

Algoritmi greedy V parte

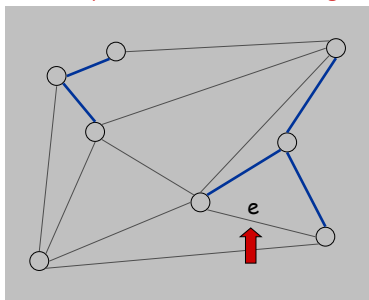
Progettazione di Algoritmi a.a. 2022-23
Matricole congrue a 1
Docente: Annalisa De Bonis

148

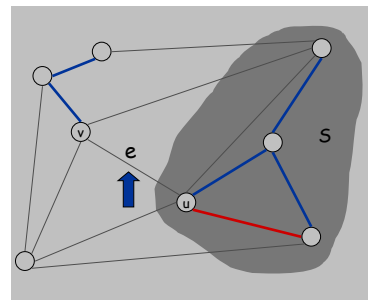
148

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

149

149

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. 2022-23
A. De Bonis

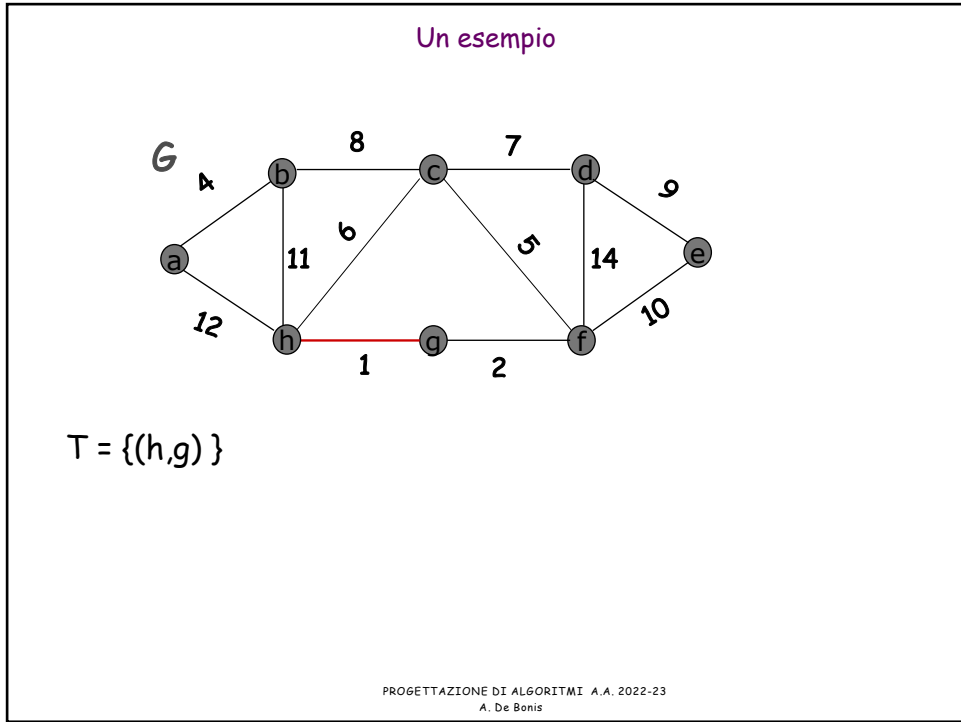
150

Un esempio

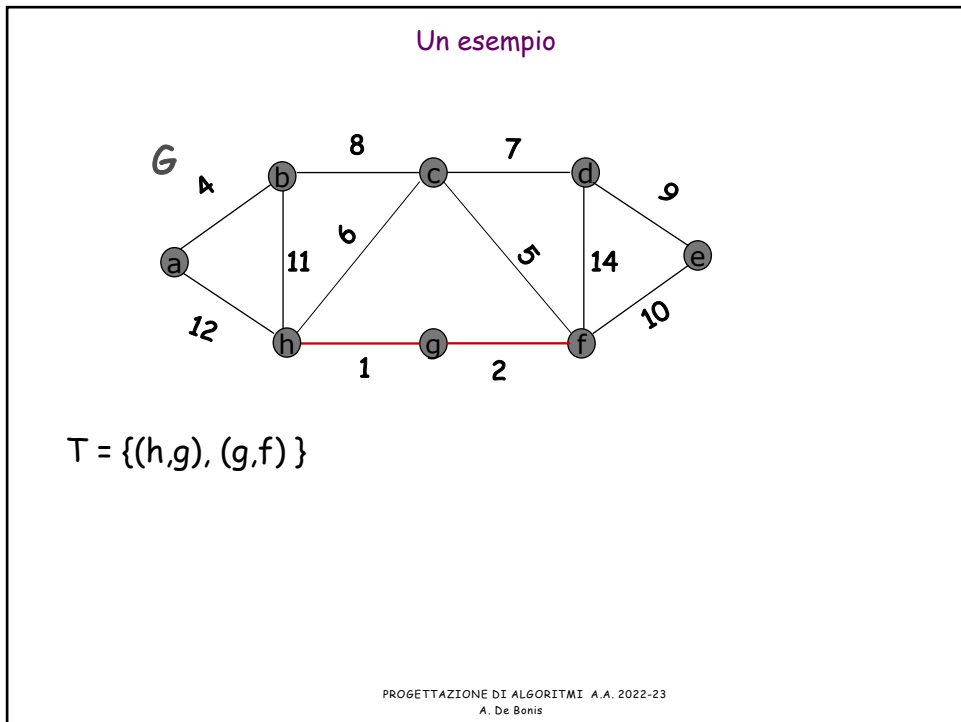
$T = \{\}$

PROGETTAZIONE DI ALGORITMI A.A. 2022-23
A. De Bonis

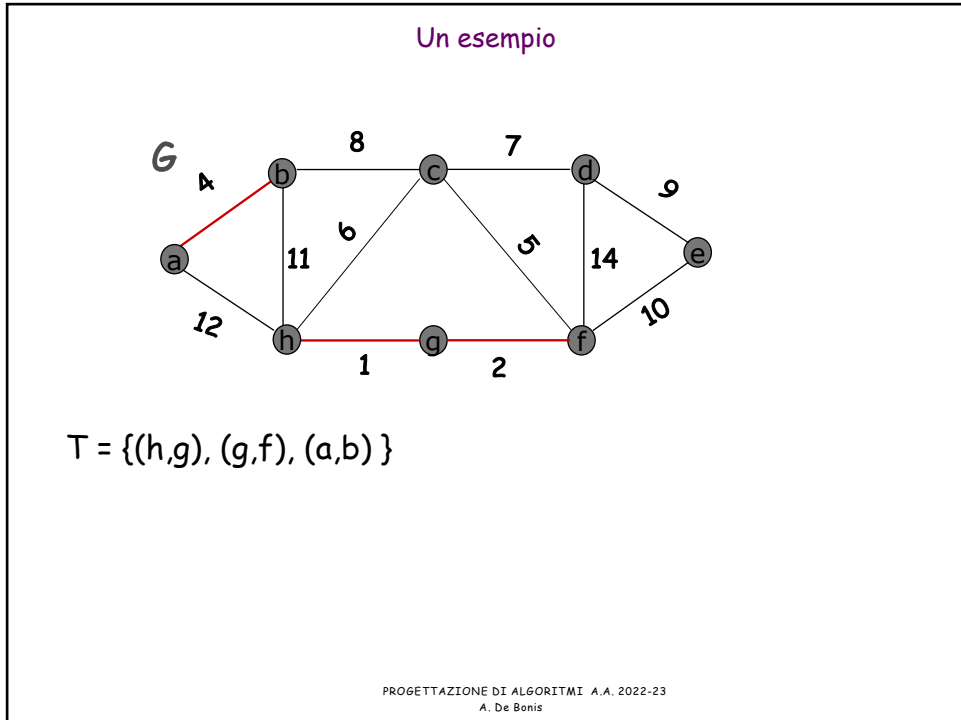
151



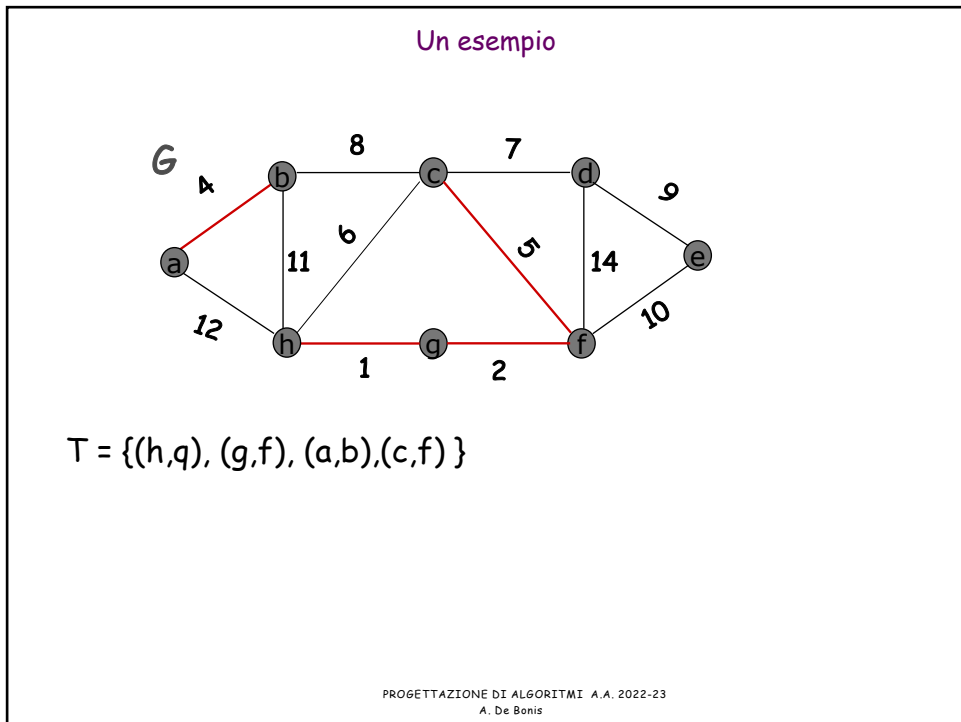
152



153



154



155

Un esempio

$T = \{(h,g), (g,f), (a,b), (c,f), (c,d)\}$

PROGETTAZIONE DI ALGORITMI A.A. 2022-23
A. De Bonis

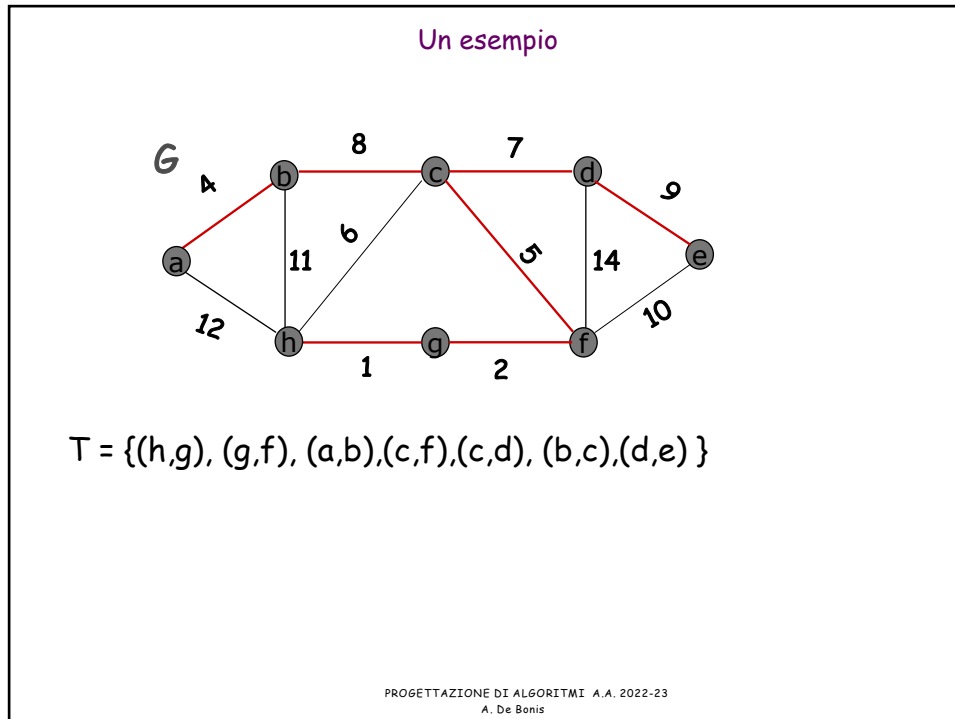
156

Un esempio

$T = \{(h,g), (g,f), (a,b), (c,f), (c,d), (b,c)\}$

PROGETTAZIONE DI ALGORITMI A.A. 2022-23
A. De Bonis

157



158

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 della foresta 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. 2022-23
A. De Bonis

159

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 un 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 altro arco (x,y) che attraversa $[S,V-S]$, cioè un arco 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 e 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 un arco di peso minimo che attraversa il taglio $[S, V-S]$

160

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 un arco di peso minimo tra quelli che attraversano il taglio $[S, V-S]$.
- Quindi per la proprietà del taglio ciascuno degli archi selezionati dall'algoritmo è contenuto in un minimo albero ricoprente.

161

Correttezza dell'algoritmo di Kruskal

- Potrebbe sorgere un'obiezione: chi mi dice che gli archi selezionati facciano però parte di uno stesso MST?
- Supponiamo che fino ad un certo passo Kruskal abbia selezionato un certo insieme di archi A e che questi siano effettivamente tutti all'interno di uno stesso MST. Per quanto visto nelle slide precedenti, A è una foresta e l'arco (u,v) introdotto da Kruskal al prossimo passo è quello di costo minimo che attraversa il taglio $[S, V-S]$ dove S è la componente connessa di u in A (l'albero della foresta che contiene u subito prima che venga inserito (u,v)). Potrebbe accadere che selezionando un arco (u,v) , $A \cup \{(u,v)\}$ non sia più un sottoinsieme di MST?
- Osserviamo che ciò non può accadere se (u,v) è l'unico arco di costo minimo che attraversa il taglio in quanto in questo caso (u,v) fa sicuramente parte di ogni MST e quindi anche di quelli che contengono gli archi di A (proprietà taglio, punto 1, slide 134)
- Consideriamo allora il caso in cui ci sono più archi di costo minimo che attraversano $[S, V-S]$.
- Sia T un MST che contiene tutti gli archi di A (siamo sicuri che esista per l'ipotesi su A)
- Il punto 2 della proprietà del taglio (slide 134) ci dice che o (u,v) è nello MST T o che esiste un arco (x,y) che attraversa $[S, V-S]$ che può essere rimpiazzato con (u,v) in modo che il nuovo albero sia ancora un MST e contenga tutti gli altri di T diversi da (x,y) → otteniamo un nuovo MST che contiene sia (u,v) che tutti gli archi di A .

PROGETTAZIONE DI ALGORITMI A.A. 2022-23
A. De Bonis

162

162

Correttezza dell'algoritmo di Kruskal

- Abbiamo dimostrato che tutti gli archi selezionati da Kruskal formano un sottoinsieme di uno MST.
- Ora dimostriamo che al termine dell'esecuzione dell'algoritmo T è un albero ricoprente.
- T è un albero ricoprente perché
 - l'algoritmo non introduce mai cicli in T (implicato dal fatto che gli archi formano un sottoinsieme dello MST oltre che da come funziona l'algoritmo)
 - 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. 2022-23
A. De Bonis

163

163

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

164

164

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

165

165

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. 2022-23
A. De Bonis

166

166

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. 2022-23
A. De Bonis

167

167

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 $\text{Find}(x)$ è identica a prima
 - $\text{MakeUnionFind } O(n)$: 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 più piccolo in quello più grande. Ciascun elemento x dell'insieme più piccolo viene aggiunto all'insieme più grande e si aggiorna la 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.

168

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

169

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. 2022-23
A. De Bonis

170

170

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. 2022-23
A. De Bonis

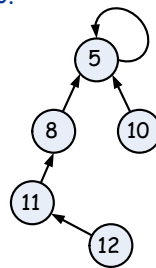
171

171

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. 2022-23
A. De Bonis

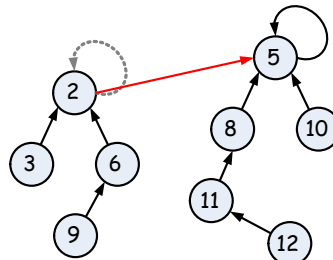
172

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.



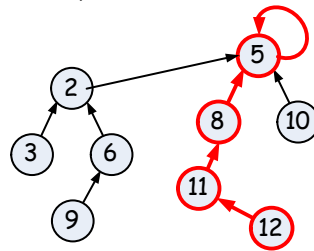
PROGETTAZIONE DI ALGORITMI A.A. 2022-23
A. De Bonis

173

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. 2022-23
A. De Bonis

174

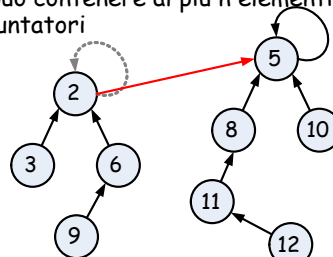
Euristica per migliorare l'efficienza

- 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



175

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

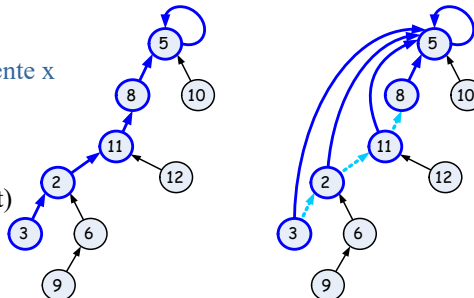
PROGETTAZIONE DI ALGORITMI A.A. 2022-23
A. De Bonis

176

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 $q(x).pointer$



177

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. 2022-23
A. De Bonis