12. Minimo albero ricoprente

Ricordiamo che due vertici x e y di un grafo sono connessi se esiste un cammino, cioè una sequenza di archi, che li collega. Un grafo non orientato è connesso se per ogni coppia di vertici x e y, x e y sono connessi. Una componente connessa di un grafo non orientato è definita da un sottoinsieme massimale di vertici del grafo che risultino connessi tra loro.

Un *albero* è un grafo non orientato connesso privo di cicli.

Valgono le seguenti proprietà:

- Un grafo non orientato è un albero se e solo se tra ogni coppia di vertici vi è un unico cammino che li connette.
- Ogni albero di n vertici contiene esattamente $n-1$ archi.
- Se ad un albero aggiungiamo un arco (ma non un vertice) otteniamo un ciclo; se rimuoviamo un arco, senza rimuovere vertici, il grafo risultante non è più connesso.

Chiamiamo *foresta* una collezione di alberi (quindi in una foresta richiediamo l'assenza di cicli, ma non la connessione). Estendendo le proprietà precedenti possiamo dimostrare che una foresta di n vertici e k alberi contiene esattamente $n-k$ archi.

Dato un grafo non orientato $G = (V, E)$ un *albero di ricoprimento* (o *albero ricoprente*, *albero di supporto* o, in inglese, *spanning tree*) è un qualunque sottografo $G' = (V', E')$ di G con $V' = V$, $E' \subseteq E$ che sia un albero. Dunque, per essere un albero ricoprente, G' deve possedere lo stesso insieme V di vertici di G, un sottoinsieme $E' \subseteq E$ degli archi, deve essere connesso, e privo di cicli.

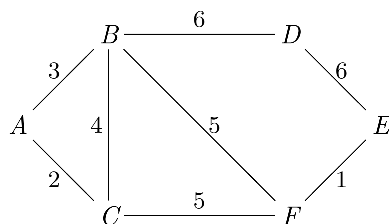
Consideriamo ora grafi nei quali ad ogni arco associamo un valore o peso, mediante una funzione $\omega : E \to \mathbb{R}$, detta anche *funzione peso*. Dato un grafo pesato $G = (V, E, \omega)$, definiamo il peso di G, indicato con $\omega(G)$, come la somma dei pesi degli archi di G, cioè $\omega(G) = \sum_{e \in E} \omega(e)$.

Dato un grafo non orientato e pesato $G = (V, E, \omega)$, *un albero ricoprente minimo* di G è un albero ricoprente $T = (V, E_T)$, il cui peso sia minimo, tra tutti gli alberi ricoprenti di G, cioè per ogni albero ricoprente T' di G risulti $\omega(T) \leq \omega(T')$.

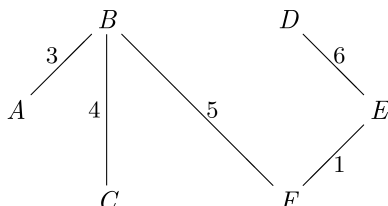
Esempio.
Sia G il grafo nella seguente figura, dove i numeri rappresentano i pesi degli archi:

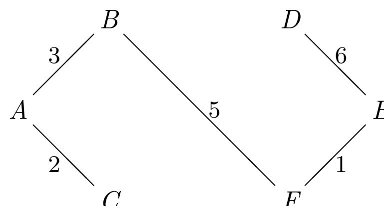
Il presente materiale integra *ma non sostituisce* il libro di testo consigliato.
Data pubblicazione: 6 dicembre 2019

© 2019 Giovanni Pighizzini
Il contenuto di queste pagine è protetto dalle leggi sul copyright e dalle disposizioni dei trattati internazionali. Il titolo ed i copyright relativi alle pagine sono di proprietà dell'autore. Le pagine possono essere riprodotte ed utilizzate liberamente dagli studenti, dagli istituti di ricerca, scolastici e universitari afferenti al Ministero dell'Istruzione, dell'Università e della Ricerca, per scopi istituzionali, non a fine di lucro. Ogni altro utilizzo o riproduzione (ivi incluse, ma non limitatamente a, le riproduzioni a mezzo stampa, su supporti magnetici o su reti di calcolatori) in toto o in parte è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte dell'autore.
L'informazione contenuta in queste pagine è ritenuta essere accurata alla data della pubblicazione. Essa è fornita per scopi meramente didattici e non per essere utilizzata in progetti di impianti, prodotti, ecc.
L'informazione contenuta in queste pagine è soggetta a cambiamenti senza preavviso. L'autore non si assume alcuna responsabilità per il contenuto di queste pagine (ivi incluse, ma non limitatamente a, la correttezza, completezza, applicabilità ed aggiornamento dell'informazione). In ogni caso non può essere dichiarata conformità all'informazione contenuta in queste pagine. In ogni caso questa nota di copyright non deve mai essere rimossa e deve essere riportata anche in utilizzi parziali.



Nella figura seguente sono rappresentati due alberi ricoprenti di G, Quello di sinistra ha peso 19, quello di destra ha peso 17. Si può verificare che non esistono alberi ricoprenti di peso inferiore a 17. Pertanto l'albero di destra è un albero ricoprente minimo per il grafo G. Si può anche osservare che esistono altri alberi ricoprenti minimi, cioè con peso 17.





□

Studiamo ora due algoritmi per trovare un albero ricoprente minimo di un grafo connesso, non orientato e pesato. In entrambi gli algoritmi l'albero viene costruito in modo incrementale utilizzando una *strategia greedy.*

12.1 L'algoritmo di Kruskal

Il primo algoritmo risolve il problema costruendo un grafo T che ha gli stessi vertici di G e, inizialmente, è privo di archi. L'algoritmo esamina gli archi di G in ordine di peso non decrescente. Un arco viene aggiunto a T se, insieme a quelli già scelti, non forma cicli, altrimenti viene scartato e non sarà più considerato. Pertanto, ad ogni passo, il grafo T è una foresta di alberi. Ogni volta che si aggiunge un grafo, si connettono tra loro due alberi della foresta che diventano, con l'arco aggiunto, un unico albero. Alla fine, quando sono stati esaminati tutti gli archi, T contiene un unico albero che, come dimostreremo, è un albero ricoprente di peso minimo per il grafo G dato.

Lo schema dell'algoritmo di Kruskal è presentato nel Codice 12.1.

Codice 12.1 Algoritmo di Kruskal

```
Algoritmo Kruskal (grafo connesso non orientato G = (V, E, ω)) → albero
  /* Costruzione di un albero ricoprente minimo                        */
  ordina l'insieme E in base ai pesi in modo non decrescente
  T ← (V, ∅)
  foreach (x, y) ∈ E secondo l'ordine do
      if x e y non sono connessi in T then
          aggiungi al grafo T l'arco (x, y)
  return T
```

Dimostriamo ora che l'algoritmo trova sempre la soluzione ottima, cioè trova sempre un albero ricoprente di peso minimo.

Teorema 1 *Dato un grafo connesso e non orientato $G = (V, E, \omega)$, l'albero ottenuto dall'algoritmo di Kruskal (Codice 12.1) è un albero ricoprente minimo di G*

Dimostrazione.
Sia $T = (V, E_T)$ l'albero costruito dall'Algoritmo di Kruskal. Indichiamo con $T_0 = (V, E_0)$ un albero ricoprente minimo di G con *il maggior numero di archi* in comune con T.

Supponiamo per assurdo che $T \neq T_0$ e consideriamo il primo arco (x, y), secondo l'ordine utilizzato dall'algoritmo, con $(x, y) \in E_T \setminus E_0$.

Poiché T_0 è un albero, i vertici x e y sono connessi da un cammino ρ in T_0. Pertanto, aggiungendo (x, y) a T_0 si otterrebbe un ciclo. Ciò implica che esiste un arco (u, v) su tale cammino ρ con $(u, v) \in E_0 \setminus E_T$ e dunque, poiché l'algoritmo di Kruskal procedendo in ordine di peso ha scelto (x, y) e non (u, v), si ha $\omega(x, y) \leq \omega(u, v)$.

Consideriamo ora $T' = (V, E')$ con $E' = E_0 \setminus \{(u, v)\} \cup \{(x, y)\}$. Possiamo osservare che anche T' è un albero ricoprente di G. Inoltre:

$$\omega(T') \leq \omega(T_0) - \omega(u, v) + \omega(x, y) \leq \omega(T_0)$$

Dunque T' è un albero ricoprente che ha al più lo stesso peso di T_0 e ha un arco in più di T_0 in comune con T. Questo contraddice l'ipotesi iniziale su T_0. Pertanto $T_0 = T$. □

Studiamo ora una possibile implementazione dell'algoritmo di Kruskal. È utile rappresentare il grafo come *lista di archi*. La lista può essere rappresentata direttamente in un array, sul quale applicare uno degli algoritmi di ordinamento (in base ai pesi degli archi). Insieme al grafo T che viene costruito, utilizziamo una struttura che quando si ispeziona un arco (x, y) permetta di decidere facilmente se i vertici x e y sono già connessi in T, cioè se esiste un cammino che li collega.

A tale scopo possiamo considerare partizioni dell'insieme dei vertici V, in cui due vertici appartengono allo stesso elemento della partizione se e solo se sono connessi in T. In altre parole, ogni elemento della partizione rappresenta una componente connessa di T.

- Inizialmente ogni vertice di V costituisce un singolo insieme della partizione (T non contiene archi e dunque non ci sono vertici connessi tra di loro).
- Quando esaminiamo un arco (x, y) vi sono due possibilità:
  - Se *x e y appartengono allo stesso elemento della partizione* significa che sono già connessi in T. In tal caso l'arco (x, y) non viene aggiunto a T perché creerebbe un ciclo.
  - Se *x e y appartengono a elementi diversi della partizione* allora non sono connessi: aggiungendo l'arco (x, y) a T rendiamo ciascun vertice dell'elemento cui appartiene x connesso con ciascun vertice dell'elemento cui appartiene y, cioè rendiamo le due componenti connesse a cui appartengono x e y un'unica componente connessa. Pertanto, oltre ad aggiungere l'arco (x, y) all'insieme T, fondiamo i due elementi della partizione.

In altre parole, durante l'esecuzione dell'algoritmo, la partizione rappresenta una foresta di alberi, i cui vertici sono i vertici di G e gli archi sono un sottoinsieme degli archi di G. Ogni elemento della partizione rappresenta un albero. Ogni volta che aggiungiamo un arco, connettiamo tra loro due alberi della partizione.

La partizione può essere rappresentata mediante le strutture Union-Find. Per verificare se x e y appartengono allo stesso insieme della partizione è sufficiente confrontare i risultati di $find(x)$ e $find(y)$. Per unire due elementi della partizione utilizziamo *union*. Si veda il Codice 12.2.

Stimiamo ora il tempo di calcolo, in funzione del numero $n = \#V$ di vertici e del numero $m = \#E$ di archi del grafo G in input. Assumendo il criterio di costo uniforme, supponiamo che i confronti tra i pesi degli archi avvengano in tempo costante. Dobbiamo tenere conto dei seguenti tempi:

- Ordinamento di E:
  Possiamo utilizzare l'algoritmo `heapSort` ed effettuare l'ordinamento in tempo $O(m \log m)$.

Codice 12.2 Algoritmo di Kruskal (mediante Union-Find)

Algoritmo *Kruskal (grafo connesso non orientato $G = (V, E, \omega)$) $\rightarrow$ albero*
```
/* Costruzione di un albero di ricoprimento di peso minimo */
ordina l'insieme E in base ai pesi in modo non decrescente
T ← (V, ∅)
Sia P una partizione inizialmente vuota
foreach v ∈ V do P.makeSet(v)
foreach (x, y) ∈ E secondo l'ordine do
    t_x ← P.find(x)
    t_y ← P.find(y)
    if t_x ≠ t_y then
        P.union(t_x, t_y)
        aggiungi a T l'arco (x, y)
return T
```

- Operazioni Union/Find:
  Supponiamo di utilizzare la tecnica `QuickUnion` con bilanciamento in altezza in cui ciascuna operazione `MakeSet` viene effettuata in tempo costante, `Find` in tempo $O(\log n)$ e `Union` in tempo costante, dove n è il numero di elementi presenti complessivamente negli insiemi della partizione. L'algoritmo effettua le seguenti operazioni:
  - n operazioni `MakeSet`: *tempo totale* $O(n)$,
  - $2m$ operazioni `Find` (2 per ogni arco): *tempo totale* $O(m \log n)$,
  - $n - 1$ operazioni `Union` (una per ogni arco che viene aggiunto, si ricordi che un albero con n vertici contiene $n - 1$ archi): *tempo totale* $O(n)$.

Sommando i vari tempi e osservando che $n-1 \leq m \leq n^2$, essendo G connesso, otteniamo che il tempo totale è $O(m \log m)+O(n)+O(m \log m = O(m \log n^2)+O(m)+O(m \log n)$ e dunque $O(m \log n)$, che può anche essere approssimato con $O(m \log m)$.

Se i pesi sono interi, si potrebbe ridurre il tempo dell'ordinamento utilizzando `radixSort`. Si possono inoltre ridurre i tempi legati alle operazioni sulla partizione utilizzando la compressione di cammino. In questo modo si ottiene tempo totale $O(m \log^* m)$.

12.2 L'algoritmo di Prim

Un altro algoritmo greedy che permette di risolvere il problema dell'albero di ricoprimento minimo è quello proposto da Prim (v. Codice 12.3). Dato in ingresso un grafo connesso, non orientato con pesi sugli archi, l'algoritmo inizia costruendo un albero T formato da un unico vertice s qualsiasi del grafo. Ad ogni passo, l'albero T viene espanso scegliendo, tra tutti gli archi che hanno un vertice in T e l'altro non in T, un arco di peso minimo. Tale arco viene aggiunto a T (insieme al vertice che non era in T). Si può dimostrare che, come l'algoritmo di Kruskal, anche questo algoritmo trova sempre una soluzione ottima.

L'algoritmo di Prim può essere implementato ricorrendo a una coda con priorità C, contenente un elemento per ogni vertice che deve essere ancora inserito nell'albero, secondo la tecnica che descriviamo ora (Codice 12.4).

- Ad ogni passo, per ogni vertice v non ancora in T consideriamo le seguenti informazioni:
  - $d[v]$: minimo peso di un arco tra un vertice appartenente all'albero T già costruito e v, cioè $d[v] = \min(\{\omega(u, v) \mid u \in E_T\} \cup \{\infty\})$ (il valore ∞ indica che non c'è nessun arco da vertici in T a v).

Codice 12.3 Algoritmo di Prim (schema ad alto livello)

Algoritmo *Prim (grafo connesso non orientato* $G = (V, E, \omega)$*)* $\rightarrow$ *albero*
`/* Costruzione di un albero di ricoprimento di peso minimo */`
$T \leftarrow (V_T \leftarrow \{s\}, E_T \leftarrow \emptyset)$ `// albero formato da un solo nodo` s
while T *ha meno di* n *nodi* **do** `//` $n = \#V$
  sia $(x, y) \in E$ di peso minimo con $x \in V_T$ e $y \notin V_T$
  $V_T \leftarrow V_T \cup \{y\}$
  $E_T \leftarrow E_T \cup \{(x, y)\}$
return T

  - *vicino*$[v]$: un vertice u nell'albero T già costruito con distanza minima da v, cioè $u \in V_T$ tale che $\omega(u, v) = d[v]$.

- La coda con priorità C contiene ciascun vertice v, non ancora inserito in T, con priorità $d[v]$.

- Inizialmente l'albero è vuoto. Pertanto, per ogni vertice v si pone $d[v] = \infty$, mentre il valore di *vicino*$[v]$ non è definito. Ogni vertice viene inserito in C.

- Ad ogni passo si sceglie un vertice y corrispondente al minimo in C. (Al primo passo, poiché tutti i vertici hanno priorità uguale a ∞, verrà scelto un qualsiasi vertice.).

- Nei passi successivi al primo, si considera il "vicino" x in T del vertice y scelto (cioè $x =$ *vicino*$[y]$). L'arco (x, y) è pertanto un arco di peso minimo con un vertice x in T e l'altro vertice y non in T. Il vertice y e l'arco (x, y) vengono aggiunti all'albero. (Al primo passo, poiché T è vuoto, il "vicino" x non esiste. In questo caso si aggiunge solo il vertice y all'albero, senza aggiungere archi. Questa situazione viene riconosciuta dal fatto che $d[y] = \infty$.)

- Si ricalcolano le priorità dei vertici, tenendo conto del nuovo vertice y inserito in T. Per ogni arco (y, z) uscente da y, con z non in T, nel caso il peso $\omega(y, z)$ risulti minore di $d[z]$, si modifica $d[z]$ e si aggiorna la coda con priorità e l'informazione relativa al vicino di z.

- Queste operazioni vengono ripetute sino a svuotare la coda. A quel punto si può restituire T.

Forniamo ora una stima del tempo di calcolo. Prima di tutto assumiamo che il grafo in ingresso sia rappresentato mediante liste di adiacenza o di incidenza. Questo permette di trovare facilmente tutti gli archi incidenti su un vertice (ciclo *for-each*). La coda con priorità può essere rappresentata con un array di n elementi (come nello `heapSort`) e riempita in tempo $O(n)$. Verranno eseguite in totale n operazioni `deleteMin()`, ciascuna delle quali impiega tempo al più $O(\log n)$, per un tempo complessivo $O(n \log n)$. Il corpo del ciclo *for-each* interno al ciclo principale viene eseguito in totale al più 2 volte per ciascun arco[1], pertanto $O(m)$ volte. Dunque anche il numero di operazioni `changeKey` è $O(m)$. Ognuna di esse impiega tempo $O(\log n)$, in quanto la coda contiene al massimo n elementi. Pertanto il tempo complessivo delle operazioni `changeKey` è $O(m \log n)$. Sommando questi tempi otteniamo $O(n) + O(n \log n) + O(m \log n)$. Poiché il grafo dato è connesso, si ha $m \geq n - 1$. Pertanto il tempo totale è $O(m \log n)$, come per l'algoritmo di Kruskal.

Con una differente implementazione delle code con priorità, basata sugli *heap di Fibonacci*, si può ottenere un tempo $O(m + n \log n)$, che risulta migliore del precedente quando il numero di archi nel grafo è "alto". Per ulteriori dettagli consultate il libro di testo.

Una variante nella quale si sceglie un vertice s da cui iniziare la costruzione dell'albero, assegnandogli priorità 0, è presentata nel Codice 12.5.

[1]Poiché il grafo non è orientato, lo stesso arco potrebbe essere considerato una volta per ognuno dei suoi estremi.

Codice 12.4 Algoritmo di Prim (mediante code con priorità)

Algoritmo *Prim (grafo connesso non orientato* $G = (V, E, \omega)$*)* $\rightarrow$ *albero*
`/* Costruzione di un albero di ricoprimento di peso minimo */`
Sia C una coda con priorità inizialmente vuota
Siano $vicino[V]$ e $d[V]$ due array con insieme di indici V
foreach $v \in V$ **do**
  $d[v] \leftarrow \infty$
  $C.insert(v, \infty)$
$T \leftarrow (V_T \leftarrow \emptyset, E_T \leftarrow \emptyset)$ `// albero vuoto`
do
  $y \leftarrow C.deleteMin()$
  $V_T \leftarrow V_T \cup \{y\}$
  if $d[y] \neq \infty$ **then** `// condizione falsa solo nella prima iterazione`
    $x \leftarrow vicino[y]$
    $E_T \leftarrow E_T \cup \{(x, y)\}$
  foreach $(y, z) \in E$ **do**
    if $z \notin V_T$ **and** $\omega(y, z) < d[z]$ **then**
      $d[z] \leftarrow \omega(y, z)$
      $C.changeKey(z, \omega(y, z))$
      $vicino[z] \leftarrow y$
while $C \neq \emptyset$
return T

Codice 12.5 Algoritmo di Prim (mediante code con priorità, vertice di partenza fissato)

Algoritmo *Prim (grafo connesso non orientato* $G = (V, E, \omega)$*,* $s \in V$*)* $\rightarrow$ *albero*
`/* Costruzione di un albero di ricoprimento di peso minimo */`
Sia C una coda con priorità inizialmente vuota
Siano $vicino[V]$ e $d[V]$ due array con insieme di indici V
$d[s] \leftarrow 0$
$C.insert(s, 0)$
foreach $v \in V \setminus \{s\}$ **do**
  $d[v] \leftarrow \infty$
  $C.insert(v, \infty)$
$T \leftarrow (V_T \leftarrow \emptyset, E_T \leftarrow \emptyset)$ `// albero vuoto`
while $C \neq \emptyset$ **do**
  $y \leftarrow C.deleteMin()$
  $V_T \leftarrow V_T \cup \{y\}$
  if $y \neq s$ **then** `// condizione falsa solo nella prima iterazione`
    $x \leftarrow vicino[y]$
    $E_T \leftarrow E_T \cup \{(x, y)\}$
  foreach $(y, z) \in E$ **do**
    if $z \notin V_T$ **and** $\omega(y, z) < d[z]$ **then**
      $d[z] \leftarrow \omega(y, z)$
      $C.changeKey(z, \omega(y, z))$
      $vicino[z] \leftarrow y$
return T