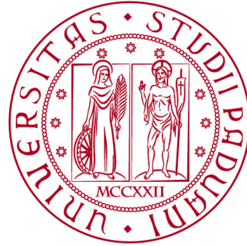

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di laurea in Ingegneria dell'Informazione

# Analisi sperimentale delle prestazioni di un suffix tree troncato

Laureando

**Daniel Zucchetto**

Relatore

**Dott.ssa Cinzia Pizzi**

Anno Accademico 2011/2012

# Indice

**1 Introduzione** **1**

**2 Strutture dati basate sui suffissi** **3**
2.1 Suffix tree . . . . . 3
2.1.1 Algoritmi lineari per la costruzione di un Suffix Tree . . . . 5
2.1.2 Codifica illi . . . . . 7
2.2 Suffix tree troncato . . . . . 9
2.2.1 Algoritmi di costruzione . . . . . 9
2.2.2 Codifica derivata da illi . . . . . 10
2.2.3 Codifica TruST . . . . . 10
2.3 Suffix Array . . . . . 13
2.4 Confronto tra suffix tree ed enhanced suffix array . . . . . 17

**3 Analisi sperimentale** **19**
3.1 Descrizione delle sequenze analizzate . . . . . 19
3.2 Sequenze casuali . . . . . 21
3.3 Sequenze non casuali . . . . . 29
3.3.1 Sequenze aaa e alphabet . . . . . 29
3.3.2 Sequenze AE000111, asyoulik e pi . . . . . 35

**4 Conclusioni** **45**

**Bibliografia** **47**

**Sommario**

Le strutture dati sono un ambito di ricerca estremamente vivo, poiché adatte a risolvere un ampio numero di problemi d'indicizzazione. Tra le varie strutture dati, particolare rilevanza hanno le strutture dati basate sui suffissi, che si prestano particolarmente bene alla ricerca/scoperta di pattern su stringhe e alla loro caratterizzazione. Attualmente esistono diverse strutture dati di questo tipo e non è facile capire caso per caso quale sia la migliore da utilizzare. Per facilitare tale compito, questa tesi si propone di analizzare le prestazioni in termini di occupazione di spazio di diverse strutture dati basate sui suffissi, effettuando un'analisi comparativa tra di esse e prestando particolare attenzione al *suffix tree troncato*, sia con la codifica derivata da illi che con la codifica TruST, e al *suffix array* in versione *enhanced*. I risultati ottenuti hanno mostrato come la scelta tra le strutture dati enhanced suffix array e suffix tree troncato dipenda dal valore $k$ di troncamento dell'albero (ovvero la lunghezza massima dei pattern oggetto di studio). Inoltre, l'occupazione di spazio del suffix tree troncato è influenzata da fattori come il tipo di sequenza da indicizzare, la sua lunghezza e la cardinalità del suo alfabeto. D'altra parte, l'occupazione di spazio dell'enhanced suffix array è fortemente dipendente dal tipo di applicazione, necessitando la memorizzazione di tabelle aggiuntive. In conclusione, la scelta della struttura dati che fornisce prestazioni migliori non è univoca, ma può essere effettuata utilizzando questi parametri come criteri di valutazione.

# Capitolo 1

# Introduzione

Le strutture dati basate sui suffissi rappresentano un'area di ricerca estremamente importante per quanto riguarda i problemi di ricerca di sottostringhe all'interno di sequenze più grandi. In generale, questo tipo di strutture permette la soluzione in modo efficiente di algoritmi di tipo *pattern matching*, ma riesce a comportarsi altrettanto bene con diversi tipi di algoritmi operanti su sequenze. Verranno forniti ulteriori esempi di applicazioni di tali strutture dati all'inizio del Capitolo 2.

Nel tempo sono state definite diverse strutture dati basate sui suffissi, tra cui le più diffusamente utilizzate sono il *suffix tree* e il *suffix array*. Tra queste strutture dati sembra essere inspiegabilmente trascurato dalla letteratura il *suffix tree troncato*. Questa struttura dati potenzialmente combina due aspetti chiave delle due strutture dati sopra citate: la struttura ad albero del suffix tree (che però richiede molto spazio) e la ridotta occupazione spaziale (seppur in dipendenza del parametro di troncamento), caratteristica, quest'ultima, che negli ultimi anni ha favorito l'utilizzo e la diffusione del suffix array. Questa tesi si propone, quindi, di esaminare sperimentalmente le prestazioni in termini di occupazione di spazio di un suffix tree troncato, al variare di diversi parametri, e di confrontare questa struttura dati con il suffix array in versione *enhanced*, che garantisce la stessa complessità temporale in fase di elaborazione del suffix tree (troncato o non).

La tesi è organizzata come segue. Nel Capitolo 2 verranno descritte le strutture dati basate sui suffissi. In particolare verranno esaminati il suffix tree, il suffix tree troncato ed il suffix array. Di ogni struttura dati verranno descritti gli algoritmi di costruzione e le codifiche adottate per implementarli. Nel Capitolo 3 verrà descritta l'analisi sperimentale effettuata. All'inizio si descriveranno le sequenze scelte per l'analisi e le motivazioni per la loro scelta. Quindi si analizzeranno i risultati degli esperimenti svolti prima su sequenze casuali e poi su sequenze non casuali. Al termine verranno fatte delle osservazioni sui risultati ottenuti, comparando le prestazioni delle strutture dati in esame ed argomentando sulla scelta di quale struttura dati utilizzare in base al tipo di sequenze che vi si devono memorizzare.

# Capitolo 2

# Strutture dati basate sui suffissi

In questo capitolo verranno descritte le principali strutture dati basate sulla memorizzazione dei suffissi di una sequenza: *suffix tree*, *suffix tree troncato* e *suffix array*.

Le strutture dati basate sui suffissi hanno vasti ambiti di applicazione, principalmente nell'indicizzazione di testi, per poter effettuare successivamente delle operazioni in tempo efficiente. In [11] sono elencate numerose applicazioni, tra cui la ricerca esatta di sottostringhe e la compressione con il metodo Ziv-Lempel. Ulteriori importanti applicazioni riguardano l'esecuzione di algoritmi di tipo *profile matching* su sequenze biologiche [9], il *motif discovery* [4], il *co-occurrence count* [16, 12, 6, 7] e la ricostruzione di alberi filogenetici [5].

La prima struttura dati presentata è il suffix tree, storicamente la prima ad essere stata utilizzata. Del suffix tree verrà analizzata in particolare la codifica *illi* (improved linked list implementation) [14], che è attualmente la codifica più efficiente per i suffix tree. In seguito verrà presentata una variante del suffix tree, il suffix tree troncato, che, memorizzando solo alcuni suffissi della sequenza, riesce ad ottenere dei notevoli vantaggi in termini di occupazione di spazio. Di tale struttura dati verrà analizzata sia una variante [2] della codifica illi, sia una sua evoluzione, la codifica TruST [9]. Al termine del capitolo verrà introdotto il suffix array, che è una trasposizione del suffix tree basata sull'uso di array e che riesce a sua volta a garantire alcuni vantaggi rispetto alla struttura dati da cui storicamente deriva.

## 2.1 Suffix tree

In questa sezione si daranno innanzitutto le definizioni fondamentali relative al suffix tree e successivamente si descriveranno brevemente gli algoritmi di costruzione lineari e la codifica illi [14].

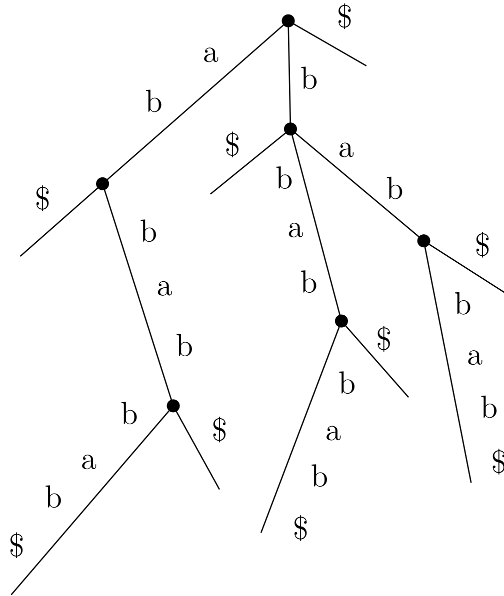


Figura 2.1: Suffix tree per la sequenza *abbabbab*\$

**Definizione 1.** *Data una sequenza $S$ di lunghezza $n$ definita su un alfabeto $\Sigma$, il suffix tree $T$ di $S$ è un albero con le seguenti proprietà:*

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

Queste proprietà non garantiscono, però, l'esistenza di una foglia per ogni suffisso della sequenza. Infatti, si supponga che il suffisso $i$-esimo sia anche prefisso del suffisso $j$-esimo. Allora il percorso relativo al prefisso $i$-esimo non terminerà su una foglia, ma su un punto intermedio del percorso che parte dalla radice e arriva alla foglia relativa al suffisso $j$-esimo. Per prevenire che ciò accada si costruisce il suffix tree non per $S$, ma per $S\$$, dove \$ è un simbolo appartenente a $\Sigma$ ma non presente in $S$ (Figura 2.1). Questo garantisce che l'albero abbia $n$ foglie e al massimo $n-1$ nodi interni, quindi un numero lineare di nodi rispetto alla lunghezza della sequenza. \$ viene chiamato simbolo terminale.

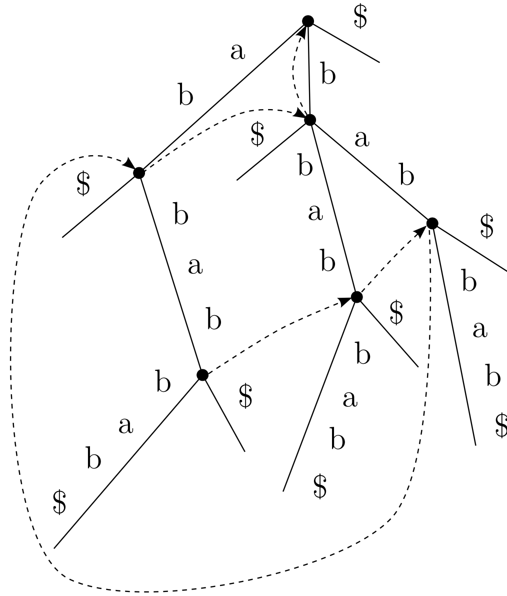


Figura 2.2: Suffix tree per la sequenza *abbabbab*\$. I suffix link sono disegnati in linea tratteggiata

Per ottenere un'occupazione di spazio lineare nella lunghezza di $S$ è tuttavia necessario attuare un ulteriore accorgimento. Le etichette degli archi non devono essere memorizzate come stringhe, altrimenti lo spazio occupato diventerebbe quadratico, bensì si devono memorizzare come coppia $(i, j)$ se l'etichetta coincide con la sottostringa $S[i..j]$.

Prima della descrizione della codifica e degli algoritmi di costruzione è necessario dare una definizione che tornerà utile anche in seguito.

**Definizione 2.** *Sia $x\alpha$ una stringa arbitraria, dove $x$ denota un singolo carattere ed $\alpha$ denota una stringa (anche nulla). Si consideri un qualsiasi nodo interno $v$ la cui concatenazione delle etichette dalla radice al nodo stesso sia $x\alpha$. Se esiste un altro nodo $s(v)$ la cui concatenazione delle etichette dalla radice al nodo stesso è $\alpha$, allora un puntatore da $v$ a $s(v)$ è chiamato suffix link.*

In Figura 2.2 si può osservare un suffix tree con i suffix link relativi.

### 2.1.1 Algoritmi lineari per la costruzione di un Suffix Tree

La costruzione del suffix tree per una sequenza di lunghezza $n$ può essere eseguita in $O(n)$. Gli algoritmi lineari per la costruzione di suffix tree sono quelli di Weiner [21], McCreight [18] e Ukkonen [20].

**Algoritmo di Ukkonen**

L'algoritmo di Ukkonen si basa sulla costruzione successiva di suffix tree impliciti, di cui ora si dà la definizione, e trasformandone l'ultimo in un suffix tree.

**Definizione 3.** *Un suffix tree implicito per una stringa $S$ è un albero ottenuto dal suffix tree di $S\$$ rimuovendo ogni simbolo terminale $\$$ dai rami dell'albero, poi rimuovendo ogni ramo che ha un'etichetta vuota e quindi eliminando ogni nodo che non abbia almeno due figli.*

*Un suffix tree implicito per un prefisso $S[1..i]$ di $S$ è analogamente definito prendendo il suffix tree di $S[1..i]\$$ ed eliminando i simboli terminali, i rami ed i nodi come descritto precedentemente.*

Si consideri di voler costruire il suffix tree per una stringa $S$ di lunghezza $n$. L'algoritmo si divide in $n$ fasi eseguite in ordine crescente. Nella fase $i+1$ viene costruito il suffix tree implicito $I_{i+1}$ di $S[1..i+1]$ partendo dal suffix tree implicito $I_i$. Ogni fase $i+1$ è ulteriormente suddivisa in $i+1$ estensioni, una per ciascuno degli $i+1$ suffissi di $S[1..i+1]$. L'estensione $j$ della fase $i+1$ trova il termine del percorso che, partendo dalla radice, è etichettato con la sottostringa $S[j..i]$. L'albero viene quindi esteso aggiungendo il simbolo $S[i+1]$ al termine dell'etichetta del percorso trovato, a meno che esso non sia già presente in quella posizione. Le estensioni vengono anch'esse eseguite in ordine crescente, con l'estensione $i+1$ che mette semplicemente il simbolo $S[i+1]$ nell'albero, sempre che non sia già presente.

Il punto cruciale dell'algoritmo è riuscire a trovare in modo efficiente il termine degli $i+1$ suffissi di $S[1..i]$. Un approccio naïve consiste nell'effettuare la ricerca, per ogni suffisso, partendo dalla radice dell'albero. Ciò porta la complessità dell'algoritmo ad $O(n^3)$, poichè il numero di estensioni da eseguire è proporzionale ad $n^2$ e, per ogni estensione, il tempo della ricerca è proporzionale ad $n$.

Un aiuto fondamentale deriva dall'uso dei suffix link. Avendo costruito i suffix link per l'albero $I_i$, nell'estensione $j$ della fase $i+1$ la ricerca del percorso etichettato con $S[j..i]$ avviene facilmente seguendo tali puntatori. Il funzionamento dettagliato è il seguente: alla fine di ogni fase $k$ si può conservare un puntatore al termine del percorso etichettato con $S[1..k]$. Per trovare il termine del percorso etichettato con $S[2..k]$ si parte dall'elemento indicato dal puntatore e si sale verso la radice fino ad incontrare il primo nodo interno $v$. Supponiamo che il ramo percorso abbia etichetta $\beta$. Seguendo il suffix link di $v$ si raggiunge $s(v)$ e si attraversa il sottoalbero con radice $s(v)$ seguendo il percorso etichettato con $\beta$. In questo modo la ricerca del percorso dalla radice etichettato con $S[2..k]$ non richiede più l'attraversamento dell'intero albero, ma solo di un suo sottoalbero. L'uso dei suffix link e l'implementazione di altri accorgimenti volti a migliorare la complessità temporale di caso peggiore portano ad avere una complessità dell'algoritmo proporzionale ad $n$.

**Algoritmo di McCreight**

L'algoritmo di McCreight deriva da quello sviluppato da Weiner. I miglioramenti che McCreight ha sviluppato permettono di ottenere una minor occupazione di spazio pur mantenendo una complessità temporale lineare nella lunghezza della sequenza.

L'algoritmo di McCreight costruisce il suffix tree $T$ di una sequenza $S$, di lunghezza $n$, inserendo i suffissi di $S$, uno alla volta, iniziando dal suffisso di indice 1, cioè quello coincidente con l'intera sequenza. Ad ogni fase, partendo dall'albero che contiene i suffissi di $S$ di indice $1, 2, 3, ..., i$, viene costruito l'albero che codifica i suffissi di $S$ di indice $1, 2, 3, ..., i, i + 1$.

L'implementazione naïve di questo algoritmo ha una complessità temporale di $O(n^2)$ ma, usando i suffix link ed altri accorgimenti implementativi, la complessità temporale diventa $O(n)$.

Sebbene l'algoritmo di McCreight possa sembrare piuttosto diverso da quello di Ukkonen, si sottolinea che lo strumento cardine in entrambi gli algoritmi è l'uso dei suffix link. Giegerich e Kurtz [10] hanno dimostrato come in realtà sia gli algoritmi di Ukkonen e McCreight, sia quello di Weiner, usino ulteriori concetti affini, presentando una visione d'insieme che unisce concettualmente i tre algoritmi citati.

### 2.1.2 Codifica illi

In questa codifica i nodi interni e le foglie sono memorizzati usando due diverse tabelle. In entrambe le tabelle gli indici dei nodi corrispondono all'ordine con il quale i nodi sono stati aggiunti durante la costruzione dell'albero. In particolare, l'indice di ogni foglia corrisponde alla posizione del corrispondente suffisso nella stringa.

La codifica di un nodo interno $N$ contiene le seguenti informazioni:

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

I nodi interni vengono a loro volta suddivisi in *large nodes* e *small nodes*. Durante la costruzione dell'albero, infatti, alcuni nodi interni possono essere creati in serie. Se durante una fase dell'algoritmo di costruzione $z$ nodi sono creati consecutivamente, i primi $z-1$ nodi sono chiamati small nodes, mentre lo $z$-esimo nodo è un large node. Supponiamo che tali nodi siano chiamati $N_1, ..., N_z$. Allora, per ogni nodo $N_i, 1 \leq i < z$, si ha che $N_{i+1}$ è il nodo a cui punta il suffix link di $N_i$, la lunghezza del percorso dalla radice a $N_i$ è pari alla lunghezza del percorso dalla radice a $N_z + (z-i)$ e $head(N_i) = head(N_z)-(z-i)$. Non è dunque necessario memorizzare tutte le informazioni per ogni nodo e per questo i large nodes occupano 16 byte, mentre gli small nodes, che devono solo memorizzare un puntatore al primo figlio, un puntatore al fratello destro e la distanza dal large node, occupano 8 byte ciascuno.

La codifica di una foglia $F$ contiene solo l'informazione riguardante il suo fratello destro. Questa informazione è sufficiente, poiché $head(F)$ e la lunghezza del suffisso a essa relativo possono essere ricavati direttamente dall'indice della foglia nella tabella. In questo modo l'occupazione di spazio di una foglia è pari ad 4 byte.

## 2.2 Suffix tree troncato

Nelle applicazioni pratiche non sempre è necessario conoscere tutti i suffissi di una sequenza. Spesso l'interesse è limitato a sottostringhe della sequenza di lunghezza limitata. Un esempio è la ricerca di pattern all'interno di sequenze biologiche, dove i pattern hanno una lunghezza molto piccola rispetto a quella dell'intera sequenza. Il suffix tree troncato [2, 19], come dice il nome, è un suffix tree che non memorizza completamente tutti i suffissi della sequenza. Le parti del suffix tree che non appaiono nel corrispettivo troncato dipendono da un valore chiamato *fattore di troncamento*, solitamente indicato con il simbolo $k$, secondo le seguenti regole:

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

Il vantaggio principale del suffix tree troncato riguarda i requisiti di spazio, che in esso sono minori rispetto a quelli del relativo suffix tree. In Figura 2.3 è illustrato il suffix tree troncato a $k = 3$ per la stessa sequenza del suffix tree di Figura 2.1.

### 2.2.1 Algoritmi di costruzione

La costruzione di un suffix tree troncato può essere effettuata, in maniera naïve, costruendo prima il suffix tree non troncato e poi eliminando le parti in eccesso, ottenendo così il relativo suffix tree troncato. Ovviamente questa procedura, richiedendo la costruzione in memoria dell'intero suffix tree, non migliora le performance in termini di occupazione di spazio rispetto ad esso. Prestazioni migliori si possono ottenere usando algoritmi sviluppati appositamente per il suffix tree

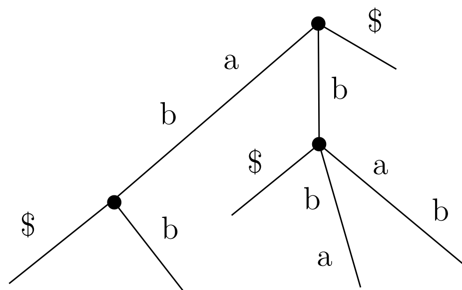


Figura 2.3: Suffix tree troncato con $k$=3 per la sequenza *abbabbab*$

troncato modificando gli algoritmi di McCreight e di Ukkonen [2]. In entrambi i casi non è richiesta la costruzione dell'intero suffix tree in memoria, riducendo l'occupazione di spazio richiesta a quanto effettivamente occupato dal suffix tree troncato.

### 2.2.2 Codifica derivata da illi

La codifica esposta in questo capitolo [2], è un adattamento della codifica illi.

La modifica principale rispetto alla codifica originale riguarda le foglie. Infatti, se per due o più suffissi della sequenza i primi k simboli coincidono, nel suffix tree troncato tali suffissi hanno in comune una singola foglia. Le foglie devono quindi poter ospitare le informazioni relative alla posizione di più di un suffisso. La soluzione a questo problema deriva dall'uso di liste concatenate per memorizzare le posizioni dei suffissi relativi alla stessa foglia.

Ogni foglia del suffix tree troncato è dunque in realtà composta da una lista concatenata, dove ogni elemento della lista contiene l'informazione relativa ad un suffisso. L'ultimo elemento della lista concatenata è chiamato *master leaf* e contiene il puntatore al fratello destro, se presente, altrimenti contiene il suffix link. Negli altri elementi della lista vengono usati 4 bit per memorizzare una parte dell'indice della master leaf corrispondente. In questo modo, visto che l'indice di una foglia richiede 27 bit, dopo aver attraversato i primi 7 elementi della lista (e dunque con un tempo costante) si può saltare direttamente alla master leaf e quindi al fratello destro.

L'occupazione di spazio per ciascun nodo interno è uguale a quella della codifica illi. Le foglie, sia master che non, occupano ognuna 4 byte.

### 2.2.3 Codifica TruST

La codifica appena presentata funziona per qualsiasi algoritmo su un suffix tree troncato. In molti casi, però, gli algoritmi sui suffix tree troncati non necessitano di tutte le informazioni in essi memorizzate. Nello specifico, la codifica TruST è stata pensata per funzionare con algoritmi che effettuano una visita in profondità dell'albero. Questo consente di liberare spazio non memorizzando le informazioni che permettono di effettuare altri tipi di visite dell'albero, in particolare i suffix link.

Ulteriori ottimizzazioni possono essere ottenute sfruttando le constatazioni a cui si giunge con il seguente ragionamento.

Si supponga di avere un puntatore $p$ verso un elemento di una struttura a puntatori. In tale elemento è presente il puntatore $q$ all'elemento successivo della struttura per poter accedere ad esso. Spesso, però, la memorizzazione della diffe-

renza $q - p$ richiede uno spazio inferiore rispetto alla memorizzazione di $q$. $q$ può poi essere ricavato semplicemente sommando $p$ alla differenza $q - p$.

La codifica TruST applica questa tecnica in ogni nodo dell'albero memorizzando la differenza tra indirizzi di memoria, invece degli indirizzi stessi, quando ciò è conveniente. Per ogni tipo di nodo esistono infatti delle versioni diverse a seconda che la differenza tra il proprio indirizzo e quello delle locazioni di memoria a cui punta sia o meno sotto una certa soglia. Per questo motivo, per ogni tipo di nodo non si potrà indicare un'occupazione di spazio costante, ma verrà fornito un intervallo nel quale tale occupazione può variare in base agli indirizzi assegnati ai nodi.

### Codifica delle foglie

La codifica TruST distingue tra tre tipi di foglie:

- *master leaf*: sono le foglie alle estremità destre delle liste concatenate;
- *small leaf*: sono le foglie intermedia nelle liste concatenate (potrebbero non essere presenti in tutte le liste);
- *large leaf*: sono le prime foglie delle liste concatenate (potrebbero non essere presenti in tutte le liste);

Come nella codifica precedente, la master leaf contiene il puntatore al fratello destro, se presente, mentre le altre foglie nella lista puntano all'elemento successivo. Per cercare di risparmiare spazio sulle foglie intermedie della lista viene memorizzata, ove sia vantaggioso, la differenza tra gli indirizzi di una foglia e della successiva, invece dell'effettivo indirizzo di quest'ultima. Questo accorgimento permette di risparmiare 1 byte sulle foglie dove è possibile applicarlo. Per distinguere se il valore memorizzato nella foglia è l'indirizzo del nodo successivo o la sua differenza con l'indirizzo della foglia attuale sono necessari dei bit di controllo. Per far spazio a tali bit vengono liberati i 4 bit per foglia che contenevano parte dell'indirizzo della master leaf relativa. Per poter comunque accedere in tempo costante alla master leaf viene memorizzato il suo indirizzo nella large leaf.

Ogni lista concatenata ha dunque sempre la master leaf. Se ci sono 2 elementi nella lista, il primo è una small leaf e il secondo è la master leaf. In questo caso, infatti, sarebbe inutile usare la large leaf, poiché l'indirizzo della foglia successiva coincide con quello della master leaf. Se ci sono tre o più foglie tutti i tipi di foglia sono presenti nella lista. L'occupazione di spazio per ogni tipo di foglia è la seguente:

- master leaf: 1, 3 o 4 byte;

- small leaf: 3 o 4 byte;
- large leaf: da 6 a 8 byte.

### Codifica dei nodi interni

La codifica dei nodi interni è simile alla codifica di Allali e Sagot, sono stati però implementati i seguenti accorgimenti per sfruttare più efficientemente lo spazio in memoria:

- come già accennato, non vengono memorizzati i suffix link;
- vengono liberati dei bit inutili quando il nodo non ha un fratello destro;
- vengono memorizzati, se possibile, le differenze tra valori e non i valori stessi per i motivi già citati.

L'occupazione di memoria per i vari tipi di nodo interno è la seguente:

- small node: da 4 a 8 byte;
- large node: da 8 a 15 byte.

## 2.3 Suffix Array

In questa sezione si darà la definizione di *suffix array* [17] e di *enhanced suffix array* [1]. Verrà spiegato in che modo è possibile costruirli e qual è la loro occupazione di spazio.

**Definizione 4.** *Data una sequenza $S$ di lunghezza $n$ definita su un alfabeto $\Sigma$, il suffix array $suf$ per $S$ è un array di interi nell'intervallo da 1 a $n$ che descrive l'ordine lessicografico degli $n$ suffissi di $S$. In altre parole, il suffisso $S_{suf[1]}$ è quello che precede tutti gli altri suffissi nell'ordine lessicografico.*

Come per il suffix tree, nel suffix array spesso si aggiunge al termine della sequenza che si vuole elaborare un simbolo terminale \$.

Il suffix array si può costruire in un tempo proporzionale alla lunghezza della sequenza pertendo dal suffix tree per quella stessa sequenza [11]. Algoritmi più recenti permettono di costruire direttamente il suffix array senza passare dal suffix tree e sempre in un tempo $O(n)$ [3, 13, 15]. L'occupazione spaziale è di 4 byte per simbolo.

Il suffix array, però, non è così potente come il suffix tree. Infatti, non per tutti gli algoritmi pensati per lavorare su un suffix tree esiste un algoritmo equivalente che lavora su un suffix array. Aggiungendo delle informazioni aggiuntive al suffix array, però, questa nuova struttura, chiamata enhanced suffix array [1], diventa potente quanto un suffix tree. In altre parole, per ogni algoritmo pensato per operare su un suffix tree è possibile creare un algoritmo con la stessa complessità temporale che opera su un enhanced suffix array. Per questo motivo, in questa tesi l'analisi è diretta verso le prestazioni in termini di occupazione di spazio, mentre non vengono approfondite le questioni legate alle prestazioni temporali poiché sia per il suffix tree che per l'enhanced suffix array gli algoritmi che vi operano possono essere considerati ugualmente veloci.

Le informazioni aggiuntive necessarie per ottenere una struttura dati equivalente al suffix tree variano in base alla tipologia di algoritmi che si vuole eseguire sull'enhanced suffix array. In Tabella 2.1 si fornisce l'elenco delle tabelle aggiuntive rispetto al suffix array di base per poter eseguire su di esso diverse classi di algoritmi.

In generale, per poter eseguire un attraversamento *bottom-up* è necessario avere a disposizione, oltre al suffix array di base, anche la tabella *lcp*, mentre per l'attraversamento *top-down* sono necessari il suffix array di base e le tabelle *lcp* e *child*. Per poter operare attraversamenti seguendo i suffix link è invece necessario avere la tabella *link* in aggiunta al suffix array.

È necessario fornire una definizione prima di poter descrivere in dettaglio le tabelle aggiuntve.

| Applicazione | Enhanced suffix array | | | | | | |
|---|---|---|---|---|---|---|---|
| | *suf* | *lcp* | *child* | *link* | *S* | *bwt* | *skp* |
| Supermaximal repeats | ✓ | ✓ | | | | ✓ | |
| Maximal unique matches | ✓ | ✓ | | | | ✓ | |
| Maximal repeated pairs | ✓ | ✓ | | | | ✓ | |
| Ziv-Lempel decomposition | ✓ | ✓ | | | | | |
| Pattern searching | ✓ | ✓ | ✓ | | ✓ | | |
| Shortest unique substrings | ✓ | ✓ | ✓ | | | | |
| Matching statistics | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| Profile matching | ✓ | ✓ | | | | | ✓ |

Tabella 2.1: Informazioni aggiuntive necessarie per l'esecuzione di diverse classi di algoritmi [1, 9, 11]

**Definizione 5.** *Un intervallo* $[i..j]$, $0 \leq 0 < j \leq n$, *è un lcp-intervallo di lcp-valore* $\ell$ *se*

- $lcp[i] < \ell$;
- $lcp[k] \geq \ell$ *per tutti i* $k$ *con* $i + 1 \leq k \leq j$;
- $lcp[k] = \ell$ *per almeno un* $k$ *con* $i + 1 \leq k \leq j$;
- $lcp[j + 1] < \ell$.

Ora verranno definite le tabelle che possono essere presenti nell'enhanced suffix array considerando di dover memorizzare la sequenza $S\$$, dove la lunghezza di $S$ è $n$:

- *suf* è il suffix array di base;
- *lcp* è un array di interi nell'intervallo $[0, n]$ tale che $lcp[0] = 0$ e $lcp[i]$ è la lunghezza del più lungo prefisso comune tra $S_{suf[i-1]}$ e $S_{suf[i]}$ per $i \in [1, n]$;
- *child* è una tabella che memorizza la relazione padre-figlio degli lcp-intervalli;
- *link* è una tabella che contiene i suffix link.

Le altre tabelle citate in Tabella 2.1 contengono informazioni relative alle specifiche applicazioni per le quali sono usate.

Sia la tabella *lcp* che la tabella *child* richiedono 4 Byte per simbolo per la loro memorizzazione, mentre la tabella link necessita di uno spazio di 8 Byte per simbolo. In realtà, come illustrato in [1], con alcuni accorgimenti implementativi le tabelle *lcp* e *child* possono essere memorizzate in 1 Byte per simbolo ciascuna,

mentre per la memorizzazione della tabella link servono solo 2 Byte per simbolo. Queste ottimizzazioni peggiorano la complessità temporale di caso peggiore degli algoritmi che operano sull'enhanced suffix array ma, all'atto pratico, non si riscontrano peggioramenti delle prestazioni temporali.

## 2.4 Confronto tra suffix tree ed enhanced suffix array

Già dalle definizioni e dagli esempi è evidente che è più semplice lavorare con i suffix tree che con gli enhanced suffix array. Algoritmi concettualmente complessi che usano i suffix tree diventano estremamente difficili da implementare se se ne vuole creare un equivalente per gli enhanced suffix array. Questo porta a lentezza nello sviluppo e alla difficile individuazione di *bug*. Ciononostante, l'enhanced suffix array è una struttura dati che viene spesso utilizzata nelle situazioni in cui è estremamente importante che l'occupazione di spazio della struttura dati sia modesta.

Il suffix tree con la codifica illi occupa in media 11 byte per simbolo e addirittura 21 byte per simbolo nel caso peggiore (incluso lo spazio per la memorizzazione della sequenza) [14]. L'enhanced suffix array a sua volta ha un'occupazione di spazio che dipende dal tipo di problema a cui viene applicato. Questo rende piuttosto complicato il confronto tra queste strutture dati che, per quanto detto sopra, va fatto separatamente per ogni tipo di applicazione (o classe di applicazioni con i medesimi requisiti di elaborazione). Inoltre, la scelta della struttura dati da utilizzare deve essere fatta in base alle prestazioni richieste dall'applicazione e dall'incremento di complessità nello sviluppo che si è disposti a sostenere per avere un'occupazione di spazio minore.

Per la maggior parte delle applicazioni, comunque, l'enhanced suffix array non ha bisogno di più di 7 Byte per simbolo, o 9 Byte per simbolo se si includono i suffix link, rendendolo in questi casi preferibile al suffix tree in termini di occupazione di spazio.

Il confronto tra suffix tree troncato ed enhanced suffix array verrà invece dettagliato nel capitolo successivo.

# Capitolo 3

# Analisi sperimentale

In questo capitolo verrà descritta l'analisi sperimentale effettuata per diverse sequenze memorizzate su suffix tree troncati. Saranno utilizzate, in particolare, la codifica derivata da illi e la codifica TruST per mettere in evidenza il differente comportamento in termini di occupazione di spazio. Verranno anche illustrati i comportamenti delle codifiche al variare della dimensione dell'alfabeto, della lunghezza della sequenza e del fattore di troncamento e fatto il confronto con un equivalente enhanced suffix array.

Tutti i valori delle occupazioni di spazio riportati in questo capitolo comprendono lo spazio per la memorizzazione della sequenza, necessario per poter ottenere i simboli della sequenza a partire dagli indici memorizzati nei suffix tree. Anche per gli enhanced suffix array, per ottenere un confronto equo, è stato incluso lo spazio per la memorizzazione della sequenza, pari ad 1 Byte per simbolo.

## 3.1 Descrizione delle sequenze analizzate

La scelta delle sequenze da analizzare è stata fatta cercando di coprire la più ampia casistica possibile. Le prime analisi che verranno in seguito presentate sono state fatte su sequenze casuali. Ciò rende possibile stabilire le prestazioni generali delle codifiche e definire un caso base col quale confrontare poi le prestazioni con le sequenze non casuali.

Le sequenze non casuali di tipo testuale sono state estratte dal *Canterbury Corpus* [8] e sono delle sequenze testuali tipiche nell'analisi delle prestazioni di algoritmi su stringhe. Dato che un'applicazione tipica del suffix tree troncato riguarda l'analisi di sequenze biologiche è stata inserita anche una sequenza di questo tipo tra le sequenze analizzate. Ora si spiegheranno in dettaglio i contenuti e le caratteristiche delle sequenze non casuali:

| **Sequenza** | **Lunghezza** | $\|\Sigma\|$ |
|---|---:|---:|
| asyoulik | 125180 | 69 |
| aaa | 100001 | 2 |
| alphabet | 100001 | 27 |
| pi | 1000001 | 11 |
| AE000111 | 4639222 | 5 |

Tabella 3.1: Caratteristiche delle sequenze utilizzate (la lunghezza e la cardinalità dell'alfabeto includono il simbolo terminale $)

- *asyoulik*: questa sequenza contiene la commedia di Shakespeare *As You Like It* in lingua inglese ed è rappresentativa di generici testi in lingua inglese.
- *aaa*: questa sequenza contiene una successione di lettere *a* ed è stata inserita tra le sequenze analizzate per scoprire come le codifiche si comportino nella situazione estrema in cui l'alfabeto è composto da un solo simbolo più il simbolo terminale.
- *alphabet*: in questa sequenza sono presenti tutte le lettere dell'alfabeto inglese una di seguito all'altra e tale successione viene ripetuta per tutta la lunghezza del file. La scelta di questa sequenza è stata dettata dalla volontà di capire il comportamento delle codifiche nel caso ci sia un elevato numero numero di prefissi in comune tra i diversi suffissi della sequenza.
- *pi*: questa sequenza è composta dal primo milione di cifre decimali di $\pi$. Anche in questo caso l'alfabeto non è grande, ma la situazione non è così estrema come nel caso della sequenza *aaa*. La sequenza in questione, dunque, può rappresentare i casi in cui la cardinalità dell'alfabeto è bassa.
- *AE000111*: questa sequenza rappresenta il DNA dell'*Escherichia Coli* ed è un esempio di sequenza biologica. Come già fatto notare in precedenza, l'analisi di sequenze biologiche è un tipico campo di applicazione dei suffix tree ed è dunque necessario includere una sequenza di questo tipo tra quelle analizzate.

In Tabella 3.1 sono dettagliate le caratteristiche delle sequenze citate.

## 3.2 Sequenze casuali

In Figura 3.1 si può osservare la variazione dell'occupazione di spazio di un suffix tree troncato con la codifica derivata da illi e con la codifica TruST per sequenze casuali di lunghezza fissata nelle quali sono state variate la cardinalità dell'alfabeto della sequenza e il valore di $k$. In Figura 3.2 si può invece osservare come varia tale occupazione di spazio per entrambe le codifiche variando la cardinalità dell'alfabeto e la lunghezza delle sequenze e mantenendo fissato, invece, il fattore di troncamento, pari a 20.

La prima osservazione da fare è che, in entrambe le figure, i comportamenti delle due codifiche in esame sono estremamente simili. L'unica differenza osservabile è infatti un vantaggio in termini di occupazione di spazio della codifica TruST rispetto alla codifica derivata da illi. Questo vantaggio, che è dell'ordine di alcuni Byte/simbolo, non è costante e varia in base alla situazione in esame. In particolare, più l'occupazione di spazio per simbolo nella codifica derivata da illi è maggiore, più il risparmio di spazio nella codifica TruST diventa consistente. Le osservazioni che però ora si faranno sulle dipendenze tra occupazione di spazio e caratteristiche della sequenza valgono per entrambe le codifiche, poiché, come mettono in evidenza i grafici, tale dipendenza è comune per le due codifiche analizzate.

#### Variazione dell'occupazione di spazio al variare di $k$ e della cardinalità dell'alfabeto

Nei grafici di Figura 3.1 si possono notare gli effetti di saturazione indotti dal variare del valore di $k$. Per alfabeti grandi l'effetto di saturazione si nota già con valori di $k$ piccoli, mentre, al diminuire della cardinalità dell'alfabeto, il valore di $k$ per il quale insorge il fenomeno di saturazione cresce. Come si avrà occasione di osservare nell'analisi delle sequenze non casuali, tale valore di $k$ può dipendere dalla lunghezza della sequenza in esame e il fatto che si verifichi o meno questa dipendenza è determinato dal tipo di sequenza analizzata.

#### Variazione dell'occupazione di spazio al variare della lunghezza della sequenza e della cardinalità dell'alfabeto

Le variazioni del valore di occupazione di spazio, in funzione di $|\Sigma|$ e della lunghezza della sequenza, in condizioni di saturazione sono invece osservabili nei grafici di Figura 3.2. Per $k = 20$, infatti, come si è già notato nei grafici di Figura 3.1, il fenomeno di saturazione è già presente nelle sequenze casuali per tutte le cardinalità dell'alfabeto esaminate.

In particolare, fissata una lunghezza della sequenza, esiste una cardinalità dell'alfabeto che minimizza lo spazio occupato per simbolo. Tale punto di minimo (che vale circa 7 Byte per simbolo per la codifica derivata da illi e circa 5 Byte per simbolo per la codifica TruST) varia in funzione della lunghezza della sequenza, spostandosi verso cardinalità dell'alfabeto più alte per lunghezze della sequenza crescenti.

Un discorso a parte deve essere fatto per le parti dei grafici più vicine agli assi. Se si osservano le aree relative a lunghezze della sequenza molto piccole sui grafici di Figura 3.2, infatti, si può notare come l'occupazione di spazio minima si abbia per alfabeti molto grandi. Ciò è causato dal fatto che in questo caso ci sono pochi suffissi della sequenza che hanno prefissi in comune, portando ad avere un piccolo numero di nodi interni. Essendo quindi il numero dei nodi nell'albero simile a quello delle foglie, a loro volta in numero pari alla lunghezza della sequenza, ecco come l'occupazione di spazio per simbolo sia approssimabile all'occupazione di spazio di una singola foglia.

L'area dei grafici in Figura 3.2 relativa a cardinalità dell'alfabeto bassa evidenzia, invece, come le sequenze con alfabeti piccoli abbiano delle occupazioni di spazio in condizioni di saturazione molto elevate. Ciò sarà confermato anche dalle analisi delle sequenze non causali.

### Confronto con l'enhanced suffix array

Il confronto dei risultati ottenuti con l'occupazione di spazio dell'enhanced suffix array porta ad osservare come sia vantaggioso l'uso del suffix tree rispetto all'enhanced suffix array nel caso in cui per quest'ultimo non si applichino gli accorgimenti implementativi atti a ridurne l'occupazione spaziale, come citato nel capitolo precedente. Infatti, considerando un enhanced suffix array che permetta l'attraversamento bottom-up e top-down si occupano 13 Byte per simbolo e, se si aggiungono i suffix link, si arrivano ad occupare ben 21 Byte per simbolo. Nel caso tali accorgimenti vengano applicati, invece, l'occupazione di spazio dell'enhanced suffix cala a 7 Byte per simbolo e 9 Byte per simbolo, rispettivamente senza e con i suffix link. Questo implica che, come si può rilevare dai grafici, in alcuni casi è vantaggioso l'uso dei suffix tree, mentre negli altri casi si ottiene una minore occupazione di spazio utilizzando l'enhanced suffix array.

Un'analisi dettagliata dei risultati illustrati in Tabella 3.2 evidenzia che, per questo tipo di sequenze, usando la codifica illi, il suffix tree troncato, per $|\Sigma| < 12$, si comporta meglio dell'enhanced suffix array quando $k \leq 6$ e peggio per $k > 6$. Per $12 \leq |\Sigma| < 52$ il suffix tree troncato si comporta meglio, rispetto all'enhanced suffix array, per $k \leq 4$, mentre le due strutture dati sono confrontabili per $k > 4$. Per $|\Sigma| \geq 52$ il suffix tree troncato ha una minore occupazione di spazio rispetto all'enhanced suffix array per ogni valore di $k$.

Analizzando i risultati ottenuti con la codifica TruST, invece, risulta che, per $|\Sigma| < 12$, il suffix tree troncato ha un'occupazione spaziale, rispetto all'enhanced suffix array, minore per $k \leq 6$, superiore per $k \geq 10$ e confrontabile per $6 < k < 10$. Per $12 \leq |\Sigma| < 36$ il suffix tree troncato si comporta meglio, rispetto all'enhanced suffix array, per $k \leq 5$, mentre le due strutture dati sono confrontabili per $k > 5$. Per $|\Sigma| \geq 36$ il suffix tree troncato ha una minore occupazione di spazio rispetto all'enhanced suffix array per ogni valore di $k$.

Osservando le Tabelle 3.4 e 3.5 si può invece osservare che le migliori prestazioni in termini di occupazione di spazio si ottengono con l'enhanced suffix array per $|\Sigma| < 36$, mentre, per $|\Sigma| \geq 36$, le prestazioni in termini di occupazione di spazio per le due strutture dati sono confrontabili.

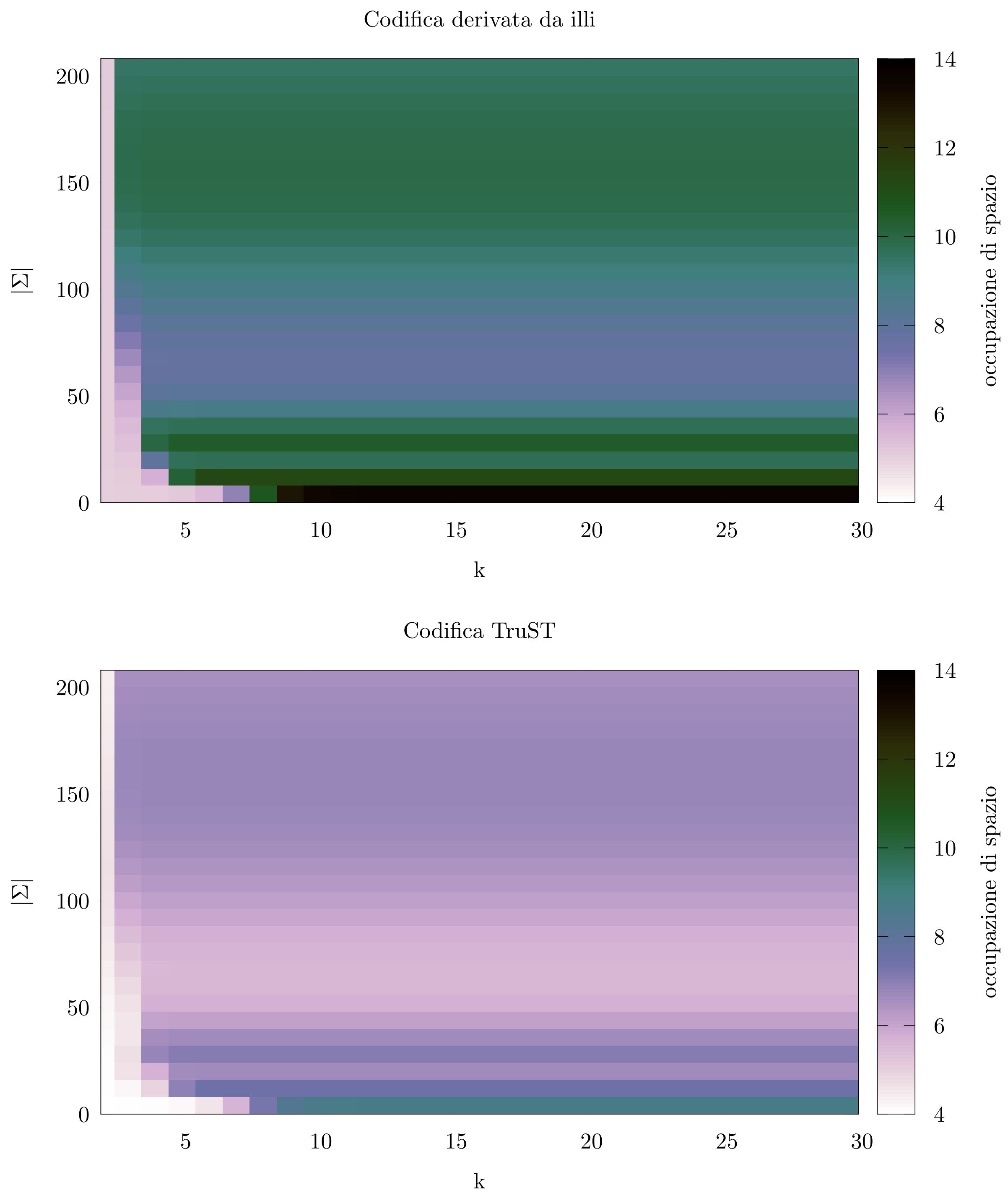


Figura 3.1: Occupazione di spazio (in Byte/simbolo) per sequenze casuali di lunghezza 45000

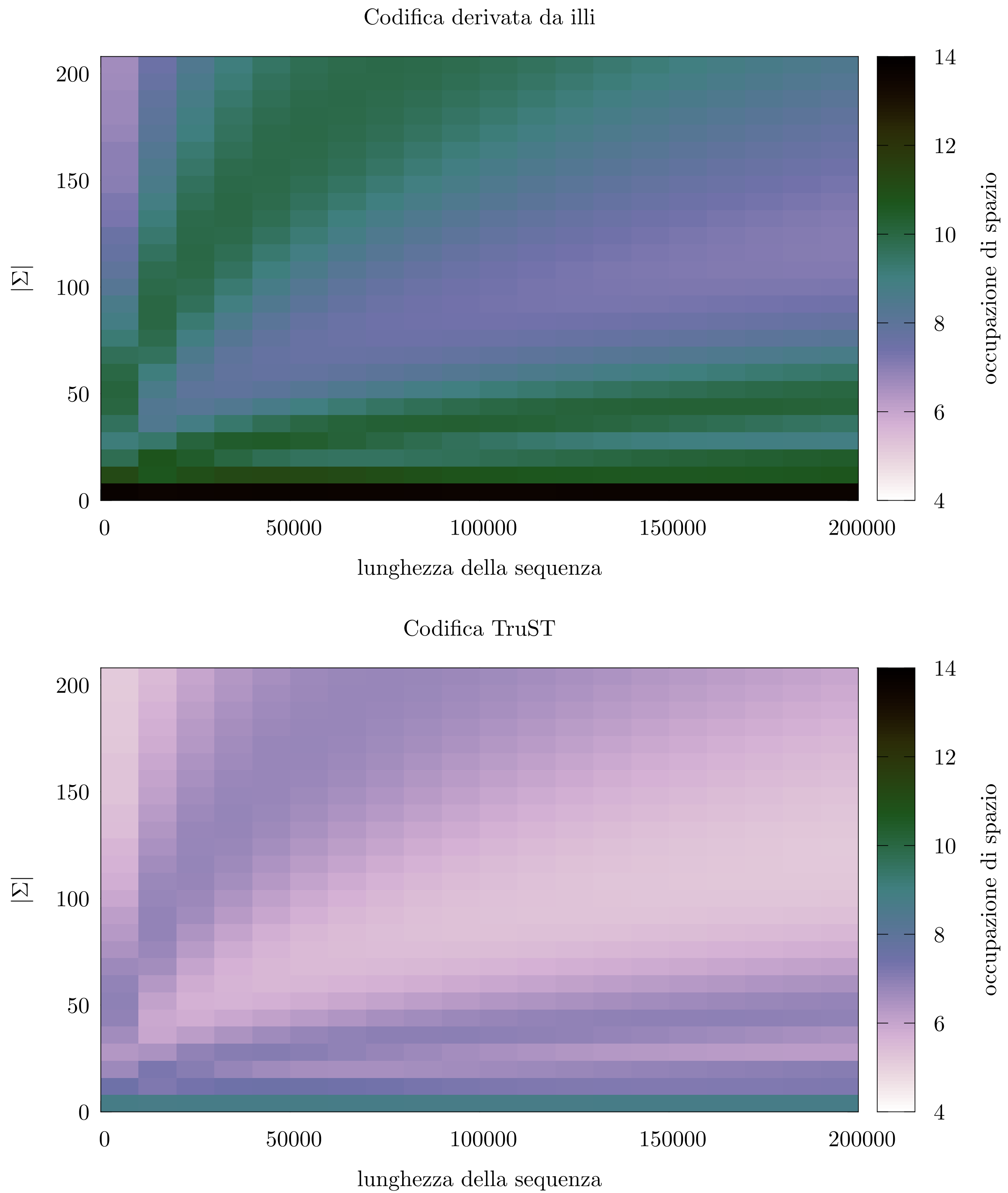


Figura 3.2: Occupazione di spazio (in Byte/simbolo) per sequenze casuali fissato $k = 20$

| | | $\|\Sigma\|$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 4 | 12 | 20 | 28 | 36 | 52 | 76 | 100 | 124 | 156 | 204 |
| | 2 | 5,00 | 5,00 | 5,01 | 5,01 | 5,01 | 5,02 | 5,03 | 5,04 | 5,04 | 5,06 | 5,07 |
| | 3 | 5,01 | 5,06 | 5,15 | 5,29 | 5,47 | 5,98 | 7,08 | 8,37 | 9,35 | 9,79 | 9,40 |
| | 4 | 5,03 | 5,67 | 7,91 | 9,98 | 9,54 | 7,98 | 7,83 | 8,71 | 9,52 | 9,87 | 9,43 |
| | 5 | 5,12 | 10,18 | 9,63 | 10,42 | 9,68 | 8,01 | 7,83 | 8,71 | 9,52 | 9,87 | 9,43 |
| $k$ | 6 | 5,47 | 11,16 | 9,69 | 10,43 | 9,68 | 8,01 | 7,83 | 8,71 | 9,52 | 9,87 | 9,43 |
| | 8 | 10,63 | 11,22 | 9,69 | 10,43 | 9,68 | 8,01 | 7,83 | 8,71 | 9,52 | 9,87 | 9,43 |
| | 10 | 13,41 | 11,22 | 9,69 | 10,43 | 9,68 | 8,01 | 7,83 | 8,71 | 9,52 | 9,87 | 9,43 |
| | 12 | 13,58 | 11,22 | 9,69 | 10,43 | 9,68 | 8,01 | 7,83 | 8,71 | 9,52 | 9,87 | 9,43 |
| | 14 | 13,59 | 11,22 | 9,69 | 10,43 | 9,68 | 8,01 | 7,83 | 8,71 | 9,52 | 9,87 | 9,43 |

Tabella 3.2: Occupazione di spazio in Byte per simbolo della codifica derivata da illi per sequenze casuali di lunghezza 45000 in base al valore di k e alla cardinalità dell'alfabeto

| | | $\|\Sigma\|$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 4 | 12 | 20 | 28 | 36 | 52 | 76 | 100 | 124 | 156 | 204 |
| | 2 | 4,00 | 4,01 | 4,03 | 4,06 | 4,10 | 4,20 | 4,40 | 4,58 | 4,61 | 4,51 | 4,33 |
| | 3 | 4,01 | 4,15 | 4,58 | 4,65 | 4,50 | 4,60 | 5,18 | 5,90 | 6,46 | 6,71 | 6,50 |
| | 4 | 4,03 | 4,89 | 5,66 | 6,79 | 6,54 | 5,67 | 5,59 | 6,09 | 6,55 | 6,75 | 6,51 |
| | 5 | 4,13 | 6,90 | 6,58 | 7,04 | 6,62 | 5,69 | 5,60 | 6,09 | 6,55 | 6,75 | 6,51 |
| $k$ | 6 | 4,53 | 7,43 | 6,61 | 7,04 | 6,63 | 5,69 | 5,60 | 6,09 | 6,55 | 6,75 | 6,51 |
| | 8 | 7,23 | 7,46 | 6,62 | 7,04 | 6,63 | 5,69 | 5,60 | 6,09 | 6,55 | 6,75 | 6,51 |
| | 10 | 8,62 | 7,46 | 6,62 | 7,04 | 6,63 | 5,69 | 5,60 | 6,09 | 6,55 | 6,75 | 6,51 |
| | 12 | 8,72 | 7,46 | 6,62 | 7,04 | 6,63 | 5,69 | 5,60 | 6,09 | 6,55 | 6,75 | 6,51 |
| | 14 | 8,72 | 7,46 | 6,62 | 7,04 | 6,63 | 5,69 | 5,60 | 6,09 | 6,55 | 6,75 | 6,51 |

Tabella 3.3: Occupazione di spazio in Byte per simbolo della codifica TruST per sequenze casuali di lunghezza 45000 in base al valore di k e alla cardinalità dell'alfabeto

| | | $\vert\Sigma\vert$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 4 | 12 | 36 | 60 | 84 | 108 | 132 | 156 | 180 | 204 |
| lunghezza | 5000 | 13,63 | 11,13 | 9,55 | 9,94 | 8,84 | 7,91 | 7,22 | 6,94 | 6,70 | 6,58 |
| | 35000 | 13,58 | 11,16 | 9,31 | 7,84 | 8,55 | 9,52 | 9,91 | 9,80 | 9,46 | 9,04 |
| | 65000 | 13,58 | 11,08 | 10,12 | 7,88 | 7,54 | 8,20 | 9,03 | 9,64 | 9,85 | 9,82 |
| | 95000 | 13,54 | 10,76 | 10,26 | 8,26 | 7,34 | 7,54 | 8,19 | 8,91 | 9,44 | 9,77 |
| | 125000 | 13,55 | 10,61 | 10,08 | 8,66 | 7,38 | 7,23 | 7,64 | 8,28 | 8,89 | 9,40 |
| | 155000 | 13,57 | 10,58 | 9,79 | 9,02 | 7,50 | 7,11 | 7,32 | 7,82 | 8,39 | 8,96 |
| | 185000 | 13,58 | 10,62 | 9,49 | 9,31 | 7,66 | 7,07 | 7,12 | 7,50 | 8,00 | 8,55 |

Tabella 3.4: Occupazione di spazio in Byte per simbolo della codifica derivata da illi per sequenze casuali in base alla lunghezza della sequenza e alla cardinalità dell'alfabeto

| | | $\vert\Sigma\vert$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 4 | 12 | 36 | 60 | 84 | 108 | 132 | 156 | 180 | 204 |
| lunghezza | 5000 | 8,75 | 7,43 | 6,59 | 6,84 | 6,25 | 5,74 | 5,38 | 5,25 | 5,14 | 5,09 |
| | 35000 | 8,72 | 7,43 | 6,42 | 5,60 | 6,00 | 6,55 | 6,78 | 6,72 | 6,53 | 6,30 |
| | 65000 | 8,71 | 7,38 | 6,87 | 5,62 | 5,43 | 5,81 | 6,27 | 6,62 | 6,74 | 6,73 |
| | 95000 | 8,70 | 7,20 | 6,95 | 5,83 | 5,31 | 5,43 | 5,80 | 6,20 | 6,51 | 6,69 |
| | 125000 | 8,70 | 7,12 | 6,85 | 6,06 | 5,33 | 5,26 | 5,49 | 5,85 | 6,19 | 6,48 |
| | 155000 | 8,71 | 7,10 | 6,68 | 6,25 | 5,40 | 5,19 | 5,30 | 5,59 | 5,91 | 6,23 |
| | 185000 | 8,72 | 7,13 | 6,51 | 6,42 | 5,49 | 5,17 | 5,19 | 5,41 | 5,69 | 6,00 |

Tabella 3.5: Occupazione di spazio in Byte per simbolo della codifica TruST per sequenze casuali in base alla lunghezza della sequenza e alla cardinalità dell'alfabeto

## 3.3 Sequenze non casuali

In questo capitolo si analizzeranno i risultati di occupazione di spazio dagli esperimenti sulle sequenze non casuali. Gli esperimenti sono stati condotti cercando una possibile correlazione tra l'occupazione di spazio per simbolo e i valori di $k$ e di lunghezza della sequenza. Nel fare i vari esperimenti le sequenze originali sono state troncate a varie lunghezze, avendo però l'accortezza di controllare che l'alfabeto della sequenza non subisse modifiche a causa del troncamento.

È altresì importante osservare come, in tutte le analisi che seguiranno, le due codifiche si comportino in modo identico, se non per lo stesso vantaggio in termini di occupazione di spazio evidenziato nell'analisi delle sequenze casuali.

### 3.3.1 Sequenze aaa e alphabet

Nonostante la cardinalità dell'alfabeto delle due sequenze in esame sia molto diversa, i risultati sperimentali ottenuti, osservabili in Figura 3.3 e Figura 3.4, sono molto simili. Il perché di questo è spiegabile analizzando la struttura delle sequenze: entrambe hanno un numero estremamente elevato di ripetizioni. Ciò fa sì che il fenomeno di saturazione non si inneschi per alcun valore di $k$.

Si può osservare, però, che, a parità di $k$, si ottiene un'occupazione di spazio per simbolo minore usando sequenze più lunghe. Questo perché, a causa della struttura delle sequenze, un incremento in lunghezza non aggiunge nuovi rami alla parte superiore dell'albero. Avendo fissato il valore del fattore di troncamento, dunque, l'occupazione totale di spazio del suffix tree troncato non cambia, ma quest'ultima viene divisa per un numero maggiore di simboli per ottenere l'occupazione di spazio per simbolo, che dunque cala per l'aumentare della lunghezza della sequenza.

Un'ulteriore osservazione è che, per una certa lunghezza della sequenza, il valore di occupazione di spazio per simbolo cresce linearmente al crescere di $k$, fino a raggiungere un valore di $k$ uguale a quello della lunghezza della sequenza, eguagliando, quindi, i risultati ottenibili per un suffix tree non troncato.

#### Analisi dell'occupazione di spazio in condizioni di saturazione

È interessante fornire un'analisi più dettagliata dell'occupazione di spazio in condizioni di saturazione: per la codifica derivata da illi esso è pari a 13,25 Byte per simbolo, mentre per la codifica TruST tale valore è di 8,22 Byte per simbolo. Si ricorda che l'enhanced suffix array, affinché permetta gli attraversamenti di tipo bottom-up e top-down, necessita di appena 7 Byte per simbolo (13 Byte per simbolo nella versione che non intacca le prestazioni di caso peggiore degli algoritmi). È necessario, però, notare che nei casi in cui gli algoritmi operanti su tale struttura

dati necessitino di ulteriori estensioni dell'enhanced suffix array, come nel caso del profile matching, la codifica TruST può risultare più efficiente.

### Confronto dei risultati ottenuti per il suffix tree troncato con l'enhanced suffix array

I risultati ottenuti mostrano come il suffix tree troncato fornisca prestazioni superiori, in termini di occupazione di spazio, rispetto all'enhanced suffix array per valori di $k$ fino al 60% della lunghezza della sequenza. Per valori di $k$ tra il 60% e il 75% della lunghezza della sequenza le prestazioni in termini di occupazione di spazio del suffix tree troncato e dell'enhanced suffix array sono confrontabili, mentre quest'ultimo risulta migliore rispetto al primo per valori di $k$ maggiori rispetto al 75% della lunghezza della sequenza.

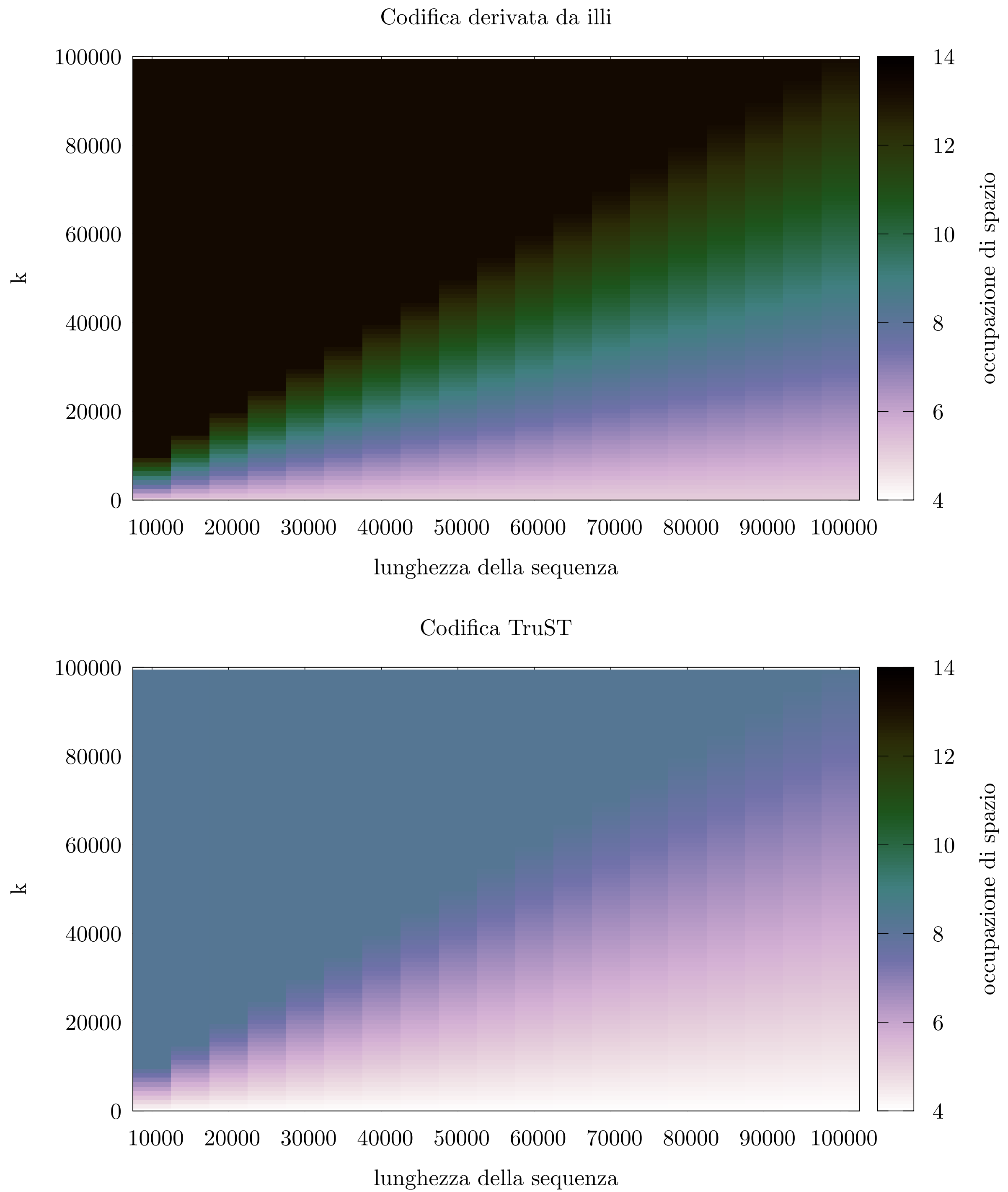


Figura 3.3: Occupazione di spazio (in Byte/simbolo) per la sequenza aaa

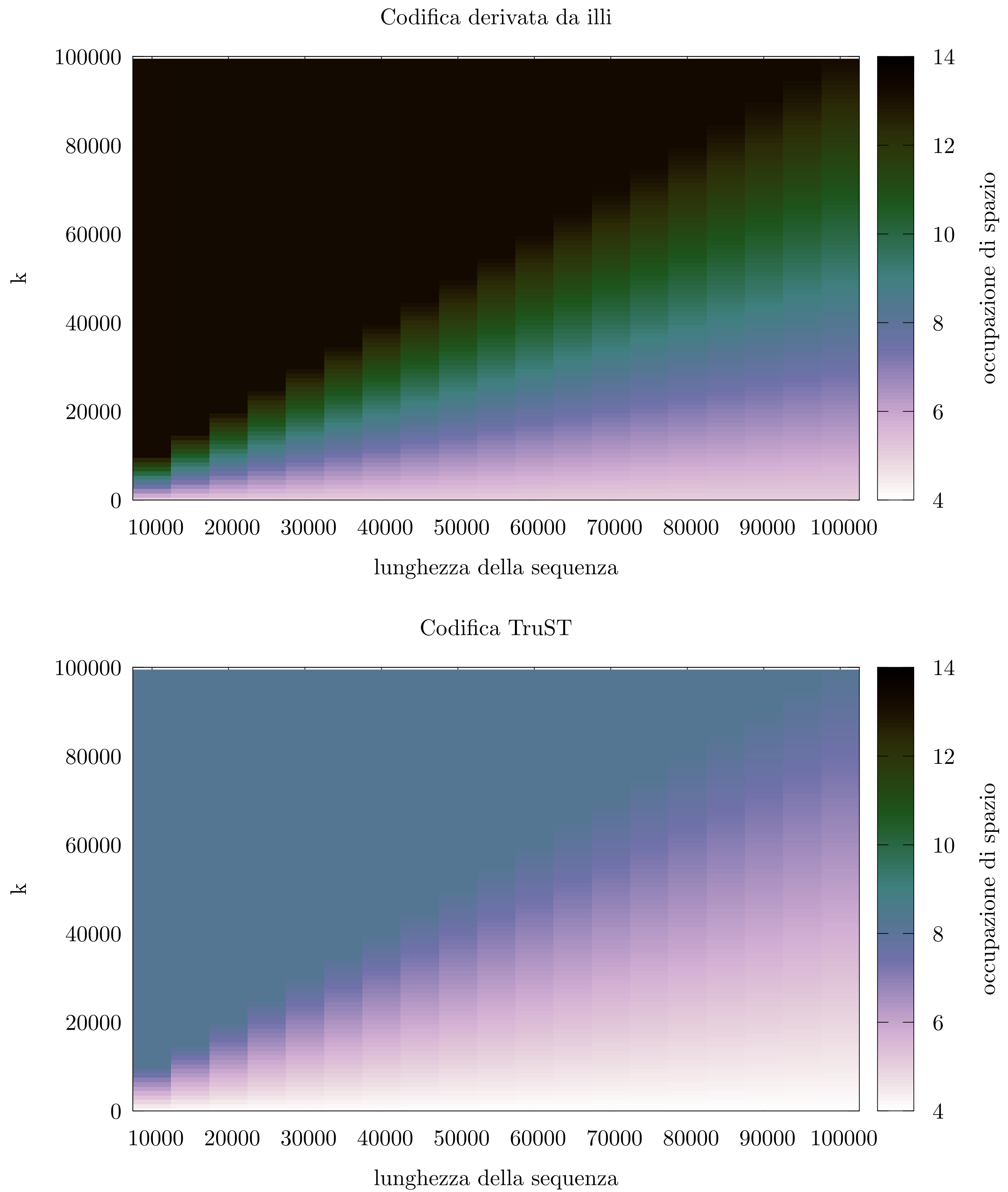


Figura 3.4: Occupazione di spazio (in Byte/simbolo) per la sequenza alphabet

| lunghezza \ $k$ | 2 | 10002 | 20002 | 30002 | 40002 | 50002 | 60002 | 70002 | 80002 |
|---|---|---|---|---|---|---|---|---|---|
| 10000 | 5,00 | 13,25 | 13,25 | 13,25 | 13,25 | 13,25 | 13,25 | 13,25 | 13,25 |
| 20000 | 5,00 | 9,13 | 13,25 | 13,25 | 13,25 | 13,25 | 13,25 | 13,25 | 13,25 |
| 30000 | 5,00 | 7,75 | 10,50 | 13,25 | 13,25 | 13,25 | 13,25 | 13,25 | 13,25 |
| 40000 | 5,00 | 7,06 | 9,13 | 11,19 | 13,25 | 13,25 | 13,25 | 13,25 | 13,25 |
| 50000 | 5,00 | 6,65 | 8,30 | 9,95 | 11,60 | 13,25 | 13,25 | 13,25 | 13,25 |
| 60000 | 5,00 | 6,38 | 7,75 | 9,13 | 10,50 | 11,88 | 13,25 | 13,25 | 13,25 |
| 70000 | 5,00 | 6,18 | 7,36 | 8,54 | 9,71 | 10,89 | 12,07 | 13,25 | 13,25 |
| 80000 | 5,00 | 6,03 | 7,06 | 8,09 | 9,13 | 10,16 | 11,19 | 12,22 | 13,25 |

Tabella 3.6: Occupazione di spazio in Byte per simbolo della codifica derivata da illi per la sequenza aaa

| lunghezza \ $k$ | 2 | 10002 | 20002 | 30002 | 40002 | 50002 | 60002 | 70002 | 80002 |
|---|---|---|---|---|---|---|---|---|---|
| 10000 | 4,00 | 8,21 | 8,21 | 8,21 | 8,21 | 8,21 | 8,21 | 8,21 | 8,21 |
| 20000 | 4,00 | 6,11 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 |
| 30000 | 4,00 | 5,40 | 6,81 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 |
| 40000 | 4,00 | 5,05 | 6,11 | 7,16 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 |
| 50000 | 4,00 | 4,84 | 5,69 | 6,53 | 7,37 | 8,22 | 8,22 | 8,22 | 8,22 |
| 60000 | 4,00 | 4,70 | 5,41 | 6,11 | 6,81 | 7,51 | 8,22 | 8,22 | 8,22 |
| 70000 | 4,00 | 4,60 | 5,20 | 5,81 | 6,41 | 7,01 | 7,62 | 8,22 | 8,22 |
| 80000 | 4,00 | 4,53 | 5,05 | 5,58 | 6,11 | 6,64 | 7,16 | 7,69 | 8,22 |

Tabella 3.7: Occupazione di spazio in Byte per simbolo della codifica TruST per la sequenza aaa

| | | $k$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2 | 10002 | 20002 | 30002 | 40002 | 50002 | 60002 | 70002 | 80002 |
| lunghezza 10000 | 5,00 | 13,23 | 13,23 | 13,23 | 13,23 | 13,23 | 13,23 | 13,23 | 13,23 |
| 20000 | 5,00 | 9,13 | 13,24 | 13,24 | 13,24 | 13,24 | 13,24 | 13,24 | 13,24 |
| 30000 | 5,00 | 7,75 | 10,50 | 13,24 | 13,24 | 13,24 | 13,24 | 13,24 | 13,24 |
| 40000 | 5,00 | 7,06 | 9,13 | 11,19 | 13,24 | 13,24 | 13,24 | 13,24 | 13,24 |
| 50000 | 5,00 | 6,65 | 8,30 | 9,95 | 11,60 | 13,25 | 13,25 | 13,25 | 13,25 |
| 60000 | 5,00 | 6,38 | 7,75 | 9,13 | 10,50 | 11,88 | 13,25 | 13,25 | 13,25 |
| 70000 | 5,00 | 6,18 | 7,36 | 8,54 | 9,71 | 10,89 | 12,07 | 13,25 | 13,25 |
| 80000 | 5,00 | 6,03 | 7,06 | 8,09 | 9,13 | 10,16 | 11,19 | 12,22 | 13,25 |

Tabella 3.8: Occupazione di spazio in Byte per simbolo della codifica derivata da illi per la sequenza alphabet

| | | $k$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2 | 10002 | 20002 | 30002 | 40002 | 50002 | 60002 | 70002 | 80002 |
| lunghezza 10000 | 4,03 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 |
| 20000 | 4,01 | 6,12 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 |
| 30000 | 4,01 | 5,41 | 6,82 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 |
| 40000 | 4,01 | 5,06 | 6,11 | 7,17 | 8,22 | 8,22 | 8,22 | 8,22 | 8,22 |
| 50000 | 4,01 | 4,85 | 5,69 | 6,53 | 7,38 | 8,22 | 8,22 | 8,22 | 8,22 |
| 60000 | 4,00 | 4,71 | 5,41 | 6,11 | 6,82 | 7,52 | 8,22 | 8,22 | 8,22 |
| 70000 | 4,00 | 4,61 | 5,21 | 5,81 | 6,41 | 7,02 | 7,62 | 8,22 | 8,22 |
| 80000 | 4,00 | 4,53 | 5,06 | 5,58 | 6,11 | 6,64 | 7,17 | 7,69 | 8,22 |

Tabella 3.9: Occupazione di spazio in Byte per simbolo della codifica TruST per la sequenza alphabet

### 3.3.2 Sequenze AE000111, asyoulik e pi

Le sequenze AE000111 (Figura 3.5), pi (Figura 3.6) e asyoulik (Figura 3.7) si comportano in modo estremamente simile. Innanzitutto si deve notare che per tutte e tre le sequenze, così come è avvenuto per le altre sequenze casuali e per quelle non casuali, l'andamento dell'occupazione di spazio per simbolo è lo stesso per entrambe le codifiche. La codifica TruST, però, come già notato, permette un risparmio di spazio che, a seconda delle situazione, varia da 1 a 4 Byte per simbolo.

Si può notare, altresì, che l'occupazione di spazio per simbolo non è dipendente dalla lunghezza della sequenza. Ciò è in forte contrasto rispetto ai risultati ottenuti con le sequenze altamente ripetitive viste in precedenza.

Fatte queste considerazioni, è interessante comparare i risultati qui ottenuti con i grafici di Figura 3.2. Nei risultati per la codifica della sequenza biologica AE000111 (Figura 3.5) si può osservare come il valore minimo di occupazione di spazio (che vale circa 5 Byte per simbolo per la codifica derivata da illi e circa 4 Byte per simbolo per la codifica TruST) si mantenga fino a $k = 5$. Se si analizzano i risultati ottenuti per la sequenza pi (Figura 3.6) si scopre, invece, come tale valore minimo sia mantenuto solo fino a $k = 3$. Per quanto riguarda la sequenza asyoulik (Figura 3.7) addirittura non esiste un intervallo di valori di $k$ in cui il valore minimo di occupazione di spazio venga mantenuto. Tutto questo è in accordo con quanto mostrato in Figura 3.2 per le sequenza non casuali: aumentando la cardinalità dell'alfabeto la transizione dal minimo valore di occupazione spaziale al valore di saturazione inizia per valori di $k$ sempre minori.

#### Analisi dell'occupazione di spazio in condizione di saturazione

È opportuno osservare che il valore di occupazione di spazio in condizioni di saturazione, coincidente alla situazione di suffix tree non troncato, segue l'andamento osservato nei file casuali (Figura 3.1). La sequenza AE000111 mostra, in tale situazione, un valore di occupazione di spazio di circa 13,5 Byte per simbolo con la codifica derivata da illi e di circa 8,7 Byte per simbolo con la codifica TruST. Tali valori coincidono con quelli rilevati dagli esperimenti con sequenze casuali aventi una cardinalità dell'alfabeto bassa. Anche la sequenza pi mostra tale similitudine con le sequenze casuali, facendo rilevare, in condizioni di saturazione, un valore di occupazione spaziale di circa 11 Byte per simbolo con la codifica derivata da illi e di poco più di 7 Byte per simbolo con la codifica TruST. I risultati ottenuti con la sequenza asyoulik mostrano invece una certa discrepanza rispetto ai valori di occupazione di spazio in condizioni di saturazione per sequenze casuali con uguale cardinalità dell'alfabeto. Infatti, i valori di occupazione spaziale ottenuti in condizioni di saturazione per la sequenza asyoulik sono di circa 2 Byte per simbolo peggiori rispetto a quelli della rispettiva sequenza casuale. Tale discrepanza è

verosimilmente attribuibile al fatto che, mentre nelle sequenze AE000111 e pi ciascuno dei simboli dell'alfabeto appare in numero simile agli altri all'interno della sequenza, nella sequenza asyoulik alcuni simboli, come, ad esempio, i simboli di punteggiatura, ricorrono più raramente all'interno della sequenza rispetto ad altri simboli. Questo rende la sequenza difficilmente assimilabile ad una successione casuale di simboli e dimostra che gli esperimenti effettuati su sequenze casuali non sono atti a rappresentare il comportamento della struttura dati per una sequenza qualsiasi.

#### Confronto dei risultati ottenuti per il suffix tree troncato con l'enhanced suffix array

È necessario anche confrontare i valori appena citati con quelli ottenibili usando l'enhanced suffix array. Come già detto, l'occupazione di spazio di quest'ultima struttura dati, per poter operare gli attraversamenti bottom-up e top-down, è pari a 7 Byte per simbolo (13 Byte per simbolo nella versione che non influisce sulle prestazioni temporali di caso peggiore degli algoritmi che operano su di esso). La codifica TruST garantisce una minore occupazione di spazio per ogni valore di $k$ rispetto alla versione più avida di memoria dell'enhanced suffix array, mentre, considerando la versione dell'enhanced suffix array che richiede meno memoria, il vantaggio del suffix tree con codifica TruST è presente solo per valori di $k$ molto piccoli.

Nello specifico, per la sequenza AE000111, osservando la Tabella 3.11 si nota che il suffix tree troncato si comporta, rispetto all'enhanced suffix array, meglio per $k \leq 6$, peggio per $k \geq 10$, mentre per i valori di $k$ intermedi le due strutture dati mostrano occupazioni di spazio confrontabili. Per la sequenza pi, i cui risultati sono in Tabella 3.13, il suffix tree troncato ha una minore occupazione di spazio rispetto all'enhanced suffix array per $k \leq 5$, mentre per valori di $k$ superiori le due strutture dati hanno mostrato occupazioni di spazio confrontabili. Analogamente, per la sequenza asyoulik (Tabella 3.15) il suffix tree troncato ottiene migliori prestazioni in termini di occupazione di spazio rispetto all'enhanced suffix array per $k \leq 10$, mentre per valori di $k$ superiori le due strutture dati sono confrontabili in termini di occupazione spaziale.

Prendendo in considerazione, invece, l'enhanced suffix array che permette gli attraversamenti bottom-up, top-down e seguendo i suffix link, l'occupazione di spazio sale a 9 Byte per simbolo (21 Byte per simbolo nella versione che non influisce sulle prestazioni temporali di caso peggiore degli algoritmi che operano su di esso). In questo caso il confronto con il suffix tree si deve effettuare considerando la codifica derivata da illi. Anche in questo caso, utilizzando la versione che occupa meno memoria dell'enhanced suffix array, il vantaggio di quest'ultimo sul suffix tree è presente per i valori di $k$ che sono tipicamente utilizzati nella pratica, mentre per

valori di $k$ più bassi il suffix tree riesce ad offrire prestazioni migliori in termini di occupazione di spazio. Utilizzando, invece, la versione del suffix array che occupa 21 Byte per simbolo, questa risulta avere delle prestazioni in termini di occupazione di spazio peggiori rispetto al suffix tree per ogni valore di $k$.

Nello specifico, per la sequenza AE000111, dalla Tabella 3.10 si nota che il suffix tree troncato si comporta, rispetto all'enhanced suffix array, meglio per $k \leq 6$, peggio per $k \geq 10$, mentre per i valori di $k$ intermedi le due strutture dati mostrano occupazioni di spazio confrontabili. Per la sequenza pi (Tabella 3.12) il suffix tree troncato ha un'occupazione di spazio, rispetto all'enhanced suffix array, minore per $k \leq 4$ e maggiore per $k \geq 6$, mentre per valori di $k$ intermedi le occupazioni spaziali delle due strutture dati sono confrontabili. Analogamente, per la sequenza asyoulik (Tabella 3.14) il suffix tree troncato ottiene migliori prestazioni in termini di occupazione di spazio rispetto all'enhanced suffix array per $k \leq 6$, mentre per valori di $k$ superiori le due strutture dati hanno un'occupazione di spazio confrontabile.

In tutti i casi, se gli algoritmi che operano sull'enhanced suffix array richiedono ulteriori informazioni aggiuntive oltre a quelle già incluse, l'occupazione di spazio aumenta e le situazioni in cui l'enhanced suffix array offre una minore occupazione di spazio rispetto al suffix tree possono diminuire.

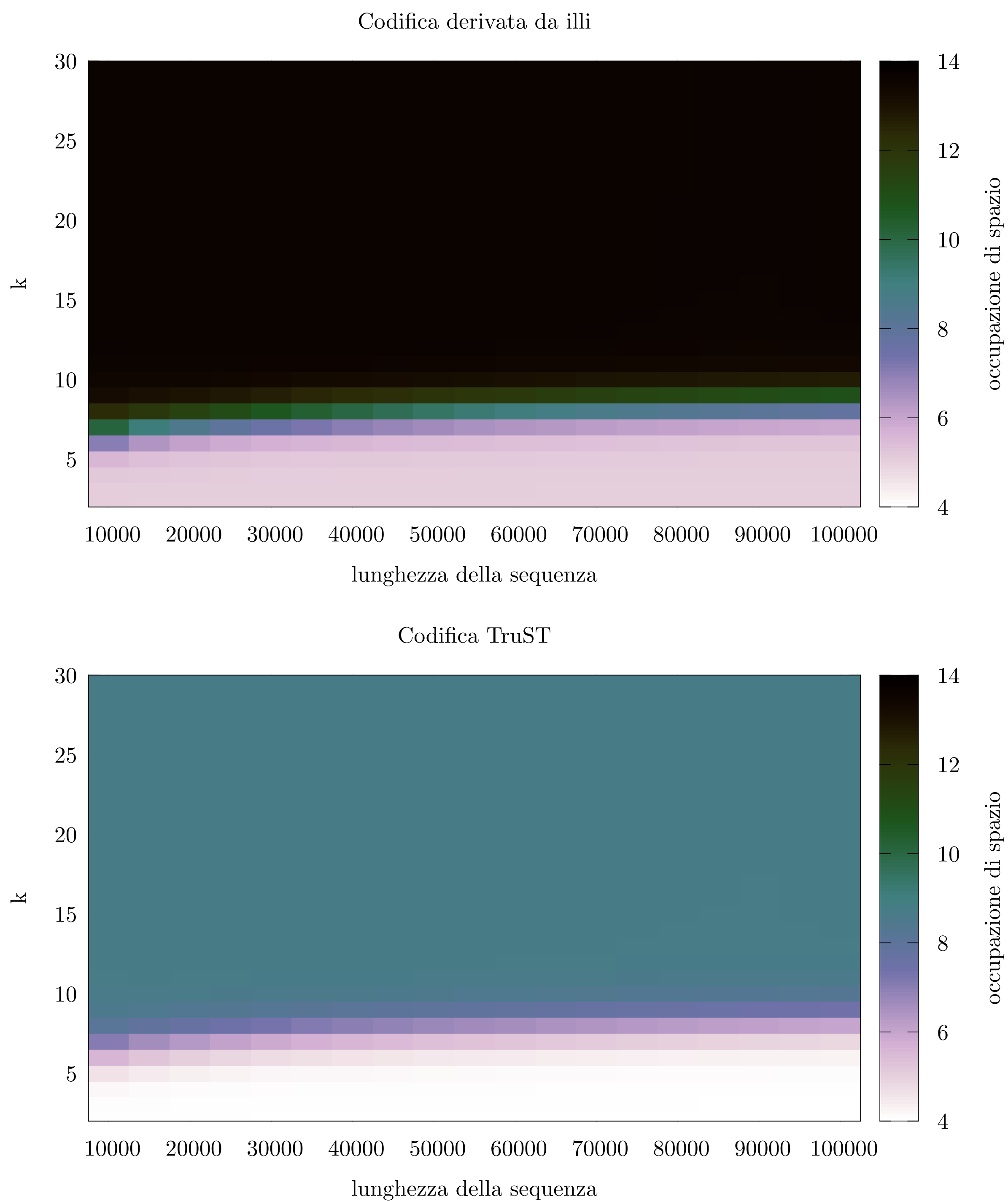


Figura 3.5: Occupazione di spazio (in Byte/simbolo) per la sequenza AE000111

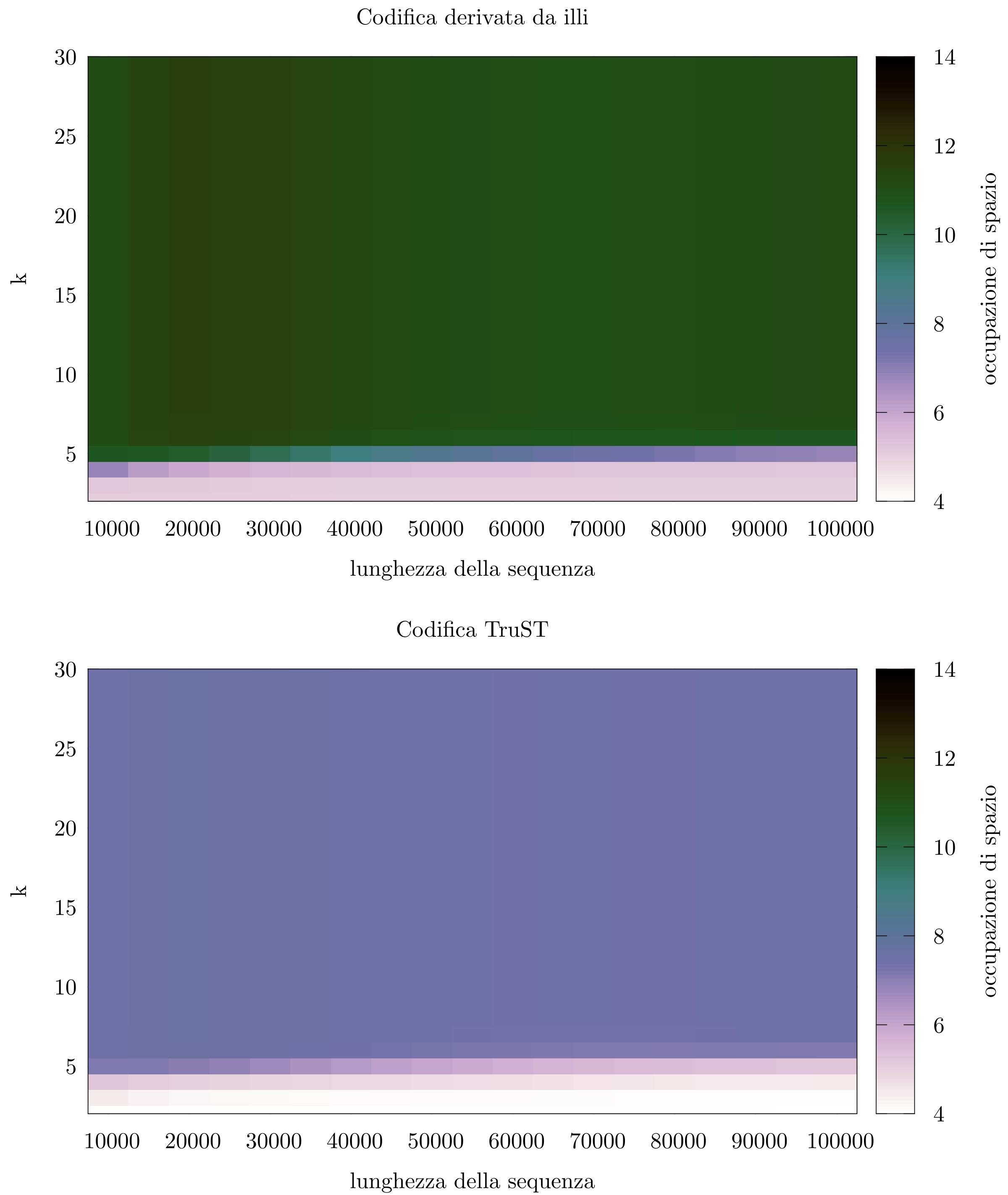


Figura 3.6: Occupazione di spazio (in Byte/simbolo) per la sequenza pi

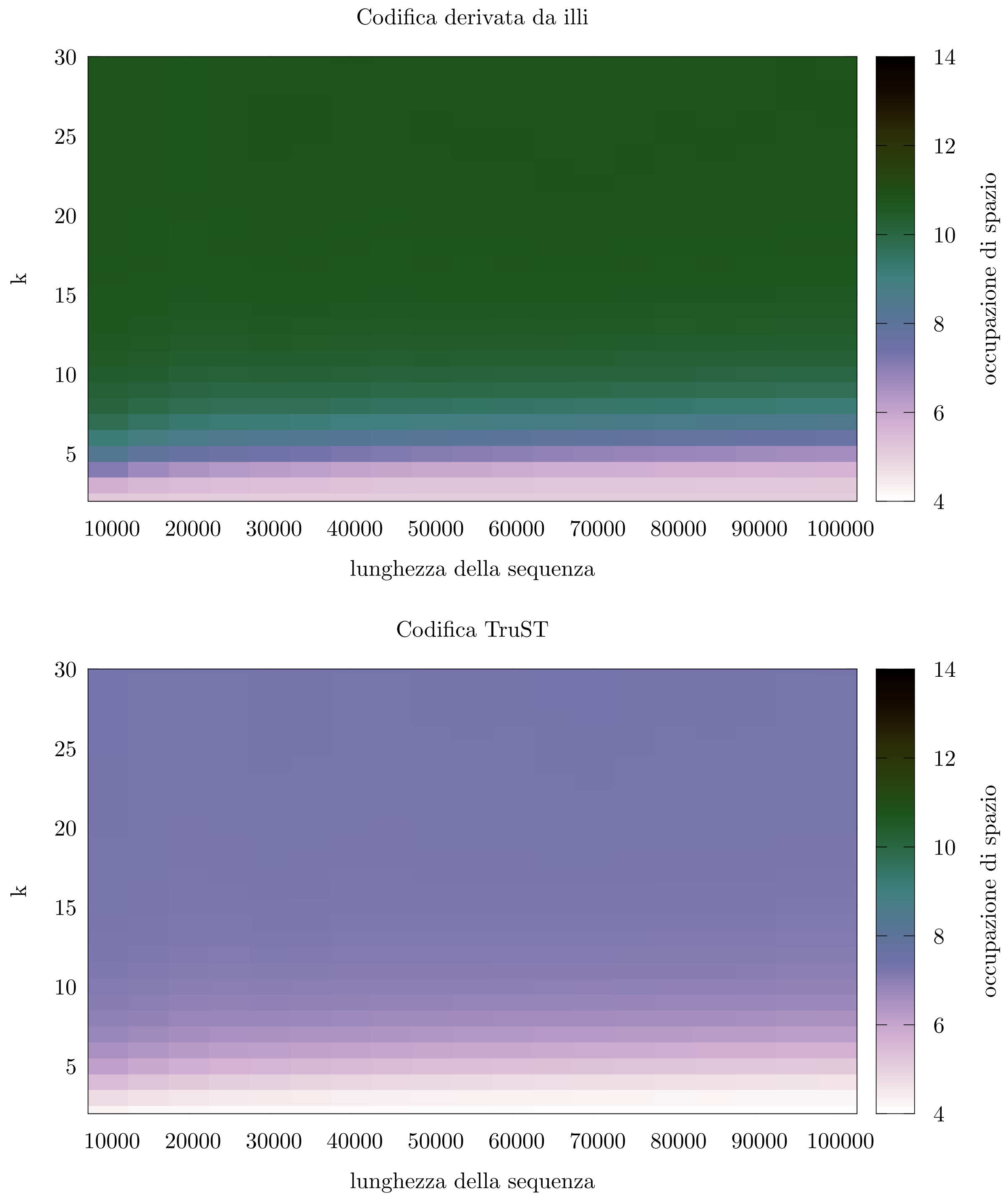


Figura 3.7: Occupazione di spazio (in Byte/simbolo) per la sequenza asyoulik

| | $k$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| lunghezza | 2 | 3 | 4 | 5 | 6 | 8 | 10 | 12 | 14 | 16 |
| 10000 | 5,01 | 5,03 | 5,13 | 5,52 | 6,98 | 12,32 | 13,43 | 13,55 | 13,56 | 13,56 |
| 20000 | 5,00 | 5,02 | 5,07 | 5,26 | 6,03 | 11,45 | 13,39 | 13,56 | 13,57 | 13,58 |
| 30000 | 5,00 | 5,01 | 5,04 | 5,17 | 5,69 | 10,64 | 13,30 | 13,56 | 13,57 | 13,58 |
| 40000 | 5,00 | 5,01 | 5,03 | 5,13 | 5,52 | 9,98 | 13,23 | 13,57 | 13,60 | 13,60 |
| 50000 | 5,00 | 5,01 | 5,03 | 5,10 | 5,42 | 9,43 | 13,12 | 13,55 | 13,59 | 13,59 |
| 60000 | 5,00 | 5,01 | 5,02 | 5,09 | 5,35 | 9,04 | 13,03 | 13,53 | 13,56 | 13,57 |
| 70000 | 5,00 | 5,00 | 5,02 | 5,07 | 5,30 | 8,64 | 12,93 | 13,51 | 13,56 | 13,56 |
| 80000 | 5,00 | 5,00 | 5,02 | 5,07 | 5,26 | 8,31 | 12,82 | 13,49 | 13,54 | 13,55 |
| 90000 | 5,00 | 5,00 | 5,01 | 5,06 | 5,23 | 8,05 | 12,76 | 13,47 | 13,54 | 13,54 |

Tabella 3.10: Occupazione di spazio in Byte per simbolo della codifica derivata da illi per la sequenza AE000111

| | $k$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| lunghezza | 2 | 3 | 4 | 5 | 6 | 8 | 10 | 12 | 14 | 16 |
| 10000 | 4,01 | 4,04 | 4,15 | 4,58 | 5,57 | 8,08 | 8,65 | 8,71 | 8,72 | 8,72 |
| 20000 | 4,01 | 4,02 | 4,08 | 4,30 | 5,00 | 7,64 | 8,62 | 8,72 | 8,72 | 8,72 |
| 30000 | 4,00 | 4,01 | 4,05 | 4,20 | 4,72 | 7,27 | 8,57 | 8,71 | 8,72 | 8,72 |
| 40000 | 4,00 | 4,01 | 4,04 | 4,15 | 4,56 | 6,97 | 8,53 | 8,72 | 8,73 | 8,73 |
| 50000 | 4,00 | 4,01 | 4,03 | 4,12 | 4,46 | 6,71 | 8,47 | 8,71 | 8,73 | 8,73 |
| 60000 | 4,00 | 4,01 | 4,03 | 4,10 | 4,39 | 6,54 | 8,43 | 8,69 | 8,71 | 8,72 |
| 70000 | 4,00 | 4,01 | 4,02 | 4,09 | 4,33 | 6,36 | 8,38 | 8,69 | 8,71 | 8,72 |
| 80000 | 4,00 | 4,01 | 4,02 | 4,07 | 4,29 | 6,20 | 8,32 | 8,67 | 8,70 | 8,71 |
| 90000 | 4,00 | 4,00 | 4,02 | 4,07 | 4,26 | 6,08 | 8,29 | 8,67 | 8,70 | 8,70 |

Tabella 3.11: Occupazione di spazio in Byte per simbolo della codifica TruST per la sequenza AE000111

| lunghezza | $k$ = 2 | 3 | 4 | 5 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10000 | 5,02 | 5,18 | 6,77 | 10,58 | 11,03 | 11,06 | 11,06 | 11,06 | 11,06 | 11,06 |
| 20000 | 5,01 | 5,09 | 5,88 | 10,37 | 11,39 | 11,48 | 11,48 | 11,48 | 11,48 | 11,48 |
| 30000 | 5,01 | 5,06 | 5,59 | 9,72 | 11,29 | 11,41 | 11,41 | 11,41 | 11,41 | 11,41 |
| 40000 | 5,00 | 5,04 | 5,44 | 8,99 | 11,01 | 11,19 | 11,19 | 11,19 | 11,19 | 11,19 |
| 50000 | 5,00 | 5,04 | 5,35 | 8,38 | 10,81 | 11,05 | 11,05 | 11,05 | 11,05 | 11,05 |
| 60000 | 5,00 | 5,03 | 5,29 | 7,89 | 10,70 | 10,99 | 10,99 | 10,99 | 10,99 | 10,99 |
| 70000 | 5,00 | 5,03 | 5,25 | 7,50 | 10,63 | 10,97 | 10,97 | 10,97 | 10,97 | 10,97 |
| 80000 | 5,00 | 5,02 | 5,22 | 7,20 | 10,59 | 11,00 | 11,00 | 11,00 | 11,00 | 11,00 |
| 90000 | 5,00 | 5,02 | 5,20 | 6,96 | 10,59 | 11,06 | 11,06 | 11,06 | 11,06 | 11,06 |

Tabella 3.12: Occupazione di spazio in Byte per simbolo della codifica derivata da illi per la sequenza pi

| lunghezza | $k$ = 2 | 3 | 4 | 5 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10000 | 4,05 | 4,41 | 5,23 | 7,11 | 7,36 | 7,38 | 7,38 | 7,38 | 7,38 | 7,38 |
| 20000 | 4,02 | 4,20 | 4,98 | 7,00 | 7,56 | 7,61 | 7,61 | 7,61 | 7,61 | 7,61 |
| 30000 | 4,02 | 4,14 | 4,91 | 6,66 | 7,49 | 7,56 | 7,56 | 7,56 | 7,56 | 7,56 |
| 40000 | 4,01 | 4,10 | 4,82 | 6,28 | 7,33 | 7,43 | 7,43 | 7,43 | 7,43 | 7,43 |
| 50000 | 4,01 | 4,08 | 4,72 | 5,97 | 7,22 | 7,36 | 7,36 | 7,36 | 7,36 | 7,36 |
| 60000 | 4,01 | 4,07 | 4,64 | 5,73 | 7,16 | 7,33 | 7,33 | 7,33 | 7,33 | 7,33 |
| 70000 | 4,01 | 4,06 | 4,56 | 5,54 | 7,12 | 7,32 | 7,32 | 7,32 | 7,32 | 7,32 |
| 80000 | 4,01 | 4,05 | 4,49 | 5,41 | 7,10 | 7,33 | 7,33 | 7,33 | 7,33 | 7,33 |
| 90000 | 4,01 | 4,05 | 4,44 | 5,30 | 7,11 | 7,36 | 7,37 | 7,37 | 7,37 | 7,37 |

Tabella 3.13: Occupazione di spazio in Byte per simbolo della codifica TruST per la sequenza pi

| lunghezza \ $k$ | 2 | 3 | 4 | 5 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10000 | 5,10 | 5,75 | 7,07 | 8,30 | 9,14 | 10,04 | 10,38 | 10,55 | 10,62 | 10,65 |
| 20000 | 5,05 | 5,45 | 6,44 | 7,67 | 8,62 | 9,74 | 10,20 | 10,43 | 10,54 | 10,60 |
| 30000 | 5,03 | 5,34 | 6,19 | 7,40 | 8,40 | 9,67 | 10,21 | 10,46 | 10,58 | 10,64 |
| 40000 | 5,03 | 5,27 | 6,02 | 7,19 | 8,22 | 9,58 | 10,16 | 10,42 | 10,55 | 10,62 |
| 50000 | 5,02 | 5,23 | 5,91 | 7,04 | 8,10 | 9,52 | 10,15 | 10,42 | 10,55 | 10,63 |
| 60000 | 5,02 | 5,20 | 5,82 | 6,93 | 7,99 | 9,44 | 10,11 | 10,40 | 10,54 | 10,62 |
| 70000 | 5,01 | 5,17 | 5,76 | 6,82 | 7,88 | 9,37 | 10,08 | 10,39 | 10,54 | 10,63 |
| 80000 | 5,01 | 5,16 | 5,70 | 6,72 | 7,78 | 9,30 | 10,04 | 10,36 | 10,52 | 10,61 |
| 90000 | 5,01 | 5,14 | 5,65 | 6,63 | 7,69 | 9,23 | 10,00 | 10,35 | 10,51 | 10,61 |

Tabella 3.14: Occupazione di spazio in Byte per simbolo della codifica derivata da illi per la sequenza asyoulik

| lunghezza \ $k$ | 2 | 3 | 4 | 5 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10000 | 4,23 | 4,74 | 5,47 | 6,08 | 6,48 | 6,92 | 7,09 | 7,17 | 7,21 | 7,23 |
| 20000 | 4,13 | 4,50 | 5,11 | 5,74 | 6,21 | 6,75 | 6,98 | 7,10 | 7,15 | 7,18 |
| 30000 | 4,10 | 4,40 | 4,96 | 5,60 | 6,10 | 6,72 | 6,98 | 7,11 | 7,17 | 7,20 |
| 40000 | 4,08 | 4,34 | 4,85 | 5,49 | 6,01 | 6,67 | 6,95 | 7,08 | 7,15 | 7,18 |
| 50000 | 4,06 | 4,29 | 4,78 | 5,40 | 5,94 | 6,64 | 6,95 | 7,08 | 7,15 | 7,19 |
| 60000 | 4,06 | 4,26 | 4,72 | 5,33 | 5,88 | 6,60 | 6,93 | 7,07 | 7,15 | 7,19 |
| 70000 | 4,05 | 4,24 | 4,67 | 5,28 | 5,83 | 6,56 | 6,91 | 7,07 | 7,15 | 7,19 |
| 80000 | 4,04 | 4,22 | 4,63 | 5,22 | 5,77 | 6,53 | 6,89 | 7,05 | 7,14 | 7,18 |
| 90000 | 4,04 | 4,20 | 4,59 | 5,17 | 5,72 | 6,50 | 6,88 | 7,05 | 7,13 | 7,18 |

Tabella 3.15: Occupazione di spazio in Byte per simbolo della codifica TruST per la sequenza asyoulik

# Capitolo 4

# Conclusioni

In questa tesi sono state analizzate diverse strutture dati basate sui suffissi. Si è potuto concludere che, in ogni caso, le prestazioni, in termini di occupazione di spazio, della codifica TruST sono state superiori rispetto a quelle della codifica derivata da illi. Si deve, però, notare che le due codifiche non sono equivalenti e gli algoritmi che necessitano dei suffix link possono operare solo nel caso si usi la codifica derivata da illi.

Inoltre, si è osservato come, considerando le versioni dell'enhanced suffix array che non usano gli accorgimenti implementativi atti a ridurre l'occupazione di spazio delle tabelle *lcp*, *child* e *link*, tale struttura dati necessiti di uno spazio di memorizzazione più ampio rispetto al suffix tree.

Se, al contrario, vengono usati tali accorgimenti implementativi, l'enhanced suffix array può occupare un minor spazio rispetto al suffix tree. Alcuni casi in cui si verifica ciò sono già stati analizzati, ma, in generale, tale situazione si verifica dipendentemente dal valore di $k$ usato e dalle caratteristiche della sequenza (lunghezza e struttura della sequenza stessa e cardinalità dell'alfabeto).

È comunque doveroso notare che, come è già stato osservato in precedenza, molti algoritmi necessitano di informazioni aggiuntive rispetto al solo enhanced suffix array con le tabelle $suf$, *lcp*, *child* ed, eventualmente, *link*. In tali casi il suffix tree, con la codifica derivata da illi o TruST a seconda della necessità di memorizzare i suffix link, può comportarsi meglio dell'enhanced suffix array indipendentemente dal valore di $k$. Un esempio di tali situazioni è costituito dal profile matching, dove in [9] viene descritto l'enhanced suffix array necessario per tale specifica applicazione, che risulta occupare uno spazio di 10 Byte per simbolo. La codifica TruST, invece, occupa 10 Byte per simbolo solo nel caso peggiore ma, in generale, le sue prestazioni sono migliori rispetto all'enhanced suffix array.

# Bibliografia

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

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

[3] S. Aluru e P. Ko. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2–4):143–156, 2005.

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

[5] A. Apostolico e O. Denas. Fast algorithms for computing sequence distances by exhaustive substring composition. *Algorithms for Molecular Biology*, 3(13), 2008.

[6] A. Apostolico, C. Pizzi e G. Satta. Optimal discovery of subword associations in strings. In *Proceedings of the 7th International Conference on Dicovery Science*, DS'04, pagine 270–277, 2004.

[7] A. Apostolico, C. Pizzi e E. Ukkonen. Efficient algorithms for the discovery of gapped factors. *Algorithms for Molecular Biology*, 6(1), 2011.

[8] R. Arnold e T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Data Compression Conference*, DCC '97, pagine 201–210, 1997.

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

[10] R. Giegerich e S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(13):331–353, 1997.

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

[12] S. Inenaga, H. Bannai, H. Hyyrö, A. Shinohara, M. Takeda, K. Nakai e S. Miyano. Finding optimal pairs of cooperative and competing patterns with bounded distance. In *Proceedings of the 7th International Conference on Discovery Science*, DS'04, pagine 32–46, 2004.

[13] D. K. Kim, H. Park, K. Park e J. S. Sim. Linear-time construction of suffix arrays. In *Proceedings of the 14th annual conference on Combinatorial pattern matching*, CPM'03, pagine 186–199, 2003.

[14] S. Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[15] J. Kärkkäinen, P. Sanders e S. Burkhardt. Simple linear work suffix array construction. In *Proceedings of the 30th international conference on Automata, languages and programming*, ICALP'03, pagine 943–955, 2003.

[16] M.-F. Sagot L. Marsan. Extracting structured motifs using a suffix tree – algorithms and application to promoter consensus identification. In *Proceedings of the fourth annual international conference on Computational molecular biology*, RECOMB '00, pagine 210–219, 2000.

[17] U. Manber e G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

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

[19] J. C. Na, A. Apostolico, C. S. Iliopoulos e K. Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304:87–101, 2003.

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

[21] P. Weiner. Linear pattern matching algorithms. In *IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory*, pagine 1–11, 1973.