# Elementi di Bioinformatica

UniShare

Davide Cozzi
@dlcgold

# Indice

# 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/dlcgold/Appunti`.
**Grazie mille e buono studio!**

# Capitolo 2

# Introduzione alla Bioinformatica

Un po' di notazione per le stringhe:

- **simbolo:** $T[i]$
- **stringa:** $T[1]T[2][n]$
- **sottostringa:** $T[i:j]$
- **prefisso:** $T[:j] = T[1:j]$ (inclusi gli estremi)
- **suffisso:** $T[i:] = T[i:|T|]$ (inclusi gli estremi)
- **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 maching 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, ha complessità $O(nm)$.

```
for i = 1 → n do
    trovato ← true
    for j = 1 → m do
        if T[1 + j − 1] <> P[j] then
            trovato ← 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*, *xor* $\oplus$, *left-shift* $<<$ *e right-shift* $>>$. Ricordiamo che il left shift sposta di $k$ posizioni a sinistra i bit, scartandone $k$ in testa e aggiumgendo altrettanti zeri in coda (lo shift a destra sposta a destra, scarta in coda e aggiunge zeri in testa). Queste sono operazioni bitwise e sono mappate direttamente sull'hardware, rendendo tutto estremamente efficiente.

### 2.1.1 Algoritmo Dömölki/Baeza-Yates

Questo algoritmo viene anche chiamato **algoritmo shift-and** o anche **bit parallel string matching**.
Si definisce in input una stringa $T$ di cardinalità $n$ e un pattern $P$ di cardinalità $m$.
Si costruisce una matrice $M$ *ipotetica*, di dimensione $n \times m$, con un indice $i$ per $P$ e uno $j$ per $T$ dove:

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

Quindi $M(i,j) = 1$ sse i primi $i$ caratteri del pattern sono uguali alla sottostring lunga $i$ in posizione $j - i + 1$ del testo.
Questa matrice è veloce da costruire e si ha:

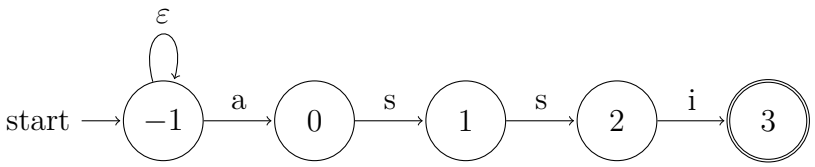
$$M(m,\cdot) = 1,\; M(0,\cdot) = 1,\; M(\cdot,0) = 0$$

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

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. Qindi 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 $P[i] = T[j]$. Ovvero, avendo $P = assi$ e $T = apassi$ si avrebbe (omettendo la prima riga e la prima colonna in quanto banali):

| | j | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| i | | a | p | a | s | s | i |
| 1 | a | 1 | 0 | 1 | 0 | 0 | 0 |
| 2 | s | 0 | 0 | 0 | 1 | 0 | 0 |
| 3 | s | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | i | 0 | 0 | 0 | 0 | 0 | 1 |

Con un automa non deterministico che accetta una stringa terminante con $P$ sarebbe:




La matrice la costruisco con due cicli e controllo solo l'ultima riga. Non si ha un guadagno a livello di complessità, dato che rimane $O(nm)$, ma grazie all'architettura a 64 bit della cpu. Infatti 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:

- la colonna $C[j]$ corrisponde al right shift della colonna $C[j-1]$
- 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 $\omega$ in caso di pattern più grandi di 64bit.

ottengo:

$$\text{C[j]} = ((\text{C[j-1]})\text{»}1)| \ (1 \text{ « } (\omega\text{-}1)\&\text{U[T[j]]})$$

Conoscendo una colonna della matrice voglio calcolare la successiva. Quindi $M[i,j] = M[i-1, j-1] \ AND \ P[i] = T[j]$ (per esempio, $M[1,j] = TRUE \ AND \ (p[i] = T[j])$), cioè conta solo il confronto dei caratteri.
Ogni 1 nell'ultima riga corrisponde ad un'occorrenza.
Questo algoritmo ha il vantaggio di non avere branch if/else, però si ha ul limite nella lunghezza del pattern (64 bit) 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.

## 2.2 Algoritmo Karp-Rabin

Vediamo un altro algoritmo di pattern matching che sfrutta una codifica binaria e che, pur non risultando sempre corretto, è estremamente più veloce, viene infatti eseguito in tempo lineare.
Uso un alfabeto binario e devo fare il match di due stringhe con ciascuna la sua codifica $H(S) = \sum_{i=1}^{|S|} 2^{i-1} H(S[i])$. Ad ogni carattere di una string si associa un numero nel range $[0, 2^{m-1}]$. Praticamente si usano due funzioni hash che trasformano una stringa in un decimale rappresentate in binario (ogni numero intero è facilmnete rappresentabile come somma di potenze di 2 e quindi in binario). Viene quindi facile paragonare le due fingerprints. Mi muovo sul testo $T$ mediante finestre di ampiezza $m$ pari a quella del pattern e controllo il fingerprint di quella porzione con quella del pattern. Inoltre il fingerprint di una finestra è facilmente calcolabile da quello della precedente. Per farlo elimino il contributo del carattere della finestra precedente e includo l'unico aggiunto dalla finestra successiva, in quanto mi sposto di 1:

$$H(T[i+1 : i+m]) = \frac{H(T[i : i+m-1])T[i]}{2} + 2^{m-1} T[i+m]$$

Essendo il primo carattere quello meno pesante viene rimosso ad ogni spostamento sfruttando la divisione per due per lo shift

La sottostringa è uguale al pattern solo se le fingerprint lo sono:

$$T[i : i + m - 1] = P \Leftrightarrow H(T[i : i + m - 1]) = H(P)$$

*Per estendere la codifica binaria in k caratteri avrò la finestra che si sposta di k con la divisione per k anziché per 2.*
Si ha il problema della lunghezza del pattern in quanto ho un $2^{m-1}$ che fa esplodere l'algoritmo perché usa un numero di bit grandissimo. Si ricorda che un'operazione "costa 1" solo se sono piccoli i numeri in gioco, nel nostro caso il costo diventa proporzionale al numero di bit coinvolti. La soluzione di Karp-Rabin è di continuare con la logica di sopra ma solo con numeri piccoli, cambiando la definizione di fingerprint prendendo il resto di quanto sopra con un numero primo $p$:

$$H(T[i + 1 : i + m]) = \left( \frac{H(T[i : i + m - 1])T[i]}{2} + 2^{m-1}T[i + m] \right) \mod p$$

ma in questo modo la fingerprint non è più iniettiva, con la possibilità che più stringhe abbiano la stessa fingerprint e di conseguenza si avranno degli errori. Si ha che $2^{m-1}T[i+m]$ viene calcolato iterativamente facendo $\mod p$ ad ogni passo. Si può quindi avere una sottostringa di T con lo stesso fingerprint del pattern che però non è uguale al pattern, è un **falso positivo**. Non si possono tuttaavia avere falsi negativi, quindi tutte le occorrenze sono trovate con la possibilità di trovare occorrenze false in più:

$$H(T[i : i + m - 1]) \mod p = H(P) \mod p \Leftarrow T[i : i + m - 1] = P$$

Se il numero primo $p$ è scelto a caso minore di un certo $I$ so che l'errore è minore di $O(\frac{nm}{I})$.
Vogliamo sfruttare però che si hanno solo falsi positivi e provare ad eseguire l'algoritmo con due $p$ diverse, le vere occorrenze saranno trovate da entrambe mentre i falsi positivi probabilmente no. Itero quindi su $k$ numeri primi e il risultato sarà l'intersezione di tutte le $k$ iterazioni dell'algoritmo, riducendo moltissimo le probabilità di avere un risultato errato. Paghiamo quindi un incremento di un prodotto $k$ delle operazioni (diventa $O(k(n + m))$) per ridurre esponenzialmente le chances di errore.

Proponiamo una versione semplificata dell'algoritmo (lunghezza del testo $= n$ e del pattern $= m$):

```
function RabinKarp(text, pattern)
    patternHash ← hash(pattern[1 : m])
    for i ← 1 to n − m + 1 do
        textHash ← hash(text[i : i + m − 1])
        if textHash = patternHash then
            if text[i : i + m − 1] = pattern[1 : m] then
                return(i)
            end if
        end if
    end for
    return(NotFound)
end function
```

È quindi un algoritmo probabilistico in quanto i $p$ sono scelti a caso. Ci sono due categorie di algoritmi probabilistici:

1. **Monte Carlo**, come Karp-Rabin, veloci ma non sempre corretti
2. **Las Vegas**, sempre corretti ma non sempre veloci, come per esempio il quicksort con pivot random (dove il caso migliore è un pivot che è l'elemento mediano mentre il peggiore è che il pivot sia un estremo, portando l'algoritmo ad essere quadratico).

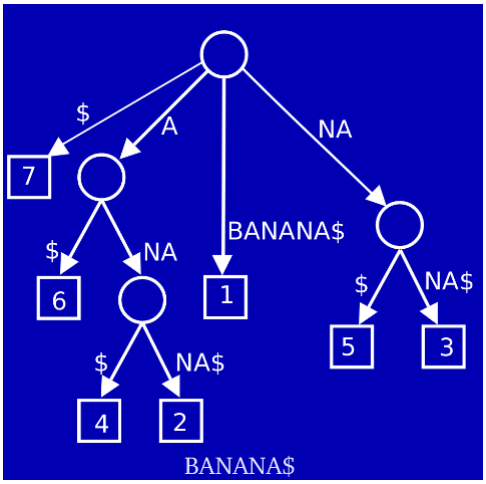
*È possibile rendere Karp-Rabin un algoritmo della categoria Las Vegas controllando tutti i falsi positivi (anche se non è una procedura utlizzata).*

## 2.3 Trie

Si usera la filosofia che prevede il preprocessamento del testo.
Il **trie** è una struttura ad albero, con archi etichettati, che, preso un insieme di parole, detto dizionario, controlla che quella sequenza sia nell'insieme di parole. Si tratta di un **problema di membership**. Voglio un tempo $O(n)$ con $n$ lunghezza della query. Quindi si preprocessa una volta sola il dizionario in un albero e poi si procede con le query. Nella pratica si ha che un percorso radice-foglia deve essere esattamente la query richiesta. Si ha però un problema, essendo ogni ramo un elemento del dizionario, non si possono avere parole diverse nel dizionario che siano l'una prefissa dell'altra (per

esempio se ho *abraabra* non posso avere anche *abra* nel dizionario), perché non riuscirei ad andare da radice a foglia. Introduco quindi il carattere \$, il quale non appartiene all'alfabeto che viene aggiunto alla fine di ogni stringa, viene infatti detto **terminatore**, così *abra*\$ non è prefisso di *abraabra*\$ etc..., rimuovendo così ogni ambiguità.

Usiamo una struttura dati chiamata **suffix tree**, che è il trie di tutti i suffissi di $T$\$, che quindi è un insieme più specifico di stringhe (i suffissi). È una sorta di trie compatto, dove un suffisso diventa l'etichetta di un arco. Le etichette degli archi uscenti, i figli, da $X$ inziano con simboli diversi. I suffissi sono il percorso radice-foglia.




Una sottostringa è il *prefisso di un suffisso*. Dato un pattern voglio trovare tutti i suffissi che iniziano col pattern. Quindi, dato che ogni nodo ha un solo figlio con un certo prefisso, la procedura di pattern matching nel suffix-tree naviga nell'albero sequendo il pattern nell'unico arco possibile con quel pattern, posto che esista. Se non esiste il pattern cercato termina, così come termina nel momento in cui lo trova raggiungendo un \$. Il compattamento da trie a suffix tree serve a migliorare le performance di costruzione della struttura dati, non quelle del pattern matching in sè. Un suffix-tree ha un rapporto tra il numero di nodi e il numero di foglie (?).
Questo pattern matching ha $O(m)$ con $m$ lungheza del pattern da ricercare ma poi ho tante foglie $k$ sotto il nodo a cui sono arrivato quante sono le occorrenze del pattern nel testo, che quindi visito in $O(k)$. Nel complesso ho la costruzione dell'albero in $O(n)$, $n$ lunghezza testo, matching in $O(m)$ e visita finale delle foglie in $O(k)$, quindi nel complesso ho

$$O(n + m + k)$$

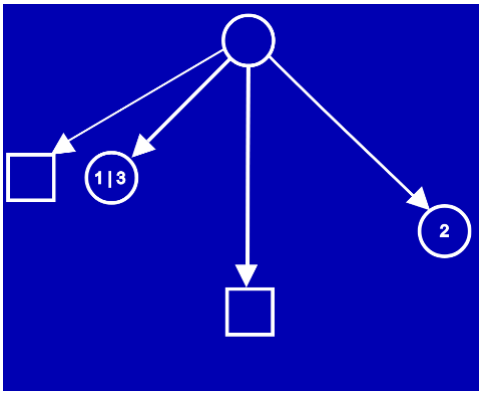
Se il pattern termina prima del passaggio ad un nodo successivo non mi interessa in quanto i suffissi corrispondono (se ho $NA$ e mi fermo a $N$ va bene lo stesso, in quanto i suffissi di $NA$ sono dello stesso numero di quelli di $N$). Nelle foglie ho indicato l'indice dove inizia ogni occorrenza.
Definisco la **path-label(X)** di un nodo X è la concatenazione delle stringhe fino a quel nodo. Definisco invece **string-depth(X)** di un nodo X la lunghezza del path-label, che è calcolabile in $O(n)$. La string-depth di X sarà la lunghezza dell'etichetta sommata alla string-depth del padre.
*Molti algoritmi sfruttano molte visite per arricchire le informazioni dell'albero ai fini di rendere semplice la risoluzione di un problema.*
Non posso fare lo stesso ragionamento per la path-label perché, concatendando quella del padre alla propria raggiungerei quasi $O(n^2)$. Si usa quindi una tecnica basata su puntatori al testo e non su stringhe, questo fa si che ogni arco sia etichettato da una coppia di numeri (posizione di inizio e lunghezza), quindi raggiungo tempo costante per etichettare ogni albero.
Il grande problema del suffix-tree è lo spazio occupato, circa $20n$ byte, quindi per il genoma umano servono 60gb di memoria. Per risolvere questo problema si usano i **suffix array (SA)**, che occupa meno spazio, ed è *l'array dei suffissi in ordine lessicografico.* Il suffix array non permette il pattern matching in tempo lineare quindi viene usato il suffix array per costruire il suffix tree. Tutto quello che si può fare sul suffix tree si può fare sul suffix array con tempi diversi ma simili. Non memorizzo esplicitamente i suffissi ma memorizzo le posizioni iniziali del suffisso. Al suffix array si aggiunge l'array ausiliario *LCP (longest common prefix)* con la lunghezza del prefisso comune $LPC[i]$ tra $SA[i]$ e $SA[i+1]$.

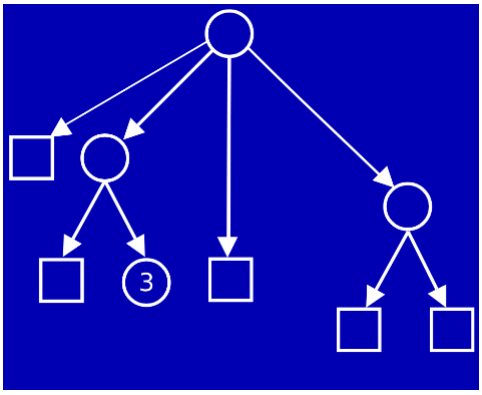
BANANA$

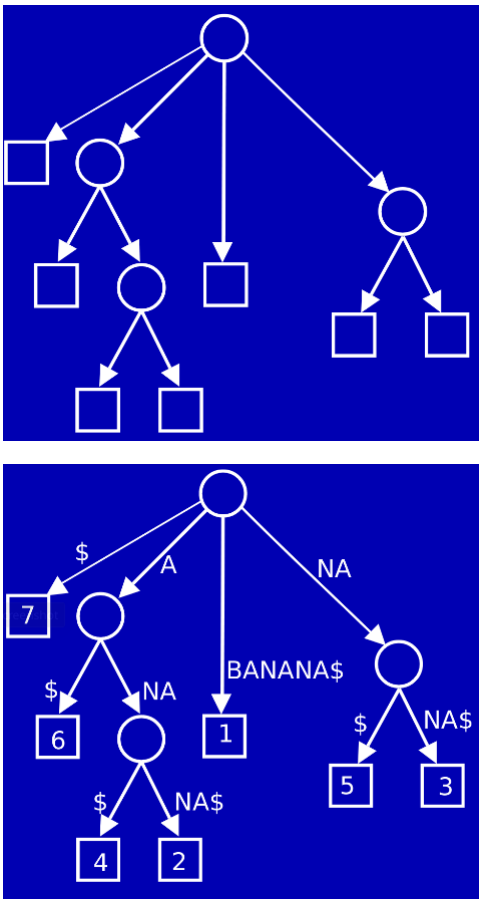
| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $SA$ | 7 | 6 | 4 | 2 | 1 | 5 | 3 |
| $Lcp$ | 0 | 1 | 3 | 0 | 0 | 2 | - |

Lo spazio diventa $4n$ bytes, quindi per il genoma 12gb.
Passiamo ora dall'albero all'array. Facciamo una visita depth-firts (pre-order) del suffix array, assumento che a sinistra ci siano etichette in ordine lessicografico minore (a sinistra ho suffissi con una lettera dell'alfabeto iniziale "precedente").
Ho quindi che $LPC[i]$ è la string-depth di $LCA(i, i+1)$, ovvero *least common ancestor*, che quindi mi indica dove due percorsi divergono e vedo quanto vale lì la string-depth.
Calcolare LPC costa un tempo quadratico. Ma possiamo fare un altra cosa. Ogni arco viene visitato almeno due volte inq aundo una volta arrivati ad una foglia si torna indietro. Mi salvo la string depth dell'LCA nell'array ogni

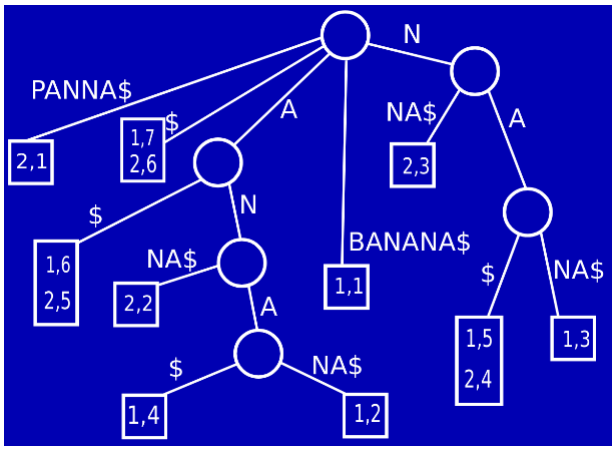
volta che ho un cambio di direzione nella visita dell'albero lavorando quindi in tempo lineare. Nel disegno guardiamo due foglie consective e contiamo i nodi che hanno in comune sopra esclusa la radice.
Dati SA e LCP passiamo ora a costruire l'albero. Gli LCP nulli partizionano il SA e corrispondono ai figli della radice del suffix array aggiungendo 1 (tre zeri corrispondono 4 figli). Di questi 4 figli prendo i valori no nulli e saranno un cammino che prosegue ripetutamente sul valore minimo.
*Quindi se l'array è 013002 avrò 4 figli della radice, uno 0, uno 13, uno 00 (che conta come 0) e uno 2. Il 13 avrà una foglia a sinistra e poi scenderà di un nodo. Avendo solo un numero (3) avrà due foglie. Infine avrò due, che essendo un solo numero, avrà solo 2 foglie.*





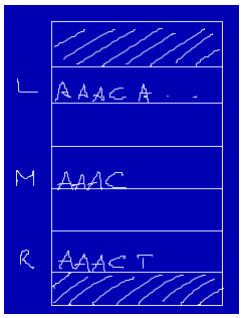

Un **suffix tree generalizzato** rappresenta un insieme di testi. Prendo x stringhe con terminatore, le concateno in un unico testo e ne genero il suffix tree ma nelle foglie avremo le coppie (numero stringa, posizione inizio suffisso) che potrebbero essere una per ogni stringa e lo costruisco in tempo lineare alla lunghezza del testo completo.



Cerco ora la sottostringa comunque più lunga tra due stringhe sfruttando il suffix tree generalizzato. Una sottostringa comune corrisponde ad un cammino tra una radice e un nodo X e tra i discendenti di X devo avere almeno un discendente per ogni stringa. La più lunga sarà quindi quella che arriva al nodo con string-depth massimo. Visito quindi due volte l'albero, alla prima vedo se ha le foglie giuste e la seconda per cercare il max delle string-depth. La prima visita la faccio dalle foglie verso la radice, creo un array di booleani per il nodo X lungo quanto il numero di stringhe i cui elementi valgono true sse esiste un disendente di X tale che è suffisso della stringa S. Alla fine faccio l'and tra tutti i valori dell'array, se è true significa che ho un discendente per ogni stringa. Se X è nodo interno faccio l'or con l'array del nodo discendente per determinare se X va bene. Se è foglia determino il vettore leggendone l'etichetta. La seconda visita può essere in qualsiasi direzione e dove ho nodi con la proprietà di essere comune alle stringhe e trovo quello con la string-depth max, ottengo quindi un tempo che è $O(k \cdot n)$.

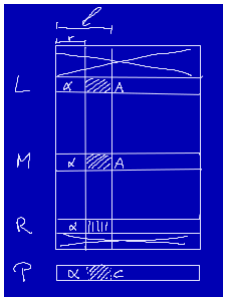
### 2.3.1 Pattern Matching su Suffix Array

Il suffix array contiene le posizioni di inizio dei suffissi ordinati lessicograficamente e viene utilizato insieme all'LCP. Il pattern matching in questo caso non sarà ottimale ma si otterrà $O(m \log n)$, con $m$ lunghezza del pattern. Si sfrutta la ricerca dicotomica, che si basa sulla ricerca a partire dall'elemento mediano di un array, per poi cercare su una sola delle metà dell'array, dimezzando di volta in volta l'array. La ricerca dicotomica funziona sul suffix array ma ogni iterazione consisite nel confrontare il pattern con i primi $m$ caratteri del suffisso mediano, fermandosi ovviamente al primo carattere discordante per capire poi su quale metà del suffix array continuare a cercare, basandosi sui singoli caratteri che non matchano. Quest'ultimo passaggio è possibile grazie al fatto che il suffix array è ordinato lessicograficamente. Si avranno 3 aggiustamenti all'algoritmo, detti **acceleranti**, che abbasseranno la complessità del caso pessimo (col terzo si arriva a $O(m + \log n)$). Con gli acceleranti si evitano confronti inutili.
Partiamo col **primo accelerante**.
Si parte dal presupposto che ci sia un ordine lessicografico posso individuare regioni, dal $L$ a $R$, in cui tutti i possibili pattern iniziano con gli stessi caratteri iniziali.




Formalmente si ha che tutti i fuffissi in $SA(L, R)$ iniziano con lo stesso prefisso lungo $Lcp(SA[L], SA[R])$ e quindi potrò non controllare i primi $Lcp(SA[L], SA[R])$ caratteri.
Vediamo il **secondo accelerante**.
Si ha un elenco di casi che tengono in considerazione il pattern. Si indica con $l$ l'Lcp tra il pattern e il primo suffisso dell'intervallo e $r$ l'Lcp tra il primo e l'ultimo suffisso dell'intervallo. Queste due variabili vengono aggiornate man mano e ho:

- **caso 1:** se $l > r$ quindi il pattern somiglia più al primo suffisso che all'ultimo




  Quindi dovrò poi controllare nella seconda metà del suffix array quindi scarterò in tempo costante tutta la prima metà solo se $Lcp(L, M) > l$, assegnando a $L$ il valore di $M$.
  Se fosse minore farei uguale scartando la seconda metà, assegnando a $R$ il valor edi $M$ e ad $r$ l'$Lcp(M, L)$ (perché dovrò fermarmi a metà).
  Se fossero uguali dovrei confrontare $P[l+1:]$ e $M[l+1:]$ per decidere quale metà scartare, se avrò un carattere che non matcha tra il suffisso mediano e il pattern con l'ultimo carattere del suffisso mediano più piccolo allora cercherò nella seconda metà, altrimenti nella prima (sempre grazie all'ordine lessicografico). A differenza dei due sottocasi sopra questo impiega un tempo che dipende

dal numero di caratteri uguali che trovo, più sono e più impiego tempo, anche se per assurdo più caratteri uguali trovo più mi sto stringendo alla regione che contiene il match del mio pattern. **Si ha consumo di tempo solo nel caso in cui mi stia avvicinando alla soluzione, se "pago" tanto in un'iterazione avrò la certezza di "pagare" meno la volta seguente. Al massimo potro avere un costo pari a** $2m$ **perché ogni volta incremento o** $l$ **o** $r$**, che sommati sono per forza** $\leq 2m$**, quindi alla peggio ho complessitò** $O(2m)$. *In ogni caso $l$ e $r$ potrebbero restare invariati o crescere in base al numero di caratteri che matchano e al massimo avranno valore pari alla lunghezza del pattern $m$*

- **caso 2:** se $l = r$. Se ho $Lcp(L, M) > l$ tengo la prima metà mentre se ho l'opposto $Lcp(L, M) > l$ tengo seconda metà come nel primo caso. Se invece $Lcp(L, M) = Lcp(M, R) = l$ ragiono come nel terzo sottocaso del primo caso

- **caso 3:** se $l < r$ faccio lo speculare del primo caso

**Esempio 1.** *ho i seguenti suffissi: \$, A\$, ANA\$, ANANA\$, BANANA\$, NA\$ e NANA\$ e pattern BA ho quindi* $L = 0$, $R = 6$ *e* $M = 3$.
$Lcp(L, M) = 0$ *e* $Lcp(R, M) = 0$.
*Confronto il primo carattere del pattern e il primo del mediano* $B \neq A$ *e* $B > A$ *quindi ho* $L \leftarrow 3$ *mentre R resta uguale (cosi come r e l). Il nuovo mediano è tra 3 e 6, quindi, arrotonando, 5.*
*Confronto B e N e quindi cerco nella prima metà, che ha un solo elemento, quindi controllo i caratteri e scopro che, in questo caso, ho il match con BANANA.*

**Esempio 2.** *ho i seguenti suffissi: \$, A\$, ABRA\$, ABRACADABRA\$, ACADABRA\$, ADABRA\$, BRA\$, BRACADABRA\$, CADABRA\$, DABRA\$, RA\$ e RACADABRA\$ e pattern BRACA ho quindi* $L = 0$, $R = 11$ *e* $M = 6$.
$Lcp(L, M) = 0$ *e* $Lcp(R, M) = 0$.
*Confronto con BRA\$ e vedo che ci sono 3 caratteri uguali, quindi ora ho* $L = 6$, $R = 11$, $M = 9$, $l = 3$ *(i 3 caratteri uguali) e* $r = 0$.
*Confronto con DABRA\$ e vedo che il primo carattere non matcha e* $D > B$ *quindi cerco nella metà sopra. Quindi ora ho* $L = 6$, $R = 9$, $M = 8$, $l = 3$ *(i 3 caratteri uguali) e* $r = 0$.
*Confronto con CADABRA\$ e vedo che il primo carattere non matcha e* $C > B$ *quindi cerco nella metà sopra. quindi ora ho* $L = 6$, $R = 8$, $M = 7$, $l = 3$ *(i*

*3 caratteri uguali) e $r = 0$.*
*Confronto con BRACADABRA\$ e vedo che ho $Lcp(L, M) = 3$ (prima era sempre stata nulla cosìccome $Lcp(R, M)$) e quindi sono nel terzo caso e confronto solo a partire dal terzo carattere escluso, trovando il match (di 5 caratteri).*

Vediamo ora il **terzo accelerante (DA SISTEMARE)**.
Con questo accelerante studiamo il calcolo dell'Lcp. Alla prima iterazione avrò $L = 1$ e $R = n$, alla seconda avrò o $L = 1$ e $= \frac{n}{2}$ oppure $L = \frac{n}{2}$ e $R = n$ (a seconda della metà scelta). Ad interazione $k$ avrò una forma del tipo $L = h\frac{n}{2^{k-1}}$ e $R = (h+1)\frac{n}{2^{k-1}}$ con $h$ che rappresenta l'indice dell'intervallo (al primo giro ho $h = \{1, 2\}$, al secondo in $h = \{1, 2, 3, 4\} \dots h = \{1, \dots, 2^k\}$). Al termine della mia ricerca dicotimica ho intervalli di cardinalità due. Sempre alla fine della ricerca dicotomica si avranno al massimo $n$ intervalli. Entra quindi in gioco il terzo accelerante, che consiste nel preprocessare tutti gli intervalli, quindi precalcola tutti i vari Lcp. **Questo accelerante è di solo interesse teorico**. Voglio calcolare abbastanza velocemente questi Lcp. Per ogni intervallo calcolo l'Lcp ma in realtà per intervalli consecutivi (quindi nel caso k-simo) ho già l'array Lcp e per ottenere gli altri aggrego i risultati dell'interazione. Quindi Lcp tra un $L$ e un $R$ è il minimo tra l'Lcp della prima metà, l'Lcp della seconda meta e l'Lcp tra $M$ e $M + 1$ che è nell'array Lcp (mentre quelli delle due metà sono già calcolate). Ottengo quindi un $O(n)$.
Ho quindi ottenuto l'algoritmo per trovare un'occorrenza, che non è per forza la prima. Voglio ora espandere per trovare tutte le occorrenze in un tempo proprorzionale al numero stesso di occorrenze. So che tutte le occorenze sono in un intorno dell'occorrenza trovata. Cerco quindi i suffisi precedente e seguenti fino a non trovare alcun match. Parto con l'occorrenza precedente e uso l'array Lcp, in quanto controllo suffissi consecutivi, con l'array vedo quanti caratteri iniziali condidono e se ne condividono almeno $m$ (cardinalità del pattern) allora ho trovato un'altra occorrenza. Dovendo solo leggere un valore ho tempo costante e quindi nel complesso ho un $O(n + m + k)$ con $k$ numero di occorrenze.
In totale calcolo suffix array, array Lcp, preprocessamento e scansione per trovare tutte le occorrenze, il tutto in $O(m \log n)$

### Pattern Matching

Per calcolare la sottostringa comune più lunga ho i seguenti step:

1. calcolo il suffix tree generalizzato della stringa $S$
2. cerco il nodo $x$ tale che:

(a) per ogni stringa $s_i \in S$ esiste un discendente di $x$ corrispondente ad un suffisso di una delle stringhe $s_i$

(b) abbia la string-depth di massima fra tutti i nodi con la stessa caratteristica

Dobbiamo "rimappare" questo ragionamento sul suffix array.
Una volta che i suffissi sono ordinati lessicograficamente essi appaiono consecutivamente nel suffix tree (di $n$ nodi), come se fossero una porzione contigua del suffix array. Però dal suffix array posso estrarre $\binom{n}{2}$ intervalli.
Parto quindi da un suffix array generalizzato (col suo array LCP). Estraggo un generico intervallo $[i : j]$ del suffix array, provando ad estrarli tutti, mi salvo il prefisso comune $p$ di tutti i suffissi in quell'intervallo e se quell'intervallo contiene almeno un suffisso per ogni stringa $s_i \in S$ e la lunghezza del prefisso estratto $p$ è maggiore di $t$ (maggior lunghezza fino a quel momento) allora mi salvo quella $p$ in $t$. Ma questa tecnica non può essere lineare. Posso quindi diminuire il numero di intervalli da considerare e devo calcolare velocemente la lunghezza del prefisso comune in $p$ (nel tree era facile calcolarlo in quanto si usava la string-depth).
Iniziamo col vedere come ridurre gli intervalli, cercando quelli inutili. Fissato un punto finale $j$ guardo tutti gli intervalli che finisono in $j$. Per ogni $j$ che controllo voglio considerare solo un punto di inizio $i$. Non voglio intervalli con suffissi di una sola della stringhe in ingresso. Presi un $l \leq i \leq j \leq q$ e considero il $SA[i : j]$ e $SA[l : q]$ con il primo intervallo quindi incluso nel secondo. Se $SA[l : q]$ non contiene nemmeno un suffisso per ogni stringa in ingresso allora anche $SA[i : j]$ farà lo stesso. Sia quindi $t_1$ un prefisso comune di $SA[i : j]$ e $t_2$ di $SA[l : q]$ allora la lunghezza di $t_1$ è $\geq$ di quella di $t_2$. Cerco quindi l'unico punto di inizio per ogni fine $j$ che ha senso considerare, ovvero calcolo il massimo $i$ tale per cui $SA[i : j]$ contiene almeno un suffisso per ogni stringa in ingresso, ho quindi un numero lineare di intervalli da considerare. Per calcolare $i$ tengo traccia, per ogni stringa in ingresso, di ogni ultima volta che ho un suffisso (creo quindi un array $last$). Fissato $j$ quindi prendo l'intervallo minimo che finisce in $j$ e sfrutto l'array $last$. Quindi $i$ sarà il massimo valore tale che $SA[i : j]$ contiene almeno un suffisso per ogni stringa in ingresso. **capire** Quindi considero un intervallo $[i : j]$ e la lunghezza del prefisso comune a tutti i suffissi che è uguale al minimo dell'LCP tra $i$ e $j-1$
Questo sistema comporta un tempo quadratico e bisogna arrivare ad un tempo lineare.
Io ho un array $A$ di $n$ elementi e voglio calcolare il minimo in un certo intervallo. È il **range minimun query**. Vogliamo indicizzare in tempo $O(n \log n)$ e calcoleremo, grazie a questo preprocessamento, il minimo in di un intervallo in tempo costante $O(1)$. Il sistema consiste nel precalcolare i

minimi per alcuni intervalli che bastano per tutti. Il calcolo del minimo sarà poi un confronto tra due intervalli. Se voglio il minimo in un intervallo e ho precalcolato il minimo di due intervalli che insieme coprono l'intero intervallo allora mi basta confrontare i due minimi dei sottointervalli pre calcolati.
Preprocesso quindi piccoli intervalli ma che nel complesso mi coprano, a coppie, tutto l'intervallo completo. Partiamo cercando un $z$ tale che $2^z \leq j-i+1$ e prendo poi due intervalli, uno $[i : i+2^z-1]$ e uno $[j-2^z+1 : j]$. Ho quindi che $z = \lfloor \log_2 j - i + 1 \rfloor$. Uso quindi la programmazione dinamica con una matrice $B[x, y] = \min_{x \leq z < x+2^y} A[z]$:

$$\begin{cases} B[x, 0] = A[x] \\ B[x, y] = \min\{B[x, y-1], B[x+2^{y-1}, y-1]\} \text{ se } y \geq 0 \end{cases}$$

nel complesso ho quindi $n \log n$ elementi. Costruita questa matrice vado a prendere il:

$$\min\{B[i, wr], B[j-2^w+1, w]\}$$

dove $w$ è la più grande potenza di 2 minore o uguale a $j-i+1$. **finire**

# Capitolo 3

# Allineamenti

Ovviamente in bioinformatica uno dei problemi principali è il confronto di sequenze biologiche per lo studio, per esempio, delle proteine, alla ricerca di similitudini tra sequenze. Una sequenza biologica si basa su un alfaberto formato dalle basi azotate:

$$\Sigma = \{A, C, T, G\}$$

In biologia molecolare si ha che **se due sequenze sono strutturalmente simili allora hanno anche una funzione simile**. Si ha quindi la *ricerca di omologie*, anche per lo studio dell'evoluzione.

**Definizione 1.** *Si definsice* ***distanza di Hamming*** *il numero totale di caratteri differenti, a parità di posizione, tra due stringhe:*

$$d : S \times S \to \mathbb{R}^+$$

*con $S \times S$ insieme delle coppie di stringhe.*
*La distanza di Hamming gode di:*

- ***riflessività****:* $d(x, y) = 0 \iff x = y,\ \forall x, y \in S$
- ***simmetria****:* $d(x, y) = d(y, x),\ \forall x, y \in S$
- ***diseguaglianza triangolare****:* $d(x, y)+d(y, z) \leq d(x, z),\ \forall x, y \in S$

***La distanza di Hamming è definita unicamente per stringhe di egual lunghezza***

## 3.1 Allineamento globale

Si analizza l'allineamento di due intere stringhe, si ha quindi un problema di **ottimo gloable**. Si analizzano due stringhe di lunghezza arbitraria e non uguale tra di loro. Ci si riconduce all'uso della distanza di Hamming inserendo spazi fino ad ottenere due stringhe di egual lunghezza per poter lavorare colonna per colonna. Si usano caratteri **indel** (*in* inserisco e *del* tolgo). Si hanno delle proprietà:

- non si possono avere colonne di soli indel
- le stringhe estese con gli indel devono essere lunghe uguali

Per esempio si ha quindi:

Input
```
ABRACADABRA
BANANA
```
sequenze allineate 1
```
ABRACADABRA
-B-ANA---NA
```
sequenze allineate 2
```
ABR-AC-ADABRA
---B-ANA---NA
```
sequenze allineate 3
```
ABRACADABRA
-BANA----NA
```

Un buon allineamento deve contenere pochi indel, molti caratteri allineati e pochi non allineati.

**Definizione 2.** *Un problema di ottimizzazione richiede:*

- *un'istanza che è un insieme infinito di casi*
- *un insieme di soluzioni ammissibili verificabili in tempo polinomiale*
- *una funzione obiettivo, che è una soluzione ammissibile che mappa in* $\mathbb{Q}$
- *una soluzione che massimizza (massimizza il valore) o minimizza (minimizza il costo) la funzione obiettivo*

Si usa quindi la programmazione dinamica per stabilire uno score che cresce quando i valori delle colonne coincidono e quindi si cerca di massimizzare questo valore.
Cerchiamo quindi l'equazione di ricorrenza di questo problema di massimizzazione.
Innazitutto abbiamo come variabile una matrice di score:

$$d : (\Sigma \cup \{-\}) \times \Sigma \cup \{-\}) \to \mathbb{Q}$$

Quindi date due stringhe $s_1$ e $s_2$ si ha che:

$$M[i,j] = \text{ottimo su}s_1[:i],\ s_2[:j]$$

con la seguente equazione di ricorrenza, detta di **Needleman-Wunsch** (si indicano con "-" gli indel):

$$M[i,j] = \max \begin{cases} M[i-1,j-1] + d(s_1[i], s_2[j]) & \text{se non ho indel} \\ M[i,j-1] + d(-, s_2[j]) & \text{se ho indel solo in } s_1 \\ M[i-1,j] + d(s_1[i], -) & \text{se ho indel solo in } s_2 \end{cases}$$

con le seguenti condizioni a contorno:

$$\begin{cases} M[0,0] = 0 \\ M[i,0] = M[i-1,0] + d(-, s_2[j]) \\ M[0,j] = M[0,j-1] + d(-, s_2[j]) \end{cases}$$

Si ha quindi un doppio ciclo for e un tempo pari a $O(nm)$.
Fino ad ora si è aprlato di **allineamento globale**. Si definisce invece **allineamento locale** l'individuazione di due sottostringhe $t_1 \subseteq s_1$ e $t_2 \subseteq s_2$. Date altre due sottostringhe $u_1, u_2$ delle 2 stringhe si ha che:

$$M[t_1, t_2] \geq M[u_1, u_2]$$

Un algoritmo banale avrebbe tempi assurdi: $O(n^3 m^3)$.
Questo problema si può risolvere velocemente con il metodo **Smith-Waterman**.
Con questo metodo l'allineamento viene ricercato tra tutte le sottostribghe terminanti nelle posizioni $i$ e $j$, ovvero $t_1 = s_1[h,i]$ e $t_2 = s_2[k,j]$, con $h$ e $k$ incogniti e con massimo valore di $M[t_1, t_2]$. Si richiede inoltre che l'allineamento ottimo abbia un'ultima colonna senza indel. Grazie a questa clausola si ha che $s_1[h, i-1]$ e $s_2[h, j-1]$ è l'allineamento ottimale delle sottostrighe terminanti un carattere prima di $i$ e $j$. Supponiamo che queste ultime due sottostringhe inizino da $a$ e $b$ e che:

$$M(s_1[a, i-1], s_2[b, j-1]) > M(s_1[h, i-1], s_2[k, j-1])$$

Aggiungiamo ora anche la colonna $i$ e quella $j$. Se si hanno caratteri uguali nell'ultima colonna si ha che:

$$M(s_1[a,i], s_2[b,j]) > M(s_1[h,i], s_2[k,j])$$

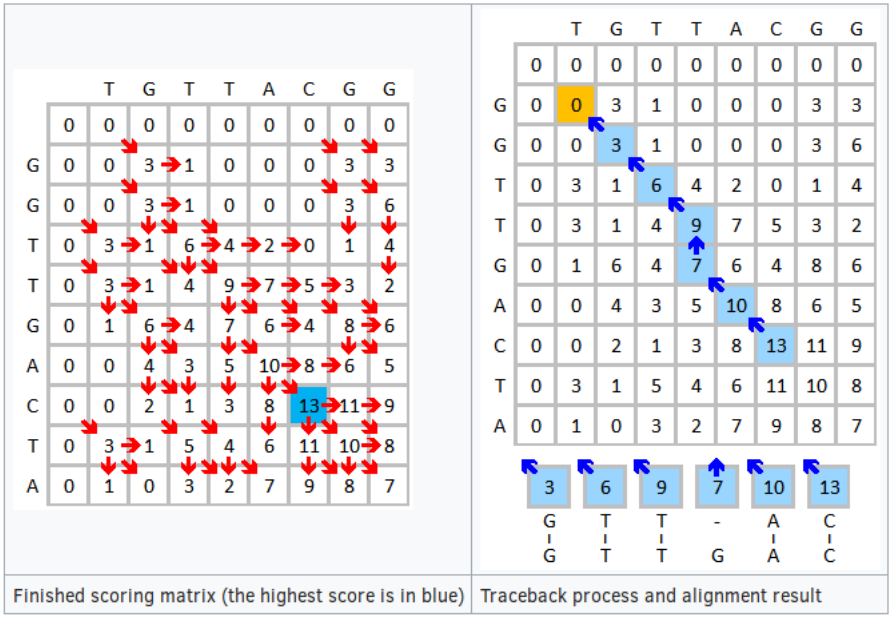
Ma questo è un assurdo essendo $M(s_1[h,i], s_2[k,j]$ massimo per ipotesi.
Questo ragionamento si estende anche ai casi in cui ci sia un'indel a termine di una delle due stringhe. Si introduce anche un nuovo caso, aggiuntivo rispetto a quelli di Needleman-Wunsch: il caso in cui nessuna sottostringa ha un allineamento positivo. In questo caso se precedentemente non ci sono match si setta la casella a 0 e si evitano i valori negativi. Una conseguenza diretta è che:

$$M[0,0] = M[i,0] = M[0,j] = 0$$

Dopo aver costruito la matrice cerco il valore massimo e le coordinate $x, y$ indicano le posizioni finali delle sottostringhe mentre le coordinate del primo 0 indicano le posizioni iniziali.
Definiamo quindi l'equazione di ricorrenza:

$$M[i,j] = \max \begin{cases} M[i-1, j-1] + d(s_1[i], s_2[j]) & \text{se non ho indel} \\ M[i, j-1] + d(-, s_2[j]) & \text{se ho indel solo in } s_1 \\ M[i-1, j] + d(s_1[i], -) & \text{se ho indel solo in } s_2 \\ 0 & \text{altrimenti} \end{cases}$$

Si è quindi raggiunto un $O(nm)$.
Graficamente si ha:



**Definizione 3.** *La **distanza di Edit** è la trasformazione di una stringa in un'altra tramite l'inserimento, la cacncellazione o la modifica di un carattere. Inoltre si ha che la distanza di edit è un caso particolare dell'allineamento globale tra due sequenze.*
***L'allineamento globale racchiude tutte le possibili casistiche delle distenze di Edit**.*
*Si ha la seguente equazione di ricorrenza:*

$$M[i,j] = \min \begin{cases} M[i-1,j-1] & \text{se } s_1[i] = s_2[j] \\ M[i-1,j] & \text{se viene cancellato un carattere} \\ M[i,j-1] & \text{se viene aggiunto un carattere} \\ M[i-1,j-1]+1 & \text{se viene modificato un carattere} \end{cases}$$

Sulla stringa si procede da sinistra a destra.
Quindi il costo di ogni mismatch è pari a 1.
La distanza di Edit ha tempo $O(nm)$ **rivedere completamente**

Con LCS abbiamo un problema di massimizzazione dove mettiamo nella matrice di score 1 in corrispondenza di caratteri uguali e 0 negli altri casi. Con la distanza di Edit abbiamo 0 in caso di uguali caratteri e 1 nelle altre posizioni. Le due matrici di score sono quindi complementari.

Facciamo un altro ragionamento. Penso ad un allineamento di minimo costo dove "taglio" un prefisso e un suffisso delle due stringhe, ovviamente un prefisso o un suffisso di una delle due stringhe. Quindi, dato che parliamo di minimizzazione, quanto taglio ha costo 0. Bisogna trovare un modo per imporre costo 0 per i caratteri scartati ragionando sull'equazione di ricorrenza **senza toccare la matrice di score**. Basta modificare le condizioni a contorno, mettendo a 0 la prima riga e la prima colonna:

$$\begin{cases} M[0,0] = 0 \\ M[i,0] = 0 \\ M[0,j] = 0 \end{cases}$$

**Abbiamo quindi una nuova variante dell'allineamento globale dove minimizziamo il costo e non massimiziamo la validità, scartando un prefisso di una delle due stringhe**.
*Questo proedimento è comodo se si hanno stringhe di lunghezza molto diversa, trovando un allineamento buono in modo che dove si allineano si abbiano pochi indel.*

Abbiamo visto come scartare un prefisso, ora ragioniamo sul suffisso. Cerco quindi il minimo nell'ultima riga e nell'ultima colonna e decido di conseguenza quale suffisso scartare e i quale stringa (se è nell'ultima riga scarto da $s_2$, se nella colonna è in $s_1$).
Un altra variante prevede una lunghezza $k$ del prefisso o suffisso da scartare. Per il suffisso ragiono solo le ultime $k$ posizioni nell'ultima riga e dell'ultima colonna, mentre per il prefisso metto 0 solo nelle prime k posizioni della prima riga e della prima colonna.
In ogni caso per ricorstruire risalgo fino ad un 0.
Analizziamo un ulteriore approccio. Questa volta si è interessati solo al costo di un allineamento globale, cercando però di ottimizzare sullo spazio. Per calcolare $M[i, j]$, essendo una LCS, mi bastano solo il valore a sinistra $M[i-1, j]$, quello sopra $M[i, j-1]$ e quello precedente sulla diagonale $M[i-1, j-1]$. Quindi mi basta tenere la riga o la colonna in cui sono e la riga o la colonna precedente, scegliendo la riga o la colonna in base alla loro lunghezza, che corrisponde quindi alla lunghezza della stringa (se $s_2$ è più corta scelgo la riga altrimenti, se $s_1$ è più corta, la colonna). Non serve quindi tenere tutta la matrice e posso quindi risparmiare spazio.
Ora restringiamo ancora l'allineamento, cercando di risolvere l'allineamento su stringhe molto simili, in maniera ancora più ottimizzata sfruttando la similarità. In altre parole si ha la distanza di Edit con un numero di operazioni che è al massimo un valore $k$, tutto in $O(kn)$. Voglio ridurre il tempo di calcolo dela distanza di Edit sapendo che vale al massimo $k$. Avrò quindi al massimo $kn$ caselle.
Sapendo che ho la distanza di Edit tra le due stringhe vale al massimo $k$ e che le due stringhe sono lunghe $n$ e $m$ so che:

$$|n - m| \leq k$$

Si ha quindi una matrice di programmazione dinamica che è quasi quadrata. Si ha che le parti che non costano corrispondono ad uno spostamento sulla diagonale, quindi al diminuire della distanza di Edit mi avvicino alla diagonale principale. Posso quindi non considerare le caselle lontane della diagonale. È l'**allineamento mediante distanza di Edit con banda**, ovvero seleziono un intorno della diagonale principale (appunto una banda) da analizzare, perché fuori dalla stessa avrò solo soluzioni non ottimali.
**disegno**
Si arricchiscono ancora le condizioni a contorno della ricorrenza della distanza di Edit, ponendo a $-\infty$ le caselle in cui $|i - j| \geq k$ (nel codice si avrebbe solo questo controllo). Si potrebbe però non conoscere $k$ o si potrebbe avere un $k$ sbagliato, portando ad una soluzione non ottima (magari perché questa soluzione vorebbe "uscire" nell'area fuori dalla banda). Si procede quindi

raddoppiando $k$ e mi fermo quando la distanza di Edit *edit* è, per un certo $h$ che alla fine varrà $h^* = \log_2 edit$:

$$2^{h-1} < edit < 2^h$$

Ho quindi tempo complessivo pari a:

$$\sum_{i=0}^{h^*} O(2^i n) = O(n) \cdot \sum_{i=0}^{h^*} O(2^i) = O(n) \cdot (2^{h^*+1} - 1) \sim O(n) \cdot (2^{h^*} - 1)$$

$$\Downarrow$$

$$O(n) \cdot (2 \cdot edit - 1) = O(n \cdot edit)$$

Abbiamo quindi un buon tempo, rispetto a $O(nm)$.
*Si raddoppia per essere comodi coi conti con il logaritmo.*
**Ovviamente questa strategia funziona bene sse le stringhe sono abbastanza simili, con $k << n$ e $k << m$.**
Vediamo un'altra variante di allineamento. Ogni volta che si introducono indel si ha un forte "significato" biologico. Quando si aggiunge un indel si sposta il frame di lettura e si shiftano di uno i codoni.

**Definizione 4.** *Si definisce **gap** una sequenza contigua di indel*

Si ha che tra inserire un indel e un gap si ha poca differenza. Si ha quindi che ogni gap è associato ad un costo legato alla sua lunghezza $P(l)$. Per l'allineamento separo quindi in match e mismatch tra caratteri e in costi in presenza di gap.
Si vuole ottenere che allungare di uno un gap deve avere costo costante (allungare un gap da 1 a 2 deve costare come da 11 a 12).
L'ultima colonna dell'allineamento ottimo quindi varia rispetto a quella standard dell'allineamento globale.
Quindi $M[i, j]$ è l'allineameno ottimo si $s_i[1:]$ e $s_2[:j]$.
Cerco quindi la mia equazione di ricorrenza dove si devono considerare i gap. Ma non posso scrivere un'unica equazione di ricorrenza che consideri tutte le possibili lunghezze dei gap. Cerco quindi l'ultima componente per la matrice della programmazione dinamica, che sono l'ultima colonna in assenza di gap o la posizione dell'ultimo gap. Ottengo quindi:

$$M[i,j] = \max \begin{cases} M[i-1, j-1] + d(s_i[i], s_2[j]) & \text{se non ho gap} \\ \max_{l>0} M[i, j-l] + P(l) & \text{se ho gap in } s_1 \\ \max_{l>0} M[i-l, j] + P(l) & \text{se ho gap in } s_2 \end{cases}$$

Ponendo come condizione a contorno che:

$$M[0,0] = 0$$

$$M[i,0] = P(i)$$
$$M[0,j] = P(j)$$

Per i tempi ho, con un'analisi iniziale grezza, sapendo che riempire una casella costa $i + j + 1$:

$$\sum_{i=0}^{n}\sum_{j=0}^{m}(i+j) \leq nm(n+m) = n^2m + nm^2$$

E non si può scendere sotto questo caso pessimo. Ipotizziamo di dividere in 4 la matrice, del quadrante in basso a destra ho $i \geq \frac{n}{2}$ e $j \geq \frac{m}{2}$ e quindi ho

$$t \geq \frac{nm}{4}\left(\frac{n}{2} + \frac{m}{2}\right) \geq \frac{1}{8}nm(n+m)$$

tutto questo solo nell'ultimo quadrante. È quindi impossibile scendere sotto quel caso pessimo. **Questo, $O(nm(n+m))$, è il "prezzo da pagare" per avere costo di gap generico, dove un gap lungo $n$ costa meno di $n$ gaps lunghi 1**.
Bisogna quindi cambiare approccio per migliorare l'algoritmo, riducendo i valori da calcolare, in quanto ora leggo tutti i valori sopra, tutti quelli a sinistra e quello precedente sulla diagonale. Si passa al **gap affine o lineare**, dove si ha una penalità $P_o$ di apertura del gap più $l$ volte quella di estenione del gap $P_e$:

$$costo = P_o + l \cdot P_e, \;\; P_e, P_o < 0$$

Quindi la creazione costa un valore che tiene conto di $P_o$ e $P_e$ mentre l'aumento del gap varia solo su $P_e$. Vista questa situazione cerco le casistiche dell'ultima componente da guardare. Alla situazione di prima si aggiungono dei casi. Si cerca l'allineamento ottimo di $s_1[i-1]$ e $s_2[j:]$ sotto la condizione che l'ultima colonna di tale allineamento abbia un indel per $s_2$ (e si cerca anche il problema simmetrico a questo per $s_1$). Si hanno quindi ottimi vincolati.

Si ha quindi:

$$M[i,j] = \text{ ottimo su } s_1[:i],\, s_2[:j]$$

$$E_1[i,j] = \text{ ottimo su } s_1[:i],\, s_2[:j] \text{ con estensione di gap finale in } s_1$$

$$E_2[i,j] = \text{ ottimo su } s_1[:i],\, s_2[:j] \text{ con estensione di gap finale in } s_1$$

$$E_1[i,j] = \text{ ottimo su } s_1[:i],\, s_2[:j] \text{ con apertura di gap alla fine di } s_1$$

$$N_2[i,j] = \text{ ottimo su } s_1[:i],\, s_2[:j] \text{ con apertura di gap alla fine di } s_2$$

quindi:

$$M[i,j] = \max \begin{cases} M[i-1,j-1] + d(s_1[i], s_2[j]) \\ E_1[i,j], E_2[i,j] \\ N_1[i,j], N_2[i,j] \end{cases}$$

$$E_1[i,j] = \max \begin{cases} E_1[i,j-1] + P_e \\ N_1[i,j-1] + P_e \end{cases}$$

$$E_2[i,j] = \max \begin{cases} E_2[i-1,j] + P_e \\ N_2[i-1,j] + P_e \end{cases}$$

$$N_1[i,j] = M[i,j-1] + P_o + P_e$$

$$N_2[i,j] = M[i-1,j] + P_o + P_e$$

Si ha quindi tempo costante per ogni casella e ottengo tempo pari a $O(nm)$ anche se ho una costante superiore (5) che moltiplica $nm$ da considerare i termini di spazio.
*Si può volendo compattare in 3 matrici.*
**Ho programmazione dinamica in termini di soluzioni di sottoistanze ottime di altri problemi collegati al primo**

### 3.1.1 Allineamento Multiplo

Estendiamo il problema dell'allineamento avendo ora $k$ stringhe in input, oltre alla matrice di score. Si nota che l'allineamento globale è solo un sottocaso dei quello multiplo, ovvero è un allineamto multiplo con $k = 2$.
Per risolvere il problema cerco gli allineamenti per ogni coppia possibile di stringhe. Chiamiamo $t_i$ le stringhe estese. Si hanno alcune proprietà necessarie:

- $|t_1| = \cdots = |t_n|$, ovvero tutte le stringhe estese hanno lunghezza uguale

- non ho colonne di soli indel

Si ha quindi quindi l'equazione per il caso passo della ricorrenza:

$$\sum_{i=1}^{h} \sum_{j=1}^{h} \sum_{n=0}^{k} d(t_i[j], t_n[j])$$

Il tempo è $O(2^k n^k)$ e si ha uno spazio di $O(n^k)$. Avendo $k$ arbitrario quindi si ha a che fare con un problema NP-completo.

### 3.1.2 Matrici di Score

Si hanno matrici di score "famose" per valutare gli allineamenti. I loro dati possono essere letti come le probabilità che avvenga una transizione. Vediamo le più importanti:

- **PAM (*Point/Percent Accepted Mutation*)**. Viene usata per capire la "distanza tra due sequenze" usando la quantità di mutazioni. In realtà si hanno più matrici PAM pvvero $PAMk$, dove $k$ indica l'accuratezza. Queste matrici si calcolano numericamente prendnedo varie seguenze distanti $kPAM$ e allineandole. SI ha che 1PAM è il numero di mutazioni: $\frac{1}{k}|s_i|$. Sia $f(i)$ la frequenza di ogni carattere e sia $f(i,j)$ la frequenza di ogni coppia di caratteri. *Ogni casella, quindi, indica la probabilità che l'aminoacido di quella riga sia stato sostituito con l'aminoacido di quella colonna attraverso il tempo.*
  Si ha quindi la formula:

  $$PAM_k(i,j) = \log \frac{f(i,j)}{f(i)f(j)}$$

  e il risultato è detto **Logs Odd Ratio**

- **BLOSUM (*BLOck SUbstitution Matrix*)** si usano, a differenza delle PAM, per allineare sequenze lontane, infatti divide i blocchi di regioni conservate da quelle mutate, scegliendole manualmente. Anche le BLOSUM sfruttano il Logs Odd Ratio. Si hanno anche qui varie $BLOSUMk$ e la più usata è la $BLOSUM62$

Tra i tool più usati nominiamo invece **BLAST (*Basic Local Alignment Search Tool*)** che confronta sequenze, allineate con *Smith Watermn* in un database sfruttando i *seed*, ovvero pattern matching con una sottostringa di lunghezza 3. L'algoritmo usato su basa su **HSP (*High-scoring Segment***

***Pair*)** che è l'estensione di un seed che massimizza i valori dell'allineamento. Si tengono solo gli HSP significativi mentre gli altri vengono fusi tra loro.
I risultati di BLAST vengono analizzati statisticamente con le **statistiche Karlin-Altschul**, che assegnano un punteggio ad ogni query. Queste statistiche tengono conto di simboli indipendenti ed equiprobabili, sequenze infinitamente lunghe e allineamenti senza gap. Gli HSP con massimo score sono distribuiti casualmente, e si può considerare che la probabilità che questi punteggi superino una determinata soglia segua una distribuzione di Poisson.
Vedno $E$ apri al numero di allineamenti senza gap, $k$ costante dipendente dallo score, $n$ numero di caratteri nel database, $m$ lunghezza della stringa di cui si fa la query e $\lambda S$ il punteggio normalizzato (secondo Poisson) si ha:

$$E = kmne^{-S}$$

# Capitolo 4

# Assemblaggio di Genomi

Si parla di assemblaggio da 0. Nella prima metà degli anni 90 c'è stato il **progetto genoma umano**, che si proponeva di ricostruire la sequenza del genoma umano, lungo circa 3 miliardi di basi. Non si ha una tecnologia per leggere tutto il genoma ma ho delle tecnologie che ne leggono pezzetti, detti **read**, che con la tecnologia di "prima generazione" erano lunghi tra i 50 e i 10000 basi (*base pairs*, nucleotidi), con un errore del $\sim 1\%$ ma un costo economico elevatissimo. Le read "di seconda generazione" sono molto più precise (errore sotto all'uno per mille) ma molto corte, circa 200 basi. La "terza generazione" offre lunghezze estese, circa 10000 basi, ad un costo d'errore elevatissimo ($10\% - 15\%$). Estratte le read non so da quale porzione del genoma arrivino. Si ha quindi una grande collezione di read che devono essere assemblate per ottenere il genoma, a partire da un numero di read vincolate solo dai costi economici (superano comunque in lunghezza il genoma, magari avendo un totale di 60 miliardi di basi). Il rapporto tra read archiviate e lunghezza del genoma originale è detto **copertura**.
Si sfruttano le sovrapposizioni delle read per ottenere il genoma (come se fosse un puzzle).
Spesso le read sono prodotte a coppie e vengono dette *mate pairs*, derivate dalla natura a doppio filamento del del DNA e la **legge di complementarietà di Dawson-Crick**, infatti una base ha la suia complementare nell'altro filamento. In modo deterministico quindi si ottengono due sequenze che si sa essere "vicine".
Formalmente le sovrapposizioni si riducono a cercare un suffisso di una read che sia il prefisso di un'alra read, è la tecnica dell'**overlap**. La differenza di una singola base (o comunque una differenza minima) può essere data dal tasso d'errore del macchinario che produce la sequenza oppure si ha che è una conseguenza della natura diploide dell'uomo, ovvero che si hanno due copie genomiche, una ereditata da ciascun genitore. Magari si ha a che fare quindi

con un confronto tra queste due copie che genera queste minime differenze.

### 4.0.1 Grafi di Overleap

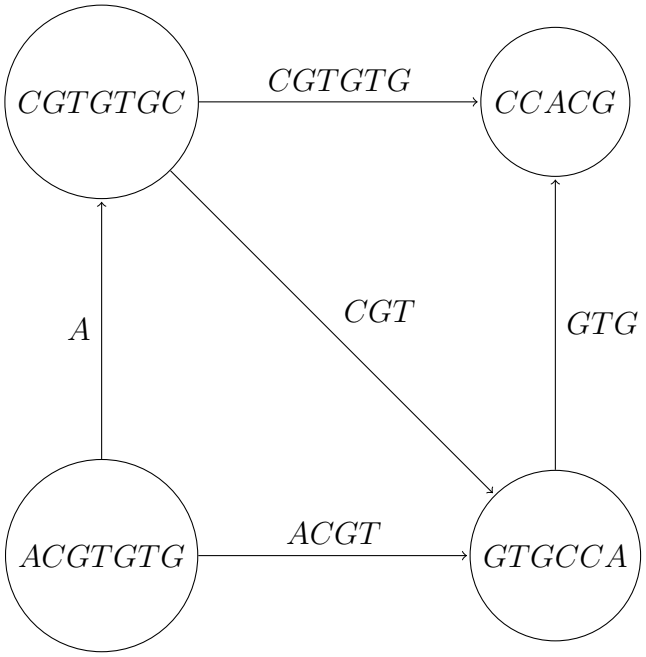
Iniziamo a vedere come ricostruire il genoma. Il primo approccio potrebbe essere quello di trovare una stringa, detta *superstringa* $T$ tale che ogni $s_i \in S$ $S = \{s_1, \dots s_n\}$ sia sottostringa di $T$. Praticamente si ha che $T$ è il genoma (con $|T|$ che rappresenta la nostra funzione obiettivo) e le $s_i$ le reads. Si hanno però diverse ripetizioni e grandi regioni di basi ripetute.

**Esempio 3.** *ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long*
***g__time ng__tim***
*ng_time ng_lon* ***__long__*** *a_long long_l ong_ti ong_lo long_t* ***g__long***
*ng_time g_long_ ng_lon a_long long_l* ***ong__ti*** *ong_lo* ***long__t***
*ng_time long_ti g_long_* ***ng__lon*** *a_long long_l* ***ong__lo***
***ng__time*** *ong_lon* ***long__ti*** *g_long_ a_long long_l* ***ong__lon*** *long_time*
*g_long_ a_long* ***long__l*** *long_lon* ***long__time g__long__*** *a_long*
***long__lon g__long__time*** *a_long*
***long__long__time a__long***
*a_long_long_time*

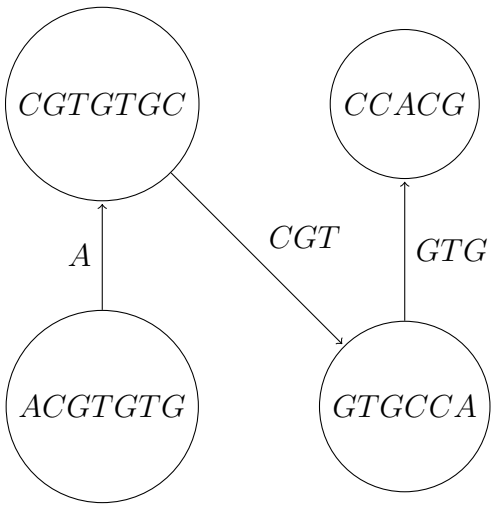
Si procede con un algoritmo greedy fondendo le due stringhe com massimo overleap iterativamente finchè non si ottiene una sola stringa. Questa è una soluzione **NP-hard** oltre al fotto che si hanno regioni estremamente simili che verranno considerate solo una volta.

Si passa quindi ad un ragionamento che sfrutta i grafi. In un **grafo di overleap** i nodi sono rappresentati da dalle read. I nodi sono collegati sse l'overleap è abbastanza lungo e l'arco tra un vertice $A$ e uno $B$ è etichettato con la sottosequenza di $A$ che precede la sovrapposizione. Si tiene conto anche dell'ordine di lettura delle reads. Vediamo un esempio avendo:

$$read = \{ACGTGTG,\ CGTGTGC,\ GTGCCA,\ CCACG\}$$



Dal grafo precedente si passa allo **string graph** rimuovendo gli archi transitivi, ottenendo:




**TSP e Superstringa**

Per inizare ad approcciarci ai grafi vediamo il probolema del **Traveling Salesman Problem, *TSP***.
Dato un grafo orientato $G = (V, A)$ con archi pesati seocndo $w : A \to \mathbb{Q}^+$. Si cerca una permutazione $\Pi = \{\pi_1, \ldots, \pi_n\}$ di $V$ di costo (peso totale degli archi che vengono attraversati) minimo che visiti tutti i nodi e torni al punto di partenza. Si ha la seguente funzione obiettivo:

$$\min z = w(\pi_n, \pi_1) + \sum_{i=1}^{n} w(\pi_i, \pi_{i+1})$$

Si dimostra che questo è un problema **NP-Complete** risolvibile grazie all'hardware.
**Una soluzione è un percorso che tocca ogni città esattamente una volta e torna al punto di partenza**
Torniamo ora al problema originario della superstringa. Ipotizziamo che una read non sia altro che un città del TSP. Però si ha che l'asseblaggio non è il "ciclo" tra le città e la lunghezza della stringa non è il costo del percorso TSP. Le stringhe in ingresso vengono mappate nel grafo di TSP, diventando nodi. Gli archi devono avere peso minimo, quindi diventano la parte della stringa che non è sovrapposta. e si ha quindi:

$$|T| = \sum_{i=1}^{n} |s_i| - \sum_{i=1}^{n-1} |overleap(s_i, s_{i+1})|$$

*Per permettere di tornare all'inizio della stringa dalla fine si usa il carattere $ a fine stringa che viene collegata al suo inizio.*
**Si mappano quindi tutti i possibili cammini e si individua il percorso ottimo.**

### 4.0.2 OLC

L'**OLC (*Overlap, Layout, Consensus*)** è una tecnica che si usa per ridurre il grafo di overleap e si basa su 3 passaggi:

1. **overleap,** dove si calcolano le sovrapposizioni e si costruisce il grafo di overleap. Si usa il suffix array per ottenere un *metodo esatto* oppure la programmazione dinamica (che può comportare errori). Usando il primo metodo si calcola il suffix tree generalizzato di tutte le read *L'albero viene visitato carattere per carattere, usando la read come pattern, e cercando tutti i nodi da cui esce un simbolo di terminazione che non sia l'ultimo.*
   Usando invece la programmazione dinamica si definisce un problema che prende in input due stringhe $s$ e $t$ si cercano i suffissi $x$ di $s$ e $y$ di $t$ tali che si abbia:

   $$\max\{|x| + |y| - 2edit(x, y)\}$$

   dove il $2\cdot$ è stato introdotto in modo arbitrario per dare un maggior peso alle sovrapposizioni. Si confrontano quindi un prefisso e un suffisso e si ha che $m[i, j]$ è l'ottimo del prefisso lungo $j$ di $t$ e del suffisso lungo $i$ di $s$

2. **layout,** dove si fondono i cammini per ottenre i cosiddetti **contigs**, ovvero sottoseuqneze continue. In questa fase vengono rimossi i *branching nodes*, ovvero le ripetizioni e con una sola visita determinare per ogni prefisso se esso sia anche un suffisso

3. **consensus,** dove si calcolano i nucleotidi

### 4.0.3 SBH

Si ha che un *k-mero* è una sottostringa di lunghezza $k$. **SBH (*Sequencing By Hybridation*)** è una vecchia tecnologia che analizza gli *oligonucleotidi* (ovvero sequenze nucleotidiche di un numero di basi da 6 a 10). Per ogni *k-mero*, che viene chiamato **chip**, con $k \approx 8$, si conosce se appare nel genoma. Il processo è chiamato DNA chip perché la logica di fondo è la stessa che

viene usata per i chip, ognuno di essi può tenere migliaia di oligonucleotidi. Trascurando tutta la parte tecnica e passando a quella algoritmica si ha che SBH si differenzia dal grafo di overleap in quanto in *k-meri* hanno tutti la stessa lunghezza (al conbtrario delle reads) e quindi si hanno sovrapposizioni di lunghezza $k-1$

### 4.0.4 Grafi di De Bruijin

Si ha che ogni k-mero viene diviso in *(k-1)-meri*.
In un **grafo di De Bruijin** ogni vertice corrisponde ad un *(k-1)-mero* e un arco corrisponde ad un *k-mero*. Due archi sono collegati sse ci sono sovrapposizioni. Avendo stringhe tutte di lunghezza $k$ che sono presenti in almeno una read si costruisce facilmente il grafo. I *(k-1)-meri* identici vengono eliminati ottenendo unicamente nodi distinti.
Si definisce **cammino Euleriano** un cammino che attraversa ogni nodo una e una sola volta.
Si definisce **ciclo** un cammino che torna al nodo di partenza.
Si definisce **ciclo sempice** un cammino che torna al nodo di partenza passando una e una sola volta per ogni nodo.
Trovando il cammino Euleriano dei grafi di De Bruijin si ottiene il genoma.

**Esempio 4.** *Si abbiano le seguenti reads:*

$$reads = \{ACGTGTG,\ CGTGTGC,\ GTGCCA,\ CCACG\}$$

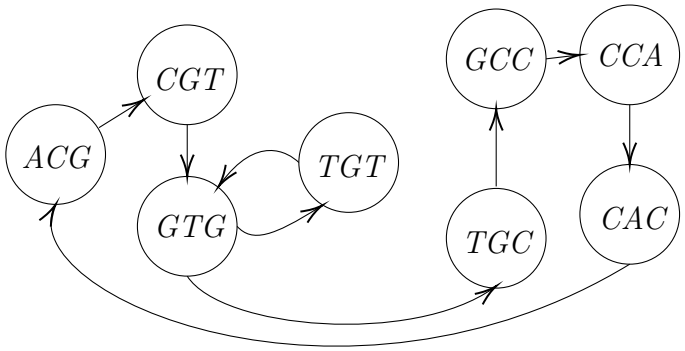
*Scegliamo* $k = 4$ *e otteniamo i 4-meri:*

$ACGT,\ CACG,\ CCAC,\ CGTG,\ GCCA,\ GTGC,\ GTGT,\ TGCC,\ TGTG$

*e i seguenti 3.meri unici:*

$$ACG,\ CAC,\ CCA,\ CGT,\ GCC,\ GTG,\ TGC,\ TGT$$

*e quindi:*



tornando sul discorso dei cammini diamo qualche definizione:

**Definizione 5.**

- ***grafo Semi-Euleriano****: dato un grafo orientato $G = (V, A)$ esso è definitito semi-Euleriano se esistono due vertici $s$ e $t$ tali che:*

$$N_G^-(s) = N_G^+(s) + 1, N_G^-(t) = N_G^+(v) - 1$$

*mentre per ogni altro vertice $w$ si ha che:*

$$N_G^-(w) = N_G^+(w)$$

*Si ottiene che ogni nodo avrà lo stesso numero di archi entranti e uscenti, tranne due nodi, uno che ne avrà uno uscente in più e l'altro uno entrante meno*

- ***grafo Euleriano:*** *dato un grafo orientato $G = (V, A)$ esso è definitito Euleriano se:*

$$N_G^-(w) = N_G^+(w)$$

*ovvero ogni vertice ha lo stesso numero di archi entranti e uscenti.*

**Teorema 1.** *Dato un grafo orientato $G = (V, A)$ e sia $C$ un suo cilo. Sia $G_1$ il grafo senza gli archi di $C$. $G_1$ è Euleriano*

**Teorema 2.** *Un grafo connesso $G = (V, A)$ ha un cammino Euleriano sse $G$ è Semi-Euleriano, inoltre $G$ ha un ciclo Euleriano sse $G$ è Euleriano*

**Teorema 3.** *Dato un grafo Semi-Euleriano $G = (V, A)$ e sia $P$ un cammino tra $s$ e $t$. Sia $G_1$ il grafo ottenuto togliendo a $G$ gli arhci di $P$. Si ha che $G_1$ è Euleriano*

- ***ciclo Euleriano:*** *è un assemblaggio in un grafo orientato e connesso che attraversa ogni* ***arco*** *esattamente una volta. Si possono avere cicli Euleriani semplici*

- ***ciclo Hamiltoniano*** *è un cammino che attraversa ogni* ***vertice*** *esattamente una volta. Non si possono avere cicli Hamiltoniani semplici*

### 4.0.5 Reverse and Complement