

Algoritmi greedy IV parte

Progettazione di Algoritmi a.a. 2020-21
Matricole congrue a 1
Docente: Annalisa De Bonis

107

107

Minimo albero ricoprente (Minimum spanning tree)

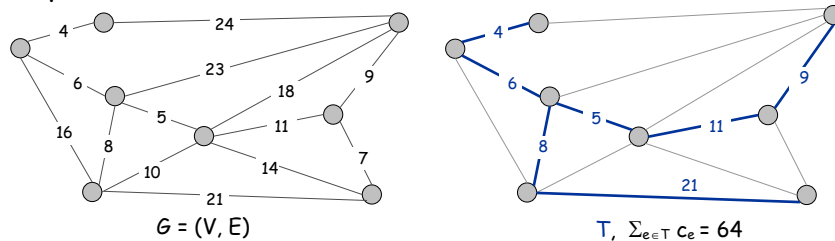
- Supponiamo di voler creare una rete che interconnetta un insieme di posizioni $V = \{v_1, v_2, \dots, v_n\}$ in modo che per ogni coppia di posizioni esista un percorso che li collega.
- Alcune coppie di posizioni possono essere collegate direttamente.
- Stabilire un collegamento diretto tra una coppia di posizioni ha un costo che dipende da vari parametri.
- L'obiettivo è di utilizzare esattamente $n-1$ di questi collegamenti diretti tra coppie di posizioni in modo da connettere l'intera rete e da minimizzare la somma dei costi degli $n-1$ collegamenti stabiliti .
- Esempi di applicazione alla progettazione di una rete sono.
 - Reti telefoniche, idriche, televisive, stradali, di computer

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

108

Minimo albero ricoprente (Minimum spanning tree o in breve MST)

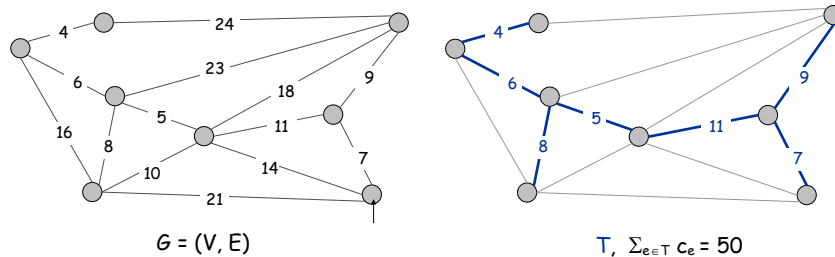
- Grafo non direzionato connesso $G = (V, E)$.
- Per ogni arco e , $c_e =$ costo dell'arco e (c_e numero reale).
- **Def. Albero ricoprente (spanning tree).** Sia dato un grafo non direzionato connesso $G = (V, E)$. Uno spanning tree di G è un sottoinsieme di archi $T \subseteq E$ tale che $|T|=n-1$ e gli archi in T non formano cicli (in altre parole T forma un albero che ha come nodi tutti i nodi di G).
- **Def.** Sia dato un grafo non direzionato connesso $G = (V, E)$ tale che ad ogni arco e di G è associato un costo c_e . Per ogni albero ricoprente T di G , definiamo il **costo di T** come $\sum_{e \in T} c_e$.



PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

Minimo albero ricoprente (Minimum spanning tree o in breve MST)

- **Input:**
 - Grafo non direzionato connesso $G = (V, E)$.
 - Per ogni arco e , $c_e =$ costo dell'arco e .
- **Minimo albero ricoprente.** Sia dato un grafo non direzionato connesso $G = (V, E)$ con costi c_e degli archi a valori reali. Un minimo albero ricoprente è un sottoinsieme di archi $T \subseteq E$ tale che T è un albero ricoprente di costo minimo.



PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

Minimo albero ricoprente (Minimum spanning tree o in breve MST)

- Il problema di trovare un minimo albero ricoprente non può essere risolto con un algoritmo di forza bruta
- **Teorema di Cayley.** Ci sono n^{n-2} alberi ricoprenti del grafo completo K_n .

111

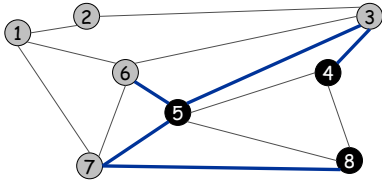
Algoritmi greedy per MST

- **Kruskal.** Comincia con $T = \emptyset$. Considera gli archi in ordine non decrescente di costo. Inserisce un arco e in T se e solo il suo inserimento non determina la creazione di un ciclo in T
- **Inverti-Cancella.** Comincia con $T = E$. Considera gli archi in ordine non crescente dei costi. Cancella e da T se e solo se la sua cancellazione non rende T disconnesso.
- **Prim.** Comincia con un certo nodo s e costruisce un albero T avente s come radice. Ad ogni passo aggiunge a T l'arco di peso più basso tra quelli che hanno esattamente una delle due estremità in T (se un arco avesse entrambe le estremità in T , la sua introduzione in T creerebbe un ciclo)

112

Taglio

- **Taglio.** Un taglio è una partizione $[S, V-S]$ dell'insieme dei vertici del grafo.
- **Insieme di archi che attraversano il taglio $[S, V-S]$.**
Sottoinsieme D di archi che hanno un'estremità in S e una in $V-S$.



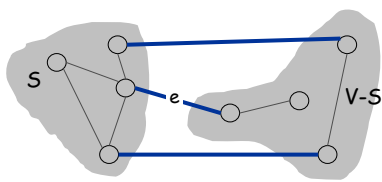
Taglio $[S, V-S] = (\{4, 5, 8\}, \{1, 2, 3, 6, 7\})$
 Archi che attraversano $[S, V-S]$ $D = \{5-6, 5-7, 3-4, 3-5, 7-8\}$

113

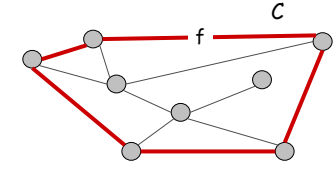
113

Algoritmi Greedy

- **Per il momento** assumiamo per semplicità che i pesi degli archi siano a due a due distinti
 - Vedremo che questa assunzione implica che il minimo albero ricoprente è unico.
- **Proprietà del taglio.** Sia S un qualsiasi sottoinsieme di nodi e sia e l'arco di costo minimo che attraversa il taglio $[S, V-S]$. Ogni minimo albero ricoprente contiene e .
- **Proprietà del ciclo.** Sia C un qualsiasi ciclo e sia f l'arco di costo massimo in C . Nessun minimo albero ricoprente contiene f .



e è nello MST



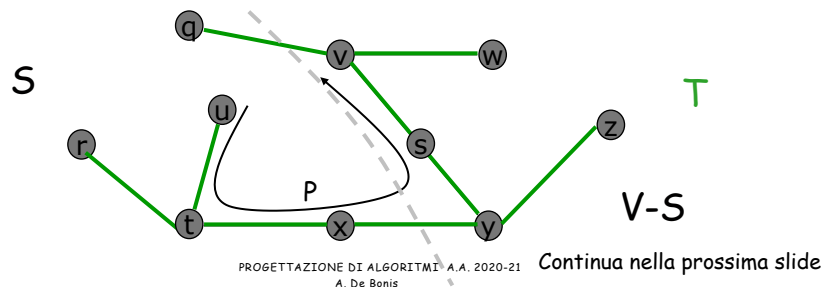
f non è nello MST

114

114

Proprietà del taglio

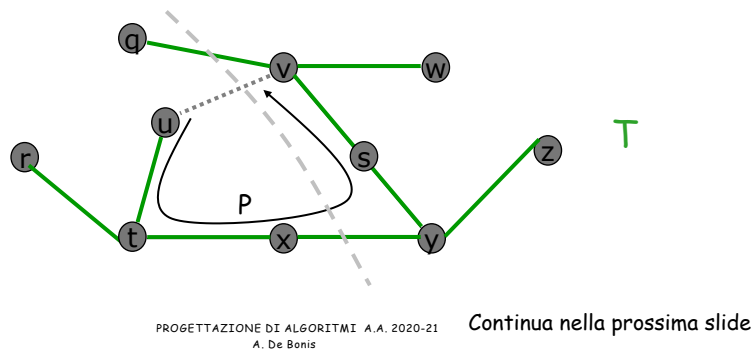
- Stiamo assumendo (per il momento) che i costi degli archi siano a due a due distinti
- **Proprietà del taglio.** Sia S un qualsiasi sottoinsieme di nodi e sia $e=(u,v)$ l'arco di costo minimo che attraversa il taglio $[S, V-S]$. Ogni minimo albero ricoprente contiene e .
- **Dim. (nel caso in cui i costi degli archi sono a due a due distinti)**
- Sia T un albero ricoprente tale che $e=(u,v) \notin T$. Dimostriamo che T non può essere un minimo albero ricoprente.
- Sia P il percorso da u a v in T . In T non ci sono altri percorsi da u a v altrimenti ci sarebbe un ciclo



115

Proprietà del taglio

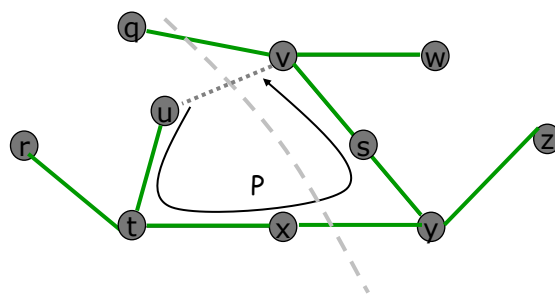
- Siccome u e v sono ai lati opposti del taglio $[S, V-S]$ allora il percorso P da u a v in T deve comprendere un arco $f=(x,y)$ che attraversa il taglio $[S, V-S]$



116

Proprietà del taglio

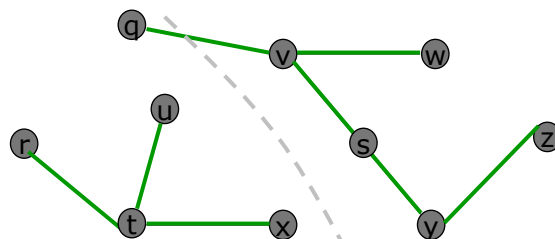
- Poichè $(x,y) \neq (u,v)$ e poichè entrambi attraversano il taglio e (u,v) è l'arco di peso minimo tra quelli che attraversano il taglio allora si ha $c_e < c_f$



117

Proprietà del taglio

- Poichè $(x,y) \neq (u,v)$ e poichè entrambi attraversano il taglio e (u,v) è l'arco di peso minimo tra quelli che attraversano il taglio allora si ha $c_e < c_f$
- $f=(x,y)$ si trova sull'unico percorso P che connette u a v in T
- Se togliamo $f=(x,y)$ da T dividiamo T in due alberi, uno contenente u e l'altro contenente v

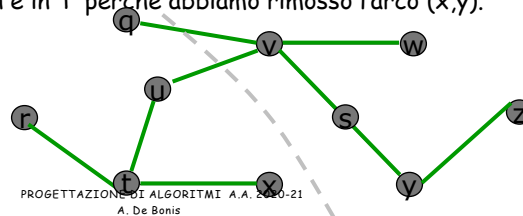


PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

118

Proprietà del taglio

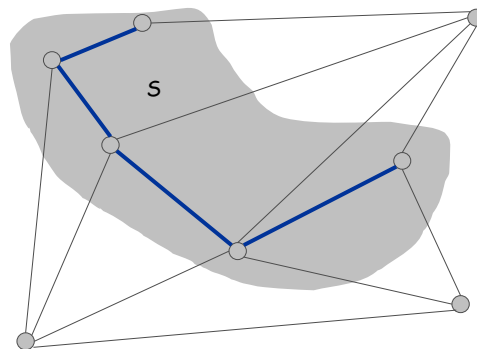
- Poichè $(x,y) \neq (u,v)$ e poichè entrambi attraversano il taglio e (u,v) è l'arco di peso minimo tra quelli che attraversano il taglio allora si ha $c_e < c_f$
- $f=(x,y)$ si trova sull'unico percorso che connette u a v in T
- Se togliamo $f=(x,y)$ da T dividiamo T in due alberi, uno contenente u e l'altro contenente v
- Se introduciamo $e=(u,v)$ riconnettiamo i due alberi ottenendo un nuovo albero ricoprente $T' = T - \{f\} \cup \{e\}$ dove tutte le coppie di nodi che prima erano connesse da un percorso contenente (x,y) , ora sono connesse da un percorso che passa attraverso (u,v)
- Il costo di T' è $c(T') = c(T) - c_f + c_e < c(T)$. Ciò vuol dire che T non è un minimo albero ricoprente.
- Si noti che T' non contiene cicli perché l'unico ciclo su cui si potrebbe venire a trovare (u,v) quando lo aggiungiamo a T è quello formato da P e dall'arco (u,v) ma il percorso P non è in T' perchè abbiamo rimosso l'arco (x,y) .



119

Algoritmo di Prim

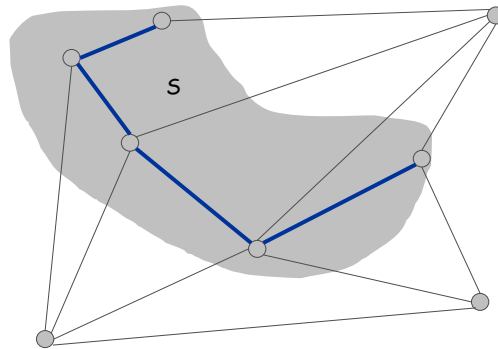
- **Algoritmo di Prim.** [Jarník 1930, Prim 1957, Dijkstra 1959,]
- Ad ogni passo T è un sottoinsieme di archi dello MST
- S = insieme di nodi di T
- **Inizializzazione:** Pone in S un qualsiasi nodo u . Il nodo u sarà la radice dello MST
- Ad ogni passo aggiunge a T l'arco (x,y) di costo minimo tra tutti quelli che congiungono un nodo x in S ad un nodo y in $V-S$ (scelta greedy)
- Termina quando $S=V$



120

Correttezza dell'algoritmo di Prim

- L'algoritmo di Prim ad ogni passo inserisce in T l'arco di costo minimo tra quelli che attraversano il taglio $[S, V-S]$. Quindi per ogni arco e di T esiste un taglio per cui e è l'arco di costo più piccolo tra quelli che lo attraversano.
- La proprietà del taglio implica che ogni albero ricoprente deve contenere ciascuno degli archi selezionati dall'algoritmo $\rightarrow T$ è un sottoinsieme di MST



continua nella prossima slide

121

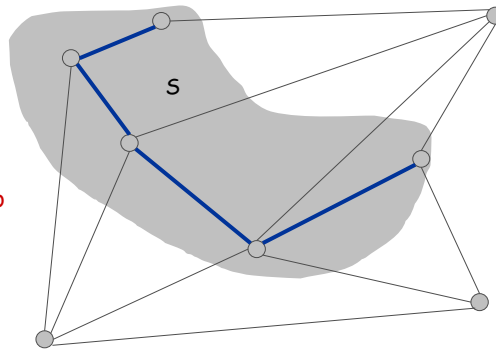
121

Correttezza dell'algoritmo di Prim

- Ci resta da dimostrare che al termine dell'algoritmo di Prim, l'albero T costruito dall'algoritmo è un albero ricoprente, cioè che T effettivamente connette ogni nodo di V .
- Ciò è un'ovvia conseguenza del fatto che l'algoritmo si ferma solo quando $S=V$, cioè quando ha attaccato tutti i vertici all'albero, si ha che T è un albero

In conclusione T è un albero ricoprente che contiene esclusivamente archi che fanno parte dello MST e quindi T è lo MST.

NB: quando i costi sono a due a due distinti c'è un unico MST in quanto per ogni taglio c'è un unico arco di costo minimo che lo attraversa



122

Implementazione dell' algoritmo di Prim con coda a priorità

- Mantiene un insieme di vertici esplorati S .
- Per ogni nodo non esplorato v , mantiene $a[v]$ = costo dell' arco di costo più basso tra quelli che uniscono v ad un nodo in S
- Mantiene coda a priorità Q delle coppie $(a[v], v) \forall v \notin S$
- Stessa analisi dell' algoritmo di Dijkstra con coda a priorità:
- $O(n^2)$ con array o lista non ordinati;
- $O(m \log n + n \log n)$ con heap. Siccome nel problema dello MST il grafo è connesso allora $m \geq n-1$ e $O(m \log n + n \log n) = O(m \log n)$

```

Prim's Algorithm (G,c)
Let S be the set of explored nodes
For each u not in S, we store the cost a[u]
Let Q be a priority queue of pairs (a[u],u) s.t. u is not in S

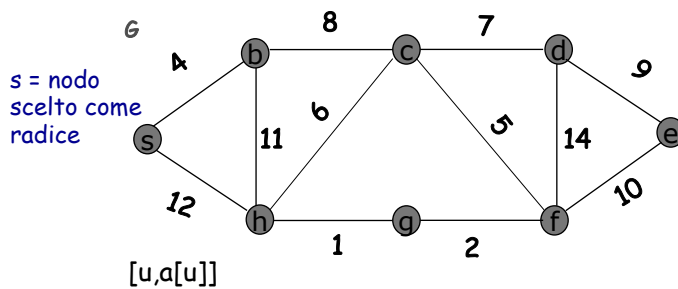
For each u in V insert (Q, ∞,u) in Q EndFor
While (Q is not empty)
  (a[u],u) ← ExtractMin(Q)
  Add u to S
  For each edge e=(u,v)
    If ((v not in S) && (ce < a[v]))
      ChangeKey(Q,v, ce)
EndWhile

```

123

123

Un esempio



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

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

$Q = \{[b, 4], [c, \infty], [d, \infty], [e, \infty], [f, \infty], [g, \infty], [h, 12]\}$

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

124

Un esempio

G

$Q = \{[b, 4], [c, \infty], [d, \infty], [e, \infty], [f, \infty], [g, \infty], [h, 12]\}$

- Si estrae **b** da Q .
- Si aggiornano i campi a dei nodi adiacenti a **b** che si trovano in Q

$Q = \{[c, 8], [d, \infty], [e, \infty], [f, \infty], [g, \infty], [h, 11]\}$

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

125

Un esempio

G

$Q = \{[c, 8], [d, \infty], [e, \infty], [f, \infty], [g, \infty], [h, 11]\}$

- Si estrae **c** da Q .
- Si aggiornano i campi a dei nodi adiacenti a **c** che si trovano in Q

$Q = \{[d, 7], [e, \infty], [f, 5], [g, \infty], [h, 6]\}$

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

126

Un esempio

$Q = \{[d, 7], [e, \infty], [f, 5], [g, \infty], [h, 6]\}$

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

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

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

127

Un esempio

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

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

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

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

128

Un esempio

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

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

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

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

129

Un esempio

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

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

$Q = \{[e, 9]\}$

Si estrae **e** da Q

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

130

Algoritmo di Prim

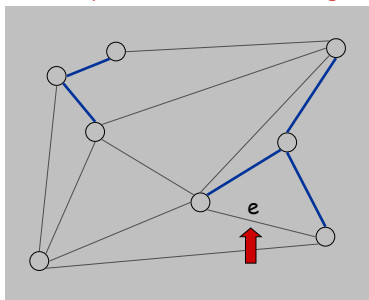
- Esercizio: modificare il codice dell'algoritmo di Prim in modo che l'algoritmo restituisca l'insieme di T degli archi che fanno parte dello MST.

131

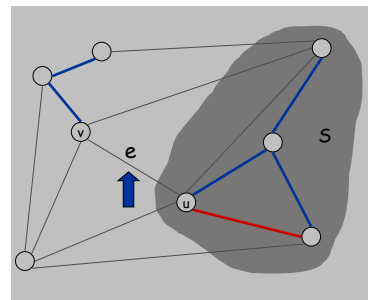
131

Algoritmo di Kruskal

- **Algoritmo di Kruskal's** . [Kruskal, 1956]
 - Considera ciascun arco in ordine non decrescente di peso
 - Caso 1: Se e crea un ciclo allora scarta e
 - Case 2: Altrimenti inserisce e in T
 - **NB:** Siccome gli archi selezionati dall'algoritmo non creano cicli in T allora durante l'esecuzione dell'algoritmo, il grafo formato dai nodi di V insieme agli archi di T è una foresta di alberi, cioè le componenti connesse del grafo (V,T) sono alberi.



Caso 1



Caso 2

132

132

Esempio

Archi in MST : **rossi** (già selezionati) e **verdi** (non ancora selezionati)

$T = \{(r,q), (t,u), (t,x), (v,s), (s,y)\}$ archi dello MST già inseriti

Componenti connesse (alberi) in $G_T = (V, T)$:

$C_1 = \{(r,q), \{(r,q)\}\}$

$C_2 = \{(u,t,x,v,s,y), \{(u,t), (t,x), (v,s), (s,y), (x,y)\}\}$

$C_4 = \{(w), \emptyset\}$

$C_5 = \{(z), \emptyset\}$

Il prossimo arco selezionato è (v,w)

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

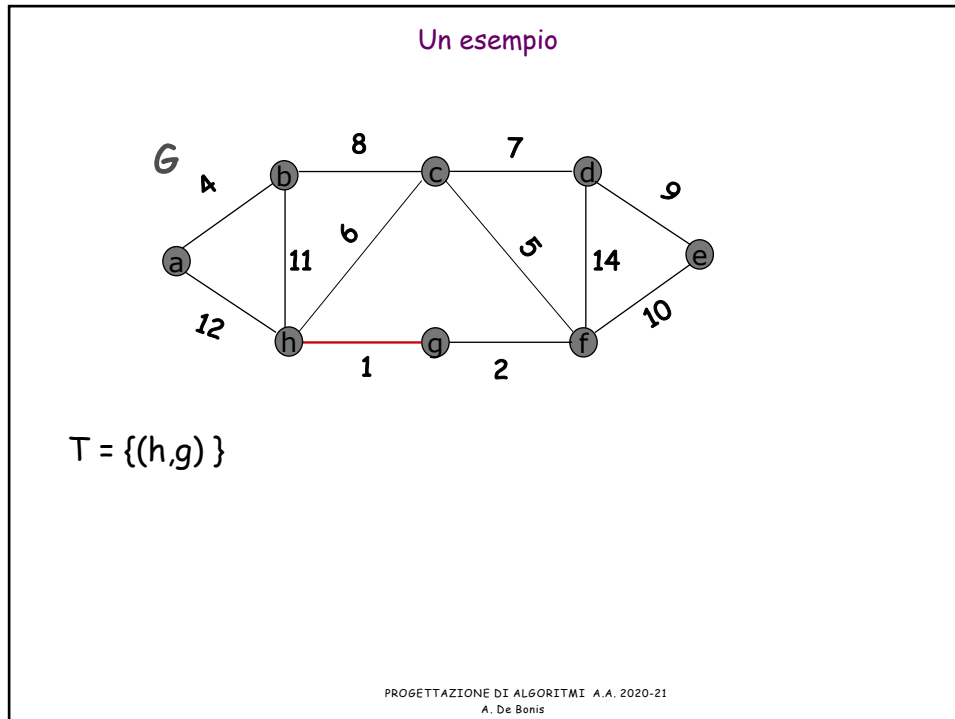
133

Un esempio

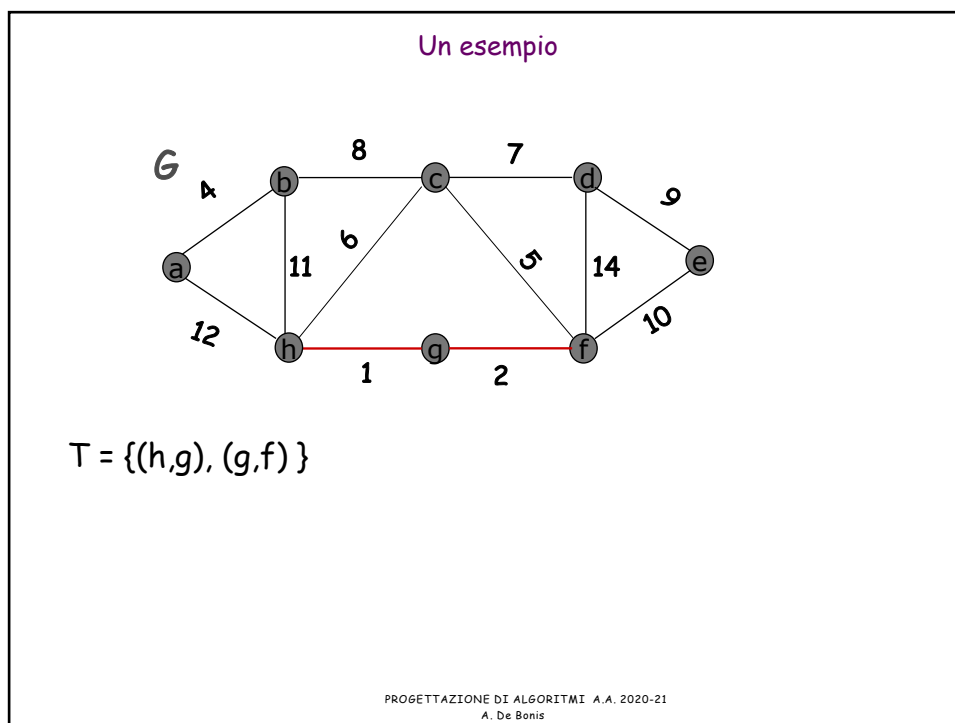
$T = \{\}$

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

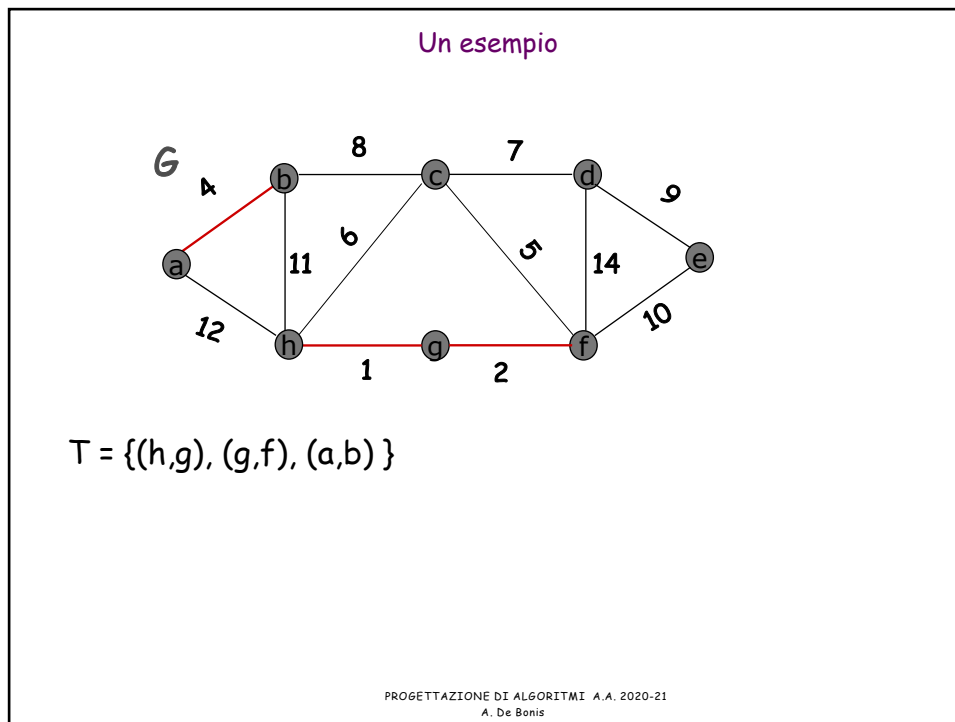
134



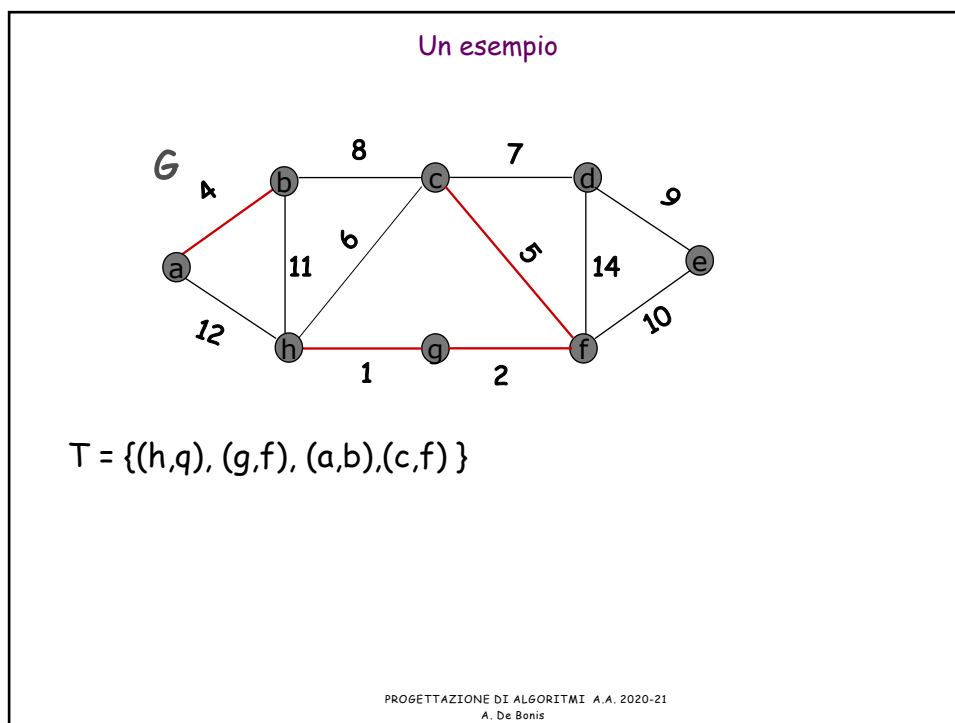
135



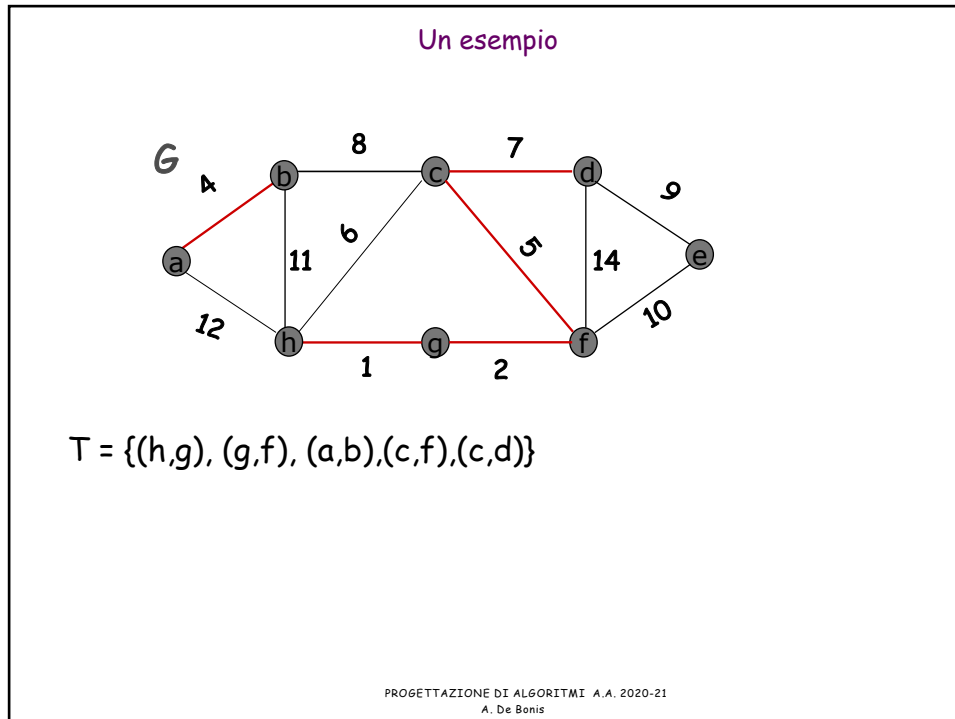
136



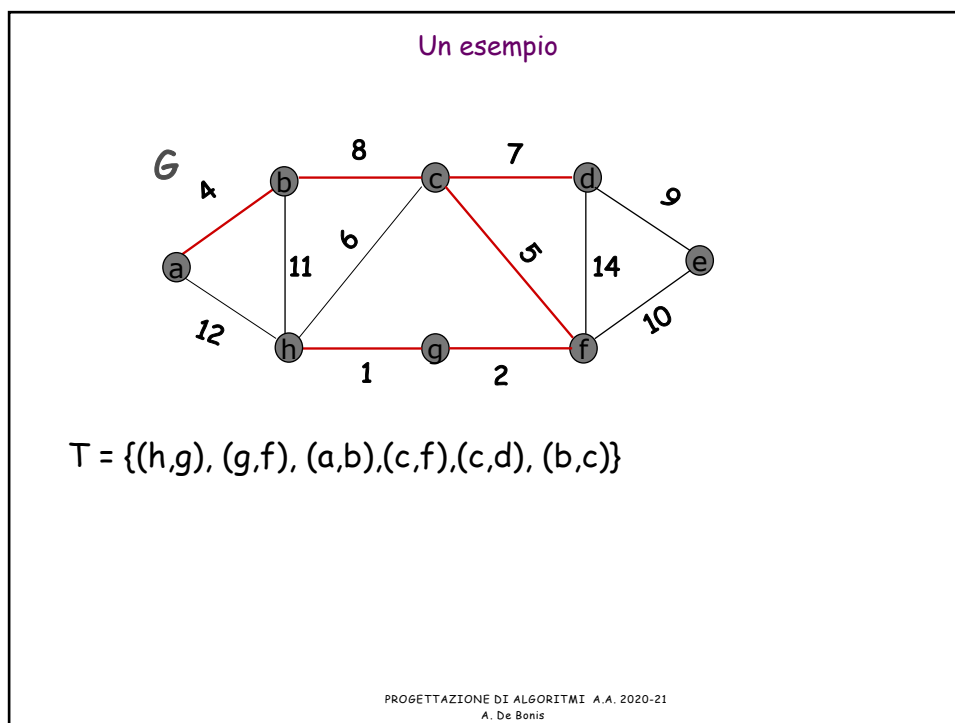
137



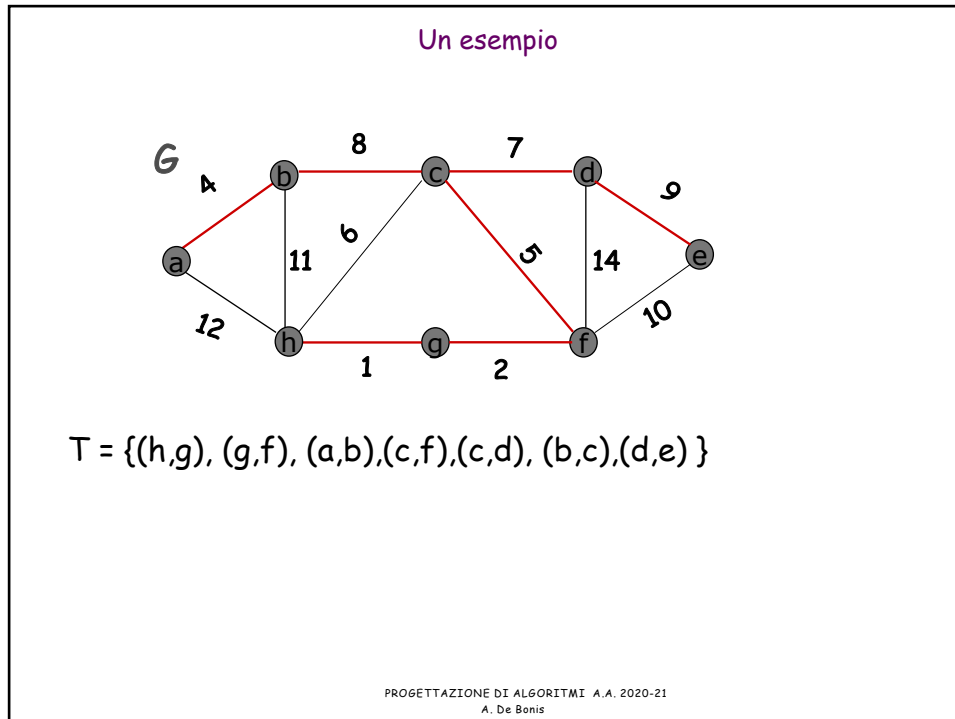
138



139



140



141

Correttezza dell'algoritmo di Kruskal

L'insieme di archi T prodotto dall'algoritmo di Kruskal è un MST

- La dimostrazione è per il caso in cui gli archi hanno costi a due a due distinti
- Prima dimostriamo che ogni arco di T è anche un arco di MST
- Sia $e=(u,v)$ l'arco inserito in un certo passo. **Dimostriamo che esiste un taglio per il quale $e=(u,v)$ è l'arco di peso minimo che lo attraversa.**
 - Osserviamo che T fino a quel momento non contiene un percorso che collega u a v , altrimenti introducendo $e=(u,v)$ in T si creerebbe un ciclo e l'algoritmo scarterebbe (u,v) senza inserirlo in T .
 - Consideriamo l'albero contenente u nel momento in cui $e=(u,v)$ viene inserito in T . Chiamiamo S l'insieme dei vertici contenuti in questo albero, cioè l'insieme dei nodi connessi ad u fino a quel momento. Ovviamente v non è in S altrimenti esisterebbe un percorso da u a v . Quindi $e=(u,v)$ attraversa il taglio $[S, V-S]$.

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

142

Correttezza dell'algoritmo di Kruskal

- Nella slide precedente abbiamo dimostrato che $e=(u,v)$ attraversa il taglio $[S, V-S]$
- Ora dimostriamo che $e=(u,v)$ è l'arco di peso minimo tra quelli che attraversano il taglio $[S, V-S]$
- Dimostriamo che nessuno degli archi esaminati dall'algoritmo prima di $e=(u,v)$ unisce un nodo di S ad un nodo di $V-S$.
- Supponiamo per assurdo che l'algoritmo abbia esaminato prima dell'arco $e=(u,v)$ un arco (x,y) che unisce un nodo x di S ad un nodo y di $V-S$. L'algoritmo avrebbe aggiunto y ad S visto che la sua introduzione non avrebbe creato cicli. Di conseguenza y non potrebbe trovarsi in $V-S$ quando $e=(u,v)$ viene esaminato e introdotto in T . Siamo arrivati ad una contraddizione in quanto stiamo supponendo che y sia in $V-S$ quando ciò accade.
- Quindi $e=(u,v)$ è il primo arco esaminato dall'algoritmo tra quelli che attraversano il taglio $[S, V-S]$. Siccome gli archi sono esaminati in ordine non decrescente di peso, $e=(u,v)$ è l'arco di peso minimo che attraversa il taglio $[S, V-S]$

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

143

143

Correttezza dell'algoritmo di Kruskal

- Abbiamo dimostrato che per ciascuno degli archi (u,v) selezionati dall'algoritmo di Kruskal esiste un taglio $[S, V-S]$ per cui (u,v) è l'arco di peso minimo tra quelli che attraversano il taglio $[S, V-S]$.
- Quindi per la proprietà del taglio tutti gli archi selezionati dall'algoritmo sono nel minimo albero ricoprente.
- Ora dimostriamo che al termine dell'esecuzione dell'algoritmo T è un albero ricoprente.
- T è un albero ricoprente perchè
 - l'algoritmo non introduce mai cicli in T (ovvio!)
 - connette tutti i vertici
 - Se così non fosse esisterebbe un insieme W non vuoto, di al più $n-1$ vertici, tale che non c'è alcun arco di T che connette un vertice di W ad uno di $V-W$.
 - Siccome il grafo input G è connesso devono esistere uno o più archi in G che connettono vertici di W a vertici di $V-W$
 - Dal momento che l'algoritmo di Kruskal esamina tutti gli archi avrebbe selezionato sicuramente l'arco di costo minimo tra quelli che connettono un vertice di W ad uno di $V-W$. Non è quindi possibile che in T non vi sia un arco che connette un nodo in W ad uno in $V-W$.

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

144

144

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  $c_1 \leq c_2 \leq \dots \leq c_m$ .
  T  $\leftarrow \phi$ 

  foreach (u  $\in$  V) make a set containing singleton u

  for i = 1 to m
    (u,v) =  $e_i$ 
    if (u and v are in different sets) {
      T  $\leftarrow$  T  $\cup$  { $e_i$ }
      merge the sets containing u and v
    }
  return T
}

```

are u and v in different connected components?

merge two components

145

145

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 ciclo 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

146

146

Implementazione dell'algoritmo di Kruskal con Union-Find

```

Kruskal(G, c) {
  Sort edges weights so that  $c_1 \leq c_2 \leq \dots \leq c_m$ .
  T  $\leftarrow \emptyset$ 

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

  for i = 1 to m
    (u,v) =  $e_i$ 
    if (Find(u)  $\neq$  Find(v)) {
      T  $\leftarrow$  T  $\cup$  { $e_i$ }
      Union(Find(u), Find(v))
    }
  return T
}

```

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

147

147

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 $\rightarrow O(m)$

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

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

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

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

148

148

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 altro array per mantenere traccia della cardinalità di ciascun insieme. Inoltre per l'analisi che faremo è utile che l'array S memorizzi in ciascuna cella un puntatore ad una lista contenente gli elementi dell'insieme. Ciò consente, in un'operazione di unione, di trovare le celle degli elementi che vengono spostati senza dover scandire l'intero array. Ovviamente si mantiene un'unica lista per ogni insieme.
 - La $\text{Find}(x)$ è identica a prima
 - $\text{MakeUnionFind } O(n)$: occorre creare n liste ciascuna 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 più piccolo in quello più grande. Ciascun elemento x dell'insieme più piccolo viene aggiunto alla lista degli elementi dell'insieme più grande e si aggiorna la cella di indice x di S con il puntatore a questa lista. 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.

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

149

149

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

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

150

150

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 $2^k \rightarrow x$ 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 $O(n \log n)$.

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

151

151

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

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

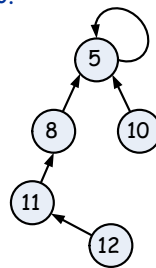
152

152

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



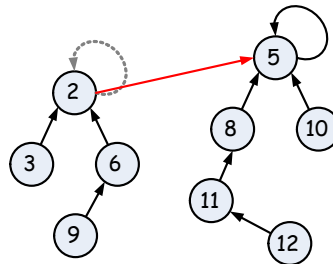
PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

153

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.



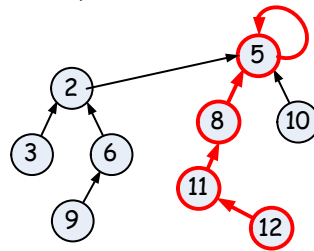
PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

154

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)



PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

155

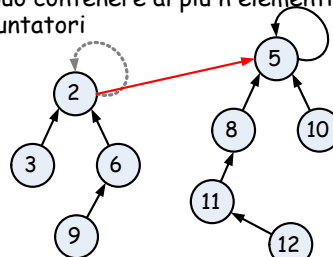
Euristica per migliorare l'efficienza

- 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



156

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

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis

157

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

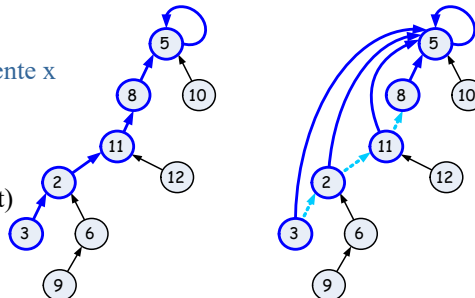
`Find(x)`

if $q(x) \neq q(x).pointer$

then $p \leftarrow q(x).pointer$

$q(x).pointer \leftarrow Find(p.element)$

return $q(x).pointer$



158

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

PROGETTAZIONE DI ALGORITMI A.A. 2020-21
A. De Bonis