieme due importanti qualità nel campo del pattern matching: la struttura ad albero (per cui esistono molti efficienti algoritmi di attraversamento, ma che richiede molto spazio) e la ridotta occupazione di spazio di archiviazione (in dipendenza dal fattore di troncamento) tipica dei suffix array.

Una buona codifica per il suffix tree troncato è quella presentata in [3], derivata dalla codifica illi [1]. In [2] ne è stata proposta una versione modificata, senza suffix link, la codifica TruST. Questa è stata progettata per risolvere più efficientemente un problema specifico di bioinformatica, il profile matching, o ricerca di Position Weight Matrices, con l'algoritmo TruSTsearch, tuttavia questa codifica può essere utilizzata anche per altri scopi (in cui non servano i suffix link).

Questa tesi si propone di migliorare, in termini di spazio, la costruzione di un suffix tree troncato con codifica TruST. Infatti lo spazio occupato dal procedimento originariamente usato per la costruzione di un TST con codifica TruST, dovendo costruire prima il suffix tree troncato come in [3], dipende fortemente dal fattore di troncamento del suffix tree.

La tesi è articolata nel seguente modo. Nel Capitolo 1 saranno descritte le strutture dati basate sui suffissi, con i loro algoritmi di costruzione e le loro codifiche di implementazione. Nel Capitolo 2 verrà presentato il suffix tree troncato e, in particolare, la codifica TruST. Nel Capitolo 3 verranno esposte le modifiche alla costruzione di un suffix tree troncato con codifica TruST, utilizzando come base di partenza un suffix array, invece del suffix tree. Infine, nel Capitolo 4 e nelle Conclusioni, si discuteranno i risultati dei confronti sperimentali tra la soluzione proposta e quella precedente.

Capitolo 1

Suffix tree e suffix array

Questo capitolo sarà dedicato alla presentazione delle due principali strutture dati basate sull'indicizzazione dei suffissi di una stringa: *suffix tree* e *suffix array*.

I suffix tree nascono per l'elaborazione dei testi e in particolare nell'ambito del pattern matching. Le strutture dati basate sui suffissi, tuttavia, trovano applicazione in altri svariati ambiti, come la bioinformatica, la biologia computazionale, l'analisi di serie temporali e la compressione dati [4] [10]. La ricerca di strutture dati sempre più efficienti per l'elaborazione di grandi moli di dati è un campo vasto e ancora aperto a nuove soluzioni. Lo stato dell'arte presenta, fra le migliori soluzioni per quanto riguarda l'indicizzazione di sequenze di caratteri, le strutture basate sui suffissi.

Verrà descritta innanzitutto la struttura che storicamente per prima è stata usata: il suffix tree. Di questa verrà descritto uno dei principali algoritmi lineari di costruzione, l'algoritmo di Ukkonen [8], e la codifica illi [1], attualmente la più efficiente per il suffix tree.

In seguito si presenterà il suffix array e la sua versione estesa: l'*enhanced suffix array* (ESA), con analogie e collegamenti con il suffix tree. Per questa struttura dati verranno descritti due algoritmi di costruzione lineari nella taglia dell'input.

1.1 Suffix tree

La struttura dati suffix tree è stata proposta esattamente quarant'anni fa da Weiner [6]. Di seguito verranno introdotti concetti e algoritmi fondamentali che la riguardano.

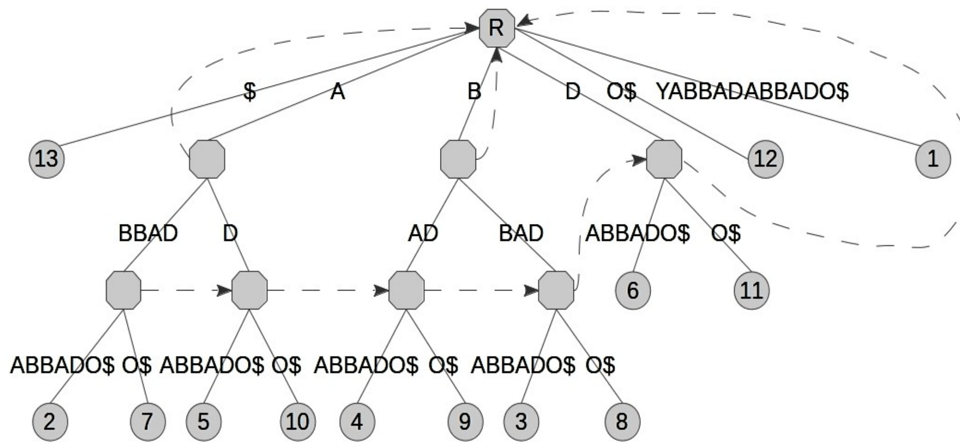

Figura 1.1: suffix tree della stringa YABBADABBADO$. I suffix link sono disegnati in linea tratteggiata

Definizione: *Data una sequenza S di lunghezza n definita su un alfabeto Σ, il suffix tree T di S è un albero con le seguenti proprietà :*

- *ci sono n foglie etichettate da 1 a n;*
- *ogni nodo interno ha più di un figlio;*
- *ogni arco è etichettato con una sottostringa di S;*
- *due archi che escono da un nodo non possono avere un'etichetta che inizia con lo stesso carattere;*
- *la concatenazione delle etichette dalla radice alla foglia i è l'i-esimo suffisso di S.*

Per garantire che ad ogni suffisso della sequenza corrisponda una foglia, si costruisce il suffix tree di S$, invece che di S, dove $ è un simbolo contenuto in Σ, ma non presente in S. $ è detto simbolo terminale. Tale accorgimento garantisce l'esistenza nell'albero di n foglie e al più n-1 nodi interni (Figura 1.1).

È tuttavia necessario un ulteriore accorgimento per rendere l'occupazione di spazio lineare: per etichettare un arco non si memorizza la sottosequenza, ma, piuttosto, lo si identifica con la coppia *(i,j)*, con i e j posizioni di inizio e fine della sottostringa corrispondente al ramo. Questa tecnica di memorizzazione evita che lo spazio occupato sia quadratico.

Viene data, ora, la definizione di suffix link necessaria nell'algoritmo di costruzione di Ukkonen e nella codifica illi.

Definizione: *Sia $x\alpha$ una stringa arbitraria, dove x denota un singolo carattere ed α denota una stringa (anche nulla). Si consideri un qualsiasi nodo interno v la cui concatenazione delle etichette dalla radice al nodo stesso sia $x\alpha$. Se esiste un altro nodo s(v) la cui concatenazione delle etichette dalla radice al nodo stesso è α, allora un puntatore da v a s(v) è chiamato suffix link.*

In Figura 1.1 si può osservare un suffix tree con i relativi suffix link.

1.1.1 Algoritmo di Ukkonen

Il suffix tree, oltre ad un'occupazione spaziale lineare, offre alcuni algoritmi di costruzione che possono essere eseguiti in $O(n)$. Fra questi quelli di Weiner [6], McCreight [7] e Ukkonen [8].

L'algoritmo di Ukkonen è uno dei più presenti in letteratura. Questo algoritmo costruisce ricorsivamente dei suffix tree sempre più grandi, aggiungendo i prefissi di S, e ne trasforma poi l'ultimo nel suffix tree di S$.

Il procedimento elabora la stringa da sinistra a destra, partendo dal primo carattere e aggiungendone uno per passo.

L'algoritmo si divide in n fasi, ognuna compie un certo numero di estensioni. Le fasi corrispondono alla creazione del suffix tree per il prefisso di lunghezza pari alla fase. Le estensioni sono l'inserimento di tutti i suffissi del prefisso nell'albero.

Ogni fase *i+1* si compone di *i+1* estensioni, una per ognuno dei suffissi del prefisso *i+1-esimo*. Ogni estensione j trova l'arco etichettato con *(j..i)* e lo trasforma in *(j..i+1)*. Quindi in ogni fase viene aggiunto un simbolo a ciascun arco che termina in una foglia.

Questo procedimento risulta essere di complessità cubica. Per evitare che questo accada, si utilizzano i suffix link nella ricerca dell'arco etichettato con *(j..i)*.

Altri accorgimenti, oltre ai suffix link, sono usati per rendere la costruzione lineare temporalmente nel caso peggiore.

1.1.2 Codifica illi

La codifica illi (improved linked list implementation) [1] è, attualmente, la codifica più efficiente per il suffix tree. Questa memorizza foglie e nodi dell'albero in due diverse tabelle.

I nodi si suddividono in *large node* e *small node* in questo modo: se viene costruita una serie di nodi $N_1, ..., N_q$, questa può essere suddivisa in catene.

Una catena di nodi è una sottosequenza di nodi adiacenti $N_l, ..., N_r,\ r \geq l$ con queste proprietà:

- N_{l-1} non è uno small node;
- $N_l, ..., N_{r-1}$ sono small node;
- N_r è un large node.

La codifica di un large node N contiene le seguenti informazioni:

- il fratello destro di N, se esiste;
- il primo figlio di N;
- la lunghezza del percorso dalla radice a N, chiamata *path(N)*;
- la posizione iniziale della prima occorrenza del percorso sopracitato nella sequenza che ha richiesto la creazione del nodo. Tale posizione è chiamata *head(N)*;
- il suffix link di N.

Per le proprietà del suffix tree, è possibile calcolare *head(N)* e la profondità di uno small node a partire dai dati del large node ad esso associato e dalla distanza fra i due nodi. Questo permette di memorizzare alcune informazioni solo nei large node, risparmiando spazio.

La codifica di uno small node contiene, infatti, solo le seguenti informazioni:

- il fratello destro di N, se esiste;
- il primo figlio di N;
- la distanza dal large node associato.

Ogni foglia F è, invece, codificata con la memorizzazione del suo fratello destro, infatti *head(F)* e *path(F)* si possono ricavare, facilmente, dal numero della foglia.

Lo spazio necessario per la memorizzazione di un large node è di 16 byte, per uno small node di 8 byte, mentre per una foglia è di 4 byte. Questo riduce lo spazio occupato dal suffix tree, rispetto all'algoritmo di McCreight, di 4 byte per i large node e di 12 byte per foglie e small node.

1.2 Suffix array

In questa sezione verrà descritto il *suffix array* e la sua versione estesa, l'*enhanced suffix array* (ESA). Nella descrizione saranno presentati degli algoritmi di costruzione lineare del suffix array.

Definizione: *Data una sequenza S di lunghezza n definita su un alfabeto Σ, il suffix array SA per S è un array di tutti i suffissi di S ordinati lessicograficamente. Ogni suffisso è rappresentato dalla sua posizione iniziale: $SA[i] = j$ se $Suff_j$ è l'$i-$esimo suffisso nell'ordine lessicografico di S.*

Nella Tabella 1.1 si può osservare un esempio di suffix array (SA) con il relativo longest common prefix array (LCP) che adesso verrà definito.

	1	2	3	4	5	6	7	8	9	10	11	12	13
SA	13	2	7	5	10	4	9	3	8	6	11	12	1
LCP	0	4	1	2	0	3	1	4	0	1	0	0	/

Tabella 1.1: suffix array (SA) e longest common prefix (LCP) della stringa YABBADABBADO$.

Il longest common prefix array è una delle principali tabelle che arricchiscono l'informazione del suffix array.

Definizione: *Dato un suffix array SA della stringa S di lunghezza n, l'i-esima posizione del longest common prefix (LCP) contiene la lunghezza del più lungo prefisso fra l'i-esimo e il j-esimo suffisso di SA.*
Si definisce $lcp(\alpha, \beta)$ come il più lungo prefisso comune fra α e β. Quindi $LCP[i] = lcp(S[SA[i]..n], S[SA[i+1]..n])$

Un algoritmo di costruzione del LCP verrà presentato nel Capitolo 3 nella spiegazione della costruzione del suffix tree troncato da ESA.

L'implementazione della tabella SA richiede $4n$ byte, mentre per la tabella LCP sono necessari n byte.

1.2.1 Enhanced suffix array

Per essere uno strumento equivalente al suffix tree, il suffix array ha, però, bisogno di essere arricchito con alcune altre tabelle contenenti informazioni

sulla struttura della sequenza. La struttura composta da SA e dalle tabelle aggiuntive è chiamata *enhanced suffix array*.

Ogni applicazione necessita di informazioni diverse, quindi non esiste l'ESA, ma si può parlare di un ESA specifico per ogni algoritmo da implementare. Alcune informazioni che corredano l'ESA sono i suffix link, gli lcp-interval e le relazioni genitore-figlio.

La tabella *link* contiene le informazioni relative ai suffix link, necessita di 2 byte per simbolo.

Di seguito si descriveranno gli *lcp-interval* che, insieme alla tabella *child*, la quale contiene le relazioni genitore-figlio, rendono possibile lavorare sul suffix array come su un suffix tree. Per memorizzare *child* serve 1 byte per simbolo.

Definizione: *Un intervallo* $[i..j]$, $0 \leq i < j \leq n$, *è un lcp-interval di lcp-valore* l *se*

- $LCP[i] < l$;
- $LCP[k] \geq l$ *per ogni* k *con* $i + 1 \leq k \leq j$;
- $LCP[k] = l$ *per almeno un* k *con* $i + 1 \leq k \leq j$;
- $LCP[j + 1] < l$.

I principali algoritmi che vengono trasposti da suffix tree a suffix array sono gli attraversamenti. Per ognuno di questi, oltre al suffix array di base, sono necessarie altre tabelle particolari:

- attraversamento bottom-up: LCP;
- attraversamento top-down: LCP e child;
- attraversamento depth-first: LCP e child;
- attraversamento con i suffix link: link.

In Tabella 1.2 si possono osservare alcune delle informazioni necessarie per altri algoritmi presenti in letteratura.

Applicazione	Enhanced suffix array						
	SA	LCP	child	link	S	bwt	skp
Supermaximal repeats	√	√				√	
Maximal unique matches	√	√				√	
Maximal repeated pairs	√	√				√	
Ziv-Lempel decomposition	√	√					
Pattern searching	√	√	√		√		
Shortest unique substrings	√	√	√				
Matching statistics	√	√	√	√	√		
Profile matching	√	√					√

Tabella 1.2: informazioni aggiuntive necessarie per l'esecuzione di diverse classi di algoritmi [4] [2] [10].

1.2.2 Algoritmi di costruzione

In questa sezione verranno presentati due algoritmi lineari di costruzione del suffix array. In particolare verranno descritti lo *skew algorithm* [11] e uno dei due algoritmi per *pure induced sorting* presentati in [13]. Lo skew algorithm è tra i primi algoritmi lineari per il calcolo di un suffix array. Gli algoritmi per pure induced sorting hanno una struttura simile allo skew algorithm, ma offrono una migliore efficienza spaziale, per questo uno dei due è usato nel procedimento proposto in questa tesi.

Skew algorithm

Uno dei primi algoritmi lineari per la costruzione di suffix array è lo skew algorithm presentato in [11]. Questo procedimento è una modifica dell'originale skew algorithm per la costruzione del suffix tree. L'algoritmo è basato sul tipico approccio *divide et impera*.

Per meglio comprendere lo skew algorithm per il suffix array verrà presentato prima quello per i suffix tree:

1. costruire il suffix tree dei suffissi che iniziano in posizione dispari. Questo si fa attraverso la costruzione di un suffix tree di una stringa lunga la metà, che viene elaborata ricorsivamente;

2. costruire il suffix tree dei rimanenti suffissi, usando il risultato del primo passo;

3. unire i due suffix tree in uno.

Questo procedimento non è ripetibile interamente per un suffix array, perchè l'unione di array ordinati richiede alcuni accorgimenti. Questi sono i passi dello skew algorithm per la costruzione del suffix array:

1. costruire il suffix array dei suffissi che iniziano nelle posizioni $i \ mod3 \neq 0$. Questo si fa attraverso la costruzione di un suffix array di una stringa lunga due terzi, che viene elaborata ricorsivamente;

2. costruire il suffix array dei rimanenti suffissi, usando il risultato del primo passo;

3. unire i due suffix array in uno.

Sorprendentemente, l'uso di due terzi invece della metà dei suffissi, nel primo passo, rende l'ultimo passo semplice: è sufficiente un'unione basata sul confronto. Per comparare due suffissi che iniziano a i e j con $i \ mod3 = 0$ e $j \ mod3 = 1$, prima si confrontano i caratteri iniziali, e, se sono uguali, vengono confrontati i suffissi che iniziano a *i+1* e *j+1* dei quali l'ordine è già noto dal primo passo.

Costruzione con almost pure induced sorting

L'algoritmo presentato in [13], come lo skew algorithm, elabora ricorsivamente la stringa.

Il procedimento si divide in:

- riduzione del problema;
- risoluzione ricorsiva del problema;
- soluzione finale per induzione.

Riduzione del problema: sia S$ una stringa di dimensione n, SA il suo suffix array e $Suff_i$ il suffisso i-esimo di S. Un suffisso $Suff_i$ si dice di tipo S o L se $Suff_i < Suff_{i+1}$ o $Suff_i > Suff_{i+1}$ rispettivamente. Il suffisso $ è definito di tipo S. È possibile definire il tipo di ogni suffisso in tempo lineare.

Si definisce LMS-carattere un simbolo S_i se è di tipo S e S_{i-1} è di tipo L. Inoltre si dice LMS-sottostringa una sottostringa S*[i..j]*, per $i \neq j$, con S_i e S_j LMS-caratteri, e nessun altro LMS-carattere nella sottostringa. Il simbolo terminale $ è definito come LMS-sottostringa.

L'ordine di due LMS-sottostringhe, a parità di ordine lessicografico, è dato dalla precedenza del tipo S sul tipo L.

Si supponga di avere tutte le LMS-sottostringhe ordinate in dei bucket, si associ ad ogni elemento dell'array P_1 di puntatori ai suffissi il numero del suo bucket, ottenendo la stringa R_1.

Si può dimostrare che:

1. $|R_1| \leq \frac{|S|}{2}$;

2. l'ultimo carattere di R_1 è unico ed è il più piccolo in R_1;

3. per $R_1[i] = R_1[j]$, deve essere $P_1[i+1] - P_1[i] = P_1[j+1] - P_1[j]$;

4. l'ordine relativo di due qualsiasi suffissi, i e j, in R_1 rimane lo stesso di $Suff_{P_1[i]}$ e $Suff_{P_1[j]}$.

Da questo si può dedurre che per ordinare tutti gli LMS-suffissi di S, è possibile ordinare R_1. La complessità si riduce, così, di almeno la metà.

Induzione: si supponga di aver calcolato linearmente il suffix array SA_1 di R_1 e si voglia calcolare SA di S. Si osserva banalmente che in SA tutti i suffissi che iniziano con lo stesso carattere appaiono consecutivamente. Ci si riferisca al subarray dei suffissi che iniziano con lo stesso carattere come ad un bucket. Nel bucket, i suffissi di tipo S sono divisi da quelli di tipo L. Quindi il bucket può essere diviso in due parti L ed S.

Il procedimento di induzione viene effettuato in tre passi:

1. trovare la fine di ogni bucket S, inserire tutti gli elementi di SA_1 nei corrispondenti bucket in SA in ordine inalterato;

2. ordinare ogni bucket L;

3. trovare, in SA, la fine di ogni bucket S e inserire gli elementi del bucket L ordinato.

Ognuno dei passi descritti può essere compiuto in tempo lineare e si può dimostrare che l'algoritmo restituisce il suffix array della stringa in esame.

Capitolo 2

Suffix tree troncati e TruST

In questo capitolo si descriverà il *suffix tree troncato* [14]. Di questa struttura dati verrà presentata, in particolare, la codifica TruST [2].

Il suffix tree troncato ha due caratteristiche fondamentali per l'efficienza nell'elaborazione di sequenze di grandi dimensioni: la duttilità della struttura ad albero e l'efficiente occupazione spaziale tipica dei suffix array.

2.1 Suffix tree troncato

Fra le applicazioni che utilizzano il suffix tree, molte non necessitano della memorizzazione dei suffissi di tutta la stringa. Infatti molti algoritmi per la soluzione di problemi di pattern matching visitano solo alcune parti dell'albero, limitandosi a sottostringhe di lunghezza limitata. Per questo motivo si può pensare di memorizzare una versione ridotta del suffix tree: il suffix tree troncato.

Il *suffix tree troncato* [14], o *k-factor tree* [3], di fattore di troncamento k, come evidenziato dal nome, è un suffix tree che memorizza solo i primi k caratteri di ogni suffisso, tagliando il resto dell'albero.

A partire dalla definizione di suffix tree, viene così definita la sua versione troncata:

- se il suffisso ha lunghezza inferiore o uguale a k, tale suffisso apparirà interamente anche nel suffix tree troncato;
- se il suffisso ha lunghezza superiore a k, nel suffix tree troncato ne verranno memorizzati solo i primi k simboli.

Il suffix tree troncato riesce così ad evitare la memorizzazione di alcuni nodi, rendendolo più efficiente in termini spaziali rispetto al suffix tree intero.

In Figura 2.1 e 2.2, si possono osservare i suffix tree troncati per due diversi valori di k, i k-factor tree, per la stessa sequenza di Figura 1.1.

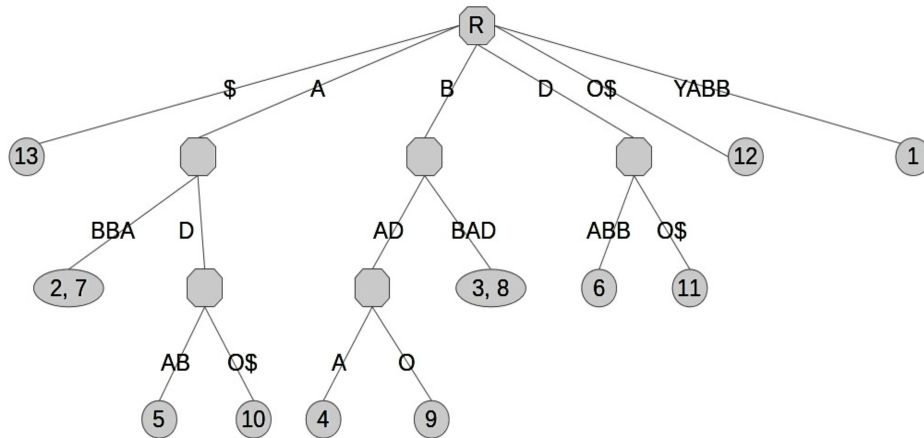

Figura 2.1: 4-suffix tree per la stringa YABBADABBADO$; le foglie 2 e 7 condividono i primi 4 simboli (ABBA) e quindi c'è una linked list che contiene queste foglie, come per 3 e 8 (BBAD)

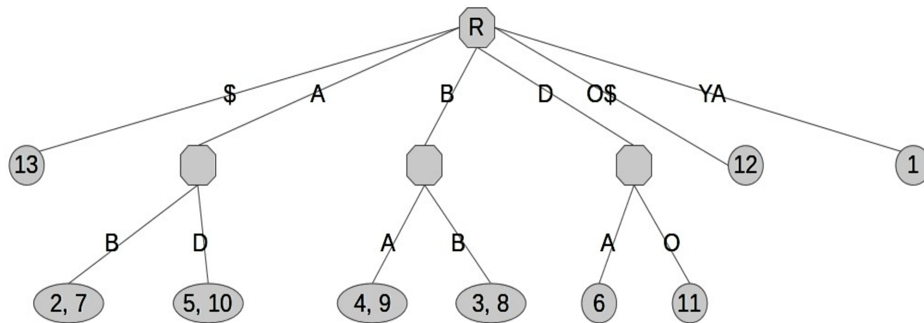

Figura 2.2: 2-suffix tree per la stringa YABBADABBADO$; le foglie 2 e 7 condividono i primi 2 simboli (AB) e quindi c'è una linked list che contiene queste foglie, come per 5 e 10 (AD), 4 e 9 (BA) e per 3 e 8 (BB)

2.1.1 Algoritmi di costruzione

Il modo più semplice per costruire un suffix tree troncato è quello di costruire l'intero suffix tree, per poi tagliarlo alla profondità del fattore di troncamento. Questo procedimento non dà, ovviamente, un miglioramento nell'occupazione spaziale. Algoritmi più efficienti sono derivati da una modifica dell'algoritmo di Ukkonen o di quello di McCreight [14].

La costruzione modificata rispetto a quella di Ukkonen si compone di due parti:

1. Costruire il suffix tree di $S_{i..k-1}$;
2. Aggiungere all'albero tutti i suffissi di $S_{i-k+1..i}$ per $k \leq i \leq n$.

La prima parte segue l'algoritmo di Ukkonen fino ad arrivare al $k-1$-esimo suffisso di S, aggiungendo però le foglie create in $queue_{leaf}$.

La seconda parte è simile all'algoritmo di Ukkonen, sempre con la memorizzazione in $queue_{leaf}$ delle foglie create. La differenza rispetto al procedimento di Ukkonen è che alla fase i possono esserci due casi:

- c'è una foglia F alla fine di $queue_{leaf}$;
- $queue_{leaf}$ è vuota.

Nel primo caso si procede normalmente con la fase i dalla foglia F, che corrisponde all'ultima foglia creata.

Nel caso in cui $queue_{leaf}$ è vuota, non si procede come nell'algoritmo di Ukkonen dall'ultima foglia creata, ma dall'ultima posizione raggiunta nell'albero durante la fase precedente, posizione che può essere nel mezzo di un arco. Alla fine della fase i, si rimuove la foglia F in testa di $queue_{leaf}$.

Il procedimento ha una complessità temporale lineare, grazie alla modifica dell'algoritmo di Ukkonen nel caso in cui $queue_{leaf}$ è vuota [3].

2.1.2 Codifica derivata da illi

In questa sezione viene esposta una codifica [3] per il suffix tree troncato che è un adattamento della codifica illi [1].

Le modifiche principali sono nella codifica delle foglie. Queste infatti, nell'originale codifica illi, vengono memorizzate con il solo fratello destro, in quanto head e path sono facilmente ricavabili dal numero della foglia. Questo non è valido nel suffix tree troncato, dove, a meno di un fattore di troncamento superiore al path della foglia, il path della foglia è pari a k.

È quindi necessario memorizzare anche head e path nella codifica delle foglie. Per questo scopo le foglie sono inserite in linked list: le foglie del

sottoalbero con radice nel nodo di path pari a k, o nel nodo subito successivo se il path di lunghezza k termina in mezzo ad un arco, formano una linked list.

È necessario sapere quando si giunge alla fine di una linked list durante un attraversamento. Serve un modo efficiente per ottenere il fratello o il suffix link del genitore di ogni foglia.

Per risolvere questi problemi viene modificata la codifica dei nodi. Se il suffix link punta ad uno small node, il suffix link non viene memorizzato, ma si ricava dal nodo stesso. Se invece il suffix link punta ad un large node è necessario memorizzarlo. La memorizzazione del suffix link dipende dalle dimensioni di path e head del nodo in questione.

Per conoscere il suffix link o il fratello di una linked list, serve giungere all'ultima foglia della lista (master) in tempo costante. Per questo nelle foglie della lista vengono usati 4 bit per memorizzare il puntatore all'ultima foglia (bisogna attraversare 7 foglie per conoscere i 27 bit dell'indice della foglia).

Nella foglia master sono memorizzate anche alcune informazioni per ricavare facilmente il path di tutte le foglie della linked list.

Con questa codifica si ottiene un'occupazione di spazio che per i nodi rimane uguale a quella della codifica originale, mentre per ogni foglia, master e non, vengono occupati 4 byte.

Il miglioramento nell'occupazione spaziale è dato dal ridotto numero di nodi da memorizzare.

2.2 Codifica TruST

La codifica TruST [2] nasce per risolvere un problema di bioinformatica, ovvero l'individuazione di fattori di trascrizione rappresentati come matrici (profile matching o position weight matrices search).

TruSTsearch è un algoritmo per il profile matching basato sull'utilizzo di un suffix tree troncato. La scelta di questa struttura dati deriva da un'analisi delle prestazioni di suffix tree e suffix array. I secondi sono più efficienti, ma i primi memorizzano informazioni non utili allo scopo del profile matching. Spesso, infatti, è sufficiente avere informazioni su stringhe di lunghezza limitata e il k-factor tree offre la praticità della struttura ad albero con una ridotta occupazione spaziale.

La codifica derivata da illi presentata sopra funziona per ogni algoritmo che lavora su un suffix tree troncato, permettendo tutti i possibili attraversamenti dell'albero. Non tutte le applicazioni hanno tuttavia necessità di compiere ogni tipo di attraversamento, ad esempio TruSTsearch si basa solo su un tipo: l'attraversamento depth-first. Questo significa che

le informazioni memorizzate nella codifica illi modificata non sono tutte necessarie. L'eliminazione di alcune di queste porta ad un miglioramento dell'occupazione spaziale del suffix tree troncato. Queste considerazioni portano a elaborare una nuova codifica: la *codifica TruST*, che verrà ora esposta.

Considerazioni di base

Si supponga di avere una permutazione S dei primi N numeri naturali. Si immagini ora che ogni elemento di S abbia un puntatore all'elemento successivo. È possibile verificare che, per una permutazione S casuale, è conveniente conoscere la differenza fra un elemento e l'altro piuttosto che il valore effettivo dell'elemento.

Un altro importante aspetto da considerare sono le informazioni necessarie da memorizzare.
Per un nodo interno N:

- $head(N)$;
- $|path(N)|$;
- primo figlio di N;
- fratello destro di N (se presente).

Per una foglia F:

- $head(F)$;
- $|path(F)|$;
- fratello destro di F (se è una foglia alla fine di una linked list e se ha un fratello).

L'algoritmo non usa mai i suffix link, quindi non serve memorizzarli. Questo permette di liberare spazio rispetto all'originale codifica.

Lo spazio recuperato può essere usato per memorizzare altre informazioni. Inoltre alcuni dati possono essere immagazzinati in un numero di bit inferiore a quello della codifica originale. Infatti, definendo diversi casi a seconda della grandezza del numero da memorizzare, si possono usare meno bit nella codifica. Si ricorda inoltre la memorizzazione delle differenze fra i valori invece dei valori stessi.

Codifica delle foglie

La codifica TruST distingue fra tre tipi di foglie:

- master leaf: la foglia più a destra in una linked list;
- small leaf: le foglie intermedie in una linked list;
- large leaf: la prima foglia in una linked list.

Si consideri una linked list con almeno tre foglie. La foglia più a destra è una master leaf e punta al fratello destro, le altre foglie puntano al proprio successore (memorizzando la differenza da questo).

Per giungere in tempo costante alla master leaf, se ne memorizza il valore nella large leaf, oltre a quello del successore di quest'ultima.

Nel caso in cui nella linked list ci sia solo una foglia questa è una master leaf. In una linked list con due foglie, la seconda è una master leaf mentre la prima è una small leaf (il suo successore è la master leaf associata e non serve quindi memorizzarne il valore come in una large leaf).

Si usano alcuni bit di controllo per distinguere i vari casi.

Codifica di una master leaf:

Si definiscano:

- master_leaf = numero della foglia;
- sib = numero del fratello destro;
- diff = master_leaf - sib.

Il secondo bit di una master leaf si usa per discriminare se $|\text{diff}| < 2^{19}$ o no. Sono necessari altri bit per differenziare i casi in cui diff < 0 o diff > 0 e se il fratello è un nodo o una foglia.

Riassumendo i casi possibili:

Caso	Ha fratello	|diff|	Spazio
1	NO		1 byte
2	SÌ	$< 2^{19}$	3 byte
3	SÌ	$\geq 2^{19}$	4 byte

Codifica di una small leaf:

Si definiscano:

- leaf = numero della foglia;
- next_leaf = numero del successore;
- diff = leaf - next_leaf.

Serve un bit per discriminare se $|\text{diff}| < 2^{21}$ o meno. Essendo le foglie di una linked list in ordine decrescente non serve un bit per decidere se diff > 0.

Riassumendo i casi possibili:

Caso	|diff|	Spazio
1	$< 2^{21}$	3 byte
2	$\geq 2^{21}$	4 byte

Codifica di una large leaf:

Si definiscano:

- large_leaf = numero della foglia;
- next_leaf = numero del successore;
- master_leaf = master leaf della linked list;
- diff_leaf = large_leaf - next_leaf;
- diff_master = large_leaf - master_leaf.

Quattro bit servono per discriminare la dimensione e il segno di diff_master e di diff_leaf.

Riassumendo i casi possibili:

Caso	|diff_leaf|	|diff_master|	Spazio
1	$< 2^{21}$	$< 2^{23}$	6 byte
2	$< 2^{21}$	$\geq 2^{23}$	7 byte
3	$\geq 2^{21}$	$< 2^{25}$	7 byte
4	$\geq 2^{21}$	$\geq 2^{25}$	8 byte

Codifica dei nodi

La codifica dei nodi è fortemente modificata rispetto a quella della codifica originale. Le principali differenze sono:

- non memorizza i suffix link;
- libera bit inutili quando il nodo non ha fratello destro;
- memorizza le differenze fra i valori invece dei valori stessi.

Codifica di uno small node:

Si definiscano:

- node = numero del nodo;
- sib = numero del fratello;
- fc = numero del primo figlio;
- diff_sib = node - sib;
- diff_fc = node - fc.

Si usano solo 3 bit per la distanza invece che 5. Questo non è un problema perchè si inserisce ogni 7 nodi un large node, raggiungendo così in tempo costante le informazioni ricercate.

Riassumendo i casi possibili:

Caso	Ha fratello	|diff_sib|		|diff_fc|	Spazio
1	NO			$< 2^{24}$	4 byte
2	NO			$\geq 2^{24}$	5 byte
3	YES	$< 2^{23}$	e	$< 2^{23}$	7 byte
4	TES	$\geq 2^{23}$	o	$\geq 2^{23}$	8 byte

Codifica di un large node:

Si definiscano:

- node = numero del nodo;
- sib = numero del fratello;
- fc = numero del primo figlio;

- head = posizione di head;
- |path| = lunghezza del percorso dalla radice al nodo;
- diff_sib = node - sib;
- diff_fc = node - fc;
- diff_head = node - head.

La codifica di un large node usa come soglia per |diff_sib|, |diff_fc| e |diff_head|, 2^{23}, mentre per |path|, $2^7 = 128$, valore ragionevole per il pattern matching, dove il fattore di troncamento è attorno a 30. Alcuni bit sono necessari per discriminare il segno delle differenze e il tipo di fratello e primo figlio.

Riassumendo i casi possibili:

Caso	Ha fratello	|diff_sib|	|diff_fc|	|diff_head|	|path|	Spazio
1	NO		$< 2^{23}$	$< 2^{23}$	$< 2^{10}$	8 byte
2	NO		$< 2^{23}$	$< 2^{23}$	$\geq 2^{10}$	11 byte
3	NO		$< 2^{23}$	$\geq 2^{23}$	$< 2^{7}$	8 byte
4	NO		$< 2^{23}$	$\geq 2^{23}$	$\geq 2^{7}$	11 byte
5	NO		$\geq 2^{23}$	$< 2^{23}$	$< 2^{14}$	9 byte
6	NO		$\geq 2^{23}$	$< 2^{23}$	$\geq 2^{14}$	11 byte
7	NO		$\geq 2^{23}$	$\geq 2^{23}$	$< 2^{11}$	9 byte
8	NO		$\geq 2^{23}$	$\geq 2^{23}$	$\geq 2^{11}$	11 byte
9	YES	$< 2^{23}$	$< 2^{23}$	$< 2^{23}$	$< 2^{8}$	11 byte
10	TES	$< 2^{23}$	$< 2^{23}$	$< 2^{23}$	$\geq 2^{8}$	14 byte
11	YES	$< 2^{23}$	$< 2^{23}$	$\geq 2^{23}$	$< 2^{13}$	12 byte
12	TES	$< 2^{23}$	$< 2^{23}$	$\geq 2^{23}$	$\geq 2^{13}$	14 byte
13	YES	$< 2^{23}$	$\geq 2^{23}$	$< 2^{23}$	$< 2^{12}$	12 byte
14	TES	$< 2^{23}$	$\geq 2^{23}$	$< 2^{23}$	$\geq 2^{12}$	14 byte
15	YES	$< 2^{23}$	$\geq 2^{23}$	$\geq 2^{23}$	$< 2^{9}$	12 byte
16	TES	$< 2^{23}$	$\geq 2^{23}$	$\geq 2^{23}$	$\geq 2^{9}$	15 byte
17	YES	$\geq 2^{23}$	$< 2^{23}$	$< 2^{23}$	$< 2^{12}$	12 byte
18	TES	$\geq 2^{23}$	$< 2^{23}$	$< 2^{23}$	$\geq 2^{12}$	14 byte
19	YES	$\geq 2^{23}$	$< 2^{23}$	$\geq 2^{23}$	$< 2^{9}$	12 byte
20	TES	$\geq 2^{23}$	$< 2^{23}$	$\geq 2^{23}$	$\geq 2^{9}$	15 byte
21	YES	$\geq 2^{23}$	$\geq 2^{23}$	$< 2^{23}$	$< 2^{8}$	12 byte
22	TES	$\geq 2^{23}$	$\geq 2^{23}$	$< 2^{23}$	$\geq 2^{8}$	15 byte
23	YES	$\geq 2^{23}$	$\geq 2^{23}$	$\geq 2^{23}$	$< 2^{13}$	13 byte
24	TES	$\geq 2^{23}$	$\geq 2^{23}$	$\geq 2^{23}$	$\geq 2^{13}$	15 byte

Usando la codifica TruST, invece di quella derivata da illi, c'è un guadagno in termini di occupazione spaziale [9], particolarmente nella memorizzazione dei nodi interni, sia per i large che per gli small. Tuttavia, pur se in minor quantità, anche le foglie contribuiscono al miglioramento dell'efficienza spaziale.

La codifica TruST non può, però, essere utilizzata se sono necessari i suffix link. In ogni caso permette altri tipi di attraversamento tipici di algoritmi che lavorano su suffix tree troncati, oltre all'attraversamento depth-first usato in TruSTsearch.

Capitolo 3

Codifica TruST da ESA

In questo capitolo verrà presentato il contributo principale della tesi, ovvero un nuovo algoritmo di costruzione per suffix tree troncato con codifica TruST.

Nel tool originale viene utilizzata una costruzione a partire da un suffix tree troncato (TST) con codifica illi (di seguito illi-TST) adattata a k-factor tree, ottenuta tramite un adattamento dell'algoritmo di Ukkonen per suffix tree. Del suffix tree troncato costruito vengono memorizzate nel file di output solo le informazioni necessarie alla codifica TruST. Tuttavia, nella prima parte di questo procedimento vengono salvati temporaneamente alcuni dati che risultano poi superflui per ottenere la codifica TruST come, ad esempio, i suffix link (necessari per garantire la linearità temporale dell'algoritmo di Ukkonen).

La costruzione descritta in questa tesi si propone di ridurre l'occupazione spaziale del primo passo attraverso la costruzione del suffix tree troncato a partire da un ESA associato alla sequenza in esame.

L'ESA che viene utilizzato è composto da suffix array (SA) e longest common prefix (LCP), dai quali vengono costruiti gli *lcp-interval*, che non vengono memorizzati in una tabella, bensì direttamente trasformati in nodi o foglie con codifica TruST.

3.1 Costruzione del k-factor tree da ESA

La costruzione del suffix tree troncato con codifica TruST viene fatta a partire da un ESA che comprende SA e LCP. Entrambe queste tabelle vengono costruite in tempo e spazio lineari nella taglia dell'input.

3.1.1 Suffix Array

La costruzione del SA si basa sull'algoritmo per *pure induced sorting* [13] presentato nel Capitolo 1. Questo algoritmo, presa in esame la stringa, la riduce in sottostringhe sempre più piccole di cui poi, ricorsivamente, calcola il suffix array.

È stato scelto questo algoritmo per la sua struttura ricorsiva e semplice implementazione. Inoltre questo algoritmo è caratterizzato da linearità temporale e da una buona efficienza spaziale sia nella memorizzazione finale del suffix array, sia nella sua costruzione, proprietà fondamentali nell'elaborazione di lunghe sequenze.

3.1.2 Longest Common Prefix

La costruzione dell'array dei longest common prefix (LCP) è fatta con l'algoritmo presentato in [12]. Questo algoritmo si basa su alcune proprietà dei *lcp*.

Il *lcp* fra due suffissi qualsiasi è il minimo dei *lcp* di tutte le coppie di suffissi adiacenti compresi fra le posizioni che i due suffissi occupano nel SA:

$$lcp(S_{SA[x]}, S_{SA[z]}) = min\{lcp(S_{SA[y-1]}, S_{SA[y]})\},\ x < y \leq z.$$

Questo implica che il *lcp* di una coppia di suffissi è maggiore o uguale dei *lcp* fra i suffissi che formano un intervallo che contiene la coppia.

$$lcp(S_{SA[y-1]}, S_{SA[y]}) \geq lcp(S_{SA[x]}, S_{SA[z]}),\ x < y \leq z.$$

Quando il *lcp* fra un paio di suffissi adiacenti in SA è maggiore di 1, l'ordine lessicografico dei suffissi è preservato se il primo carattere di ogni suffisso viene cancellato, e il *lcp* fra i due nuovi suffissi è uguale al precedente diminuito di 1.

Per riempire il LCP, si voglia calcolare il *lcp* fra un suffisso S_i e il suo successivo nel SA, noto quello fra S_{i-1} e S_i.

Sia *rank* un array che rappresenta la funzione inversa di SA, così ottenuta: se $SA[k] = i$ allora $rank[i] = k$.

Si definiscano: $p = rank[i-1]$ e $q = rank[i]$, inoltre $j - 1 = SA[p-1]$ e $k = SA[q-1]$. Dato $LCP[p]$ si vuole calcolare il valore di $LCP[q]$.

Si può provare che se $lcp(S_{j-1}, S_{i-1}) > 1$ allora $lcp(S_k, S_i) \geq lcp(S_j, S_i)$. Inoltre se $LCP[p] = lcp(S_{j-1}, S_{i-1}) > 1$ allora $LCP[q] = lcp(S_k, S_i) \geq LCP[p] - 1$.

Lo pseudocodice dell'algoritmo:

Dati una sequenza S e il suo SA

  $for(i = 1\ to\ n)\ rank[SA[i]] = i$

  $h = 0$

  $for(i = 1\ to\ n)$

    $if(rank[i]) > 1)$

      $k = SA[rank[i] - 1]$

      $while\ (S[i + h] = S[k + h])$

        $h = h + 1$

      $LCP[rank[i]] = h$

      $if(h > 0)\ h = h - 1$

3.1.3 Creazione di nodi e foglie come *lcp-interval*

Per ricavare la struttura del TST a partire da SA e LCP si utilizza il concetto di *lcp-interval* [10]. Questi infatti rappresentano un mapping diretto fra il suffix tree e il suffix array e permettono di lavorare sul secondo come se fosse un albero. Questa corrispondenza viene data dal fatto che ogni lcp-interval è assimilabile ad un nodo o ad una foglia, se il lcp-interval ha lunghezza pari a 1.

Nell'algoritmo presentato in [10] viene attraversato LCP per trovare gli lcp-interval (come descritti nel Capitolo 1), questi vengono memorizzati in uno stack, per poi, se necessario per l'implementazione, essere elaborati. Rispetto a questo procedimento sono state apportate alcune modifiche per ottenere la struttura del suffix tree troncato con codifica TruST:

- l'elaborazione prevede che ogni lcp-interval creato venga processato ricorsivamente per trovarne i sottointervalli;
- il caso base è il lcp-interval con inizio e fine coincidenti;
- ogni lcp-interval con inizio e fine diversi è codificato come large o small node a seconda di quando viene creato (ogni 7 small node si inserisce un large node);
- gli lcp-interval corrispondenti al caso base sono codificati come foglie (master, large o small leaf a seconda della posizione nella linked list);
- la ricorsione sugli lcp-interval viene interrotta se il valore l del *lcp-intervallo* è maggiore del fattore di troncamento k;

- se la ricorsione viene interrotta per $l \geq k$ ogni suffisso del lcp-interval in esame è una foglia da inserire in una linked list;
- ad ogni sottointervallo viene assegnato un livello corrispondente al numero dei suoi avi nell'albero degli lcp-interval;
- nella costruzione degli lcp-interval vengono definiti i fratelli e i primi figli di ogni lcp-interval/nodo.

Si riporta di seguito lo pseudocodice dell'algoritmo per la costruzione, attraverso gli lcp-interval, del suffix tree troncato con codifica TruST:

```
Dati una sequenza S, il suo SA e il suo LCP
function lcpinterval(l,start,end,level)
  l è il valore dell'lcp da cercare
  start è l'inizio del lcp-interval
  end è la fine del lcp-interval
  level è il livello del lcp-interval
  {
  if(start = end)
      {
      leaves[leaf] = master leaf
      if(level_of_lastins = level) then
          leaf è fratello destro/successore di lastins
      else if(level_of_lastins > level) then
          leaf è fratello destro di last_of_level[level]
      else if(level_of_lastins = level − 1) then
          leaf è primo figlio di lastins
      lastins = last_of_level[level] = leaf
      leaf = leaf + 1
      }
  if(start < end)
      {
      if(LCP[start] < l)
          lcpinterval(l, start, start, level)
          lcpinterval(l, start + 1, end, level)
      else {
          if(l ≥ k)
              while(start + lk ≤ end)
                  lcpinterval(k, start + lk, start + lk, level)
                  lk = lk + 1
              definire i tipi delle foglie nella linked list
```

```
        else
            i = start
            while(LCP[i] ≥ l)
                if(LCP[i] = l) write = yes
                i = i + 1
            if(write = yes)
                nodes[node] = small node
                    if(level_of_lastins = level) then
                        node è fratello destro di lastins
                    else if(level_of_lastins > level) then
                        node è fratello destro di last_of_level[level]
                    else if(level_of_lastins = level − 1) then
                        node è primo figlio di lastins
                small_nodes_created = small_nodes_created + 1
                if(small_nodes_created = 7)
                    trasformare in large node
                lastins = last_of_level[level] = node
                node = node + 1
                lcpinterval(l + 1, start, i, level + 1)
                if(i < end) lcpinterval(l, i + 1, end, level)

            else lcpinterval(l + 1, start, end, level)
        }
    }
}
```

Come visto nel Capitolo 2, i nodi interni e le foglie nella codifica TruST sono codificati con alcune informazioni. I nodi memorizzano head, path, primo figlio e fratello. Le foglie vengono, invece, codificate con head, path e fratello o successore nella linked list.

Per definire primo figlio e fratello di un lcp-interval, questo viene memorizzato in una variabile temporanea e in un array temporaneo. La prima è *lastins* e serve a tenere nota dell'ultimo lcp-interval inserito, di cui il successivo lcp-interval creato può essere primo figlio o fratello. Mentre l'array *last_of_level* memorizza nella posizione di indice *level* il lcp-interval appena creato, di cui il successivo lcp-interval dello stesso livello può essere fratello.

Ogni lcp-interval, quando è da memorizzare, viene salvato con la codifica TruST, con le stesse modalità usate nel tool originale. In particolare le foglie sono create tutte come master leaf, caratteristica tolta se si crea una linked list. Se la foglia è una master leaf, oltre all'eventuale fratello, viene

memorizzata anche la sua profondità, pari al minimo fra il fattore di troncamento e la lunghezza del suffisso a cui corrisponde. Se invece la foglia è small viene associato ad essa il suo successore nella linked list, mentre nel caso di large leaf si aggiunge, oltre al successore, un puntatore alla master leaf della linked list.

Se invece il lcp-interval corrisponde ad un nodo, questo viene memorizzato come small node, per poi, se necessario, trasformarlo in large node. Ad ogni nodo viene associato il primo figlio. Inoltre se è uno small node, si memorizza la sua distanza dal relativo large node, mentre se è un large node se ne salva la profondità, pari all'*lcp-valore* del lcp-interval, e il valore di head, pari a $SA[start]$, essendo questa la prima posizione per cui il nodo viene creato.

Capitolo 4

Analisi delle prestazioni

In questo capitolo vengono riportati e discussi i risultati dei test di confronto tra le costruzioni per suffix tree troncato con codifica TruST basate su Illi-TST ed ESA.

Il capitolo, dopo la presentazione delle sequenze di benchmark, proseguirà con un'analisi teorica dell'occupazione spaziale, per poi, infine, presentare i risultati sperimentali.

Descrizione delle sequenze di benchmark

Le sequenze scelte per i confronti, temporali e spaziali, fra le due costruzioni di suffix tree troncato con codifica TruST basate, rispettivamente, su illi-TST ed ESA, sono principalmente stringhe casuali (questo per verificare le prestazioni generali degli algoritmi e poterle confrontare con quelle su sequenze non casuali). A queste sono state aggiunte alcune sequenze tipiche di ambiti dove queste strutture dati trovano applicazione, come, ad esempio, la bioinformatica, l'analisi dei testi e la compressione.

Le stringhe casuali sono state generate utilizzando un web tool (http://www.dave-reed.com/Nifty/randSeq.html), che prende in input lunghezza (n) della stringa e simboli con cui comporla e, chiamando la funzione random di Java sui simboli possibili n volte, crea stringhe random. Le stringhe non casuali sono tratte dal Canterbury Corpus (http://corpus.canterbury.ac.nz/descriptions/), mentre le sequenze biologica dal sito del NCBI (http://www.ncbi.nlm.nih.gov/).

Vengono ora presentate le sequenze usate, raggruppate per tipologia, la loro lunghezza è riportata nelle tabelle di confronto caso per caso:

- alfanumerica random: stringa casuale di caratteri a, b, 0 e 1;

- numeri random: stringa casuale composta da tutte le cifre;
- alfabeto random: stringa casuale composta dalle prime venti lettere dell'alfabeto inglese;
- Taeniopygia 16: sequenza di nucleotidi del cromosoma 16 di Taeniopygia guttata;
- Canis 17: sequenza di nucleotidi del cromosoma 17 di Canis lupus familiaris;
- aaa: stringa di caratteri 'a' ripetuti per tutta la lunghezza;
- alfabeto ripetuto: stringa composta dalla ripetizione dell'alfabeto inglese per tutta la lunghezza;
- simboli random: sequenza casuale su alfabeto di cardinalità 64, comprendente lettere, numeri e simboli;
- alice in wonderland: testo del famoso libro di Lewis Carrol;
- pi greco: il primo milione di cifre del numero irrazionale π.

4.1 Analisi dell'occupazione spaziale

In questa sezione verrà analizzato lo spazio necessario alla costruzione del suffix tree troncato con codifica TruST, prima con il procedimento originariamente adottato, cioè attraverso la costruzione di Ukkonen adattata a TST con codifica illi, e poi con il procedimento basato su ESA.

Le considerazioni verranno fatte tralasciando lo spazio occupato dalla memorizzazione della sequenza, dai nodi e dalle foglie con codifica TruST, in quanto questo è uguale nei due procedimenti. Saranno anche tralasciati tutti gli elementi che hanno un'occupazione spaziale $O(1)$, ovvero le variabili e gli array di dimensioni dell'ordine delle decine di byte.

4.1.1 Algoritmo basato su illi-TST

Il procedimento originariamente utilizzato per la costruzione del suffix tree troncato con codifica TruST prevede l'iniziale costruzione di un TST con codifica illi [3].

La costruzione di illi-TST utilizza una variante per suffix tree troncati del noto algoritmo di Ukkonen [8] per la costruzione online in tempo lineare

di un suffix tree. Questa costruzione utilizza degli elementi che non vengono poi utilizzati per la costruzione di un TST con codifica TruST:

- suffix link di ogni nodo;
- suffix link delle foglie di tipo master;
- array delle informazioni relative ai tipi di codifica di nodi e foglie;
- array delle profondità delle foglie di tipo master leaf.

Ogni nodo del suffix tree ha un suffix link, e il numero di nodi creati è lineare rispetto alla lunghezza n della sequenza di input. Essendo n l'upper bound del numero di foglie dell'albero il numero di nodi interni è, nel caso peggiore, $n\text{-}1$, anche se, nel caso di un suffix tree troncato, è, solitamente, sublineare.

Oltre ai suffix link dei nodi bisogna, però, tenere conto dei suffix link delle foglie di tipo master leaf.

Il numero di foglie di tipo master leaf e quello dei nodi interni dipende fortemente dal fattore di troncamento k e dalla stringa in esame. Lo spazio necessario per ogni suffix link è di 4 byte. Vengono quindi occupati $4n \times (t(k) + f(k))$ byte per i suffix link, con $f(k)$ e $t(k)$ funzioni che identificano, rispettivamente, la dipendenza del numero di nodi interni e del numero di foglie di tipo master leaf dal fattore di troncamento. Sperimentalmente si è visto che $f(k)$ e $t(k)$ possono variare da valori nell'ordine dei millesimi (per sequenze ripetute) a 1 (nei casi peggiori).

Nella costruzione del suffix tree troncato vengono definiti anche gli array leaves_color, inodes_color, nb_leaves_color e nb_inodes_color. Questi contengono informazioni sul tipo di codifica di foglie e nodi a cui sono associati, per differenziare, dopo la divisione in large, small e master, la modalità di memorizzazione a seconda dei valori di profondità, differenze e tipo di fratelli e figli (come dettagliato nel Capitolo 2). Gli array riferiti alle foglie hanno lunghezza proporzionale al numero delle foglie di tipo master leaf, mentre quelli dei nodi al numero di nodi interni. Le tabelle leaves_color e inodes_color hanno dimensione, per ogni elemento, di 1 byte, mentre nb_leaves_color e nb_nodes_color usano 2 byte per elemento.

Ripetendo le considerazioni precedenti, lo spazio occupato dalle tabelle degli attributi di nodi e foglie è $(2+1)n \times (t(k) + f(k)) = 3n \times (t(k) + f(k))$ byte.

Ulteriore spazio è allocato per la memorizzazione delle profondità delle foglie di tipo master leaf. Il numero di queste foglie è, nel caso peggiore, pari a n, nel caso migliore è invece pari a 2 (essendo il più piccolo fattore

di troncamento accettato dal programma). Con considerazioni simili alle precedenti, il numero di foglie di tipo master leaf è pari a $n \times t(k)$. Ogni elemento di questo array occupa 4 byte, quindi lo spazio occupato è $4n \times t(k)$ byte.

Sommando i contributi dell'occupazione di memoria, si ottiene: $4n \times (t(k) + f(k)) + 3n \times (t(k) + f(k)) + 4n \times t(k) = 7n \times f(k) + 11n \times t(k)$ byte.

4.1.2 Algoritmo basato su ESA

Nella costruzione del suffix tree troncato con codifica TruST a partire da ESA, gli elementi che richiedono uno spazio dipendente dalla stringa di input sono:

- suffix array (SA);
- longest common prefix (LCP);
- array *last_of_level*.

La memorizzazione del suffix array richiede uno spazio lineare nella lunghezza n della sequenza S. Lo spazio necessario è quello della misura di un intero per simbolo, ossia $4n$ byte. Durante la costruzione del SA viene occupata ulteriore memoria pari a $2.25n$ byte come affermato in [13].

La costruzione del longest common prefix necessita anch'essa di uno spazio lineare in n. L'occupazione per il LCP è di poco più di 1 byte per simbolo. Questo risultato è ottenuto memorizzando gli $lcp < 256$ nello spazio di 1 byte, mentre per gli $lcp \geq 256$ in LCP viene memorizzato un puntatore all'indirizzo di memoria dove viene memorizzato l'effettivo lcp nello spazio di 4 byte [10]. Solitamente gli elementi di un LCP maggiori di 256 sono pochi rispetto alla lunghezza della sequenza, quindi si può affermare che per memorizzare LCP sono necessari $n + O(1)$ byte.

Nella definizione degli lcp-interval, l'unico elemento di dimensioni non costanti che viene creato è l'array *last_of_level*, che serve per la definizione dei fratelli di alcuni lcp-interval/nodi. Questo array occupa, nel caso migliore, uno spazio $O(log(n))$, mentre nel caso peggiore la memoria necessaria è $O(n)$. La dimensione dello spazio da allocare dipende dalla sequenza in esame e dal fattore di troncamento k. Solitamente la dimensione di questo array si avvicina di più ad un andamento logaritmico, questo perchè coincide con l'altezza di un suffix tree troncato. Si può quindi considerare lo spazio occupato da *last_of_level* come $4 \times a\ log(n)$ byte, con a fattore di aggiustamento.

Sommando i contributi dell'occupazione di memoria presi in considerazione, si ottiene: $6.25n + n + 4 \times a\ log(n) = 7.25n + 4 \times a\ log(n)$ byte.

Confronto

Per il confronto sperimentale vengono usati i risultati teorici ottenuti dalle considerazioni fatte precedentemente. In particolare per l'algoritmo basato su illi-TST vengono contati i nodi interni e le foglie di tipo master leaf, mentre per l'algoritmo a partire da ESA viene calcolato la dimensione dell'array $last_of_level$.

Confrontando lo spazio utilizzato nella costruzione del suffix tree troncato con i due algoritmi si vede che non è possibile stabilire a priori quale dei due sia migliore. Infatti la scelta dipende dal fattore di troncamento.

In Tabella 4.1 sono presentati i risultati di occupazione spaziale dei test svolti sulle sequenze presentate in precedenza.

Si può notare come, per alcuni tipi di stringhe, lo spazio occupato da illi-TST cresca drasticamente per fattori di troncamento superiori al 4, al 7, o al 10, mentre quello dell'algoritmo a partire da ESA si attesti su 7,25 byte per simbolo. Addirittura, per alcune sequenze, già per un fattore di troncamento 4 lo spazio occupato dagli attributi di nodi e foglie di tipo master leaf è più che lineare, favorendo la costruzione attraverso ESA.

Stringa S	$\|S\|$	$\|\Sigma\|$	**K factor**	**byte per simbolo**	
				illi-TST	**TruST-ESA**
Taeniopygia 16	9510	4	4	**0,36**	7,25
			7	**4,38**	7,25
			10	7,88	**7,26**
			15	11,43	**7,26**
alfanumerica random	10000	4	4	**0,35**	7,25
			7	10,99	**7,25**
			10	15,29	**7,25**
			15	15,36	**7,26**
numeri random	10000	10	4	7,66	**7,25**
			7	13,84	**7,25**
			10	13,85	**7,25**
			15	13,85	**7,25**
alfabeto random	10000	20	4	12,85	**7,25**
			7	13,43	**7,25**
			10	13,43	**7,25**
			15	13,43	**7,25**
aaa	100000	1	4	**0,01**	7,25
			7	**0,02**	7,25
			10	**0,02**	7,25
			15	**0,02**	7,25
alfanumerica random	100000	4	4	**0,04**	7,25
			7	**2,18**	7,25
			10	14,54	**7,25**
			15	15,35	**7,25**
numeri random	100000	10	4	**1,18**	7,25
			7	13,74	**7,25**
			10	13,83	**7,25**
			15	13,83	**7,25**
alfabeto random	100000	20	4	8,77	**7,25**
			7	13,09	**7,25**
			10	13,09	**7,25**
			15	13,09	**7,25**

Stringa S	$\|S\|$	$\|\Sigma\|$	**K factor**	**byte per simbolo**	
				illi-TST	**TruST-ESA**
alfabeto ripetuto	100000	26	4	**0,02**	7,25
			7	**0,02**	7,25
			10	**0,02**	7,25
			15	**0,03**	7,25
simboli random	100000	64	4	12,29	**7,25**
			7	12,34	**7,25**
			10	12,34	**7,25**
			15	12,34	**7,25**
alice in wonderland	152089	73	4	**1,57**	7,25
			7	**6,93**	7,25
			10	10,93	**7,25**
			15	13,55	**7,25**
Canis 17	885108	4	4	**0,01**	7,25
			7	**0,60**	7,25
			10	7,87	**7,25**
			15	12,36	**7,25**
alfanumerica random	1000000	4	4	**0,01**	7,25
			7	**0,22**	7,25
			10	9,24	**7,25**
			15	15,35	**7,25**
numeri random	1000000	10	4	**0,18**	7,25
			7	12,96	**7,25**
			10	13,82	**7,25**
			15	13,82	**7,25**
pi greco	1000000	10	4	**0,12**	7,25
			7	12,96	**7,25**
			10	13,82	**7,25**
			15	13,82	**7,25**
alfabeto random	1000000	20	4	**1,82**	7,25
			7	13,06	**7,25**
			10	13,06	**7,25**
			15	13,07	**7,25**

Tabella 4.1: confronto di occupazione spaziale fra la costruzione del suffix tree troncato attraverso illi-TST e attraverso ESA (TruST-ESA).

In Tabella 4.2 sono riassunti i risultati.

Tipo stringa	**Fattore troncamento**	**Algoritmo**
Casuale	$k < 4$ $k > 4$	illi-TST ESA
Ripetuta	$\forall k$	illi-TST
Non casuale	$k < 7$ $7 < k < 10$ $k > 10$	illi-TST Indifferente ESA

Tabella 4.2: costruzione suggerita in base al tipo di stringa e al valore di troncamento.

4.2 Analisi temporale

Il confronto tra i tempi di costruzione dei due approcci per i vari tipi di stringa in ingresso è riportato in Tabella 4.3.

Come si può vedere, l'algoritmo che costruisce il suffix tree troncato a partire da ESA è più veloce di quello che utilizza l'algoritmo di Ukkonen modificato.

La differenza tra i tempi di costruzione aumenta all'aumentare della lunghezza della stringa, anche se in alcuni casi, dipendenti dalla particolare stringa in esame, il procedimento originale risulta più veloce di quello proposto in questa tesi.

Stringa S	$\|S\|$	$\|\Sigma\|$	**K factor**	**tempo** (s) **illi-TST**	**TruST-ESA**
alfanumerica random	10000	4	5 30	0,01 0,01	0,01 0,01
numeri random	10000	10	5 30	0,01 0,01	0,01 0,01
alfabeto random	10000	20	5 30	0,02 0,02	**0,01** **0,01**
aaa	100000	1	5 30	0,05 **0,04**	0,05 0,05
alfanumerica random	100000	4	5 30	0,11 0,15	**0,07** **0,09**
numeri random	100000	10	5 30	0,13 0,17	**0,08** **0,09**
alfabeto random	100000	20	5 30	0,15 0,16	**0,07** **0,09**
alfabeto ripetuto	100000	26	5 30	0,19 0,20	**0,10** **0,10**
simboli random	100000	64	5 30	0,37 0,38	**0,14** **0,14**
alice in wonderland	152089	73	5 30	0,31 0,38	**0,25** **0,25**
Canis 17	885108	4	5 30	0,51 0,55	**0,43** **0,46**
alfanumerica random	1000000	4	5 30	**0,88** 2,18	1,51 **1,54**
numeri random	1000000	10	5 30	2,38 2,99	**1,76** **1,82**
pi greco	1000000	10	5 30	2,28 3,03	**1,76** **1,78**
alfabeto random	1000000	20	5 30	4,07 4,28	**1,61** **1,59**

Tabella 4.3: confronto temporale fra la costruzione del suffix tree troncato attraverso illi-TST e attraverso ESA (TruST-ESA).

Conclusioni

Questa tesi ha presentato un nuovo procedimento di costruzione del suffix tree troncato con codifica TruST, che risulta essere più efficiente di quello originariamente utilizzato, in termini sia spaziali che temporali per molti tipi di stringhe.

Si può concludere che nella costruzione di un suffix tree troncato con codifica TruST, l'algoritmo basato su enhanced suffix array è preferibile a quello a partire da illi-TST nelle applicazioni che richiedono efficienza spaziale.

Tuttavia, in alcuni casi dipendenti dal fattore di troncamento k e dalla struttura della sequenza, rimane comunque migliore l'algoritmo basato su illi-TST. Questi casi sono, soprattutto, quelli in cui k è piccolo. La scelta del procedimento da usare, nel caso di fattore di troncamento superiore a 10, è sempre, a meno di sequenze ripetute, a favore di quello basato su ESA.

Una delle possibili applicazioni della codifica TruST è il profile matching, che utilizza, solitamente, sottostringhe di lunghezza approssimativamente 30 e quindi $k \simeq 30$. Si può, dunque, affermare che, per questo tipo di applicazione, la soluzione proposta risulta più efficiente di quella originariamente utilizzata.

Il nuovo procedimento, oltre ad essere più efficiente in termini spaziali, ha prestazioni temporali migliori di quelle dell'algoritmo originale. Nelle applicazioni dove è necessaria la velocità di esecuzione è sempre preferibile la costruzione a partire da ESA a quella basata su illi-TST.

Bibliografia

[1] S. Kurtz, Reducing the space requirement of suffix trees, 1999.

[2] A. Favaretto, Design and analysis of a profile matching algorithm based on truncated suffix tree, Tesi di laurea magistrale, Università di Padova, 2009.

[3] J. Allali e M.F. Sagot, The at-most k-deep factor tree, Relazione tecnica, 2004.

[4] D. Gusfield, Algorithms on String, Trees and Sequences: computer science and computational biology, Cambridge University press, 1997.

[5] A. Apostolico, M. E. Bock e S. Lonardi. Monotony of surprise and large-scale quest for unusual words, Journal of Computational Biology, 10(3-4), pp. 283-311, 2003.

[6] P. Weiner, Linear pattern matching algorithms, In IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory, pp. 1-11, 1973.

[7] E. M. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM, 23(2), pp. 262-272, 1976.

[8] E. Ukkonen, On-line construction of suffix trees, Algorithmica, 14(3), pp. 249-260, 1995.

[9] D. Zucchetto, Analisi sperimentale delle prestazioni di un suffix tree troncato, Tesi di laurea triennale, Università di Padova, 2012.

[10] M. I. Abouelhoda, S. Kurtz e E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, Journal of Discrete Algorithms, 2(1), pp. 53-86, 2004.

[11] J. Kärkkäinen e P. Sanders, Simple linear work suffix array construction, Max Planck Institut für Informatik, ICALP 2003, LNCS 2719, pp. 943-955, 2003.

[12] T. Kasai, G. Lee, H. Arimura, S. Arikawa e K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, CPM 2001, LNCS 2089, pp. 181-192, 2001.

[13] G. Nong, S. Zhang e W.H. Chan, Two efficient algorithms for linear time suffix array construction, IEEE Transactions on Computers, Vol. 60, N 10, Oct. 2011.

[14] J.C. Na, A. Apostolico, C.S. Iliopoulose K. Park, Truncated suffix trees and their application to data compression, Theoretical Computer Science 304, pp. 87-101, 2003.