Algoritmi greedy
IV parte

Progettazione di Algoritmi a.a. 2016-17

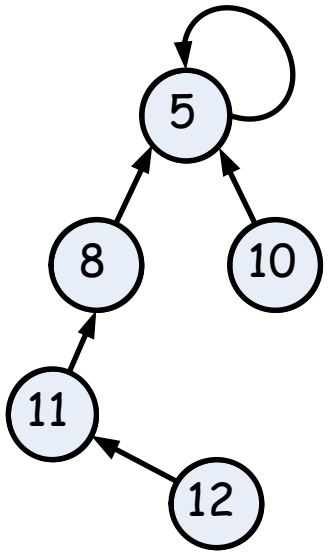
Matricole congrue a 1

Docente: Annalisa De Bonis

Implementazione basata su struttura a puntatori

- Insiemi rappresentati da strutture a puntatori
- Ogni nodo contiene un campo per l'elemento ed un campo con un puntatore ad un altro nodo dello stesso insieme.
- In ogni insieme vi è un nodo il cui campo puntatore punta a sé stesso. L'elemento in quel nodo dà nome all'insieme
- Inizialmente ogni insieme è costituito da un unico nodo il cui campo puntatore punta al nodo stesso.
-

Insieme chiamato 5

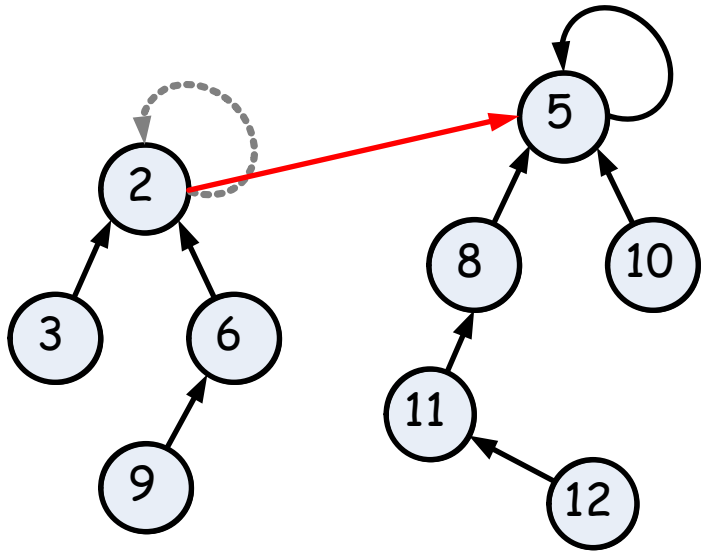


Union

- Per eseguire la union di due insiemi A e B, dove A è indicato con il nome dell' elemento x mentre B con il nome dell'elemento y, si pone nel campo puntatore del nodo contenente x un puntatore al nodo contenente y. In questo modo y diventa il nome dell'insieme unione. Si può fare anche viceversa, cioè porre nel campo puntatore del nodo contenente y un puntatore al nodo contenente x. In questo caso, il nome dell'insieme unione è x.

- Tempo: $O(1)$

- Unione dell'insieme di nome 2 con quello di nome 5. L'insieme unione viene indicato con 5.

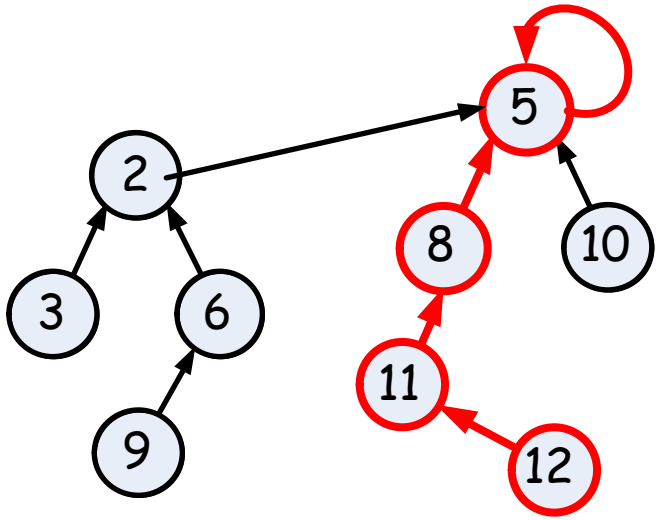


Find

- Per eseguire una find, si segue il percorso che va dal nodo che contiene l' elemento passato in input alla find fino al nodo che contiene l'elemento che dà nome all'insieme (nodo il cui campo puntatore punta a se stesso)

- Tempo: O(n) dove n è il numero di elementi nella partizione.
- Il tempo dipende dal numero di puntatori attraversati per arrivare al nodo contenente l'elemento che dà nome all'insieme.
- Il caso pessimo si ha quando la partizione è costituita da un unico insieme ed i nodi di questo insieme sono disposti uno sopra all'altro e ciascun nodo ha il campo puntatore che punta al nodo immediatamente sopra di esso

Find(12)



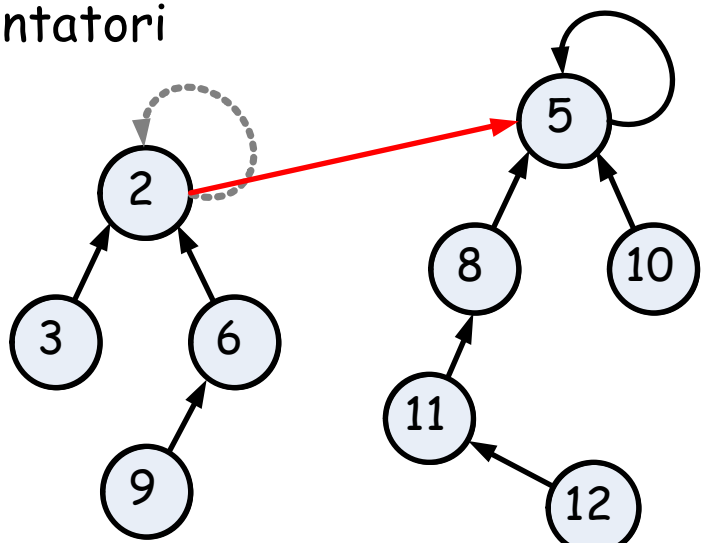
Euristica per migliorare l' efficienza

- **Union-by-size:** Diamo all'insieme unione il nome dell'insieme più grande

In questo modo find richiede tempo O(log n)

Dim.

- Contiamo il numero massimo di puntatori che possono essere attraversati durante l'esecuzione di un'operazione di find
- Osserviamo che un puntatore viene creato solo se viene effettuata una unione. Quindi attraversando il puntatore da un nodo contenente x ad uno contenente y passiamo da quello che prima era l'insieme x all'insieme unione degli insiemi x e y. Poiché usiamo la union-by-size, abbiamo che l'unione di questi due insiemi ha dimensione pari almeno al doppio della dimensione dell'insieme x
- Di conseguenza, ogni volta che attraversiamo un puntatore da un nodo ad un altro, passiamo in un insieme che contiene almeno il doppio degli elementi contenuti nell'insieme dal quale proveniamo.
- Dal momento che un insieme può contenere al più n elementi, in totale si attraversano al più O(log n) puntatori



Euristica per migliorare l' efficienza

- **Union-by-size**
  - Consideriamo la struttura dati Union-Find creata invocando MakeUnionFind su un insieme S di dimensione n. Se si usa l'implementazione della struttura dati Union-Find basata sulla struttura a puntatori che fa uso dell'euristica union-by-size allora si ha
  - Tempo Union : $O(1)$ (manteniamo per ogni nodo un ulteriore campo che tiene traccia della dimensione dell'insieme corrispondente)
  - Tempo MakeUnionFind: $O(n)$ occorre creare un nodo per ogni elemento.
  - Tempo Find: $O(\log n)$ per quanto visto nella slide precedente

Kruskal con questa implementazione di Union-Find richiede
$O(m \log m) = O(m \log n^2) = O(m \log n)$ per l'ordinamento
$O(m \log n)$ per le $O(m)$ find
$O(n)$ per le n-1 Union.

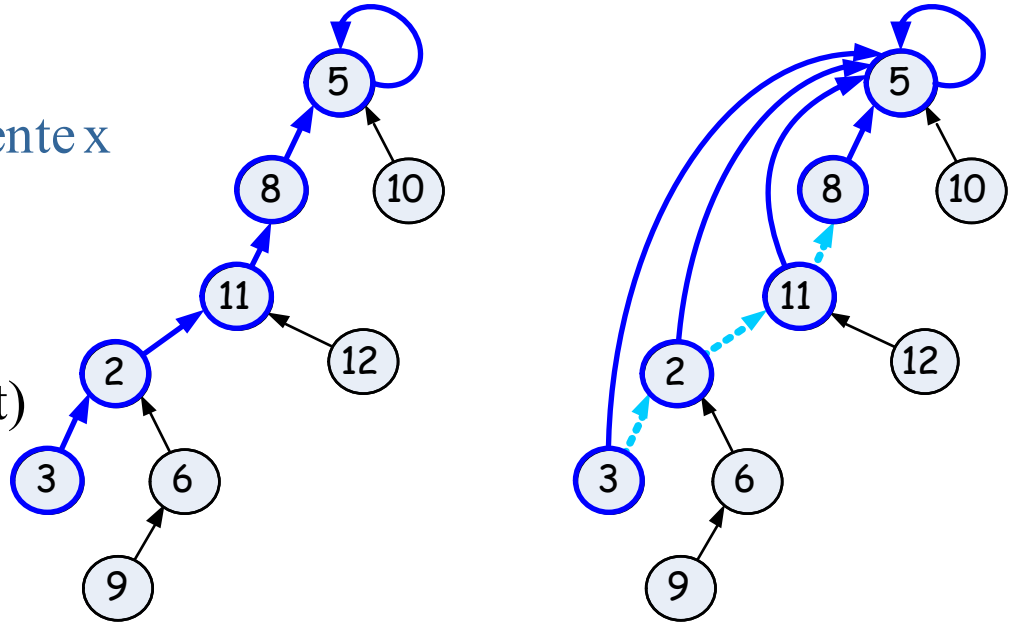
In totale $O(m \log n)$
come nel caso in cui si usa l'implementazione di Union-Find basata sull'array con uso dell'euristica union-by-size

Un' altra euristica per l' efficienza

- **Path Compression** (non ci serve per migliora il costo dell'alg. Di Kruskal)
  - Dopo aver eseguito un' operazione di Find, tutti i nodi attraversati nella ricerca avranno come il campo puntatore che punta al nodo contenente l'elemento che dà nome all'insieme
  - Intuizione: ogni volta che eseguiamo la Find con in input un elemento x di un certo insieme facciamo del lavoro in più che ci fa risparmiare sulle successive operazioni di Find effettuate su elementi incontrati durante l'esecuzione di Find(x). Questo lavoro in più non fa comunque aumentare il tempo di esecuzione asintotico della singola Find.

Indichiamo con q(x) il nodo contenente x

```
Find(x)
 if q(x) ≠q(x).pointer
  then  p← q(x).pointer
        q(x).pointer ← Find(p.element)
return q(x).pointer
```



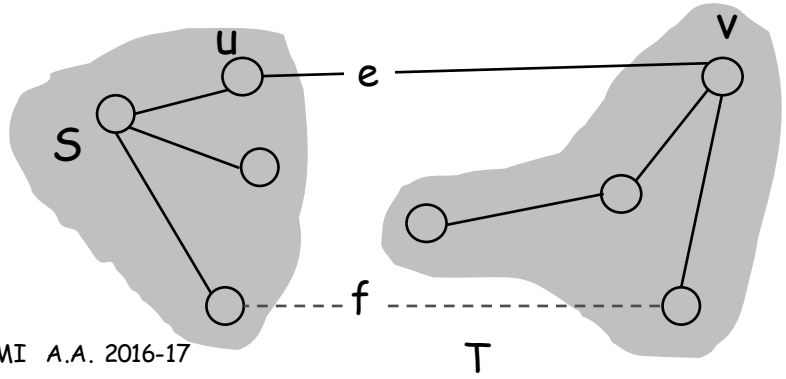
Union-by-size e path-compression

- Se si utilizza l'euristica path-compression allora una sequenza di n operazioni di find richiede tempo $O(n\,\alpha(n))$
- $\alpha(n)$ è l'inversa della funzione di Ackermann
- $\alpha(n))\leq 4$ per tutti i valori pratici di n

Proprietà del ciclo

- **Per semplicità assumiamo che tutti i costi c_e siano distinti.**
- Proprietà del ciclo. Sia C un ciclo e sia e=(u,v) l'arco di costo massimo tra quelli appartenenti a C. Ogni minimo albero ricoprente non contiene l'arco e.
- Dim. (tecnica dello scambio)
  - Sia T un albero ricoprente che contiene l'arco e. Dimostriamo che T non può essere un MST.
  - Se rimuoviamo l 'arco e da T disconnettiamo T e otteniamo un taglio [S,V-S] con u in S e v in V-S tale che nessun arco di T- {e} attraversa [S,V-S].
  - Il ciclo C contiene due percorsi per andare da u a v. Uno è costituito dall'arco e=(u,v), l'altro percorso va da u a v attraversando archi diversi da (u,v). Tra questi archi deve essercene uno che attraversa il taglio [S,V-S] altrimenti non sarebbe possibile andare da u che sta in S a v che sta in V-S. Sia f questo arco.

Se al posto dell'arco e inseriamo in T l'arco f, otteniamo un albero ricprente T' di costo
$c(T')=c(T)- c_e+c_f$
Siccome $c_f < c_e$, $c(T') < c(T)$.



Correttezza dell'algoritmo Inverti-Cancella

- L'algoritmo Inverti-Cancella produce un MST.
- Dim. (nel caso in cui i costi sono a due a due distinti)
- Sia T il grafo prodotto da Inverti-Cancella.
- Prima dimostriamo che gli archi che non sono in T non sono neanche nello MST.
  - Sia e un qualsiasi arco che non appartiene a T.
  - Se e=(u,v) non appartiene a T vuol dire che quando l'arco e=(u,v) è stato esaminato l'arco si trovava su un ciclo C (altrimenti la sua rimozione avrebbe disconnesso u e v).
  - Dal momento che gli archi vengono esaminati in ordine decrescente di costo, l'arco e=(u,v) ha costo massimo tra gli archi sul ciclo C.
  - La proprietà del ciclo implica allora che e=(u,v) non può far parte dello MST.
- Abbiamo dimostrato che ogni arco dello MST appartiene anche a T. Ora dimostriamo che T non contiene altri archi oltre a quelli dello MST.
  - Sia T^* lo MST. Ovviamente (V,T^*) è un grafo connesso.
  - Supponiamo **per assurdo** che esista un arco (u,v) di T che non sta in T^*.
  - Se agli archi di T^* aggiungiamo l'arco (u,v), si viene a creare un ciclo. Poiché T contiene tutti gli archi di T^* e contiene anche (u,v) allora T contiene un ciclo C. Ciò è impossibile perché l'algoritmo rimuove l'arco di costo più alto su C. Abbiamo quindi ottenuto una contraddizione.

Correttezza degli algoritmi quando i costi non sono distinti

- In questo caso la correttezza si dimostra perturbando di poco i costi c_e degli archi, cioè aumentando i costi degli archi in modo che valgano le seguenti tre condizioni
  - i nuovi costi $\hat{c}_e$ risultino a due a due distinti
  - se $c_e < c_{e'}$ allora $\hat{c}_e < \hat{c}_{e'}$
  - la somma dei valori aggiunti ai costi degli archi sia minore del minimo delle quantità $|c(T_1)-c(T_2)|$, dove il min è calcolato su tutte le coppie di alberi ricoprenti T_1 e T_2 tali che $c(T_1) \neq c(T_2)$ (Questo non è un algoritmo per cui non ci importa quanto tempo ci vuole a calcolare il minimo)

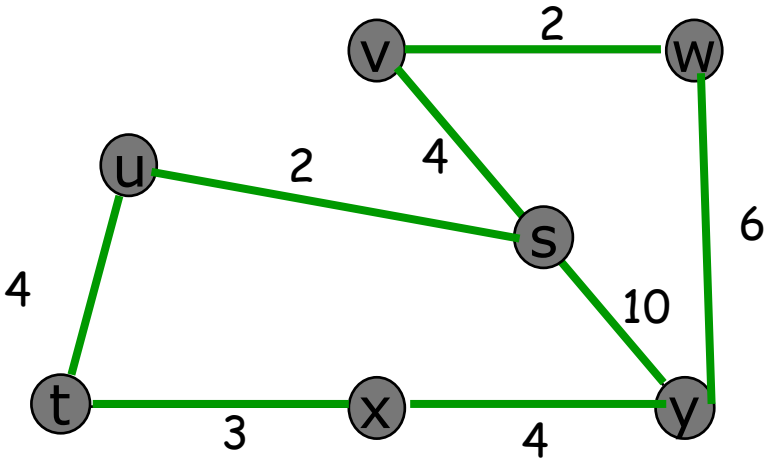
Continua nella prossima slide

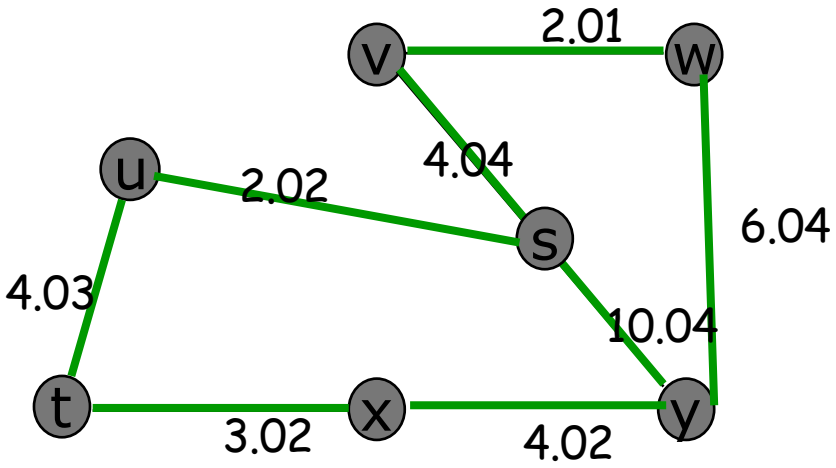
Correttezza degli algoritmi quando i costi non sono distinti

- Chiamiamo G il grafo di partenza (con i costi non pertubati) e $\hat{G}$ quello con i costi perturbati.
- Sia T un minimo albero ricoprente del grafo $\hat{G}$. Dimostriamo che T è un minimo albero ricoprente anche per G.
- Se così non fosse esisterebbe un albero T^* che in G ha costo minore di T.
- Siano c(T) e $c(T^*)$ i costi di T e T^* in G. Per come abbiamo perturbato i costi, si ha che $c(T)-c(T^*) >$ somma dei valori aggiunti ai costi degli archi
- Vediamo di quanto può essere cambiato il costo di T^* dopo aver perturbato gli archi.
- Il costo di T^* non può essere aumentato più della somma totale dei valori aggiunti ai costi degli archi
- Poiché la somma dei valori aggiunti ai costi degli archi è minore di $c(T)-c(T^*)$ allora il costo di T^* è aumentato di un valore minore di $c(T)-c(T^*)$. Di conseguenza, dopo aver perturbato i costi, la differenza tra il costo di T e quello di T è diminuita di un valore inferiore a $c(T)-c(T^*)$ per cui è ancora maggiore di 0. Ne consegue che T non può essere lo MST di $\hat{G}$ perché T^* ha costo più piccolo di T anche in $\hat{G}$

Correttezza degli algoritmi quando i costi non sono distinti

- In questo esempio i costi sono interi quindi è chiaro che i costi di due alberi ricoprenti di costo diverso differiscono almeno di 1.
- Se perturbiamo i costi come nella seconda figura, si ha che
  - I nuovi costi sono a due a due distinti
  - Se e ha costo minore di e' all'inizio allora e ha costo minore di e' anche dopo aver modificato i costi.
  - La somma dei valori aggiunti ai costi è 0.01+0.02+0.02+0.02+0.03+00.4+0.04+0.04 < 1





Correttezza degli algoritmi quando i costi non sono distinti

- Proprietà del taglio (senza alcun vincolo sui costi degli archi) Sia S un qualsiasi sottoinsieme di nodi e sia e **un** arco di costo minimo che attraversa il taglio [S,V-S]. Esiste un minimo albero ricoprente che contiene e.
- Dim.
- Siano $e_1,e_2,\ldots,e_p$ gli archi di G che attraversano il taglio ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_p)$ con $e_1=e$.
- Perturbiamo i costi degli archi di G come mostrato nelle slide precedenti e facendo in modo che $\hat{c}(e_1) < \hat{c}(e_2) < \ldots < \hat{c}(e_p)$. Per fare questo basta perturbare i costi c di G nel modo già descritto e stando attenti che se $c(e_i)=c(e_{i+1})$, per un certo $1 \leq i \leq p-1$, allora deve essere $\hat{c}(e_i) < \hat{c}(e_{i+1})$.
- Consideriamo un MST T di $\hat{G}$.
- La proprietà del taglio per grafi con costi degli archi a due a due distinti implica che lo MST di $\hat{G}$ contiene l'arco e (in quanto e è l'arco di peso minimo che attraversa [S,V-S] in $\hat{G}$) → T contiene e.
- Per quanto dimostrato nelle slide precedenti, T è anche un MST di G.
- Abbiamo quindi dimostrato che esiste un MST di G che contiene e.
- NB: MST distinti di G possono essere ottenuti permutando tra di loro archi di costo uguale nell'ordinamento $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_p)$

Correttezza degli algoritmi quando i costi non sono distinti

- Proprietà del ciclo (senza alcun vincolo sui costi degli archi) Sia C un ciclo e sia e **un** arco di costo massimo in C. Esiste un minimo albero ricoprente che non contiene e.
- Dim.
- Siano $e_1,e_2,\dots,e_p$ gli archi del ciclo C, ordinati in modo che $c(e_1) \leq c(e_2) \leq \dots \leq c(e_p)$ con $e_1=e$.
- Perturbiamo i costi degli archi di G come mostrato nelle slide precedenti e facendo in modo che $\hat{c}(e_1) < \hat{c}(e_2) < \dots < \hat{c}(e_p)$. Per fare questo basta perturbare i costi c di G nel modo già descritto e stando attenti che se $c(e_i)=c(e_{i+1})$, per un certo $1 \leq i \leq p-1$, allora deve essere $\hat{c}(e_i) < \hat{c}(e_{i+1})$.
- Consideriamo un MST T di $\hat{G}$.
- La proprietà del ciclo per grafi con costi degli archi a due a due distinti implica che lo MST di $\hat{G}$ non contiene l'arco e (in quanto e è l'arco di peso massimo nel ciclo C in $\hat{G}$)→ T NON deve contenere e.
- Per quanto dimostrato nelle slide precedenti T è anche un MST di G.
- Abbiamo quindi dimostrato che esiste un MST di G che non contiene e.
- NB: MST distinti di G possono essere ottenuti permutando tra di loro archi di costo uguale nell'ordinamento $c(e_1) \leq c(e_2) \leq \dots \leq c(e_p)$.

Correttezza degli algoritmi quando i costi non sono distinti

- Si è visto che la proprietà del taglio può essere estesa al caso in cui i costi degli archi **non** sono a due a due distinti
- Possiamo quindi dimostrare la correttezza degli algoritmi di Kruskal e di Prim nello stesso modo in cui abbiamo dimostrato la correttezza di questi algoritmi nel caso in cui gli archi hanno costi a due a due distinti.

- Si è visto che la proprietà del ciclo può essere estesa al caso in cui i costi degli archi **non** sono a due a due distinti
- Possiamo quindi dimostrare la correttezza dell'algoritmo Inverti-Cancella nello stesso modo in cui abbiamo dimostrato la correttezza dell'algoritmo nel caso in cui gli archi hanno costi a due a due distinti.

Clustering

- Clustering. Dato un insieme U di n oggetti $p_1, \ldots, p_n$, vogliamo classificarli in gruppi coerenti
- Esempi: foto, documenti, microorganismi.

- Funzione distanza. Associa ad ogni coppia di oggetti un valore numerico che indica la vicinanza dei due oggetti
  - Questa funzione dipende dai criteri in base ai quali stabiliamo che due oggetti sono simili o appartengono ad una stessa categoria.
  - Esempio: numero di anni dal momento in cui due specie hanno cominciato ad evolversi in modo diverso

Problema. Dividere i punti in cluster (gruppi) in modo che punti in cluster distinti siano distanti tra di loro.

- Classificazione di documenti per la ricerca sul Web.
- Ricerca di somiglianze nei database di immagini mediche
- Classificazione di oggetti celesti in stelle, quasar, galassie.

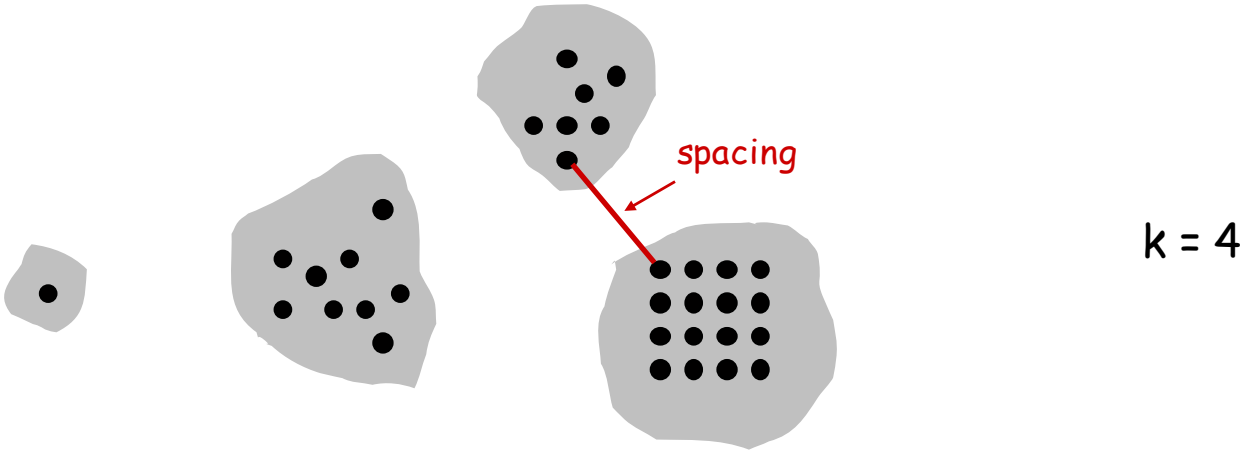
Clustering con Massimo Spacing

- k-clustering. Dividi gli oggetti in k gruppi non vuoti.

- Funzione distanza. Soddisfa le seguenti proprietà
  - $d(p_i, p_j) = 0$ se e solo se $p_i = p_j$
  - $d(p_i, p_j) \geq 0$
  - $d(p_i, p_j) = d(p_j, p_i)$

- Spacing. Distanza più piccola tra due oggetti in cluster differenti

- Problema del clustering con massimo spacing. Dato un intero k, trovare un k-clustering con massimo spacing.



Algoritmo greedy per il clustering

- Algoritmo basato sul single-link k-clustering.
  - Forma un grafo sull'insieme di vertici U.
  - Inizialmente ogni vertice u è in un cluster che contiene solo u
  - Ad ogni passo trova i due oggetti x e y più vicini e tali che x e y sono in cluster distinti. Aggiunge un arco tra x e y.
  - Va avanti fino a che ha aggiunto n-k archi: a quel punto ci sono esattamente k clusters.

- Osservazione. Questa procedura corrisponde ad eseguire l'algoritmo di Kruskal su un grafo completo in cui i costi degli archi rappresentano la distanza tra due oggetti. L'unica differenza è che l'algoritmo si ferma prima di inserire i k-1 archi più costosi dello MST.

- NB: Corrisponde a cancellare i k-1 archi più costosi da un MST

Algoritmo greedy per il clustering: Analisi

- Teorema. Sia C^* il clustering $C^*_1, \ldots, C^*_k$ ottenuto cancellando i k-1 archi più costosi da un MST T. C^* è un k-clustering con massimo spacing.
- Dim. Sia C un clustering $C_1, \ldots, C_k$ diverso da C^*
  - Sia d* lo spacing di C^*. La distanza d* corrrisponde al costo del (k-1)-esimo arco più costoso dello MST T.
  - Facciamo vedere che lo spacing tra due cluster di C non è maggiore di d*
  - Siano p_i e p_j due oggetti che si trovano nello stesso cluster in C^* e in cluster differenti in C. Chiamiamo rispettivamente C^*_r il suddetto cluster di C^* e C_s e C_t i suddetti due cluster di C.
  - Sia P il percorso tra p_i e p_j che passa esclusivamente per nodi di C^*_r e sia q il primo vertice di P che non appartiene a C_s e sia p il suo predecessore lungo P. In altre parole p è in C_s e q è in C_t
  - Tutti gli archi sul percorso P e quindi anche (p,q) hanno costo≤ d* in quanto sono stati scelti da Kruskal nei primi n-k passi.
  - Lo spacing di C è minore o uguale del costo dell'arco (p,q) che per quanto detto è ≤ d*