Q & A Bioinformatica

Parte di Botta - Pattern Matching

Z-Algorithm

Lo Z-Algorithm funziona come efficace metodo di preprocessing per il problema del pattern matching singolo in tempo lineare. Qui di seguito verrà scritto il sentimento di questa tecnica in pillole:

- **Calcolo degli Zi**: Data una stringa, si va alla posizione i e si confronta quel carattere con il primo della stringa. Se fa match, allora proseguo (considero il carattere i+1 e lo confronto col carattere 2). Mi fermo al primo mismatch. Zi sarà quindi uguale al numero di match incontrati fino al primo mismatch.

- **Costruzione di uno z-box**: lo z-box è un tipo di costruzione che si può creare a partire dalla conoscenza degli Zi che consente di visualizzare graficamente le aree dove avviene un match. Servono quindi 2 elementi per determinare gli estremi di uno z-box: la posizione i ed il valore di Zi. I due estremi di uno zbox, li ed ri si calcolano come segue :
    1. li = i
    2. ri = i + Zi - 1

Per **garantire la linearità** quello che dobbiamo fare è calcolare gli Zi successivi senza dover ogni volta fare tutti i confronti possibili sulla sottostringa, bensì usando i valori di **li, Zi ed ri precedenti.** L'idea è usare un approccio per casi. Supponiamo quindi di dover trovare Zk con k > i. Ecco a cosa fare attenzione.

1. **k > ri** allora **non abbiamo nessuna info** precedente che ci può aiutare a fare **prediction**. Dobbiamo usare l'approccio naif.
2. **k < ri** allora il nuovo Zbox inizia di sicuro all'interno di uno zbox precedente. possiamo dunque avere 2 sottocasi:
    a. Lo zbox successivo **sfora quello precedente**: allora chiamiamo q lo sforamento ed aggiorniamo i valori come segue: **Zk = q-k , li = k, ri = q-1**
    b. Lo zbox successivo **è uguale ad uno individuato in precedenza.** Allora il carattere S(k) compare anche nella posizione **k' = k - l + 1**. posso tenere **uguali l ed r** e dico che **Zk = Zk'**.

Questa tecnica da sola ci dà un buon algoritmo di pattern matching, basta infatti usare come stringa **S = P$T** è poi **contare tutti gli Zi di cardinalità uguale a quella di P**. Dato che il **$ non compare in T** questi specifici Zi devono essere per forza occorrenze di P. C'è un teorema visto a lezione che garantisce che gli Zi vengono calcolati **tutti** per i > 2 è che l ed r vengono aggiornati correttamente.

Boyer-Moore

L'algoritmo di Boyer-Moore consiste in qualche variante rispetto a prima. Ad esempio il pattern viene confrontato col testo da destra verso sinistra. L'idea è la seguente: al primo mismatch che troviamo spostiamo il pattern a destra fin quando non riusciamo ad incolonnare due lettere che fanno match.

Regola del **bad-character shift**: una volta trovato un mismatch col pattern (x è il carattere di T che non fa match):

- Scansiono il pattern **da sx verso dx** e scelgo **l'occorrenza di x in P più a destra**
- Avrò quindi individuato una distanza y che separa le due occorrenze di x, per cui sposto tutto il pattern a destra di y, evitando altri confronti inutili

Regola del **bad-character shift estesa**: per alfabeti con pochi caratteri (tipo DNA) conviene usare questa regola estesa. Sostanzialmente è uguale, ma si scansiona il pattern **da dx verso sx** partendo **direttamente dalla posizione in cui è avvenuto un mismatch**. Si prende poi la **prima occorrenza** a sinistra di x come nuovo riferimento.

Regola dello **strong good suffix**: in breve, se t è una **sottostringa che matcha un suffisso di P** ed **x il carattere subito prima che ha fatto mismatch**, quello che posso fare è **spostare a destra P** fin quando non incolonno **un'altra occorrenza t' che matcha sempre un suffisso di P** al netto del carattere subito prima y con **y != x**. Questo non lo faccio con la sola idea di trovare y che matchi, ma soprattutto perché così sposto di tanto a destra P velocizzando di molto la scansione generale.

I Teoremi forniti a lezione ci garantiscono che nello spostarci a destra non ci spostiamo comunque **mai** più di **un'intera occorrenza di P**.

Fase di preprocessing in Boyer-Moore

Definizione di L(i): per ogni suffisso di P, cerco una sua occorrenza precedente e definisco L(i) come il suo ultimo carattere. es. CABDABDAB , L(i) = 6.

Definizione di L'(i): per ogni suffisso di P, cerco la prima sottostringa diversa da un suffisso subito precedente un suffisso. Definisco poi L'(i) il suo ultimo carattere, es. CABDABDAB , L'(i) = 3.

L'ultima cosa è definire gli Nj(P) : sostanzialmente è uno Z-Algorithm al contrario (applicato cioè ad una sottostringa reversata), però quello che significa è sostanzialmente presa una sottostringa di P lunga j **P[1...j]**, determinare il più lungo suffisso di questa entità che sia anche suffisso di P. es. CABDABDAB con N3(P) = CABDABDAB = 2.

Knuth Morris Pratt

L'importante in questo algoritmo (sebbene performi peggio di Boyer-Moore) è il calcolo di sp(i) ed sp'(i).

Partiamo dagli sp(i): data una stringa, es. ABCAEABCABD ed un i, ad esempio 4, andiamo a cercare dentro il prefisso [1...i] il suffisso più lungo uguale ad un prefisso. Nel nostro caso quindi: ABCAEABCABD è quindi su ABCA il suffisso maggiore è quello di lunghezza 1 (ABCA).

Per quanto riguarda gli sp'(i), possiamo dire che sono la stessa cosa degli sp(i) con l'unica accortezza di interrompersi subito se si incontrano ripetizioni di caratteri consecutive, ad esempio BBCCEABB (non va bene perché ho trovato 2 B consecutive).

L'algoritmo funziona nel modo seguente:
Per ogni allineamento, se il primo mismatch si presenta in posizione i+1 di P e
k di T, sposto P a destra in modo che P[1,...,sp(i)] sia allineato con T[k-sp(i)....k-1].
Se si trova un'occorrenza di P, sposto P di n − sp(n) posizioni.

Motif Finding

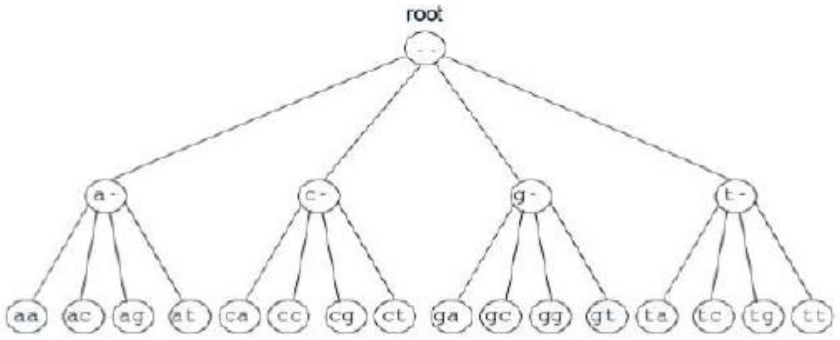
È una tecnica per trovare dei **candidati pattern** quando si sta studiando un genoma per la prima volta. Come problema giocattolo si può pensare a trovare i motifi più probabili tra tutte le stringhe di 15 caratteri all'interno di un genoma di 600 caratteri. A quel punto si effettua un'**analisi delle frequenze** e si stabilisce **il candidato più probabile**. A partire da questo candidato si **calcolano le differenze con gli altri candidati** in termini di **frequenze**. Ma questo solo in generale. Come **rappresentazione grafica** si utilizza il **Motif Logo**.

A questo punto si calcola la matrice di consenso come matrice delle varie frequenze. Ora, il problema è che noi vogliamo trovare la stringa di consenso ottima tra tutte le possibili (nel nostro esempio giocattolo abbiamo tutte le combinazioni C(n,k) con n=600 è k=15, ovvero 301 403 213 218 069 597 428 666 705 840 candidati possibili. Figurarsi con n=3.4 GB e k=100…..

Non va bene, perché diventa a **complessità esponenziale**…

Si è proceduto quindi ad un cambio di visione considerando il problema della stringa mediana. Idea: anzichè provare le combinazioni di T, provo invece tutti i possibili motif (es. Da AA…..AA a TT….TT) . Utilizziamo il concetto di **distanza di Hamming**, d(v,w), ovvero il **numero di mismatch** che ci sono tra le stringhe v e w.

Un ulteriore step è quello di convertire il problema in numeri, in modo da poter fare i conti in base 4 tolto lo zero. Si costruisce quindi una sorta di albero dei prefissi



Una funzione comoda che possiamo inventarci è quella che chiamiamo **nextLeaf** che ci consente di **esplorare** la foglia successiva. In questo algoritmo, **a** è il nostro **array** di cifre in base 4, **L** la **lunghezza** di a e **k** è il **massimo valore** di una cifra. In sostanza l'algoritmo parte dalla cifra meno significativa ed incrementa man mano di 1 spostandosi verso sinistra.

Algorithm 3 NextLeaf

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

Un aiuto ci viene da questo esempio:

nextLeaf(21,2,4) = 22, infatti
21
2 (1+1)
22 (2 non è < 2, per cui mi fermo)

Un altro algoritmo utile è il nextVertex, col cui navighiamo al nodo successivo anche se non è una foglia.

Algorithm 5 NextVertex

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

Praticamente la parte sopra è la vera parte nuova, ovvero l'incremento di 1 se i < L di a(i+1). Sotto è una nextLeaf.

L'ultimo algoritmo è il bypass:

Algorithm 6 Bypass

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

Praticamente **bypass(2-, 1, 4) = 3(0)** (ho saltato tutto il sottoalbero 2X)

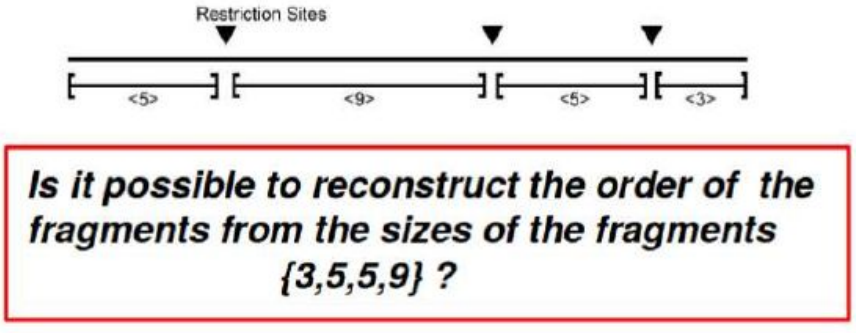
bypass(22,1,4) = 2(0) (sono salito su al mio nodo padre)

Con queste primitive è possibile implementare un'algoritmo di tipo **Branch & Bound**, saltando alcuni sottoalberi quando il punteggio che ricaveremmo è minore di quello nostro attuale (ad esempio all'inizio e alla fine dell'albero, quando compariamo con 11...11 e 44….44).

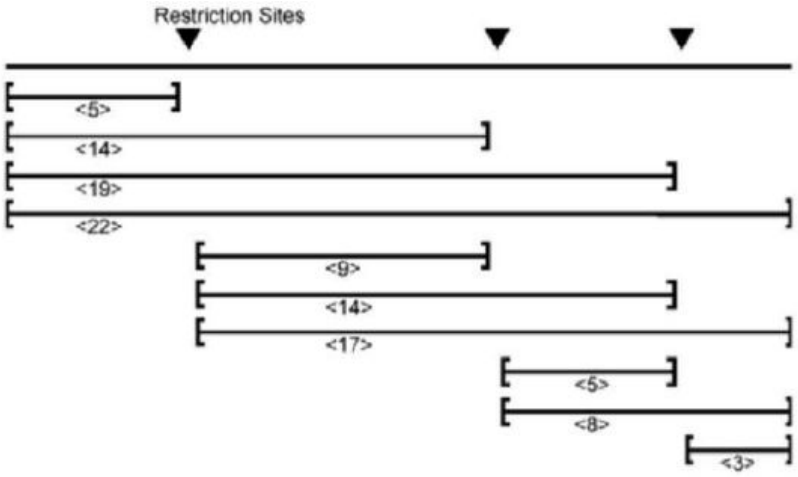
```
BranchAndBoundMotifSearch(DNA,t,n,ℓ)
s ← (1,...,1)
bestScore ← 0
i ← 1
while i > 0
    if i < t
            optimisticScore ← Score(s, i, DNA) +(t – i ) * ℓ
            if optimisticScore < bestScore
              (s, i) ← Bypass(s,i, n-ℓ+1)
            else
              (s, i) ← NextVertex(s, i, n-ℓ+1)
    else
            if Score(s,DNA) > bestScore
              bestScore ← Score(s)
              bestMotif ← (s1, s2, s3, ..., st)
          (s,i) ← NextVertex(s,i,t,n-ℓ+ 1)
return bestMotif
```

Partial Digest Problem

È un problema intrinseco nella natura del **sequenziamento**. Per farla breve, si usa un approccio su un DNA da sequenziare per step successivi, nei quali si individuano alcuni **siti** in cui la stringa è stata divisa. Ora, il nostro problema nasce dal fatto che dobbiamo trovare il modo di “riattaccare” queste sottostringhe “come erano originariamente”.



Nel nostro problema abbiamo quindi a che fare con **Multiset**. Quello che dobbiamo fare è risalire in base a questo multiset ad un unica stringa originale di partenza. Per esempio, possiamo avere la seguente situazione:



Quindi il nostro multiset sarà X = {3, 5, 5, 8, 9, 14, 14, 17, 19, 22} (ordinando i numeri).

Si può impostare questo problema con un cambio di notazione ed il seguente schema:

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

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

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

Ora, come risolvere il problema ? Un modo possibile è l'**approccio Bruteforce**. Come prima cosa dobbiamo fissare le lunghezze min e max. Questo per fortuna è semplice dato che **min=0 e max= la stringa più lunga che ho**.
Successivamente, **tenendo fissi questi due valori**, in mezzo inserisco via via tutte le **combinazioni senza ripetizioni degli altri valori.** Ad esempio:

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

```
BruteForcePDP(L, n)
    M <- maximum element in L
    for every set of n – 2 integers 0 < x2 < ... xn-1 < M
        X <- {0,x2,...,xn-1,M}
        Form DX from X
        if DX = L
            return X
    output "no solution"
```

La complessità è quindi **esponenziale**, O(N^m-2). Possiamo migliorare di poco andando a scegliere **solo i valori presenti in L** (ma dobbiamo a questo punto **barare** e farci dare un L in **input**). N.B: All'esame **diffida** da tutte le soluzioni proposte che **non** contemplano la possiblità di **backtracking** !

Per quanto riguarda un buon algoritmo possiamo usare il seguente:

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

Ci manca però la definizione di **Place**. Prima però dobbiamo definire **D(y,X)** che non è altro che un insieme costituito dalle varie **differenze (in modulo)**, t**ra y e ciascun elemento di X**.

```
PLACE(L, X)
 if L is empty
    output X
    return
y <- maximum element in L
Delete(y,L)
if D(y, X ) ⊆ L
    Add y to X and remove lengths D(y, X) from L
    PLACE(L,X )
    Remove y from X and add lengths D(y, X) to L
if D(width-y, X ) ⊆ L
    Add width-y to X and remove lengths D(width-y, X) from L
    PLACE(L,X )
    Remove width-y from X and add lengths D(width-y, X ) to L
return
```

È bene fare il seguente esempio pratico:

$$L = \{2, 2, 3, 3, 4, 5, 6, 7, 8, 10\}$$
$$X = \{0\}$$

Si parte aggiungendo il numero max ad X rimuovendolo da L (solo la prima volta)

$$L = \{2, 2, 3, 3, 4, 5, 6, 7, 8, 10\}$$
$$X = \{0, 10\}$$

Ora, si prende il secondo max e si calcola D(y,X). Nel nostro caso **D(8,X) = {8-0,8-10}**. Quindi possiamo sceglere come candidati **8** e **2**. Nell'esempio si sceglie **arbitrariamente** 2, Ma dato che 2 è **contenuto** il L **dobbiamo calcolare anche D(2,X) = {2,8}**. Rimuoviamo quindi **anche questi 2 elementi** ed aggiungiamo 2 ad X**.**

$$L = \{2, 2, 3, 3, 4, 5, 6, 7, 8\}$$
$$X = \{0, 10\}$$

Ricominciamo quindi da capo.

$$L = \{2, 3, 3, 4, 5, 6, 7\}$$
$$X = \{0, 2, 10\}$$

D(7,X) = {7,3}, scegliamo 7, ricalcoliamo **D(7,X)={7,5,3}**, per cui rimuoviamo anche questi numeri

$$L = \{2, 3, 4, 6\}$$
$$X = \{0, 2, 7, 10\}$$

E così via fino a quando L non si svuota. **Attenzione ! Quando calcolo D(y,X) e trovo N valori, poi devo eliminare tutti quegli N da L.**

Alberi filogenetici

In bioinformatica è molto utile avere uno strumento per graficare le **discendenze** delle specie. Questo strumento si chiama albero filogenetico.
Un **albero filogenetico** ha le seguenti caratteristiche :

1. Le **foglie** corrispondono a varie **specie esistenti**
2. I **nodi a livello superiore** rappresentano man mano **gli antenati più distanti**
3. I **rami** rappresentano uno **step evolutivo ben specifico**
4. La **radice** dell'albero non è altro che l'antenato **più antico conosciuto**

Possiamo avere 2 casi, la **root nota** oppure **no**. Nel primo caso è semplice risalire all'albero, basta graficamente prendere l'albero per la root ed alzarlo. Gli altri nodi si disporranno automaticamente nelle posizioni corrette. Nel secondo caso è un problema, perché dobbiamo attuare qualche strategia.

A questo punto definiamo **d(i,j)** è **D(i,j)** :

1. **d(i,j)** è la **distanza** che intercorre tra due nodi **all'interno della struttura dati** dell'albero (può essere un singolo numero se i e j sono direttamente connessi, oppure una sommatoria se tra i due nodi c'è un path.

2. **D(i,j)** invece è la distanza **"evolutiva"** tra due specie (sempre calcolabile direttamente)

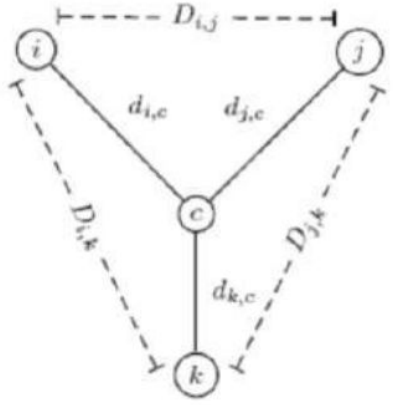
Ora, il discorso è che dobbiamo cercare di far coincidere il più possibile i due valori. Possibilmente, bisogna ricondurlo alla seguente struttura:

Tree reconstruction for any 3x3 matrix is straightforward.

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

Observe:

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



Trucco matematico:

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

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

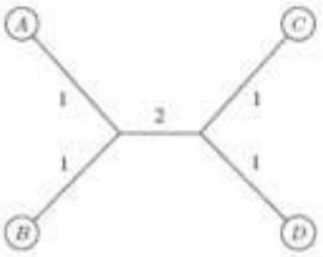
- Data una distanza $d_{x,c}$ i membri della frazione che si sommano sono tutti quelli in cui comprare la x come pedice mentre si sottrae l'unico membro dove non compare.
- Per quanto riguarda la divisione per 2, è necessaria perchè altrimenti calcoleremmo un pezzo di distanza due volte.

Ora, per un albero di n foglie il numero di rami sarà **2n - 3**. Esso equivale a risolvere un sistema col numero di equazioni pari al **coefficiente binomiale** (n su 2). Il set di equazioni **non** è sempre divisibile per 3.

Per ogni albero possiamo costruire la propria **Matrice di distanza**, dove semplicemente riportiamo per ogni etichetta la distanza da tutte le altre.

Additive Matrix

	A	B	C	D
A	0	2	4	4
B	2	0	4	4
C	4	4	0	2
D	4	4	2	0



Nonadditive Matrix

	A	B	C	D
A	0	2	2	2
B	2	0	3	2
C	2	3	0	2
D	2	2	2	0

?

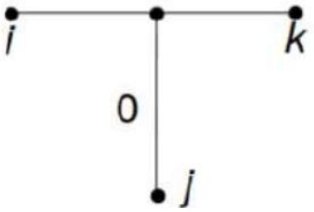
C'è poi una considerazione da fare sul concetto di additività: una matrice è **additiva** se **D(i,j) = d(i,j)**. Ovviamente se non viene soddisfatta questa condizione la matrice è non additiva. Meno formalmente, se tra le righe di una matrice non vedi un minimo di progressività tra i numeri, allora è non additiva.

Neighbor-Joining Algorithm

Questo algoritmo funziona con la seguente idea: si vogliono cercare **foglie "vicine tra di loro"** ma **"lontane da tutte le altre"**. Questo ci da una buona **euristica** che nella pratica funziona bene.

Triple Degeneri

Spesso, per la risoluzione di un problema informatico si sfrutta il concetto di **divide et impera**, oppure si costruiscono soluzioni riconducendosi a **pattern** che conosciamo bene. Nel nostro caso il pattern consiste in **triple degeneri**, ovvero etichette **0 < i,j,k < n** tali che **D(i,j) + D(j,k) = D(i,k)**. Graficamente:



Questo ci dice una cosa molto simpatica: tra i e k c'è in mezzo j. Il che significa che **j** è nello **stesso percorso evolutivo** che c'è **tra i e k** (i 3 nodi sono allo stesso livello), il che vuol dire che j occupa un **ramo** di **peso nullo**. Questo è il trucco che useremo. Scegliamo un valore **delta**, e lo togliamo progressivamente da tutte le distanze della tabella fin quando non troviamo **situazioni "degeneri"**. Alla fine avremo individuato tutte le **gerarchie**, per cui per ricostruire l'albero dovremo solo ripercorrere al **contrario** gli step. (l'ultimo nodo trovato è la **root**). Nota: possiamo anche cambiare il valore di delta tra un'iterazione e l'altra, purchè lo si faccia in modo furbo.

L'algoritmo è il seguente:

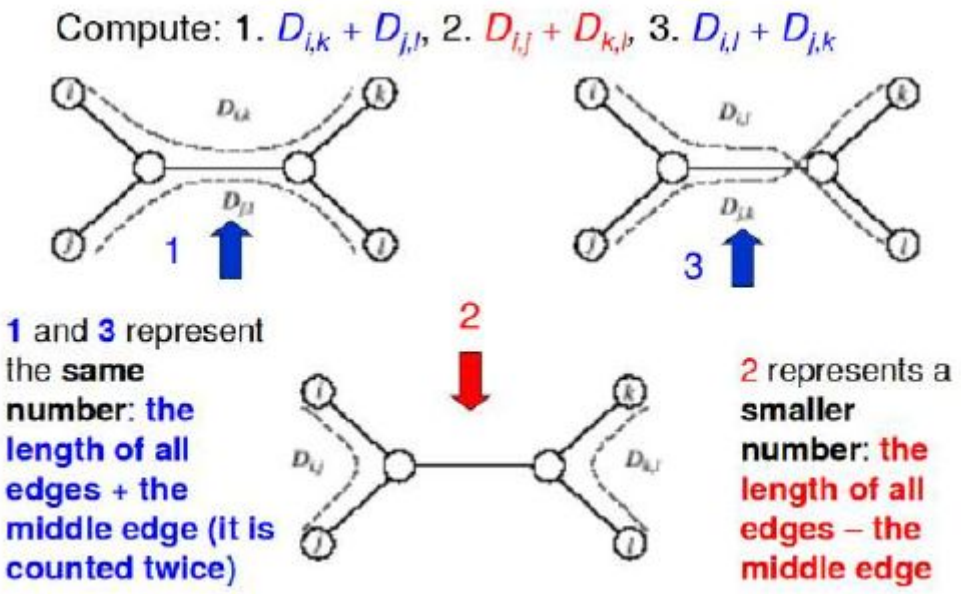
```
AdditivePhylogeny(D)
  if D is a 2 x 2 matrix
    T = tree of a single edge of length D_{1,2}
    return T
  if D is non-degenerate
    δ = trimming parameter of matrix D
    for all 1 ≤ i ≠ j ≤ n
      D_ij = D_ij - 2δ
  else
    δ = 0
Find a triple i, j, k in D such that D_ij + D_jk = D_ik
x = D_ij
Remove j^th row and j^th column from D
T = AdditivePhylogeny(D)
Add a new vertex v to T at distance x from i to k
Add j back to T by creating an edge (v,j) of length 0
for every leaf l in T
  if distance from l to v in the tree ≠ D_{l,j}
    output "matrix is not additive"
    return
Extend all "hanging" edges by length δ
return T
```

Four point condition

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

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

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



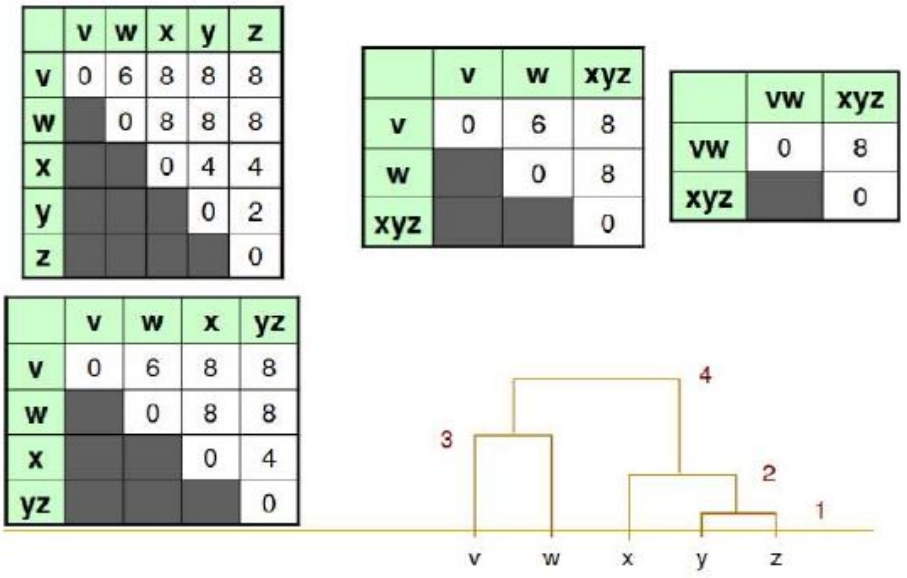
UPGMA

Algoritmo di **clustering gerarchico**. Si parte da tutte le etichette come **cluster** a parte. Si procede poi nel seguente modo.

1. Calcolo da **distanza media** tra **tutti** i cluster.
2. Trovo 2 cluster C1 e C2 con la distanza **minima tra tutte**.
3. Faccio **C1 U C2** e metto i due cluster ad altezza **d(i,j) / 2**.
4. Reitero fin quando non arrivo alla **root**

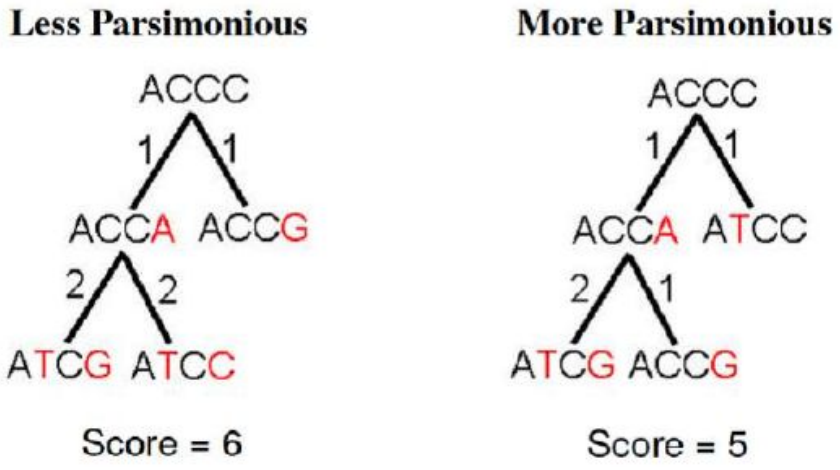
Problema di questo approccio: ciascuna foglia ha la **stessa** distanza con la **root**. Si dice che l'albero è **ultrametrico**. <u>In sostanza c'è troppa normalizzazione, col rischio di perdere informazioni.</u>

Es. grafico:



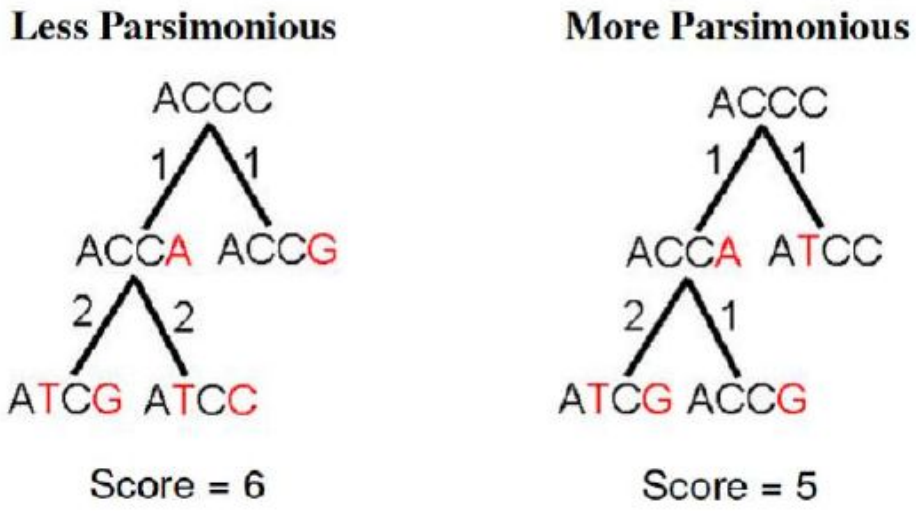
Character based phylogeny:

Il concetto è chiaro: al posto di usare le **distanze**, usiamo direttamente il **dna** ! **Allineiamo** tutte le **sequenze** di dna delle specie e poi **accoppiamo** quelle sequenze che **differiscono** per il **più basso** numero di nucleotidi. Possiamo anche rilassare un po' il concetto scegliendo tratti chiave (es. numero di gambe, colore degli occhi….). Solo a questo punto andiamo a calcolarci le **distanze** ma in termini di **distanza di Hamming** tra i vari nodi.
A questo punto dobbiamo introdurre il concetto di parsimonia. In questo caso però conviene applicare il Rasoio di Occam (un nome diverso del metodo KISS) e fornire un esempio grafico:



Small Parsimony Problem

“A parità di fattori, la soluzione più semplice è quella da preferire” -> Rasoio di Occam



Se utilizziamo **direttamente** il **DNA** ecco che possiamo costruire gli alberi a partire dalle distanze di **Hamming** tra le varie mutazioni. Si vuole **minimizzare** lo “score” tra ciascun **nodo** ed il proprio **padre**, come in figura qui sopra.

Questo problema si configura nelle seguenti modalità:

- Matrice di distanze in input, per esempio, nel caso dello Small Parsimony Problem le differenze sono solo i numeri 0 ed 1, 0 se le lettere sono uguali, uno se non lo sono :

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

- Nel caso in cui per esempio si sappia da conoscenze pregresse, ad esempio che la lettera x muta più facilmente in y piuttosto che in z, posso cambiare la funzione di score e ottenere il **Weighted Parsimony Problem**:

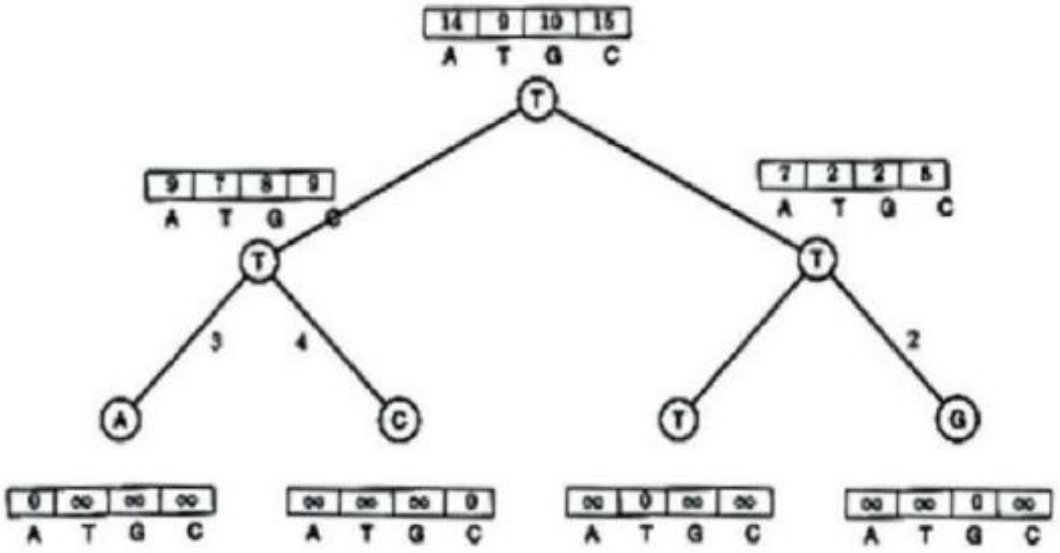
Weighted Parsimony Problem

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

Algoritmo di Sankoff

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

Il risultato è il seguente:



Adesso arriverà il commento chiarificatore:

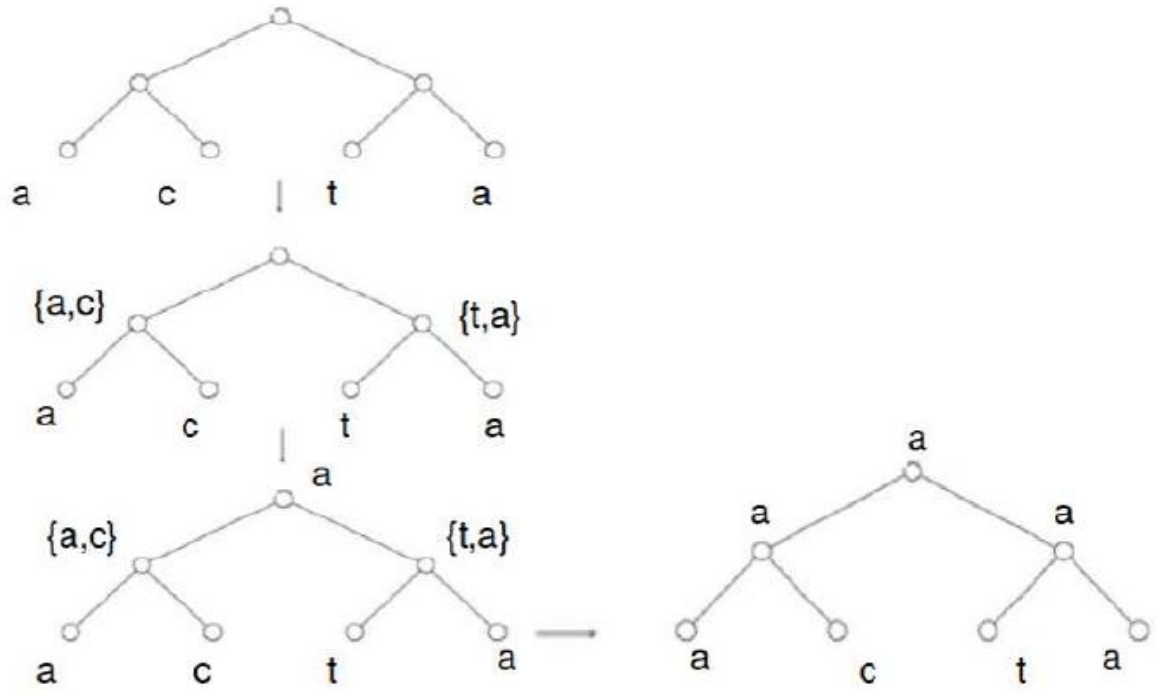
Ogni foglia viene etichettata banalmente con la lettera corrispondente, per cui le altre possibili sono settate ad ∞. Salendo di un livello vado a spiegare il sottoramo sinistro: quello che si fa è calcolare le somme delle distanze tra i 2 figli ed ogni possibile altra lettera. Per fare un esempio pratico, La A dalla A dista 0, mentre la C dalla A dista 9. Per cui andiamo a scrivere 9 nella casella del nodo padre di A e C.
Stessa cosa facciamo per la T : La A dalla T dista 3, mentre la C dalla T dista 4. Di conseguenza, La T del nodo padre avrà valore 7. E così via fin quando

.....

Una volta finito si riprende dalla root e in modo Top Down si scende scegliendo sempre il **cammino minimo**.

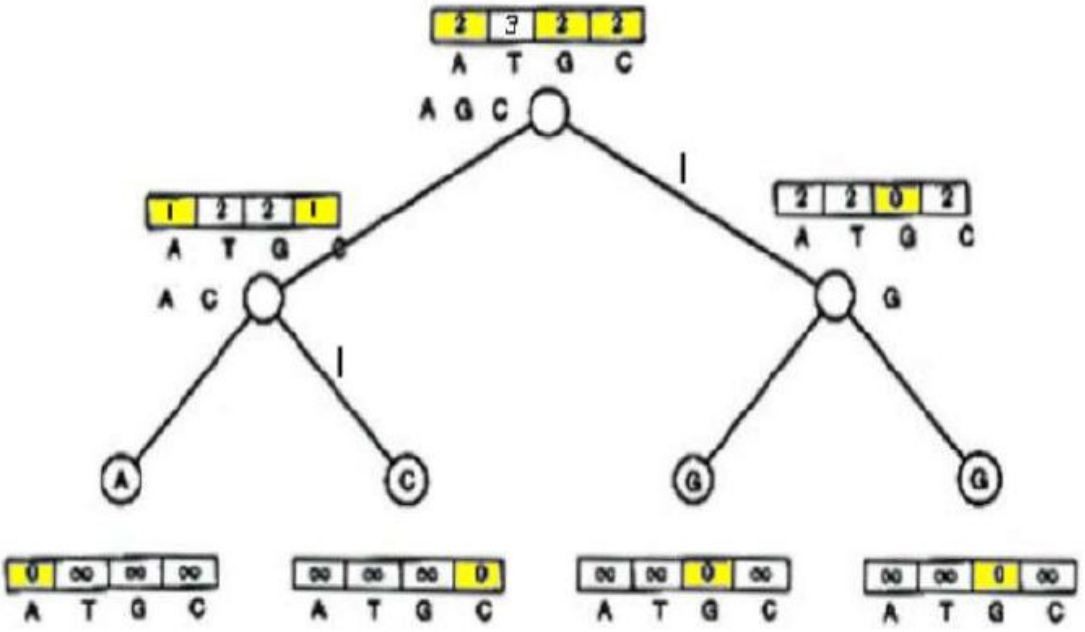
Algoritmo di Fitch:

Il sentimento dietro a questo algoritmo è semplice: all'inizio abbiamo solo le **foglie**. In seguito il **padre** di due figli non è altro che **l'unione** dei due. Da quel momento in poi i successivi padri vengono determinati come **intersezione** dei nodi **figli**.



6.5.3 Sankoff vs Fitch

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



Large Parsimony Problem

Il Large Parsimony Problem è una generalizzazione dello Small Parsimony Problem applicato ad n sequenze anzichè a 2. Di conseguenza avremo una matrice n x m, dato che sono n sequenze da m caratteri l'una. È abbastanza chiaro che in questo caso un approccio brute force come quelli adottati finora non è la scelta più saggia fattibile. Dobbiamo necessariamente accettare il fatto di saltare qualche confronto (in realtà parecchi, ma vabbè...) ed implementare sistemi di Branch & Bound oppure qualche euristica.
Per la precisione, Il LPP è un probema NP-Completo. Infatti il numero di possibili alberi con root con n foglie è:

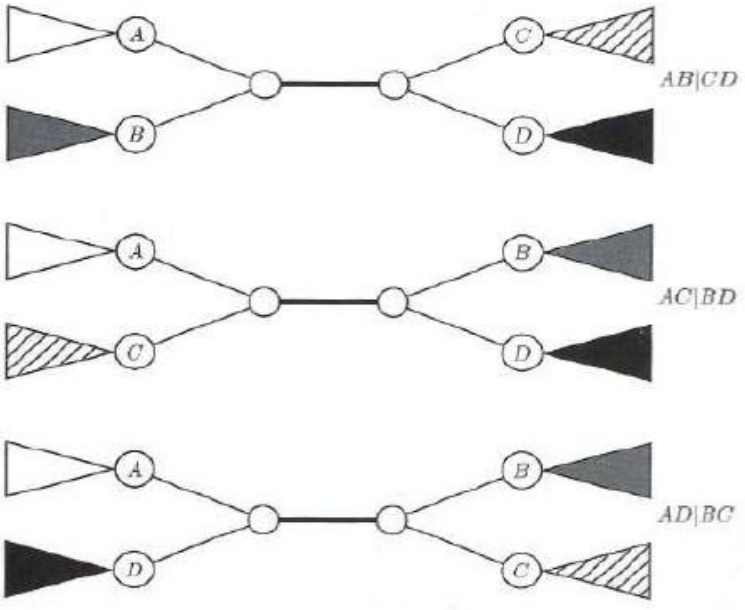
$$\frac{(2n-3)!}{2^{n-2}(n-2)!} \tag{4}$$

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

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

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

6.6.2 Nearest Neighbor Interchange



Questo algoritmo è abbastanza semplice, ma funziona bene se si accetta di non ottenere necessariamente la soluzione ottima.
Di fatto quello che facciamo è partire da un albero casuale per poi andare a fare swapping dei sottoalberi nella speranza che si riduca lo score di parsimonia.

ALGORITMI DI CLUSTERING:

Tecniche per fare il più possibile unsupervised learning su dataset non etichettato. I vari tipi sono:

1. Partizionamento
2. Gerarchici
3. Density-based
4. Basati su griglia
5. Model-based

Per quanto riguarda il partizionamento i due principali sono senz'altro l'algoritmo k-means e k-metoid. Essi di fatto partono da un database D contenente n oggetti e costruiscono k gruppi a partire da esso. La formula di costo che si minimizza è quella dell'errore quadratico.

K-means fa la media con punti anche "a metà" tra i membri del dataset, il k-metoid sceglie come centro un rappresentante di D.

Pro:

- **Veloce** (O(tkn)), con k numero di cluster, n numero di oggetti, t numero di iterazioni
- Trova sempre un ottimo locale.

Contro:

- Va in crisi se per un certo dataset non siamo in grado di definire il concetto di "**media**", per cui non si può usare per dati categorici
- Dobbiamo fornire noi k
- Sbarella male con **dati rumorosi e NO CLUSTER CONCAVI**

PAM - Partitioning around metoids

È un esempio di k-metoids:

- Scelgo a caso k **rappresentanti** e faccio i cluster.
- Per ogni elemento i != k lo confronto e cerco di capire se vale la pena fare cambio di rappresentante
- Vado avanti fin quando non avvengono più spostamenti significativi

Contro: O(n^2)

CLARA - Clustering Large Applications

È un **PAM** fatto per **partizioni** del dataset.

Contro: scala comunque male, la scelta delle partizioni iniziali può influenzare la qualità finale

CLARANS

Trasforma il **clustering** in una **ricerca su grafo** dove ogni nodo è costituito da alcuni elementi del dataset. Due nodi sono vicini se differiscono di al più un elemento.
Si usa runnare molte istanze di questo algoritmo, **uno** perché scala bene, **due** perché così è più facile ottenere buona qualità di clustering.

Algoritmi gerarchici

In realtà uno di questi lo abbiamo già visto (alberi ultrametrici ricorda qualcosa ?). Comuque qui abbiamo quelli Top Down e Bottom Up.

AGNES

Parte da **tutte foglie** ed arriva ad un **solo cluster (la root)**. Poi ripercorre al contrario e si costruisce l'albero. Spetta poi all'analista fare approssimazioni, spezzando il diagramma dove gli serve. (la soluzione se la si va a costruire un po' noi insomma...)

DIANA

Top Down, **O(n^2), no UNDO**, i raggruppamenti **successivi** dipendono dai **precedenti (perché appunto, è Top Down)**

CURE

Grandissima **porcheria** fatta per accontentare tutti (secondo me). Si scelgono **sottogruppi casuali** di r elementi e su **ognuno** si esegue **un algoritmo di clustering diverso**. Si devono **scremare** i dati e si uniscono poi i vari cluster in una logica di **gravity center**.

Algoritmi Density-based

DBSCAN

Concetto di **distanza** mediante quanti punti **giacciono** dentro un **cerchio** centrato su un **punto** in esame. Punti **density connected: stessa filosofia della proprietà transitiva a->b , b->c quindi a->c.**

Algoritmi basati su griglia

CLIQUE: idea geniale! si rappresenta tutto il dataset dentro un **sistema cartesiano** di riferimento (anche multidimensionale). Per ogni feature, Si suddivide l'asse che la rappresenta in **multipli intervallini**. A questo punto si determinano di fatto **celle multidimensionali** , per cui facciamo una sorta di battaglia navale, mettendo nello stesso clustering elementi di una singola cella.

Scale-free, anche in 1000 dimensioni, gli intervallini sono sempre quelli. Funziona bene, mi piace ! **Lineare col dataset**.

Algoritmi basati su modello

Si usano per classificare dati categorici.

ROCK

Racchiude in **sottoinsiemi** e determina cose simili in base al numero di elementi presenti dentro **l'intersezione insiemistica**.

tra i due ($A \cap B$). Si tratta quindi di un problema di ottimizzazione, ma questo comporta il dover fare molti confronti, per cui la sua complessità è elevata $O(n^2 + nm_m m_a + n^2 logn)$. Inoltre, clusters tra loro simili sono collegati da un arco (non per forza 1 a 1) dal peso pari al numero di vicini. La foto in questo caso è risolutiva:

$$\{1,2,3\}, \{1,2,4\}, \{1,2,5\}, \{1,3,4\}, \{1,3,5\}$$
$$\{1,4,5\}, \{2,3,4\}, \{2,3,5\}, \{2,4,5\}, \{3,4,5\}$$

$$\{1,2,3\} \xleftarrow{3} \xrightarrow{} \{1,2,4\}$$