Algoritmi greedy
VI parte

Progettazione di Algoritmi a.a. 2021-22
Matricole congrue a 1
Docente: Annalisa De Bonis

155

Implementazione dell'algoritmo di Kruskal

- Abbiamo bisogno di rappresentare le componenti connesse (alberi della foresta)
- Ciascuna componente connessa è un insieme di vertici disgiunto da ogni altro insieme.

```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
   T ← ϕ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m                    are u and v in different connected components?
      (u,v) = ei                     ↙
      if (u and v are in different sets) {
         T ← T ∪ {ei}
         merge the sets containing u and v
      }
   return T
}                                    ↖
                                     merge two components
```

156

Implementazione dell'algoritmo di Kruskal

- Ciascun albero della foresta è rappresentato dal suo insieme di vertici
- Per rappresentare questi insiemi di vertici, si utilizza la struttura dati Union-Find per la rappresentazione di insiemi disgiunti
- Operazioni supportate dalla struttura dati Union-Find
- MakeUnionFind(S): crea una collezione di insiemi ognuno dei quali contiene un elemento di S
  - Nella fase di inizializzazione dell'algoritmo di Kruskal viene invocato MakeUnionFind(V): ciascun insieme creato corrisponde ad un albero con un solo vertice.
- Find(x): restituisce l'insieme che contiene x
  - Per ciascun arco esaminato (u,v), l'algoritmo di Kruskal invoca find(u) e find(v). Se entrambe le chiamate restituiscono lo stesso insieme allora vuol dire che u e v sono nello stesso albero e quindi (u,v) crea un ciclio in T.
- Union(X,Y): unisce gli insiemi X e Y
  - Se l'arco (u,v) non crea un ciclo in T allora l'algoritmo di Kruskal invoca Union(Find(u),Find(v)) per unire le componenti connesse di u e v in un'unica componente connessa

157

Implementazione dell'algoritmo di Kruskal con Union-Find

```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
   T ← ϕ

   MakeUnionFind(V)   //create n singletons for the n vertices

   for i = 1 to m
      (u,v) = ei
      if (Find(u)≠Find(v)) {
         T ← T ∪ {ei}
         Union(Find(u),Find(v))
      }
   return T
}
```

Implementazione di Union-Find con array

- La struttura dati Union-Find può essere implementata in vari modi
- Implementazione di Union-Find con array
  - Gli elementi sono etichettati con interi consecutivi da 1 ad n e ad ogni elemento è associata una cella dell'array S che contiene il nome del suo insieme di appartenenza.
  - Find(x): $O(1)$. Basta accedere alla cella di indice x dell'array S
  - Union: $O(n)$. Occorre aggiornare le celle associate agli elementi dei due insiemi uniti.
  - MakeUnionFind $O(n)$: Occorre inizializzare tutte le celle.

Analisi dell'algoritmo di Kruskal in questo caso:

Inizializzazione $O(n)+O(m \log m)=O(m \log n^2)=O(m \log n)$.

- $O(n)$ per creare la struttura Union-Find e $O(m \log m)$ per ordinare gli archi

Per ogni arco esaminato: $O(1)$ per le 2 find.

- In totale, 2m find → $O(m)$

Per ogni arco aggiunto a T : $O(n)$ per la union

- In totale n-1 union (perché?) → $O(n^2)$

Algoritmo: $O(m \log n)+ O(n^2)$

Implementazione di Union-Find con array e union-by-size

- Implementazione di Union-Find con array ed union-by-size
  - Stessa implementazione della slide precedente ma si usa anche un array A per mantenere traccia della cardinalità di ciascun insieme.
  - La Find(x) è identica a prima
  - MakeUnionFind $O(n)$: occorre creare n insiemi ciascuno contenente un unico elemento e inizializzare tutte le celle S e tutte le celle di A.
    - Inizialmente le celle di A sono tutte uguali ad 1.
  - Union: si spostano gli elementi dell'insieme piu` piccolo in quello piu` grande. Ciascun elemento x dell'insieme piu` piccolo viene aggiunto all'insieme piu` grande e si aggiorna le cella di indice x di S con il nome del nuovo insieme. La cella dell'array A corrispondente all'insieme più piccolo viene posta a 0 mentre quella corrispondente all'insieme più grande viene posta uguale alla somma delle cardinalità dei due insiemi uniti.

Implementazione di Union-Find con array e union-by-size

- Nell'implementazione con union-by-size la singola operazione di unione richiede ancora O(n) nel caso pessimo perché i due insiemi potrebbero avere entrambi dimensione pari ad n diviso per una costante.
- Vediamo però cosa accade quando consideriamo una sequenza di unioni.
- Inizialmente tutti gli insiemi hanno dimensione 1.

Continua nella prossima slide

Implementazione di Union-Find con array e union-by-size

Affermazione. Una qualsiasi sequenza di unioni richiede al più tempo O(n log n).

Dim. Il tempo di esecuzione di una sequenza di Union dipende dal numero di aggiornamenti che vengono effettuati nell'array S.

- Calcoliamo quanto lavoro viene fatto per un qualsiasi elemento x.
- Questo lavoro dipende dal numero di volte in cui viene aggiornata la cella S[x] e cioè dal numero di volte in cui x viene spostato da un insieme ad un altro per effetto di una Union.
- Ogni volta che facciamo un'unione che coinvolge x , x cambia insieme di appartenenza solo se proviene dall'insieme che ha dimensione minore o uguale dell'altro. Ciò vuol dire che l'insieme risultante dall'unione ha dimensione pari almeno al doppio dell'insieme da cui proviene x.
- Dopo un certo numero k di unioni che richiedono lo spostamento di x, l'elemento x si troverà in un insieme di taglia almeno 2^k→ x viene spostato al più log(n) volte in quanto, al termine della sequenza di unioni, x si troverà in un insieme B di cardinalità al più n.
- Quindi per ogni elemento viene fatto un lavoro che richiede tempo O(log n). Siccome ci sono n elementi, in totale il tempo richiesto dalla sequenza di Union è O(n log n).

Implementazione di Union-Find con array e union-by-size

Analisi algoritmo di Kruskal in questo caso:

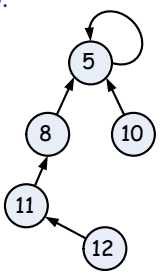
- Inizializzazione $O(n)+O(m \log m)=O(m \log m)$.
  - $O(n)$ creare la struttura Union-Find e
  - $O(m \log m)$ ordinare gli archi
- In totale il numero di find è 2m che in totale richiedono $O(m)$
- Si effettuano esattamente n-1 union (perché?). Queste n-1 union, per il risultato sopra dimostrato, richiedono $O(n \log n)$

Algoritmo: $O(m \log m+n \log n)=O(m \log m)=O(m \log n^2) = O(m \log n)$

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.

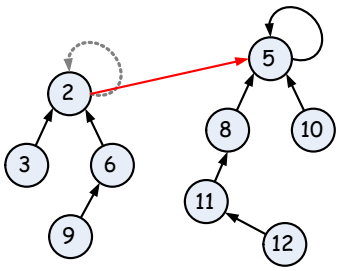
Insieme chiamato 5



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.

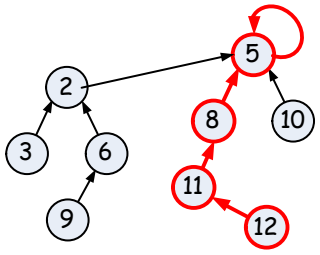


165

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)



166

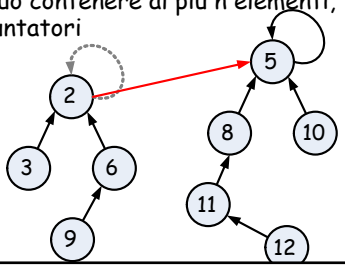
Euristica per migliorare l' efficienza

- **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



167

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

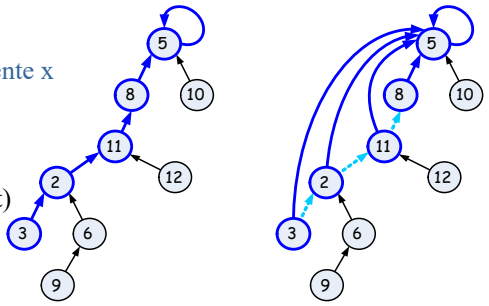
168

Un' altra euristica per l' efficienza

- **Path Compression** (non ci serve per migliorare il costo dell'algoritmo di Kruskal)
  - Dopo aver eseguito un'operazione di Find, tutti i nodi attraversati nella ricerca avranno 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
```



169

Union-by-size e path-compression

- Se si utilizza le euristiche union-by-size e path-compression allora una sequenza di n operazioni union-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

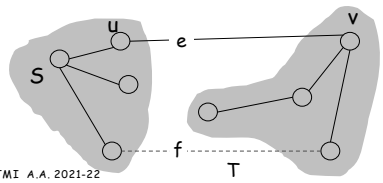
170

Proprietà del ciclo

- **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 in due alberi uno contenente u e l'altro contenente v. Chiamamo S l'insieme dei nodi dell'albero che contiene u.
  - Il ciclo C contiene due percorsi per andare da u a v. Un percorso è costituito dall'arco e=(u,v) mentre l'altro va da u a v attraverso gli archi di C 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 di e inseriamo in T l'arco f, otteniamo un albero ricoprente T' di costo $c(T')=c(T)- c_e+c_f$
    Siccome $c_f < c_e$ allora $c(T') < c(T)$. Ne consegue che T non è uno MST.



171

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 e quindi elimina i cicli. Abbiamo quindi ottenuto una contraddizione.

172

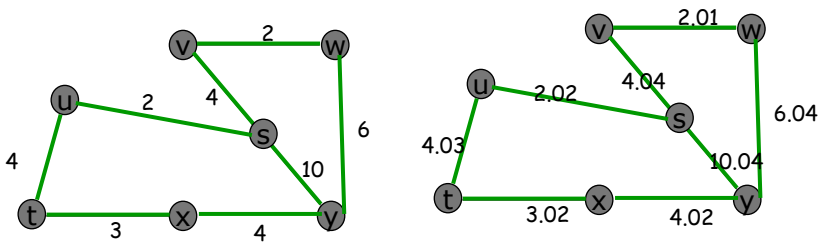
172

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
  1. i nuovi costi $\hat{c}_e$ risultino a due a due distinti
  2. se $c_e < c_{e'}$ allora $\hat{c}_e < \hat{c}_{e'}$
  3. 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)

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+0.04+0.04+0.04 < 1



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 **per assurdo** non fosse così esisterebbe un albero T^* che in G ha costo minore di T → $c(T)-c(T^*)>0$, dove $c(T)$ e $c(T^*)$ sono i costi di T e T^* in G.
- Sia s la somma totale dei valori aggiunti ai costi degli archi di G

✸ Per come abbiamo perturbato i costi, si ha che $c(T)-c(T^*) > s$
  - in quanto s è minore della differenza in valore assoluto tra i costi di due qualsiasi alberi ricoprenti di G con costo diverso.

- Se mostrassimo che il costo $\hat{c}(T^*)$ di T^* in $\hat{G}$ è minore del costo $\hat{c}(T)$ di T in $\hat{G}$ allora si otterrebbe una contraddizione al fatto che T è un MST per $\hat{G}$.

175

Correttezza degli algoritmi quando i costi non sono distinti

- Vediamo di quanto può essere cambiato il costo di T^* dopo aver perturbato gli archi. Stimiamo quindi $\hat{c}(T^*) - c(T^*)$
- Osserviamo che il costo di T^* è aumentato di un valore minore di s (perché?) → $\hat{c}(T^*) - c(T^*) < s$

1. $\hat{c}(T^*) - c(T^*) < s \rightarrow \hat{c}(T^*) < s + c(T^*)$

- La 1. implica

2. $\hat{c}(T) - \hat{c}(T^*) > \hat{c}(T) - c(T^*) - s > (c(T) - c(T^*)) - s$

per cui la differenza tra il costo di T e quello di T^* è diminuita di un valore minore di s

Per la ✸ si ha $c(T)-c(T^*) > s$ e quindi

3. $(c(T)-c(T^*))-s > 0$

La 2 e la 3 implicano $\hat{c}(T) - \hat{c}(T^*) > 0$ per cui T non può essere lo MST di $\hat{G}$ perché T^* ha costo più piccolo di T anche in $\hat{G}$.

176

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 $e_1,e_2,\dots,e_p$ gli archi di G che attraversano il taglio 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 $\hat{c}(e_i) < \hat{c}(e_{i+1})$.
- Sia T lo MST 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 → 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 potrebbero essere ottenuti permutando tra di loro archi di costo uguale nell'ordinamento $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$

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 $e_p=e$.
- 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 $\hat{c}(e_i) < \hat{c}(e_{i+1})$.
- Sia T un MST 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 → 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 potrebbero essere ottenuti permutando tra di loro archi di costo uguale nell'ordinamento $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$.

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

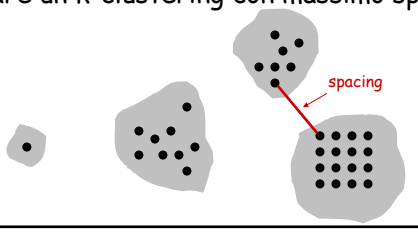
- Clustering. Dato un insieme U di n oggetti $p_1, \ldots, p_n$, vogliamo classificarli in gruppi coerenti
- Esempi: foto, documenti, microorganismi.

- 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. Partizione dell'insieme U in k sottoinsiemi non vuoti (cluster).
- 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.



k = 4

181

181

Algoritmo greedy per il clustering

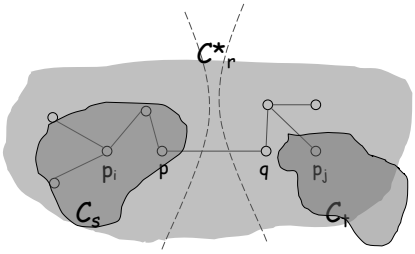
- Algoritmo basato sul single-link k-clustering.
  - Costruisce un grafo sull'insieme di vertici U in modo che alla fine abbia k componenti connesse. Ogni componente connessa corrisponderà ad un cluster.
  - Inizialmente il grafo non contiene archi per cui ogni vertice u è in un cluster che contiene 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 cluster.
- 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 (costo dell'arco (u,v) = d(u,v)). 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

182

182

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 del grafo completo in cui ogni arco e=(u,v) ha costo $c_e = d(u,v)$. 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^* corrisponde al costo del (k-1)-esimo arco più costoso dello MST T (il meno costoso tra quelli cancellati dallo MST T)
  - Facciamo vedere che lo spacing di C non è maggiore di d^*
  - Siccome C e C^* sono diversi allora devono esistere due oggetti p_i e p_j che si trovano nello stesso cluster in C^* e in cluster differenti in C. Chiamiamo rispettivamente C^*_r il cluster di C^* che contiene p_i e p_j e C_s e C_t i due cluster di C contenenti p_i e p_j , rispettivamente.

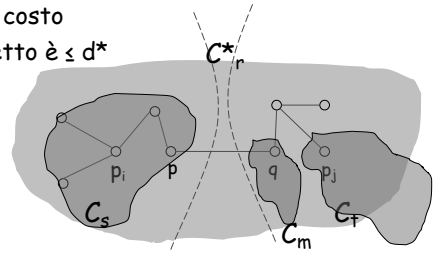


183

183

Algoritmo greedy per il clustering: Analisi

- Sia P il percorso tra p_i e p_j che passa esclusivamente per nodi di C^*_r (cioe` attraverso archi selezionati da Kruskal nei primi n-k passi) e sia q il primo vertice di P che non appartiene a C_s. Indichiamo con C_m la componente in cui si trova q.
- Sia p il predecessore di q lungo P. Il nodo p è in C_s in quanto q è il primo nodo incontrato lungo il percorso che non sta in C_s
- Tutti gli archi sul percorso P e quindi anche (p,q) hanno costo $\leq d^*$ in quanto sono stati scelti da Kruskal nei primi n-k passi.
- Lo spacing di C è minore o uguale della distanza tra i punti piu` vicini di C_s e C_m e quindi è anche minore del costo dell'arco (p,q) che per quanto detto è $\leq d^*$



184

184