Appunti di
Elementi di Bioinformatica

A cura di:
Francesco Refolli
Matricola 865955

Anno Accademico 2022-2023

Chapter 1

Note sul Corso

todo: segnare delle note

Chapter 2

Pattern Matching

2.1 Introduzione

Pattern Matching Per **Pattern Matching** si intende trovare tutte le occorrenze di un pattern P di lunghezza m all'interno del testo T di lunghezza n.

Notazione Data una stringa o sequenza S, si identifica con $S[i:j]$ la sottostringa che contiene gli elementi da i a j (compreso).

2.2 Algoritmo Banale

Ragionamento L'algoritmo piu' semplice a cui si puo' pensare consiste nello scorrere linearmente il pattern P e provare per ogni posizione $i \in [1, n]$ la corrispondenza con una porzione di testo di uguale lunghezza.

Algorithm Confronta Stringhe

```
procedure ConfrontaStringhe(X, Y)
    for i ← 1 to n do
        if X[i] ≠ Y[i] then
            return false
        end if
    end for
    return true
end procedure
```

Algorithm Pattern Matching banale

```
procedure PMB(T, P)
    for i ← 1 to n do
        if ConfrontaStringhe(T[i : i + m − 1], P) then
            print(i)
        end if
    end for
end procedure
```

Procedura

Complessita' Come si puo' notare, sia PMB che $ConfrontaStringhe$ sono procedure la cui complessita' e' legata principalmente al singolo ciclo che contengono.
PMB contiene un ciclo di n iterazioni fisse, quindi il suo tempo nel caso medio sara' $T_{PMB} = \Theta(n)$.
ConfrontaStringhe al contrario contiene si un ciclo di m iterazioni, ma il numero di volte in cui saranno ripetuto il confronto dipendera' dalla similitudine delle due stringhe.
Nel caso medio sara' circa meta' dei caratteri, quindi $T_{ConfrontaStringhe} = \Theta(m/2) = \Theta(m)$.
Visto che PMB incorpora una chiamata a ConfrontaStringhe, la complessita' totale sara': $T(n, m) = \Theta(n * m)$.
Per quanto riguarda lo spazio, e' facilmente intuibile che sia $S(n) = \Theta(n + m)$.

2.3 Baeza-Yates-Gonnet

Approccio Bit-Parallel Quando si devono effettuare piu' operazioni dello stesso tipo poco costose e ripetitive e' possibile ridurre il problema ad azioni elementari che la CPU puo' processare in parallelo per ridurre il numero di passi da fare per completare un algoritmo. Per esempio se si devono sommare vettori di bit e' possibile assemblare una word che li contenga in modo da poi effettuare operazioni bit-wise (ovvero bit-a-bit, ogni bit non interferisce con quello successivo) per ottimizzare il calcolo. Si puo' applicare in certi casi anche agli algoritmi di pattern matching.

Ragionamento Si supponga di disporre di una matrice M di dimensione $m \cdot n$. Ogni cella e' riempita come segue:

Con $j = 0$:

$$M[i][j] = \begin{cases} 1 & \text{sse } P[i] = T[0] \\ 0 & \text{altrimenti} \end{cases}$$

Per ogni $j > 0$:

$$M[i][j] = \begin{cases} 1 & \text{sse } P[:i] = T[j - i + 1 : j] \\ 0 & \text{altrimenti} \end{cases}$$

Sappiamo per certo che:
$P[:i] = T[j - 1 + 1 : j] \iff P[:i - 1] = T[j - i + 1 : j - 1] \wedge P[i] = T[j]$.
Quindi possiamo scrivere la cella generica $M[i][j]$ in modo ricorsivo rispetto alla matrice:

Con $j = 0$:

$$M[i][j] = \begin{cases} 1 & P[i] \wedge T[0] \\ 0 & \text{altrimenti} \end{cases}$$

Per ogni $j > 0$:

$$M[i][j] = \begin{cases} 1 & M[i-1][j-1] = 1 \wedge P[i] = T[j] \\ 0 & \text{altrimenti} \end{cases}$$

Visto che $\wedge$ e' un'operazione bitwise rispetto a $(0, 1)$ possiamo scrivere:
Con $j = 0$:

$$M[i][j] = \begin{cases} 1 & P[i] \wedge T[0] \\ 0 & \text{altrimenti} \end{cases}$$

Per ogni $j > 0$:

$$M[i][j] = \begin{cases} 1 & M[i-1][j-1] \wedge P[i] \wedge T[j] \\ 0 & \text{altrimenti} \end{cases}$$

Definiamo un vettore U con $|U| = |P|$ come:

$$U(j) = \begin{cases} 1 & P[i] \wedge T[j] \\ 0 & \text{altrimenti} \end{cases}$$

Visto che ogni cella $M[i][j]$ e' il risultato di un $\wedge$ tra la cella $M[i-1][j-1]$ e $U(j)[i]$, possiamo sfruttare la colonna $M[\cdot][j-1]$ per generare la colonna $M[\cdot][j]$.

Da qui in poi chiamero' ω la dimensione in bit della word della CPU.
Per poter utilizzare $M[i-1][j-1]$ per $M[i][j]$ possiamo sfruttare l'operazione bitwise rshift su $M[\cdot][j-1]$ e inserire un 1 in posizione 0:
$tmp_{M[\cdot][j-1]} = (M[\cdot][j-1] >> 1) \vee (1 << (\omega - 1))$.

Quindi:
$M[\cdot][j] = tmp_{M[\cdot][j-1]} \wedge U(j)$

Per trovare quindi le occorrenze di P occorrera' quindi controllare l'ultima riga di M. Se $M[|P|-1][j] = 1$ si dice che P occorre in T in posizione j.

Visto che queste operazioni sono bitwise possiamo usare il paradigma Bit-Parallel trattando le colonne $M[\cdot][j]$ come un numero intero processabile dalla CPU. Lo stesso vale per il vettore $U(j)$.

Algorithm Baeza Yates Gonnet

```
procedure BYG(T, P)
    V ← U(0)
    for j = 1 to n do
        tmp ← (V >> 1) ∨ (1 << (ω − 1))
        V ← (tmp ∧ U(j))
        if V mod 2 = 1 then
            print(j)
        end if
    end for
end procedure
```

Procedura

Complessita' Di primo acchito potrebbe sembrare che $T(n,m) = \Theta(n)$. Nell'algoritmo abbiamo un solo ciclo con $\Theta(n)$ iterazioni.
Tuttavia questo avviene sse $|P| \leq \omega$. In quel caso infatti $T_{U(j)} = \Theta(1)$, perche' la costruzione di U richiedera' sempre $O(\omega)$ iterazioni. L'algoritmo necessita solo di una piccola modifica, ma che porta la complessita' a cambiare radicalmente.
Nel caso $|P| > w$, la transizione $M[\cdot][j-1] \leftarrow M[\cdot][j]$ sara' $T_{U(j)}(n,m) = \Theta(\frac{m}{\omega})$, perche' il processo di transizione occupera' piu' operazioni della CPU. Quindi la complessita' dell'algoritmo nel caso generale e' $T(n,m) = O(n \cdot m)$.

2.4 Karp Rabin

Ragionamento L'idea e' quella di usare una funzione di hash $H(S)$ per confrontare il pattern P con il testo T.
Si itera con una finestra scorrevole W con $|W| = m$. Si calcola una sola volta $H(P)$, quindi si confronta $H(W)$ con $H(P)$ alla ricerca di occorrenze.

La funzione di hash usata in questo caso e':
$H(S) = \Sigma_{i=1}^{|S|} 2^{i-1} H(S[i])$

Il calcolo di $H(T[i+1 : i+m])$ puo' essere ottimizzato usando $H(T[i : i+m-1])$:
$H(T[i+1 : i+m]) = \frac{H(T[i:i+m-1]) - T[i]}{2} + 2^{m-1} T[i+m]$.

Algorithm Karp Rabin

```
procedure KB(T, P)
    H_P ← H(P)
    H_W ← H(T[: m])
    if H_P H_W then
        print(0)
    end if
    for i = 1 to n − m do
        H_W ← ((H_W − T[i])/2) + 2^{m−1} T[i + m]
        if H_P = H_W then
            print(i)
        end if
    end for
end procedure
```

Procedura

Complessita' Visto che l'aggiornamento di H_W costa $\Theta(1)$, la creazione del primo H_W costa $\Theta(m)$ ed e' presente solo un ciclo di $\Theta(n-m)$ iterazioni, la complessita' temporale e' $T(n,m) = \Theta(n+m)$.

Problemi Purtroppo i calcolatori hanno capacita' limitata, e, come nel caso dello Baeza-Yates-Gonnet, se $m > \omega$ allora la transizione di H_W viene a costare $\Theta(\frac{m}{\omega})$ e quindi l'algoritmo costera' $T(n,m) = O(n \cdot m)$.

Si puo' pensare di risolvere questo problema limitando la dimensione di H_W applicando un modulo p. Tuttavia questo introduce necessariamente la possibilita' di ottenere **Falsi Positivi** a causa delle collisioni che si possono creare con il modulo sbagliato.

Se descriviamo la probabilita' che un falso positivo si verifichi possiamo scrivere: $P(FP) = \frac{1}{2}$. Ad ogni modo e' possibile generalizzare questa variante provando in successione k moduli diversi per ottenere una probabilita' di errore di $P(FP) = \frac{1}{2^k}$. Dopo ogni falso positivo si sostituisce il modulo che ha generato la collisione. Per questo motivo questa variante del Karp Rabin rientra nella classe degli algoritmi probabilistici.

Algoritmi Probabilistici Esistono due class di algoritmi probabilistici.

Monte Carlo Puo' essere non corretto, ma e' comunque veloce ed efficiente. La vairiante del Karp Rabin rientra in questa categoria.

Las Vegas E' sicuramente corretto ma potenzialmente costoso. Il quicksort con pivot randomico rientra in questa categoria.

Chapter 3

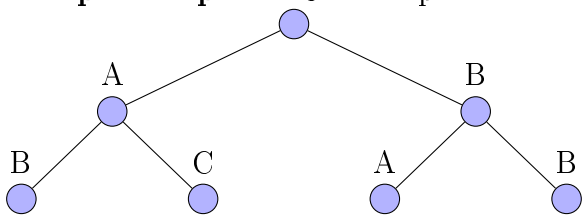
Suffix Tree e Suffix Array

3.1 Trie

Un **Trie** e' un albero ordinato in cui i nodi non mantengono una copia della loro chiave, ma essa e' rappresentata dal cammino radice-nodo. Infatti ogni arco tra nodi e' etichettato con una stringa.

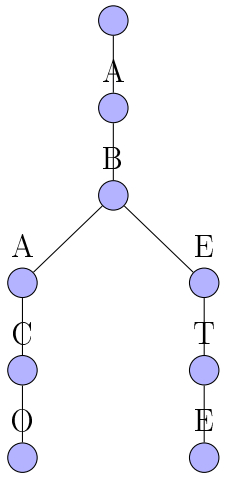
Esempi

esempio semplice Un esempio di Trie:



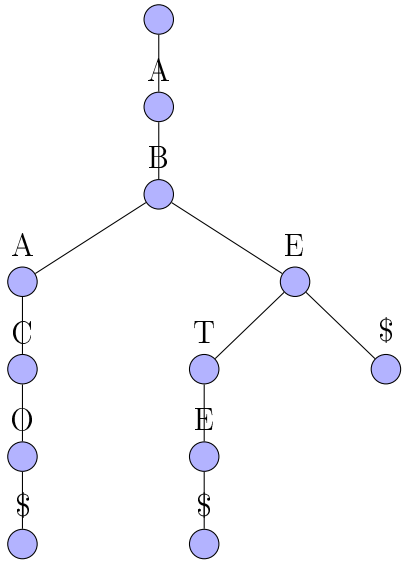
con dizionario Puo' essere generato anche a partire da un dizionario, per esempio:

$$\text{ABE} \tag{3.1}$$
$$\text{ABETE} \tag{3.2}$$
$$\text{ABACO} \tag{3.3}$$



con terminatori Facendo terminare ogni parola del dizionario con un terminatore $\$ \notin \Sigma$, e' possibile costruire un Trie basato su dizionario che indichi chiaramente quali siano le parole del dizionario. Infatti se il dizionario non gode della prefix-free property, ovvero se $\exists w, x \in \Sigma^*$ tali che $\exists s \in \Sigma^* \mid w \cdot s = x$, il Trie conterra' per x e w due foglie distinte che segnalano la fine delle due parole. Ovvero per ogni parola nel dizionario sara' presente la rispettiva e distinta foglia corrispondente al terminatore \$.

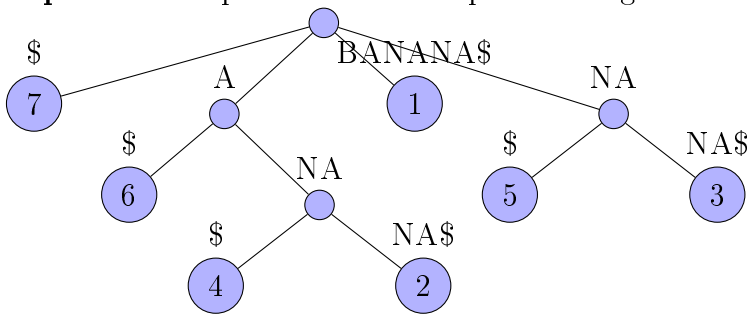
$$\text{ABE\$} \tag{3.4}$$
$$\text{ABETE\$} \tag{3.5}$$
$$\text{ABACO\$} \tag{3.6}$$



3.2 Suffix Tree

Un **Suffix Tree** e' un **Trie** che rappresenta un insieme di suffissi in cui tutti i nodi hanno archi che iniziano con simboli diversi e solo la radice puo' avere meno di due archi. Ogni nodo contiene come valore la posizione di inizio del suffisso rispetto al testo di partenza. Un Trie su una stringa S e' generato a partire dall'elenco di tutti i suffissi di S terminati in \$ (quindi un dizionario di lunghezza $|S|$). Questa struttura dati richiede spazio in memoria con una complessita' di $O(n^2)$.

Esempi Un esempio di Suffix Tree per la stringa BANANA\$:



L'esempio riprende quello dei lucidi del prof. Della Vedova.

3.3 Suffix Array

E' un array di tutti i suffissi di S in ordine lessicografico associati alle rispettive posizioni di inizio nel testo.

Esempi Riprendendo il Suffix Tree della sezione precedente e il rispettivo esempio:

i	1	2	3	4	5	6	7
SA	\$	A\$	ANA\$	ANANA\$	BANANA\$	NA\$	NANA\$

Possiamo quindi calcolare la Lunghezza del Prefisso Comune ai suffissi utilizzando la formula di ricorrenza:
$Lcp[i] = |CommonPrefix(SA[i], SA[i+1])|$

i	1	2	3	4	5	6	7
SA	\$	A\$	ANA\$	ANANA\$	BANANA\$	NA\$	NANA\$
LCP	0	1	3	0	0	2	–

3.4 Costruzione di Suffix Array

Un metodo naive potrebbe essere riassunto in tre fasi.

Creazione Si crea l'array raccogliendo tutti i suffissi di S, con una complessita' lineare: $T(n) = \Theta(n)$

Ordinamento Si puo' quindi riordinare i suffissi con un algoritmo a piacere, per esempio con un heap sort, il quale ha $T(n) = O(n \cdot logn)$. Siccome confrontare due stringhe ha $T(n) = O(n)$, il totale e': $T(n) = O(n^2 \cdot logn)$.

LCP Si tratta semplicemente di calcolare LCP per ogni due entry consecutive: $T(n) = \Theta(n)$

Totale La complessita' totale e':
$T(n) = \Theta(n) + O(n^2 \cdot logn) + \Theta(n)$
$T(n) = O(n^2 \cdot logn)$

3.5 Da Suffix Tree a Suffix Array

Per creare il Suffix Array e' necessaria una visita depth-first del Suffix Tree in cui i nodi fratelli sono visitati secondo l'ordine lessicografico delle stringhe dei relativi archi uscenti al nodo padre. Questo processo puo' essere usato per calcolare anche la Lunghezza del Prefisso Comune, modificando la formula usata nella sezione precedente:
$Lcp[i] = NodeDepth(LCA(i, i+1))$, dove LCA e' il Lowest Common Ancestor, trattato piu' avanti.

3.6 Da Suffix Array a Suffix Tree

Innanzitutto si deve disporre del Suffix Array con le informazioni sul LCP. L'algoritmo si divide in due fasi:

- Ricostruzione del Suffix Tree.
- Riempimento del Suffix Tree.

Si considerino le seguenti procedura:

Ricostruzione del Suffix Tree Sia dato un array LCP. Se $|LCP| = 0$ allora ritorno una foglia. Altrimenti creo un nodo e procedo.

Sia quindi $k \in LCP$ l'elemento piu' piccolo in LCP, tratto quell'elemento come un separatore e applico ricorsivamente la procedura sui sotto array.

Esempio:
Dato $LCP = [0, 1, 3, 0, 0, 2]$. Il valore piu' piccolo e' 0, quindi formo i sotto array $< [], [1, 3], [], [2] >$ e creo un sottoalbero che come figli i nodi creati dall'applicazione ricorsiva di questa procedura ai singoli sottoarray.

Riempimento del Suffix Tree Per riempire il ST e' sufficiente una visita depth-first in cui si assegnano i valori delle foglie con i valori di SA e i rispettivi archi.

3.7 Suffix Tree Generalizzato

Un **Suffix Tree Generalizzato** e' un **Suffix Tree** per un insieme di stringhe. Ovvero e' Suffix Tree di tutti i suffissi w che appartengono a una delle stringhe. I nodi saranno decoranti non piu' solo con la posizione nel testo ma con le posizioni nelle relative stringhe, ovvero un insieme di coppie (stringa, posizione).

Il modo piu' semplice per ottenerlo e' la concatenazione delle stringhe con terminali speciali distinti:
$S_{tot} = S_0 \cdot \$_0 \cdot S_1 \cdot \$_1 \cdot ...S_n \cdot \$_n$.

Come esempio, siano $S_0 = MONDIALE$ e $S_1 = CAMBIALE$. Quindi costruisco $S_{tot} = S_0 \cdot \$_0 \cdot S_1 \cdot \$_1$, ovvero $S_{tot} = MONDIALE \cdot \$_0 \cdot CAMBIALE \cdot \$_1$.

Quindi costruisco il Suffix Tree di S_{tot}. Posso usare il Suffix Tree Generalizzato per cercare la LCS di piu' stringhe al posto di usare un algoritmo banale che avrebbe complessita' temporale $T(S_0, S_1, ...S_n) = \Pi_{i=0}^{n} |S_i|$.

3.8 Pattern Matching con Suffix Array

Visto che il Suffix Array e' un array ordinato lessicograficamente di suffissi, il pattern matching si riduce ad una ricerca dicotomica dei suffissi che hanno come prefisso il pattern P. In particolare:

- Visto che tutti i suffissi del range considerato nella singola chiamata ricorsiva $SA[L, R]$ condividono lo stesso prefisso, si puo' evitare di confrontare $LCP(SA[L], SA[R])$ caratteri.

3.9 Longest Common Substring con Suffix Tree

Come sappiamo, le foglie di un Suffix Tree contengono le informazioni su a quali stringhe appartengono i suffissi descritti dalle foglie stesse.

Siano $Node_i$ l'i-esimo nodo in SuffixTree e $String_j$ la j-esima stringa da confrontare. Si puo' estendere la proprieta' sopra destritta $P(Node_i, String_j) \Leftrightarrow Leaf(Node_i) \wedge String_j \in Node_i$ a tutti i nodi, in questo modo: $P(Node_i, String_j) \Leftrightarrow \exists Node_k \in Node_i \mid P(Node_k, String_j)$.

Ebbene l'algoritmo consiste nel cercare il nodo interno $Node_i$ piu' profondo nel Suffix Tree tale che: $\forall String_j \mid P(Node_i, String_j)$.

3.10 Lowest Common Ancestor

Un algoritmo semplice potrebbe essere questo.

Detti A e B i due nodi di cui si deve calcolare LCA. Detti P_A e P_B i percorsi dalla radice R ai due nodi.

Si confrontano i percorsi fino a trovare dove divergono.

Trovare P_A e P_B impiega $T(n) = O(n)$. Confrontare P_A e P_B si usa $T(n) = O(n)$.

Quindi l'algoritmo ha una complessita' totale di $T(n) = O(n)$.

3.11 Range Minimum Query

TODO