

Elementi di Bioinformatica

UniShare

Davide Cozzi
@dlcgold

Indice

| | | |
|----------|---|----------|
| 1 | Introduzione | 2 |
| 2 | Introduzione alla Bioinformatica | 3 |
| 2.1 | Bit-Parallel | 3 |
| 2.1.1 | Algoritmo Dömölki/Baeza-Yates | 4 |

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: <https://github.com/dlccgold/Appunti>.

Grazie mille e buono studio!

Capitolo 2

Introduzione alla Bioinformatica

Un po' di notazione per le stringhe:

- **simbolo:** $T[i]$
- **stringa:** $T[1]T[2]\dots T[n]$
- **sottostringa:** $T[i : j]$
- **prefisso:** $T[: j] = T[1 : j]$
- **suffisso:** $T[i :] = T[i : |T|]$
- **concatenazione:** $T_1 \cdot T_2 = T_1T_2$

In bioinformatica si lavora soprattutto con le stringhe, implementando algoritmi, per esempio, di pattern matching. Nel pattern matching si ha un testo T come input e un pattern P (solitamente di cardinalità minore all'input) da ricercare. Si cerca tutte le occorrenze di P in T . L'algoritmo banale prevede due cicli innestati e ha complessità $O(nm)$ con n lunghezza di T e m lunghezza di P . Il minimo di complessità sarebbe $O(n + m)$ (è il **lower bound**). Si ragiona anche sulla costante implicita della notazione O-Grande cercando di capire quale sia effettivamente l'algoritmo migliore con la quantità di dati che si deve usare. Bisogna quindi bilanciare pratica e teoria.

2.1 Bit-Parallel

È un algoritmo veloce in pratica ma poco performante a livello teorico, infatti ha complessità $O(nm)$.

```

for  $i = 1 \rightarrow n$  do
   $trovato \leftarrow true$ 
  for  $j = 1 \rightarrow m$  do
    if  $T[1 + j - 1] <> P[j]$  then
       $trovato \leftarrow false$ 
    end if
  end for
  if  $trovato$  then
     $print(i)$ 
  end if

```

end for questo algoritmo è facilmente eseguibile dall'hardware del pc.

In generale si hanno **algoritmi numerici** che trattano i numeri e gli **algoritmi simbolici** che manipolano testi.

Si hanno poi gli **algoritmi semi-numerici** che trattano i numeri secondo la loro rappresentazione binaria, manipolando quest'ultima con *or* \vee , *and* \wedge , *wedge*, *xor* \oplus , *left-shift* $<<$ e *right-shift* $>>$. Queste operazioni sono tutte bitwise e sono mappate direttamente sull'hardware, rendendo tutto estremamente efficiente.

2.1.1 Algoritmo Dömölki/Baeza-Yates

Si costruisce una matrice M dove:

$$M(i, j) = 1 \text{ sse } P[:j] = T[j - i + 1 : j], \quad 0 \leq i \leq m, \quad 0 \leq j \leq n$$

Questa matrice è veloce da costruire e si ha:

$$M(m, \cdot) = 1, \quad M(0, \cdot) = 1, \quad M(\cdot, 0) = 0$$

$$M(i, j) = 1 \text{ sse } M(i = 1, j = 1) \text{ AND } P[i] = T[j]$$

la prima riga saranno tutti 1 ($M(0, \cdot) = 1$) in quanto la stringa vuota c'è sempre mentre la prima colonna saranno tutti 0 ($M(\cdot, 0) = 0$) in quanto un testo vuoto non matcha mai con una stringa non vuota.

Quindi la matrice avrà 1 solo se i primi caratteri del pattern $P[i]$ sono uguali alla porzione di testo $= T[j - i + 1 : j]$. Ma in posizione $M(i - 1, j - 1)$ mi accorgo che ho 1 se ho un match anche con un carattere in meno di P e T . Quindi se $M(i - 1, j - 1) = 0$ lo sarà anche $M(i, j)$. Se invece

$M(i-1, j-1) = 1$ devo controllare solo il carattere $P[i]$ e $T[j]$ e vedere se $P[i] = T[j]$. La matrice la costruisco con due cicli e controllo solo l'ultima riga ma il guadagno non si ha a livello di complessità, in quanto sempre $O(nm)$, ma dall'architettura a 64bit della cpu. Con una word della cpu posso memorizzare una colonna intera, in quanto vista come numero binario. Ora lavoro in parallelo su più bit, con un algoritmo **bit-parallel**, facendo ogni volta 64 confronti tra binari. In questo modo crolla la costante moltiplicativa nell'O-grande.

Ma come passo da una colonna $C[j]$ a una $C[j-1]$? Con questi step:

- right shift di $C[j-1]$
- aggiungo 1 in prima posizione per compensare lo shift
- faccio l'AND con $U[T[j]]$, che è un array binario lungo come il pattern dove ho un binario con 1 se è il carattere di riferimento:

P=abca
 U[a]=1001
 U[b]=0100
 U[c]=0010

- ragiono sul word size ω

ottengo:

$$C[j] = ((C[j-1] \gg 1) | (1 \ll (\omega-1) \& U[T[j]]))$$

Quindi sapendo una colonna della matrice voglio calcolare la colonna seguente. Quindi $M[i, j] = M[i-1, j-1] \text{ AND } P[i] = T[j]$ quindi, per esempio, $M[1, j] = \text{TRUE AND } (p[i] = T[j])$ quindi conta solo il confronto dei caratteri. Quindi praticamente col lo shift spostato in basso di uno la colonna e faccio il confronto.

Quindi, ricapitolando, costruisco gli array U, itero controllando di avere l'1 in posizione più significativa e ottengo i match nell'ultima riga, in pratica la matrice non la stiamo costruendo, è solo un concetto teorico.

Però si ha il limite dei 64bit di lunghezza del pattern e l'uso di più word comporta il riporto sulla colonna seguente, fattore che si complica all'aumentare della lunghezza del pattern, soprattutto se arbitraria. Abbiamo il vantaggio che non abbiamo if/else.