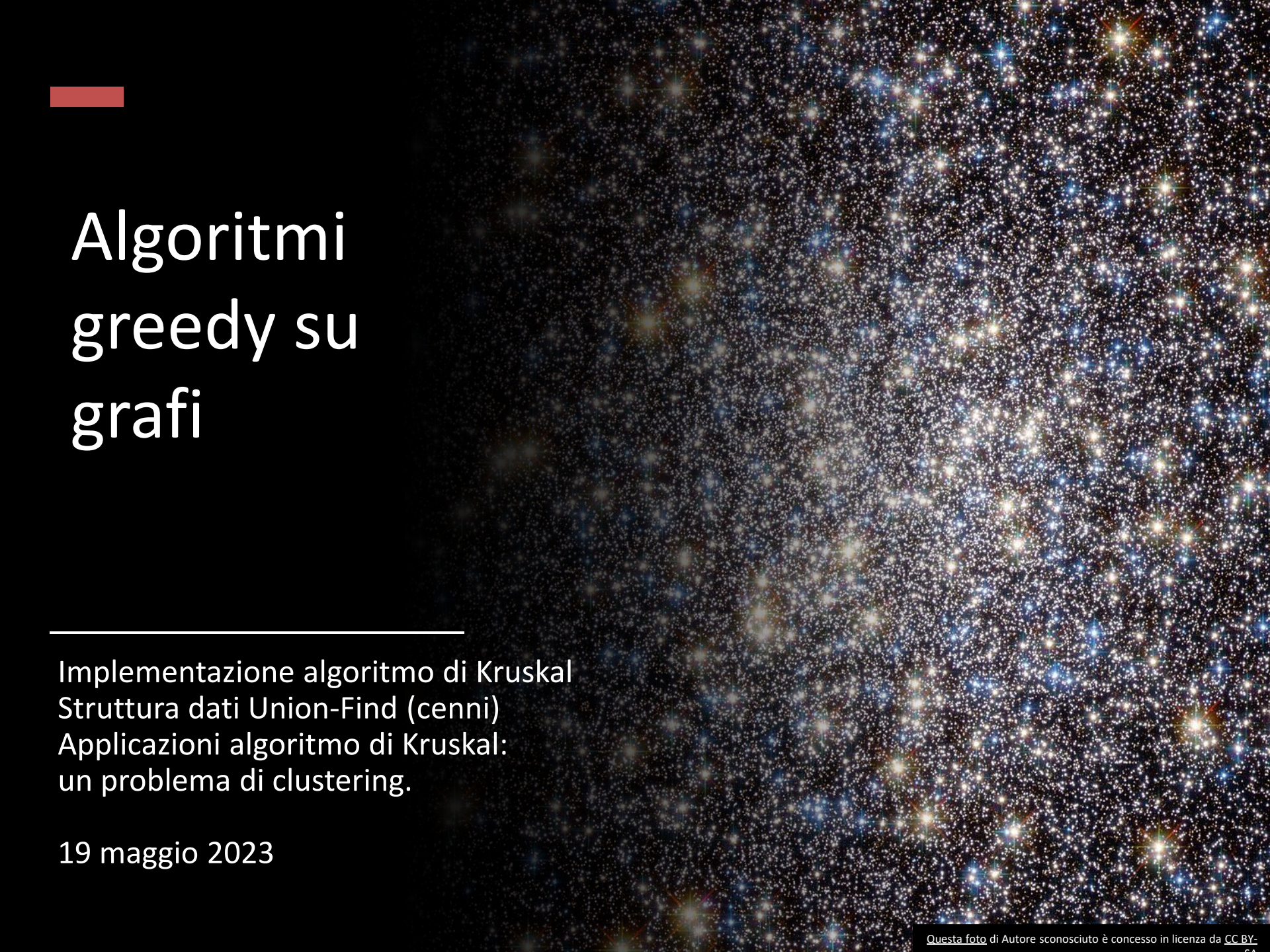
Algoritmi greedy su grafi

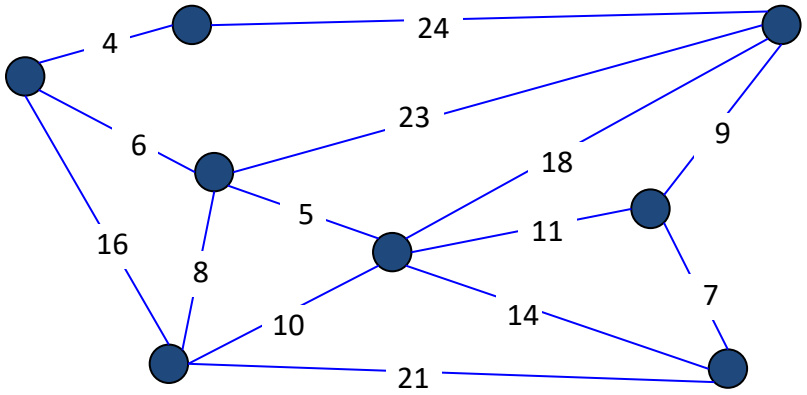
Implementazione algoritmo di Kruskal
Struttura dati Union-Find (cenni)
Applicazioni algoritmo di Kruskal:
un problema di clustering.

19 maggio 2023

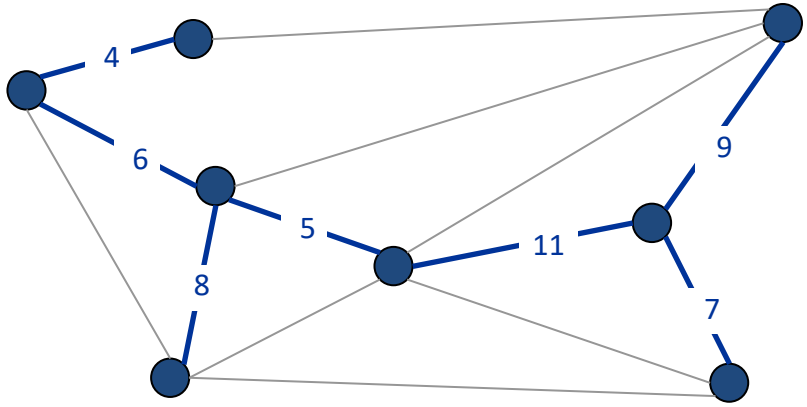


Minimum Spanning Tree (MST)

Minimum spanning tree. Given a connected graph G = (V, E) with real-valued edge weights c_e, an **MST** is a subset of the edges $T \subseteq E$ such that (V,T) is a tree (connected and acyclic), denoted spanning tree, whose sum of edge weights is minimized.



G = (V, E)



T, $\Sigma_{e \in T} c_e = 50$

Recall: a tree with n nodes has n-1 edges.

Cayley's Theorem. There are n^{n-2} spanning trees of K_n: can't solve by brute force!

Greedy Algorithms

All three algorithms produce an MST!!!

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

Kruskal's algorithm. Start with T = ϕ. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Reverse-Delete algorithm. Start with T = E. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

Prim's Algorithm: Implementation

```
Prim(G,c,s) {
foreach (v∈V) {a[v]← ∞; π[v]= NULL}; a[s]←0;
Initialize an empty priority queue Q
foreach (v ∈ V) insert v onto Q
Initialize set of explored nodes S ← ϕ

 while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
          if ((v ∉ S) and (c_e < a[v]))
           {decrease priority a[v] to c_e; π[v]=u;}
}
```

Priority Queue

PQ Operation	Prim	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	1	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
Decr		1	log n	$\log_d n$	1
I		1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

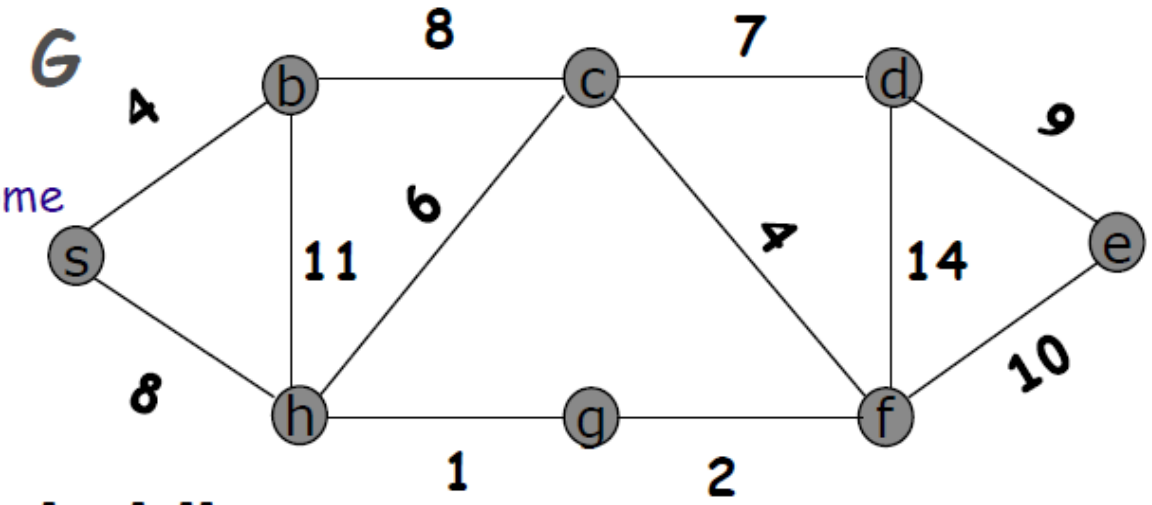
Quale delle due è preferibile?

† Individual ops are amortized bounds

Un esempio

In questo esempio, per ciascun nodo u manteniamo anche un campo $\pi[u]$ che alla fine è uguale al padre di u nello MST

s = nodo scelto come radice



[u,a[u]]

$Q = \{[s, 0],[b, \infty],[c, \infty],[d, \infty],[e, \infty],[f, \infty],[g, \infty],[h, \infty]\}$ $S = \{\ \}$

Si estrae s da Q e si aggiornano i campi a e π dei nodi adiacenti ad s

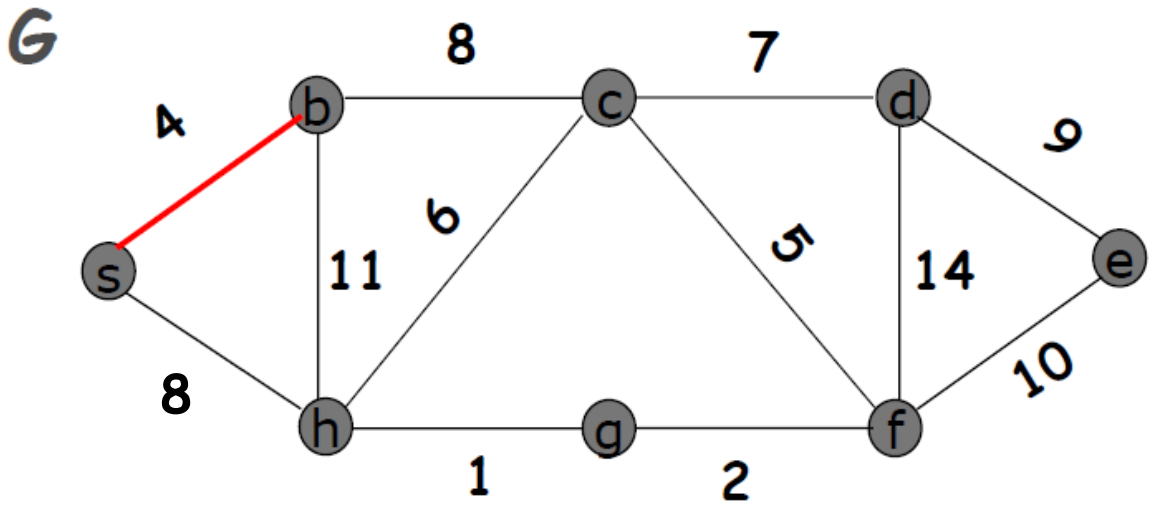
$S = \{s\}$

$Q = \{[b, 4],[c, \infty],[d, \infty],[e, \infty],[f, \infty],[g, \infty],[h, 8]\}$

$\pi(b)=s$

$\pi(h)=s$

Un esempio



$Q = \{[b,4],[c,\infty],[d,\infty],[e,\infty],[f,\infty],[g,\infty],[h,8]\}$ $S = \{s\}$

- Si estrae b da Q.

$S = \{s,b\}$

- Si aggiornano i campi a dei nodi adiacenti a b che si trovano in Q

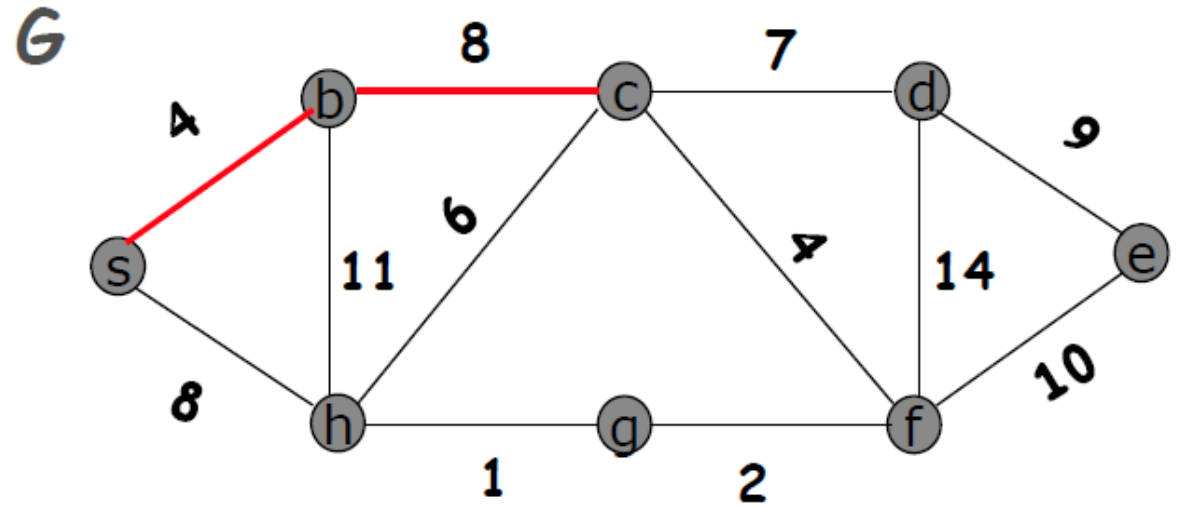
$\pi(b)=s$

$\pi(h)=s$

$Q = \{[c,8],[d,\infty],[e,\infty],[f,\infty],[g,\infty],[h,8]\}$

$\pi(c)=b$

Un esempio



$Q = \{[c, 8],[d, \infty],[e, \infty],[f, \infty],[g, \infty],[h, 8]\}$ $S = \{s,b\}$

- A seconda dell'implementazione di Q si estrae c oppure h da Q.

$S = \{s,b,c\}$

- Assumiamo che venga estratto c: si aggiornano i campi a e π dei nodi adiacenti a c che si trovano in Q

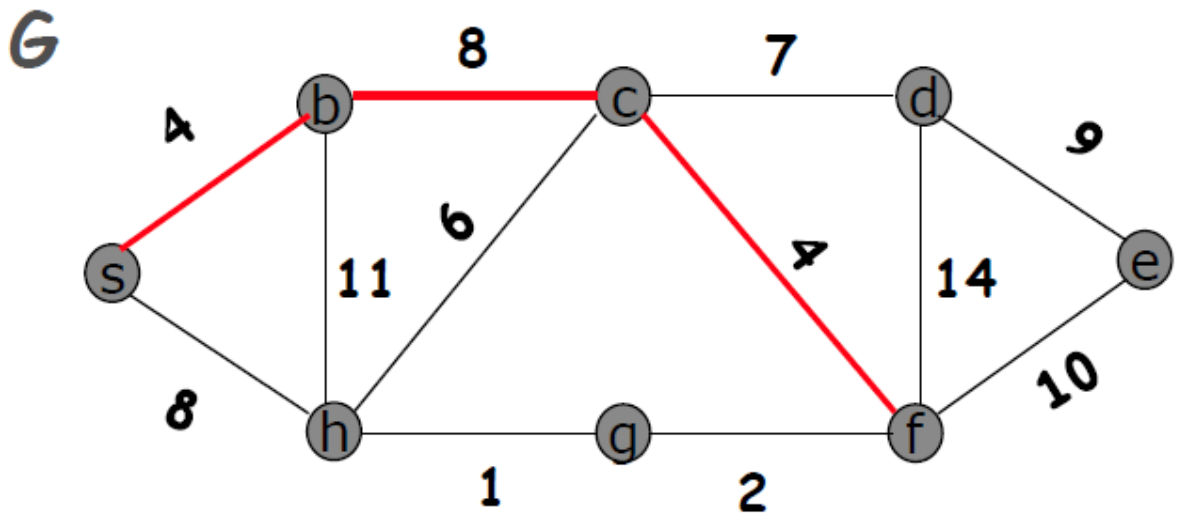
$Q = \{[d, 7],[e, \infty],[f, 4],[g, \infty],[h, 6]\}$

$\pi(b)=s, \pi(c)=b$

$\pi(h)=c, \pi(d)=c$

$\pi(f)=c$

Un esempio



$Q = \{[d, 7],[e, \infty],[f, 4],[g, \infty],[h, 6]\}$ $S = \{s,b,c\}$

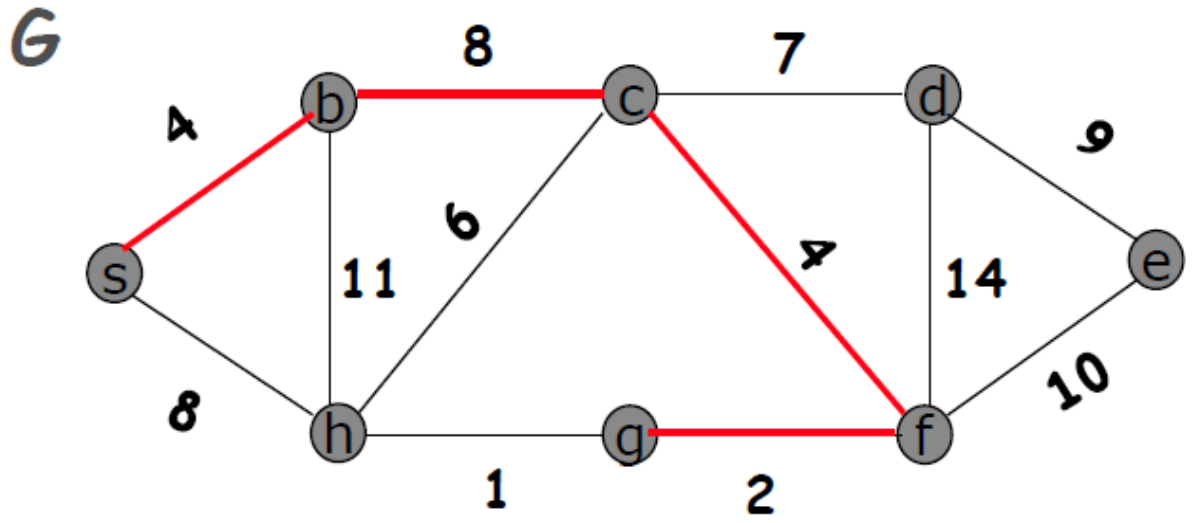
Si estrae f da Q e si aggiornano i campi a e π dei nodi adiacenti a f che si trovano in Q

$S = \{s,b,c,f\}$

$Q = \{[d, 7],[e, 10],[g, 2],[h, 6]\}$

$\pi(b)=s, \pi(c)=b, \pi(f)=c,$
$\pi(h)=c, \pi(d)=c, \pi(g)=f,$
$\pi(e)=f$

Un esempio



Q = {[d, 7],[e, 10],[g, 2],[h, 6]}

S = {s,b,c,f}

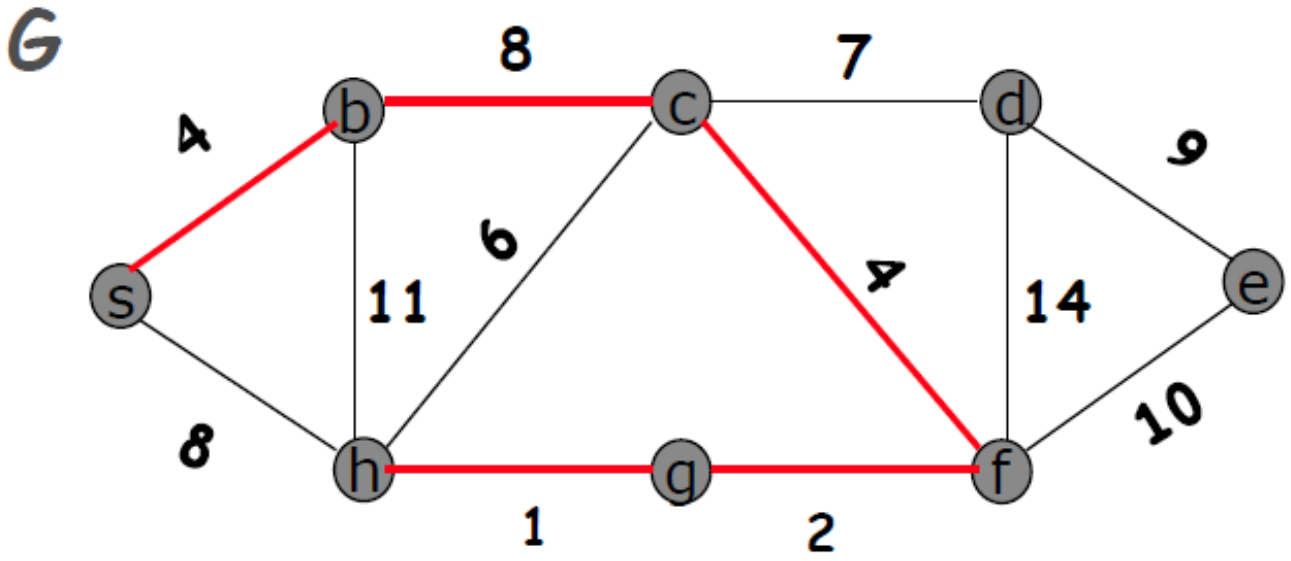
Si estrae g da Q e si aggiornano i campi a e π dei nodi adiacenti a g che si trovano in Q

S = {s,b,c,f, g}

Q = {[d, 7],[e,10],[h, 1]}

π(b)=s,π(c)=b,π(f)=c,π(g)=f,
π(d)=c, π(e)=f, π(h)=g

Un esempio



$Q = \{[d, 7],[e, 10],[h, 1]\}$

$S = \{s,b,c,f,g\}$

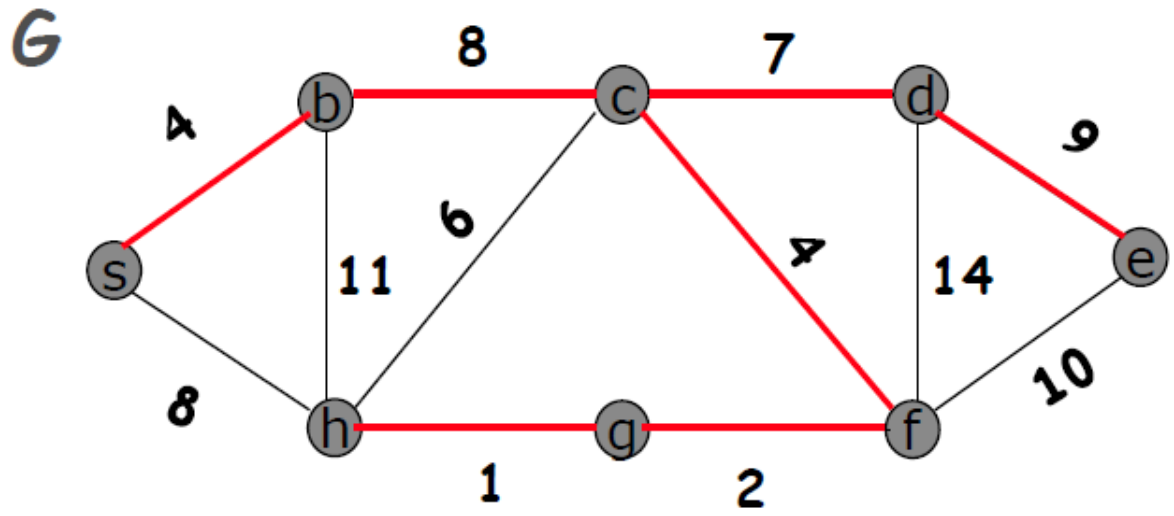
Si estrae h da Q e si aggiornano i campi a e π dei nodi adiacenti a h che si trovano in Q

$S = \{s,b,c,f,g,h\}$

$Q = \{[d, 7],[e, 10]\}$

$\pi(b)=s, \pi(c)=b, \pi(f)=c, \pi(g)=f, \pi(h)=g,$
$\pi(d)=c, \pi(e)=f$

Un esempio



Q = {[d, 7],[e, 10]}

S = {s,b,c,f,g, h}

Si estrae d da Q e si aggiorna il campo a e π di e

Q = {[e, 9]}

S = {s,b,c,f,g,h,d}

Si estrae e da Q

π(b)=s,π(c)=b,π(f)=c,π(g)=f,π(h)=g,
π(d)=c, π(e)=d

Implementazione dell'algoritmo di Kruskal

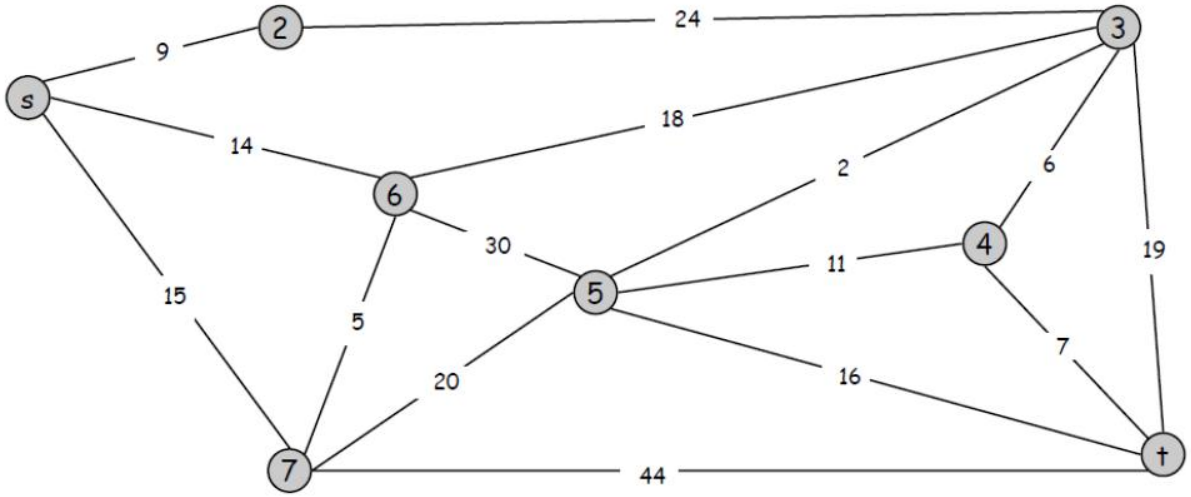
- Abbiamo bisogno di rappresentare le componenti connesse (alberi della foresta)
- Ciascuna componente connessa è un insieme di vertici disgiunto da ogni altro insieme.

```
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
```

Esempio



L'algoritmo continuerebbe considerando **tutti** i rimanenti archi, senza aggiungere altro a T

{s}, {2}, {3}, {4}, {5}, {6}, {7}, {t}	T=∅
{s}, {2}, {3,5}, {4}, {6}, {7}, {t}	T={(3,5)}
{s}, {2}, {3,5}, {4}, {6,7}, {t}	T={(3,5), (6,7)}
{s}, {2}, {3,5,4}, {6,7}, {t}	T={(3,5), (6,7), (3,4)}
{s}, {2}, {3,5,4,t}, {6,7}	T={(3,5), (6,7), (3,4), (4,t)}
{s,2}, {3,5,4,t}, {6,7}	T={(3,5), (6,7), (3,4), (4,t) (s,2)}
per l'arco (4,5) Find(4)=Find(5)	
{s,2, 6,7}, {3,5,4,t}	T={(3,5), (6,7), (3,4), (4,t), (s,2), (s,6)}
per l'arco (s,7) Find(s)=Find(7)	
{s,2, 6,7,3,5,4,t}	T={(3,5), (6,7), (3,4), (4,t), (s,2), (s,6), (3,6)}

Struttura dati Union-Find

Utilizziamo una struttura dati per mantenere una collezione di insiemi disgiunti che supporta le seguenti operazioni:

MakeUnionFind(S): crea una collezione di insiemi ognuno dei quali contiene un elemento distinto di S

Find(x): restituisce l'insieme a cui appartiene l'elemento x (un suo rappresentante)

Union(A,B): unisce l'insieme A con l'insieme B

Implementazione dell'algoritmo di Kruskal con Union-Find

```
Kruskal(G, c) {
   Sort edges weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
   T ← φ

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

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

O(m log m)

MakeUnionFind (V)

2m Find

n-1 Union

Complessità dell'algoritmo di Kruskal

La struttura dati Union-Find può essere implementata in vari modi con varie complessità.

Con qualsiasi implementazione il tempo richiesto da MakeUnionFind, le 2m Find e le (n-1) Union sarà sempre minore del tempo necessario per l'ordinamento iniziale, per cui il tempo di esecuzione dell'algoritmo di Kruskal sarà in ogni caso O(m log n).

Più precisamente:

$O(m \log n) + O(m) + O(n \log n)$ per impl. con liste+euristica

$O(m \log n) + O(m\, \alpha(m)) + O(n)$ per impl. con puntatori+ euristica

dove α (m) è l'inversa della funzione di Ackermann e α (m) <4 per tutti gli usi pratici

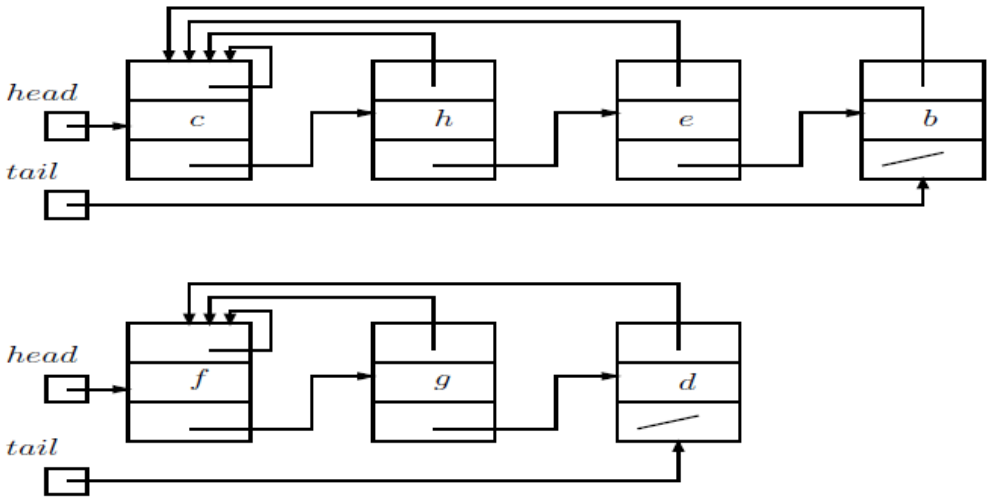
Accenniamo alle due implementazioni…..

Implementazione con liste

Prima (semplice) rappresentazione: liste linkate.

- Ogni insieme S_i della collezione $\mathcal{S}$ é rappresentato mediante una lista linkata;
- L'elemento nella testa della lista é il *rappresentante* dell'insieme corrispondente;
- Ciascun oggetto nella lista linkata contiene un elemento dell'insieme, un puntatore all'oggetto contenente il successivo elemento dell'insieme, ed un puntatore al rappresentante dell'insieme;
- Ciascuna lista mantiene un puntatore *head* alla testa della lista (rappresentante dell'insieme) ed un puntatore *tail* all'ultimo elemento della lista.

Esempio: $S_1 = \{b, c, e, h\}$, $S_2 = \{d, f, g\}$



Operazioni

MAKEUNIONFIND(S)
For ogni $x \in S$
`crea una lista con l'unico elemento` x
tail `e` *head* `puntano entrambi ad` x

Complessità: Θ(n)

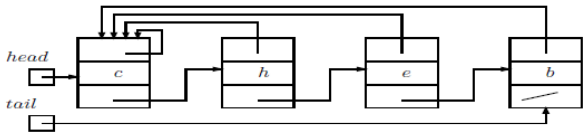
FIND(x)
`RETURN`(puntatore di x al rappresentante)

Complessitá: Θ(1)

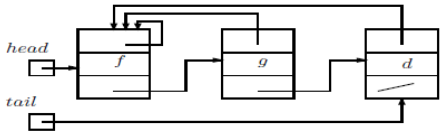
UNION(A,B)
Appendi la lista A alla fine della lista B.
Aggiorna *head*, *tail* e i puntatori della lista A a *head*

Complessità: O(n)

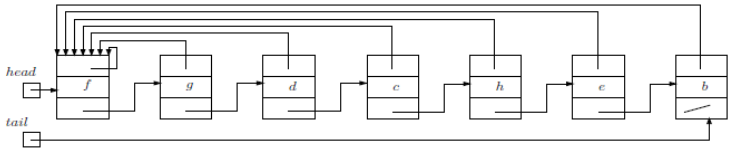
A



B



Supponiamo ora di eseguire UNION(A,B)



Nota: la complessità è proporzionale alla lunghezza della lista che viene appesa

Implementazione con liste e euristica

Euristica per unione pesata: manteniamo per ogni lista un contatore del numero di suoi elementi e appendiamo sempre la lista più corta.

- Ogni singola operazione ha lo stesso caso peggiore, ma…..
  (si può dimostrare che) …… una sequenza di k Union richiede O(k log k).

Analisi ammortizzata

Quindi l'algoritmo di Kruskal richiede:

$O(m \log m) + O(n) + O(m) + O(n \log n) = O(m \log n)$

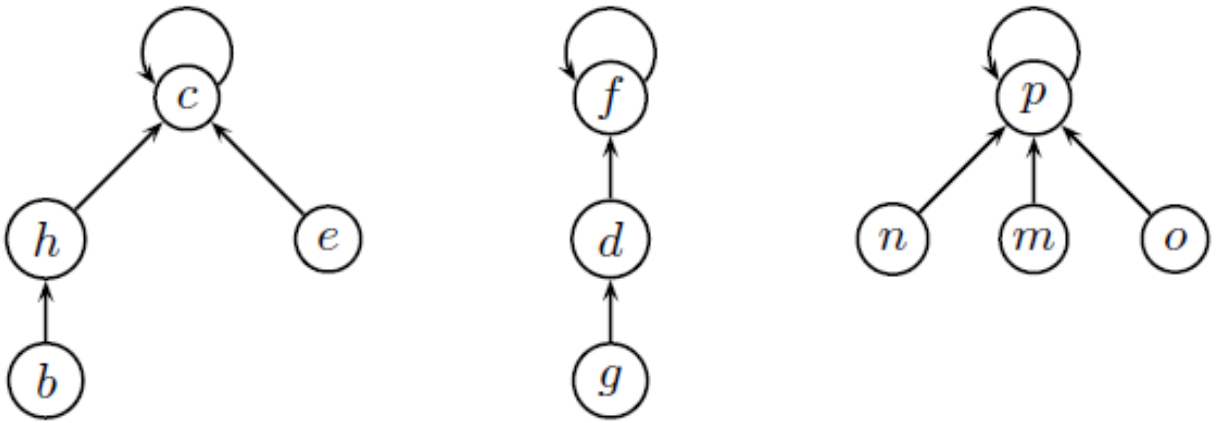
Implementazione con puntatori/alberi

E possibile ottenere una implementazione piú veloce delle operazioni su insiemi disgiunti usando una diversa rappresentazione, in cui:

- Ogni insieme $S \in \mathcal{S}$ é rappresentato da un albero;
- Ogni elemento di S é un nodo dell'albero, con un unico puntatore al suo padre;
- La radice dell'albero rappresentante l'insieme S contiene il rappresentante dell'insieme S.

Esempio di rappresentazione di insiemi mediante alberi

$\mathcal{S} = \{S_1, S_2, S_3\}$, $S_1 = \{b, c, e, h\}$, $S_2 = \{d, f, g\}$, $S_3 = \{n, m, o, p\}$



Operazioni

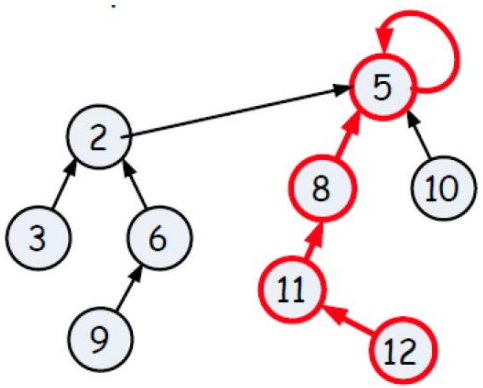
MAKEUNIONFIND(S)
for ogni $x \in S$
crea un albero con il solo nodo x che punta a sé stesso

Complessità: Θ(n)

FIND(x)
segui i puntatori al padre da x fino alla radice dell'albero, restituisci il puntatore alla radice dell'albero

Complessità: O(n)

Esempio: FIND(12)

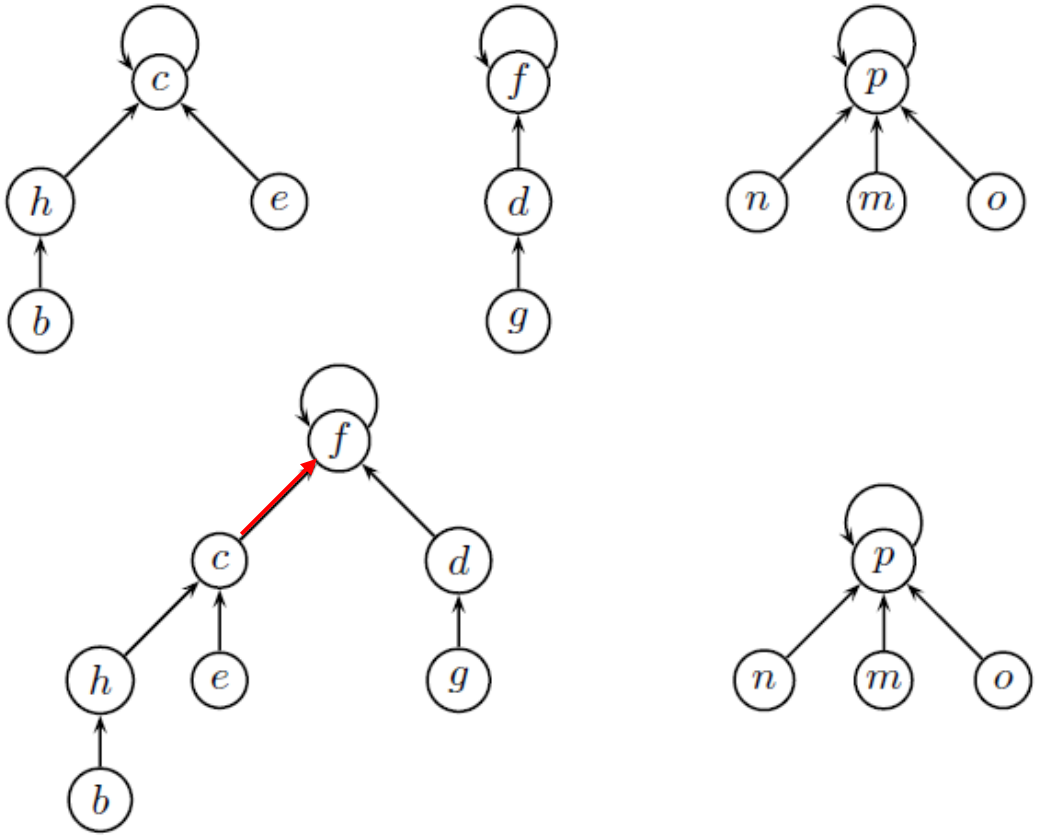


Union con alberi

UNION(A,B)
`Poni il puntatore padre di A a B`

Complessità: O(1)

Esempio: $S_1 = \{b, c, e, h\}$, $S_2 = \{d, f, g\}$, $S_3 = \{n, m, o, p\}$, e UNION(S_1,S_2)



Implementazione con euristiche

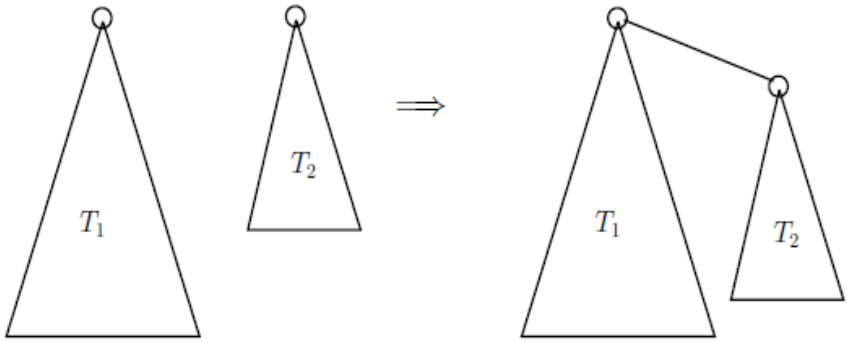
Purtroppo le complessità non sono migliori dell'implementazione con liste.
Lo saranno adottando delle euristiche.

Prima euristica: unione per rango.

Nella euristica dell'unione per rango, procediamo in maniera simile all'euristica dell'unione pesata usata nella rappresentazione di insiemi disgiunti mediante liste.

Piú precisamente, manteniamo un valore (chiamato **rango**) associato ad ogni nodo dell' albero nella struttura. Tale valore corrisponde ad una limitazione superiore all'altezza del nodo nell'albero.

Nella euristica dell'unione per rango, ogni qualvolta si dovrá effettuare una operazione di UNION, la radice dell'albero con il *rango minore* punterá alla radice con il *rango maggiore*.



Implementazione con euristiche

Seconda euristica: compressione dei cammini.

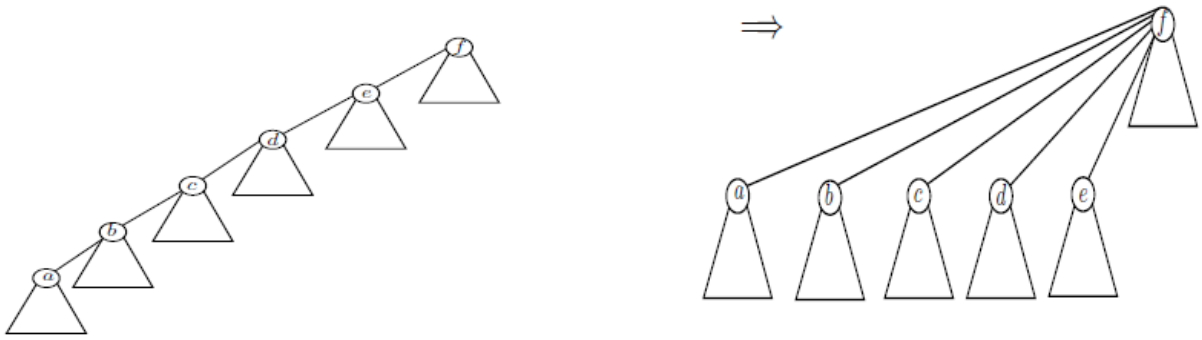
Ricordiamo la implementazione della operazione di FIND-SET:

FIND(x)
`segui i puntatori al padre da` x `fino alla radice dell'albero, restituisci il puntatore alla radice dell'albero`

Siano $x_1 = x, x_2, \ldots, x_k = r$ i nodi dell'albero sul percorso da x fino alla radice r dell'albero, che vengono visitati durante l'esecuzione di FIND-SET(x).

Nella euristica della compressione dei cammini, *tutti* i nodi $x_1 = x, \ldots, x_{k-1}$ avranno il loro puntatore al padre assegnato ad r (ovvero, tutti i nodi $x_1 = x, \ldots, x_{k-1}$ diventeranno figli della radice)

Esempio di compressione di cammini durante una operazione di FIND(a)



Implementazione basata su puntatori e euristica

Con l'euristica «union-by-size e path-compression»

- MakeUnionFind(V) richiede $O(n)$
- Union(A,B) richiede $O(1)$
- Find(x) richiede $O(\log n)$

ma (si può dimostrare che) una sequenza di k Find richiede $O(k\ \alpha(k))$, dove α (k) è l'inversa della funzione di Ackermann e α (k) < 4 per tutti gli usi pratici

Quindi l'algoritmo di Kruskal richiede:

$O(m \log m) + O(n) + O(m\ \alpha(m)) + O(n) = O(m \log n)$

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)

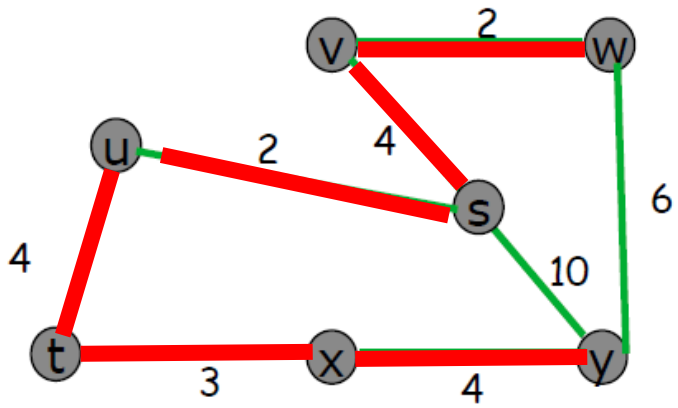
Sul libro a p. 149

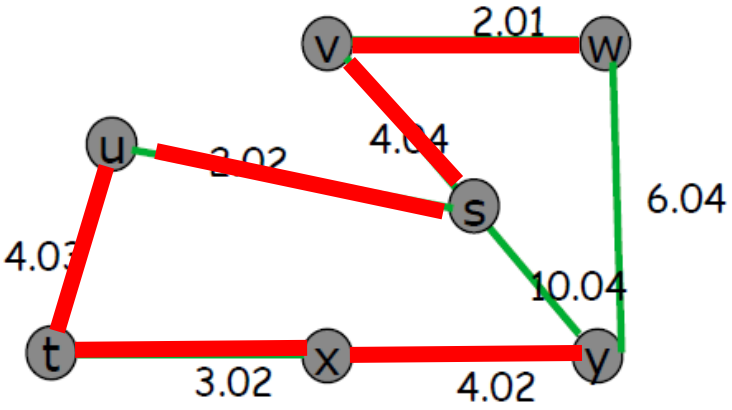
Continua nella prossima slide

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

0,22 = 0.01+0.02+0.02+0.02+0.03+00.4+0.04+0.04 < 1





Costo di **T** in questo grafo = 19
Può esistere un albero di ricoprimento di costo ≤ 18?

Costo MST **T**= 19,14

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 $\hat{G}$.

An application: Clustering

Clustering. Given a set U of n objects labeled p_1, …, p_n, classify into coherent groups.

↑ photos, documents, micro-organisms

Distance function. Numeric value specifying "closeness" of two objects.

↑ number of corresponding pixels whose intensities differ by some threshold

Fundamental problem. Divide into clusters so that points in different clusters are far apart.

Routing in mobile ad hoc networks.

Identify patterns in gene expression.

Document categorization for web search.

Similarity searching in medical image databases

Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

Clustering of Maximum Spacing

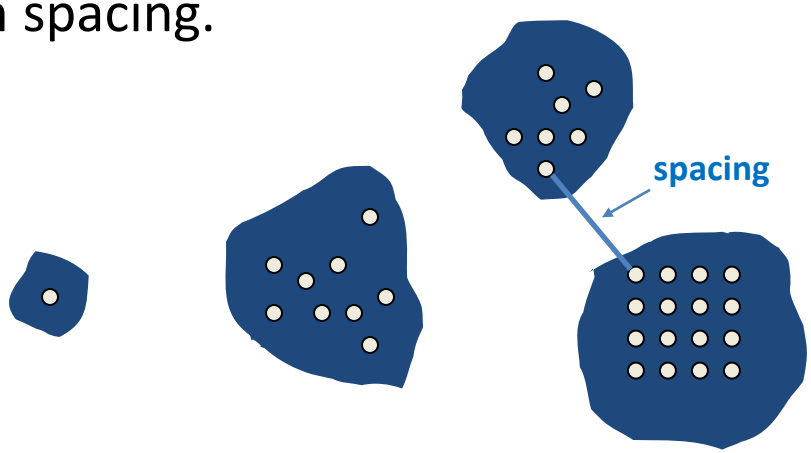
k-clustering. Divide objects into k non-empty groups.

Distance function. Assume it satisfies several natural properties.

$d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
$d(p_i, p_j) \geq 0$ (nonnegativity)
$d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

Spacing. Min distance between any pair of points in different clusters.

Clustering of maximum spacing. Given an integer k, find a k-clustering of maximum spacing.



k = 4

Greedy Clustering Algorithm

Single-link k-clustering algorithm.

Form a graph on the vertex set U, corresponding to n clusters.

Find the closest pair of objects such that each object is in a different cluster and add an edge between them.

Repeat n-k times until there are exactly k clusters.

Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

Remark. Equivalent to finding an MST and deleting the k-1 most expensive edges.

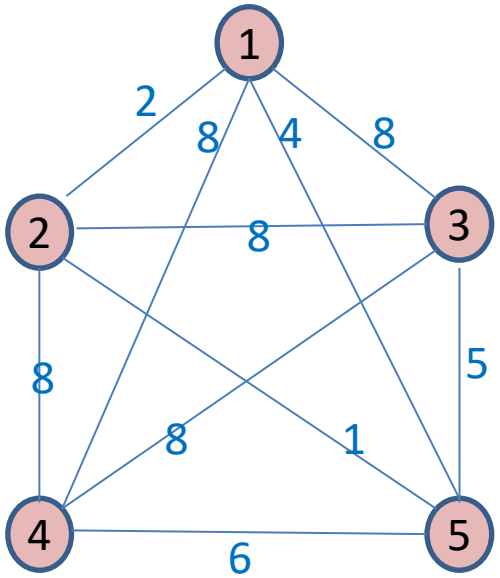
Esempio dell'algoritmo di clustering

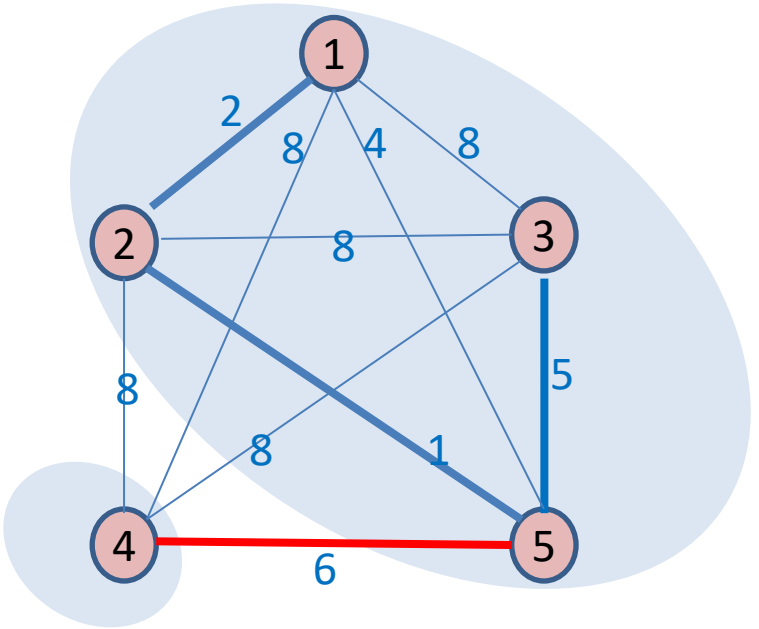
U = {1,2,3,4,5}; d(i,j) come indicata sull'arco (i,j); k=2.

Costruisco il grafo come in figura a sinistra.

Eseguo l'algoritmo di Kruskal. Seleziono in ordine:

(2,5), (1,2), (3,5), (4,5). Cancello gli ultimi k-1=1 archi inseriti (ovvero (4,5)). Ottengo il 2-clustering: C1={1,2,3,5}, C2={4} il cui spacing è d*=6





Nota: d*= costo del (k-1)-esimo arco più costoso del MST.

Greedy Clustering Algorithm: correctness

Theorem. Let C* denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the k-1 most expensive edges of a MST. C* is a k-clustering of max spacing.

Pf. Let C denote some other clustering $C_1, \ldots, C_k$.

The spacing of C* is the length d* of the $(k-1)^{st}$ most expensive edge.

Let p_i, p_j be in the same cluster in C*, say C^*_r, but different clusters in C, say C_s and C_t.

Some edge (p, q) on p_i-p_j path in C^*_r spans two different clusters in C.

All edges on p_i-p_j path have length $\leq$ d* since Kruskal chose them.

Spacing of C is $\leq$ d* since p and q are in different clusters. ▪