IL PARTO DI BIOINFORMATICA

Adrian Castro, Ilaria Battiston

Anno scolastico 2018-2019

Contents

1 Notazione

Durante il corso verranno usate delle specifiche notazioni.

1.1 Stringhe

Da notare che l'array inizia da 1, e non da 0.

1. Simbolo: $T[i]$;
2. Stringa: $T = T[1]T[2]T[3]T[4]\dots T[l]$;
3. Sottostringa: $T[i : j]$;
4. Prefisso: $T[: j] = T[1 : j]$ (estremi inclusi),
   Esempio: $T = \text{abcdefg}, T[: 3] = \text{abc}$;
5. Suffisso: $T[i :] = T[i : |T|]$ (estremi inclusi),
   Esempio: $T = \text{abcdefg}, T[3 :] = \text{cdefg}$;
6. Concatenazione: $T_1 \cdot T_2 = T_1T_2$.

2 Pattern matching

2.1 Problema

Il problema è il seguente: dato un testo $T = T[1]T[2]\dots T[n]$ e un pattern $P = P[1]P[2]\dots P[m]$ (con alfabeto Σ qualunque), trovare **tutte** le occorrenze di P in T, ovvero trovare tutti gli i tale che $T[i]\dots T[i + m - 1] = P$.

Questo algoritmo viene utilizzato in bioinformatica per riconoscere le sequenze nei geni, o anche dagli editor di testo. Il problema è semplice da risolvere ma c'è il rischio di una grande complessità computazionale: con un algoritmo banale si arriva ad una complessità di $O(nm)$, ma essa si può ottimizzare arrivando a un tempo di $O(n + m)$ (senza contare il tempo k per contare le occorrenze).

2.2 Algoritmi semi-numerici

Gli algoritmi semi-numerici hanno la caratteristica che in essi vengono trattati numeri, ma in realtà viene esaminata la loro rappresentazione binaria. Le operazioni sono effettuate su insiemi di bit, anche relativamente grandi. Le sequenze di bit possono essere interpretate anche come valori booleani.

2.2.1 Rappresentazione binaria

Esempio:

$$25 = 00011001$$
$$25 = 00011001 = FFFTTFFT \ (F = 0, T = 1)$$

Sfrutta il parallelismo implicito nell'usare operazioni della CPU.

2.2.2 Operazioni bit-level

Vengono trattate principalmente 5 operazioni base a livello di bit:

1. Or: $x \vee y$;
2. And: $x \wedge y$;
3. Xor: $x \oplus y$;
4. Left Shift: $x << k$,
   Prende i bit e li *sposta* di k posizioni a sinistra, scartando k bit dalla testa, e aggiungendo k zero (0) in coda (es.: $10111 << 2 = 11100$);
5. Right Shift: $x >> k$,
   Prende i bit e li *sposta* di k posizioni a destra, scartando k bit dalla coda, e aggiungendo k zero (0) in testa (es.: $10111 >> 2 = 00101$).

Tutte queste operazioni sono *bitwise* (a livello di bit), e tutte eseguite a livello di hardware.

2.3 Dömölki / Baeza-Yates, Gonnet

2.3.1 Un paio di considerazioni

Questo algoritmo di **string matching** viene comunemente chiamato **Shift-And**. Viene descritta però una semplificazione dello **Shift-And** (*bit parallel string matching*). Infatti i caratteri dell'alfabeto non verranno codificati con una bit mask, ma verrà semplicemente controllato il valore del carattere vero e proprio.

Lo scopo dell'algoritmo non è quello di ridurre il numero di operazioni (infatti la complessità rimane uguale all'algoritmo più efficiente che risolve lo stesso problema, cioè $O(nm)$), piuttosto cercare di ridurre al minimo il tempo che ci mette ogni singola operazione sfruttando la velocità delle operazioni sui bit.

2.3.2 L'algoritmo SPIEGATO DA DIO

Si definisce T (lunghezza $|T| = n$) la stringa, e P (lunghezza $|P| = m$) il pattern da matchare all'interno di essa.

Si ipotizza quindi di avere una matrice M di dimensione $m \times n$ (m righe, una per ogni lettera di P ed n colonne, una per ogni lettera di T), e due indici i, j, corrispondenti agli indici di P e di T. Vale il seguente:

$$M[i,j] = 1 \Leftrightarrow P[:i] = T[j-i+1:j]$$

Che in altre parole vuol dire: $M(i,j) = 1$ *sse* $P[:i]$ *(i primi i caratteri del prefisso) è uguale alla sottostringa di lunghezza* i *in posizione* $j - i + 1$.

Esempio 2.1

Usando $T = AGCAGBGCA,\ n = 9,\ P = GCA,\ m = 3$

			1	2	3	4	5	6	7	8	9
			A	G	C	A	G	B	G	C	A
M =	1	G	0	1	0	0	1	0	1	0	0
	2	C	0	0	1	0	0	0	0	1	0
	3	A	0	0	0	1	0	0	0	0	1

L'ultima riga, in particolare, mostra se il pattern è presente o meno nel testo: ci sarà 1 in corrispondenza di ogni occorrenza, precisamente nella posizione del carattere finale.

L'algoritmo funziona in questo modo: vengono definiti dei vettori $U[\sigma]$ lunghi $m = |P|$ per ogni $\sigma \in \Sigma$. $U[\sigma]$ avrà 1 in posizione i se $P[i] = \sigma$, 0 altrimenti.

Esempio: $P = cacao$, $U(c) = 10100$, $U(o) = 00001$, $U(x) = 00000$.

La prima colonna della matrice viene sempre inizializzata a 0, a eccezione del primo valore che sarà 1 se e solo se $T[1] = P[1]$.

Tutte le altre colonne dipendono dalla precedente in questo modo:

- La colonna i (con $i \geq 2$) corrisponde alla colonna $i - 1$ shiftata di una posizione;
- Il primo valore della colonna i è settato di default a 1;
- La colonna ottenuta in questo modo viene paragonata tramite AND a $U[T[i]]$;
- In questo modo ci sarà 1 solo nelle posizioni con caratteri di T e P uguali.

L'equazione di ricorrenza di questo algoritmo è

$$C[j] = ((C[j-1] >> 1) \vee (1 << (\omega - 1))) \wedge U[T[j]]$$

dove ω è la word size.

Le sottostringhe sono considerate uguali se il carattere corrente i è uguale al carattere j, e la sottostringa di T precedente è uguale alla sottostringa di J precedente. Ogni 1 nell'ultima riga della matrice corrisponde a un'occorrenza (carattere finale).

Alla fine dell'algoritmo, nella matrice risultante, risulterà una sottomatrice **identità** di dimensioni $m \times n$ per ogni occorrenza del pattern nel testo.

Se n non occupa una word intera, bisogna copiare l'ultimo bit della metà precedente sul primo 0 della metà successiva (come il riporto nell'addizione). Queste operazioni devono essere eseguite $(n/m)+1$ volte, infatti con m non troppo grande (una word) l'algoritmo è veloce. Al contrario, con m grande la complessità computazionale cresce.

3 Fibonacci

Esempi in Python di implementazioni dell'algoritmo di Fibonacci.

3.1 Soluzione ricorsiva

```
import argparse

parser = argparse.ArgumentParser(description='Fibonacci.')
parser.add_argument('limit', metavar='N', type=int, nargs=1, help='How␣many␣
    ↪ Fibonacci␣numbers␣to␣compute')
args = parser.parse_args()

def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

print(fib(args.limit[0]-1))
```

3.2 Soluzione iterativa

```
import argparse

parser = argparse.ArgumentParser(description='Fibonacci.')
parser.add_argument('limit', metavar='N', type=int, nargs=1, help='How␣many␣
    ↪ Fibonacci␣numbers␣to␣compute')
args = parser.parse_args()

def fib():
    a,b = 1, 1
    while True:
        yield a
        a, b = b, a+b

for index, f in enumerate(fib()):
    print(f)
    if index == args.limit[0]-1:
        break
```

Questa variante utilizza un iteratore, yeld ritorna il valore di *a* senza uscire dalla funzione. Potenzialmente vengono generati un numero infinito di valori. *enumerate* restituisce le coppie indice-valore da una lista.

4 Karp-Rabin

4.1 Introduzione

Karp-Rabin apprtiene alla categoria degli algoritmi probabilistici, i quali non danno la garanzia che il risultato sia corretto e con lo stesso input potrebbero esserci output diversi. Un altro importante elemento è un numero pseudo-casuale: proprio esso rende l'algoritmo non deterministico.

4.2 L'algoritmo

L'algoritmo di Karp-Rabin viene eseguito in tempo lineare, e si basa sulle moltiplicazioni come operazioni bit. A ogni carattere di una stringa viene associato un numero che va da 0 a 2^{m-1}. Tutti gli elementi sono considerati numeri binari di n bit, e vengono trasformati in un codice hash.

Questo algoritmo viene utilizzato per il pattern matching, in particolare per individuare casi di plagio di testi.

Stringa P, testo T, $\Sigma = \{0, 1\}$ Per una stringa di testo T, T_r sia la sottostringa di T di lunghezza n che inizia al carattere r.

Per la sequenza binaria P,

$$H(P) = \sum_{i=1}^{i=n} 2^{n-i} P(i)$$

Similmente,

$$H(T_r) = \sum_{i=1}^{i=n} T(r + i - 1)$$

Queste sono funzioni hash, trasformano una stringa di bit in un numero decimale (generalmente) univoco. La sliding window ha ampiezza m su T, in altre parole la finestra (lunga quanto la sottostringa) inizia dalla posizione 0 e si sposta a destra di una posizione per volta fino alle ultime posizioni.

Ogni intero può essere scritto in modo unico come somma di potenze positive di 2, e questo algoritmo trova le occorrenze paragonando le sequenze di bit dei due numeri. A ogni spostamento la fingerprint lunga m del testo viene ricalcolata, per poi eseguire il controllo se essa è uguale alla fingerprint della sottostringa P.

Il problema è: si riesce a rendere il calcolo della fingerprint più veloce a partire dalla fingerprint precedente? Si usa la seguente formula:

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

Il primo carattere è quello che pesa di meno (1), l'ultimo ha il peso più elevato (2^{m-1}) a causa del fattore moltiplicativo. A ogni spostamento il primo carattere (1) viene rimosso, il numero viene diviso per due e viene aggiunto l'n-esimo carattere della finestra elevato alla m.

Cambiando alfabeto è sufficiente cambiare la base, utilizzando la più piccola potenza di 2 maggiore o uguale a p (con p numero di simboli) per rendere più veloci le operazioni. Questo comporta la

perdita di univocità della stringa associata a una fingerprint insieme all'utilizzo di una quantità maggiore di memoria, p^n (irrilevante al momento).

Esempio: $P = 0101$, $n = 4$
$H(P) = 2^3 \cdot 0 + 2^2 \cdot 1 + 2^1 \cdot 0 + 2^0 \cdot 5$

Esempio: $T = 101101010$, $n = 4$, $r = 2$
$H(T_r) = 6$

C'è un'occorrenza di P che inizia in posizione R se e solo se $H(P) = H(T_r)$, quindi $P = T[i : i+m-1]$.

Uno dei maggiori problemi di questo algoritmo è la complessità computazionale. C'è una differenza tra il costo delle operazioni unitario (il costo di una semplice operazione) e il caso in cui i numeri considerati non sono polinomiali rispetto alla dimensione dell'input.

Se $H(P)$ e $H(T_r)$ sono numeri grandi, l'algoritmo è inefficiente a causa del rapido aumento delle potenze di 2, il cui numero di bit cresce esponenzialmente (costo logaritmico). Ogni operazione ha un costo proporzionale rispetto al logaritmo della somma dei due numeri.

Tempo migliore e medio: $O(n + m)$ (con la garanzia che i numeri sono piccoli).
Tempo peggiore: $O(nm)$.

Con i numeri grandi viene utilizzato il modulo P, in modo da ridurli. In questo modo diventa

$$2^{m-1}T[i + m] \ mod \ P$$

che viene calcolato iterativamente. Il modulo viene calcolato a ogni passo.
P è un numero primo casuale, quindi l'algoritmo diventa non deterministico. Il modulo viene applicato iterativamente all'ultima somma e non a tutta la formula per evitare l'elevata grandezza dei numeri (in questo modo ogni valore è minore di 2P).

I possibili errori sono falsi positivi FP in cui un'occorrenza non è vera, e i falsi negativi FN in cui un'occorrenza non viene trovata. Questi problemi sono causati dal fatto che non sempre lo stesso hash corrisponde alla stessa stringa. L'univocità è garantita solo se il modulo non viene utilizzato.

Questo algoritmo non può avere falsi negativi, cioè se una stringa non viene trovata nel testo è sicuro che essa non sia presente: il problema sono i falsi positivi.

La probabilità di errore è $P[\#FP \geq 1] \leq O(nm/I)$ se il numero primo P è scelto fra tutti i primi $\leq I$. Maggiore è l'insieme dei primi possibili, minore è la probabilità di errore.
Alcuni valori di I prefissati danno probabilità fisse, quindi l'analisi della probabilità di errore è accurata.

Per abbassare le probabilità di errore è anche possibile scegliere **k** primi casuali (indipendenti senza ripetizioni) e cambiare ogni numero primo dopo FP. I falsi positivi possono essere riconosciuti leggendo le sottostringhe carattere per carattere, aggiungendo un tempo computazionale di m. Questo funziona quando il numero di occorrenze che ci si aspetta di trovare è basso, ma elimina completamente i falsi positivi.
In pratica, i tempi di calcolo aumentano, ma la probabilità di errore diminuisce.

Con un numero primo, $T = O(n + m)$, $P[FP] = q$.
Con k numeri primi, $T = O(k(n + m))$, $P[FP] = q^k$.

Karp-Rabin quindi ha un tempo peggiore lineare, il che rende l'algoritmo utile per studiare altri problemi come il pattern matching in due dimensioni.

4.3 Classificazione degli algoritmi probabilistici

Il tempo di esecuzione di Karp-Rabin è essenzialmente sempre lo stesso indipendentemente dal numero casuale scelto, e il fattore probabilistico è l'errore. Questa tipologia di algoritmi è chiamata **Monte Carlo**: veloce ma forse non corretto.

La tipologia di algoritmi **Las Vegas** (quick sort) ha tempo variabile in base alla scelta del pivot probabilistica (casuale): se la scelta è "corretta" i tempi saranno migliori e viceversa, ma l'algoritmo è sempre corretto.

Se in Karp-Rabin viene effettivamente controllato ogni falso positivo, l'algoritmo rientra nella categoria Las Vegas. Generalmente ciò non viene fatto, perché non vale la pena di incrementare il tempo con una probabilità così bassa di errore.
Un altro modo di riconoscere i falsi positivi è considerare le sovrapposizioni, le quali sono distanti al massimo $m/2$ caratteri: eventuali valori fuori da questa ampiezza capitano solo in situazioni ristrette e vengono gestiti come semiperiodici (una parte ripetuta più volte, periodo), d è la lunghezza del periodo.

- $d = l_2 - l_1$;
- P semiperiodico con periodo d;
- $P = \alpha\beta^{k-1}$, α suffisso di β;
- Ogni run occupa $\leq n$ caratteri di T, ogni carattere di T è in massimo 2 run.

5 Comandi Unix

5.1 Essenziali

- `ls` vedere il contenuto di una directory;
- `cd` cambiare la directory corrente;
- `pwd` vedere la directory corrente;
- `cp` copiare un file;
- `mv` spostare o rinominare un file (o una directory);
- `rm` rimuovere un file;
- `cat` mostrare il contenuto di un file;
- `mkdir` creare un directory;
- `rmdir` eliminare una directory vuota;
- `man` mostrare la pagina di manuale di un comando.

5.2 Importanti

- `gzip`, `bzip2`, `xz` comprimere un file;
- `tar` creare, verificare, ripristinare un archivio;
- `grep` cercare un pattern in un testo;
- `head` mostrare le prime righe di un file;
- `tail` mostrare le ultime righe di un file.

5.3 Altri concetti

- `stdin` standard input, normalmente la tastiera;
- `stdout` standard output, normalmente lo schermo;
- `stderr` standard error,normalmente lo schermo;
- `.` standard input/output/error, possono essere un file;
- `>` redirezione di standard output (sovrascrive);
- `>>` redirezione di standard output (appende);
- `<` redirezione di standard input;
- `2>` redirezione di standard error;
- `|` pipe, lo stdout del primo programma diventa stdin del secondo.

5.4 Altri comandi

- `ssh` per collegarsi da un altro computer/server;
- `scp` per copiare un file da/a un altro computer;
- `rsync` come scp, ma copia solo se necessario;
- `make` per compilare solo quello che serve;
- `tr` cambiare alcuni caratteri in altri;
- `sort` ordinare un insieme di righe;
- `uniq` fondere le righe consecutive identiche;
- `cut` estrarre alcune colonne;
- `less` paginatore;
- `find` cercare file.

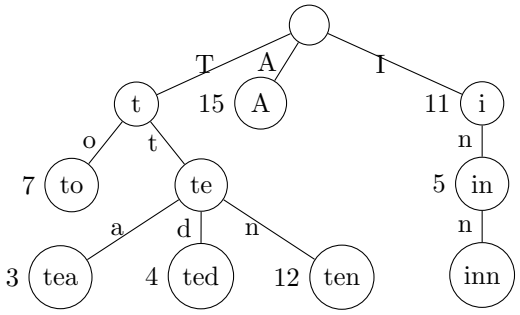
6 Suffix array e suffix tree

6.1 Trie

Definizione 6.1: Trie

I trie sono una struttura dati ordinata (un albero) utile per il pattern matching. *Sono caratterizzati da indicizzazione, l'assegnamento di un valore univoco con lo scopo di rendere le operazioni (come la ricerca) più veloci.* La forma più semplice di indice è un array ausiliario che contiene il puntatore agli elementi in ordine lessicografico, tramite il quale è possibile effettuare una ricerca dicotomica.

Figure 1: Trie



Definizione 6.2: Ricerca Dicotomica

Anche chiamata **Binary Search**, è un algoritmo di ricerca che opera selezionando due alternative (dicotomica) ad ogni step. È un particolare caso di algoritmo **Dividi e Conquista**.

Definizione 6.3: Self-Index

Il self-index è una struttura dati che viene utilizzata da sola, i dati sono contenuti all'interno di essa (esempio: array ordinato con puntatore alla posizione iniziale del numero).

Riassumendo, un **Trie** è:

- Un albero etichettato;
- Alfabeto ($\Sigma + \$$);
- **Terminatore**: $ (discriminante usato per determinare la fine di una parola $\in$ dizionario, esempio: **ABRA$**). È sempre una foglia dell'albero, senza **non è possibile** determinare la presenza di prefissi;
- Dizionario: insieme di parole $\in \Sigma$;
- Query di ricerca: la parola $\in$ dizionario?
- Archi etichettati (le etichette $\in \Sigma$).

Un trie può essere rappresentato:

1. Come **array**, dove le celle vuote immagazzinano predecessori e successori.
   Tempo: $O(P)$, spazio: $O(T\Sigma)$;
2. Come **Binary Search Tree** bilanciato.
   Tempo: $O(Plog\Sigma)$, spazio: $O(T)$.
3. Come **tabella hash**, anche se non supporta query con il predecessore e sorting.
   Tempo: $O(P)$, spazio: $O(T)$.
4. ...

Definizione 6.4: Compressed Trie

Un **Compressed Trie** contrae i cammini con un solo figlio in un solo nodo, e i relativi simboli nello stesso arco.

6.2 Suffix tree

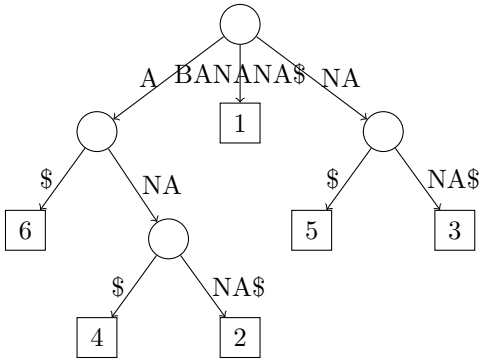
Un **suffix tree** è un compressed trie di tutti i suffissi di $T\$$ (più il simbolo di terminazione), dove **le foglie sono etichettate dalla posizione di inizio suffisso**. Si ricorda che un compressed tree ha ogni singolo cammino senza diramazioni rappresentato come un unico arco.

Le etichette degli archi uscenti da x iniziano con simboli diversi.

Definizione 6.5: Suffix tree

Un suffix tree per una stringa T di m caratteri ($|T| = m$) è un albero radicato con m foglie con indici da 1 a m. Ogni nodo interno oltre alla radice ha almeno due figli, e ogni arco è etichettato con una sottostringa non vuota di S. Due archi che non appartengono allo stesso nodo non possono iniziare con lo stesso carattere. Per ogni foglia i, la concatenazione delle etichette degli archi dalla radice alla foglia corrisponde al suffisso di S che inizia alla posizione i, $S[i \dots m]$.

Figure 2: Suffix Tree



Un suffisso implica un percorso radice-foglia (e viceversa).
Nella Figura 2 abbiamo i seguenti suffissi:

A$, ANA$, ANANA$, BANANA$, NA$, NANA$

I prefissi sono aggregati in un solo arco, e possono avere più suffissi. Le foglie sono etichettate con la posizione di inizio del suffisso, in modo da avere una corrispondenza $1:1$ tra le foglie e i suffissi. C'è al massimo un arco uscente per ogni simbolo dell'alfabeto.

Ci sono $|T+1|$ foglie. Ogni label è una sottostringa $T[i:j]$, è importante avere in memoria entrambi gli indici (i,j). Lo spazio di attraversamento dell'albero è $O(T)$, quindi gli alberi possono essere utilizzati per risolvere problemi di string matching in tempo lineare.

- Label(x): concatenazione delle sottostringhe che etichettano gli archi di un nodo;
- Path-label(x): concatenazione delle sottostinghe (etichette) contenute negli archi in un percorso dalla radice al nodo;
- String-depth(x): lunghezza (numero di caratteri) del path-label(x);
- Pattern matching: visita.

Un cammino che termina a metà di un arco (u,v) lo divide in un determinato punto. Il label di questo cammino è il label di u concatenato con i caratteri di (u,v) fino al punto di divisione.

Definizione 6.6: Suffix Tree Generalizzato

Un suffix tree generalizzato è un suffix tree derivante da un insieme di stringhe. Ogni stringa deve essere caratterizzata da un unico simbolo non appartenente all'alfabeto, per assicurarsi che nessun suffisso sia una sottostringa (e quindi ogni suffisso sia rappresentato da un'unica foglia).

6.2.1 Applicazioni

- Ricerca di una stringa P, il cui risultato corrisponde a tutte le occorrenze di P;
- Lista delle prime k occorrenze (foglie connesse tramite una linked list in cui ogni nodo punta al discendente sinistro);
- Ricerca della stringa più lunga che si ripete in T;
- Ricerca della sottostringa comune più lunga;
- Ricerca di stringhe palindrome.

6.2.2 Problemi

- Lo spazio esponenziale occupato dall'albero ($O(n^2)$), il che rende difficoltosa la ricerca in tempo lineare. Questo è causato dalla lunghezza delle etichette, e viene risolto memorizzando le posizioni di inizio e fine con un puntatore ($O(n)$?);
- La gestione delle posizioni dei puntatori al testo;

- Lo spazio di $20n$ bytes nel caso migliore e medio.

6.3 Pattern matching con suffix tree

Dato un pattern P di lunghezza n e un testo T di lunghezza m, si vogliono trovare tutte le occorrenze di P in T. I suffix tree permettono di affrontare questo problema con un approccio in tempo lineare che consiste nella costruzione dell'albero e nel matching dei caratteri.

Come si risolve il problema del pattern matching usando un suffix tree?
La ricerca in un trie permette di capire se una stringa è suffisso di un'altra. Bisogna estendere il problema alle sottostringhe, e capire le posizioni e il numero di occorrenze in cui essa si trova nel testo.

Si definisce la sottostringa in termini di suffisso. Un suffisso è sicuramente una sottostringa, e *ogni sottostringa è il prefisso di un suffisso*. Siccome ogni suffisso corrisponde a un percorso radice-foglia univoco, si devono cercare i percorsi iniziali.

Tutte le occorrenze del pattern P nel testo sono tutte le porzioni iniziali dei suffissi che iniziano con P. Per cercare stringhe a metà tra due archi, l'etichetta non viene consumata interamente, ma non ci sono errori.

I casi possibili sono:

1. La stringa non è presente in T;
2. Ogni foglia del sotto-albero sottostante l'ultimo match è enumerata con un punto di partenza di P in T, e ogni punto di partenza corrisponde a una foglia.

Dopo aver speso un tempo $O(m)$ per processare T, si riescono a trovare tutti i match di P in un tempo lineare $O(n + k)$, dove k è il **numero di occorrenze**.

Una volta trovata un'occorrenza, è sufficiente scorrere verso l'alto fino al primo suffisso che non inizia con P. Poi la ricerca riprende verso il basso.

Osservazione: ogni occorrenza di P in T è un prefisso di qualche suffisso in T.

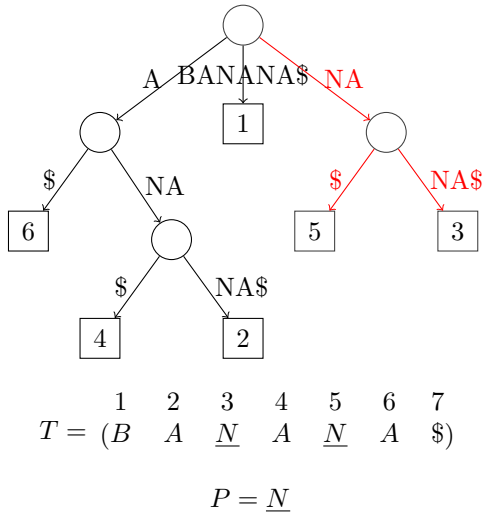
6.3.1 Procedimento

I passaggi sono i seguenti, partendo dal primo carattere del pattern e dalla radice dell'albero:

1. Per il carattere corrente del pattern, se c'è una corrispondenza con la stringa dell'arco, allora si prosegue verso il basso;
2. Se il pattern non esiste nell'arco corrente, allora esso non esiste in assoluto;
3. Si continua fino ad arrivare a una o più foglie dell'albero. Il valore delle foglie indicheranno **l'indice al quale si trovano le corrispondenze**.

6.3.2 Esempio

Figure 3: Esempio Matching Suffix Tree



Nell'esempio in *Figura 3* ci sono due corrispondenze agli indici $3, 5$, infatti $T[3] = N, T[5] = N$.

Il totale delle foglie discendenti una volta consumato il pattern corrisponde al numero k delle occorrenze.

È importante ricordare che a questi tempi va aggiunto il tempo per costruire l'albero ($O(n)$). Il numero di foglie incontrate è proporzionale al numero di archi attraversati (ogni nodo ha almeno 2 figli, e con k nodi si hanno k nodi interni al massimo).

6.4 Suffix Array

Definizione 6.7: Suffix Array

Un suffix array è l'array di interi dei suffissi ottenuti dal suffix tree ordinati lessicograficamente. Contiene **le posizioni iniziali del suffisso nell'array** e occupa solo $4n$ byte.

La principale utilità dei **Suffix Array** è il risparmio di tempo e spazio: memorizza le stesse informazioni del **Suffix Tree** [2] con maggiore località.

Definizione 6.8: Lcp

Lcp$[i]$ è un array ausiliario che contiene la lunghezza del prefisso comune più lungo per ogni coppia di suffissi consecutivi: $SA[i]$, $SA[i+1]$. Generalmente il più piccolo suffisso è la sentinella. La ricerca tramite array viene fatta usando *Lcp*.

Nella *Figura 4* viene costruito il **Suffix Array** partendo dalla stringa iniziale T.

Figure 4: Esempio Suffix Array

$$T = \begin{array}{cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ = & (b & a & n & a & n & a & \$) \end{array}$$

$$\text{Suffissi} = \begin{array}{c} \text{Suffix} \\ \begin{pmatrix} \text{banana\$} \\ \text{anana\$} \\ \text{nana\$} \\ \text{ana\$} \\ \text{na\$} \\ \text{a\$} \\ \text{\$} \end{pmatrix} \end{array} \begin{array}{c} i \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \xrightarrow{\text{Ordinati}} \begin{array}{c} \text{Suffix} \\ \begin{pmatrix} \text{\$} \\ \text{a\$} \\ \text{ana\$} \\ \text{anana\$} \\ \text{banana\$} \\ \text{na\$} \\ \text{nana\$} \end{pmatrix} \end{array} \begin{array}{c} i \\ 7 \\ 6 \\ 4 \\ 2 \\ 1 \\ 5 \\ 3 \end{array}$$

$$\text{SA} = \begin{array}{cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ = & (7 & 6 & 4 & 2 & 1 & 5 & 3) \end{array}$$

Figure 5: Lcp Example

$$\text{LCP} = \begin{array}{cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ = & (- & 0 & 1 & 3 & 0 & 0 & 2) \end{array}$$

Facendo riferimento all'esempio 4, la *Figura 5* rappresenta l'**Lcp** relativo al **SA** di **T**.

$LCP[0] = -$ perché \$ non ha, ovviamente, sottostringhe comuni con il suo elemento precedente (non esiste, duh).

$LCP[4] = 3$ perché **ana** è un prefisso di $\underline{ana}$\$ e di $anana$\$.

Nel pattern matching, tutte le occorrenze di P corrispondono a una porzione dell'array *Lcp*, dove gli elementi sono tutti minori o uguali della lunghezza del pattern. La scansione di *Lcp* permette di trovare il pattern in tempo costante.

Nella pratica, i suffix tree non vengono mai costruiti: prima si costruiscono i suffix array e da lì viene effettuata la trasformazione.

6.5 Da suffix tree a suffix array

L'algoritmo converte il tree in un array (quindi il tree non esiste più alla fine).
Si ricorda che, per ogni nodo in un suffix tree, si ha al massimo un arco che inizia con una lettera

e quindi un ordine non ambiguo. Una qualsiasi visita garantisce l'ottenimento di tutti i suffissi in ordine lessicografico, se i figli vengono visitati in ordine lessicografico.

Per ottenere un suffix array a partire dal suffix tree corrispondente bisogna effettuare la visita depth-first dell'albero ST, per poi salvare gli archi uscenti da ogni nodo in ordine lessicografico.

Per calcolare *Lcp* bisogna trovare il nodo più in basso che ha la proprietà di essere progenitore (*Lca*, Least Common Ancestor) di entrambe le foglie, e calcolarne la string-depth.
$Lcp[i]$ = string-depth di $Lca(i, i+1)$.

Dati due nodi di un albero, il percorso che li collega è unico: quindi *Lca* non è mai ambiguo. Questi calcoli possono essere eseguiti velocemente aggiungendo la string-depth tramite visita: il progenitore è il punto in cui finisce la discesa e inizia la salita nel percorso tra un nodo e un altro.

Il tempo è $O(n)$. Viene utizzato un algoritmo (tempo lineare) che si basa su merge sort e radix sort (Kärkkäinen), oppure *sais-lite*.

6.6 Da suffix array a suffix tree

La conversione da suffix array a suffix tree è più complessa, e viene eseguita ricorsivamente. Ha bisogno di *Lcp*, che serve per codificare la topologia dell'albero, mentre il suffix array definisce le etichette.

Si ricorda che la visita depth-first tocca le foglie esattamente nell'ordine del suffix array.

Se $Lcp = 0$ è una partizione di SA, le due stringhe corrispondono a figli della radice e non condividono nessun altro nodo. Ci sono tanti 0 quante coppie che non condividono nessun carattere, quindi i sottoalberi dovranno essere figli diversi della radice.

Sul primo sottoalbero c'è tutta la porzione a sinistra del primo 0; sul secondo, tra il primo e il secondo 0, e così via. Le porzioni di *Lcp* vuote sono foglie.

Ogni elemento minimo corrisponde a una partizione, i cui sottoalberi sono le classi. Esaurita una partizione, si procede ricorsivamente.

Se i suffissi non condividono nulla, il punto di inversione sarà la radice dell'albero. Se due sottostringhe hanno caratteri in comune, il nodo avrà il valore minimo della *Lcp* compresa tra le posizioni i.

In pratica, viene calcolato il minimo *Lcp* dell'albero o della porzione considerata, e l'albero viene diviso in $k+1$ regioni (dove k sono le occorrenze del numero). Alla fine della ricorsione ci saranno k regioni di un singolo elemento, le quali corrispondono a foglie che saranno collegate al rispettivo $SA[i]$.

Le etichette vengono assegnate tramite una visita depth-first con gli elementi del suffix array in ordine a partire dalle foglie. Il tempo resta lineare.

Riassumendo:

1. Ricerca del più piccolo valore di *Lcp* e conteggio delle occorrenze;
2. Divisione dell'albero in $k+1$ regioni, cioè sottoalberi;
3. Considerazione di ogni regione specifica compresa tra ogni coppia di occorrenze, e ricerca del minimo;

4. Assegnamento dei sottoalberi (considerando che \$ è sempre a sinistra, in base all'ordine lessicografico);
5. Nuove iterazioni fino all'esaurimento dell'array;
6. Visita depth-first e assegnamento delle etichette in ordine a ogni foglia.

6.7 Sottostringa comune più lunga

Date due stringhe s_1 e s_2, questo algoritmo ricerca la sottostringa comune più lunga utilizzando un suffix tree generalizzato (un insieme di stringhe). Ogni stringa rappresenta un suffisso di una delle due stringhe (o entrambe).
Il testo ha la forma $ST(s_1\$_1 s_2\$_2)$, cioè i due testi vengono concatenati. Viene usato un singolo terminatore e viene costruito l'albero.

A ogni foglia è associata una coppia di valori: la stringa di riferimento e la sua posizione di partenza. Tutti i suffissi che appaiono in entrambe le stringhe (i nodi in cui *Lcs* termina) hanno due coppie di valori.

Ogni nodo interno viene etichettato con 1, 2 o entrambi. Ogni path-label con sia 1 che 2 è una sottostringa comune, resta il problema di trovare il massimo path-label tra essi.

Viene cercato un nodo x con foglie di s_1 e s_2, in modo da avere $ST(s_1\$s_2\$)$, tale che string-depth risulti il massimo. Essa viene calcolata sommando la string-depth del padre e la lunghezza dell'etichetta che li collega.

L'albero viene visitato in post-order: in questo modo è possibile confrontare il contenuto dei figli con il contenuto del padre. Le visite sono due, una per identificare le sottostringhe comuni e una per trovare quella di lunghezza massima.

La costruzione dell'albero e la ricerca hanno entrambi tempo lineare.

6.8 Pattern matching con suffix array

Per trovare un pattern P con lunghezza $|P| = m$ avendo a disposizione un suffix array A, si potrebbe utilizzare la **ricerca dicotomica** sfruttando l'ordinamento lessicografico. Essa si basa sui singoli caratteri dell'array e le stringhe comprese tra gli indici.

Partendo dal presupposto che la ricerca dicotomica usa le posizioni L (left) e R (right), in base all'ordine si ha che a ogni iterazione $R > P > L$, dove P è il pattern. Se P è maggiore della media M tra L e R, la ricerca sarà nel sottoalbero sinistro, e viceversa.

Questo algoritmo ha tempo logaritmico, approssimabile a $O(mlogn)$: è possibile arrivare a $O(m)$?

La soluzione è utilizzare il **self-index**, menzionato precedentemente (indice che punta a se stesso). Vengono usati **T**, il suffix array **SA** e **LCP**.

I suffissi che contengono P sono consecutivi: tutti iniziano con P. Per accelerare il pattern matching si possono utilizzare tre alternative, definite **acceleranti**.

6.9 Acceleranti

6.9.1 Accelerante 1

Tutti i suffissi nell'intervallo $SA(L, R)$ iniziano con lo stesso prefisso lungo $\mathbf{Lcp}(SA[L], SA[R])$: **LCP** ha la stessa lunghezza in prefissi successivi, quindi è sufficiente controllare il primo carattere dopo il pattern comune. Se due suffissi condividono i primi x caratteri, tutti i suffissi tra essi condividono x caratteri.

Si parte da $L = 1$ e $R = m$, dove m è la lunghezza della stringa T. Poi ad ogni iterazione della ricerca dicotomica si fa una query alla posizione $M = [(R+L)/2]$ di SA. Si definisce quindi $mlr = \min(l, r)$, dove l, r sono le lunghezze dei prefissi in posizione L, R rispettivamente.

Il valore di mlr può essere usato per accelerare il confronto lessicale di P ed il suffisso in $SA[M]$. Dal momento che SA fornisce l'ordine lessicale dei suffissi di T, se i è un indice qualunque tra L e R, i primi mlr caratteri del suffisso $SA[i]$ devono essere gli stessi dei primi mlr caratteri del suffisso $SA[L]$, e di conseguenza del pattern P. Dunque, il confronto lessico tra P e $SA[M]$ può iniziare dalla posizione $mlr + 1$ delle due stringhe, invece di dover iniziare dalla prima posizione.

Alla prima iterazione per ottenere l, r, mlr si forza la comparazione di P con il suffisso $SA[1]$ ed il suffisso $SA[M]$.

Questo algoritmo però ha come caso peggiore comunque $O(nlogn)$.

6.9.2 Accelerante 2

Si definisce *ridondante* una verifica di un carattere di P se il carattere è già stato analizzato in precedenza. Lo scopo degli acceleranti è quello di **ridurre** il numero di analisi di caratteri ridondanti riducendole ad al massimo a **una per iterazione** della ricerca dicotomica - quindi $O(logm)$.

L'utilizzo di solamente mlr non permette di ottenere questo risultato. Dal momento che mlr è il *minimo* tra l e r (che ricordiamo essere la lunghezza dei prefissi L, R), quando $l \neq r$, tutti i caratteri in P a partire da $mlr + 1$ fino al massimo di l e r saranno già stati esaminati.

Ogni confronto di quei caratteri sarà ridondante. Ciò di cui si ha bisogno è di iniziare il confronto lessicografico al *massimo* tra l e r.

Viene calcolato il numero di caratteri condivisi tra il primo (ultimo) elemento dell'array e il pattern. Siano $l = Lcp(L,\ P)$ e $r = Lcp(R,\ P)$. A ogni iterazione, l e r vengono ricalcolati.

Si ha che l e r non sono mai maggiori di M, da cui consegue un tempo massimo di $O(M)$ ottenibile approssimando $O(log_2 N) + O(N) + O(M)$.

1. Tutti i suffissi nell'intervallo $SA(L,\ R)$ iniziano con lo stesso prefisso lungo $Lcp(SA[L],\ SA[R])$: *Lcp* ha la stessa lunghezza in prefissi successivi, quindi è sufficiente controllare il primo carattere dopo il pattern comune. Se due suffissi condividono i primi x caratteri, tutti i suffissi tra essi condividono x caratteri;
2. Viene calcolato il numero di caratteri condivisi tra il primo (ultimo) elemento dell'array e il pattern. Siano $l = Lcp(L,\ P)$ e $r = Lcp(R,\ P)$. A ogni iterazione, l e r vengono ricalcolati. Si ha che l e r non sono mai maggiori di M, da cui consegue un tempo massimo di $O(M)$ ottenibile approssimando $O(log_2 N) + O(N) + O(M)$;

(a) Caso 1, $l > r$:

i. $Lcp(L,\ M) > l$, il pattern sarà tra M e R;

ii. $Lcp(L,\ M) < l$, il pattern sarà tra L e M;

iii. $Lcp(L,\ M) = l$, è necessario controllare il carattere successivo.

(b) Caso 2, $l = r$:

i. $Lcp(L,\ M) > l$, il pattern sarà tra M e R;

ii. $Lcp(M,\ R) > l$, il pattern sarà tra L e M;

iii. $Lcp(L,\ M) = l$, è necessario controllare il carattere successivo.

(c) Caso 3, $l < r$: simmetrico al caso 1.

6.9.3 Accelerante 3

Calcolo di *Lcp* in tempo $O(n)$, con n intervalli. Ogni intervallo ha punti di inizio e di fine determinati, che vengono calcolati ricorsivamente tramite iterazioni e poi aggregati.

Si ha che, nell'accelerante 2, i suffissi si trovano negli estremi o nei punti mediani della ricerca dicotomica. L'accelerante 3 consiste in due osservazioni:

- Il numero delle coppie non è arbitrario e non è quadratico, ma lineare (si dimezza a ogni iterazione);
- Anche *Lcp* può essere calcolato in tempo lineare. Con intervalli di ampiezza 2, il valore di $Lcp(SA[i], SA[i+1])$ è contenuto nell'array iniziale.

In generale, gli intervalli hanno ampiezza 2^i con $i > 1$. Il minimo può essere nella prima o seconda metà, oppure nel mezzo.

Il calcolo di *Lcp* ha tempo $O(n)$, con n intervalli. Ogni intervallo ha punti di inizio e di fine determinati, che vengono calcolati ricorsivamente tramite iterazioni e poi aggregati.

1. Iterazione 1: $(L, R) = (1, M)$;
2. Iterazione 2: $(L, R) = (1, m/2)$ oppure $(m/2, n)$;
3. Iterazione k: $L = h\frac{m}{2^{k-1}},\ R = L +\ Lcp(h,\ h+1)$;
4. Iterazione $\lceil log_2 m \rceil$: $R = L + 1, Lcp(h,\ h+1)$;
5. Iterazione $\lceil log_2 m \rceil\ -\ 1$: aggregazione dei risultati dell'iterazione $\lceil log_2 m \rceil$;
6. Iterazione k: $Lcp(h\frac{m}{2^{k-1}},\ (h+1)\frac{m}{2^{k-1}})$.

6.10 Considerazioni sugli acceleranti

L'accelerante 1 si basa sul fatto che gli estremi abbiano un suffisso comune dato appunto da *Lcp*, e questa porzione non va controllata; per il resto è molto simile alla ricerca dicotomica.

L'accelerante 2 consiste nella ricerca ricorsiva, simile a quella dicotomica ma usando i valori di *Lcp*: sfruttando l'ordine lessicografico, è possibile riconoscere la posizione di interi pattern e suffissi. Introducendo nell'analisi le lunghezze di *Lcp* in relazione con il pattern, è possibile risolvere alcuni casi in tempo costante evitando il confronto tra alcune coppie.

Nell'accelerante 2 ci sono $\binom{n}{2}$ possibili coppie: questi valori non vengono memorizzati.

Al contrario, l'accelerante 3 definisce un minor numero di coppie e un calcolo più veloce (ricorsivo). Esso consiste in un preprocessamento iniziale dove vengono calcolate tutte le coppie di suffissi che servono, cioè gli estremi oppure estremi e mediano di una ricerca dicotomica. Vengono anche calcolati i *Lcp*.

All'iterazione k, si ha che i due intervalli soddisfano la relazione definita precedentemente. Dopo un numero logaritmico di iterazioni, gli intervalli sono formati da due elementi consecutivi, e il loro numero può essere rappresentato da una sommatoria geometrica (lineare).

Gli intervalli dell'ultima iterazione sono precalcolati e memorizzati nell'array, quindi è sufficiente aggregarne due consecutivi. Si ha che $Lcp(l_1,\ r_2) \leq Lcp(l_1,\ r_1)$ e $Lcp(l_1,\ r_1) \leq Lcp(l_2,\ r_2)$: i due intervalli saranno della forma αx e αy, dove α è la stringa comune.

Il cambiamento di lettera (passaggio da x a z o qualsiasi altra lettera maggiore) può avvenire prima della metà, dopo la metà o esattamente a metà. Se la transizione avviene nella prima metà, $Lcp(l_1,\ r_2) = Lcp(l_1,\ r_1)$; se avviene nella seconda, $Lcp(l_1,\ r_1) \leq Lcp(l_2,\ r_2)$; altrimenti, $Lcp(l_1,\ r_1) \leq Lcp(r_1,\ l_2)$.

In altre parole,

$$Lcp = min \begin{cases} Lcp(l_1,\ r_1) \\ Lcp(l_2,\ r_2) \\ Lcp(r_1,\ l_2) \end{cases}$$

La maggiore differenza tra gli acceleranti (che diventano via via più sofisticati) e l'algoritmo di ricerca dicotomica è appunto l'uso di *Lcp*, che permette operazioni in tempo lineare.

6.11 Implementazione

L'implementazione proposta ha una variabile che rappresenta il tipo di accelerante da utilizzare, dove 0 di default indica la semplice ricerca dicotomica.
La scelta dell'algoritmo è tramite un costrutto in C che assegna a una variabile *search_f* (dichiarata come puntatore a una funzione) una diversa funzione, predefinita in base all'opzione.

L'indice del suffisso che inizia con il prefisso (la prima occorrenza) è salvato in una variabile $found$. Per trovare le occorrenze successive si confrontano tutti i *Lcp* successivi.

C'è una funzione di appoggio *suffix_cmp* (dove *cmp* = compare) che restituisce un numero minore di 0 se la prima stringa è minore della seconda; maggiore di 0 se è maggiore, 0 altrimenti. Si ferma al primo carattere diverso e restituisce il valore di $X_i - Y_i$ convertendo il carattere in intero.

Il calcolo dell'elemento mediano è $(l + (r - l))/2$ invece che $(l + r)/2$ per evitare l'overflow in caso l e r siano numeri molto grandi.

Un'ulteriore funzione di appoggio calcola *Lcp* date due stringhe generiche: questa viene utilizzata nelle acceleranti 1 e 2 (e 3, ma essa non viene trattata). Questa si basa su un ciclo for con un contatore che viene incrementato se i caratteri sono uguali.

I puntatori permettono di considerare l'intera porzione di stringa compresa tra ogni puntatore e la fine, ma possono essere incrementati in modo semplice.

Il secondo accelerante ha circa la stessa struttura degli altri, solo che ci sono numerosi casi che devono essere controllati.

7 Sottostringa comune di k stringhe

Si ricorda che l'algoritmo per trovare la sottostringa comune di due stringhe risolve il problema in tempo lineare, utilizzando un suffix tree generalizzato.

7.1 Variante con suffix tree

L'obiettivo è estendere l'algoritmo conosciuto con k stringhe di lunghezza complessiva v, e tradurre l'albero dei suffissi in un array. La costruzione dell'albero è la stessa, e le visite indicano quali stringhe hanno almeno un suffisso sotto ogni nodo interno.

Si ha che, in un suffx tree generalizzato, a partire da un nodo x, esso deve avere come discendente almeno una foglia per ogni stringa dell'insieme. Se questa condizione è verificata, allora il tratto dalla radice a x è comune a tutte le stringhe.

Tra tutti, viene scelto il nodo di string-depth massima che rispetti questo vincolo, ovvero quello con il cammino più lungo dalla radice a x.

Ogni foglia rappresenta un suffisso di una delle k stringhe, ed è distinta dalla propria stringa di provenienza (identificatore). Per ogni nodo, viene memorizzato il numero di identificatori che appaiono nelle foglie ($C[i]$).
Le foglie hanno solo un identificatore a causa dei diversi simboli terminali.

Un'alternativa è un array A lungo k di valori booleani per ogni nodo, dove $A[i]$ vale T se e solo se x è antenato di una foglia che rappresenta un suffisso di s_i.

Per ottenere la sottostringa più lunga, la visita costruisce un array V che contiene la string-depth del nodo più in fondo tale che il numero di identificatori associati sia k, per ogni $k \in \{1, \ldots, v\}$.

Riassumendo (alternativa con array booleani):

1. Costruzione del suffix tree generalizzato;
2. Visita dell'albero per etichettare ogni vertice con l'insieme delle stringhe di cui un suffisso etichetta un discendente (post-order);
3. Per ogni nodo, esiste un array di k booleani in cui ogni posizione indica se esiste un suffisso della i-esima stringa tra i discendenti;

4. L'array di ogni nodo, quindi, è l'*or* di tutti gli array dei figli;
5. L'albero viene visitato ancora per capire quali nodi prendere in considerazione;
6. Viene eseguita un'ultima visita per trovare la string-depth massima.

Il tempo non è più lineare: diventa $O(k^h)$ dove h è la somma della lunghezza delle stringhe.
In caso di stringhe tutte lunghe n, si ha $O(k^2 n)$: il fattore k deriva dalla computazione del numero di identificatori, salvati in un altro array C. Le altre due visite hanno tempo lineare.

Esistono algoritmi in tempo migliore, ma sono molto complessi.

7.2 Variante con suffix array

Come si traduce il suffix tree in suffix array?
Ogni foglia corrisponde a un elemento del suffix array, e c'è una corrispondenza 1 : 1 tra gli elementi (in ordine lessicografico) e le foglie. Un nodo interno corrisponde a un determinato intervallo (porzione contigua) del suffix array; non c'è una corrispondenza 1 : 1.

Ogni nodo interno del suffix tree corrisponde a un insieme di suffissi che condividono una porzione iniziale nel suffix array, quindi suffissi contigui.

Il suffix array generalizzato è diviso in due parti: una contiene gli indici delle posizioni, e una contiene l'indice della stringa del suffisso.

L'algoritmo di costruzione del suffix array generalizzato riceve in input un insieme di stringhe; usa una variabile *len* con la lunghezza complessiva di tutte le stringhe, e poi procede normalmente. Vengono utilizzati k terminatori diversi.

Ogni intervallo del suffix array associato a un nodo del suffix tree corrisponde a tutti e soli i suffissi che iniziano con il percorso dalla radice a quel nodo.

Gli intervalli associati a un nodo e il padre sono uno un sottoinsieme dell'altro. Se due nodi sono foglie, i sottoinsiemi sono disgiunti (così come gli intervalli).

L'algoritmo trova, per ogni nodo, se contiene suffissi delle stringhe di ingresso, e tra essi quello con string-depth massima. Vengono calcolati tutti gli intervalli, controllando di avere almeno un suffisso della stringa d'ingresso, e poi scelto il massimo.

Il numero di intervalli sarebbe $\binom{n}{2}$, che è troppo elevato per poterli calcolare tutti; bisogna quindi restringere il campo: si cerca quello con *Lcp* massimo tra quelli che contengono almeno un suffisso per ogni stringa.

Dato un suffix array SA, con il relativo *Lcp*, si considerano tutti gli intervalli che finiscono in una determinata posizione j (per ogni j da 1 a n, ma non è conosciuto il punto di inizio). L'obiettivo è trovare l'intervallo che corrisponde alla più lunga sottostringa comune di tutto l'input, fra quelli che soddisfano due proprietà:

- Devono contenere tutti gli indici da 1 a k: questo viene controllato in modo veloce associando un numero intero per ogni stringa in un array ausiliario, che indica la posizione dell'ultima occorrenza di un suffisso. In questo modo, con k interi è possibile testare tutti gli intervalli e sapere che sono accettabili se e solo se il punto di inizio è minore o uguale del minimo dell'array;

  - Es: $prev = [57, 99, 98, 60]$, l'intervallo $[80, 99]$ non contiene nessuna sottostringa comune perché l'ultima occorrenza di $S1$ è in posizione 57;

- Per ogni intervallo in posizione j, si controlla solo l'intervallo j dato che tutti quelli con indice minore contengono i successivi;
  - Si ha che il *Lcp* dell'intervallo più piccolo è sempre maggiore o uguale del *Lcp* dell'intervallo più grande. Dato un intervallo, la lunghezza della sottostringa comune è il minimo *Lcp*: quindi, se un intervallo contiene un altro, *Lcp* dell'intervallo maggiore sarà minore o uguale. Pertanto, sarà inutile controllare i sovrainsiemi.

L'implementazione in C è una traduzione diretta di questi step.

Per ogni j, viene aggiornato l'array *prev* in tempo costante, e calcolato il minimo in tempo k. Poi bisogna trovare *Lcp* minimo: il tempo dell'operazione dipende dall'ampiezza.
Alla fine, l'algoritmo trova la stringa comune più lunga in tempo $O(nlogn)$, utilizzando l'algoritmo Range Minimum Query.

8 Range Minimum Query

Il problema *Range Minimum Query* si occupa di trovare il minimo tra due qualsiasi intervalli.
Lo scopo di questo algoritmo è calcolare il minimo *Lcp* nel problema del pattern matching con suffix array.

Si ha in input un array A di interi, con cardinalità n. Viene richiesto di:

1. Indicizzare l'array A, tempo $O(nlogn)$;
2. Rispondere a query, tempo $O(1)$:
   - Dati i, j (due numeri qualsiasi), calcolare il minimo nella porzione di array A fra indice i e indice j compreso: $min_{i \leq z \leq j}\{A[z]\}$.

Non è possibile calcolare tutti i minimi di ogni intervallo (il tempo di indicizzazione sarebbe troppo), ma non è nemmeno possibile rispondere troppo lentamente alle query: è necessario trovare un bilanciamento, preferibilmente in tempo lineare.

L'indicizzazione è in tempo $O(nlogn)$ in cui il logaritmo è un fattore moltiplicativo. Il preprocessamento (non ottimale) viene eseguito *una tantum* in modo da permettere una risposta in tempo costante. Esso consiste nel calcolare degli intervalli, e su questi andare a cercare il minimo.

8.1 Preprocessamento

Consiste nel calcolare degli intervalli, e su essi andare a trovare il minimo.

Viene costruito un array bidimensionale B, indicizzato da x, y, che conterrà il minimo della porzione dell'array A, che inizia con x e finisce con $x + 2^y - 1$.
Quindi: $A[x : x + 2^y - 1]$, l'inizio è in x e l'ampiezza è 2^y, e vengono calcolati solo gli intervalli che corrispondono a una potenza di 2. Questa matrice avrà $nlogn$ elementi.

- Caso base: $y = 0$ ha come minimo $A[x]$, intervallo ampio 1 perché contiene solo x, $A[x : x + 1 - 1]$;
- $B[x, y] = min(B[x, y - 1], B[x + 2^{y-1}, y - 1])$, si confrontano il minimo della prima e quello della seconda metà.

Dopo aver indicizzato gli intervalli, si ricerca in tempo costante qual è il minimo tra tutti gli intervalli compresi tra i, j. Con due intervalli sovrapposti, si confrontano semplicemente i due minimi dei rispettivi intervalli.

Il più grande intervallo precalcolato in B è: w = la più grande potenza di $2 \leq j - i + 1 \Rightarrow \lfloor log_2(j - i + 1) \rfloor$. Si ha che, con $|A| \geq 1$, $0 \leq j \leq \lceil log_2 |A| \rceil$.

In pratica: si costruisce B, quindi si trova il minimo di un certo intervallo utilizzando la matrice B.

8.2 Implementazione

L'algoritmo in C è la semplice implementazione di quanto presentato.
La prima modifica è l'inizializzazione di *__matrix__*, che viene mantenuta in modo tale da poter riutilizzarla quando necessario. La scelta implementativa è tra due intervalli di lunghezza diversa o due che si sovrappongono: è meglio il secondo caso, perché permette il semplice confronto del minimo.

Per evitare "buchi", è meglio suddividere in intervalli grandi (è meno probabile) cioè della più grande potenza di 2 minore o uguale della lunghezza di $j-i+1$. In termini matematici, $w = \lfloor log_2(j-i+1) \rfloor$.

Dimostrazione per assurdo: si suppone che ci sia un buco, quindi i due intervalli sono disgiunti e c'è un elemento nel mezzo. Perché ciò sia possibile, l'ampiezza dell'intervallo dev'essere maggiore di w, quindi della più grande potenza di 2, che è un controsenso.

I valori trovati vengono poi confrontati in tempo lineare.

9 Allineamento di 2 sequenze

Un problema fondamentale in bioinformatica è quello del confronto di due sequenze biologiche (proteine, genomi, geni...) per capire quante esse siano simili. Un possibile alfabeto in questo caso è $\Sigma = \{A,\ C,\ T,\ G\}$.

Il dogma centrale della biologia molecolare enuncia che se due sequenze sono simili (come struttura), hanno anche una funzione simile. Questo è il motivo per cui è molto utile capire le omologie. Un altro ambito in cui ciò viene applicato è il confronto di specie dal punto di vista evolutivo.

Il problema computazionale relativo è definito allineamento: il pattern matching non è sufficiente, perché in questo caso si vuole riconoscere la somiglianza.

Si introduce il concetto di **distanza di Hamming** (numero di caratteri differenti nella stessa posizione tra due stringhe): per "distanza" si intende una funzione $d : S \times S \to \mathbb{R}^+$, con $S \times S$ insieme delle coppie di stringhe. Il calcolo di d è molto semplice, ed è possibile utilizzarla per confrontare delle sequenze.

Esempio: al**b**ero, al**t**ero, distanza di Hamming = 1.

Proprietà:

1. Riflessività: $d(x, y) = 0 \Leftrightarrow x = y, \forall\ x, y \in S$;
2. Simmetria: $d(x, y) = d(y, x), \forall\ x, y \in S$;
3. Disuguaglianza triangolare: $d(x, y) + d(y, z) \leq d(x, z), \forall\ x, y \in S$.

La distanza di Hamming ha un difetto: è definita solamente su stringhe di **uguale lunghezza**, un problema in bioinformatica quando si vorrebbe poter confrontare stringhe di lunghezza diversa. È quindi un modello estremamente semplificato, ridotto, del concetto di confronto di sequenze.

9.1 Allineamento

Anche chiamato problema di allineamento ottimo **globale**, perché le stringhe vengono analizzate nella loro interezza.

Se le due stringhe avessero la stessa lunghezza (come nella Distanza di Hamming), si potrebbe fare un confronto colonna per colonna: ma non è così, quindi si cerca un algoritmo per ricondursi a questo caso.

Per riempire i buchi vengono inseriti degli spazi, (**indel**, dove **del** indica un carattere cancellato; **in**, un carattere inserito). Gli indel devono rispettare le seguenti proprietà:

- Non possono esistere colonne solo di **indel**;
- Le stringhe estese devono avere la stessa lunghezza.

Esempio 9.1

$s_1 = ABRACADBRA, s_2 = BANANA$.
Alcuni possibili allineamenti con indel sono:

A	B	R	A	C	A	D	A	B	R	A
-	B	-	A	N	A	-	-	-	N	A

A	B	R	A	C	A	D	A	B	R	A
-	-	-	B	-	A	N	A	-	N	A

A	B	R	A	C	A	D	A	B	R	A
-	B	A	N	A	-	-	-	-	N	A

Esistono innumerevoli modi per inserire questi spazi, tutti validi perché rispettano le proprietà definite prima. Il fatto che siano tutti e 3 allineamenti validi, non vuol dire che siano equamente *buoni*.

9.2 Problema di ottimizzazione

Formalmente si definisce problema di ottimizzazione:

- Insieme delle istanze (una coppia per esempio è un'istanza), un numero infinito di casi;
- Soluzioni ammissibili: ammissibilità verificabile in tempo **polinomiale**;
- Funzione obiettivo: Istanza $\rightarrow \mathbb{R}^+$;
- Soluzione che *massimizza* (**valore**) o *minimizza* (**costo**) la funzione obiettivo.

Avendo l'istanza e l'insieme delle soluzioni ammissibili, è necessario definire il valore dell'allineamento delle sequenze incolonnate, che andrà massimizzato. Esso è la somma dei valori delle singole colonne, e dev'essere una buona misura dell'omologia.

Il valore di una colonna viene calcolato ricevendo in ingresso una matrice, e associando a ogni coppia di elementi un numero. Esistono delle matrici (es. BLOSUM62) di *scoring* con valori predefiniti per ogni singola colonna possibile.

Tecnicamente quindi, se si confrontano due stringhe random, si otterrà un valore negativo (a seconda di come è costruita la matrice in ingresso). Con due stringhe simili tra di loro, o tante colonne con caratteri uguali, ci saranno dei valori positivi. Tanto più è alto il valore dell'allineamento, tanto meno è probabile che le due stringhe siano incorrelate o casuali.

Il problema quindi diventa: per ogni allineamento possibile, scegliere quello con il massimo valore. Se le stringhe sono effettivamente simili e l'allineamento è ottimale, ci saranno tanti contributi positivi. I caratteri uguali devono essere sempre allineati.

La soluzione sfrutta il concetto di **Programmazione Dinamica**.

9.3 Needleman-Wunsch: equazione di ricorrenza

L'algoritmo usa le soluzioni dei sottoproblemi per trovare la soluzione del problema corrente. Ci sono 3 casi da confrontare: indel in s_1, indel in s_2 e nessun indel (quando i caratteri sono uguali).

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

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

Si ricorda che non può esistere il caso di due indel sulla stessa colonna.

Condizioni di contorno:

- $M[0,0] = 0$;
- $M[i,0] = M[i-1,0] + d(s_1[i], -)$;
- $M[0,j] = M[0,j-1] + d(-, s_2[j])$.

Una volta trovato il valore ottimo, per trovare l'allineamento corrispondente è sufficiente memorizzare lo step per arrivare alla casella successiva e tornare indietro a partire dall'ultima.

Per costruire questa matrice M bastano due cicli for innestati banali. Il tempo computazionale è pertanto $O(nm)$.

10 Allineamento locale

Nel caso in cui due sequenze abbiano una parte limitata simile e tutto il resto diverso, il valore complessivo di omologia è relativamente basso: l'allineamento globale non riuscirebbe a evidenziare il tratto altamente omologo.

Dato che lo scopo di confrontare le sequenze è anche funzionale, avere due regioni fortemente simili potrebbe essere interessante: questa problematica è definita allineamento locale.

Dal punto di vista formale, l'input è lo stesso: due stringhe s_1, s_2, una matrice di score d, ma l'algoritmo individua le sottostringhe t_1 e t_2 tale che $All[t_1, t_2] \geq All[u_1, u_2]$ per ogni coppia di sottostringhe u_1, u_2 di s_1, s_2.

Questo concetto deriva dal fatto che lo score è positivo o negativo in base al valore assegnato alla somiglianza. Tutte le parti non simili hanno un contributo fortemente negativo, quindi confrontare i valori delle sottostringhe è utile per trovare ciò che interessa.

Una strategia consiste nella ricerca dei *seed*, cioè sottostringhe comuni. Il problema è che in questo caso le sottostringe potrebbero essere simili, non necessariamente comuni, e il procedimento fallisce in rare occasioni.

L'approccio brute-force calcola tutte le sottostringhe e le confronta: con $|s_1| = n$, $|s_2| = m$, si ha un tempo di $O(n^3 m^3)$, dove n^2 e m^2 sono il numero di sottostringhe e il restante è il tempo di confronto. Ovviamente questa soluzione non è ottimale.

Il tempo minimo raggiungibile per una soluzione ottima è $O(nm)$.

10.1 Variante con i prefissi

Una variante dell'algoritmo utilizza i prefissi: cerca i prefissi t_1, t_2 di s_1, s_2 tali che $All[t_1, t_2]$ sia il massimo.

Viene ripreso Needleman-Wusch: la matrice usata per l'allineamento globale contiene l'allineamento ottimo di tutti i prefissi di s_1 e s_2. I valori temporanei sono sfruttati cercando il massimo di essi. La matrice deve anche contenere la cronologia degli spostamenti, in modo da poter seguire a ritroso il percorso.

Riassumendo, la matrice $M[i, j]$ memorizza l'allineamento di tutte le coppie di prefissi, il cui massimo è appunto il massimo di M.

$M[0, 0] = 0$. quindi non ci sono sottostringhe con allineamento negativo.

Il problema è che vengono trovati tutti i prefissi, non tutte le sottostringhe. Questo algoritmo ha un tempo di $O(n^2m^2)$.

10.2 Smith-Waterman

Un'altra variante cerca $M[i, j]$, l'allineamento globale ottimo tra tutte le sottostringhe che finiscono nelle posizioni i e j, cioè $t_1 = s_1[\cdot, i]$ e $t_2 = s_2[\cdot, j]$.

Siano h e k gli indici iniziali delle sottostringhe t_1 e t_2, essi sono incogniti.

Siano $t_1 = s_1[h : i]$, $t_2 = s_2[k : j]$ tali che t_1 e t_2 sono due sottostringhe di s_1 e s_2 rispettivamente che terminano in posizione i e j, con massimo valore di $All[t_1, t_2]$. Inoltre, l'allineamento ottimo di t_1 e t_2 presenta un'ultima colonna senza indel.

Allora le altre colonne sono l'allineamento ottimale di $s_1[h, i-1]$, $s_2[k, j-1]$, che è anche l'allineamento ottimale di due sottostringhe di s_1, s_2 che terminano in $i - 1$, $j - 1$.

Nell'ultima colonna, logicamente, ci sono l'i-esimo carattere di s_1 e il j-esimo di s_2. La porzione a sinistra è quindi l'allineamento tra $[h, i - 1]$ e $[k, j - 1]$. Viene dimostrato che $s_1[h, i - 1]$, $s_2[k, j - 1]$ è la migliore coppia di sottostringhe precedenti, cioè che i punti di inizio non cambiano.

Si suppone che le migliori sottostringhe che terminano in $i - 1$, $j - 1$ siano $s_1[a, i - 1]$ e $s_2[b, j - 1]$, ovvero $All(s_1[a, i - 1], s_2[b, j - 1]) > All(s_1[h, i - 1], s_2[k, j - 1])$.

Si aggiunge la colonna $[s_1[i], s_2[j]$ ad $All(s_1[a, i - 1], s_2[b, j - 1])$. Questo nuovo allineamento, considerando uguali caratteri nell'ultima colonna, ha un valore maggiore di $All(s_1[h, i], s_2[k, j])$, il che è assurdo visto che è stato precedentemente definito quest'ultimo come massimo.

Un ragionamento analogo è applicabile agli altri due casi, con un indel nella colonna: questo algoritmo introduce però un quarto caso, in cui nessuna sottostringa ha un allineamento positivo. Per evitare valori negativi piuttosto viene scelta la sottostringa vuota: se precedentemente ci sono mismatch, la casella corrente viene settata a 0. Si ha anche che $M[0, 0] = 0$.

Le coordinate del valore massimo indicano le posizioni finali delle sottostringhe, mentre le coordinate del primo 0 a ritroso indicano il punto di inizio.
Le condizioni al contorno sono leggermente differenti, ma nella prima riga e nella prima colonna i valori sono tutti comunque 0.

Il tempo di questo algoritmo è $O(nm)$.

10.3 Considerazioni

Durante il confronto di genomi allineati, il valore di due colonne con coppie di caratteri uguali è lo stesso. Implicitamente, la probabilità delle mutazioni è la stessa: se ci sono A e C in colonna nella posizione 0 e nella posizione 15, teoricamente non cambierà nulla.

Non esiste un modello corretto per interpretare la soluzione, ma i modelli più sofisticati sono più vicini alla realtà. L'esempio più classico è rappresentato dalle previsioni del tempo: esso richiede una forte competenza in dominio, ma è accurato.

Uno studio più approfondito delle sequenze richiederebbe conoscenze di biologia (genetica, ...) e un maggior tempo di calcolo. Per risultati migliori, si potrebbero avere diverse matrici di score, ma sarebbe più complicato; si cerca un equilibrio nel modello (il più semplice che funzioni, che di solito è anche il più veloce).

Se si vuole confrontare una sequenza di DNA, un passaggio prevede che una tripletta (codone) di basi venga codificata in un aminoacido. Con 4 nucleotidi, ci sono 20 possibili aminoacidi: più codoni quindi codificano lo stesso aminoacido.

Una mutazione che trasforma diversi nucleotidi nella stesso aminoacido è molto probabile e importante da considerare. Un cambiamento nel codone di start o in quello di stop avrà una rilevanza significativa rispetto a qualche variazione nel mezzo, quindi un modello semplice non rappresenterà in modo esaustivo queste situazioni.

Per questo motivo esistono dei modelli più complessi.

11 Distanza di edit

La distanza di edit è la trasformazione di una stringa in un'altra tramite inserimento di un nuovo carattere, cancellazione o modifica.

Esempio: $s_1 = ABRACADABRA$, $s_2 = ABRABANANA$, l'obiettivo è trovare il minimo numero di operazioni che trasformano s_1 in s_2.

In questo caso, al contrario rispetto all'allineamento, la funzione obiettivo riguarda la minimizzazione: questo è un classico esempio di problema con distanza, utilizzato negli algoritmi di clusterizzazione.

Il problema è semplice da definire ma non da calcolare, quindi è necessario ricorrere alla programmazione dinamica.

11.1 Equazione di ricorrenza

L'equazione di ricorrenza è simile a quella dell'allineamento. La soluzione ottima è contenuta in una matrice: $M[i,j]$ rappresenta la minima distanza di edit fra i primi i caratteri di s_1 e i primi j di s_2.

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

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

Per rendere più semplice l'algoritmo, si considera anche l'opzione in cui un'operazione avviene con una coppia di caratteri uguali (che poi non verrà effettuata in pratica, perché sconveniente).

L'ultima componente della soluzione ottima riguarda l'ordine: prima viene modificato il carattere a sinistra, poi quello di destra.

Considerando anche le no-op, ci saranno caratteri che non vengono mai coinvolti? La risposta è no: al contrario, ogni carattere sarà soggetto a esattamente un'operazione (l'uguaglianza).

Paragonando questo algoritmo all'allineamento, si ha che l'indel corrisponde a un inserimento o una cancellazione. La distana di edit, quindi, è una versione semplificata della minimizzazione dell'allineamento globale, dove il costo di ogni mismatch è sempre uguale a 1.

Questo procedimento ha un tempo di $O(nm)$. In bioinformatica, la distanza di edit non è solitamente utilizzata.

12 Allineamento globale ottimo

L'allineamento globale richiede tempo e spazio di $O(nm)$. L'occupazione in spazio è più limitante in pratica di quella in tempo: la creazione della matrice è più dispendiosa, ma l'algoritmo funziona sempre qualsiasi siano le stringhe.

Esempio 12.1

$s_1 =$ A B R A C A D A B R A
$s_2 =$ A B R A B A N A N A
$s_3 =$ A C G T C G G T C C T A
$s_4 =$ C G T G G A A T C A T T T

L'algoritmo in entrambi i casi ci mette lo stesso tempo perché non è in grado di capire che le prime metà di s_1 e s_2 sono uguali.

Si vuole considerare l'allineamento globale sotto il vincolo di avere al massimo k colonne con indel o mismatch.

La programmazione dinamica risolve questo problema con un percorso nella matrice che sarà sempre diagonale tranne al più k volte.

Il percorso contiene al massimo k spostamenti in orizzontale o verticale, questo vale se ci sono k indel. Per accelerare questo processo, è possibile sfruttare il fatto che alcune celle possono non essere calcolate. Il massimo numero di mismatch in una casella $[i, j]$ è $|i-j|$, che dovrà essere strettamente minore di k.

Viene considerata solo una banda della matrice, la cui ampiezza sarà al più $2k$. Solo i valori che rientrano in essa vengono calcolati: al massimo ci saranno $2kn$ caselle, mentre il tempo e lo spazio diventano $O(kn)$.

Il problema principale è la decisione del valore di k. Il test, fondamentalmente, è: il percorso ottimale con l'allineamento globale è contenuto nella banda? Non ha senso confrontare la matrice ottenuta con quella originaria dell'allineamento, perché troppo dispendioso.

La stima iniziale più sensata di k corrisponde alla differenza tra le lunghezze delle due stringhe. In seguito, però, bisogna controllare che esso sia ottimale e in caso contrario ridefinire il valore.

Un modo semplice è l'estensione della banda di 1 a destra e 1 a sinistra, calcolare l'allineamento ottimale e verificare che i due siano uguali. In caso contrario, k non è ottimale. Il tempo del controllo è simile.

Quando la stima è sbagliata, la banda può aumentare di ampiezza di una posizione (da entrambi i lati). Il tempo di ciò è $\sum_{i=0}^{k}(2i+n)n = O(k^2n)$, che è troppo.

Raddoppiando l'ampiezza della banda, $\sum_{i=0}^{\lceil log_2 k \rceil}(2^i+1)n \leq \sum_{i=0}^{\lceil log_2 k \rceil} 2^{i+1}n = O(kn)$.

13 Gap

Il problema principale degli algoritmi trattati finora consiste nella presenza degli indel. Ogni problema computazionale, così come ogni matrice di score, corrisponde a un determinato modello: la probabilità di inserimento o cancellazione di un determinato carattere è identica.

Un altro aspetto fondamentale da considerare, però, è la presenza di indel nell'allineamento globale.

Esempio 13.1

$s_1 =$	A	C	G	T	T	A	T	A	C	G
$s_2 =$	-	-	G	T	-	A	T	C	G	-
$s_3 =$	-	-	G	T	***A***	T	C	G	-	-

Le sequenze s_2 e s_3 differiscono soltanto di un carattere, che nel primo caso è un indel nel mezzo. Se s_1 è il genoma di riferimento, l'indel in s_2 (un carattere ignoto) potrebbe cambiare in modo drastico il significato della sequenza, o frame di lettura.

Un gap è una sequenza contigua di indel in un allineamento. Un gap sposta il frame di lettura, quindi l'aggiunta di un indel può essere un'operazione costosa e rilevante, ma tutti gli indel successivi sono meno significativi.

Il gap dev'essere in funzione della lunghezza l, senza considerare le lettere associate agli indel. In input ci sono la matrice di score, le stringhe da allineare e un valore che corrisponde alla penalità P.

Quest'ultima è definita come $P : \mathbb{N} \to \mathbb{Q}^+$ e ha le seguenti proprietà:

- La funzione è monotona crescente;
- Il costo dell'elemento $n + 1$ è sempre minore (o uguale) di quello di n: la derivata prima non è crescente, quindi la derivata seconda non è strettamente positiva. Il costo del gap tende a essere sempre uguale dopo n indel, quindi la funzione tende a una retta dopo il primo costo.

L'obiettivo è la risoluzione delle due varianti:

1. Gap arbitrario, in cui la funzione è sconosciuta ma rispetta le proprietà precedenti;
2. Gap lineare (affine), con funzione lineare.

13.1 Gap arbitrario

Il gap arbitrario è un problema risolvibile con programmazione dinamica. Si ha che il costo di un gap lungo l è definito da $P(l)$.

L'ultima componente dell'allineamento ottimo, cioè l'ultima colonna, può essere composta da una coppia carattere-carattere oppure carattere-indel, ma bisogna considerare anche il contenuto delle colonne precedenti, altrimenti non si può capire quanto è lungo un gap.

Nel caso di due indel vicini, la somma dei valori presi separatamente è diversa da quella dei valori consecutivi (il gap lungo 2 è meno costoso), di conseguenza è essenziale che ci sia una somma ma che essa sia accurata.

In $M[i, j]$ si vuole avere l'allineamento ottimale dei primi i caratteri di s_1 e i primi j caratteri di s_2. Non viene considerata una colonna per volta, ma un intero gap: è necessario controllarne la lunghezza.

Il caso peggiore è un gap lungo i, cioè tutta la lunghezza della parte a sinistra della stringa fino a i quando ci sono solo indel.

L'equazione di ricorrenza (dove ϕ rappresenta il valore nella matrice di score) è rappresentata come:

$$M[i,j] = max \begin{cases} M[i-1, j-1] + \phi(s_1[i], s_2[j]) & \text{con carattere-carattere} \\ max_{\ l>0}\ M[i, j-l] + P(l) & \text{gap in } s_1 \\ max_{\ l>0}\ M[i-l, j] + P(l) & \text{gap in } s_2 \end{cases}$$

Il numero dei casi dipende dalla lunghezza del gap, perché esse vanno valutate tutte.

Il tempo computazionale totale è $O(nm(n + m))$, dove nm sono il numero delle caselle e $n + m$ il costo del calcolo di ogni casella.

13.2 Implementazione

L'implementazione di questo algoritmo è simile a quella di Needleman-Wunsch, essendo entrambi risolti con la programmazione dinamica. L'allineamento locale, infatti, può essere visto come allineamento globale con gap di costo 0 ai lati.

La matrice di score viene data in input, insieme alle sue dimensioni ed eventualmente un puntatore a essa. La parte più pesante è il riempimento delle matrici, che utilizza funzioni di appoggio *min* e *max*. Dato che una matrice in C viene gestita come array di array, per effettuare i calcoli senza errori si converte ogni riga e colonna in un array unidimensionale, mappando anche le caselle limitrofe.

13.3 Gap lineare

In questo caso, il costo di un gap lungo l è $P_o + lP_e$ (eventualmente $l - 1$), dove P_o è il costo dell'apertura del gap e P_e è il costo dell'estensione (strettamente positivi).

$M[i, j]$ è l'allineamento ottimo di $s_1[: i]$ e $s_2[: j]$. Questo algoritmo è una via di mezzo tra Needleman-Wunsch e il gap arbitrario, ma i casi si restringono a due: apertura o estensione.

Bisogna conoscere il contributo dell'ultima colonna, e questo permette di arrivare ai seguenti casi:

1. Nessun indel;
2. Indel all'inizio di un gap, allineato con un carattere;
3. Indel che estende un gap, allineato con un carattere;
4. Carattere, allineato con un indel all'inizio di un gap;
5. Carattere, allineato con un indel che estende un gap.

L'apertura di un gap è semplice, perché non ci sono restrizioni a sinistra. L'estensione implica che ci sia un indel precedentemente, quindi è necessario eseguire un controllo: c'è bisogno di un modo per rappresentare l'allineamento ottimo sotto il vincolo che nell'ultima colonna ci sia un indel. Questo viene rappresentato tramite un'altra matrice.

In totale ci sono 5 matrici:

1. M, con $M[i, j]$ allineamento ottimo di $s_1[: i]$, $s_2[: j]$;
2. N_1, con $M[i, j]$ allineamento ottimo di $s_1[: i]$, $s_2[: j]$ e apertura di gap in s_1;
3. N_2, con $M[i, j]$ allineamento ottimo di $s_1[: i]$, $s_2[: j]$ e apertura di gap in s_2;
4. E_1, con $M[i, j]$ allineamento ottimo di $s_1[: i]$, $s_2[: j]$ e estensione di gap in s_1;
5. E_2, con $M[i, j]$ allineamento ottimo di $s_1[: i]$, $s_2[: j]$ e estensione di gap in s_2.

Esse sono strettamente collegate, e permettono di ottimizzare tempo e spazio grazie alla costante moltiplicativa della notazione O-grande.

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

L'equazione di ricorrenza finale è:

$$
M[i,j] = max \begin{cases} M[i-1,j-1] + \phi(s_1[i], s_2[j]) & \text{ultima colonna senza indel} \\ M[i-1,j] + P_o & \text{apertura gap in } s_1 \\ +P_e & \text{estensione gap in } s_1 \end{cases}
$$

N ed E vengono calcolate con cicli innestati, e i loro risultati sono utilizzati per trovare il valore di M. L'algoritmo è più complicato, ma i casi sono legati fra di loro tramite le formule precedenti.

Il tempo computazionale è $O(nm)$ (così come lo spazio): ognuna delle matrici occupa $O(nm)$ caselle, ed essendo i casi un numero costante non è necessario aumentare il tempo con calcoli aggiuntivi.

14 Matrici di sostituzione

Le matrici di sostituzione (score) vengono usate per valutare un allineamento (proteine, in campo bioinformatico), e implicitamente misurano la probabilità di transizione (mutazione da un carattere all'altro). Il valore aumenterà se un cambiamento è probabile.

Tanto maggiore è la lunghezza temporale tra due sequenze, tanto maggiore è la probabilità di mutazione (es. i virus che cambiano nel corso degli anni) per selezione naturale, quindi questo è un fattore importante da considerare.

14.1 PAM

PAM (Point/percent Accepted Mutation) serve per capire quanto due sequenze siano distanti utilizzando la quantità di mutazioni. Si utilizza per esempio quando un singolo aminoacido nella struttura di una proteina viene rimpiazzato.

1PAM è appunto il *numero di mutazioni*: $\frac{1}{100}|s_1|$, con 100 variabile in base al peso che si vuole dare a ogni mutazione. In assenza di indel, questo valore è semplice da calcolare, ma quando le mutazioni diventano ricorrenti il metodo perde affidabilità.

Esempio 14.1

Se s_1 e s_2 sono distanti 100PAM, una singola base ha il 36% di probabilità di non essere mutata.

Quindi, l'accuratezza dipende dalla distanza attesa: per coprire parzialmente questo problema, ci sono diverse matrici (PAM250, PAM200, PAM1).

La costruzione di ogni matrice PAMk è eseguita matematicamente: si prendono varie sequenze distanti kPAM e le si allineano, poi si calcolano le frequenze $f(i)$, $f(i,j)$ di tutti i singoli caratteri e tutte le coppie di caratteri.

Ogni casella, quindi, indica la probabilità che l'aminoacido di quella riga sia stato sostituito con l'aminoacido di quella colonna attraverso il tempo.

La formula finale è:

$$\mathrm{PAM}_k(i,j) = \log \frac{f(i,j)}{f(i)f(j)}$$

$f(i)$ frequenza della mutazione misurata

$f(i)f(j)$ ipotesi nulla (caratteri indipendenti)

Il valore ottenuto dalla formula è chiamato **Log odds ratio** ed è definito generalmente come $\frac{p}{1-p}$ dove p è la probabiltà dell'evento interessante (target).

Sono state allineate sequenze molto simili, con distanza temporale breve, e poi è stato applicato il ragionamento matematico: questo però ha comportato una perdita di affidabilità se le sequenze sono lontane.

Come allineare se non si conosce la matrice PAM? Essa viene costruita tramite catene di Markov, a partire da PAM1.
Senza la presenza di indel, si ha che:

$$PAM_k(i,j) = \log \frac{f(i)M_1^k(i,j)}{f(i)f(j)} = \log \frac{M_1^k(i,j)}{f(j)}$$

I valori vengono poi moltiplicati per 10 e arrotondati all'intero più vicino, e si somma un intero a tutti i valori.

14.2 BLOSUM

Le matrici BLOSUM (BLock SUbstitution Matrix) sono state introdotte per rimediare al problema di PAM: esso infatti viene utilizzato per allineare sequenze lontane, ma è stato studiato per le sequenze vicine (distanza temporale). Inoltre, le regioni conservate e quelle non conservate hanno la stessa importanza.

BLOSUM, invece, divide i blocchi di regioni conservate da quelle mutate, scegliendole manualmente. Il cambiamento più probabile ha valore maggiore, sempre secondo il concetto di Log odds ratio: si ha che $B[i,j] = \log \frac{f(i,j)}{f(i)f(j)}$.

Esistono diverse versioni di BLOSUM, generalizzate con BLOSUMx. In ogni caso, le sequenze che sono simili più di $x\%$ vengono clusterizzate: sono rimosse tutte tranne una, per evitare di sovrappesare parti ripetute nel campione. La matrice più usata per gli allineamenti è BLOSUM62.

15 Karlin-Altschul e BLAST

BLAST (Basic Local Alignment Search Tool) è uno strumento per il confronto di sequenze in un database che si basa sui seed. Un *seed* è un pattern matching con una sottostringa di lunghezza 3.

L'algoritmo si basa sulla costruzione di HSP, High-scoring Segment Pair, cioè un'estensione di un seed che massimizzi il valore dell'allineamento. Gli HSP vengono poi filtrati e tenuti solo quelli con alta significatività, mentre gli HSP vicini vengono fusi.

Per allineare le regioni, BLAST utilizza Smith-Waterman.

Le statistiche Karlin-Altschul servono per verificare le probabilità di una ricerca in un database tramite BLAST. Alle query viene assegnato un punteggio, che possibilmente è positivo ma in media è negativo. Queste statistiche tengono conto di simboli indipendenti ed equiprobabili, sequenze infinitamente lunghe e allineamenti senza gap.

Gli HSP con massimo score sono distribuiti casualmente, e si può considerare che la probabilità che questi punteggi superino una determinata soglia segua una distribuzione di Poisson.

Si ha che $E = kmne^{-\lambda S}$, dove:

- E è il numero degli allineamenti senza gap;
- k è una costante che dipende dallo score S;

- n è il numero dei caratteri nel database;
- m è la lunghezza della stringa nella query;
- λS è il punteggio normalizzato (Poisson).

E è proporzionale alla dimensione dello spazio di ricerca (mn) e diminuisce esponenzialmente con lo score S.

16 Allineamento multiplo

Il problema di allineamento viene esteso, con k stringhe in input. Le sequenze quindi sono un insieme $\{s_1,\ s_2,\ \ldots,\ s_k\}$.

Esempio 16.1

$s_1 =$ A C T G G A T
$s_2 =$ T G G A T
$s_3 =$ A C G G G A T T
Un possibile allineamento è:
$s_1 =$ A - C T G G - A T
$s_2 =$ - - T G G - A - T
$s_3 =$ A C G G - G A T T
In questo caso, è necessario considerare la presenza di più di un indel sulla stessa colonna: è accettabile? Sicuramente non potranno esserci colonne formate unicamente da indel, ma è possibile aggiungerne fino a $k-1$.

La soluzione ammissibile di questo problema è l'allineamento ottimo che rispetta le seguenti proprietà:

1. Le sequenze estese devono avere tutte la stessa lunghezza;
2. Non devono esserci colonne in cui sono presenti tutti indel.

La matrice dev'essere bidimensionale, per poterla confrontare con PAM e BLOSUM. Per questo motivo si introduce la funzione SP, Sum of Pairs: tutte le coppie di righe sono considerate come sequenze da allineare, ed esse vengono sommate.

Si usa la somma delle coppie di tutte le sequenze per ottenere uno score complessivo.
Si ha $\{s_1,\ s_2,\ \ldots,\ s_k\} \to \{s_1^*,\ s_2^*,\ \ldots,\ s_k^*\}$ allineamento ottimo, con valore $\{s_1^*[h],\ s_2^*[h],\ \ldots,\ s_k^*[h]\}$. $\sum_{i<j} d(s_1^*[i],\ s_k^*[j])$.

Il problema di ciò è ancora l'inserimento di indel, è permesso averne due finché non sono k ma l'allineamento globale di due sequenze questo non lo prevede.

L'algoritmo sfrutta la programmazione dinamica: viene individuato l'ultimo componente della soluzione ottimale, e per ogni caso sfruttate le soluzioni dei sottoproblemi. L'ultima colonna può trovarsi in $2^k - 1$ casi: ogni sottoinsieme delle stringhe in ingresso potrebbe avere un indel.

Se k è arbitrario, l'algoritmo è NP-completo. Altrimenti, il tempo computazionale è $O(n^k)$ (dove n è la lunghezza maggiore), con n^k caselle nella matrice, il che è molto elevato.

17 Grafi di assemblaggio

La bioinformatica sta ancora cercando di capire in modo chiaro come siano fatti i nucleotidi del genoma umano. Il problema del sequenziamento umano fino a qualche decennio fa era troppo complicato, quindi i primi studi su organismi più semplici come il lievito o i moscerini della frutta.

L'assemblaggio dei genomi è effettuata da porzioni di genoma definite *read*, di 50-10.000bp (base pairs). Questi spesso vengono in coppia (mate pairs), ma non è possibile identificarne la posizione originaria a partire dai singoli pezzi: da qui il problema della ricostruzione.

Per la costruzione, vengono utilizzati dei sequenziatori, che ottengono read a partire dal genoma sfruttandone le proprietà biochimiche. Il numero medio di read estratte è definito *copertura*.

L'evoluzione tecnologica ha permesso sviluppi di attrezzature in grado di classificare read sempre più lunghi, in base ai gigabyte per run. Sanger è uno dei più importanti elaboratori che ha permesso di identificare i read prima del 2006, con costi elevatissimi. Il progresso nel campo dell'informatica ha portato a tempi e costi molto più brevi, ma ci sono altri importanti fattori da tenere in considerazione: il tasso di errore e la distribuzione dell'errore.

Il sistema che individua i read più lunghi (10.000) ha un tasso di errore di circa il 15%, mentre altri hanno meno dell'1‰ ma errori sistematici (che invalidano l'analisi).

Per rimediare a questi problemi sono utilizzati agenti chimici (adattatori) che rendono il DNA più leggibile, tagliandolo in entrambi i lati. Le read si possono estrarre di una lunghezza relativamente bassa conoscendo la distanza tra ogni coppia di esse.

Il problema, in questo caso, è la ricostruzione del genoma di partenza a partire dalle read. Non è conosciuto il punto di inizio e di fine di esse nella stringa iniziale.

C'è una proprietà che si può sfruttare: l'overlap. Il suffisso di una read può essere prefisso di un'altra read (sovrapposizione). Pezzi di sequenze uguali vengono quindi unite, ma questo causa ulteriori problemi:

1. Casualità, se le sovrapposizioni sono brevi (quindi si è interessati solo a quelle sufficientemente lunghe);
2. Errore nella lettura;
3. Organismi diploidi (come l'uomo), con una copia del genoma di ogni genitore che comportano sequenze con piccole differenze (SNIP, Single Nucleotide Polymorphism).

Esempio 17.1

$$\begin{array}{cccccccccccccccccccc} T & C & T & A & T & A & T & C & T & C & G & G & C & T & C & T & A & G & G & \\ & & & & | & | & | & | & | & | & | & & | & | & | & | & | & | & | & \\ & & & & T & A & T & C & T & C & G & A & C & T & C & T & A & G & G & C & C \end{array}$$

Probabile motivo:

$$\begin{array}{cccccccccccccccccccc} & & & & C & T & A & T & A & T & C & T & C & G & G & C & T & & & & & \\ G & C & G & T & C & T & A & T & A & T & C & T & C & G & G & C & T & C & T & A & G & G & C \\ & & & & & & & & A & T & C & T & C & G & A & C & T & C & T & & & & \end{array}$$

17.1 Shortest superstring

Questo problema è una semplificazione della ricostruzione del genoma, che non lo gestisce interamente ma ha in comune molti elementi.

Dato un insieme $S = \{s_1, \ldots, s_n\}$ di stringhe, trovare una stringa T tale che ogni stringa s_i è sottostringa di T.

Applicando questo problema alle read, la funzione obiettivo è $|T|$, T rappresenta il genoma assemblato e S le read. Bisogna però gestire le regioni ripetute, ovvero tenere conto della possibilità di grandi regioni quasi uguali.

Una risoluzione greedy consiste nel fondere le due stringhe con massimo overlap iterativamente finché non ne rimane una sola. Tanto più lunghe sono le read, tanto più lunghe saranno le sottosequenze che è possibile distinguere.

Esempio: **a_long_long_long_time**

1. ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long g_time **ng_tim**
2. ng_time ng_lon **_long_** a_long long_l ong_ti ong_lo long_t **g_long**
3. ng_time g_long_ ng_lon a_long long_l **ong_ti** ong_lo **long_t**
4. ng_time long_ti g_long_ **ng_lon** a_long long_l **ong_lo**
5. **ng_time** ong_lon **long_ti** g_long_ a_long long_l
6. **ong_lon** long_time g_long_ a_long long_l
7. long_lon **long_time g_long_** a_long
8. **long_lon g_long_time** a_long
9. **long_long_time a_long**
10. a_long_long_time

L'algoritmo è NP-hard, e dato che ci sono porzioni quasi perfettamente ripetute (duplicazione), ci saranno regioni estremamente simili che verranno prese in considerazione solo una volta.

18 String graph per l'assemblaggio

Il problema di assemblaggio del genoma, quindi, è ridotto a un problema su grafi. Ogni read corrisponde a un vertice, e i collegamenti sono rappresentati tramite archi con un'etichetta che ne indica la lunghezza. Le rappresentazioni più comuni sono **OLC** e **De Bruijn**.

Ogni superstringa viene convertita in un grafo completo, orientato e pesato. Data una funzione ov che calcola l'overlap (orientato), ogni etichetta è ottenuta con $|A| - ov(A, B)$ con A nodo di partenza, B nodo di arrivo. Si vuole trovare il cammino più corto che tocca ogni vertice una e una sola volta.

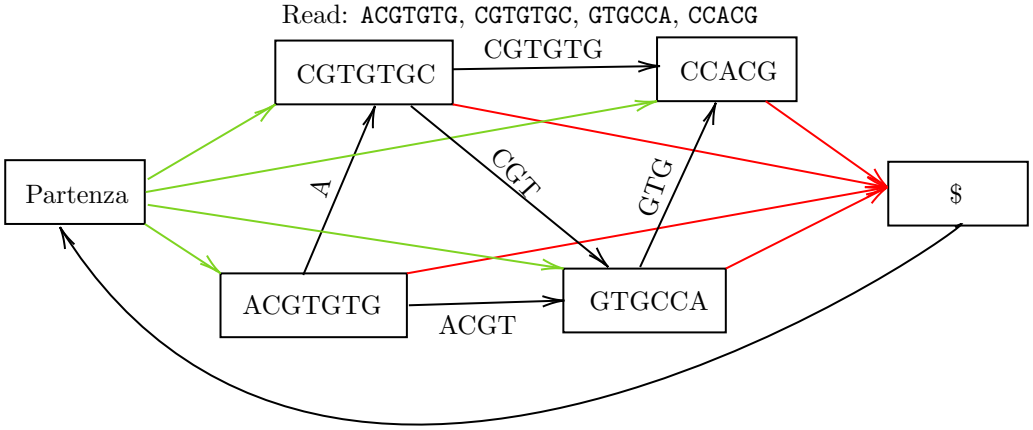
Considerando che il genoma umano è formato da 4 nucleotidi in una struttura a doppia elica, secondo la legge di Watson-Crick si ha $A \leftrightarrow T$ e $C \leftrightarrow G$: i due filamenti sono uno il complemento dell'altro. Fra una coppia di vertici si potrebbe avere fino a 4 archi, di cui due riferiti al *reverse and complement*. Per ridurre questo problema ci sono tecniche euristiche.

18.1 Grafi di overlap

Un grafo di overlap è una rappresentazione astratta in cui ogni read è un nodo, e se l'overlap è *abbastanza lungo* i nodi sono collegati con un arco. L'etichetta di ogni arco è la parte che precede la sovrapposizione.

Tutte le possibili sovrapposizioni sono rappresentate, ma la stringa di partenza va ricostruita in modo preciso tramite visita del grafo in base alla lunghezza dell'etichetta. Alcuni archi, però, non sono informativi (stringhe non consecutive), e per questo motivo essi vengono rimossi.

Figure 6: Grafo di Overlap



18.2 TSP

Un problema intermedio tra la visita nei grafi e la ricostruzione di stringhe è il Traveling Salesman Problem: dato un grafo orientato $G = \langle V, A \rangle$ con archi pesati $w : A \to \mathbb{Q}^+$, trovare una permutazione $\Pi = \langle \pi_1, \ \ldots, \ \pi_n \rangle$ di V di costo minimo che visiti tutti i nodi e torni al punto di partenza. Il costo è il peso totale di tutti gli archi attraversati.

La funzione obiettivo è:

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

Nonostante sia risolvibile in pratica con grafi anche di grandi dimensioni (grazie alla potenza dell'hardware), il problema è NP-completo.

Partendo dall'istanza della superstringa, si vuole ottenere un'istanza di TSP per poi arrivare alla relativa soluzione. Una volta trovata la soluzione TSP, essa viene convertita in quella della superstringa.

Le stringhe in ingresso vengono mappate nel grafo di TSP, diventando nodi. Gli archi devono avere peso minimo, quindi diventano la parte della stringa che non è sovrapposta.
Ogni read è una città, ma l'assemblaggio non è un ciclo e la lunghezza della stringa è diversa dal costo del percorso TSP.

Si ha che:

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

dove *ov* è l'overlap.

Per individuare la fine della stringa si usa un simbolo di appoggio \$, collegato all'inizio, per poter tornare al punto di partenza. Vengono mappati tutti i possibili cammini, e poi viene individuato il percorso ottimo.

18.3 OLC

OLC (Overlap, Layout, Consensus) è un modo di eseguire la riduzione tramite questi passaggi:

1. Overlap, calcolo delle sovrapposizioni e costruzione del grafo. Per un metodo esatto si utilizza il suffix array, altrimenti la programmazione dinamica;
2. Layout, fusione dei cammini per ottenere i *contigs*, sottosequenze continue. Le ripetizioni (branching nodes) vengono rimosse;
3. Consensus, calcolo dei nucleotidi.

Si vuole calcolare la lunghezza dell'overlap per ogni coppia di stringhe in modo veloce, usando i suffix array. Il primo step è calcolare il suffix tree generalizzato di tutte le read, e con una sola visita determinare per ogni prefisso se esso sia anche un suffisso.

L'albero viene visitato carattere per carattere, usando la read come pattern, e cercando tutti i nodi da cui esce un simbolo di terminazione che non sia l'ultimo.

18.3.1 OLC con errori

Per avere overlap permettendo la presenza di errori, al contrario, si definisce un problema tale che: date due stringhe s, t, trovare:

1. Suffisso x di s;
2. Suffisso y di t;

tale che $|x| + |y| - 2edit(x, y)$ sia massima. Il 2 viene introdotto arbitrariamente come fattore moltiplicativo per dare un peso maggiore alle sovrapposizioni.

In questo caso non vengono confrontati prefissi, ma un prefisso e un suffisso. $M[i, j]$ = ottimo del prefisso lungo j di t e del suffisso lungo i di s. L'algoritmo è simile a LCS, con questa differenza.

18.4 SBH

SBH, Sequencing By Hybridation, è una vecchia tecnologia (metà anni '90) per gli array di DNA, che analizza gli oligonucleotidi (6-10 basi). Per ogni k-mero (chip), con $k \sim 8$, si conosce se appare nel genoma. Il processo è chiamato DNA chip perché la logica di fondo è la stessa che viene usata per i chip, ognuno di essi può tenere migliaia di oligonucleotidi.

Esempio 18.1

DNA Chip	A	C	G	T	G	G	C	A					
DNA Replicato	T	G	G	**T**	**G**	**C**	**A**	**C**	**C**	**G**	**T**	G	G

Ogni nucleotide viene messo in contatto con il DNA dopo che esso è stato replicato più volte tramite PCR (un processo chimico), nel quale avviene un'ibridazione: nelle giuste condizioni biochimiche, il nucleotide sul chip forma legami covalenti con una porzione di DNA che è perfettamente complementare (eliche), in modo che essi siano ancora uniti al momento dell'estrazione.

Alcuni oligonucleotidi non reagiranno, perché non hanno trovato il proprio complementare: è possibile individuarli sapendo che il DNA replicato viene marcato con qualche sostanza che lo renda riconoscibile (fluorescente).

Questa tecnologia in pratica non viene utilizzata, ma ha degli interessanti aspetti algoritmici. Una differenza essenziale tra un grafo di overlap e SBH è che i k-meri hanno tutti la stessa lunghezza, al contrario delle read, e ci si aspettano sovrapposizioni lunghe esattamente $k - 1$.

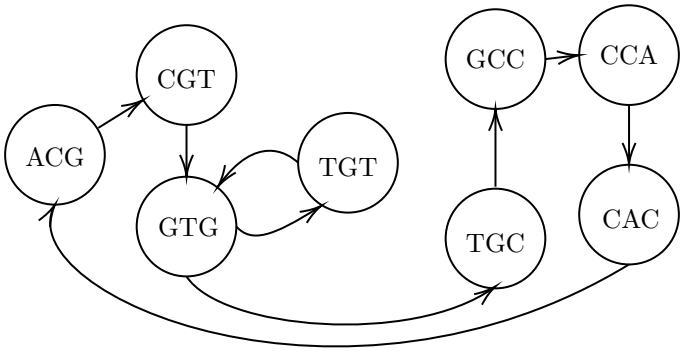
18.4.1 Grafi di De Bruijn

Ogni k-mero (sottostringa di lunghezza k) viene diviso in $(k - 1)$-meri: in un grafo di De Bruijn, ogni **arco** corrisponde a un k-mero, ogni **vertice** corrisponde a un $(k - 1)$-mero, e due vertici sono collegati se ci sono sovrapposizioni.

Dato che tutte le stringhe in ingresso hanno lunghezza k e sono presenti in almeno una read, il problema della costruzione del grafo diventa più semplice.

I $(k-1)$-meri identici vengono eliminati, in modo da avere nodi distinti. Il problema diventa trovare in un grafo il *percorso che attraversi ogni arco esattamente una volta* (cammino Euleriano), per ricostruire il genoma.

Figure 7: Grafo di De Bruijn



18.5 Cicli e cammini di Eulero

Definizione 18.1: Grafo Semi-Euleriano

Sia $G = \langle V, A \rangle$ un grafo orientato. G è semi-Euleriano se esistono due vertici s, t tali che $N_G^-(s) = N_G^+(s) + 1, N_G^-(t) = N_G^+(v) - 1$, mentre per ogni altro vertice w, $N_G^-(w) = N_G^+(w)$. Ogni nodo ha lo stesso numero di archi entranti e uscenti, tranne per due nodi di cui uno ha un arco uscente in più e l'altro uno in meno.

Definizione 18.2: Grafo Euleriano

Sia $G = \langle V, A \rangle$ un grafo orientato. G è Euleriano se $N_G^-(w) = N_G^+(w)$, per ogni vertice (stesso numero di archi entranti e uscenti).

Sia $G = \langle V, A \rangle$ un grafo Euleriano e sia C un ciclo di G. Sia G_1 il grafo ottenuto da G togliendo tutti gli archi di C. Allora G_1 è Euleriano.

Teorema 18.1: Grafo Euleriano

Un grafo connesso $G = \langle V, A \rangle$ ha un cammino Euleriano se e solo se G è semi-Euleriano. G ha un ciclo Euleriano se e solo se G è Euleriano.

Sia $G = \langle V, A \rangle$ un semi-Euleriano e sia P un cammino da s a t. Sia G_1 il grafo ottenuto da G togliendo tutti gli archi di P. Allora G_1 è Euleriano.

Definizione 18.3: Ciclo Euleriano e Hamiltoniano

Un ciclo (cammino) Euleriano è un assemblaggio in un grafo orientato e connesso che attraversa ogni **arco** esattamente una volta.

Un ciclo Hamiltoniano (caso particolare di TSP) è un cammino che attraversa ogni **vertice** esattamente una volta.

Il primo problema è risolvibile con un algoritmo in tempo lineare, mentre il secondo è NP-completo.

Un ciclo viene chiamato semplice se non tocca due volte lo stesso vertice: è possibile trovare cicli Euleriani in questi casi, ma non ci saranno cicli Hamiltoniani.

L'algoritmo è in tempo lineare: ciò significa che per ogni arco si spende un tempo costante. Si parte dal primo vertice, e si prosegue verso un arco uscente s. Si cercano sempre archi uscenti che non sono già stati visitati, e si termina quando non ne esistono più.

L'ultimo vertice è la destinazione, che ha un numero di archi entranti superiore di 1 rispetto agli archi uscenti. Togliendo il cammino Euleriano, il grafo diventa sicuramente Euleriano (formato da un insieme di cicli), anche se non necessariamente connesso.

Ci dev'essere un vertice toccato dal cammino Euleriano che ha un arco uscente non ancora visitato. Come prima, si guarda quest'ultimo arco del vertice, e la procedura termina nello stesso di partenza: si è trovato un altro ciclo.

I vari cicli, dopo essere stati trovati, vengono ricombinati.

18.6 Reverse and complement

All'algoritmo precedente viene aggiunta una complicazione: non si conosce lo strand di DNA. La stessa stringa potrebbe aver subito un reverse and complement, e non si sa quale delle due possibilità è effettivamente quella corretta.

Per evitare il raddoppio dello spazio di memoria, si indica la read canonica come la prima in ordine lessicografico, però a ogni calcolo dell'overlap viene considerato anche il complemento.

L'idea di base è che i valori più lunghi di k sono più espressivi, perché permettono di gestire meglio le sovrapposizioni. I numeri dispari sono migliori, perché altrimenti potrebbero esistere k-meri che hanno reverse and complement uguali all'originale.

Il genoma è diploide (copie quasi perfettamente identiche), ma le differenze formano percorsi separati in un grafo chiamato *bolla* (**bubble popping**). La lunghezza dei percorsi è variabile, e alcuni di essi non portano da nessuna parte: vengono rimossi in un processo chiamato **tip removal**. Attraverso questo perfezionamento delle sequenze sono costruiti i contig.

Un'altra fase importante è, dato un frammento del DNA di origine, capire se ci sono match tra contigs per poterli inserire in porzioni adiacenti (perché sono diploidi). Non sono ancora conosciuti tutti i possibili match.

19 Ricostruzione della storia evolutiva

Con il passare del tempo, nella storia evolutiva umana, ci sono stati cambiamenti attraverso le generazioni. Questi sono probabilmente casuali, ma secondo la teoria della sopravvivenza del più adatto solo alcuni sono rimasti nel futuro.

Il processo di sviluppo nell'ambiente (di qualsiasi tipo) è definito *pressione selettiva*: le mutazioni dominanti sono quelle che hanno garantito un vantaggio sul numero di figli sani. Riconoscere i cambiamenti è utile per ricostruire la storia evolutiva.

Ci sono numerosissime tipologie di mutazioni: traslazione di DNA, replicazione di genoma eccetera, ma molte di esse non avranno effetti concreti. Per esempio, i pesci nella loro storia hanno avuto la duplicazione del loro intero DNA.

Se tanti fenomeni avvengono rapidamente in un intervallo di tempo breve, ci sarà rumore. Le mutazioni interessanti sono quelle che avvengono lentamente.

19.1 Evoluzione individuale

L'evoluzione individuale avviene in ogni individuo nel corso della sua vita: le cellule accumulano mutazioni, come nei tumori. Buona parte di esse sono comunque completamente innocue.

L'aspetto importante è che le cellule proliferano in modo casuale, in un ambiente avverso (a causa degli anticorpi e delle condizioni di pochissimo ossigeno), e di fatto per adattarsi c'è un'esplosione del numero di mutazioni.

19.2 Evoluzione basata sul carattere

Carattere può riferirsi a due tipologie: genetico, cioè la presenza di una specifica mutazione o meno nel DNA, oppure fenotipico, cioè visibile esteriormente (morfologico).

Essendo l'evoluzione un processo così complesso, sono state introdotte delle semplificazioni per classificare i caratteri ed essere in grado di ricostruire la storia delle specie, assumendo il modello binario più semplice possibile: **ogni carattere è stato acquisito esattamente una volta in tutta la storia, e una volta acquisito non viene mai perso** (avere quel carattere risulta vantaggioso).

Questo nella pratica non succede, ma è necessario un algoritmo che risolva il modello elementare per poi passare a quelli più complessi.

20 Filogenesi su caratteri

Il problema della filogenesi è definito con una matrice le cui righe rappresentano gli individui, e le colonne rappresentano i caratteri. Un 1 indica che quella determinata specie ha un determinato carattere, e viceversa (es. la tartaruga non ha il pelo).

Si vuole trovare un albero che, data la matrice binaria M, la rappresenti. L'algoritmo è in tempo lineare, e consiste nel radix sort delle colonne decrementando il numero di 1 e inserendo le specie una alla volta.

L'albero T ha le seguenti proprietà:

- Ognuno degli n oggetti (specie) corrisponde a una foglia di T;
- Ognuno degli m caratteri etichetta esattamente un arco di T (ogni etichetta indica l'acquisizione di un carattere);
- Per ogni oggetto p, i caratteri che etichettano gli archi nel percorso dalla radice alla foglia hanno come stato 1.

Se si ammette la possibilità di avere archi che non sono necessariamente etichettati da un carattere, i nodi interni possono essere trasformati in foglie: per questo motivo si usano solo foglie della matrice M, che hanno la stessa etichetta dei genitori (interni), senza corrispondenza $1:1$.

La radice non ha nessuno degli m caratteri; ogni carattere cambia stato da 0 a 1 esattamente una volta e il contrario non avviene mai.

Non è detto che per ogni matrice esista un albero, quindi la verifica è il primo problema. Se è possibile costruire un albero, esso è unico, e si parla di **filogenesi perfetta**.

In altre parole, il problema della filogenesi perfetta consiste nel determinare se esiste un albero filogenetico per una matrice 0-1 M di dimensioni $n \times m$.

	A	B
s_1	0	0
s_2	0	1
s_3	1	0
s_4	1	1

Figure 8: Filogenesi perfetta non ammissibile (sottomatrice).

Un carattere X è trasmesso da un nodo padre ai figli tramite un arco: solo i figli lo possiedono. Dato un altro carattere Y, la relazione tra i due insiemi può essere:

- X, Y disgiunti;
- X sottoinsieme di Y;
- X superinsieme di Y.

Data la matrice M, $1(c)$ è l'insieme di specie che hanno il carattere c, cioè $1(c) = \{s : M[s, c] = 1\}$. Questo ha di conseguenza il lemma definito in seguito.

Lemma 1.
Sia M una matrice che ammette filogenesi perfetta T, e siano c_1, c_2 due suoi caratteri. Allora (se e solo se) uno dei seguenti casi è vero:

1. $1(c_1) \cap 1(c_2) = \emptyset$;
2. $1(c_1) \subseteq 1(c_2)$;
3. $1(c_1) \supseteq 1(c_2)$.

Se non esiste una sottomatrice come quella sopra, è possibile costruire l'albero (condizione necessaria e sufficiente). Bisogna notare che molti caratteri ne includono altri, quindi una volta ricostruiti gli insiemi e capite le relazioni tra di essi, la costruzione diventa banale.

20.1 Algoritmo

L'algoritmo è semplice, e impiega un tempo computazionale di $O(nm)$, considerando che i confronti sono effettuati in tempo costante. Esso si basa sull'ordinamento di M.

Si cerca di ordinare le colonne in modo tale che quelle che ne includano altre vengano prima: questo è possibile grazie al fatto che un carattere non viene mai perso. Le colonne vengono interpretate come numeri interi binari, decrescenti.

La matrice M ha un albero filogenetico se e solo se per ogni coppia di colonne i, j, gli insiemi delle caratteristiche corrispondenti sono disgiunti o uno un sottoinsieme dell'altro.

Ogni carattere è un arco, e ogni specie viene collegata subito sotto l'ultimo carattere che possiede. Se un carattere ne include un altro, viene collocato a sinistra. Questo è il principio del radix sort.

1. Ogni colonna di M viene ordinata con radix sort decrescente;
2. Per ogni riga p della matrice ottenuta, viene costruita la stringa dei caratteri che p possiede, in ordine crescente;
3. Viene costruito l'albero T per le n stringhe ottenute nello step precedente, inserendole una per una;
4. Si controlla se T è una filogenesi perfetta.

20.1.1 Radix sort

Il radix sort ha una proprietà fondamentale: richiede tempo lineare. Non è in grado di ordinare elementi arbitrari, è pensato per numeri di b bit.

Dato che esso lavora sulle righe, è sufficiente trasporre la matrice. Alla fine, le colonne saranno in ordine decrescente: se non ci sono sovrapposizioni, ogni colonna è superinsieme della successiva.

Il tempo computazionale è $\Theta(nk)$, dove n è il numero di stringhe e k è il numero di caratteri (m nel caso dell'algoritmo originario).

20.2 Altri casi

I modelli sono sempre basati sui caratteri, che sono posseduti o meno da una specie. Il caso in cui una mutazione viene persa è più complesso: si parla di **backmutation**, cioè il cambiamento di stato di un arco da 1 a 0.

Si ricorda che la filogenesi perfetta è il caso più semplice, risolvibile in tempo lineare.

- Filogenesi perfetta: non ci sono cambiamenti all'indietro;
- Filogenesi persistente, rappresentabile con un modello 012 (intermedio ma più complicato da implementare), in cui ogni carattere viene acquisito una sola volta e perso al più una volta nell'albero;
- Parsimonia di Dollo, senza limitazioni sul numero di perdite ma ogni carattere è sempre acquisito una volta sola;
- Camin-Sokal, ogni carattere può essere acquisito più volte e non viene mai perso.

In queste tipologie, l'albero associato alla matrice può non essere unico, tranne nella soluzione banale. Una filogenesi Camin-Sokal o Dollo, però, garantisce l'esistenza dell'albero: è molto più generale, quindi i problemi sono più facilmente adattabili.

Il tumore è anch'esso un caso a parte: è formato da più popolazioni diverse, cellule sane e cancerogene (eterogeneità intratumorale). Ogni cellula reagisce in modo differente ai trattamenti.

Le tipologie sono rappresentabili con matrici di frequenza, che indicano la percentuale delle cellule con ogni determinata mutazione. Le matrici di frequenza vengono poi trasformate in binarie.

21 Approcci basati su parsimonia

I due problemi confrontati consistono nella grande e piccola parsimonia, risolti con gli algoritmi di **Fitch** e **Sankoff**.

Il criterio di parsimonia si basa sul concetto di *rasoio di Occam*, cioè la scelta della soluzione più semplice. L'albero dovrebbe avere la massima verosimiglianza, ma il modello di evoluzione sottostante dev'essere statistico e i tempi di calcolo sono elevati.

21.1 Grande parsimonia

Il **problema della grande parsimonia** consiste nel trovare la filogenesi (l'albero, sconosciuto) che minimizzi il numero di cambiamenti di stato, il momento in cui un carattere viene acquisito. In questo caso la matrice non è binaria, perché ogni carattere può avere più stati, non necessariamente in sequenza.

NB: il problema in versione semplificata considera solo caratteri presenti o meno, quindi la matrice corrispondente è binaria e ogni valore può essere solo 0 o 1.

Questo algoritmo è NP-completo, però può essere ridotto al **problema della piccola parsimonia**, in cui l'istanza in input è data dalla matrice M e dalla topologia dell'albero. Si vuole determinare quale specie etichetta ogni foglia, e quali passaggi di stato sono rappresentati da ogni arco.

21.2 Piccola parsimonia

La differenza tra la piccola e la grande parsimonia è che nella prima in input c'è la topologia dell'albero, quindi il problema è solo l'etichettatura.

Questo problema, in altre parole, consiste nel riempire l'albero a partire dalla struttura. Esso ha come istanze:

- Una matrice triangolare M con n specie e un insieme di caratteri C;
- Un albero T, le cui foglie corrispondono alle specie di M;
- Per ogni carattere $c \in C$, un costo w_c fra ogni coppia di stati.

Una soluzione ammissibile è un'etichettatura λ che assegna a ogni nodo uno degli stati possibili per i caratteri C.
La nostra funzione obiettivo andrà quindi minimizzata:

$$min \sum_{c \in C} \sum_{(x,y) \in E(T)} w_c(\lambda_c(x), \lambda_c(y))$$

dove $E(T)$ è l'insieme dei lati di T.

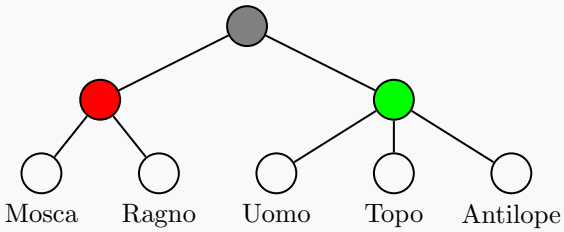
Esempio 21.1: Piccola parsimonia

Si ha in input una matrice N di questo tipo:

# arti	0	1	2	4	6	8
0	0	1000	2000	4000	6000	8000
1		0	60	150	190	232
2			0	30	70	120
4				0	25	80
6					0	10
8						0

Si indica che il costo di passare da 0 a 1 arto è elevato, da 0 a 2 arti ancora più elevato, da 6 a 8 arti invece è meno elevato, eccetera (in pratica, si assegna un costo legato al numero di arti che la specie aveva prima e dopo essersi evoluta, completamente inventato). È un esempio di caratteri non binari, possono avere più stati. Una matrice di questo tipo è *parte dell'istanza.*

M	# arti	Gene X
Topo	4	0
Uomo	4	0
Ragno	8	1
Mosca	6	0
Antilope	4	0



Nella piccola parsimonia bisogna capire qual è l'etichettatura dei nodi interni (grigio, rosso e verde) in modo che la *somma totale dei costi su ogni arco sia minimizzata.* Si possono trattare separatamente ogni singolo carattere, poiché questi sono **indipendenti**, quindi l'algoritmo è scritto in maniera tale da trattare un carattere per volta.

21.3 Algoritmo di Sankoff

Questo è un algoritmo risolutivo per il problema della piccola parsimonia. Ogni carattere è completamente indipendente, quindi si può lavorare su ognuno singolarmente.

Per determinare l'etichettatura ottimale, è possibile sfruttare la programmazione dinamica salendo dalle foglie fino alla radice e usando un insieme di sottoistanze ordinate.

L'etichettatura di ogni nodo è l'ultima componente della soluzione nei nodi precedenti. L'ordinamento nell'albero è parziale, ma è sufficiente esaminare le sottoistanze formate dai sottoalberi. Per ogni nodo x nell'albero T, il valore $sol(x, y)$ nella matrice sarà la soluzione ottimale del sottoalbero di T che ha radice X, con etichetta di x uguale a y.

Formalmente, $sol(x, y) = min_{z_1, \ ..., \ z_n}\{\sum_{i=1}^{h} sol(f_i, z_i) + d(z_i, y)\}$, dove $\{f_i\}$ sono tutti i figli di $x_i T$ e $\{z_i\}$ sono tutte le possibili etichette di f_i.

In altre parole:

- $M[x, z]$ è la soluzione ottimale del sottoalbero di T che ha radice x, sotto la condizione che x abbia etichetta z;
- $M[x, z] = 0$ se x è una foglia con etichetta (conosciuta, perché presente nella matrice) z;
- $M[x, z] = +\infty$ se x è una foglia con etichetta diversa da z (insensata);
- $M[x, z] = \sum_{f \in F(x)} min_s\{w(z, s) + M[f, s]\}$ dove $f(x)$ è l'insieme dei figli in x di T, se x è un nodo interno;
- La soluzione ottimale è $min_s\{M[r, s]\}$ dove r è la radice di T.

A differenza dell'allineamento, le istanze sono alberi e non stringhe o sequenze; i casi possibili dell'ultimo componente dipendono dal numero di caratteri dell'etichetta, per questo ci si appoggia sull'indice z.

Per ogni figlio dell'albero viene calcolata la funzione *min* (tutte le possibilità), e la soluzione del padre viene ricavata da quella dei figli. Ogni cambiamento ha un costo: se i figli sono già etichettati nel modo desiderato allora esso sarà 0.

Il costo dell'arco (x, f_1) è 0 se l'etichetta è uguale a y, 1 altrimenti.
z rappresenta ogni etichetta diversa da y.

21.4 Algoritmo di Fitch

Funziona solo per il caso non pesato, in cui T è un albero binario. Anch'esso sfrutta la programmazione dinamica, utilizzando le seguenti regole:

1. $S(x) = \delta_c(x)$ se x è una foglia;
2. $S(x) = S(f_l) \cap S(f_r)$ dove f_l e f_r sono i figli di x in T, se $S(f_l) \cap S(f_r) \neq \emptyset$.
   Se l'intersezione non è vuota, si etichetta il padre con l'**intersezione** dei figli;
3. $S(x) = S(f_l) \cup S(f_r)$ dove f_l e f_r sono i figli di x in T, se $S(f_l) \cap S(f_r) = \emptyset$.
   Se l'intersezione è vuota, si etichetta il padre con l'**unione** dei figli.

La soluzione $B(x)$ consiste nell'insieme degli stati z tale che $M[x, z]$ (il costo del sottoalbero che ha radice x e con stato z) sia minimo, con $B(x) = S(x)$. Quindi $B(x)$ è lo stato migliore che posso ottenere con un determinato nodo x. L'algoritmo di Fitch funziona solo con l'albero binario: come estenderlo al caso generico (sempre non pesato)?

Per generalizzare, si usa un'etichettatura con caratteristiche (colori) non pesate, di cui ognuna vale 1, e si cerca l'insieme degli stati che compare il maggior numero di volte.

22 Filogenesi su distanze

L'idea di base di questo approccio è che, più piccola è la distanza (non negativa) tra due specie, più simili esse sono.

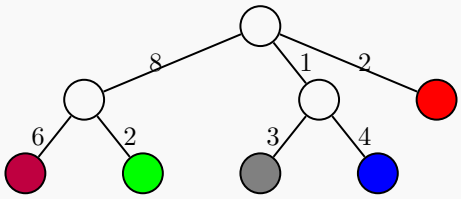
La distanza è una funzione $S \times S \to \mathbb{R}^+$, dove S è l'insieme delle specie. Si ricordano le proprietà:

- Simmetria;
- $d(x, y) = 0 \leftrightarrow x = y$;
- Disuguaglianza triangolare.

L'input è una matrice (simmetrica) in cui, per ogni coppia di individuo, è contenuta una sua stima della distanza dal punto di vista evolutivo. Ogni arco è etichettato con la misura del tempo trascorso dall'evento di speciazione a oggi. Si otterranno alberi senza radice (questo è un problema).

Esempio 22.1

M	M	U	R	T	A
Mosca	-	10	2	12	9
Uomo		-	9	2	3
Ragno			-	13	14
Topo				-	4
Antilope					-



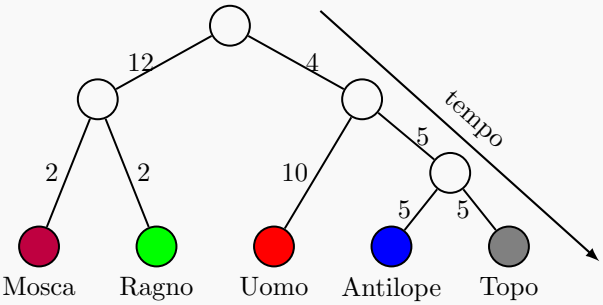
Per ogni arco c'è un intervallo temporale, ed è possibile trovare una matrice delle distanze indotte da T (es. tra il nodo grigio e quello blu c'è una distanza di $4 + 3 = 7$).

Il problema da risolvere è: data una matrice numerica M di distanze, trovare un albero T la cui matrice di distanza indotta M_T abbia minima distanza da M (sia identica, o abbastanza simile, a quella in input). Non sempre è possibile trovare l'istanza minima, quindi a volte è sufficiente ottenere una matrice abbastanza simile.

Quando una matrice M ha un albero T che la induce?

22.1 Orologio molecolare

Esempio 22.2



La freccia indica che il tempo scorre in egual misura per tutte le specie. Quindi, se il tempo è misurato correttamente, la distanza tra tutti i percorsi da una foglia qualunque alla radice deve essere uguale. Questo deve valere anche per ogni nodo interno: *la distanza fra ogni nodo e tutte le sue foglie deve essere uguale.*

Questo, però, presuppone che si riesca a calcolare una misura corretta del tempo: si possono confrontare solo le sequenze attualmente a disposizione.

L'ipotesi che la differenza genomica sia in funzione proporzionale al tempo trascorso è chiamata **ipotesi dell'orologio molecolare**, cioè l'ipotesi che il numero di variazioni genomiche sia una misura precisa dello scorrere del tempo.

L'ipotesi non è però vera in maniera assoluta: è troppo semplificativa, quindi è utile solamente per trattare il concetto di distanza. Il caso è ideale, non reale, altrimenti ogni coppia di figli dovrebbe avere la stessa etichetta che li collega al proprio genitore (il che è improbabile).
In questo caso, la matrice delle distanze soddisfa la definizione di **ultrametrica**.

22.2 Ultrametrica

Comunque siano prese tre specie, vengono considerate le distanze tra tutte e tre le coppie di punti:

$$\forall\ s_1, s_2, s_3 \quad max\{d(s_1, s_2), d(s_1, s_3), d(s_2, s_3)\} \quad \text{è realizzato da almeno due coppie.}$$

Esempio 22.3

Per esempio, si hanno Uomo(s_1), Antilope(s_2) e Topo(s_3):

$$\begin{aligned} d(s_1, s_2) &= 10 + 5 + 5 & &= 20 \\ d(s_1, s_3) &= 10 + 5 + 5 & &= 20 \\ d(s_2, s_3) &= 5 + 5 & &= 10 \end{aligned}$$

Due saranno tipicamente più vicini tra di loro (s_2, s_3), il resto invece ha valori massimi uguali (uguaglianza triangolare).

Le matrici ultrametriche permettono di usare un algoritmo efficiente per ottenere l'unico albero associato.

Ogni metodo basato su distanze, senza l'ultrametrica che dà una proprietà in più, non è in grado di individuare la radice dell'albero (perché la matrice è simmetrica). Questo è il motivo per cui spesso gli alberi sono senza radice, ed è un problema: la radice serve appunto per capire se le specie hanno dei nodi in comune.

Figure 9: Albero senza radice

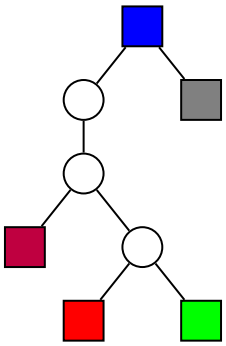
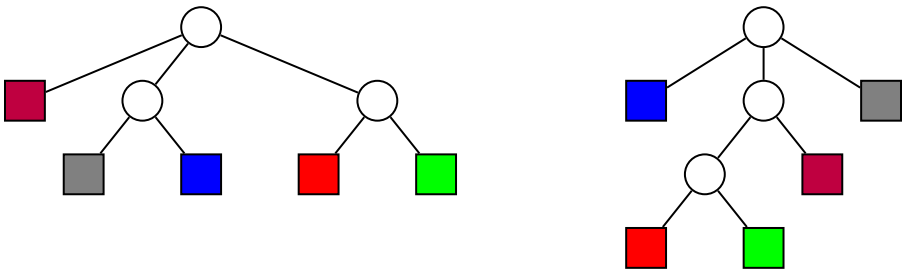


Figure 10: Alberi con radice



Gli alberi con radice danno delle storie evolutive profondamente diverse: è necessario quindi trovare il modo di capire quale dei nodi interni sia il miglior candidato a radice.

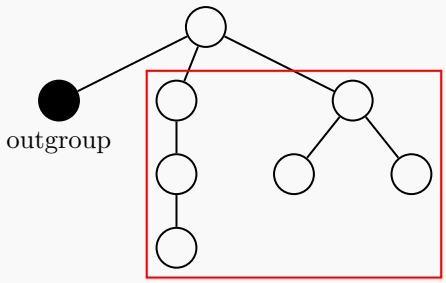
Per poterlo fare, bisogna aggiungere dei cosiddetti **outgroup**, specie che non c'entrano niente con il soggetto da analizzare.

22.3 Outgroup

L'introduzione dell'outgroup serve da una parte per individuare la radice, e dell'altra come *controllo qualità*. È inoltre comune a tutti gli algoritmi che si basano sulle distanze.

A volte si possono introdurre più specie outgroup. Un esempio può essere il seguente:

Esempio 22.4



La parte evidenziata è la parte di interesse, in cui il nodo padre dell'outgroup viene preso come radice. L'albero evolutivo avrà subito da una parte l'outgroup e dall'altra le specie che interessano.

Se gli outgroup sono nel mezzo e non vicini fra loro e alla radice, la matrice di partenza non era corretta.

23 Ricostruzione dell'albero

Se l'albero è binario, ogni nodo ha esattamente due figli, quindi la speciazione può essere vista come una divisione in due, e la matrice delle distanze prende il nome di *distanza additiva*.

Sia T un albero binario senza radice, e D la matrice ultrametrica delle distanze totali associata (si ricorda che D è simmetrica). Si vuole calcolare l'albero T che induce una matrice D_T simile a D.

Quando una matrice M ha un albero T che la induce? Per capire se esiste l'albero, si controllano tutte le **quaterne** di punti.

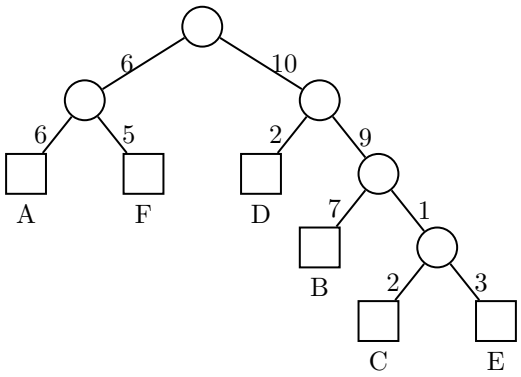
D è una distanza additiva se soddisfa la condizione dei 4 punti:

1. $D[v, w] + D[x, y]$;
2. $D[v, x] + D[w, y]$;
3. $D[v, y] + D[w, x]$.

Il massimo dei tre valori è ottenuto esattamente da due di questi tre casi. La condizione è anche inversa: se questa proprietà è soddisfatta, la distanza è additiva.

Si ha la matrice M associata a T, e si vuole ottenere una proprietà della matrice: l'idea di base è che le distanze non possano essere completamente slegate tra loro, ma rispettino la condizione dei quattro punti.

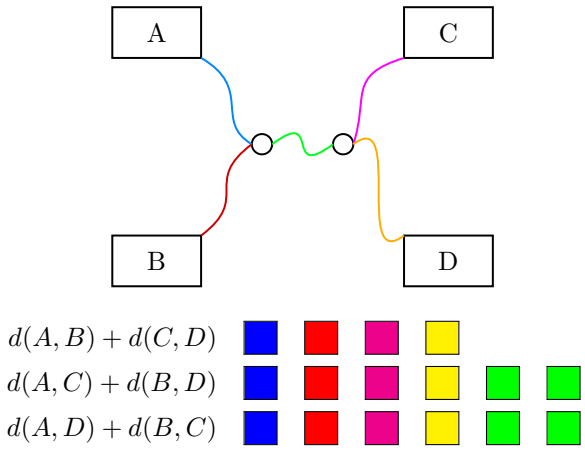
Figure 11: Albero generico binario, distanza additiva



In un albero, vengono estratte quattro specie (foglie), considerando soltanto la porzione di cammini che li collega. Essendo l'albero binario, la struttura sarà sempre la stessa. Sono poi considerate tutte le coppie di distanze: due di esse avranno un valore più elevato, e uno sarà minore. Questo accade perché per due coppie non sarà necessario attraversare il cammino intermedio.

Per calcolare le distanze è sufficiente sfruttare la struttura dell'albero, non necessariamente i valori. Se un cammino ha etichetta 0, la disuguaglianza diventa uguaglianza, e per posizionarlo si contrae un arco.

Figure 12: Condizione dei quattro punti



23.1 Procedura con minimo valore

L'algoritmo è molto diverso da quelli visti finora: funziona solo nel caso di alberi con radice, e si basa sulla minima distanza. L'input è una matrice ultrametrica D.

Il procedimento segue questi step:

1. Viene considerato sempre il valore della minima distanza d, per individuare le specie più vicine;
2. Queste avranno un genitore in comune;
3. Si ha che, per ogni nodo, la distanza fra esso e tutte le sue foglie dev'essere uguale a d;
4. Le etichette degli archi tra ogni specie e il padre avranno valore $d/2$;
5. La matrice viene aggiornata, togliendo una delle due specie;
6. Viene cercato il nuovo valore minimo;
7. Se un nodo ha già un genitore, non si deve dimezzare la distanza, ma si deve tenere in considerazione l'etichetta esistente;
8. Sia a il valore dell'arco, viene creato un altro arco sopra, con $d/2 - a$ come etichetta.

23.2 Procedura con minimo squilibrio

Questa procedura è simile alla precedente, e si basa sulla disuguaglianza triangolare. Si ha che tutte le foglie sono incidenti (collegate) su un arco solo, e si vuole trovare l'etichetta minima.

Togliendo una quantità δ da tutti i pesi degli archi incidenti sulle foglie si ottiene un nuovo albero, la cui matrice corrispondente avrà 2δ sottratto a tutti i valori.

Per ogni $< s_1, s_2, s_3 >$ con minimo squilibrio δ, si sottrae δ a tutti i valori in D (e $\delta/2$ a ogni arco), una specie sarà inclusa in un nodo interno (perché la nuova distanza sarà 0). Essa viene salvata,

rimossa e il processo viene ripetuto fino ad arrivare a un solo nodo. A ritroso poi viene ricostruito l'albero.

In altre parole, partendo dalla matrice D e togliendo $\delta/2$ a tutti i numeri, si ha un albero contratto (anche non conosciuto) in cui ogni percorso foglia-foglia attraversa esattamente due archi ridotti.

Vale la disuguaglianza triangolare: date specie s_1, s_2, s_3, si ha che $D(s_1, s_2) \leq D(s_1, s_3) + D(s_2, s_3)$, cioè $D(s_1, s_2) \geq D(s_1, s_3) - D(s_2, s_3)$. Nell'albero senza radice, quando vale l'uguaglianza?

Si ha che il valore dello squilibrio è 2 volte la lunghezza dell'arco incidente sulla foglia la cui lettera non appare nel termine $D(s_1, s_2)$.

Per ogni terna di specie, viene calcolata la distanza, che è sempre maggiore o uguale a 0 (l'eccesso viene definito squilibrio). Il minimo δ corrisponde al peso minimo di un arco incidente a una foglia: sottraendo un $\delta/2$ (e collassando l'albero), è possibile togliere dalla matrice la specie corrispondente e reiterare. Nonostante le caselle in D siano diverse da 0, l'intera riga corrispondente alla specie viene eliminata.

	M	U	R	T	A
Mosca	-	14	7	14	19
Uomo		-	17	8	5
Ragno			-	17	22
Topo				-	9
Antilope					-

$\rightarrow$

	M	U	R	T	A
Mosca	-	14 - 2δ	7 - 2δ	14 -2δ	19 - 2δ
Uomo		-	17 - 2δ	8 - 2δ	5 - 2δ
Ragno			-	17 - 2δ	22 - 2δ
Topo				-	9 - 2δ
Antilope					-

Se l'albero è senza radice, perché sia binario deve semplicemente rispettare la condizione che tutti i nodi interni abbiano 3 vicini.

Il problema principale di questo approccio è che, se la matrice D non soddisfa le condizioni dei 4 punti, non è possibile costruire un albero T associato: si vuole estendere (ottimizzare) l'algoritmo in modo che esso soddisfi anche il caso generale.

La più semplice nozione di "più vicino" è quella di varianza minima, cioè la somma dei quadrati degli scarti o del valore assoluto. Nonostante questa formulazione sia banale, il problema corrispondente è NP-hard: la strategia che viene usata in pratica è il clustering gerarchico.

24 Clustering gerarchico

L'idea dell'algoritmo è la continua fusione (iterativa) di parti di alberi fino all'ottenimento dell'albero completo. Questo funziona anche se la matrice di input non è ultrametrica o distanza additiva, ma ha proprietà simili (essendo anch'essa una distanza, è simmetrica).

Un albero è un insieme di cluster gerarchici, di cui alcuni sono inclusi in altri. Vedendo un insieme di foglie come un cluster, si cerca di unirli con una procedura intuitiva.

La decisione a ogni passo è quella dei due gruppi da fondere, basandosi sulla matrice delle distanze in input.

24.1 UPGMA

Viene utilizzata *UPGMA* (Unweighted Pair Group with Arithmetic Mean), cioè la media delle distanze. Il valore minore sarà formato dalla coppia da eliminare, prendendo in considerazione un valore h che rappresenta l'etichetta dell'arco.

$$D(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{i \in C_1} \sum_{j \in C_2} D(i, j)$$

All'inizio, $h = 0$ per ogni specie: i due cluster con distanza minima vengono fusi, ottenendo C. Il processo viene poi ripetuto per tutti gli altri, ricalcolando la distanza. Si ha che $h(C) = \frac{1}{2} D(C_1, C_2)$, e ogni etichetta (C, C_1) viene ottenuta con $h(C) - h(C_1)$.

Il presupposto è che ogni cluster già contenga nodi accorpati con distanza minima: al momento dell'unione, viene creato un nuovo nodo interno tale che il valore delle distanze h sia identico.

L'albero ottenuto alla fine è un'ultrametrica, cioè la distanza tra la radice e tutte le sue foglie è la stessa. Questo algoritmo, però, ha un limite: funziona solo con cluster vicini, e fallisce con un numero elevato di essi.

24.2 Neighbor-joining

Neighbor-joining controlla, per ogni cluster, quanto esso è lontano dagli altri: l'idea di base è che la fusione avvenga tra gruppi (classi) vicini tra di loro ma distanti da tutti gli altri.

Viene introdotta una nuova quantità $u(C)$, che rappresenta la misura di quanto un cluster sia separato da tutti gli altri. Il fattore $n - 2$ al denominatore è una correzione del termine.

$$D(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{i \in C_1} \sum_{j \in C_2} D(i, j)$$

$$u(C) = \frac{1}{\text{num. cluster} - 2} \sum_{C_3} D(C, C_3)$$

All'inizio, $h = 0$ per ogni specie: i due cluster che hanno minimo $D(C_1, C_2) - u(C_1) - u(C_2)$ vengono fusi, ottenendo C. Le etichette sono ottenute con $\frac{1}{2}(D(C_1, C_2) + u(C_1) - u(C_2))$.

Vale la proprietà della consistenza: al tendere di n a infinito (lunghezza della sequenza), la probabilità che l'algoritmo costruisca l'albero corretto tende a 1. Neighbor-joining, però, è completamente deterministico.

La procedura è relativamente semplice da implementare e funziona in un tempo computazionale cubico rispetto al numero di specie. Neighbor-joining è uno dei metodi più usati per la costruzione di filogenesi.

La matrice delle distanze D è ottenuta con allineamento, senza tenere in considerazione l'ipotesi dell'orologio molecolare (quindi con risultati approssimati, ma piuttosto accurati).

24.3 Modelli di evoluzione

Un altro algoritmo molto comune segue l'ipotesi di massima verosomiglianza, e utilizza la matrice delle sequenze genomiche. La precisione del metodo è maggiore, a discapito del tempo computazionale. Con le nuove risorse di calcolo, però, è possibile calcolare gli alberi.

Il modello di evoluzione dipende dalla probabilità di transizione fra stati (A, C, T, G), considerando anche il tempo trascorso tra i due eventi. Un altro fattore che ha influenza è il tasso istantaneo di mutazione: in un intervallo temporale ϵ, si vuole sapere la probabilità che avvenga un cambiamento.

Il tasso istantaneo è la parte realmente interessante, e da esso è possibile calcolare la variazione in relazione al tempo. La probabilità di mutazione in una generazione ha come sonna di ogni riga 1.

Jukes-Cantor è uno dei modelli di evoluzione più semplice: si basa sul concetto che ogni mutazione sia equiprobabile con parametro μ: Il fatto che nessuna mutazione avvenga ha probabilità $1 - \mu$, mentre una mutazione avviene con probabilità $\mu/3$ essendo 3 i nucleotidi rimanenti.

Kimura è più complesso, si basa su due parametri μ e R e prende in considerazioni transizioni e trasversioni. Una transizione è una mutazione $A \leftrightarrow G$ o $C \leftrightarrow T$, mentre una trasversione è qualsiasi altra mutazione. Le transizioni sono molto più frequenti.

Si ha che:

- Nessuna mutazione avviene con probabilità $1 - \mu$;
- Una transizione ha probabilità $\frac{R}{R+1}\mu$;
- Una trasversione ha probabilità $\frac{R}{2(R+1)}\mu$, di cui le possibilità sono:
  - $A \leftrightarrow C$;
  - $G \leftrightarrow T$;
  - $A \leftrightarrow T$;
  - $C \leftrightarrow G$;
- $R = \frac{R}{R+1}\mu/(2\frac{1}{2(R+1)}\mu)$ è il rapporto tra le probabilità delle transizioni e quella delle trasversioni (generalmente 2).

Un'estensione di questi due modelli di evoluzione è definita **General time-reversible**: rende possibile ogni valore con la condizione che la matrice sia simmetrica. In questo modo, la probabilità di una mutazione $A \to T$ è uguale a quella di $T \to A$.

Un numero elevato di parametri genera overfitting e risulta inutile, ma dal punto di vista della biologia non è possibile vincolarne alcuni, quindi questo modello è troppo generale.

Inoltre, la conseguenza della matrice simmetrica è che genera alberi senza radice (con lo stesso valore di verosomiglianza qualsiasi sia la radice).