Università degli Studi di Milano Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

Costruzione di un Suffix Array a partire da fattorizzazione inversa di Lyndon

Relatore: Raffaella Rizzi
Co-relatore: Paola Bonizzoni

Relazione della prova finale di:
Elia Leonardo Martin
Matricola 886366

Anno Accademico 2023-2024

A Luca

Indice

Introduzione

Il Suffix Array è una struttura dati di indicizzazione efficiente che permette di risolvere il problema di pattern matching in un testo in tempi logaritmici. Nonostante la sua importanza, lo studio e la costruzione del Suiffix Array rimangono una sfida computazionale rilevante.
Questo lavoro di stage propone un approccio alla realizzazione basato su una particolare fattorizzazione di un testo, denominata Fattorizzazione Inversa di Lyndon.
Nella fase preliminare, sono stati studiati ed esposti i principali teoremi atti all'ordinamento dei suffissi locali, fondamentali per la corretta costruzione del Suffix Array, i quali hanno guidato la progettazione di una struttura ad albero utile alla rappresentazione dei suffissi locali.
Verranno successivamente forniti ed analizzati gli pseudocodici delle funzioni chiave, facilitando la traduzione degli algoritmi teorici in implementazioni pratiche.
L'implementazione della costruzione del Suffix Array, realizzata in linguaggio C++, sarà focalizzata sull'utilizzo di librerie per l'uso di Bitvector e sull'ottimizzazione di memoria, garantendo una solida base per le applicazioni pratiche del modello.

Capitolo 1

Definizioni preliminari e costruzione di un Suffix Array a partire da un Testo T

1.1 Suffix Array di un testo

Dato un alfabeto Σ, definito come un insieme finito di caratteri, Σ^* rappresenta l'insieme di tutte le stringhe che possono essere ottenute concatenando zero o più simboli presenti in Σ, dove 1 indica la stringa vuota.

Data una stringa $S = a_1\ a_2\ ...\ a_n$ con $a_i \in \Sigma^*$ denoteremo con $|S|$ la dimensione n di S. Una sottostringa di S è la stringa $a_i\ a_{i+1}\ ...\ a_j$ con $1 \leq i \leq j \leq n$ riferita nel seguito anche come $S[i : j]$.
Se $j = n$, allora $S[i : n]$ (che indicheremo come $S[i :]$ per semplicità) è un suffisso di S. Nel caso in cui $i \neq 1$ il suffisso è proprio poiché non coincidente con S.
Analogamente, data la sottostringa $S[i : j]$, se $i = 1$ allora $S[1 : j]$ (che indicheremo come $S[: j]$ per semplicità) è un prefisso di S. Nel caso in cui $j \neq 1$ il prefisso è proprio poiché non coincidente con S.

Esempio: Sia $S = aabcde$, $S[: j] = aab$ con $j = 3$ è un prefisso proprio di S. $S[i :] = cde$ con $i = 4$ è un suffisso proprio di S.

Specificando un ordinamento su Σ, è possibile definire una relazione di precedenza tra due stringhe x e y.
Sia $(\Sigma, <)$ un alfabeto totalmente [1] ordinato. L'ordine lessicografico $\prec$ definito su $(\Sigma^*, <)$ è definito impostando $x \prec y$ se si verifica una delle due seguenti condizioni:

- x è un prefisso proprio di y
- $x = ras,\ y = rbt,\ per\ a, b \in \Sigma \wedge\ r, s, t \in \Sigma^*$

[1] Un alfabeto viene definito totalmente ordinato se per ogni coppia di caratteri è possibile stabilire quale elemento viene prima secondo un criterio specificato.

Per due stringhe non vuote x e y, scriveremo $x \ll y$ se $x \prec y$ e x non è un prefisso proprio di y.

Esempio: Sia $x = abcc$ e $y = abccdef$, avremo che $x \prec y$ poiché x prefisso proprio di y. Sia $x = abcc$ e $y = abcd$, avremo che $x \ll y$. Nonostante i primi tre caratteri che compongono le stringhe coincidano, notiamo che il quarto carattere di x risulta lessicograficamente minore di quello di y.

Il **Suffix Array** [7] è una struttura di indicizzazione che permette di effettuare il pattern matching in tempo $O(nlog(m))$. Il Suffix Array di una stringa T con $|T| = n$, che nel seguito chiameremo *testo*, è un SA di dimensione n, tale che $SA[i] = q$ se e solo se $T[q :]$ è l'i-esimo suffisso nell'ordinamento lessicografico dei suffissi di T.

Esempio: Sia $T = aacbed$ un testo su un alfabeto Σ^* totalmente ordinato.
I suffissi di T elencati per posizione crescente sono:
I suffissi $T[i :]$ di T, con $1 \leq i \leq n$, elencati per posizione i crescente sono: *aacbed*, *acbed*, *cbed*, *bed*, *ed*, *d*.
Il loro ordinamento lessicografico è: *aacbed*, *acbed*, *bed*, *cbed*, *d*, *ed* e quindi $SA(T) = [1, 2, 4, 3, 6, 5]$.

Il problema computazionale [4] di trovare il Suffix Array di un testo T viene formulato in modo molto semplice:

- input: un testo T definito su un alfabeto ordinato Σ
- output: il Suffix Array di T

1.2 Fattorizzazione di Lyndon

Una stringa $S \in \Sigma^+$ è una **parola di Lyndon** [8] se e solo se per ogni fattorizzazione non banale [2] di $S = uv$ si ha che $u \prec v$.

Esempio: Sia $S = abac$. Le fattorizzazioni non banali sono:

1. $S =$ "a" + "bac" con $u =$ "a" e $v =$ "bac", $u \prec v$.
2. $S =$ "ab" + "ac" con $u =$ "ab" e $v =$ "ac", $u \prec v$.
3. $S =$ "aba" + "c" con $u =$ "aba" e $v =$ "c", $u \prec v$.

[2] Decomposizione della stringa in una sequenza di sottostringhe non vuote in cui nessuna sottostringa è un prefisso di un'altra sottostringa.

Una stringa $S \in \Sigma^+$, che non è una parola di Lyndon, può essere espressa come prodotto di $h > 1$ parole di Lyndon tali che:

$$S = l_1 l_2 ... l_h \text{ e } l_1 \gg l_2 \gg l_3 \gg ... \gg l_h.$$

La sequenza $(l_1 ... l_h)$, denotata con $CFL(S)$ prende il nome di **fattorizzazione di Lyndon**. [2]

Una stringa $S \in \Sigma^+$ è una **parola inversa di Lyndon** se $s \prec S$ per ogni suffisso $s = S[i :]$ con $2 \leq i \leq |S|$, cioè per ogni suffisso proprio di S.

Esempio: Le stringhe $a, b, aaaaa, bbba, baaab, bbaba, bbababbaa$ sono parole inverse di Lyndon. Contrariamente, $aaba$ non è una parola inversa di Lyndon in quanto $aba \not\prec aaba$.

La **fattorizzazione inversa di Lyndon** di una stringa $S \in \Sigma^+$, denotata come $ICFL(S)$ è una sequenza $m_1, ..., m_k$ di parole inverse di Lyndon tali che $m_1 ... m_k = S$ e $m_i \ll m_{i+1}$ con $1 \leq i \leq k - 1$.

$$S = m_1 m_2 ... m_k$$

Essa mantiene le proprietà principali della fattorizzazione di Lyndon. [1]

Esempio: Sia $T = aaabaacaabcadcaabca$ testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$, allora $ICFL(T) =< aaa, baa, caabca, dcaabca >$.

Data una stringa $S \in \Sigma^+$, un **bordo** di S è $b = S[: i]$ uguale al suffisso proprio $S[n - i + 1 :]$ con $1 \leq i \leq n - 1$. Il più lungo prefisso proprio di S uguale al più lungo suffisso proprio di S viene denotato come "*il bordo*".

La **proprietà del bordo** afferma che se $z \in \Sigma^+$ è un bordo di m_i, allora z non può essere un prefisso di m_{i+1} per $1 \leq i \leq k - 1$.

Questo implica che i bordi delle parole inverse di Lyndon non si sovrappongono con i prefissi della parole successive, mantenendo corretta la struttura dei fattori che compongono una parola inversa di Lyndon.

1.3 Suffissi globali e suffissi locali

Per la trattazione dei teoremi e delle conoscenze presentati nella sezione seguente, faremo riferimento a [8].

Siano S, x, u, y stringhe appartenenti a Σ^*, con u fattore non vuoto di $S = xuy$. Denoteremo con $first(u)$ e $last(u)$ la posizione del primo e dell'ultimo simbolo del fattore u nella stringa S. Se $S = a_1 ... a_n$, $a_i \in \Sigma$ con $1 \leq i \leq j \leq n$, allora scriveremo $S[i : j] = a_i ... a_j$.

Un **suffisso locale** di S è un suffisso di un fattore u di S, nello specifico $suf_u(i) = S[i, last(u)]$ denota il suffisso locale di S in posizione i rispetto al fattore u, con $i \geq first(u) \wedge i \leq last(u)$.
Il corrispondente **suffisso globale** $suf_u(i)y$ della stringa $S = xuy$ alla posizione i si denota semplicemente con $suf(i)$.

Siano $S \in \Sigma^+$ e u un fattore non vuoto di S. L'ordinamento dei suffissi locali di S rispetto a u è compatibile con l'ordinamento dei corrispondenti suffissi globali di S, per ogni i, j con $fisrt(u) \leq i < j \leq last(u)$,

$$suf_u(i) \prec suf_u(j) \Longleftrightarrow suf(i) \prec suf(j)$$

Esempio: Sia $S = bceadb$, si consideri il fattore $u = bce$. Avremo che $suf_u(2) = ce$ e $suf_u(3) = e$ con $suf_u(2) \prec suf_u(3)$. I corrispettivi suffissi globali saranno $suf(2) = ceadb$ e $suf(3) = eadb$ con $suf_2) \prec suf(3)$.

Questa relazione risulta dimostrabile analizzando gli indici dei suffissi in riferimento al fattore u e alla stringa S. Se il suffisso locale in posizione i precede quello in posizione j nel fattore u, allora il suffisso globale in posizione i precederà quello in posizione j nella stringa S, in quanto il suffisso globale include tutti i caratteri dalla posizione iniziale i fino a termine della stringa, mentre il suffisso locale risulta essere limitato al fattore. Segue che la precedenza all'interno del fattore non varierà in termini di suffissi globali.

Supponendo che un suffisso locale $suf_u(i)$ di un fattore u inizi in posizione i di S, con $first(u) \leq i \leq last(u)$, avremo che il suffisso globale che inizia con tale suffisso è $S[i :]$.

Sia $S \in \Sigma^+$ una stringa che non è una parola inversa di Lyndon e sia $ICFL(S) = (m_1, ...m_k)$ la sua fattorizzazione inversa. Sia $u = m_i m_{i+1} ... m_h$ con $1 \leq i \leq h \leq k$. Assumiamo che $suf_u(j_1) \prec suf_u(j_2)$, dove $first(u) \leq j_1 \leq last(u)$ e $first(u) \leq j_2 \leq last(u)$ con $j_1 \neq j_2$.
La **proprietà di compatibilità generale** afferma che: se $suf_u(j_1)$ è un prefisso proprio di $suf_u(j_2)$ e $h < k$, allora $suf(j_2) \prec suf(j_1)$, altrimenti $suf(j_1) \prec suf(j_2)$.

Esempio: Sia $S = a^{12}bbab \in \{a, b\}^+$ con $a < b$. Avremo che $ICFL(S) = (m_1, m_2) = (a^{12}, bbab)$.
Sia $u = m_1 = a^{12}$. Consideriamo $suf_u(4) = a^9$ e $suf_u(12) = a$. Segue che $suf_u(12) = a \prec a^9 = suf_u(4)$ a cui corrisponde $suf(4) = a^9bbab \prec abbab = suf(12)$.

Successivamente, ci riferiremo con il termine *suffisso locale* a una stringa x che occorre come suffisso di qualche fattore m_t di $ICFL(S)$ e assegneremo a x la posizione di inizio i_t tale che $S[i_t : i_t + |x| - 1] = x$ e $first(m_t) \leq i_t \leq last(m_t)$. Alla posizione i_t è associato il suffisso globale $suf(i_t)$. Il suffisso globale che inizia con tale occorrenza di x è il suffisso $S[i_t :]$ e lo denoteremo come $S_{x,t}$.
Si noti che un suffisso globale $S_{x,k}$, con k ultimo fattore di $ICFLm_k$, coincide con x stesso, cioè x non è prefisso proprio di $S_{x,k}$.

Teorema 1.1. *Il suffisso globale $S_{x,k}$ è minore di qualsiasi suffisso globale $S_{x,i}$ per $i < k$*

Essendo k ultimo fattore della $ICFL(S)$, avremo che $S_{x,k} = x$. Di conseguenza $S_{x,k}$ è prefisso proprio di qualsiasi $S_{x,i}$ per $i < k$ e quindi $S_{x,k} \prec S_{x,i}$ per $i < k$.

Teorema 1.2. *Dati i e j diversi da k, il suffisso globale $S_{x,i}$ è minore del suffisso globale $S_{x,j}$ se e solo se $i < j$.*

I suffissi globali $S_{x,i}$ e $S_{x,j}$ hanno x come prefisso proprio, seguito rispettivamente dai fattori m_i e m_j. Essendo i fattori di una $ICFL(S)$ disposti in ordine crescente per definizione, avremo che $m_i \prec m_j$ se $i \prec j$. Di conseguenza, $S_{x,i} \prec S_{x,j}$.

Teorema 1.3. *Il suffisso globale $S_{x,k}$ è minore del suffisso globale $S_{z,k}$ per $z = xy$, con m_k ultimo fattore di $IFCL(S)$.*

Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baa, caabca, dcaabca \rangle$. Allora per $x = a$ con $S_{x,k} = a$ e per $z = xy = abca$ con $S_{x,k} = abca$, avremo che $S_{x,k} = a \prec abca = S_{z,k}$, riferito a $k = dcaabca$ ultimo fattore di $ICFL(T)$.
Si noti che, in questo teorema, la stringa x dovrà essere un bordo di $z = xy$. Quest'ultima potrà essere espressa come $z = xvx$, con $vx = y$. Essendo $S_{x,k}$ il più piccolo suffisso globale dell'occorrenza di x, avremo che esso risulterà essere minore di qualsiasi suffisso globale derivante da una stringa di cui x è bordo.

Teorema 1.4. *per $z = xy$, il suffisso globale $S_{z,i}$, per $i < k$, è minore del suffisso globale $S_{x,j}$ per $i \leq j < k$ e tutti i suffissi globali $S_{z,q}$, per $i < q < k$, saranno minori del suffisso globale $S_{x,j}$.*

Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baa, caabca, dcaabca \rangle$. Siano i suffissi locali $x = a$ con $j = 2$ e $z = xy = aaa$ con $i = 1$.

Allora il suffisso globale $S_{z,i} = aaabaacaabcadcaabca \prec acaabcadcaabca = S_{x,j}$ con $i \leq j$.

Corollario 4: *per $z = xy$, sia m_l il fattore più a sinistra ($l < k$) che contiene z come suffisso. Sia $S_{x,j}$ il suffisso globale che inizia con l'occorrenza di x nel fattore m_j tale che $j \geq l$. Allora, tutti i suffissi globali che iniziano con z precedono $S_{x,j}$.*

Il rapporto di precedenza tra i due suffissi deriva dal teorema precedentemente illustrato.
Se $j = l$, x e $z = xy = xvx$ saranno entrambi suffissi dello stesso fattore, avremo che $S_{z,l} \prec S_{x,j}$.
Se $j > l$, essendo z una stringa avente x come prefisso (il quale risulta essere un suffisso per un fattore $j > l$), il corrispettivo suffisso globale $S_{z,l}$ risulterà essere minore di $S_{x,j}$.

Teorema 1.5. *per $z = xy$, sia m_l il fattore più a sinistra ($l < k$) che contiene z come suffisso. Sia m_q il fattore più a sinistra, per $q < i$, che ha x come suffisso e tale che m_{q+1} abbia un prefisso α, per $|\alpha| \leq |y|$, che sia maggiore di y. Allora tutti i suffissi globali $S_{z,*}$ che iniziano con z precedono $S_{x,q}$.*

Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baa, caabcaaa, dcaabca \rangle$. Siano i suffissi locali $x = a$ in m_q con $q = 2$ e $z = xy = aaa$ in m_l con $l = 1$ e in m_3. Sia $y = aa$ con $|y| = 2$ e $\alpha = ca$ con $|\alpha| = |y| = 2$ prefisso di m_3, tale che $y = aa \prec ca = \alpha$. Allora $S_{z,1} = aaabaacaabcaaadcaabca \prec acaabcaaadcaabca = S_{x,2}$ e $S_{z,3} = aaadcaabca \prec acaabcaaadcaabca = S_{x,2}$.

Si noti come $S_{z,l} = xym_{l+1}...m_k$ e $S_{x,q} = x\alpha m_{q+1}...m_k$. Essendo x prefisso di entrambi i suffissi globali, il rapporto di precedenza sarà determinato da y e α. Considerando che $y \prec \alpha$, in quanto suffisso del fattore m_l precedente rispetto a m_q in cui *alpha* è prefisso, avremo che $S_{y,*} \prec S_{\alpha,q}$.

Teorema 1.6. *per $z = xy$, sia m_l il fattore più a sinistra ($l < k$) che contiene z come suffisso. Sia m_q il fattore più a destra, per $q < l$, che ha x come suffisso e tale che m_{q+1} abbia un prefisso α, per $|\alpha| \leq |y|$, minore o uguale a y. Allora tutti i suffissi globali $S_{z,*}$ che iniziano con z seguono $S_{x,q}$.*

Capitolo 2

Algoritmo di costruzione

2.1 Definizioni e strutture di supporto

Sia x un suffisso locale che occorre in almeno un fattore di $ICFL(T)$, con T testo definito su Σ. Definiamo $[x] - list$ come la lista ordinata in modo crescente degli indici di inizio dei suffissi locali $suf_u(x)$, con $m_1 \leq m_u \leq m_k$ fattore di $ICFL(T)$.

Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baa, caabca, dcaabca \rangle$. Allora $[x]-list = [3, 6, 12, 19]$.

Sia x un suffisso locale che occorre in almeno un fattore di $ICFL(T)$, con T testo definito su Σ. Definiamo $[x]_g - list$ la lista degli indici di inizio dei suffissi globali $S_{x,u}$, con $m_1 \leq m_u \leq m_k$) fattore di $ICFL(T)$, ordinati in modo lessicografico crescente.

Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baa, caabca, dcaabca \rangle$. Allora $[x]_g-list = [19, 3, 6, 12]$.

Dati i suffissi locali x e z, con $z = xy$, definiamo $[x, z] - list$ la lista ordinata in modo crescente degli indici di inizio dei suffissi locali $suf_u(x)$ e $suf_u(z)$, con $m_1 \leq m_u \leq m_k$ fattore di $ICFL(T)$.

Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baa, caabca, dcaabca \rangle$. Siano $x = a$ e $z = xy = abca$. Allora $[x, z] - list = [3, 6, 9, 12, 16, 19]$.

Dati i suffissi locali x e z, con $z = xy$, definiamo $[x, z]_g - list$ la lista degli indici di inizio dei suffissi globali $S_{x,u}$ e $S_{z,u}$, con $m_1 \leq m_u \leq m_k$ fattore di $ICFL(T)$, ordinati in modo lessicografico crescente.

Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baa, caabca, dcaabca \rangle$. Siano $x = a$ e $z = xy = abca$. Allora $[x, z]_g - list = [19, 3, 16, 9, 6, 12]$.

Siano $[z]_g - list$ e $[x]_g - list$ le due liste di due suffissi locali x e z tali che $z = xy$.

Definiamo l'**Insertion Target** come la posizione h in $[x]_g - list$ relativa all'elemento prima del quale deve essere inserita $[z]_g - list$ per ottenere $[x, z]_g - list$.
Un valore $h = |[x]_g - list| + 1$ indica che $[z]_g - list$ è da aggiungere in coda a $[x]_g - list$, mentre un valore $h = 1$ indica che $[z]_g - list$ è da aggiungere in testa.

Risulta necessario rappresentare e tenere traccia delle occorrenze di x e z nei diversi fattori di $ICFL(T)$. Per farlo utilizzeremo una struttura dati denominata ***Bitvector*** [3] [6], rappresentante un vettore in grado di memorizzare una sequenza di bit posti a 0 o a 1.
La struttura dati presenta due diverse funzioni: [5]

- *rank1(B, r)*, ritorna il numero di bit uguali a 1 nel prefisso $B[: r]$
- *select1(B, r)*, ritorna la posizione dell'r-esimo 1 in B

Esempio: Sia il bitvector $B = [0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1]$. Allora $rank1(B, 5) = 1$ e $select1(B, 5) = 10$.

Si suppone che la distribuzione delle occorrenze di un certo suffisso locale sia data da un bitvector B di dimensione $|ICFL|$ definito in modo tale che il bit i-esimo è 1 se e solo se il fattore i-esimo contiene un'occorrenza del suffisso locale (con $1 \leq i \leq k$), 0 altrimenti.

Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baa, caabca, dcaabca \rangle$. Sia il suffisso locale $x = bca$, allora $B_x = [0, 0, 1, 1]$.

2.2 Algoritmo *GetInsertionTarget*

Procederemo ora a descrivere e illustrare l'algoritmo per reperire l'insertion target di $[x]_g - list$ all'inserimento di $[z]_g - list$, dati i loro bitvector B_z e B_x.

Come primo passo troveremo l'indice del fattore m_i più a sinistra contenente il suffisso locale z, attraverso la funzione $select1(B_z, 1)$. Distingueremo due differenti casi: $(i < k)$ e $(i = k)$.

1. $i < k$
   Se il suffisso locale x occorre nell'ultimo fattore m_k, allora la prima posizione in $[x]_g - list$ si riferisce a tale occorrenza e, per il Teorema 1.3, l'insertion target h risulterà essere maggiore di 1, in quanto $S_{x,k} \prec S_{z,*}$.

Sia u la posizione in $[x]_g - list$ che si riferisce all'occorrenza più a sinistra di x in un fattore m_j tale che $i \leq j < k$ (cioè, m_j è il fattore più a sinistra tra quelli da m_i a m_{k-1} contenente x come suffisso).
Se nessun fattore tra m_i a m_{k-1} contiene x come suffisso, allora u sarà uguale a $|[x]_g - list| + 1$. Dal Corollario 4 deriva che l'insertion target h deve essere minore o uguale a u, cioè u è un **upper bound** per h.
Per il calcolo di u necessito di conoscere:

(a) il numero di fattori fino a m_{i-1} in cui occorre x, che è uguale al numero di elementi $[x]_g - list$ che sono occorrenze di x nei fattori fino a m_{i-1} uguale a $rank1(B_x, i-1)$.

(b) se x occorre nell'ultimo fattore, quindi $B_x[k]$.

L'upper bound u sarà quindi uguale a $rank1(B_x, i-1) + B_x[k] + 1$.
È importante osservare che se nessun fattore da m_i a m_{k-1} contiene x, si ottiene (correttamente) un valore di u pari a $|[x]_g - list| + 1$. Per il Teorema 1.6, l'insertion target h sarà la posizione più a sinistra in $[x]_g - list$, minore di u, che corrisponde a un'occorrenza di x in un fattore m_q (con $q < i$), tale che m_{q+1} abbia un prefisso α con $|\alpha| = |y|$ che sia maggiore di y. Se tale posizione non esiste, avremo $h = u$.
Procederemo quindi ad analizzare le posizioni di $[x]_g - list$ a partire dalla posizione $u - 1$ fino ad arrivare alla posizione $B_x[k] + 1$.
Per ogni posizione p tra $u - 1$ e $B_x[k] + 1$, trovo l'indice q del fattore che contiene l'occorrenza di x contenuta nella posizione p di $[x]_g - list$, che è dato da $q = select1(B_x, p - B_x[k])$.
Considero il prefisso α lungo $|y|$ di m_{q+1} e lo confronto con y: se $\alpha \leq y$, allora mi fermo e si pone $h = p+1$, altrimenti vado avanti a considerare la posizione successiva.
In particolare, se per nessuna posizione p il prefisso α è minore o uguale di y, allora h sarà uguale a $B_x[k]+1$. Se $u-1 < B_x[k]+1$, non ci sono posizioni da esaminare e h sarà uguale a u.

2. $i = k$
Il calcolo dell'insertion target h avviene come per il caso $i < k$ considerando in questo caso un upper bound u uguale alla dimensione di $[x]_g - list$ incrementato di 1.

Algorithm 1 GetInsertionTarget

Procedure $GetInsertionTarget(B_z, B_x)$
$i \leftarrow Select(B_z, 1)$ ▷ m_i è il fattore più a sinistra che contiene $z = xy$ come suffisso
▷ Trovo l'upper bound u per l'insertion target h
if $i < k$ **then** ▷ Se m_i è diverso dall'ultimo fattore
$u \leftarrow Rank_1(B_x, i-1) + B_x[k] + 1$
else ▷ Se m_i coincide con l'ultimo fattore (z occorre solo sull'ultimo fattore)
$u \leftarrow Rank_1(B_x, |B_x|) + 1$ ▷ Dimensione di $[x]_g - list$ incrementata di 1
$p \leftarrow u - 1$
while $p \geq B_x[k] + 1$ **do**
▷ Trovo l'indice q del fattore che contiene l'occorrenza di x in posizione p di $[x]_g - list$
$q \leftarrow select_1(B_x, p - B_x[k])$
$\alpha \leftarrow m_{q+1}[: |y|]$ ▷ prefisso α del fattore m_{q+1} successivo a m_q
if $\alpha \leq y$ **then**
$p \leftarrow B_x[k]$
else
$p \leftarrow p - 1$
$h \leftarrow p + 1$ ▷ Insertion target
return h

2.3 Prefix-chain e Prefix-tree

Sia x_i un suffisso locale che occorre in almeno un fattore di $ICFL(T)$, con T testo definito su Σ. $C = \langle x_1, x_2, ..., x_c \rangle$ è una **prefix-chain** se x_i è prefisso di x_{i+1} per $1 \leq i < c$ e non esiste un suffisso locale non presente in C che è prefisso di x_{i+1} e contiene x_i come prefisso per $1 \leq i \leq c$. In altre parole, non esiste un suffisso locale che, se inserito in C in una qualche posizione purché diversa dall'ultima, mantiene C una prefix-chain.

Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baaba, cbaabac, dcaabacb \rangle$. Allora $C = \langle a, aba, abac, abacb \rangle$.

C viene definita **massimale** se non esiste un suffisso locale non presente in C che contiene x_c come prefisso, cioè C non può essere ulteriormente estesa con un suffisso x_{c+1}.

Data una prefix-chain $C = \langle x_1, x_2, ..., x_c \rangle$, si indica con $[x_1, x_2, \ldots, x_c]_g - list$ la lista delle occorrenze dei suffissi locali in C che è consistente con l'ordinamento dei suffissi globali che iniziano con tali suffissi, chiamata anche $C_g - list$.

Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baaba, cbaabac, dcaabacb \rangle$ con$C = \langle a, aba, abac, abacb \rangle$. Allora $C_g - list = \langle 3, 6, 12, 19 \rangle$.

Si può dimostrare che, per un certo x_i (con $1 \leq i < c$), tutti gli x_j per $j > i$ hanno lo stesso insertion target rispetto a x_i. Chiamo h_i l'insertion target di $[x_i]_g - list$ in $[x_{i-1}]_g - list$ per $1 < i \leq c$. Di conseguenza per calcolare $C_g - list$ si può procedere a ritroso: si calcola h_c e si inserisce $[x_c]_g - list$ in $[x_{c-1}]_g - list$ immediatamente prima della posizione h_c per ottenere $[x_{c-1}, x_c]_g - list$, poi si calcola h_{c-1} e si inserisce $[x_{c-1}, x_c]_g - list$ immediatamente prima della posizione h_{c-1} in $[x_{c-2}]_g - list$ per ottenere $[x_{c-2}, x_{c-1}, x_c]_g - list$.
Alla generica iterazione, si calcola dunque h_i (insertion target di $[x_i]_g - listin[x_{i-1}]_g - list$) e si inserisce $[x_i, \ldots, x_{c-1}, x_c]_g - list$ immediatamente prima della posizione h_i in $[x_{i-1}]_g - list$ per ottenere $[x_{i-1}, x_i, \ldots x_{c-1}, x_c]_g - list$. All'ultima iterazione, si calcola h_2 e si inserisce $[x_2, \ldots, x_{c-1}, x_c]_g - list$ immediatamente prima della posizione h_2 in $[x_1]_g - list$ per ottenere $[x_1, x_2, \ldots x_{c-1}, x_c]_g - list$ corrispondente a $C_g - list$.

Si considerino a questo punto due catene C_1 e C_2 che condividono i primi suffissi locali. Sia x il suffisso comune più a destra e siano $z_1 = xy_1$ e $z_2 = xy_2$ i due suffissi seguenti rispettivamente in C_1 e C_2, supponendo $y_1 < y_2$. Chiamiamo $[C_1, C_2]_g - list$ l'unione di $C_{1g} - list$ e $C_{2g} - list$ consistente con l'ordinamento dei suffissi globali che iniziano con i suffissi locali di tali catene.
Si può dimostrare che ogni suffisso globale che inizia con un suffisso locale di C_2 da z_2 in poi, deve seguire ogni suffisso globale che inizia con un suffisso locale di C_1 da z_1 in poi.

Per calcolare $[C_1, C_2]_g - list$, si procede a ritroso lungo C_1 nel modo precedentemente descritto (per ricostruire la lista della singola prefix-chain) fermandosi non appena viene costruita $[z_1, ...]_g - list$. Si procede analogamente per ricostruire $[z_2, ...]_g - list$ relativamente a C_2, in seguito si inseriscono $[z_1, ...]_g - list$ e $[z_2, ...]_g - list$ in $[x]_g - list$ e la lista ottenuta viene inserita per intero nella lista relativa al suffisso x' (comune alle due catene) immediatamente precedente a x. Infatti, l'insertion target di z_1 e z_2 in x' coincide con quello di x in x' e con tutti quelli che precedono x' nel parte condivisa dalle due catene. Si procede poi a ritroso con inserimenti successivi lungo la parte comune.

È possibile rappresentare i suffissi locali relativi ad una $ICFL(T)$ come un albero così definito:

Prefix tree:

1. albero radicato dove la radice rappresenta il suffisso locale vuoto

2. Ogni suffisso locale è rappresentato da un nodo del tree
3. Il padre di un nodo z è il suffisso locale x più lungo che è prefisso di z

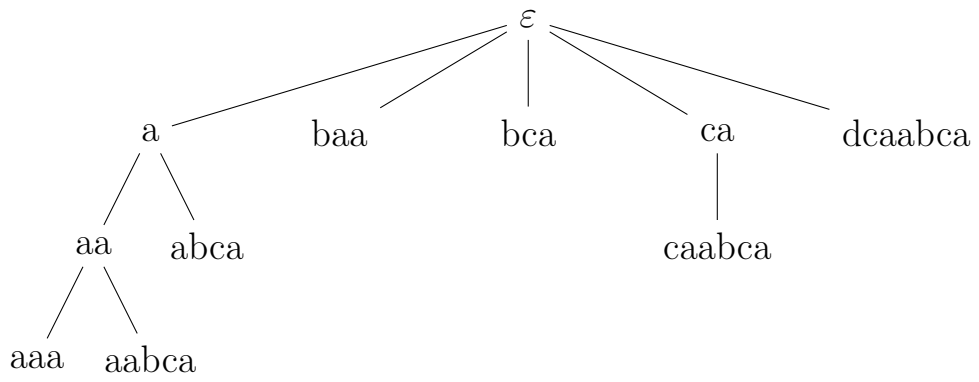
Esempio: Sia T testo definito su $\Sigma = a, b, c, d$ con $a < b < c < d$ e sia $ICFL(T) = \langle aaa, baa, caabca, dcaabca \rangle$. Estraendo tutti i suffissi locali otterremo le seguenti prefix-chain massimali non banali:

- $C_1 = \langle a, aa, aaa \rangle$
- $C_2 = \langle a, aa, aabca \rangle$
- $C_3 = \langle a, abca \rangle$
- $C_4 = \langle ca, caabca \rangle$

e le seguenti prefix-chain banali, costituite un singolo suffisso locale, poiché non vi è alcun altro suffisso locale che possa essere prefisso del suffisso all'interno della catena:

- $C_5 = \langle baa \rangle$
- $C_6 = \langle bca \rangle$
- $C_7 = \langle dcaabca \rangle$

Seguendo la definizione sopra descritta, risulta possibile costruire il prefix-tree corrispondente:



2.4 Algoritmo *BuildPrefixTree*

L'algoritmo sotto illustrato provvede a costruire il prefix-tree di un testo T data la sua fattorizzazione inversa.

Dopo aver inizializzato il suffisso vuoto ε, definiamo due insiemi distinti: V, rappresentante i nodi del prefix-tree, e A, rappresentante gli archi del prefix-tree. Inizialmente, V conterrà solo ε, mentre A sarà vuoto.

Attraverso un ciclo *for*, che itererà in modo crescente su una variabile l fino alla lunghezza massima L tra i fattori di *ICFL*, procederemo con l'estrazione e lo studio dei suffissi aventi lunghezza pari a tale parametro. Utilizzeremo due hash map di supporto per tenere traccia dei suffissi analizzati ad ogni iterazione:

- *suffix map*, dove ogni valore risulta essere rappresentato dalla coppia $\langle suffisso, g-list \rangle$
- *f distribution map*, il cui contenuto é costituito da coppie $\langle suffisso, bitvector \rangle$.

Al termine di ogni iterazione,coincidente con il termine dell'estrazione di tutti i suffissi aventi lunghezza esattamente l, entrambe le mappe verranno svuotate.
Data la variabile i identificativa del fattore analizzato, distingueremo due casi:

1. $l \leq |m_i|$
   Se la lunghezza del suffisso risulta essere minore o uguale a quella del fattore m_i, procederemo con l'estrazione del suffisso nel fattore. Successivamente verrà calcolata l'occorrenza del suffisso in m_i e, se non già presente nella *suffix map*, verrà aggiunto. Se il suffisso risulta essere già stato inserito in precedenza, l'occorrenza verrà aggiunta in coda alla lista delle occorrenze presenti del suffisso studiato solo se $i \neq k$; altrimenti, in testa, secondo il Teorema 1.3. Verrà poi aggiornato il bit in riferimento al fattore analizzato nella *f distribution map*, impostandolo a 1.
2. $l > |m_i|$
   Se la lunghezza del suffisso risulta essere maggiore della lunghezza del fattore, non si potrà procedere con l'estrazione.

Successivamente, per ogni suffisso s presente nella *suffix map*, si procederà con l'aggiunta dei suffissi all'albero.

Sia p il suffisso più lungo, corrispondente al nodo più profondo, che funge da prefisso di s. Poniamo s figlio di p aggiungendo s all'insieme V e l'arco (p, s) ad A.
Proseguiamo con l'iterazione successiva attraverso l'incremento di l da parte del ciclo *for*.

Algorithm 2 BuildPrefixTree

```
INPUT: ⟨m_1, m_2, ..., m_k⟩, fattorizzazione ICFL di un testo T
OUTPUT: (V, A), prefix tree
ε ← empty local suffix
V ← {ε}                                          ▷ insieme dei nodi del prefix-tree
A ← {}                                           ▷ insieme degli archi del prefix-tree
for l from 1 to L do
    //ricavo i suffissi di lunghezza l
    suffix_map ← empty hash
    f_distribution_map ← empty hash
    for i from 1 to K do
        if l ≤ |m_i| then
            //estrazione dei suffissi di lunghezza l
            s ← m_i[|m_i| − l + 1                                   ▷ suffisso
            length ← total length of factors m_1, m_2, ..., m_i
            occ ← length − l + 1          ▷ occorrenza del suffisso di lunghezza l in m_i
            if s not in suffix_map then
                suffix_map[s] ← occ
                f_distribution_map ← inizialize a bitvector with k 0s
            else
                if i is not k then
                    append occ to suffix_map[s]
                else
                    prepend occ to suffix_map[s]
            f_distribution_map[s][i] ← 1
    //Aggiunta dei suffissi locali di lunghezza l
    for each s in suffix_map do
        p ← deepest node in V that is prefix of s                   ▷ parent
        V ← V ∪ {s}
        A ← A ∪ {(p, s)}
return (V, A)
```

2.5 Algoritmo BuildSuffixArray

Il passo successivo dell'algoritmo di costruzione prevede la visita dei nodi dell'suffix-tree ,precedentemente costruito, al fine di realizzare il suffix array di riferimento. L'algoritmo è suddiviso funzioni differenti: *BuildSuffixArray-Helper* e *BuildSuffixArray*:

- *BuildSuffixArrayHelper*: Questa funzione è responsabile della visita e dell'aggiornamento delle liste dei nodi. Ad ogni nodo sono associate due informazioni: la *g-list* corrispondente e l'*insertion target* in riferimento al padre. La visita in profondità avviene iterando sui nodi figli in ordine lessicografico decrescente. Questa decisione realizzativa è conseguenza della semplicità con cui le liste dei nodi figli verranno inserite in quelle del padre. Le relazioni di precedenza tra suffissi aventi un prefisso comune si propagano ai loro *insertion target* di riferimento. Prevedendo un inserimento delle liste decrescente, sarà evitato l'aggiornamento degli *insertion target* dei nodi che condividono lo stesso padre. Al raggiungimento di un nodo foglia, verrà inserita la lista del nodo analizzato in quella del padre nella rispettiva posizione indicata dall'*insertion target*.

Algorithm 3 BuildSuffixArrayHelper

INPUT: *node*, nodo del *suffix-tree*
OUTPUT: //
//visita in profondità
for i from $|node.children|$ to 1 **do**
    $BuildSuffixArray(node.children[i])$
$h \leftarrow node.insertionTarget$
insert $node.g-list$ in $parent.g-list$ at position h

- *BuildSuffixArray*: Al termine dell'esecuzione della funzione precedente, ogni nodo avrà la propria *g-list* aggiornata. La *g-list* della radice ε del *suffix tree* corrisponderà al Suffix Array della *ICFL(T)* su cui è stato realizzato l'albero, il quale verrà restituito da questa funzione.

Algorithm 4 BuildSuffixArray

INPUT: ε, radice del *suffix-tree*
OUTPUT: SA, suffix array
$BuildSuffixArrayHelper(\varepsilon)$
return $\varepsilon.g-list$

Capitolo 3

Implementazione

La fase di implementazione ha previsto la scrittura di un diverse funzioni e classi, in C++ 20, per la costruzione del *Suffix Array* a partire dalla fattorizzazione inversa di Lyndon di un testo T.

L'intero progetto consente la lettura di un file input.txt, contenente i fattori della fattorizzazione inversa di Lyndon (*ICFL*). Tramite le funzioni e le classi sviluppate, viene prodotto in output il *Suffix Array* di riferimento.

3.1 Struttura del progetto

Il progetto è strutturato nelle seguenti file e directory:

- *main.cpp*: contiene il punto di ingresso del programma e i test di verifica per il corretto funzionamento delle funzioni realizzate.
- *func.hpp*: include le funzioni descritte nel capitolo precedente, le funzioni di stampa e alcune funzioni di supporto per la comunicazione tra le diverse componenti del progetto.
- *Node.hpp*: definisce la classe Node, rappresentante un nodo del suffix tree, con i metodi e gli attributi necessari affinché ogni nodo rappresenti un suffisso utile per la costruzione del Suffix Array.
- *Tree.hpp*: definisce la classe Tree, rappresentante il suffix tree.
- *input.txt*: file di testo contenente i fattori della fattorizzazione inversa di Lyndon (*ICFL*) del testo *T*.

Il progetto utilizza le seguenti librerie:

- Librerie standart C++:
  `<iostream>`: per le funzionalità input e output.
  `<fstream>`: per la gestione dei file.
  `<vector>`: per l'utilizzo di vettori dinamici.
  `<string>`: per la gestione delle stringhe.

`<list>`: per l'utilizzo delle liste.
`<unordered_map>`: per l'utilizzo delle hash map.

- `<pasta/bitvector>`[9] Libreria esterna per le funzionalità degli oggetti bitvector.

3.2 Flusso di esecuzione

Il flusso di esecuzione del progetto può essere riassunto nei seguenti passaggi:

1. Lettura del file di input:
   Attraverso la funzione `std::list<std::string> build_input_ICFL` viene prevista la lettura dei fattori *ICFL* e la costruzione di una lista di stringhe rappresentante la fattorizzazione.
2. Realizzazione Suffix Tree:
   La lista ottenuta in precedenza viene passata come parametro alla funzione `Tree build_suffix_tree`. Questa funzione costruisce il Suffix Tree, creando i nodi-suffissi attraverso il costruttore appropriato e stabilendo le dipendenze necessarie tra di essi per la realizzazione del *Suffix Array*.
3. Generazione Suffix Array:
   La funzione `std::vector<int> build_suffix_array`, attraverso una visita in profondità dell'albero passato come parametro, aggiorna le *g-list* di ogni nodo con la liste dei propri figli secondo i criteri precedentemente esposti. Restituisce successivamente la *g-list* della radice dell'albero, corrispondente al *Suffix Array*.

3.3 Osservazioni e Problemi Riscontrati

Nel processo di sviluppo delle classi, è stata prestata particolare attenzione alla gestione della memoria, evitando potenziali memory leak e assicurandosi che le risorse fossero deallocate correttamente.

Durante l'utilizzo degli oggetti `pasta/bitvector`, è stato necessario apportare alcune modifiche ai file presenti. La libreria non prevedeva iteratori di tipo costante, rendendo complesso il passaggio di questi oggetti alle funzioni. Di conseguenza, è stato necessario implementare i *const iterator* nella classe `bitvector.hpp`. Inoltre, sono stati esplicitamente eliminati il costruttore di

copia e l'operatore di assegnazione, rendendo difficoltosa la costruzione di un nodo avente come parametro un oggetto BitVector e la modifica dei suoi bit. Tuttavia, è possibile modificare l'oggetto tramite un puntatore, poiché l'accesso e la modifica diretta degli oggetti esistenti non sono influenzati da queste eliminazioni.

Capitolo 4

Conclusioni

In questo lavoro di stage, è stata studiata ed implementata una soluzione per la costruzione del Suffix Array basata su una particolare fattorizzazione di un testo, denominata Fattorizzazione Inversa di Lyndon (*ICFL*). Sono stati analizzati e studiati i teoremi fondamentali atti all'ordinamento dei suffissi locali e sviluppata una struttura ad albero per la rappresentazione delle dipendenze.

Lo studio teorico dei principali teoremi approfonditi e delle strutture dati utilizzate ha guidato la progettazione delle funzioni utili alla costruzione del Suffix Array, confermando l'efficacia e la validità dell'approccio proposto, permettendo la traduzione in codice C++.

Il lavoro presenta alcune limitazioni, in particolare per quanto riguarda la complessità e il tempo richiesti per la costruzione dell'albero dei suffissi locali e l'implementazione del Suffix Array, che aumentano in modo significativo con l'incremento del numero dei fattori ICFL.

Durante questo progetto di stage, ho acquisito numerose competenze e conoscenze che hanno arricchito il mio bagaglio culturale e professionale.
Ho migliorato la conoscenza del linguaggio C++, approfondendo l'uso delle sue caratteristiche, imparando a valutare la soluzione migliore a seconda del problema riscontrato. Inoltre, ho imparato a scrivere in pseudocodice e a valutare la complessità temporale degli algoritmi, il che ha facilitato la traduzione da concetti teorici in implementazioni pratiche.
Un ulteriore aspetto fondamentale del mio apprendimento è stata la capacità di saper leggere e interpretare file esterni scritti da altri utenti, sviluppando la competenza di integrare e comprendere il lavoro altrui nei miei progetti. Questo è stato particolarmente utile nella fase di debugging per la risoluzione dei problemi. Infine, ho acquisito la capacità di valutare e gestire i tempi necessari per lo sviluppo, migliorando la pianificazione e la gestione del progetto. Ho appreso a comunicare in modo efficace con il tutor, esponendo i problemi incontrati e facendo tesoro dei feedback ricevuti per il progresso del lavoro.

Bibliografia

[1] P. Bonizzoni et al. «Inverse Lyndon words and inverse Lyndon factorizations of words». In: (2018). Accepted: 2018-09-27T09:44:40Z Publisher: Academic Press Inc. DOI: 10.1016/j.aam.2018.08.005. URL: https://boa.unimib.it/handle/10281/206213?mode=full.416 (visitato il giorno 13/07/2024) (cit. a p. 4).

[2] P. Bonizzoni et al. «Numeric Lyndon-based feature embedding of sequencing reads for machine learning approaches». In: *Information Sciences* 607 (1 ago. 2022), pp. 458–476. ISSN: 0020-0255. DOI: 10.1016/j.ins.2022.06.005. URL: https://www.sciencedirect.com/science/article/pii/S0020025522005795 (visitato il giorno 13/07/2024) (cit. a p. 4).

[3] J. Cleary e I. Witten. «Data Compression Using Adaptive Coding and Partial String Matching». In: *IEEE Transactions on Communications* 32.4 (apr. 1984). Conference Name: IEEE Transactions on Communications, pp. 396–402. ISSN: 1558-0857. DOI: 10.1109/TCOM.1984.1096090. URL: https://ieeexplore.ieee.org/abstract/document/1096090 (visitato il giorno 13/07/2024) (cit. a p. 9).

[4] Gawrychowski, P., & Kociumaka. «Lyndon factorization of trees and applications». In: *Lyndon factorization of trees and applications.* (2012), pp. 350–359. (Cit. a p. 3).

[5] Alexander Golynski. «Optimal lower bounds for rank and select indexes». In: *Theoretical Computer Science*. The Burrows-Wheeler Transform 387.3 (22 nov. 2007), pp. 348–359. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2007.07.041. URL: https://www.sciencedirect.com/science/article/pii/S0304397507005312 (visitato il giorno 13/07/2024) (cit. a p. 9).

[6] *High-Order Entropy-Compressed Text Indexes*. URL: https://kuscholarworks.ku.edu/handle/1808/7192 (visitato il giorno 13/07/2024) (cit. a p. 9).

[7] Udi Manber e Gene Myers. «Suffix arrays: a new method for on-line string searches». In: *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. SODA '90. USA: Society for Industrial e Applied Mathematics, 1 gen. 1990, pp. 319–327. ISBN: 978-0-89871-251-3. (Visitato il giorno 13/07/2024) (cit. a p. 3).

[8] Paola Paola Bonizzoni et al. «On the longest common prefix of suffixes in an inverse Lyndon factorization and other properties». In: *Theoretical Computer Science*. A Fascinating Rainbow of Computation – Honoring Gheorghe Păun on the Occasion of His 70th Birthday 862 (16 mar. 2021), pp. 24–41. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2020.10.034. URL: https://www.sciencedirect.com/science/article/pii/S0304397520306253 (cit. alle pp. 3, 4).

[9] *pasta-toolbox/bit_ vector*. original-date: 2021-10-20T15:13:09Z. 7 Lug. 2024. URL: https://github.com/pasta-toolbox/bit_vector (cit. a p. 18).

Ringraziamenti

Vorrei riservare questo spazio finale ai ringraziamenti verso tutti coloro che hanno contribuito al raggiungimento di questo importante traguardo, rendendo il mio percorso più agevole e gratificante grazie al loro prezioso sostegno.

Un sentito ringraziamento va alla Professoressa Rizzi, che mi ha guidato con infinita pazienza e professionalità nella realizzazione di questo elaborato. I suoi preziosi consigli e la grande disponibilità dimostrata sono stati fondamentali per il successo di questo percorso.

Grazie a Mamma e Papà, per aver creduto in me più di quanto non lo abbia fatto io e per avermi insegnato che nella vita ogni ostacolo rappresenta un'opportunità di crescita. Il vostro amore e sostegno incondizionato è stata la chiave per la riuscita di questa mia crescita professionale e soprattutto personale.

Grazie a Maura, che, nonostante lo dica poco, è sempre stata un punto di riferimento. I tuoi consigli, la tua gioia e la tua capacità di mettere a soqquadro qualsiasi cosa mi hanno fatto comprendere che ammirare qualcuno significa imparare qualcosa da quest'ultimo. La tua autenticità e il tuo affetto sono stati fondamentali.

Grazie a Nonna Tonia, Nonna Flavia e a tutti i parenti che mi sono stati vicino. Essere parte di questa grande famiglia mi ha arricchito di preziosi insegnamenti e di affetto sincero. Ogni istante condiviso insieme è stato un dono di gioia e sostengo.

Grazie ad Alessandro, per avermi sopportato ed aiutato ogni volta che ne ho avuto bisogno e per essere stato in grado di trasmettermi la passione per questa materia. Le sessioni, i progetti e i drammatici momenti vissuti insieme resteranno un ricordo indelebile di questo cammino.

Grazie a Marco, ad Ajo e agli amici che ho incontrato in questi anni. Per aver condiviso con me gioie e dolori, alleviando e rendendo unico questo percorso.

Grazie a Matthew, per essere sempre stato presente. La nostra amicizia mi ha garantito un sostegno sincero e costante. Le risate e le avventure condivise rimarranno una ricchezza che custodirò gelosamente, più prezioso di qualsiasi leggendario tesoro disperso tra i mari e raccontato nelle storie.

Grazie a Luca, il mio angelo custode.
Desidero dedicare questo traguardo a te e a noi, ancora bambini in riva al mare. Mi hai sempre spronato a essere la migliore versione di me stesso, che nemmeno io sono mai riuscito a vedere. Sei la persona che più sento vicina in questo momento di gioia.