

# Appunti Bioinformatica

Antonio Cavuoto

A.A 2019/2020 - Documento Sempre Aggiornato

## Indice

<b>1</b>	<b>Introduzione sulla Biologia Molecolare</b>	<b>5</b>
1.1	NGS . . . . .	7
1.2	DNA . . . . .	8
1.3	RNA . . . . .	10
1.3.1	Dal DNA all'RNA: La trascrizione . . . . .	10
1.3.2	Splicing . . . . .	13
1.4	Proteine . . . . .	14
1.4.1	Il codice genetico . . . . .	17
1.5	Epigenetica . . . . .	17
<b>2</b>	<b>Algoritmi di Pattern Matching</b>	<b>18</b>
2.1	Metodo naif . . . . .	18
2.2	Z Algorithm . . . . .	18
2.2.1	Fase di preprocessing . . . . .	18
2.3	Boyer-Moore . . . . .	22
2.3.1	Regola del "Bad character shift" . . . . .	23
2.3.2	Regola del bad character shift estesa . . . . .	23
2.3.3	Regola dello strong good suffix . . . . .	24
2.3.4	Preprocessing in Boyer-Moore . . . . .	25
2.4	Knuth-Morris-Pratt . . . . .	27
2.4.1	Preprocessing in Knuth-Morris-Pratt . . . . .	27
<b>3</b>	<b>Algoritmi di Motif finding</b>	<b>28</b>
3.1	Metodo Brute Force . . . . .	32
3.2	Problema della stringa mediana . . . . .	32
<b>4</b>	<b>Partial Digest Problem</b>	<b>39</b>
4.1	Metodo Brute Force . . . . .	42
<b>5</b>	<b>Algoritmi di allineamento</b>	<b>46</b>
5.1	Introduzione (Breve) . . . . .	46
5.2	Similitudini . . . . .	47
5.3	Allineamento Globale . . . . .	48
5.3.1	Etichettare gli archi . . . . .	48
5.3.2	Dynamic programming nel problema del resto . . . . .	49
5.3.3	Problema di Manhattan con Dynamic Programming . . . . .	51
5.3.4	Gestione corretta delle indel . . . . .	54
5.4	Allineamento Locale . . . . .	55
5.4.1	Free Taxi Rides . . . . .	56
5.4.2	Penalizzare correttamente le Indel . . . . .	57
5.5	Allineamento Space-Efficient . . . . .	61
5.5.1	Middle Node Problem . . . . .	61
5.6	MultiSequence Alignment . . . . .	65

<b>6</b>	<b>Alberi filogenetici</b>	<b>66</b>
6.1	Struttura . . . . .	66
6.1.1	Matrici di distanza . . . . .	68
6.2	Neighbor-Joining Algorithm . . . . .	69
6.2.1	Additive Phylogeny . . . . .	69
6.2.2	Triple degeneri . . . . .	69
6.2.3	Four point condition . . . . .	72
6.3	UPGMA . . . . .	73
6.4	Character Based Phylogeny . . . . .	75
6.5	Small Parsimony Problem . . . . .	75
6.5.1	Algoritmo di Sankoff . . . . .	78
6.5.2	Algoritmo di Fitch . . . . .	79
6.5.3	Sankoff vs Fitch . . . . .	80
6.6	Large Parsimony Problem . . . . .	81
6.6.1	Formalizzazione del Problema . . . . .	81
6.6.2	Nearest Neighbor Interchange . . . . .	81
6.6.3	Problemi della parsimonia in generale . . . . .	83
<b>7</b>	<b>Algoritmi di Clustering</b>	<b>84</b>
7.1	Algoritmi di partizionamento . . . . .	85
7.1.1	PAM (Partitioning Around Metoids) . . . . .	86
7.1.2	CLARA (Clustering Large Application) . . . . .	87
7.1.3	CLARANS (Clustering Large Applications based upon RANDOMized Search) . . . . .	87
7.2	Algoritmi gerarchici . . . . .	88
7.2.1	AGNES (Agglomerative Nesting) . . . . .	88
7.2.2	DIANA (Divisive Analysis) . . . . .	89
7.2.3	CURE (Clustering Using REpresentatives) . . . . .	89
7.3	Algoritmi basati su densità . . . . .	89
7.3.1	DBSCAN . . . . .	89
7.4	Algoritmi basati su griglia . . . . .	90
7.4.1	CLIQUE . . . . .	90
7.5	Algoritmi basati su Modello . . . . .	91
7.6	Clustering di dati categorici . . . . .	91
7.6.1	Algoritmo ROCK . . . . .	91
7.7	Considerazioni finali sul Clustering . . . . .	92
7.7.1	Gene Ontology ed Implicazioni . . . . .	92
7.8	Co-clustering . . . . .	93
7.8.1	Conoscenze pregresse . . . . .	94
<b>8</b>	<b>Motif Identification</b>	<b>97</b>
8.1	Analisi statistica delle sequenze . . . . .	100
8.2	Incertezza media . . . . .	102
8.3	Motif Detection Algorithm . . . . .	102
8.4	Greedy Motif Search . . . . .	105
8.4.1	Laplace's rule of succession . . . . .	107

---

8.4.2	Rolling dice to find motifs . . . . .	108
8.4.3	Gibbs Sampling . . . . .	108
<b>9</b>	<b>Introduzione al Deep Sequencing</b>	<b>110</b>
9.1	Library Preparation . . . . .	110
9.2	Cluster Amplification . . . . .	111
9.3	Sequencing . . . . .	112
<b>10</b>	<b>Genome Assembly</b>	<b>115</b>
10.1	The Newspaper problem . . . . .	115
10.2	String Composition Problem . . . . .	116
10.3	String Composition as a Graph path . . . . .	117
10.3.1	Hamiltonian Path . . . . .	117
10.3.2	Eulerian path in De Bruijn Graph . . . . .	118
10.3.3	Unique solution with paired end reads . . . . .	121
<b>11</b>	<b>Combinatorial Pattern Matching</b>	<b>123</b>
11.1	Trie . . . . .	123
11.2	Suffix Tree . . . . .	125
11.3	Burrows-Wheeler . . . . .	126
11.4	Reversing BWT . . . . .	128
11.5	Finding Matched . . . . .	131
11.5.1	Suffix Array . . . . .	131
11.5.2	Moving Backward through a pattern . . . . .	132
11.6	Epilogo . . . . .	138
11.6.1	Pattern Matching Inesatto . . . . .	138
<b>12</b>	<b>Conclusione</b>	<b>139</b>

## 1 Introduzione sulla Biologia Molecolare

Partiamo dalla definizione di Bioinformatica:

*“Il termine BIOINFORMATICA è stato coniato per indicare l'applicazione di tecniche informatiche nel dominio applicativo delle scienze della vita”*

Il motivo per il quale si studia Biologia Molecolare è perchè ormai si è compreso che non basta studiare i singoli geni, bensì come questi interagiscono tra loro.

In particolare:

- La Biologia Computazionale pone l'accento sugli aspetti algoritmici dei problemi biologici e sull'efficienza delle loro soluzioni
- La Biologia dei Sistemi è un approccio introdotto recentemente che si basa sull'utilizzo della teoria dei sistemi per studiare fenomeni biologici
- Il DNA Computing è una forma di computazione che usa il DNA e la biologia molecolare al posto delle tecnologie computazionali tradizionali (silico-based)

Le informazioni rilevanti per il nostro corso, sostanzialmente girano attorno al sequenziamento del genoma umano e le sue implicazioni. Nei primi anni 2000 siamo riusciti a sequenziare totalmente il genoma umano, che si è scoperto essere lungo 3.4 Gb (dove la b non rappresenta i Byte, ma le basi). Per fare un confronto, l'orzo possiede un genoma da 5.5 Gb ma a livello di complessità siamo a livelli molto inferiori rispetto a quello umano. Di conseguenza se ne deduce che la complessità non è rapportata alla lunghezza del genoma.



Figura 1: Copertine di riviste scientifiche

I biologi sono poi arrivati ad una conclusione importante: c'è una sorta di legge naturale che per forza si deve verificare, una sorta di assioma. Partendo dai

singoli geni, si compone un mRNA (trascrizione) che poi sarà necessario alla produzione delle proteine (traduzione).

Il ruolo dell'informatica in tutto questo è capire la funzione biochimica delle nuove proteine utilizzando algoritmi specifici sulle sequenze di geni. Un'altra cosa interessante da dire è che un insieme di geni può determinare la nascita di proteine diverse (per cui non c'è corrispondenza 1:1). Questo complica di molto il lavoro. Inoltre, ad oggi non si conoscono strumenti sperimentali per andare a studiare le interazioni tra proteine. Quello che si fa è andare a livello più basso (geni e RNA) e osservare come reagiscono le proteine alle varie modifiche. Un esempio di tecnologia per lo studio dei geni è il microarray:

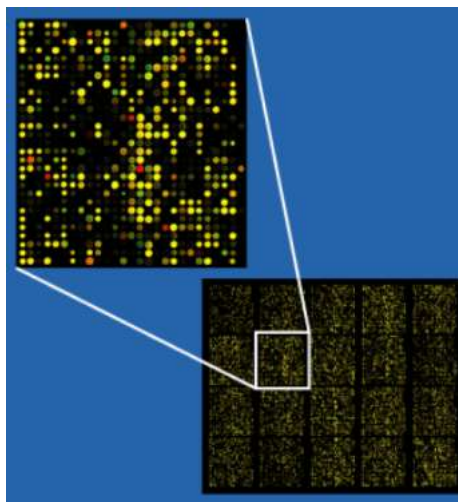


Figura 2: Struttura di un microarray

Attraverso questo chip di silicio è possibile studiare in una volta sola le interazioni di multiple combinazioni di geni (a patto di impilarli in modo corretto in una struttura verticale sulla sua superficie).

In pochi anni l'evoluzione delle tecnologie sperimentali ha portato l'aumento dei dati prodotti a seguito degli esperimenti, per cui è stato necessario creare sistemi di trattamento ed organizzazione dei dati (high-throughput). Un esempio è la heat map<sup>1</sup>.

Con lo sviluppo dell'informatica è cambiato il paradigma della ricerca della Biologia. Utilizzando modelli matematici le simulazioni al computer hanno consentito maggiore comprensione e determinismo dei sistemi biologici e delle loro interazioni. Innovazione dopo innovazione siamo giunti ai giorni nostri, con la Next Generation Sequencing (NGS).

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Heat\\_map](https://en.wikipedia.org/wiki/Heat_map)

## 1.1 Next Generation Sequencing

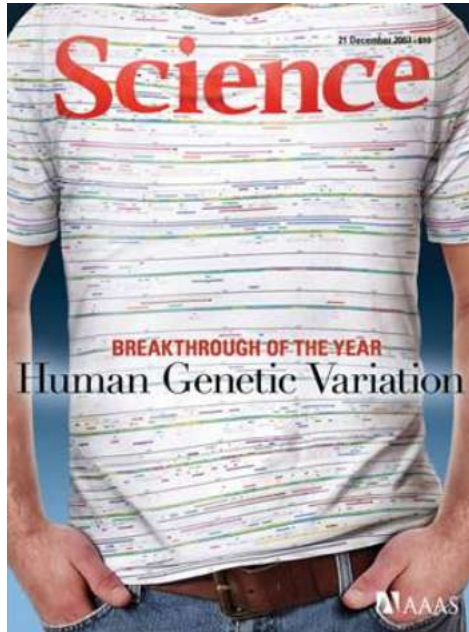


Figura 3: Copertina di Science

La tecnologia attuale consente perfino di arrivare alla conservazione di informazioni attraverso del DNA. Se pensiamo che 3.4 Gb sono contenuti in 3nm, allora nello spazio di una microSD attuale (15x11x1mm) potremmo stipare una quantità enorme di informazione 62.33 Zettabyte ( $62.33 \times 10^{21}$  bytes). Per avere un'idea molto generica, nel 2014 si stimava che l'intera dimensione della rete Internet fosse di circa 1000 Zettabyte. In altre parole, con 17 microSD avremmo l'intera rete Internet in tasca (ed anche qualcosa in più). Se usassimo genomi più grandi si potrebbero addirittura migliorare questi numeri (a parità di spazio occupato).

Nella prossima sezione andiamo a spiegare un po' come funzione la biologia. Per semplicità ci soffermeremo solo su ciò che riguarda gli aspetti che vedremo noi (geni, trascrizione e traduzione).

## 1.2 DNA

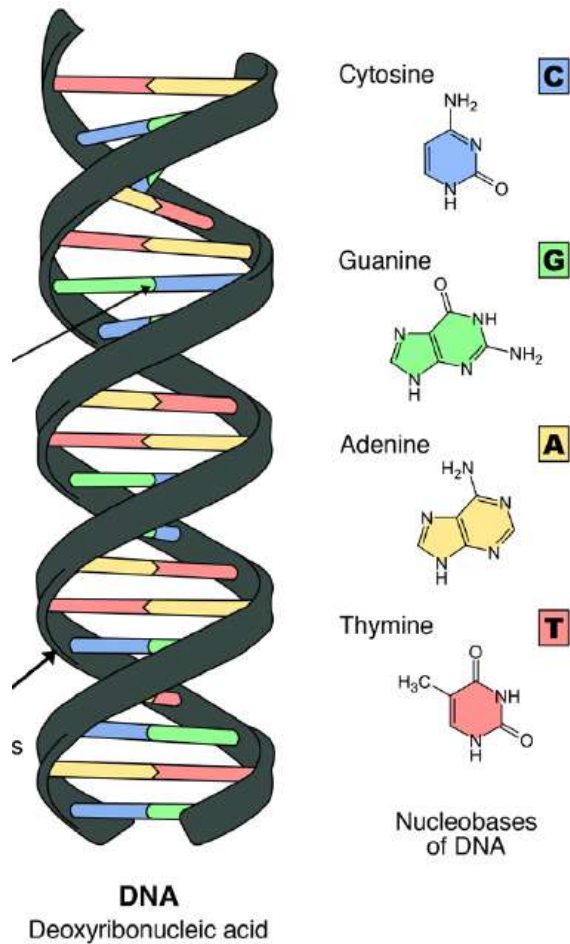


Figura 4: Descrizione schematica del DNA

Il DNA è una singola molecola che contiene tutte le informazioni necessarie per sintetizzare le proteine in ognuna delle nostre cellule.

Ha la forma di una scala di corda attorcigliata su se stessa. Due lati della scala fungono da spina dorsale dei singoli filamenti di DNA, ciascuno dei quali è un lungo polimero. Le sottounità che si ripetono compresi i polimeri sono chiamati nucleotidi. Ogni nucleotide è costituito da 3 parti:

1. Uno zucchero formato da 5 atomi di carbonio.
2. Una base azotata (Riportate in colonna a destra della figura)
3. Uno o più fosfati



Possiamo vedere il DNA come una stringa, appartenente all'alfabeto del DNA  $A_{DNA} = \{A, C, G, T\}$  dove i membri dell'insieme sono i 4 in figura: Adenina, Citosina, Guanina e Timina.

Il DNA è autocomplementare, nel senso che la doppia elica si tiene in piedi quando si accoppiano le basi azotate. Esiste infatti una complementarità bene precisa tra le basi (dipendente dal numero di legami a idrogeno che esse formano unendosi, in una logica di “il simile si accoppia al simile”). Questa regola sostanzialmente è la seguente :  $\{A \rightarrow T, G \rightarrow C\}$  (e viceversa naturalmente). Inoltre, il DNA viene letto sempre dal 5' al 3'.

La funzione del DNA è quella di trasportare i geni, l'informazione che specifica tutto l'insieme di proteine che costituiscono un'organismo, con aggiunte informazioni di controllo (quante proteine produrre in quale tipo di cellule). Nella cellula eucariote il genoma è sostanzialmente raggruppato nei cosiddetti cromosomi<sup>2</sup>. Un'immagine esplicativa può essere la seguente:

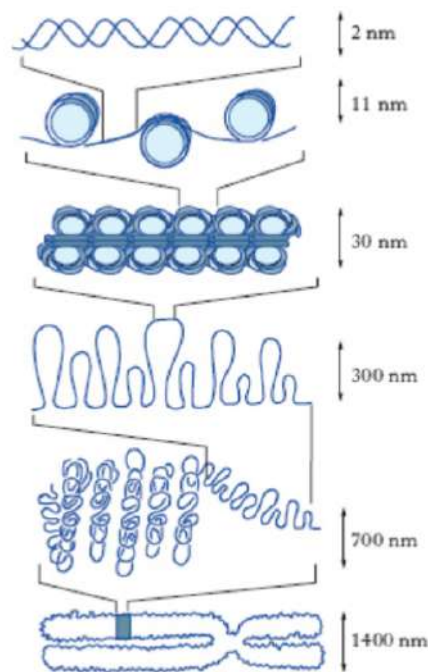


Figura 5: DNA Packing

A questo punto è arrivato il momento di parlare di come viene utilizzato il DNA per produrre l' mRNA. Per questo motivo andremo nella prossima sezione ad approfondire questo aspetto.

<sup>2</sup><https://en.wikipedia.org/wiki/Chromosome>

### 1.3 RNA

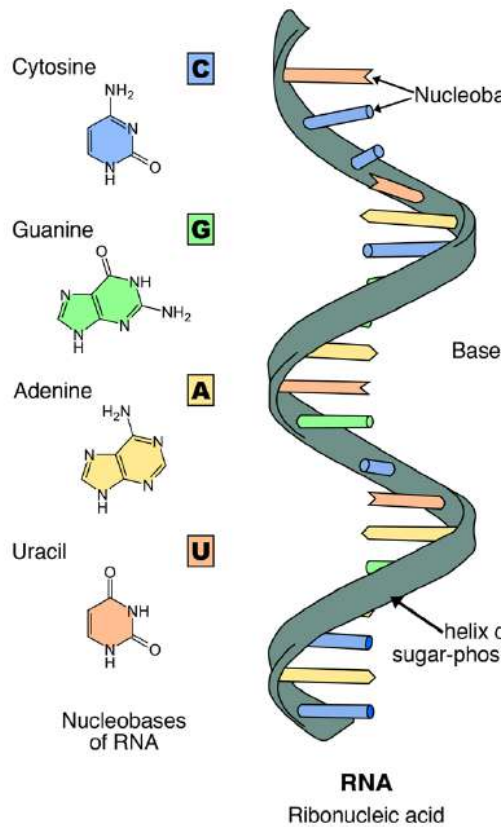


Figura 6: Descrizione schematica dell'RNA

L'RNA ha un suo alfabeto particolare: è uguale a quello del DNA ma la Timina viene sostituita da un Uracile :  $A_{DNA} = \{A, C, G, U\}$ . Inoltre, l'RNA è letto a triplette di basi. A differenza del DNA, l'RNA è composto da una singola elica. L'RNA viene prodotto a partire dal DNA e dai geni mediante l'enzima RNA polimerasi, ma facciamo un passo indietro e cerchiamo di tenere un filo logico del discorso più rigoroso.

#### 1.3.1 Dal DNA all'RNA: La trascrizione

Ogni gene è un'unità fondamentale di eredità. Un gene è una stringa di basi, ed ogni molecola di DNA contiene più di 10000 geni diversi. Il DNA non partecipa attivamente alla sintesi delle proteine, bensì c'è bisogno di costruire un intermediario, l'mRNA (RNA messaggero). Ogni volta che c'è bisogno di costruire una proteina viene creato un opportuno RNA che viene usato come

base per attivare tutta quella macchina che ha il compito di produrre proteine. Ogni mRNA ha un'emivita limitata, questo per evitare che la cellula produca solo un tipo di proteina in loop. Il processo che dal DNA crea un RNA è detto Trascrizione, che avviene con tempi e metodologie ben specifici. Ma prima di descrivere questo procedimento è bene capire come avviene la "Fase preparatoria".

Essendo l'RNA sostanzialmente una copia temporanea del DNA c'è bisogno di fare una lettura ed una conseguente scrittura (per usare un gergo informatico). L'unico problema è che il DNA è impacchettato, ovvero non è un filo rettilineo. Di conseguenza la molecola di DNA va prima divisa nelle sue due eliche, va scelto il filamento da seguire ( $5' \rightarrow 3'$ ) e va raddrizzato per bene. All'inizio il DNA si trova raccolto da strutture particolari note come Istoni:

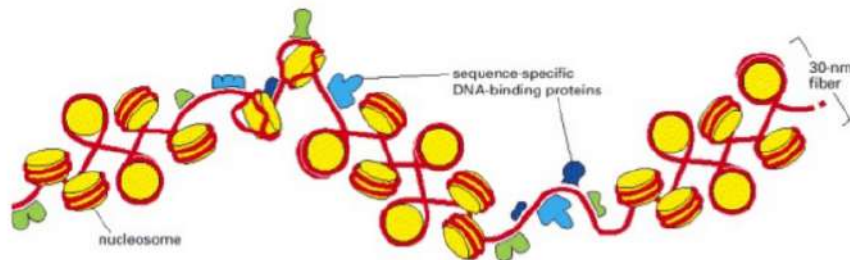


Figura 7: Rappresentazione grafica Istoni (i dischi in giallo)

Successivamente c'è bisogno di individuare il gene da cui partire per la lettura. Questo compito è svolto da particolari proteine che individuano la cosiddetta TATA box, ovvero una sequenza di partenza (TATA) dalla quale 25 nucleotidi dopo inizia il gene da leggere. A partire dalla TATA box avviene un richiamo di proteine fin quando non arriva nel sito l'enzima RNA polimerasi, che fisicamente si ancora al punto determinato ed inizia a "srotolare" e sintetizzare l'RNA vero e proprio. Qui di seguito viene riportato un cartoon che spiega proprio questi passaggi.

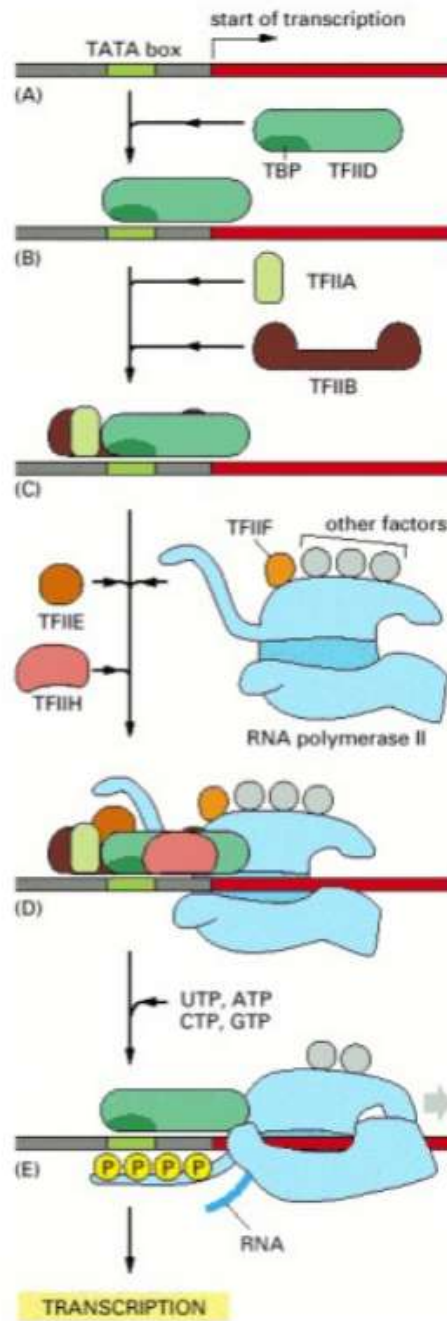


Figura 8: Il processo di trascrizione

Ora, ci sono vari tipi di DNA in base alla forma tridimensionale che può assumere:

1. mRNA : è responsabile del trasporto delle informazioni dei codoni ai ribosomi, che sono il sito dove avviene la sintesi proteica.
2. tRNA : ha fondamentalmente due ruoli: trasporta gli amminoacidi nei ribosomi e fornisce ai codoni i rispettivi anticodoni.
3. rRNA : serve a costruire una base sulla quale il ribosoma può agganciarsi per iniziare il processo di sintesi.
4. spRNA: derivato da reazioni di splicing.

Purtroppo l'RNA così come esce dai punti precedenti non può ancora svolgere il suo ruolo biologico: infatti il DNA presenta molte sequenze inutili e ridondanti (non codificanti) che per ovvie ragioni vengono copiate insieme alle informazioni rilevanti. Per questo motivo è necessario separare la parte buona da quella inutile. In particolare, le sequenze codificanti sono chiamate esoni, mentre quelle interposte sono dette introni. La separazione tra esoni e introni è un processo che prende il nome di Splicing.

### 1.3.2 Splicing

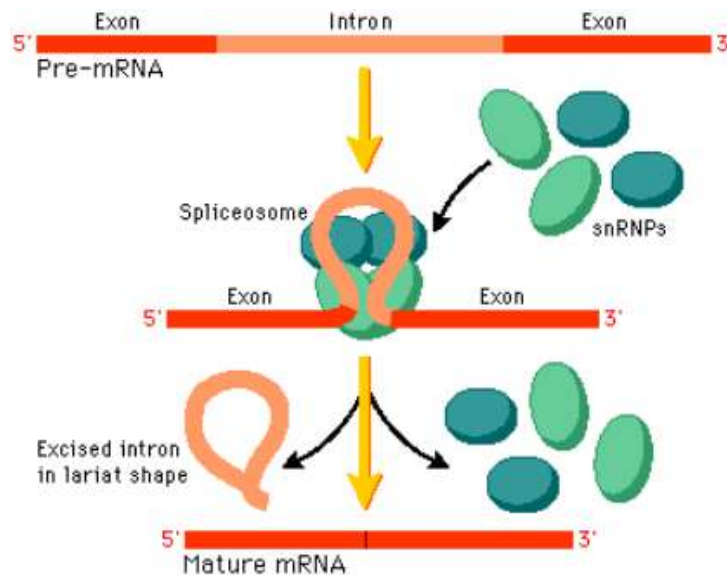


Figura 9: Il processo di splicing

Non penso ci sia nulla da aggiungere rispetto all'immagine precedente e a quello che abbiamo detto prima. C'è però un fattore che complica non poco tutta la

trattazione: non esiste un solo modo per fare lo splicing. Anzi, spesso qualche introne rimane lo stesso. Inoltre, gli stessi esoni possono essere alternati in modo diverso, per cui non abbiamo particolari possibilità d'azione se non metterci lì e vedere un po' a cosa corrisponde ogni ordine parziale.

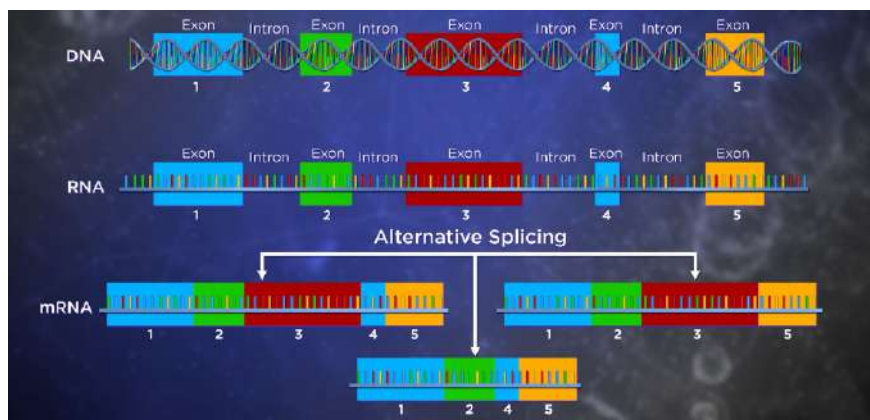


Figura 10: Alternative splicing

Bene, con questo abbiamo terminato il processo di trascrizione. Nella prossima sezione andiamo a vedere come a partire da un mRNA possiamo giungere alla produzione delle proteine.

## 1.4 Proteine

Le proteine sono entità molto più complesse dei singoli geni, basti pensare che gli acidi nucleici hanno 4 blocchi di base mentre le proteine hanno 20 amminoacidi di base, ciascuno dei quali ha funzioni particolari. Ogni ordine nella sequenza amminoacidica determina una forma tridimensionale complessiva diversa. Abbiamo infatti un tRNA per ogni amminoacido, in modo che per ognuno si disponga della “chiave di codifica”. Il codice genetico nel mRNA è scritto in codoni, 3 basi specifiche per ogni amminoacido. Invece nel tRNA viene usato l'anticodone per codificare l'informazione. Codoni e anticodoni sono sequenze complementari, il che vuol dire che per ogni codone ci può essere un unico anticodone che completa l'informazione. La complementarità è quella del DNA con una piccola modifica :  $\{A \rightarrow U, G \rightarrow C\}$  (perchè la Timina è stata sostituita da un Uracile). Nella pagina seguente ho inserito un'immagine utile a rappresentare almeno visivamente la situazione.

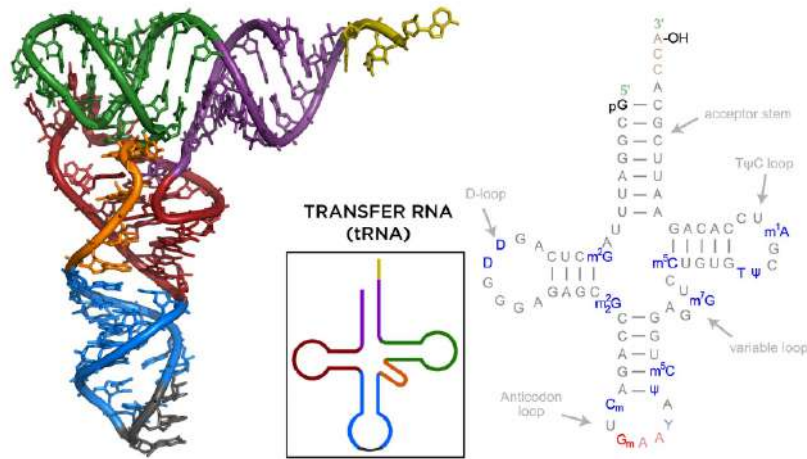


Figura 11: Esempio di tRNA di codifica

Ora, il processo che da RNA porta alla formazione delle proteine è detto Traduzione. Andiamo ora a parlarne più nel dettaglio:

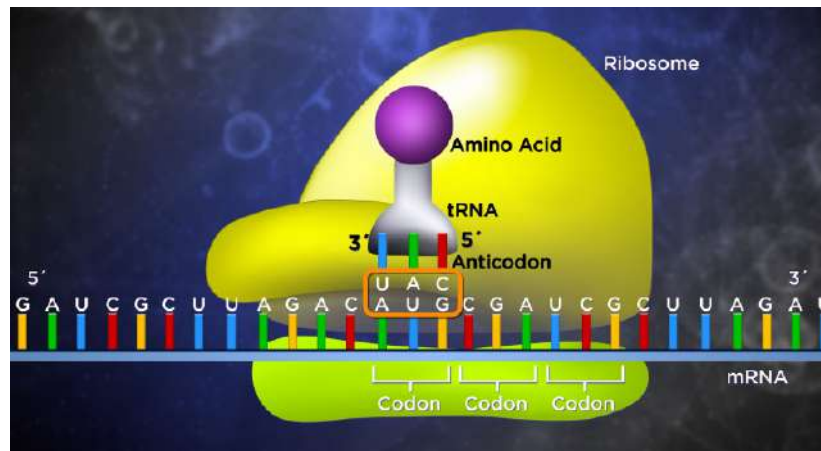


Figura 12: Processo di Traduzione

La sintesi proteica avviene in ogni cellula all'interno dei ribosomi. In questi ultimi inizia a scorrere l'mRNA, e si alternano i tRNA che cercano di fissarsi ad un codone complementare al loro anticodone. In caso positivo, l'amminoacido in coda del tRNA viene aggiunto alla proteina in fase di costruzione. L'immagine seguente raffigura tutto il procedimento:



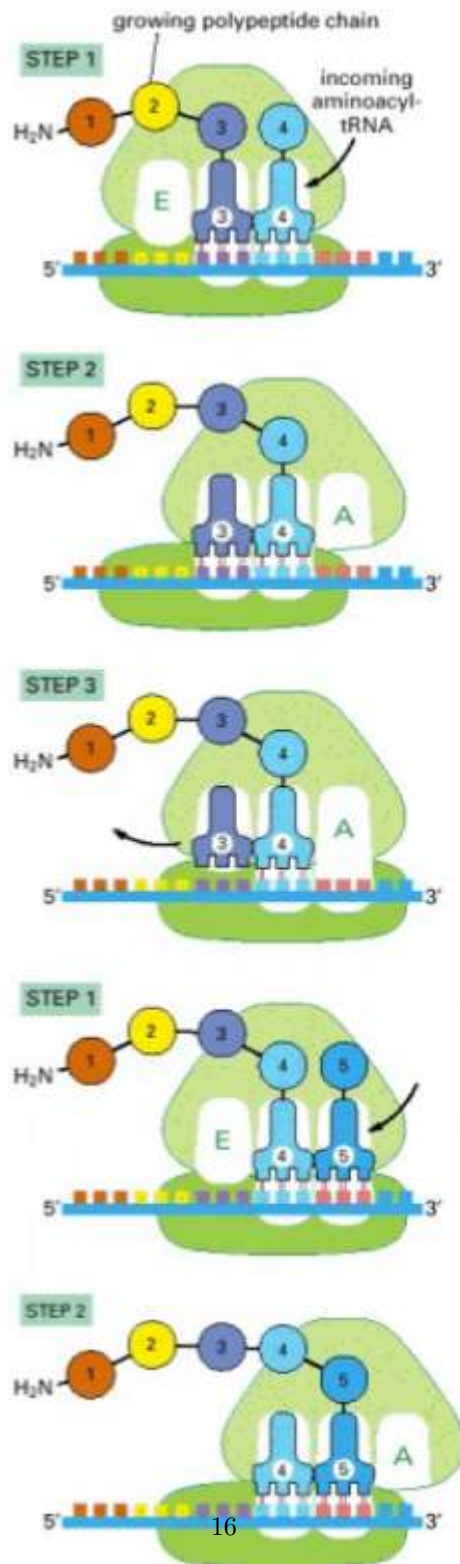


Figura 13: Processo di Traduzione in sequenza



### 1.4.1 Il codice genetico

Per codificare 20 amminoacidi non sono sufficienti due nucleotidi ( $4^2 = 16$ ), ma tre sono troppi ( $4^3 = 64$ ). Di conseguenza, sequenze diverse possono codificare lo stesso amminoacido. Come se non fosse già abbastanza complicato, può capitare che certi amminoacidi possano essere sostituiti senza cambiare sostanzialmente la struttura della proteina. Questo apre a tutto un mondo di errori possibili (che per semplicità non tratteremo).

Finita questa parte andiamo brevemente a fare un cenno di ciò che si sta facendo in questi anni.

## 1.5 Epigenetica

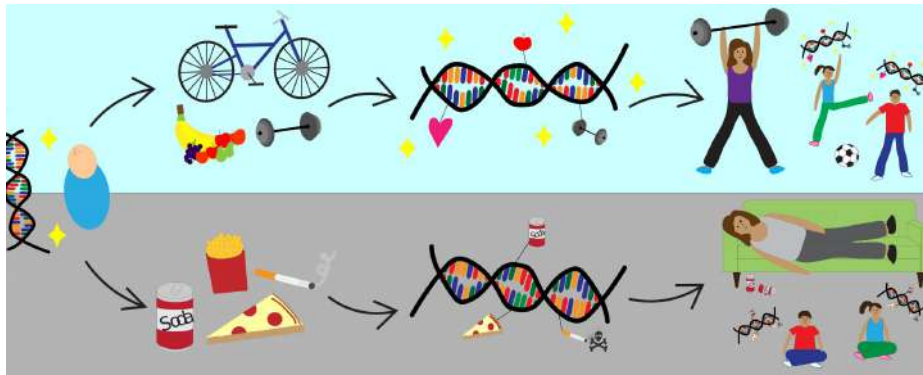


Figura 14: Processo di Traduzione in sequenza

L'epigenetica è il ciclo di feedback tra i nostri geni e l'ambiente circostante. I geni all'interno delle nostre cellule sono in continuo aggiornamento dato che devono fronteggiare sempre nuove minacce. Fornendo ai nostri geni input positivi (stile di vita incluso), possiamo ottenere salute e longevità, e lasciare questi marker positivi per la prossima generazione.

Al giorno d'oggi, la biologia dei sistemi si sovrappone allo studio delle reti biologiche nella misura in cui l'obiettivo è diventato quello di sviluppare una comprensione quantitativa delle funzioni biologiche dei geni all'interno delle reti biologiche.

## 2 Algoritmi di Pattern Matching

In questa sezione andiamo a descrivere le metodologie per cercare sostanzialmente stringhe in altre stringhe più grandi.

### 2.1 Metodo naif

Il metodo naif è il classico confronto di un pattern  $P$  in una stringa  $T$  per ogni carattere di  $T$ . Per esempio, consideriamo la stringa  $T = \text{"aaaaaaaaa"}$  e il pattern  $P = \text{"aaa"}$ .

Allora con il metodo naif dovremmo effettuare 24 confronti (esattamente  $|P| \cdot (|T| - (|T| - |P| - 1))$ ), nel nostro caso  $3 \cdot (10 - (3 - 1)) = 3 \cdot 8 = 24$  confronti. Parlando di complessità, è chiaro che siamo nell'ordine di  $\mathcal{O}(n \cdot m)$  per cui vorremmo provare a fare di meglio. Per fare questo l'idea più semplice è quella di tentare di eliminare i confronti inutili, così da poter avvicinarsi ad una complessità in termini di confronti di  $\mathcal{O}(n)$ , quindi lineare.

Molti algoritmi quindi separano l'operazione di pattern matching in 2 fasi distinte:

1. Fase di preprocessing: si cerca di imparare qualcosa di utile sulla struttura del pattern o del testo.
2. Fase di ricerca: con le informazioni raccolte nel precedente step si effettua la ricerca vera e propria evitando il più possibile di fare confronti superflui.

### 2.2 Z Algorithm

#### 2.2.1 Fase di preprocessing

Quello che studieremo noi è il calcolo dei valori  $Z_i$ , ovvero per ogni carattere in posizione  $i > 1$ , troveremo il numero di caratteri che fanno match con un prefisso della stringa  $T$ .

Esempio: Calcoliamo  $Z_5$  della stringa  $S = \text{AABCAABXAA}$ . Per farlo, posizioniamoci al carattere  $i$  in posizione 5 (iniziamo da 1 la numerazione)

AABC**A**ABXAA

A questo punto scorriamo a destra sia i caratteri dopo  $i$ , sia i caratteri iniziali della stringa, fin quando non ne troviamo uno che fa mismatch. In steps :

```

1: AABCAABXAA
2: AABCAABXAA
3: AABCAABXAA
4: AABCAABXAA

```

Per cui concludiamo che la più lunga sottostringa che matcha un prefisso è  $AAB$  di lunghezza 3. Abbiamo determinato quindi  $Z_5 = 3$ . Gli  $Z_i$  sono calcolabili per ogni  $i > 1$ , per cui diremo che  $Z_1 = \#$ .

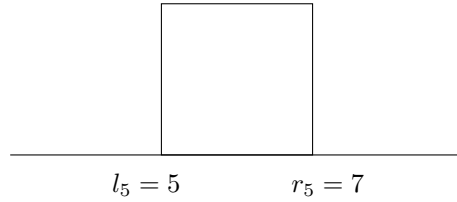
Ora, facciamo un passo in avanti ed occupiamoci di definire il cosiddetto Z-box. Uno Z-box non è nient'altro che un intervallo di una stringa che inizia in posizione  $i$  e finisce in posizione  $i + Z_i - 1$ . Costruiamo un esempio visivo sulla stringa precedente e prendiamo come esempio  $i = 5$ :

$$Zbox_{i=5} = AABC\textcolor{brown}{A}B\textcolor{brown}{X}AA$$

Definiamo ora  $l$  ed  $r$ , rispettivamente come il limite inferiore e superiore dell'intervallo chiuso  $[i, i + Z_i - 1]$ , per cui nel nostro caso lo Z-box individuato sarà l'intervallo chiuso  $[5, 7]$ . Formalmente diciamo che :

$$l_5 = 5 \text{ ed } r_5 = 7$$

In una forma più grafica:



È chiaro che la lunghezza della Z-box corrisponda a  $r_i - l_i$ . Ora, è essenziale che il calcolo di tutti gli  $Z_i$  venga effettuato in tempo  $\mathcal{O}(n)$ , perchè ora come ora la procedura che abbiamo indicato ha complessità  $\theta(|S|^2)$ . Fortunatamente esiste un algoritmo che riesce a svolgere questo calcolo in tempo  $\mathcal{O}(|S|)$ .

Questo algoritmo inizia dalla posizione  $i = 2$ . I valori calcolati vengono ricordati, ma in ogni iterazione  $i$  l'algoritmo ha bisogno solo dei valori di  $l_{i-1}$  e di  $r_{i-1}$ .

**ATTENZIONE: ESEMPIO SBAGLIATO IN ATTESA DI CHIARIMENTO!!!**

Facciamo un esempio con  $k = 121$  e con  $l_{120}$  e  $r_{120}$  che valgono rispettivamente 120 e 150:

Allora possiamo dedurre il valore di  $Z_{120}$ , infatti:

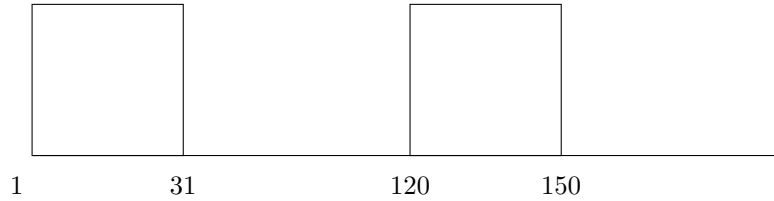
$$r_i = i + Z_i - 1 \text{ (per definizione)}$$

Di conseguenza nel nostro caso di esempio avremo che:

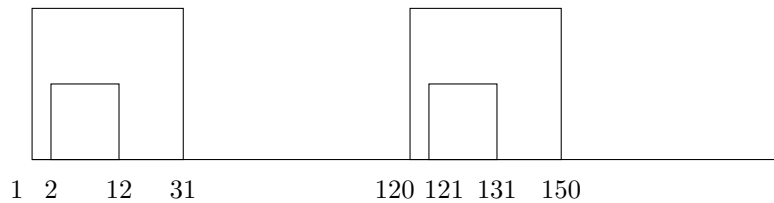
$$Z_i = r_i - i + 1 \text{ ovvero } Z_{120} = 150 - 120 + 1 = 31$$

Allora, se supponiamo  $Z_{121} = 10$ , esisterà necessariamente una sottostringa di S che fa match con un prefisso di S esattamente lungo 10. Quindi stiamo dicendo che la sottostringa che parte dalla posizione 2 di S e che è lunga 10 sarà uguale alla sottostringa che parte da k e arriva a k+10.

Di conseguenza  $Z_2$  è utile per calcolare  $Z_{121}$  perchè avranno lo stesso valore. Graficamente, rappresentiamo prima cosa vuol dire che  $Z_{120} = 31$ :



Possiamo notare che se  $Z_{120} = 31$ , allora i primi 31 caratteri di S devono necessariamente fare match con la sottostringa  $S[120, 150]$ . Ora, supponiamo che  $Z_{121} = 10$ . Mantenendo il disegno di prima andiamo a rappresentare graficamente ciò che questo comporta :



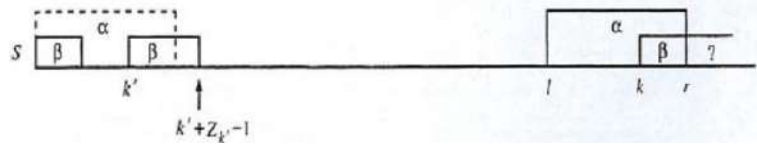
Da questo disegno capiamo subito che se  $Z_2 = 10$  allora anche  $Z_{121} = 10$ .

**FINE ESEMPIO**

In pratica, come trovo  $Z_k$  una volta noti tutti i precedenti  $Z_i, l_i, r_i$ ?  
Abbiamo 2 casi:

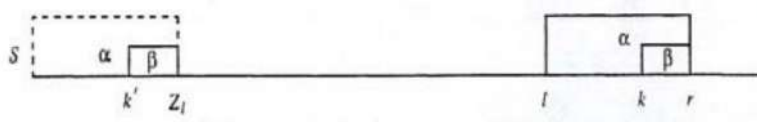
1.  $k > r$  : allora non ho una scorciatoia utile per saltarmi il confronto con l'inizio della stringa  $S$  perchè  $k$  sta fuori da un Zbox noto. Procedo quindi a calcolare  $Z_k$  normalmente, per cui se ad esempio  $Z_k = 10$  e  $k = 100$ , allora  $l_k = 100$  ed  $r_k = 100 + 10 - 1 = 109$ .
2.  $k < r$  : allora  $k$  si trova in uno Zbox precedente. Qui abbiamo nuovamente 2 casi ovvero che lo Zbox successivo sia interno a quello attuale oppure che si estenda oltre.

(a) Lo zbox successivo non è contenuto in quello attuale:



Allora diciamo che il nuovo Zbox sforerà di un valore  $q$ . Avremo quindi:  $Z_k = q - k$ ,  $l = k$ ,  $r = k - q - k - 1 = q - 1$

(b) Lo zbox successivo è contenuto in quello attuale:



Di conseguenza il carattere  $S(k)$  compare anche in posizione  $k' = k - l + 1$ . Mi basta andare a vedere quanto valeva  $Z_{k'}$  e posso ottenere gratis il valore di  $r_{k'}$ . Non avendo rilevato nessun problema posso dire che  $Z_k = Z_{k'}$  e lascio inalterati  $l$  ed  $r$ .

A corredo della spiegazione, è importante fornire i seguenti Teoremi e Corollari (banalmente perchè basta citarli al momento del bisogno in sede d'esame senza doverli dimostrare):

**Teorema 1.4.1 :** Usando lo Z-algorithm il valore  $Z_k$  viene calcolato correttamente ed i valori  $l$  ed  $r$  vengono correttamente aggiornati.

**Corollario 1.4.1 :** Usando lo Z-algorithm per valori  $i > 2$ , vengono calcolati correttamente tutti gli  $Z_i$

Questo preprocessing da solo ci fornisce un semplice algoritmo di matching in tempo lineare:

- Si consideri  $S = P\$T$  dove  $\$$  è un carattere che non appartiene nè al testo nè al pattern. La sua lunghezza è dunque  $n + m + 1$ , di fatto  $\mathcal{O}(m)$ .
- Calcola  $Z_i(S)$  per  $i$  che va da 2 a  $n + m + 1$
- Poichè  $\$$  non compare da nessuna parte, la massima lunghezza per un qualsiasi  $Z_i$  deve essere esattamente pari ad  $n$ , ovvero proprio la lunghezza del nostro pattern.
- Di conseguenza, ogni  $Z_i = n$  corrisponde ad un'occorrenza trovata di  $P$  in  $T$ .

Una caratteristica interessante è che l'algoritmo mantiene una complessità lineare indipendentemente dall'alfabeto considerato.

## 2.3 Boyer-Moore

L'algoritmo di Boyer-Moore così come nell'algoritmo naif allinea  $P$  con  $T$  e poi verifica i match.  $P$  viene spostato a destra ma utilizza delle idee più furbe :

- Scansione da destra verso sinistra
- Regola di shift del “bad character”
- Regola di shift del “good suffix”

La combinazione di queste idee porta a risparmiare un numero rilevante di confronti pur continuando ad essere lineare nel caso peggiore (“Tanto meglio”, cit. Prof).

Consideriamo il testo  $T = xpbctbxabpqxctbpq$  ed il pattern  $P = tpabxab$  con il seguente posizionamento:

XPBCTBXABPQXCTBPQ  
TPABXAB

L'algoritmo di Boyer-Moore ci dice che dobbiamo partire a fare il matching dei caratteri da destra verso sinistra rispetto al pattern, in questo modo:

XPBCTBXABPQXCTBPQ  
TPABXAB

XPBCTBXABPQXCTBPQ  
TPABXAB

XPBCTBXABPQXCTBPQ  
TPABXAB

E così via, fino ad arrivare al primo mismatch che avviene quando confronto  $T(5)$  con  $P(3)$

```
XPBCTBXABPQXCTBPQ
TPABXAB
```

Arrivati a questo punto l'idea è quella di spostare a destra  $P$  fin quando non metto in colonna il carattere che ha creato mismatch con una sua occorrenza nel pattern. Nel nostro esempio lo spostamento corretto sarebbe :

```
XPBCTBXABPQXCTBPQ
  TPABXAB
```

Come possiamo notare abbiamo spostato tutto il pattern direttamente di 2, evitando parecchi confronti. Tuttavia, per essere in condizione di fare ciò, bisogna sapere qual'è di volta in volta il carattere uguale ad  $x$  più a destra (partendo da sinistra) del pattern, in modo da poter spostare tutto di quella distanza. Chiameremo  $R(x)$  la posizione dell'occorrenza più a destra del carattere  $x$ , che può anche valere zero, se non ci sono occorrenze di quel carattere.

### 2.3.1 Regola del “Bad character shift”

Formalmente, una volta trovato il mismatch faccio i seguenti passi:

- $x$  è il carattere di  $T$  che non matcha col carattere  $y$  di  $P$ .
- Cerco in  $P$  partendo da sinistra l'occorrenza di  $x$  più a destra e la sua distanza da  $y$ .
- Sposto a destra tutto il pattern del valore che ho ottenuto prima.
- Nel caso in cui  $R(x) = 0$  allora torno al comportamento naif e sposto a destra solo di 1.

Questa regola è un'utile euristica quando il mismatch è verso l'estremo destro di  $P$ . Non ha effetto se a mismatch avvenuto, a sinistra di quel punto non ci sono occorrenze di  $x$  in  $P$ . Risulta molto efficace in pratica, specialmente in testi come l'inglese.

La sua pecca invece è sfortunatamente proprio gli alfabeti a cardinalità bassa e con tante ripetizioni nelle stringhe, come il DNA e le proteine (rispettivamente 4 e 20 simboli).

### 2.3.2 Regola del bad character shift estesa

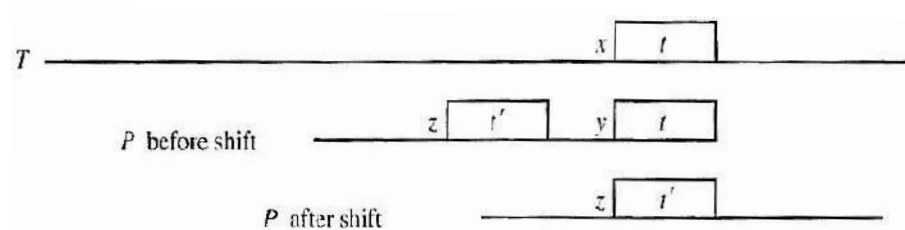
Molto brevemente, la differenza con la regola “classica” è questa :

Nella regola “classica” cercavamo l'occorrenza di  $x$  partendo dalla sinistra di  $P$  e cercando l'occorrenza più a destra. Con la regola estesa invece partiamo direttamente dal punto dov'è avvenuto il mismatch e andando verso sinistra ci

fermiamo alla prima istanza di  $x$  che troviamo, supponiamo alla distanza  $d$ . A questo punto spostiamo a destra  $P$  di  $d$ . Inoltre, scendendo da destra verso sinistra possiamo per ogni carattere che incontriamo, memorizzare la distanza con le occorrenze di tutti i caratteri a sinistra, rendendo le operazioni a complessità  $\mathcal{O}(n)$ .

### 2.3.3 Regola dello strong good suffix

L'immagine è autoesplicativa:



Il procedimento è semplice: se  $t$  è una sottostringa che matcha un suffisso di  $P$  e tale per cui il successivo confronto causa un mismatch, posso cercare nel pattern se esiste un'occorrenza  $t'$  della stringa  $t$  avente come carattere subito a sinistra un carattere diverso da quello che ha causato il mismatch. Nel caso, l'occorrenza di  $t'$  non deve essere un suffisso di  $P$ .

Esempio :

```

PRSTABSTUBABVQXRST
Q CABDABDAB

PRSTABSTUBABVQXRST
Q CABDABDAB

```

Nota: nell'esempio non è stata scelta per lo shift la stringa **DAB** in quanto suffisso di  $P$ .

Anche qui abbiamo un teorema da segnalare:

**Teorema 2.2.1 :** L'uso della good suffix rule non sposta mai  $P$  oltre un'occorrenza in  $T$ .



### 2.3.4 Preprocessing in Boyer-Moore

Bisogna definire i concetti di  $L(i)$  ed  $L'(i)$ :

- Definiamo  $L(i)$  come la posizione del carattere più a destra della copia più a destra della sottostringa  $P[i, \dots, n]$  che è suffisso di  $P$ .

$$\begin{array}{c} \text{CABDABDAB} \\ L(8) = \text{CABDABDAB} \\ \text{DAB} = 6 \end{array}$$

Nell'esempio la stringa subito a sinistra che fosse anche suffisso di **DAB** è proprio **DAB**. Di essa prendo la posizione dell'ultimo carattere, ovvero la **B**, quindi  $L(8) = 6$ .

- Definiamo  $L'(i)$  come la posizione del carattere più a destra della copia più a destra della sottostringa  $P[i, \dots, n]$  che però NON è suffisso di  $P$ .

$$\begin{array}{c} \text{CABDABDAB} \\ L'(8) = \text{CABDABDAB} \\ \text{CAB} = 3 \end{array}$$

In questo caso invece non potevamo scegliere **DAB** in quanto suffisso di  $P$  (= **DAB**).

Allora la sottostringa più a destra del pattern e compatibile con ciò che volevamo è **CAB**, della quale prendiamo la posizione dell'ultimo carattere **B**. Concludiamo che  $L'(8) = 3$ .

Sappiamo che  $L(i)$  e  $L'(i)$  possono essere calcolati in  $\mathcal{O}(n)$ .

L'ultima cosa che manca è la definizione degli  $N_j(P)$ .

Sia  $N_j(P)$  la lunghezza del più lungo suffisso della sottostringa  $P[1 \dots j]$  che è anche suffisso dell'intera stringa  $P$ .

$$\text{CABDABDAB}$$

$$N_3(P) = \text{CABDABDAB} \text{ (Prendo la sottostringa } t = P[1,3])$$

$$\text{CABDABDAB} \text{ (Cerco il più lungo suffisso contenuto in } t)$$

$$N_3(P) = 2$$

Osservazione :  $N$  è l'inverso di  $Z$ , per cui posso utilizzare lo  $Z$ -algorithm per calcolare i vari  $N_j$  semplicemente utilizzando la stringa riflessa  $P^R$  anziché  $P$ . Infine, è bene introdurre 2 teoremi importanti:

**Teorema 2.2.2 :**  $L_i$  è il più grande indice  $j < n$  tale per cui  $N_j(P) \geq |P[i, \dots, n]|$  (Ovvero  $n - i + 1$ ).  $L'(i)$  è il più grande indice  $j < n$  tale per cui  $N_j(P) = |P[i, \dots, n]|$  (Ovvero  $n - i + 1$ ).

Per trattare il caso in cui  $L'(i) = 0$  dobbiamo introdurre un secondo teorema:

**Teorema 2.2.4 :**  $L'_i$  è uguale al più grande indice  $j < |P[i, \dots, n]|$ , cioè  $n - i + 1$

e tale che  $N_j(P) = j$  Ed eccoci arrivati all'algoritmo completo:

### The Boyer-Moore algorithm

{Preprocessing stage}

Given the pattern  $P$ ,

Compute  $L'(i)$  and  $L''(i)$  for each position  $i$  of  $P$ ,  
and compute  $R(x)$  for each character  $x \in \Sigma$ .

{Search stage}

$k := n$ ;

while  $k \leq m$  do

begin

$i := n$ ;

$h := k$ ;

while  $i > 0$  and  $P(i) = T(h)$  do

begin

$i := i - 1$ ;

$h := h - 1$ ;

end;

if  $i = 0$  then

begin

report an occurrence of  $P$  in  $T$  ending at position  $k$ .

$k := k + n - L'(2)$ ;

end

else

shift  $P$  (increase  $k$ ) by the maximum amount determined by the  
(extended) bad character rule and the good suffix rule.

end;

## 2.4 Knuth-Morris-Pratt

È raramente usato nella pratica perchè in molti casi è peggiore di Boyer-Moore. Costituisce la base di un altro algoritmo (Aho-Corasick) che può cercare in parallelo più pattern nello stesso testo.

L'idea è la seguente:

Se  $P = abcxabcde$  e in un allineamento il mismatch avviene col carattere  $P(8)$ , allora posso tranquillamente spostare  $P$  di 4 posizioni senza rischiare di saltare nessuna occorrenza di  $P$  in  $T$ .

ABCXABCDE (Mismatch in posizione 8)

ABCXABCDE

ABCXABCDE

### 2.4.1 Preprocessing in Knuth-Morris-Pratt

Devo calcolare gli  $sp_i(P)$  e gli  $sp'_i(P)$ . Per definizione :

- Definisco  $sp_i(P)$  la lunghezza del più lungo suffisso proprio della sottostringa  $t = P[1..i]$  uguale ad un prefisso di  $P$  Esempio : Calcoliamo  $sp_4(P)$ :

ABC**A**EABCABD (Prendo il carattere in posizione 4)

AB**CA**EABCABD (Prendo la sottostringa  $t[1,4]$ )

AB**CA** (Trovo il suffisso più lungo che sia uguale ad un prefisso di  $t$ )

Per cui concludo che  $sp_4(P) = 1$ . Per citarne un altro,  $sp_8(P) = 3$

- Definisco  $sp'_i(P)$  la lunghezza del più lungo suffisso proprio della sottostringa  $t = P[1..i]$  uguale ad un prefisso di  $P$  ma tale per cui i caratteri  $P(i+1)$  e  $P(sp'_i(P)+1)$  siano diversi. Ad esempio calcoliamo  $sp'_8(P)$

BBCCEAB**B**CABD (Prendo il carattere in posizione 8)

BBCCEAB**B**CABD (Prendo la sottostringa  $t[1,8]$ )

BBCCEAB**B** (Trovo il suffisso più lungo che sia uguale ad un prefisso di  $t$  ma "senza ripetizione di caratteri consecutivi")

Concludiamo quindi che  $sp'_8(P) = 1$  anche se  $sp_8(P) = 2$

L'algoritmo funziona nel modo seguente:

Per ogni allineamento, se il primo mismatch si presenta in posizione  $i+1$  di  $P$  e  $k$  di  $T$ , sposto  $P$  a destra in modo che  $P[1, \dots, sp'_i]$  sia allineato con  $T[k-sp'_i, \dots, k-1]$ . Se si trova un'occorrenza di  $P$ , sposto  $P$  di  $n - sp'_n$  posizioni.

Esempio:

XYABCXABCXADCDQFEG

ABCXABCDE

mismatch con  $i + 1 = 8$  e  $k = 10$

Allora trovo  $sp'_7 = 3$ , infatti:

ABCXABCDE

ABCXABC (prendo la sottostringa  $t[1,7]$ )

ABCXABC (prendo il suffisso più lungo uguale ad un prefisso senza ripetizioni di carattere col precedente)

Ergo  $sp'_7 = 3$ .

Adesso sposto P in modo da allineare  $P[1,3]$  con  $T[10-3, 9] = T[7,9]$ .

XYABCXABCXADCDQFEG ( $T[7,9]$ )

ABCXABCDE ( $P[1,3]$ )

### 3 Algoritmi di Motif finding

In questa sezione andremo a descrivere gli algoritmi che servono per individuare delle sequenze quando si conosce T ma non il pattern. Detta così sembrerebbe una cosa impossibile, ma è solo tremendamente complicata dal punto di vista delle risorse computazionali che richiede.

Esempio di problema giocattolo: trovare un motif (motivo, pattern) in 20 sequenze randomiche lunghe 600 nucleotidi.

- ogni sequenze contiene uno pseudo pattern di lunghezza 15.
- ogni pattern appare con 4 mismatch rispetto al pattern da trovare (che comunque non conosciamo).

Generalizzando, sappiamo che in un determinato testo esiste un motivo  $(n,k)$  ovvero un pattern lungo  $n$  e che presenta  $k$  da quello che vogliamo trovare (ma che non conosciamo).

Nel caso del DNA il nostro problema sarà quello di trovare un motivo  $(15,4)$  in un gruppo di 20 sequenze.

Attenzione! anche se sono 4 i mismatch “ufficiali”, 2 stringhe possono differire di molti più caratteri (infatti sappiamo solo che  $s_1$  ed  $s_2$  sono distanti 4 da  $p$ , ma è possibile che tra di loro siano distanti per esempio 8).

In biologia ogni gene è preceduto da un cosiddetto “fattore di regolazione” che influenza il grado di espressione del gene in questione (tanto, poco, per niente...)

Come già accennato in precedenza, i motif sono variabili anche tra di loro, in quanto è possibile che avvengano mutazioni a livello delle basi “poco importanti”. Una tecnica per visualizzare queste differenze è quella del *motif logo*:

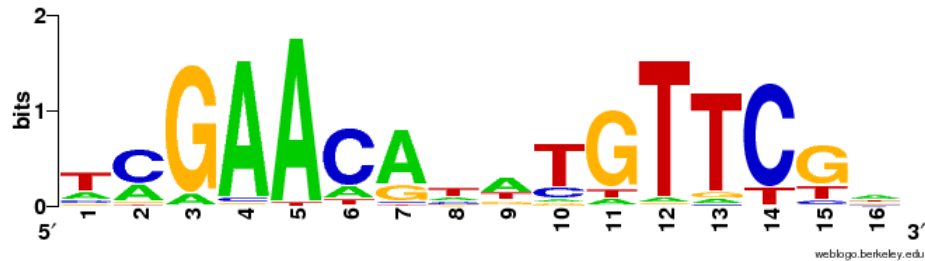


Figura 15: Esempio di motif logo

Si tratta semplicemente di una rappresentazione visiva delle frequenze delle basi in una determinata posizione. In caso di mismatch si disegna con una dimensione maggiore la base più frequente (e si disegnano le altre via via sempre minori sempre proporzionalmente alla frequenza).

Ci sono però delle complicazioni :

- Non sappiamo qual'è il motif finchè non lo troviamo
- Rispetto ad un gene, non sappiamo dove si trova questo motif.
- Tra un gene e l'altro il motif può comparire con qualche mutazione
- Gran numero di falsi positivi (alla fine il motif è uno solo e noi troviamo un insieme di candidati tra cui scegliere... ma non sappiamo come fare)

Per fare un parallelismo con la Cyber Security è più o meno lo stesso problema della crittanalisi statistica applicata a sequenze di  $n$  caratteri. Il problema è che noi “non sappiamo la grammatica del DNA”, ci limitiamo ad osservare i fenomeni in una logica anche un po' ripetitiva di azione-reazione. Per fare un esempio si veda la tecnologia microarray nel capitolo introduttivo. Un esempio più narrativo è quello del Gold Bug <sup>3</sup>.

Ora, la complessità del problema è davvero molto ampio, per cui dobbiamo necessariamente fare delle assunzioni:

- Supponiamo che la sequenza nascosta sia lunga esattamente 8
- Supponiamo che siano ammesse al più 2 mutazioni.
- Dato che i motif possono iniziare da posizioni diverse per ogni sequenza presa in esame supporremo di conoscere per ognuna il punto di partenza del motif.

<sup>3</sup>[https://en.wikipedia.org/wiki/The\\_Gold-Bug](https://en.wikipedia.org/wiki/The_Gold-Bug)

Di conseguenza quello che possiamo fare è prendere tutti i candidati motif di 8 caratteri ed allinearli a formare una matrice rettangolare (quadrata se prendiamo esattamente 8 sequenze). Un esempio è il seguente:

Alignment		a	G	g	t	a	c	T	t
		C	c	A	t	a	c	g	t
		a	c	g	t	T	A	g	t
		a	c	g	t	C	c	A	t
		C	c	g	t	a	c	g	G
<hr/>									
Profile	A	3	0	1	0	3	1	1	0
	C	2	4	0	0	1	4	0	0
	G	0	1	4	0	0	0	3	1
	T	0	0	0	5	1	0	1	4
<hr/>									
Consensus		A	C	G	T	A	C	G	T

Figura 16: Meccanismo di determinazione del candidato motif

Quello che abbiamo fatto in figura è stato appunto fare un allineamento, e poi, proprio come fosse una sfida di crittanalisi statistica abbiamo contato per ogni lettera dell'alfabeto il numero di occorrenze di una particolare colonna. Il motif cercato è quindi la stringa determinata dalle frequenze maggiori (nel nostro caso 3-4-4-5-3-4-3-4). Ora, è chiaro che non sapendo a priori il motif corretto non possiamo stabilire con certezza che quello che abbiamo ottenuto se è effettivamente ciò che ci aspettiamo. Si dimostra però che la stringa di consenso (il motif che abbiamo trovato) differisce dal vero motif per una distanza minore rispetto al confronto di due motif reali (e quindi siamo salvi, perchè rientriamo nel numero di mutazioni ammissibili).

Per avere un'idea grafica consideriamo la seguente figura autoesplicativa:

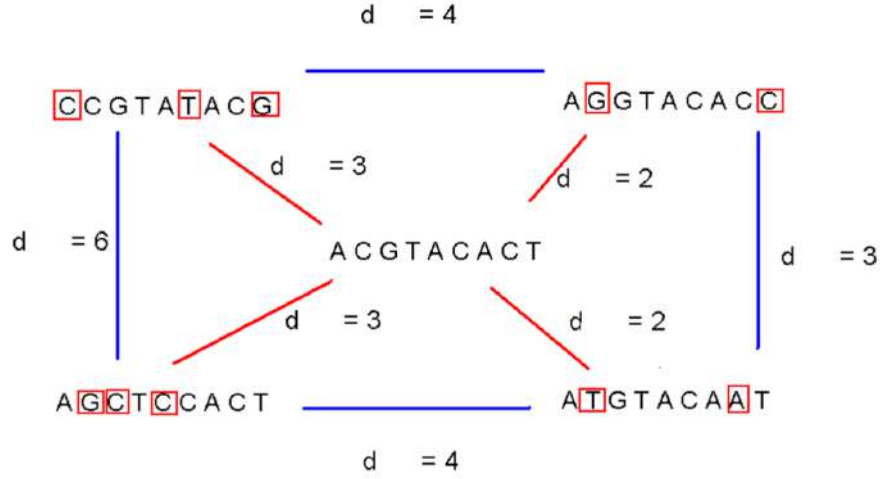


Figura 17: Esempio di distanze di sequenze dalla stringa di consenso

Facciamo ora un passo in avanti : essendo le stringhe di consenso numerose, non basta fermarsi alla prima stringa di consenso che troviamo, ma vorremmo quella “migliore”. Per fare questo dobbiamo inventarci una funzione di *score* per poter valutare la qualità delle varie stringhe di consenso. Scegliamo come funzione di score un qualcosa di siffatto:

$$\sum_{i=1}^l \max(count(k, i)) \quad (1)$$

In questo caso abbiamo come parametri  $l$  che sarebbe la lunghezza delle sequenze prese in esame,  $count(k, i)$  che rappresenta sostanzialmente la frequenza del nucleotide  $k$  all’interno del motif che inizia in  $i$ . Tornando alla figura 16, possiamo quindi calcolare lo score della nostra stringa di consenso nel seguente modo :

$$Score = 3 + 4 + 4 + 5 + 3 + 4 + 3 + 4 = 30 \quad (2)$$

Ora è facile determinare la migliore stringa di consenso una volta note le posizioni iniziali dei motif. Nella realtà però non abbiamo questa fortuna, il che vuol dire che dobbiamo cercare un sistema che ci permetta di determinare la migliore stringa possibile. Vediamo alcuni modi.

### 3.1 Metodo Brute Force

È il metodo classico ed anche il più rozzo, semplicemente faccio la matrice dei punteggi su ogni possibile combinazione di punti di partenza dei motif.

---

**Algorithm 1** BruteForceMotifSearch

---

```

BruteForceMotifSearch(DNA, t, n, l)
  bestScore  $\leftarrow$  0
  for all  $s = (s_1, s_2, \dots, s_n)$  from  $(1, 1, \dots, 1)$  to  $(n - l + 1, \dots, n - l + 1)$  do
    if ( $score(s, DNA) > bestScore$ ) then
      bestScore  $\leftarrow score(s, DNA)$ 
      bestMotif  $\leftarrow (s_1, s_2, \dots, s_n)$ 
    end if
  end for
  return bestMotif

```

---

Discutiamo ora la complessità di questo algoritmo : variamo  $n - l + 1$  posizioni per ogni  $t$  sequenze, per cui stiamo guardando  $(n - l + 1)^t$  insiemi di candidate posizioni di partenza.

La complessità è quindi  $O(ln^t)$  (yuppi, esponenziale :-( ). Per fare un esempio, con  $t = 8$ ,  $n = 1000$  ed  $l = 10$  faremo circa  $10 \cdot 1000^8 = 10 \cdot (10^3)^8 = 10 \cdot 10^{24} = 10^{25}$  computazioni, per cui anche per un problema semplice l'algoritmo impiegherebbe miliardi di anni per terminare.

**Conclusion:** dobbiamo vedere il problema con sotto un'altra luce.

### 3.2 Problema della stringa mediana

Passiamo ora a considerare un nuovo problema: dato un set  $t$  di sequenze di DNA, trovare un pattern che appare in tutte le sequenze  $t$  avente il minor numero di mutazioni.

**Idea chiave:** al posto di cercare tutte le combinazioni di punti di partenza dei motif, cerchiamo invece direttamente tutti i possibili motif.

Per proseguire andiamo quindi a definire qualcosa che ci sarà utile più tardi : la "Hamming Distance":

Dati due nucleotidi,  $v$  e  $w$ , la distanza di hamming  $d_H(v, w)$  non è altro che il numero di mismatch quando le stringhe sono perfettamente allineate.

Esempio :

$$d_H(AAAAAA, ACAAAC) = 2, \text{ infatti}$$

```

AAAAAA
ACAAAC

```



Definiamo adesso la funzione  $\text{TotalDistance}(v, \text{DNA})$ . Prima però spieghiamo l'idea che c'è dietro: quello che andremo a fare sarà per ogni sequenza  $t$  del DNA, trovare la somma delle distanze da un motif  $v$ . Alla fine prenderemo il motif con distanza minima tra tutte.

Il problema è che anche in questo caso è implicita la ricerca per forza bruta, dato che dobbiamo generare tutti i possibili motif di lunghezza  $l$ .

L'algoritmo è quindi il seguente:

---

**Algorithm 2** MedianStringSearch

---

```

MedianStringSearch(DNA, t, n, l)
  bestWord  $\leftarrow$  AAA...AA
  bestDistance  $\leftarrow$  0
  for all l-mer  $v$  from AAA...AA to TTT...TT do
    if ( $\text{TotalDistance}(v, \text{DNA}) < \text{bestDistance}$ ) then
      bestDistance  $\leftarrow \text{TotalDistance}(v, \text{DNA})$ 
      bestWord  $\leftarrow v$ 
    end if
  end for
  return bestWord

```

---

Dunque, dal punto di vista computazionale possiamo dire che non abbiamo avuto un grosso miglioramento: nella bruteforce del MotifFinder esaminavamo  $(n-l+1)^t$ , adesso dobbiamo computare  $4^l$  motifs. Il che va bene se le sequenze sono corte, ma con  $l$  sufficientemente grandi siamo punto a capo.

A questo punto dobbiamo necessariamente cercare una soluzione per ottimizzare la fase di ricerca. Potremmo pensare ad esempio di utilizzare i numeri anziché le lettere, così che si possa simulare un conteggio in base 4 (senza l'uso dello zero). Possiamo quindi organizzare le stringhe tramite i loro prefissi in una logica ad albero:

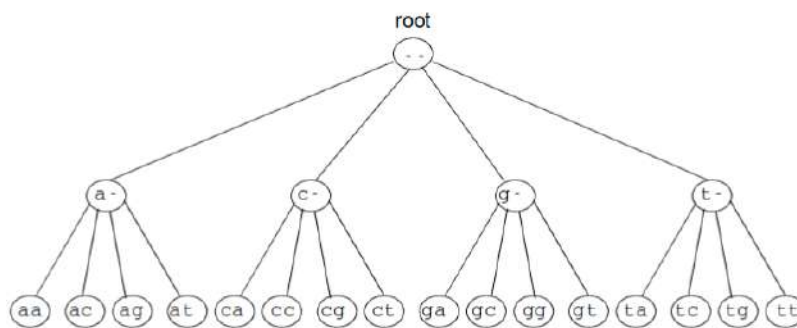


Figura 18: Organizzazione della ricerca mediante struttura ad albero di prefissi

Ora, le proprietà di un albero di ricerca sono note a chiunque abbia una laurea triennale in informatica, per cui direi che possiamo sorvolare. Quello che

salta subito all'occhio è che adesso possiamo visitare i vari nodi in modo furbo (ricerca in ampiezza o in profondità). È utile nel nostro caso crearci delle funzioni di supporto per semplificare il processo: ad esempio potremmo pensare di voler ogni tanto saltare l'esplorazione di un nodo (e di conseguenza di tutti i suoi discendenti) per risparmiare confronti. Per cui andiamo a definire la funzione `nextLeaf`, che appunto ci conduce direttamente alla prossima foglia. Per aiutarci a capire guardiamo la figura precedente: possiamo notare che ogni foglia ha come prefisso il nodo padre, e differisce con gli altri "sibling" di una cifra in base 4. Per cui quello che andremo a fare sarà partire dalla cifra meno significativa e andremo via via a sommare 1 fin quando non raggiungeremo il valore 4. Facciamo un esempio :

$$\begin{aligned} \text{nextLeaf}(21, 2, 4) &= 22, \text{ infatti} \\ &\quad \text{21} \\ &\quad \text{2 (1+1)} \\ &\quad \text{22 (2 non è } < 2, \text{ per cui mi fermo)} \end{aligned}$$

Ed ecco qui l'algoritmo completo:

---

**Algorithm 3** NextLeaf

---

```

NextLeaf(a, L, k)
for i ← L to 1 do
  if (ai < k) then
    ai ← ai + 1
    return a
  end if
ai ← 1
end for
return a

```

---

Nel disegno di prima:

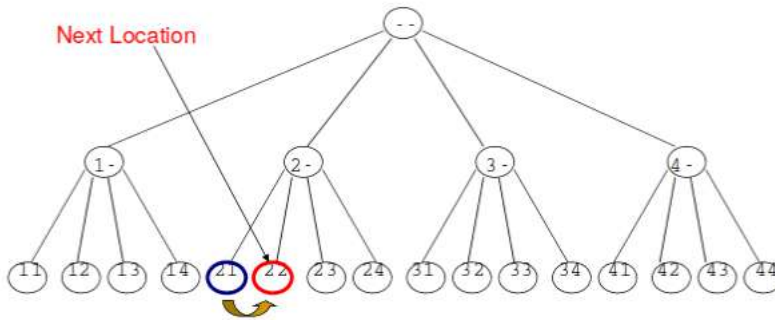


Figura 19: Applicazione di NextLeaf alla foglia 21

La visita in profondità di un albero siffatto viene effettuata dalla funzione AllLeaves, che non fa altro che visitare ogni nodo in modo DFS.

---

**Algorithm 4** AllLeaves
 

---

```

AllLeaves(L, k)
   $a \leftarrow (1, \dots, 1)$ 
  for ever do
    output a
     $a \leftarrow \text{NextLeaf}(a, L, k)$ 
    if ( $a == (1, \dots, 1)$ ) then
      return
    end if
  end for
  
```

---

Andiamo ora a vedere come poter visitare il prossimo nodo generico, non per forza una foglia. L'algoritmo è più o meno quello del NextLeaf, con una parte che si occupa di aggiungere delle cifre a destra.

---

**Algorithm 5** NextVertex
 

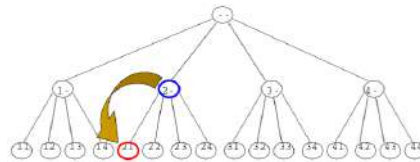
---

```

NextVertex(a, i, L, k)
  if ( $i < L$ ) then
     $a_{i+1} \leftarrow 1$ 
    return ( $a, i + 1$ )
  else
    for  $j \leftarrow L$  to 1 do
      if ( $a_j < k$ ) then
         $a_j \leftarrow a_j + 1$ 
        return ( $a, j$ )
      end if
    end for
     $a_i \leftarrow 1$ 
  end if
  return ( $a, 0$ )
  
```

---

Graficamente:



**Nota:** Se la mia posizione attuale è una foglia diversa dall'ultima allora viene applicato NextLeaf. Se invece siamo nell'ultima foglia mi sposto al nodo successivo del padre. Nel caso ci si trovi in un nodo non foglia, il successivo sarà

l'immediata foglia sottostante.

L'ultima mossa da considerare è quella nella quale mi trovo in un nodo non foglia, e voglio andare al successivo non foglia senza dover scendere di profondità e quindi andando a visitare i figli di quel nodo. Per far ciò ci aiutiamo con l'algoritmo Bypass.

---

**Algorithm 6** Bypass
 

---

```

Bypass(a, i, k)
  for  $j \leftarrow i$  to 1 do
    if  $(a_j < k)$  then
       $a_j \leftarrow a_j + 1$ 
      return (a,j)
    end if
  end for
  return (a,0)
  
```

---

Graficamente:

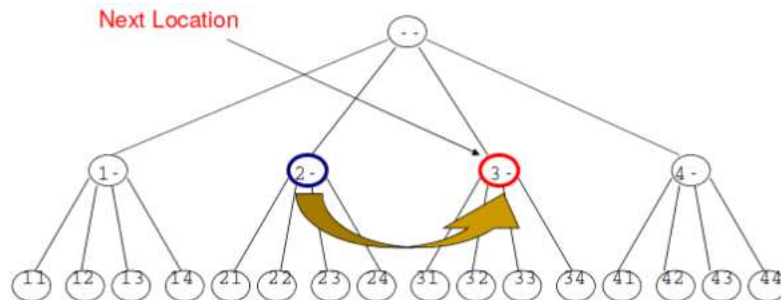


Figura 20: Applicazione del Bypass su un nodo non foglia

A questo punto ci siamo costruiti delle primitive grazie alle quali poter rivisitare i due algoritmi bruteforce visti finora. Andiamo quindi a riscriverli con le conoscenze attuali

**Algorithm 7** BruteForceMotifSearchAgain

---

```

BruteForceMotifSearchAgain(DNA, t, n, l)
  s  $\leftarrow$  (1, ..., 1)
  bestScore  $\leftarrow$  Score(s, DNA)
  for ever do
    s  $\leftarrow$  NextLeaf(s, t, n - l + 1)
    if (score(s, DNA) > bestScore) then
      bestScore  $\leftarrow$  score(s, DNA)
      bestMotif  $\leftarrow$  (s1, s2, ..., sn)
    end if
  end for
  return bestMotif

```

---

C'è però ancora un problema: anche in questo caso quello che andiamo a fare è semplicemente la visita di tutti i nodi dell'albero. Dobbiamo quindi formalizzare delle intuizioni che ci consentono di velocizzare l'algoritmo.

Intuitivamente la parte iniziale e quella finale dell'albero avranno un punteggio basso (perchè contengono molte ripetizioni, che in natura sono improbabili). Ma non basta. Supponiamo che tutte le posizioni successive (t-i) ( $s_{i+1}, \dots, s_t$ ) aggiungano allo score al più  $(t-i) \cdot l$ , per cui, se ci accorgiamo che  $\text{Score}(s, i, \text{DNA}) + (t-i) \cdot l < \text{bestScore}$ , allora non ha senso esplorare il sottoalbero, per cui possiamo utilizzare le funzioni di Bypass e di NextVertex per spostarci in un'altra zona. Realizziamo così un cosiddetto approccio "Branch & Bound".

```

1.  BranchAndBoundMotifSearch(DNA, t, n, l)
2.  s  $\leftarrow$  (1, ..., 1)
3.  bestScore  $\leftarrow$  0
4.  i  $\leftarrow$  1
5.  while i > 0
6.    if i < t
7.      optimisticScore  $\leftarrow$  Score(s, i, DNA) + (t - i) * l
8.      if optimisticScore < bestScore
9.        (s, i)  $\leftarrow$  Bypass(s, i, n - l + 1)
10.     else
11.       (s, i)  $\leftarrow$  NextVertex(s, i, n - l + 1)
12.     else
13.       if Score(s, DNA) > bestScore
14.         bestScore  $\leftarrow$  Score(s)
15.         bestMotif  $\leftarrow$  (s1, s2, s3, ..., st)
16.         (s, i)  $\leftarrow$  NextVertex(s, i, t, n - l + 1)
17.  return bestMotif

```

Figura 21: Branch &amp; Bound Motif Search Algorithm

Qui di seguito troviamo anche la versione Branch & Bound dell'algoritmo di ricerca della stringa mediana.

```

1. BranchAndBoundMedianStringSearch(DNA, t, n, l)
2. s  $\leftarrow$  (1,...,1)
3. bestDistance  $\leftarrow \infty$ 
4. i  $\leftarrow$  1
5. while i > 0
6.   if i < l
7.     prefix  $\leftarrow$  string corresponding to the first i nucleotides of s
8.     optimisticDistance  $\leftarrow$  TotalDistance(prefix, DNA)
9.     if optimisticDistance > bestDistance
10.      (s, i)  $\leftarrow$  Bypass(s, i, l, 4)
11.   else
12.     (s, i)  $\leftarrow$  NextVertex(s, i, l, 4)
13. else
14.   word  $\leftarrow$  nucleotide string corresponding to s
15.   if TotalDistance(word, DNA) < bestDistance
16.     bestDistance  $\leftarrow$  TotalDistance(word, DNA)
17.     bestWord  $\leftarrow$  word
18.   (s, i)  $\leftarrow$  NextVertex(s, i, l, 4)
19. return bestWord

```

Figura 22: Branch & Bound Median String Search Algorithm

In questo caso l'intuizione è stata quella di dire che non ha senso esplorare un nodo se la distanza del prefisso è già maggiore rispetto al mio attuale miglior risultato.

Un ultimo step che possiamo fare è quello di tentare di ottimizzare la soglia superata la quale è meglio fare direttamente Bypass. L'idea è la seguente: dividiamo una stringa in due sottostringhe  $w \rightarrow u, v$ . Avremo che necessariamente  $u$  è un prefisso di  $w$  mentre  $v$  è un suffisso. Quello che andremo a fare è dividere il problema in due. Faremo un Bypass se sommando le distanze totali di prefisso e suffisso queste saranno maggiori del mio attuale miglior risultato. Questa parte è un po' più complicata da dire a parole, per cui metto direttamente il codice.

```

1. ImprovedBranchAndBoundMedianString(DNA, t, n, l)
2.    $s = (1, 1, \dots, 1)$ 
3.    $bestDistance = \infty$ 
4.    $i = 1$ 
5.   while  $i > 0$ 
6.     if  $i < l$ 
7.        $prefix$  = nucleotide string corresponding to  $(s_1, s_2, s_3, \dots, s_l)$ 
8.        $optimisticPrefixDistance = TotalDistance(prefix, DNA)$ 
9.       if  $(optimisticPrefixDistance < bestSubstring[i])$ 
10.         $bestSubstring[i] = optimisticPrefixDistance$ 
11.        if  $(l - i < i)$ 
12.           $optimisticSuffixDistance = bestSubstring[l - i]$ 
13.        else
14.           $optimisticSuffixDistance = 0$ ;
15.        if  $optimisticPrefixDistance + optimisticSuffixDistance \geq bestDistance$ 
16.           $(s, i) = Bypass(s, i, l, 4)$ 
17.        else
18.           $(s, i) = NextVertex(s, i, l, 4)$ 
19.      else
20.         $word$  = nucleotide string corresponding to  $(s_1, s_2, s_3, \dots, s_l)$ 
21.        if  $TotalDistance(word, DNA) < bestDistance$ 
22.           $bestDistance = TotalDistance(word, DNA)$ 
23.           $bestWord = word$ 
24.           $(s, l) = NextVertex(s, l, l, 4)$ 
25.   return  $bestWord$ 

```

Figura 23: Versione Migliorata dell'algoritmo B&B della stringa mediana

**Nota:** Questi due algoritmi che abbiamo visto trovano la soluzione ottimale. Esistono però algoritmi che utilizzano un approccio Greedy, per cui si limitano a fornire un risultato “sufficientemente buono”, in modo da velocizzare ancora di più il tempo di running.

## 4 Partial Digest Problem

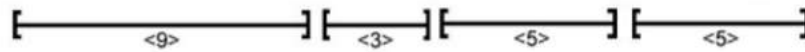
Passiamo ora a parlare del Partial Digest Problem. Per farlo dobbiamo necessariamente parlare di ciò che accade a livello biologico. Le macchine NGS utilizzano degli enzimi particolari che scindono il DNA in sequenze. In realtà però quello che fanno è fare multipli tentativi con timing diversi nei quali fanno agire un'enzima su un campione di DNA solo per un certo quanto di tempo. Il fatto è che in base a quanto tempo uno lascia agire l'enzima, esso riesce a dividere il DNA in punti diversi. Questo è un problema, perchè non riesco di solito a sequenziare tutto un DNA in una volta sola per via degli errori che ancora sono impliciti nell'uso di questa tecnica. Oltre a questo, data una singola sequenza, è possibile trovare multipli “restriction fragments” (ovvero punti nei quali la stringa originale è stata tagliata). Per fare un esempio visivo:



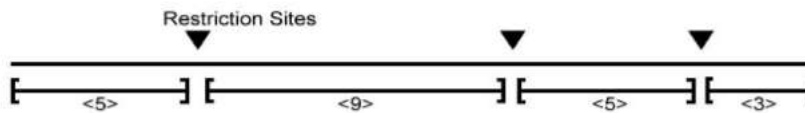
***Is it possible to reconstruct the order of the fragments from the sizes of the fragments  $\{3,5,5,9\}$  ?***

Infatti è spesso possibile ordinare in modo diverso:

- Alternative ordering of restriction fragments:

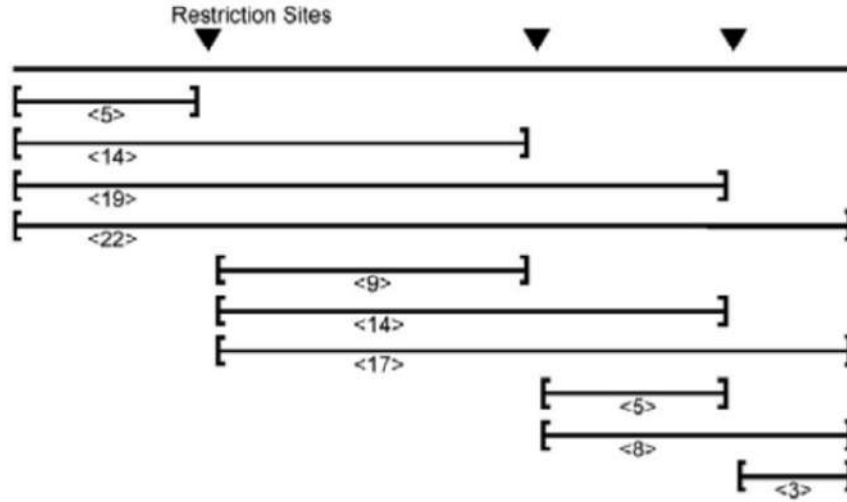


VS



Ricapitolando, l'esperimento biologico genera l'insieme di tutti i possibili restriction sites tra ogni due tagli (non necessariamente consecutivi). Come possiamo ben immaginare, è un lavoro molto oneroso, sia in spazio sia in tempo. Per rimanere sull'esempio di prima, da una sola stringa possiamo determinare molti insiemi diversi :





In questo caso viene a formarsi il seguente multinsieme<sup>4</sup>:

$$X = \{3, 5, 5, 8, 9, 14, 14, 17, 19, 22\}$$

Formalizziamo meglio il problema andando a definire i parametri opportuni : diremo che  $X$  è il multinsieme di partenza,  $n$  è il numero totale di tagli e  $DX$  è il multiset di interi che rappresenta la lunghezza di ognuno dei  $\binom{n}{2}$  frammenti prodotti da un partial digest.

Per chiarire quest'ultimo aspetto mi affido ad un'altra immagine:

X	0	2	4	7	10
0		2	4	7	10
2			2	5	8
4				3	6
7					3
10					

Representation of  $DX = \{2, 2, 3, 3, 4, 5, 6, 7, 8, 10\}$  as a two dimensional table, with elements of

$$X = \{0, 2, 4, 7, 10\}$$

Figura 24: Esempio di insieme DX

<sup>4</sup><https://en.wikipedia.org/wiki/Multiset>

In breve, quello che è stato fatto è stato allineare gli elementi di  $X$  sia in orizzontale, sia in verticale (a formare una matrice), per poi andare a calcolare la distanza dell'elemento  $j$  da quello  $i$ . Per esempio, guardando la prima riga, il 2 dallo 0 differisce di 2, e così via fino al 10 che differisce da 0 di 10. Nella seconda riga invece i conti sono (4-2), (7-2) e (10-2).

Continuiamo il nostro processo di formalizzazione andando a descrivere ciò che abbiamo e ciò che vogliamo ottenere : in particolare, date tutte le distanze tra le varie coppie di punti su una linea, ricostruire la posizione di questi punti.

In input avremo un multiset, di cardinalità  $\frac{n(n+1)}{2}$ , in output invece avremo un insieme  $X$  di interi tale per cui  $DX = L$ .

Anche in questo caso le cose tendono a complicarsi: è infatti possibile che dati due insiemi di partenza entrambi portino allo stesso identico insieme  $X$ , mettendo il dubbio su quale dei due sia quello corretto. Per fare un esempio:

$$X = \{0, 2, 5\} \text{ e } (X + 10) = \{10, 12, 15\}$$

$$\text{Infatti per entrambi } DX = \{2, 3, 5\}$$

Detto questo, passiamo alle tecniche di risoluzione di questi sistemi.

#### 4.1 Metodo Brute Force

Oramai è diventato un classico degli algoritmi, ma cerco di spiegarlo meglio applicato ad un Partial Digest Problem. Innanzitutto tra tutte le mie stringhe devo trovare quella di lunghezza massima, perchè so per certo che quella è l'intera stringa di partenza sulla quale l'enzima di scissione non è riuscito ad agire in tempo.

A questo punto, dobbiamo generare tutti i possibili insiemi mantenendo fissi il valore iniziale e quello finale, ma è meglio fornire un esempio per farlo capire: la base è l'insieme  $X = \{0, 3, 5, 6\}$  che corrisponde ad un  $L = \{1, 2, 3, 3, 5, 6\}$  (che noi facciamo finta di non conoscere). L'esempio si sviluppa così :

$X_1 = \{0, 1, 2, 6\}$	$\Rightarrow$	$DX_1 = \{1, 1, 2, 4, 5, 6\}$
$X_2 = \{0, 1, 3, 6\}$	$\Rightarrow$	$DX_2 = \{1, 2, 3, 3, 5, 6\}$
$X_3 = \{0, 1, 4, 6\}$	$\Rightarrow$	$DX_3 = \{1, 2, 3, 4, 5, 6\}$
$X_4 = \{0, 1, 5, 6\}$	$\Rightarrow$	$DX_4 = \{1, 1, 4, 5, 5, 6\}$
$X_5 = \{0, 2, 3, 6\}$	$\Rightarrow$	$DX_5 = \{1, 2, 3, 3, 4, 6\}$
$X_6 = \{0, 2, 4, 6\}$	$\Rightarrow$	$DX_6 = \{2, 2, 2, 4, 4, 6\}$
$X_7 = \{0, 2, 5, 6\}$	$\Rightarrow$	$DX_7 = \{1, 2, 3, 4, 5, 6\}$
$X_8 = \{0, 3, 4, 6\}$	$\Rightarrow$	$DX_8 = \{1, 2, 3, 3, 4, 6\}$
$X_9 = \{0, 3, 5, 6\}$	$\Rightarrow$	$DX_9 = \{1, 2, 3, 3, 5, 6\}$
$X_{10} = \{0, 4, 5, 6\}$	$\Rightarrow$	$DX_{10} = \{1, 1, 2, 4, 5, 6\}$

Figura 25: Possibili insiemi generati a partire da  $X$

Infine, andiamo a calcolare di ogni insieme generato  $DX_i$ , e se ci accorgiamo che per caso è uguale ad  $L$  allora quell'insieme è proprio quello che determina le posizioni dei restriction sites. Algoritmicamente :

```

1. BruteForcePDP( $L, n$ )
2.    $M \leftarrow$  maximum element in  $L$ 
3.   for every set of  $n - 2$  integers  $0 < x_2 < \dots x_{n-1} < M$ 
4.      $X \leftarrow \{0, x_2, \dots, x_{n-1}, M\}$ 
5.     Form  $DX$  from  $X$ 
6.     if  $DX = L$ 
7.       return  $X$ 
8.   output "no solution"

```

Figura 26: Bruteforce per problema Partial Digest

**Nota:** il numero di insieme generati è  $\binom{M-1}{n-2}$ . Inoltre l'algoritmo ha complessità  $O(M^{n-2})$ .

Possiamo però fare di meglio, ovvero limitarci a generare gli insiemi che contengono solo i valori presenti in  $L$ , che quindi adesso dobbiamo supporre in input.

```

1. AnotherBruteForcePDP( $L, n$ )
2.    $M \leftarrow$  maximum element in  $L$ 
3.   for every set of  $n - 2$  integers  $0 < x_2 < \dots x_{n-1} < M$  from  $L$ 
4.      $X \leftarrow \{0, x_2, \dots, x_{n-1}, M\}$ 
5.     Form  $DX$  from  $X$ 
6.     if  $DX = L$ 
7.       return  $X$ 
8.   output "no solution"

```

Figura 27: Bruteforce più ottimizzato su PDP

**Nota:** L'algoritmo ha complessità  $O(M^{2n-4})$ . In alcuni casi può funzionare, ma la maggior parte delle volte risulta ancora troppo lento.

Dobbiamo cambiare strategia. Ci serve un algoritmo di Branch & Bound. E ci serve la possibilità di backtracking. Per cui, ad un eventuale domanda d'esame diffida di tutte le soluzioni proposte che non comprendono la possibilità di tornare indietro se sbagliamo a scegliere i nodi da visitare. Prima

di arrivarci però ci serve una definizione preliminare : definiamo  $D(y, X) = \{|y - x_1|, |y - x_2|, \dots, |y - x_n|\}$ , per  $X = \{x_1, x_2, \dots, x_n\}$  ed  $y$  come elemento corrente in esame. Passiamo ora a definire l'algoritmo PartialDigest:

**PartialDigest(L):**

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

**Notice:** The function Delete ( $y, L$ ) removes the component  $y$  from  $L$

Figura 28: Definizione di  $D(y, X)$

La funzione Place è invece siffatta :

1. PLACE( $L, X$ )
2. **if**  $L$  is empty
3.   **output**  $X$
4.   **return**
5.  $y$  <- maximum element in  $L$
6. Delete( $y, L$ )
7. **if**  $D(y, X) \subseteq L$
8.   Add  $y$  to  $X$  and remove lengths  $D(y, X)$  from  $L$
9.   PLACE( $L, X$ )
10.   Remove  $y$  from  $X$  and add lengths  $D(y, X)$  to  $L$
11. **if**  $D(width-y, X) \subseteq L$
12.   Add  $width-y$  to  $X$  and remove lengths  $D(width-y, X)$  from  $L$
13.   PLACE( $L, X$ )
14.   Remove  $width-y$  from  $X$  and add lengths  $D(width-y, X)$  to  $L$
15. **return**

Figura 29: Funzione Place

Per aiutare nella comprensione è meglio fare un esempio procedurale. Parliamo da questa situazione:

$$L = \{2, 2, 3, 3, 4, 5, 6, 7, 8, 10\}$$

$$X = \{0\}$$

Procediamo scegliendo l'elemento di  $L$  massimo, ed aggiungiamolo ad  $X$ :

$$L = \{2, 2, 3, 3, 4, 5, 6, 7, 8, 10\}$$

$$X = \{0, 10\}$$

Adesso procediamo col prossimo, prendiamo l'elemento massimo successivo, 8:

$$L = \{2, 2, 3, 3, 4, 5, 6, 7, 8\}$$

$$X = \{0, 10\}$$

Calcoliamo  $D(y, X)$  con  $y = 8$ .  $D(8, X) = \{8 - 0, 10 - 8\} = \{8, 2\}$ . Ora dobbiamo scegliere quale dei due valori usare per l'esplorazione. Scegliamo arbitrariamente  $y = 2$ . Dato che anche 2 è contenuto in  $L$  dobbiamo fare lo stesso discorso. Procediamo quindi calcolando  $D(2, X) = \{2 - 0, 10 - 2\} = \{2, 8\}$ . A questo punto rimuoviamo 2 ed 8 da  $L$  ed aggiungiamo 2 in  $X$ .

$$L = \{2, 3, 3, 4, 5, 6, 7\}$$

$$X = \{0, 2, 10\}$$

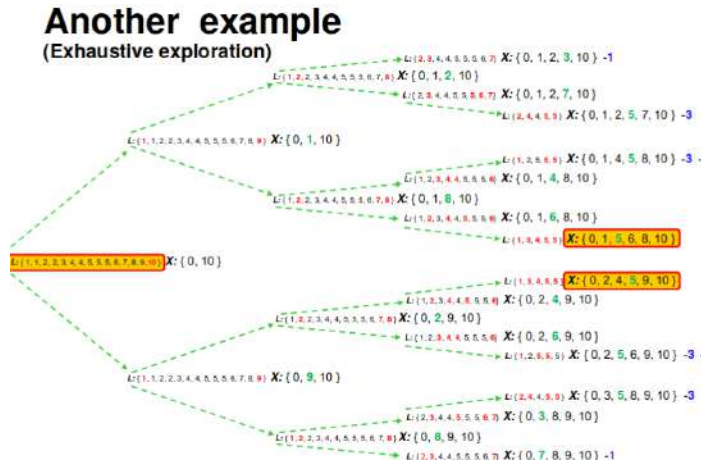
Per la prossima iterazione prendiamo il 7. Possiamo scegliere  $y$  tra  $\{7 - 0, |7 - 10|\} = \{7, 3\}$ . Noi scegliamo arbitrariamente  $y = 7$ . La distanza  $D(7, X) = \{7, 5, 3\}$ . Rimuoviamo questi numeri da  $L$  ed aggiungiamo 7 in  $X$ .

$$L = \{2, 3, 4, 6\}$$

$$X = \{0, 2, 7, 10\}$$

E si va avanti così finchè non troviamo  $X = \{0, 2, 4, 7, 10\}$  ed  $L = \{\}$ .

C'è però da dire che facendo queste scelte arbitrarie su  $y$  abbiamo trovato solo una possibile soluzione. Ce ne sono infatti altre possibili, che vengono determinate dall'esplorazione esaustiva dell'albero che si viene a creare. Per fare un altro esempio :



**Nota:** Attenzione! Quando vado a rimuovere i valori da  $L$  ne rimuovo tanti quanti quelli trovati in  $D(y, X)$ , quello di prima è solo un caso in cui ne rimuovo un'occorrenza alla volta.

Passiamo ora a discutere la complessità:

Nel caso medio l'algoritmo è veloce per quanto quadratico. Nel caso peggiore invece devo per forza esplorarmi tutti i rami, per cui rimane esponenziale.

## 5 Algoritmi di allineamento

### 5.1 Introduzione (Breve)

In bioinformatica è cruciale riuscire ad allineare le sequenze biologiche. Questo consente di studiare varie condizioni, dalle malattie genetiche fino alla costruzione di alberi filogenetici (che vedremo in seguito). In informatica quando si vuole valutare il livello di similarità tra due stringhe ci si affida alla distanza di Hamming (che abbiamo già visto in precedenza). Quindi quello che normalmente facciamo è contare il numero di cambi che dobbiamo fare su una stringa  $X$  affinché diventi uguale ad un'altra stringa  $Y$ . C'è però un problema molto serio, dovuto proprio ad una precodizione che noi informatici diamo per scontato. Vediamo con un esempio:

```
ATGCATGC
TGCATGCC
```

In questo caso la distanza di Hamming è 7. Tuttavia, se cambiamo il posizionamento della seconda stringa rispetto alla prima, otterremo questo:

```
ATGCATGC -
- TGCATGCC
```

Ed ecco che la distanza di Hamming ora vale solo 2. Dobbiamo quindi cambiare strategia e definire un cosiddetto “good alignment”, ricorrendo alla teoria dei giochi. Cercheremo di ricondurre il problema ad un gioco in single-player nel quale ad ogni mossa dobbiamo massimizzare il numero di matches tra 2 stringhe. Ci inventiamo quindi una funzione di score che assegna 1 punto quando rimuovendo un carattere a sinistra da entrambe le stringhe abbiamo un match, 0 altrimenti. Precisiamo che è anche lecito rimuovere un carattere a sinistra da una sola stringa ed ottenere 0 punti. Questo consente di allineare in modo sfasato (ovvero il carattere  $c_i$  di una stringa sarà allineato con il carattere  $c_j$  della seconda (con  $i \neq j$ )). Per convenzione, quando rimuoviamo un carattere dalla stringa di sopra inserendoci uno spazio stiamo effettuando una insertion. Nel caso in cui si rimuova un carattere dalla stringa di sotto inserendoci uno spazio allora stiamo effettuando una deletion. Questa operazione si definisce come indel, accorpando le due definizioni.

Il nostro problema ora lo definiamo col nome di “Longest Common Subsequence”.

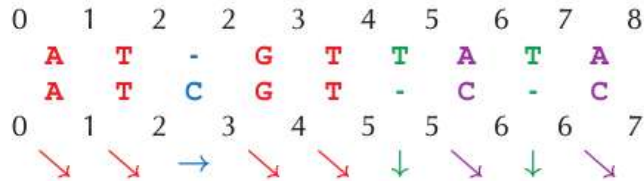
In questa sezione vado spedito al punto per evitare pagine e pagine di esempi e similitudini. L'idea di base è non inventarsi da zero un modo di allineare le stringhe, ma ispirarci ad altri problemi già risolti per trovare dei trucchi da usare a nostro vantaggio. Il problema a cui facciamo riferimento è il turista di Manhattan. In breve, data una griglia con sopra disposte delle attrazioni turistiche, e dati i vincoli di percorso (posso solo muovermi ad est  $\rightarrow$  e a sud  $\downarrow$ ) il nostro compito è trovare un percorso che massimizza le attrazioni visitate. Graficamente, è una cosa del genere:



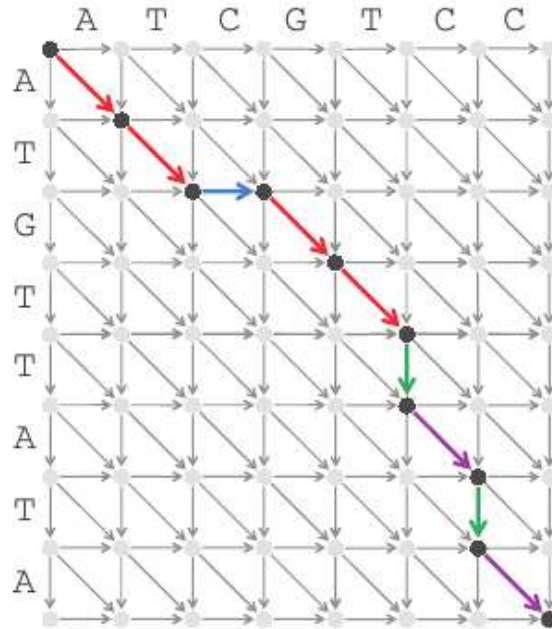
47



buone ma non ottime. Se vogliamo il cammino ottimale dobbiamo fare del lavoro in più. Intanto facciamo una congiunzione tra il problema di Manhattan ed il gioco dell'allineamento:



Possiamo quindi interpretare un match/mismatch (rosso, viola) come un movimento in diagonale, una insertion come un movimento orizzontale ed una deletion come movimento verticale. Globalmente:



Siamo quindi in grado di associare un percorso ad un allineamento. L'ultimo passo è quindi quello di risolvere il problema del cammino più lungo e virtualmente avremo gratis la nostra soluzione. Partiamo

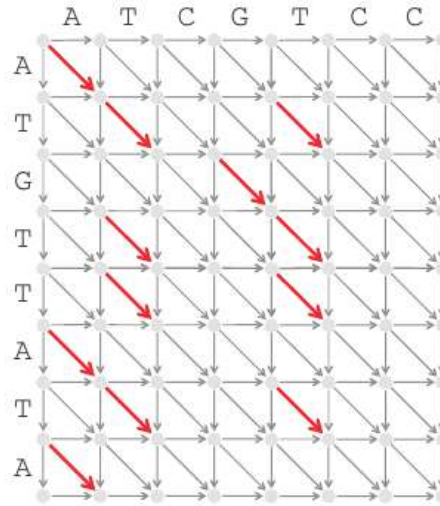
### 5.3 Allineamento Globale

#### 5.3.1 Etichettare gli archi

L'idea è quella di associare ad ogni arco un punteggio basato su una funzione di score tipo quella che abbiamo definito poco fa. Possiamo quindi fare una cosa



del genere:



A parole, abbiamo evidenziato tutti gli archi diagonali che portano ad un match, per cui andremo ad etichettarli con un peso di valore 1. Tutti gli altri archi saranno etichettati con uno zero. Essendo un algoritmo non banale ci arriviamo mediante uno step in più dove andiamo ad introdurre i concetti della Dynamic Programming su un problema diverso, quello del resto.

### 5.3.2 Dynamic programming nel problema del resto

È un problema classico in ambito informatico, e consiste nel fornire a fronte di una determinata cifra, un cambio caratterizzato dal minor numero di monete possibili (o banconote, ma poco cambia, si può generalizzare e si possono fare tutte le astrazioni che vogliamo). Un modo classico di risolvere questo problema è quello di utilizzare un approccio ricorsivo : ad ogni passo la moneta da scegliere sarà il minimo tra i vari minimi degli step precedenti. Per essere chiari :

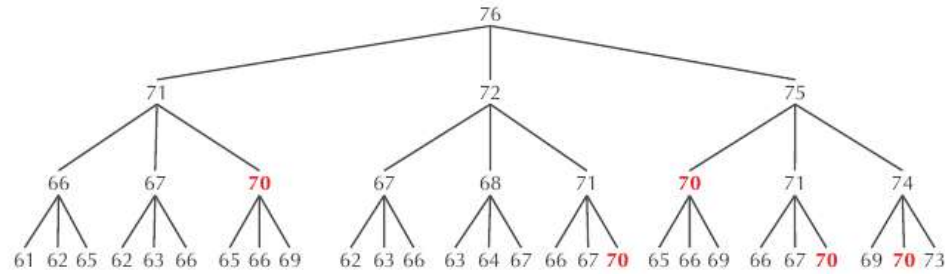
$$\text{MINNUMCOINS}(\text{money}) = \min \begin{cases} \text{MINNUMCOINS}(\text{money} - \text{coin}_1) + 1 \\ \vdots \\ \text{MINNUMCOINS}(\text{money} - \text{coin}_d) + 1 \end{cases}$$

Figura 31: Generalizzazione del metodo di resto ricorsivo

Per fare un esempio, se ricevo 9 e voglio cambiarlo con monete da 6,5 ed 1, la formula sarà la seguente:

$$MinNumCoins(9) = \min \begin{cases} MinNumCoins(9 - 6) + 1 \\ MinNumCoins(9 - 5) + 1 \\ MinNumCoins(9 - 1) + 1 \end{cases}$$

In questo caso abbiamo un problema: l'algoritmo per quanto corretto non è efficiente, infatti al crescere del parametro *money* verranno ricomputati sempre più spesso valori che in realtà abbiamo già calcolato una volta. Un'immagine autoesplicativa del problema è l'albero di ricorsione del problema con *money*=76:



Nel nostro caso il passo ricorsivo con *money*=70 viene ricalcolato 5 volte, 4 delle quali inutili.

Dobbiamo quindi risolvere questo problema di spreco di risorse temporali. Casi come questi sono molto diffusi in ambito informatico: quando ci si sottopone a un problema può capitare che sia possibile migliorarlo in tempo sacrificando qualcosa in termini di spazio. Ed è proprio questo principio che sta alla base della dynamic programming: siamo anche disposti a perdere del tempo all'inizio (magari facendo operazioni controintuitive) ma al fine di velocizzare il passo successivo.

Detto questo l'idea è la seguente: anziché computare le ricorsioni top-down (per tornare all'albero di prima da 76 a 0) adottiamo un approccio bottom up nel quale si parte da 0 e si arriva alla cifra reale da cambiare. Man mano che si calcolano i valori di cambio per somme più piccole ce li memorizziamo in un array di comodo in modo da poterli riutilizzare in seguito per evitarci una ricorsione aggiuntiva. Ed ecco che in men che non si dica non abbiamo più bisogno di ricorsioni, perché di fatto abbiamo già i precedenti valori pronti all'uso. Per completezza, ecco l'algoritmo finale.

```

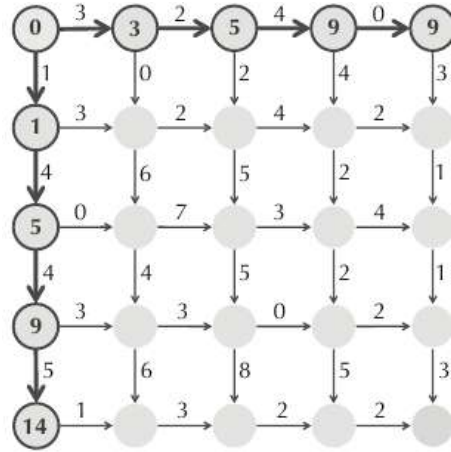
DPCHANGE(money, COINS)
  MINNUMCOINS(0)  $\leftarrow$  0
  for  $m \leftarrow 1$  to money
    MINNUMCOINS( $m$ )  $\leftarrow \infty$ 
    for  $i \leftarrow 1$  to |COINS|
      if  $m \geq \text{coin}_i$ 
        if MINNUMCOINS( $m - \text{coin}_i$ ) + 1 < MINNUMCOINS( $m$ )
          MINNUMCOINS( $m$ )  $\leftarrow$  MINNUMCOINS( $m - \text{coin}_i$ ) + 1
  return MINNUMCOINS(money)

```

Andiamo adesso ad utilizzare un approccio simile per il nostro problema iniziale.

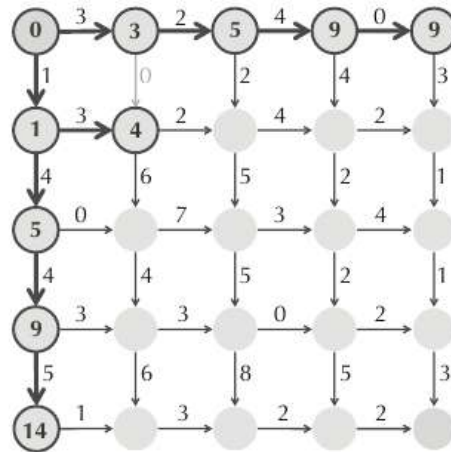
### 5.3.3 Problema di Manhattan con Dynamic Programming

Nel nostro caso l'idea è quella di calcolare il percorso minimo per ogni nodo, se vogliamo andando a risolvere i vari problemi dei sottografi una volta sola anziché milioni di volte. Partiremo quindi dal nodo (0,0) e cercheremo di calcolare il percorso migliore in modo incrementale per ogni nodo successivo. Sembra controintuitivo fare uno sforzo del genere, però funziona. Andiamo a spiegare l'idea con alcuni disegni.



Come primo passo possiamo tranquillamente calcolare in un colpo solo il cammino minimo per la prima riga e per la prima colonna, infatti entrambe le entità hanno come percorso migliore l'unico realmente fattibile, ovvero rispettivamente quello fatto da sole insertion ( $\rightarrow$ ) e quello fatto da sole deletion ( $\downarrow$ ).

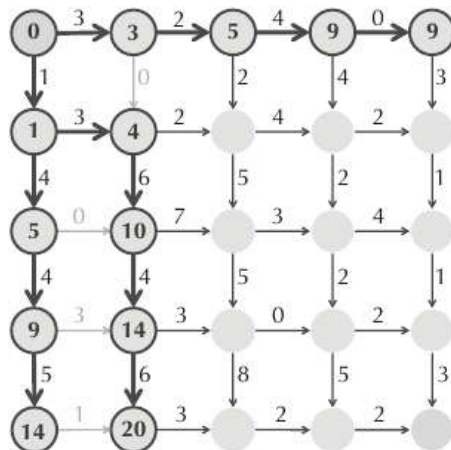
A questo punto ci troviamo di fronte ad un problema più piccolo, per cui ricominciamo dal nodo (1,1)



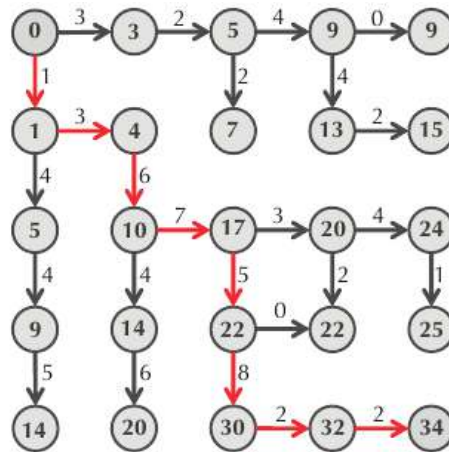
In questo caso il percorso migliore per arrivare ad  $(1,1)$  è il massimo tra i percorsi migliori per arrivare a ciascuno dei nodi entranti, per cui la nostra scelta viene determinata univocamente dal seguente conto :

$$BestPath((1,1)) = \max \begin{cases} 3 + 0 \text{ (partendo da } (0,0) \rightarrow \downarrow) \\ 3 + 1 \text{ (partendo da } (0,0) \downarrow \rightarrow) \end{cases} = 3 + 1 = 4$$

Possiamo dunque procedere sia completando prima una riga e poi una colonna o viceversa (ormai si è capito che il trucco è quello di avere già valutato i percorsi migliori dei nodi entranti quello corrente). Per cui, supponendo di fare prima i conti per colonna:



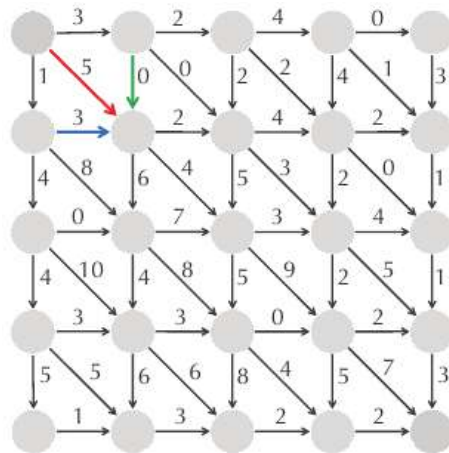
E così via fin quando non abbiamo completato i conti preliminari.



Il trucco finale per evitare le ricorsioni è usare la tecnica del backtracking dal nodo più in basso a destra fino a tornare al nodo iniziale. Quello che faremo è quindi tornare indietro scegliendo il nodo col punteggio maggiore possibile rispetto a dove mi trovo io (tipo Greedy, tanto ormai i conti preliminari ci garantiscono l'ottimalità). Per cui il percorso ottimale sarà, andando a ritroso :

34 - 32 - 30 - 22 - 17 - 10 - 4 - 1 - 0

Ovviamente per noi informatici non cambia molto generalizzare aggiungendo gli archi diagonali. Il metodo è sempre quello, ad ogni step andremo a prendere il massimo tra i migliori cammini degli archi entranti nel nodo che stiamo considerando.



Ed ecco che ora possiamo tornare al nostro problema iniziale, ovvero l'allineamento di sequenze.

Per ogni nodo  $(i,j)$ , quello che andremo a fare sarà la seguente operazione :

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of edge } \downarrow \text{ between } (i-1,j) \text{ and } (i,j) \\ s_{i,j-1} + \text{weight of edge } \rightarrow \text{ between } (i,j-1) \text{ and } (i,j) \\ s_{i-1,j-1} + \text{weight of edge } \searrow \text{ between } (i-1,j-1) \text{ and } (i,j) \end{cases}$$

Dato che stiamo considerando due stringhe  $v$  e  $w$ , torniamo al nostro gioco del buon allineamento ed andiamo a definire la funzione di score definitiva:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, \text{ if } v_i = w_j \end{cases}$$

#### 5.3.4 Gestione corretta delle indel

Abbiamo quindi trovato un buon algoritmo di allineamento che a fronte di costi ragionevoli ci trova il percorso ottimale. Aggiungiamo un po' di complessità ed andiamo a discutere la gestione delle operazioni di indel. Nello specifico, finora abbiamo assegnato 0 punti a ciascuna operazione siffatta, per cui nulla vieta al nostro algoritmo di servirsene in modo improprio, ovvero di abusare di questa possibilità. Nel mondo informatico questo tipo di discussioni sono ben poco interessanti, tuttavia in ambito biologico soluzioni trovate con questo sistema possono non voler dire assolutamente niente ! Anzi, talvolta per i biologi è importante penalizzare alcune mutazioni a scapito di altre, per cui quello che ci viene richiesto in soldoni è modificare la nostra funzione di score in modo opportuno, così che si tenga conto di tutte queste cose (che tra l'altro variano in base a quello che serve al momento).

$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma \\ s_{i,j-1} - \sigma \\ s_{i-1,j-1} + 1, \text{ if } v_i = w_j \\ s_{i-1,j-1} - \mu, \text{ if } v_i \neq w_j. \end{cases}$$

**Legenda** In verde è segnata l'operazione di inserzione, in azzurro quella di delezione, in rosso un match e in viola il mismatch.

È utile in questo caso definire anche le cosiddette “score matrix”, ovvero matrici nelle quali inseriamo gli score per fare una determinata mossa :

	A	C	G	T	-
A	<b>+1</b>	$-\mu$	$-\mu$	$-\mu$	$-\sigma$
C	$-\mu$	<b>+1</b>	$-\mu$	$-\mu$	$-\sigma$
G	$-\mu$	$-\mu$	<b>+1</b>	$-\mu$	$-\sigma$
T	$-\mu$	$-\mu$	$-\mu$	<b>+1</b>	$-\sigma$
-	$-\sigma$	$-\sigma$	$-\sigma$	$-\sigma$	

## 5.4 Allineamento Locale

Arrivati a questo punto siamo in grado di trovare un allineamento ottimale ad una qualsiasi coppia di stringhe. Quello che ora dobbiamo aggiungere è che abbiamo realizzato un cosiddetto allineamento globale, ovvero teniamo conto delle lunghezze complessive delle due sequenze in esame.

Il problema principale di questo approccio sta nel fatto che spesso in svariati contesti biologici va a perdere di senso. Per fare un esempio il nostro algoritmo di allineamento globale non riesce a trovare le similitudini di due sequenze con regioni “Omeobox”<sup>5</sup>. Tradotto, se due specie hanno zone molto corte dove i geni sono simili e zone molto lunghe dove invece non ci sono similitudini allora il nostro algoritmo fa cilecca, perchè tenta di allineare l’intera sequenza.

Intuitivamente sto passando ad un sistema che dato un intervallo si focalizza ad allineare quell’intervallo, proprio come se stessimo davvero applicando un allineamento globale ad una sottostringa nota della nostra sequenza. Per fare un esempio, vorremmo passare da questo tipo di allineamento :

```
GCC-C-AGTC-TATGT-CAGGGGGCAG--A-GCATGCACA-
GCCGCC-GTCGT-T-TTCAG----CA-GTTATGT-T-CAGAT
```

A questo :

```
---G---C-----C--CAGTCATG-TCAGGGGGCAGAGCATGCACA
GCCGCCGTCGTTTTCAGCAGT-TATGTCAG-----A-----T-----
```

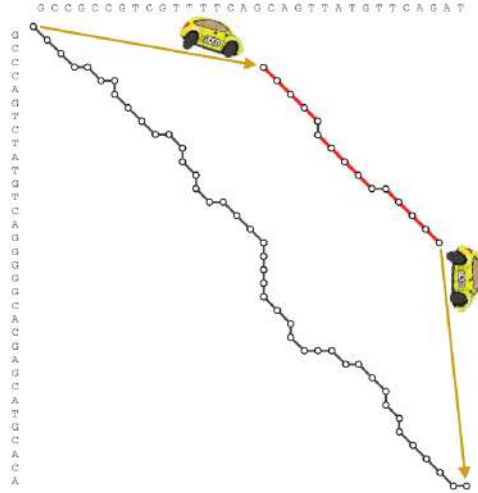
Nel quale i nostri sforzi si sono concentrati sul conservare il più possibile una sottostringa che rappresenta una caratteristica che si mantiene simile durante l’avanzamento delle generazioni.

Stabilito questo, una prima idea per risolverlo è fare multipli allineamenti globali per un certo numero di sottostringhe. Il problema è che per sequenze molto lunghe questo sistema esplode per due motivi, il primo perchè non conosciamo

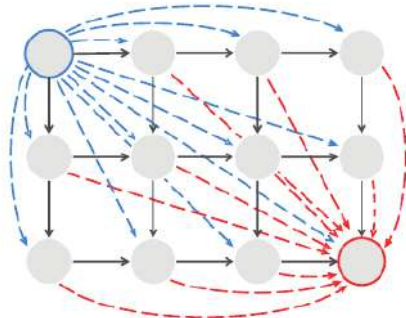
<sup>5</sup>Sequenza lunga circa 180 paia di basi

a priori se ci sono Omeobox, il secondo perchè di conseguenza il numero delle sottostringhe da considerare cresce di molto e computazionalmente diventa un bel problema, sia in termini di tempo sia in termini più spaziali.

#### 5.4.1 Free Taxi Rides



L'idea che risolve il nostro problema è semplice quanto geniale : andiamo a modificare la struttura dati di partenza in un modo tale per cui non costi nulla muoversi dall'origine fino al punto dove sappiamo che c'è una di queste zone Omeobox. In questo modo, dovendo massimizzare lo score, l'algoritmo potrà tranquillamente fare dei salti e riprendere da dove vogliamo. Ora, dato che non è vero che conosciamo a priori se ci sono zone siffatte, quello che possiamo fare è creare degli archi fittizi che dall'origine possono andare ad ogni altro nodo, a costo 0. Bisogna solo fare attenzione che dobbiamo fare la stessa cosa anche da ogni nodo al nodo sink (il finale). Graficamente:





Per quanto riguarda invece l'algoritmo vero e proprio, e quindi guardando con un occhio più informatico il discorso implementativo, la modifica che faremo sarà la seguente:

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j} + \text{Score}(v_i, -) \\ s_{i,j-1} + \text{Score}(-, w_j) \\ s_{i-1,j-1} + \text{Score}(v_i, w_j) \end{cases}$$

A parole, ad ogni nodo  $(i, j)$  andremo a scegliere come percorso migliore il massimo tra 4 possibili:

1. Il percorso migliore del nodo entrante subito a sinistra (a seguito di un'inserzione)
2. Il percorso migliore del nodo entrante subito sopra (a seguito di una delezione)
3. Il percorso migliore del nodo in diagonale (in alto a sinistra) a seguito di un match
4. Zero nel caso non banale in cui per qualche motivo i precedenti score siano tutti valori negativi.

A questi valori ricordiamoci che una volta scelto il massimo, dobbiamo ancora sommare il peso dell'arco che congiunge il nodo scelto con quello attuale.

**Attenzione!** Tutto questo funziona solo a patto di modificare opportunamente la funzione di score in modo da penalizzare un certo tipo di scelta per far sì che ogni tanto l'algoritmo possa effettivamente fare dei salti.

Per quanto riguarda le Free Taxi Rides per arrivare alla fine (dunque il secondo gruppo di archi che abbiamo aggiunto arbitrariamente), una volta fatta questa aggiunta non abbiamo bisogno di modifiche sostanziali, ma a questo punto possiamo iniziare il backtracking essenzialmente da qualsiasi nodo. Per questo motivo l'allineamento locale ottimale deve essere quello che si conclude nel nodo dal massimo score ma non necessariamente dal sink.

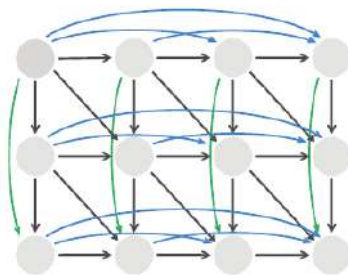
#### 5.4.2 Penalizzare correttamente le Indel

Penso che in questo corso ci si diverta ad aggiungere un po' di complessità ogni tot. tempo, giusto per non annoiarsi. Battute stupide a parte, adesso ci

tocca fare i conti con altri problemi dei biologi. In particolare, ci possono essere tranquillamente delle sequenze che presentano magari  $k$  indel consecutive, ma poi hanno tanti match. Questo ha senso perchè solitamente una serie di indel rappresenta un evento evolutivo (gap) che ha più senso di  $k$  indel separate nel genoma (anche perchè può capitare che andiamo a sostenere che a mutare sia un gene che magari nella realtà tende invece a conservarsi con l'evoluzione). Se vogliamo, l'origine di tutti i mali sta nel fatto che abbiamo assegnato alle indel una penalità fissa  $\sigma$ , quando in realtà vorremmo che con l'aumentare del numero di indel che incontriamo questa quantità diminuisca proporzionalmente nel tempo. Una penalità più ragionevole è un qualcosa del tipo:

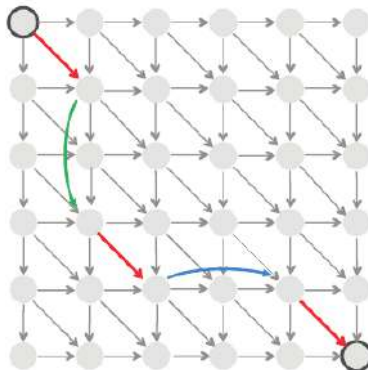
$$\sigma + \epsilon(k - 1) \quad (3)$$

Così facendo con  $k = 1$  avremo solo  $\sigma$ , con  $k = 2$  avremo  $\sigma + \epsilon$ , con  $k = 3$   $\sigma + 2\epsilon$  e così via. Tutto bene fin quando non pensiamo a come implementare correttamente questa modifica nel nostro DAG. Quello che sarebbe da fare in teoria è ad ogni nodo aggiungere tanti archi quanti sono i possibili  $k$  valori di eventuale gap. Graficamente è abbastanza brutto:



E per fortuna che il libro si è fermato a soli 12 nodi ! Per stringhe sufficientemente lunghe questa strada fa esplodere tutto...

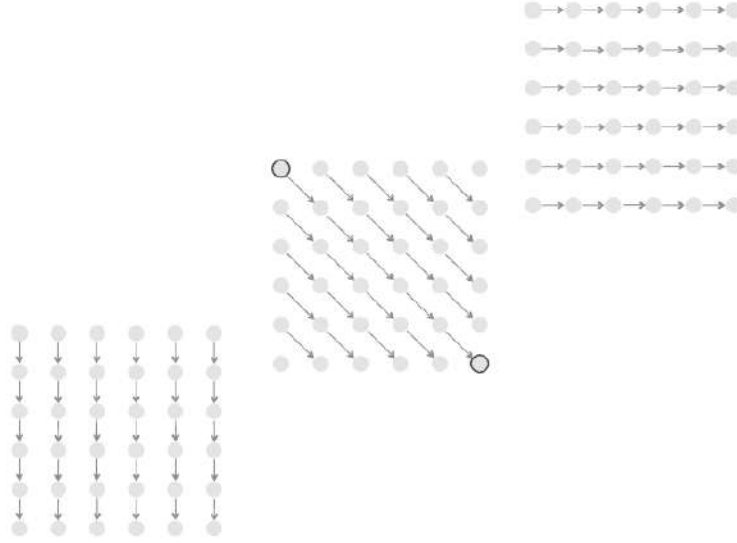
A noi bastava una cosa del genere per vivere felici :



Il problema sembra ormai insormontabile !

Per risolvere questo problema cercando di non fare tutto quel casino di archi aggiuntivi, è aggiungere altri nodi! Come al solito può sembrare una mossa controintuitiva, ma funzionerà. Per fare una similitudine, arriveremo ad avere una struttura astratta molto simile a quella degli Scacchi 3D di Sheldon Cooper.

Quello che andremo a fare è separare il DAG in 3 sotto-DAG dallo stesso numero di nodi del DAG di partenza, ma con archi di un solo tipo, come in figura.



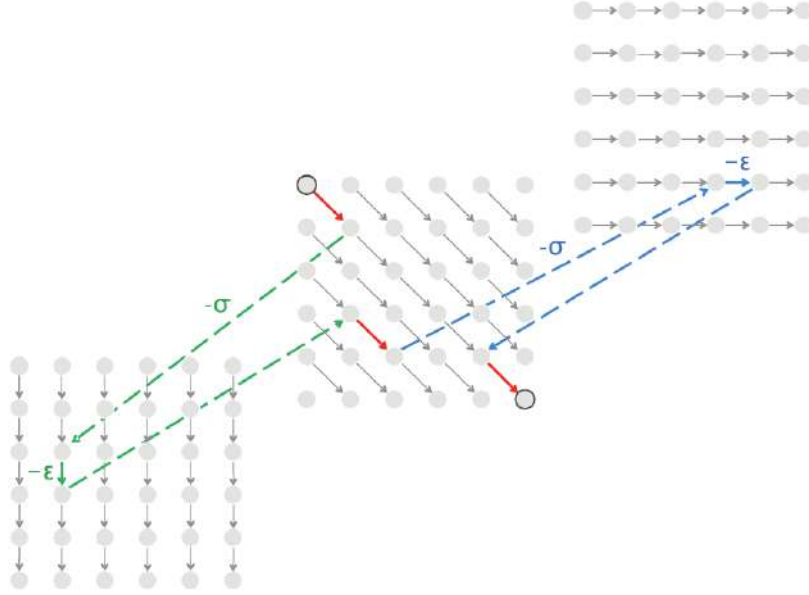
La vita in questa sorta di Manhattan 3D sarebbe molto difficile se non ci fosse un modo per spostarsi tra i livelli e di conseguenza poter intraprendere direzioni diverse. Per cui risolviamo il problema aggiungendo gli archi “mancanti”. In altre parole, andiamo ad aggiungere archi al fine di aprire e chiudere eventuali gap. Modelliamo questa situazione aggiungendo, ad ogni nodo  $(i, j)$  della matrice di mezzo un arco che lo collega al nodo  $(i + 1, j)$  della matrice di sotto e un arco che lo collega al nodo  $(i, j + 1)$  della matrice di sopra. Per aprire un gap bisognerà pagare una penalità iniziale  $\sigma$ , ma una volta fatto, tutte le successive indel saranno contate come  $\epsilon$ , per cui quando andremo a chiudere il gap la penalità totale sarà di  $\sigma - \epsilon(k - 1)$ , proprio come desiderato.

L’ultima cosa da considerare è che per chiudere un gap non è necessario pagare ulteriori penalità, per cui introduciamo archi con peso zero che vanno dal nodo  $(i, j)$  della matrice di sotto al nodo  $(i, j)$  della matrice di mezzo. La stessa cosa faremo per quanto riguarda il nodo  $(i, j)$  della matrice di sopra che sarà collegato con quello  $(i, j)$  della matrice di mezzo (in questo caso i direzionamenti sono importanti, ovvero  $lower \rightarrow middle$  e  $upper \rightarrow middle$ )

A questo punto però il sistema di equazione originale cambia per tenere conto di queste ultime modifiche, per cui è opportuno rivederlo:

$$\begin{aligned}
 \text{lower}_{i,j} &= \max \begin{cases} \text{lower}_{i-1,j} - \epsilon \\ \text{middle}_{i-1,j} - \sigma \end{cases} \\
 \text{middle}_{i,j} &= \max \begin{cases} \text{lower}_{i,j} \\ \text{middle}_{i-1,j-1} + \text{Score}(v_i, w_j) \\ \text{upper}_{i,j} \end{cases} \\
 \text{upper}_{i,j} &= \max \begin{cases} \text{upper}_{i,j-1} - \epsilon \\ \text{middle}_{i,j-1} - \sigma \end{cases}
 \end{aligned}$$

Per completezza vado ora a scrivere cosa comporta a livello di struttura dati:



La complessità di tutta questa impalcatura è  $O(nm)$  nel numero degli archi e se cerchiamo un longest path tra due stringhe, con questo sistema continueremo ad avere garanzia di ottenere in output il cammino ottimale.

## 5.5 Allineamento Space-Efficient

A questo punto siamo in grado di trovare il longest path corrispondente all'allineamento ottimale di due sequenze sia globalmente sia localmente. Il tempo di esecuzione è un tempo ragionevole (quadratico se  $n \approx m$ ). Dobbiamo ancora fare i conti con un problema di tipo spaziale: la matrice 3D è una soluzione affascinante ma allo stesso tempo dispendiosa, in quanto anche la complessità spaziale è quadratica. Vogliamo quindi ridurla affinché sia lineare nel numero di nodi. È chiaro che non si può avere tutto nella vita, per cui questo guadagno di spazio lo pagheremo raddoppiando il tempo di runtime.

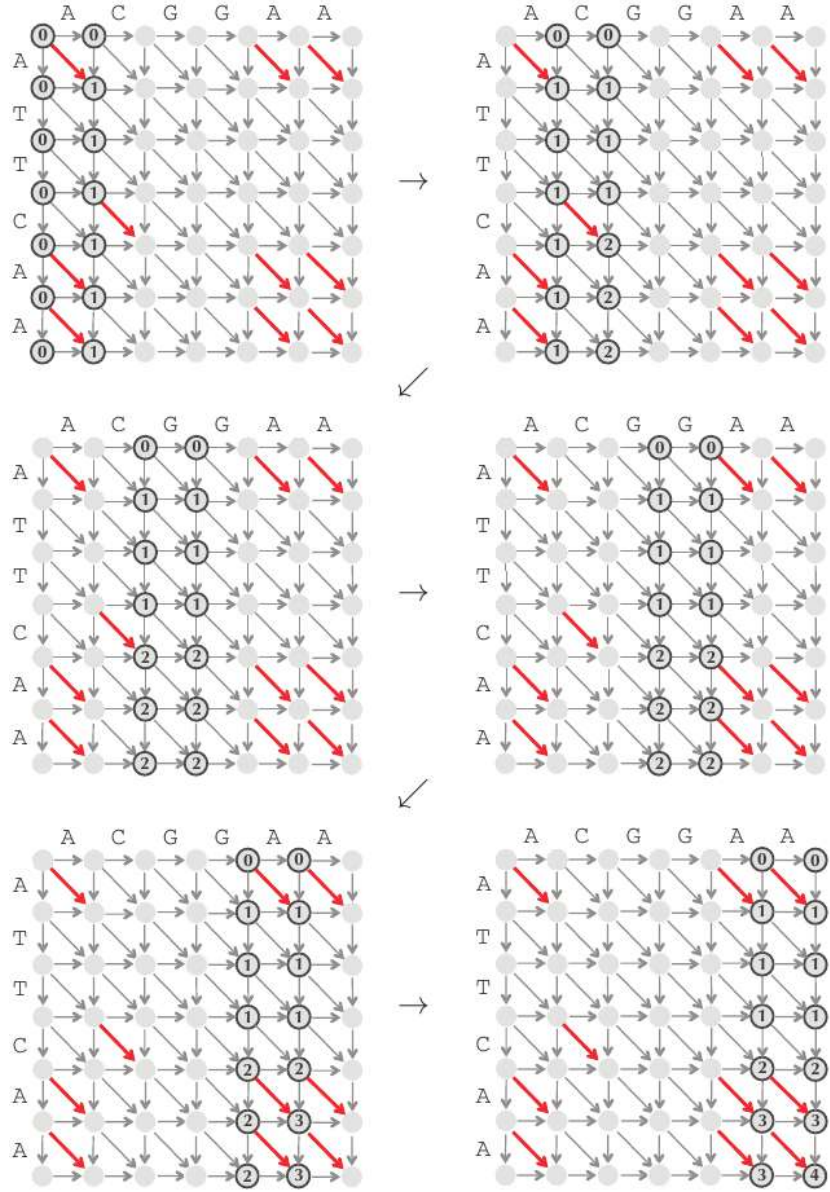
La nostra tecnica sarà quella di utilizzare un approccio “*Divide et Impera*”, ma andiamo per ordine:

- È intuitivamente chiaro che un qualsiasi problema, se siamo in grado di ridurlo a sottoproblemi più semplici può essere risolto in maniera più furba in termini di risorse occupate.
- Un approccio siffatto è il classico “*Divide et Impera*”, ovvero una fase dove ottengo varie soluzioni parziali e un'altra finale dove vado a combinarle per tornare al problema di partenza.
- Nel nostro caso, se anziché calcolare l'allineamento in sé ci limitassimo a calcolare solo gli score associati, allora siamo tutti d'accordo che in termini spaziali ci basterebbe 2 volte il numero di nodi della singola colonna, per cui di fatto avremmo raggiunto una complessità  $O(n)$ . Questa osservazione è giustificata dal fatto che per trovare tutti gli score della colonna  $j$  mi basta guardare solamente quelli della colonna precedente  $j - 1$ . Quindi non abbiamo bisogno di fare il discorso del backtracking per le colonne precedenti la  $j - 1$ .
- Avanzando di due in due non dobbiamo memorizzare nulla, a patto di accettare come conseguenza il fatto che una volta calcolati gli score dobbiamo rifare nuovamente l'allineamento “classico”.

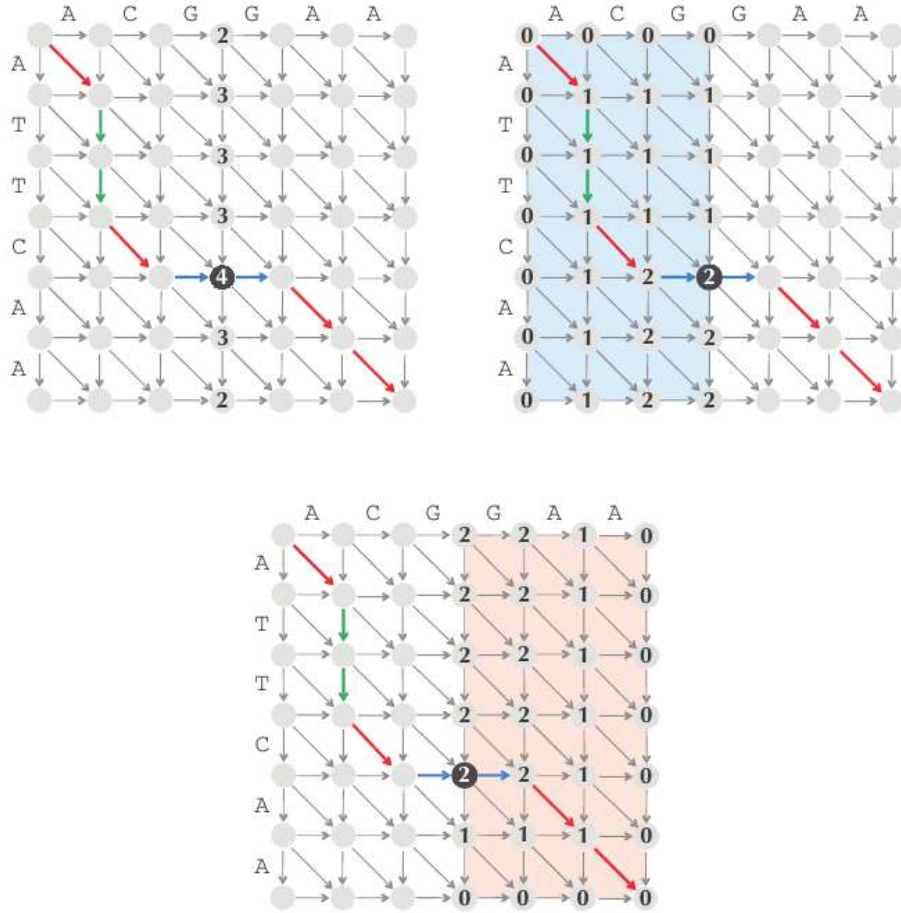
### 5.5.1 Middle Node Problem

La tecnica che andremo ad utilizzare è quella del nodo di mezzo. L'idea è che se dividiamo in 2 il nostro DAG di Manhattan sappiamo per certo che nella colonna di mezzo esisterà un nodo dal quale passa il cammino ottimale. Il nostro primo compito è quello di trovare questo nodo con complessità spaziale  $O(n)$ .

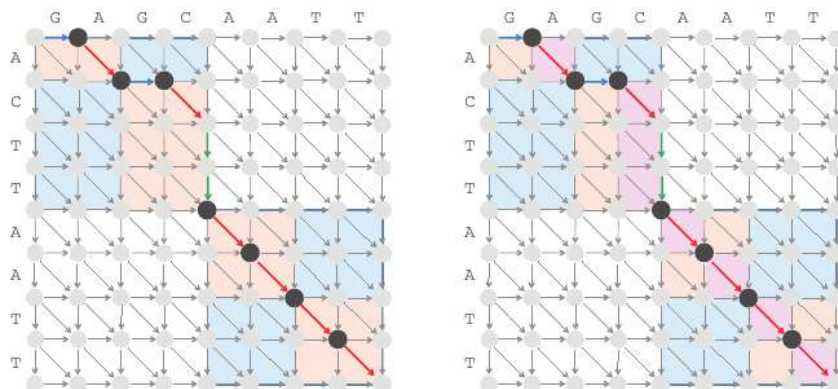
Per semplicità usiamo una funzione di score che da 1 quando c'è un match e zero altrimenti. Partiamo dall'inizio e calcoliamo solo gli score. Nella pagina seguente viene mostrato tutto il procedimento di calcolo degli score con la nuova funzione semplificata al fine di determinare il nodo di mezzo.



Se tutto questo è chiaro ecco il prossimo step. Dividiamo davvero in 2 il DAG e computiamo in parallelo il conto degli score. La prima parte di DAG la calcoliamo normalmente, per la seconda invertiamo tutte le direzioni degli archi e facciamo finta che la fine sia il nodo d'inizio, ricreando la situazione di sinistra. Una volta che entrambe le parti arriveranno alla colonna di mezzo, sommiamo semplicemente gli score parziali di quella colonna specifica e prendiamo il nodo dal punteggio maggiore: quello sarà il nostro nodo di mezzo. Graficamente :



Ed ecco che una volta fatto questo, possiamo ripetere lo stesso stratagemma dimezzando le due metà ottenute, e così via. Coloro che parlando bene direbbero che andiamo a computare in parallelo metà sempre più piccole *“In a binary search fashion”*:

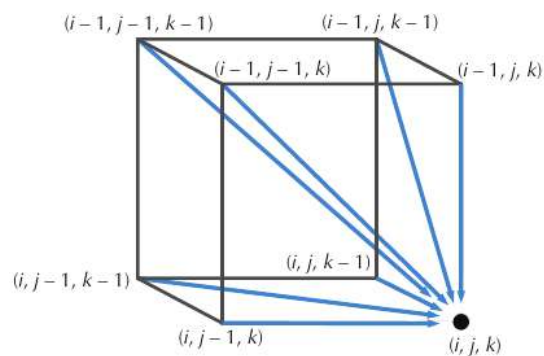


Di conseguenza, a patto di avere macchine che reggano il costo del parallelismo, possiamo risolvere il nostro problema di allineamento sia con un tempo d'esecuzione quadratico, sia con un consumo di memoria lineare.



## 5.6 MultiSequence Alignment

Finora abbiamo allineato solo coppie di sequenze. Cosa accadrebbe se volessimo allinearne  $n$  tutte insieme? Eh... Un bel casino, nel senso che il nostro DAG diventerebbe da 2D a multidimensionale. Le tecniche che abbiamo visto funzionano ancora ma sono dannatamente complicate da immaginarsi graficamente. Ad esempio, se volessimo allineare 3 sequenze il nostro DAG avrebbe la seguente forma:

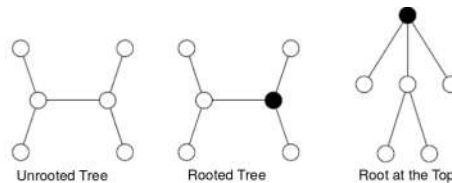


Insomma, Ufficio complicazioni affari semplici. Ma nel mondo reale questo è spesso richiesto... Fortunatamente, per il corso di Bioinformatica ci possiamo fermare qui. Nella prossima sezione andremo ad analizzare gli alberi filogenetici.

## 6 Alberi filogenetici

È importante in Bioinformatica ricostruire le relazioni tra specie diverse in base al loro DNA. Per fare ciò si utilizzano i cosiddetti alberi filogenetici, ossia un modo per graficare le discendenze. È un approccio molto utile quando non si conoscono per bene tutte le specie in gioco. Un esempio pratico è quello che ha portato alla classificazione del Panda Gigante. Per un secolo i biologi hanno cercato di capire se questo animale fosse discendente dagli orsi o dai procioni, questo perché il Panda ha caratteristiche un po' ibride tra le due famiglie (e.g. pur avendo dimensioni più vicine ad un orso, i Panda non vanno in letargo). Il problema è stato risolto nel 1985 da Steven O'Brien, utilizzando algoritmi di Bioinformatica. Ora, cerchiamo di fare una panoramica un po' più informatica.

### 6.1 Struttura



La struttura di un albero filogenetico è sostanzialmente la seguente:

- Le foglie rappresentano le specie esistenti
- I vertici interni rappresentano gli antenati
- I rami corrispondono a step evolutivi specifici
- La radice altro non è che il più antico antenato comune conosciuto.

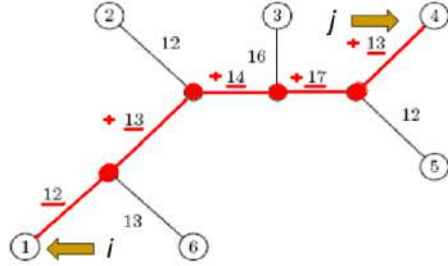
Ci possono essere inoltre 2 casi:

1. La radice non è nota: allora non sappiamo la vera forma dell'albero e dobbiamo costruircela in qualche modo
2. Sappiamo con certezza qual'è il più vecchio antenato comune. Possiamo quindi determinare intuitivamente la forma dell'albero in un modo stupido ma funzionante: immaginiamo di avere un modellino di questo albero. Se sappiamo qual'è la root allora basta prenderla con una mano ed alzare tutto il modellino di conseguenza. I successori si disporranno automaticamente nelle posizioni corrette.

I rami di un albero filogenetico hanno spesso dei pesi, che possono corrispondere al numero di mutazioni tra il nodo padre e il nodo figlio, oppure al tempo intercorso per l'evoluzione *padre*  $\rightarrow$  *figlio*.

In un albero  $T$  è spesso interessante calcolare  $d_{i,j}(T)$ , ovvero la distanza tra due nodi  $i$  e  $j$  (che può essere un singolo numero se i nodi sono direttamente connessi, oppure un percorso se ci sono nodi intermedi tra i due).

Un esempio è il seguente:



$$d_{1,4} = 12 + 13 + 14 + 17 + 12 = 68$$

Ora, al fine di evitare future incomprensioni, facciamo un distinguo:

- Definiamo  $d_{i,j}$  la lunghezza di un percorso  $i \rightarrow j$  in un albero ignoto
- Definiamo  $D_{i,j}$  la distanza tra specie (nota perchè calcolabile)

Il nostro scopo sarà quello di far coincidere queste due distanze al fine di costruire il nostro albero filogenetico.

Date  $n$  specie, possiamo costruire una matrice di distanze  $D_{i,j}$ . L'evoluzione tra esse sarà determinata dalle differenze nel DNA che possiamo calcolare.

Ora, prima di partire in quarta, osserviamo che in caso di 3 foglie e 1 vertice comune sappiamo risolvere il problema di fitting:

Tree reconstruction for any 3x3 matrix is straightforward.

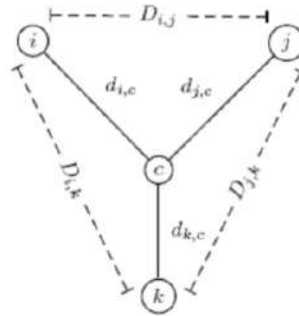
We have 3 leaves  $i, j, k$  and a center vertex  $c$ .

**Observe:**

$$d_{i,c} + d_{j,c} = D_{i,j}$$

$$d_{i,c} + d_{k,c} = D_{i,k}$$

$$d_{j,c} + d_{k,c} = D_{j,k}$$



Di conseguenza, sarebbe opportuno ridursi sempre ad un caso siffatto per poter costruire un buon albero filogenetico. Infatti, possiamo ricavarci il valore dei  $d_{i,j}$  a partire dalle  $D_{i,j}$  in questo modo

$$d_{i,c} = \frac{D_{i,j} + D_{i,k} - D_{j,k}}{2}$$

$$d_{j,c} = \frac{D_{i,j} + D_{j,k} - D_{i,k}}{2}$$

$$d_{k,c} = \frac{D_{i,k} + D_{j,k} - D_{i,j}}{2}$$

Consiglio: per ricordarsi la formula basta procedere nel seguente modo:

- Data una distanza  $d_{x,c}$  i membri della frazione che si sommano sono tutti quelli in cui comparire la  $x$  come pedice mentre si sottrae l'unico membro dove non compare.
- Per quanto riguarda la divisione per 2, è necessaria perchè altrimenti calcoleremmo un pezzo di distanza due volte.

Ora, per un albero con  $n$  foglie il numero di archi sarà pari a  $2n-3$ . Esso equivale al calcolo di un sistema lineare di equazioni pari in numero ad un coefficiente binomiale  $\binom{n}{2} = \frac{n(n+1)}{2}$ .

Questo set di equazioni non è sempre risolvibile per  $n > 3$ .

### 6.1.1 Matrici di distanza

Per ogni foglia etichettata con una lettera (A,B...) possiamo costruire quella che viene chiamata Matrice di distanza, ovvero una matrice dove ogni valore indica quanto dista l'etichetta della riga  $i$  dall'etichetta della colonna  $j$ .

Questo tipo di matrici si possono suddividere a loro volta in due tipi differenti:

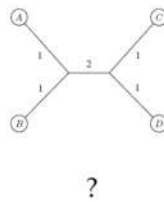
- Matrici Additive, quando  $D_{i,j} = d_{i,j}$
- Matrici non additive, altrimenti

Un esempio specifico è il seguente:

Additive Matrix		A	B	C	D
	A	0	2	4	4
	B	2	0	4	4
	C	4	4	0	2
	D	4	4	2	0

Nonadditive Matrix		A	B	C	D
	A	0	2	2	2
	B	2	0	3	2
	C	2	3	0	2
	D	2	2	2	0



Per ricordarlo in una forma meno formale, se non c'è un minimo di progressività tra le distanze presenti in una riga, allora non sappiamo risalire all'albero, in quanto ci possono essere varie soluzioni. Intuitivamente si potrebbe pensare di

cercare di accoppiare le foglie più vicine fino a ridurre tutto alla radice. Quello sarebbe l'albero definitivo. Peccato che non è vero che foglie più vicine in termini di DNA sono effettivamente discendenti da uno stesso precursore.

## 6.2 Neighbor-Joining Algorithm

Passiamo ad analizzare la prima tecnica sviluppata per la risoluzione del problema Distance based Phylogeny, ovvero l'algoritmo di join dei vicini. È stato inventato nel 1987 dai giapponesi Naruya Saitou e Masatoshi Nei. Se è vero che non possiamo dare per scontato che foglie vicine siano anche discendenti da un nodo comune, quello che viene fatto è trasformare la matrice di distanza  $D$  in modo che si possa fare questo ragionamento. L'idea è quella di trovare due foglie vicine tra loro ma lontane da tutte le altre. Si viene così a creare un'euristica che nella pratica funziona bene.

### 6.2.1 Additive Phylogeny

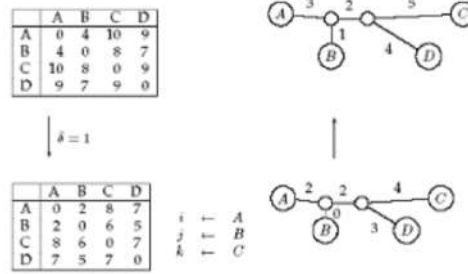
#### 6.2.2 Triple degeneri

Una cosa simpatica in questo tipo di problemi è il concetto di Triple degeneri. Formalmente una tripla degenera non è altro che una tripla  $0 < i, j, k < n$  tale per cui  $D_{i,j} + D_{j,k} = D_{i,k}$ . Graficamente questo vuol dire uno dei due casi:

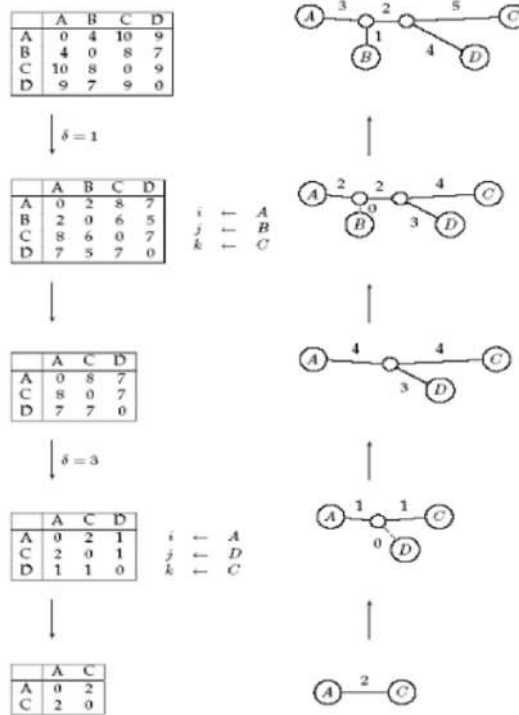


L'interpretazione è più o meno la seguente: ogni elemento  $j$  di una tripla degenera giace in un percorso evolutivo che collega  $i$  con  $k$  (oppure è attaccato a questo percorso con un ramo di peso nullo).

Questo apre le porte ad una possibilità molto interessante per la costruzione di un albero filogenetico. Come al solito, il trucco è ricondurci ad una situazione che sappiamo gestire e poi risolvere quel problema. Nel nostro caso se vogliamo trovare le triple degeneri ma non ci sono, semplicemente ce le troviamo. Un esempio è il seguente:



In questo caso abbiamo scelto una quantità  $\delta = 1$  e abbiamo tolto da ogni distanza il valore  $2\delta$ . Nel disegno, ai pesi è stato sottratto il valore  $\delta$ , ma la cosa interessante è che già in questo primo step abbiamo creato una nostra tripla degenera fittizia, (A,B,nodo intermedio). Possiamo iterare nuovamente questo processo fin quando non terminiamo le triple degeneri. A questo punto abbiamo individuato tutta la gerarchia dei rami, per cui basta ripercorrere il tutto all'indietro e si avrà il nostro albero. Ecco l'esempio completo:



L'algoritmo che implementa tutto questo procedimento è il seguente:

### AdditivePhylogeny( $D$ )

if  $D$  is a  $2 \times 2$  matrix

$T$  = tree of a single edge of length  $D_{1,2}$

return  $T$

if  $D$  is non-degenerate

$\delta$  = trimming parameter of matrix  $D$

for all  $1 \leq i \neq j \leq n$

$$D_{ij} = D_{ij} - 2\delta$$

else

$$\delta = 0$$

Find a triple  $i, j, k$  in  $D$  such that  $D_{ij} + D_{jk} = D_{ik}$

$$x = D_{ij}$$

Remove  $j^{th}$  row and  $j^{th}$  column from  $D$

$T$  = AdditivePhylogeny( $D$ )

Add a new vertex  $v$  to  $T$  at distance  $x$  from  $i$  to  $k$

Add  $j$  back to  $T$  by creating an edge  $(v, j)$  of length 0

for every leaf  $l$  in  $T$

if distance from  $l$  to  $v$  in the tree  $\neq D_{l,j}$

output "matrix is not additive"

return

Extend all "hanging" edges by length  $\delta$

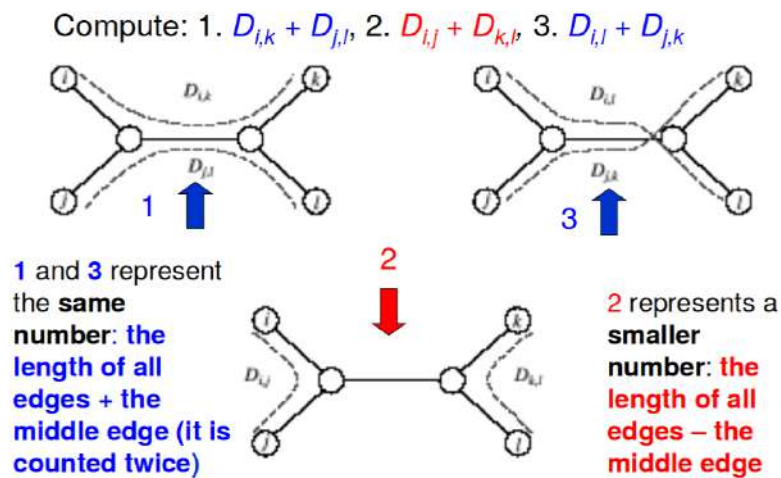
return  $T$

### 6.2.3 Four point condition

La four point condition è di fatto un teorema che ci garantisce l'additività di una matrice  $D$ . Essa stabilisce che una matrice quadrata è additiva se per ogni quadrupla  $1 < i, j, k, l < n$  e considerate le tre somme :

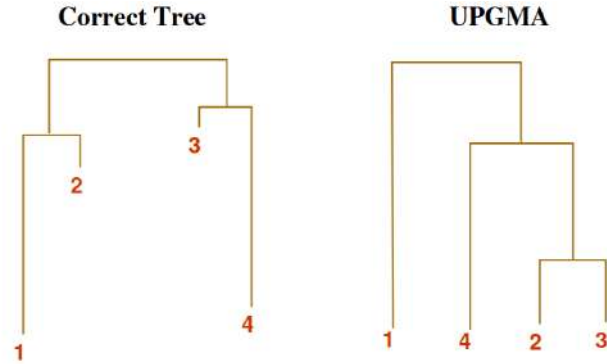
1.  $D_{i,j} + D_{k,l}$
2.  $D_{i,k} + D_{j,l}$
3.  $D_{i,l} + D_{j,k}$

Ci sono almeno due di esse che sono uguali e la terza minore delle altre. Il disegno è autoesplicativo :





### 6.3 UPGMA



L'algoritmo UPGMA è sostanzialmente un algoritmo di clustering gerarchico, che computa le distanze tra cluster utilizzando la distanza media tra tutte le coppie dei cluster in esame. Dati due cluster disgiunti  $C_i, C_j$  la distanza  $d_{i,j}$  è calcolata nel modo seguente:

$$d_{i,j} = \frac{1}{|C_i| \cdot |C_j|} \sum_{p \in C_i} \sum_{q \in C_j} d_{p,q}$$

L'algoritmo è il seguente :

**Initialization:**

Assign each  $x_i$  to its own cluster  $C_i$ .

Define one leaf per sequence, each at height 0.

**Iteration:**

Find two clusters  $C_i$  and  $C_j$  such that  $d_{i,j}$  is minimal.

Let  $C_k = C_i \cup C_j$ .

Add vertex connecting  $C_i$  to  $C_j$  and place it at height  $d_{i,j}/2$ .

Delete  $C_i$  and  $C_j$ .

**Termination:**

When a single cluster remains.

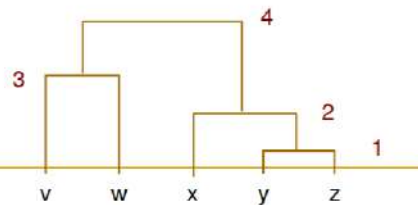
Comunque con un esempio il tutto si capisce meglio

	v	w	x	y	z
v	0	6	8	8	8
w		0	8	8	8
x			0	4	4
y				0	2
z					0

	v	w	xyz
v	0	6	8
w		0	8
xyz			0

	vw	xyz
vw	0	8
xyz		0

	v	w	x	yz
v	0	6	8	8
w		0	8	8
x			0	4
yz				0



Il problema di UPGMA è che gli alberi in output sono ultrametrici: in sostanza vi è la stessa distanza tra la root e ciascuna delle foglie.

## 6.4 Character Based Phylogeny

Questo tipo di approccio si basa su un'idea semplice : se prendiamo un gene di  $m$  nucleotidi in  $n$  specie, allora possiamo di fatto generare una matrice di allineamento  $n \times m$ . Inoltre, è possibile generare una matrice di distanza a partire da una matrice di allineamento. È necessario fare attenzione al fatto che data una matrice di distanza non è sempre possibile ricondursi ad un allineamento corretto (ci possono essere in sostanza più possibilità).

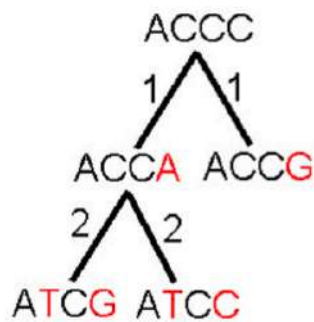
Abbiamo tutti gli elementi per comprendere la Character Based Phylogeny : anzichè utilizzare le distanze, ci focalizzeremo sulle differenze in termini di allineamento tra sequenze. In altre parole, il focus sarà su specifiche basi oppure su specifiche caratteristiche biologiche (es. numero di gambe, colore degli occhi, presenza di ali...). Solo a questo punto useremo queste informazioni per il calcolo delle distanze e successivamente daremo un valore ai pesi dei rami.

Settando la lunghezza di un ramo come la distanza di Hamming, definiremo lo **Score di parsimonia** dell'albero come la somma dei pesi dei rami.

## 6.5 Small Parsimony Problem

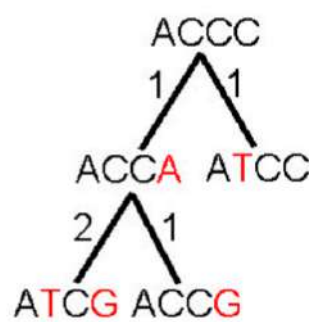
Per spiegare il concetto di parsimonia è interessante partire dal principio del **Rasoio di Occam**<sup>6</sup> che fa più o meno così : “A parità di fattori la spiegazione più semplice è da preferire”. Senza stare qui a vendere fumo, possiamo fare un'estrema sintesi dicendo che per realizzare un albero, è opportuno seguire la strada più semplice, per cui le specie con una minore presenza di mutazioni tra loro solo le preferibili per essere accoppiate. Di conseguenza quello che in linea teorica andiamo a preferire è un albero di parsimonia con lo score minore possibile. Per fare un esempio:

### Less Parsimonious



Score = 6

### More Parsimonious



Score = 5

<sup>6</sup>[https://it.wikipedia.org/wiki/Rasoio\\_di\\_Occam](https://it.wikipedia.org/wiki/Rasoio_di_Occam)

Ecco come imposteremo il nostro problema :

- Prima di tutto ogni foglia dell'albero sarà numerata da una stringa di  $m$  caratteri.
- L'output dell'algoritmo è l'albero etichettato in modo da minimizzare lo score di parsimonia
- Possiamo inizialmente fare un'assunzione e dire che in realtà ci possiamo limitare ad usare etichette di un solo carattere, perchè i caratteri in una stringa sono indipendenti.

Per uscire dalla confusione che inevitabilmente ha creato questa parte (colpa delle slide) facciamo un passo in avanti formalizzando più in modo informatico:

- In input avremo una matrice di distanze quadrata, nella quale sarà riportato il costo di una lettera  $x$  di mutare in una lettera  $y$ .
- Nello Small Parsimony Problem tutti questi costi sono per semplicità fissati ad 1. Di conseguenza, la funzione di score assumerà valore 1 se i caratteri  $i$  e  $j$  sono diversi, 0 altrimenti.
- Nel caso in cui volessimo fare ulteriori assunzioni sul comportamento delle sequenze (magari sappiamo che la lettera  $x$  muta più facilmente in una  $y$  che in una  $z$ ) allora si cambia la funzione di score dimodochè per ogni trasformazione ci sia un costo diverso.
- Il principio di base, qualunque scelta si compia, resta quello di organizzare i vari nodi dell'albero in modo tale da minimizzare il nostro score totale di parsimonia.

Graficamente:

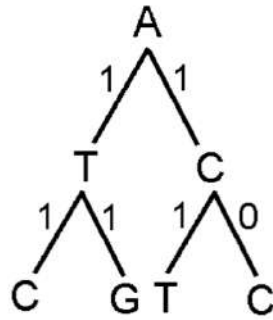
**Small Parsimony Problem**

	A	T	G	C
A	0	1	1	1
T	1	0	1	1
G	1	1	0	1
C	1	1	1	0

**Weighted Parsimony Problem**

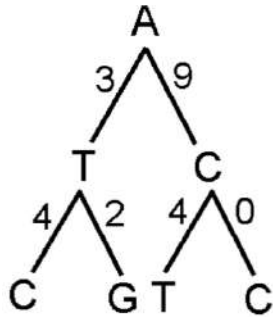
	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

Per essere pedanti al 100% :

**Small Parsimony Scoring Matrix**

	A	T	G	C
A	0	1	1	1
T	1	0	1	1
G	1	1	0	1
C	1	1	1	0

Small Parsimony Score: 5

**Weighted Parsimony Scoring Matrix**

	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

Weighted Parsimony Score: 22

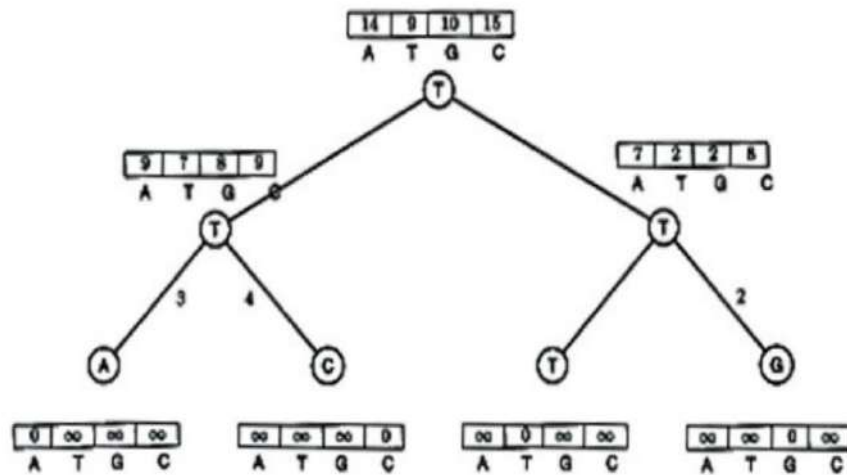
## 6.5.1 Algoritmo di Sankoff

Allora, questo algoritmo si basa sul memorizzare per ogni nodo le varie distanze con le altre lettere. Ovviamente questo tipo di ricerca è esaustivo, perchè una volta calcolate tutte le possibili combinazioni basta prendere sempre i minimi ed il risultato è garantito.

Inizio con la matrice in input:

$\delta$	A	T	G	C
A	0	3	4	9
T	3	0	2	4
G	4	2	0	4
C	9	4	4	0

Il risultato è il seguente:



Adesso arriverà il commento chiarificatore:

Ogni foglia viene etichettata banalmente con la lettera corrispondente, per cui le altre possibili sono settate ad  $\infty$ . Salendo di un livello vado a spiegare il sottoramo sinistro: quello che si fa è calcolare le somme delle distanze tra i 2 figli ed ogni possibile altra lettera. Per fare un esempio pratico, La A dalla A dista 0, mentre la C dalla A dista 9. Per cui andiamo a scrivere 9 nella casella del nodo padre di A e C.

Stessa cosa facciamo per la T : La A dalla T dista 3, mentre la C dalla T dista 4. Di conseguenza, La T del nodo padre avrà valore 7. E così via fin quando

non abbiamo finito i conti del nodo padre.

Il procedimento è analogo visto dal punto di vista di due nodi figli ed un nodo genitore, per cui siamo in grado di etichettare l'intero albero generalizzando.

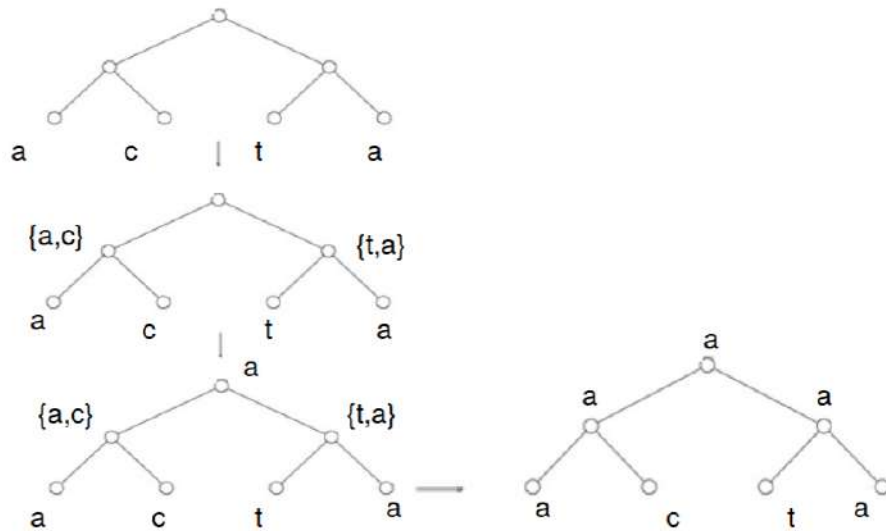
**Attenzione!** Una volta finito il primo step Bottom Up si ricomincia il percorso Top Down e si scelgono le vere etichette per gli altri nodi prendendo semplicemente i minimi presenti.

### 6.5.2 Algoritmo di Fitch

È un algoritmo che risolve lo Small Parsimony Problem adottando un approccio un po' diverso. In questo caso vale la pena raccontare prima di cosa si tratta perchè è abbastanza chiaro.

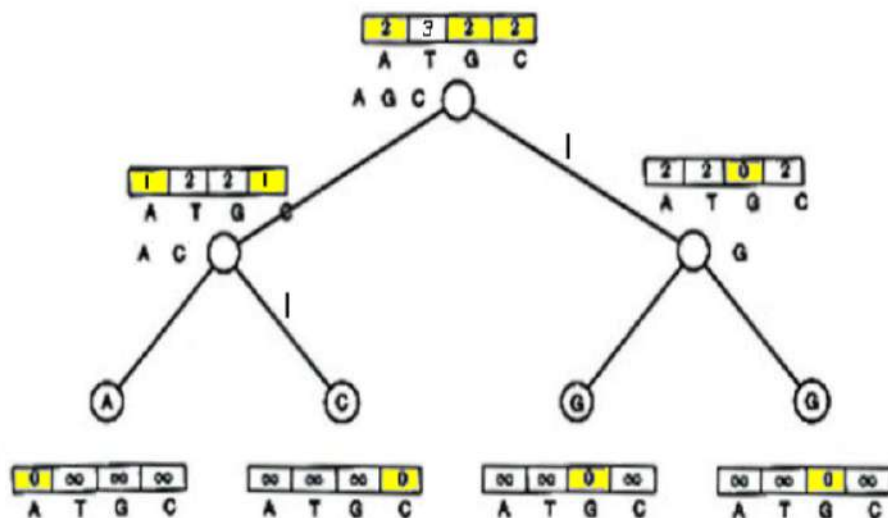
Come prima cosa si assegna alle foglie un insieme contenente un singolo carattere. Successivamente il nodo padre di due figli non è altro che l'unione dei due insiemi singleton. Da quel momento in poi, tutti i nodi dei livelli superiori si trovano attraverso l'intersezione dei nodi figli. Se per caso l'intersezione è vuota, allora prendo l'unione e ricomincio.

Esempio chiarificatore:



## 6.5.3 Sankoff vs Fitch

Entrambi i due algoritmi hanno complessità  $O(nk)$ . In realtà si dimostra che sono esattamente la stessa cosa, ma non voglio fare tecnicismi, vado solo a descrivere la sostanza delle cose : c'è una corrispondenza di fatto, ovvero quando capita in Fitch che l'intersezione esiste ed è unica allora quella lettera corrispondente è proprio la lettera che in Sankoff ha la somma delle distanze minima (dai figli). Quando invece questa intersezione è vuota, devo prendere l'unione: in questo caso tutti gli elementi dell'unione hanno in Sankoff distanza minima e uguale. Graficamente:





## 6.6 Large Parsimony Problem

### 6.6.1 Formalizzazione del Problema

Il Large Parsimony Problem è una generalizzazione dello Small Parsimony Problem applicato ad  $n$  sequenze anziché a 2. Di conseguenza avremo una matrice  $n \times m$ , dato che sono  $n$  sequenze da  $m$  caratteri l'una. È abbastanza chiaro che in questo caso un approccio brute force come quelli adottati finora non è la scelta più saggia fattibile. Dobbiamo necessariamente accettare il fatto di saltare qualche confronto (in realtà parecchi, ma vabbè...) ed implementare sistemi di Branch & Bound oppure qualche euristica.

Per la precisione, Il LPP è un problema NP-Completo. Infatti il numero di possibili alberi con root con  $n$  foglie è:

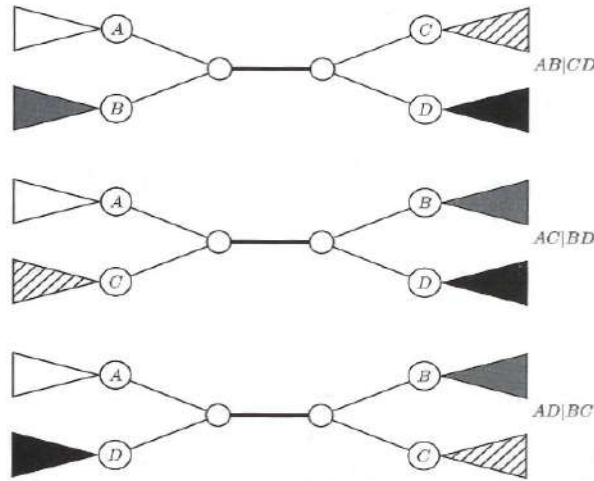
$$\frac{(2n-3)!}{2^{n-2}(n-2)!} \quad (4)$$

Per quanto riguarda la situazione senza root, non cambia poi chissà cosa:

$$\frac{(2n-5)!}{2^{n-3}(n-3)!} \quad (5)$$

La ricerca esaustiva è possibile solo per  $n < 10$

### 6.6.2 Nearest Neighbor Interchange



Questo algoritmo è abbastanza semplice, ma funziona bene se si accetta di non ottenere necessariamente la soluzione ottima.

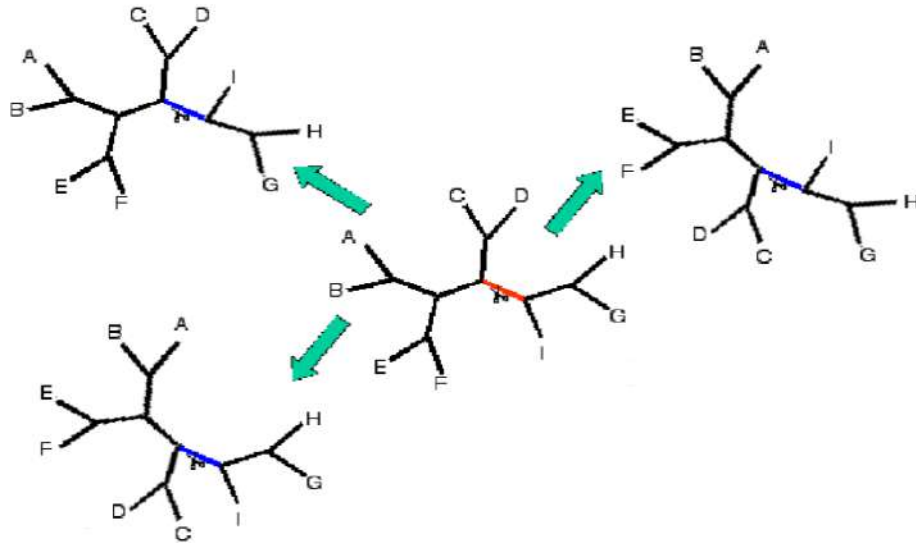
Di fatto quello che facciamo è partire da un albero casuale per poi andare a fare swapping dei sottoalberi nella speranza che si riduca lo score di parsimonia.

In pratica cerchiamo di portarci nella situazione in figura, nella quale sono

possibili 3 scambi (in quanto gli altri invertirebbero solo gli ordini ma non influirebbero sui punteggi).

Attenzione ! Potremmo rimanere bloccati in un minimo locale, per cui quello che si fa è partire da alberi randomici diversi ed applicare più volte l'algoritmo. Compensiamo quindi i pochi confronti che facciamo col numero di volte che facciamo le prove su alberi diversi. Se però molti tentativi convergono è probabile trovarsi davanti ad una buona soluzione.

Un altro esempio è il seguente:



### 6.6.3 Problemi della parsimonia in generale

Non abbiamo ancora risolto il problema in maniera definitiva (e non lo faremo in realtà). Questo tipo di approccio basato su parsimonia non tiene conto dei casi di omeoplasia.

L'omeoplasia è una condizione per cui ci possono essere multiple mutazioni evolute da un comune predecessore.

Given:

1. CAGCAGCAG
2. CAGCAGCAG
3. CAGCAGCAGCAG
4. CAGCAGCAG
5. CAGCAGCAG
6. CAGCAGCAG
7. CAGCAGCAGCAG

Most would group 1, 2, 4, 5, and 6 as having evolved from a common ancestor, with a single mutation leading to the presence of 3 and 7.

Il problema è che la parsimonia tende ad accoppiare le sequenze in maniera controintuitiva. Ecco perchè per fare un buon lavoro dobbiamo sperare nella convergenza di metodi diversi ad un unico risultato.

## 7 Algoritmi di Clustering

In questa sezione andremo ad approfondire una tecnica di apprendimento non supervisionato nota come Clustering. Tecniche come queste si basano sul fatto di non dover basarsi su parametri in input per andare ad effettuare una classificazione, a patto di scegliere accuratamente un criterio il più oggettivo possibile per fare i confronti.

In sostanza quello che vogliamo fare è partire da un insieme di dati e partizionarlo in sottoclassi che per noi hanno un significato. Tutto questo in modo automatico.

Bisogna notare la sottile differenza tra Clustering e Classificazione: il primo è un processo automatico, dove noi non sappiamo le categorie che verranno fuori, mentre la seconda deve essere basata su un qualche tipo di valore atteso con lo scopo di minimizzare la distanza tra esso e il training set.

Alla luce di questo possiamo quindi definire ciò che intendiamo con “Buon clustering”, ovvero un tipo di clustering tale che gli elementi comuni ad una stessa classe abbiamo alta similarità, mentre quelli in classi diverse abbiamo una bassissima similarità. Inoltre un buon clustering dovrebbe permettere di identificare i pattern nascosti tra i dati. Dal punto di vista del data mining ci aspettiamo ulteriori proprietà:

- Scalabilità
- Capacità di lavorare con attributi di tipo diverso
- Capacità di individuare cluster di forma qualsiasi
- Requisiti minimi in termini di conoscenza delle proprietà dei dati
- Interpretabilità e Usabilità

E chi più ne ha più ne metta...

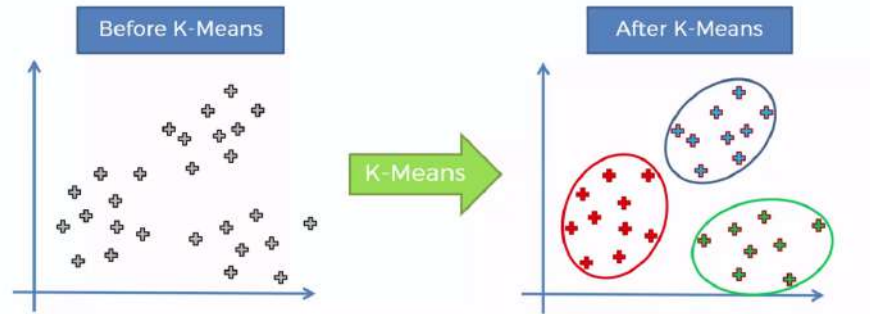
L'uso degli algoritmi di clustering spazia dai rilievi geografici alle reti neurali, intrecciandosi con molte altre discipline. Solo recentemente in Bioinformatica si è pensato di utilizzare un simile approccio per la risoluzione di problemi particolari.

Ci sono 5 tipi di algoritmi di clustering :

1. Algoritmi di partizionamento
2. Algoritmi gerarchici
3. Algoritmi basati su densità
4. Algoritmi basati su griglia
5. Algoritmi basati su modelli

Ora, in teoria queste categorie sono autoesplicative, comunque andiamo adesso a dettagiarle tutte quante con più spirito critico

## 7.1 Algoritmi di partizionamento



Gli algoritmi di partizionamento partono da un database  $D$  contenente  $n$  oggetti e costruiscono  $k$  classi a partire da esso.

Ci sono vari tipi di questi algoritmi, sostanzialmente divisi in quelli che fanno ricerca esaustiva, e quindi brute force, e quelli che attraverso qualche criterio particolare. Nel nostro caso è interessante parlare solo degli ultimi, dato che risolvono un problema di ottimizzazione.

In particolare quello che andiamo a fare è minimizzare una certa funzione di score, e la più usata è quella dell'errore quadratico:

$$E = \sum_{i=1}^k \sum_{p \in C_i} \|p - m_i\|^2 \quad (6)$$

Fondamentalmente questa classe di algoritmi si divide in K-means e K-medoids. Il k-means consiste sostanzialmente in 4 steps:

1. Gli oggetti del dataset sono partizionati in  $k$  cluster
2. Computa i centri dei cluster attraverso la media degli elementi del cluster.
3. Assegnare ogni elemento al cluster dal centro più vicino
4. Continuare dallo step 2 fin quando l'algoritmo converge (non ci sono cambi significativi di cluster)

Andiamo ora a commentare quelli che sono gli aspetti positivi e negativi di questa tecnica. Iniziamo dai pro:

- È relativamente efficiente, infatti ha complessità  $O(tkn)$ , dove  $n$  è il numero di oggetti,  $k$  è il numero di cluster e  $t$  è il numero di iterazioni.
- Trova sempre un ottimo locale. Questo non sempre corrisponde ad un ottimo globale ma è possibile fare inferenza con altre tecniche (e.g. Algoritmi genetici)

Per quanto riguarda i contro invece diremo che :

- È applicabile solo quando si riesce a dare la definizione di “media”. Questo è un problema per i dati categorici (e.g Non ha senso fare la media tra una persona con gli occhi verdi di Scarlett Johansson<sup>7</sup> e quelli marroni di Morena Baccarin<sup>8</sup>).
- C'è bisogno di fornire a priori il numero di cluster  $k$  che vogliamo ottenere in output.
- Essendo un algoritmo basato sul concetto di media se i dati sono rumorosi (non precisi) allora i centri e le conseguenti classificazioni saranno sfasate.
- Non è adatto per trovare cluster di forme concave (o comunque di forme diverse da quella sferica)

Gli algoritmi k-metoids invece anzichè cercare centri “fittizi” cercano di utilizzare direttamente i rappresentanti del dataset come elementi centrali. Vediamone alcuni esempi.

#### 7.1.1 PAM (Partitioning Around Metoids)

Il metodo PAM consiste nello scegliere inizialmente  $k$  elementi rappresentativi dal dataset in input. A questo punto si procede così :

- Ogni elemento  $h$  diverso dai  $k$  scelti in precedenza viene confrontato con ciascun elemento  $i \in k$  in modo da calcolare quanto costa fare un eventuale cambio di rappresentante.
- Per ciascuna coppia  $i, h$ , se il costo di cambio eventuale è negativo, allora faccio lo scambio e a questo punto riassegno tutti gli elementi dei cluster in modo da rispettare “il nuovo centro”.
- Si ripete la procedura fin quando l'algoritmo non converge, ovvero non ci sono più cambi sostanziali tra cluster.

---

<sup>7</sup><http://tiny.cc/lo6efz>

<sup>8</sup><http://tiny.cc/pq6efz>

### 7.1.2 CLARA (Clustering Large Application)

È chiaro che applicare PAM a grossi dataset è un problema a livello di complessità computazionale, in quanto l'algoritmo è  $O(n^2)$ . Per questo è stato ideato CLARA, che sostanzialmente crea piccole partizioni randomiche e su queste ultime esegue un'istanza di PAM.

**Weaknesses:**

- Pur comportandosi mediamente meglio di PAM, è una tecnica che scala poco all'aumentare delle dimensioni del dataset
- La scelta iniziale è proprio il discriminante che può peggiorare la qualità complessiva del clustering.

### 7.1.3 CLARANS (Clustering Large Applications based upon Randomized Search)

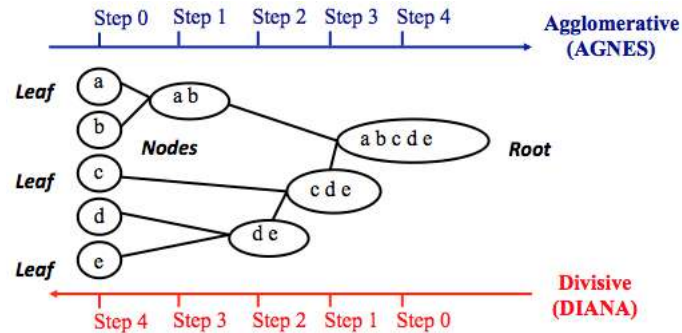
Questo algoritmo trasforma il processo di clustering in una ricerca in un grafo, nel quale ogni elemento può essere la soluzione. Il grafo in questione è costituito da un insieme di nodi. Secondo questa astrazione, ogni nodo non è altro che un insieme di  $k$  metoids. Due nodi sono detti vicini solo se hanno la differenza di al più un elemento.

In ogni iterazione, l'algoritmo considera un set randomico di nodi vicini, e tra essi determina qual'è il migliore in termini di  $k$ -metoids. Nel caso in cui non ci siano nodi migliori di altri allora si è trovata una convergenza ad un ottimo locale. A questo punto si potrebbe finire qui.

Per avere più certezze, quello che si fa di solito è lanciare istanze multiple dell'algoritmo (in modo che le scelte dei nodi siano sempre diverse) e andare a studiare i vari nodi di convergenza. Intuitivamente, se abbiamo avuto  $n$  risposte uguali, allora potremmo trovarci davanti all'ottimo globale.

In ogni caso questo algoritmo scala molto meglio dei precedenti.

## 7.2 Algoritmi gerarchici

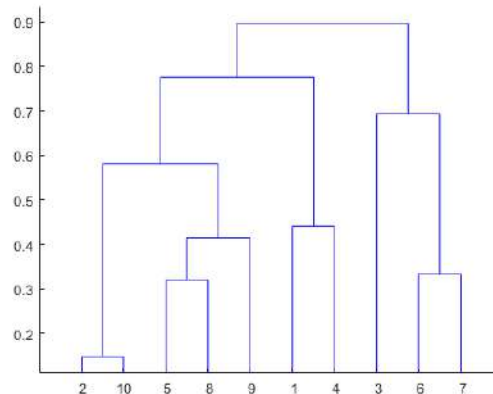


Gli algoritmi gerarchici si dividono sostanzialmente in Top-Down e Bottom-Up. I primi partono generalmente da un singolo dataset e tentano di scomporlo in pezzi più piccoli, anche se spesso questo è difficile senza introdurre una opportuna parametrizzazione. I secondi invece partono dal considerare ogni singolo elemento come cluster unitario, e poi tentano di accorpare più partizioni a formare cluster sempre più grandi.

Entrambe queste tecniche utilizzano una matrice di distanze per poter decidere come effettuare i raggruppamenti, di conseguenza non sono richiesti a priori il numero di cluster che ci si aspetta in output. Andiamo ora a descriverne alcuni.

### 7.2.1 AGNES (Agglomerative Nesting)

AGNES è un algoritmo gerarchico di tipo Bottom-Up, la sua idea iniziale è quella di fare il merge dei cluster che hanno meno disparità fin quando non si arriva ad un unico cluster finale. L'output di questo tipo di algoritmi è il cosiddetto "Dendrogramma" ovvero un albero di clusters. Ecco una foto di esempio:





Ora, è necessario introdurre il concetto di dendrogramma per approfondire un attimo la questione del clustering.

È chiaro che al termine di un algoritmo gerarchico Bottom Up si ottenga un solo cluster (di root). Spetta quindi all'analista andare a spezzare il dendrogramma al fine di ottenere un numero di cluster adatto a descrivere il fenomeno analizzato o più banalmente che abbia un senso.

### 7.2.2 DIANA (Divisive Analysis)

DIANA invece utilizza un approccio Top Down, e già questo basterebbe per finire la trattazione. Per cui adesso vado a descrivere le debolezze:

- Intanto non scala, ha complessità  $O(n^2)$
- Ancora peggio, non c'è l'UNDO, per cui i raggruppamenti successivi dipendono da quelli attuali.

### 7.2.3 CURE (Clustering Using REpresentatives)

CURE è un algoritmo che prova ad ottimizzare la categoria di questo tipo di algoritmi. In sostanza si diversifica per il tipo di operazioni iniziali. Quello che si fa infatti è scegliere a caso un sottoinsieme del dataset  $r$  e partizionarlo in un numero fisso di clusters. A questo punto si applica un'istanza di algoritmo di clustering per ognuno di questi nuovi insiemi. L'unica accortezza in questo caso è quella di eliminare gli elementi anomali (perché distanti dal trend degli altri) detti outliers.

A questo punto quello che si fa è andare a fare il merge di clusters vicini semplicemente avvicinando i loro centri, in una logica di "gravity center".

## 7.3 Algoritmi basati su densità

Gli algoritmi gravity-based si basano sul fare i raggruppamenti localmente, ed uniformare il resto mediante il concetto di densità. Questo tipo di tecnica segue fedelmente la distribuzione dei dati nello spazio e questo consente di individuare cluster di ogni forma. Inoltre sono in grado di lavorare bene anche su dataset rumorosi (con errori). Andiamo ora ad approfondire la conoscenza dei più famosi.

### 7.3.1 DBSCAN

DBSCAN è un algoritmo di densità che per funzionare necessita di due parametri fondamentali: il raggio  $r$  di vicinanza di un generico punto e il minimo numero di punti che devono essere compresi nell'area determinata dal cerchio di raggio  $r$  costruito intorno al punto in esame per poter essere considerati suoi vicini.

Di conseguenza dati due punti  $p, q$ , diremo che  $p$  è direttamente raggiungibile da  $q$  se:

- $p$  è compreso nel cerchio di raggio  $r$  con centro in  $q$ .
- il numero di punti interni al cerchio siffatto è superiore alla soglia prestabilita

Ci sarebbe anche la definizione di punto density-connected, ma io la semplifico al massimo dicendo: consideriamo 3 punti  $a, b, c$ . Diremo che  $c$  è density-connected a partire da  $a$  se tra loro due si trova  $b$  che di fatto è direttamente raggiungibile dai due. È un po' come la proprietà transitiva, ma solo la filosofia:  $a \rightarrow b, c \rightarrow b$  quindi di fatto  $a$  è collegato a  $c$  tramite  $b$ .

## 7.4 Algoritmi basati su griglia

Questo tipo di algoritmi secondo me è molto furbo, perchè si basano su un'idea geniale! O almeno, così mi è sembrato la prima volta che ne ho sentito parlare.

In sostanza si prende tutto il dataset e lo si rappresenta su un sistema di coordinate cartesiano (che può anche essere multidimensionale, non serve che sia per forza disegnabile, possiamo ragionarci su in astratto).

A questo punto, per ogni feature del dataset, (facciamo finta che ogni punto abbia 3 proprietà così ci viene un disegno 3D) andiamo a suddividere l'asse che la rappresenta in molti intervallini.

Da qui in poi il procedimento è quello di battaglia navale, ovvero si viene a creare una griglia multidimensionale nella quale i punti sono immersi. È quindi possibile fare un ragionamento molto immediato: tutti i punti nella stessa cella automaticamente li considero nello stesso cluster.

Andiamo a vedere un esponente di questa categoria

### 7.4.1 CLIQUE

CLIQUE è un algoritmo grid-based ma anche density-based. Quello che va a fare è sostanzialmente l'approccio generico esposto in precedenza. La particolarità è che una volta individuate le celle, su di esse viene applicato un approccio di clustering a densità. Così facendo godiamo delle proprietà di questi algoritmi senza sacrificare le performance.

Andiamo ad elencare pro e contro:

- Innanzitutto è buono che sia adatto per  $n$  dimensioni.
- Scala linearmente con le dimensioni del dataset, perchè anche se ho miliardi di elementi io sempre divido gli assi cartesiani allo stesso modo (oserei dire costante, o in altre parole con un criterio a priori)
- Non è schizzinoso sugli ordini da rispettare, lavora e basta.

Attenzione! L'accuratezza del clustering spesso non è elevata proprio a causa dell'estrema semplicità.

### 7.5 Algoritmi basati su Modello

Questo tipo di algoritmi utilizzano modelli spesso probabilistici per studiare la distribuzione dei dati nello spazio. Per esempio una cosa che si fa è quella di selezionare con probabilità equamente distribuita un primo sottoinsieme iniziale dei dati. Gli elementi di questo sottoinsieme vengono poi considerati i centri (ovvero i valori medi) di curve Gaussiane.

Quello che viene fatto sostanzialmente è calcolare quanto ogni Gaussiana influisce su un punto per poi poter effettuare gli spostamenti in cluster all'incirca come nel k-means.

### 7.6 Clustering di dati categorici

Per effettuare questo tipo di clustering non possiamo avvelerci di scorciatoie come la distanza. Quello che però possiamo fare è se vogliamo farlo rientrare dalla finestra chiamandolo similarità. Questo è quello che per esempio accade con l'algoritmo ROCK.

#### 7.6.1 Algoritmo ROCK

ROCK utilizza i link tra sottoinsiemi per capire quanto sono simili. Per tradurlo in un linguaggio comprensibile, dati due insiemi di partenza  $A, B$ , possiamo dire che più sono simili, maggiore è il numero degli elementi presenti nell'intersezione tra i due ( $A \cap B$ ). Si tratta quindi di un problema di ottimizzazione, ma questo comporta il dover fare molti confronti, per cui la sua complessità è elevata  $O(n^2 + nm_m m_a + n^2 \log n)$ . Inoltre, clusters tra loro simili sono collegati da un arco (non per forza 1 a 1) dal peso pari al numero di vicini. La foto in questo caso è risolutiva:

$$\begin{array}{c} \{1,2,3\}, \{1,2,4\}, \{1,2,5\}, \{1,3,4\}, \{1,3,5\} \\ \{1,4,5\}, \{2,3,4\}, \{2,3,5\}, \{2,4,5\}, \{3,4,5\} \\ \{1,2,3\} \xleftrightarrow{3} \{1,2,4\} \end{array}$$

## 7.7 Considerazioni finali sul Clustering

Questo pezzo non credo sarà di particolare rilevanza per l'esame, quindi scrivo giusto qualche pensiero sparso.

Innanzitutto ci sono molte soluzioni diverse, che a loro volta contengono parametri diversi. È quindi opportuno fare attenzione, perché i risultati finali vanno interpretati. Spesso quello che si fa è usare varie tecniche insieme e sperare che convergano ad un risultato simile.

Tornando per un attimo alla Bioinformatica, quello che facciamo è adattare un algoritmo di clustering a quello che ci serve a noi in un determinato momento. Dato che parliamo di geni, allora ci possiamo affidare a conoscenze pregresse per influenzare gli algoritmi a fare ciò che ci aspettiamo.

Esempio stupido: se so che un gene  $A$  e un gene  $B$  intervengono nello stesso processo, mi aspetto che vengano messi nello stesso cluster. Questa tecnica consiste nello sfruttare la "Gene Ontology"

### 7.7.1 Gene Ontology ed Implicazioni

La Gene Ontology è una sorta di database nel quale è possibile cercare i collegamenti logici tra geni. Non utilizziamo più criteri matematico/statistici bensì criteri sperimentali. È possibile dato un gene andare a vedere in quale processo biologico è coinvolto.

Di conseguenza abbiamo già tutto quello che ci serve, bisogna solo formalizzare. Quello che faremo è la seguente cosa: innanzitutto partiamo da un output di un qualunque algoritmo di clustering. In secondo luogo andiamo a valutare "quanto bene un gene sta nel suo cluster", attraverso la consultazione della GO (Gene Ontology). Avremo pertanto un indicatore affidabile sulla bontà del clustering.

Ma non ci fermiamo qui : il passo successivo è completare la valutazione attraverso la tecnica statistica del p-value.

Questo conto si fa in due step, ma l'idea di base è avere un numero che indichi una cosa del genere: "Se prendo a caso dei punti dal dataset, quanta probabilità c'è che siano concordi al mio output clusterizzato?". Ovviamente minore è questo valore, minore è la componente casuale e quindi statisticamente il nostro clustering è di qualità superiore. Il primo passo che si fa è calcolare la probabilità di avere  $x$  geni presi a caso su un campione di  $n$  appartenenti ad una specifica categoria della GO, dato che su tutto il genoma  $M$  solo  $N$  hanno quella stessa annotazione. Questo lavoro viene fatto con la distribuzione ipergeometrica.

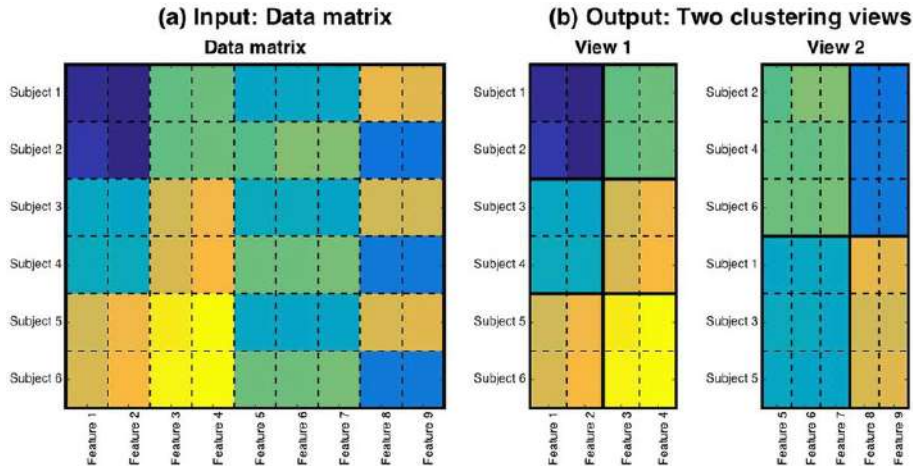
A questo punto possiamo a calcolare il nostro p-value:

To calculate a P-value, we calculate the probability of having *at least*  $x$  of  $n$  annotations:

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

Infine, si vanno a fare ipotesi specifiche caso per caso.

## 7.8 Co-clustering



Il Co-clustering è a quanto pare una tecnica inventata da alcuni professori universitari (tra cui Pensa e la Cordero) che serve per ottenere a partire da valori numerici delle informazioni più precise. Ora, come mai abbiamo bisogno di un clustering 2.0 ? Sostanzialmente perchè il clustering tradizionali ha delle mancanze:

- Non sappiamo il motivo per cui due geni A e B vengono ad esempio messi nello stesso cluster (parliamo di motivo biologico, non matematico)
- Magari nel mondo reale solo alcuni sottoinsiemi di cluster hanno senso (perchè magari solo in certe condizioni i geni A e B lavorano insieme,

e magari sono disattivati da un gene C che però risulta nel loro stesso cluster)

Quindi, l'idea è sostanzialmente quella di costruire una matrice dove le righe rappresentano i geni e le colonne le condizioni nelle quali sono attivi quei geni. A questo punto quello che si fa è andare sostanzialmente a costruire una sottomatrice più "sensata" dal punto di vista biologico, un po' come nella figura di inizio sezione.

Algoritmicamente quello che si va a fare consiste nelle seguenti operazioni:

1. Inizializzo la matrice di partenza e vado a calcolare i valori numerici
2. Mantenendo fisse le colonne, provo a trovare le righe che raggruppate mi danno un miglior risultato.
3. Mantenendo ora fisse le righe, trovo le colonne che raggruppate mi danno un miglior risultato.

Conviene sempre aiutarsi con l'immagine della sezione: in quella configurazione siamo partiti da una matrice dai colori molto sparsi a sottomatrici nelle quali c'è solo un colore. L'idea è che raggruppamenti siffatti rispecchino meglio nella realtà quelle che sono le relazioni tra i vari geni.

Di fatto, essendo il clustering una forma di apprendimento non supervisionato l'analista di turno ha pochi gradi di libertà (e.g. decidere se i cluster possono sovrapporsi, calcolare dei coefficienti come quelli di Goodman-Kruskal e di Cheng and Church, ...)

### 7.8.1 Conoscenze pregresse

Facciamo finta di sapere che i geni XYZ e ABC lavorano insieme in certe malattie e supponiamo che il mio algoritmo di clustering li separi in 2 gruppi diversi. La domanda è: come fare per forzare l'algoritmo a metterli insieme? Per non saper nè leggere nè scrivere a lezione mi è subito venuto in mente il problema del simplesso di Ricerca Operativa. Aggiungiamo vincoli all'algoritmo dimodochè faccia quello che ci aspettiamo. A questo punto il nostro problema si è trasformato in un problema di ottimizzazione.

In realtà non ci sono andato molto lontano, perchè quello che si fa è aggiungere delle condizioni, i **must-link** e i **cannot-link**, rispettivamente per i geni che vogliamo accoppiare e quelli che non vogliamo che vengano accoppiati. Formalmente indichiamo questi vincoli come  $C_=(i, j)$  e  $C_{\neq}(i, j)$ , dove  $i$  indica una riga e  $j$  una colonna.

Un'idea che sembra funzionare (Pensa, 2006) è quella di fare i raggruppamenti almeno visivamente consecutivi, nel senso che i vincoli sono maggiormente soddisfatti se un clustering è nella forma  $\{\{1, 2, 3\}, \{4, 5\}\}$  piuttosto che in quella  $\{\{1, 3\}, \{2, 4, 5\}\}$ .

**Nota** Capiamo che un clustering è buono perchè esiste una funzione obiettivo da ottimizzare, in questa sezione stiamo solo dicendo che mentre facciamo la nostra ottimizzazione, vogliamo introdurre anche dei vincoli da soddisfare (almeno la maggior parte possibile)

Il nuovo algoritmo è più o meno il seguente:

Partiamo sempre da una situazione in cui ho inizializzato la matrice e ho i miei insiemi di vincoli must-link per riga e colonna, rispettivamente  $M_r$  ed  $M_c$ . Le operazioni che faccio sono le seguenti:

**Ripeto fino alla convergenza**

- Calcola la funzione di score attuale
- Per ogni colonna  $j$ 
  - Se esiste un vincolo  $M \in M_c$  che dice che la colonna  $j$  sostanzialmente **gli appartiene** allora assegno **arbitrariamente** tutte le colonne specificate dal vincolo  $M$  **al cluster più vicino** dimodochè **nessun cannot-link sia violato**
  - Altrimenti assegno **solo  $j$**  al cluster più vicino sempre senza violare nessun cannot-link
- Faccio la stessa cosa per quanto riguarda le righe

Un esempio è quello delle slide che mi sembra autoesplicativo che metto nella pagina seguente.

## Example 1

	C1	C2	C3	C4	C5
G1	3	0	0	2	4
G2	1	4	5	1	2
G3	4	1	0	4	5
G4	2	0	1	3	4
G5	1	4	5	1	2
G6	1	4	6	0	0
G7	0	5	6	0	0

→

	C1	C4	C5	C2	C3
G1	3	2	4	0	0
G2	1	1	2	4	5
G3	4	4	5	1	0
G4	2	3	4	0	1
G5	1	1	2	4	5
G6	1	0	0	4	6
G7	0	0	0	5	6

- Optimization constraint only
- Best objective function value: 3.38

23

## Example 2

	C1	C4	C5	C2	C3
G1	3	2	4	0	0
G2	1	1	2	4	5
G3	4	4	5	1	0
G4	2	3	4	0	1
G5	1	1	2	4	5
G6	1	0	0	4	6
G7	0	0	0	5	6

→

	C1	C4	C5	C2	C3
G1	3	2	4	0	0
G3	4	4	5	1	0
G4	2	3	4	0	1
G2	1	1	2	4	5
G5	1	1	2	4	5
G6	1	0	0	4	6
G7	0	0	0	5	6

- Optimization constraint only +  $c_{\neq}(G1, G2)$
- Best objective function value: 3.38 → 3.99
- **G5 changes clusters too**

24



## 8 Motif Identification

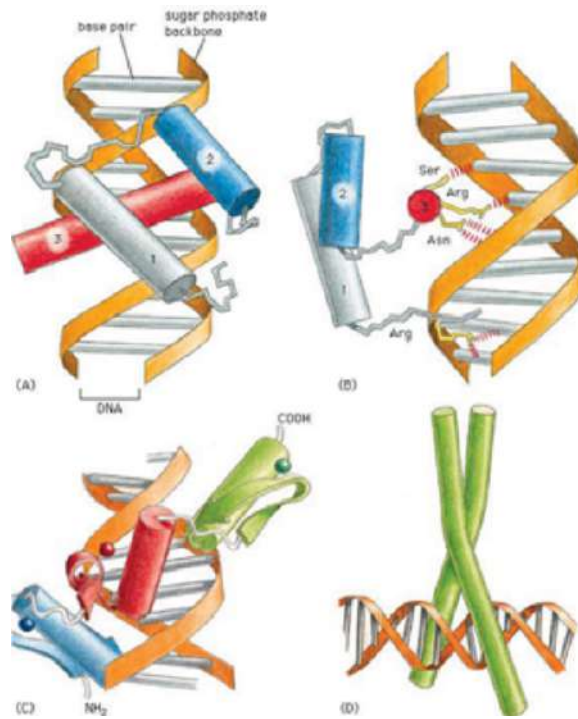
Riprendiamo ora la parte di teoria della Prof.ssa Cordero. In particolare andiamo ad analizzare il problema del Motif finding. Per fare un breve recap, possiamo dire che i motifs sono sostanzialmente delle porzioni particolari di DNA che regolano l'espressione di un determinato gene. Di fatto è come se i motifs fungano da entità modulatrici nella trascrizione degli RNA a partire dai geni.

Ora, i problemi sono molti, perchè spesso quello che si vuole fare è trovare nuovi motifs, il che ci porta a sbattere contro difficoltà oggettive, ad esempio:

- I motifs non sono sempre a distanza fissa dal gene che regolano.
- I motifs possono mutare nel tempo, per cui non c'è certezza assoluta che una determinata sequenza sia o meno un motif.
- Non sono legati a basi specifiche e hanno lunghezza variabile.

Se a questo aggiungiamo il discorso dello splicing ci stiamo davvero complicando la vita: ricordo che lo splicing è quell'attività che a partire da un gene arriva a costruire un RNA messaggero. In particolare lo splicing avviene in modo standard rimuovendo tutti gli introni di un gene, oppure in modo alternativo, che comprende combinazioni di istoni e introni (il che vuol dire che spesso 1 trascritto mappa n proteine diverse o simili, non sappiamo nemmeno questo).

A questo punto, noi vorremmo una metodologia che ci aiuti a trovare nuovi motifs senza brancolare nel buio in stile capopalestra Rudi di Bluvia nella regione di Hoenn. Di conseguenza quello che facciamo è la classica cosa che si fa quando non conosci una lingua e vuoi trovare informazioni utili: metti a confronto. Nel caso del DNA quello che si fa è confrontare DNA diversi per trovare le zone uguali: l'assunzione che facciamo è che quindi le zone uguali siano molto poco propense a mutare, e questo può indicare che effettivamente ci si trova davanti ad un motif. La verifica di questo di solito avviene a livello sperimentale: sappiamo che le proteine interagiscono col DNA, nel senso che proprio meccanicamente hanno una struttura 3d che "si attacca" proprio dove serve, un po' come se fossero ciechi che si orientano col tatto. Per cui almeno le proteine sanno dove sono i motifs, perchè quando viene fatta la trascrizione devono sapere quante proteine produrre e come. Un'idea per aiutarci nella comprensione è data dalle seguenti immagini:



Scer	TATCCATACTAATCTTATATGTTGT-GGAAAT-GTAAAGAGCCCATATCTTAGCCTAAAAAACC--TTCTCTTTGGAACTTTCAGTAATACG	TBP
Spar	TATCCATACTAGTCTTACITATCTGTTGT-GAGAGT-GTTGATAACCCAGIATCTTAACCCAGAAAGCC--TT-TCTATGAACTTGAACIG-TACG	
Smik	TACCGAIGCTAGTCTTACITATCTGTTAC-GGTAC-GGGAATGTGGTAATCCAGTCTCCAGATCAAAAAGGT--CTTCTATGAGCTTGTG-CTA-TATG	
Sbay	TAGTATTTCTGATCTTCTTATCTATATAGACAGATGCCAATAAACGTGCTACCTCGAACAAAAGAGGGGATTTTCTGTAGGCTTTCCCTATTGTG	
Scer	CTTAACCTGCTCATTCG-----TATAATTGAAGTATGGATTAGAAGCCGCGAGCGGCGACAGCCCTCGAGGAAGACTCTCTCCGTCGCTCTCTCTCT	GAL4
Spar	CTAAACCTGCTCATTCG-----AATAATTGAAGTATGGATCAGAAAGCCGCGAGCGGACGACAGCCCTCGAGGAATATTCCCTCCGTCGCTCTCTCTCT	
Smik	TTTAGCTGTTCAG-----AATAATTGAAGTATGGATGAGAAAGCCGCGAGCGGACGACAAATTCCTCCGTCGCTCTCTCTCTCTCTCTCTCTCT	
Sbay	TCTTATTGTCCATTACTTCGCAATGTTGAAATATGGATCAGAAAGCTGCGACGGATGACAGTACTCCGGAAGAACTGTCTCCGTCGGAAGTCTCTCT	
Scer	TCACCGG-TCCGCTTCGTAAGCCGAGATGTGCTCGCGCCGCACTGCTCCGAACAATAAGATTCTACAA-----TACTAGCTTTT--ATGGTTATGAA	GAL4
Spar	TCGTCGCGTTGTCTCCCTTAA-CATCGATGTACTCGCGCCGCGCTGCTCCGAACAATAAGCAATTCTACAGAAA-TACTGTTTTTTTATGTTATGAC	
Smik	ACGTTGG-TCCGCTCCCTGAA-CATAGGTACGCTCGCACCCGCTGCTCCGAACAATAAGCAATTCTACAGAAA-TACTGTTTTTTTATGTTATGAC	
Sbay	GTG-CGGATCACGTCCCTGAT-TACTGAAGCGTCTCGCGCCGCGCATACCCGGAACAATGCAATGCAAGAACAAA-TGCCGTGTAGT--GCAGTTATGGT	
Scer	GAGGA-AAAATTGGCAGTAA-----CCTGCCCCCAAACTT-CMAATTACGAATCMAATTAAACACATA-GGATGATAATGCGA-----TTAG--T	MIG1
Spar	AGGACAAAATAAGCAGCC-----ATGTCGCCCAATACCTTCAAACTATGATCAATGGCCAGCATA-TGGTATATGTCAG-----TTAG--G	
Smik	CAACGCAAAATAAGCAGTCC-----CCCGCCCCACATACCTTCAAACTGATCGTAAACTGGCTAGCATA-GAATTTGGTAGCAA-AATAATTAG--G	
Sbay	GAACGTGAATGACATTCCTGCCCT-CCCCAATATCTTGTTCGCTGTACAGCACACTGGATAGAACAATGATGGGTGCGGCTCAAGCTACTCTG	
Scer	TTTTAGCTTATTCTGGGCTTAATTAATCAGCGAAGCG--ATGATTTTT-GATCTATTAACAGATATATAATGGAAAAGCTGCATAACCAC-----TT	MIG1
Spar	TTTTTT--TCTATTCTGAGCAATTCAATCCGCAAAAATAATGGTTTTT--GGTCTATTAGCAACATATAAATGCAAAAAGTTCATAGCCAC-----TT	
Smik	TCTCA--CTTCTCTGTGTAATTCATCCAGCAATG--ATGGTTTA--GGACTATTAGCAACATATAAATGCAAAAAGTTCATAGCCAC-----TT	
Sbay	TTTTCCGTTTACTTCTGTAGGGCTCAT--GCAGAAAATAATGGTTTCTGTCTCTTGCACAACTATAATATGAAAAGTAAAGTGCCTCAATTGTGA	
Scer	TAACATACTCTCAACATTTTCAGT--TTGTATTACTTTCTATTCAAAAT-----GTCAATAAAGTATACACA-AAAAATTGTTAATATAC	TBP
Spar	TAAATAC-ATTTGCTCCCTCCAGATT--TTTAATTTCTGTTTGTTTTATTT--GTCAATGCAAAATTAACA-ACAAAGTATGTTAATATAC	
Smik	TCATTC--ATTCGAACCTTTGAGACTAATTAATTTAGTACTAGTTTCTTTGGAGTTATAGAAAATACAAAA-AAAAATAGTCAGTATCT	
Sbay	TAGTTTTTCTTATTCGTGTGTACTTCTTAGATTGTATTTCGGTTTACTTTGTCTCAATTATCAAAACATCAATACAGATATCT	
	ATGACTA	GAL1



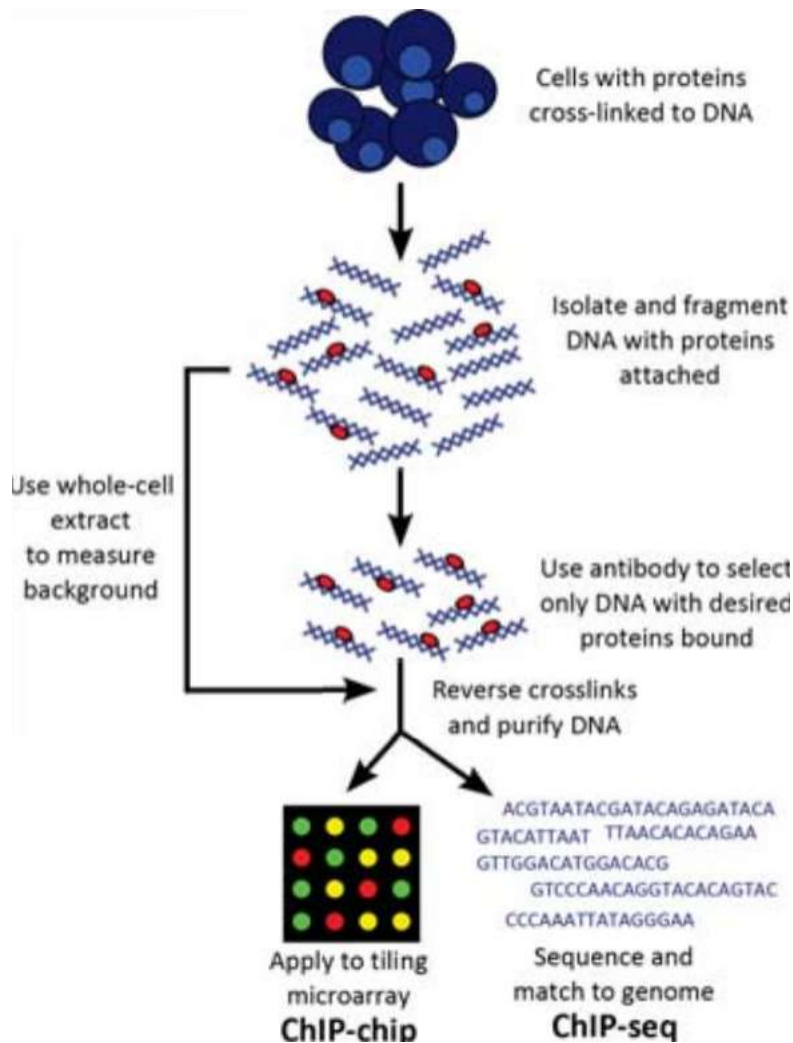
Transcription factor binding

Conservation island

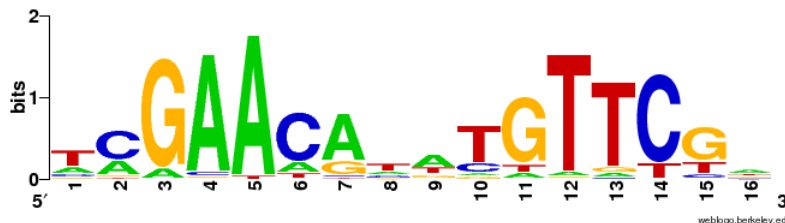
Ora, sapendo tutto questo possiamo utilizzare una tecnica che personalmente reputo molto interessante per l'identificazione dei motifs: abbiamo detto che le proteine sostanzialmente sanno quali sono questi motifs (perchè li “sentono al tatto”). Ebbene, allora facciamo degli esperimenti dove lasciamo che queste proteine si leghino ai motifs e poi noi cerchiamo di rimuovere il blocco che si crea in questo modo (proteina + motif) in un qualche modo separandolo dall'intero DNA.

Nello specifico andiamo ad inventarci un anticorpo che si attacca alla proteina e la fa precipitare con attaccato il motif. Solo in seguito si rimuove l'anticorpo dalla proteina lasciando quindi il nostro tanto voluto motif.

Un cartoon che spiega questo procedimento in modo più schematico è il seguente:



Ci manca però un ultimo pezzo: può capitare che le proteine si leghino in modi e luoghi diversi, per cui l'esperimento che abbiamo spiegato prima deve essere ripetuto molte volte, e poi si vanno a confrontare tutte le sequenze lette. Per avere poi una rappresentazione schematica probabilistica si utilizza la tecnica del Motif Logo che abbiamo già visto con Botta ma che ripropongo in un'immagine:



Così facendo andiamo a fare una selezione probabilistica di quello che può essere il nostro motif definitivo considerando le frequenze dei singoli nucleotidi nelle varie sequenze lette.

### 8.1 Analisi statistica delle sequenze

Una volta che abbiamo stabilito il nostro piano d'azione ed ottenuto i primi risultati sperimentali è arrivato il momento della parte di data mining. E qui ci si può sbizzarrire, nel senso che possiamo adottare approcci diversi, per esempio:

- Tipo di apprendimento (supervised o unsupervised)
- Tipo di pattern (deterministico, rigido, flessibile...)
- Misure di rilevanza statistica (p-value)
- Conoscenze a priori

Comunque, la prima cosa da fare è identificare quelli che si chiamano i TFBS (Transcription Factor Binding Site). Questi fattori hanno le stesse difficoltà descritte per i motifs, perchè sono corti, possono variare senza cambiare significativamente la loro funzione biologica, si ripetono spesso... Di conseguenza abbiamo il problema di considerare i falsi positivi ed i falsi negativi, il che di certo non ci semplifica la vita. Per affrontare al meglio questa parte, rimando al problema del Gold Bug che abbiamo visto con Botta. In quell'occasione noi conoscevamo il linguaggio originale del messaggio cifrato (Inglese) per cui potevamo sfruttare bene la crittoanalisi statistica. Col DNA non possiamo fare questo discorso, perchè abbiamo solo l'alfabeto di 4 lettere... Come se non bastasse, solo una piccolissima parte del genoma codifica i motif, per cui dobbiamo per forza lavorare su dati grossi in termini di spazio occupato. Discutiamo adesso delle varie forme di pattern matching:

1. Pattern deterministici (e.g. cerco direttamente TATA) : è chiaro che non sono molto utili in ambito biologico

2. Pattern rigidi: usano le lettere dell'alfabeto IUPAC per mappare i singoli nucleotidi. Le lettere usate sono  $\{A, C, G, T, U, M, R, W, S, Y, K, V, H, D, B, X, N\}$ , dove le lettere che non appartengono alle 4 di base del DNA indicano range probabilistici di aver letto una di quelle 4. Per fare un esempio, la R indica che in quella posizione posso leggere una A oppure una G. Insieme, avremo che  $R \rightarrow \{A, G\}$ . Questo è molto utile perchè così siamo in grado di rappresentare una cosa grafica come un Motif Logo con una stringa di testo. L'unico contro è che le sostituzioni di lettere sono sempre 1:1, per cui la lunghezza del pattern è fissa.
3. Pattern flessibili: le sostituzioni possono essere fatte con multiple lettere. Un esempio è F-x(5)-G-x(2,4)-G-\*H nel quale x(5) indica che ci sono 5 X, x(2,4) indica che ci può essere un numero variabile di X che va da 2 a 4 e \* indica che in quel punto non ci ho capito niente di quello che c'è.

Per riuscire a districarsi da tutta questa complessità utilizziamo le matrici di profilo per arrivare ad una sorta di stringa consensus (che in ambito accademico è l'unica cosa che conta quando laboratori diversi si parlano). La matrice di profilo è una matrice siffatta:

<b>A</b>	<i>0.26</i>	<i>0.22</i>	<i>0.00</i>	<i>1.00</i>	<i>0.11</i>
<b>C</b>	<i>0.17</i>	<i>0.18</i>	<i>0.59</i>	<i>0.00</i>	<i>0.35</i>
<b>G</b>	<i>0.09</i>	<i>0.15</i>	<i>0.00</i>	<i>0.00</i>	<i>0.00</i>
<b>T</b>	<i>0.48</i>	<i>0.45</i>	<i>0.41</i>	<i>0.00</i>	<i>0.54</i>

Da notare che la somma dei valori di una colonna deve sempre dare 1. Ultimamente, a causa di diversi lavori scientifici, non si usa la probabilità stretta ma si discriminano le varie stringhe attraverso il concetto di contenuto informativo IC, definito dalla seguente formula:

$$IC_i = 2 \cdot \sum_{B=A}^T f_{b,i} \log_2(f_{b,i}) \quad (7)$$

Dove  $f_{b,i}$  è la frequenza della base  $b$  in posizione  $i$ .

## 8.2 Incertezza media

In questa sezione andiamo a fare un piccolo approfondimento probabilistico per aiutare la comprensione. Parliamo di incertezza come l'inverso della probabilità, ergo se un evento ha probabilità  $p$  la sua incertezza sarà data da  $\frac{1}{p}$ . È ovvio quindi notare che all'aumentare della mia probabilità si riduca l'incertezza. Per quanto riguarda l'incertezza media essa è modellata dalla seguente formula:

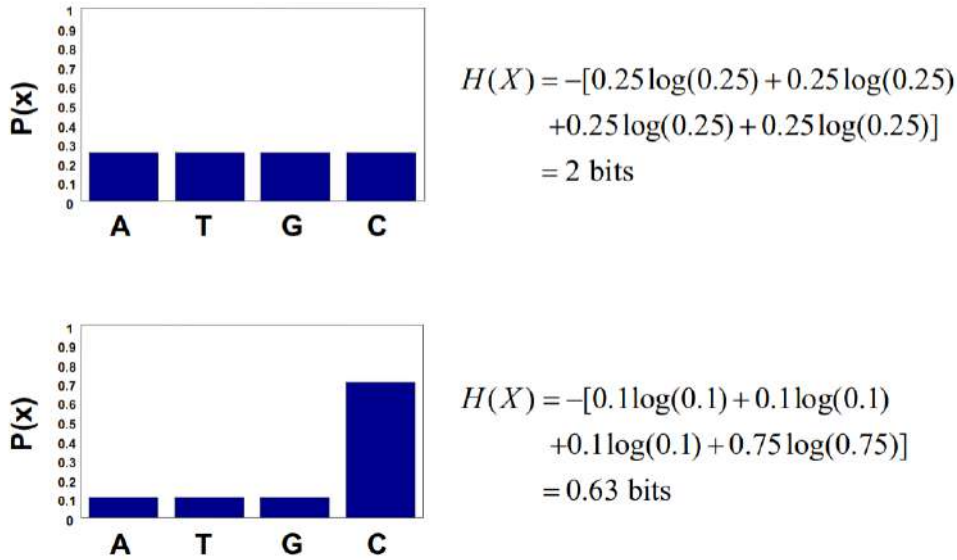
$$P(A)Uncertainty(A) + P(B)Uncertainty(B) = -p_1 \log_2(p_1) - p_2 \log_2(p_2) \quad (8)$$

Che generalizzando diventa:

$$-\sum p_i \log_2(p_i) = Entropia \quad (9)$$

L'entropia è maggiore quando le probabilità sono eque, per esempio nel lancio di una moneta non truccata (ho sempre probabilità  $\frac{1}{2}$ , per cui ho la massima incertezza).

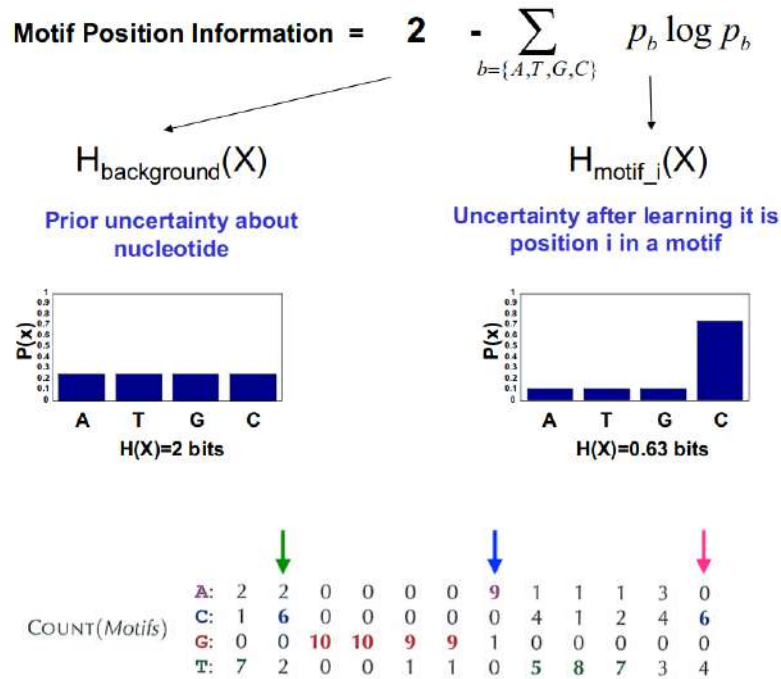
Un esempio tratto dalle slide è il seguente:



Possiamo dire che l'informazione sia un decremento di entropia, ovvero una differenza di entropia in due momenti differenti. Per citare di nuovo le slide faccio un esempio legato al nostro caso di studio:

## 8.3 Motif Detection Algorithm

Allora, innanzitutto andiamo a costruire la matrice delle frequenze:



→  $0.2 \log_2 0.2 + 0.6 \log_2 0.6 + 0 \log_2 0 + 0.2 \log_2 0.2 = 1.371$

→  $0 \log_2 0 + 0.6 \log_2 0.6 + 0 \log_2 0 + 0.4 \log_2 0.4 = 0.971$

→  $0 \log_2 0 + 0 \log_2 0 + 0.9 \log_2 0.9 + 0.1 \log_2 0.1 = 0.467$

A questo punto andiamo a calcolare lo score, ovvero più mismatch hai più lo score è alto:

	T	C	G	G	G	G	g	T	T	T	t	t
	c	C	G	G	t	G	A	c	T	T	a	C
	a	C	G	G	G	G	A	T	T	T	t	C
	T	t	G	G	G	G	A	c	T	T	t	t
Motifs	a	a	G	G	G	G	A	c	T	T	C	C
	T	t	G	G	G	G	A	c	T	T	C	C
	T	C	G	G	G	G	A	T	T	c	a	t
	T	C	G	G	G	G	A	T	T	c	C	t
	T	a	G	G	G	G	A	a	c	T	a	C
	T	C	G	G	G	t	A	T	a	a	C	C
SCORE(Motifs)	3 + 4 + 0 + 0 + 1 + 1 + 1 + 5 + 2 + 3 + 6 + 4 = 30											

A partire dalla matrice delle frequenze è un attimo calcolare la matrice delle probabilità dato che basta dividere ogni numero di una colonna per la somma dei valori totali.

	A:	.2	.2	.0	.0	.0	.0	.9	.1	.1	.1	.3	.0
Profile	C:	.1	.6	.0	.0	.0	.0	.0	.4	.1	.2	.4	.6
	G:	.0	.0	1	1	.9	.9	.1	.0	.0	.0	.0	.0
	T:	.7	.2	.0	.0	.1	.1	.0	.5	.8	.7	.3	.4

$$\Pr(\text{ACGGGGATTACC} | \text{Profile}) = .2 \cdot .6 \cdot 1 \cdot 1 \cdot .9 \cdot .9 \cdot .9 \cdot .5 \cdot .8 \cdot .1 \cdot .4 \cdot .6 = 0.000839808$$

In questa immagine è riportata anche la probabilità di trovare la stringa ACGGG-GATTACC data la matrice di profilo.

L'ultimo step sta nel determinare quanto è distante la sequenza trovata rispetto all'ipotesi nulla. In altre parole quello che vogliamo fare è andare a vedere se prendiamo una stringa totalmente randomica, quanto è probabile che sia quella di test. Questo è molto utile, perchè è come andare a calcolare un p-value, ovvero l'idea è quella di verificare che le nostre stringhe di test non siano comparabili ad altre completamente random. Questa è una misura indiretta di bontà, nel senso che se io non riesco ad ottenere facilmente la mia stringa di test in modo random, allora quella stringa avrà implicitamente un contenuto informativo statisticamente significativo. In formule:

$$\text{Score} = \log \frac{P(S | PFM)}{P(S | B)} = \log \prod_{i=1}^N \frac{P_i(S_i | PFM)}{P(S_i | B)} = \sum_{i=1}^N \underbrace{\log \frac{P_i(S_i | PFM)}{P(S_i | B)}}_{\text{score}_i}$$

Se i motif che cerchiamo non sono deterministici, allora si dice che avremo a che fare con i cosiddetti (k,d)-motifs. Un algoritmo bruteforce per la visita e la valutazione dei motifs non è certamente una soluzione accettabile quando lavoriamo con dati molto grandi.



Comunque, questa parte è la ripetizione degli algoritmi di Motif finding che abbiamo fatto con Bottà, per cui dovremmo essere a posto già così.

## 8.4 Greedy Motif Search

Il nuovo problema che abbiamo riformulato a partire da quello della stringa mediana visto con Bottà è il seguente : abbiamo in input una collezione di stringhe e quello che vogliamo trovare in output è un pattern che minimizza la distanza tra tutti i possibili pattern e tutti i possibili k-mers.

A prima vista sembra che abbiamo solo aumentato la difficoltà rispetto al problema della stringa mediana, perchè ora vogliamo il minimo globale. Ed in effetti è proprio così... Fortunatamente in informatica spesso è necessario fare un passo controintuitivo per vedere bene cosa si cela dietro ad un muro. Nel nostro caso l'idea è che troveremo un modo per evitare di esplorare ogni singolo k-mer per minimizzare la distanza  $d(\text{Pattern}, \text{Motifs})$ .

Il trucco in questo caso sta nell'utilizzare la matrice di profilo per generare probabilisticamente il k-mer più probabile. Per cui, facciamo una cosa del genere:

A	1/2	7/8	3/8	0	1/8	0
C	1/8	0	1/2	5/8	3/8	0
T	1/8	1/8	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	1/8

**Profile- most probable k-mer in a sequence: the k-mer with the highest**

$Pr(k\text{-mer} | \text{Profile})$

among all k-mers in this sequence.

6-mer	Prob(6-mer Profile)	
CTATAAACCTTACAT	$1/8 \times 1/8 \times 3/8 \times 0 \times 1/8 \times 0$	0
CTATAAACCTTACAT	$1/2 \times 7/8 \times 0 \times 0 \times 1/8 \times 0$	0
CTATAAACCTTACAT	$1/2 \times 1/8 \times 3/8 \times 0 \times 1/8 \times 0$	0
CTATAAACCTTACAT	$1/8 \times 7/8 \times 3/8 \times 0 \times 3/8 \times 0$	0
CTATAAACCTTACAT	$1/2 \times 7/8 \times 3/8 \times 5/8 \times 3/8 \times 7/8$	<b>.0336</b>
CTATAAACCTTACAT	$1/2 \times 7/8 \times 1/2 \times 5/8 \times 1/4 \times 7/8$	.0299
CTATAAACCTTACAT	$1/2 \times 0 \times 1/2 \times 0 \times 1/4 \times 0$	0
CTATAAACCTTACAT	$1/8 \times 0 \times 0 \times 0 \times 0 \times 1/8 \times 0$	0
CTATAAACCTTACAT	$1/8 \times 1/8 \times 0 \times 0 \times 3/8 \times 0$	0
CTATAAACCTTACAT	$1/8 \times 1/8 \times 3/8 \times 5/8 \times 1/8 \times 7/8$	.0004

Possiamo utilizzare quindi questo approccio attraverso l'implementazione del seguente algoritmo Greedy:

```

GREEDYMOTIFSEARCH(Dna, k, t)
  BestMotifs  $\leftarrow$  motif matrix formed by first k-mers in each string from Dna
  for each k-mer Motif in the first string from Dna
    Motif1  $\leftarrow$  Motif
    for i = 2 to t
      form Profile from motifs Motif1, ..., Motifi-1
      Motifi  $\leftarrow$  Profile-most probable k-mer in the i-th string in Dna
    Motifs  $\leftarrow$  (Motif1, ..., Motift)
    if SCORE(Motifs) < SCORE(BestMotifs)
      BestMotifs  $\leftarrow$  Motifs
  return BestMotifs

```

È chiaro che è computazionalmente esoso fare la suddivisione di un intero genoma in *k*-mers, per cui quello che si vuole fare è evitare di dover necessariamente esplorare tutti i possibili casi.

Per spiegare bene cosa fa questo algoritmo partiamo dalla seguente stringa di *Dna*:

```

GGCGTTCAGGCA
AAGAATCAGTCA
CAAGGAGTTCGC
CACGTCAATCAC
CAATAATATTTCG

```

Ora, inizialmente costruiamo un insieme di *k*-mers iniziali formato dai primi *k*-mers per ogni stringa. Con *k*=3 avremo

```

GGCGTTCAGGCA
AAGAATCAGTCA
CAAGGAGTTCGC
CACGTCAATCAC
CAATAATATTTCG

```

A questo punto avremo un insieme di *k*-mers iniziali sui quali costruiamo la matrice di profilo. Lo step successivo è utilizzare questa matrice anziché una globale per trovare *k*-mers più probabili nelle altre stringhe di *dna*. Adesso entriamo nel primo loop. Quello che facciamo è semplicemente scorrere la prima stringa di *Dna* per trovare tutti i *k*-mers.

```

GGCGTTCAGGCA GGCGTTCAGGCA GGCGTTCAGGCA

```

E così via.

Per ognuno di essi entriamo nel secondo loop e lo confrontiamo con tutte le altre stringhe alla ricerca di un miglior candidato. Quindi per ogni motif della prima stringa otterremo *t* nuovi motifs dove *t* è il numero di righe di *DNA* originali.

A questo punto si calcola lo score di questo nuovo insieme, e se è minore di quello iniziale (BestMotifs) allora BestMotifs cambia e verrà utilizzato nell'iterazione successiva. Graficamente:

GGCGTTCAGGCA AAGAATCAGTCA CAAGGAGTTCGC CACGTCAATCAC CAATAATATTCG Score = 7	GGCGTTCAGGCA AAGAATCAGTCA CAAGGAGTTCGC CACGTCAATCAC CAATAATATTCG Score = 5	GGCGTTCAGGCA AAGAATCAGTCA CAAGGAGTTCGC CACGTCAATCAC CAATAATATTCG Score = 4
---	---	---

Così facendo non dobbiamo utilizzare una matrice di Profilo globale ottenendo un tempo di running quantomeno accettabile.

#### 8.4.1 Laplace's rule of succession

Per migliorare la qualità in termini di score dei k-mers trovati si è visto che è utile applicare la regola di successione di Laplace (Laplace's rule of succession). Il problema è che sostanzialmente i calcoli probabilistici si fanno tramite produttorie, per cui può comunque capitare un caso del genere:

$$\begin{array}{l}
 \text{Profile} \\
 \begin{array}{l}
 \text{A: } .2 \ .2 \ .0 \ .0 \ .0 \ .0 \ .9 \ .1 \ .1 \ .1 \ .3 \ .0 \\
 \text{C: } .1 \ .6 \ .0 \ .0 \ .0 \ .0 \ .0 \ .4 \ .1 \ .2 \ .4 \ .6 \\
 \text{G: } .0 \ .0 \ .1 \ .1 \ .9 \ .9 \ .1 \ .0 \ .0 \ .0 \ .0 \ .0 \\
 \text{T: } .7 \ .2 \ .0 \ .0 \ .1 \ .1 \ .0 \ .5 \ .8 \ .7 \ .3 \ .4
 \end{array}
 \end{array}$$

$$\Pr(\text{TCGTGGATTTC} | \text{Profile}) = .7 \cdot .6 \cdot .1 \cdot .0 \cdot .9 \cdot .9 \cdot .9 \cdot .5 \cdot .8 \cdot .7 \cdot .4 \cdot .6 = 0$$

In questo caso abbiamo la stringa TCGTGGATTTC e vogliamo vedere quanto è probabile ottenerla data la matrice di profilo. Per cui quello che facciamo è moltiplicare le varie probabilità delle singole lettere per ogni posizione (es. per la prima T vado a vedere nella prima colonna della matrice quanto è probabile avere T e uso quel dato, e così via). Ora, di solito noi prendiamo la stringa con lo score più basso, ma in questo caso c'è un grosso problema: la quarta colonna ha un elevato numero di zeri, per cui è appena capitato che la probabilità di avere la T come quarta lettera è 0. Ed ecco che abbiamo appena falsato lo score di tutta la stringa, perchè l'algoritmo crede di aver trovato un ottimo quando invece è un k-mer qualitativamente molto basso!

Per fortuna la regola di Laplace è banale da applicare e risolve il problema elegantemente. Dal punto di vista grafico andiamo a modificare la matrice delle frequenze in questo modo:

	A: 2+1 1+1 1+1 1+1		3/8 2/8 2/8 2/8
COUNT(Motifs)	C: 0+1 1+1 1+1 1+1	PROFILE(Motifs)	1/8 2/8 2/8 2/8
	G: 1+1 1+1 1+1 0+1		2/8 2/8 2/8 1/8
	T: 1+1 1+1 1+1 2+1		2/8 2/8 2/8 3/8

A questo punto l'algoritmo Greedy può essere riscritto imponendo di correggere le matrici di profilo nel ciclo interno utilizzando la regola di Laplace.

#### 8.4.2 Rolling dice to find motifs

Un'altra tattica che possiamo usare per cercare i motifs all'interno del nostro Dna è quella di utilizzare algoritmi randomici. A primo impatto potrebbe sembrare disastroso (es. Gioco a scacchi muovendomi a caso), tuttavia spesso le performance sono di notevole livello, come nel caso del problema dell'Ago di Buffon<sup>9</sup>. Detto questo, questo tipo di algoritmi vengono generalmente divisi in algoritmi di Las Vegas (che trovano la soluzione ottima) ed algoritmi Monte Carlo (che invece si accontentano di una buona soluzione). Nell'ambito del motif finding vengono utilizzati gli algoritmi del secondo tipo. Questo perchè essendo molto veloci possono essere lanciati parecchie volte (anche in parallelo), permettendo di fare confronti tra gli insiemi di soluzioni.

Tutto questo discorso ci porta alla formalizzazione di questo algoritmo stile Monte Carlo:

```

RANDOMIZEDMOTIFSEARCH(Dna, k, t)
  randomly select k-mers Motifs = (Motif1, ..., Motift) in each string from Dna
  BestMotifs ← Motifs
  while forever
    Profile ← PROFILE(Motifs)
    Motifs ← MOTIFS(Profile, Dna)
    if SCORE(Motifs) < SCORE(BestMotifs)
      BestMotifs ← Motifs
  else
    return BestMotifs

```

Ora, si dimostra che la chiave di tutto diventa la scelta del set di motifs iniziale. Questo però non ci spaventa più di tanto, perchè l'algoritmo di per sè è veloce, per cui non mi costa nulla provare in parallelo per 100k o anche 1MLN di volte. Prima o poi saremo fortunati quel tanto che basta per andare nella direzione giusta.

#### 8.4.3 Gibbs Sampling

Dobbiamo necessariamente fare un'osservazione: L'approccio puramente randomico ha un senso solo se le stringhe del Dna fossero tutte equiprobabili (ogni base ha sempre probabilità 0.25). Nella realtà le cose non stanno così, proprio

<sup>9</sup>[https://it.wikipedia.org/wiki/Ago\\_di\\_Buffon](https://it.wikipedia.org/wiki/Ago_di_Buffon)

perchè il Dna contiene dei motifs ! Per questo motivo è utile utilizzare un approccio più conservativo che prende il nome di Gibbs Sampling. Dal punto di vista del sentimento che c'è dietro posso dire che sostanzialmente l'unica differenza sta nel ciclo interno : al posto di cambiare il set di k-mers in modo deterministico l'idea è di cambiarne solo uno scegliendo il rimpiazzo a random dall'insieme di k-mers appena costruito. Questo perchè magari il problema della qualità di quell'insieme risiede in un numero molto limitato di elementi e non in tutti ! Per esempio:

ttacctt <b>aac</b>	ttaccttaac	ttacctt <b>aac</b>	ttacctt <b>aac</b>
g <b>ata</b> tctgtc	gata <b>at</b> ctgtc	g <b>ata</b> tctgtc	gata <b>at</b> ctgtc
<b>acg</b> gcgttcg	acggcg <b>tt</b> c	<b>acg</b> gcgttcg	<b>acg</b> gcgttcg
ccct <b>aaa</b> gag	ccctaa <b>ag</b> ag	ccct <b>aaa</b> gag	ccct <b>aaa</b> gag
cg <b>ta</b> ga <b>g</b> gt	cg <b>t</b> cagaggt	cg <b>ta</b> ga <b>g</b> gt	cg <b>ta</b> ga <b>g</b> gt
<b>RANDOMIZEDMOTIFSEARCH</b>		<b>GIBBSAMPLER</b>	
(may change all k-mers in one step)		(changes one k-mer in one step)	

**Attenzione!** È possibile che l'algoritmo si blocchi in un ottimo locale, che alla fine è il solito problema degli algoritmi che approssimano una qualche funzione generica.

Ed eccoci arrivati a descrivere l'implementazione:

```

GIBBSAMPLER(Dna, k, t, N)
  randomly select k-mers Motifs = {Motif1, ..., Motift} in each string from Dna
  BestMotifs ← Motifs
  for j ← 1 to N
    i ← RANDOM(t)
    Profile ← profile matrix formed from all strings in Motifs except for Motifi
    Motifi ← Profile-randomly generated k-mer in the i-th sequence
    if SCORE(Motifs) < SCORE(BestMotifs)
      BestMotifs ← Motifs
  return BestMotifs

```

Per ulteriori informazioni rimando a consultare le slide della Cordero, un po' perchè sono fatte bene e un po' perchè non credo sia necessario raggiungere un livello di dettaglio alto al fine di passare l'esame.

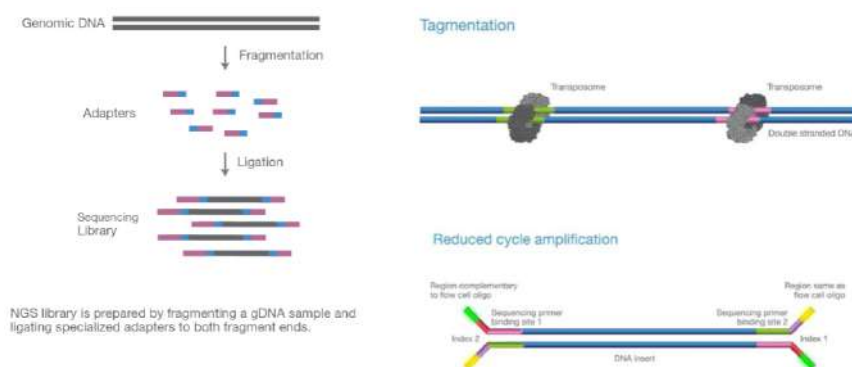
## 9 Introduzione al Deep Sequencing

In questa sezione andremo a fare una panoramica teorica su quello che è il mondo del sequenziamento con le moderne tecniche Deep Sequencing. Cercherò di fare questa parte in modo breve e sintetico perchè all'esame non ci viene chiesta la parte di Biologia spicciola.

Dunque, partiamo dicendo che nel tempo il costo di sequenziamento del singolo genoma è sceso molto al di sotto della legge di Moore, per cui oggi paghiamo circa 1000 \$ per sequenziamento. Anche dal punto di vista ingegneristico si sono fatti molti progressi, sono usciti macchinari che riescono a tirare fuori milioni di stringhe in 48 ore.

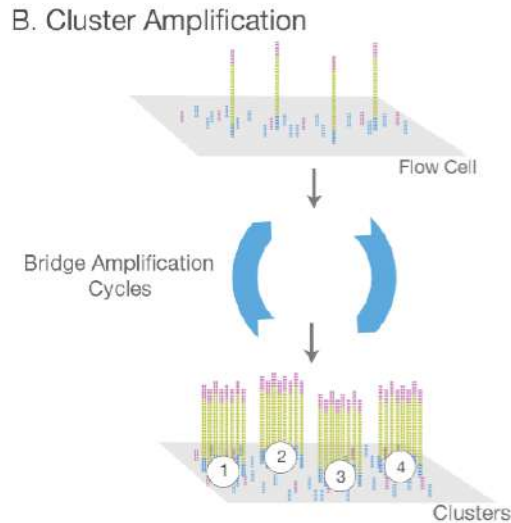
Un esempio è la macchina della serie Illumina, ma ai nostri fini ci manca ancora un pezzo di spiegazione, ovvero come viene preparato l'esperimento. Parliamo quindi di come costruire gli input e come la macchina riesce a fornirci in output le sequenze di DNA.

### 9.1 Library Preparation



Prima di poter dare in pasto il DNA alla nostra macchina di Deep Sequencing dobbiamo necessariamente fare dei passi intermedi. Il primo tra questo è la Library Preparation. In sostanza a partire da un set di cellule si estrae il DNA, successivamente avviene una frammentazione in pezzi più piccoli e, per ogni pezzo, andiamo ad aggiungere alle sue estremità quelli che vengono chiamati adattatori. Gli adattatori sono componenti utili per poter svolgere il secondo step.

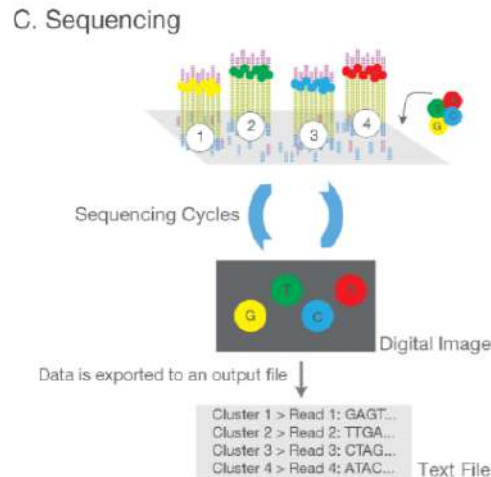
## 9.2 Cluster Amplification



A questo punto si attua la Cluster Amplification, ovvero si inseriscono i vari pezzi di DNA dentro un dispositivo noto come Flow Cell. Gli adattatori della fase di Preparazione si mettono proprio per far sì che i pezzi di DNA si possano inserire in questo dispositivo. È voluto che nella Flow Cell ci siano più spazi di quello che mi serve. Questo perchè alla fine del processo, i vari pezzi di DNA si moltiplicano lungo un pezzo di Flow Cell grazie ad enzimi (DNA polimerasi), proprio come nel disegno qui sopra.

Ora, ci sono molte tecnologie inventate per fare lettura di DNA, un paio le abbiamo viste a lezione e come al solito sono tecniche inventate per altri scopi che sono state adattate alla Biologia. Andiamo ad approfondire un attimo la situazione.

### 9.3 Sequencing



Le macchine che fanno sequenziamento cercano sostanzialmente di far legare il nostro DNA ad un'altra sequenzina detta Primer. In base a quale nucleotide si lega il Primer si ha la produzione di un particolare fluoruro che alla macchina appare come luce di un certo colore (nella figura si può notare lo schema dei colori). È necessario fare questa operazione di generazione di ridondanza sostanzialmente perché così posso fare clustering e quando leggo sono più sicuro di quello che leggo (forse perché viene prodotta luce più intensa e grande in area).

Una macchina NGS produce in output un file con estensione FASTQ. Per ogni read, viene riportata la seguente struttura:

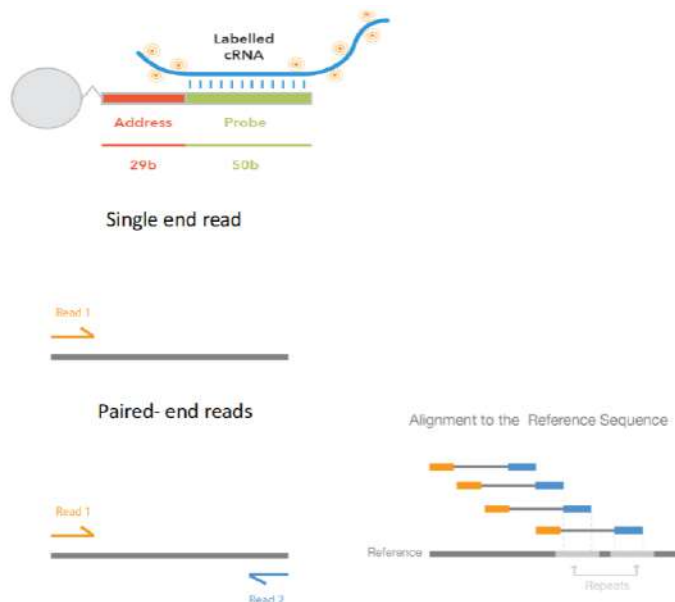
```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTGTGCAACTCACAGTTT
+
!''*((( (**+))%%%++) (%%%) .1***-+*'') **55CCF>>>>>CCCCCCC65
```

Il carattere + funge da separatore. Le lettere A,T,C,G sono effettivamente le lettere che compongono la lunga stringa della read, @SEQ\_ID è un id univoco mentre l'ultima riga rappresenta il livello di qualità. In pratica ad ogni lettera letta della stringa corrisponde un carattere ASCII nella riga di sotto che indica quanto siamo sicuri che si sia letta proprio quella lettera (perché la chimica non sempre è precisa). Questi ultimi “caratteri di qualità” sono i seguenti (in ordine crescente).

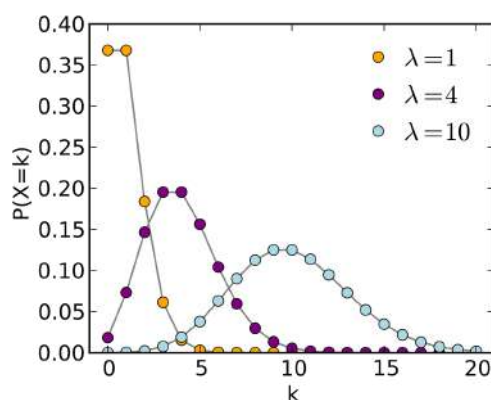
```
! "# $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```



Ora, introduciamo il concetto di **coverage**. Si parla di coverage (ad esempio a 100X), quando in media ogni read viene letta almeno 100 volte. Questo consente di avere più read per poter successivamente allineare il tutto e ricostruire il genoma. Le reads possono essere single end o paired end, proprio come in figura:



Senza entrare troppo nello specifico (perchè non credo sia necessario) si dimostra che il numero di read che viene letto con un coverage ad esempio a 100X segue una distribuzione di poisson, dal seguente grafico:

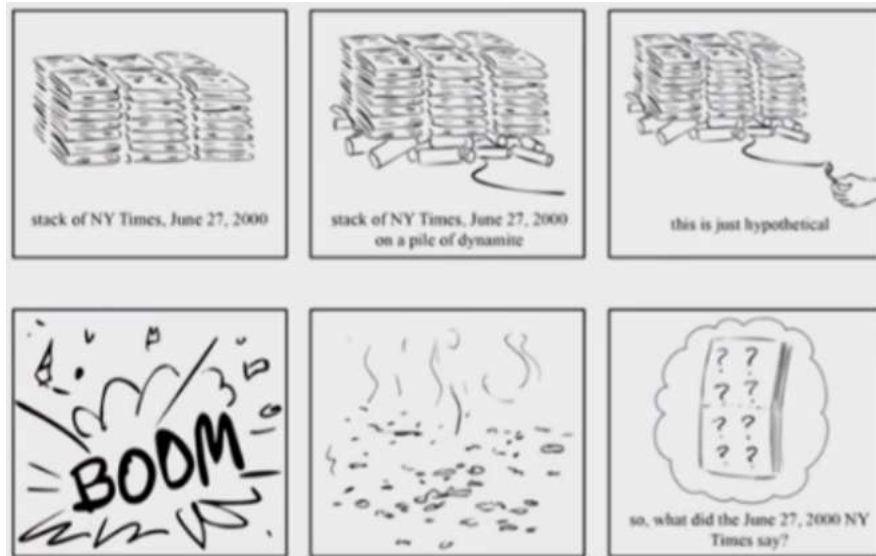


In breve, la maggior parte delle read viene letta effettivamente  $n$  volte (quanto previsto dal coverage) mentre allontanandoci dalla centratura della distribuzio-

ne risulterà che alcune (anche se relativamente poche) saranno lette di meno (sempre per colpa della chimica che fa un po' quel che vuole).

## 10 Genome Assembly

### 10.1 The Newspaper problem



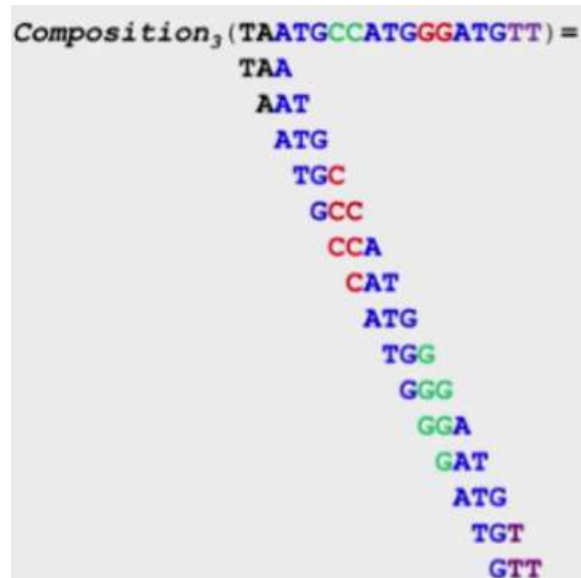
Il problema si configura nel seguente modo: abbiamo una pila di copie dello stesso giornale (e.g. New York Times del 27 Giugno 2000), la si cosparge di dinamite e poi facciamo saltare tutto in aria. I pezzetti che rimangono verranno usati per la ricostruzione di una sola copia.

Anche se potrebbe sembrare un problema stupido, è una metafora molto azzeccata del problema della ricostruzione del DNA. Infatti ad oggi non siamo in grado di leggere tutto un genoma dall'inizio alla fine, per colpa della chimica. Inoltre, c'è bisogno di una pila di copie di un giornale dello stesso giorno proprio come per l'estrazione del DNA vengono usate cellule dello stesso tipo (o dello stesso tessuto).

Possiamo dire che questo problema di ricostruzione non è necessariamente un problema computazionale, è solo un problema logico proprio della biologia. Dal punto di vista informatico questo lo possiamo vedere come un problema di generazione di un set di sottostringhe a partire da un'unica stringa iniziale.

## 10.2 String Composition Problem

La seguente immagine è autoesplicativa:



Una volta trovate le sottostringhe esse vengono ordinate in modo lessicografico:

$$\text{COMPOSITION}_3(\text{TATGGGGTGC}) = \{\text{ATG}, \text{GGG}, \text{GGG}, \text{GGT}, \text{GTG}, \text{TAT}, \text{TGC}, \text{TGG}\}.$$

Ora, se invertiamo il problema ecco che a partire da un dizionario di k-mers vogliamo ottenere un Genoma completo. L'idea immediata è riallineare i k-mers sfruttando gli overlap di k-1 simboli, ad esempio:

TAA  
AAT

Penso sia altrettanto immediato il problema: in caso di k-mer con gli stessi caratteri in overlap noi non abbiamo informazione su quale dei due usare (perché dal punto di vista informatico quei 2 o più k-mers hanno la stessa importanza). Ora, posso risolvere questo problema introducendo una sorta di backtracking, ad esempio scelgo di fare backtracking quando arrivo in un punto dove ho usato meno k-mer di quelli di partenza. Tutto bene fin quando non mi rendo conto che sono capitato in un caso in cui sono possibili più soluzioni anche usando gli stessi k-mers...

Vabbè, allora, cerchiamo di cambiare modo di vedere il problema: come facciamo per tenere conto delle diramazioni in un percorso? Usiamo un grafo!

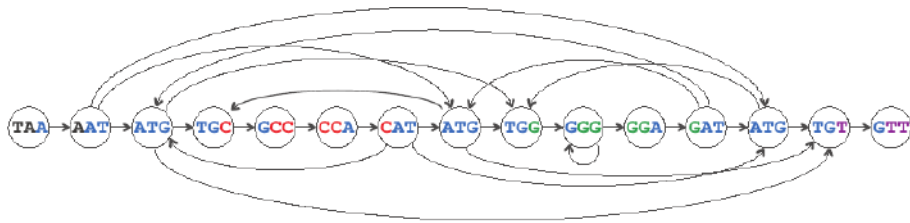
### 10.3 String Composition as a Graph path

#### 10.3.1 Hamiltonian Path

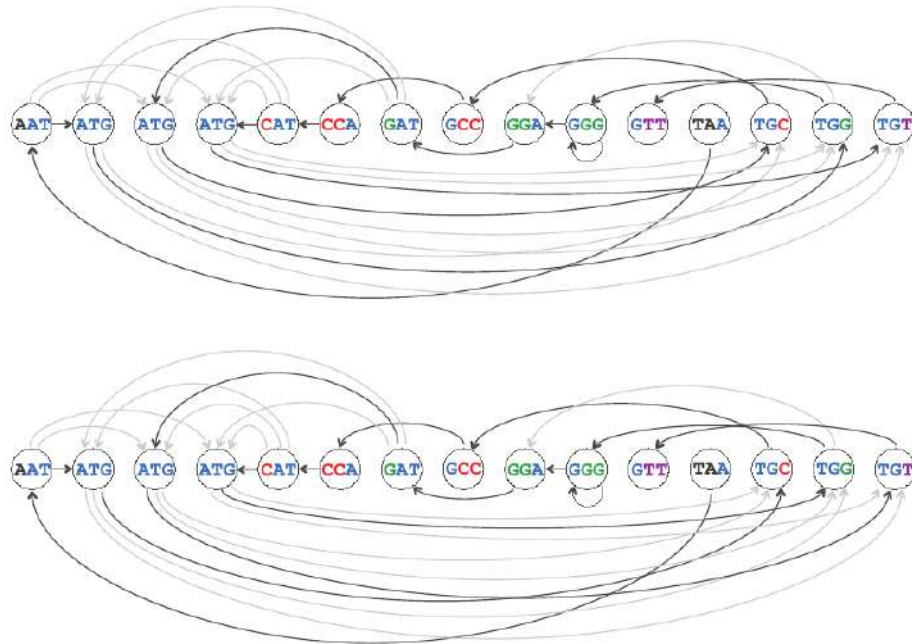
Quello che vogliamo è a partire da un insieme di k-mers arrivare ad un cammino di questo tipo:



Ma questa è la soluzione finale, perchè abbiamo tolto archi inutili, come riportato qui:



Ma anche qui, questa rappresentazione non tiene conto del fatto che in realtà noi ordiniamo i k-mer in ordine lessicografico, per cui per evitare di barare troppo dovremmo fare riferimento alla seguente configurazione:



E quindi c'è pure il rischio di contemplare più di un modo per generare un path usando tutti i k-mers una sola volta.

Ecco che in questo caso l'informatica ci viene parzialmente in soccorso, perchè questo problema è già stato studiato: si tratta della costruzione di un cammino Hamiltoniano<sup>10</sup> su un grafo (orientato nel nostro caso). Per come ci siamo inventati la suddivisione in k-mers sappiamo che la regola per la connessione di un nodo ad un altro è semplicemente la verifica della condizione secondo cui il suffisso del nodo precedente è uguale al prefisso di quello successivo, con l'unica accortezza di scegliere la lunghezza opportuna.

Attenzione! Come se non avessimo abbastanza problemi, trovare un path hamiltoniano in un grafo è un problema NP-Completo ! Per cui non sono noti algoritmi che risolvono il problema in tempo polinomiale, e questo lo rende inapplicabile nel nostro caso, perchè dobbiamo fare i conti con giga basi (nel caso umano)

### 10.3.2 Eulerian path in De Bruijn Graph

Ci serve un modo migliore per risolvere il problema, però lo vado a descrivere senza tutto il cinema che abbiamo fatto a lezione...

La soluzione è switchare dal cammino Hamiltoniano al cammino Euleriano<sup>11</sup> perchè sappiamo che esistono algoritmi polinomiali. L'unico problema è che il grafo deve essere fortemente connesso, nel senso che ogni nodo deve essere connesso almeno ad un altro, per cui noi aggiriamo l'ostacolo collegando il primo nodo con l'ultimo.

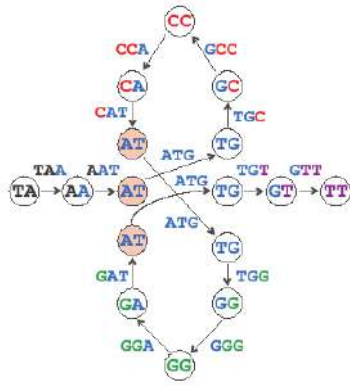
Ma andiamo per ordine. Il primo step è assegnare i k-mers agli archi anzichè ai nodi, mentre, presi due nodi collegati, all'interno del ci sarà o il prefisso dell'arco, mentre nel secondo il suffisso. Graficamente:



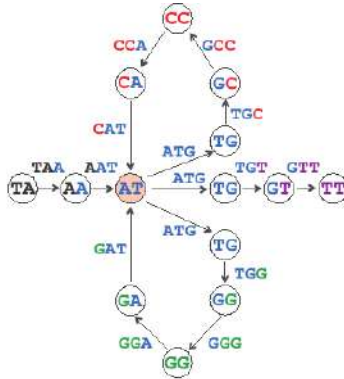
Sembrerebbe nulla di nuovo, fin quando non introduciamo uno step aggiuntivo: uniamo i nodi con le stesse label collegati da archi che hanno la stessa annotazione. Questa operazione si chiama **Glu-ing**. Graficamente:

<sup>10</sup>[https://en.wikipedia.org/wiki/Hamiltonian\\_path](https://en.wikipedia.org/wiki/Hamiltonian_path)

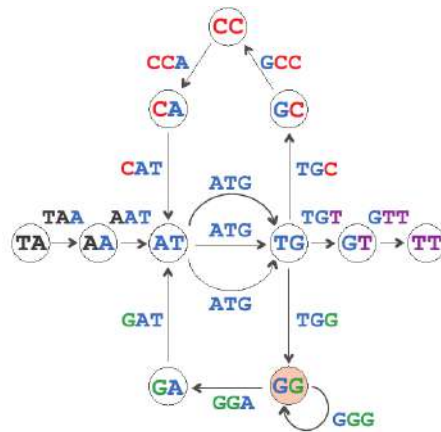
<sup>11</sup>[https://en.wikipedia.org/wiki/Eulerian\\_path](https://en.wikipedia.org/wiki/Eulerian_path)



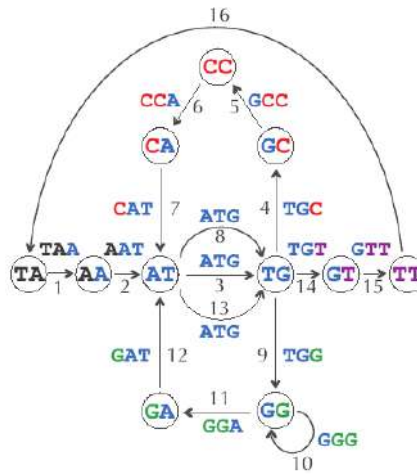
Una volta individuati i 3 nodi uguali AT, caratterizzati da archi uscenti etichettati tutti ATG, li andiamo ad unire avendo cura di mantenere i vari archi, in questo modo:



Se facciamo lo stesso per tutti i nodi uguali arriveremo ad avere un grafo Euleriano su cui far girare un algoritmo che sappiamo riesce a trovare una soluzione.



Ricordiamo che dobbiamo collegare l'inizio alla fine per ottenere un grafo davvero Euleriano:



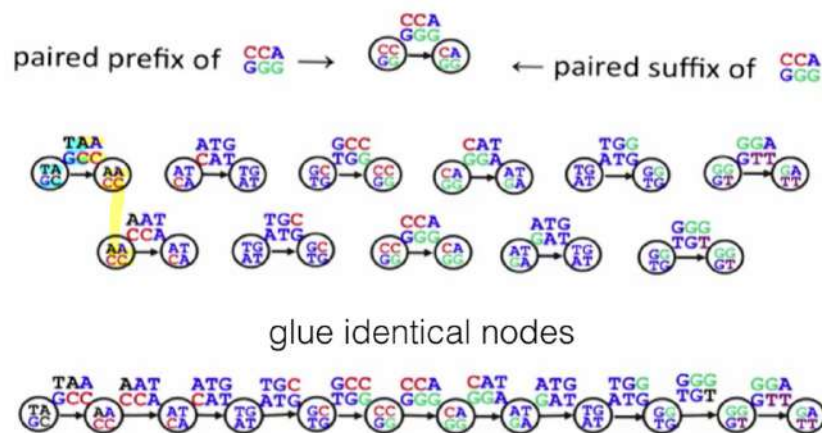
In sintesi dico che un grafo è Euleriano se ogni nodo ha almeno un arco entrante ed uno uscente. Il path Euleriano è più semplice da trovare rispetto a quello Hamiltoniano perchè richiede che ogni arco venga visitato esattamente una volta, al contrario del secondo che assume la stessa cosa ma per i nodi.

Attenzione ! Abbiamo solo spostato il problema sugli archi, perchè anche in questo caso l'algoritmo, per quanto termini in tempo ragionevole, può ritornare in output soluzioni multiple !



### 10.3.3 Unique solution with paired end reads

Per fortuna possiamo risolvere abbastanza agevolmente il problema. Se usiamo le paired end reads, avremo di fatto due De Bruijn graphs: banalmente quello con le frecce che vanno da sinistra verso destra e quello con le frecce invertite da destra verso sinistra. La soluzione unica è garantita dal fatto che entrambi i grafi devono valere: si dimostra che le soluzioni ammissibili informaticamente ma non biologicamente hanno questi due grafi incompatibili (è una mia estrema sintesi della materia). Ma andiamo per ordine: costruiamo un De Bruijn graph in questo modo:



Ed ecco che possiamo rifare tutti i passaggi avendo la certezza che solo accoppiando prefissi e suffissi allo stesso tempo otteniamo l'unica soluzione giusta.

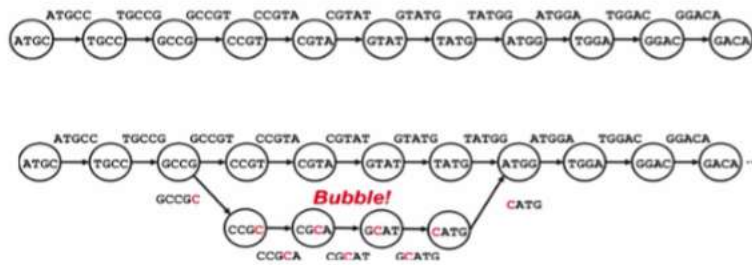
Fin qui tutto bene... ma anche no. Nel senso che abbiamo fatto come al solito assunzioni implicite molto forti, come ad esempio:

- Il coverage è perfetto ed ogni parte del genoma ha esattamente  $n$  reads che lo mappano
- le reads sono senza errori (di alta qualità)
- I k-mer sono tutti della stessa lunghezza
- Se prendiamo una read paired end, allora i due paia di k-mer iniziali e finali sono lunghi esattamente  $k$  e lo spazio in mezzo è lungo esattamente  $d$  (formalmente una read è nella forma  $k$ - $d$ - $k$ )

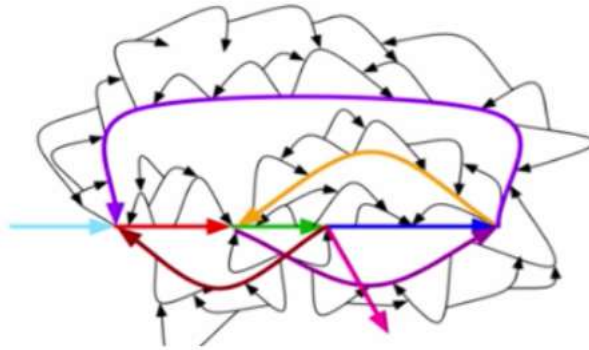
È abbastanza evidente che nella realtà le cose non funzionano così, in quanto tutte le precedenti condizioni possono essere violate. Purtroppo così facendo invalidiamo le precondizioni necessarie alla costruzione di un De Bruijn graph. Per tamponare il problema possiamo risolvere il discorso delle read di diverse

dimensioni scomponendole in k-mers fin quando non otteniamo tutte read di lunghezza uguale. Per quanto riguarda gli errori e la qualità possiamo tarare la macchina NGS in un modo diverso oppure possiamo fare preprocessing sui dati ed eliminare quello che non ci piace. Nella realtà però il metodo è abbastanza tollerante agli errori.

Vale la pena fare un esempio: in presenza di uno o più errori si vengono a creare le seguenti “Bubbles” :



Generalizzando non è raro trovarsi di fronte a milioni di errori:



Ora, esistono algoritmi efficienti per la rimozione di queste bolle (credo funzionino proprio trovando il logest path come nella blockchain), per cui non è necessario preoccuparsi in questo corso.

## 11 Combinatorial Pattern Matching

In questa sezione andiamo a studiare un modo per fare Pattern Matching combinatorio. Per fare un esempio pratico potremmo trovarci nella situazione clinica in cui dobbiamo capire se i problemi di un paziente sono riconducibili ad una particolare mutazione genetica. Caso emblematico è quello di Nicholas Volker che nel 2010 è scampato ad un intervento di rimozione del colon proprio grazie all'analisi del gene che causava l'infiammazione (bloccato con un farmaco ad-hoc successivamente).

Ora, il discorso è il seguente: come facciamo per assemblare ex novo un genoma umano ? E soprattutto: riusciamo a farlo velocemente ?

Per quanto riguarda la prima domanda quello che verrebbe in mente è di usare una tecnica di Assembly come ad esempio il De Bruijn Graph. Sfortunatamente questa tecnica non è abbastanza veloce. Allora potremmo partire da zero e ripercorrere tutte le strategie a nostra disposizione per poter trovare una soluzione al problema.

La prima cosa che possiamo fare è la solita brute-force, ma a questo punto possiamo subito tagliare corto e dire che non va bene. Anche perchè quello che vorremmo è magari comparare il genoma di un paziente ad uno “sano” di riferimento. Quindi l'idea è quella di mettere a confronto le differenze. Ma torniamo un momento a noi: vogliamo sapere per ogni pattern le eventuali posizioni in cui si trova. Per ora accontentiamoci di questo.

Dunque, la brute-force per un singolo pattern ha complessità  $O(|Genome| \cdot |Pattern|)$  e cresce all'aumentare dei pattern multipli.

### 11.1 Trie

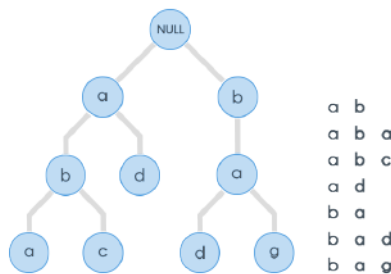


Fig. 1

La prima tecnica che vediamo fa uso di una struttura dati nota come Trie (da non confondere con gli alberi classici). Questa struttura consente di ragionare in

```

graph TD
    root((root)) -- a --> n1((n))
    root -- b --> a1((a))
    root -- n --> a2((a))
    root -- p --> a3((a))
    n1 -- n --> a4((a))
    n1 -- d --> d1((d))
    n1 -- t --> t1((t))
    a4 -- n --> n2((n))
    a4 -- a --> a5((a))
    a4 -- s --> s1((s))
    d1 -- e --> e1((e))
    t1 -- n --> n3((n))
    t1 -- n --> n4((n))
    t1 -- a --> a6((a))
    a1 -- a --> a7((a))
    a7 -- n --> n5((n))
    n5 -- a --> a8((a))
    n5 -- d --> d2((d))
    a8 -- n --> n6((n))
    a8 -- a --> a9((a))
    d2 -- a --> a10((a))
    d2 -- n --> n7((n))
    d2 -- a --> a11((a))
    a2 -- b --> b1((b))
    a2 -- n --> n8((n))
    b1 -- a --> a12((a))
    n8 -- a --> a13((a))
    a3 -- a --> a14((a))
    a14 -- n --> n9((n))
    a14 -- a --> a15((a))
  
```

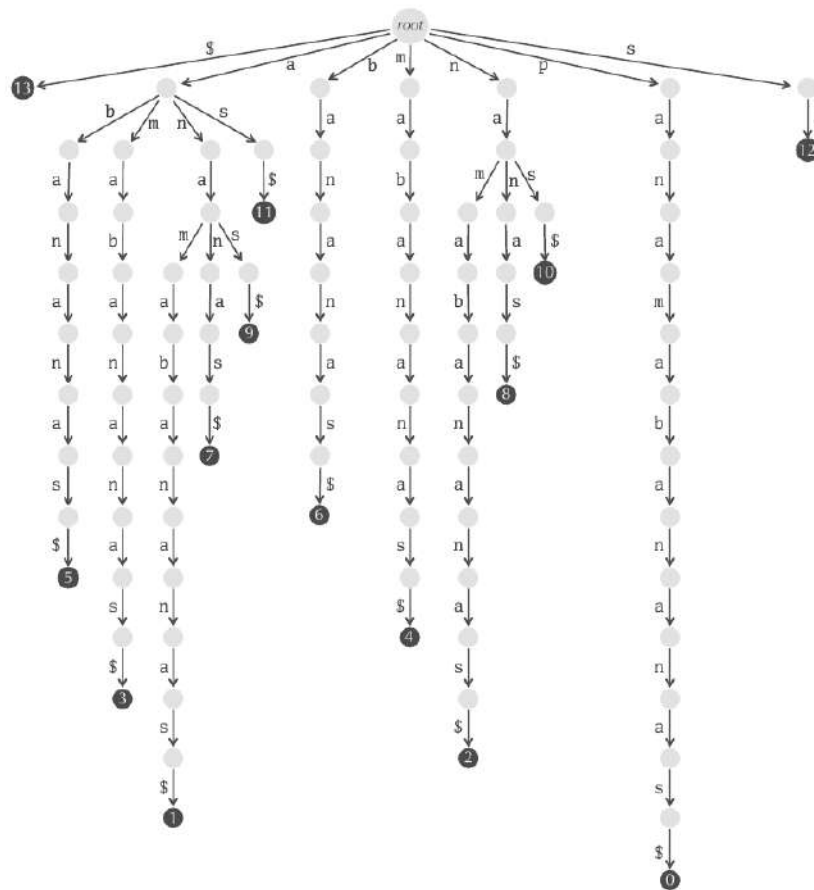
Figure 1 consists of two search trees. The left tree, labeled 'root', has three main branches: 'a', 'b', and 'p'. The 'a' branch leads to a node 'n', which then branches into 'a' and 't'. The 'b' branch leads to a node 'n', which then branches into 'a' and 'd'. The 'p' branch leads to a node 'a', which then branches into 'n' and 'a'. The right tree, also labeled 'root', has three main branches: 'a', 'b', and 'p'. The 'a' branch leads to a node 'n', which then branches into 'a' and 't'. The 'b' branch leads to a node 'n', which then branches into 'a' and 'd'. The 'p' branch leads to a node 'a', which then branches into 'n' and 'a'. In the right tree, the nodes 'a' and 'n' under the 'p' branch are highlighted in red, indicating a larger search space compared to the left tree.

dal punto di vista della complessità la situazione è la seguente: per costruire il Trie il tempo è  $O(|Patterns|)$ , però la ricerca rimane troppo costosa  $O(|Genome| \cdot |LongestPattern|)$ . Dal punto di vista della memoria è anche peggio: se consideriamo un classico Genoma umano di 3.3 Gb ed un coverage

al 50% , con una lunghezza delle read di 200 nucleotidi, allora generiamo 750 milioni di reads da moltiplicare per 200. Sono un fottio di archi !

## 11.2 Suffix Tree

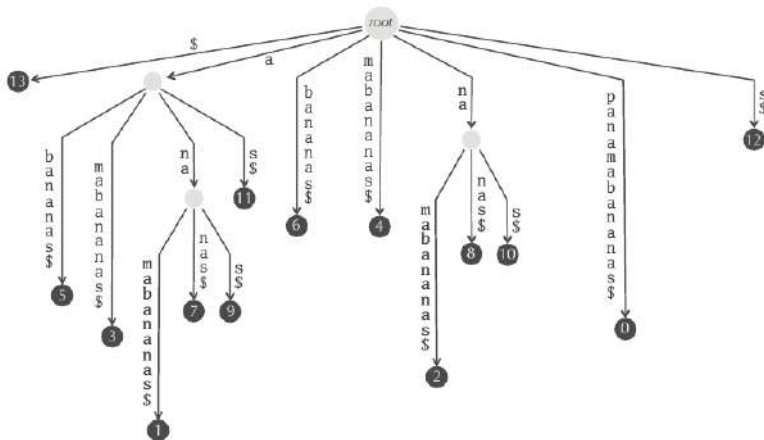
Ed ecco che ci conviene passare ad una nuova tecnica: costruiamo l'albero dei suffissi direttamente a partire dal Genoma ! Questa volta usiamo direttamente un Suffix Tree classico. A questo punto il genoma di prima, **panamabananas** (col dollaro come terminatore) da origine al seguente Suffix Tree:



Tutto bello, fin quando sorge un nuovo problema: se un pattern è più piccolo di un suffisso ci possono essere match esatti senza arrivare necessariamente ad una foglia: come gestiamo questa situazione ?

La soluzione consiste nell'aggiungere alla fine di ogni path un numero intero che rappresenta da quale punto nel genoma parte quel suffisso. Riusciamo quindi a mappare il pattern perchè inizierà dall'intero  $x$  di fine path e terminerà ad

x + lunghezza del pattern cercato. Ora, dal punto di vista della memoria non abbiamo certamente fatto un affare, infatti un albero siffatto occupa in spazio  $O(|Suffissi|)$ , quindi, per un genoma lungo  $n$  ecco che occupiamo  $O(n^2)$ . Lo step successivo è quello di tentare di comprimere quest'albero, e lo si fa inglobando tutti i nodi che hanno un solo arco entrante (in altre parole quelli da cui si può fare un unico percorso). Graficamente:



Così facendo abbiamo risolto il nostro problema di memoria, in quanto diventa tutto lineare nella lunghezza del genoma. Per quanto riguarda la ricerca siamo sempre su  $O(|Genome| + |Patterns|)$ .

Quindi, l'idea è la seguente: vogliamo aumentare ancora di più le prestazioni. Quindi siamo disposti a pagare qualcosa in termini di memoria occupata per velocizzare ancora di più l'esecuzione: questo perchè la costruzione dell'albero di un genoma viene fatta solo una volta, per cui chisseneffrega se ci mettiamo anche giorni, perchè una volta fatto useremo sempre quella struttura per ogni successivo confronto.

### 11.3 Burrows-Wheeler

Allora, facciamo un minimo di ragionamento critico: noi informatici siamo abituati a pensare con la logica degli O-grandi, per cui nei simboli di Landau sappiamo che le costanti vengono mangiate, in favore di un'indicazione approssimativa degli upper-bound. Ora nel nostro caso la migliore implementazione di un Suffix Tree richiede 20 volte la lunghezza del Genoma, ovvero lineare sì ma con quel 20 che non ci piace per niente.

Prima idea: comprimiamo il Genoma a monte attraverso un algoritmo Run-Length (quello delle telecomunicazioni che in sostanza trasmette 1000 nero anzichè 1000 volte la parola nero, quindi compressione lossless). Problema ! Il Genoma non ha molti caratteri uguali consecutivi.

Il prossimo step è quello di fare un preprocessing atto a rendere il Genoma adatto per la compressione Run Length, ovvero si vuole fare un riarrangiamento delle lettere per mettere quelle uguali il più vicino possibile. Ovviamente vogliamo fare questo senza dover memorizzarci da qualche parte la mappa delle permutazioni, perchè altrimenti non ha senso. Vogliamo quindi ricostruire in tempo costante. Ecco che entra in gioco la Trasformata di Burrows-Wheeler. Ora, per questioni pratiche vado a fare la super sintesi senza tutti i passaggi che abbiamo fatto a lezione.

Partiamo costruendo tutte le rotazioni della stringa di partenza (per semplicità riprendiamo la stringa **panamabananas\$**) ed ordiniamole successivamente in ordine lessicografico (dove il \$ ha priorità maggiore)

Cyclic Rotations	$M(\text{"panamabananas\$"})$
panamabananas\$	\$ p a n a m a b a n a n a <b>s</b>
\$panamabananas	a b a n a n a s \$ p a n a <b>m</b>
s\$panamabanana	a m a b a n a n a s \$ p a <b>n</b>
as\$panamabanan	a n a m a b a n a n a s \$ <b>p</b>
nas\$panamabana	a n a n a s \$ p a n a m a <b>b</b>
anas\$panamaban	a n a s \$ p a n a m a b a <b>n</b>
nanas\$panamaba	a s \$ p a n a m a b a n a <b>n</b>
ananas\$panamab	b a n a n a s \$ p a n a m <b>a</b>
bananas\$panama	m a b a n a n a s \$ p a n <b>a</b>
abananas\$panam	n a m a b a n a n a s \$ p <b>a</b>
mabananas\$pana	n a n a s \$ p a n a m a b <b>a</b>
amabananas\$pan	n a s \$ p a n a m a b a n <b>a</b>
namabananas\$pa	p a n a m a b a n a n a s <b>\$</b>
anamabananas\$p	s \$ p a n a m a b a n a n a <b>a</b>

La magia è presto fatta: questo metodo ci dice che la prima colonna può essere compressa attraverso un intero per ogni lettera (nel nostro caso 4 lettere per 4 byte di un intero = 16 bytes), questo perchè abbiamo ottenuto tutte lettere uguali consecutive. Per quanto riguarda l'ultima colonna diciamo che applicando un Run-Lenght abbiamo un'altra probabilità che questa stringa abbia lettere uguali consecutive (comunque molto più rispetto al genoma di partenza).

Ed ecco che ci basta solo l'ultima colonna compressa per ricostruirci tutto il Genoma originale ! Davvero un gran bel colpo !

## 11.4 Reversing BWT

L'idea è la seguente: a partire dall'ultima colonna la copio e ordino questa copia in ordine alfabetico. A questo punto mi bastano queste 2 colonne per ricostruire il genoma originale. Questo grazie ad una proprietà che ha del magico: se io indicizzo le lettere (esempio  $a_1, a_2, \dots, g_1, g_2, \dots$ ) gli indici delle occorrenze della colonna ordinata che mettiamo a sinistra corrispondono esattamente a quelli delle prime occorrenze di quei caratteri a destra. Graficamente :

\$	p	a	n	a	m	a	b	a	n	a	n	a	s
$a_1$	b	a	n	a	n	a	s	\$	p	a	n	a	m
$a_2$	m	a	b	a	n	a	n	a	s	\$	p	a	n
$a_3$	n	a	m	a	b	a	n	a	n	a	s	\$	p
$a_4$	n	a	n	a	s	\$	p	a	n	a	m	a	b
$a_5$	n	a	s	\$	p	a	n	a	m	a	b	a	n
$a_6$	s	\$	p	a	n	a	m	a	b	a	n	a	n
b	a	n	a	n	a	s	\$	p	a	n	a	m	$a_1$
m	a	b	a	n	a	n	a	s	\$	p	a	n	$a_2$
n	a	m	a	b	a	n	a	n	a	s	\$	p	$a_3$
n	a	n	a	s	\$	p	a	n	a	m	a	b	$a_4$
n	a	s	\$	p	a	n	a	m	a	b	a	n	$a_5$
p	a	n	a	m	a	b	a	n	a	n	a	s	\$
s	\$	p	a	n	a	m	a	b	a	n	a	n	$a_6$

Questo ci porta diretti al metodo che usiamo: se leggiamo l'immagine diremo che ad esempio, prima del \$ troviamo  $s_1$ , prima di  $a_1$  troviamo la lettera m, prima di  $a_2$  troviamo una n. E così via.

Facciamo un esempio pratico: vogliamo ricostruire **abracadabra\$**. Partiamo dalle 2 colonne solite:

$\$_1$	a	?	?	?	?	?	?	?	?	?	?	?	$a_1$
$a_1$	?	?	?	?	?	?	?	?	?	?	?	?	$r_1$
$a_2$	?	?	?	?	?	?	?	?	?	?	?	?	$d_1$
$a_3$	?	?	?	?	?	?	?	?	?	?	?	?	$\$_1$
$a_4$	?	?	?	?	?	?	?	?	?	?	?	?	$r_2$
$a_5$	?	?	?	?	?	?	?	?	?	?	?	?	$c_1$
$b_1$	?	?	?	?	?	?	?	?	?	?	?	?	$a_2$
$b_2$	?	?	?	?	?	?	?	?	?	?	?	?	$a_3$
$c_1$	?	?	?	?	?	?	?	?	?	?	?	?	$a_4$
$d_1$	?	?	?	?	?	?	?	?	?	?	?	?	$a_5$
$r_1$	?	?	?	?	?	?	?	?	?	?	?	?	$b_1$
$r_2$	?	?	?	?	?	?	?	?	?	?	?	?	$b_2$



Il procedimento è il seguente: partiamo dal dollaro, andiamo a vedere a destra da cosa è preceduto. Ci rendiamo conto che nel nostro caso c'è  $a_1$  e lo segniamo:

$$\$ a_1$$

Ora torniamo alla prima colonna e vediamo a cosa corrisponde  $a_1$ . Notiamo che c'è  $r_1$  e lo segniamo:

$$\$ a_1 r_1$$

Ad  $r_1$  corrisponde  $b_1$ , per cui :

$$\$ a_1 r_1 b_1$$

E così via fino ad arrivare alla stringa al contrario:

$$\$ a_1 r_1 b_1 a_2 d_1 a_5 c_1 a_4 r_2 b_2 a_3$$

Se reversiamo la stringa otterremo infatti :

$$abracadabra\$$$

Questo ci suggerisce che effettivamente le ricerche vanno fatte con l'input al contrario, così che il risultato sia coerente con quello che ci aspettiamo.

Ora, facciamo un rapido recap di quello che vogliamo:

- Dobbiamo fare Pattern Matching, per cui nella logica di aumentare le prestazioni siamo passati dai Trie, ai Suffix Tree ed ora alla BWT.
- La migliore implementazione di Suffix Tree ha come runtime  $O(|Genome| + |Patterns|)$
- Sempre parlando di Suffix Tree sappiamo che l'occupazione di memoria è lineare nella lunghezza del Genoma al netto di un fattore 20 che non ci piace per niente.
- La domanda era quindi se possiamo usare la BWT come dato di partenza per fare Pattern Matching.

La risposta per fortuna (una delle poche del corso) è affermativa. Ma prima di sorridere troppo dovremo affrontare il problema di fornire anche i dati sul posizionamento dei pattern (una vera seccatura). Nella pagina seguente metto un blocco di slide che fa vedere più o meno come fare il pattern matching senza i dati del posizionamento:

- Search for **ana** in **panamabananas**

```

$1 panamabananas1
a1 bananass$panam1
a2 mabananass$pan1
a3 namabananass$pa1
a4 nanass$panama1
a5 nas$panamabana2
a6 s$panamabanan3
b1 ananass$panama1
m1 abananass$pana2
n1 amabananass$pa3
n2 anass$panamaba4
n3 as$panamabana5
p1 anamabananass1
s1 s$panamabanan6

```

- We can only store the first and the last columns.

M. Sussan BIOINFORMATICS November 2018 36

### Finding Pattern Matches Using BWT

- Search for **ana** in **panamabananas**

```

$1 panamabananas1
a1 bananass$panam1
a2 mabananass$pan1
a3 namabananass$pa1
a4 nanass$panama1
a5 nas$panamabana2
a6 s$panamabanan3
b1 ananass$panama1
m1 abananass$pana2
n1 amabananass$pa3
n2 anass$panamaba4
n3 as$panamabana5
p1 anamabananass1
s1 s$panamabanan6

```

- We can only store the first and the last columns.

M. Sussan BIOINFORMATICS November 2018 37

### Finding Pattern Matches Using BWT

- Search for **ana** in **panamabananas**

```

$1 panamabananas1
a1 bananass$panam1
a2 mabananass$pan1
a3 namabananass$pa1
a4 nanass$panama1
a5 nas$panamabana2
a6 s$panamabanan3
b1 ananass$panama1
m1 abananass$pana2
n1 amabananass$pa3
n2 anass$panamaba4
n3 as$panamabana5
p1 anamabananass1
s1 s$panamabanan6

```

## 11.5 Finding Matched


In questa ultima sezione andremo a vedere quelle tecniche che ci consentono (pur pagandolo in spazio) di ottenere anche le posizioni delle stringhe che abbiamo trovato. La prima tecnica che vediamo è il Suffix Array.

### 11.5.1 Suffix Array

$M(\text{Text})$	$\text{SUFFIXARRAY}(\text{Text})$
\$ p a n a m a b a n a n a s	13
a b a n a n a s \$ p a n a m	5
a m a b a n a n a s \$ p a n	3
a n a m a b a n a n a s \$ p	1
a n a n a s \$ p a n a m a b	7
a n a s \$ p a n a m a b a n	9
a s \$ p a n a m a b a n a n	11
b a n a n a s \$ p a n a m a	6
m a b a n a n a s \$ p a n a	4
n a m a b a n a n a s \$ p a	2
n a n a s \$ p a n a m a b a	8
n a s \$ p a n a m a b a n a	10
p a n a m a b a n a n a s \$	0
s \$ p a n a m a b a n a n a	12

Per spiegarla in modo comprensibile da me, ad ogni matrice  $M$  associamo un array verticale le cui componenti sono il numero di lettere dopo il \$ per ogni riga di  $M$ . Diventa quindi immediato fare un check con l'immagine qui sopra. Da un punto di vista più formale l'idea è che dato che le righe sono ordinate in modo lessicografico, è chiaro che i suffissi simili compariranno in righe della matrice consecutive. In figura possiamo vedere come questo sia vero nel caso della ricerca della parola *ana*. Nel nostro caso, se vogliamo sapere dove inizia questa parola all'interno dell'intero Genoma *panamabananas\$* diciamo senza ombra di dubbio in 3 punti, ovvero nelle posizioni 1 7 e 9. Graficamente:

Thus, **ana** occurs at  
positions **1, 7, 9** of  
**panamabananas\$**



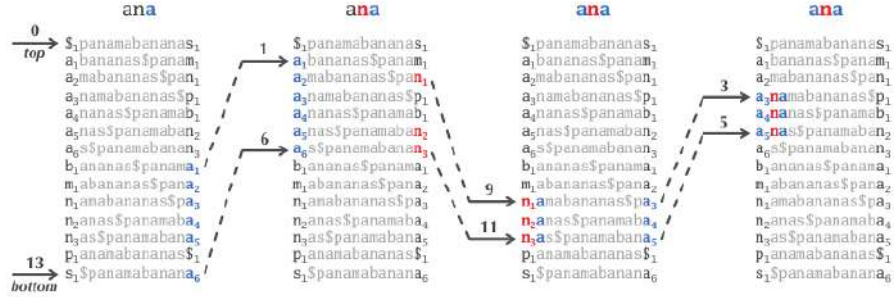
Ora facciamo un po' di conti della serva: usando un Suffix Array con interi di 4 bytes ci accorgiamo che il nostro impatto sulla memoria è diventato di 4 volte il Genoma anzichè il 20 iniziale. Non male !

Ma come al solito abbiamo un piccolo problema: dobbiamo memorizzarci la matrice intera ! E questo ci distrugge tutto, perchè torniamo ad un impatto di memoria di  $O(|Genoma|^2)$ .

### 11.5.2 Moving Backward through a pattern

Torniamo quindi indietro a quando avevamo solo le 2 colonne: sappiamo comunque trovare i pattern ma non sappiamo dire dove essi appaiano. Quello che facciamo è cercare qualche trucco per aiutarci.

La prima tecnica che ci viene in mente è quella di usare dei puntatori mobili, in questo modo:



L'idea molto alla buona è la seguente: introduciamo una funzione LASTTO-FIRST() che ci conta quante lettere ci sono tra top e bottom (bottom-top+1, perchè partiamo da 0). Se dobbiamo cercare la solita parola *ana*, cerchiamo prima dove sono tutte le *a* (ricerca dicotomica) poi passiamo a cercare tutte le *n* che precedono queste *a* ed infine tutte le *a* che precedono le *n* appena trovate. In generale, Last-To-First(i) si calcola nel seguente modo:

<i>i</i>	FirstColumn	LastColumn	LASTTOFIRST(i)	COUNT
0	\$ <sub>1</sub>	s <sub>1</sub>	13	\$ a b m n p s
1	a <sub>1</sub>	m <sub>1</sub>	8	0 0 0 0 0 0 1
2	a <sub>2</sub>	n <sub>1</sub>	9	0 0 0 1 0 0 1
3	a <sub>3</sub>	p <sub>1</sub>	12	0 0 0 1 1 0 1
4	a <sub>4</sub>	b <sub>1</sub>	7	0 0 0 1 1 1 1
5	a <sub>5</sub>	n <sub>2</sub>	10	0 0 1 1 1 1 1
6	a <sub>6</sub>	n <sub>3</sub>	11	0 0 1 1 2 1 1
7	b <sub>1</sub>	a <sub>1</sub>	1	0 0 1 1 3 1 1
8	m <sub>1</sub>	a <sub>2</sub>	2	0 1 1 1 3 1 1
9	n <sub>1</sub>	a <sub>3</sub>	3	0 2 1 1 3 1 1
10	n <sub>2</sub>	a <sub>4</sub>	4	0 3 1 1 3 1 1
11	n <sub>3</sub>	a <sub>5</sub>	5	0 4 1 1 3 1 1
12	p <sub>1</sub>	\$ <sub>1</sub>	0	0 5 1 1 3 1 1
13	s <sub>1</sub>	a <sub>6</sub>	6	1 5 1 1 3 1 1
				1 6 1 1 3 1 1

Allora, per capire quanto vale  $\text{Last-To-First}(0)$  vado a vedere a quale carattere corrisponde quello in posizione 0. Scopro quindi che  $s_1$  corrisponde a  $s_1$ .  $\text{Last-To-First}(0)$  allora è uguale al valore di  $i$  a cui corrisponde  $s_1$ . Dalla figura possiamo notare che  $s_1$  nella prima colonna ha  $i = 13$ , per cui concludiamo che  $\text{Last-To-First}(0) = 13$ . E così via per tutti quanti gli  $i$ . Il problema di questa tecnica è che non riusciamo ad aggiornare i puntatori in modo costante... Infatti l'algoritmo è il seguente:

```

BWMATCHING(FirstColumn, LastColumn, Pattern, LASTTOFIRST)
  top  $\leftarrow$  0
  bottom  $\leftarrow$  |LastColumn| - 1
  while top  $\leq$  bottom
    if Pattern is nonempty
      symbol  $\leftarrow$  last letter in Pattern
      remove last letter from Pattern
      if positions from top to bottom in LastColumn contain symbol
        topIndex  $\leftarrow$  first position of symbol among positions from top to bottom
          in LastColumn
        bottomIndex  $\leftarrow$  last position of symbol among positions from top to
          bottom in LastColumn
        top  $\leftarrow$  LASTTOFIRST(topIndex)
        bottom  $\leftarrow$  LASTTOFIRST(bottomIndex)
      else
        return 0
    else
      return bottom - top + 1

```

Nella tabella precedente però ho introdotto il concetto di funzione count, ovvero  $\text{COUNT}_l(i, \text{Genoma})$ , che sostanzialmente calcola il numero di occorrenze della lettera  $l$  all'interno della sottostringa lunga  $i$  del Genoma. Ad esempio:

$$\text{COUNT}_{\text{"n"}}(10, \text{"smnpbnnaaaaa$a"}) = 3$$

Ora, riproponiamo la tabella di prima e spieghiamo come fare a calcolare il campo dei count.

<i>i</i>	<i>FirstColumn</i>	<i>LastColumn</i>	LASTTOFIRST( <i>i</i> )	COUNT						
				\$	a	b	m	n	p	s
0	\$ <sub>1</sub>	s <sub>1</sub>	13	0	0	0	0	0	0	0
1	a <sub>1</sub>	m <sub>1</sub>	8	0	0	0	0	0	0	1
2	a <sub>2</sub>	n <sub>1</sub>	9	0	0	0	1	0	0	1
3	a <sub>3</sub>	p <sub>1</sub>	12	0	0	0	1	1	0	1
4	a <sub>4</sub>	b <sub>1</sub>	7	0	0	0	1	1	1	1
5	a <sub>5</sub>	n <sub>2</sub>	10	0	0	1	1	1	1	1
6	a <sub>6</sub>	n <sub>3</sub>	11	0	0	1	1	2	1	1
7	b <sub>1</sub>	a <sub>1</sub>	1	0	0	1	1	3	1	1
8	m <sub>1</sub>	a <sub>2</sub>	2	0	1	1	1	3	1	1
9	n <sub>1</sub>	a <sub>3</sub>	3	0	2	1	1	3	1	1
10	n <sub>2</sub>	a <sub>4</sub>	4	0	3	1	1	3	1	1
11	n <sub>3</sub>	a <sub>5</sub>	5	0	4	1	1	3	1	1
12	p <sub>1</sub>	\$ <sub>1</sub>	0	0	5	1	1	3	1	1
13	s <sub>1</sub>	a <sub>6</sub>	6	1	5	1	1	3	1	1
				1	6	1	1	3	1	1

Dunque, è molto semplice: all'inizio, quindi nello step 0, non abbiamo incontrato nessuna occorrenza di nulla. Nello step successivo incontriamo  $s_1$ , per cui andiamo ad aumentare di 1 il contatore delle  $s$ . Nello step 2 invece incontriamo  $m_1$ , per cui dobbiamo incrementare il contatore opportuno. E così via fin quando non arriviamo alla fine. Andiamo adesso ad inventarci una funzione FISTOCCURRENCE(symbol) che ci dice la posizione del primo simbolo trovato.

<i>i</i>	<i>FirstColumn</i>	FIRSTOCCURRENCE
0	\$ <sub>1</sub>	0
1	a <sub>1</sub>	1
2	a <sub>2</sub>	
3	a <sub>3</sub>	
4	a <sub>4</sub>	
5	a <sub>5</sub>	
6	a <sub>6</sub>	
7	b <sub>1</sub>	7
8	m <sub>1</sub>	8
9	n <sub>1</sub>	9
10	n <sub>2</sub>	
11	n <sub>3</sub>	
12	p <sub>1</sub>	12
13	s <sub>1</sub>	13

A questo punto includo la spiegazione del libro perchè è semplice:

When  $top = 1$ ,  $bottom = 6$ , and  $symbol = "n"$ , we have that

$$\begin{aligned}\text{FIRSTOCCURRENCE}("n") &= 9 \\ \text{COUNT}_{"n"}(top, \text{LastColumn}) &= 0 \\ \text{COUNT}_{"n"}(bottom + 1, \text{LastColumn}) &= 3\end{aligned}$$

Recalling Figure 9.14, this again implies that  $(top = 1, bottom = 6)$  will be updated as

$$\begin{aligned}top &= 9 + 0 = 9 \\ bottom &= 9 + 3 - 1 = 11\end{aligned}$$

Ed ecco che abbiamo sostituito la prima colonna della matrice con la funzione FIRSTCOLUMN, mentre LAST-TO-FIRST è stata sostituita da COUNT. Abbiamo ottenuto quindi un grande aumento di velocità.

```
BETTERBWMATCHING(FIRSTOCCURRENCE, LastColumn, Pattern, COUNT)
  top ← 0
  bottom ← |LastColumn| - 1
  while top ≤ bottom
    if Pattern is nonempty
      symbol ← last letter in Pattern
      remove last letter from Pattern
      top ← FIRSTOCCURRENCE(symbol) + COUNTsymbol(top, LastColumn)
      bottom ← FIRSTOCCURRENCE(symbol) + COUNTsymbol(bottom + 1,
        LastColumn) - 1
    else
      return bottom - top + 1
  return
```

Come al solito tutto bello ma non abbiamo risolto il problema, non sappiamo infatti dove sono i pattern matchati... Allora potremmo pensare di ritirare fuori dal cilindro il nostro Suffix Array, e la cosa funzionerebbe:

panamabananas\$	panamabananas\$	panamabananas\$	Partial Suffix Array
s <sub>1</sub> panamabananas <sub>1</sub>	s <sub>1</sub> panamabananas <sub>1</sub>	s <sub>1</sub> panamabananas <sub>1</sub>	13
a <sub>1</sub> bananas\$panam <sub>1</sub>	a <sub>1</sub> bananas\$panam <sub>1</sub>	a <sub>1</sub> bananas\$panam <sub>1</sub>	5
a <sub>2</sub> nabananas\$pan <sub>1</sub>	a <sub>2</sub> nabananas\$pan <sub>1</sub>	a <sub>2</sub> nabananas\$pan <sub>1</sub>	3
a <sub>3</sub> nabananas\$pan <sub>1</sub>	a <sub>3</sub> nabananas\$pan <sub>1</sub>	a <sub>3</sub> nabananas\$pan <sub>1</sub>	1
a <sub>4</sub> nanas\$panamab <sub>1</sub>	a <sub>4</sub> nanas\$panamab <sub>1</sub>	a <sub>4</sub> nanas\$panamab <sub>1</sub>	7
a <sub>5</sub> nas\$panamaban <sub>2</sub>	a <sub>5</sub> nas\$panamaban <sub>2</sub>	a <sub>5</sub> nas\$panamaban <sub>2</sub>	9
a <sub>6</sub> \$panamabanan <sub>2</sub>	a <sub>6</sub> \$panamabanan <sub>2</sub>	a <sub>6</sub> \$panamabanan <sub>2</sub>	11
b <sub>1</sub> ananas\$panama <sub>1</sub>	b <sub>1</sub> ananas\$panama <sub>1</sub>	b <sub>1</sub> ananas\$panama <sub>1</sub>	6
n <sub>1</sub> abananas\$pana <sub>2</sub>	n <sub>1</sub> abananas\$pana <sub>2</sub>	n <sub>1</sub> abananas\$pana <sub>2</sub>	4
n <sub>2</sub> abananas\$pana <sub>3</sub>	n <sub>2</sub> abananas\$pana <sub>3</sub>	n <sub>2</sub> abananas\$pana <sub>3</sub>	2
n <sub>3</sub> anas\$panamaba <sub>4</sub>	n <sub>3</sub> anas\$panamaba <sub>4</sub>	n <sub>3</sub> anas\$panamaba <sub>4</sub>	8
n <sub>4</sub> as\$panamabana <sub>5</sub>	n <sub>4</sub> as\$panamabana <sub>5</sub>	n <sub>4</sub> as\$panamabana <sub>5</sub>	10
p <sub>1</sub> anamabananas\$ <sub>1</sub>	p <sub>1</sub> anamabananas\$ <sub>1</sub>	p <sub>1</sub> anamabananas\$ <sub>1</sub>	0
s <sub>2</sub> \$panamabanan <sub>6</sub>	s <sub>2</sub> \$panamabanan <sub>6</sub>	s <sub>2</sub> \$panamabanan <sub>6</sub>	12

FIGURE 9.17 One of the matches of "ana" in "panamabananas\$" is highlighted in the matrix on the right. By walking backward, we find that "ana" is preceded by "b<sub>1</sub>", which in turn is preceded by "a<sub>1</sub>". The partial suffix array above, generated for  $K = 5$ , indicates that "a<sub>1</sub>" occurs at position 5 of "panamabananas". Since it took us two steps to walk backward to "a<sub>1</sub>", we conclude that this occurrence of "ana" begins at position  $5 + 2 = 7$ .

Ora, dobbiamo aggiustare questo Trade-off di fatto che si è creato, perchè non vogliamo usare troppa memoria precomputando tutto ma non vogliamo nemmeno ricalcolarci on-the-fly tutti gli step intermedi. Allora adottiamo la tecnica dei checkpoint:

$i$	LastColumn	COUNT						
		\$	a	b	m	n	p	s
0	s <sub>1</sub>	0	0	0	0	0	0	0
1	m <sub>1</sub>	0	0	0	0	0	0	1
2	n <sub>1</sub>	0	0	0	1	0	0	1
3	p <sub>1</sub>	0	0	0	1	1	0	1
4	b <sub>1</sub>	0	0	0	1	1	1	1
5	n <sub>2</sub>	0	0	1	1	1	1	1
6	n <sub>3</sub>	0	0	1	1	2	1	1
7	a <sub>1</sub>	0	0	1	1	3	1	1
8	a <sub>2</sub>	0	1	1	1	3	1	1
9	a <sub>3</sub>	0	2	1	1	3	1	1
10	a <sub>4</sub>	0	3	1	1	3	1	1
11	a <sub>5</sub>	0	4	1	1	3	1	1
12	s <sub>1</sub>	0	5	1	1	3	1	1
13	a <sub>6</sub>	1	5	1	1	3	1	1
		1	6	1	1	3	1	1

FIGURE 9.18 The COUNT checkpoint arrays for  $Text = \text{"panamabananas\$"} and  $C = 5$  are highlighted in bold. If we want to compute  $COUNT_{\text{"a"}}(13, \text{"smnpbnnaaaaa\$a"})$ , then the checkpoint array at position 10 tells us that there are 3 occurrences of "a" before position 10 of "smnpbnnaaaaa\$a". We then check whether "a" is present at position 10 (yes), 11 (yes), and 12 (no) of LastColumn to conclude that  $COUNT_{\text{"a"}}(13, \text{"smnpbnnaaaaa\$a"}) = 3 + 2 = 5$ .$



L'idea è quindi fare storage solo se  $i$  è divisibile per una costante  $C$  di nostra scelta (nella figura  $C=5$ ). Ed ecco quindi che possiamo fare questo pattern matching in questo modo:

	FirstCol	BWT(Text)	Suffix Array
0	\$	s <sub>1</sub>	13
1	<b>a<sub>1</sub></b>	m <sub>1</sub>	<b>5</b>
	a <sub>2</sub>	n <sub>1</sub>	3
	a <sub>3</sub>	p <sub>1</sub>	1
	<b>a<sub>4</sub>na</b>	<b>b<sub>1</sub></b>	7
	a <sub>5</sub>	n <sub>2</sub>	9
	a <sub>6</sub>	n <sub>3</sub>	11
7	<b>b<sub>1</sub></b>	a <sub>1</sub>	6
8	m <sub>1</sub>	a <sub>2</sub>	4
9	n <sub>1</sub>	a <sub>3</sub>	2
	n <sub>2</sub>	a <sub>4</sub>	8
	n <sub>3</sub>	a <sub>5</sub>	<b>10</b>
12	p <sub>1</sub>	\$ <sub>1</sub>	<b>0</b>
13	s <sub>1</sub>	a <sub>6</sub>	12

## 11.6 Epilogo

### 11.6.1 Pattern Matching Inesatto

Come ultimo argomento del corso andiamo a vedere concettualmente come risolvere il problema del pattern matching inesatto. Innanzitutto il problema è uguale a quello del pattern matching di prima con l'aggiunta di un numero intero  $d$  che indica il numero massimo di mismatch tollerati. Noi per semplicità supporremo  $d = 1$ .

Ora, l'idea è la seguente (la dico molto alla buona): abbiamo una stringa molto lunga di partenza, sulla quale vogliamo fare pattern matching con al più un errore. È chiaro che essendoci al più un errore una volta capito dove esso si trova tutte le sottostringhe prima e dopo quel punto fanno per forza match esatto.

Un esempio grafico può essere il seguente:

<i>Pattern</i>	acttggt
<i>Text</i>	...ggcacactaggctcc...

Perfetto, non ci resta che sfruttare questa proprietà per trovare una soluzione a questo problema. Questo perché altrimenti ogni qualvolta facciamo una comparazione con la BWT dovremmo anche riscrivere il programma affinché accetti anche un errore e questo vorrebbe dire aggiungere backtracking aumentando di fatto la complessità dell'algoritmo.

Per fortuna a questo punto del corso ci viene in soccorso un potente teorema:

**Teorema** : Se un pattern compare in un genoma con  $d$  mismatch, allora possiamo

dividere il pattern in  $d + 1$  sottopattern e otterremo per forza di cose almeno un match esatto. Visivamente:

<i>Pattern</i>	acttaggctcgggataatcc
<i>Text</i>	...actaagtctcgggataagcc...

In questo esempio abbiamo supposto  $d = 3$  ed abbiamo diviso il nostro pattern in  $d + 1 = 4$  parti (verde, blu, arancione e viola). Si dimostra che per testi molto lunghi conviene questo approccio, perché più efficiente del backtracking.

Per quanto riguarda valori di  $d$  molto piccoli e testi più corti va bene fare backtracking con la BWT modificata. Per ulteriori informazioni si rimanda alle slide.

## 12 Conclusione

Con questo si conclude la parte da 6 crediti del corso. Grazie per l'attenzione fin qui, se ci sono commenti/suggerimenti/errori segnalatemeli via Telegram.

Ultimo ma non ultimo, la licenza :

*Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. You can find a copy of the license at <https://www.gnu.org/licenses/fdl-1.3.html>*