Algoritmi e Strutture Dati

Grafi pesati
Minimo albero ricoprente

Basato su materiale di C. Demetrescu, I. Finocchi, G.F. Italiano

Definizioni

Sia G=(V,E) un grafo non orientato e connesso.

Un albero ricoprente di G è un sottografo T⊆G tale che:

- T è un albero;
- T contiene tutti i vertici di G.

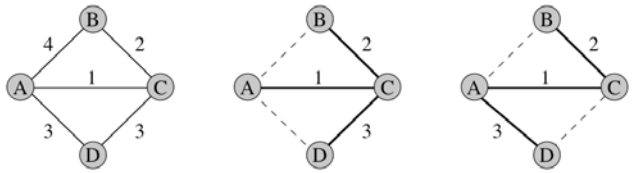
Sia w(e) il costo (o peso) di un arco e∈E.

Un minimo albero ricoprente di G è un albero ricoprente di costo minimo

- Il costo dell'albero è la somma dei costi degli archi che contiene

Esempi

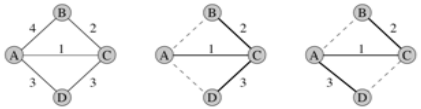
Il minimo albero ricoprente non è necessariamente unico



Proprietà dei minimi alberi ricoprenti

Tagli e cicli

- Dato un grafo non orientato G=(V,E),
  un **taglio** in G è una partizione dei vertici V in due insiemi: X e V-X.
- Un arco e=(u,v) attraversa il taglio (X,X) se u∈X e v∈X
- Un **ciclo** è un cammino in cui il primo e l'ultimo vertice coincidono



Un approccio "goloso"

- Si può costruire un minimo albero ricoprente scegliendo un arco alla volta, in modo "goloso", es:
  - includere nella soluzione archi di costo piccolo
  - escludere dalla soluzione archi di costo elevato
- Si può formalizzare come un processo di colorazione:
  - archi blu: inclusi nella soluzione
  - archi rossi: esclusi dalla soluzione

Regola del taglio (regola blu)

Scegli un taglio che non contiene archi blu.
Tra tutti gli archi non colorati del taglio, scegline uno di costo minimo e coloralo blu.

Ogni albero ricoprente, infatti, deve contenere almeno un arco del taglio

E' naturale includere quello di costo minimo

Regola del ciclo (regola rossa)

Scegli un ciclo che non contiene archi rossi.
Tra tutti gli archi non colorati del ciclo, scegline uno di costo massimo e coloralo rosso.

Ogni albero ricoprente deve escludere almeno un arco del ciclo

E' naturale escludere quello di costo massimo

L' approccio "goloso"

- Idea:
  - L'approccio goloso applica una delle due regole ad ogni passo, finché tutti gli archi sono colorati
  - Esiste sempre un minimo albero ricoprente che contiene tutti gli archi blu e nessun arco rosso.
- Diversi algoritmi
  - Differiscono in base alla scelta della regola da applicare e del taglio/ciclo usato ad ogni passo

Algoritmo di Kruskal

Algoritmo di Kruskal: versione semplificata

```
algoritmo Kruskal (grafo G) → grafo T  {
    crea un vettore nodiScelti di G.nrNodi() boolean, inizializzato a false
    inserisci in un vettore vArchi gli archi di G (triple <in, fin, peso>)
    ordina il vettore vArchi per peso crescente
    for ( i = 1; i <= vArchi.length && T.nrArchi() < G.nrNodi() -1; i++ ) {
        ArcoP ap = vArchi [i];
        // l'arco viene introdotto solo se non forma un ciclo
        if ( !scelti[ap.in] || !scelti[ap.fin] || ! esisteCammino(T,ap.in,ap.fin)) ){
            inserisci l'arco <in(p),fin(ap),peso(ap)> in T;
            scelti[in(ap)] = scelti[fin(ap)] = true;
        }
    }
    if ( T.nrArchi() == G.nrNodi() -1) return T
    else return null;
}
```


Algoritmo di Kruskal: complessità

- Per valutare la complessità dell'algoritmo di *Kruskal* consideriamo separatamente i casi di struttura a liste e a matrice di adiacenza
- Nel caso di liste di adiacenza, la complessità è $O(m^2)$
  - il vettore degli archi è costruito in tempo $O(m)$ e ordinato in $O(m \log m)$;
  - il ciclo *for* è eseguito al più m volte e, poiché la funzione più complessa, *esisteCammino*, ha complessità $O(m)$, esso è eseguito in tempo $Q(m^2)$.
- Nel caso di matrice di adiacenza, la complessità è $O(m\, n^2)$
  - il vettore degli archi è costruito in tempo $Q(n^2)$ e ordinato in $O(m \log m)$
  - il ciclo *for* è eseguito al più m volte e, poiché la funzione più complessa, *esisteCammino*, ha complessità $Q(n^2)$; quindi la complessità è $O(m\, n^2)$.
- Si può realizzare l'algoritmo di Kruskal in modo più efficiente con particolari strutture dati per la gestione efficiente di insiemi disgiunti

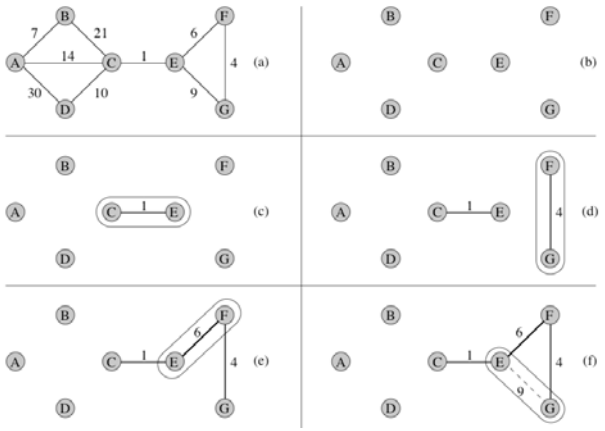
Strategia

- Mantiene una foresta di alberi blu, all'inizio tutti disgiunti
  - memorizzati in strutture union-find
- Per ogni arco, in ordine non decrescente di costo, applica il seguente passo:
  - se l'arco ha entrambi gli estremi nello stesso albero blu, applica la regola del ciclo e coloralo rosso,
  - altrimenti applica la regola del taglio e coloralo blu

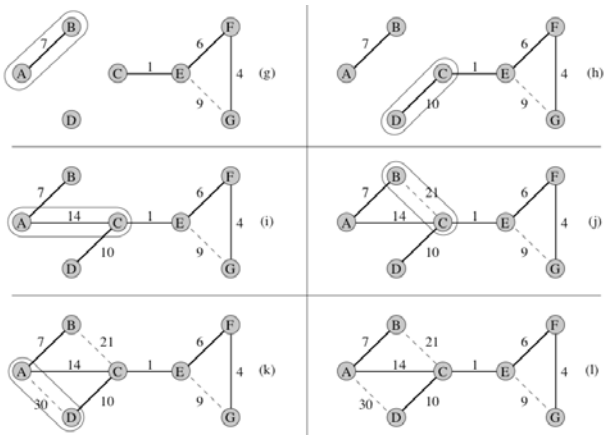
Pseudocodice

```
algoritmo Kruskal (grafo G) → albero
1.    UnionFind UF
2.    T ← albero vuoto
3.    ordina gli archi di G = (V, E) secondo costi non decrescenti
4.    for each ( vertice v in G ) do UF.makeSet(v)
5.    for each ( arco (x, y) in G in ordine non decrescente di costo ) do
6.       T_x ← UF.find(x)
7.       T_y ← UF.find(y)
8.       if ( T_x ≠ T_y ) then
9.          UF.union(T_x, T_y)
10.         aggiungi l'arco (x, y) a T
11.   return T
```


Esempio (1/2)



Esempio (2/2)



Analisi

Il tempo di esecuzione dell'algoritmo di Kruskal è O(m log n) nel caso peggiore

(Utilizzando un algoritmo di ordinamento ottimo e la struttura dati union-find)

Algoritmo di Prim

Strategia

- Mantiene un unico albero blu T, che all'inizio consiste di un vertice arbitrario
- Applica per n-1 volte il seguente passo:

  scegli un arco di costo minimo incidente su T e coloralo blu (applica la regola del taglio)
- Definiamo arco azzurro un arco (u,v) tale che $u \in T$, $v \notin T$, (u,v) ha il costo minimo tra tutti gli archi che connettono v ad un vertice in T:

  l'algoritmo mantiene gli archi azzurri in una coda con priorità da cui è estratto il minimo ad ogni passo

Pseudocodice

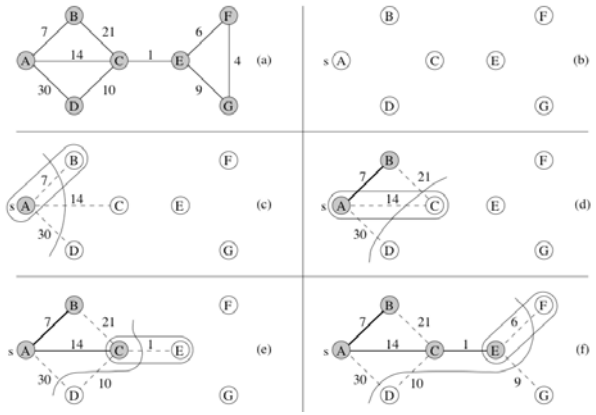
```
algoritmo Prim (grafo G) → albero
1.   for each ( vertice v in G ) do d(v) ← +∞
2.   T ← albero formato da un solo nodo s
3.   CodaPriorita S
4.   d(s) ← 0
5.   S.insert(s, 0)
6.   while ( not S.isEmpty() ) do
7.      u ← S.deleteMin()
8.      for each ( arco (u, v) in G ) do
9.         if (d(v) = +∞) then
10.           S.insert(v, w(u, v))
11.           d(v) ← w(u, v)
12.           rendi u padre di v in T
13.        else if (w(u, v) < d(v)) then
14.           S.decreaseKey(v, w(u, v))
15.           d(v) ← w(u, v)
16.           rendi u nuovo padre di v in T
17.  return T
```

Nr. max operazioni:
- *n* `insert`
- *n* `deleteMin`
- *m* `decreaseKey`

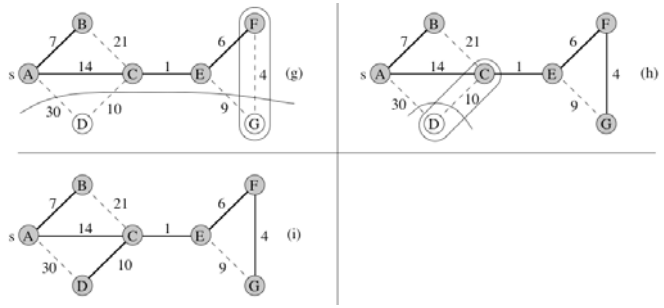
O(m log n) con heap

O(m+n log n) con heap di Fibonacci (costi amm.)

Esempio (1/2)



Esempio (2/2)



Riepilogo

- Paradigma generale per il calcolo di minimi alberi ricoprenti
- Applicazione della tecnica golosa
- Due algoritmi specifici ottenuti dal paradigma generale: Prim e Kruskal
- L'uso di strutture dati sofisticate ci ha permesso di implementare tali algoritmi in modo efficiente