Appunti di Bioinformatica

A.A. 2020-2021

Matteo Brunello

Indice

1 Introduzione

La disciplina della bioinformatica e' molto vasta, per cui e' possibile riassumerla in 3 grandi sottocategorie:

- **Biologia Computazionale**: che pone l'accento sugli aspetti algoritmici dei problemi biologici e sull'efficienza delle loro soluzioni. Principalmente riguarda lo studio algoritmi e tecniche che riguardano stringhe di sequenze di caratteri.
- **Biologia dei Sistemi**: approccio introdotto recentemente che si basa sull'utilizzo della teoria dei sistemi per studiare i fenomeni biologici. L'idea e' quella di sfruttare formalismi matematico-computazionali per modellare interazioni biologiche.
- **DNA Computing**: e' una forma di computazione che usa il DNA e la biologia molecolare al posto delle tecnologie computazionalii tradizionali (silico-based)

La biologia computazionale in particolare fa riferimento alla biologia molecolare, la branca della biologia che si occupa di studiare le interazioni e le attivita' biologiche a livello molecolare. Quando si parla di biologia molecolare e' impossibile non parlare del cosiddetto *dogma centrale della biologia molecolare*, che descrive il flusso principale di informazione genetica all'interno di un sistema biologico. In dettagli tale dogma e' descritto dai seguenti passaggi:

1. Tutta l'informazione di una cellula e' codificata nel **DNA**. L'informazione e' organizzata nei "geni" che sono delle sequenze particolari all'interno dello stesso
2. Nelle cellule eucariote, il DNA deve "uscire" dal nucleo della cellula, e per far cio' viene letto e tradotto in **mRNA**. L'RNA e' una piccola molecola in grado di passare attraverso degli appositi "pori" situati nella membrana nucleare [1]
3. L'RNA viene trasformato in **proteine** (cioe' il "*carburante*" che permette al meccanismo genetico di funzionare) attraverso un processo di sintesi
4. Le proteine determinano il **fenotipo**, cioe' le caratteristiche *morfologiche* e le *funzionalita'* di un determinato organismo
5. Il fenotipo si riflette all'interno di una popolazione di individui. Tramite dei meccanismi di incrocio (nella maggior parte degli esseri viventi e' l'accoppiamento) si ottengono nuovi individui con fenotipi differenti. In questa fase avviene una **selezione** dei fenotipi migliori
6. Il risultato della selezione dei fenotipi migliori determina l'**evoluzione** della specie
7. Il nuovo fenotipo determina un nuovo DNA, il ciclo si ripete.

In generale, e' possibile riassumere questi punti nei seguenti passaggi

$$DNA \overset{trascrizione}{\rightarrow} mRNA \overset{traduzione}{\rightarrow} Proteine$$

Idealmente, un genoma (DNA) codifica un mRNA che a sua volta codifica una proteina. Tale derivazione pero' non e' corretta dal momento che ogni genoma puo' codificare molte proteine (e quindi determina molte funzioni). Basti pensare ad esempio che il genoma umano che condividiamo con le altre persone ha similarita' per il 99,9% rispetto agli altri. Cio' significa che anche la piu' piccola variazione nella presenza di geni puo' causare una differenza fenotipica molto accentuata. Oltre a questo, e' valido anche il contrario, cioe' che esistono all'interno della sequenza genomica gli stessi geni che codificano proteine diverse. Principalmente, un DNA e' composto da 4 nucleotidi (Adenosina, Citosina, Guanina e Tinina), mentre una proteina che viene codificata dal DNA e' composta da amminoacidi. Gli amminoacidi sono al massimo 20. Ogni amminoacido e' codificato da una sequenza di 3 nucleotidi. Quindi, per calcolare quanti amminoacidi sono codificati da una tripla:

$$4 \cdot 4 \cdot 4 = 4^3 = 64$$

cio' significa che tramite una tripla possono essere codificati 64 amminoacidi, ma dal momento che sono solo 20, alcune sequenze sono duplicate, per cui lo stesso amminoacido e' codificato da diverse triple. E' evidente come sia fondamentale per una branca come la biologia molecolare trattare una grande mole di dati biologici dal punto di vista computazionale, ed e' qui che entra in gioco la biologia computazionale. La bioinformatica nasce essenzialmente negli anni 2000 in seguito ad un evento scientifico molto importante: il sequenziamento del genoma umano. Per la prima volta veniva raccolta una mole di dati mai raccolta prima. Questo evento mise luce alla mancanza di adeguatezza da parte delle tecniche e tecnologie informatiche (riguardanti la biologia computazionale) utilizzate fino a quel momento. Naque quindi la necessita' di sviluppare algoritmi e tecniche della biologia computazionale particolarmente sviluppate per trattare grandi quantita' di dati. Al giorno d'oggi e' addirittura fondamentale sviluppare tecnologie high-thoughput per il processamento di dati biologici. Una di questa ragione e' che molte delle tecnologie per il sequenziamento producono di norma una quantita' di dati molto grande. Un esempio di questo sono i microarrays, cioe' dei chip hardware specificatamente progettati per il sequenziamento parallelo di migliaia di geni contemporaneamente. (Piu' recentemente si utilizza il deep sequencing)

1.1 Introduzione alla biologia

1.1.1 Dai geni alle proteine

- Il dogma fondamentale della biologia molecolare negli anni e' diventato sempre meno un "dogma" dal momento che si sono scoperte alcune eccezioni e variazioni.
- Il DNA contiene tutto il nostro codice genetico. Esso determina ogni nostro fenotipo e ogni reazione che abbiamo a stimoli sia interni che esterni.

- Il DNA e' formato puo' essere rappresentato come una stringa formata da 4 caratteri che rappresentano i nucleotidi: **A**(*denina*),**T**(*inina*), **G**(*uanina*), **C**(*itosina*)
- Il DNA concettualmente *dirige* le operazioni principali della cellula (come ad esempio determina la risposta ad un determinato stimolo)
- Il DNA umano e' di circa 3Gb (*basi*). Di tutto questo genoma, la parte che effettivamente contiene informazioni operative e' una porzione molto piccola.
- La maggior parte del DNA non sono delle porzioni che codificano dei geni (e quindi proteine), ma sono delle parti che o vanno a regolare la codifica delle stesse proteine, oppure per la produzione di alcuni tipi di proteine che non vengono prodotte normalmente (storicamente veniva definito come *"junk"* DNA perche' si pensava fosse inutile)
- Dal momento che i geni codificati sono molto pochi, per ottenere una combinazione di molte piu' proteine possibili la cellula usa l'mRNA messaggero
- La trascrizione di una sequenza di n nucleotidi di DNA produce una sequenza di n nucleotidi di mRNA. (la trascrizione preserva la lunghezza)
- Le proteine sono codificate come una sequenza di amminoacidi. In tutto sono 20
- La traduzione *NON* preserva la lunghezza della sequenza

Il DNA: La doppia elica

- NOTA: Una sequenza di DNA e' contrassegnata dal carattere `5'` che ne indica la testa e dal carattere `3'` che ne indica invece la coda
- Il DNA e' composto da due filoni di nucleotidi connessi tra di loro
- Esiste una complementarieta' tra i nucleotidi che compongono il DNA. Tale complementarieta' e' derivata da una caratteristica intrinseca dei nucleotidi che lo compongono
- In generale l'Adenina si lega in modo complementare alla Tinina, mentre la Citosina si va a legare in modo complementare alla Guanina
- L'Adedina e la Tinina sono legate da 2 ponti a idrogeno, mentre la Guanina e la Citosina sono legate da 3 ponti a idrogeno. Questo implica che quando si deve fare un esperimento che presuppone una divisione delle due stringhe di DNA, se la quantita' di Citosina e Guanina e' piu' presente, bisognera' usare molta piu' energia per poterle dividere
- Il DNA e' in grado di replicare se stesso tramite un processo di replicazione in cui i due filoni di DNA si dividono per poi essere "letti" da una proteina apposita in grado di generare il nucleotide complementare
- Dal momento che il DNA e' una sequenza molto lunga che deve risiedere interamente dentro ad una porzione specifica (nelle cellule eucariote) detta nucleo, il DNA assume una forma piu' compatta detta cromosoma
- Il compattamento in cromosoma del DNA e' dato da una proteina a forma di racchetta chiamata *Histone*. Concettualmente, il DNA non fa altro che "*arrotolarsi*" su di questa proteina
- La sequenza genomica si arrotola piu' volte su di questa proteina, ottenendo infine il cromosoma

- E' importante notare che il DNA rimane sempre in questa forma compatta con l'eccezione della fase di replicazione. Inoltre durante il processo di trascrizione vengono scompattate solo le parti di DNA (i genomi) da trascrivere.

RNA:

- Esistono diverse forme di RNA, ognuna atta a svolgere una funzione specifica. Ad esempio, lo scopo dell'mRNA e' quelo di trasportare informazione genomica-protenomica.

- Contrariamente al DNA, l'mRNA e' costituito da un singolo filamento formato dagli stessi nucleotidi che formano il DNA, con la differenza che la Tinina e' sostituita all'Uracile
- Durante il processo di trascrizione, solo uno dei due filamenti di DNA viene letto. Non ci sono limitazioni sul quale, l'importante e' che entrambi vengano letti nel verso giusto (`5'-3'`)
- Il processo di trascrizione avviene per mezzo di una proteina chiamata *RNA-Polimerasi*. Quando il DNA entra all'interno dell'RNA polimerasi, il DNA viene diviso nei suoi due filamenti. Uno di questi filamenti e' il template della trascrizione, su sui si legheranno a loro volta i nucleotidi di cui e' composto l'mRNA man mano che il DNA passa all'interno dell'RNA polimerasi. Una volta uscito, il DNA si richiude da solo.
- Generalmente l'inizio di un gene e' contraddistinto dalla seguenza `ATG`
- Il promotore e' la zona a monte di un gene che contiene al suo interno delle sequenze (*Motif*) che hanno la funzione di regolazione
- Una di queste sequenze e' la cosiddetta `TATA` box. Essa e' una sequenza situata a 25 nucleotidi prima dell'inizio della sequenza del gene e serve essenzialmente a legarsi con una proteina (TBP-TFIIID) il cui scopo e' quello di "sedersi" sopra la struttura del DNA per scompattarla e funge da segnalatore per l'inizio della trascrizione. Successivamente altri fattori di trascrizione si assemblano nei pressi del promotore, compresa la proteina RNA-Poly. Infine, un fattore di trascrizione TFIIH, utilizza l'ATP per spezzare il DNA all'inizio del punto di trascrizione dando il via al processo stesso.
- Esistono principalmente 4 tipi di RNA:
  - mRNA: messaggero dell'informazione genetica
  - tRNA: codon-to-amino acid apecificity
  - rRNA: nucleo del ribosoma
  - snRNA: reazioni di splice
- Da un gene presente nel DNA si ottengono n proteine. Questo avviene per mezzo dell'mRNA messaggero, tramite un'operazione chiamata *splicing*. Come detto, un filamento di mRNA messaggero corrisponde ad un gene particolare nel DNA. Tutta l'informazione presente nell'mRNA, pero', non e' solo riguardante alla proteina da sintetizzare. Possiamo quindi definire 2 parti dell'mRNA:

  - Esoni: codificano effettivamente la proteina da produrre
  - Introni: sono porzioni che non contribuiscono all'informazione della proteina da produrre
    Ci sono due tipologie di splicing principali:

1. Splicing classico: che consiste nella rimoziione degli introni dalla sequenza. il risultato e' l'unione degli esoni.
2. Splicing alternativo: consiste nella rimozione di alcuni introni (anche tutti), e di alcuni esoni. Tramite lo splicing alternativo, quindi si possono ottenere tante combinazioni dalla stessa sequenza di mRNA (e di conseguenza diverse proteine)

- Splicing nel dettaglio:

1. Per effettuare lo splicing, all'inizio nei siti `5'-3'` dell'introne, ci sono dei Motif per il riconoscimento, su cui si andranno a legare delle proteine specifiche per la funzione chiamate *Spliceosomi* (contrassegnate con l'iniziale U). Il motif per il riconoscimento del sito `3'` e' un nucleotide di Adenina, ed e' spostato di x posizioni rispetto al sito `3'`.
2. Una volta che gli spliceosomi si sono legati ai motif di segnalazione, un ulteriore spliceosoma `U5` si leghera' agli spliceosomi di segnalazione avvicinandoli, formando un "cappio" di mRNA.
3. Tramite una reazione biochimica, i siti si dividono nei punti specifici, separando l'introne (a forma di cappio) e dando modo alla sequenza dei due entroni di ricongiungersi formando un solo entrone.

4. L'introne rimosso verra' poi degradato da delle proteine specifiche per la degradazione.

Traduzione:

- Quando si parla di traduzione non si parla piu' di una corrispondenza 1-1, ma quello che si ottiene e' una sequenza molto piu' corta di amminoacidi data una sequenza di RNA, poiche' 3 nucleotidi vengono tradotti in 1 amminoacido.
- La traduzione viene effettuata per mezzo dei ribosomi e delle molecole di tRNA. Esse hanno una forma a trifoglio capovolto in cui possono essere individuate 4 zone:
  - D-loop ("Foglia" sx)
  - T-loop ("Foglia" dx)
  - Anticodone ("Foglia" inferiore) - Complementare del codone
  - Codone (Gambo superiore) - Ha un amminoacido legato al sito `3'` Nota: Il tRNA e' il risultato da un processo di trascrizione e di successivo splicing, proprio come l'mRNA
- Quando l'mRNA e' pronto dopo l'operazione di splicing, entra nel ribosoma che guida la traduzione utilizzando 3 molecole di tRNA in modo progressivo. Ogni molecola di tRNA si lega all'mRNA attraverso il suo anticodone, per poi rilasciare il suo specifico amminoacido. L'operazione si

ripete e man mano gli amminoacidi di legano tra di loro a formare una catena. Una volta che il tRNA ha "rilasciato" il suo amminoacido viene rilasciato dal ribosoma
- Una nota interessante e' che nel codice genetico potrebbero esserci degli errori. Alcune proteine hanno il compito di rilevare e correggere questi errori ma cio' potrebbe non succedere a volte. Quindi, dal momento che con 3 nucleotidi si possono esprimere 64 amminoacidi, alcune sequenze sono ripetute in modo che anche se ci fosse un errore nell'ultimo nucleotide della tripletta, si sintetizzerebbe lo stesso amminoacido.

2 Algoritmi di Pattern Matching

2.1 Pattern Matching

- Stringa: Lista di caratteri contigui, l'i-esimo carattere si indica con $S(i)$
- Sottostringa: $S[i..j]$ - Lista dei caratteri contigui che vanno dalla posizione i di S alla posizione j
- Stringa vuota: $S[i..j]$ con $i > j$
- Prefisso: Sottostringa che inizia dal primo carattere di S
- Suffisso: Stottostring che inizia da un generico indice e termina nell'ultimo carattere di S
- Prefissi e suffissi si dicono propri (e viceversa non propri) quando la stringa vuota non e' considerata prefisso o suffisso
- Quando due caratteri sono uguali si dice che c'e' un *match*, altrimenti si dice *mismatch*
- Una sottosequenza e' un sottoinsieme dei caratteri della stringa NON contigui. Una sottostringa e' una sottosequenza, ma una sottosequenza non e' una sottostringa.

> **Definizione (Pattern Matching)** Data una string P chiamata *pattern* e una stringa piu' lunga T, trovare tutte le occorrenze del pattern P nel testo T.

- Una soluzione naive a questo problema potrebbe essere quella di prendere i caratteri di P con T e confrontarli da sinistra a destra, usando P come *"stencil"* e confrontandone carattere per carattere. Successivamente si sposta a destra di un carattere rispetto a T e il confronto ricomincia.
- Sia $|P| = n$ e $|T| = m$, la complessita' dell'algoritmo e' $O(nm)$.
- E' possibile migliorare la tecnica andando a spostare P di piu' di un carattere quando si presenta un mismatch, oppure riducendo i confronti saltando alcune parti del pattern dopo lo shift.
- Molti algoritmi di pattern matching funzionano in 2 fasi
  - Fase di preprocessing: estraggono delle informazioni sulla struttura del pattern e del testo
  - Fase di ricerca: utilizzano l'informazione ottenuta dalla prima fase per effettuare la ricerca
- Per poter prima introdurre algoritmi di preprocessing o di pattern matching, e' necessario prima introdurre alcuni concetti e notazione specifica.

- Con $Z_i(S)$ si indica la lunghezza della sottostringa piu' lunga di S che inizia nella posizione i e matcha un prefisso di S
  - es. $S = a\ a\ b\ c\ a\ a\ b\ x\ a\ a$
  - $Z_5(S) = 3$, $Z_6(S) = 1$, $Z_9(S) = 2$
- ZBox: intervallo che inizia in i e termina in $i + Z_i - 1$
- r_i indica il valore massimo di $j + Z_j - 1$ per tutti i $1 < j <= i$ tali che $Z_j > 0$
- l_i indica l'indice iniziale dello ZBox che termina in r_i
- Il tempo richiesto per calcolare tutti gli Z_i e' $O(|S|^2)$
- Esiste pero' un algoritmo che sfrutta le iterazioni precedente che ha complessita' $O(|S|)$, chiamato Z-Algoritm
- Per poter capire come funziona a linee generali, facciamo un esempio e supponiamo che $k = 121$, $r_{120} = 130$, $l_{120} = 100$. Si vuole calcolare Z_{121}.

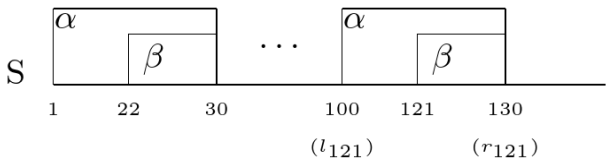


Figura 1: Raffigurazione schematica della situazione descritta nell'esempio

- Come si vede nella figura 1, per la definizione di Z-Box e secondo i dati dati dal problema, esiste una stringa α di lunghezza 30, all'inizio del testo e alla posizione 100. Allora la sottostringa β lunga 10 che inizia nella posizione 121, deve matchare la sottostringa β lunga 10 che inizia nella posizione 22 di *S*, percio' Z_{22} puo' essere utile per calcolare Z_{121}. Se Z_{22} e' per esempio 3, allora anche Z_{121} sara' 3.

2.2 Z-Algorithm

- Possiamo sfruttare questa caratteristica per sviluppare il cosiddetto *Z-Algorithm*:
  - All'inizio, Z_2 e' trovato confrontando carattere per carattere la sottostringa $S[1 \dots |S|]$ con la sottostringa $S[2 \dots |S|]$ fino a quando non si trova un mismatch, proprio come nel metodo Naive.

$$\overbrace{\underset{\uparrow}{a}bcdefg}^{S[1\dots|S|]}\dots$$

$$a\overbrace{\underset{\uparrow}{b}cdefg}^{S[2\dots|S|]}\dots$$

  - Se $Z_2 > 0$, $r = Z_2 + 1$ e $l = 2$, altrimenti $r = l = 0$ (*non esiste una Z-Box*)

- Procediamo induttivamente a calcolare i restanti Z. Dati gli Z_i con $2 <= i <= k-1$, $l = l_{k-1}$ e $r = r_{k-1}$, Z_k e i nuovi valori di $l = l_k$ e $r = r_k$ sono calcolati nel modo seguente:
  * Se $k > r$, vuol dire che la *Z-box* tra l ed r e' stata *"superata"*, per cui bisogna per forza utilizzare il metodo naive: Z_k viene calcolato confrontando i caratteri dalla posizione k con quelli che iniziano nella posizione 1 di S finche' non si trova un mismatch. Una volta trovato, se $Z_k > 0$, aggiorna i valori $r = k + Z_k - 1$ e $l = k$.
  * Altrimenti ($k \leq r$), denotiamo Z'_k la *Z-Box* corrispondente al prefisso, cioe' $k' = l-k+1$, la situazione e' quella raffigurata in figura 2.

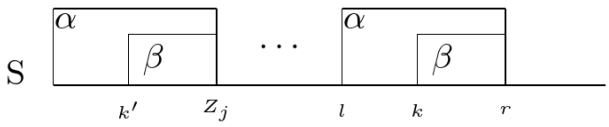
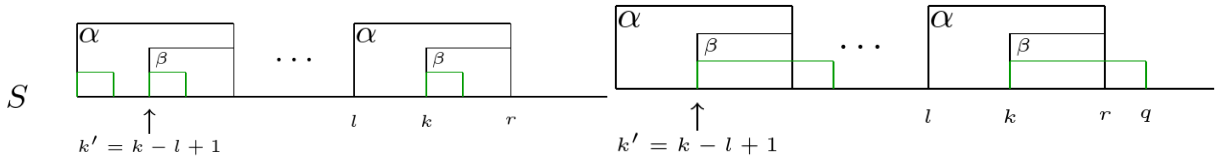


Figura 2: Situazione in cui $k \geq r$

Possiamo ora distinguere due casi differenti:

- Se $Z_{k'} < \beta$, allora imposta $Z_k = Z_{k'}$ e lascia l ed r invariati
- Se $Z_{k'} \geq \beta$, allora utilizza il metodo naive a partire dalla posizione $r+1$ e $|\beta|+1$ fino a quando non trovi un mismatch. Posto che il mismatch avvenga in una posizione $q \geq r+1$, allora setta $Z_k = q - k, r = q - 1$ ed $l = k$



(a) Situazione in cui $Z_{k'} < \beta$ **(b)** Situazione in cui $Z_{k'} < \beta$

- Lo *Z-Algorithm* puo' essere utilizzato per cercare un pattern nel testo andando a imporre che la stringa S sia la concatenazione di $S = P\gamma T$ dove γ e' un carattere che non occorre ne' in P ne' in T.
- Siccome γ non compare ne in T ne in S, allora per ogni $i < n + m + 1$, $Z_i <= n$, dove $n = |P|$, per cui basta contare il numero di volte in cui il valore di $Z_i = n$

2.3 Algoritmo di Boyer-Moore

- Come nell'algoritmo naive, l'algoritmo di Boyer-Moore allinea successivamente P con T e verifica il match carattere per carattere, ma differisce dal metodo naive per 3 caratteristiche.

  - La scansione del pattern avviene da destra a sinistra quando si confrontano i caratteri
  - Regola di shift del "*bad character*" (e conseguentemente la regola "*bad character estesa*")
  - Regola di shift del "*good suffix*"

- Queste idee portano a un metodo che di solito esamina meno di $m+n$ caratteri e ha complessita' in tempo lineare in caso peggiore, proprio come nello *Z-Algorithm*.

2.3.1 Bad Character Shift Rule

> *"Sposta il pattern quando c'e' un mismatch in modo da allineare il carattere in cui c'e' stato il mismatch nel testo con lo stesso carattere piu' a destra nel pattern"*

- Sia i il punto in cui in P si presenta il mismatch, mentre gli $n-i$ caratteri precedenti sono uguali ai corrispondenti in T (ne siamo sicuri dal momento che il confronto avviene da destra a sinistra)
- Sia $T(k)$ il carattere allineato con $P(i)$ nel testo T (il carattere in cui si presenta il mismatch nel testo).
- Sia $R(x)$ la posizione dell'occorrenza piu' a destra del carattere x in P ($R(x) = 0$ se x non occorre in P).
- Allora, sposta P a destra di N posizioni, con $N = max\{1, i - R(T(k))\}$. Se in P l'occorrenza piu' a destra di $T(k)$ e' in posizione $j < i$, sposta P in modo che il carattere j di P sia allineato con il carattere k di T, altrimenti sposta P di una posizione.
- La bad character shift rule, non ha effetto se il carattere $T(k)$ con in quale si trova il mismatch, si trova in P a destra del punto in cui e' stato trovato il mismatch (perche', come detto nel punto precedente, viene spostato di una sola posizione).

2.3.2 Bad Character Shift Rule Estesa

- Quando si presenta un mismatch nella posizione i di P e il carattere in T e' x, sposta P a destra in modo che il carattere x a sinistra piu' vicino alla posizione i in P sia allineato con x in T.
- Questa regola ha bisogno di una fase di preprocessing per calcolare le posizioni delle occorrenze piu' vicine a i a sinistra, la quale, per ogni posizione i in P e per ogni carattere x, calcola la posizione dell'occorrenza di x piu' vicina a i alla sua sinistra.
- Si puo' utilizzare un array bidimensionale $n \times |\Sigma|$ per memorizzare queste informazioni.
- Scandendo P da destra a sinistra si possono memorizzare per ogni carattere le posizioni in cui occorre. Ogni lista e' in ordine decrescente, e la loro costruzione richiede sia tempo che spazio $O(n)$.

2.3.3 Strong Good Suffix rule

- Sia t un suffisso che viene matchato sia in P che in T per cui il prossimo carattere fa mismatch. Trova allora un suffisso $t' = t$ in P per cui il carattere successivo non e' uguale a quello che ha fatto mismatch. Trovato il suffisso t', sposta P a destra in modo che la sotto stringa t' in P sia allineata con la sotto stringa t in T
- In caso t' non esista, allora sposta l'estremita' sinistra di P della minima quantita' in modo che un prefisso del pattern matchi un suffisso di t in T, se cio' e' possibile.
- Se cio' non e' possibile, sposta P di n posti a destra.
- Se si trova un'occorrenza di P, sposta P del minimo numero di posizioni in modo che un prefisso proprio di P matchi un suffisso dell'occorrenza di P in T, altrimenti sposta P di n posizioni, cioe' oltre t in T.

Esempio di Strong Suffix Rule:

```
T: PRSTABSTUBABQXRST
               ||
P:    QCABDABDAB
               ||
P':          QCABDABDAB
```

- Anche la good suffix rule ha bisogno di una fase di preprocessing:
  - Sia $L(i)$ la massima posizione minore di n tale che la stringa $P[i \dots n]$ matcha un suffisso di $P[1 \dots L(i)]$. $L(i) = 0$ se la condizione non e' soddisfatta
  - Sia $L'(i)$ la massima posizione minore di n tale che la stringa $P[i \dots n]$ matcha un suffisso di $P[1 \dots L'(i)]$ e il carattere che precede il suffisso non e' uguale a $P(i-1)$.
  - Sia $L(i)$ che $L'(i)$ possono essere calcolati in tempo lineare sulla lunghezza del pattern:
    * Sia $N_j(P)$ la lunghezza del piu' lungo *suffisso* della sotto stringa $P[1..j]$ (e' l'inverso di Z_i in sostanza)
    * Dal momento che N e' l'inverso di Z (cioe' vale $N_j(P) = Z_{n-j+1}(P^R)$) possiamo utilizzare lo *Z-algorithm* per calcolare tutti gli N_j, applicandolo sulla stringa riflessa P^R
  - *Z-Based Boyer Moore Preprocessing*

```
1  for i:=1 to n do L'(i) := 0
2  for j:=1 to n-1 do
3      i := n - N_j(P) + 1;
4      L'(i) := j;
```

- Per trattare nella fase di preprocessing anche i casi di $L'(i) = 0$ e occorrenza di P trovata, poniamo che $l'(i)$ sia la lunghezza del massimo suffisso di $P[i..n]$ che e' anche prefisso di P, e supponiamo che durante la ricerca si presenta un mismatch alla posizione $i-1$ di P:
  - Se $L'(i) > 0$, la good suffix rule sposta P a destra di $n - L'(i)$ posizioni in modo che il prefisso di P di lunghezza $L'(i)$ venga allineato al suffisso di P nella posizione precedente
  - Se $L'(i) = 0$ la good suffix rule sposta P di $n - l'(i)$ posizioni
  - Se si trova un'occorrenza di P, lo shift e' di $n - l'(2)$ posizioni
  - Se il risultato del primo confronto e' un mismatch, P viene spostato di una posizione

2.4 Algoritmo di Knuth-Morris-Pratt

- Viene raramente usato in pratica perche' in molti casi ha prestazioni inferiori rispetto a Boyer-Moore, ma costituisce la base di un noto algoritmo per cui vale la pena studiarlo
- Sia $sp_i(P)$ la lunghezza del piu' lungo suffisso proprio di $P[1..i]$ uguale ad un prefisso di P. $sp_1 = 0$ per ogni stringa
- Sia $sp'_i(P)$ la lunghezza del piu' lungo suffisso proprio di $P[1..i]$ uguale ad un prefisso di P, e tale che i caratteri $P(i+1)$ e $P(sp'_i + 1)$ siano diversi
- Per tutte le stringhe e per tutte le posizioni i, $sp'_i(P) <= sp_i(P)$
- Per ogni allineamento di P e T, se il primo mismatch si presenta in posizione $i+1$ di P, con la corrispondente posizione k in T, sposta P in modo che $P[1..sp'_i]$ sia allineato con $T[k - sp'_i..k-1]$. Cioe', di $i - sp'_i$ posizioni in modo che il carattere $sp'_i + 1$ di P sia allineato con il carattere k di T
- Se si trova un'occorrenza di P, sposta P di $n - sp'_n$ posizioni
- Anche in questo caso, gli sp'_i e gli sp_i possono essere calcolati in fase di preprocessing utilizzando una versione modificata dello *Z-algorithm*
- Z-Based Knuth Morris Pratt Preprocessing

```
for i:= 1 to n do
  sp'_i := 0
for j := n to 2 do
  i := j + Z_j(P) - 1
  sp'_i := Z_j
```

3 Motif Finding

- In generale, il problema del motif finding e' simile ad un problema di pattern matching, con la differenza che il pattern non e' conosciuto a priori

- Un'altra grande differenza e' che i *motifs* possono presentare delle mutazioni, per cui anche se si conoscesse il pattern (motifs) da cercare, impiegare algoritmi di pattern matching sarebbe inutile (perche' non si troverebbero tutti i motifs con almeno una mutazione)
- Piu' formalmente, possiamo dire che un (n, k) motif e' un pattern di lunghezza n che appare con k mismatches all'interno di una sequenza di DNA
- **Obiettivo**: trovare un pattern P (motif) all'interno di diverse sequenze di DNA, data la sua lunghezza e il numero di mutazioni ((n, k))
- **Problemi**: Possiamo conoscere solo le singole istanziazioni di P all'interno delle diverse sequenze, per cui e' difficile ritrovare il P originale!
- Considerando un motif $(15, 4)$, prendendo due istanze del pattern qualsiasi P_i, P_j, e' possibile che possano differire al piu' di $4 + 4 = 8$ mutazioni, per cui un approccio che consiste nel confrontare direttamente le sequenze tra loro non funzionerebbe.
- I motifs sono molto importanti perche' corrispondono ai cosiddetti TFBS (*Transcription Factors Binding Sites*), cioe' opportune sezioni a monte del gene che ne regolano la silenziazione, il potenziamento e la trascrizione (regolazione genica)
  - I transcription factors sono delle proteine che sono a loro volta codificate da altri geni che si legano in questi binding sites
  - Alcuni geni necessitano di particolari TF per indurre la trascrizione, per questo i TF sono anche chiamati *proteine regolatorie*
  - Ovviamente, i TFBS non sono tutti uguali, ma saranno diversi tra loro in base al gene da regolare (da qui la presenza di mutazioni)
- Per tener conto delle mutazioni piu' probabili, possiamo utilizzare un metodo grafico chiamato *Motif Logo*. In ogni posizione del motif vengono segnate le lettere piu' grandi o piccole in dimensioni in funzione della frequenza con cui appaiono in quella posizione.
- Usando i Motif Logos si possono vedere graficamente i punti del pattern che sono piu' frequentemente soggetti a mutazione
- In generale, l'identificazione di motifs presenta diverse sfide di non facile risoluzione:
  - Non si a priori la sequenza di caratteri che compone un motif
  - Non si sa dove il motif e' localizzato all'interno della zona promotoria del gene
  - Da un gene all'altro un motif puo' avere delle mutazioni
- Come possiamo distinguere un motif che effettivamente ha una funzione di regolazione da una sequenza casuale che non ha nessuna funzione di regolazione?

3.1 The Gold Bug's Problem

- Possiamo fare una piccola digressione su un problema di traduzione, in cui si possono utilizzare diversi metodi per la soluzione che possono essere utilizzati poi per la soluzione del motif finding.
- In questo caso, il problema in questione e' il problema posto da Edgar Allan Poe nel suo breve raccondo *The Gold Bug's Problem*.
- Il succo e' che nella storia c'e' un mittente che codifica un messaggio e lo manda ad un destinatario. L'idea e' che quando il destinatario riceve il messaggio, dovra' decifrarlo, cercando di capire cosa c'e' scritto dentro
- Ci sono pero' diverse assunzioni che si possono fare per poter decifrare il messaggio, che in qualche modo si legano al problema del motif finding nel DNA:
  - Il messaggio e' codificato in lingua Inglese
  - Ad ogni simbolo corrisponde una lettera dell'alfabeto Inglese
  - Nessuna lettera corrisponde a piu' di un simbolo
  - I segni di punteggiatura non sono codificati
- Un approccio molto semplice (quasi banale) e' il seguente:
  - Conta il numero di occorrenze dei simboli all'interno del messaggio.
  - Conta la frequenza dei simboli delle parole Inglesi in un vocabolario.
  - Ordina entrambe le liste di simboli e lettere in modo decrescente per frequenza, ottenendo quindi la mappa di traduzione simbolo -> lettera.
- Questo approccio si rivela pero' molto scadente (ne risulta un testo insensato)
- (*l-tuple count*) Un approccio alternativo potrebbe essere quello di calcolare le frequenze di coppie oppure terne di simboli al posto dei singoli caratteri.
- In questo modo si tiene conto dell'occorrenza di un simbolo dopo l'altro, per cui rappresenta un metodo piu' intelligente del precedente.
- Una volta trovata la tupla con la frequenza piu' alta, si vanno a sostituire tutte le occorrenze dei simboli nel testo. A questo punto si possono fare delle inferenze che man mano portano a decifrare l'intero testo.
  - Es. "t(ee" molto probabilmente significa "tree", per cui si puo' inferire che '(' = 'r'.
- Ovviamente per poter risolvere il problema sono necessari due prerequisiti:
  1. La frequenza delle tuple-terne di lettere di tutte le parole nel testo devono essere conosciute
  2. La frequenza di tutte le parole della lingua Inglese devono essere conosciute
- Ci sono diverse similarita' rispetto al problema del Motif Finding:
  - I nucleotidi nel motifs codificano un messaggio nel linguaggio "genetico" = simboli nel *The Golden Bug Problem* codificano un messaggio nella lingua Inglese

  - Per poter risolvere il probelma si puo' analizzare la frequenza dei pattern nelle sequenze nucleotidiche = per poter risolvere il problma si puo' analizzare la frequenza dei pattern nel testo codificato
  - E' necessaria una conoscenza di motifs regolatori gia' scoperti = e' necessaria la conoscenza delle parole in Inglese

- Il Motif Finding e' pero' piu' difficile per diverse ragioni:

  1. Non abbiamo un dizionario completo di tutti i Motifs esistenti
  2. Non abbiamo una grammatica standard per un linguaggio genetico, per cui non possiamo fare i passi di inferenza
  3. Sono una piccola porzione nelle sequenze decodifica il vero e proprio motifs (la quantita' di dati da elaborare e' enorme)

3.2 Definizione informale e primo approccio al problema

- Siccome il problema di motif finding e' molto difficile, consideriamone una versione rilassata, in cui si introduce una restrizione sulla lunghezza del pattern.
- Definizione dei parametri:

  - t numero di sequenze di DNA
  - n lunghezza di ogni sequeza di DNA
  - DNA insieme complessivo delle sequenze di dna (matrice bidimensionale $t \times n$)
  - l lunghezza del motif (*l-mer*)
  - $s = (s_1, s_2, s_3, \ldots, s_t)$ array delle posizioni iniziali dei motifs

- Sapendo la lunghezza del motif, allora possiamo individuare un motif all'interno della sequenza solo mediante l'indice iniziale, per cui $s = (s_1, s_2, s_3, \ldots, s_t)$ indica l'indice in cui inizia il motif all'interno della sequenza $1, 2, .., t$
- Supponiamo per semplicita' di sapere a priori s, allora possiamo allineare ogni motif all'interno delle sequenze e calcolare la cosiddetta matrice di profilo, che contiene la frequenza di ogni nucleotide per ogni colonna.
- E' possibile ricostruire la **stringa di consenso** utilizzando i nucleotidi con frequenza maggiore nella matrice di profilo. Tale stringa puo' essere considerata come una stringa "antenata" dalle quale sono emerse altre istanze mutate dello stesso motif
- Piu' formalmente, la consensus sequence e' una sequenza tale per cui il numero di mutazioni rispetto a qualsiasi altro motif e' sempre minore al numero di mutazioni tra tutte le possibili coppie di motifs.

- L'operazione descritta e' possibile ripeterla per qualsiasi insieme s di indici di sequenza, per cui e' utile avere un'euristica per determinare se una stringa di consenso decodifica effettivamente un motif oppure e' semplicemente una sequenza random
- Dal momento che la stringa di consenso e' determinata da s, e' possibile definire la funzione di scoring direttamente su s nel modo seguente:

$$score(s, DNA) = \sum_{i=1}^{l} \max_{k \in \{A,C,G,T\}} count(k, i)$$

- Dove:
  - $count(k, i)$ rappresenta la frequenza del nucleotide k nel motif s_i (e' l'entrata della matrice di profilo)
  - l e' la lunghezza del motif
  - s e' l'insieme di indici iniziali dei motif
- Lo score corrisponde ad estrarre la stringa di consenso e a sommarne le frequenze dei nucleotidi che la compongono nella matrice di profilo.
- Fino ad ora abbiamo supposto che le posizioni iniziali s siano conosciute a priori, ma nel caso reale non lo sono.
- Possiamo quindi sfruttare questa proprieta' per formalizzare il problema del motif finding come un problema di massimizzazione:
  - **Goal**: Dato un insieme di sequenze di DNA, trova un insieme di t *l-mers* per ogni sequenza, che massimizzino la funzione score
  - **Input**: Una matrice $t \times n$ di DNA e l, la lunghezza del pattern da trovare
  - **Output**: Un array $s = (s_1, s_2, \ldots, s_t)$ di t posizioni che indicano l'inizio di ogni motif nella sequenza *i-esima*, che massimizza la funzione di score

3.3 Metodo Brute Force

- Un approccio molto semplice potrebbe essere l'approccio brute force, computando ogni possibile combinazione delle posizioni iniziali s
- L'obiettivo e' quello di massimizzare $score(s, DNA)$ variando le posizioni iniziali s_i dove:

$$s_i = [1, \ldots, n - l + 1],\ i = [1, \ldots, t]$$

```
1 BruteForceMotifSearch(DNA, t, n, l)
2 bestScore <- 0
3     for each s = (s_1,s_2,...,s_t) from (1,...,1) to (n-l+1,...,n-l+1)
4         if(score(s, DNA) > bestScore)
```

```
            bestScore <- score(s, DNA)
            bestMotif <- s
return bestMotif
```

- Il problema di questo metodo e' che per ogni sequenza t combina $n-l+1$ posizioni, cio' significa che vengono considerati $(n-l+1)^t$. Per ogni set di posizioni, la funzione di score richiede l operazioni, per cui la complessita' dell'algoritmo e' di $O(ln^t)$.
- Nonostante l'algoritmo Brute Force (in quanto algoritmo esaustivo) ritorni la soluzione **ottima**, puo' essere applicato solo in istanze molto, molto piccole a causa della sua complessita' esponenziale nel numero di sequenze t.
- Per dare un'idea, se si scegliessero dei parametri tipici quali $t = 8, n = 1000, l = 10$, l'algoritmo richiederebbe 10^{20} computazioni, per cui richiederebbe miliardi di anni per terminare.

3.4 Problema della Stringa Mediana

- Possiamo migliorare l'algoritmo cambiando il punto di vista con cui guardiamo il problema
- Il problema della stringa mediana consiste nel trovare il pattern che compare in tutte le t sequenze di DNA con il numero minimo di mutazioni
- Al posto di provare variando le posizioni iniziali e provare a trovare una stringa consenso che rappresenti il motif, si possono variare invece tutti i possibili motifs direttamente
- Distanza di Hamming: Date due stringhe v e w, $d_H(v, w)$ e' il numero di coppie di nucleotidi che non fanno match quando v e w vengono allineate
- Per ogni sequenza di DNA i, computa ogni $d_H(v, x)$ dove x e' un *l-mer* con e v e' la sequenza di motif
- Una volta calcolate le corrispondenti distanze di Hamming, considera solo quella minima
- E' possibile poi calcolare la distanza totale, che e' la somma di tutte le distanze minime associate ad ogni sequenza
- E' possibile quindi formulare un algoritmo per trovare la stringa mediana:
  - **Goal**: Dato un insieme di sequenze di DNA, trova la stringa mediana
  - **Input**: Una matrice $n \times n$ di DNA, e l, la lunghezza della stringa
  - **Output**: Una stringa mediana, cioe' una stringa v di lunghezza l che minimizzi la distanza totale su tutte le stringhe di lunghezza l

```
MedianStringSearch(DNA, t, n, l)
bestWord <- AAA..AA
bestDistance <- INFINITY
    for each l-mer v from AAA..AA to TTT..TT
        if(TotalDistance(v, DNA) < bestDistance)
            bestDistance <- TotalDistance(v, DNA)
            bestWord <- v
```

```
8  return bestWord
```

- Il motif finding problem e' **equivalente** al problema della stringa mediana
- E' possibile dimostrare che *minimizzare* la distanza totale e' equivalente a *massimizzare* lo score:
  - Per ogni colonna della matrice di profilo i, vale la seguente relazione $Score_i + TotalDistance_i = t$
  - Siccome ci sono l colonne (tante quante la lunghezza del motif), vale anche $Score + TotalDistance = l * t \rightarrow Score = l * t - TotalDistance$
  - Dal momento che l e t sono costanti, minimizzare la distanza totale equivale a massimizzare lo score
- Il problema della stringa mediana deve esaminare solamente 4^t (il numero di nucleotidi possibili e' 4) ($|\Sigma|^t$) combinazioni di v, per cui e' leggermente piu' efficiente della soluzione brute force
- Per aumentare l'efficienza dell'algoritmo, si puo' pensare di ridurre lo spazio di ricerca, evitando di considerare alcune sequenze *l-mer*
- L'idea e' quella di rappresentare le sequenze non piu' come sequenze di caratteri ma come sequenze di numeri, applicando la seguente trasformazione: $A = 1, C = 2, G = 3, T = 4$
- In questo modo per passare alla prossima stringa da considerare basta aumentare di 1 il numero della sequenza
- Questo metodo rende piu' facile generare le nuove stringhe, ma non aumenta particolarmente l'efficienza dell'algoritmo

3.4.1 Metodi Branch and Bound

- Tramite questa numerazione, pero', e' possibile raggruppare tutte le sequenze in una struttura ad albero in base al prefisso delle sequenze
- Un percorso all'interno di questo albero aggiunge/rimuove lettere alla sequenza in caso si scendesse/salisse
- Per muoversi all'interno di quest'albero e' possibile definire 4 diverse operazioni:

1. Discendi verso la prossima foglia

```
1  NextLeaf(a, L, k)
2    for i <- L to 1
3      if a_i < k
4        a_i <- a_i + 1
5        return a
6       a_i <- 1
7    return a
```

   - k = 4 (lunghezza dell'alfabeto)

- a = array che contiene la sequenza (in cifre)
- L = lunghezza della sequenza

2. Visita tutte le foglie

```
AllLeaves(L, k)
  a <- (1, ..., 1)
  while forever
    output a
    a <- NextLeaf(a, L, k)
    if a == (1,...,1)
      return
```

3. Visita il prossimo vertice (incrementa l'*i-esima* cifra)

```
NextVertex(a, i, L, k)
  if i < L
    a_{i+1} <- 1
    return (a, i + 1)
  else
    for j <- L to 1
      if a_j < k
        a_j <- a_j + 1
        return (a, j)
  return (a, 0)
```

- i = lunghezza del prefisso, indica la profondita' attuale nell'albero

4. Bypassa il figlio di un nodo (trova il prossimo vertice saltando la generazione di tutti i figli)

```
Bypass(a, i, L, k)
  for j <- i to 1
    if a_j < k
      a_j <- a_j + 1
      return (a, j)
  return (a, 0)
```

- E' possibile utilizzare questi metodi per riscrivere brute force in modo che possa navigare lo spazio in modo piu' efficiente, semplicemente facendo le seguenti osservazioni:
  - Ogni riga nella matrice di profilo puo' aggiungere al massimo l allo score
  - Tutte le prossime $t-i$ posizioni $(s_{i+1}, \ldots, s_t)$ aggiungono al massimo $(t-i) \cdot l$ a `Score(s, i, DNA)`, per cui se $Score(s, i, DNA) + (t-i) \cdot l < BestScore$ non ha senso continuare a cercare nei vertici del sottoalbero corrispondente al prefisso corrente i
  - Per potare il sottoalbero, si utilizza `ByPass`
- Questo modo di operare e' detto **Branch&Bound**

```
BranchAndBoundMotifSearch(DNA, t, n, l)
  s <- (1,...,1)
```

```
3   bestScore <- 0
4   i <- 1
5   while i > 0
6     if i < t
7       optimisticScore <- Score(s, i, DNA) + (t - i) * l
8       if optimisticScore < bestScore
9         (s, i) <- ByPass(s, i, n-l+1)
10      else
11        (s, i) <- NextVertex(s, i, n-l+1)
12    else
13      if Score(s, DNA) > bestSCore
14        bestScore <- Score(s)
15        bestMotif <- (s_1, s_2, ..., s_t)
16    (s, i) <- NextVertex(s, i, t, n-l+1)
17  return bestMotif
```

- Dal momento che i due problemi sono equivalenti, e' possibile ragionare in maniera analoga anche per il problema della Stringa Mediana.
- Anche in questo caso, se la distanza totale di un prefisso e' piu' grande di quella migliore calcolata fino al punto attuale (`TotalDistance(prefix, DNA)> BestDistance`), allora non ha senso esplorare la restante parte della sequenza
- *(Algoritmo sulle Slides)*
- Gli algoritmi ottenuti con la tecnica di Branch&Bound hanno sempre complessita' esponoenziale in caso peggiore, pero' mediamente performano meglio rispetto alle controparti Brute Force
- E' possibile inserire ulteriori miglioramenti, ad esempio nel caso della stringa mediana:
  - Si puo' dividere l'*l-mer* in due parti nel punto i
    - u: prefisso
    - v: suffisso
  - Trova una distanza minima per u in una sequenza, per cui nessun'altra istanza avra' una distanza **minore** di quella trovata
  - Si ripete lo stesso per la sottostringa v
  - Essenzialmente, si possono utilizzare questi bounds in congiunzione con la migliore distanza per bypassare un numero maggiore di nodi (tutti quelli compresi tra i bounds) - distanza tra prefisso e suffisso e' maggiore di quella trovata fin'ora

4 Partial Digest Problem

- Il Partial Digest Problem consiste nel ricostruire la sequenza originale di DNA a partire da frammenti dello stesso

- **Molecular Scissors**: Sono molecole particolari che permettono di dividere il DNA nei punti in cui si lega
- Queste molecole si utilizzano per sequenzializzare le sequenze, siccome gli strumenti di sequenzializzazione funzionano solo su sequenze non troppo lunghe
- Gli enzimi che tagliano le sequenze per mezzo delle molecular scissor in sequenze piu' piccole sono chiamati **Restriction Enzymes**
  - Es. Hind II e' un RE che taglia la sequenza non appena incontra `GTGCAC` e `GTTAAC`
  - Le sequenze a cui si lega un Restriction Enzyme sono chiamate **Restriction Sites**
- Come detto, il Partial Digest Problem consiste nel ricostruire la sequenza originale a partire da tante restrizioni ottenute per mezzo di Restriction Enzymes
- In geneale, il PDP puo' essere riassunto con la seguente domanda:

> E' possibile ricostruire l'ordine dei frammenti ristretti solamente sapendo la loro lunghezza?

- Tramite un processo di **Gel Elecrophoresis** e' possibile misurare la lunghezza in maniera esatta dei frammenti
- **Partial Restriction Digest**: la sequenza di DNA viene esposta ad un tempo piu' limitato agli enzimi di restrizione in modo da evitare che il DNA venga tagliato su tutti i siti possibili, ma solo in un sottoinsieme di essi
- Il risultato della restrizione sono diversi frammenti, ma il problema viene complicato ulteriormente dal fatto che ci possono essere frammenti con lunghezze uguali
- **Multiset**: Insieme che permette di contenere duplicati. Contiene le lunghezze dei frammenti
- Indichiamo con:
  - X: L'insieme degli n interi che rappresentano i punti di taglio nel DNA (restriction sites)
  - n: il numero totale dei tagli
  - DX: il multiset di interi rappresentanti le lunghezze di ognuno dei $\binom{n}{2}$ frammenti prodotti dal partial digest
- Possiamo a questo punto dare una formulazione formale del problema:
  - **Input**: il multiset delle distanze a coppie L, contenente $n(n-1)/2$ interi
  - **Output**: Un set X di n interi tale che $DX = L$

> A partire da tutte le coppie di distanze tra i punti di una linea, ricostruisci le posizioni originali di tutti questi punti

- Nel problema del partial digest non e' sempre possibile determinare in modo **univoco** il set X basandosi solo su DX
  - Ad esempio, il set $X = \{0, 2, 5\}$ e $X = \{10, 12, 15\}$ hanno la stessa soluzione $DX = \{2, 3, 5\}$

4.1 Approccio Brute Force

- Approccio di ricerca esaustivo che ricerca ogni variante possibile per cercare una soluzione
- Come nel cazo del Motif Finding, sono generalmente non praticabili
- L'approccio consiste nei seguenti passi:

  1. Trova il frammento di restrizione di lunghezza massima M, con M la lunghezza della sequenza di DNA
  2. Per ogni possibile set $X = \{0, x_2, \ldots, x_{n-1}, M\}$ calcola il corrispondente DX
  3. Se DX e' uguale al partial digest sperimentale L, allora X e' la mappa di restrizione corretta

```
ButeForcePDP(L, n)
  M <- maximum element in L
  for every set of n-2 integers 0 < x_2 <...<x_{n-1} < M
    X <- {0, x_2, ..., x_{n-1}, M}
    Form DX from X
    if DX == L
      return X
    output "no solution"
```

- La complessita' di questo metodo e' pari a $O(M^{n-2})$ dal momento che devono essere esaminati tutti i possibili set di posizioni
- Siccome un grande numero di set generati non generano l'insieme L, un miglioramento all'algoritmo puo' essere fatto andando a limitare i valori di x_i ai soli valori che occorrono in L
- Questa e' una buona idea perche' in L ci sono le lunghezze, ma ovviamente anche le posizioni dovranno essere posizionate in quelle lunghezze

```
BetterButeForcePDP(L, n)
  M <- maximum element in L
  for every set of n-2 integers 0 < x_2 <...<x_{n-1} < M from L
    X <- {0, x_2, ..., x_{n-1}, M}
    Form DX from X
    if DX == L
      return X
    output "no solution"
```

- In questo algoritmo vengono esamimnati meno insiemi, ma la complessita' rimane esponenziale: $O(n^{2n-4})$

4.2 Approccio Branch&Bound

- Possiamo formulare anche una soluzione branch and bound, che anziche' costruire l'intero set ad ogni iterazione, lo si costruisce in maniera *incrementale* (cioe' andando a inserire posizione

per posizione)

- L'algoritmo funziona a grandi linee nel modo seguente:
  1. Inizializza il valore di $X = \{0\}$
  2. Prendiamo il valore piu' grande in L e inseriamolo in X
  3. Calcola le differenze delle distanze corrispondenti a X
  4. Se le differenze sono tutte contenute L, allora sappiamo che fino a questo punto X e' consistente. In questo caso, quindi, rimuovi tutte le distanze corrispondenti dall'insieme L
  5. Ripeti dal passo 2. fino a quando L e' vuoto
- Questo approccio e' pero' sbagliato! Perche' non si e' in grado di tornare indietro quando si rimuovono tutte le lunghezze da L nel passo 4. (non abbiamo l'altro branch dell'if)
- Per formulare l'algoritmo finale del partial digest e' necessario definire un insieme di supporto $D(y, X)$ come il multiset di tutte le distanze tra il punto y e tutti gli altri punti nell'insieme X
  - In altri termini, $D(y, X) = \{|y - x_1|, |y - x_2|, \ldots, |y - x_n|\}$
- Possiamo quindi formulare l'algoritmo branch and bound per il partial digest:

```
PartialDigest(L):
  width <- Maximum element in L
  Delete(width, L)
  X <- {0, width}
  Place(L, X)

// Backtracking
Place(L, X):
  if L is empty
    output X
    return
  y <- maximum element in L
  Delete(y, L)
  if D(y, X) proper subset of L
    Add y to X and remove lengths D(y, X) from L
    Place(L, X)
    Remove y from X and add lengths D(y, X) to L
  if D(width - y, X) proper subset of L
    Add width-y to X and remove lengths D(width-y, X) from L
    Place(L, X)
    Remove width-y from X and add lengths D(width-y, X) to L
  return
```

- Guardare l'esempio sulle slides
- La complessita' dell'algoritmo e' sempre esponenziale, ma e' molto veloce in media
- Informalmente, se $T(n)$ e' il tempo necessario al PartialDigest per inserire n tagli
  - Se esiste solo un'alternativa, allora $T(n) < T(n-1) + O(n)$

  - In caso contrario, allora e' esponenziale $T(n) < 2T(n-1) + O(n)$

5 Tecniche di Clustering

- Un cluster e' un insieme omogeneo di oggetti che sono tutti vicini tra loro secondo una relazione di distanza
- Il clustering puo' essere sia supervisionato che non supervisionato. Di solito, le tecniche di clustering piu' utilizzate sono di tipo non supervisionato
- Un buon clustering e' caratterizzato dalle seguenti caratteristiche:
  - I dati all'interno di un singolo cluster sono molto simili tra loro
  - I dati tra clusters differenti sono molto dissimili tra loro
- Ovviamente la qualita' di un clustering e' determinata:
  - Dalla misura con cui si definisce il concetto di similarita' (*distanza*)
  - Dal modello utilizzato (algoritmo di inferenza del modello)
- Le tecniche principali di clustering possono essere classificate in 5 approcci differenti:
  - *Algoritmi di Partizionamento*: costruiscono diverse partizioni e le valutano secondo determinati criteri. Ripetono la procedura fin quando la soluzione di clustering non soddisfa i criteri
  - *Clustering Gerarchico*: ritornano in output una descrizione dei cluster secondo una relazione gerarchica
  - *Density-Based*: formano i clusters in base alla densita' degli oggetti
  - *Grid-Based*: si basano su una struttura a diversi livelli di granularita'. Essenzialmente trattano i dati in basi a livelli livelli di dettaglio
  - *Model-Based*: derivano dall'ambito statistico. Inferisce delle distribuzioni di probabilita' che descrivono i clusters

5.1 Algoritmi di Partizionamento

> **Obiettivo**: Costruisci una partizione di un dataset D di n oggetti in un insieme k di clusters

- Dato k, trova una partizione di k clusters che ottimizzino un dato criterio di Partizionamento
- Siccome e' un algoritmo di ottimizzazione, possiamo differenziarli in metodi
  - **Esaustivi**: ritornano l'ottimo globale, ma hanno una complessita' elevata solitamente
  - **Euristici**: utilizzano un qualche tipo di euristica per ritornare l'ottimo locale, ma senza aver la garanzia che si tratti di un ottimo globale

- Alcuni algoritmi famosi (di tipo euristico) sono *k-means* e la sua variante *k-medoids*
  - *k-means*: i clusters sono rappresentati mediante i centroidi del cluster
  - *k-medoids*: i clusters sono rappresentati mediante i medoidi del cluster
- Come gia' detto, il clustering puo' essere visto come un problema di ottimizzazione, in cui bisogna massimizzare una funzione obiettivo. Tale funzione e' la funzione *errore*, definita nel modo seguente

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} \|p - m_i\|^2$$

  dove:
  - k e' il numero di clusters
  - m_i e' il medoide del cluster $i - esimo$
  - p e' un oggetto appartenente al cluster C_i
  - C_i e' il cluster *i-esimo*
- Si vogliono trovare delle soluzioni di cluster tali per cui le distanze di tutti gli elementi rispetto ai loro centri sono minori possibili

5.1.1 K-Means

- L'algoritmo *k-means* riesce a minimizzare questa funzione, trovando un ottimo locale, e lo fa mediante 4 passaggi:
  1. Partiziona gli oggetti in k clusters non vuoti
  2. Calcola i centroidi dei k clusters (il centroide e' l'oggetto che ha come valore degli attributi la media dei valori di ogni attributo)
  3. Assegna ad ogni oggetto al cluster corrispondente al centroide piu' vicino
  4. Se la soluzione di clustering non e' cambiata, termina, altrimenti ripeti dal passo 2.
- La complessita' di *k-means* e' pasi a $O(kn)$, per cui e' molto efficiente
- Il problema e' che e' possibile che non converga mai ad una soluzione, per cui e' opportuno prevedere un numero di iterazioni prefissato massimo
- Alcune limitazioni di *k-means* sono:
  - Difficile applicarlo quando le features non sono di tipo numerico
  - Bisogna per forza specificare in anticipo il numero di clusters k
  - Non e' in grado di gestire bene dati *rumorosi* e gli *outliers*
  - Non e' in grado di trovare soluzioni di clusters che hanno una forma *non convessa*

- Alcune varianti come *k-modes* servono ad ovviare ad alcuni di queste limitazioni. Esso difatti consiste nel calcolare la *moda* al posto della *media*.
- *k-prototype* utilizza invece dei metodi misti per calcolare i medoidi, applicando la moda o la media in base al fatto che le features siano categoriche oppure numeriche

5.2 Algoritmi Gerarchici

- Un algoritmo di clustering gerarchico puo' funzionare in due modi:
  - Agglomerativo: si parte da n clusters composti da un solo elemento e si fondono mano a mano fino ad ottenere la soluzione finale
  - Divisivo: si fa l'opposto, cioe' si parte da un solo cluster singolo e si va a suddividere iterativamente man mano
- La maggior parte degli algoritmi gerarchici sono di tipo agglomerativo, siccome sono di piu' facile implementazione
- La soluzione di questa tipologia di algorimi e' un albero gerarchico, chiamato *dendrogramma*, in cui i nodi sono connessi mediante (solitamente, ma non necessariamente) una relazione binaria di inclusione
- Alcuni di questi algoritmi sono `AGNES` (*Agglomerative Nesting*), `DIANA` (*Divisive Analysis*) e `CURE` (*Clustering Using REpresentatives*)

```
Hierarchical Clustering(d, n)
  form n clusters each with one element
  construct a graph T by assigning one vertex to each cluster
  while there is more than one cluster
    find two closest clusters C_1 and C_2
    merge C_1 and C_2 ito new cluster  C with |C_1| + |C_2| elements
    compute distance from C to all other clusters
    add a new vertex C to T and connect to vertic C_1 and C_2
    remove rows and columns of d corresponding to C_1 and C_2
    add a row and column to d corresponding to new cluster C
    return T
```

5.2.1 CURE

- Uno dei piu' interessanti e' proprio `CURE`, poiche' impiega dei meccanismi differenti per effettuare il Clustering
- Essenzialmente, fino ad ora, ogni algoritmo utilizzava un solo rappresentativo (es. medoide, centroide) per rappresentare il cluster.
- `CURE` utilizza invece una molteplicita' di rappresentativi per rappresentare i clusters

- Difatti, il drawback dei metodi basati sulla distanza, sono buoni solamente in casi in cui i clusters hanno una forma convessa, una dimensione relativamente piccola e se k puo' essere stimato in modo semplice
- L'algoritmo `CURE(s, p, q)` mira a sopperire a questi drawbacks. Il concetto dell'algoritmo e' il seguente:

  1. Estrai un cluster *"campione"* s in modo casuale
  2. Partiziona il campione in p partizioni con dimensione s/p
  3. Clusterizza parzialmente le partizioni in p/pq clusters
  4. Elimina gli outliers
  5. Raggruppa i clusters intermedi piu' vicini in un solo cluster

- Si noti che con l'operazione di raggruppamento un singolo cluster avra' piu' rappresentativi e non solo uno. In questo modo, le istanze rappresentative seguono la forma del cluster qualunque essa sia.
- In realta' l'algoritmo applica poi ulteriomente un'operazione di *shrinking* verso un centro di gravita' di una certa frazione α, su tutti i punti rappresentativi

5.3 Clustering basato sulla densita'

- Sono metodi che utilizzano come criterio per il clustering la densita' di punti (tipicamente all'interno di un determinato raggio)
- Questi algoritmi hanno la capacita' di rappresentare clusters di qualsiasi forma e riescono anche a gestire bene il rumore nei dati.
- Diversi famosi algoritmi quali:

  - `DBSCAN`
  - `OPTICS`
  - `DENCLUE`
  - `CLIQUE`

- Tutti questi algoritmi necessitano di solito di specificare un *raggio* per determinarne la terminazione

5.3.1 DBSCAN

- *(Density Based Spatian Clustering of Applications with Noise)*
- L'algoritmo utilizza due parametri principali:

  - ϵ: determina il raggio

  - $MinPts$: determina il numero minimo di punti (densita') che devono stare all'interno del cerchio di raggio ϵ

- *Density Reachable*: Se un punto p fa parte di uno dei punti $MinPts$ che stanno all'interno del raggio di un punto q
- *Density Connected*: Se un punto p e' connesso tramite un "percorso" di punti density reachable tra loro a un punto q
- L'algoritmo essenzialmente va a considerare punto per punto di tutto il dataset, andando a costruire man mano gli insiemi di densita'
- Una volta calcolati gli insiemi di densita', va a fondere tutti i punti che sono density connected in un singolo cluster
- **Punti di forza**:
  - Fa una sola scansione dei dati (complessita' lineare)
  - I cluster hanno dimensione e forma arbitraria
  - E' in grado di gestire gli outliers e i dati rumorosi
  - Non c'e' bisogno di assegnare un numero di cluster predefinito
- **Punti deboli**:
  - La qualita' di clustering dipende sull'ordine con cui si analizzano i dati
  - La qualita' di clustering dipende dalla definizione di vicinato che si da in input

5.4 Algoritmi Grid-Based

- Partizionano lo spazio andando a determinare degli intervalli sui valori degli attributi

5.4.1 CLIQUE

- (*Clustering In QUEst*) puo' essere considerato come una combinazione tra un algoritmo basato su densita' e uno basato sul concetto di griglia
- Utilizza un meccanismo per partizionare i **valori** degli attributi descrittivi, e non le istanze vere e proprie

1. Partiziona lo spazio dei dati e trova il numero di punti che ricadono all'interno di ogni cella della partizione
2. Identifica i sottospazi che contengono clusters utilizzando un pricipio *a-priori*:
   - Determina le unita' dense in tutti i sottospazi di interesse
   - Determina le unita' densamente connesse in tutti i sottospazi di interesse

- Guardare esempio sulle slides
- Punti di forza:
  - Determina automaticamente i sottospazi di piu' alta dimensionalita' tali per cui la densita' di punti e' alta
  - Non e' sensibile rispetto all'ordine con cui si considerano le istanze
  - Scala linearmente con la dimensione degli input e ha una buona scalabilita' all'aumentare il numero di dimensioni
- Punti deboli:
  - L'accuratezza della soluzione di clustering potrebbe degradare alle spese di un'implementazione del metodo piu' semplice

5.5 Clustering basato su Modelli

- L'idea e' quella di avere un qualche modello di come si distribuiscono i dati all'inteno del cluster. Essenzialmente utilizzano dele distribuzioni di probabilita' per determinare i clusters
- `SOM` (*Self Organizing Feature Map*) che e' basato sulle reti neurali
- Basati su una funzione di densita' di probabilita' quali `COWEB`. Questo algoritmo assume pero' ch ele distribuzioni dei valori degli attributi sono *indipendenti* tra di loro
- Un'altro esempio probabilistico e' un metodo che si chiama *Gaussian Mixture Model*, che rappresenta in un certo senso la variante probabilistica di *k-means*.
  - Si rappresentano i dati come distribuzioni Gaussiane
  - Si stimano i parametri iniziali delle Gaussiane
  - Si assegnano i punti al cluster in base alla probabilita' piu' alta
  - Si vanno a ri-stimare i parametri delle Gaussiane utilizzando i punti di appartenenza ottenuti precedentemente

5.6 Clique Clustering

- Un'altro modo per far clustering e' quello che si basa sulla rappresentazione di clusters con i *cliques*
- Un **clique graph** e' un grafo in cui ogni vertice e' connesso a qualsiasi altro vertice
- E' possibile trasformare la matrice di distanze in un grafo. Ogni riga e' un vertice del grafo, per cui ogni vertice viene connesso ad un altro se la loro distanza e' minore di un threshold definito
- Il grafo risultante puo' contenere dei *cliques*, per cui per il modo in cui si e' costruito il grafo, i cliques rappresentano clusters composti da punti vicini tra loro

- Trasformando il grafo delle distanze in un clique graph, si possono ottenere i diversi clusters dei dati. Per trasformarlo, si recidono o aggiungono connessioni al grafo. Il risultato possono essere piu' clique graphs.
- Il problema di trovare una soluzione chiusa al problema di trovare i clique graphs e' NP-Completo, per cui si utilizzano delle opportune euristiche per ricercare lo spazio delle soluzioni

5.6.1 CAST

- (*Cluster Affinity Search Technique*) e' un algoritmo che si basa sul clique Clustering
- $d(i, C) = $ distanza media tra l'istanza i e tutte le altre istanze appartenenti al cluster C
- L'istanza i e' vicina al cluster C se $d(i, C) < \theta$, dove θ e' il threshold
- Utilizzando questa nozione di distanza, l'algoritmo prende in input il grafo e va a iterare su di esso andando a suddividerlo in sottografi

```
Cast(S, G, Theta)
  P <- Empty Set
  while S is not empty
    V <- vertex of maximal degree in the distance graph G
    C <- {V}
    while a close istance i not int C or distant gene i in C exists
      Find the nearest close istance i not in C and add it to C
      remove the farthest distant istance i in C
    Add cluster C to partition P
    S <- S \ C
    Remove vertices of cluster C from the distance graph G
return P
```

5.7 Clustering per dati Biologici

- Il clustering e' un task che viene applicato molto in ambito biologico
- Esso e' utile sopratutto per *visualizzare* e *analizzare* una quantita' di dati biologici enorme
- Un esempio tipico e' l'analisi della funzionalita' dei geni, cioe' il task che ha l'obiettivo di capire quale funzione ha un gene appena sequenzializzato. Tipicamente consiste nel confrontare i geni sconosciuti con geni la cui funzionalita' e' gia' nota
- Per il 40% dei geni sequenzializzati, pero', comparare le sequenze non e' sufficiente a determinarne la funzionalita'
- Gli sperimenti con microarrays servono appunto a inferire la funzionalita' di un gene, anche quando la similarita' non e' sufficiente a determinarne la funzione
  - I microarray misurano l'attivita' espressiva dei geni sotto diverse condizioni

  - Il livello di espressione e' *stimato* misurando la quantita' di mRNA che un certo gene produce (piu' il gene e' attivo, piu' trascrizione di quel gene ci sara')

- Negli esperimenti a microarray, l'output e' essenzialmente una matrice di dati in cui sono indicati i diversi livelli di expressivita' dei geni scelti per l'esperimento. Ogni riga e' un gene differente, e ogni colonna indica il tempo di esposizione. Il valore e' l'espressione genica per quel lasso di tempo.
  - Ogni osservazione e' un punto nello spazio N-dimensionale
  - Calcolando la distanza tra ogni punto, i punti piu' vicini tra loro possono rappresentare geni che hanno le stesse caratteristiche espressive e possono quindi essere funzionalmente correlati tra loro
  - Attraverso il clustering, ad esempio, e' possibile trovare gruppi di geni che sono funzionalmente correlati tra di loro
- Di solito, gli algoritmi utilizzati su risultati degli esperimenti con microarrays sono quelli di tipo gerarchico, siccome si vuole fissare a posteriori il numero di clusters
- Il clustering gerarchico viene anche utilizzato per fare la ricostruzione evolutiva delle specie

5.7.1 Valutazione e Interpretazione dei Risultati

- Quando si fanno esperimenti di microarrays si rilevano le espressioni geniche di diversi geni
- Gli algoritmi di clustering applicati a risultati di esperimenti microarrays sono quindi dei *raggruppamenti di geni*.
- Come detto, molto spesso si utilizzano algoritmi di clustering gerarchico, cosi' che si possa "tagliare" il dendrogramma e ottenere cosi' un numero variabile di clusters a posteriori
- Dal momento che si possono utilizzare diversi algoritmi di clustering, nasce spontanea la domanda di come si possa effettivamente valutare la bonta' di una soluzione di clustering
- In generale, formulare una metrica di clustering necessita di fare delle assunzioni su cosa rappresentino i dati

> I Geni con dei patterns di espressione molto simili sono quelli che partecipano allo stesso processo biologico

- Una direzione possibile, e' quindi quella di sfruttare un'ontologia dei geni per poter formulare una metrica di bonta'
- La *Gene Ontology* e' un'ontologia che correla la conoscenza dei geni, rappresentandolo in termini di
  - **Processo Biologico** a cui prende parte

  - **Funzione Molecolare** che rappresenta il gene (se codifica un enzima, una proteina, ecc..)
  - **Componente Cellulare** dove avviene il processo

- Un modo possibile di procedere utilizzando la **GO**, e' quello di etichettare ogni elemento dei clusters con la sua rappresentazione corrispondente nell'ontologia. Se la maggior parte degli elementi di un cluster prendondono parte allo stesso processo biologico, allora e' probabile che la soluzione di clustering sia buona
- In questo modo si ottiene una valutazione in termini di *coerenza*, ma non e' sufficiente per determinare la bonta' dei clusters
- Essenzialmente, si utilizza il *p-value* come metrica addizionale
- Si puo' calcolare la probabilita' di avere x di n possibili geni che hanno la stessa annotazione nella Gene Ontology, sapendo inoltre che nel genoma, M di N geni hanno quell'annotazione. Tale probabilita' e' modellata da una distribuzione ipergeometrica:

$$p = \frac{\binom{M}{x}\binom{N-M}{n-x}}{\binom{N}{n}}$$

- In sostanza, abbiamo un cluster di n geni, x di questi hanno una determinata annotazione nella GO. Si va quindi a confrontare questo valore rispetto all'insieme totale dei geni nel genoma, e rispetto al numero totale dei geni nel genoma che hanno quella specifica annotazione
- Quindi, se il numero di geni che sono stati clusterizzati nello stesso cluster e' simile al numero di geni totale che hanno la stessa annotazione, allora p sara' molto alta
- E' possibile migliorare questa stima, considerando la probabilita' di avere almeno x su n annotazioni

$$\text{p-val} = 1 - \sum_{i=0}^{x-1} \frac{\binom{M}{i}\binom{N-M}{n-i}}{\binom{N}{i}}$$

- Questa formula, ritorna la probabilita' che l'insieme di geni clusterizzati all'interno di un cluster, abbiano un particolare insieme di annotazioni rispetto a un insieme totale dei geni che hanno la stessa annotazione
- Questo valore ci permette di calcolare la significativita' dei geni clusterizzati rispetto alla *Gene Ontology*

5.8 Co-Clustering

- Per giustificare lo studio del co-clustering, si consideri un esempio

- Supponiamo di essere nel caso in cui due geni abbiano inizialmente un'espressione molto simile. Dopo un determinato lasso di tempo, l'espressione degli stessi diventa molto diversa.
- Un algoritmo di clustering, clusterizzerebbe lo stesso questi geni nello stesso cluster, a causa della somiglianza dei valori iniziali
- Questo perche' gli algoritmi visti considerano solo i valori delle *righe* (e quindi dei geni)
- Gli algoritmi di co-clustering determinano invece i clusters per righe e colonne, anziche' solamente per righe
- L'idea e' quindi quella di fare clustering simultaneamente sia per righe che per colonne
- In questo senso, determinano un sottoinsieme di *geni* che condividono gli stessi profili di esperessione in un sottoinsieme di *condizioni* (nel caso visto, sono e' un lasso di tempo)
- ***Es.*** *G1, G2, G4 condividono gli stessi profili di espressione nell'esposizione nel lasso di tempo T2, T3*
- Questo tipo di clusters e' piu' preferito e gestibile dai biologi

5.8.1 Algoritmo di Co-Clustering Generico

- Un algoritmo di co-custering famoso e' il *Generic Co-clustering Algorithm* (Banerjee et al. 2007)
- L'algoritmo ha i seguenti parametri e ouputs:
  - **Input**: Matrice X di dati, k numero di suddivisioni nelle righe, l numero di suddivisioni nelle colonne
  - **Output**: Partizioni C_{cols}, C_{rows}
- Algoritmo:

```
Repeat until convergence (clusters don't change):
    Compute the approximation matrix w.r.t C_cols and C_rows
    Keep $C_{cols}$ fixed and find a better $C_{rows}$
    Keep $C_{rows}$ fixed and find a better $C_{cols}$
```

- Essenzialmente, consiste nel creare un cluster e provare a ottimizzarlo con un algoritmo di ottimizzazione sia sulle righe che sulle colonne
- Un buon algoritmo di co-clustering dovrebbe essere inoltre in grado di sfruttare della conoscenza sul dominio di applicazione. Facendo un esempio pratico:
  - Sappiamo che il gene `XYZ` e il gene `ABC` agiscono spesso insieme in condizioni cancerogene
  - L'algoritmo di co-clustering non deve inserire `XYZ` e `ABC` in due cluster differenti
  - Analogamente, se `XYZ` e `ABC` non agiiscono insieme, non devono essere clusterizzati nello stesso cluster
- Piu' formalmente possiamo definire questi vincoli come:

  - **Must-Link** ($c_{=}(i,j)$) e' un vincolo che indica due le due righe (o colonne) i e j devono essere clusterizzate insieme
  - **Cannot-Link** ($c_{\neq}(i,j)$) indica che le due righe (o colonne) i e j non devono essere clusterizzate insieme

- E' possibile integrare questi due vincoli, per cui il risultato dell'algoritmo finale sara' il seguente:
  - **Input**: Matrice X di dati, k numero di suddivisioni nelle righe, l numero di suddivisioni nelle colonne, insieme di righe e colonne cannot link C_r, C_c, insieme di righe e colonne must link M_r, M_c
  - **Output**: Partizioni C_{cols}, C_{rows}

- Algoritmo:

```
Initialize C_cols and C_rows
Repeat until convergence (clusters don't change):
    Compute the approximation matrix w.r.t C_cols and C_rows
    Foreach column j do:
        If exists M in M_c such that j belongs to M then:
            Assign all columns in M to the closest column cluster such
               that no
            cannot-link is violated
        else
            Assign j to the closest column cluster such that no cannot-
               link is
            violated
    Define C_cols w.r.t the resulting assignment
    Foreach row i do:
        If exists M in M_r such that i belongs to M then:
            Assign all rows in M to the closest row cluster such that
               no
            cannot-link is violated
        else
            Assign i to the closest row cluster such that no cannot-
               link is
            violated
    Define C_rows w.r.t the resulting assignment
```

- Le metriche per la valutazione di co-clusters sono due:
  1. **Hartigan**'s Residue: $h_{ij} = x_{ij} - \mathbb{E}(I,J)$ (scarto rispetto alla media tra riga I e colonna J)
  2. **Cheng&Church**'s Residue: $h_{ij} = x_{ij} - \mathbb{E}(I) - \mathbb{E}(J) + \mathbb{E}(I,J)$ (scarto sulla media discriminata di colonne e righe)

Bibliografia

[1] «Central Dogma of Molecular Biology». mar. 06, 2021, [Online]. Disponibile su: https://bio.libretexts.org/@go/page/6508.