UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

Corso di Laurea in Informatica

STUDIO, ANALISI E OTTIMIZZAZIONE DI UN ALGORITMO IN C SULLA COSTRUZIONE DEL SUFFIX ARRAY

Relatore:
Ch.ma Prof.ssa
Rosalba Zizza

Candidato:
Francesco Maddaloni
Mat. 0512105733

ANNO ACCADEMICO 2022/2023

Dedicata al mio cagnolone e ora angelo, Chopper

CONTENTS

CHAPTER 1

INTRODUZIONE

1.1 Panoramica

Uno dei problemi più interessanti nel campo delle strutture dati e nell'elaborazione ed analisi di stringhe è l'ordinamento lessicografico dei suffissi di una parola, così da ricavare il suo array dei suffissi ordinati. Quest'ultimo corrisponde alla struttura che conserva la posizione iniziale dei suffissi della parola in ordine lessicografico crescente. Questa struttura ha utilizzo principale nel campo della bioinformatica e, in particolare, nello studio di sequenze genetiche le quali altro non sono che sequenze di caratteri (stringhe).

E' ben noto che il Suffix Array, denominato SA da ora in avanti, può essere calcolato in tempo lineare, sebbene nella pratica le prestazioni dei programmi utilizzati per calcolarlo non siano ancora stabili.

Negli anni sono stati tentati diversi approcci.
Nel 2013 e nel 2014 fu elaborata da alcuni ricercatori dell'Università di Palermo un'idea prettamente teorica sul problema, la quale fu in parte la base per la stesura dell'algoritmo in esame di questa tesi.
Nel 2016 ci fu un primo tentativo di scrittura di un algoritmo non ricorsivo che presentava diverse criticità, mentre nel 2021 è stato elaborato un primo

algoritmo che utilizzi le proprietà delle *parole di Lyndon* e la *Fattorizzazione Canonica di Lyndon*. Questo approccio si basa sull'algoritmo mai pubblicato di Yuta Mori.

L'applicazione di questa struttura è così vasta che spinge a trovare altri algoritmi che, nella pratica (e non asintoticamente), siano più efficienti.

L'obiettivo della tesi è quello di ottimizzare il lavoro nel quale è stata effettuata un'operazione di conversione di un programma da Python in C aggiungendo gestione della memoria (prima del tutto assente), modificando alcuni algoritmi e funzioni, debuggando l'esecuzione per avere un riferimento step by step sui passi del programma oltre che automatizzare l'esecuzione dello stesso tramite lettura da file `FASTA`.

Per affrontare il problema principale, innanzitutto, viene utilizzata la Fattorizzazione Inversa di Lyndon di una stringa S, denotata $ICFL(S)$. L'algoritmo per costruire SA usa $ICFL(S)$ sfruttando delle proprietà che verranno descritte nella sezione successiva.
Vengono inizialmente estratti e ordinati tutti i suffissi locali x di ogni fattore di $ICFL(S)$. Per ogni x distinto, viene associata una lista delle posizioni contenente le posizioni in cui è presente x in S. Successivamente questi suffissi verranno assemblati in speciali catene. Prima vengono fuse le catene le quali sono in relazione d'ordine dei prefissi e poi le rimanenti. Infine si concatenano tutte le liste per ottenere l'array dei suffissi SA della stringa S.

La strategia, prima di essere implementata in C, ha ricevuto delle modifiche dal punto di vista implementativo. Infatti l'operazione che avrà il compito di unire le catene in gruppi, che in seguito denoteremo con $common\_prefix\_merge$, è stata ottimizzata in una precedente tesi di laurea [15], per poter svolgere lo stesso compito in meno tempo. Questa modifica ha però generato alcuni problemi nella creazione delle catene (portando a perdere un riferimento intermedio completo delle stesse) non intaccando però l'integrità del risultato finale.

Il lavoro di questa tesi, come anticipato, ha avuto lo scopo di ottimizzare ed analizzare l'algoritmo sviluppato in C.
Per quanto riguarda l'ottimizzazione del codice, è stata liberata memoria all'interno dell'intera applicazione, esaminando tutti quegli oggetti instanziati che avevano terminato il loro utilizzo e appesantivano l'esecuzione del programma.
Successivamente è stata ottimizzata la funzione `hope_insert`.

Si è poi passati ad un lavoro di analisi del codice dell'applicazione.
Sono stati effettuati nuovamente tutti i test sulle prestazioni (prendendo in esame la versione aggiornata e non aggiornata del codice C), è stato scritto uno script per l'esecuzione automatica tramite lettura da file `FASTA` e sono state aggiunte stampe intermedie dei gruppi di catene.
É stato fatto un tentativo per la stampa delle catene (attraverso la funzione `printChain`) riportando tutte le osservazioni fatte durante tale lavoro.

Grazie all'ampio spettro di operazioni effettuate è stato possibile avere un'idea più chiara sulle possibili migliorie da apportare al codice, soprattutto nella gestione delle strutture dati.

Segue una breve introduzione delle nozioni generali per permettere di capire al meglio i successivi concetti di CFL e $ICFL$.

1.2 Cenni preliminari

In questa sezione andremo ad introdurre alcune nozioni per poter comprendere i successivi capitoli. [4, 7, 10]

Di seguito, Σ è l'alfabeto finito non vuoto e assumiamo sia ordinato.

Una parola (o stringa) S di lunghezza n sull'alfabeto Σ è una sequenza infinita di n caratteri di Σ e λ denota la stringa vuota.

Sia i, $j \in \{1, \ldots, n.\}$. Indichiamo con $S[i]$ l'i-esimo carattere della parola S, e con $S[i, \ldots, j]$ indichiamo la sotto-stringa (o fattore) di S che inizia alla i-esimo carattere di S e termina al j-esimo carattere di S.

Se $i = 1$ (risp. $j = n$), $S[1, \ldots, j]$ (risp. $S[i, \ldots, n]$) è un prefisso di S (risp. un suffisso di S) e quando $i{=}\ n$ (risp. $j \neq n$) il prefisso (risp. suffisso) è proprio.
Denotiamo con S_i il suffisso di S che inizia alla i-esima posizione, ossia $S_i = S[i, \ldots, n]$. Inoltre, diremo che due stringe x e y sono in *relazione di prefisso* se x è un prefisso di y oppure y è un prefisso di x.

Data una stringa non vuota S, un *bordo* di S è una stringa non vuota la quale è sia un prefisso proprio che un suffisso di S. Il più lungo *bordo* è anche chiamato il *bordo* di S. Una stringa non vuota S è una *primitiva* se $S = x^k$ e $k{=}1$. Una stringa senza *bordo* è *primitiva*.

Date $S, S'{\in}\ \Sigma^*$, indichiamo con $S \prec S'$, (risp. $S \preceq S'$) se S è lessicograficamente più piccola rispetto a S' (risp. più piccola o uguale a S). Inoltre per due stringhe non vuote S, S' , scriviamo $S << S'$ se $S \prec S'$ e in aggiunta S non è un $prefisso\ proprio$ di S.

Ricordiamo che una fattorizzazione di una stringa S è una sequenza $F(S){=}\ (f_1, f_2, \ldots, f_k)$ di fattori tale che $S = f_1 f_2, \cdots f_k$.

Una *parola di Lyndon* è definita come una parola tale che è minore strettamente, in ordine lessicografico, di tutte le sue rotazioni, o in modo analogo, una parola tale che è minore strettamente di ogni suo suffisso proprio non vuoto.[7]

La *Fattorizzazione Canonica di Lyndon* $CFL(S) = (f_1, f_2, \ldots, f_k)$ è una speciale fattorizzazione di S tale che $(f_1 \geqslant f_2 \cdots \geqslant f_k)$ e ogni f_i é una

parola di Lyndon.

Una *parola inversa* di *Lyndon* è definita come una parola tale che è maggiore strettamente, in ordine lessicografico, di ogni suo suffisso proprio non vuoto.

La *Fattorizzazione Canonica Inversa di Lyndon* $ICFL(S) = (f_1, f_2, \ldots, f_k)$ è stata introdotta come una speciale fattorizzazione di S tale che $(f_1 << f_2 << \cdots << f_k)$ e ogni f_i'e una *parola inversa di Lyndon*.

Per esempio, se $\Sigma = \{a, c, g, t\}$, con $a < c < g < t$, abbiamo che *cac* e *tcaccgc* sono *parole inverse di Lyndon*.

Sia $S = gcatcaccgctctacagaac$. Abbiamo che $ICFL(S) = (gca, tcaccgc, tctacagaac)$.

A differenza delle *parole di Lyndon*, le *parole inverse di Lyndon* potrebbero avere un *bordo*.
Importante notare che $ICFL(S)$ conserva le principali proprietà di $CFL(S)$, può essere calcolata in tempo lineare ed è determinata in modo univoco.
In aggiunta $ICFL(S)$ ha ulteriori interessanti proprietà:

- (i) esiste un limite superiore della lunghezza del più lungo prefisso comune di due fattori di S a partire da posizioni diverse di S.
- (ii) esiste una relazione tra l'ordinamento dei suffissi globali, ossia i suffissi di S, e l'ordinamento dei suffissi locali, ossia i suffissi dei fattori di $ICFL(S)$.

Dati una stringa S e due fattori x, y di S, indichiamo con $lcp(x, y)$ il più grande prefisso comune di x, y e poniamo $LCP(x, y) = |lcp(x, y)|$.

Sia $ICFL(S) = (f_1, f_2, \ldots, f_k)$ e indichiamo con $M = max\{(|f_i f_{i+_1}| \mid 1 \leqslant i < k)\}$.

Proposizione. Sia $S \in \Sigma^*$ una stringa non vuota che non è una *parola inversa di Lyndon* e sia $ICFL(S) = (f_1, f_2, \ldots, f_k)$. Per ogni fattore proprio non vuoto u, v di S, abbiamo $LCP(u, v) = |lcp(u, v)| \leqslant M$.

Sia x un fattore non vuoto di S e siano $first(x)$ e $last(x)$ rispettivamente le posizioni del primo e dell' ultimo carattere di x in S.
Un *suffisso locale* di S è un suffisso di un fattore x di S, nello specifico $\overline{S}_{i,x} = S[i, last(x)]$ indica il *suffisso locale* di S alla posizione i rispetto a x, con $first(x) \leqslant i \leqslant last(x)$. Il corrispondente *suffisso globale* di $\overline{S}_{i,x}$ in S alla posizione i è S_i.

Lemma. Sia $S \in \Sigma^+$ una stringa la quale non è una *parola inversa di Lyndon* e sia $ICFL(S) = (f_1, f_2, \ldots, f_k)$. Sia $x = (f_i f_{i+1} \cdots f_h)$ con $1 \leqslant i < h \leqslant k$.

Assumiamo che $S_{j_1,x} \prec S_{j_2,x}$, dove $first(x) \leqslant j_1 \leqslant last(x)$, e $first(x) \leqslant j_2 \leqslant last(x)$, $j_1 \neq j_2$.
Se $S_{j_1,x}$ è un *prefisso proprio* di $S_{j_2,x}$ e $h < k$ allora $S_{j_2} \prec S_{j_1}$, altrimenti $S_{j_1} \prec S_{j_2}$.

Di seguito specializziamo la definizione di *suffisso locale*, riferendoci all'unico fattore di $ICFL(S)$, il quale contiene la posizione i, ossia, elimineremo il pedice x dato che x è sempre l'unico fattore f_j di $ICFL(S)$ tale che $first(f_j) \leqslant last(f_j)$.

1.3 Stato dell'arte

La ricerca di una tecnica per la costruzione del Suffix Array è stata approfondita a più riprese negli ultimi anni. Nel 2013 e nel 2014 fu elaborata un' idea prettamente teorica, non sviluppata nella pratica, dai ricercatori

Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, e Marinella Sciortino dell'Università degli studi di Palermo. La loro ricerca proponeva una struttura simile nella divisione dei suffissi di una parola in gruppi, sfruttando le proprietà delle parole di Lyndon.

Nel 2016 Uwe Baier riprese la tematica sviluppando un algoritmo non ricorsivo nominato $GSACA$, il quale utilizza tecniche prettamente differenti ma espone criticità nelle performance a causa di problemi di caching dovuti all'utilizzo intenso di puntatori e metodi greedy.

L'algoritmo pratico migliore, è quello ideato, ma mai pubblicato, di Yuta Mori, chiamato `DivSufSort` il quale propone un nuovo approccio utilizzando le proprietà delle parole di Lyndon.
Uno step importante avviene nel 2021 quando Nico Bertram, Jonas Ellert e Johannes Fischer elaborano un algoritmo basato sul sopra citato `DivSufSort` che utilizza le proprietà sopra citate e sfrutta la *Fattorizzazione Canonica di Lyndon* CFL. Queste proprietà sono state usate per accelerare l'ordinamento dei suffissi.

L'idea generale è separare i suffissi in gruppi, ordinare i suffissi nei gruppi per poi usarli per indurre l'ordinamento dei suffissi dell' intera parola. L'approccio di dividere i suffissi in gruppi è stato proposto, usando la $CFL(S)$. Questa fattorizzazione può essere calcolata in tempo lineare.

Analizziamo la seguente Proprietà di Compatibilità sopra dimostrata: se ordiniamo i suffissi dei fattori (tali suffissi vengono chiamati suffissi locali), presi due *suffissi locali* x e y, il loro ordine viene preservato quando estendiamo x e y ai *suffissi globali*, ossia i relativi *suffissi globali* che hanno come prefisso rispettivamente x e y e che iniziano nella stessa posizione di x e y. Il nocciolo dell'approccio è trovare un modo efficiente per fondere i vari gruppi ed aprire la strada alla ricerca di tecniche alternative per effettuare fusioni efficienti dei raggruppamenti.

Tale lavoro, unito allo studio della *Fattorizzazione Inversa di Lyndon* $ICFL$ e alle sue proprietà per accelerare ulteriormente i tempi, ha inspirato lo studio e la stesura dell'algoritmo oggetto della tesi.
Nel seguito vedremo l'utilizzo della $ICFL(S)$ proprio per questo aspetto.

Un' osservazione importante riguarda le prestazioni generali della costruzione del Suffix Array: la velocità del programma è particolarmente legata all'alfabeto preso in considerazione dove, su alfabeti più piccoli, il programma è meno prestante rispetto ad alfabeti più grandi per via delle proprietà offerte dalle *parole di Lyndon* e dalla $ICFL$. Infatti su alfabeti più piccoli il numero di fattori individuati è maggiore, cosa che comporta un incremento sostanzioso delle operazioni di fusione.
Inoltre sono presenti situazioni nella pratica, ove programmi di complessità asintotica $O(n)$ risultano più lenti di altri di complessità asintotica $O(nlogn)$.

CHAPTER 2

IDEA ALGORITMICA DI BASE

Come anticipato, per una qualsiasi parola S, la ricerca suggerisce di utilizzare la proprietà di Compatibilità dei suffissi locali di $CFL(S)$ per effettuare l'ordinamento dei suffissi globali di S.

L'algoritmo in esame, utilizza quanto elaborato negli ultimi anni,come descritto nello state dell'arte, scegliendo di utilizzare la *fattorizzazione inversa di Lyndon* $ICFL$ per migliorare ulteriormente le operazioni di fusione grazie alle sue proprietà.
Partendo dall'ordinamento dei suffissi locali dei diversi fattori, si cerca un modo efficiente per poterli utilizzare e fonderli ottenendo l'ordinamento dei suffissi globali.

Proponiamo una strategia che divide la costruzione del Suffix Array in due fasi.
La prima fase impiega la $ICFL$ per ottenere e, successivamente, ordinare i suffissi locali distinti di $ICFL$. In seguito, per ciascun suffisso locale distinto, viene composta la sua lista delle posizioni che sono tracciate sulla parola S e seguono l'ordine lessicografico.

Il risultato dello step 1, in conclusione, è un dizionario che ha come chiavi i suffissi locali distinti ordinati lessicograficamente. Alle chiavi sono associate le liste delle posizioni del suffisso stesso nella stringa di partenza.

La seconda fase si concentra nell'unire le liste delle posizioni ottenute durante la prima fase, creando così delle catene e in seguito gruppi di catene. Questa operazione viene effettuata sulla base dei prefissi creando catene con prefisso comune. Successivamente queste catene verranno fuse in gruppi di catene, i quali saranno poi uniti a formare il suffix array SA finale, grazie ad un ordinamento indotto.
Descriveremo ora in modo esteso tali procedimenti.

2.1 Step 1: Creazione delle liste delle posizioni

Data un stringa S e la sua fattorizzazione $ICFL(S) = (f_1, \ldots, f_k)$, vengono calcolati ed ordinati tutti i suoi suffissi locali distinti. In seguito per costruire le loro liste delle posizioni procediamo come segue.

- Analizziamo ogni fattore partendo da f_{k-1} (il secondo fattore più grande) fino a f_1 ed infine consideriamo f_k(il fattore maggiore). Questo metodo è utilizzato per preservare l'ordine all'interno delle liste delle posizioni e avrà utilizzo anche negli step successivi. Si osservi che i suffissi locali di f_k sono anche suffissi globali di S.

- Sia f_j un fattore e, per ogni posizione (da sinistra a destra) sia $\overline{S}_i$ un suffisso locale di f_j alla posizione i . Inseriamo quindi i in testa a $[\overline{S}_i]$`-positions` (inizialmente vuota). Di conseguenza, ogni $[\overline{S}_i]$`-positions` può essere computata in tempo lineare non dovendo eseguire alcun confronto.

Esempio. Sia $S = aaabcaabcadcaabca$ e sia $ICFL(S) = \{aaa, b, caabca, dcaabca\}$ I suoi fattori locali distinti sono a, aa, aaa, $aabca$, $abca$, b, bca, ca, $caabca$, $dcaabca$.

Inizialmente, [s]-positions = [] per ogni suffisso locale distinto $s \in \{a$, aa, aaa, $aabca$, $abca$, b, bca, ca, $caabca$, $dcaabca\}$.
Ricordiamo che analizziamo i fattori di $ICFL(S)$ partendo da $caabca$.

Ogni suffisso locale s di $caabca$ è estratto da destra verso sinistra e per ciascuno di loro aggiorniamo la corrispondente lista delle posizioni inserendo la posizione alla testa di [s] -positions. Quindi, estraiamo prima il suffisso locale a alla posizione 9, e quindi [a]-positions = [9].
Poi, estraiamo il suffisso locale ca alla posizione 8, e così [ca]-positions = [8], e così via, per bca, $abca$, $aabca$, e $caabca$.

Finita l'analisi del fattore $caabca$ abbiamo le seguenti liste delle posizioni: [a]-positions = [9], [ca]-positions= [8], [bca]-positions = [7], [$abca$]-positions = [6], [$aabca$]-positions = [5], and [$caabca$]-positions = [4].
Le altre liste sono vuote.

Procediamo con il fattore precedente b. In questo caso abbiamo un solo suffisso, b e quindi settiamo [b]-positions = [3].

Procediamo ancora con il fattore precedente aaa. Estraiamo il suffisso locale a alla posizione 2 e quindi [a]-positions = $[2, 9]$ (9 era già contenuto nella lista mentre 2 è inserito in testa), e così via, per i suffissi rimanenti aa e aaa.

Una volta che l'analisi di aaa è terminata, avremo [a]-positions = $[2, 9]$, [aa]-positions= [1], [aaa]-positions = [0], [ca]-positions = [8], [bca]-positions = [7], [$abca$]-positions = [6], [$aabca$]-positions = [5], [$caabca$]-positions = [4], e [b]-positions = [3]

In conclusione, nello Step 1 l'algoritmo prende in input una stringa S, e ritorna `position-lists` di S, un dizionario dove le chiavi sono i suffissi locali distinti di $ICFL(S)$ in ordine lessicografico. per ogni chiave, il corrispondente valore è $[Chiave]$-`positions` cioè la posizione di partenza in S della chiave del suffisso locale preservando l'ordine dei suffissi globali.

Local Suffix x	**[x]-position-list]**
a	$[16, 2, 9]$
aa	$[1]$
aaa	$[0]$
$aabca$	$[12, 5]$
$abca$	$[13, 6]$
b	$[3]$
bca	$[14, 7]$
ca	$[15, 8]$
$caabca$	$[11, 4]$
$dcaabca$	$[10]$

Table 2.1: Tabella del dizionario dei suffissi locali della stringa di esempio

2.2 Step 2: Fusione delle liste delle posizioni

Siano $[x]$-`positions` and $[y]$-`positions` liste delle posizioni, costruite allo Step 1, per due suffissi locali x, y.

L'obiettivo dello Step 2 è di costruire una nuova lista contenente gli elementi di $[x]$-`positions` e $[y]$-`positons` in modo tale che i corrispondenti suffissi globali degli elementi siano ordinati lessicograficamente.

Questa nuova lista è una lista delle posizioni di suffissi globali che inizia nella posizione di $[x]$-`positions` e di $[y]$-`positions`.
Ciò è fatto in modo tale che le posizioni di $[x]$-`positions` (e di $[y]$-`positions`) mantengano lo stesso ordine relativo all'interno della nuova lista.
Successivamente, iteriamo il procedimento come descritto più avanti.

2.2.1 Catene ordinate di prefissi e le loro liste delle posizioni

Partendo da `position-list`$_S$ computata nello step 1, la nostra strategia sfrutta la loro fusione per costruire l'array dei suffissi di S. A tale scopo, definiamo ora una catena di suffissi.

Definizione Una lista di suffissi locali distinti $C = [x_1, \ldots, x_n]$ è una catena ordinata di prefissi se x_i è un prefisso di x_{i+1} per ogni $i = 1, \ldots, n-1$, ed è massimale, non esiste alcun suffisso locale y che è un prefisso di x_i o x_i è un prefisso di y, con $i = 1, \ldots, n-1$, che non è contenuto in C.

Esempio Abbiamo che $[a, aabca]$ non è una catena ordinata di prefissi, essendo aa un prefisso di $aabca$. Invece possiamo verificare che $[a, aa, aabca]$ è una catena ordinata di prefissi.

Nel nostro esempio abbiamo le seguenti catene: $C_1 = [a,\ aa,\ aaa]$, $C_2 = [a,\ aa,\ aabca]$, $C_3 = [a,\ abca]$, $C_4 = [b,\ bca]$, $C_5 = [ca,\ caabca]$, e

$C_6 = [dcaabca]$.

La strategia computa C_i-`positions` per $i = 1, \ldots, m$ come segue: sia $C_i = [x_1, \ldots, x_n]$ una catena ordinata di prefissi.

Per ogni $j = 1, \ldots, n-1$, si esegue `in-prefix-merge`(X_j-*positions*, $[x_{j+1}]$-*positions*), dove X_j è la lista contentente gli elementi distinti di $[x_1]$-*positions*, $[x_2]$-*positions*, ..., $[x_j]$-*positions*.

Alla fine, settiamo, C_i-*positions* = `in-prefix-merge`(X_{n-1}-*positions*], $[x_n]$-*positions*]).

Più precisamente, data una catena $C = [x_1, \ldots, x_n]$, allora C-`positions` è computata eseguendo prima `in-prefix-merge`($[x_1]$-`positions`, $[x_2]$-`positions`), poi `in-prefix-merge`(X_2-`positions`, $[x_3]$-`positions`) dove X_2 è la lista contente gli elementi distinti di x_1 e x_2, e così via. Ad ogni passo vengono fuse due liste, dove la prima è il risultato della precedente operazione di fusione.

Nel nostro esempio, sia $C_1 : [a, aa, aaa]$ la prima catena. Per computare [*a*, *aa*, *aaa*]-`positions`, l'algoritmo esegue prima `in-prefix-merge`([*a*]-`positions`], [*aa*]-`positions`]) ottenendo[*a*, *aa*]-`positions` = [16, 1, 2, 9]. Poi `in-prefix-merge`([*a*, *aa*]-`positions`], [*aaa*]-`positions`]) ottenendo [*a*, *aa*, *aa*]-`positions` = [16, 0, 1, 2, 9].

Ora, sia $C_2 : [a, aa, aabca]$ la seconda catena. Eseguiamo prima `in-prefix-merge`([*a*]-`positions`, [*aa*]-`positions`), dove [*a*]-`positions` = $[16, 2, 9]$ e [*aa*]-`positions` = [1].
Il risultato è [*a*, *aa*]-`positions` = $[16, 1, 2, 9]$.

E, per concludere, eseguiamo `in-prefix-merge`($[a, aa]$-`positions`, [*aabca*]-`positions`), dove [*aabca*]-`positions` = $[12, 5]$.

Il risultato è $[a, aa, aabca]$-`positions` = $[16, 12, 1, 5, 2, 9]$ C_2-`positions`.

2.2.2 Common-prefix-merge

Date le catene ordinate di prefissi $C_1, \ldots, C_m$, e le loro C_i-`positions` per $i = 1, \ldots, m$, la strategia ora le separa in gruppi di catene in questo modo: se due catene condividono qualche suffisso, allora appartengono allo stesso gruppo di catene, altrimenti appartengono a gruppi diversi.

Nel nostro esempio, [C_2, C_1, C_3] non è un gruppo di catene. Abbiamo i seguenti gruppi di catene:
$G_1 : [C_1, C_2, C_3] = [[a,\ aa,\ aaa],\ [a, aa, aabca], [a, abca]]$, $G_2 : [C_4] = [[b,$ $bca]]$, $G_3 : [C_5] = [[ca,\ caabca]]$, $G_4 : [C_6] = [[dcaabca]]$.

Dati i gruppi di catene $G_1, \ldots, G_s$, tali che $G_i = [C_{in_1}, \ldots, C_{in_i}]$ per $i = 1, \ldots, s$, la strategia computa X_i-positions per ogni $i = 1, \ldots, s$ eseguendo `common-prefix-merge`(C_{in_1}-`positions`, ..., C_{in_i}-`positions`).

Nel nostro esempio, consideriamo $G_1 = [C_1, C_2, C_3]$ dove $C_1 = [a, aa, aaa]$, $C_2 = [a, aa, aabca]$, e $C_3 = [a, abca]$. Inoltre, C_1-positions = [16, 0, 1, 2, 9], C_2-positions = [16, 12, 1, 5, 2, 9], and C_3-positions = [16, 13, 2, 6, 9]. Eseguiamo quindi,
`common-prefix-merge`(C_1-`positions`, C_2-`positions`, C_3-*positions*) = [16, 0, 12, 1, 5, 13, 2, 6, 9].

2.2.3 Concatenation-prefix-merge

Per ogni $i \in [x]$-`positions` e $j \in [y]$-`positions`, $\overline{S}_i < \overline{S}_j$ e quindi $\overline{S}_i$ non è prefisso di $\overline{S}_j$.

In questo caso [z]-`positions` = [x]-`positions` · [y]-`positions`, possiamo semplicemente concatenare [y]-`positions` ad [x]-`positions`.

Infatti, siano $k_1, k_2 \in \{0, \ldots, |[z]\texttt{-positions}| - 1\}$ e denotiamo $q_1 = [z]\texttt{-positions}[k_1]$ e $q_2 = [z]\texttt{-positions}[k_2]$. Se $k_1 < k_2$, allora $S_{q_1} < S_{q_2}$.
Infatti, entrambi q_1, q_2 appartengono a $[x]$`-positions` (o a $[y]$`-positions`), o a $q_1 \in [x]$`-positions`, $q_2 \in [y]$`-positions`.
Nel primo caso, se $k_1 < k_2$, abbiamo $S_{q_1} < S_{q_2}$.

Nel secondo caso, essendo $\overline{S}_{q_1} < \overline{S}_{q_2}$ ed essendo che $\overline{S}_{q_1}$ non è prefisso di $\overline{S}_{q_2}$, chiaramente implica che $S_{q_1} < S_{q_2}$.

Nel nostro esempio, consideriamo [*a*, *aa*, *aaa*, *aabca*, *abca*]`-positions` = [16, 0, 12, 1, 5, 13, 2, 6, 9] e [*b*, *bca*]`-positions` = [14, 3, 7].
Come possiamo notare, queste liste ricadono in questo caso.

Quindi, [*a*, *aa*, *aaa*, *aabca*, *abca*, *b*, *bca*]`-positions` = [*a*, *aa*, *aaa*, *aabca*, *abca*]`-positions` · [*b*, *bca*]`-positions` = [16, 0, 12, 1, 5, 13, 2, 6, 9] · [14, 3, 7] = [16, 0, 12, 1, 5, 13, 2, 6, 9, 14, 3, 7].

Quindi per concludere, dati i gruppi di catene $G_1, \ldots, G_s$ and $X_i = G_i$-*positions* per ogni $i = 1, \ldots, s$, eseguiamo `concatenation_prefix_merge`$(X_1, \ldots, X_s)$, il quale è effettivamente il suffix array finale. In conclusione, viene ritornato $\texttt{SA}_S$.

2.2.4 Una visione di insieme utilizzando alberi binari

É possibile rappresentare visivamente quanto descritto fino ad ora, attraverso l'utilizzo di alberi (non utilizzati all'interno dell'implementazione) denominati **alberi di prefissi**.

Immaginiamo un albero con ogni nodo che ha come chiave un fattore locale distinto e come valore la sua lista delle posizioni. Ogni albero rappresenta un gruppo, con le catene formatesi con delle visite in profondità dell'albero, partendo dalla radice.
Possiamo quindi pensare a tutti i fattori locali distinti organizzati in una foresta di alberi di prefissi.

In altre parole, rappresentiamo il dizionario $\texttt{positions}_S$ come una foresta di alberi di prefissi, tale che se due suffissi locali condividono un prefisso comune, essi appartengono allo stesso albero. La strategia computa C_i-`positions` per $i = 1, \ldots, m$ utilizzando una visita in profondità e la in_prefix_merge.

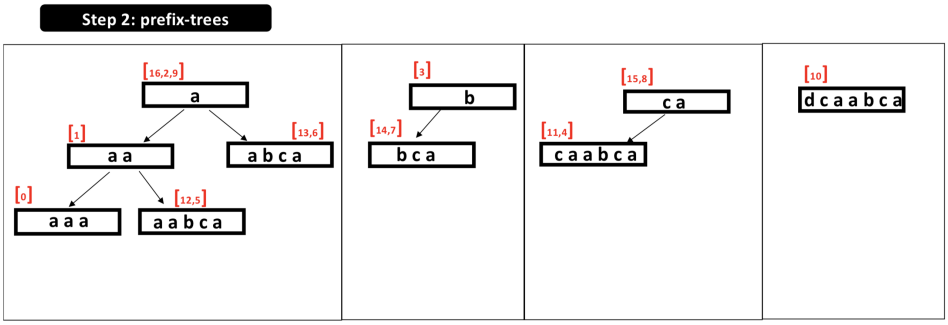

Figure 2.1: Esempio di albero di prefissi per $ICFL(S) = (aaa, b, caabca, dcaabca)$

Nel nostro esempio, abbiamo le seguenti catene ordinate di prefissi, elencate in ordine lessicografico:
$C_1 : [a, aa, aaa]$, $C_2 : [a, aa, aabca]$, $C_3 : [a, abca]$, $C_4 : [b, bca]$, $C_5 : [ca, caabca]$, $C_6 : [dcaabca]$.

Sia $C_i = [x_1, \ldots, x_n]$ una catena ordinata di prefissi, $i \in \{1, \ldots, m\}$, la sequenza di suffissi che leggiamo in un percorso dalla radice x_1 alla foglia x_n. Computiamo C_i-`positions` come segue.

Iniziamo con $[x_1]$-`positions` e $[x_2]$-`positions`, in modo incrementale dalla radice alle foglie. Sia x_1 l'etichetta di una radice di un albero di prefissi e, x_2 un'etichetta di un figlio della radice (partendo da quello più a sinistra).

Ricordiamo che ogni x_j è un prefisso di x_{j+1}, $j \in \{1, \ldots, n-1\}$. In particolare, x_1 è un prefisso di x_2 e, per definizione, $[x_1]$-`positions` and $[x_2]$-`positions` sono ordinati in maniera tale che i suffissi globali associati alle liste delle posizioni siano ordinati lessicograficamente, quindi otteniamo $[x_1, x_2]$-`positions`.

Osservazione su di uno step generale di questa operazione quando uniamo un generico $[x]$-`positions` e $[y]$-`positions`:
x, y possono essere sia entrambi suffissi locali, (con x prefisso di y), sia con x una catena ordinata di prefissi e y un suffisso locale, tale che ogni elemento in x è un prefisso di y. Nel secondo caso, x è la sequenza di nodi in un percorso nell'albero di prefissi dalla radice al nodo padre di y.

CHAPTER 3

ANALISI DELLE STRUTTURE DATI E DELL'ALGORITMO IMPLEMENTATO IN C

In questo capitolo analizzeremo le strutture dati utilizzate nel codice C dell'applicazione e vedremo in dettaglio il funzionamento della stessa, soffermandosi in particolare sulle differenze tra le operazioni effettuate nel codice e l'algoritmo illustrato nel capitolo precedente.

3.1 Strutture dati: Libreria Utils

- **array_int_t**: E' una struttura dati che simula un array dinamico di interi. E' utilizzata principalmente per conservare indici all'interno dell'applicazione i quali servono per gestire le liste delle posizioni.
  Essa presenta metodi che consentono: l'aggiunta di un intero all'array, la concatenazione di array, il controllo della presenza di un elemento nell'array.

- **positions_lists**: E' una struttura che funge da dizionario e ha come

chiave un suffisso e, come valore ad esso associato, una lista delle posizioni. Rispettivamente questi campi corrispondono ad una stringa e ad un `array_int_t`.

Sono presenti metodi che permettono: l'aggiunta di una chiave, l'aggiunta di una posizione in una lista delle posizioni legata ad una determinata chiave, l'ottenimento di una lista data una chiave, il controllo della presenza di una chiave all'interno del dizionario.
Segue una descrizione approfondita dei metodi sopra riportati:

- **contains_suffix:** prende in input una `positions_list` e una stringa. La funzione sfrutta l'ordinamento dei suffissi presente nel dizionario per effettuare una ricerca binaria della chiave.

  Il metodo presenta un **funzionamento inusuale** rispetto ai normali algoritmi di ricerca. Esso infatti, restituisce un numero negativo nel caso in cui la chiave sia presente all'interno del dizionario, mentre restituisce la posizione in cui tale chiave andrebbe inserita, nel caso in cui quest'ultima non fosse già presente.

  Questo comportamento è funzionale all'utilizzo del metodo che viene richiamato principalmente nella funzione successivamente descritta, la `add_suffix`.

- **add_suffix:** prende in input una `positions_lists` e una stringa. Verifica se la chiave è già presente nel dizionario, invocando il metodo `contains_suffix` che, in caso essa non sia effettivamente presente, restituisce la posizione nella quale la chiave deve essere inserita per mantenere l'ordine lessicografico, altrimenti termina l'esecuzione della funzione.

  Nel caso i metodi di `realloc` dovessero fallire, vengono reinstanziati sia l'array dei suffissi che le liste delle posizioni con un allocazione maggiore di memoria per permettere l'aggiunta della nuova chiave.

  - **add_position:** prende in input un dizionario `positions_lists`, una chiave (suffisso) e una posizione (intero) . Il metodo aggiunge alla lista delle posizioni , relativa ad una chiave(suffisso), la posizione passata nel caso in cui non sia presente nella lista.

- **array_strings:** E' una struttura che simula un array dinamico di stringhe. La struttura è usata principalmente per la definizione delle catene ordinate, che corrispondono a sequenze di suffissi (array di stringhe), oltre che alla definizione dei gruppi di catene, che corrisponde ad uno dei miglioramenti svolti per questa tesi. La struttura presenta un metodo per l'aggiunta di una stringa.

3.2 Descrizione in dettaglio dell'implementazione principale: ICFL2SA.c

In questa sezione analizzeremo il cuore dell'applicazione, riportando nel dettaglio il comportamento dell'algoritmo attraverso la descrizione dei metodi implementati.

In breve, l'applicazione prende in input una parola, ne computa la scomposizione in fattori, esegue lo step 1 calcolando il dizionario dei suffissi locali distinti, come descritto nel capitolo precedente.

Successivamente lo step 2 procede in modo differente rispetto a quanto descritto.
Esso infatti prima di tutto computa le catene lavorando esclusivamente sui fattori e, ignorando temporaneamente le liste delle posizioni.

Al momento della creazione dei gruppi, invece, si lavora principalmente sugli indici delle posizioni, effettuando prima le operazioni di $common\_prefix\_merge$ sui gruppi computati e, solo dopo, viene effettuata la in_prefix_merge sulle catene.
Segue una descrizione in dettaglio dei principali metodi dell'applicazione:

- **sorting_suffixes_via_icfl:** Questo metodo racchiude l'intera strategia. Esso si occupa di ricevere una parola in input e di richiamare le varie fasi per poi ottenere e restituire il Suffix Array finale della parola data. Vengono definiti gli oggetti che serviranno durante l'intera esecuzione. Oggetto di maggiore importanza è il dizionario dei suffissi, `suffix_dict` definito con il tipo `position_list_t`.

- **step_1:** Il metodo racchiude interamente la fase 1 descritta nel capitolo precedente.
  In questa fase vengono creati tutti gli oggetti necessari per poter realizzare la fase 2.

  Viene inizialmente effettuata la fattorizzazione della parola data. Successivamente calcola l'array `mask_index` che indica la posizione iniziale all'interno della parola di ogni fattore appena calcolato; `mask_index` verrà utilizzato nei passi successivi per tener conto dei limiti di un fattore. Successivamente viene costruito il dizionario dei suffissi locali distinti, `suffix_dict` e, attraverso i fattori, viene popolata la lista delle posizioni di ogni suffisso locale distinto.

  Una nota importante è che la testa della lista delle posizioni si trova alla fine dell'array. Infine viene calcolato l'array `lcp_sorted_suffixes` nel quale, ogni posizione fa riferimento alla relativa posizione dei suffissi locali distinti ordinati lessicograficamente. Tale posizione corrisponde anche alla lunghezza del più lungo prefisso comune tra un suffisso e quello precedente. Sarà utile alla creazione delle catene di suffissi.

- **step_2**: Questo metodo comprende la fase 2 descritta nel capitolo precedente. Esso presenta numerose modifiche nell'implementazione rispetto a quanto descritto nella strategia. Con l'uso dell'array `lcp_sorted_suffixes` vengono create, da subito, le catene dei suffissi con il metodo `speed_structure` e conservate all'interno dell'`array_strings speed_keys_array`.

  In questo punto, la modifica principale rispetto alla strategia esposta nei capitoli precedenti, è che **le catene vengono costruite solo attraverso i suffissi**, non tenendo conto delle lista delle posizioni degli stessi che verranno direttamente uniti ed ordinati durante la creazione dei gruppi. Cio avviene eseguendo contemporaneamente in_prefix_merge e $common\_prefix\_merge$.

  Successivamente si passa alla creazione dei gruppi. Vengono scandite le catene attraverso un ciclo per effettuare direttamente le varie operazioni di fusione sopra citate e descritte nel capitolo precedente. Si inzia, quindi, scandendo le catene in `speed_keys_array` e, utilizzando l'array `lcp_sorted_suffixes` è possibile capire se la catena che si sta analizzando in quella iterazione è parte o meno del gruppo delle catene analizzato e computato fino a quel momento.

  Nel dettaglio, se all'iterazione i stiamo scandendo la catena situata in `speed_keys_array[i]` e la relativa posizione in `lcp_sorted_suffixes` è **diversa da 0**, allora l'esecuzione procede fino alla prossima iterazione, altrimenti vuol dire che è iniziato un nuovo gruppo.

  Arrivati a questo punto viene eseguita la funzione `hope_merge` che corrisponde alla $common\_prefix\_merge$ e, diversamente da come descritto, per i gruppi formatisi viene già effettuata l'operazione di $concatenation\_prefix\_merge$. Solo successivamente ai controlli su `lcp_sorted_suffixes` e le relative fusioni effettuate con `hope_merge`, viene eseguita la in_prefix_merge alla catena considerata, utlizzando

il metodo `merge_hash_list_prefix` sfruttando il `suffix_dict` ottenuto alla fine dello step 1.

Durante questo ciclo, viene computato man mano il *suffix array* finale, sfruttando un oggetto di appoggio denominato `SA_Group`. Esso contiene la lista delle posizioni dei gruppi o della catena singola computata fino a quel momento.

In definitiva, mentre la **creazione delle catene viene basata sui soli fattori**, non tenendo conto delle liste delle posizioni che verranno fuse solo in seguito, la formazione dei gruppi è invece computata utilizzando principalmente le liste delle posizioni dei singoli fattori delle catene analizzate. Si ottiene dunque nel codice, un **riferimento ai fattori per le catene** in `speed_keys_array` e **un riferimento alle posizioni ordinate lessicograficamente per i fattori dei gruppi** in `SA_Group`.

- **hope_merge:** Prende in input un array di `array_int_t` denominato `group` e la sua lunghezza. Il metodo simula l'operazione descritta nel capitolo precedente: la $common\_prefix\_merge$.

  Come già detto nella descrizione di `step_2`, il metodo è implementato diversamente rispetto a come è descritto nel capitolo precedente. L'algoritmo prevede di comporre prima una lista, denominata `main_lista`.
  Quest'ultima viene prodotta concatenando tutte le liste delle posizioni delle catene analizzate, tenendo traccia **degli indici dove inizia ogni catena** nella lista appena creata.

  Da `main_lista`, vengono ricavati tutti gli elementi distinti che verranno inseriti in un array denominato `key_lista`, nell'ordine in cui sono stati valutati.

Per ogni *key* viene poi creata una lista di indici che indicano la posizione di *key* in `main_lista` in un dizionario denominato `group_dict`.

Un'importante osservazione: ogni *key* è contenuta al massimo una volta in una catena, quindi se `group_dict` contiene più di un elemento, allora vuol dire che *key* è comune a più catene. Vengono quindi ciclate le chiavi prese in ordine di presenza in `main_lista` **attraverso** `key_lista`: se una chiave è stata già valutata, (controllo eseguito attraverso un array parallelo a `main_lista`) si passa alla successiva, altrimenti si esegue la funzione `hope_insert` e successivamente il suo **risultato parziale**, presente in `SA_Temp`, si concatena alla lista delle posizioni calcolate fino a quel momento e presenti in `SA`, oggetto che verrà utilizzato per produrre il **suffix array finale**.

- **hope_insert:** Questo metodo svolge il cuore dell'operazione ed è stato oggetto di una principale modifica di ottimizzazione all'interno del lavoro svolto per questa tesi.
  Tale modifica verrà descritta nel dettaglio nel capitolo successivo.
  Presa in input la *key*, il metodo ricava dal dizionario le posizioni di `main_lista` in cui è presente la chiave.
  Ogni posizione viene esaminata e per ognuna di esse vengono analizzate le posizioni precedenti, le quali vengono inserite all'interno di un `array_int_t` denominato `not_taken_list` per tenere conto delle *key* ancora non prese.

  Nel frattempo è presente un array parallelo denominato `group_dict_is_taken` che tiene conto delle *key* già prese. Questa operazione continua fino a quando all'iterazione *i* `group_dict_is_taken[i]` vale 1. A questo punto su **ogni** elemento in `not_taken_list` viene eseguita **ricorsivamente** la funzione `hope_insert` e viene modificato a 1 il corrispondente elemento di `group_dict_is_taken`.

Infine l'elemento viene inserito alla fine della lista delle posizioni calcolata fino a quel momento di quel gruppo. Viene infine restituita la lista delle posizioni ordinata lessicograficamente.

CHAPTER 4

OPERAZIONI EFFETTUATE

In questo capitolo descriviamo in dettaglio tutte le operazioni svolte per questa tesi, integrando con commenti e osservazioni sui risultati ottenuti.
Durante il lavoro, sono state tentate più strade le quali hanno prodotto risultati o osservazioni con lo scopo di al migliorare le performance ed il funzionamento dell'applicazione in futuro.

4.1 Gestione della memoria e ottimizzazione hope_insert

4.1.1 Gestione della memoria

Il primo lavoro svolto ha avuto l'obiettivo di integrare gestione della memoria completamente mancante nel programma.

Dopo un'attenta analisi e studio del codice sorgente, sono state effettuate numerose de-allocazioni attraverso la funzione `free()` di C che permette di de-allocare la memoria di un certo oggetto in modo del tutto sicuro.
E' stata quindi liberata memoria di tutti quegli oggetti dall'utilizzo limitato, i quali esaurivano il loro scopo all'interno di una data funzione, risultando,

così, pesanti a livello prestazionale nell'economia generale dell'esecuzione del programma su input molto grandi.

Questa operazione però non ha risolto il principale problema dell'applicazione, in quanto tutte le strutture dati implementate utilizzano array per conservare dati.
Problema principale di questa scelta consiste nell'impossibilità di deallocare parzialmente memoria, causando, un problema di memory-leak che arresta prematuramente il programma causando segmention-faults, su un'esecuzione continua su input molto grandi.

Questo problema verrà approfondito nella sezione successiva relativa alla lettura di input da file.

4.1.2 Ottimizzazione funzione hope_insert

Tale funzione è parte del processo della formazione dei gruppi di catene. Essa presenta un ciclo `while(1)` dove vengono esaminati tutti gli elementi di *key* in `main_lista` e tutti i suoi precedenti fin quando non è analizzato un elemento già preso o che faccia parte di un'altra catena.

Successivamente è presente un ulteriore ciclo, che scandisce gli elementi di `not_taken_list`, dove in seguito ad alcune operazioni, presenti alla riga 21 nel codice sotto riportato, era presente la funzione `get_pos_key_lista` che aveva il compito di restituire l'indice di `key_lista` dell'elemento in analisi. Ciò comportava dover scorrere l'intero `array_int` presente in `key_lista` ad ogni iterazione per due volte all'interno della funzione, in quanto tale metodo era richiamato già alla riga 9 del codice senza però salvare l'intero restituito per il ciclo successivo.

Per ottimizzare la funzione è stato creato, alla riga 3 del codice sotto riportato, un array parallelo a `not_taken_list` denominato `not_taken_list_index` , il quale tiene conto degli indici relativi a `key_lista` degli elementi inseriti in `not_taken_list`. Ciò permette, come riportato nella riga 21, di accedere all'indice direttamente in tempo costante.

Di seguito è riportata la porzione di codice interessata della funzione hope_insert:

```
  int k = lista->ints[j] - 1;
  array_int_t* not_taken_list = init_array_int(0);
  array_int_t* not_taken_list_index= init_array_int(0);//FM

  //creo un array parallelo con gli indici
  while(1){
           if(binary_search(group_start_index,
           k+1,0,len_group-1) >= 0)
                break;
           int item = main_lista[k];
           int index=get_pos_key_lista(key_lista,item);//FM
           int taken = group_dict_is_taken[index];
           if(taken == 0){
               k = k - 1;
               add_int_array(not_taken_list,item);
               add_int_array(not_taken_list_index,index);//FM
           }
           else
               break;
           }
           //head is at end of array
 for (k = (not_taken_list->len)-1; k>= 0; k--){
   int position_key = not_taken_list_index->
   ints[k];
   array_int_t* SA_temp = hope_insert(len_group,
   not_taken_list->ints[k],position_key, group_dict,
   group_start_index, main_lista, key_lista,
   group_dict_is_taken);
   SA = concat_new_array_int(SA_temp, SA);
   group_dict_is_taken[position_key] = 1;
   }
   ...
```

Listing 4.1: Porzione di codice interessata funzione hope_insert

4.2 Osservazioni sulle prestazioni e test aggiornati

Le operazioni di ottimizzazione svolte hanno avuto un buon impatto sulle prestazioni generali del programma ottenendo un **piccolo (seppur significativo) incremento** su input molto grandi (principale oggetto dello scopo del programma), con stringhe di **oltre 3000 caratteri**, mentre **non ha avuto impatti su input inferiori ai circa 2900 caratteri**.

Durante le analisi svolte sulle prestazioni dell'applicazione, sono stati effettuati tutti i test su **entrambe le versioni del programma (prima e dopo le operazioni di ottimizzazione)** , su un ampio spettro di lunghezze di input per valutarne gli effettivi miglioramenti.
Tale operazione è stata effettuata anche perchè i test effettuati precedentemente al lavoro di questa tesi, non erano validi a causa della loro esecuzione su un laptop con alimentazione a batteria. Ciò degradava, di molto, le performance della macchina e quindi del programma, restituendo risultati non veritieri rispetto alle prestazioni effettive del programma su macchina in condizioni standard e quindi con alimentazione continua.
Su uno stesso input di 16734 caratteri, infatti, il programma si attesta su un tempo di circa **4244ms** contro i circa **5700ms** riscontrati dai **test precedenti su batteria**.

Riportiamo ora una tabella con i risultati dei test svolti su input di varie lunghezze, esaminando anche un precedente test effettuato sull'originale programma in Python.
Un osservazione preliminare: I test sono stati effettuati sulla stessa macchina e nelle stesse condizioni operative. I tempi sono stati calcolati su una media degli stessi su più esecuzioni.

Lunghezza Input	Tempo attuale]	Tempo precedente	Tempo in Python]
16734	∼4238ms	∼4244ms	∼3995ms
2882	∼112ms	∼112ms	-
22160	∼9433ms	∼9561ms	-
1323	∼28ms	∼ 28ms	-
32651	∼19352ms	∼ 19672ms	-
249	∼3ms	∼3ms	-
763	∼19ms	∼19ms	-

Table 4.1: Tabella delle prestazioni prima e dopo il lavoro di ottimizzazione

Da come si evince dal primo test, le prestazioni del programma in C sono ancora **peggiori del programma in Python** e ciò è probabilmente dovuto all'**utilizzo intenso di array** all'interno del codice.

Sebbene le prestazioni siano comunque peggiori, la situazione è meno grave di quanto è stato precedentemente analizzato. Infatti, se i 5700ms del test precedente a questa tesi faceva risultare una differenza di quasi 2 secondi tra le due applicazioni, il risultato effettivo attuale di **4238ms contro i 3995ms** di Python, pone una luce ben diversa sulla situazione, seppur ancora non positiva.

4.3 Lettura input automatizzata da file e problema del memory leak

Una delle strade percorse durante il lavoro è stata quella di automatizzare l'utlizzo del programma attraverso la scrittura di uno script che leggesse automaticamente una sequenza di input da un file FASTA e computasse il Suffix Array singolarmente per ogni stringa letta prima di passare alla successiva. Seppur funzionante, tale operazione ha permesso di rilevare un problema importante del codice dell'applicazione (accennato nella sezione

4.0.1), cioè quello del **memory leak**. Grazie allo script, il programma riesce a computare il Suffix Array di 10 stringhe prima di finire in **segmentation fault**, non riuscendo a costruire neanche il dizionario dei suffissi della undicesima stringa.

Le stringhe prese in esame sono di lunghezze molto grandi, che spaziano dai 10000 ai 33000 caratteri.

Per appurare che il problema fosse relativo ad un'impossibilità di allocare ulteriore memoria nel programma, è stato effettuato un test su di uno stesso file FASTA, composto da 50 stringhe più piccole con lunghezze di al più 700 caratteri, seguite da stringhe di lunghezze maggiori equivalenti a quelle analizzate nel test precedente. Questo cambiamento ha concesso al programma di computare esattamente 60 stringhe interrompendosi nello stesso punto durante la computazione della 61esima. L'interruzione, dunque, avveniva esattamente dopo la computazione di 10 stringhe di lunghezze molto grandi come descritte precedentemente.

Un esempio di situazione in cui l'utlizzo di array comporta spreco di memoria è durante la formazione delle catene nella struttura `speed_keys_array` che contiene array di stringhe. Tale oggetto contiene al suo interno tutti i fattori locali distinti della stringa all'inizio dell'operazione.

Esempio. Sia $S = aaabcaabcadcaabca$ la stringa trattata negli esempi della tesi,
prendiamo come esempio la creazione della catena $C_1 : [a, aa, aaa]$.
Durante l'operazione, la situazione iniziale di `speed_keys_array` sarà la seguente: $[a], [aa], [aaa]...[dcaabca]$.

Al termine della creazione della catena avremo questa situazione in `speed_keys_array`: $[], [], [a, aa, aaa]...[dcaabca]$.

Come possiamo notare, esistono 2 celle vuote, in quanto nell'array è stato effettuato un **semplice shift** degli elementi rimanendo memoria inutilmente allocata.

Questa situazione è solo un piccolo esempio di ciò che comporta l'utilizzo di array per tali operazioni specialmente su stringhe di grandi dimensioni. Daremo, in seguito, un accenno ad una possibile soluzione a tale problema. Qui di seguito è riportato il codice dello script di lettura da file utilizzato per tali test.

```
1    int main(int argc, char **argv) { //Main con
2    //lettura da file fasta
3    char * word=NULL;
4    char buff[BUFF_SIZE];
5    char *ch;
6    FILE *fp;
7    size_t len =0;
8    long read;
9
10
11   struct timeval stop, start;
12
13
14  fp=fopen("sampled_read.fasta","r");
15   if(fp==NULL)
16       exit(EXIT_FAILURE);
17   int wordN=0;
18
19   while((len=getline(&word, &len, fp))){
20       if(word[0] == '>') //Segue la formattazione del
21       //file >i,stringa i
22           //fprintf(stdout, "Non mi serve\n");
23       else {
24           //fprintf(stdout,"Mi serve\n");
25           word[len-1]='\0';// aggiungo il '\0' alla riga letta
26           fprintf(stdout,"Word Number: %d ----\n",wordN);
27           fprintf(stdout,"%s\n",word);
28           fflush(stdout);
29
30
```

```
31              wordN++;//Numero della stringa in analisi
32              //algoritmo principale del programma sulla
33              //stringa letta
34              gettimeofday(&start, NULL);
35              array_int_t * SA_w =
36              sorting_suffixes_via_icfl(word);
37
38              int *arr = SA_w->ints;
39              int size = SA_w->len;
40
41              for (int i = 0; i < size/2; i++){
42                   int temp = arr[i];
43                   arr[i] = arr[size - 1 - i];
44                   arr[size - 1 - i] = temp;
45              }
46
47              gettimeofday(&stop, NULL);
48              printf("time: %lu\n", ((stop.tv_sec
49              start.tv_sec) * 1000000 + (stop.tv_usec
50              start.tv_usec) )/ 1000 );
51
52
53
54               int len_w = strlen(word);
55               printf("Lunghezza input: %d\n",len_w);
56               printf("Lunghezza suffissi: %d\n", SA_w->len);
57
58               printf("FINAL SA:\n[");
59
60              for(int i = 0; i < SA_w->len;i++){
61                  fprintf(stderr,"%d, ",SA_w->ints[i]);
62               }
63              printf("]\n");
64              word=NULL;
65
66               //Test di correttezza del suffix, oneroso
67               //in fase di test
```

```
68              /* char temp_string_1[len_w+1];
69               char temp_string_2[len_w+1];
70               int flag = 1;
71
72               for(int i = 0; i < (SA_w->len) - 1 ;i++){
73                  int cheaj = strcmp( strcpy(temp_string_1,
74                  &word[SA_w->ints[i]]),
75                  strcpy(temp_string_2,&word[SA_w->ints[i+1]]) );
76                  if( cheaj > 0 ){
77                      flag = 0;
78                  }
79              }
80               if (flag)
81                  printf("SA ordinato: True\n");
82               else
83                  printf("SA ordinato: False\n");
84               */
85              fflush(stdout);
86              free(SA_w);
87              free(speed_structure);
88          }
89      }
90
91  }
```

Listing 4.2: Codice funzione main con lettura da file

4.4 Stampe intermedie dei gruppi e delle catene

L'ultimo lavoro svolto per questa tesi ha avuto lo scopo di aggiungere delle stampe intermedie nell'esecuzione dell'applicazione, più precisamente, delle catene e dei gruppi e degli indici ad essi associati. Sebbene la stampa delle catene sia stata possibile soltanto per i suffissi presenti nella singola catena (a causa di come sono calcolate all'interno del codice) è stato invece possibile

stampare i gruppi di catene. Seguono dettagli.

4.4.1 Stampe delle catene e funzione printChain

Sebbene sia stata scritta una funzione per la stampa delle catene, la `printChain`, non è precisa in quanto, come spiegato nel capitolo precedente, le catene vengono formate all'interno del codice solo attraverso la concatenazione dei suffissi, non effettuando, temporaneamente, l'ordinamento degli indici delle posizioni ad esse associate.

A causa del posporre della fusione degli indici, la funzione stampa gli indici concatenando solamente le posizioni dei fattori all'interno della catena. La funzione è stata mantenuta per un futuro miglioramento o per una stampa dei soli suffissi delle catene.

Tale funzione include una sezione già presente all'interno del codice (come ciclo di debug per `speed_keys_array`) che presentava un punto critico alla riga 3, dove l'indice h se uguale a len,portava alla stampa o di una catena vuota inesistente o al `segmentation fault` per alcuni input.
La condizione di entrata del ciclo `for` è stata posta come $h \leqslant len - 1$ piuttosto che $h \leqslant len$ per risolvere il problema.
Di seguito è riportato il codice della funzione in esame:

```
1  void printChain(array_strings_t** speed_keys_array,
2  positions_lists_t* posList, int len){
3  //Stampa dei soli suffissi delle catene
4      printf("Stampa Catene ---- taglia: %d\n", len);
5      for (int h=0; h<=len-1;h++) {//h<=len
6      //porta a seg.fault o stampa di
7      //cella vuota
8      printf("%d ",speed_keys_array[h]->len);
9      print_array_strings(speed_keys_array[h]);
10     //stampa indici delle posizioni non ordinate
11    /* int chainLen=speed_keys_array[h]->len;
12     if(chainLen>=1){ //FM: Stampa non ordinata delle posizioni
```

```
13          printf(" (");
14          for(int p=0;p<chainLen;p++){
15               print_array_int(get_positions_array(posList,
16 speed_keys_array[h]->strings[p]));
17          }
18
19          printf(")");
20      }
21       */
22      printf("\n");
23          //aggiunta
24      }
25
26
27 }
```

Listing 4.3: Funzione di stampa dei suffissi delle catene

4.4.2 Stampe dei Gruppi

La stampa dei gruppi è stata ottenuta aggiungendo opportune stampe nelle due principali funzioni che si occupano della loro creazione: `step_2` e `hope_merge`.

Di seguito un esempio di output sull'esempio base della tesi $S = aaabcaabcadcaabca$:

```
[0] -[aaa, aa, a]
[aabca, aa, a]
[abca, a]
[3] -(9, 6, 2, 13, 5, 1, 12, 0, 16, )
[bca, b]
[1] -(7, 3, 14, )
[caabca, ca]
[1] -(8, 4, 11, 15, )
[dcaabca]
(10, )
```

Figure 4.1: Stampa dei gruppi

L'output consiste nella Stampa delle catene che formano il gruppo G, precedute dalla lunghezza del gruppo esaminato iterazione dopo iterazione per motivi di manutenibilità, seguiti, infine, dagli indici delle posizioni ordinati

lessicograficamente e associati al gruppo G, sia esso formato da una o più catene.

Di seguito vengono riportate le funzioni `hope_merge` e `step_2` aggiornate nella loro interezza, essendo le stampe disposte lungo tutto il codice:

```
1       array_int_t* hope_merge(array_int_t** group,
2       int len_group,array_strings_t** speed_keys_array){
3    array_int_t* SA = init_array_int(0);
4
5    //______________________
6    // Creating main_lista and group_start_index
7   int len_main_lista = 0;
8    int group_start_index[len_group];
9    for(int i = 0; i<len_group;i++){
10      group_start_index[i] = len_main_lista;
11      len_main_lista += group[i]->len;
12   }
13
14  int main_lista [len_main_lista];
15  //index track the main_lista array
16   for(int i = 0,index = 0; i < len_group;i++){
17      //the chain's head is at end of it
18      int len_chain = group[i]->len;
19      for(int j = 0; j < len_chain; j++,index ++){
20          main_lista[index] = group[i]->ints[len_chain-1-j];
21      }
22   }
23   //______________________
24   // Creating group_dict and keys_lista
25   positions_lists_t* group_dict = initPositionsLists();
26   array_int_t* key_lista = init_array_int(0);
27
28
29   for(int j = 0; j < len_main_lista;j++){
30      // the suffix are int, they are converted in chars
```

```
31          char suffix_int[12];
32          sprintf(suffix_int, "%d\0", main_lista[j]);
33          int check = contains_suffix(group_dict, suffix_int);
34          if(check >= 0){
35              char *suffix = malloc(sizeof(*suffix) * 12);
36              strncpy(suffix,suffix_int,12);
37              add_suffix(group_dict,suffix);
38              add_position(group_dict,suffix_int,j);
39              //need be dynamically updated
40              add_int_array(key_lista,main_lista[j]);
41          }
42          else{
43              //it is not an addition in the head
44              add_position(group_dict,suffix_int,j);
45          }
46
47      }
48      //  array parallel to key_lista
49      int *group_dict_is_taken =
50      calloc(key_lista->len,sizeof(int));
51
52
53      add_int_array(SA,key_lista->ints[0]);
54      group_dict_is_taken[0] = 1;
55
56      //--------------------
57      // find key
58      array_int_t* SA_temp;
59      for(int i = 1; i < key_lista->len;i++){
60          int key = key_lista->ints[i];
61
62          if(group_dict_is_taken[i] == 0){
63              SA_temp = hope_insert(len_group, key, i, group_dict,
64              group_start_index, main_lista, key_lista,
65              group_dict_is_taken);
66              SA = concat_new_array_int(SA_temp, SA);
67          }
```

```
68      }
69      printf("("); //STAMPA INDICE DEL GRUPPO COMPUTATO
70      print_array_int(SA);
71      printf(")\n");
72      free(SA_temp);//FM
73      return SA;
74  }
75
76  array_int_t* step_2(char* word, positions_lists_t* suffix_dict,
77  int* lcp_sorted_suffixes, int* mask_index, int mask_len){
78      //printPositionsLists(suffix_dict);
79      array_strings_t** speed_keys_array=
80      speed_structure(suffix_dict,
81      lcp_sorted_suffixes);
82
83      //the lenght is the same
84      int len_speed = suffix_dict->len;
85      array_int_t* SA = init_array_int(0);
86      array_int_t* SA_group = init_array_int(0);
87      array_int_t** group = malloc(sizeof(*group));
88      array_int_t* current_SA;
89      int len_group = 0;
90
91      for(int i=0; i < len_speed; i++){
92          array_strings_t* current_speed = speed_keys_array[i];
93
94          if(lcp_sorted_suffixes[i] == 0){
95               printf("[%d] -",len_group);
96              if(len_group > 1){
97                  array_int_t* l = hope_merge(group,len_group,
98                  speed_keys_array);
99                  //(SA = SA + l)
100                 //head is at end of l
101                 SA = concat_new_array_int(l,SA);
102
103             }
104             else{
```

```
105                 if(SA_group->len!=0){ //Stampa degli
106                 //indici delle catene non formanti gruppi
107                 printf("(");
108                 print_array_int(SA_group);
109                 printf(")\n");
110                 }
111                 SA = concat_new_array_int(SA_group,SA);
112             }
113             //reset group resetting its lenght
114 
115             len_group = 0;
116         }
117 
118         if(current_speed->len != 0){
119             //current_speed it sorted in reverse so
120             //start to the end
121             //current_speed is the reverse of current_key_lista
122             print_array_strings(current_speed);
123             //Stampa delle catene
124             //dei suffissi per la stampa dei gruppi
125             current_SA = get_positions_array(suffix_dict,
126             current_speed->strings[current_speed->len-1]);
127             SA_group = current_SA;
128             for(int index = 1; index <
129             current_speed>len;index++){
130                 array_int_t* current_list =
131                 get_positions_array(suffix_dict,
132                 current_speed->strings[current_speed->
133                 len-1-index]);
134                 SA_group = merge_hash_lists_prefix(current_SA,
135                 current_list,mask_index,mask_len,word);
136                 current_SA = SA_group;
137             }
138             printf("\n");
139             len_group++;
140             if(len_group>0)
141 
```

```
142             group = realloc(group,
143             sizeof(array_int_t*)*len_group);
144             group[len_group-1] = SA_group;
145             if( i == len_speed - 1){
146                 if(len_group == 1)
147                     SA = concat_new_array_int(current_SA,SA);
148                 else{
149                     array_int_t* l = hope_merge(group,len_group,
150                     speed_keys_array);
151                     SA = concat_new_array_int(l,SA);
152                 }
153
154             }
155         }
156
157     }
158     if(len_group!=0){ //stampa indici ultimo gruppo di
159     //una sola catena quando presente
160      printf("(");
161      print_array_int(SA_group);
162      printf(")\n");
163     }
164
165     free(group);//FM
166     free(current_SA);//FM
167     free(SA_group);//FM
168     return SA;
169 }
170
```

Listing 4.4: funzioni hope_merge e step_2 aggiornate

CHAPTER 5

CONCLUSIONI E PROBLEMI APERTI

5.1 Conclusioni, problemi aperti e possibili soluzioni

In conclusione, l'applicazione presenta ancora diversi punti critici e delle performance ancora non soddisfacenti rispetto al codice Python.
Le diverse operazioni svolte sul codice hanno fornito numerose informazioni sullo stato attuale dell'applicazione, in particolare:

- il modo in cui vengono calcolate le catene, che andrà modificato per ottenere un riferimento agli indici ordinati lessicograficamente delle posizioni ad esse associate, senza rallentamenti .
- Diverse problematiche dovute all'utilizzo degli array.

Il principale problema, risiede nell'utilizzo intenso di array in tutte le strutture dati, il che porta a numerosi rallentamenti in diverse occasioni (come nell'ottimizzazione eseguita nella funzione `hope_insert`) oltre ad un problema di memory leak.

Principale modifica da effettuare e possibile soluzione al problema più importante, sarebbe una conversione totale di tutte le strutture dati, **passando dall'utilizzo di array all'utilizzo di liste linkate**, le quali permetterebbero, innanzitutto, di evitare problemi di spreco di memoria descritti nella sezione 4.0.3 grazie alla possibilità di eliminare un elemento deallocando anche lo spazio ad esso associato ed evitando operazioni onerose di shift.

RINGRAZIAMENTI

Ringrazio infinitamente mia mamma, mio papà e mio fratello Ciro per avermi supportato e sopportato nei momenti più difficili. Sono stati anni duri in cui nonostante le difficoltà eravate lì a coprirmi le spalle, e ciò non sarà mai dimenticato. Ci sono stati tantissimi momenti bui, ma voi eravate lì pronti a non farmi pesare nulla e a darmi la spinta necessaria a continuare. Ci sono tante cose che vorrei dirvi, tante cose che sento. Ma ne basta una. Se sono cresciuto con certi valori, come sono oggi lo devo a voi, a dei super genitori che nonostante tante difficoltà, da veri supereroi avete continuato a rincuorarmi anno dopo anno. So di non essere stato capace di esprimere questo sentimento negli ultimi mesi, ma **vi voglio tanto bene** e un giorno la mia più grande aspirazione è di essere un genitore come lo siete voi e di continuare a condividere la mia vita come ho sempre fatto con tutti e tre. Ci abbracciamo dopo.

A te che sei lassù **cucciolone mio, Chopper,** sappi che ogni sforzo è dedicato a te, a te che mi hai reso la persona che sono oggi. Sei entrato nella mia vita quando ero un bambino di 7 anni, e mi hai insegnato il valore dell'amore incondizionato e della gioia che si prova nel crescere con un pelosone al proprio fianco in ogni step importante della vita. Purtroppo non sono riuscito a farti quest'ultimo dono prima che ci salutassi, ma sono sicuro, lo sento, che non te ne sei andato davvero e che tutto questo tempo, da quel 22 dicembre 2022, tu sei rimasto qui, a vegliare su di me, su di noi e sul tuo nipotino, Poldino.

Grazie di tutto, Choppy, aspettami così che **un giorno potremo correre di nuovo insieme**.

Voglio dare un immenso grazie alla **metà del mio cuore**, **Chiara**. Grazie alla tua immensa forza e coraggio mi hai insegnato tanto, mi hai supportato, rincuorato in momenti davvero bui dove solo una persona che ama davvero non si sarebbe arresa. Sei stata, sei e sarai sempre la **fonte di energia del mio cuore**. Abbiamo condiviso tanto, davvero tanto, emozioni di ogni tipo che in questi 5 anni sono state scolpite dentro di me, dai momenti di tenerezza, alle piccole avventure e progetti vissuti insieme (Da quelli universitari ad un intero centro ripieno dei nostri sogni e della nostra fantasia in Minecraft, un tesoro per me davvero prezioso, un simbolo del nostro legame), non vedo l'ora di scoprirne di altre in ogni momento. Insieme concluderemo anche la tua di avventura e spiccheremo il volo non dubitarne mai, neanche per un istante. Ti amo piccola Spunf, **sei il mio universo**.

Ringrazio i miei Zii per avermi dato un forte aiuto in questi anni e per la presenza costante che non sarà mai data per scontata. Un grande abbraccio a tutti.

Voglio ringraziare davvero tanto Francesca o meglio la Chicca per tutto il sostegno, l'aiuto e le chiacchierate fatte in questi anni. Sei come una sorella maggiore per me e ogni passeggiata e momento insieme è sempre bello e ristorativo. Grazie di tutto.

Un ringraziamento speciale va ai miei due fratelli di avventure, i miei due JoBros, Pietro e Andrea. Ragazzi con voi ho condiviso tutto dalla A alla Z. 10 anni di piccole e grandi pazzie, dai 100 Kebab da waqas (con colesterolo alto annesso),alle carte magia, dalle cucinate Gourmet meravigliose alle chiacchierate di vita lungo le strade. Il mio augurio è di avervi sempre come riferimento da ora e verso l'infinito e oltre. Grazie e ancora grazie ragazzi, per tutto.

Voglio ringraziare Luca, conosciuto anche come Bergor sul web per tutte

le scazzottate tra un argomento di studio e l'altro e per la onnipresente disponibilità, sei un grande amico per me. Metticela tutta Bergor che poi si va girovagando con la BergorMobile!

Un grande grazie va alla mia relatrice, la Professoressa Rosalba Zizza, per avermi seguito con sacrosanta pazienza in questi mesi e per avermi dato tanto supporto morale nei momenti di sconforto e di singhiozzo. Lei è una grande Professoressa e nel mio futuro avrò sempre un ricordo piacevolissimo dei suoi insegnamenti e del suo modo di porsi con noi studenti.

REFERENZE

[1] Uwe Baier. “Linear-time Suffix Sorting - A New Approach for Suffix Array Construction”. In: 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016). Ed. by Roberto Grossi and Moshe Lewenstein. Vol. 54. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016, 23:1–23:12. isbn: 978-3-95977-012-5.

[2] Nico Bertram, Jonas Ellert, and Johannes Fischer. “Lyndon Words Accelerate Suffix Sorting”. In: 29th Annual European Symposium on Algorithms (ESA 2021). Ed. by Petra Mutzel, Rasmus Pagh, and Grzegorz Herman. Vol. 204. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fur Informatik, 2021, 15:1–15:13. isbn: 978-3-95977-204-4.

[3] Kuo-Tsai Chen, Ralph H. Fox, and Roger C. Lyndon. “Free Differential Calculus, IV. The Quotient Groups of the Lower Central Series”. In: Ann. Math. 68 (1958), pp. 81–95.

[4] Jean-Pierre Duval. “Factorizing Words over an Ordered Alphabet”. In: J. Algorithms 4.4 (1983), pp. 363–381.

[5] Johannes Fischer and Florian Kurpicz. “Dismantling DivSufSort”. In: Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017. Ed. by Jan Holub and Jan

Zd´arek. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017, pp. 62–76. url: http://www.stringology.org/event/2017/p07.html.

[6] Keisuke Goto. "Optimal Time and Space Construction of Suffix Arrays and LCP Arrays for Integer Alphabets". In: Prague Stringology Conference 2019, Prague, Czech Republic, August 26-28, 2019. Ed. by Jan Holub and Jan Zd´arek. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019, pp. 111–125. url: http://www.stringology.org/event/2019/p11.html.

[7] M. Lothaire. Algebraic Combinatorics on Words, Encyclopedia Math. Appl. Vol. 90. Cambridge University Press, 1997.

[8] Roger Lyndon. "On Burnside problem I". In: Trans. Amer. Math. Soc. 77 (1954), pp. 202–215.

[9] Sabrina Mantaci et al. "Suffix array and Lyndon factorization of a text". In: J. Discrete Algorithms 28 (2014), pp. 2–8.

[10] Paola Bonizzoni et al. "Inverse Lyndon words and inverse Lyndon factorizations of words". In: Adv. Appl. Math. 101 (2018), pp. 281–319.

[11] Paola Bonizzoni et al. "Lyndon Words versus Inverse Lyndon Words: Queries on Suffixes and Bordered Words". In: Language and Automata Theory and Applications - 14th International Conference, LATA 2020, Milan, Italy, March 4-6, 2020, Proceedings. Vol. 12038. Lecture Notes in Computer Science. Springer, 2020, pp. 385–396.

[12] Paola Bonizzoni et al. "On the longest common prefix of suffixes in an inverse Lyndon factorization and other properties". In: Theor. Comput. Sci. 862 (2021), pp. 24–41.

[13] Lorraine Ayad et al. "Sorting suffixes via inverse Lyndon factorization".

[14] Uwe Baier Linear-time Suffix Sorting – A New Approach for Suffix Array Construction

[15] M. Di Matteo,Suffix Array: Analisi ed implementazione in C di un algoritmo di costruzione tramite fattorizzazione inversa di Lyndon , Tesi di

Laurea in Informatica, 2022.

LIST OF FIGURES

LIST OF TABLES