Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in Ingegneria Informatica

Algoritmi per la costruzione in tempo lineare di alberi dei suffissi

Laureando

Matteo Lora

Relatore

Prof. Carlo Ferrari

Anno Accademico 2011/2012

Indice

Capitolo 1

Gli alberi dei suffissi

Un albero dei suffissi è una struttura dati ad albero utilizzata per contenere stringhe in grado di metterne in evidenza la struttura interna, facilitando la soluzione di diversi problemi, o più in generale, permettendo un approccio differente nella loro risoluzione. L'applicazione classica dell'albero dei suffissi è la soluzione del "problema della sottostringa", ossia: dato un testo T di lunghezza m, dopo un tempo di preelaborazione che richiede tempo $O(m)$, un algoritmo di matching è in grado di determinare la posizione di una stringa S di lunghezza n all'interno del testo T in un tempo linearmente proporzionale $O(n)$ alla lunghezza di S, ossia un tempo indipendente dalla lunghezza m di T. Questo risultato non può essere ottenuto se il testo T è immagazzinato in una sequenza di caratteri. Gli alberi dei suffissi sono dunque spesso usati per risolvere i problemi che occorrono durante particolari operazioni di ricerca all'interno di testi oppure operazioni di modifica degli stessi. Sono inoltre ampiamente utilizzati in applicazioni inerenti alla bioinformatica, nella ricerca di pattern nel DNA o di sequenze di proteine (che possono essere visualizzate come lunghe stringhe di caratteri).

Il primo algoritmo in grado di costruire un albero dei suffissi in tempo lineare è stato presentato da Weiner nel 1973 ed è stato nominato da Donald Knuth Algoritmo dell'anno 1973. Tuttavia tale algoritmo richiedeva uno spazio di esecuzione quadratico $O(n \times \sigma)$ dove σ è la dimensione dell'alfabeto utilizzato. La prima implementazione che richiede tempo e spazio lineari è stata presentata nel 1976 da McCreight. L'algoritmo di McCreight non procede linearmente seguendo l'ordine dei caratteri della stringa ma bensì ad ogni passo esso inserisce nell'albero un suffisso della parola in ordine decrescente rispetto alla loro lunghezza; si dice pertanto che funziona off-line. Un algoritmo sequenziale on-line, lineare nel tempo e nello spazio è stato presentato solo nel 1995 da Esko Ukkonen, algoritmo il cui unico limite è quello di richiedere che l'alfabeto utilizzato sia finito.

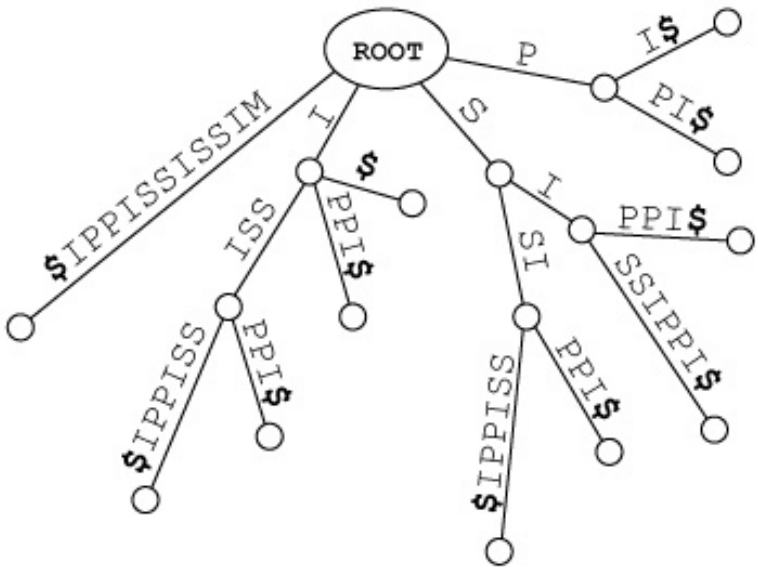

Figura 1.1: Esempio di albero di suffissi per la stringa Mississippi

1.1 Definizione di base

Un albero dei suffissi A per una stringa S di lunghezza m è un albero radicato e orientato che possiede esattamente m foglie numerate da 1 ad m. Ogni nodo interno, esclusa la radice, ha almeno due figli ed ognuno dei rami che lo connettono ad essi è etichettato con una sottostringa di S non vuota. Due rami uscenti da un nodo non possono avere etichette che iniziano con lo stesso carattere. La caratteristica chiave dell'albero dei suffissi è che per ogni cammino dalla radice alla foglia i, la concatenazione delle etichette dei rami attraversati costituisce il suffisso di S a partire dal carattere nella posizione i. Tale cammino è detto $S[i...m]$. E' inoltre fondamentale che nessun suffisso x della stringa S corrisponda ad un prefisso di un altro suffisso di S, altrimenti non può esistere un albero che rispetta la precedente definizione, poichè il cammino relativo ad x non terminerebbe in un nodo foglia. Per evitare questo problema viene introdotto un carattere speciale, detto di terminazione, diverso da ogni altro carattere dell'alfabeto e posto come

ultimo carattere della stringa in modo tale che nessun suffisso della stringa risultante possa essere un prefisso di altri suffissi. L'etichetta di un cammino dalla radice che termina in un nodo è la concatenazione delle sottostringhe che sono etichette dei rami attraversati da tale cammino. L'etichetta di un nodo è la concatenazione delle etichette corrispondenti ai rami attraversati da un cammino dalla radice a tale nodo. Per ogni nodo v in un albero dei suffissi, la profondità di v è il numero di caratteri nell'etichetta di v. Il ramo che congiunge due nodi u e v è denotato con (u, v). Un cammino che termina nel mezzo di un ramo tra i nodi u e v, spezza l'etichetta di (u, v) nel punto stabilito. L'etichetta di tale cammino è definita come l'etichetta di u concatenata ai caratteri dell'etichetta del ramo (u, v) fino al punto stabilito.

1.2 Algoritmo ingenuo di costruzione

Un algoritmo di facile comprensione, sebbene inefficiente, per la costruzione di un albero dei suffissi, è il seguente:
Sia data una stringa S di lunghezza m e il suo relativo albero dei suffissi T inizialmente vuoto e costituito solamente dal nodo radice. Si inizia la costruzione dell'albero dei suffissi inserendo un solo ramo per il suffisso $S[1...m]$ ossia l'intera stringa, quindi aggiunge uno alla volta i suffissi $S[i...m]$ per $i = 2, 3, ..., m$. Si indica con T_i l'albero intermedio che contiene i suffissi che iniziano nelle posizioni da 1 a i. L'albero T_1 consiste di un unico ramo etichettato con la stringa $S[1...m]$ che congiunge la radice ad una foglia numerata 1. Ogni albero T_{i+1} viene costruito a partire da T_i nel seguente modo: Partendo dalla radice di T_i cerca il più lungo cammino dalla radice ad un nodo la cui etichetta è prefisso del suffisso $S[i + 1...m]$\$ da aggiungere. Tale ricerca si effettua partendo dal cammino nullo (che inizia e termina nella radice e che ha la stringa nulla come etichetta) ed estendendolo, finché è possibile, aggiungendo uno alla volta i caratteri del suffisso $S[i + 1...m]$\$. Il cammino che si ottiene in questo modo è unico in quanto le etichette dei rami uscenti da un nodo esplicito iniziano sempre con caratteri tra loro diversi. Inoltre l'estensione deve terminare prima della fine del suffisso $S[i + 1...m]$ in quanto la presenza della sentinella \$ ci assicura che il suffisso $S[i + 1...m]$ non è prefisso di nessuno dei suffissi più lunghi inseriti precedentemente nell'albero. Se il nodo in cui tale cammino termina è un nodo implicito, tale nodo viene sostituito con un nodo esplicito spezzando il ramo che lo contiene in due rami (e l'etichetta in due etichette). A questo punto il nodo u a cui si arriva è un nodo esplicito con etichetta il prefisso $S[i + 1...j]$ di $S[i + 1...m]$. Si aggiunge un nuovo ramo $(u, i + 1)$ con etichetta $S[j + 1, n]$ che congiunge il nodo u con una nuova foglia numerata $i + 1$. A questo punto l'albero contiene un unico cammino dalla radice alla foglia $i + 1$ la cui etichetta è il suffisso $S[i + 1, n]$ e si è quindi ottenuto l'albero T_{i+1}. Questo algoritmo richiede tempo $O(n^2)$ e non è dunque in grado di costruire l'albero in tempo

lineare. L'analisi temporale che porta a questo risultato è piuttosto semplice: l'algoritmo richiede m fasi, cioè un numero pari alla lunghezza della stringa data. Ogni fase i richiede poi che l'albero sia percorso interamente dalla radice fino al più lungo prefisso della sottostringa $S[i+1...m]$ presente nell'albero, nel quale avranno luogo le successive operazioni che richiedono un tempo costante. L'algoritmo non è pertanto efficiente. Nei capitoli successivi sono presentati gli algoritmi di Weiner e Ukkonen, i quali sono in grado, sfruttando diverse osservazioni ed accorgimenti, di costruire l'intero albero dei suffissi in un tempo lineare.

Capitolo 2

L'algoritmo di Weiner

L'algoritmo di Weiner opera sull'intera stringa $S\$$ inserendo un suffisso alla volta in un albero crescente. Inserisce prima la sottostringa $S(m)\$$ nell'albero, poi la sottostringa $S[m-1\ldots m]\$, \ldots,$ e alla fine l'intera stringa S$.

Per comodità si denota con $Suff_i$ il suffisso $S[i...m]$ di S a partire dalla posizione i. Per esempio, $Suff_1$ rappresenta l'intera stringa mentre $Suff_m$ rappresenta il singolo carattere $S(m)$.

Si definisce T_i come l'albero che ha $m-i+2$ foglie numerate da i ad $m+1$ tali che il percorso dalla radice ad ogni foglia j $(i \leq j \leq m+1)$ abbia etichetta $Suff_j\$$. Quindi T_i è un albero che codifica tutti e solamente i suffissi della stringa $S[i...m]\$$.

L'algoritmo di Weiner costruisce alberi T_i da T_{m+1} fino a T_1 (in ordine decrescente di i).

2.1 Una prima semplice implementazione

Il primo albero T_{m+1} consiste semplicemente di un ramo uscente dalla radice etichettato con il carattere di terminazione $. Poi, per ogni i da m ad 1, l'algoritmo costruisce ogni albero T_i partendo dal suo predecessore T_{i+1} ed il carattere $S(i)$. Mano a mano che l'algoritmo procede, ogni albero T_i conserverà la proprietà che per ogni nodo v di T_i non vi sono due rami uscenti da v la cui etichetta inizia con lo stesso carattere.

Definizione 2.1. Per una qualunque posizione i, $Head(i)$ denota il più lungo prefisso di $S[i\ldots m]$ che corrisponde ad una sottostringa di $S[i+1...m]\$$.

Si nota che $Head(i)$ potrebbe anche la stringa vuota. Per esempio infatti $Head(m)$ è sempre la stringa vuota dato che $S[i+1...m]$ è la stringa vuota quando $i+1$ è maggiore di m ed il carattere $S(m) \neq \$$.

2.1.1 Algoritmo ingenuo Weiner

1. Si trova il termine del percorso etichettato $Head(i)$ in T_{i+1}.

2. Se non vi è un nodo esplicito al termine di $Head(i)$ allora ne viene creato uno, denotato con w. Se w è stato effettivamente creato, dividendo a metà un ramo esistente, allora l'etichetta di tale ramo è divisa facendo si che l'etichetta del percorso dalla radice al nodo w corrisponda ad $Head(i)$. Successivamente viene creata una nuova foglia numerata i ed un nuovo ramo (w, i) etichettato con i caratteri rimanenti di $Suff_i\$$.

2.2 Verso un'implementazione più efficiente

L'albero dei suffissi finale $T = T_1$ è costruito in un tempo $O(m^2)$ da questo semplice approccio. E' chiaro che la parte più onerosa dell'algoritmo è costituita dalla ricerca di $Head(i)$, dato che le operazioni successive richiedono un tempo costante.

2.2.1 Ricerca di Head(i) efficiente

Il problema della ricerca di $Head(i)$ dell'algoritmo di Weiner è risolto con l'inserimento in ogni nodo non foglia (radice inclusa) di due vettori, chiamati rispettivamente indicator vector I e link vector L. Ogni vettore contiene un numero di oggetti pari alle dimensioni dell'alfabeto, e ognuno di essi è legato ad un simbolo dell'alfabeto tramite il loro indice. Per esempio, per l'alfabeto inglese ogni vettore avrà lunghezza 26, oppure 27 se si considera anche il simbolo di terminazione \$.

Definizione 2.2 (Indicator vector)**.** L'indicator vector I è un vettore contenente valori binari o booleani. Si denota con $I_{v(x)}$ il valore nell'indicator vector del nodo v nella posizione indicizzata dal carattere x-esimo. Per ogni carattere x dell'alfabeto ed ogni nodo u, $I_{u(x)} = 1$ in T_{i+1} se e solo se esiste un percorso dalla radice di T_{i+1} etichettato xa, dove a è l'etichetta del nodo u.

Definizione 2.3 (Link vector)**.** Il link vector è un vettore contenente oggetti che possono essere *null* oppure puntatori a nodi. Si denota con $L_{v(x)}$ il valore nel link vector del nodo v nella posizione indicizzata dal carattere x-esimo. Per ogni carattere x dell'alfabeto ed ogni nodo u, $L_{u(x)}$ in T_{i+1} punta ad un nodo u' in T_{i+1} se e solo se u' ha etichetta xa, ed u ha etichetta a. Altrimenti $L_{u(x)}$ è *null*.

E' chiaro che per ogni nodo u e ogni carattere x, $L_{u(x)}$ è *nonnull* solo se $I_{u(x)} = 1$, ma non è necessariamente vero il contrario. E' inoltre immediato che se $I_{u(x)} = 1$ allora $I_{v(x)} = 1$ per ogni nodo v padre di u. L'algoritmo di Weiner utilizza gli indicator vector e i link vector per trovare $Head(i)$

e costruire T_i in modo efficiente. L'algoritmo deve inoltre tenere in considerazione due casi degeneri, anche se essi non sono molto diversi dal caso generico.

2.2.2 L'algoritmo nel caso standard

Si assume che l'albero T_{i+1} sia appena stato costruito e si prosegue con T_i. L'algoritmo inizia alla foglia $i + 1$ di T_{i+1} e percorre l'albero verso la radice fino al primo nodo v, se esiste, tale che $I_{v(S(i))} = 1$. Se trovato, continua a percorrere l'albero verso la radice in cerca del primo nodo v' (possibilmente esso potrebbe coincidere con v) dove $L_{v'(S(i))}$ sia *nonnull*. Se anch'esso viene trovato, si denota con n_i il numero di caratteri che compongono l'etichetta del percorso tra v e v', con c il primo di tali caratteri, e con v'' il nodo puntato dal link vector di v'. In questo caso $Head(i)$ si trova n_i caratteri al di sotto di v'' lungo uno dei rami uscenti da v''.

In generale, è possibile che ne v ne v' esistano o che v esista ma v' no. Inoltre v o v' potrebbero essere la radice. Il caso generico definito good case è il caso in cui entrambi v e v' esistono. In questo caso, dunque, $Head(i)$ può essere trovato in tempo costante dopo che v' è stato trovato.

Dunque quando $n_i = 0$, si sa che $Head(i)$ termina in v'', e quando $n_i > 0$, si trova $Head(i)$ esaminando tra i rami uscenti da v'' quale ramo, denotato con e, ha come primo carattere dell'etichetta uguale a c. Quindi $Head(i)$ termina esattamente dopo n_i caratteri percorrendo e da v''. L'albero T_i è dunque costruito spezzando il ramo e, creando un nodo w in tale posizione ed aggiungendo un nuovo ramo dal nodo w alla foglia i etichettato con il rimanente di $Suff_i\$$.

2.2.3 Analisi dei casi degeneri

Esistono due casi detti degeneri che si differenziano dal caso standard e che necessitano di particolare attenzione:

1. ll nodo v non esiste e dunque nemmeno il nodo v'.
   In questo caso il percorso termina alla radice e non viene trovato il nodo v. Ne consegue che il carattere $S(i)$ non appare in nessuna posizione e $Head(i)$ è la stringa vuota e termina alla radice.

2. Il nodo v esiste ma il nodo v' non esiste.
   $I_{v(S(i))} = 1$ per qualche v (possibilmente anche la radice), ma v' non esiste pertanto il percorso termina alla radice con $L_{r(S(i))} = null$. Si denota con t_i il numero di caratteri dalla radice a v. $Head(i)$ termina esattamente a t_{i+1} caratteri dalla radice. Dato che v esiste, esiste un ramo $e = (r, z)$ la cui etichetta inizia con il carattere $S(i)$. Questo è vero sia per $t_i = 0$ che per $t_i > 0$. Se $t_i = 0$ allora $Head(i)$ termina al primo carattere, $S(i)$, sul ramo e. In modo del tutto analogo, se

$t_i > 0$ allora $Head(i)$ termina esattamente dopo t_{i+1} caratteri dalla radice sul ramo e.

In entrambi questi casi (come nel caso generico), $Head(i)$ viene trovato in tempo costante una volta raggiunta la radice. Dopo che è stato trovato il termine di $Head(i)$ ed è stato creato o trovato il nodo w, l'algoritmo procede esattamente come nel caso generico.

2.3 L'algoritmo completo

Questo procedimento incorpora tutti i casi possibili presentati in precedenza:

1. Si comincia dalla foglia $i + 1$ di T_{i+1} (la foglia per il suffisso $Suff_{i+1}$) e si percorre l'albero verso la radice in cerca del primo nodo v sul percorso tale che $I_{v(S(i))} = 1$.

2. Se si raggiunge la radice e $I_{r(S(i))} = 0$, allora $Head(i)$ termina alla radice. Saltare al punto 4.

3a. Se si raggiunge la radice e $L_{r(S(i))}$ è *null*, allora si denota con t_i il numero di caratteri nel percorso tra la radice e v. Si cerca un ramo e uscente dalla radice la cui etichetta inizi con $S(i)$. $Head(i)$ termina esattamente a t_{i+1} caratteri dalla radice sul ramo e.

3b. Se è trovato un nodo v' tale che $L_{v'(S(i))}$ è *nonnull*, si segue il link da v' al nodo che sarà detto v''. Si denota con n_i il numero di caratteri nel percorso tra v' e v e sia c il primo di questi caratteri. Se $n_i = 0$ allora $Head(i)$ termina in v'', altrimenti si cerca un ramo e uscente da v'' il cui primo carattere dell'etichetta sia c. $Head(i)$ termina esattamente a n_i caratteri da v'', sul ramo e.

4. Se al termine di $Head(i)$ esiste già un nodo, allora si denota con w tale nodo, altrimenti ne viene creato uno. Viene creato un nuovo nodo foglia numerato i ed un ramo (w, i) etichettato con i caratteri rimanenti della sottostringa di $Suff_i$ (gli ultimi $m - i + 1 - |Head(i)|$ caratteri di $Suff_i$), seguiti dal carattere di terminazione \$.

2.3.1 Aggiornamento dei vettori

Dopo aver trovato o creato il nodo w, è necessario aggiornare i vettori I ed L cosicché siano corretti per il nuovo albero T_i. Se l'algoritmo trova un nodo v tale che $I_{v(S(i))} = 1$, allora il nodo w avrà etichettà $S(i)a$ in T_i, dove il nodo v ha etichetta a. Allora in $L_{v(S(i))}$ deve essere inserito un puntatore a w in T_i.

Questo è l'unico aggiornamento necessario per quanto riguarda i link vectors in quanto solo un nodo può puntare ad ogni altro nodo tramite il link vector ed un solo nodo viene creato in ogni fase. Inoltre, se il nodo w è stato creato durante l'ultima fase, tutte le entry della sua tabella dei link-vector devono essere *null*. Per quanto riguarda gli indicator vectors invece, per ogni nodo u sul percorso dalla radice alla foglia $i + 1$, $I_{u(S(i))}$ dev'essere impostato a *1* in T_i perché esiste ora un cammino che corrisponde alla stringa $Suff_i$ in T_i. E' facile stabilire induttivamente che se un nodo v tale che $I_{v(S(i))} = 1$ è trovato durante il percorso dalla foglia $i + 1$ alla radice, allora ogni nodo u collocato sopra v nel percorso ha già $I_{u(S(i))} = 1$. Quindi solamente i vettori per i nodi collocati sotto a v nel percorso verso la foglia $i + 1$ devono essere aggiornati. Se invece non è stato trovato nessun nodo v, allora si è costretti ad attraversare e quindi aggiornare tutti i nodi del percorso dalla foglia $i + 1$ alla radice.

Gli aggiornamenti dei vettori richiesti per i nodi collocati sotto v possono essere eseguiti durante la ricerca di v e non sono richieste ulteriori operazioni. Durante il percorso dalla foglia $i + 1$, $I_{u(S(i))}$ è impostato a *1* per ogni nodo u incontrato durante il percorso. E' poi eseguito successivamente l'aggiornamento dell'indicator vector per il nodo eventualmente appena creato w. Quando invece un nodo w è creato all'interno del ramo e (v'', z), l'indicator vector per w è copiato dall'indicator vector di z.

2.3.2 Analisi temporale

Il tempo per costruire T_i da T_{i+1} ed aggiornare i vettori è proporzionale al tempo richiesto dal percorrimento dell'albero dalla foglia $i + 1$ al nodo v' o alla radice. Questo percorso è seguito muovendo un puntatore da un nodo al suo nodo padre, richiedendo per tale operazione un tempo costante. Tempo costante è richiesto anche per seguire un link tra due nodi, e per le operazioni richieste per aggiungere il nodo w ed il ramo (w, i). Dunque il tempo necessario a costruire T_i è proporzionale al numero di nodi incontrati nel percorso dalla foglia $i + 1$ alla radice. Si ricorda che la profondità di un nodo v è il numero di nodi che si incontrano nel percorso da tale nodo alla radice. Per l'analisi temporale si immagina che man mano che l'algoritmo procede si tenga traccia di quale nodo si è incontrato di recente e di quale sia la sua profondità. Si denota con profondità attuale la profondità dell'ultimo nodo incontrato. Per esempio, quando l'algoritmo inizia, la profondità attuale è 1 e subito dopo che l'albero T_m è stato creato la profondità attuale è 2. Chiaramente, man mano che l'algoritmo percorre l'albero dalla foglia alla radice la profondità attuale decresce di una unità ad ogni passaggio. Inoltre, quando l'algoritmo si trova al nodo v'' o alla radice e quindi crea un nuovo nodo w sotto v'' (o sotto la radice), la profondità attuale aumenta di uno. Quando invece il puntatore attraversa un link da un nodo v' a un nodo v'' in T_{i+1}, la profondità attuale aumenta al massimo di una unità. La profondità

attuale può dunque aumentare di una unità al massimo ogni volta che un nuovo nodo viene creato e ogni volta che viene attraversato un link tra due nodi. Quindi il numero totale di incrementi della profondità attuale è al massimo $2m$. Ne consegue che la profondità attuale può anche decrescere un numero massimo di $2m$ volte durante il percorrimento dell'albero, quindi il numero totale di nodi visitati è al massimo $2m$. Si può dunque affermare certamente che l'algoritmo di Weiner costruisce l'albero dei suffissi per una stringa di lunghezza m in un tempo $O(m)$, assumendo finito l'alfabeto utilizzato.

Capitolo 3

L'algoritmo di Ukkonen

Esko Ukkonen individuò un algoritmo per la costruzione in tempo lineare di alberi dei suffissi che è possibilmente il più semplice a livello concettuale. Esso permette inoltre di risparmiare spazio in memoria rispetto all'algoritmo di Weiner, non utilizzando vettori per ogni nodo. L'algoritmo di Ukkonen costruisce una sequenza di alberi dei suffissi impliciti, l'ultimo dei quali è poi trasformato in un vero e proprio albero di suffissi per la stringa S.

Definizione 3.1. Un albero dei suffissi implicito per la stringa S è un albero ottenuto dall'albero dei suffissi per S\$ rimuovendo ogni copia del simbolo di terminazione \$ dalle etichette dei rami dell'albero e rimuovendo poi ogni ramo privo di etichetta e ogni nodo che non possiede almeno due figli.

Si denota l'albero dei suffissi implicito per la stringa $S[1...i]$ con I_i, per i da *1* ad m.

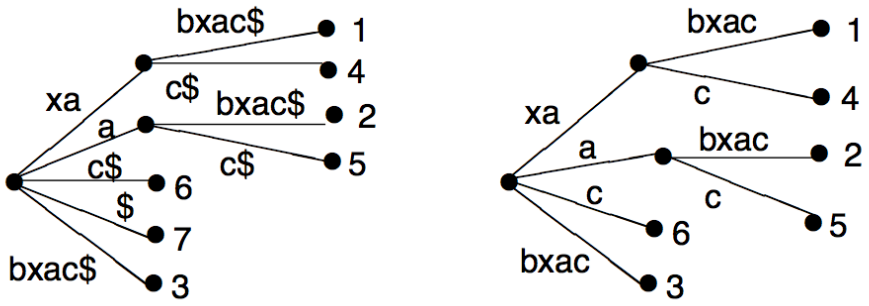

Figura 3.1: Esempio di un albero dei suffissi implicito ed esplicito per la stringa xabxac [3]

3.1 L'algoritmo ad alto livello

L'algoritmo di Ukkonen costruisce un albero dei suffissi implicito I_i per ogni prefisso $S[1...i]$ di S, a partire da I_1 e incrementando i di una unità fino alla costruzione di I_m. Il vero albero dei suffissi per S è poi successivamente costruito a partire da I_m. Tale algoritmo viene presentato partendo da uno più semplice, eseguito in $O(m^3)$ e la cui implementazione è poi ottimizzata per raggiungere le prestazioni desiderate.

L'algoritmo si divide in m fasi. Nella fase $i+1$, è costruito l'albero I_{i+1} a partire da quello precedente I_i. Ogni fase è a sua volta suddivisa in $i+1$ estensioni, una per ognuno degli $i+1$ suffissi di $S[1...i+1]$. Durante la j-esima estensione della fase $i+1$, l'algoritmo trova prima la fine del percorso che ha origine nella radice e che ha etichetta $S[j...i]$, denotato con B. Poi estende il ramo aggiungendo il carattere $S(i+1)$ al termine di essa, a meno che $S(i+1)$ compaia già in tale posizione. Più precisamente, l'estensione viene effettuata mediante l'applicazione di una delle seguenti regole:

1. Nell'albero corrente il cammino B termina in una foglia.

   Per effettuare l'estensione il carattere $S(i+1)$ viene aggiunto al termine dell'etichetta del ramo precedente la foglia.

2. Non esiste un cammino alla fine della stringa B che inizi per $S(i+1)$, ma è già presente nell'albero corrente un cammino (seppur con etichetta non compatibile con il carattere $S(i+1)$) al termine della stringa B.

   In questo caso viene aggiunto un nuovo ramo a partire dal termine di B ed etichettato con il carattere $S(i+1)$. Viene inoltre creato un nuovo nodo se B termina nel mezzo di un ramo. La foglia al termine del ramo appena creato sarà numerata con j.

3. E' presente nell'albero corrente, al termine della stringa B, un percorso che inizia con il carattere $S(i+1)$.

   In questo caso la stringa $BS(i+1)$ è già presente nell'albero corrente e l'estensione non viene eseguita.

Una volta trovato il termine del suffisso B di $S[1...i]$, è necessario un tempo costante per eseguire le regole di estensione. Ingenuamente è possibile cercare il termine di ogni suffisso B in $O(|B|)$ partendo dalla radice dell'albero. Mediante questo approccio, l'estensione j della fase $i+1$ impiegherebbe un tempo $O(i+1-j)$, I_{i+1} potrebbe essere creato da I_i in $O(i^2)$ e I_m potrebbe dunque essere creato in $O(m^3)$. Questo tempo verrà ridotto a O(m) mediante alcune osservazioni e trucchi, i quali, implementati tutti assieme, permettono di raggiungere il tempo lineare nel peggiore dei casi o worst-case. Il più importante di essi è detto suffix links.

3.2 Implementazione e miglioramenti

La chiave per l'ottimizzazione dell'algoritmo di Ukkonen, come visto in precedenza, è dunque la ricerca del termine di tutti gli $i + 1$ suffissi di $S[1...i]$ in un modo più efficiente, problema del tutto analogo alla ricerca di $Head(i)$ nell'algoritmo di Weiner. A differenza dell'algoritmo di Weiner, tuttavia, gli inserimenti dei suffissi vengono effettuati secondo un ordine diverso, un ordine che permette di sfruttare gli accorgimenti che sono in seguito presentati.

3.2.1 Suffix links

Definizione 3.2. Si denota con xa una stringa arbitraria, dove x denota un singolo carattere mentre a denota una sottostringa (possibilmente vuota). Per un nodo interno v con etichetta xa, se esiste un altro nodo $s(v)$ con etichetta a, allora esiste un puntatore da v ad $s(v)$ che è detto suffix link.

Come caso speciale, se a è vuota, allora il suffix link da un nodo interno con etichetta xa termina nella radice. La radice non è considerata nodo interno e non ha suffix link che hanno origine in essa. Inoltre, anche se la definizione non lo implica, ogni nodo interno di un albero dei suffissi implicito avrà di fatto un suffix link che origina in esso. Si osserva infatti che un nodo interno v con etichetta xa viene aggiunto all'albero corrente durante una estensione j di una qualche fase $i + 1$, allora o il cammino con etichetta a termina già in un nodo interno dell'albero corrente oppure tale nodo etichettato a sarà creato dalla successiva regola di estensione $j + 1$ della stessa fase $i + 1$. Di conseguenza nell'algoritmo di Ukkonen, ogni nuovo nodo interno creato avrà un suffix link a partire da esso entro l'estensione successiva. Inoltre, in ogni albero dei suffissi implicito I_i, se il nodo interno v ha etichetta xa, allora esiste un nodo $s(v)$ di I_i con etichetta a.

Si ricorda che nell'estensione j-esima della fase $i + 1$ l'algoritmo trova il suffisso $S[j...i]$ di $S[1..i]$, per j che varia da 1 a $i + 1$. Ingenuamente, questo può essere fatto confrontando la stringa $S[j...i]$ durante il percorso dalla radice nell'albero corrente. I suffix links forniscono una scorciatoia per questo percorso in ogni estensione. La prima estensione (per $j = 1$) della fase $i + 1$ è la più facile da descrivere. Il termine della stringa $S[1...i]$ deve terminare in una foglia di I_i dato che $S[1...i]$ è la stringa più lunga rappresentata in tale albero. Questo rende facile trovare il termine di tale suffisso (mano a mano che gli alberi vengono costruiti, si tiene un puntatore alla foglia che corrisponde alla stringa intera corrente $S[1..i]$), e la sua estensione è gestita tramite la prima delle regole di estensione. Quindi la prima estensione di ogni fase è speciale e richiede ogni volta tempo costante dato che l'algoritmo tiene traccia del puntatore della stringa intera corrente.

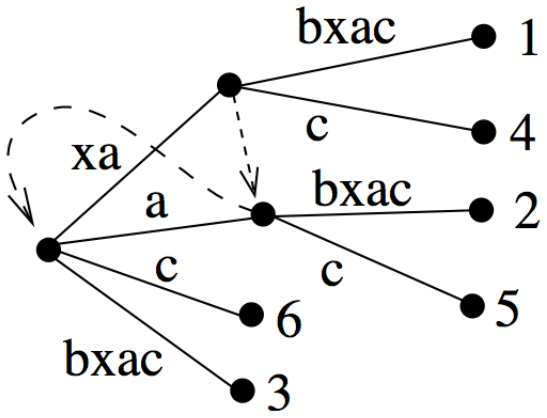

Figura 3.2: Albero dei suffissi implicito per la stringa xabxac con suffix links. [3]

3.2.2 Regole di estensione con suffix links

Le estensioni $j \geq 2$ della fase $i + 1$ in un albero implementato con suffix links sono così eseguite:

1. Si trova il primo nodo v alla fine o al di sopra di $S[j-1...i]$ che abbia un suffix link a partire da esso o sia la radice. Questo richiede di risalire al massimo un ramo dalla fine di $S[j-1...i]$ nell'albero corrente. Si denota con y la stringa (possibilmente vuota) tra v e il termine di $S[j-1...i]$.

2. Se v non è la radice, si attraversa il suffix link da v fino al nodo $s(v)$ e quindi si percorre verso il basso il percorso corrispondente alla stringa y. Se v è la radice, allora si segue il percorso corrispondente a $S[j...i]$ dalla radice.

3. Utilizzando le regole di estensione (vedi 3.1.1), si assicura che $S[j...i]S(i+1)$ sia presente nell'albero.

4. Se è stato creato un nuovo nodo interno w durante l'estensione $j-1$ (per la regola di estensione 2), allora la stringa a deve terminare al nodo $s(w)$, nodo di arrivo per il suffix link da w. Viene creato dunque tale suffix link da w a $s(w)$.

Assumendo che l'algoritmo tenga sempre un puntatore alla stringa intera $S[1...i]$ dell'albero corrente, la prima estensione della fase $i+1$ non richiede di percorrere rami, inoltre per effettuare la prima estensione è sempre applicata la regola di estensione 1.

3.2.3 Skip/count

L'uso dei suffix links è un notevole passo avanti rispetto al dover percorrere ad ogni estensione l'albero partendo dalla radice, tuttavia di per se non migliora il tempo di esecuzione nel caso peggiore. Per far questo si utilizza una procedura che riduce il tempo nel caso peggiore a $O(m^2)$. Nel secondo passaggio dell'estensione j+1 l'algoritmo percorre l'albero dal nodo $s(v)$ lungo il percorso etichettato come y. Implementato direttamente, questo percorso richiede un tempo proporzionale alla lunghezza di y. Tuttavia, un semplice trucco detto skip/count, riduce il tempo di attraversamento di questo percorso ad un tempo proporzionale al numero di nodi presenti nel percorso.

Si denota con g la lunghezza di y, e si ricorda che non ci possono essere due rami uscenti da $s(v)$ la cui etichetta inizia con lo stesso carattere, quindi il primo carattere di y deve apparire come il primo di uno dei rami uscenti da $s(v)$. Si denota con g' il numero di caratteri in tale ramo. Se g' è inferiore a g, allora l'algoritmo non controlla i caratteri in tale ramo, ma salta direttamente al nodo che si trova alla sua estremità. A questo punto si impostano le variabili $g = g - g'$, $h = g' + 1$, e si guarda intorno in cerca del ramo uscente dal nuovo nodo che abbia iniziale uguale al carattere h di y. In generale, quando l'algoritmo identifica il prossimo ramo da percorrere, confronta il valore corrente di g con il numero di caratteri in tale ramo. Quando g è almeno grande quanto g', l'algoritmo salta direttamente al nodo al termine di tale ramo, e riassegna i valori delle variabili nel seguente modo: $g = g - g'$ ed $h = h + g'$. Poi si cerca quale ramo uscente ha il carattere equivalente a quello nella posizione h di y e così via. Quando viene raggiunto ramo in cui g è inferiore o uguale a g', allora l'algoritmo passa direttamente al carattere g in tale ramo e si interrompe.

Assumendo vere alcune condizioni legate all'implementazione dell'albero, come il conoscere il numero di caratteri nell'etichetta di ogni ramo ed essere in grado di estrarne uno particolare in qualunque posizione esso si trovi in tempo costante, l'utilizzo di skip/count permette di muoversi da un nodo al seguente in tempo costante, ed il tempo totale di attraversamento del percorso è dunque proporzionale al numero di nodi piuttosto che al numero di caratteri.

Utilizzando skip/count, l'algoritmo di Ukkonen può essere interamente eseguito in un tempo $O(m^2)$. Tale stima temporale si ottiene moltiplicando il tempo richiesto da ogni fase per il numero delle fasi. Per ottenere un tempo totale lineare è necessario apportare dei cambiamenti all'implementazione. Infatti poiché un albero di suffissi può richiedere uno spazio di $\Theta(m^2)$ ed il tempo di esecuzione dell'algoritmo è almeno grande quanto le dimensioni del suo output, per raggiungere il tempo lineare è necessario introdurre uno schema differente per la rappresentazione delle etichette dei rami.

3.2.4 Compressione delle etichette

Esiste uno schema semplice ed alternativo per etichettare i rami. Invece di scrivere esplicitamente una sottostringa in un ramo dell'albero, viene scritta una coppia di indici che indicano le posizioni di inizio e di fine di tale sottostringa in S. Dal momento che l'algoritmo possiede una copia di S, può localizzare ogni carattere all'interno di essa in tempo costante conoscendone la posizione.

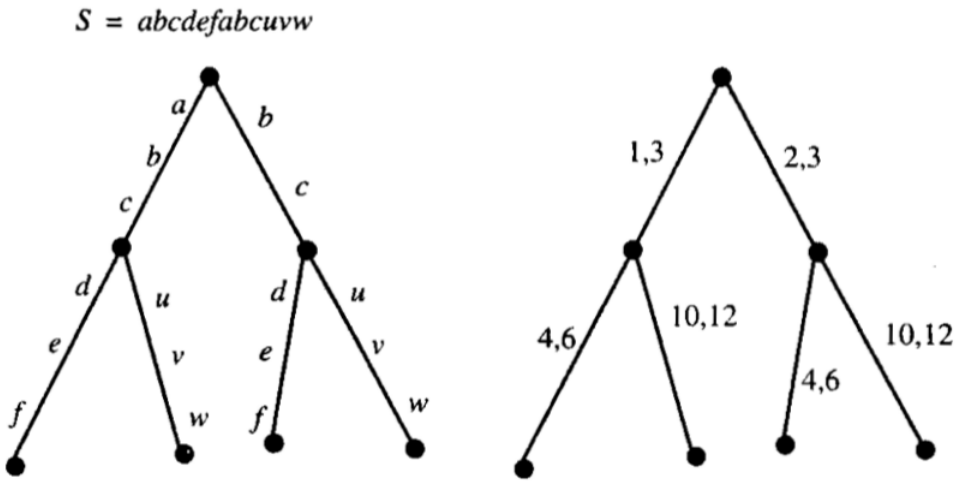

Figura 3.3: Esempio di albero dei suffissi con compressione delle etichette [1]

Nell'algoritmo di Ukkonen dunque, quando si eseguono i confronti lungo un lato, si utilizzano gli indici scritti nell'etichetta di tale lato per recuperare i caratteri richiesti di S si eseguono i confronti su di essi. In questo modo è semplice utilizzare anche le regole di estensione presentate in precedenza: quando viene applicata la regola 2 durante una fase $i + 1$, viene etichettato il nuovo ramo creato con la coppia $(i + 1, i + 1)$, e quando viene applicata la regola 1 (su un ramo che termina in un nodo foglia), la sua etichetta cambia da (p, q) a $(p, q + 1)$. Utilizzando una coppia di indici per specificare l'etichetta di un ramo, l'albero dei suffissi utilizza solo $O(m)$ simboli e richiede $O(m)$ spazio, dal momento che il numero di rami è al massimo $2m - 1$.

3.2.5 Due ulteriori accorgimenti

Sono presentate due ulteriori euristiche sviluppate in seguito a due importanti osservazioni, l'implementazione delle quali permette il raggiungimento del tempo di esecuzione desiderato.

1. In ogni fase, se si applica la regola 3 durante l'estensione j, si applicherà anche in tutte le successive estensioni (da $j + 1$ a $i + 1$) fino al termine

di tale fase. Il motivo è che quando la regola 3 viene applicata, il percorso etichettato $S[j...i]$ nell'albero corrente deve proseguire con il carattere $S(i+1)$, e così anche il percorso etichettato $S[j+1..i]$, quindi la regola 3 si applica poi a tutte le estensioni $j+1, j+2 \ldots i+1$.

Ogni fase $i+1$ viene terminata nel momento in cui viene applicata la regola *3*. Se questo avviene durante l'estensione j, non è necessario dunque cercare il termine di ogni stringa $S[k...i+1]$ per $k > j$. Per questo si dice che tali ricerche vengano eseguite implicitamente, al contrario di tutte le altre estensioni per $k < j$ dove invece il termine di $S[k...i+1]$ viene esplicitamente trovato.

2. Se in un punto dell'esecuzione dell'algoritmo viene creata una foglia etichettata j, allora tale nodo rimarrà una foglia in tutti gli alberi creati successivamente dall'algoritmo. Questo è sempre vero in quanto l'algoritmo non possiede un meccanismo per estendere un ramo che termina in un nodo foglia oltre il nodo stesso. Questo avviene in quanto dal momento che viene creato un nodo foglia j, verrà sempre applicata la regola *1* all'estensione j di tutte le fasi successive.

   Nella fase $i+1$, quando un ramo foglia viene creato e verrebbe normalmente etichettato con la sottostringa $S[p...i+1]$, invece di scrivere gli indici $(p, i+1)$ nell'etichetta del ramo, si scrive (p, e), dove e è un simbolo che denota la fine attuale. Essa è una variabile globale che viene aggiornata ad $i+1$ durante ogni fase. Nella fase $i+1$, dato che l'algoritmo sa che verrà applicata la regola 1 nell'estensione 1 attraverso j_i, almeno, non è necessario ulteriore lavoro per implementare tali estensioni. Invece, viene incrementata solamente la variabile e, e poi eventualmente il lavoro esplicito inizierà dall'estensione j_{i+1}.

3.3 L'algoritmo completo

Applicando le osservazioni viste in (3.2.5) le estensioni esplicite nella fase $i+1$ sono dunque necessarie a partire dall'estensione j_{i+1} fino alla prima estensione in cui si applica la regola 3, oppure fino all'estensione $i+1$. Tutte le altre estensioni sono eseguite implicitamente.

1. Si incrementa il valore di e ad $i+1$. (Per il 3.2.5.2 questo passaggio esegue correttamente tutte le estensioni implicite da 1 fino a j_i.

2. Si eseguono le estensioni successive partendo da j_{i+1} fino al raggiungimento della prima estensione j' dove viene applicata la regola 3 o fino all'estensione $i+1$.

3. Si aggiorna j_{i+1} a $j' - 1$ in preparazione per la prossima fase.

Il terzo passaggio imposta correttamente j_{i+1} in quanto la sequenza iniziale delle estensioni dove vengono applicate le regole 1 o 2 deve terminare al punto in cui viene applicata per la prima volta la regola 3.

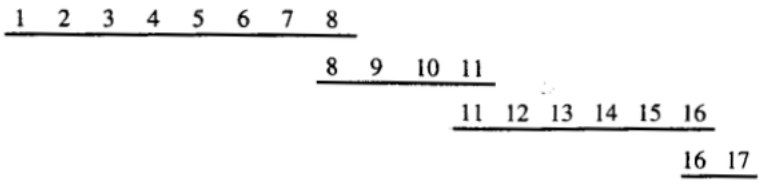

Figura 3.4: Illustrazione di un possibile andamento dell'algoritmo di Ukkonen. Ogni riga rappresenta una fase dell'algoritmo, ogni numero rappresenta una estensione esplicita. In questa illustrazione vi sono 4 fasi e 17 estensioni esplicite. Ogni fase condivide con la sua successiva al massimo un indice, per il quale la stessa estensione esplicita viene eseguita in entrambe le fasi. [1]

Il punto chiave dell'algoritmo appena presentato è che la fase $i + 2$ inizierà ad eseguire estensioni esplicite a partire dall'estensione j', dove j' era l'ultima estensione esplicita eseguita nella fase precedente. Di conseguenza, due fasi consecutive condividono al massimo un indice j' in corrispondenza del quale è eseguita una estensione esplicita. Quindi, la fase $i + 1$ termina tenendo traccia di dove termina la stringa $S[j'...i + 1]$, così che l'estensione ripetuta di j' nella fase $i + 2$ può essere eseguita nuovamente senza dover ripercorrere l'albero, attraversare suffix link o saltare nodi. Questo significa che la prima estensione esplicita di ogni fase è eseguita in tempo costante.

3.3.1 Analisi temporale

Utilizzando i suffix link ed implementando le euristiche l'algoritmo di Ukkonen costruisce gli alberi dei suffissi impliciti da I_1 a I_m in un tempo totale $O(m)$.

Man mano che l'algoritmo esegue le estensioni esplicite, si considera un indice k corrispondente all'estensione esplicita che si sta attualmente eseguendo. Durante tutta l'esecuzione dell'algoritmo, k non decresce mai, bensì rimane uguale solo durante il passaggio tra una fase ed un altra. Dal momento che ci sono solo m fasi, e che k è limitato da m, l'algoritmo esegue solamente $2m$ estensioni. E' stabilito che il tempo di esecuzione di una estensione esplicita è costante ma va sommato ad un tempo proporzionale al numero di nodi saltati durante il cammino percorso durante tale estensione. Tale percorso si può dividere in due fasi; la prima fase di risalita, che attraversa al massimo due nodi (raggiunge il primo nodo avente un suffix link che origina da esso al di sopra del punto di partenza e ne attraversa il relativo suffix link) e richiede dunque tempo costante, e la seconda fase di discesa in

cerca del punto in cui effettuare l'estensione. Questa seconda parte, utilizzando l'euristica skip/count è dipendente solamente dal numero di nodi, che è al massimo m. Dal momento, dunque, che la profondità massima dei nodi è m, e vengono effettuate $2m$ estensioni esplicite, il numero massimo di nodi attraversati durante il cammino percorso dall'intero algoritmo è limitato ad $O(m)$.

3.3.2 Costruzione dell'albero dei suffissi vero e proprio

L'albero dei suffissi implicito finale I_m può essere convertito in un vero albero dei suffissi in un tempo $O(m)$. Basta infatti aggiungere un simbolo terminale \$ al termine di S e lasciar proseguire l'algoritmo con questo nuovo carattere. L'effetto sarà che ora nessun suffisso sarà prefisso di un altro suffisso, quindi l'esecuzione dell'algoritmo di Ukkonen ha come risultato un albero dei suffissi implicito in cui ogni suffisso termina in una foglia, e si può dire dunque definito esplicitamente. E' inoltre necessario percorrere i rami foglia per sostituire gli indici e con il numero m. Questa operazione può essere eseguita in tempo $O(m)$ in quanto consiste in un attraversamento dell'albero. Apportate queste modifiche, l'albero risultante sarà un vero albero dei suffissi. Di conseguenza si può affermare che l'algoritmo di Ukkonen costruisce un albero dei suffissi esplicito per S, insieme a tutti i suoi suffix links in $O(m)$.

Conclusioni

In questa tesi si è analizzata la struttura dell'albero dei suffissi, una struttura dati che è fondamentale oggi in molte applicazioni legate all'analisi di stringhe. E' stata presentata una dettagliata descrizione di due degli algoritmi per costruire tali strutture dati in tempo lineare, gli algoritmi di Weiner e Ukkonen. Sulla base di tale descrizione, si possono mettere in mostra i pregi del più moderno ed efficiente algoritmo di Ukkonen, rispetto al più datato algoritmo di Weiner:

L'algoritmo di Ukkonen è eseguito on-line, cioè è in grado di processare l'input carattere dopo carattere senza la necessità di avere a disposizione l'intera stringa di input dall'inizio. Esso è inoltre più efficiente, sia in termini di tempo che in termini di spazio per l'esecuzione. E' concettualmente il più semplice ed elegante presentato, è l'algoritmo che permette di capire più facilmente le modalità di costruzione degli alberi dei suffissi.

L'algoritmo di Weiner non presenta particolari vantaggi, risulta essere più lento e meno efficiente dal punto di vista dello spazio di esecuzione (principalmente a causa dei vettori richiesti dall'algoritmo per ogni nodo). E' tuttavia un algoritmo di notevole importanza poiché è stato il primo del suo genere ad essere realizzato e costituisce pertanto un vero monumento storico nel campo dell'analisi delle stringhe.

Bibliografia

[1] Gusfield D., *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 28 Maggio 1997

[2] Goodrich M. T., Tamassia R., *Strutture dati e algoritmi in Java* , Zanichelli, 2007

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

[4] *http://www.cs.uku.fi/∼kilpelai/BSA05/lectures/*

[5] *http://www.wikipedia.org*

[6] *http://stackoverflow.com*