Bioinformatica

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

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

Capitolo 2

Introduzione alla bioinformatica

La genomica ha dimostrato negli ultimi anni ha dimostrato una capacità incredibile di produrre dati e questo ha portato alla nascita del bioinformatico, che diventa un esperto della gestione di questi dati sia dal punto di vista algoritmico che dal punto vista sistemistico.
A partire dal 2000/2001 ma sopratutto poco prima del 2010 si ha una crescita dei **dati genomici** non indifferente. I dati genomici sono quelli provenienti dal sequenziamento del DNA. Negli ultimi anni questa crescita ha superato la curva della **legge di Moore** quindi la crescita in termini di hardware (che si stima migliorare ogni 18 mesi) non riesce più a soddisfare la stima di richiesta di hardware necessario per il sequenziamento. Questa stima di sequenziamento è basata su Illumina, che produce le più diffuse macchine fisiche per il sequenziamento. Case farmaceutiche e laboratori che studiano il sequenziamento hanno almeno una macchina Illumina. La quantità di dati ha raggiunto i livelli dei petabyte e quindi ci si aspetta (e in parte già è così) che l'hardware non sia più in grado di elaborare tali dati.
La bioinformatica riceve quindi questa tipologia di dati. La bioinformatica è cruciale nell'ambito della ricerca in biologia molecolare (riguardante prettamente DNA), dove sempre più si ha necessità dell'appoggio dell'informatica, avendo a che fare con dati, nel dettaglio grandi dati.
Un altro aspetto è quello legato alle nanotecnologie e alla così detta **DNA-based computation**. Un esempio è legato al fatto che ormai si è in grado di manipolare il DNA al punto di essere in grado di assemblarlo in laboratorio, tramite un meccanismo a *tiling (tasselli)*, dove il tiling tendenzialmente è una figura regolare (triangolare, rettangolare, esagonale, etc...) con cui si compone del materiale biologico. Si riescono a fare letteralmente figure con il DNA (anche stelle, smile etc...) ma, sopratutto di questi tempi, vaccini,

che sono appunto manipolazione genetica di DNA o RNA. **Questa parte non è trattata nel corso**.

2.1 Breve introduzione biologica

Nel corso tratteremo prevalentemente sequenze di DNA. All'interno della cellula si hanno i **cromosomi** e un **genoma** altro non è che la collezione di cromosomi all'interno di un individuo. Il singolo cromosoma è rappresentato da filamenti di DNA "attorcigliati". Il cromosoma sostanzialmente è formato dalla coppia di due filamenti che si uniscono in una parte centrale detta **centromero**. I cromosomi, dal punto di vista informatico, sono vere e proprie sequenze (con i 4 nucleotidi, adenina, citosina, guanina e timina, ricordando la complementarietà delle basi A-T C-G), anche se si hanno varie regole per gestire questa "semplificazione". Un altro aspetto è il passaggio dal DNA alle **proteine**, anche se nel corso non verrà trattata la **proteomica**, ovvero lo studio delle proteine in se. In merito al passaggio da DNA a proteine si ha che il DNA contiene i **geni** da cui poi derivano le proteine. Un gene può portare a più di una proteina e questo si è scoperto grazie al sequenziamento. Allo stato attuale per "leggere" il DNA di un individuo dobbiamo passare per macchine di sequenziamento che però non possono leggerlo interamente ma, prendendo il DNA da una provetta (anche a partire da una singola cellula nel **sequenziamento single-cell**), si ha in output un file con dei frammenti del DNA originale, replicati in coppie, dette **read**. Tramite vari algoritmi siamo poi in grado di arrivare a capire e studiare il DNA per poi arrivare, si spera, ad uno dei principali fini della bioinformatica, quello di curare la vita, tramite terapie mediche (si parla di **medicina traslazionale**, ovvero non curo un paziente tramite protocolli generali ma sulla base del DNA del paziente, che viene studiato ai fini di stabilire la migliore terapia, che diventa personalizzata per l'individuo). Le scoperte biologiche più attuali sono ottenute praticamente sempre grazie all'intervento anche dell'informatica e della bioinformatica.
Un esempio di uso delle sequenze è confrontare regioni genomiche di varie specie per valutare eventuali somiglianze. Un primo modo è diretto, un secondo è confrontare dopo l'allineamento, con l'inserimento di gap (studieremo la cosa nel dettaglio).
Il bioinformatico fornisce al biologo/biotecnologo la strumentazione necessaria per fare le varie analisi.

2.2 Progetto Genoma Umano

Un elemento chiave nella bioinformatica è il **Human Genome Project (*progetto genoma umano*)**, progetto partito prima del 2000 (la prima base è del 1990) con vari obiettivi:

- identificare tutti i circa 30.000 geni nel DNA umano
- determinare le sequenze dei 3 miliardi di coppie di basi chimiche che compongono il DNA umano
- memorizzare queste informazioni in banche dati/db
- migliorare gli strumenti per l'analisi dei dati

La bioinformatica è andata avanti quasi sempre con progetti globali e il Progetto Genoma Umano è stato il primo di questi progetti, diciamo che lì nacque la bioinformatica. Si hanno vari *milesstones*:

- *1990:* progetto avviato come sforzo congiunto del U.S. Department of Energy e del National Institutes of Health (NIH)
- *Giugno 2000:* completamento di una bozza di lavoro dell'intero genoma umano
- *Febbraio 2001:* vengono pubblicate le analisi della bozza di lavoro
- *Aprile 2003:* Il sequenziamento del Progetto Genoma Umano è completato e il progetto è dichiarato finito due anni prima del previsto

Quest'anno, nel 2020, è stato lanciato un progetto ulteriore in quanto ora si è anche in grado di sequenziare il DNA nei pressi dei **telomeri**, ovvero le terminazioni dei cromosomi, che sono le regioni più difficili da ricostruire tramite il sequenziamento. Per farlo si hanno algoritmi e software davvero molto sofisticati.
Vediamo qualche numero:

- il genoma umano contiene 3 miliardi (3×10^9) di basi nucleotidiche chimiche che sono 4:
  - adenina (A)
  - citosina (C)
  - guanina (G)

  - timina (T)

- il gene mediamente è composto da 3000 basi, ma le dimensioni variano molto, con il più grande gene umano noto che è la Distrofina con 2.4 milioni di basi

- il numero totale di geni è stimato a circa 30000, molto inferiore alle stime precedenti da 80000 a 140000 (in quanto prima c'era il dogma che un gene codificasse una sola proteina, e si avevano circa 140000 proteine, che si conoscevano anche solo per le analisi del sangue)

- quasi tutte (99.9%) le basi nucleotidiche sono esattamente le stesse in tutte le persone. Basta lo 0.1% di differenze tra basi per "fare la differenza", anche differenziando predisposizioni geniche per una certa malattia

- le funzioni sono sconosciute per oltre il 50% del gene scoperto

Vediamo anche qualche numero (in stima) in merito agli organismi più studiati dai bioinformatici (spesso organismi con poche basi), più l'attualissimo *sars-cov-2*:

organismo	numero basi	numero di geni
uomo (Homo sapiens)	3 miliardi	30000
topo di laboratorio (M. musculus)	2.6 miliardi	30000
arabetta comune (A. thaliana)	100 milioni	25000
nematoda (C. elegans)	97 milioni	19000
mosca della frutta (D. melanogaster)	137 milioni	13000
lievito (S. cerevisiae)	12.1 milioni	6000
batterio (E.coli)	4.6 milioni	3200
Human immunodeficiency virus (HIV)	9700	9
sars-cov-2	$\sim$27 milioni	$\sim$15

2.3 Variazioni

Una volta conosciuta la sequenza dell'uomo si è cercato di studiare quello 0.1% di differenze tra vari esseri umani. Queste differenze sono dette **SNPs (*single nucleotide polymorphisms*)** (*detti a voce "snips"*) che rappresentano la variabilità nella popolazione umana. Sono le differenze a livello di singolo nucleotide. Subito dopo il Progetto Genoma Umano è partito, sempre

tramite il National Institutes of Health (NIH), un progetto che confrontasse popolazione africana, asiatica e statunitense per calcolare queste differenze, individuate tramite tool informatici, tramite il cosiddetto **assemblaggio di aplotipi**, che è prettamente un problema informatico, *NP-complete*, la cui soluzione più recente è data da un **algoritmo parametrico**. Dagli aplotipi vengono estratti gli SNPs e questo sarà visto tra qualche lezione. Gli SNPs sono serviti a determinare differenze tra le varie popolazioni campione in merito, ad esempio alla predisposizione alla Talassemia nelle popolazioni mediterranee. Questi studi servono appunto capire le predisposizioni delle varie popolazioni. Se una popolazione ha, nella maggior parte dei casi, una certa base in una certa posizione allora si ha uno SNPs. Il famoso 0.1% forma questi SNPs, il 99.9% della popolazione porta il cosiddetto **allele di maggioranza** mentre lo 0.1% l'**allele di minoranza**.
Uno studio ha dimostrato che, in Italia, solo i Sardi hanno un profilo genetico ben definito, tutti gli altri sono dei "mix genetici" e questo si è scoperto studiando gli SNPs.
Dal Progetto genoma Umano si è poi passati a confrontare il genoma di piccolissimi campioni, ad esempio 1000 individui, con il 1000 Genomes Project, un altro progetto con sforzi internazionali, fatto per mappare le variazioni su una popolazione di 1000 individui. Si segnala che per sequenziare un individuo ci sono voluti 10 anni nel primo caso ma poi ci è voluto molto meno. Ora un singolo individuo si sequenzia in qualche ora, a costi molto ridotti.
Dal DNA si sono anche ricavati i flussi migratori avvenuti nel corso della storia.

2.4 Pangenoma

Si vedrà, durante il corso, che dire **il genoma è una singola sequenza**, è ormai sostanzialmente errato. Avendo sequenziato milioni di individui si parla di **pangenoma** e le analisi devono ormai essere fatte non su un singolo genoma di riferimento ma si usa quello abbinato a tutta la serie di 0.1% di SNPs individuati finora. Nel dettaglio un pangenoma è una collezione di genomi multipli che sono correlati tra loro (variando solo in pochi punti). Si ha il pangenoma dell'uomo, di un batterio etc...
Dal punto di vista informatico diciamo comunque che il DNA è una sequenza sotto l'assunzione della **complementarietà delle basi**:

- adenina e timina sono complementari
- citosina e guanina sono complementari

e questo mi permette di poter studiare solo uno dei due filamenti del DNA.

Esempio 1. *Sia data la sequenza:*

$$S = acctacga$$

la complementare è:

$$S' = tggatgct$$

Se prendo la sequenza (o meglio una porzione di essa) di S_1 di un individuo h_1 e la sequenza S_2 di un individuo h_2 avrò un alta somiglianza con eventualmente uno o più SNPs.

La posizione dello SNP è detto **locus**. Uno SNP si ha quando nel 99.9% dei casi tutti gli individui hanno una certa base in una data posizione, avendo l'*allele di maggioranza*, mentre lo 0.1% degli individui ne ha una diversa, avendo l'*allele di minoranza* (e lo rilevo confrontando una popolazione).

Esempio 2. *Si hanno:*

$$S_1 = acc\mathbf{t}acga$$

$$S_2 = acc\mathbf{g}acga$$

ho uno SNP nel locus 4. Ipotizzando che il 99.9% degli individui siano come l'individuo con la sequenza s_1 ho che la base t è un allele di maggioranza mentre la base g è un allele di minoranza.

L'uomo si dice essere **biallelico** in quanto le "opzioni" per una certa posizione sono solo due. Alcuni cambiamenti possono anche essere del tipo *inserzione/delezione* (anche per sequenze di più basi contigue), parlando di **variazioni strutturali** (che sono comunque più complesse e meno tipiche). Per rappresentare il fatto che si hanno più sequenze con queste variazioni, soprattutto se sono inserimenti e delezioni, ma considerando che il 99.9% delle basi è uguale (cercando quindi una rappresentazioni che ottimizzi questa cosa), rappresentando quindi un pangenoma, dal punto di vista computazionale è un **grafo**. Ogni sequenza identica collassa in un solo nodo, avendo poi singoli nodi per le variazioni.

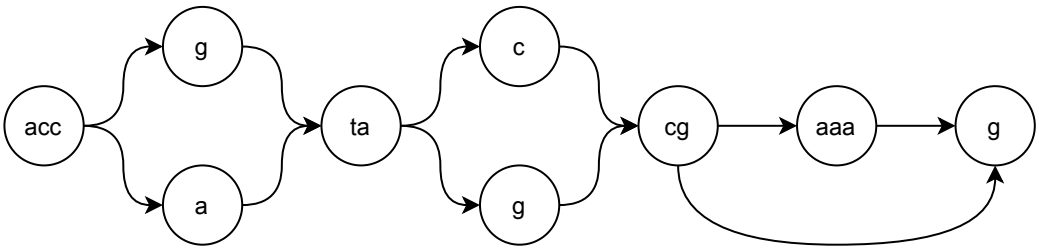
Esempio 3. *Ipotizzo di avere (con $-$ per indicare delezioni):*

$$S_1 = acc\mathbf{g}ta\mathbf{c}cg\mathbf{aaa}g$$

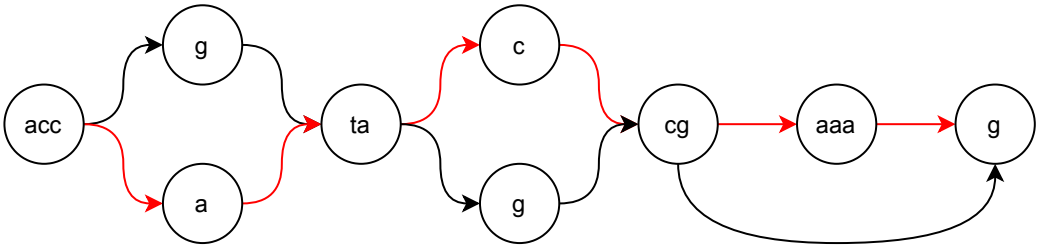
$$S_2 = acc\mathbf{a}ta\mathbf{g}cg\mathbf{aaa}g$$

$$S_3 = acc\mathbf{g}ta\mathbf{c}cg\texttt{---}g$$

E ottengo un grafo del tipo:



Studiando i cammini dei grafi ottengo tutte le rappresentazioni.
Questa rappresentazione però ha dei difetti, in quanto potrei avere cammini che non rappresentano nessuna sequenza di partenza. Pensando all'esempio sopra potrei avere il cammino in rosso che non rappresenta nessuna delle tre sequenze:



Rappresento quindi più di quello che voglio rappresentare.
Un pangenoma è un grafo che rappresenta una popolazione senza fare grandi distinzioni, avendo percorsi che non sono riscontrabili in nessun individuo della popolazione. Si ha comunque che il concetto di sequenza non è più adeguato. Il grafo di una popolazione è enorme e comunque, tramite colori, si possono distinguere i vari percorsi della popolazione (distinguendo facilmente "tracce comuni"). Parlando quindi di **genoma di riferimento** o si parla di quello specifico di un individuo o si parla del pangenoma di una popolazione, con le varianti.
Dal punto di vista di *file* le varianti vengono date in un file **Variant Call Format (*VCF*)**. L'input classico dei software è quindi spesso un VCF, così come l'output.

2.5 Progetti attuali

Vediamo ora quali sono i grandi progetti su larga scala attualmente in corso:

- **The Cancer Genome Atlas Pan-Cancer Analysis Project (*TCGA*)**, che cerca di costruire un catalogo delle caratteristiche

genomiche dei tumori, ovvero un catalogo delle mutazioni genomiche associate a tumori (ad esempio quello del seno si sa che è legato alla mutazione del gene BRCA che si sa bene dov'è)

- **The 1000 Genomes Project Consortium: A global reference for human genetic variation**, che cerca di ricostruire e raffinare un sequenziamento di diversi genomi per costruire un genoma di riferimento per una popolazione, nel dettaglio umana, (in formato VCF)
- **Trans-Omics for Precision Medicine**, il progetto per la medicina traslazionale
- **The Computational Pangenome Consortium**, che mira a studiare nuovi strumenti software che possano trattare il grafo del pangenoma visto che la maggioranza del software attuale ancora funziona su sequenze e non su grafi

2.6 Sequenziamento del DNA

Il sequenziamento (che letteralmente significa "produrre la sequenza") solitamente si svolge concatenando diverse operazioni:

1. estrazione del DNA
2. si ha una "libreria preparatoria" dove si mette del materiale genetico su un materiale preparatorio
3. si ha un meccanismo di "copie" tramite PCR o simili
4. si mettono i sample genomici in una macchina di sequenziamento che produce in output i dati

Un genoma non può essere letto "nucleotide per nucleotide" e i biologi, con la tecnologia attuale producono le cosiddette **read** del DNA originali. Si hanno due tipi di read:

- **read**, dette anche **short read**, lunghe circa 100 basi. Illumina produce tendenzialmente 100 o al più 150 basi
- **long read**, lunghe circa 10000 basi (se non di più, anche 20000)

Per ottenere il sequenziamento si ha un processo in cui:

- si divide il genoma in due parti, "aprendo" il filamento di DNA per permetterne la lettura
- si ha la **generazione delle read** da copie multiple del genoma tramite un processo biologico svolto dai macchinari, che sfruttano processi chimici
- si ha poi **l'assemblaggio dei frammenti**, ovvero un processo computazionale dove tramite algoritmi si assemblano le varie read per ottenere il genoma di partenza, avendo che le read hanno pezzi in *overlap*

Il problema del sequenziamento risale alla fine degli anni settanta con Sander e Gilbert che avevano studiato un processo di replicazione dando le basi allo studio del sequenziamento.
Dopo il sequenziamento dell'uomo si è passati a sequenziare molti altri organismi.
Oggi il sequenziamento è reso semplice dalla tecnologia. Un esempio è la tecnologia MinION, così piccola sta stare in una mano, che produce *long read* (anche se comunque con diversi errori). MinION è una tecnologia di *Oxford Nanopore*. MinION è USB ed è fatta per biologi che devono sequenziare in situazioni d'emergenza (esempio banale un biologo in Africa in piena emergenza Ebola). L'elaborazione dati viene fatta da un server.
Il primo sequenziamento è costato 3 miliardi di dollari per diversi anni, ora si fa in meno di 40 ore a 5000 dollari. Di recente si è passati addirittura a poche ore per un costo di circa 1000 dollari. Tornando alla *legge di Moore* si ha che il costo è collassato rispetto alla legge e quindi la capacità delle tecnologie di sequenziamento è molto maggiore della capacità di processare i dati, per quanto visto ad inizio capitolo. Si hanno quindi tanti dati ma non si è in grado di elaborarli.
Si tratterà anche il **confronto di genomi** per studiare poi gli aspetti evoluzionistici, tramite **alberi evolutivi**, anche **alberi evolutivi tumorali**. Il **confronto tra sequenze** permette di studiare le evoluzioni, anche quelle tumorali, dove si hanno mutazioni radicali di DNA. Approfondiremo anche tali mutazioni e il loro effetto (basta il cambio di una base per portare, ad esempio, all'anemia falciforme). Studieremo quindi anche come fare gli **allineamenti**. Approfondiremo il discorso della **filogenesi** e della **filogenesi tumorale**.
Tutto questo, in questo ultimo anno, è stato applicato allo studio di **sars-cov-2**, avendo lo studio delle variazioni.
Verrà approfondito anche il discorso del **riarrangiamento**.

Capitolo 3

Grafi di assemblaggio

La prima tematica che affrontiamo è l'assemblaggio delle read tramite grafi. Per questo problema abbiamo quindi:

- **input**: collezioni di read (short read e/o long read)
- **output**: grafo di assemblaggio da cui estrarre un cammino o un'unica sequenza

Si hanno principalmente due tipi di grafo:

- **grafo di De Bruijn (*DBG*)** (*si legge "grafo di de broin"*), che si prestano più per *short read* (da 100 o 150 basi)
- **grafo di overlap**, più comodo in caso di *long read*

Si useranno per questi scopi varie nozioni, tra cui:

- relazione di prefisso/suffisso tra k-mers
- relazione di prefisso/suffisso tra read
- Longest Common Prefix tra sequenze
- estrazione di cammino di Eulero dal grafo
- estrazione di cammino Hamiltoniano dal grafo
- Maximal Exact Matches (*SMEMs*)
- Burrows Wheeler Transform (*BWT*)
- indici succinti (come FM-Index)

- suffix tree e suffix array
- bloom filters, nati in ambito fisico e usati ora in ambito BigData
- min-hash e min-sketch, usati anche nelle reti neurali e nel Deep Learning quando si ha a che fare con grandi moli di dati

Studiare i grafi di assemblaggio può essere utile anche in ottica di applicare procedimenti simili ad altri problemi posti dai biologi.

3.1 Grafi in bioinformatica

In bioinformatica infatti uno strumento molto usato, anche oltre il sequenziamento, è quello dei **grafi**.
In letteratura la nozione di grafo compare nel 1735 con il **grafo di Eulero**, con Eulero che, si dice, fosse ossessionato dal problema dei **ponti di Königsberg**, volendo trovare il ciclo che attraversasse ogni ponte solo una volta. Ogni isola di Königsberg diventava un nodo e ogni ponte tra isole un arco tra nodi. Da qui la definizione del problema.

Definizione 1. *Il **problema del ciclo Euleriano** consiste nel trovare un ciclo in un grafo tale che visiti ogni arco una e una sola volta prima di tornare al punto di partenza. Si può passare dallo stesso nodo più volte.*
*Questo problema si dimostra risolvibile in **tempo lineare** sull'input $G = (V, E)$.*

Vediamo poi il "problema duale", quello in cui si vuole fare un cilo che non visiti due volte uno stesso nodo.

Definizione 2. *Il **problema del ciclo Hamiltoniano** consiste nel trovare un ciclo in un grafo tale che visiti ogni vertice una e una sola volta prima di tornare al punto di partenza.*
*Questo problema si dimostra essere **NP-complete**.*

La differenza di complessità di questi due problemi sarà qualcosa che bisognerà considerare parlando dello studio dei grafi in bioinformatica. Anche solo il problema dell'assemblaggio si vedrà è riducibile alla visita di un grafo (quindi non potremo formularlo come un problema di ciclo Hamiltoniano, la cui soluzione potrebbe richiedere anni).
La comparsa dei grafi nel mondo chimico è intorno a metà del 1800 con Cayley che li usò per rappresentare strutture chimiche, nel dettaglio usò **alberi** (che ricordiamo esserei grafi connessi aciclici) per contare gli isomeri strutturali.

In biologia l'uso dei grafi è stato introdotto a metà 1900 con l'esperimento di Benzer, che capì l'importanza dei grafi mentre cercava di distinguere quando determinati virus attaccano determinati batteri. Benzer è riuscito a mostrare che il DNA di questi virus era *lineare* mentre prima si congetturava che il DNA avesse delle biforcazioni. Per capire che non avesse delle biforcazioni ha sfruttato la capacità di alcuni geni dei virus di aggredire batteri, rappresentando la cosa coi **grafi ad intervallo**.

Definizione 3. *Nella teoria dei grafi, un* ***grafo d'intervallo*** *è il grafo d'intersezione di un multiinsieme di intervalli sulla linea reale. Ha un solo vertice per ciascun intervallo dell'insieme, e uno spigolo tra ogni coppia di vertici corrispondenti agli intervalli che intersecano.*
In poche parole associo una lettera ad ogni intervallo e collego nel grafo i vertici corrispondenti alla lettera qualora i due intervalli abbiano sovrapposizioni.

Il punto di svolta si ha però nel 1977 col sequenziamento e i due metodi di Sanger (che è tutti gli effetti il primo metodo di sequenziamento) e Gilbert, entrambi chimici. Entrambi i metodi generano frammenti etichettati di lunghezza variabile che vengono "letti" tramite elettroforesi.
Non approfondiamo nel dettaglio i metodi, essendo prettamente chimici e biologici.

3.1.1 Superstringhe e grafo di overlap

Siamo in ottica **short read sequencing** del **fragments assembly**.
L'assemblaggio dei frammenti del DNA è invece un problema prettamente computazionale, avendo l'assemblaggio dei singoli frammenti, ovvero delle **read** prodotte dal sequenziamento, anche in più copie, in un'unica sequenza genomica, detta **superstringa**. *Fino alla fine degli anni '90 l'assemblaggio di frammenti del genoma umano era visto come un problema intrattabile.*

Definizione 4. *Definiamo* ***stringa*** *come la concatenazione di simboli di un alfabeto* Σ.
In bioinformatica spesso si ha $\Sigma = \{a, c, g, t\}$

Definizione 5. *Definiamo il* ***shortest superstring problem (SSP)*** *come la ricerca, dato un insieme di stringhe, di trovare la più corta superstringa che le contiene tutte. So hanno quindi:*

- ***input****: una collezione* $s_1, s_2, \ldots, s_n$ *di stringhe che possono anche essere lunghe uguali o a lunghezza variabile*

- ***output**: una stringa s che contiene tutte le stringhe $s_1, s_2, \ldots, s_n$ dell'input come sottostringhe tale che $|s|$, ovvero la lunghezza della stringa s, sia **minima***

*Questo problema è **NP-complete** e assume che non ci siano errori di sequenziamento nella produzione delle stringhe $s_1, s_2, \ldots, s_n$.*
***La shortest superstring potrebbe non essere unica**.*

Esempio 4. *Vediamo un esempio di shortest superstring. Si assume per semplicità alfabeto binario $\Sigma = \{0, 1\}$.*
Si ha la collezione di stringhe binarie in input (che nel dettaglio sono tutte le possibili combinazioni di 3 simboli binari):

$$C_I = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

Si può verificare che la shortest superstring è:

$$s = 0001110100$$

Con la shortest superstring ho letteralmente assemblato le stringhe in input.
Le read determinano la **coverage (*copertura*)** del DNA. Per valutare il coverage vado a vedere ogni base da quante read è coperta. Con Illumina ho un coverage di almeno 50x, quindi ogni posizione è coperta da almeno 50 read (lunghe ciascuna ∼150 basi). Per poter ricostruire la sequenza di DNA originale serve una certa quantità di coverage. Una coverage bassa potrebbe impedire la ricostruzione. Illumina va dal 50x minimo anche a 80x. MinION, della Oxford Nanopore, produce long read anche di 20000 basi ma con basso coverage, anche 3x, ma avendo read lunghe si riesce comunque ad assemblare. Quindi se ho long read mi basta un basso coverage mentre se ho short read mi serve un elevato coverage, avendo un insieme di read molto "fitto" e con poca "sparsità", in quanto si avrebbero gap, con zone non coperte. Il coverage è comunque dato "per media" e quindi poter comunque avere buchi.
Il punto chiave che mi permette di ricostruire il DNA è la sovrapposizione tra le varie read. Il DNA inoltre ha ripetizioni e questo costituisce, purtroppo, un limite all'assemblaggio e in merito studieremo il **fragment assembly problem**, che serve anche in altri contesti, oltre a quello dell'assemblaggio del DNA. Fin'ora abbiamo anche trascurato anche un altro problema, gli **errori di sequenziamento**, dati dal fatto che il processo di sequenziare non è *ottimo*, ovvero privo di errori, dove con errore si intende che nel DNA si ha una certa base e nella read prodotta dal sequenziamento se ne ha un'altra. In fase di assemblaggio questo tipo di errore comporta che non si riesce a

sovrapporre bene le read, non potendo vedere più alcuni **overlap** tra coppie read. Si ha quindi **perdita di informazione dell'overlap** e diventa più complicato assemblare il DNA, non impossibile ma più complicato.

Esempio 5. *Si hanno un pezzo di DNA e tre read che sono sovrapponibili:*

	1	2	3	4	5	6	7	8
$DNA =$	a	c	c	g	t	a	c	g
$R_1 =$	a	c	c	g	t			
$R_2 =$		c	c	g	t	a		
$R_3 =$				g	t	a	c	g

Possiamo quindi assemblare il pezzo di DNA.
Ma se ipotizziamo di avere un errore di sequenziamento con la terza base della seconda read:

	1	2	3	4	5	6	7	8
$DNA =$	a	c	c	g	t	a	c	g
$R_1 =$	a	c	c	g	t			
$R_2 =$		c	c	**c**	t	a		
$R_3 =$				g	t	a	c	g

Diventa più difficile assemblare.

Il tasso di errore nei macchinari Illumina è dello 0.01%, avendo circa due errori per read lunga 150. Per MinION si ha un tasso d'errore anche di circa il 10%, quindi ogni 50 basi ho una serie d'errore. Di recente, in ambito long read, si stanno progettando i **PacBio HiFi** (con HiFi che qui sta per "high quality framents") che producono long read con tasso d'errore allo 0.1%, facendo ben sperare per il futuro.
Tra i primi informatici che hanno fatto sequenziamento abbiamo Eugene Myers che era un esperto di algoritmi su stringhe e di pattern matching (parte attiva nella creazione dei suffix array), nonché responsabile della creazione dell'algoritmo di assemblaggio (famoso anche per BLAST). Eugene Myers era un esperto del problema della shortest superstring. A partire dalla tecnica di costruzione della shortest superstring ha sviluppato l'algoritmo di assemblaggio. Vediamo quindi, in primis, come costruire la shortest superstring. Per farlo bisogna in primis capire come confrontare le varie stringhe in input e come "foldarle". Per farlo faccio l'overlap che però a questo punto necessita di una definizione formale.

Definizione 6. *Definiamo **overlap** tra una coppia di stringhe s_i e s_j in input come il più lungo prefisso di s_j che ha un match perfetto (coincide) con un suffisso di s_i. Posso anche dire che è il più lungo suffisso di s_i che ha un match perfetto con un prefisso di s_j, ribaltare la definizione non cambia. L'overlap tra le due stringhe si indica con:*

$$ov(s_i, s_j)$$

Ricordiamo che una stringa la posso scrivere in modo scomposto in due modi:

- $s_i = s'_i x$, *con x suffisso*
- $s_j = x s'_j$, *con x prefisso*

*Tendenzialmente si prende l'**overlap più lungo**.*

Esempio 6. *Siano:*

$$s_i = accgtgtgt$$

$$s_j = gtgtgtccaa$$

Allora si ha che:

$$ov(s_i, s_j) = gtgtgt$$

con l'overlap lungo 6.

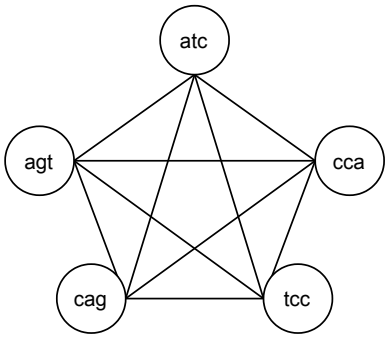
Proseguiamo quindi con il calcolo della shortest superstring dopo aver calcolato l'overlap di tutte le stringhe in input.
Creo un grafo con un nodo per ogni sequenza, etichettato con la sequenza stessa. Tracciamo quindi un arco tra due nodi sse i due nodi sono in overlap, associando all'arco la lunghezza dell'overlap.
Una tecnica per fare il grafo consiste in:

- collegare a priori di tutti i nodi ottenendo un grafo completo non orientato
- per ogni coppia di stringhe s_i e s_j metto l'arco pesato con l'overlap massimo $ov(s_i, s_j)$, dando anche direzione all'arco. Eventualmente posso anche dare doppio peso all'arco in base alla direzione. Si è ottenuto il **grafo di overlap**, che quindi è un grafo orientato (se in entrambi i versi non ho overlap lo lascio per praticità senza orientamento con peso 0)

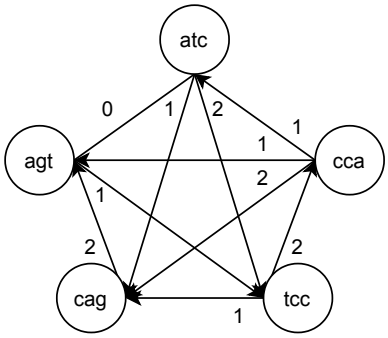
Esempio 7. *Sia la collezione di stringhe in input:*

$$C_i = \{atc, cca, cag, tcc, agt\}$$

e costruisco il grafo completo come detto sopra:



Aggiungo quindi i pesi relativi agli overlap dando l'eventuale orientamento e ottengo il grafo di overlap:



Per calcolare la shortest superstring dovremo calcolare un certo cammino sul grafo di overlap. Sicuramente un cammino che visita tutti i nodi mi porta ad avere una superstringa. Vediamo quindi una prima idea intuitiva:

Esempio 8. *Riprendendo il grafo di overlap dell'esempio precedente faccio:*

- *parto dal nodo atc e lo aggiungo alla superstringa, che per ora è* $s = atc$
- *seguo l'arco di peso 2 e arrivo in tcc*
- *aggiungo c (ovvero la parte non in overlap) alla superstringa, che per ora è* $s = atcc$
- *seguo l'arco di peso 2 e arrivo in cca*

- *aggiungo a (ovvero la parte non in overlap) alla superstringa, che per ora è $s = atcca$*
- *seguo l'arco di peso 2 e arrivo in cag*
- *aggiungo g (ovvero la parte non in overlap) alla superstringa, che per ora è $s = atccag$*
- *seguo l'arco di peso 2 e arrivo in agt*
- *aggiungo t (ovvero la parte non in overlap) alla superstringa, che per ora è $s = atccagt$*
- *mi fermo avendo visitato tutti i nodi*

Alla fine ho:

$$s = atccagt$$

che so essere una superstringa.

Si vede che il cammino, a conferma, tocca ogni vertice una e una sola volta, avendo un cammino Hamiltoniano ma, avendo i pesi, abbiamo a che fare con un **Traveling Salesman Problem (*TSP*)**. Dobbiamo però dimostrare che la superstringa ottenuta è anche la più breve.
Diamo però una piccola definizione formale del grafo di overlap.

Definizione 7. *Definiamo il **grafo di overlap** $G_{ov} = (V, E)$ tale che, data una collezione di stringhe $s_1, \ldots, s_n$:*

- *$V = \{s_1, \ldots, s_n\}$*
- *E è definito in modo che ogni arco $(s_i, s_j) \in E$ è un arco orientato da s_i a s_j di peso $|ov(s_i, s_j)|$ (quindi pesato con la lunghezza dell'overlap)*

Si dimostra poi che il **cammino Hamiltoniano di massimo costo** "produce" una shortest superstring. Facciamo una dimostrazione non formale. Innanzitutto per "produce" si intende che, dato il cammino prodotto da Hamilton di massimo costo, con i vertici etichettati dalle stringhe $s_{i,1}, s_{i,2}, \ldots, s_{i,n}$, la superstringa si ottiene sapendo che una stringa $s_{i,j+1}$ che è ha un prefisso in overlap con al precedente stringa $s_{i,j}$ la si può scrivere come:

$$s_{i,j+1} = ov(s_{i,j}, s_{i,j+1}) \cdot x_{i,j+1}$$

Possiamo anche dire che:

$$r(s_{i,j}, s_{i,j+1}) = x_{i,j+1}$$

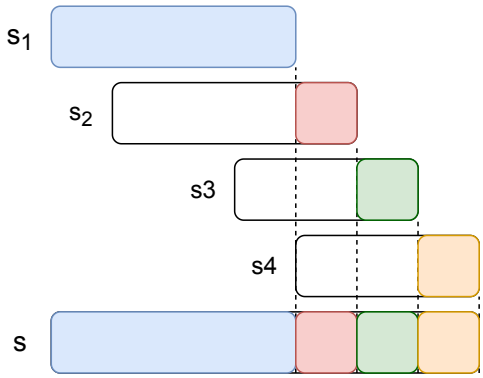


Figura 3.1: Esempio di formazione di una shortest superstring a partire da una collezione di stringhe sfruttando i "resti" degli overlap

indicando con $x_{i,j+1}$ la parte della stringa fuori dall'overlap, il "resto" possiamo dire. A questo punto so che la superstringa parte con $s_{i,1}$ e prosegue concatenando i vari $x_{i,j+1}$ (come si può vedere in figura 3.1):

$$s = s_{i,1} \cdot x_{i,2} \cdot x_{i,3} \cdot \ldots \cdot x_{i,n}$$

Bisogna dimostrare che il cammino Hamiltoniano di massimo peso coincide con la shortest superstring. Bisogna dimostrare che:

- un cammino Hamiltoniano di massimo peso calcolato come sopra è una superstringa, e questo di dimostra perché tocca ogni vertice e quindi ogni stringa che di conseguenza viene inclusa
- la superstringa appena calcolata è la più breve e per dimostrarlo si ha l'intuizione che se massimizzo l'overlap "globale" minimizzo la lunghezza della superstringa

Il calcolo della shortest superstring può quindi risolvere l'assemblaggio di stringhe anche se **si ricorda che il problema del cammino Hamiltoniano è NP-complete**.
La miriade di read però, quando ci lavorò per primo Eugene Myers, rendeva davvero difficile il calcolo (problema NP-complete e hardware storicamente poco potente). Servirono quindi anni per il primo calcolo, circa una quindicina, usando appunto il metodo della superstringa.
Si ha però un'euristica per calcolare la superstringa, usando un **algoritmo 2-approssimante** usando la **tecnica greedy**. In base a questa tecnica si sceglie sempre l'arco che pesa di più nel senso che ordino in ordine di peso tutti gli archi e faccio gli overlap tra le stringhe collegate. Dopo avere selezionato l'arco si prendono i due estremi e se ne fa la superstringa. Si continua

quindi cercando sempre gli archi che pesano di più creando poi la superstringa. **Non si vede nel dettaglio il funzionamento** e ovviamente non si ha la soluzione ottima e in realtà si congettura sia 2-approssimante ma non si ha una dimostrazione in merito, è un problema aperto da trent'anni e solo a livello sperimentale si è ipotizzata la 2-approssimazione.
Si ricorda che con il cammino Hamiltoniano ottimo si ottiene comunque una soluzione che potrebbe non essere unica.

3.1.2 Grafi di De Bruijn e k-mers

Siamo sempre in ottica **short read sequencing** del **fragments assembly**. Il metodo della superstringa è stato quindi usato per l'assemblaggio del primo sequenziamento (quello con il metodo Sanger) ma l'appoggio ad un problema NP-complete (il *ciclo Hamiltoniano*) rendeva il tutto troppo dispendioso. Il primo *assemblatore*, quello di Celera, usava però questo metodo più lento per il *fragment assembly*.
Vediamo ora una soluzione diversa, basata sui **grafi di De Bruijn** che invece come problema sottostante ha il *ciclo Euleriano* che sappiamo avere soluzione lineare.
Vediamo in primis qualche definizione.

Definizione 8. *Definiamo* ***k-mer*** *come è una sottostringa di lunghezza k. I k-mers sono quindi tutte le sottostringhe distinte di lunghezza k, non estraggo più volte lo stesso k-mer. Si segnala però che troppe ripetizioni dello stesso k-mer, che vengono trascurate, possono rendere difficile l'assemblaggio. Quindi il caso ideale è che tutti i k-mer estratti da una stringa siano distinti ma si seleziona il k in modo che ci siano al più due o tre ripetizioni dello stesso k-mer nella sequenza.*

Definizione 9. *Data una stringa s definiamo* ***spettro di s di dimensione/ampiezza*** *l è il* ***multiinsieme****,* ***avendo quindi ripetizioni****, di tutte le occorrenze di sottostringhe di lunghezza l (gli l-mers) e si indica con:*

$$spectrum(s, l)$$

Spesso lo spettro poi si rappresenta con i vari l-mer in ordine lessicografico*.*

Esempio 9. *Prendiamo una stringa s:*

$$s = tatggtac$$

Fissiamo $k = 3$ e si ha lo spettro di dimensione 3:

$$spectrum(s, 3) = \{tat, atg, tgg, ggt, gta, tac\}$$

(che in questo caso, non avendo ripetizioni, è un insieme di 3-mers.)

Dati i frammenti/read (che sono short read lunghe circa 150 basi) si estraggono da essi i **k-mers**, con k usualmente pari a 32, 31 o 28 per avere il minor numero di ripetizioni (i numeri sono stati identificati sperimentalmente). Si segnala che questa "proprietà" di avere sequenze circa lunghe 32 che non si ripetono è probabilmente legata anche al fatto che il DNA, preso come sequenza di simboli, è difficile da comprimere con i tool standard (zip, uso della BWT etc...), creando non pochi problemi alle banche dati anche se ancora non si è scoperto bene ne perché ne come risolvere la cosa.
Il problema di assemblaggio diventa quindi ricostruire una stringa da un insieme di k-mer:

- **input**: un insieme di stringhe $s_1, s_2, \ldots, s_n$
- estraggo i k-mer da tutte le s_i in input, per un k fissato
- assemblo i k-mer usando i grafi di De Bruijn

Ma prima di introdurre i grafi di De Bruijn vediamo un esempio di cosa significhi assemblare k-mers, partendo dal caso **senza ripetizioni**, avendo quindi un **insieme di k-mers** e non uno **spettro** (che è un multiinsieme).

Esempio 10. *Si prenda in input una collezione di k-mers con $k = 3$ (quindi 3-mers):*

$$C = \{atg, agg, tgc, tcc, gtc, ggt, gca, cag\}$$

Non essendo coincidenti se facessi gli overlap tra ogni coppia avrei al più overlap di lunghezza 2. Ipotizzando di fare il grafo di overlap avrei il seguente cammino Hamiltoniano (uno dei possibili):

$$atg \to tgc \to gca \to cag \to agg \to ggt \to gtc \to tcc$$

che produrrebbe la superstringa:

$$s = atgcaggtcc$$

Vedendo che anche con i k-mer posso ragionare in ottica di superstringa.

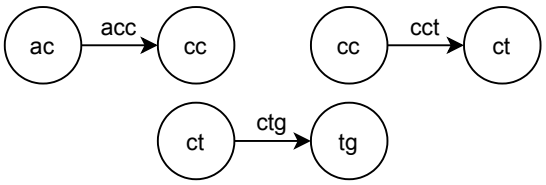
Ma con l'approccio che vogliamo ora non si passa per ogni vertice una e una sola volta ma per ogni arco una e una sola volta, usando il cammino Euleriano.
Dobbiamo fare si che quindi prendere ogni arco una e una sola volta corrisponde a prendere tutti i k-mers, avendo quindi che ogni arco deve essere

associato ad un k-mer. Costruiamo quindi un grafo che soddisfi queste condizioni (grafo che poi definiremo essere un **grafo di De Bruijn**). Si ha quindi un grafo dove gli archi sono i k-mer mentre i vertici all'estremo di un arco sono etichettati con il prefisso di lunghezza $k-1$ e il suffisso di lunghezza $K-1$ del k-mer (quindi i suffissi e i prefissi unici formano l'insieme dei vertici). L'arco è orientato dal prefisso al suffisso.
In altri termini i vertici agli estremi di un arco etichettato con il k-mer x altro non sono che i due unici (k-1)-mer estraibili da x.

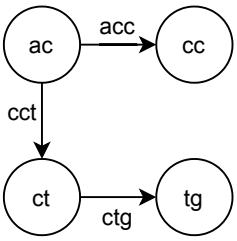
Esempio 11. *Prendendo una collezione di k-mer:*

$$C = \{acc, cct, cgt\}$$

so che, per il grafo $G = (V, E)$, ho $E = C$ mentre per V so che:



Quindi $V = \{ac, cc, ct, tg\}$.
Costruiamo quindi il grafo:



Quindi il cammino di Eulero (qui banale) e otteniamo la superstringa (e quindi l'assemblaggio) concatenando le etichette degli archi nell'ordine in cui vengono visitati, ragionando nello stesso modo in cui si faceva coi grafi di overlap, quindi partendo con la prima etichettata e proseguendo concatenando solo i resti dei vari overlap tra etichette degli archi:

$$s = acctg$$

Si ha quindi che:

Definizione 10. *Un grafo $G = (V, E)$* ***orientato*** *è un* ***grafo di De Bruijn*** *di ordine k se:*

- *i vertici sono un sottoinsieme di* Σ^{k-1}*, ovvero* $V \subseteq \Sigma^{k-1}$
- $\forall u, v \in V$ *si ha che* $(u, v) \in E$ *se esiste una parola* $w \in \Sigma^*$ *tale che* u *è (ha come etichetta) il prefisso di lunghezza* $k-1$ *di* w *e* v *è (ha come etichetta) il suffisso di lunghezza* $k-1$ *di* u

Quindi possiamo dire che, in modo astratto:

- *i vertici sono etichettati con un (k-1)-mer*
- *gli archi sono etichettati con un k-mer*

Tali grafi sono stati introdotti dal matematico De Bruijn nel 1946.
Questa è una definizione ***edge-centric*** *in quanto i* ***k-mer*** *vengono usati per gli archi. Si può avere anche una definizione* ***node-centric*** *dove estraggo i nodi per poi ottenere gli archi, collegando due vertici quando il prefisso nel primo nodo coincide con il suffisso del secondo ma avendo i nodi etichettati coi k-mer. Non necessariamente il k-mer dell'arco potrebbe non esistere. Normalmente per l'assemblaggio si usa la versione edge-centric. L'edge-centric implica il node-centric ma non viceversa. In entrambi i casi etichetto l'arco con il resto/estensione.* ***Capire meglio!***

Definizione 11. *Si ha che un vertice è* ***bilanciato*** *sse,* $\forall v \in V$*:*

$$in(v) = out(v)$$

con:

- $in(v)$ *numero di archi entranti in* v
- $out(v)$ *numero di archi uscenti in* v

Teorema 1 (Teorema di Eulero)**.** *Un grafo* ***connesso*** *è un* ***grafo Euleriano*** *sse suo ogni vertice è* ***bilanciato****.*

Dimostrazione. Vediamo le due direzioni della dimostrazione, avendo il *sse*.
Se si ha che per ogni arco entrante nel vertice v deve esistere almeno un arco uscente da v in quanto altrimenti mi "bloccherei" in un nodo, arrivando ad una contraddizione e non avendo quindi un cammino di Eulero. Quindi se non è bilanciato sicuramente non è Euleriano.
Se si ha un grafo bilanciato allora esiste (facciamo il caso "semplice") un ciclo Euleriano (e di conseguenza il caso "difficile" del cammino Euleriano). Infatti possiamo dare l'algoritmo che trova il ciclo Euleriano. Si ha quindi una **dimostrazione costruttiva**. Si ha quindi che:

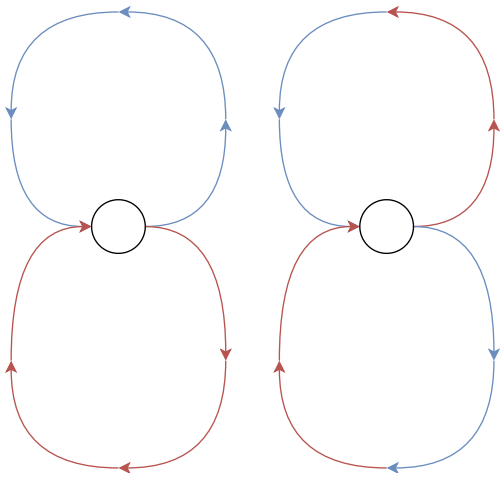


Figura 3.2: Idea del processo di apertura dei cicli dove si hanno due cicli, a sinistra in rosso e blu. Dopo l'apertura il cammino passa dal ciclo sopra a quello sotto, seguendo i colori nell'immagine a destra.

1. si parte da un vertice arbitrario v
2. si connette v ad un altro vertice usando archi, si riparte da tale arco e si fa la stessa operazione. Si ripete l'operazione fino a che non si è formato un ciclo
3. alla fine o uso tutti gli archi o altrimenti chiudo un ciclo lasciando vertici non usati. Nel secondo caso ripeto lo step 1) cercando di usare altri archi non usati. Facendo la **decomposizione del grafo in cicli**. In caso mi fermo quando ho usato tutti gli archi ma mi trovo con tanti cicli. Per derivare un unico ciclo da due cicli sfrutto il vertice di congiunzione in modo che se entro partendo da un ciclo in quel nodo esco con l'arco che mi porta nell'altro ciclo, facendo la cosiddetta **apertura dei cicli** (vedere figura 3.2). Riassumendo si combinano due cicli in uno unico e si itera questo passaggio fino alla creazione di un singolo ciclo. *Per fare questo uso l'ipotesi che G è bilanciato*

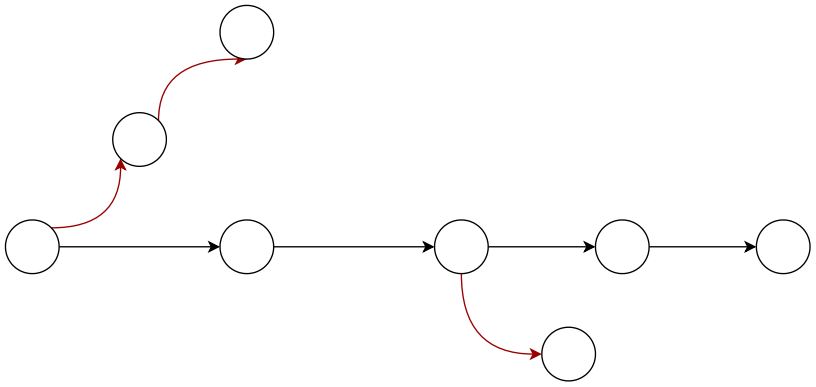
Questo algoritmo è lineare nella dimensione di un grafo, ovvero $O(|V| + |E|)$. □

Quindi per costruire il grafo di De Bruijn:

- **input:** una collezione F di frammenti
- si genera da F l'insieme dei k-mer (non lo spettro)

- per ogni k-mer dell'insieme dei k-mer estraggo il prefisso e il suffisso di lunghezza $k-1$
- si verifica l'esistenza del cammino di Eulero. Quindi:
  - se esiste si prosegue
  - se non è bilanciato si deve passare alla cosiddetta **pulitura del grafo**, eliminando gli archi detti **tip** (comportando la rimozione anche del nodo "destinatario" del grafo). Potrei avere perdita di k-mer. Le tip solamente sono singoli archi ma porrebbero anche essere cammini.
    Si usa anche il concetto di **bubble**, come effetto degli errori di sequenziamento. Una bubble è una situazione in cui si viola il teorema di Eulero ma che si elimina essendo un errore di sequenziamento, **risolvendo la bolla**. Per capire quale sia l'errore di sequenziamento sfrutto il fatto che ho più read che coprono la stessa porzione, scegliendo la base che è più frequente. **Il k condiziona le bolle oltre che alla connettività del grafo (*se è troppo grande rischio di non avere un grafo connesso*)**.
    Spesso si ha anche lo **scaffolding** per ottenere la connettività
- trovo il vertice "*head*" (l'unico che non ha vertici entranti) e il vertice "*tail*" (l'unico che non ha vertici uscenti), che saranno inizio e fine del cammino di Eulero
- si estrae il cammino di Eulero

Esempio 12. *Vediamo un esempio di tip, con le tip in rosso:*



quindi gli archi in rosso e i nodi a cui puntano possono essere rimossi. Vediamo un esempio di bubble. Ipotizzo di avere:

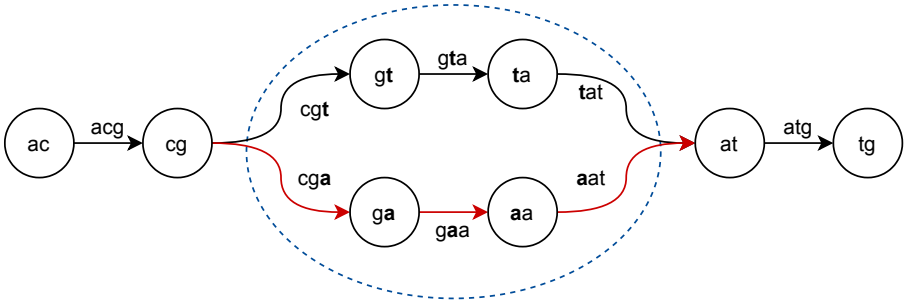
$$F = \{acgtatg, acgaatg\}$$

Notiamo quindi che:

$$acg\mathbf{t}atg$$

$$acg\mathbf{a}atg$$

Ma sappiamo che, tra le varie read, la t occorre molto più frequentemente della a in quella posizione. Si assuma di avere $k = 3$. Ho quindi, con la bolla (notando come la sua presenza impedisce di avere un cammino Euleriano) segnalata dall'ovale:



(dove si nota come la k influisca sulla natura della bolla).
Avendo però che la t occorre più frequentemente in quelle posizioni posso rimuovere il cammino con gli archi rossi e i due nodi centrali.

Esempio 13. *Sia:*

$$F = \{atgcc, caatg, gcgtt\}$$

e sia $k = 3$.
Avendo lo spettro:

$$\{atg, tgc, gcc, caa, aat, atg, gcg, cgt, gtt\}$$

ho l'insieme dei k-mer, ovvero l'insieme degli archi del grafo, è:

$$E = \{atg, tgc, gcc, caa, aat, gcg, cgt, gtt\}$$

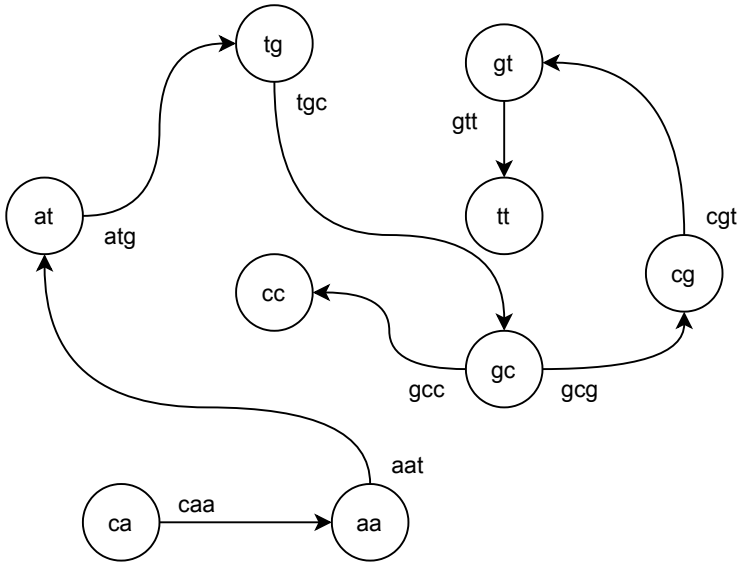
Genero quindi i prefissi e i suffissi di lunghezza $k - 1$ di ogni k-mer:

$$\{at, tg, tg, gc, gc, cc, ca, aa, aa, at, at, tg, gc, cg, cg, gt, gt, tt\}$$

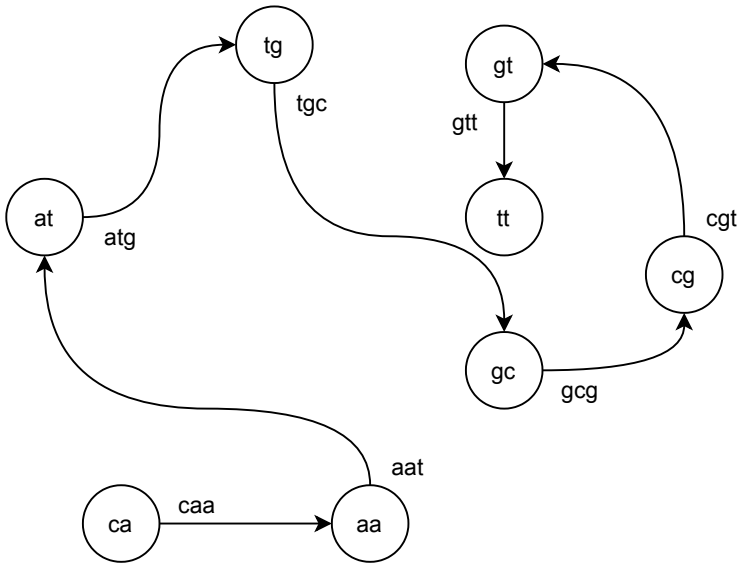
e quindi l'insieme dei vertici è:

$$V = \{at, tg, gc, cc, ca, aa, cg, gt, tt\}$$

Avendo quindi il grafo:



Le etichette degli sono per pura comprensione.
Bisogna quindi capire se esiste un cammino di Eulero. Purtroppo si hanno i vertici con tt e con cc che creano problemi. Inoltre il vertice gc non è bilanciato. Quindi il grafo non ammette cammini di Eulero.
Elimino quindi un arco "problematico", quello tra gc e cc, ottenendo:



Dove ho un cammino Euleriano, partendo dal vertice ca e arrivando al vertice tt. Si compone così la superstringa, ovvero l'assemblaggio:

$$s = caatgcgtt$$

Definizione 12. *Si indica con il termine **de novo** quando si fa una cosa "da zero".*
*Ad esempio si ha il **de novo genome assembly**.*

Definizione 13. *Si indica con il termine **reference bases** quando non si fa una cosa "da zero" ma si parte da una reference già preesistente.*

3.2 Note extra sui Grafi

I grafi di assemblaggio vengono usati anche nella **metagenomica**, ovvero lo studio di campioni di DNA di diversi organismi assieme. Questa cosa può essere utile quando il campione contiene vari organismi, si pensi ad esempio allo studio di un campione d'acqua o allo studio di un tampone faringeo.

Definizione 14. *Definiamo il paradigma **overlap-layout-consesus (OLC)**, proposto da Eugene Myers, come il paradigma per il quale si procede alla costruzione del grafo a partire da un insieme di read. Tramite le read si calcola poi il grafo di overlap per poi inferire il cammino, ovvero la **sequenza di consenso**. Il costo computazionale di calcolo non è indifferente.*
Rispetto ai grafi di De Bruijn è che possono immediatamente disambiguare brevi ripetizioni che i grafici di de Bruijn potrebbero risolvere solo nelle fasi successive.

Definizione 15. *Definiamo formalmente il **grafo di overlap**.*
Dato un insieme di read R un grafo di overlap è un grafo orientato:

$$G = (R, E)$$

dove i vertici sono le read stesse e si ha un arco tra r_i e r_j sse un suffisso di r_i è un prefisso di r_j, essendo in overlap. Il resto dell'overlap è detto e_{ij} ed etichetta l'arco (e si nota che il prefisso di r_i prima dell'overlap può essere usato come etichetta).

Si ha che:

- un cammino nel grafo di overlap rappresenta una stringa che è ottenuta assemblando le read. Tale stringa è ottenuta tramite $r_1 e_{i1} e_{i2} \cdots e_{ik}$

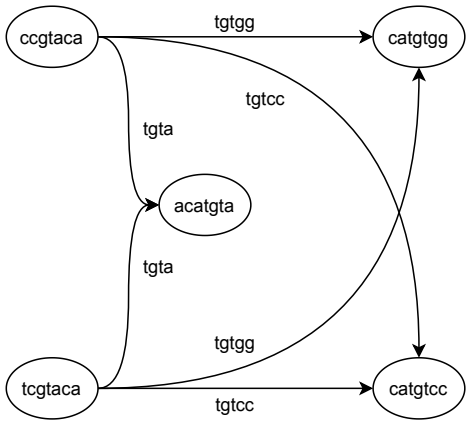


Figura 3.3: Esempio di grafo di overlap

- se un arco (r_i, r_j) è un cammino con solo due vertici è la stringa $s = r_i e_{ij}$
- un arco (r_i, r_j) è detto **riducibile** se esiste un percorso da r_i a r_j, con anche altri vertici, che rappresenta la stessa stringa di (r_i, r_j). Tali archi possono essere eliminati da un grafo di overlap ottenendo uno **string graph**, che offre una struttura più semplice e utile per la ricostruzione del genoma

Il grafo di overlap può essere usato per trovare le **varianti**.

Definizione 16. *Dato un insieme R di read si ha che un* ***edge-centric De Bruijn Graph****:*

$$G = (V, E)$$

di ordine k è un grafo dove i vertici sono i vari (k-1)-mer e si ha un arco tra u e v sse esiste un k-mer $w \in R$ tale che si il prefisso di lunghezza k-1 di w è uguale a u e il suffisso lungo k-1 di w è v. L'arco tra u e v è etichettato con l'ultimo carattere di v.

Se teniamo conto che i k-mer sono ripetuti possiamo usare un **multigrafo**.

Definizione 17. *Un* ***multigrafo*** *è un grafo in cui gli archi sono specificati da un* ***multinsieme****, il che significa che lo stesso arco può essere ripetuto più volte.*

Definizione 18. *Definiamo* ***isola*** *in un grafo di De Bruijn sono un insieme di nodi isolati. Sono generate da k-mer non frequenti e quindi probabilmente sono errori di sequenziamento.*

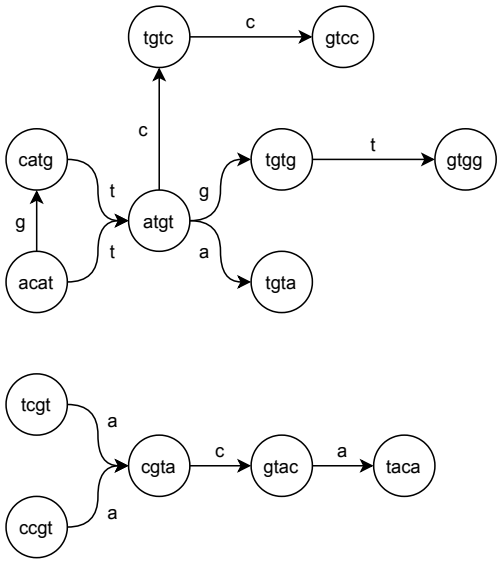


Figura 3.4: Esempio di grafo di De Bruijn con $k = 5$ edge-centric (anche se rappresentato non con l'intero 5-mer ma solo con il resto).

Definizione 19. *Definiamo **tip** in un grafo di De Bruijn come un nodo che si separa singolarmente con un arco dagli altri.*

I k-mer relativi a isole e tip vanno eliminati.
Se si ha che un k-mer è troppo ripetuto si è davanti probabilmente ad una **ripetizione**. L'uomo ha molti geni replicati, avendo magari varie versioni di un certo gene. Posso avere ripetizioni corte o molto lunghe.
Oltre alle bolle legate ad errori ho anche bolle legate magari al fatto che l'uomo è **diploide**, avendo **due copie** di ciascun cromosoma, una ereditata dalla madre e una dal padre, che possono avere comunque il 0.1% di differenza. Le read mi arrivano in eguale proporzione dai due cromosomi, sto assemblando quindi due fonti in una sola sequenza. Nelle banche solitamente si ha comunque solo l'**allele di maggioranza** (l'**allele**, dal punto di vista informatico, è la base variata a livello di cromosoma).
Vanno rimosse anche le **bubble** e i cammini vengono assemblate nelle stringhe dette **contigs**.
Un assemblatore che usa i grafi di De Bruijn è davvero complesso.
Nell'ultimo periodo in realtà si lavora con una reference e con un file VCF contenente le varianti. Le read da cui si parte solitamente sono a lunghezza fissata.
In fase di metagenomica si parte da read "colorate" provenienti da diversi organismi. Si costruisce un **grafo di De Bruijn misto** da cui si riesce ad

estrarre le sequenze delle singole specie. Per farlo si sfruttano varie informazioni, ad esempio la concertazione di copie di una certa read per le varie specie. Come informazione si usa anche il fatto che alcune specie hanno k-mer **unici**.
Diamo uno sguardo anche ai tempi computazionali, con n basi, g lunghezza del genoma e a lunghezza overlap (con ST indichiamo con *suffix array*, con DP *programmazione dinamica* e con NE *senza errori*):

	De brujin	***Overlap***
tempo	$O(n)$	$O(n+a)$ ST, $O(n^2)$ DP
spazio	$O(n)$, $O(\min\{N, g\})$ NE,	$O(n+a)$

Normalmente il sequenziamento viene fatto tramite **elettroforesi** ma picchi irregolari portano sicuramente ad errori di sequenziamento.
Tornando al discorso **ripetizioni** si ha che essere conferiscono "confusione" all'assemblaggio.
Risentire ultimi minuti lezione 16 marzo
Potrei avere, con Illumina, buchi di read non coperti che vengono colmati in modo algoritmico, tramite **contigs** che si collegano in **supercontigs**. Questa operazione è detta **scaffolding**.

3.2.1 Sequenziamento per Ibridazione

Sezione molto oscura.
Per leggere porzioni di DNA si usa il **sequenziamento per ibridazione *SBH***, ad esempio per leggere le porzioni atta a capire il test di paternità. Viene fatta tramite **affymetrix chip**, ovvero una piastra di materiale plastico dove viene depositato, dentro celle, il DNA da studiare. Il DNA sfrutta la complementarietà delle basi e queste celle contengono particolari k-mer così che le celle mi restituiscono i k-mer completi, ottenuti tramite il processo chimico dell'ibridazione che fa attaccare basi complementari.
Si ha quindi il **SBH problem**, che consiste di ricostruire una stringa dai suoi l-mer, con:

- **input**: un insieme S con tutti gli l-mer di una stringa s
- **output**: una stringa s tale che $Spectrum(s, l) = S$

Si usa il cammino Hamiltoniano per risolvere il problema oppure posso usare il cammino di Eulero usando i (k-1)-mer e il grafi di De Bruijn.

Capitolo 4

Allineamento di Sequenze

L'**allineamento** è la procedura principe usata per comparare sequenze biologiche.
Il confronto di sequenze, tramite sequenze di genomi, permette di creare **alberi di filogenesi**, alberi no-rooted con le specie attuali come foglie, che rappresentano l'evoluzione delle varie specie. La ricostruzione di tale albero avviene grazie al confronto, tramite allineamento, di sequenze.
Confrontare le sequenze consiste nello studiare il processo di evoluzione, risalendo al processo evolutivo tramite le varie mutazioni del DNA.
SI hanno due tipi di mutazioni:

- a livello nucleotidico, su singole basi, che sono **operazioni di edit**. Una singola base comporta la creazione di una diversa proteina e potrebbe portare a varie patologie. Le varie forme di una malattia genetica (tipo l'anemia) possono derivare da proteine diverse. A questo tema si affianca la medicina traslazionale

- a livello più ampio, cromosomico, che riguardano **riarrangiamenti**, ovvero scambi di pezzi di DNA o inversione di orientamento di alcune porzioni. Questo è legato magari a fenomeni ambientali e comporta il cambiamento di proteine e caratteristiche della specie, comportando un processo di adattamento e selezione naturale

Con l'allineamento si confrontano sequenze per vedere quanto sono simili/diversi. Si vuole o massimizzare la simiglianza o minimizzare le differenze. Per allineare si inseriscono i **gap**.

v	a	t		g	t	t	a	t	
w	a	t	c	g	t		a		c

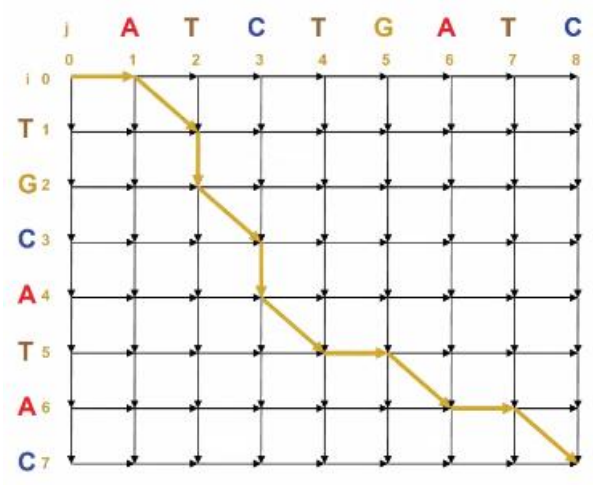


Figura 4.1: Esempio di matrice di programmazione dinamica per calcolare l'allineamento, dove si hanno 4 match, due inserzioni e due delezioni (si ragiona dal punto di vista di v).

Il biologo attribuisce alle mutazioni un punteggio, uno **score**. Tale score è stabilito attraverso uno studio probabilistico sulle mutazione avvenute nel corso dell'evoluzione.
Se allineo più di due sequenze parlo di **allineamento multiplo**. Si segnala che il 98% dei geni è pari tra due mammiferi e si ha più del 70% di similarità nelle sequenze proteiche.
L'allineamento dal punto di vista algoritmico è fatto tramite programmazione dinamica (mentre all'inizio erano fatte a mano dai biologi). Dal punto di vista bioinformatico l'allineamento è il processo base per praticamente qualsiasi software.

Definizione 20. *Definiamo l'allineamento di due sequenze A e B su Σ^* è una matrice $A^{2\times l}$ in cui la prima riga è associata ad A e la seconda a B. Ciascuna riga contiene i caratteri della sequenza associata intervallati da gap/spazi, rappresentati con il trattino.*
Non è permesso avere due gap nelle due righe per la stessa posizione. In altri termini non posso avere una colonna di gap.
Ne segue che al massimo, avendo $|A| = n$ e $|B| = m$ allora:

$$l \leq n + m$$

e la matrice $2 \times n + m$, ovvero quella massima, consiste nell'avere la prima sequenza allineata con soli gap (che saranno sotto) e la seconda allineata con soli gap (che saranno sopra).

Per definire il problema dell'allineamento ottimo ho bisogno di una:

$$\delta : (\Sigma \cup \{-\})^2 \to \mathbb{R}^+$$

Potrei avere anche versioni della delta con pesi negativi:

$$\delta : (\Sigma \cup \{-\})^2 \to \mathbb{R}^-$$

dove δ rappresenta la **funzione di score**, tramite **matrice di punteggio**, che ha i caratteri dell'alfabeto che etichettano righe e colonne. Ogni a_{ij} è lo score. Sulla diagonale avrò soli 0.
Dati in input quindi A, B e δ si ha che il costo della matrice di allineamento M è il minimo (o a seconda massimo) della somma del costo delle colonne:

$$c(M) = \sum_{1 \leq i \leq l} \delta(M_{1i,2i})$$

Dove $A_{1i,2i}$ è una scrittura $M[1,i], M[2,i]$.
In altri termini è la somma delle delta delle colonne (delta prima colonne più delta seconda, più delta terza etc...).
La **distanza di edit** è un caso particolare dell'allineamento.
Il problema **LCS** (che quindi può essere vista come caso particolare dell'allineamento), dove date due sequenze A e B si cerca la lunghezza della più lunga sottostringa comune, è calcolabile tramite la **matrice di allineamento**, dove l'unico elemento della matrice che costa 1 è quando si ha un match ($\delta(\sigma, \sigma) = 1$), avendo alla fine che che $c(M) = |lcs(A, B)|$.
Se passo all'**allineamento multiplo di sequenza** ho in input ho $A_1, A_2, \ldots, A_k$ con k sequenze.

Definizione 21. *Definiamo l'allineamento multiplo di $A_1, A_2, \ldots, A_k$ è una matrice M di k righe e l colonne con:*

$$l \leq k \cdot \max\{|A_i|\}$$

tale che la riga i-esima, $\forall 1 \leq i \leq k$, di M contiene la sequenza A_i intervallata da gap. Anche qui non si possono avere colonne di soli gap. È quindi un'estensione/generalizzazione di quanto detto per due sequenze.

Definizione 22. *Definiamo anche in questo caso lo scoring, avendo che è identico al caso di due colonne. Infatti rappresenta il corso di una operazione tra **coppie** di simboli.*

Definizione 23. *Definiamo il **costo di una matrice di allineamento multiplo**. Tale costo dipende da cosa si vuole evidenziare e si hanno varie*

definizioni alternativa.
*La definizione classica è quella della **sum of pairs**, ma anche le alternative, tipo **consesus**, si basa sul dire che il costo è comunque in qualche modo la somma dei costi delle colonne della matrice. Bisogna però capire come stabilire tali costi. Per farlo penso ad ogni cella di una colonne come nodo di un grafo e costruisco un grafo completo. Il **sum of pairs** è il costo di quel grafo completo. Ogni arco tra coppie σ_1 e σ_2 (con σ_i contenuto della cella i-esima della colonna) è di peso pari al $\delta(\sigma_1, \sigma_2)$. Confronto quindi ogni cella della colonna con ogni altra, calcolando il delta, e faccio la sommatoria:*

$$c(M) = \sum_{1 \leq t,s \leq k} \delta(\sigma_t, \sigma_s)$$

*Nel caso del **consesus** dovrei vedere solo le differenze rispetto ad una sequenza reference, anch'essa in input. Confronto tutte le altre sequenze contro questa sequenza reference A^*. Avrei quindi come grafo un albero con la reference come radice. Alla fine sommo i costi degli archi come per il sum of pairs. Tale allineamento è anche detto **star alignment**.*
*Questi due sono i due metodi principali ma ci possono anche essere altri "mapping" tra cui prendere alberi specifici basati sull'evoluzione delle specie. Questa tecnica è detta **tree alignment**, confrontando solo coppie di un albero specifico di evoluzione.*
A seconda dello scoring poi si procede massimizzando o minimizzando.

Bisogna quindi capire come calcolare M tale che il costo di M sia ottimo. Per farlo si applica la programmazione dinamica. Per due sequenze lunghe rispettivamente n e m il costo sarebbe $O(n \cdot m)$, $O(n^2)$ se lunghe uguali.
Per k sequenze si ha sempre la programmazione dinamica ma il costo è più elevato. Se le sequenze sono in numero fissato si ha $O(l^k)$ ma se non conosco il numero di sequenze in input il problema diventa **NP-hard** anche quando prendo uno score delta banale che soddisfa la disuguaglianza triangolare su alfabeto binario. Si hanno quindi diverse euristiche per risolvere il problema e si basano sempre sull'idea del **consesus**, con tecniche greedy. Un esempio di software per farlo è **MAFFT**.

Esempio 14. *Date:*

- $A_1 = accgtc$
- $A_2 = ccgcc$
- $A_3 = accttc$

Ho una possibile (sarebbero diverse) matrice di allineamento, fatta per massimizzare i match:

A_1	a	c	c	g	t	c	-
A_2	-	c	c	g	-	c	c
A_3	a	c	c	t	t	c	-

Per esempio per la prima colonne avrei:

- $\delta(a, -)$
- $\delta(a, a)$
- $\delta(-, a)$

coi tre valori che andrebbero sommati nel caso di sum of pairs.

Fondamentalmente allineare sequenze significa trovare un cammino in una griglia, un problema del tipo **Manhattan Tourist Problem (MTP)**. La griglia rappresenta diversi modi per attraversare la città da un punto A ad uno B dove si hanno pesi per ogni arco (i lati dei quadrati della gruglia hanno un peso associato) per specificare il numero di attrazioni. Lo scopo è arrivare alla fine massimizzando il numero di attrazioni viste.
Si cerca quindi il cammino più pesante in una griglia pesata. In input si ha la griglia, il punto di partenza e di arrivo. Una strategia greedy per la quale ad ogni istante scelgo l'arco più pesante non sempre porta alla soluzione ottima. Si ha quindi un algoritmico di programmazione dinamica con il criterio di memorizzazione.
L'algoritmo ricorsivo si preoccupa di capire come arrivare al nodo finale ragionando a partire dal nodo finale stesso, sapendo che posso fare solo certi movimenti sulla griglia. Derivo il costo ottimo di un nodo a partire dai costi ottimi dei nodi che portano a quel nodo (scegliendo il migliore). Con la pro-

Algorithm 1 Algoritmo MTP con ricorsione

if $n = 0 \vee m = 0$ **then**
  return $MT(n, m)$
$x \leftarrow MT(n-1, m) + w(u, v), \;\; u = (n-1, m), v = (n, m)$
$y \leftarrow MT(n, m-1) + w(z, v), \;\; u = (n, m-1), v = (n, m)$
return $\max\{x, y\}$

grammazione dinamica parto dalla sorgente e accumulo il punteggio per ogni percorso possibile, scegliendo poi il miglior percorso. Visito l'intera griglia e

calcolo ogni volta il modo migliore per arrivare ad un nodo, risolvendo ogni volta problemi di massimo. Alla fine vado a ritroso per capire che percorso fare mentre nell'ultimo nodo potrò già vedere il peso ottimo del percorso scelto.

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + w(u,v), & u = (i-1,j), v = (i,j) \\ s_{i,j-1} + w(z,v), & z = (i,j-1), v = (i,j) \end{cases}$$

Si a quindi, per una griglia $n \times m$, un costo pari a $O(n \cdot m)$.

Potrei aggiungere al problema l'uso delle diagonali, dovendo cambiare il sistema, aggiungendo il costo della diagonale:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + w(u,v), & u = (i-1,j), v = (i,j) \\ s_{i,j-1} + w(z,v), & z = (i,j-1), v = (i,j) \\ s_{i-1,j-1} + w(o,v), & o = (i-1,j-1), v = (i,j) \end{cases}$$

Trovare un cammino ottimo (uno di quelli più pesanti) della griglia corrisponde all'allineamento tra sequenze.
Dato il costo quadratico di confrontare due sequenze si passa a considerare anche metodi di confronto **alignment-free**, dove si decide che due sequenze sono molto simili senza fare l'allineamento.
Per allineare nel modo più veloce possibile si è passati poi dalla programmazione dinamica i metodi **bwt-based**.
L'allineamento può essere usato sui k-mer per eliminare a priori errori prima della costruzione del grafo di De Bruijn.

Definizione 24. *Definiamo il seguente **paradigma**. La somiglianza/similarità a livello di sequenza implica "stessa funzione".*
Due regioni genomiche, che si sa essere geni, uguali in due specie si ha che rappresentano lo stesso gene (e questo permette di studiare i topi in ambito medico e farmacologico).
Si rileva che la stessa funzione non implica per forza stessa sequenza e quindi stesso gene. Non vale quindi il viceversa.

Ricordiamo che il problema MTP consisteva nel trovare il cammino che massimizzava il cammino in una grigia. La risoluzione usa appunto la programmazione dinamica.
Interpretiamo il riempimento della griglia in termini di confronto di sequenze.
Un allineamento senza mismatch altro non è che il calcolo della Longest Common Subsequence LCS.

Per trovare l'LCS posso usare un grafo pesato che parte da un *source* e arriva ad un *sink*, trovando all'interno il cammino più pesante dal source al sink. Ogni cammino è tra il source e il sink. I nodi rappresentano i confronti tra due simboli delle due stringhe di partenza. Se ho un arco diagonale mi sto spostando su entrambe le sequenze, che in quel nodo hanno lo stesso simbolo. Con gli archi verticali e orizzontali mi sposto solo in una delle due sequenze, confrontando un simbolo con un gap (se l'arco è verticale ho un gap nella stringa che sta sulle colonne e viceversa se ho un arco orizzontale). Il percorso migliore è quello con più diagonali possibili visto che rappresentano un match. La diagonale ha quindi costo $\delta(\sigma, \sigma)$, mentre le altre due mosse hanno costo $\delta(-, \sigma)$ e $\delta(\sigma, -)$. Una matrice di allineamento è quindi un percorso nella griglia.
Ogni volta che ci si allontana dalla diagonale principale si sta inserendo un gap. Potrei quindi ragionare **allineando per banda**, avendo al più k gap e quindi costruendo solo una banda distante al più k dalla diagonale principale, avendo quindi un algoritmo più veloce per costruire l'allineamento.
Si nota quindi che LCS è un MTP e anche la distanza di edit è calcolabile con la griglia.
Per LCS, per due stringhe v e w, si ha infatti che:

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

Quindi nella griglia le diagonali costano 1 e gli altri costano 0.
Quando si fa l'allineamento si sta calcolando in realtà la **distanza** tra due sequenze, calcolando quanto sono diverse. La distanza di edit è una di queste distanze, che rappresenta il minimo numero di operazioni epr trasformare una stringa in un'altra. Un altra è la **distanza di Hamming**, che si riferisce a sequenze di distanza uguale, senza ammettere spazi, calcolando il numero di posizioni in cui si hanno mismatch. Non si hanno inserzioni e delezioni in Hamming, mentre si hanno entrambe in edit.
Le matrici di confronto più note, per il confronto proteico, sono le:

- **PAM (*point accepted mutations*)**
- **BLOSUM (*blcok substitution*)**

Allineamento Multiplo

Ricordiamo che l'allineamento multiplo permette di confrontare più specie, cercando di studiare i cambiamenti evolutivi. L'allineamento multiplo rivela

similarità che con l'allineamento normale non si riescono a comprendere. Per praticità consideriamo tre sequenze e quindi 3 dimensioni.
Confrontando 3 sequenze ho quindi bisogno di un cammino in uno spazio a tre dimensioni. Si usa il cosiddetto **3-D Manhattan cube** che comporta un **3-D edit graph**.
Pensando a come posso arrivare ad un punto ho quindi non più 3 modi, come nel caso a due dimensioni, ma ne ho 7:

$$s_{i,jk} = \max \begin{cases} s_{i-1,j-1,k-1} + \delta(v_j, w_k, u_k) \\ s_{i-1,j-1,k} + \delta(v_j, w_k, \_) \\ s_{i-1,j,k-1} + \delta(v_j, \_, u_k) \\ s_{i,j-1,k-1} + \delta(\_, w_k, u_k) \\ s_{i-1,j,k} + \delta(v_j, \_, \_) \\ s_{i,j-1,k} + \delta(\_, w_k, \_) \\ s_{i,j,k-1} + \delta(\_, \_, u_k) \end{cases}$$

Si arriva a:

$$O(n^3)$$

mentre per k dimensioni si avrebbe:

$$O(2^k n^k)$$

Per k fissato è quindi parametrico ma se non so quante k sequenze ho è un problema particolarmente complesso, esponenziale.
Ogni allineamento multiplo comporta, nei passaggi interni, allineamenti a coppie, per poi "chiudere" tramite la transitività.
Dall'allineamento multiplo induco all'allineamento a coppie ma non vale il viceversa, avendo che partendo da coppie di allineamento potrei avere tipologie diverse di allineamento a parità di sequenza non potendo quindi usare la transitività. Potrei quindi arrivare a inconsistenze.
Per allineare possiamo usare un approccio greedy. Si considerino 4 sequenze, si hanno quindi:

$$\binom{4}{2} = 6$$

possibilità di allineamento. Si usa il **progressive alignment** e non si usa quindi una strategia di programmazione dinamica.
Un altro metodo è quello già anticipato del **sum of pairs score (*SP-Score*)**.
L'allineamento multiplo può anche essere visualizzato come un grafo, tramite il **grafo del pangenoma**. In tale grafo si ha un nodo in corrispondenza di match perfetto e vari branch in presenza di non match. Posso tenere

traccia dei "collassamenti" in un solo nodo dei vari match e della sequenza di provenienza di ogni nodo. Valutando i percorsi studio gli allineamenti anche se potrei avere un percorso non corrispondente ad alcuna sequenza e questo è un side effect. Questa rappresentazione ha i suoi vantaggi, definendo il **partial order multiple sequence alignment** per cui ogni volta che arriva una nuova sequenza il confronto avviene direttamente con il grafo, arricchendolo, ottenendo un modo efficienze per fare allineamento multiplo in modo incrementale. Il creatore di questo metodo ha anche sviluppato un metodo simile per confrontare più grafi.

4.1 Algoritmi di Allineamento

4.1.1 Allineamento Globale di Needleman-Wunsch

Vediamo quindi l'allineamento globale di due sequenze. Si ricorda che:

- **input:** due sequenze, $A = a_1, \dots a_m$ e $B = b_1, \dots, b_n$, definite su algoritmo Σ, e uno schema di punteggio δ
- **output:** un allineamento M tra A e B che corrisponde all'ottimo (a seconda dell'impostazione del problema massimo/minimo) score D

Si ha che:

- il problema è di massimo se δ fornisce una misura di somiglianza tra simboli
- il problema è di minimo se δ fornisce una misura di diversità tra simboli

Dato $A_i = A[1, i]$, ovvero il prefisso di i caratteri di A, e dato $B_J = B[1, j]$, ovvero il prefisso dei primi j caratteri di B, ho che, $\forall i = 0, \dots, m$ e $\forall j = 0, \dots, n$:

$$D(i, j)$$

è lo **score** di allineamento globale tra A_i e B_j.
Si ha quindi che:

$$D(m, n)$$

è lo **score di allineamento globale** tra A e B.
Si ha quindi che:

$$D(i,j) = opt \begin{cases} D(i-1, j-1) + \delta(a_i, b_j) \\ D(i-1, j) + \delta(a_i, -) \\ D(i, j-1) + \delta(-, b_j) \end{cases}$$

avendo che, per un certo k, s e d (in base ai quali si capisce se è un problema di massimo o minimo):

- $\delta(a_i, b_j) = k$ se $a_i = b_j$
- $\delta(a_i, b_j) = s$ se $a_i \neq b_j$
- $\delta(a_i, -) = \delta(-, b_j) = d$

Come casi base si ha quindi:

- $D(0, 0) = 0$
- $D(i, 0) = i \cdot d = D(i - 1, 0) + d$
- $D(0, j) = j \cdot d = D(0, j - 1) + d$

L'algoritmo consiste quindi in:

1. riempire la matrice di score calcolando tutti i $D(i, j)$
2. individuare la cella ottima, ovvero $D(m, n)$
3. ricostruzione della matrice di allineamento M a partire da tale cella ottima

Bisogna quindi capire come riempire tale matrice D, dati k, s e d:

1. si specifica che D è una matrice $(m + 1) \times (n + 1)$, con righe indicizzate per $i \in [0, m]$ e colonne indicizzate per $j \in [0, n]$. La riga di incide $i, i > 0$, corrisponde ad a_i e la colonne di indice $j, j > 0$ al simbolo b_j
2. inizializzo $D(i, 0) = i \cdot d, \forall i \in [0, m]$ (quindi inizializzo la prima colonna) e $D(0, j) = j \cdot d, \forall j \in [0, n]$ (quindi inizializzo la prima riga)
3. scorro riga per riga (o colonna per colonna) e riempio al matrice secondo il calcolo sopra definito per $D(i, j)$. Tengo traccia della cella di partenza da cui ho ottenuto l'ottimo (se quella sopra, sotto o sulla diagonale). Se ho pari valori posso scegliere di dare priorità a considerare di aver fatto un movimento sula diagonale
4. identifico in $D(m, n)$ la cella ottima

In merito alla ricostruzione dell'allineamento si procede quindi a ritroso:

1. si inizializza una matrice M di allineamento vuota
2. parto dalla cella ottima di D e seguo un percorso a ritroso fino alla cella $(0, 0)$
3. ogni passaggio da una cella aggiungo una coppia di simboli a M. Si hanno quindi tre casi:
    - il punteggio migliore lo ottengo dalla cella sulla diagonale (ovvero l'ottimo è il valore $D(i-1, j-1)+\delta(a_i, b_j)$) e in tal caso aggiungo:
    $$\binom{a_i}{b_j}$$
    alla matrice di allineamento M
    - il punteggio migliore lo ottengo dalla cella in alto (ovvero l'ottimo è il valore $D(i-1, j) + \delta(a_i, -)$) e in tal caso aggiungo:
    $$\binom{a_i}{-}$$
    alla matrice di allineamento M
    - il punteggio migliore lo ottengo dalla cella in basso (ovvero l'ottimo è il valore $D(i, j-1) + \delta(-, b_j)$) e in tal caso aggiungo:
    $$\binom{-}{b_j}$$
    alla matrice di allineamento M

Questo algoritmo ha complessità sia in tempo che in spazio pari a:

$$O(m \cdot n)$$

Su slide esempio pratico.

4.1.2 Allineamento Locale di Smith-Waterman

Vediamo quindi l'allineamento locale di due sequenze. Si ricorda che:

- **input:** due sequenze, $A = a_1, \dots a_m$ e $B = b_1, \dots, b_n$, definite su algoritmo Σ, e uno schema di punteggio δ

- **output:** la coppia di sottostringhe di A e B a cui corrisponde il massimo (l'ottimo in questo caso può solo essere il massimo) score di allineamento globale

Vista la natura del problema, che può essere solo di massimo, si ha che, riprendendo la stessa nomenclatura dell'algoritmo di Needleman-Wunsch, si ha che lo score δ è solo di **similarità**, avendo quindi che:

- $s, d < 0$
- $k > 0$

L'algoritmo di Smith-Waterman permette di identificare una regione comune a due sequenze che non verrebbe evidenziata dall'allineamento globale con Needleman-Wunsch. Un approccio di allineamento globale permette di evidenziare le sottostringhe più simili.
Si procede quindi fissando i e j e, considerando tra tutte le coppie di sottostringhe che finiscono in a_i su A e b_j su B quella che ha il massimo score di allineamento globale $D(i, j)$.
Si ha che:

$$\max\{D(i, j)\}$$

è lo score di allineamento locale tra A e B.
Il calcolo di $D(i, j)$ varia quindi rispetto all'algoritmo di Needleman-Wunsch in quanto, per come sono stati definiti k, s e d, potrei avere valori negativi ma si impone un limite inferiore pari a 0 che tonerà poi utile nell'algoritmo.
Si ha quindi:

$$D(i, j) = \max \begin{cases} D(i-1, j-1) + \delta(a_i, b_j) \\ D(i-1, j) + \delta(a_i, -) \\ D(i, j-1) + \delta(-, b_j) \\ 0 \end{cases}$$

con:

- $\delta(a_i, b_j) = k$ se $a_i = b_j$
- $\delta(a_i, b_j) = s$ se $a_i \neq b_j$
- $\delta(a_i, -) = \delta(-, b_j) = d$

Anche i casi base variano, avendo che la prima riga e la prima colonna sono interamente inizializzate a 0:

- $D(0, 0) = 0$

- $D(i, 0) = 0, \forall i = 1, \ldots m$
- $D(0, i) = 0, \forall j = 1, \ldots n$

L'algoritmo è analogo, a livello di "procedura" (salvo ovviamente l'individuazione della cella ottima), a quello di Needleman-Wunsch e consiste quindi in:

1. riempire la matrice di score calcolando tutti i $D(i, j)$
2. individuare la cella ottima, ovvero la cella con in massimo valore in D
3. ricostruzione della matrice di allineamento M a partire da tale cella ottima

Bisogna quindi capire come riempire tale matrice D, dati k, s e d:

1. si specifica che D è una matrice $(m + 1) \times (n + 1)$, con righe indicizzate per $i \in [0, m]$ e colonne indicizzate per $j \in [0, n]$. La riga di incide $i, i > 0$, corrisponde ad a_i e la colonne di indice $j, j > 0$ al simbolo b_j
2. inizializzo a 0 la prima riga e la prima colonna
3. scorro riga per riga (o colonna per colonna) e riempio al matrice secondo il calcolo sopra definito per $D(i, j)$. Si ricorda che se dalle tre direzioni si ha massimo negativo si inserisce 0
4. identifico $D(j, i)$ la cella ottima $D(j, i)$

In merito alla ricostruzione dell'allineamento si procede quindi a ritroso:

1. si inizializza una matrice M di allineamento vuota
2. parto dalla cella ottima di D e seguo un percorso a ritroso fino alla prima cella contente 0 che si incontra
3. ogni passaggio da una cella aggiungo una coppia di simboli a M. Si hanno quindi tre casi:
   - il punteggio migliore lo ottengo dalla cella sulla diagonale (ovvero l'ottimo è il valore $D(i-1, j-1)+\delta(a_i, b_j)$) e in tal caso aggiungo:
     $$\begin{pmatrix} a_i \\ b_j \end{pmatrix}$$
     alla matrice di allineamento M

- il punteggio migliore lo ottengo dalla cella in alto (ovvero l'ottimo è il valore $D(i-1,j)+\delta(a_i,-)$) e in tal caso aggiungo:
$$\begin{pmatrix} a_i \\ - \end{pmatrix}$$
alla matrice di allineamento M
- il punteggio migliore lo ottengo dalla cella in basso (ovvero l'ottimo è il valore $D(i,j-1)+\delta(-,b_j)$) e in tal caso aggiungo:
$$\begin{pmatrix} - \\ b_j \end{pmatrix}$$
alla matrice di allineamento M

Questo algoritmo ha complessità sia in tempo che in spazio pari a:

$$O(m \cdot n)$$

Su slide esempio pratico.

4.1.3 Allineamento Semiglobale

Vediamo quindi l'allineamento semiglobale tra due sequenze. Si ha che;

- **input:** due sequenze, $A = a_1, \dots a_m$ e $B = b_1, \dots, b_n$, definite su algoritmo Σ, e uno schema di punteggio δ
- **output:** si hanno 8 possibili output:
  1. il prefisso di A più simile a B
  2. i due prefissi più simili
  3. i due suffissi più simili
  4. la sottostringa di A e il suffisso di B più simili
  5. il suffisso di A più simile a B
  6. la sottostringa di A più simile a B
  7. la sottostringa di A e il prefisso di B più simili
  8. il suffisso di A e il prefisso di B più simili

Bisogna quindi determinare opportunamente $D(i,j)$ a partire da:

- le equazioni di ricorrenza per il calcolo di $D(i,j)$

- i valori di $D(i, 0)$ e $D(0, j)$
- particolari valori i e j che risolvano il problema

A questo punto si può:

- riempire la matrice degli score D
- trovare la cella ottima
- ricostruire la matrice di allineamento M andando a ritroso a partire dalla cella ottima di D

Vediamo quindi i vari casi (**per ognuno esempio pratico su slide**) per ottenere gli 8 output:

1. **ricerca del prefisso di A più simile a B**. Per farlo si ricerca tra tutti i prefissi di A quello con il massimo score di allineamento globale con B. In altri termini $D(i, j)$ è lo score di allineamento globale tra il prefisso di A che finisce in posizione i e il prefisso di B che finisce in posizione j e nel dettaglio:
    - $D(i, n)$ è lo score di allineamento globale tra il prefisso di A che termina in i e tutta la sequenza B
    - i_m, tale che $D(i_m, n)$ è massimo, è la posizione di fine su A del prefisso che ha il massimo score di allineamento globale con tutta B. Applico quindi Needleman-Wunsch con ottimo pari al massimo presente nella colonna di indice n di D, ripartendo da lì per ricostruire M (ricostruendo quindi fino a che non arrivo in $D(0, 0)$
2. **ricerca dei due prefissi più simili**, ovvero tra tutte le coppie di prefissi di A e di B cercare quella che ha il massimo score di allineamento globale. In altri termini $D(i, j)$ è lo score di allineamento globale tra il prefisso di A che finisce in posizione i e il prefisso di B che finisce in posizione j e nel dettaglio:
    - i_m, j_m, tali che $D(i_m, j_m)$ è massimo, sono rispettivamente la posizione di fine su A e B dei prefissi che hanno il massimo score di allineamento globale. Applico quindi Needleman-Wunsch con ottimo pari al massimo presente nella matrice D, ovvero tale $D(i_m, j_m)$ e parto da lì per ricostruire M (ricostruendo quindi fino a che non arrivo in $D(0, 0)$

3. **ricerca dei due suffissi più simili**, ovvero tra tutte le coppie di suffissi di A e di B cercare quella che ha il massimo score di allineamento globale. In altri termini $D(i, j)$ è lo score di allineamento globale tra la coppia di sottostringhe più simili che finiscono in posizione i su A e in posizione j su B. Nel dettaglio:

    - $D(m, n)$ è lo score di allineamento globale tra la coppia di suffissi che hanno il massimo score di allineamento globale. Applico quindi Smith-Waterman con cella ottima pari a $D(m, n)$ e ricostruisco, ovvero costruisco M procedendo a ritroso dalla cella ottima fino a che non trovo uno 0 nella matrice D. Tale 0 si trova in un certo $D(i_0, j_o)$ e ho che $i_0 + 1$ e $j_0 + 1$ sono rispettivamente la posizione di inizio del suffisso su A e B

4. **ricerca della sottostringa di A e del suffisso di B che sono più simili**, ovvero tra tutte le coppie formate da una sottostringa di A e un suffisso di B cercare quella che ha il massimo score di allineamento globale. In altri termini $D(i, j)$ è lo score di allineamento globale tra coppie di sottostringhe più simili che finiscono in posizioni i di A e j di B. Nel dettaglio:

    - $D(i, n)$ è lo score di allineamento globale tra la coppia più simile composta da una sottostringa che finisce in posizione i su A e un suffisso di B
    - i_m, tale che $D(i_m, n)$ è massimo, è la posizione di fine su A della sottostringa su A più simile ad un suffisso di B. Applico quindi Smith-Waterman con ottimo pari al massimo presente nella colonna di indice n di D, ripartendo da lì per ricostruire M, ricostruendo quindi fino a che non arrivo in al primo 0. Tale 0 si trova in un certo $D(i_0, j_o)$ e ho che $i_0 + 1$ e $j_0 + 1$ sono rispettivamente la posizione di inizio della sottostringa su A e del suffisso B

5. **ricerca del suffisso di A più simile a B**, ovvero tra tutti i suffissi di A cercare quello ha il massimo scoredi allineamento globale con tutta la sequenza B. In altri termini $D(i, j)$ è lo score di allineamento globale della coppia più simile formata da una sottostringa di A che finisce in posizione i e il prefisso di B che termina in posizione j. Costruisco D in modo che sia inizializzata

come previsto per Needleman-Wunsch ma con la prima colonna pari a 0. Nel dettaglio:

- $D(m, n)$ è lo score di allineamento globale tra tutta la sequenza B e il suffisso più simile su A. Applico quindi Needleman-Wunsch con ottimo pari a $D(m, n)$ e parto da lì per ricostruire M, ricostruendo quindi fino a che non arrivo in $D(i_0, 0)$. Si ha che $i_0 + 1$ è la posizione di inizio del suffisso su A

6. **ricerca della sottostringa di A più simile a B**, ovvero tra tutte le sottostringhe di A cercare quella ha il massimo score di allineamento globale con tutta la sequenza B. In altri termini $D(i, j)$ è lo score di allineamento globale della coppia più simile formata da una sottostringa di A che finisce in posizione i e il prefisso di B che termina in posizione j. La matrice D è la inizializzata come la D di Needleman-Wunsch ma con tutti 0 sulla prima colonna. Nel dettaglio:

   - $D(i, n)$ è lo score di allineamento globale tra tutta B e la sottostringa di A più simile
   - i_m, tale che $D(i_m, n)$ è massimo, è la posizione di fine su A della sottostringa più simile a B. Applico quindi Needleman-Wunsch con ottimo pari a $D(i_m, n)$ e parto da lì per ricostruire M, ricostruendo quindi fino a che non arrivo in $D(i_0, 0)$. Si ha che i_0 è la posizione di inizio della sottostringa su A mentre i_m quella di fine

7. **ricerca della sottostringa di A e del prefisso di B più simili**, ovvero tra tutte le coppie formate da una sottostringa di A e un prefisso di B cercare quella che ha il massimo score di allineamento globale. In altri termini $D(i, j)$ è lo score di allineamento globale tra la coppia formata da una sottostringa di A che finisce in posizione i e il suffisso di B che finisce in posizione j. La matrice D è la inizializzata come la D di Needleman-Wunsch ma con tutti 0 sulla prima colonna e nel dettaglio:

   - i_m, j_m, tali che $D(i_m, j_m)$ è massimo, sono rispettivamente la posizione di fine su A e B della sottostringa e del prefisso più simile su A e B. Applico quindi Needleman-Wunsch con ottimo pari al massimo presente nella matrice D, ovvero tale $D(i_m, j_m)$ e parto

da lì per ricostruire M (ricostruendo quindi fino a che non arrivo in $D(i_0, 0)$. Si ha che i_m e j_m sono rispettivamente la posizione di fine della sottostringa su A e del prefisso su B. Inoltre $i_0 + 1$ è la posizione di inizio della sottostringa su A

8. **ricerca del suffisso di A e del prefisso di B più simili**, ovvero tra tutte le coppie formate da un suffisso di A e un prefisso di B cercare quella che ha il massimo score di allineamento globale. In altri termini $D(i, j)$ è lo score di allineamento globale della coppia più simile formata da una sotto stringa di A che finisce in posizione i e il prefisso di B che termina in posizione j. D è costruita come la matrice di Needleman-Wunsch con però la prima colonna a 0 nel dettaglio:

    - $D(m, j)$ è lo score di allineamento globale tra il prefisso di B che termina in j e il suffisso di A più simile
    - j_m, tale che $D(m, j_m)$ è massimo, è la posizione di fine su B del prefisso più simile ad un suffisso di A. Applico quindi Needleman-Wunsch con ottimo pari al massimo presente nell'ultima riga di D, ripartendo da lì per ricostruire M, ricostruendo fino a che non arrivo in $D(i_0, 0)$. S ha chej_m è la posizione di fine del prefisso su B mentre $i_0 + 1$ è la posizione di inizio del suffisso su A

Capitolo 5

Alberi Filogenetici

Vediamo come ricostruire la storia evolutiva in bioinformatica.
Useremo la nozione di **albero evolutivo**.
L'evoluzione è guidata da, per Darwin:

- **diversità**, con individui diversi che portano variazioni genetiche
- **mutazioni**, ovvero cambiamenti nella sequenze di DNA, con mutazioni nucleotidiche o genomiche
- **selezione naturale**, con mutazioni che favoriscono la sopravvivenza o altre che la impediscono, portando morte o impossibilità di trasmettere la mutazione

Definizione 25. *Definiamo* ***filogenesi****, o* ***albero filogenetico****, come un albero che descrive le sequenze di eventi di speciazione che hanno portato alle specie attuali, rappresentate nelle foglie di questo albero. I nodi interni non rappresentano specie viventi.*

La costruzione di un albero si usano i **caratteri morfologici** (lunghezza, numero di arti etc...) e attualmente anche l'informazione molecolare data da sequenze geniche e proteiche. Un classico tipo di sequenza genomica usato è il gene dell'emoglobina, che è un gene si è evoluto con mutazioni specifiche per le specie che si analizzano.

La **topologia** è la forma dell'albero evolutivo. Ogni nodo interno è un antenato comunque delle specie nelle foglie del sottoalbero che ha tale nodo come radice

La filogenesi dipende dall'input, usando ad esempio il **DNA mitocondriale** (che è un DNA particolarmente conservato) rispetto che altro, come

l'emoglobina, si ottengono alberi diversi, studiando nel primo caso la somiglianza a livello di DNA mitocondriale. Un altro tipo di DNA spesso usato è il **DNA nucleare**.
Con il termine **speciazione** si indica la creazione di specie differenti, portando a due gruppi con una certa differenza genetica. Ogni coppia di specie da questi due gruppi condivide uno stesso antenato.
Quando si ha la speciazione si ottengono due gruppi con differenza genetica predominante, quindi molto "distanti".
Si hanno varie assunzioni:

- organismi vicini hanno genomi simili
- geni simili sono omologhi, ovvero hanno lo stesso antenato. A seguito di una duplicazione genica però si possono avere eventi di speciazione che porterà poi a specie che differiscono
- esiste un antenato universale comune a tutti, si ipotizza
- le differenze molecolari in geni omologhi sono correlate al tempo di evoluzione. Geni duplicato omologhi hanno differenze molecolari dipendenti dalla distanza evolutiva rispetto al tempo in cui è avvenuto l'evento di speciazione (???). Ad esempio le emoglobine di specie differenti, quanto più le specie sono vicine, tanto meno si hanno differenze a livello di mutazioni

Vediamo l'albero. Si hanno:

- le foglie sono le specie attuali
- la lunghezza/costo degli archi è correlata alla distanza con l'antenato comune
- i nodi interni sono antenati comuni

A seconda dell'evento si hanno geni omologhi per diversi motivi:

- **orthologs** dopo una speciazione
- **paralogs** dopo una duplicazione
- **xenolog** dopo un trasferimento orizzontale, come quello di un virus

Abbiamo anche un **albero di geni** dove nelle foglie si ha la sequenza genica, con varie copie dello stesso gene dopo un evento di speciazione (magari la versione 1A, la versione 2A e quella 3A). Posso avere sia eventi di duplicazione che di speciazione.
Un evento di duplicazione non dovrebbe creare nuove specie, cosa che avviene con la speciazione.
Posso avere alberi di diverso tipo:

- **con radice**, che comporta più versioni a parità di specie, quando si vuole cercare l'albero migliore che rappresenti l'evoluzione
- **senza radice**, dove comunque posso studiare percorsi tra specie

Una radice in un albero, qualora non sia presente, si sceglie un **outgroup**, che è molto distante dalle specie di interesse e quindi lo si attacca direttamente alla radice.
Si hanno tre metodi di ricostruzione per l'evoluzione:

1. basato su **distanza**, dove ricorsivamente si combinano due nodi a minima distanza, collegandole e creando antenati comuni. Circa quello che si fa nel clustering con il neighbor joining
2. basato su **parsimonia**, dove si minimizza il numero di cambiamenti. È il **rasoio di Occam**
3. basato su **maximum likelihood**, dove si usano reti Bayesiane a forma di albero. Sia associa una misura all'albero e si fa un calcolo statistico. È un metodo oneroso

5.1 Metodi basati su distanza

Si ha:

- **input**: matrice delle distanze tra le specie, calcolate tramite allineamento globale e si fa un calcolo basato sulle differenze. Si ha il numero delle mutazioni richieste
- **outline**: si clusterizzano le specie, avendo all'inizio dei singleton. Ad ogni iterazione si combinano due cluster vicini in uno nuovo, tramite un algoritmo come UPGMA, che si basa su neighbor joining. Quindi si clusterizza e si crea l'albero. I singleton sono le foglie e ogni clusterizzazione successiva un antenato ai due nodi dei cluster che unisco. Dopo la creazione di un cluster devo riaggiornare le distanze tra il nuovo cluster e i cluster restanti

- **output**: una filogenesi con radice

esempio su slide. Bisogna capire come calcolare la distanza tra cluster.

Definizione 26. *Si definisce* ***orologio molecolare*** *come l'assunzione per cui tutte le foglie sono allo stesso livello, dimenticando i tempi di evoluzione, avendo tempo identico su tutti i cammini. È una forzatura della realtà in quanto normalmente l'orologio molecolare dovrebbe tenere traccia del tempo reale.*
Un albero soddisfa l'assunzione dell'orologio molecolare quando le foglie sono tutte allo stesso livello.

UPGMA costruisce una filogenesi con radice. Si hanno punti nello spazio e in modo iterativo si raggruppano i cluster.
Si ha quindi:

- **input**: una matrice D $n \times n$ con l'insieme delle specie $S = \{s_1, \dots s_n\}$, con all'interno le distanze tra due specie, ovvero il numero di mutazioni necessarie per passare da una specie all'altra. Nella matrice D ho quindi le n specie che identificano righe e colonne. La matrice è ovviamente simmetrica. Per capire il numero di mutazioni si usa l'allineamento globale usando apposite matrici di punteggio. Tramite queste distanze si ricostruisce l'evoluzione.

- **output:** un albero filogenetico T le cui foglie sono le specie (che in questo caso sono rappresentate come sequenze) e l'albero è tale che $\forall s_i, s_j \in S$ si ha che $D(s_i, s_j) = d_T(s_i, s_j)$, ovvero è uguale alla distanza stimata nell'albero. Si dice che T è consistente con la matrice D

Vediamo quindi come fare un algoritmo per ottenere T da D. È un algoritmo base di **neighbor joining**.
Si ha un'iterazione di passi:

1. tra tutte le coppie di specie della matrice D scelgo la coppia x, y tale che $D(x, y)$ è la minima

2. aggiungo x e y all'albero T collegandole con un nodo padre che rappresenta $\{x, y\}$

3. si assegna agli archi che collegano le due specie al nodo $\{x, y\}$ dei pesi pari a $\frac{D(x,y)}{2}$

4. aggiorno D:
    - sostituisco x e y con $\{x, y\}$
    - preso un nodo z si ha che $D(z, \{x, y\}) = D(z, x) - D(x, \{x, y\})$
    - chiamo questa nuova D come D', si noti che sarà di dimensioni minori
5. ripeto dal passo 1

Non tutte le matrici ammettono un albero (questo a causa della semplicità della formula dell'algoritmo) perché devono valere certe condizioni. La matrice deve essere **ultrametrica**. Per rendere una matrice una valida in caso posso modificare i costi, normalizzando la matrice delle distanze.
Nei metodi basati su distanze la matrice in input deve avere distanze consistenti con la costruzione dell'albero. Qualora non lo fosse si devono normalizzare in qualche modo le distanze.
Per UMGMA si avrebbe invece una formula più complessa che non viene approfondita nel corso. Comunque, dati C_i e C_j due cluster si ha che la loro distanza è pari a:

$$d(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{p \in C_i} \sum_{q \in C_j} d(p, q)$$

e quando si combinano per ottenere C_k si ha, dato un qualsiasi altro C_j:

$$d(C_k, C_l) = \frac{|C_i| d(C_i, C_l) + |C_j| d(C_j, C_l)}{|C_i||C_j|}$$

si definisce quindi il nodo k e si pongono i nodi a profondità:

$$\frac{d(C_i, C_j)}{2}$$

UPGMA soddisfa l'orologio molecolare. Lato score abbiamo ad esempio la **matrice Jukes-Cantor**, che assume eguale tasso di mutazione, che confronta singoli nucleotidi (usando pesi negativi per la diagonale principale dove non si hanno mutazioni). D'altro canto si ha che ogni altra mutazione è egualmente probabile.
Si ha anche il **modello Kimura** usato per le proteine, che da un peso diverso a transizioni e trasversioni (dove una purina diventa una piramidina), avendo quindi score diversi a seconda della coppia di basi. Questo modello considera la chimica del DNA.

5.1.1 Metodi Basati su Caratteri

Il modello discreto basato su caratteri è importante per diversi aspetti. La storia evolutiva è basata anche su caratteri discreti e non solo su distanze. La presenza di caratteristiche è un aspetto generale e ha importanza nella ricostruzione della filogenesi tumorale, tramite la **infinite site assumption**, per la quale si ritiene che una volta acquisita una mutazione resta per sempre, ottenendo la **filogenesi perfetta**.
Le mutazioni puntuali però spesso improvvisamente spariscono non potendo più usare l'assunzione. Si procede quindi a costruire l'albero evolutivo tramite caratteristiche comuni delle varie specie. Quando compare un carattere discreto in un nodo non viene più perso e tutte le specie sottostanti hanno quella caratteristica. Ogni carattere appare una volta sola nell'albero.
Bisogna tradurre l'albero in una matrice le colonne sono etichettate dalle caratteristiche e le righe dalle specie. Si ha che $a_{ij} = 1$ se la specie i ha la caratteristica j, 0 altrimenti.
Gusfield trovò un algoritmo lineare per costruire tale albero.
Tra gli scopi dell'algoritmo si ha lo studio di genotipi e aplotipi, essendo tra l'altro l'uomo un organismo diploide. Gli organismi diploidi hanno due copie di ogni cromosoma con piccole differenze. Il problema fondamentale è che sequenziando non ho modo di capire se un mismatch sia dovuto al fatto che ho le due sequenze materne e paterne.
L'uomo è anche biallelico, avendo che le opzioni per una certa posizione, sono due, che si indicano con 0 e 1. Si ha che 0 è l'allele dominante e 1 è detto allele di minoranza o allele recessivo. Se su due aplotipi ho valori diversi si parla di eterozigosi rispetto un certo allele.
La radice di un albero dove 0 rappresenta l'assenza di allele non si ha alcuna mutazione.
Si ha quindi il seguente problema:

- **input**: una matrice di aplotipi su $\{0, 1, *\}$
- **output**: una matrice binaria di aplotipi che risolve i genotipi in M e ammette una filogenesi perfetta

Ultima lezione da rivedere interamente.
Riassumendo il problema fondamentale è quello della ricostruzione dell'evoluzione tenuto conto che essa è legata a eventi, ovvero mutazioni, di tipo nucleotidico (che studiamo con l'allineamento globale) e di tipo genomico (che vedremo essere studiate tramite riarrangiamenti, con trasposizioni, inversioni, duplicazione genica etc...), anche a livello di cromosomi.
L'evoluzione è legata a questi eventi che comportano speciazione.

Si hanno quindi metodi basati su distanze, caratteri e parsimonia.
SI usa il DNA mitocondriale perché è quello che si "porta dietro" la storia delle specie, essendo quello meno modificato nel corso della storia. Si usa anche quello nucleare. Spesso si hanno problemi di **riconciliazione** dell'evoluzione, infatti prendendo geni diversi delle stesse specie ottengo alberi evolutivi diversi e quindi si vuole appunto "riconciliare" tali alberi, ottenendo l'**agreement** di essi. Anche metodi computazionali diversi producono alberi diversi e non si può sapere facilmente quale sia giusto. Ricostruire l'albero della vita è comunque un tempo di spazio e tempo computazionale.
In generale gli alberi evolutivi sono utilizzati nell'analisi delle sequenze. Quando si fa allineamento, specialmente multiplo, si avvale anche dell'albero filogenetico per comparare le sequenze. Nella matrice di allineamento multiplo il costo di una colonna è legato al costo degli archi in un albero evolutivo, che quindi è usato per stimare l'allineamento. Un esempio pratico è l'analisi del SARS-CoV2 dove si fanno le analisi in termini di mutazioni, associando alle sequenze dei cluster che seguono un'evoluzione nel tempo.
L'evoluzione non riguarda solo le specie ma anche l'inferenza di aplotipi da genotipi e anche l'evoluzione tumorale. Anche nel caso dei tumori ho alberi diversi a seconda del metodo ma in questo caso anche i nodi interni sono etichettati, a differenza dell'albero filogenetico per le specie.

5.1.2 Metodi Basati su Parsimonia

In questi metodi su ha che l'ipotesi più veritiera è quella che minimizza gli eventi, ovvero con il **rasoio di Occam**. Questo metodo si applica anche in altri contesti in quanto la natura è in se "parsimoniosa". Si assume in questo metodo che ciascuna specie è specificata dai caratteri o stati di attributi. Un albero di massima parsimonia ha come foglie gli tali stati di attributi delle specie e i nodi interni sono etichettati con i cambiamenti di stato, che, in tutto l'albero, sono minimizzati. Nel dettaglio un **carattere** è ad esempio un fenotipo, ovvero un certo attributo visibile, ma anche una certa informazione genomica (tramite ad esempio lo studio degli SNPs). Questi ultimi sono anche detti **caratteri genomici**. Con il termine **WGS (*whole genome scale*)** si indicano caratteri e attributi che sono ottenuti da lavori a larga scala. WGS fornisce vari caratteri o attributi, come indicazioni sui geni, combinazioni proteiche, SNP, studio di esoni e introni etc. . ..
La scelta dei caratteri influenza l'albero. I caratteri possono essere binari con 0 assenza e 1 presenza, usando quindi un vettore binario per gli stati.
Si ha la **parsimonia di Dollo**, dove si ha un solo passaggio da 0 a 1 ma tanti passaggi da 1 a 0 per un certo carattere c. Quindi dati c_i caratteri ho un albero che presenta l'etichetta c_i una e una sola volta. Una volta comparso un

carattere comunque può sparire. Le foglie sono quindi gli stati finali, ovvero le specie attuali, e i nodi interni degli stati che poi sono cambiati. Si hanno quindi:

- **input:** le foglie
- **output:** l'albero

Nel modello **Camin-Sokal** si ha invece che un carattere viene guadagnato ma non perso, ho quindi che c può passare da 0 a 1 quante volte si vuole. Posso quindi avere più archi etichettati con un certo c_i. Questo modello non si può usare per le assunzioni fatte con gli alberi tumorali. Questo modello ammette che due specie possano acquisire in modo indipendentemente una caratteristica.

Filogenesi Perfetta Binaria

Combinando, ovvero facendo l'intersezione, i modelli di Dollo e Camin-Sokal ottengo la **filogenesi perfetta binaria**, dove posso avere una sola volta il cambiamento da 0 a 1 e mai i cambiamenti da 1 a 0. Un carattere etichetta quindi un solo arco dell'albero e una volta guadagnato non è mai perso. Ho quindi un **gain** e **no loss**. Tale modello è risolvibile in tempo lineare mentre i due problemi di partenza sono **NP-complete**.
Per la filogenesi binaria perfetta si ha:

- **input:** una matrice binaria $n \times m$, con n specie, sulle righe s_i, e m, sulle colonne c_i caratteri
- si crea un albero dove ogni nodo x è etichettato da un vettore lungo m, detto l_x, dove $l_x[j]$ è lo stato del carattere c_j. La radice è etichettata con un vettore di m zeri. Se sono in $(0, 0, 0)$, ipotizzando tre caratteri, e passo con un arco etichettato con c_i ad un altro nodo esso sarà etichettato con $(1, 0, 0)$. Per ogni carattere c_j esiste esattamente un arco e etichettato c_j con c_j che rappresenta il cambiamento $0 \to 1$. Ciascuna riga della matrice etichetta esattamente una foglia dell'albero
- **output:** l'albero, se esiste

Se faccio, nell'albero di filogenesi perfetta, in un cammino da radice a foglia incontro solo i cambiamenti di stato che portano ad ottenere la foglia.

La matrice in input potrebbe non permettere la costruzione dell'albero, per le condizioni imposte. Ad esempio:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

è una **matrice proibita**.
La filogenesi perfetta è una **dollo 0**, ovvero non posso perdere caratteri.
Posso avere una **dollo k** dove il carattere può essere perso k volte.
Il calcolo della **dollo 0** è lineare per l'algortimo di Gusfield e ogni foglia sottostante un nodo dove si è acquisito un carattere contiene quel carattere.
La **dollo 1** si pensa sia comunque polinomiale e si usa nella filogenesi tumorale.
La **dollo k**, per $k \geq 2$ è NP-complete con però algoritmi parametrici.
La matrice proibita la posso rappresentare assumendo la loss del carattere.
Capiamo quando una matrice binaria ammette filogenesi perfetta.

Teorema 2. *Una matrice binaria ammette filogenesi perfetta se non contiene la matrice proibita.*

Ma possiamo dirlo diversamente.

Definizione 27. *Definiamo:*

$$O_j = \{i |\, M_{ij} = 1\}$$

come l'insieme di tutte le specie i con il carattere j.

Definizione 28. *Una collezione di insiemi $O_1, \ldots O_n$ è detta **laminare** se per $\forall i, j$, con $1 \leq i$ e $j \leq n$, o O_i e O_j sono disgiunti o un insieme include l'altro.*

Teorema 3. *Una matrice binaria è una filogenesi perfetta sse la collezione delle colonne di M è laminare.*

Quindi studio le colonne delle matrici. Per ogni coppia di colonne o gli insiemi, rappresentati dai vettori colonna, sono disgiunti o uno contiene l'altro.
La laminarità corrisponde alla non esistenza della matrice proibita. Infatti, prendendo la matrice, $(0, 1, 1)$ e $(1, 0, 1)$ sono disgiunti ma nessuno dei due contiene l'altro.

Teorema 4. *La laminarità è una condizione necessaria e sufficiente per la costruzione dell'albero.*
L'albero di filogenesi perfetta sse la matrice in input non contiene la matrice proibita.

Potrei quindi confrontare ogni colonna della matrice, in $O(n \cdot m^2)$ ma si può fare di meglio, avendo che la lettura di una matrice è in $O(n \cdot m)$.
Per vedere se una matrice è laminare, per Gusfield:

- si riordinano le colonne per numero di 1 non crescenti (la prima colonna è quella con più 1 e l'ultima quella con meno). Questo porta vantaggi anche per l'albero in quanto il primo carattere, quello della prima colonna nella matrice riordinata, è quello che probabilmente è in cima all'albero. Chiamo M tale matrice

- si costruisce una matrice ausiliaria quando si legge M. In questa nuova matrice L, quando si legge M, si memorizza per ogni posizione di M che è a 1 dove si trova a sinistra di quella posizione il primo 1 (che tra i vari 1 a sinistra è quindi quello più destra). Qualora tale 1 non esistesse (sto leggendo il primo 1 di una riga) si mette -1 in L. Se si legge 0 in M metto 0 in L.
  Qualora si faccia la matrice ausiliaria di una matrice proibita si ha che nelle colonne si ottengono colonne con valori non nulli diversi (ho quindi -1 e 1)

- eseguo il vero e proprio test, ovvero, per ogni colonna della matrice ausiliaria, verifico di avere solo valori uguali non nulli in quanto significherebbe

L'algoritmo per la costruzione dell'albero ha quindi tempo $O(n \cdot m)$ (che è anche il tempo per costruire la matrice d'appoggio per il test di laminarità). Si ha quindi un algoritmo molto performante e decidere se esso esiste.
L'algoritmo lavora su ogni riga della matrice ripartendo ogni volta dal source e aggiungendo gli archi ogni volta che ho un 1 nella matrice, etichettandoli con il carattere necessario.

Si ha quindi l'algoritmo:

Algorithm 2 algoritmo di creazione albero di filogenesi a partire da matrice laminare già ordinata

```
function CreatePerfectPhylogeny(M')
    create root of tree T
    for i = 1 to n do
        curr ← root
        for j = 1 to m do
            if M_ij == 1 then
                if already exist edge (curr, u) labeled C_j then
                    curr ← u
                else
                    create u
                    create edge (curr, u) labeled C_j
                    curr ← u
        label curr as S_i
    for each node u except root do
        Create as many links to u species labeling u
    return T
```

Le applicazioni dell'algoritmo sono l'inferenza di aplotipi da genotipi e l'applicazione agli alberi tumorali.

5.1.3 I Tumori

Un **clone** è una famiglia di cellule con un antenato comune con una certa mutazione. In generale un **tumore** si crea a causa di un accumulo di mutazioni ed è un agglomerato di **cloni cancerogeni**, è quindi un misto di cellule cancerogene e sane.
Diversi cloni formano diverse parti del tumore.
All'inizio si ha una sola cellula mutata da cui scaturisce tutta l'evoluzione tumorale, dove si accumulano altre mutazioni. Si formano quindi vari cloni con anche le nuove mutazioni, seguendo le regole della filogenesi perfetta, coi nuovi cloni come foglie dell'albero.
Tramite chemioterapia si uccidono tali cloni ma tipicamente qualche clone sopravvive, che a sua volta fa partire una nuova evoluzione tumorale.
Le mutazione che si accuulano sono dette *somatic* e sono diverse da quelle *germline*, che sono quelle ereditate di genitori. Le mutazioni somatic sono acquisite dal singolo individuo nel corso della sua esistenza. Anche il corpo

stesso cerca comunque di "aggiustarsi" limitando la diffusione di tali cellule. Si hanno quindi varie sezioni del tumore caratterizzate da diverse mutazioni. Un misto di cloni viene chiamato **sample**. Un sample viene rappresentato come un insieme contenente cerchi con all'interno le varie mutazioni. Un cerchio senza pallini interni indica l'assenza di mutazioni.
Quando si sequenzia un genoma tumorale non so quale frammento sto guardando ma posso contare la frequenza delle basi per un certo pezzo per capire la frequenza di una mutazione. Per farlo confronto il pezzo di genoma con la reference di un pezzo sano. Questa frequenza è detta **VAF (variant allele frequency)**.
Si costruisce quindi la VAF matrix che useremo poi per la filogenesi perfetta. La matrice ci dice la frequenza delle mutazioni avendo come caratteri le varie mutazioni. Le righe sono invece i **sample**, che in se sono collezioni di read prese da un campione tumorale. La matrice contiene quindi la frequenza di una mutazione in un certo sample, avendo quindi valori tra 0 e 1. Da questo dobbiamo arrivare a costruire l'albero, ricostruendo l'ordine con cui sono avvenute le mutazioni. Quando si misura la frequenza devo anche considerare il fatto che l'organismo è diploide, dovendo contare tutto doppio. Per ora trascuriamo la cosa.
Una volta ottenuta la matrice VAF devo capire come sono formato i vari cloni nei vari sample, stabilendo anche un ordine evolutivo delle mutazioni.
Dalla matrice VAF si costruisce una matrice di filogenesi perfetta e da qui l'albero, che avrà come foglie i vari cloni ma non solo. Anche i nodi interni sono etichettati come cloni, eventualmente come una foglia se essa è ottenuta da quel nodo interno con un arco non etichettato. I cloni, quando sono generati, rimangono tutti.
Con i tumori ci interessa infatti sapere tutta l'evoluzione tumorale.
Per ogni riga della matrice ho quindi una foglia con quelle mutazioni, posso comunque chiamarle specie.
Si crea anche una **matrice di utilizzo** che ci dice quanti cloni usano una determinata mutazione, che si usa per costruire il disegno di sample e cloni. Si costruisce con i sample sulla riga e le specie (che sarebbero le righe della matrice di filogenesi) sulle colonne e ci dice la frequenza assoluta di un clone in un certo sample. Si ha quindi l'utilizzo dei cloni nei vari sample.
Si ha che:

$$VAF = \frac{1}{2}UB$$

con B matrice di filogenesi e U (*Usage*) matrice di utilizzo. La divisione per 2 è dovuta sempre al discorso dell'essere diploide.
La numerosità del clone è legata alla frequenza della matrice VAF, avendo che già ci ice una storia evolutiva e circa ci dice la forma dell'albero.

Si può quindi costruire l'albero direttamente dalla matrice VAF, dovendo inferire B e U per ottenere la matrice VAF. Si usano metodi di programmazione lineare o tramite euristiche, cercando di indovinare matrici B e U valide per ottenere la matrice di frequenze VAF.
Non sempre comunque il modello della filogenesi perfetta riflette l'evoluzione tumorale, potrei non avere un modello Dollo, avendo mutazioni che spariscono (quindi con una Dollo 1) o una mutazione che etichetta più di un arco.
La filogenesi perfetta è comunque un'ottimo modello di base per ottenere, appunto **l'albero dei cloni**.

5.1.4 Assemblaggio di Aplotipi

L'**assemblaggio di aplotipi** è un problema prettamente computazionale.
L'uomo è diploide quindi abbiamo due insiemi di cromosomi, uno dal padre e uno dalla madre. Si hanno 23 coppie di cromosomi e quindi ogni coppia è costruita da due aplotipi, ovvero da due sequenze che costituiscono l'aplotipo.
Un aplotipo è quindi una sequenza nucleotidica e ogni posizione è detta locus.
Il valore di un locus è un allele specifico. L'uomo è biallelico potendo avere due alleli per locus, se studio contemporaneamente i due aplotipi, dicendo che si sta studiando il genotipo, che è appunto la coppia di alleli di un locus.
Le posizioni in cui si hanno due alleli identici sono omozigote, altrimenti son o eterozigote. Anche in questo caso si ricollega il discorso fatto tempo fa sugli SNPs, che sono appunto le differenze di aplotipi.
Su 150 basi di una lettura di Illumina posso avere anche solo uno o due SNPs.
A livello informatico per l'aplotipo, come coppia di cromosomi, usiamo un vettore binario, con 0 per l'allele di maggioranza e 1 per quello di minoranza.
I siti/locus umani sono biallelici, potendo avere due alleli possibili. Le piante sono addirittura poliploidi. Avendo i due aplotipi ho che il genotipo è quindi un vettore di coppie di valori binari.
Per ogni cromosoma abbiamo due copie.

Teorema 5 (Legge Di Mendel)**.** *Per ogni cromosoma su hanno due coppie, una ereditata dalla madre e una dal padre.*

A questo si aggiungono le **ricombinazioni nell'ereditarietà** (per la quale magari un genitore portatore sano non fa ereditare la malattia al figlio). In questo caso un figlio prende un pezzo del primo cromosoma e un pezzo del secondo per un certo genitore. Quindi non eredita uno dei due cromosomi di un certo genitore ma un mix di entrambi i cromosomi di quel genitore, ereditando quindi anche dai nonni. Con questo discorso si va oltre la mera legge di Mendel avendo un mescolamento degli aplotipi.
Avere la coppia di array binari rappresentanti gli aplotipi permette di vedere

omozigosi e eterozigosi, qualora si abbiano o meno 1/0 da entrambe le parti. Due 0/1 li rappresento spesso con un solo 0/1, dicendo che è homo0 o homo1, mentre la differenza con *.
Tendenzialmente gli SNPs sono posizioni eterozigote, in quanto è dove si ha un allele di minoranza su uno degli aplotipi.
Gli SNPs e gli aplotipi sono la vera informazione interessante e quindi bisogna ricostruire gli aplotipi in quanto una costruzione sperimentale non è fattibile e quindi bisogna procedere in modo computazionale, anche in modo statistico o combinatorio. Bisogna inferire gli aplotipi, cercando di indovinarli in quanto nessuno può dire che sia corretto se non tramite studio sperimentale costoso.
Si fa sequenziamento anche per questo motivo. Si generano le read di lettura dei cromosomi, si prende il reference e si allineano le read al reference studiano le differenze. Le read sono quindi bipartite, tramite SNPs comuni, per ricostruire gli aplotipi. Si fa quindi un'operazione di cluster. In tutto questo però abbiamo comunque gli errori di sequenziamento, anche se su short read con Illumina essi sono davvero pochi. Quando si sequenzia, si ricorda, non si sa se si sta analizzando la parte ereditata dal padre o dalla madre del cromosoma.
Se non avessi errori di sequenziamento avrei un algoritmo polinomiale ma con gli errori si passa ad un problema di ottimizzazione. Tramite i siti eterozigoti con l'allineamento con il reference si vedono le differenze. In alcune read corte posso avere due SNPs per ogni read.
Dall'allineamento si ottiene una matrice binaria detta **fragment matrix**, con le read come righe e le posizioni SNPs come colonne. Il contenuto è 0/1 a seconda di avere un sito eterozigota o meno. Tramite la matrice posso fare il bi-partizionamento in tempo polinomiale, trovando conflitti, ovvero valori diversi a parità di posizione. La presenza di errori di sequenziamento porta però a incoerenze. La matrice binaria è creata dopo aver già fatto pulizia di errori di sequenziamento ma potrebbe non bastare. A questo punto posso pensare di eliminare le righe della matrice binaria che creano problemi ma non sempre è così semplice, dovendo capire cosa va bene e cosa no. Si ha quindi un problema NP-Complete, che è stato però parametrizzato. Si parla quindi di **Minimum Fragment Removal (*MFR*)**.
Un'altra soluzione è rimuovere le colonne con gli SNPs problematici, parlando quindi di **Minimum SNP Removal (*MSR*)**. Anche questo è un problema NP-Completo.
Si procede in realtà con la **Minumum Error Correction (*MEC*)**, correggendo le entry errate, che si individuano errate studiando le colonne. Ogni coppia di colonna o le due colonne sono uguale o le colonne sono complementari, altrimenti ho ambiguità che impediscono la bipartizione. Da questo

discorso si eliminano eventuali gaps, di cui non ho interesse. Quindi le colonne devono essere uguali o complementari al più dei gap in una delle due colonne. Si dice che le colonne devono essere **a coppie concordanti**.
Con MEC si prende in input la fragment matrix e si cerca il numero minimo di correzioni per permettere la disambiguazione. Si ha anche **wMEC**, che assegna un peso ad ogni elemento in modo da minimizzare il peso totale di correzione me migliorare l'accuratezza. Anche MEC è NP-Complete ma si ha che si possono applicare euristiche per il discorso di studio di colonne concordanti (cosa verificabile in tempo quadratico). MEC ha anche un buon algoritmo parametrico, con k pari al numero di righe, che è tendenzialmente piccolo visto che si hanno anche 5 read, per coverage pari a 5.
Si hanno altre varianti di MEC, al variare dei tipi di gap (senza, con, con holes etc...).
Si ha anche la versione $K-cMEC$ dove si hanno al più k correzioni per colonna. Questa versione si presta molto alla programmazione dinamica. Dopo aver partizionato le read bisogna anche assemblare (quindi pensare di correggere con gap in MEC è comunque qualcosa che andrebbe evitato, meglio correggerla per migliorare l'assemblaggio).

5.1.5 Inferenza di Aplotipi tramite Filogenesi Perfetta

Si estrae per le posizioni di interesse il genotipo, ricostruito dall'allineamento. A partire dal genotipo faccio inferenza per capire gli aplotipi. In questo contesto entra in gioco nuovamente la filogenesi perfetta.
Come detto "risolvere il genotipo" è un problema prettamente computazionale.
In input si prende un elenco di genotipi. Si ha quindi una matrice con i genotipi come righe e le posizioni degli SNPs come colonne. Si hanno tre valori nella matrice:

- 0 per (0,0)
- 1 per (1,1)
- * per (0,1) e (1,0)

Presa questa matrice si vuole costruire una matrice binaria di aplotipi che risolve i genotipi, ovvero si cerca le coppie di aplotipi che permettono il genotipo, e che ammette un albero di filogenesi. Le coppie (1,0) e (0,1) sono diverse ma entrambe rappresentate con * quindi possono vere più combinazioni di aplotipi che risolvono il genotipo.

Definizione 29. *Formalmente un genotipo g è risolto dalla coppia di aplotipi* $\langle h_1, h_2 \rangle$ *sse:*

$$\forall i, (g[i] = h_1[i] = h_2[i]) \vee (h_1[i] \neq h_2[i] \wedge g[i] = *)$$

le due parti dell'or distinguono quindi omozigoti ed eterozigoti.

Potrei avere comunque una matrice in input che causa * comporta l'impossibilità di avere filogenesi, avendo la matrice proibita che rende la matrice incoerente con l'evoluzione:

$$\begin{bmatrix} * & 1 \\ 1 & * \\ 1 & 1 \end{bmatrix}$$

Vediamo un caso buono, con 3 genotipi:

$$\begin{bmatrix} * & * \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Che posso risolvere con $h_1 = 0\ 0$ e $h_2 = 1\ 1$, come **unica soluzione**. Si ottiene quindi la matrice di filogenesi:

$$\begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

con 6 aplotipi per i 3 genotipi (i primi 2 aplotipi, h_1 e h_2, i secondi due e gli ultimi due, che sono uguali non avendo *).
Da questa posso costruite una matrice di filogenesi.
Se la soluzione è unica allora è valida. In assenza di ricombinazioni, che sono detti blocchi, si ha un modello coerente con la filogenesi perfetta.
Per ottenere la matrici di aplotipi da quella di genotipi in modo che si abbia un albero di filogenesi è un problema che si risolve in tempo lineare.

5.1.6 Formalizzazione Problemi Bioinformatici

Uno dei punti chiave della bioinformatica è impostare il problema dal punto di vista computazionale. Bisogna passare da un problema biologico ad uno algoritmico.

Parliamo sempre di aplotipizzazione, *haplotyping*, e genotipizzazione, *genotyping*.
I vertebrati in generale sono diploidi, avendo che si hanno due copie di ogni cromosoma che sono appunto gli aplotipi. Le differenze delle due copie sono comunque piccole e sono rappresentate dagli SNPs, avendo che in un certo locus si hanno due possibili nucleotidi, potendo quindi indicare la cosa in modo binario, 0 per l'allele di maggioranza e 1 per quello di minoranza.
Vediamo varie problematiche

Esempio 15. *Vediamo il problema dell'inferenza di aplotipi da genotipi, il* ***phasing****.*
Dati i genotipi di una popolazione vogliamo ottenere gli aplotipi della popolazione o di un singolo individuo.
In merito all'aplotipo diciamo che l'aplotipo h è un vettore binario su alfabeto $\{0,1\}$*, avendo quindi* $(h_1, \dots h_m) \in \{0,1\}^m$ *con* h_i *che è lo SNP del locus* i*. Nell'aplotipo non ho quindi tutto il cromosoma ma solo quel 10% di differenze circa nella popolazione, ho solo i locus con problemi, tralasciando le parti uguali dei cromosomi che si stanno confrontando.*
Il genotipo invece è un vettore $g = (g_1, \dots, g_m) \in \{0,1,2\}^m$*, avendo 0 per omozigosi sull'allele di maggioranza, 1 di minoranza e 2 per l'eterozigosi (nelle definizioni sopra si era usato * al posto di 2).*
Dato un vettore di genotipi presi da n individui G_I*, con associato ad ogni individuo il rispettivo genotipo (avendo ricostruito il suo DNA vedendo cosa succede negli SNPs, avendo quindi 0, 1, 2). Magari ho fatto la genotipizzazione degli SNP solo di un cromosoma e prendo un insieme di SNPs* $S = \{s_1, \dots, s_m\}$*.*
Rappresentiamo G_I *come una matrice. Si ha quindi:*

- ***input****: matrice* G_i *di genotipi*
- ***output****: matrice* M_H *di aplotipi che spiega* G_I

Non so quanti aplotipi si avranno quindi dico che saranno l*.*
Si vuole che:

$$\forall g_i \in G_I \ \ \exists (h_{i1}, h_{i2}) \ \textit{t.c. risolvono} \ g_i$$

G_i *ha quindi n righe con* g_i *che è la riga* i*.* M_H *ha* l *righe.*
Mi aspetto che $l \geq n$ *e si vuole il minimo numero di aplotipi distinti che spiega* G_I *e questo è un problema NP-complete. Ma sappiamo anche che* $l \geq 2n$ *e quindi* $n \leq l \leq 2n$*, potendo riutilizzare lo stesso aplotipo per più genotipi. Quest'ultima è la* ***regola di Clark****, che ha definito una regola iterativa per il phasing di un gruppo di genotipi. Vediamo il procedimento. Si*

hanno 3 genotipi, ad esempio:

$$g_1 = (0, 2, 0, 0)$$

$$g_2 = (2, 0, 0, 2)$$

$$g_3 = (1, 2, 2, 1)$$

e si supponga di conoscere già un aplotipo per il primo genotipo, ad esempio:

$$h_{11} = (0, 1, 0, 0)$$

e in automatico si ha anche l'altro aplotipo del primo genotipo:

$$h_{12} = (0, 0, 0, 0)$$

A questo punto vedo se questi aplotipi risolvono altri genotipi. Vedo quindi quali aplotipi noti risolvono già i genotipi, ottenendo in caso nuovi aplotipi. Avendo h_{12} so che posso applicarlo anche a g_2, ottenendo automaticamente il terzo mancante per poter risolvere g_2, insieme ad h_{12}, ovvero $h_{22} = (1, 0, 0, 1)$. Inoltre h_{22} può spiegare g_3 e posso ottenere anche il quarto mancante per risolvere g_3 insieme a h_{22}, ovvero $h_{32} = (1, 1, 1, 1)$.
Si sono ottenuti tutti gli aplotipi distinti:

1. $h_{11} = (0, 1, 0, 0)$
2. $h_{12} = (0, 0, 0, 0)$
3. $h_{22} = (1, 0, 0, 1)$
4. $h_{32} = (1, 1, 1, 1)$

Avendo che 4 aplotipi spiegano 3 genotipi, avendo $3 \leq 4 \leq 6$, avendo $n = 3$. La regola di Clark è esponenziale.
Si hanno altre regole per il phasing, come si è visto ad esempio con la filogenesi perfetta. Con la filogenesi perfetta, grazie alla soluzione di Gusfield nel 1994, si ha la seguente formulazione:

- ***input**: matrice G_I di n genotipi, con m SNPs (che sono le colonne), avendo quindi una matrice $n \times m$*
- ***output**: matrice di lunghezza $2n$ di aplotipi M_h tale che:*

  $$\forall g_i \in G_I \exists \ (h_{i1}, h_{i2}) \text{ t.c. risolvono } g_i,$$

  avendo il vincolo che M_H ammetta filogenesi perfetta (altrimenti restituisco errore). Si ha il vincolo che gli SNPs seguono il vincolo della filogenesi perfetta

*Il problema del phasing con filogenesi perfetta è polinomiale. All'inizio la soluzione quadratica (tramite **graph realization**) per poi arrivare ad una soluzione lineare.*
*Il problema del **graph realization** ha:*

- ***input**: una lista di insiemi su lettere (ciascuno con all'interno basi azotate)*
- ***output**: un albero etichettato con le lettere (nel nostro caso le basi azotate) tale che ciascun insieme in input rappresenti un cammino dell'albero*

Ad esempio con:

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

mi basterebbe un albero coi nodi in sequenza:

$$A \to C \to G \to T$$

Questo problema ha soluzione polinomiale.
*Per il phasing si usa anche la **legge di Mendel** quando si ha a disposizione il **pedigree**, avendo il **trio** (ovvero la rappresentazione di padre, madre e un figlio, coi rispettivi genotipi).*

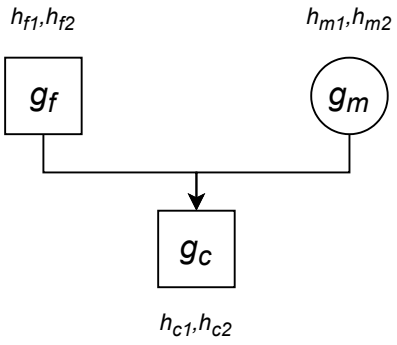


Figura 5.1: Esempio di trio, con quadrati per i maschi e cerchi per le femmine, si ha quindi un figlio maschio

Quindi:

- ***input**: il trio, g_f, g_m e g_c*
- ***output**: le tre coppie di aplotipi del trio calcolati con la legge di Mendel, escludendo però le ricombinazioni (altrimenti il problema diventa troppo complesso)*

Il problema di determinare gli SNP dal trio è importante per patologie ereditarie come la leucemia o comunque in altri discorsi in cui centra l'ereditarietà, anche per esempio per la selezione di specie d'allevamento etc...
Si hanno ad esempio:

$$g_f = (2, 2, 1, 1)$$

$$g_m = (1, 1, 0, 0)$$

$$g_c = (1, 1, 2, 2)$$

Vediamo quindi di inferire i genotipi.
Per il padre ho:

$$h_{f1} = (0, 1, 1, 1)$$

$$h_{f2} = (1, 0, 1, 1)$$

Per la madre:

$$h_{m1} = (1, 1, 0, 0)$$

$$h_{m2} = (1, 1, 0, 0)$$

Il figlio e vedo che $g_c = (1, 1, 2, 2)$ *comporta:*

$$h_{c1} = (1, 1, 1, 1)$$

$$h_{c2} = (1, 1, 0, 0)$$

Ma questo non è coerente col padre, nel dettaglio non è coerente h_{c1}*. Ma per il padre potrei avere anche:*

$$h_{f1} = (1, 1, 1, 1)$$

$$h_{f2} = (0, 0, 1, 1)$$

e in questo caso:

$$h_{c1} = (1, 1, 1, 1)$$

$$h_{c2} = (1, 1, 0, 0)$$

sono consistenti.
La soluzione è quindi unica *e il problema ha soluzione in tempo polinomiale. Questo problema, quello di assegnare i valori agli aplotipi sulla base dei genotipi, posso vederlo come il problema di* ***2SAT****, che è appunto polinomiale e per il quale si hanno risolutori veloci. Quindi il phasing, in assenza di ricombinazioni è polinomiale.*
In alternativa potrei usare la ***programmazione lineare/intera*** *(come in ricerca operativa).*

Esempio 16. *Vediamo il problema dell'assemblaggio, ovvero come ricostruire gli aplotipi. È il problema del* ***fragment haplotype assembly****.*
Dal punto di vista biologico si ha appunto il problema di assemblare aplotipi da dati di sequenziamento, ovvero dalle read. Per semplicità si parte da ***read allineate****. Si ha quindi il* ***reference****, espresso da una sequenza, con gli alleli di maggioranza in tutta la popolazione umana, più varianti,gli SNP della popolazione, in un file VCF. Un esempio di reference è l'****NA128****, depositato nelle varie banche dati. L'obiettivo sarebbe arrivare ad un* ***gold standard****, ovvero una sequenza che rappresenterà tutti gli alleli da maggioranza con in aggiunta tutte le variazioni possibili della popolazione (per ora si sta studiando la popolazione degli Askenasi, con il progetto GIAB). L'idea è quindi di avere una sequenza più le variazioni, per un dato indice i, della popolazione. Si prende l'individuo singolo I con le read relative e tali read vengono allineate al reference, per cui* ***read allineate****, estraendo i frammenti allineati degli SNPs. Si produce quindi, da ogni read, una riga della matrice dei frammenti, con 1 e 0 in presenza di SNPs per dire se l'allele è di maggioranza o minoranza.*
Si hanno quindi:

- ***input****: dati $F = \{f_1, \ldots, f_n\}$ frammenti per m SNPs si ha in input quindi la SNP matrix, $n \times m$, su alfabeto $\{0, 1, -\}$*
- ***output****: due aplotipi consistenti con i frammenti*

Ad esempio, presi 4 frammenti, potrei avere come input:

$$\begin{bmatrix} - & 0 & 1 & 1 & - \\ 1 & 0 & 0 & - & - \\ 1 & 0 & 1 & - & - \\ - & 0 & 1 & 1 & - \end{bmatrix}$$

Si parla prima di un altro problema, ovvero quello di ***aplotipizzazione di un singolo individuo****, detta* ***error free****, avendo:*

- ***input****: matrice M_F di frammenti senza errori, error-free, $n \times m$*
- ***output****: partizione di F in due insiemi H_1 e H_2, che rappresentano i due aplotipi, tale che il* ***conflitto tra frammenti*** *esiste solo tra due aplotipi differenti. All'interno di H_1 non ho conflitti tra i frammenti, idem all'interno di H_2*

Si definisce che f e f' sono in conflitto se $f \in H_1$ e $f' \in H_2$ e quindi sse $\exists i,\ 1 \leq i \leq n$ tale che:

$$f[i] \neq f'[i]$$

quindi uno è 0 e l'altro è 1. Basta quindi una sola posizione.
Preso l'esempio sopra, con in ordine f_1, f_2, f_3 e f_4, con matrice error-free essendo tutto consistente:

$$\begin{bmatrix} - & 0 & 1 & 1 & - \\ 1 & 0 & 0 & - & - \\ 1 & 0 & 1 & - & - \\ - & 0 & 1 & 1 & - \end{bmatrix}$$

so che sono in conflitto:

$$(f_1, f_2), (f_2, f_3), (f_2, f_4)$$

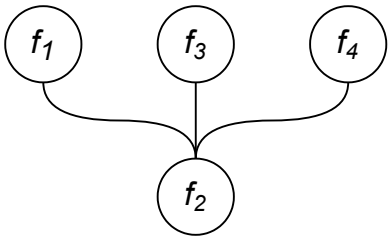
Avendo quindi:

$$H_1 = \{f_1, f_3, f_4\}$$
$$H_2 = \{f_2\}$$

Dobbiamo rappresentare il conflitto, nel caso error-free, riducendolo ad un problema su grafo. Tale grafo ha $V = F$, quindi come vertici i frammenti, e come archi, non orientati, si ha che:

$$(f, f') \in E \iff f \text{ in conflitto con } f'$$

Si crea quindi un grafo bipartito G_F. Il problema si riduce quindi a capire se un grafo è bipartito.
Per l'esempio si avrebbe G_F:



Decidere che è un grafo è bipartito mi genera H_1 e H_2 e ci sono già algoritmi polinomiali in merito. Un grafo per non essere bipartito basta verificare che non ci siano cicli semplici dispari.
Aggiungiamo ora gli errori e quindi bisogna applicare la correzione nella matrice dei frammenti. Si usa ***K-MEC*** *che è prevede il minimo numero di correzioni per colonna facendo al più K correzioni per colonna nella matrice dei frammenti tale che esista H_1, H_2 come bipartizione.*
Vediamo un esempio:

$$\begin{bmatrix} - & 1 & 0 & 0 & - \\ - & 0 & 1 & 1 & 1 \\ - & 0 & 0 & 0 & - \end{bmatrix}$$

Avendo i conflitti:

$$\{(f_1, f_2), (f_1, f_3), (f_2, f_3)\}$$

non potendo bipartire. Applico K-MEC e modifico la terza riga della seconda colonna:

$$\begin{bmatrix} - & 1 & 0 & 0 & - \\ - & 0 & 1 & 1 & 1 \\ - & \mathbf{1} & 0 & 0 & - \end{bmatrix}$$

facendo quindi una sola correzione, ottenendo una matrice su cui poter fare bipartizione.
Il problema K-MEC (Minimum Error Correction) si risolve con programmazione dinamica.
K-MEC contiene la sottostruttura ottima, ovvero presa una matrice con colonne da 1 a $i + 1$ se rimuovo l'ultima colonna ho comunque la soluzione ottima.
Si ha quindi:

- ***input**: M_F matrice dei frammenti non error-free e $k \geq n$ di correzioni per colonna massime*
- ***output**: matrice M'_F ottenuta da M_F tramite il minimo numero di flip (con al più k correzioni per colonna) tale che sia error-free, ammettendo quindi bipartizione*

Si ha una soluzione parametrica.

Capitolo 6

Indexing dei k-mer

Una questione essenziale in bioinformatica è la mole di dati.
Per tenere traccia dei dati si fa un *fully-text-index* del testo T, avendo una struttura dati atta a tenere in memoria il testo. Su questa struttura si fanno query, in merito a posizioni o frequenza di sottostringhe. Le query devono essere in tempo sub-lineare rispetto a $n = |T|$. Ad esempio calcolare la BWT può essere dispendioso ma le query poi sono sulla dimensione del pattern. Alcune query sono addirittura in tempo costante.
Un altro problema è il tradeoff tra tempo e spazio, cercando una struttura performante in entrambi i sensi. Si parla di **next generation computing**.
Sulle strutture dati è comodo avere le due operazioni di *rank* e *select*, che per la BWT sono C e Occ.
Un tecnica per accedere in tempo costante al testo è l'hashing, tramite **funzione di hash**. Si ha un numero arbitrario di chiavi k_i. Le chiavi possono essere di vario tipo ma sono tendenzialmente numeriche. Si ha poi una funzione h che mappa le chiavi negli ingressi di una tabella, tramite $h(k_i)$. Tale tabella hash è a dimensione fissata. La funzione di hash modifica la chiave per poter accedere in modo ottimizzato alla tabella. Qualora con due chiavi si abbia il tentativo di accesso alla stessa cella della tabella si parla di **collisione** e questo è un problema normalmente. In realtà le collisioni sono un vantaggio per grandi moli di dati, trovando somiglianza di oggetti qualora collidano nella stessa posizione e questo è d'aiuto in bioinformatica (dove il processamento è sulla scala di 10^{13}, in clustering di genomi simili), ma anche in ambiti di studio di immagini. Questo sistema di hashing è detto **local sensitive hashing (*LSH*)**.
Si usano i **q-gram** che sono sottostringhe di di lunghezza q, sono infatti detti anche **q-mers**. Si sfrutta LSH per ridurre la dimensione del dato, usando collisioni per capire la somiglianza, in termini di distanza di Jaccard/Hamming/Edit, processando i **bucket**, ovvero i contenitori dove finiscono elementi

con lo stesso hash. Si hanno trilioni di buckets di cui hashare gli elementi. Oggetti con lo stesso hash sono molto probabilmente simile e quindi serve una funzione di hash che sfrutti questa cosa, in un tempo accettabile. Non confronto quindi direttamente gli elementi. Il singolo costo di hash su un elemento è **costante** quindi il tutto è lineare sul numero di elementi.
Le tipiche query sono cercare stringhe simili ad una di query q oppure fare clustering dei vari elementi.
Per fare LSH si riassumono le sequenze, o alcune parte, nei cosiddetti **sketch**, che sono rappresentazioni più piccoli. Gli sketch sono usate come chiavi di hash e si confrontano le sequenze tramite appunto le collisioni.
La tecnica che usa Jaccard è detta MinHash.
La distanza tra due stringhe è detta d mentre la somiglianza $1 - d$. La distanza di Jaccard tra due insiemi A e B è:

$$J(A, B) = \frac{A \cap B}{a \cup B}$$

quindi se A e B sono uguali $J(A, B) = 1$.
Una famiglia di funzioni hash è detta **locality sensitive** se elementi simili hanno probabilità alta di portare ad una collisione, avendo la stessa funzione di hash.

Definizione 30. *LSH per una relazione $s()$ di simiglianza è una famiglia D di funzioni hash tale che:*

$$P_{h \in F}[h(x] = h(y) = s(x, y)$$

per ogni coppia x, y nel dominio di F.
Nel dettaglio $s(x, y)$ viene stimata tramite gli sketch di x e y:

$$sketch(x) =< h_1(x) \dots h_t(x)$$

$$sketch(y) =< h_1(y) \dots h_t(y)$$

con t funzioni hash. SI ha quindi:

$$s(x, y) = \frac{numero\ di\ volte\ che\ su\ ha\ h_i(x) = h_i(y)}{t}$$

con il numero di collisioni che indica appunto la somiglianza.
Se uso minHash LSH, che sua Jaccard, ho che:

$$P[h(A) = h(B)] = J(A, B)$$

Scegliendo a caso la funzioni di hash tramite permutazione π_i si ha che:

$$h_{\pi_i}(x) = \min \pi_i(x)$$

Manca frase da slide

Le permutazioni dell'elenco dei kmer sono le permutazioni di posizioni di una matrice caratteristiche dell'insieme che contiene i k-mer. **Esempio su slide per il calcolo della funzione di hash (probabilmente cannati i conti finali)**.
Dimostrazioni non fatte a lezione.
Si riesce quindi a confrontare le sequenze tramite hash sui kmer basandosi su un certo numero di permutazioni scelte. Rappresento quindi un insieme di kmer tramite un sottoinsieme di kmer. Non sempre la somiglianza è comunque indicativa e la cosa varia sul k e sull'utilizzo degli stessi.
Tutto questo, lato genomico, è stato studiato nel software **MASH**, che studia il metagenoma. MASH si basa sul prendere genomi da confrontare a coppie, si estraggono i kmer e si estraggono i kmer canonici, ovvero quello più piccoli lessicograficamente per le due direzioni. Si fissa una dimensioni dello sketch e si calcolano i più piccoli hashes su tutti i kmer. La comprensione è incredibile, da centinaia di giga a meno di un centinaio di mega. Si fa confronto quadratico sugli sketch. Anche in questo caso $k \sim 32$. MASH viene anche usato per fare assemblaggio di reads, sempre usando gli sketch.
Questi risultati sono interessanti sia lato tempo che lato spazio.

Capitolo 7

Bloom Filters

Un **bloom filter** è una seconda variante di struttura dati basata su hash. Volendo solo sapere se un elemento ci sia o meno posso in primis togliere i valori dalla tabella di hash, tanto se c'è ho il mapping. Ma posso anche eliminare le chiavi, mi basta sapere che ci sia la funzione di hashing che, non interessandomi di recuperare il valore effettivo. Mi riduco quindi ad un vettore binario, con 1 in corrispondenza di dove prima aveva dei valori hash, con eventuali collisioni (sempre prima), e 0 altrimenti. Quindi con 1 flaggo i bucket dove finirebbe almeno un elemento. La query di esistenza diventa quindi solo verificare che ci sia 1 nella posizione calcolata dall'hash. Un bit vector inoltre è rappresentabile facilmente in memoria, soprattutto se sparso. Il problema è che posso avere falsi positivi se ho collisioni (magari due elementi puntano ad una posizione, ne vedo uno e metto 1 ma faccio la query sull'altro) ma se ho 0 sicuro non c'è. Una soluzione è usare più funzioni di hash per abbassare le chance di falsi positivi, devo avere 1 per tutti i vari **filtri**, i vari bit vector sono detti filtri. Ovviamente la query non da una risposta certa ma altamente probabile (se tutti i filtri alla posizione x hanno valore 1). Aumentare le funzioni di hash riduce le collisioni. Possono inoltre fare un bit vector come and di tutti gli altri e usare quello per fare le query, anche se resta una stima probabilistica. Il **bloom filter** è appunto questo bit vector finale (questa cosa non è chiara).
In bioinformatica i bloom filter sono un modo efficiente per memorizzare i k-mer e farci query sopra per verificare se esistono o meno. Minhash più bloom filter più skatch sono un'ottima soluzione. Viene usato ora per memorizzare i VCF del COVID.
Su slide vari tool con bloom filter e alcuni pseudocodici.

7.1 Conteggio dei K-mer

Un altro problema interessante è il conteggio dei kmer:

- **input:** un insieme di T di testi e un intero k
- **output:** un insieme di coppie $(s, Occ(s))$, con s k-mer e $Occ(s)$ numero di occorrenze del kmer

Questo problema è utile in fase di error correction, de novo assembly, repeat detenction e stima del coverage in fase di sequenziamento.
È inoltre frequente che in cellule tumorali il conteggio dei cromosomi è sballato, con il **whole genome duplication**.
Su slide lista di tool per il conteggio, tra cui KMC2 (anche se ormai siamo a KMC3).
Questo problema è semplice concettualmente ma ha seri limiti prestazionali. Tali software usano spesso il parallelismo, tabelle hash, bloom filters, e compressione. **Su slide spiegazione su KMC2**.

Capitolo 8

Succint De Bruijn Graph

Si vede ora una rappresentazione efficiente dei grafi di De Bruijn tramite BWT, ovvero in modo succinto.
I DBG sono, come sappiamo, costruiti su un insieme di k-mer, coi nodi che sono prefisso e suffisso dei vari kmer, di lunghezza $k-1$.
Si ha quindi:

- **input**: un insieme S di sequenze, da cui si estraggono i k-mer
- **output**: il DBG

Se aggiungiamo davanti ad ogni stringa un numero di dollari (che è sempre il simbolo più piccolo in ordine lessicografico) pari a $|S|$ possiamo procedere con l'indicizzazione della struttura. Avremo quindi potenzialmente un nodo con soli dollari che è il più piccolo. Potrei però a causa del k non avere un nodo di soli dollari. Usiamo nodi univo. Gli archi uscenti da un nodo mi danno informazione sui k-mer successivi. Vogliamo quindi un modo per sfruttare la cosa per rappresentare il grafo in modo succinto in memoria. Useremo quindi la BWT che permette anche di navigare nel grafo. L'idea è quindi di "decomprimere" solo la parte di grafo che mi interessa.
Si procede ordinando i (k-1)-mer, dei nodi, però in ordine rovesciato, da destra a sinistra. Questa fase si fa in costruzione, durante la quale può servire più memoria di quella che poi serve per lo studio del grafo a struttura completata. Si da per assunto che i k-mer sono stati già estratti. La BWT questa volta rappresenta gli archi uscenti dai nodi (che possono essere più di uno e in tal caso vengono messi in ordine lessicografico), riordinati appunto in ordine lessicografico da destra a sinistra. Se un nodo non ha archi uscenti metto il dollaro. La BWT è quindi lunga almeno quanto $|V|$ ma probabilmente di più, essendo lungo $|E|$. Tengo in memoria solo la BWT, che occupa molto meno dei k-mer. Posso anche comprimerla. La BWT, che chiamo w, ha tutte

le caratteristiche per ricostruire il grafo a partire dal nodo iniziale (quello di soli dollari, per cui serve un k adeguato).
Vediamo quindi come navigare il grafo a partire da w. Si sfrutta la solita LF-function. Dato un nodo e l'etichetta dell'arco uscente si capisce dove si trova il nodo successivo, che sappiamo quale sia come etichetta (aggiungendo l'etichetta dell'arco al nodo) ma dobbiamo capire dove sia sulla BWT. Si usano quindi C e Occ dell'FM-index sul vettore w, sulla BWT. Dal nodo di tutti dollari posso far partire la navigazione tramite w. Serve anche un vettore binario per capire se un nodo ha più archi uscenti, lungo quando w. Questo vettore contiene 0 dove si ha un nodo che esce con più nodi, e si ha 0 per tutti questi archi uscenti (che trovo etichettai in corrispondenza sulla BWT) tranne l'ultimo, che ha 1. Se si ha un solo nodo uscente metto 1. Questo vettore binario lo si può chiamare *last*, tale che $last[i] = 1$ se i rappresenta in w l'ultimo arco uscente, altrimenti metto 0.
In memoria ho solo bwt e last, per di più compressi. Questa struttura dati succinta per i DBG è chiamata **BOSS**. Quindi dato un simbolo σ si rank i cerco il corrispondente simbolo σ nell'ordinamento "inverso" per capire il prossimo nodo. Si fa quindi il seguente conto partendo da un simbolo di w per capire, sempre in ottica di w, a che indice passare:

$$j = C(\sigma) + Occ(\sigma, i)$$

Quindi l'ordine delle stringhe è rovesciato perché poi dobbiamo basarci sull'ultimo carattere del nodo a cui si arriva, per questo sono da destra a sinistra, avendo nella BWT il simbolo che precede la stringa che voglio ottenere. Si permette quindi di muoversi in modo forward sul grafo, tramite LF-mapping. Con BOSS si è passati anche da 500gb ad una trentina.
Esempi su slide (senza è tutto un poco confuso).

Capitolo 9

I dati in Bioinformatica

Dobbiamo introdurre i dati fondamentali della bioinformatica, ovvero **DNA** (figura 9.1) e **RNA**, intese come molecole biologiche che vengono trattate come stringhe.
Il **DNA (*acido deossiribonucleico*)** è una molecola composta da nucleotidi, che è formato da:

- il deossiribosio, uno zucchero, **D**
- un gruppo fosfato, **p**
- una base azotata (Adenina, Citosina, Guanina, Timina). Citosina e Timina sono dette **piramidine**, le altre due **purine**

Si ha la cosiddetta **direzione 5'3'** per leggere le sequenze del DNA (avendo la direzione posso leggere in sequenza le basi, solitamente è dal basso all'alto).
I nucleotidi si legano tramite **legame fosfodiesterico tra D e P**.
L'unità di lunghezza di un molecola di DNA è il **base pair (*bp*)** (base pair in quanto il DNA è in se un'accoppiamento di basi a doppia elica), ovvero il numero delle basi.
Il **genoma** è la lunga molecola di DNA contenuta in ogni nucleo cellulare. Nel 1953 Watson e Crick hanno scoperto essere a doppia elica. Il genoma è frammentato in **cromosomi** (22 autosomi e i due cromosomi X e Y). Il più lungo cromosoma è il primo e il più corto è il ventiduesimo.
Tra le due catene di un genoma si ha che la Timina si appaia all'Adenina (e viceversa) mentre la Citosina alla Guanina (e viceversa), questa è la **regola delle basi complementari**. Se una catena ha la direzione 5'3' dall'alto verso il basso l'altra lo ha dal basso all'alto (ma sempre da 5' a 3') e viceversa. Si legge sempre in direzione 5'3' e si hanno così le **due sequenze primarie appaiate**, una detta **forward strand (+,1,+1)** e una detta **reverse**

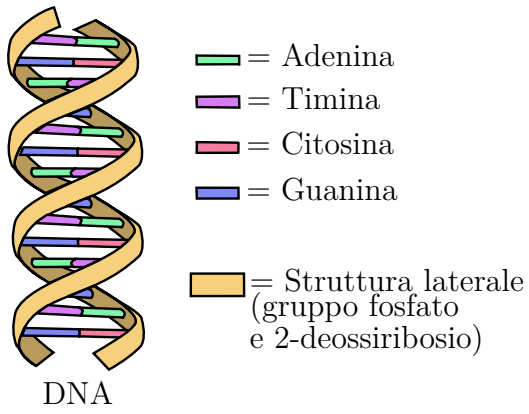


Figura 9.1: Rappresentazione grafica del DNA tratta da `https://it.wikipedia.org/wiki/DNA`

strand (-,-1).
Tra Adenina e Timina abbiamo due legami idrogeno mentre tra Citosina e Guanina se ne hanno tre (si richiede quindi più energia per eventualmente separare).
Si ha quindi a che fare con un alfabeto $\Sigma = \{a, c, g, t\}$ o $\Sigma = \{A, C, G, T\}$ per costruire le sequenze. Data la sequenza primaria di una delle due catene del DNA genomico, la sequenza primaria della catena appaiata è ottenuta per mezzo di un'**operazione di reverse&complement**, ottenuta sostituendo alla catena primaria ribaltata ogni base con la complementare (potrei anche prima sostituire e poi invertire). Se prendo entrambe le stringhe le chiamo **paired strands**.
Aggiungiamo altre definizioni:

- una regione di DNA genomico che ha una certa funzione prende il nome di **locus**
- la sequenza primaria di un locus è chiamata **sequenza genomica**, che di fatto è una sottostringa del genoma di un organismo

Per l'**RNA (*acido ribonucleico*)** si ha una composizione di nucleotidi del tipo:

- il ribosio, uno zucchero, **D**
- un gruppo fosfato, **p**
- una base azotata (Adenina, Citosina, Guanina, Uracile)

simbolo	sigla	nome	simbolo	sigla	nome
A	Ala	Alanina	M	Met	Metionina
C	Cys	Cisteina	N	Asn	Asparagina
D	Asp	Acido aspartico	P	Pro	Prolina
E	Glu	Acido glutammico	Q	GIn	Glutammina
F	Phe	Fenilalanina	R	Arg	Arginina
G	Gly	Glicina	S	Ser	Serina
H	His	Istidina	T	Thr	Treonina
I	Ile	Isoleucina	V	Val	Valina
K	Lys	Lisina	W	Trp	Triptofano
L	Leu	Leucina	Y	Tyr	Tirosina

Tabella 9.1: Tabella con l'elenco degli amminoacidi

Anche per l'RNA si ha la **direzione 5'3'** e si ha un alfabeto $\Sigma = \{a, c, g, u\}$ o $\Sigma = \{A, C, G, U\}$. La vera differenza è che l'RNA è sempre in **singola catena**.
Una **proteina** p una catena di amminoacidi e la sua sequenza primaria è una stringa costruita su un alfabeto di 20 simboli che rappresentano i 20 **amminoacidi** presenti in natura. I **geni** è il "responsabili" della produzione di proteine, che poi "regolano" la vita. Ogni proteina è prodotta da un gene e un gene genera più proteine. **I geni sono il 10% del genoma**, quindi solo una piccola parte. Il gene è quindi un locus del genoma che esprime una proteina. La sequenza primaria del locus di un gene è la sequenza genomica del gene. Ogni gene ha un identificativo detto **HUGO NAME**, con una precisa nomenclatura che è "circa" un acronimo. Un gene può appartenere al genoma di diversi organismi.
Sia il forward strand che il reverse strand contengono geni e gli uni non c'entrano con gli altri (figura 9.2). L'uomo ha circa 20000 geni codificanti (più i cosiddetti **geni non codificanti** che hanno altri scopi), ciascuno che codifica una o più proteine. Ogni cellula contiene l'intero set di geni che regolano la vita dell'organismo, ma nelle varie cellule si **esprime** solo un sotto-set di geni mentre gli altri rimangono **inattivi**, è il **profilo di espressione**. Può variare anche la quantità di proteina espressa. In due cellule posso esprimere lo stesso gene che però porta a diverse proteine.
Si ha quindi:

1. il locus viene trascritto (con anche lo splicing) nell'm-RNA detto trascritto

2. il trascritto porta alla proteina

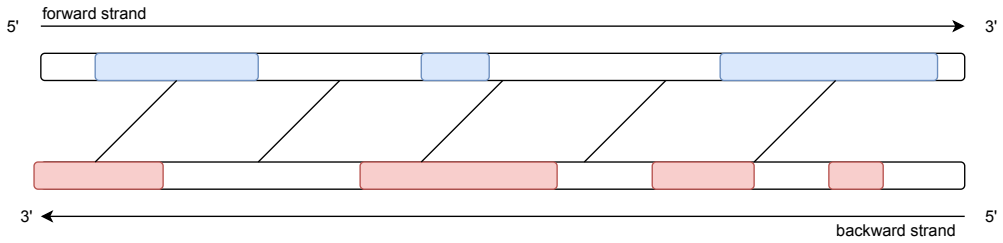


Figura 9.2: Rappresentazione stilizzata dei due strand con in azzurro e rosso, rispettivamente, i geni sul forward strand e sul backward strand

Per determinare la sequenza primaria di DNA e RNA usiamo il sequenziamento, producendo varie read e procedendo poi alla ricostruzione **in-silico**. Posso avere due tipi di output dal sequenziamento:

- **single-end reads**, che è un'unica sequenza
- **paired-end reads** e **mate-pair reads**, dove si parla di coppie di sequenze che sappiamo circa quanto distavano sulla sequenza originale

Per il sequenziamento ci sono stati vari step:

- **first generation**, per il **metodo Sanger** nel 1975. Si hanno:
  - read lunghe fino a 1000bp
  - elevata qualità
  - bassa coverage
  - costi elevati
- **second generation**, con le **Next-Generation-Sequencing (*NGS*) technologies**, dagli inizi degli anni 2000. Si hanno:
  - read corte, da 100bp a 400bp
  - alta coverage
  - bassi costi

  Tra le tecnologie *NGS* abbiamo:
  - Illumina (Solexa) con:
    - HiSeq System

        - Genome analyzer lix
        - **MySeq**, che produce read di 300bp, con il 99.90% di accuratezza, producendo 25 milioni di read per run
    - Ion Torrent-Life technologies con:
        - **Personal Genome Machine (*PGM*)**, che produce read di 200bp, con il 99.00% di accuratezza, producendo 11 milioni di read per run
        - Proton

  Per indicare i dati prodotti diciamo **NGS data**

- **third generation**, con le **Next Next-Generation-Sequencing technologies**, più avanti negli anni 2000. Si hanno:
    - read lunghe fino a 10000bp
    - qualità relativamente bassa

  Tra le tecnologie *Next-Next* abbiamo:

    - Pacific Biosciences con PacBio RS
    - Oxford Nanopore Technologies con:
        - GridION System
        - MinION

9.1 Formato FASTA

Vediamo il formato standard per sequenze primarie nucleotidiche:

- **FASTA** per le sequenze ottenute al metodo Sanger, quindi con pochi errori
- **FASTQ** per le sequenze ottenute con NGS. È un'evoluzione del FASTA in cui si associa alle sequenze dei reads un valore di qualità, uno per base

Il FASTA ha le seguenti caratteristiche:

- è un plain text

- ha la sequenza primaria (ma anche sequenze di amminoacidi) più eventuali informazioni extra a scelta. Si ha:
  - un header introdotto da > con tutte le informazioni extra in una sola riga (tra cui, volendo, cromosoma di partenza, indici di inizio e fine per specificare la porzione, ricordando che le posizioni sono 1-based, la "release", ad esempio tramite la sigla *GRC*, lo strand tramite +1 e -1 etc...)
  - la sequenza genomica separata in righe di 60 o 80 bp (quindi dopo 60 o 80 simboli vado a capo)
- è stato pensato come formato di input del software FASTA per l'allineamento
- ha come estensione `.fa` o `.fasta`

9.1.1 ENSEMBL

ENSEMBL è il **genome browser** attualmente più usato. Un genome browser è un database con associata un'interfaccia di esplorazione. È un progetto partito nel 1999 da **EMBL (*European Molecular Biology Laboratory*)** e dal **Wellcome Trust Sanger Center**. Dal 2017 fa capo a **EMBL-EBI (*European Bioinformatics Institute*)**, sempre di **EMBL**. Si hanno i dati di 250 specie di cordati, tra cui uomo, topo, ratto e zebrafish. Si hanno vari livelli di informazione (ovvero di annotazioni), tra cui; genoma, gene, proteina etc...
I genome browser sono consistenti rispetto a progetti di ricerca e analisi genomiche. SI può accedere, tra le altre cose a:

- la sequenza del genoma di una data specie o dei singoli cromosomi
- le sequenze dei geni annotati su di un genoma e i relativi trascritti espressi
- le variazioni annotate rispetto al genoma di riferimento che caratterizzano i singoli individui

Le coordinate sono sempre rispetto alla catena forward.
ENSEMBL si può raggiungere tramite `www.ensembl.org`.
Si ha anche il formato **EMBL** stesso, pensato originariamente per memorizzare sequenze nucleotidiche in banche dati che non erano db relazionali ed erano senza accesso. Il file poteva essere scaricato tramite un ID e basta (era

poi scopo del biologo e dell'informatico riutilizzarlo). Il file EMBL è pieno di informazioni, con in fondo la sequenza nucleotidica. Banche dati così ci sono ancora, come l'**ENA (*European Necleotide Archive*)**.

9.2 Qualità dei Dati e FASTQ

Valutiamo quindi la **qualità del dato di sequenziamento** e il formato standard **FASTQ**, il formato di output dei sequenziatori di nuova generazione.
Le NGS producono una gran quantità di frammenti, piuttosto corti, per i quali è essenziale conoscere la qualità di ogni base in fase di processamento della read, per eliminare/modificare le parti problematiche. Per ogni base di una read NGS si ha quindi un *indice di qualità*, detto **Phred Quality Score**.
Phred, abbreviato, è stato creato per il *base caller Phred*, che sfrutta il cromatogramma che valutare la qualità.

Definizione 31. *Definiamo **Phred Quality Score** q tramite la formula:*

$$q = -10 \log_{10} p$$

con:

- *base b*
- *p probabilità che b sia errata*

il valore q viene arrotondato all'intero più vicino.
Chiedere una base corretta al 100% comporterebbe $q = \infty$ (avendo $p = 0$) e quindi si considera corretta una base con $q \geq 50$ (che comporterebbe probabilità che sia errata pari a $p = 0.00001$, quindi 0.001%).
Una base con $30 \leq q \leq 50$ è solitamente considerata buona ma dipende dal contesto. Avere $q = 30$ hanno probabilità delle 0.1% di essere errate.

Esempi su slide.
Il formato **FASTQ** è standard per i sequenziatori NGS, che quindi non producono un **FASTA**.
SI ha che è:

- plain text
- è stato sviluppato per associare ad una sequenza il Phred Quality Score

- ha estensione `.fq` o `,fastq`

Si hanno 4 righe, dette record:

- un header con l'identificatore. Tale riga ha "@" come simbolo iniziale. Non si hanno informazioni extra come per il FASTA
- sequenza delle basi della read in una riga
- un header dei phred values. Tale riga inizia con il simbolo "+"
- la sequenza dei valori di qualità, una per ogni base della read

Esempio 17. *Vediamo un esempio:*

```
@HWUSI-EAS522:8:5:662:692#0/1
TATGGAGGCCCAACTTCTTGTATTCACAGGTTCTGC
+HWUSI-EAS522:8:5:662:692#0/1
aaaa`aa`aa`]__`aa`_U[_a`^\\UTWZ`X^QX
```

con:

- *l'header della sequenza un identificatore tipico di illumina*
- *le basi su una riga*
- *l'header, con lo stesso identificatore della sequenza (è opzionale, potrei non averlo), dei phred*
- *la sequenza dei phred values, con una codifica che permette di associare facilmente il valore di qualità. Ogni valore è in corrispondenza di indice con la base di cui rappresenta la qualità (se usassi gli interi avrei magari un intero di due simboli che renderebbe impossibile capire a che base si riferisce)*

I valori interi di qualità vengono quindi convertiti in un certo carattere e ogni sequenziatore ha la sua funzione per convertire l'intero rappresentante la qualità q in char:

$$c = f(q)$$

Ad esempio per Illumina è (con $ASCII(X)$ che converte in char l'intero):

$$c = ASCII(\min\{q, 93\} + 33)$$

Esempi su slide.
Conoscere la qualità delle basi permette di fare **trimming** che, fissata una soglia minima di qualità (decisa di volta in volta), consiste in:

- trovare la più lunga sottostringa della read composta da basi con qualità superiore a questa soglia minima
- sostituire tale sottostringa all'intero read sse questa sottostringa è lunga a sufficienza (lunghezza anch'essa decisa di volta in volta). Qualora la sottostringa sia troppo corta si elimina la read. Su milioni di read sacrificarne qualcuna non è un problema particolare e non solo, si migliorano le analisi successive

9.3 Espressione di un Gene e GTF

Ricordiamo che un gene è un locus che codifica una proteina. La sequenza primaria del locus di DNA del gene è detta **sequenza genomica del gene** (e si ricorda che entrambi gli strand contengono geni, negli esempi si studia comunque il forward essendo già da "sinistra a destra").
I geni si identificano tramite il cosiddetto **HUGO NAME**.
I geni sono formati da **esoni**, regioni codificanti, e **introni**, regioni non codificanti. Il confine tra un esone a sinistra e un introne a destra è detto **sito di slicing al 5'**, detto anche **donor splice site**. Se ho un introne a sinistra e un esone a destra ho un **sito di slicing al 3'**, detto anche **acceptor splice site**. Nell'uomo il 99.24% degli introni iniziano con GT e finiscono con AG e si parla di **introne canonico**. Si hanno anche introni non canonici.
Vediamo gli step di sintesi proteica in modo "operativo":

- si ha la **trascrizione** dove l'intero locus del gene viene copiato in una molecola di RNA (convertendo T in U), detta **pre-mRNA**, lunga quanto il locus del gene
- avviene quindi lo **splicing**, dove si eliminano gli intorni dal pre-mRNA producendo l'**mRNA**, che è una sequenza continua di esoni, di regioni codificanti, detta anche **trascritto**
- il trascritto viene tradotto in proteina. In realtà all'interno del trascritto si ha una parte centrale detta **coding sequence (*CDS*)** che è la parte che da origine alla traduzione in proteina. La CDS inizia sempre con la **tripletta di inizio** AUG e finisce con una tra le seguenti **triplette di fine** UAG, UAA o UGA. La lunghezza della CDS è sempre un multiplo di tre per cui la si può vedere come una sequenza di triplette dette **codoni** (quindi si hanno il **codone di inizio** e **codone di fine** anziché chiamarle triplette). Le due parti a monte e valle della CDS sono dette **5'UTR** (la parte prima) e **3'UTR** (la parte dopo)

- dalla sequenza di codoni si passa alla sequenza di amminoacidi e quindi alla proteina. Questo passaggio è facilmente studiabile tramite il **codice genetico** (dove per comodità l'uracile è spesso segnato segnato con la timina, questa è una cosa standard nelle banche dati in quanto permette confronti diretti con il genoma). Ogni codone corrisponde ad un amminoacido. Il codice genetico è **degenere** quindi più codoni mappano lo stesso amminoacido. Si nota che AUG è la metionina (e solo AUG lo codifica) che è l'amminoacido che inizia ogni proteina. I codoni di fine hanno associato solo un "segnale di stop", non producendo alcun amminoacido, che ferma la traduzione

Il sequenziamento che produce read di RNA, se di tipo NGS, produce read dette **RNA-seq read**, se invece si usa Sanger si chiamano **Expressed Sequence Tag (*EST*)**, che sono ormai in disuso.
Come si era già detto 20000 geni producono centinaia di migliaia di proteine e quindi non si ha una corrispondenza "1:1". Un gene può esprimere una molteplicità di proteine. Questo accade in quanto si ha lo **splicing alternativo**, dove un gene è in grado di combinare i suoi esoni in modo diversi. Si definisce **isoforma** un particolare trascritto che un gene esprime in virtù dello splicing alternativo. Lo splicing alternativo è importante in quanto:

- è **tessuto-specifico** quindi un gene esprime trascritti diversi in tessuti diversi
- dipende dalle condizioni della cellula
- è fortemente legato alle malattie (studiare le differenze tra la sintesi di cellule sane e malate è fondamentale per trovare terapie mirate)

Si descrivono i vari eventi di splicing alternativo:

- l'**exon skipping**, ovvero *salto dell'esone*, dove un esone (o anche più esoni) può essere escluso dal trascritto primario
- l'**alternative acceptor site**, ovvero *sito di taglio alternativo* $3'$, detto anche **3' competing site**, dove una parte del secondo esone può essere considerata non codificante o, alternativamente, una porzione dell'introne adiacente può essere considerata codificante
- l'**alternative donor site**, ovvero *sito di taglio alternativo* $5'$, detto anche **5' competing site**, dove una parte del primo esone viene considerata non codificante o, alternativamente, una porzione di introne adiacente può essere considerata codificante

- i **mutually exclusive exons**, ovvero *esoni mutuamente esclusivi*, dove solo uno di due esoni viene conservato nel trascritto. In due isoforme diverse non ho mai entrambi gli esoni
- l'**intron retention**, ovvero *introne trattenuto*, dove un certo introne viene incluso nel trascritto primario o dove un esone viene considerato in più parti con un introne in mezzo
- **multiple promoters** dove non si considera un prefisso del primissimo esone
- **multiple polyA** dove non si considera un suffisso dell'ultimo esone

9.3.1 Il formato GTF

Il **formato GTF (*Gene Transfer Format*)**, `.gtf` è il formato standard che permette di annotare un dato gene su una sequenza genomica di riferimento che contiene il locus del gene (la sequenza può anche essere nello strand opposto a quello dato). Un GTF fornisce per un gene:

- la composizione in esoni delle sue isoforme di splicing (quindi di tutti i trascritti)
- la composizione delle CDS in relazione ai trascritti
- la composizione delle 5/3'UTR in relazione ai trascritti
- la presenza dello start e dello stop codon delle CDS

Da un file GTF si possono ricostruire tutti i trascritti e le coding sequence del gene annotato.
GTF deriva dal formato GFF, spesso si hanno ancora GTF in `.ggf` ed è formato da 9 campi separati da tabulazioni.
Il trascritto viene annotato "all'indietro" sul locus del gene.
Ogni record del GTF fornisce una **feature**, ovvero una regione continua della sequenza genomica, che rappresenta uno tra:

- un esone, tramite `exon`. Ad un trascritto corrispondono tante feature di tipo `exon` sulla genomica di riferimento quanti sono gli esoni che lo compongono (e per ogni feature di tipo `exon` ho un record)

- una porzione di CDS, tramite `CDS`. Una porzione in quanto la CDS completa sul locus è "spezzata" dagli introni e quindi mi servono più feature per rappresentarla. Ipotizzando che una CDS sia un pezzo di un esone e un intero secondo esone avrò due feature `CDS` con gli indici di queste due parti (se corrisponde ad un esone avrò anche una feature con gli stessi indici di tipo `exon`). Ad una CDS corrispondono tante feature quanti gli esoni che copre sul trascritto
- uno start codon, con `start_codon`. Solitamente è un unico record
- uno stop codon, con `stop_codon`. Solitamente è un unico record
- una porzione 5'UTR, con `5UTR`. A un 5'UTR corrispondono tante feature sulla sequenza di riferimento quanti sono gli esoni che copre sul trascritto
- una porzione 3'UTR, con `3UTR`. A un 3'UTR corrispondono tante feature sulla sequenza di riferimento quanti sono gli esoni che copre sul trascritto

Il file GTF ha solo gli indici di partenza e fine dei vari elementi e quindi mi serve associato un file FASTA con la genomica di riferimento.
Un GTF può avere più geni annotati.
Ogni record ha 9 campi:

1. l'**identificatore della genomica di riferimento** (presa su uno dei due strand) che copre il locus del gene
2. la **sorgente** che ha prodotto l'annotazione (ad esempio un software se prodotta "in silico")
3. il **nome della feature** (*exon, CDS, 5UTR, 3UTR, start_codon, stop_codon*)
4. la **posizione di inizio** della feature sulla genomica di riferimento (1-based, non si parte da 0 ma 1 con gli indici)
5. la **posizione di fine** della feature sulla genomica di riferimento (1-based)
6. lo **score** della feature

7. lo **strand** ("+", "-"). In un file GTF possono coesistere geni che si esprimo su strand opposti e che vengono annotati su un'unica genomica di riferimento. Il GTF permette di annotare un gene che contiene il sup locus sulla catena opposto e questo campo mi permette di segnalare la cosa. Se ho "-" so che il gene è sullo strand opposto rispetto a quello della genomica di riferimento (con "+" è sullo stesso). Con "-" e features del gene avranno coordinate decrescenti passando da sinistra a destra sul trascritto mentre con "+" avranno coordinate crescenti passando da sinistra a destra sul trascritto. Ne segue che:

    - per ottenere la sequenza di una feature con strand "+" basta estrarre la sottostringa della genomica di riferimento che corrisponde alla feature
    - per ottenere la sequenza di una feature con strand "-" bisogna estrarre la sottostringa della genomica di riferimento che corrisponde alla feature e fare *reverse&complement* della sottostringa estratta

    Ovviamente tutte le features annotate per un dato gene avranno sicuramente lo stesso campo strand

8. il **frame** (0, 1, 2) solo per feature `CDS`, `start_codon` e `stop_codon`. Nel dettaglio si ha che, per la feature `CDS`:

    - 0, se la prima base della feature è la prima base di un codone
    - 1, se la prima base della feature è la terza base di un codone
    - 2, se la prima base della feature è la seconda base di un codone

    Possiamo dire che, dato L lunghezza totale delle feature `CDS` che vengono prima di quella in analisi, si ha:

    $$(3 - L \bmod 3) \bmod 3$$

    Per le altre feature trovo un "."

9. il campo degli **attributi della feature** come coppie chiave-valore: *<attribute_name1> <value1>; <attribute_name2> <value2>; ...*
   Gli attributi *gene_id* e *transcript_id* sono attributi obbligatori

Capitolo 10

Strutture di Indicizzazione

Le strutture di indicizzazione permettono varie cose nello studio del genoma, dall'assembly, all'individuazione di match, MEMs etc..., anche in ambito pan-genomica. Tra le strutture principali abbiamo:

- suffix tree
- suffix array
- LCP
- BWT

e le studieremo sia in ottica genomica che pan-genomica.
Con $T[i; j]$ abbiamo la sottostringa da i a j (incluso) e con $T[i :]$ il suffisso che inizia in i.

10.1 Caso Genomico

10.1.1 Suffix Tree

È una delle prime strutture create, è molto efficiente e permette di risolvere tanti problemi in tempo lineare ma ha lo svantaggio dell'occupazione di memoria, memorizzando in modo ridondante. Non va quindi bene applicato direttamente a genomi e dati NGS. Volendo può essere simulato tramite SA, LCP e altre strutture.

Definizione 32. *Dato un testo T di n caratteri un suffix tree è un albero radicato con:*

- *n foglie etichettate da 1 a n*

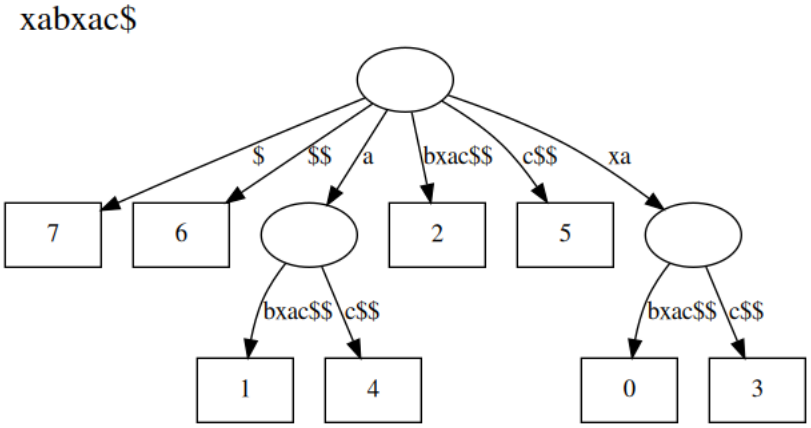


Figura 10.1: Esempio di suffix tree per la stringa "xabxac"

- *ogni nodo interno ha almeno 2 figli*
- *ogni arco è etichettato da una sottostringa di T*
- *non esistono due archi uscenti da un nodo con lo stesso carattere iniziale per l'etichetta*
- *la concatenazione delle etichette dalla radice alla foglia i è $T[i:]$*

Anche ogni etichetta di arco che porta ad una foglia è un suffisso, *da qui il discorso delle ripetizioni e della ridondanza di informazione.*

Teorema 6. *Il ST di T esiste sse nessun suffisso occorre come prefisso di qualche altro suffisso, in quanto avrei un nodo interno con figli che iniziano con lo stesso carattere come primo carattere delle sottostringhe e per evitare la cosa otterrei nodi con singoli figli.*

Per risolvere il problema si usano testi $ terminati, con $ che è il carattere minore di tutti lessicograficamente.
Si hanno tre algoritmi lineari per la costruzione, in $O(n)$:

- **algoritmo di Weiner**
- **algoritmo di McCreight's**, efficiente in memoria
- **algoritmo di Hukkonen**, online

Definizione 33. *Dato un nodo w definisco la **path label** L_w come la concatenazione delle etichette dalla root a w. Se w è una foglia i allora la path label è il suffisso $T[i:]$*

Definizione 34. *Dato un nodo w definisco la* ***string depth*** *come la lunghezza della path label L_w*

Definizione 35. *Definisco* ***dummy node*** *come un falso nodo che separa in due l'etichetta di un arco (w, w') in due stringhe l_1 e l_2. La sua path label è $L_w L_1$. È un nodo a metà di un arco.*

Teorema 7. *Due nodi diversi non possono avere la stessa path label.*

Teorema 8. *Dato un nodo interno w, le k foglie del sottoalbero radicato in w danno le posizioni di inizio dei k suffissi che condividono L_w come prefisso.*

Teorema 9. *Dato un dummy node d che separa (w, w') le k foglie del sottoalbero radicato in w danno le posizioni di inizio dei k suffissi che condividono $L_w L_1$, ovvero la path label di d, come prefisso.*

Per il pattern matching, con pattern P lungo m e testo T lungo n, con ST si ha una soluzione in tempo:

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

avendo:

- costruzione di ST in $O(n)$
- ricerca delle k occorrenze in $O(m + k)$, cercando in $O(m)$ ò'unico nodo w che ha path label pari a P e visito in $O(k)$ il sottoalbero radicato in w elencando le k foglie

In quanto si sa ha che:

- se P occorre k volte allora P è prefisso di k suffissi di T
- se P è prefisso di k suffissi di T allora P è la path label di un nodo w, anche dummy, che è unico
- le k foglie del sottoalbero radicato in w, che può essere dummy, danno le posizioni di inizio delle k occorrenze di P su T

Se P non occorre non trovo il w e mi fermo.
Avendo alfabeto Σ finito la ricerca dell'arco uscente da un certo nodo la cui etichetta inizia con un certo carattere è in tempo costante con la giusta implementazione.
Su slide esempio di pattern matching.

10.1.2 Suffix Array

Riprendere teoria da appunti di Teoria della Computazione.
Il SA è nato come struttura ausiliaria di un ST. Si ricorda che la costruzione è in $O(n)$ e il pattern matching è in $O(m \log n)$.
Si usa spesso il suffix array inverso SA^{-1}, è definito in modo tale che in posizione j ho l'indice i sse $T[j:]$ è l'i-esimo suffisso nell'ordinamento lessicografico dei suffissi di T:

$$SA^{-1}[j] = i \iff SA[i] = j$$

In tempo lineare posso costruire il SA dal ST, visitando in profondità, in $O(n)$, il ST seguendo l'ordine lessicografico delle etichette degli archi uscenti da un dato nodo, scelgo sempre l'arco con etichetta "minore". Ogni volta che arrivo ad una voglia aggiungo l'etichetta della foglia al SA, in ordine. Ne segue che il primissimo è ovviamente sempre \$ e quindi il primo indice del SA è ovviamente il suffisso nullo.
Esempio su slide.

10.1.3 Longest Common Prefix

Passiamo ora all'**LCP** è stato introdotto come struttura ausiliaria per il SA.

Definizione 36. *Dato un testo T dollaro terminato, l'LCP è un array lungo n tale che $LCP[i] = l$ sse l è la lunghezza del più lungo prefisso comune tra $T[S[i]:]$ e $T[S[i-1]:]$. In altri termini è la lunghezza del più lungo prefisso comune dei suffissi corrispondenti a $S[i]$ e $S[i-1]$.*
SI assume $LCP[1] = -1$

L'algoritmo di Kasai lo costruisce a partire da SA in tempo lineare.
SA e LCP permettono il pattern matching in $O(m + \log n)$, migliorando il $O(m \log n)$ del solo SA. Questo è permesso dal fatto che con LCP il confronto del pattern non torna mai indietro.
Quindi $lcp(i, j)$ è la **LCP-funcion** è computa lunghezza del più lungo prefisso comune tra $T[S[i]:]$ e $T[S[j]:]$. Assumendo $i < j$ si ha che $lcp(i, j)$ è il minimo tra i valori di LCP compresi tra $LCP[i+1]$ e $LCP[j]$ quindi, usando la **range minimum query** (cerco il valore minimo tra $LCP[i+1]$ e $LCP[j]$):

$$lcp(i, j) = RMQ(LCP, i + 1, j)$$

Esempio su slide.

Definizione 37. *Definiamo* ***lcp-interval*** *di valore l come l'intervallo di posizioni $[i, j]$ tale che:*

- $i < j$
- $LCP[i] < l$
- $LCP[j + 1] < l$
- $LCP[k] \geq l, i + 1 \leq k \leq j$
- $LCP[k] = l$ *per almeno un* $k \in [i + 1, j]$*, tale* k *è detto* l*-index*

L'lcp-interval $[i, j]$ è anche detto l-interval, detto anche l-$[i, j]$.
Il valore l di un lcp-interval $[i, j]$ è uguale a:

$$RMQ(LCP, i + 1, j)$$

Il valore l è la lunghezza del più lungo prefisso comune tra i suffissi di T che iniziano in posizione $S[i], S[i + 1], \ldots, S[j]$, che viene denotato con ω.
Dato $l = 0$ si ha che:

$$0\text{-}[1, n]$$

è lo 0-interval che ha come ω il prefisso nullo.

Esempio su slide.

Definizione 38. *Definisco l'-interval $[g, d]$ con prefisso ω' come* ***embdedded*** *in l-interval $[i, j]$, con prefisso ω sse;*

$$i \leq g < d \leq j$$

Quindi $[i, j]$ racchiude $[g, d]$.
In tal caso si ha:

$$l' > l$$

e ω è prefisso di ω'.

Esempio su slide.

Definizione 39. *Si dice che l'-$[g, d]$ è figlio di l-$[i, j]$ sse non esiste un altro lcp-interval che racchiude $[g, d]$ e che a sua volta è racchiuso in $[i, j]$.*
Per trovare i figli di l-$[i, j]$:

- *identifico gli l-indici $i_1, \ldots i_k$, dal più piccolo al più grande*

- *il primo figlio è* $[i, i_1 - 1]$*, il secondo* $[i, i_2 - 1]$ *e etc... fino a* $[i_k, j]$*, scartando però quelli di ampiezza 1*
- *il valore* l *di ogni figlio* $[g, d]$ *è dato da:*

$$RMQ(LCP, g + 1, d)$$

Definizione 40. *Definisco* ***lcp-interval tree*** *come un albero radicato* (V, A) *tale che:*

- V *è l'insieme degli lcp-interval*
- A *è l'insieme di tutte le relazioni parent-child*
- *la radice corrisponde a* 0-$[1, n]$

Esempio su slide.
Grazie a tutte queste nozioni possiamo legare SA e ST. Se prendo un ST e nascondo foglie e archi in esse entranti ottengo una struttura uguale a quella dell'lcp-interval tree, stessa topologia. Esiste un nodo l-$[i, j]$ con prefisso ω nell'lcp-interval tree sse si ha un nodo nel ST con path label ω.
Con SA, SA inverso, LCP posso simulare un attraversamento del ST, anche senza lcp-interval tree esplicito tramite *enhanced SA*, senza mi serve anche il lcp-interval tree.

10.1.4 BWT e FM-Index

Riprendere teoria da appunti di Teoria della Computazione.
Posso avere Q-intervalli che sono lcp-interval ma non tutti, visto che scarto gli lcp-interval di ampiezza 1. Quando si ha la corrispondenza ho $Q = \omega$.
Dato il Q-intervallo $[b, e]$ (quindi con intervallo chiuso, non come a teoria della computazione) ho la backward extension per σ: ho che:

$$b' = C(\sigma) + Occ(\sigma, b) + 1$$

$$e' = C(\sigma) + Occ(\sigma, e + 1)$$

Se $e' < b'$ allora il σQ-interval non esiste.
Se ho un Q-intervallo $[b, e]$ che sulla BWT contiene \$ allora esso caratterizza una stringa Q che è prefisso di T.

10.2 Caso Pan-Genomico

Si passa ora dal singolo testo ad una **collezione di testi**. Tranne il suffix tree si vedranno le varie strutture di indicizzazione nell'ottica di riconoscere overlap all'interno di collezione di read (che appunto è una collezione di testi). Ci si basa quindi su un **overlap graph**.

10.2.1 Suffix Tree

Definizione 41. *Si definisce* ***Generalized Suffix Tree (GST)****, costruito su un set di h testi* T_j $\$_j$*-terminati (ogni testo ha il suo dollaro), come un albero radicato tale per cui:*

- *per ogni suffisso* $T_j[i:]$ *esiste una foglia etichettata* (i, j)*, aggiunge quindi l'indice del testo all'etichetta. Si avranno n foglie con n somma delle lunghezze dei testi*
- *ogni arco è etichettato da una sottostringa di un testo e non si hanno due archi uscenti dallo stesso nodo con lo stesso carattere*
- *la concatenazione delle etichette dalla radice alla foglia* (i, j) *è uguale a* $T_j[i:]$

Esempio su slide.
Un GST si può costruire in $O(n)$, con n somma delle lunghezze dei testi. Per farlo semplicemente concateno i vari testi in un solo T, ognuno con il loro dollaro finale. A questo punto costruisco il ST come al solito, in tempo lineare sulla lunghezza del testo. Non si è arrivato alla fine però, non avendo ancora il GST come definito. Bisogna sistemare le etichettature delle foglie, in quanto allo stato attuale abbiamo solo l'indice del suffisso e non il testo di provenienza (visto che si è partiti da un testo unico).
Si parte dal caso semplice con 2 testi, $T = T_1T_2$, e una posizione p sul testo unificato. Ho che $\exists i$ tale che:

- se p cade in T_1 si ha che $T[p:] = T_1[i:]T_2$
- se p cade in T_2 si ha che $T[p:] = T_2[i:]$, essendo T_2 l'ultimo testo della concatenazione (notando che ovviamente I non coincide con p)

Si ha che l'arco entrante nella foglia p di ST avrà etichetta:

- sT_2, con s suffisso di $T_1[i:]$ se $T[p:] = T_1[i:]T_2$, ovvero il suffisso rappresentato da p è rappresentato da un percorso radice foglia e quindi l'ultimo arco del cammino è etichettato da un suffisso del suffisso di T_1 che inizia in i seguito da T_2, avendo che T contiene separatori unici, i vari $\$_j$. Sostituisco quindi con s l'etichetta sT_2 dell'arco entrante in p (tengo la parte che cade in T_1 del suffisso, fermandomi grazie al $\$_1$) e trasformo la foglia p in $(i, 1)$

- s, con s suffisso di $T_2[i:]$ se $T[p:] = T_2[i:]$, essendo l'ultimo testo ho già s come etichetta dell'arco entrante nella foglia p, che viene etichettata come $(i, 2)$. Questo caso è una generalizzazione del primo caso, semplicemente dopo s si ha ε

Il ST è quindi uguale al GST dal punto di vista topologico ma cambiano le etichette delle foglie, questo grazie ai $\$_j$, unici e che separano i vari T_j.
Generalizziamo ora ad h testi.
In ST ogni foglia è ora etichettata da una posizione p su T che si riferisce a $T[p:]$ tale che:

- esiste un indice i e uno j per cui $T[p:] = T_j[i:]T_{j+1}T_{j+2}\dots T_h$

- l'arco entrante in p ha etichetta $sT_{j+1}T_{j+2}\dots T_h$ con s suffisso di $T_j[i:]$

Quindi per trasformare ST di T nel GST dei vari T_j si procede per ogni foglia p:

- si individua il testo per cui l'etichetta dell'arco entrante nella foglia p sia $sT_{j+1}T_{j+2}\dots T_h$ con s suffisso di T_j

- si rimuove la parte di etichetta a destra del primo $\$_j$ incontrato a sinistra, lasciando quindi solo s

- si sostituisce l'etichetta di p con (i, j)

Come per ST si hanno varie osservazioni anche per il GST:

- dato un *dummy* w, le k foglie del suo sottoalbero danno le k occorrenze della sua *path label* L_w dei testi

- dato un *dummy* w, se il suo sottoalbero ha due foglie $(., j)$ e $(., j')$, $j \neq j'$, allora la *path label* L_w è sottostringa comune a T_j e $T_{j'}$

Queste due osservazioni ci permettono di calcolare la *longest common substring* tra T_1 e T_2, di lunghezza n_1 e n_2, in $O(n_1 + n_2)$:

- si costruisce il GST in $O(n_1 + n_2)$
- si visita il il GST in $O(n_1 + n_2)$ e si etichetta ogni nodo interno w con:
  - 1, se il suo subtree ha solo foglie di tipo (.,1), avendo che la path label L_w occorre solo in T_1
  - 2, se il suo subtree ha solo foglie di tipo (.,2), avendo che la path label L_w occorre solo in T_2
  - 0, se il suo subtree ha foglie di tipo (.,1) e (.,2), avendo che la path label L_w occorre in entrambi i testi
- si trova il nodo interno w, etichettato con 0, di massima profondità di string. L_w è la *longest common substring*

10.2.2 Suffix Array

Definizione 42. *Dati h testi $T_1, T_2, \ldots, T_h$ (\$-terminati, stavolta senza \$ diversi) di lunghezza totale n, il **Generalized Suffix Array (GSA)** è un array S lungo n, tale che $S[i] = (p, j)$ se e solo se $T_j[p:]$ è l'i-esimo suffisso nell'ordinamento lessicografico di tutti i suffissi dei testi.*
Per avere un solo \$ si assume che il suffisso $s\$$ di T_i è minore del suffisso $s\$$ di T_j se $i < j$, potendo ordinare suffissi uguali.

10.2.3 Longest Common Prefix

Definizione 43. *Dati h testi $T_1, T_2, \ldots, T_h$ (\$-terminati, stavolta senza \$ diversi) di lunghezza totale n, il **Generalized LCP Array** è un array LCP lungo n tale che $LCP[i] = l$ sse il più lungo prefisso comune tra l'i-esimo e l'(i-1)-esimo suffisso, nell'ordinamento lessicografico, ha l caratteri.*
l'LCP array non tiene conto del carattere \$.

10.2.4 BWT e FM-Index

Definizione 44. *Dati h testi $T_1, T_2, \ldots, T_h$ (\$-terminati, stavolta senza \$ diversi) di lunghezza totale n, la **Generalized BWT** è un array B lungo n, tale che $B[i] = Tj[p-1]$, se e solo se $S[i] = (p, j)$.*

Tutto il resto resta invariato nell'estensione ad una collezione di testi, ovvero:

- C e Occ di FM-Index

- LF function
- Q-interval e backward extension di un Q-interval
- pattern matching

10.2.5 Overlap tra Testi

Definizione 45. *Dati due testi T_i e T_j si ha che Q è un **overlap** se Q è suffisso di T_i e prefisso di T_j. Il \$ viene escluso.*

Definizione 46. *Dati h testi $T_1, T_2, \ldots, T_h$ si ha che Q è un loro **overlap** se esistono due sottoinsiemi T_s e T_p tali che, per ogni coppia (T_i, T_j) del prodotto cartesiano $T_s \times T_p$:*

- *T_i ha Q come suffisso, escluso \$*
- *T_j ha Q come prefisso*

L'overlap è quindi una stringa in comune tra un gruppo di testi che ha Q come prefisso e un altro gruppo di testi che ha Q come suffisso, escluso \$.

Si assume che nessun testo è sottostringa di alcun altro testo, si ha che il set di input è *substring-free*. Qualora non lo sia si fa un preprocessing veloce
Esempi su slide.
Data questa definizione bisogna parlare di Q-interval e delle sue eventuali caratteristiche.

Definizione 47. *Data la stringa Q, overlap di h testi, allora si ha che il Q-interval $[b, e]$, detto **overlap interval**, ha queste caratteristiche:*

- *ha ampiezza almeno due (un overlap deve occorrere almeno in due testi)*
- *contiene un sottointervallo proprio $[B, e_m]$ dove i suffissi sono esattamente $Q\$$, avendo che gli indici dei testi letti dal SA in $[b, e_m]$ sono quelli relativi ai testi che, eslcuso il terminatore, hanno Q come prefisso, ovvero i testi in T_s*
- *contiene almeno una posizione i tra $e_m + 1$ ed e dove $B[i] = \$$ e quindi gli indici dei testi letti dal SA in $[e_m + 1, e]$ sono quelli dei testi che hanno Q come suffisso, ovvero i testi in T_p*

Questo è possibile per la premessa del substring-free set, in quanto:

- $B[b, e_m]$ *non contiene caratteri* \$
- *il numero di* \$ *in* $B[b, e]$ *è uguale al numero di* \$ *in* $[e_m + 1, e]$

Dato il Q-interval $[b, e]$ di cui si sa la lunghezza di Q ma non Q si può verificare se è un overlap-interval:

- cerco la prima posizione p, tale che $p \geq b$, tale per cui la lunghezza del suffisso, \$ escluso è diverso da $|Q|$
- se $p > b$ allora la posizione finale del sottointervallo $[b, e_m]$ è $e_m = p - 1$ e la scansione prosegue fino a e, posizione finale, cercando simboli \$ nella BWT B
- se in $B[e_m + 1, e]$ si trova almeno un \$ allora $[b, e]$ è un overlap interval

Si noti che T_s e T_p si possono calcolare da SA durante la scansione.
Esempi su slide.
Vediamo quindi un algoritmo di riconoscimento overlap in un insieme di testi:

1. costruisco SA e BWT della collezione di testi
2. parto da ε-interval e lo aggiungo ad una coda C insieme alla lunghezza 0 di ε
3. estraggo da C un intervallo $[b, e]$ con la sua lunghezza L e, $\forall \sigma \in \Sigma$:
    - effettuo la backward extension di $[b, e]$ in $[b', e']$
    - se $[b', e']$ ha almeno ampiezza due verifico se è un overlap interval e aggiungo $[b', e']$ nella coda C, insieme alla sua lunghezza data da $L + 1$
    - se C non è vuota eseguo di nuovo 3)

Questo algoritmo è esponenziale e inutilizzabile in ambito genomico ma usando anche LCP si ottiene un algoritmo lineare in quanto $LCP[i] > LCP[i-1]$ implica che:

- $i - 1$ è la posizione di inizio di un numero $LCP[i] - LCP[i - 1]$ di Q-interval, di ampiezza almeno due, relativi alle lunghezze $LCP[i - 1] + 1, LCP[i - 1] + 2, \ldots, LCP[i]$ per la stringa Q
- tali intervalli sono l'uno contenuto nell'altro, a partire da quello con lunghezza di Q maggiore, ovvero $LCP[i]$

- solo l'intervallo relativo a $|Q| = LCP[i]$ può essere un overlap interval

Quindi $(i-1)$ può essere l'inizio di un Q-interval che è **overlap interval** tale per cui $|Q| = LCP[i]$ mentre gli altri che iniziano con $(i-1)$ con lunghezze di Q inferiori a $LCP[i]$ hanno sicuramente una posizione di fine che è maggiore (o uguale) a quella del candidato **overlap interval**.
D'altro canto $LCP[i] < LCP[i-1]$ implica che:

- $i-1$ è la posizione di fine di un numero $LCP[i-1]-LCP[i]$ di Q-interval, di ampiezza almeno due, relativi alle lunghezze $LCP[i]+1, LCP[i]+2, \ldots, LCP[i-1]$ per la stringa Q
- tali intervalli sono l'uno contenuto nell'altro, a partire da quello con lunghezza di Q maggiore, ovvero $LCP[i-1]$
- $i-1$ può essere la fine di un overlap interval

Esempio su slide.

Capitolo 11

Riarrangiamento Genomico

Il **riarrangiamento genomico** è uno degli effetti comuni nei tumori e nelle patologie tumorali ed è un fenomeno molto diverso dalle mutazioni.
Ricordando che il genoma è diviso in cromosomi si ha che essi possono essere:

- **lineari**, nel caso eucariote
- **circolari**, nel caso procariote

L'insieme di tutti i cromosomi è detto **cariotipo**.
Le differenze di genoma tra due individui possono essere:

- **mutazioni puntuali**, di singoli nucleotidi (come già visto nel corso con indel etc...)
- **riarrangiamenti genomici**, con mutazioni di gruppi di nucleotidi

Se le mutazioni puntuali avevano come operazioni *inserzione, delezione* e *sostituzione* si ha che il riarrangiamento genomico ha le seguenti operazioni, che prevedono cambiamenti a livello dei geni (spesso rappresentati tramite posizioni numeriche, che fissa un ordine come negli esempi sulle slide):

- **inversione**, quando una sequenza di geni cambia ordine e viene totalmente invertita:

$$[1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9] \rightarrow [1\ 2\ 3\ \mathbf{6}\ \mathbf{5}\ \mathbf{4}\ 7\ 8\ 9]$$

- **traslocazione**, quando si scambiano due pezzi genomici tra due cromosomi:

$$[1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9] \rightarrow [1\ 2\ 3\ 4\ \mathbf{13}\ \mathbf{14}\ \mathbf{15}]$$

$$\wedge$$

$$[10\ 11\ 12\ 13\ 14\ 15] \rightarrow [10\ 11\ 12\ \mathbf{5}\ \mathbf{6}\ \mathbf{7}\ \mathbf{8}\ \mathbf{9}]$$

- **fissione**, dove un pezzo cromosoma si spezza in due pezzi

$$[1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9] \rightarrow [1\ 2\ 3\ 4] \wedge [5\ 6\ 7\ 8\ 9]$$

- **fusione**, dove due pezzi di cromosoma si fondono in uno solo

$$[1\ 2\ 3\ 4] \wedge [5\ 6\ 7\ 8\ 9] \rightarrow [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$$

Fusione e *fissione* sono l'una il complementare dell'altra.
Sono in ordine di diffusione e gli ultimi due, i più rari, avvengono mediamente durante la **meiosi**.
Il riarrangiamento è spesso scatenato dalle radiazioni (che portano tumori).
Per le mutazioni puntuali si usavano distanza di edit e alberi filogenesi perfetta, con assunzione di parsimonia. Vedremo come fare lo stesso nel caso dei riarrangiamenti.
Le operazioni di edit possono quindi anche essere di questo tipo, del riarrangiamento genomico.
Il riarrangiamento si avvicina al concetto di **anagramma di una parola**.
Potrei avere anche tra due specie lo stesso contenuto di geni in posizioni diverse o semplicemente in ordine invertito (**grafici su slide**). Questo porta a vedere come tra uomo e topo ci siano molti geni simili magari solo con ordine diverso e si cerca di capire quante operazioni di riarrangiamento (quindi di edit sui geni) servavano per passare da un genoma all'altro (e sono abbastanza pochi in quanto hanno in comune un antenato). Queste differenze rappresentano anche la storia evolutiva, anche dei singoli cromosomi, usando i riarrangiamenti (**grafici su slide**).
A livello biologico l'inversione è stata osservata per la prima volta con l'**esperimento diDobzhansky** nella drosofila, a fine anni trenta. È stato osservato quindi un cambiamento nell'ordine dei geni. L'esperimento si è basato sulla classificazione delle specie di drosofila, tramite un albero filogenetico. Si è poi scoperto che l'inversione è data dal fatto che il filamento di DNA, coi vari geni, va a formare dei "nodi", formando eventualmente un'apertura, ricongiungendosi poi con l'ordine invertito dato dal fatto che c'era il nodo (**immagini su slide**).
Vediamo quindi le **inversioni non segnate** e il cambiamento nell'ordine dei geni. Si usa l'operazione binaria:

$$r(a, b)$$

con a e b posizione di inizio e fine. I geni tra gli indici a e b, inclusi, vengono invertiti.
Vediamo quindi il **problema della distanza di edit per inversioni**, dove

date due permutazioni si cerca la più corta serie di inversioni per trasformare una permutazione nell'altra:

- **input**: due permutazioni π e σ
- **output**: una serie di inversioni $r_1, \ldots, r_t$ che trasforma π in σ tale che t sia minimo

Tale t è quindi la **distanza di edit per inversioni** tra le due permutazioni, avendo:

$$t = d_{rev}(\pi, \sigma)$$

Non si ha quindi una distanza di edit alla Levenshtein, per inserzioni/delezioni/sostituzioni, ma stimata tramite l'inversione di pezzi.
Un problema analogo è quello detto **problema ordinamento per inversioni**, che data una permutazione cerca la minima serie di inversioni per trasformare la permutazione nella **permutazione identica**, $(1, 2, 3, \ldots, n)$:

- **input**: una permutazione π
- **output**: una serie di inversioni $r_1, \ldots, r_t$ che trasforma π in $(1, 2, 3, \ldots, n)$, la permutazione identica, tale che t sia minimo

Il legame tra i due problemi si ritrova nel fatto che si prende la permutazione σ e se si mappano i suoi elementi e si fa una biezione tra gli elementi di σ e quelli della permutazione identica da questa biezione si ha che si ha una forte analogia tra i due problemi (trasformare π in σ con σ permutazione identica). Si passa dal primo problema al secondo perché è più semplice, l'ordinamento avviene per inversioni.
Per capire il discorso servono alcune definizioni:

Definizione 48. *Data una permutazione π, che viene estesa all'inizio con* 0 *e alla fine con* $\max(\pi) + 1$ *definisco:*

- ***adjacency*** *come una coppia di elementi su due indici consecutivi che sono consecutivi anche come valore, in un qualsiasi ordine*
- ***breakpoint*** *come una coppia di elementi su due indici consecutivi che non sono consecutivi come valore. I breakpoint sono quindi le discontinuità tra "pezzi" di stringa che sono ordinati (crescenti o decrescenti che siano)*
- $b(\pi)$ *come il numero di breakpoint in π*

Esempio 18. *Dato:*

$$\pi = [5\ 6\ 2\ 1\ 3\ 4]$$

lo estendo a π'*:*

$$\pi' = [0\ 5\ 6\ 2\ 1\ 3\ 4\ 7]$$

e ho che:

- *le coppie* $(5,6), (2,1), (3,4)$ *sono adjacency*
- *le coppie* $(0,5), (6,2), (1,3), (4,7)$ *sono breakpoint*
- $b(\pi) = 4$

I breakpoint ci dicono dove iniziano/finiscono **variazioni strutturali**, ovvero un cambiamento genomico che coinvolge più nucleotidi consecutivi (il riarrangiamento, riguardando addirittura geni interi, è una variazione strutturale).
La distanza di inversioni è legata ai breakpoint in quanto ciascuna inversione elimina al più due breakpoint quindi si ha:

$$\text{distanza di inversioni} \geq \frac{b(\pi)}{2}$$

Esempio su slide.
Trasformare una permutazione nell'identica ha costo esponenziale, il problema del **sorting by reversal** è infatti un problema NP-complete.
Vediamo ora un algoritmo che consente di approssimare la distanza per inversione tramite una 2-approssimazione.

Definizione 49. *Si definisce uno* ***strip*** *come l'intervallo tra due breakpoint consecutivi in una permutazione.*
Si ha che:

- *uno strip è* ***crescente*** *se gli elementi dell'intervallo, quindi dello strip, sono crescenti*
- *uno strip è* ***decrescente*** *se gli elementi dell'intervallo, quindi dello strip, sono decrescenti*
- *uno strip di un solo elemento è considerabile crescente o decrescente, come si preferisce, tranne due eccezioni:*
  - *lo strip* $[0]$
  - *lo strip* $[n+1]$

Avendo una 2-approssimazione ho che, con $opt(x)$ ottimo sull'istanza x:

$$A(x) \leq k \cdot opt(x) \implies A(x) \leq 2 \cdot opt(x)$$

È quindi un'euristica con un fattore garantito di approssimazione.
Per il **sorting by reversal** si avrebbe anche un algoritmo banale 4-approssimante.
Per l'algoritmo 4-approssimante si hanno due osservazioni:

1. se una permutazione contiene uno strip decrescente, allora esiste un'inversione che diminuirà il numero di breakpoint. Si ha, per esempio, usando "|" per i breakpoint:

   $$[0\ 1 \mid \mathbf{5\ 6\ 7} \mid \mathbf{4\ 3\ 2} \mid 8\ 9\ 10] \overset{r(3,8)}{\Longrightarrow} [0\ 1\ 2\ 3\ 4 \mid 7\ 6\ 5 \mid 8\ 9\ 10]$$

   Dove si passa da 3 breakpoint a 2

2. in caso contrario si crea uno strip decrescente, invertendo uno strip crescente. Il numero di breakpoint può essere ridotto nel passaggio successivo e non in questo. Si ha ad esempio:

   $$[0\ 1 \mid 5\ 6\ 7 \mid \mathbf{2\ 3\ 4} \mid 8\ 9\ 10] \overset{r(6,8)}{\Longrightarrow} [0\ 1 \mid 5\ 6\ 7 \mid \mathbf{4\ 3\ 2} \mid 8\ 9\ 10]$$

   Arrivando poi alla situazione in cui si applica l'esempio della prima osservazione.

L'algoritmo elimina ogni due step un breakpoint:

Algorithm 3 Algoritmo 4-approssimante per **Breakpoint Reversal Sort (*BRS*)**, con $\pi \bullet r$ che segnala l'applicazione dell'inversione r alla permutazione π

```
function SBRapprox(π)
    while b(π) > 0 do
        if π has a decreasing strip then
            Choose reversal r that minimizes b(π • r)
        else
            Choose a reversal r that flips an increasing strip in π
        π ← π • r
        return π
```

Considerando di rimuovere uno o due breakpoint, riuscendo ad ordinare facendo flip almeno uguali al numero di breakpoints, ottengo una 2-approssimazione. Nel nostro caso invece si ha una 4-approssimazione infatti:

- elimina almeno un breakpoint ogni due step, avendo:

$$d_{BRS} \leq 2 \cdot b(\pi)\ step$$

- un algoritmo ottimale elimina al più due breakpoint ogni step, avendo:

$$d_{OPT} \geq \frac{b(\pi)}{2}\ step$$

Quindi si ha:

$$\frac{d_{BRS}}{d_{OPT}} \leq \frac{2 \cdot b(p)}{\frac{b(p)}{2}} = 4$$

L'algoritmo può essere migliorato a 2-approssimante tramite passaggi sofisticati, basati sul fatto che si può dimostrare che:

Se una permutazione ha un strip decrescente (incluso quello di lunghezza 1), allora è sempre possibile diminuire il numero di breakpoint

Resto della trattazione sul 2-approssimante su slide, non fatto in aula.
Parlando invece di **inversioni segnate** si ha che l'ordine inverso viene identificato tramite segno "-", ad esempio:

[1 2 3 -8 -7 -6 -5 -4 9 10]

Che nel dettaglio presenta due breakpoint.
In ogni caso quando si parla di geni si intende che possono essere su entrambi gli strand, studiandoli in modo adeguato.
Si hanno due tool per il calcolo della distanza segnata o non segnata per inversione tra due permutazioni:

- *GRIMM Web Server*, che è usato proprio per il calcolo della distanza segnata o non segnata per inversione tra due permutazioni
- *Cinteny*, anche lui un web server, che oltre allo studio del riarrangiamento è usato anche per l'identificazione di **sintenia**, ovvero l'associazione di due o più geni in uno stesso cromosoma