Scuola Superiore dell'Università
degli Studi di Udine

Classe Scientifico-Economica

Anno Accademico 2021/22

LCP sketching

Allievo
Alessandro Minisini

Relatore
Prof. Nicola Prezza

Sommario

In questa tesina verrà analizzato il problema del *longest common prefix* e verrà descritto uno sketch che ne da una $(1-\varepsilon)$-approssimazione in spazio $O(\log_{1+\varepsilon} n)$ e tempo di calcolo del *longest common prefix* logaritmico rispetto alla dimensione dello sketch. Più avanti si descrive l'utilizzo combinato degli sketches con strutture dati ordinate (BST e Y-Fast-Trie) e l'utilizzo di queste nella compressione LZ76. Nell'ultima sezione vengono presentati i risultati sperimentali ottenuti dall'implementazione della fattorizzazione LZ76 con sketch di varia dimensione, col fine di analizzare la qualità di compressione di vari valori di ε.

Indice

1 Introduzione

Il *longest-common-prefix (LCP)* rappresenta il più lungo prefisso in comune tra due stringhe. Questa misura viene spesso associata al LCP array, che salva la lunghezza del longest-common-prefix tra due suffissi lessicograficamente adiacenti di un testo. Il LCP array [23] è principalmente usato per velocizzare gli algoritmi che fanno uso del Suffix Array; è noto che combinando un Suffix Array compresso e il LCP array (anch'esso compresso) si ottiene una struttura dati tanto potente quanto un Suffix Tree[12], con il vantaggio di usare molto meno spazio. Questo fatto ha sicuramente influenzato la ricerca di nuovi risultati per la costruzione del LCP array ([21], [4], [6],[9],[13]). Tuttavia, ci sono problemi in cui sono necessari i singoli valori di LCP, senza Suffix Array. Infatti, i valori di LCP danno importanti informazioni sulla ripetitività di un testo e sono utili per l'analisi sia di testi scritti (per l'individuazione di testi plagiati[17]) e in sequenze genomiche (problemi di mappabilità[24]).
In più il calcolo del LCP su un insieme di stringhe permette di rispondere alle query *successor* e *predecessor* (vedere sezione 6.3). Le principali strutture dati che implementano questi due metodi sono divise in *comparison-based* e *digitali*[3]. Le prime assumono che due stringhe possono essere confrontate in tempo costante, mentre le seconde lavorano sulla rappresentazione binaria delle stringhe. La struttura dati *comparison-based* per eccellenza è l'albero binario bilanciato, mentre le strutture *digitali* sono diventate oggetto di studio con l'introduzione degli alberi di van Emde Boas [32] e fusion trees[8], ottime alternative ai primi.
In questa tesina viene descritto uno sketch che permette di calcolare un'approssiamzione di LCP tra due stringhe in tempo logaritmico rispetto alla dimensione dello sketch con alta probabilità. Questo risultato viene reso possibile grazie all'hashing polinomiale[29], che permette il confronto tra stringhe in tempo costante. Su questo sketch viene poi definita una relazione d'ordine che permette l'utilizzo di strutture dati dinamiche ordinate. Vengono quindi analizzate due strutture dati, una *comparison-based* (vedi sezione 5.1) e una *digitale* (vedi sezione 6.3). Nell'ultima sezione invece viene descritto l'utilizzo dello sketch nella compressione Lempel-Ziv, in particolare nella ricerca veloce di sequenze di caratteri.

2 Notazioni

Le lettere maiuscole verranno utilizzate per indicare **stringhe**, mentre le lettere minuscole indicano **interi**; $A[i]$ indica il carattere in posizione i (il pri-

mo simbolo è $A[1]$); $A \cdot B$ indica A **concatenato** B; $A[\dots i], A[i \dots j], A[j \dots]$ indicano rispettivamente un **prefisso**, una **sottostringa** e un **suffisso**; le lettere $\mathbb{A}$, $\mathbb{B}, \dots$ indicano un **insieme** di stringhe.; $(k : v)$ indica una coppia **chiave**/**valore** tipica dei dizionari; $\Gamma = [(k_1 : v_1), (k_2 : v_2), \dots]$ indica un **dizionario** e $\Gamma[k]$ restituisce v, dove (k, v) è l'unica coppia nel dizionario che contiene k come primo elemento.

3 Funzione LCP

Definizione 3.1 (Longest common prefix)**.** Siano date A, B tali che $n = |A| \leq |B|$. Si definisca

$$\operatorname{lcp}(A, B) = \max_{0 \leq l \leq n} l \mid A[\dots l] = B[\dots l]$$

la lunghezza del massimo prefisso in comune tra le due stringhe.

4 Sketching

Sia $\alpha \in \mathbb{R}$ e $\kappa_{z,q}$ una funzione di hash polinomiale [29].
Si definisca lo sketch di S ($n = |S|$) come

$$\Gamma_S = \left[\left(i : \kappa_{z,q}(S[\dots \lfloor \alpha^i \rfloor])\right), \quad i, \alpha \in \mathbb{R} \wedge \alpha^i \leq n\right]$$

ovvero l'insieme degli hash dei prefissi della stringa S lunghi quanto le potenze intere di α. Lo sketch è composto in tutto da $O(\log_\alpha n) = O\left(\frac{\log n}{\log \alpha}\right)$ valori di hash.

Teorema 1. *Si definisca un errore $0 < \varepsilon < 1$ e poniamo $\alpha = 1 + \varepsilon$. Date A, B di lunghezza n, poniamo $q = n^c$ con c costante. Sia $i^*(A, B) = \max\left[i \mid \Gamma_A[i] = \Gamma_B[i]\right]$.*
Allora α^{i^*} è una $(1 - \varepsilon)$-approximation di $\operatorname{lcp}(A, B)$ con alta probabilità[†].

Dimostrazione. Per come è stato definito lo sketch, si può osservare che $\frac{\operatorname{lcp}(A,B)}{\alpha} \leq \alpha^{i^*} \leq \operatorname{lcp}(A, B)$. quindi

$$\frac{\alpha^{i^*}}{\operatorname{lcp}(A, B)} \geq \frac{\operatorname{lcp}(A, B)}{\alpha} \cdot \frac{1}{\operatorname{lcp}(A, B)} = \frac{1}{\alpha}$$

$$\frac{1}{\alpha} = \frac{1}{1 + \varepsilon} = \frac{1 - \varepsilon}{1 - \varepsilon^2} = 1 + \frac{\varepsilon^2 - \varepsilon}{1 - \varepsilon^2} = 1 - \frac{\varepsilon}{1 + \varepsilon} > 1 - \varepsilon$$

[†]In inglese *with high probability* (*w.h.p.*), indica una probabilità è inversamente proporzionale ad una potenza del numero di elementi di cui si calcola l'hash.

Poiché $\varepsilon > 0$.
Quindi $(1-\varepsilon) \cdot \text{lcp}(A,B) < \alpha^{i^*} \leq \text{lcp}(A,B)$, dunque α^{i^*} è una $(1-\varepsilon)$-approximation di $\text{lcp}(A,B)$. □

D'ora in poi $\text{lcp}(A,B)$ indicherà anche il valore approssimato, siccome la distinzione non è più necessaria.

4.1 Relazione d'ordine

Definizione 4.1 (Operatore $\leq$)**.** Date A, B ($|A| \leq |B|$), siano Γ_A, Γ_A i relativi sketch e $i_{next}(A,B) = i^*(A,B) + 1$, con $i^*(A,B)$ definito come nel Teorema 1. Sia poi k la dimensione di Γ_A.
Si definisca $\leq\colon [0,q)^k \times [0,q)^k \to \{0,1\}$

$$(\Gamma_A \leq \Gamma_B) = \begin{cases} \text{Falso} & \text{se } \Gamma_A[i_{next}(A,B)] > \Gamma_B[i_{next}(A,B)] \\ \text{Vero} & \text{altrimenti (anche se } i_{next} > |A|) \end{cases}$$

Quindi $\leq$ rappresenta la relazione d'ordine ($\leq$ o $>$) che c'è tra le prime due *fingerprint* (numeri interi in questo caso) differenti tra le due stringhe.

Teorema 2 (Relazione d'ordine)**.** *L'operatore $\leq$ definito in 4.1 è una relazione d'ordine totale.*

Dimostrazione. Affinché $\leq$ sia una relazione d'ordine, occorre dimostrare la validità delle seguenti proprietà.

1. **Proprietà riflessiva**: Siccome $i_{next}(A,A) > |A|$, $\Gamma_A \leq \Gamma_A = Vero$
2. **Proprietà antisimmetrica**: Si noti che $i_{next}(A.B) = i_{next}(B,A)$, quindi affinché la proprietà valga per gli sketch deve valere per $\Gamma_A[i_{next}(A,B)]$ e $\Gamma_B[i_{next}(A,B)]$. Siccome queste due fingerprint sono intere, la proprietà antisimmetrica vale.
3. **Proprietà transitiva**: Date A, B, C e siano $\Gamma_A, \Gamma_B, \Gamma_C$ i rispettivi sketch. Ciò che vogliamo dimostrare è che se vale $\Gamma_A \leq \Gamma_B$ e $\Gamma_B \leq \Gamma_C$, allora vale $\Gamma_A \leq \Gamma_C$. Definiamo $j = \min(i_{next}(A,B), i_{next}(B,C)$.
   Se uno dei tre valori $\Gamma_A[j],\ \Gamma_B[j], \Gamma_C[j]$ non dovesse esistere, allora si assegni a questo il valore -1 (più piccolo di tutti i valori possibili). Questo fatto non altera la relazione d'ordine e rende più semplice la dimostrazione.
   Per ipotesi sappiamo che

$$\Gamma_A[j] \leq \Gamma_B[j] \quad e \quad \Gamma_B[j] \leq \Gamma_C[j]$$

Siccome $\Gamma_A[j],\ \Gamma_B[j], \Gamma_C[j]$ sono interi (che godono della propietà transitiva) si ottiene che:

$$\Gamma_A[j] \leq \Gamma_C[j] \Rightarrow \Gamma_A \leq \Gamma_C$$

4. **Proprietà d'ordine totale**: Siccome valgono le tre proprietà definite sopra e la relazione è definita per qualsiasi coppia di sketch, questa è una relazione d'ordine totale.

□

4.1.1 Calcolo di $\Gamma_A \leq \Gamma_B$

Si noti che il calcolo di $\leq$ si basa tutto sul trovare i^*, siccome ciò che ci interessa è l'indice immediatamente successivo, facile da trovare in tempo costante. Banalmente, per trovare i^* si potrebbe scorrere tutto lo sketch fino a che non si trova il primo mismatch, in tempo $O(\log_\alpha n)$, non efficiente. Per ottimizzare la ricerca, si noti il seguente fatto.

Osservazione 1. Date A, B e Γ_A, Γ_B i rispettivi sketch, per ogni coppia di indici $i < j \leq \log_\alpha n$ vale che:

$$\Gamma_A[i] \neq \Gamma_B[i] \Rightarrow \Gamma_A[j] \neq \Gamma_B[j]$$

$$\Gamma_A[j] = \Gamma_B[j] \Rightarrow \Gamma_A[i] = \Gamma_B[i]$$

Poiché il prefisso $[\dots \alpha^i]$ di ogni stringa è contenuto nel prefisso $[\dots \alpha^j]$.

Questo ci permette di fare binary-search su i^*, ottimizzando il calcolo a $O(\log\log_\alpha n) = O(\log\log n + \log\varepsilon^{-1})$.

4.2 Analogia con i numeri interi

Notiamo che lo sketch sopra definito è una lista di interi dell'insieme $[0, q)$ in cui è stata definita una relazione d'ordine calcolabile in modo efficiente. Grazie a questa proprietà possiamo vedere gli sketch come dei veri e propri numeri interi in base q e quindi utilizzare tutti gli algoritmi e le strutture dati definiti per gli interi conosciute fino ad oggi. Definire le operazioni di base $(+, -, \dots)$ è possibile, ma non sarebbero utili nel contesto del problema considerato. E' molto più utile concentrarsi sui confronti, quindi di seguito verranno descritte alcune strutture dati su cui si può utilizzare lo sketch.

5 LCP su un Insieme

Lo sketching sopra proposto risolve il problema tra due stringhe in modo approssimato. Ora si vedrà come usare l'idea descritta per confrontare stringhe e insiemi di stringhe.

Definizione 5.1 (Metrica estesa)**.** Dati $\mathbb{A}$ un insieme di stringhe e X una stringa, si definisca

$$\mathrm{lcp}(X, \mathbb{A}) = \max_{S \in \mathbb{A}} \mathrm{lcp}(X, S)$$

il massimo lcp tra X e una stringa di $\mathbb{A}$.

5.1 Primo approccio: Binary Search Tree

Supponiamo che l'insieme $\mathbb{A}$ contenga solo sketch di stringhe **tutte diverse** tra loro.
Grazie alla relazione d'ordine sopra descritta è possibile costruire un albero binario che supporti insertion, deletion e search in tempo $O((\log \log_\alpha n) \log k)$, con k numero di elementi nell'albero e n massima lunghezza delle stringhe in $\mathbb{A}$.
Le due operazioni utili per il calcolo di lcp sono:

- $\Upsilon.lower\_bound(k)$: il più grande valore x in Υ tale che $x \leq k$.
- $\Upsilon.upper\_bound(k)$: il più piccolo valore x in Υ tale che $x \geq k$.

Entrambe queste operazioni vengono eseguite in tempo $O((\log \log_\alpha n) \log k)$.

Lemma 0.1. *Sia Υ un BST che contiene gli sketch delle stringhe dell'insieme $\mathbb{A}$ e X una stringa. Siano poi $L \in \mathbb{A}$ una stringa tale che $\Gamma_L = \Upsilon.lower\_bound(\Gamma_X)$ e $U \in \mathbb{A}$ una stringa tale che $\Gamma_U = \Upsilon.upper\_bound(\Gamma_X)$ Allora:*

$$\mathrm{lcp}\,(X, \mathbb{A}) = \max\big\{\,\mathrm{lcp}\,(X, L), \mathrm{lcp}\,(X, U)\big\}$$

Dimostrazione. Per dimostrare il lemma, basta dimostrare che:

$$\forall\ \Gamma_Y \in \mathbb{A} \mid \Gamma_Y < \Gamma_L \quad \mathrm{lcp}(Y, X) \leq \mathrm{lcp}(X, L) \tag{1}$$

$$e$$

$$\forall\ \Gamma_Y \in \mathbb{A} \mid \Gamma_Y > \Gamma_U \quad \mathrm{lcp}(Y, X) \leq \mathrm{lcp}(X, U) \tag{2}$$

Si dimostrino separatamente.
Dimostrazione di (1): Definisco $i_1^* = \mathrm{lcp}(X, Y)$ e $i_2^* = \mathrm{lcp}(X, L)$ e supponiamo per assurdo che $i_1^* > i_2^*$.

Allora, calcolando $\text{lcp}(Y, L)$, notiamo che $\Gamma_Y[i_2^*] = \Gamma_X[i_2^*] > \Gamma_L[i_2^*]$ per ipotesi, quindi $\Gamma_Y > \Gamma_L$, assurdo.
Dimostrazione di (2): Analogamente, definisco $i_1^* = \text{lcp}(X, Y)$ e $i_2^* = \text{lcp}(X, U)$ e supponiamo per assurdo che $i_1^* > i_2^*$.
Allora, calcolando $\text{lcp}(Y, U)$, notiamo che $\Gamma_Y[i_2^*] = \Gamma_X[i_2^*] < \Gamma_U[i_2^*]$ per ipotesi, quindi $\Gamma_Y < \Gamma_U$, assurdo. □

6 Y-Fast Trie

L' Y-Fast-Trie[33] è una struttura dati definita su numeri interi e fornisce le seguenti operazioni:

- `find(`k`):` trova se il valore k è presente nella struttura
- `predecessor(`k`):` trova il più grande numero x tale che $x \leq k$
- `successor(`k`):` trova il più piccolo numero x tale che $x \geq k$
- `insert(`k`):` inserisce k nella struttura
- `delete(`k`):` rimuove k dalla struttura, se presente

e ognuna di queste operazioni viene eseguita in tempo $O(\log \log m)$ (ammortizzato per la modifica), dove m è la dimensione dell'insieme universo.

Definizione 6.1 (Modello macchina)**.** Per descrivere il funzionamento della struttura, definiamo il seguente modello macchina:

- La RAM della macchina su cui stiamo lavorando è divisa in words da w bit.
- La macchina su cui stiamo lavorando esegue le seguenti funzioni del linguaggio C in tempo costante.

```
+   -   *   /   %   «   »   &   |   ^   ==   <=
```

E supponiamo $m = 2^w$.
Divideremo la descrizione dell'Y-Fast-Trie in tre parti:

1. BinaryTrie: versione base di un trie su interi
2. X-Fast-Trie: miglioramento temporale del BinaryTrie
3. Y-Fast-Trie: miglioramento spaziale dello X-Fast-Trie

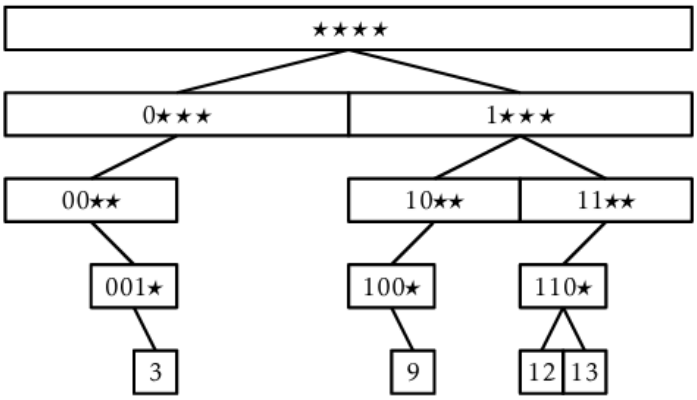

Figura 1: Rappresentazione di un BinaryTrie

6.1 BinaryTrie

Un BinaryTrie[25] codifica un insieme di interi da w bit in un albero binario. Tutte le foglie dell'albero hanno profondità w e ogni intero è codificato da un percorso radice→foglia. Il percorso dell'intero x gira a sinistra al livello i se i-esimo bit più significativo di x è uno 0, mentre gira a destra se è un 1. La figura 1 rappresenta in esempio con $w = 4$ in cui il BinaryTrie contiene i valori 3, 9, 12, 13.

Siccome il percorso di ricerca di un valore x dipendono dai bit di x, è utile chiamare i figli di un nodo u, `u.child[0]` (left) e `u.child[1]` (right). Questi puntatori avranno un doppio scopo. Poiché le foglie di un binary trie non hanno figli, i puntatori saranno usati per connettere le foglie assieme in una doubly-linked list. Per ogni foglia nel binary trie `u.child[0]` (prev) è il nodo che viene prima di u nella lista e `u.child[1]` (next) è il nodo che segue u nella lista. Da qui in avanti `u.child[0],u.left e u.prev` verranno usati per indicare lo stesso campo del nodo u, come per `u.child[1], u.right e u.next`.

Ogni nodo, u, contiene anche un puntatore `u.jump`. Se il figlio sinistro manca, `u.jump` punta alla più piccola foglia nel sottoalbero di u. La Figura 2 aggiunge i puntatori `jump` al trie descritto nella Figura 1.

Se invece manca il figlio destro di u, esso punterà alla foglia più grande del sottoalbero di u.

La procedura `find(x)` in un BinaryTrie è molto intuitiva. Proviamo a seguire il percorso di ricerca per x nel trie. Se raggiungiamo una foglia, allora abbiamo trovato x. Se raggiungiamo un nodo u dal quale non possiamo procedere (perché a u manca uno dei due figli), seguiamo `u.jump`, che ci porta o alla

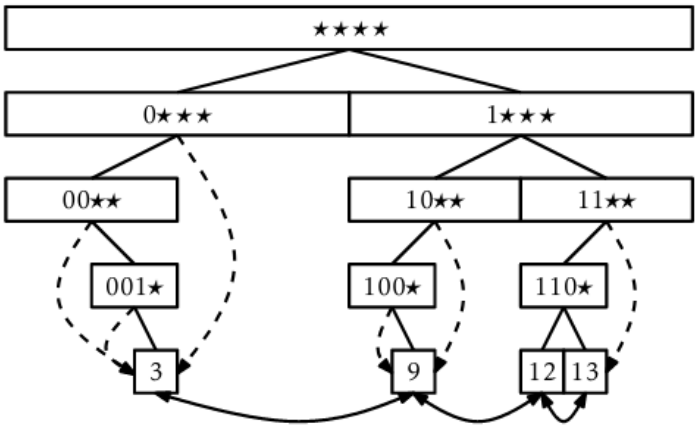

Figura 2: Un BinaryTrie con aggiunti i puntatori `jump`.

più piccola foglia maggiore di x o alla più grande foglia minore di x. Quale di questi due opzioni capiti dipende dal fatto che manchi il figlio destro o sinistro di u. A questo punto possiamo scegliere se ritornare il `successor(x)` o il `predecessor(x)`, spostandoci eventualmente tra uno e l'altro grazie alla linked-list. Questa procedura viene schematizzata nella Figura 6.1.

Il tempo di esecuzione della procedura `find(x)` è dato dal tempo che richiede per completare il percorso radice→foglia, quindi è $O(w)$.
La procedura `add(x)` in un BinaryTrie è anche lei molto intuitiva, ma più complessa:

1. Segue il percorso per x finché non raggiunge un nodo u dove non può più procedere.

2. Crea il resto del percorso da u alla foglia che contiene x.

3. Aggiunge il nodo u', che contiene x, alla linked-list di foglie (notare che l'accesso alla foglia precedente è dato da `u.jump`, dove u è il nodo incontrato nel punto 1.)

4. Ripercorre all'indietro il percorso di ricerca per x sistemando i puntatori `jump` ai nodi il cui `jump` deve puntare a x. L'inserimento viene rappresentato in Figura 4.

Questa procedura completa il percorso radice→foglia due volte, una in avanti e una indietro. Ogni step del percorso richiede tempo costante, quindi `add(x)` lavora in un tempo $O(w)$.
La procedura `remove(x)` inverte ciò che è stato fatto da `add(x)`:

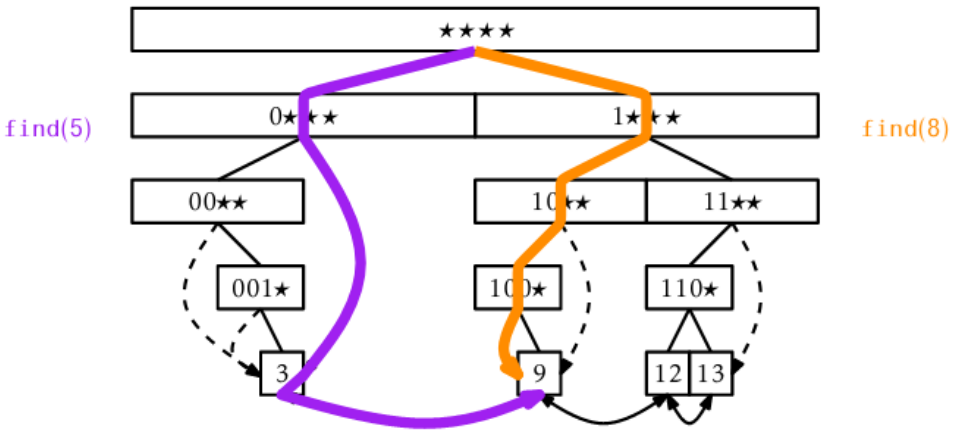

Figura 3: Percorsi seguiti dalle chiamate `find(5)` e `find(8)`.

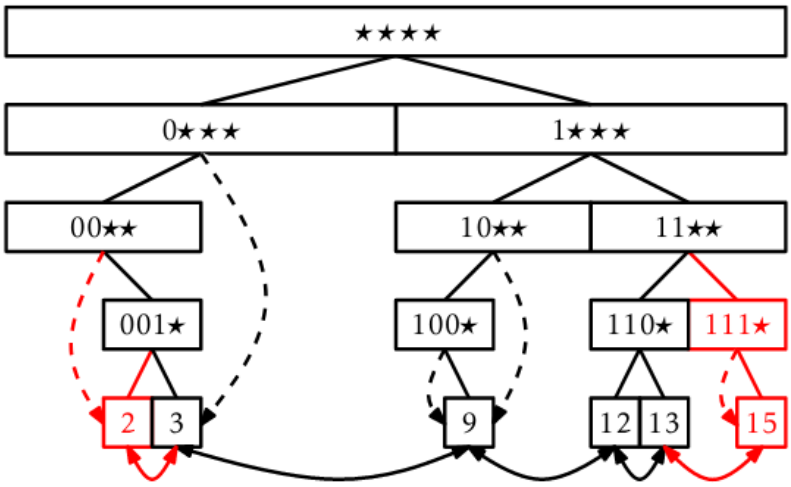

Figura 4: Inserimento dei valori 2 e 5 al BinaryTrie.

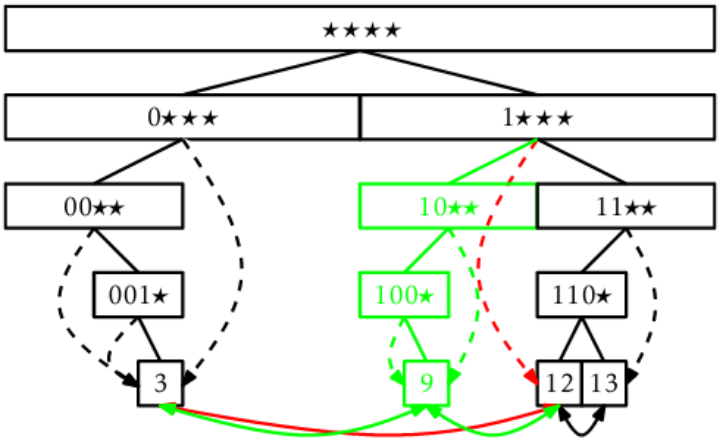

Figura 5: Rimozione del valore 9 dal BinaryTrie

1. Segue il percorso di ricerca per x finché non raggiunge la foglia u che contiene x.

2. Rimuove u dalla linked-list.

3. Elimina u e quando ripercorre il percorso all'indietro elimina tutti i nodi che incontra, finché non raggiunge un nodo v con un figlio che non sta sul percorso di x.

4. Ripercorre i passi fino alla radice da v aggiornando tutti i `jump` che puntano a u. Questa procedura viene illustrata in Figura 5.

Riassumendo, un BinaryTrie supporta le operazione `find(x)`, `successor(x)`, `predecessor(x)`, `add(x)`, `remove(x)` in tempo $O(w)$ per singola operazione. Lo spazio che usa per salvare n valori è $O(n \cdot w)$.

6.2 X-Fast-Trie

Per migliorare le prestazioni del BinaryTrie, introduciamo la struttura X-Fast-Trie, che è un BinaryTrie con $w+1$ hash tables, una per ogni livello del trie. Queste tabelle di hash servono a velocizzare la procedura `find(x)` ad un tempo $O(\log w)$. Ricordiamo che la procedura `find(x)` in un BinaryTrie è completata sostanzialmente quando viene raggiunto il nodo u, dove il percorso di x procederebbe per `u.right` (o `u.left`) se u avesse un figlio destro (o sinistro). A questo punto la ricerca passa per `u.jump` per giungere alla foglia v e terminando restituendola o restituendo il suo successore nella

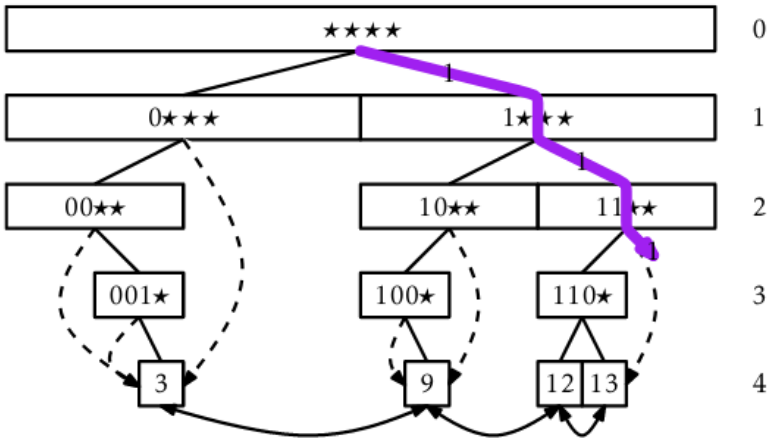

Figura 6: Siccome non esiste nessun valore con prefisso $111\star$, la ricerca si ferma al livello 2.

linked-list. L' X-Fast-Trie ottimizza il tempo di ricerca facendo ricerca binaria sui livelli del trie per localizzare il nodo u.
Per usare la ricerca binaria, è necessario determinare se il nodo u che stiamo cercando sta sopra ad uno specifico livello i o sta sotto. Questa informazione è data dagli i bit più significativi della rappresentazione binaria di x; questi bit determinano se il percorso di ricerca che x segue dalla radice al livello i. Facendo riferimento alla figura 6, se venisse fatta la chiamata `find(14)`, la ricerca si esaurirebbe al nodo $11\star\star$, siccome al livello 3 non esiste nessun nodo etichettato con $111\star$.

Quindi possiamo etichettare ogni nodo al livello i con un intero a i bit. Ora il nodo u che stiamo cercando sarà al di sotto del livello i se la sua etichetta è uguale agli i bit più significativi di x.
In un X-Fast-Trie, salviamo, per ogni $i \in \{0, \ldots, w\}$, tutti i noti al livello i in una hash table, `t[i]`. In questo modo possiamo verificare in tempo costante se esiste un nodo al livello i la cui etichetta è uguale agli i bit più significativi di x. Le hash table `t[0],...,t[w]` permettono di fare ricerca binaria per trovare u.
Inizialmente, sappiamo che u si trova ad un livello i con $0 \leq i < w + 1$. Inizializziamo quindi `l=0` e `h=w+1` e accediamo ricorsivamente alla hash table `t[i]`, con `i=`$\lfloor$`(l+h)/2`$\rfloor$. Se `t[i]` contiene un nodo la cui etichetta è uguale agli i bit più significativi di x impositamo `l=i` (u è al livello i o al di sotto); altrimenti impostiamo `h=i` (u sta sopra il livello i). Questo processo termina quando $h - l \leq 1$, nel cui caso abbiamo trovato u al livello l. A questo punto

ci basta completare `find(x)` usando `u.jump` e la linked-list delle foglie.
Ogni iterazione della procedura dimenzza l'intervallo $[l, h]$, quindi trova u in tempo $O(\log w)$. Ogni iterazione utilizza un tempo costante di operazioni e una chiamata alla hash table, che lavora in tempo atteso costante. Quindi la procedura `find(x)` lavora in tempo atteso $O(\log w)$.
Le procedure `add(x)` e `remove(x)` di un X-Fast-Trie sono sostanzialmente identiche a quelle del BinaryTrie. Le uniche modifiche sono per gestire le tabelle di hash `t[0],...,t[w]`. Durante la procedura `add(x)`, quando un nuovo noto viene creato al livello i, questo nodo viene aggiunto a `t[i]`. Durante la procedura `remove(x)`, quando un nodo viene rimosso al livello i, questo nodo viene rimosso da `t[i]`. Siccome aggiungere e togliere elementi da una hash table richiede un tempo atteso costante, questo non aumenta il tempo di esecuzione di `add(x)` e `remove(x)`.
Riassumendo, un X-Fast-Trie supporta le operazioni

- `add(x)` e `remove(x)` in tempo atteso $O(w)$ e
- `find(x), successor(x), predecessor(x)` in tempo atteso $O(\log w)$.

Lo spazio usato dalla struttura per salvare n valori è $O(n \cdot w)$.

6.3 Y-Fast-Trie

Per migliorare lo spazio utilizzato dall'X-Fast-Trie e le prestazioni delle procedure `add(x)` e `remove(x)`, introduciamo l'Y-Fast-Trie. Un Y-Fast-Trie usa un X-Fast-Trie, `xft`, ma vi ci salva solo $O(n/w)$ valori. In questo modo, lo spazio totale usato da `xft` è solo $O(n)$. Inoltre, solo uno ogni w chiamate a `add(x)` o `remove(x)` dell'Y-Fast-Trie vengono fatte a `xft`. Facendo questo, il costo medio delle chiamate a `xft.add(x)` e `xft.remove(x)` è costante.
Gli altri $\frac{n}{w}$ elementi vengono salvati BST, descritti nella sezione 5.1. Ci sono $O(n/w)$ alberi binari e in ognuno di questi verranno salvati $O(w)$ elementi. Un albero binario bilanciato supporta tutte le operazioni di un Y-Fast-Trie in tempo $O(\log w)$, come voluto.
Più concretamente, un Y-Fast-Trie contiene un X-Fast-Trie, `xft`, che contiene un sottoinsieme casuale di elementi, dove ogni elemento appare nel sottinsieme indipendentemente con probabilità $\frac{1}{w}$. Siano $x_0 < x_1 < \cdots < x_{k-1}$ gli elementi salvati in `xft`. Ad ogni elemento x_i viene associato un BST t_i, che contiene tutti i valori nell'intervallo $x_{i-1} + 1, \ldots, x_i$. Un Y-Fast-Trie è rappresentato in Figura 7
La procedura `find(x)` in un Y-Fast-Trie è molto semplice. Cerchiamo x nell'`xft` e troviamo un valore x_i associato all'albero t_i. Poi usiamo il metodo `find(x)` del BST per rispondere alla query.

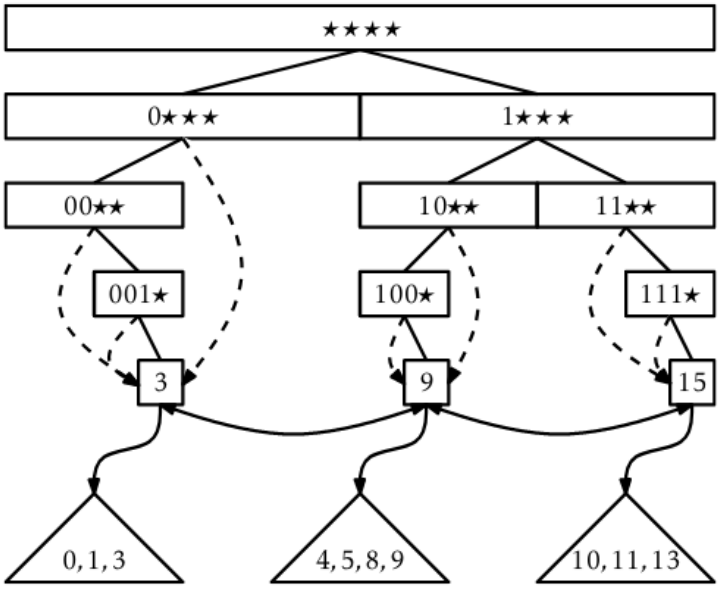

Figura 7: Un Y-Fast-Trie contenente in valori $0, 1, 3, 4, 5, 8, 9, 10, 11, 13$.

La procedura `find(x)` di `xft` richiede tempo $O(\log w)$. La procedura `find(x)` del BST richiede tempo $O(\log r)$, dove r è la grandezza del BST. Siccome il valore atteso della grandezza del BST è $O(w)$, il tempo diventa $O(\log w)$. La procedura `add(x)` chiama `xft.find(x)` per trovare il BST t in cui x deve essere inserito. Dopo chiama `t.add(x)` per aggiungere x a t. A questo punto, con probabilità $\frac{1}{w}$, x viene aggiunto a `xft`. Se questo avviene, il BST t deve essere diviso in due BST, t' e t". Il BST t' contiene tutti i valori minori o uguali a x; t" è il BST originale t, a cui sono stati rimossi gli elementi di t'. A questo punto, aggiungiamo la coppia (x, t') a `xft`. La Figura 8 mostra un esempio di questa procedura.

Aggiungere x a t richiede tempo $O(\log w)$. Dividere un BST richiede anch'esso tempo $O(\log w)$. Aggiungere (x, t') a `xft` richiede tempo $O(w)$, ma avviene solo con probabilità $\frac{1}{w}$.
Quindi il tempo atteso di `add(x)` è

$$O(\log w) + \frac{1}{w}O(w) = O(\log w)$$

La procedura `remove(x)` inverte i passaggi di `add(x)`. Usiamo `xft` per trovare la foglia u che corrisponde a `xft.find(x)`. Da t, ottenuto da u, viene rimosso x. Se x era salvato anche in `xft`, viene rimoso da quest'ultimo e gli elementi del BST associato ad x (t''')vengono salvati nel BST associato al

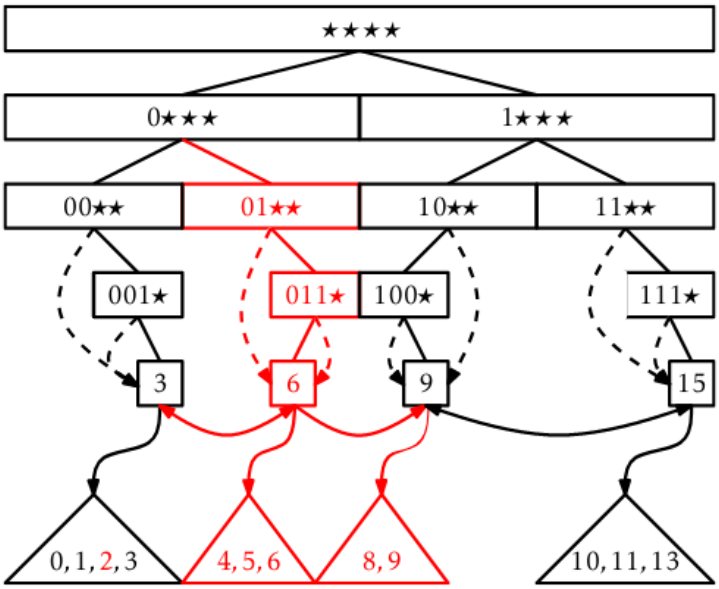

Figura 8: Inserimento dei valori 2 e 6. Inoltre, il valore 6 viene inserito anche nel X-Fast-Trie.

successore di u nella linked-list. Tutto questo è rappresentato in Figura 9
Trovare il nodo u in `xft` richiede tempo $O(\log w)$. Rimuovere x da t richiede anch'esso tempo $O(\log w)$. Unire due BST (t e t''') richiede tempo $O(\log w)$. Se necessario, rimuovere x da `xft` richiede tempo $O(w)$, ma solo se x era contenuto in `xft` con probabilità $\frac{1}{w}$. Quindi, il tempo medio della procedura `remove(x)` di un Y-Fast-Trie è $O(\log w)$.
Riassumendo, un Y-Fast-Trie supporta le operazioni `add(x)`, `remove(x)`, `find(x)`, `successor(x)`, `predecessor(x)` in tempo medio $O(\log w)$. Lo spazio usato da un Y-Fast-Trie che contiene n elementi è $O(n)$.

6.3.1 Utilizzo con lo sketch

Innanzitutto notiamo che le funzioni `predecessor` e `successor` sono analoghe a `lower_bound` e `upper_bound`, quindi vale sempre:

$$\operatorname{lcp}(X, \mathbb{A}) = \max\left\{\operatorname{lcp}(X, L), \operatorname{lcp}(X, U)\right\}$$

con L stringa tale che $\Gamma_L =$ `predecessor`(Γ_X) e U stringa tale che $\Gamma_U =$ `successor`(Γ_X).

Come detto nella sezione 4.2 possiamo utilizzare il nostro sketch come se fosse un numero intero. Siccome ogni hash appartiene a $[0, q)$ (con q modulo

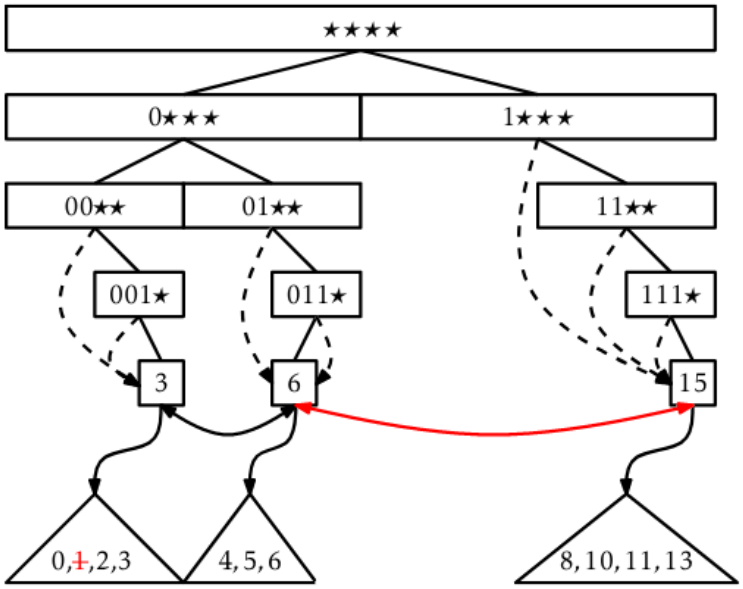

Figura 9: Rimozione dei valori 1 e 9 dalla struttura.

utilizzato in Rabin-Karp) lo sketch occupa in totale $O(\log n \cdot \varepsilon^{-1} \cdot \log q)$ bit. Siccome abbiamo scelto $q = n^c$ con c costante, lo sketch occupa $O(\log^2 n \cdot \varepsilon^{-1})$ bit. Per calcolare il tempo di ricerca quindi è necessario tenere conto del tempo richiesto da `xft.find(x)`, che analizzeremo in seguito, e dal tempo del BST di supporto, $O((\log\log n + \log \varepsilon^{-1})^2)$ (vedere la sezione 5.1). Per analizzare il tempo di esecuzione di `xft.find(x)` e necessario vedere come avviene la ricerca. Ad ogni livello i della ricerca binaria descritta in 6.2, i primi i bit di x vengono cercati nella hash table `t[i]`. Il tempo di ricerca viene considerato costante nel modello definito in 6.1, ma in questo caso il numero di livelli, che corrisponde al numero di bit di cui sono formati gli sketch, può essere molto maggiore di w. Non è possibile quindi salvare i prefissi interi degli sketch nelle hash table `t`. Per poter quindi confrontare in tempo costante i prefissi, ci basta richiamare l'Osservazione 1: nel livello i, tutti gli sketch che differiscono da x nei bit in posizione $(i - \log q), \ldots, i$ (che corrispondono alla fingerprint del prefisso $T[\ldots i]$), differiscono da x anche per tutti i bit in posizione $1, \ldots, i$. Grazie a questo fatto, è possibile etichettare i nodi di ogni livello i con $\log q$ bit invece che i bit.
La complessità di ogni accesso alle hash table rimane costante, permettendo ad ogni operazione di lavorare in tempo $O(\log\left(\log^2 n \cdot \varepsilon^{-1}\right)) = O(\log\log n + \log \varepsilon^{-1})$.

Confrontando i tempi di esecuzione si ottiene che questo secondo approccio è più efficiente del primo (vedere sezione 5.1) quando

$$\varepsilon \geq \frac{\log n}{n} \tag{3}$$

7 Compressione

Gli algoritmi di compressione stanno ricevendo sempre più attenzione nel mondo della ricerca a causa del sempre più alto tasso di crescita al quale i dati si stanno accumulando in archivi quali il web e i database genomici. Molto spesso questi insiemi di dati sono però ripetitivi, come archivi di sequenze geniche simili o repositories, e quindi possono essere compresse anche di 1000 volte. Oltre allo stoccaggio di questi dati in forma efficiente, è diventato un problema di interesse anche costruire delle strutture dati che consentano operazioni sui dati compressi, in modo da poter lavorare anche su macchine con risorse limitate. La compressione di queste strutture è basata su un insieme di tecniche che includono la compressione ad entropia, parsing Lempel-Ziv [35],[20] (LZ76/LZ77), compressione mediante grammatiche[5] e la trasformata di Burrows-Wheeler (BWT)[2]. La compressione mediante grammatiche, la trasformata di Barrows-Wheeler codificata mediante Run-Length econding [31](RLBWT) e LZ77 si sono dimostrati molto efficienti nella compressione di dati altamente ripetitivi e quindi, di conseguenza, molti ricercatori si stanno focalizzando sullo studio di queste tecniche.

7.1 Fattorizzazioni Lempel-Ziv

Questa sezione è dedicata all'applicazione di lcp su insieme nell'algoritmo che computa la fattorizzazione LZ76/LZ77.

Definizione 7.1 (Fattorizzazione Lempel-Ziv 76)**.** Sia T un testo. La fattorizzazione LZ76 di T è una sequenza di triple (o fattori)

$$LZ76(T) = \langle \pi_i, \lambda_i, c_i \rangle_{i=1\ldots z}$$

dove $\pi_i \in \{0, \ldots, n-1\} \cup \{\perp\}$ e $\perp$ rappresenta la posizione indefinita, $\lambda_i \in \{0, \ldots, n-2\}, c_i \in \Sigma$, e:

1. $T = \omega_1 c_1 \ldots \omega_z c_z$, con $\omega_i = \epsilon$ se $\lambda_i = 0$ e $\omega_i = T[\pi_i, \ldots, \pi_i + \lambda_i - 1]$ altrimenti.

2. Per ogni $i = 1 \ldots z$, la stringa ω_i è la più lunga stringa ad occorrere almeno due volte in $\omega_1 c_1 \ldots \omega_i$.

LZ77 è una variante di LZ76 in cui ogni ω_i è la più lunga stringa ad occorrere nella sottostringa $T[i-l \dots i]$, per un certo l scelto a priori. LZ77 è usato largamente per comprimere qualsiasi tipo di testo, poiché combina una compressione accettabile con una decompressione veloce. Su testi tipici, valori di l compresi tra 2^{12} e 2^{15} sono sufficienti per trovare ripetizioni abbastanza lunghe[26]. Su testi ripetitivi è comunque preferibile LZ76, poiché in questo caso è molto probabile trovare lunghe ripetizioni molto indietro nel testo.

7.2 LZ76 parsing

La fattorizzazione LZ76 può essere trovata in tempo lineare con l'aiuto di un suffix tree [30] ma questa struttura richiede molto spazio aggiuntivo, che rende più costoso il parsing di testi lunghi.Ricerche molto più recenti cercano di calcolare LZ76 in spazio ridotto ([27], [1], [16], [14], [15], [11], [10], [34], [7], [28], [19]). In questa tesina utilizzeremo lcp insieme per fattorizzare un testo T di lunghezza n in tempo $O(n \log n \log \log_\alpha n)$ e spazio $O(n)$ parole di memoria.
L'algoritmo può essere descritto dai seguenti passaggi:

Data: T testo di lungezza n
Result: Sequenza di triple (π, λ, c)
$index \leftarrow 0$
while $index \leq n$ **do**
  match$\leftarrow$ il più lungo prefisso di $T[index \dots]$ che compare in $T[\dots index-1]$
  if *esiste un match* **then**
    $\pi \leftarrow$ posizione in cui inizia il match
    $\lambda \leftarrow$ lunghezza del match
    $c \leftarrow$ carattere che segue il match in $T[index \dots]$
  else
    $\pi \leftarrow 0$
    $\lambda \leftarrow 0$
    $c \leftarrow$ primo carattere in $T[index \dots]$
  end
  output (π, λ, c)
  $index \leftarrow index + \lambda + 1$
end

Algorithm 1: Schema generale LZ76

L'algoritmo elabora tutto il testo a partire dalla prima posizione e cerca

sempre la più grande sequenza di caratteri non processati che occorre nella parte di testo già fattorizzata, e se viene trovata, genera la tripla composta da posizione del match π, la lunghezza del match λ e il carattere che segue il match nella sequenza non processata c. A questo punto retituisce in output la tripla e avanza di $\lambda + 1$ caratteri per poter processare un nuovo blocco. Se non trova il match restituisce $(0, 0, c)$, con c primo carattere non processato. Il tempo di esecuzione è dato dal tempo richiesto per trovare il match e da altre istruzioni che richiedono tempo costante. Notare che ogni carattere viene processato una sola volta dall'algoritmo, quindi il tempo richiesto per ogni carattere è $O(1)$. E' facile notare che il collo di bottiglia dell'algoritmo è la ricerca del match: se questa viene fatta banalmente scandendo ogni carattere in $T[\dots index]$, la complessità totale diventa $O(n^2)$.
Per velocizzare l'algoritmo, notiamo che:

- una sequenza di caratteri che inizia in posizione $index$ è un prefisso di $T[index \dots]$,
- la sottostringa $T[i \dots j]$ è il $(j-i+1)$-esimo prefisso del suffisso $T[i \dots]$.

Il match richiesto è quindi il più lungo prefisso di $T[index \dots]$ che è anche un prefisso dei suffissi $T[i \dots]$, con $0 \leq i < index$. Detto in modo formale, quello che cerchiamo è:

$$\max_{i \in \{0,\dots,index-1\}} \text{lcp}(T[index \dots], T[i \dots]) = \text{lcp}(T[index \dots], \mathbb{S})$$

con $\mathbb{S} = \{T[i \dots], 0 \leq i < index\}$.
Come abbiamo visto nelle sezioni 5.1 e 6.3, possiamo gestire questo tipo di query in tempo, $O(\log n \log \log_\alpha n)$.
L'algoritmo seguente schematizza l'utilizzo delle strutture in LZ76:
Il tempo di esecuzione di ogni iterazione dell'algoritmo è dato dal tempo di esecuzione delle funzioni `lower_bound(x)` e `upper_bound(x)` e `add(x)` (che viene eseguita per ogni suffisso), $O(\log n \log \log_\alpha n)$, dal tempo richiesto da lcp, $O(\log \log_\alpha n)$ e da operazioni costanti. Il tempo totale di esecuzione del ciclo while è quindi $O(n \log n \log \log_\alpha n)$.
Ora analizziamo st. Se salviamo gli sketch in st, lo spazio occupato dalla struttura è $O(n \log_\alpha n)$, siccome contiene n sketch grandi $O(\log_\alpha n)$ parole di memoria. Per migliorare questo risultato, definiamo l'array A_T come l'insieme delle fingerprint di tutti i prefissi di T. Questo array occupa $O(n)$ parole di memoria, siccome il numero di prefissi di T è n. Per accedere alla fingerprint $\kappa_{z,q}(T[i \dots j])$, posso usare la seguente formula:

$$\kappa_{z,q}(T[i \dots j)) = (A_T[j] - A_T[i-1])z^{-i} \mod q$$

Data: T testo di lunghezza n
Result: Sequenza di triple (π, λ, c)
$index \leftarrow 0$
$st \leftarrow$ BST/Y-Fast-Trie inizialmente vuoto
while $index \leq n$ **do**
  match$_1 \leftarrow$`st.lower_bound(`$T[index \dots]$`)`
  match$_2 \leftarrow$`st.upper_bound(`$T[index \dots]$`)`
  if *esiste un match* **then**
    if lcp$(match_1, T[index \dots])$ > lcp$(match_2, T[index \dots])$ **then**
      $\pi \leftarrow$ posizione in cui inizia match$_1$
      $\lambda \leftarrow$ lunghezza del match$_1$
      $c \leftarrow$ carattere che segue il prefisso in comune con $match_1$ di $T[index \dots]$
    else
      $\pi \leftarrow$ posizione in cui inizia match$_2$
      $\lambda \leftarrow$ lunghezza del match$_2$
      $c \leftarrow$ carattere che segue il prefisso in comune con $match_2$ di $T[index \dots]$
    end
  else
    $\pi \leftarrow 0$
    $\lambda \leftarrow 0$
    $c \leftarrow$ primo carattere in $T[index \dots]$
  end
  output (π, λ, c)
  for i ***in*** *index... (index+*λ*)* **do**
    st.add(fingerprint di $T[i \dots]$)
  end
  $index \leftarrow index + \lambda + 1$
end

Algorithm 2: Algoritmo per la fattorizzazione LZ76 con sketch lcp

Dove z^{-i} indica l'inverso moltiplicativo di z^i modulo q.
Precalcolando i valori di $z^{-i},\ 0 \leq i \leq n$ possiamo calcolare la fingerprint in tempo costante.
In *st* quindi possiamo salvare solamente l'indice iniziale di ogni suffisso e per accedere alle fingerprint usiamo l'array A_T. Lo spazio occupato diventa quindi $O(n)$ parole di memoria.
Riassumendo, il tempo di esecuzione totale, dato dal tempo di esecuzione del ciclo while, è $O(n \log n \log_\alpha n)$. Siccome l'algoritmo utilizza 2 strutture che occupano spazio lineare, l'algoritmo utilizza $O(n)$ parole di memoria aggiuntive.

8 Risultati Sperimentali

Ho implementato l'algoritmo 2 in C/C++ utilizzando `std::set` come struttura dati per gestire le query lcp. Le fingerprint vengono rappresentate come coppie di interi da 32 bit, in modo da ottenere $\left(\frac{1}{2^{32}}\right)^2 = \frac{1}{2^{64}}$ come probabilità di collisione, sufficientemente bassa per i testi su cui l'algoritmo è stato testato. I datasets utilizzati sono `Pizza&Chili Corpus` e `Calgary Corpus`. Il primo comprende un testo in inglese, una sequenza di frequenze musicali, un codice sorgente, una sequenza di amminoacidi, una sequenza genomica e un testo XML. Ogni testo è stato ridotto a una dimensione di 10MB. Il secondo invece è formato da testi in inglese (libri, articoli scientifici e di giornale), codici sorgenti ed eseguibili. I file del secondo dataset sono più piccoli: non superano 1MB di memoria. Tutti gli esperimenti sono stati svolti su un computer con una CPU Intel i3-8130U da 2.20GHz, 8GB di RAM DDR3 e Pop! OS 22.04 LTS a 64-bit. Il codice C++ è stato compilato con g++ 11.2.0 con l'opzione -O3. Gli ε testati per ogni dataset sono 0.001, 0.006, 0.01, 0.06, 0.1, 0.2, 0.3, 0.4, 0.6, 0.8.

8.1 Qualità di commpressione

Ogni grafico in Figura 10 e 11 rappresenta un dataset: sull'asse delle ascisse viene indicato il valore ε^{-1}, proporzionale alla dimensione dello sketch utilizzato, e sull'asse delle ordinate si trova il rapporto $\gamma = \frac{LZ76(\varepsilon)}{LZ76(0)}$, dove $LZ76(\varepsilon)$ rappresenta il numero di frasi generate dall'algoritmo 2 in cui sono stati utilizzati sketch che forniscono una $(1-\varepsilon)$-approssimazione dei prefissi, mentre $LZ76(0)$ rappresenta il numero di frasi LZ76 ottimale.
Come possiamo osservare dai grafici, il rapporto γ sembra diminuire in modo esponenziale rispetto alla dimensione degli sketch. Questo risultato può essere considerato attendibile, poiché il numero di frasi LZ76 è direttamente

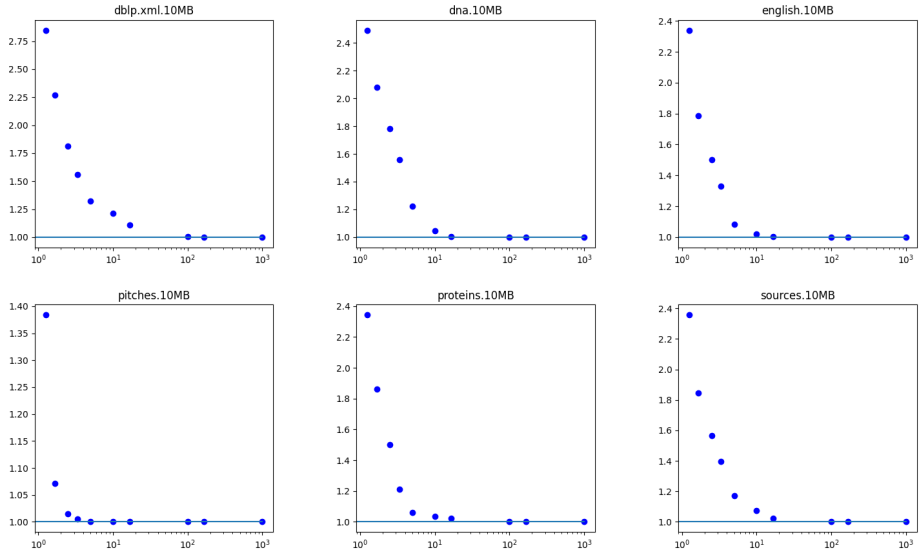

Figura 10: Grafici sulla qualità di compressione del dataset `Pizza & Chili Corpus`

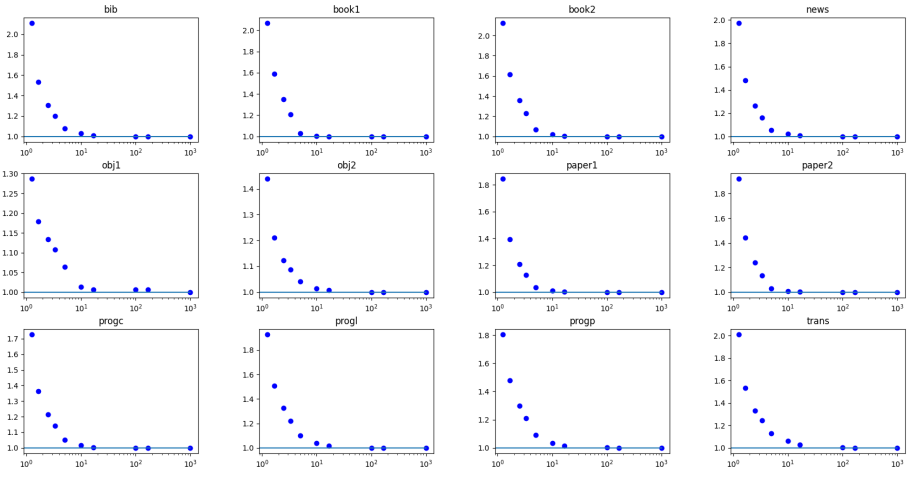

Figura 11: Grafici sulla qualità di compressione del dataset `Calgary Corpus`

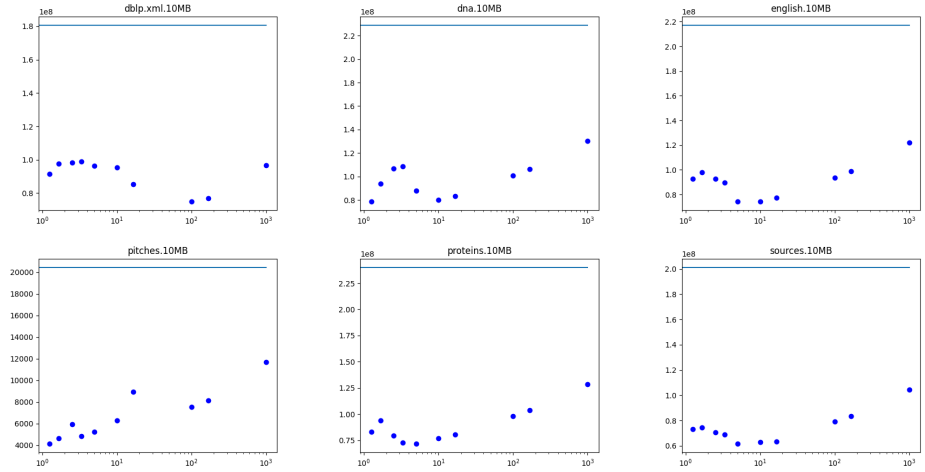

Figura 12: Grafici delle performance di $LZ76(\varepsilon)$ sul dataset `Pizza & Chili Corpus`

collegato con il numero di sottostringhe distinte del testo.[18] Le stringhe distinte di lunghezza n che condividono il prefisso lungo $(1-\varepsilon)n$ caratteri sono $\sigma^{\varepsilon n}$ (con σ numero di caratteri dell'alfabeto relativo al testo), quindi al crescere di ε il numero di sottostringhe distinte aumenta in modo esponenziale. Notiamo poi che per file di queste dimensioni il numero di frasi generate non supera mai il triplo delle frasi ottimali, e vi si avvicina considerevolmente con $\varepsilon = 0.1$. Tenendo conto di questo risultato, si può ottenere una buona compressione utilizzando sketch di dimensione molto ridotta (con un file da 10MB, la dimensione dello sketch è dell'ordine di 1KB).

8.2 Prestazioni

I grafici in Figura 12 e 13 rappresetano le performance dell'algoritmo $LZ76(\varepsilon)$ su ogni singolo file dei datasets. In questo caso i tempi sono espressi in microsecondi e la linea blu orizzontate corrisponde al tempo di esecuzione di $LZ76(0)$. Questo algoritmo è identico all'algoritmo 2 ma, per un testo lungo n, utilizza sketches con n fingerprints, una per ogni prefisso del testo. Il tempo asintotico di esecuzione dell'algoritmo è quindi $O(n \log^2 n)$. Sull'asse delle ascisse si trova sempre il valore ε^{-1}, mentre sulle ordinate viene indicato il tempo in microsecondi dell'algoritmo $LZ76(\varepsilon)$. I dati del dataset `Calgary Corpus` sono poco uniformi: nel file `bib` tutti i punti sono disposti in ordine crescente, mentre nel file `obj` i punti sono disposti senza un ordine apparente. In quest'ultimo poi l'algoritmo $LZ76(0)$ è più efficien-

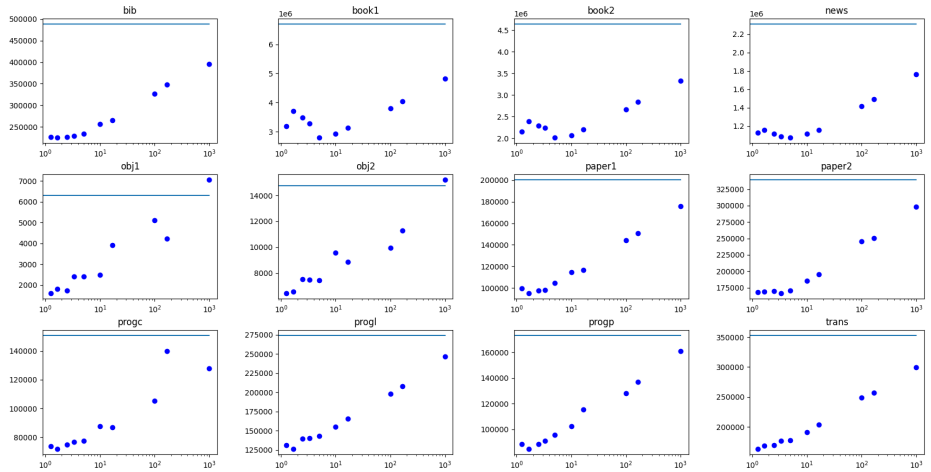

Figura 13: Grafici delle performance di $LZ76(\varepsilon)$ sul dataset `Calgary Corpus`

te dell'algoritmo $LZ76(0.001)$, assolutamente inconsistente con le premesse poste precedentemente. Questi risultati potrebbero essere legati alle piccole dimensioni dei file in questione: i file sono talmente piccoli che nel tempo di esecuzione entrano in gioco anche le fluttuazioni dovute a fattori esterni.
Molto più uniformi sono invece i grafici relativi al dataset `Pizza & Chili Corpus`: in questo caso il tempo medio di esecuzione si aggira attorno ai 100/120 secondi, sufficientemente lungo per rendere ininfluenti le piccole fluttuazioni. Sui grafici in questione si osserva un comportamento inatteso: il tempo di esecuzione decresce man mano che ci si avvicina a $\varepsilon = 0.1$, poi aumenta per ε intermedi e, per finire, riscende in corrispondenza di $\varepsilon = 0.8$. Per spiegare questo comportamento, consideriamo due operazioni che l'algoritmo $LZ76$ esegue: il calcolo di lcp e l'avanzamento di λ caratteri. Osserviamo innanzitutto che numero di volte che lcp viene calcolato è strettamente legato a λ: se si avanza sempre di tanti caratteri, il numero di volte che viene calcolato lcp diminuisce; viceversa, più i λ di ogni singola iterazione sono piccoli, più confronti saranno fatti e quindi lcp verrà calcolata più volte. Inoltre, anche la grandezza di λ è strettamente legata al calcolo di lcp: siccome questa funzione restituisce sempre una sottostima del valore vero, maggiore è la precisione dell'approssimazione, maggiore sarà il numero di caratteri trovati e quindi maggiore saranno le varie λ.
Tenendo conto di questi fatti, possiamo presumere che in 0.1 ci sia il giusto rapporto tra tempo richiesto per calcolare lcp e numero di caratteri processati per volta. Tenendo conto anche delle considerazioni fatte nella Sezione 8.1, possiamo dire che il valore di ϵ ottimale per file di queste dimensio-

ni è 0.1, siccome permette un'ottima compressione con tempi di esecuzione relativamente bassi.

9 Conclusioni

In questa tesina abbiamo descritto uno sketch che permette di calcolare in modo approssimato il *longest common prefix* tra due stringhe in spazio $O(\log_{1+\varepsilon} n)$ e tempo $O(\log \log_{1+\varepsilon} n)$ con alta probabilità grazie all'hashing polinomiale. E' stata poi definita una relazione d'ordine tra sketch che permette di salvarli in modo ordinato all'interno di strutture che implementano la funzione `successor` e `predecessor`; è stata poi descritta una loro possibile applicazione in un algoritmo di compressione. Altre possibili applicazioni di questo sketch potrebbero riguardare la ricerca di parole in dizionari, specialmente se le parole in essi contenute sono molto lunghe. In altri contesti potrebbero essere utilizzati come strutture di supporto per Suffix Array e Suffix Tree, permettendo di velocizzare il confronto - di base lineare - tra i vari suffissi.
Negli esperimenti si è poi visto che errori dell'ordine di grandezza di 10^{-1} sono pressoché ottimali, sia per qualità di compressione che per efficienza.

Riferimenti bibliografici

[1] Stephen Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, J. Ian Munro, Theis Rauhe, and Mikkel Thorup. Efficient tree layout in a multilevel memory hierarchy. *CoRR*, cs.DS/0211010, 2002.

[2] Z. Arnavut and S.S. Magliveras. Block sorting and compression. In *Proceedings DCC '97. Data Compression Conference*, pages 181–190, 1997.

[3] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic z-fast tries. In Edgar Chavez and Stefano Lonardi, editors, *String Processing and Information Retrieval*, pages 159–172, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[4] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and lcp arrays in external memory. *ACM J. Exp. Algorithmics*, 21, sep 2016.

[5] M. Charikar, E. Lehman, Ding Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[6] Johannes Fischer. Inducing the lcp-array. *CoRR*, abs/1101.3448, 2011.

[7] Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel ziv computation in small space (LZ-CISS). *CoRR*, abs/1504.02605, 2015.

[8] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, dec 1993.

[9] Simon Gog and Enno Ohlebusch. Compressed suffix trees: Efficient computation and storage of lcp-values. *ACM J. Exp. Algorithmics*, 18, may 2013.

[10] Keisuke Goto and Hideo Bannai. Simpler and faster lempel ziv factorization. *CoRR*, abs/1211.3642, 2012.

[11] Keisuke Goto and Hideo Bannai. Space efficient linear time lempel-ziv factorization on constant˜size˜alphabets. *CoRR*, abs/1310.1448, 2013.

[12] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[13] Juha Kärkkäinen and Dominik Kempa. Lcp array construction in external memory. *ACM J. Exp. Algorithmics*, 21, apr 2016.

[14] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time lempel-ziv factorization: Simple, fast, small. *CoRR*, abs/1212.2952, 2012.

[15] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight lempel-ziv parsing. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms*, pages 139–150, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[16] Dominik Kempa and Simon J. Puglisi. Lempel-ziv factorization: Simple, fast, practical. In *ALENEX*, 2013.

[17] Dmitry V. Khmelev and William J. Teahan. A repetition based measure for verification of text collections and for text categorization. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*, SIGIR '03, page 104–110, New York, NY, USA, 2003. Association for Computing Machinery.

[18] Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Towards a definitive measure of repetitiveness. *CoRR*, abs/1910.02151, 2019.

[19] Dominik Köppl and Kunihiko Sadakane. Lempel ziv computation in compressed space (LZ-CICS). *CoRR*, abs/1510.02882, 2015.

[20] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[21] Michael Luttenberger, Raphaela Palenta, and Helmut Seidl. Computing the longest common prefix of a context-free language in polynomial time. *CoRR*, abs/1702.06698, 2017.

[22] M. Léonard, L. Mouchard, and M. Salson. On the number of elements to reorder when updating a suffix array. *Journal of Discrete Algorithms*, 11:87–99, 2012. Special issue on Stringology, Bioinformatics and Algorithms.

[23] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[24] Giovanni Manzini. Longest common prefix with mismatches. In Costas Iliopoulos, Simon Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval*, pages 299–310, Cham, 2015. Springer International Publishing.

[25] Pat Morin. *Open Data Structures: An Introduction.* Athabasca University Press, 2013.

[26] Gonzalo Navarro. *Compact Data Structures: A Practical Approach.* Cambridge University Press, USA, 1st edition, 2016.

[27] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index, LZ factorization, and LCE queries in compressed space. *CoRR*, abs/1504.06954, 2015.

[28] Alberto Policriti and Nicola Prezza. From LZ77 to the run-length encoded burrows-wheeler transform, and back. *CoRR*, abs/1702.01340, 2017.

[29] M.O. Rabin. *Fingerprinting by Random Polynomials.* Center for Research in Computing Technology: Center for Research in Computing Technology. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[30] Michael Rodeh, Vaughan R. Pratt, and Shimon Even. Linear algorithm for data compression via string matching. *J. ACM*, 28(1):16–24, jan 1981.

[31] Jouni Siren, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. 11 2008.

[32] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[33] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space (n). *Information Processing Letters*, 17(2):81–84, 1983.

[34] Jun-ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster compact on-line lempel-ziv factorization. *CoRR*, abs/1305.6095, 2013.

[35] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

A Trovare ε ottimale

Per voler dare una stima dell'epsilon ottimale temporalmente in funzione della dimensione dei file, definisco un modello approssimativo basato sulle considerazioni fatte nella sezione 8.2. Supponendo di processare un testo lungo n caratteri e supponendo ogni λ_i costante, il tempo totale di esecuzione può essere descritto dalle seguenti funzioni

$$t(\varepsilon) = \frac{n}{\lambda(\varepsilon)}(\log_2 \log_2 n + \log_2 \varepsilon^{-1}) \tag{4}$$

$$\lambda(\varepsilon) = (1-\varepsilon)\log_2 n \tag{5}$$

dove $t(\varepsilon)$ rappresenta il tempo di esecuzione, formato dal tempo di calcolo di lcp moltiplicato per il numero di frasi generate. Siccome si considera λ costante, il numero di frasi generate corrisponde al rapporto tra la dimensione del testo e la dimensione di ogni match (ovvero λ). $\lambda(\varepsilon)$ corrisponde alla lunghezza di ogni match: viene considerata l'approssimazione $(1-\varepsilon)$ e la lunghezza media del *longest common prefix*, che possiamo considerare essere $\log_2 n$ su dati reali [22].
Sostituendo l'equazione 5 nella 4, si ottiene:

$$t(\varepsilon) = \frac{n}{(1-\varepsilon)\log_2 n}(\log_2 \log_2 n + \log_2 \varepsilon^{-1})$$

Calcolando il minimo della funzione per $n = 10^7$ (dimensione dei file nel dataset `Pizza & Chili Corpus`), si ottiene $\varepsilon \approx 0.16$, in linea con i dati sperimentali.