# Capitolo 2.4-2.5-2.6ish

Lorenzo Vaccarecci

25 Marzo 2024

## 1 Algoritmo di Prim

**Minimo albero ricoprente** Dato un grafo $G$ connesso, non orientato e pesato, un minimo albero ricoprente di $G$ è un albero ricoprente di $G$ in cui la somma dei pesi degli archi è minima.

Un minimo albero ricoprente di $G$ è quindi un sottografo di $G$ tale che:

- sia un albero libero, ossia connesso e aciclico
- contenga tutti i nodi di $G$
- la somma dei pesi degli archi sia minima

Simile a Dijkstra, ma si prende ogni volta, fra tutti i nodi adiacenti a quelli per cui si è già trovato il minimo (neri), quello connesso a un nodo nero dall'arco di costo minimo, cioè si cerca il nodo "più vicino" all'albero già costruito (poi, come in Dijkstra, si aggiornano gli altri nodi).

```
Prim(G,s)
    for each (u nodo in G) marca u come non visitato
    for each (u nodo in G) dist[u] = inf
    parent[s] = null; dist[s] = 0
    Q = heap vuoto
    for each (u nodo in G) Q.add(u, dist[u])
    while(Q non vuota)
        u = Q.getMin()
        marca u come visitato (nero)
        for each ((u,v) arco in G)
            if(v non visitato && c_{u,v} < dist[v])
                parent[v] = u; dist[v] = c_{u,v}
                Q.changePriority(v, dist[v])
```

A differenza di quanto accade in Dijkstra, occorre un controllo esplicito che i nodi adiacenti al nodo $u$ estratto dalla coda non siano già stati visitati, perchè non è detto che per un nodo $v$ già visitato il test $c_{u,v} <$ dist[v] sia falso.

### 1.1 Correttezza

Sia $T$ l'albero di nodi neri (non in Q) corrente. L'invariante è composta di due parti:

1. $T \subseteq MST$ per qualche $MST$ minimo albero ricoprente di $G$
2. per ogni nodo $u \neq s$ in Q dist[u] = costo minimo di un arco che collegaa $u$ a un nodo nero

L'invariante vale all'inizio:

1. vale banalmente perchè $T$ è vuoto (non ci sono nodi neri)
2. vale banalmente perchè non ci sono nodi neri e la distanza è per tutti infinito, tranne che per s

L'invariante si mantiene:

1. Viene estratto dalla coda un nodo $u$ con dist[u] minima, cioè connesso a un nodo nero $y$ da un arco di costo minimo tra tutti quelli che "attraversano la frontiera", cioè uniscono un nodo nero a un nodo non nero. L'arco $(y, u)$ diventa quindi parte dell'albero di nodi neri corrente.
2. L'insieme dei nodi risulta modificato per l'aggiunta di $u$, quindi occorre ripristinare l'invarinate (2), controllando se per qualche nodo $v$ non nero e adiacente a $u$ l'arco $(u, v)$ ha costo minore del precedente arco che univa $v$ a un nodo nero, e in tal caso aggiornare l'arco e la distanza.

Postcondizione: all'uscita dal ciclo tutti i nodi sono neri, quindi $T$ connette tutti i nodi, cioè è un albero ricoprente di $G$. Per l'invariante (1), $T \subseteq MST$ per qualche $MST$ minimo albero ricoprente, quindi $T = MST$.
L'analisi della complessità è esattamente la stessa dell'algoritmo di Dijkstra, quindi $O((n+m)\log n)$.

# 2 Algoritmo di Kruskal

Anche questo algoritmo risolve il problema del minimo albero ricoprente. In particolare, il minimo albero ricoprente viene costruito mantendeno una foresta alla quale si aggiunge a ogni iterazione un nuovo arco che unisce due sottoalberi distinti. Quindi l'algoritmo di Kruskal, a differenza di quello di Prim, non costruisce l'albero a partire da un nodo scelto come radice, ma come un insieme di archi che alla fine risulta essere un grafo connesso aciclico, quindi un albero libero.

```
Kruskal(G)
    s = sequenza archi di G in ordine di costo crescente
    T = foresta formata dai nodi di G e nessun arco
    counter = 0
    while(counter < n-1) // un albero di n nodi ha n-1 archi
        estrai da s il primo elemento (u,v)
        if(u,v non connessi in T) T = T + (u,v) // aggiunta dell'arco
        counter++
    return T
```

Non si fa nessun reinserimento o variazione di ordine quindi realizzabile per esempio con un array.

## 2.1 Correttezza

Simile a Prim. Invariante: $T \subseteq MST$ per qualche $MST$ minimo albero ricoprente di $G$.
All'inizio vale banalmente perchè in $T$ non ci sono archi. Si mantiene, infatti: si sceglie un archo $(u, v)$ di costo minimo tra quelli non ancora estratti in $T$.

- Se $u$ e $v$ appartengono a uno stesso albero nella foresta $T$, aggiungendo l'arco $(u, v)$ si avrebbe un ciclo, quindi esso viene correttamente scartato, $T$ rimane uguale e l'invariante vale ancora.
- Se $u$ e $v$ appartengono a due alberi distinti nella foresta $T$, dimostriamo che aggiungendo l'arco $(u, v)$ a $T$ si ha ancora una foresta contenuta in un minimo albero ricoprente. Per assurdo supponiamo non sia così. Se in $MST$ non c'è l'arco $(u, v)$, ci deve essere un altro cammino che connette $u$ a $v$. Se eliminiamo l'arco $(x, y)$ da $MST$, otteniamo due alberi non connessi fra loro, e quindi se aggiungiamo a questi due alberi l'arco $(u, v)$ otteniamo, analogamente a Prim:
  - nuovamente un albero
  - che contiene tutti i nodi di $G$, quindi è un albero ricoprente
  - in cui la somma dei pesi degli archi è minore o uguale di quella di $MST$

Postcondizione: all'uscita dal ciclo si ha necessariamente un unico albero, perchè l'algoritmo ha esaminato tutti gli archi di $G$ unendo ogni volta due alberi di $T$ se non ancora connessi. Quindi alla fine $T$ è un unico albero, sottoalbero di un minimo albero ricoprente di $G$ contenente tutti i nodi di $G$, quindi è un minimo albero ricoprente di $G$.

## 2.2 Strutture union-find

Servono a rappresentare una collezione di insiemi disgiunti sulla quale siano possibili le seguenti operazioni:

- `makeSet(a)` aggiunge l'insieme costituito dal solo elemento `a` (singleton)
- `union(A,B)` sostituisce gli insiemi `A` e `B` con la loro unione
- `find(a)` restituisce l'insieme che contiene l'elemento `a`

Gli alberi *QuickFind* permettono di eseguire rapidamente la `find`. Sono alberi di altezza **uno**.

- `makeSet(a)` aggiunge un nuovo albero con due nodi, radice `a` e figlio `a`: $O(1)$
- `union(a,b)` rende la radice dell'albero che contiene `a` padre di tutti i nodi dell'albero che contiene `b`: $O(n)$
- `find(a)` restituisce il padre di `a`: $O(1)$

Gli alberi *QuickUnion* permettono di eseguire rapidamente la `union`. Sono alberi di altezza arbitraria.

- `makeSet(a)` aggiunge un nuovo albero con un unico nodo `a`: $O(1)$
- `union(a,b)` tra rappresentanti rende `a` padre di `b`: $O(1)$

- `union(a,b)` generica, richiede prima una `find`: $O(n)$
- `find(a)` risale la catena dei padri di `a`: $O(n)$

Per mantenere gli alberi bilanciati, si può effettuare l'unione scegliendo sempre come radice del nuovo insieme quella dell'insieme di cardinalità maggiore, cioè la radice dell'albero con meno nodi diventa figlio della radice dell'altro (*Union-by-Size*). Con la union-by-size, l'altezza di ogni albero della struttura union-find è al pià logaritmica nel numero di nodi dell'albero. SI ha quindi:

- `union` tra rappresentanti: $O(1)$
- `union` generica: $O(\log n)$
- `find`: $O(\log n)$

Una tecnica alternativa a quella dell'unione per dimensione, detta *Union-by-Rank*, è la seguente: l'unione viene effettuata scegliendo come radice del nuovo albero quella dell'albero di altezza maggiore, cioè la radice della'lbero meno alto diventa figlio della radice di quello più alto. Con la union-by-rank, l'altezza di ogni albero della struttura union-find è al più logaritmica nel numero di nodi dell'albero. *Path compression*: la `find` rende figli della radice tutti i nodi che incontra nel suo percorso di risalita dal nodo alla radice. Si noti che, dato che la `union` tra elementi generici consiste di due operazioni di `find` seguite dall'unione vera e propria, introducendo la compressione dei cammini nella `find` la introduciamo automaticamente anche nella `union`. Il vantaggio della compressione dei cammini si ha quando si effettua una sequenza di operazioni, non può quindi venire misurato dalla complessità di una singola operazione. Si considera quindi la nozione di *complessità ammortizzata*. Si può dimostrare che la complessità per sequenza di $n$ `makeSet`, $m$ `find` e $n-1$ `union` è $O(m + n \log n)$.

```
Kruskal(G)
    s = sequenza archi di G in ordine di costo crescente
    T = foresta formata dai nodi di G e nessun arco
    UF = struttura union-find vuota
    for each (u nodo in G) UF.makeSet(u)
    while(s non vuota)
        estrai da s il primo elemento (u,v)
        if(UF.union_by_need(u,v))
            /* restituisce vero ed esegue union delle radici se
            UF.find(u) != UF.find(v)
            falso altrimenti */
            T = T + (u,v)
    return T
```

Complessità dell'algoritmo: $O(m \log m)$

# 3 Ordinamento topologico

E' facile vedere che in un grafo aciclico la relazione sull'insieme dei nodi "$v$ è raggiungibile da $u$" è un ordine parziale. Un ordine totale, che "raffina" questo ordine parziale si chiama ordine topologico.

**Ordine topologico** Un ordine topologico su un grafo orientato aciclico $G = (V, E)$ è un ordine totale (stretto) $<$ su $V$ tale che, per ogni $u, v \in V$: $(u, v) \in E \Rightarrow u < v$

In un grafo orientato definiamo un nodo sorgente (**source**) se non ha archi entranti, pozzo (**sink**) s enon ha archi uscenti. E' facile vedere che in un grafo orientato aciclico esistono sempre almento un nodo pozzo e un nodo sorgente: infatti, se per assurdo così non fosse a partire da un nodo qualunque si potrebbe sempre costruire un ciclo. Per evitare di modificare il grafo e trovare in tempo costante, a ogni passo, un nodo sorgente, possiamo memorizzare per ogni nodo il suo indegree. Invece di rimuovere un arco $(u, v)$ si decrementa il valore di indegree per il nodo $v$. Quando il valore di indegree per un nodo diventa zero, lo si inserisce in un insieme di nodi sorgente da cui si estrae ogni volta il nodo successivo da inserire nell'ordine topologico.