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.