Teoria dei grafi

Algoritmi classici

Lorenzo Ferrari, Davide Bartoli

January 12, 2023

Table of contents

Dijkstra

Problema

Single source shortest path

Dato un grafo $G = (V, E)$ **pesato**, trovare la minima distanza tra due nodi a e b, ovvero il percorso di peso totale minimo che parte da a e arriva a b.

Dijkstra

Problema

Single source shortest path

Dato un grafo $G = (V, E)$ **pesato**, trovare la minima distanza tra due nodi a e b, ovvero il percorso di peso totale minimo che parte da a e arriva a b.

Supponiamo che il grafo non sia pesato.

- potremmo usare l'algoritmo di **BFS** per trovare il percorso più corto

Dijkstra

Problema

Single source shortest path

Dato un grafo $G = (V, E)$ **pesato**, trovare la minima distanza tra due nodi a e b, ovvero il percorso di peso totale minimo che parte da a e arriva a b.

Supponiamo che il grafo non sia pesato.

- potremmo usare l'algoritmo di **BFS** per trovare il percorso più corto
- in queso caso la **BFS** funziona dato che visita i nodi in ordine di distanza crescente

Dijkstra

Problema

Single source shortest path

Dato un grafo $G = (V, E)$ **pesato**, trovare la minima distanza tra due nodi a e b, ovvero il percorso di peso totale minimo che parte da a e arriva a b.

Supponiamo che il grafo non sia pesato.

- potremmo usare l'algoritmo di **BFS** per trovare il percorso più corto
- in queso caso la **BFS** funziona dato che visita i nodi in ordine di distanza crescente

Riusciamo in qualche modo a visitare i nodi in ordine di distanza crescente anche con il grafo pesato?

Dijkstra

Problema

Vorremmo avere una struttura dati equivalente a una coda, ma che ordini i nodi in base alla distanza dal nodo sorgente.

Dijkstra

Problema

Vorremmo avere una struttura dati equivalente a una coda, ma che ordini i nodi in base alla distanza dal nodo sorgente. Per nostra fortuna questa ds esiste già all'interno della libreria standard di C++: **priority_queue**.

Dijkstra

Problema

Vorremmo avere una struttura dati equivalente a una coda, ma che ordini i nodi in base alla distanza dal nodo sorgente. Per nostra fortuna questa ds esiste già all'interno della libreria standard di C++: **priority_queue**.

Dijkstra

- inizializziamo la distanza del nodo sorgente a 0

Dijkstra

Problema

Vorremmo avere una struttura dati equivalente a una coda, ma che ordini i nodi in base alla distanza dal nodo sorgente. Per nostra fortuna questa ds esiste già all'interno della libreria standard di C++: **priority_queue**.

Dijkstra

- inizializziamo la distanza del nodo sorgente a 0
- aggiungiamo il nodo sorgente alla priority_queue

Dijkstra

Problema

Vorremmo avere una struttura dati equivalente a una coda, ma che ordini i nodi in base alla distanza dal nodo sorgente. Per nostra fortuna questa ds esiste già all'interno della libreria standard di C++: **priority_queue**.

Dijkstra

- inizializziamo la distanza del nodo sorgente a 0
- aggiungiamo il nodo sorgente alla priority_queue
- finchè la priority_queue non **è vuota**:
  - estraiamo il nodo con distanza minima dalla coda

Dijkstra

Problema

Vorremmo avere una struttura dati equivalente a una coda, ma che ordini i nodi in base alla distanza dal nodo sorgente. Per nostra fortuna questa ds esiste già all'interno della libreria standard di C++: **priority_queue**.

Dijkstra

- inizializziamo la distanza del nodo sorgente a 0
- aggiungiamo il nodo sorgente alla priority_queue
- finchè la priority_queue non **è vuota**:
  - estraiamo il nodo con distanza minima dalla coda
  - per ogni nodo adiacente a quello estratto:
    - se posso migliorare la sua distanza, la aggiorno e lo aggiungo alla priority_queue

Dijkstra

Problema

Vorremmo avere una struttura dati equivalente a una coda, ma che ordini i nodi in base alla distanza dal nodo sorgente. Per nostra fortuna questa ds esiste già all'interno della libreria standard di C++: **priority_queue**.

Dijkstra

- inizializziamo la distanza del nodo sorgente a 0
- aggiungiamo il nodo sorgente alla priority_queue
- finchè la priority_queue non **è vuota**:
  - estraiamo il nodo con distanza minima dalla coda
  - per ogni nodo adiacente a quello estratto:
    - se posso migliorare la sua distanza, la aggiorno e lo aggiungo alla priority_queue

La complessità di questo algoritmo è $O((N + M) \cdot logN)$, dato che la priority_queue ha complessità $O(logN)$ per ogni operazione.

Dijkstra

Implementazione

```cpp
vector<pair<int, int>> adj[100000];

int dijkstra(int a, int b) {
    vector<int> dist(100000, 1e9);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;
    q.push({0, a});
    dist[a] = 0;
    while (!q.empty()) {
        auto [d, cur] = q.top();
        q.pop();
        if (dist[cur] != d) continue;
        for (auto [nodo, peso] : adj[cur]) {
            if (d + peso < dist[nodo]) {
                dist[nodo] = d + peso;
                q.push({d + peso, nodo});
            }
        }
    }
    return dist[b];
}
```

Dijkstra

Implementazione

```cpp
vector<pair<int, int>> adj[100000];

int dijkstra(int a, int b) {
    vector<int> dist(100000, 1e9);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;
    q.push({0, a});
    while (!q.empty()) {
        auto [d, cur] = q.top();
        q.pop();
        if (dist[cur] != 1e9) continue;
        dist[cur] = d;
        for (auto [nodo, peso] : adj[cur]) {
            q.push({d + peso, nodo});
        }
    }
    return dist[b];
}
```

Dijkstra

Osservazione

Dijkstra non funziona se sono presenti archi con peso negativo.

Dijkstra

Osservazione

Dijkstra non funziona se sono presenti archi con peso negativo.

- se è presente un ciclo negativo, l'algoritmo può andare in loop infinito

Dijkstra

Osservazione

Dijkstra non funziona se sono presenti archi con peso negativo.

- se è presente un ciclo negativo, l'algoritmo può andare in loop infinito
- anche se non è presente un ciclo negativo, l'algoritmo può non trovare il percorso minimo o avere una complessità quadratica a seconda delle implementazioni

Disjoint Set Union

Problema

Contare le componenti connesse online

Dati $n \leq 10^6$ nodi e $m \leq 10^6$ update che aggiungono un arco (a_i, b_i), dire dopo ogni update quante componenti connesse ci sono.

Disjoint Set Union

Problema

Contare le componenti connesse online

Dati $n \leq 10^6$ nodi e $m \leq 10^6$ update che aggiungono un arco (a_i, b_i), dire dopo ogni update quante componenti connesse ci sono.

- un'opzione è fare M visite ognuna in $O(N + M)$, ma quest'approccio prenderebbe sicuramente TLE (Time Limit Exceeded)

Disjoint Set Union

Problema

Contare le componenti connesse online

Dati $n \leq 10^6$ nodi e $m \leq 10^6$ update che aggiungono un arco (a_i, b_i), dire dopo ogni update quante componenti connesse ci sono.

- un'opzione è fare M visite ognuna in $O(N + M)$, ma quest'approccio prenderebbe sicuramente TLE (Time Limit Exceeded)
- al momento non sappiamo risolverlo efficientemente

Disjoint Set Union

Problema

Contare le componenti connesse online

Dati $n \leq 10^6$ nodi e $m \leq 10^6$ update che aggiungono un arco (a_i, b_i), dire dopo ogni update quante componenti connesse ci sono.

- un'opzione è fare M visite ognuna in $O(N + M)$, ma quest'approccio prenderebbe sicuramente TLE (Time Limit Exceeded)
- al momento non sappiamo risolverlo efficientemente
- servirebbe una struttura dati che rappresenti ogni componente connessa e che sia in grado di unire due componenti connesse efficientemente

Disjoint Set Union

Idea

La struttura dati **Disjoint Set Union** (spesso chiamata anche **Union Find**) fa esattamente questo. In particolare, implementeremo le seguenti operazioni:

Disjoint Set Union

Idea

La struttura dati **Disjoint Set Union** (spesso chiamata anche **Union Find**) fa esattamente questo. In particolare, implementeremo le seguenti operazioni:

- `find(v)`, trova un nodo nella componente connessa di `v`

Disjoint Set Union

Idea

La struttura dati **Disjoint Set Union** (spesso chiamata anche **Union Find**) fa esattamente questo. In particolare, implementeremo le seguenti operazioni:

- `find(v)`, trova un nodo nella componente connessa di `v`
  - vogliamo che `find(v)` ritorni lo stesso nodo per tutti i `v` appartenenti alla stessa componente connessa
  - per il *rappresentante* di una componente connessa vale `find(v) == v`

Disjoint Set Union

Idea

La struttura dati **Disjoint Set Union** (spesso chiamata anche **Union Find**) fa esattamente questo. In particolare, implementeremo le seguenti operazioni:

- `find(v)`, trova un nodo nella componente connessa di `v`
  - vogliamo che `find(v)` ritorni lo stesso nodo per tutti i `v` appartenenti alla stessa componente connessa
  - per il *rappresentante* di una componente connessa vale `find(v) == v`
- `merge(a, b)`, aggiungi l'arco `(a, b)`

Disjoint Set Union

Idea

La struttura dati **Disjoint Set Union** (spesso chiamata anche **Union Find**) fa esattamente questo. In particolare, implementeremo le seguenti operazioni:

- `find(v)`, trova un nodo nella componente connessa di `v`
  - vogliamo che `find(v)` ritorni lo stesso nodo per tutti i `v` appartenenti alla stessa componente connessa
  - per il *rappresentante* di una componente connessa vale `find(v) == v`
- `merge(a, b)`, aggiungi l'arco `(a, b)`

La connettività non cambia se, invece dell'arco `(a, b)`, si aggiunge l'arco `(find(a), find(b))`

Disjoint Set Union

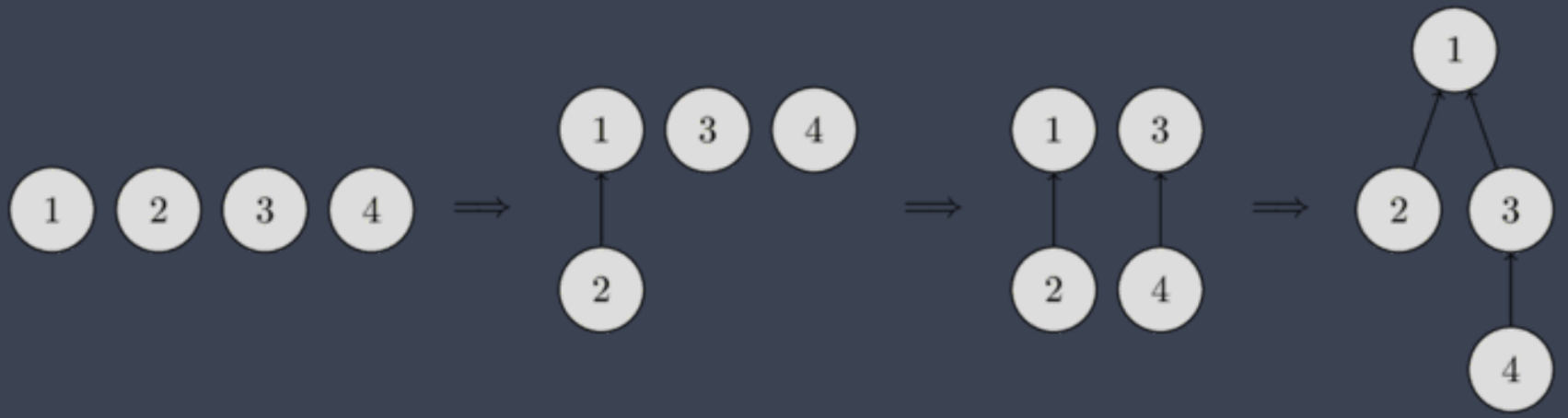
Implementazione

- manteniamo una foresta, in cui ogni albero corrisponde a una componente connessa
- la radice dell'albero è il nodo rappresentante della componente
- inizialmente ci sono n alberi ognuno composto da un unico nodo
- per ogni nodo `v`, salviamo il suo parent `par[v]`[1]

[1] per la radice vale `par[v] == v`

Disjoint Set Union

Struttura



```cpp
void init() {
    for (int i = 0; i < n; ++i) {
        par[i] = i;
    }
}

int find(int v) {
    return par[v] == v ? v : find(par[v]);
}

void merge(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        par[b] = a;
    }
}
```

Disjoint Set Union

Complessità

La struttura dati funziona, ma è abbastanza efficiente?

Disjoint Set Union

Complessità

La struttura dati funziona, ma è abbastanza efficiente?

Complessità

- la complessità di tutte le operazioni dipende dall'altezza h dell'albero

Disjoint Set Union

Complessità

La struttura dati funziona, ma è abbastanza efficiente?

Complessità

- la complessità di tutte le operazioni dipende dall'altezza h dell'albero

Si può dimostrare che, se gli archi sono inseriti in ordine casuale, l'altezza massima attesa è $O(\log n)$.

Disjoint Set Union

Complessità

La struttura dati funziona, ma è abbastanza efficiente?

Complessità

- la complessità di tutte le operazioni dipende dall'altezza h dell'albero

Si può dimostrare che, se gli archi sono inseriti in ordine casuale, l'altezza massima attesa è $O(\log n)$. Tuttavia esiste il caso pessimo in cui l'albero è una catena: in quel caso ogni operazione costa $O(h) = O(n)$.

Disjoint Set Union

Complessità

La struttura dati funziona, ma è abbastanza efficiente?

Complessità

- la complessità di tutte le operazioni dipende dall'altezza h dell'albero

Si può dimostrare che, se gli archi sono inseriti in ordine casuale, l'altezza massima attesa è $O(\log n)$. Tuttavia esiste il caso pessimo in cui l'albero è una catena: in quel caso ogni operazione costa $O(h) = O(n)$.
Si può fare meglio di così?

Disjoint Set Union

Path compression

Idea

se in qualunque momento cambiamo `par[v]` in `par[v] = find(v)`, la connettività non cambia, ma le prossime chiamate a `find(v)` saranno più veloci.

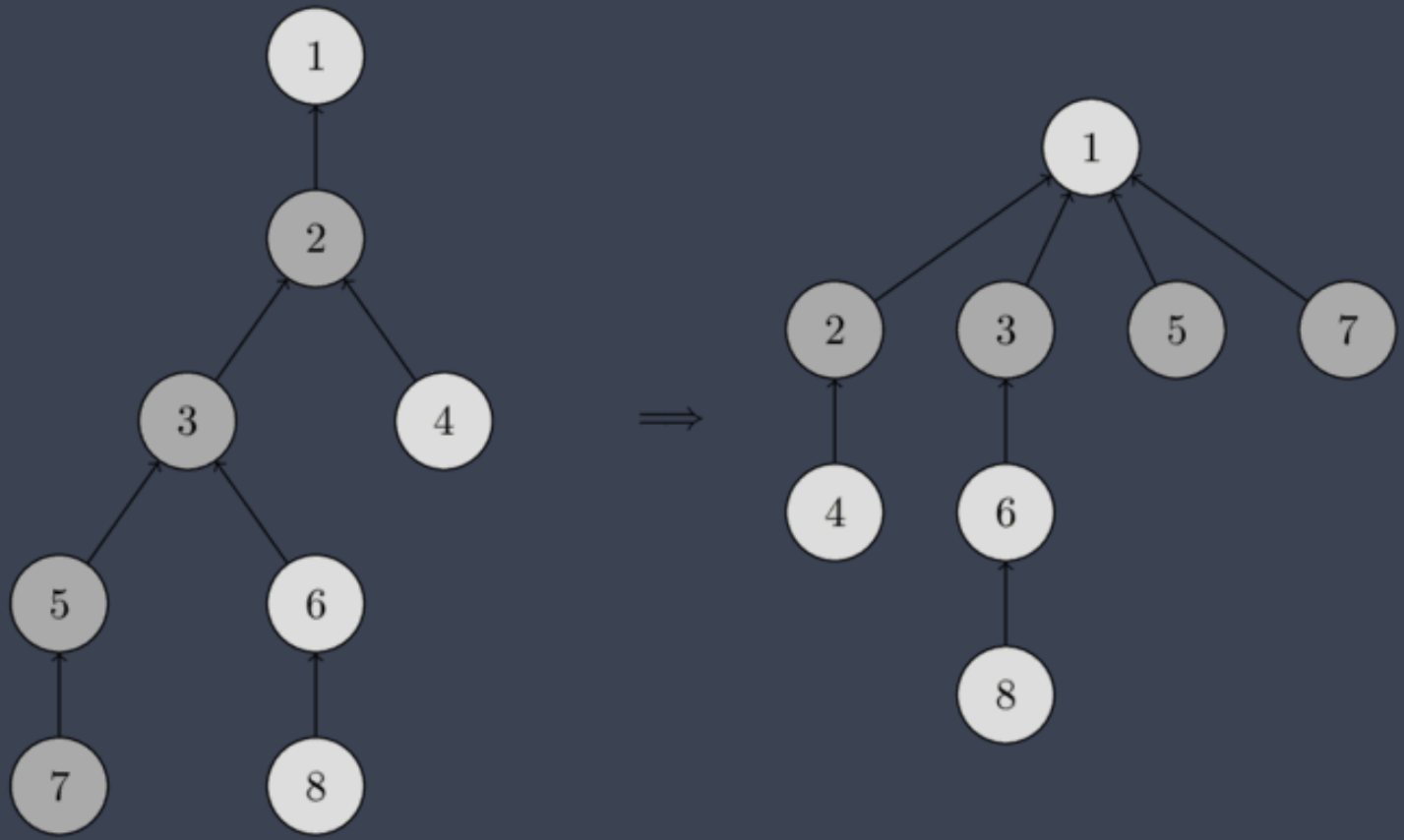
Ad ogni chiamata di `find(v)`, aggiorniamo il parent anche di tutti i suoi antenati.

Con quest'ottimizzazione la complessità ammortizzata[2] diventa $O(\log n)$

[2] complessità ammortizzata $O(f(x))$ significa che la somma di m operazioni è $O(mf(x))$

Disjoint Set Union

Path compression



Disjoint Set Union

Implementazione

```
int find(int v) {
    return par[v] == v ? v : par[v] = find(par[v]);
}
```

Disjoint Set Union

Ottimizzazioni

Se uniamo sempre l'albero più "piccolo" in quello più "grande", si dimostra che l'altezza massima di un albero con n nodi è al più $\log n$.

Union by rank

- salviamo in ogni radice l'altezza `rank[v]` dell'albero
- il rank aumenta solo quando mergiamo due alberi dello stesso rank
- per induzione si dimostra che un albero con `rank[v]` $= h$, ha almeno 2^h nodi
- $h = O(\log n)$

Disjoint Set Union

Ottimizzazioni

Union by size

- salviamo in ogni radice il numero di nodi `size[v]` nel suo albero
- dopo ogni operazione `merge`, il nuovo albero ha almeno il doppio dei nodi dell'albero più piccolo
- l'albero di un nodo può essere appeso a un nuovo albero al più $\log n$ volte
- $h = O(\log n)$

Disjoint Set Union

Ottimizzazioni

Se usiamo sia path compression che union by rank, la complessità ammortizzata scende a $O(\alpha(n))$ per operazione; $\alpha(n)$ è la funzione inversa di Ackermann e cresce molto lentamente[3] e a fini pratici può essere considerata come un fattore costante.

[3] davvero molto lentamente, possiamo assumere $\alpha(n) \leq 4$

Disjoint Set Union

Ottimizzazioni

Se usiamo sia path compression che union by rank, la complessità ammortizzata scende a $O(\alpha(n))$ per operazione; $\alpha(n)$ è la funzione inversa di Ackermann e cresce molto lentamente[3] e a fini pratici può essere considerata come un fattore costante.

In problemi particolari potrebbe essere impossibile applicare la path compression. Con union by rank/size si può comunque raggiungere complessità $O(\log n)$ per operazione.

[3] davvero molto lentamente, possiamo assumere $\alpha(n) \leq 4$

Minimo albero ricoprente

Problema

Problema

Dato un grafo pesato di $n \leq 2 \cdot 10^5$ nodi e $m \leq 5 \cdot 10^5$ archi (a_i, b_i, w_i), trovare la somma minima di un set di archi che permetta di raggiungere ogni nodo da ogni altro nodo.

Minimo albero ricoprente

Problema

Problema

Dato un grafo pesato di $n \leq 2 \cdot 10^5$ nodi e $m \leq 5 \cdot 10^5$ archi (a_i, b_i, w_i), trovare la somma minima di un set di archi che permetta di raggiungere ogni nodo da ogni altro nodo.

- non è mai ottimale avere un ciclo: posso risparmiarmi un arco e rendere gli stessi nodi connesso
  - la struttura è un albero, detto **minimo albero ricoprente** (o *minimum spanning tree, mst*)
- dato un albero ricoprente, se in uno stesso ciclo prendo un arco e ne escludo uno di peso minore, allora la soluzione non è ottimale

Minimo albero ricoprente

Problema

Problema

Dato un grafo pesato di $n \leq 2 \cdot 10^5$ nodi e $m \leq 5 \cdot 10^5$ archi (a_i, b_i, w_i), trovare la somma minima di un set di archi che permetta di raggiungere ogni nodo da ogni altro nodo.

- non è mai ottimale avere un ciclo: posso risparmiarmi un arco e rendere gli stessi nodi connesso
  - la struttura è un albero, detto **minimo albero ricoprente** (o *minimum spanning tree, mst*)
- dato un albero ricoprente, se in uno stesso ciclo prendo un arco e ne escludo uno di peso minore, allora la soluzione non è ottimale

Date queste premesse, possiamo convincerci della correttezza dell'**Algoritmo di Kruskal** per trovare il minimo albero ricoprente.

Algoritmo di Kruskal

Algoritmo di Kruskal

- ordino gli archi in ordine di peso crescente
- per ogni arco:
  - se gli estremi appartengono a due componenti connesse diverse, lo aggiungo alla soluzione
  - altrimenti non faccio nulla

Algoritmo di Kruskal

Algoritmo di Kruskal

- ordino gli archi in ordine di peso crescente
- per ogni arco:
  - se gli estremi appartengono a due componenti connesse diverse, lo aggiungo alla soluzione
  - altrimenti non faccio nulla

Con una Dsu possiamo controllare efficientemente se due nodi appartengono alla stessa componente connessa!

Algoritmo di Kruskal

Algoritmo di Kruskal

- ordino gli archi in ordine di peso crescente
- per ogni arco:
  - se gli estremi appartengono a due componenti connesse diverse, lo aggiungo alla soluzione
  - altrimenti non faccio nulla

Con una Dsu possiamo controllare efficientemente se due nodi appartengono alla stessa componente connessa!

Complessità totale: $O(m \log m)$

- sort: $O(m \log m)$
- Dsu: $O(m\alpha(n)) \approx O(m)$

Algoritmo di Prim

Algoritmo di Prim

- scelgo un nodo come radice dell'mst
- per $n - 1$ volte, aggiungo alla soluzione l'arco minimo che collega un nodo dell'mst a un nodo non ancora nell'mst
  - si implementa tenendo un array `dist` dove `dist[v]` è la distanza di un nodo dall'mst
  - ad ogni iterazione aggiungo il nodo `v` che minimizza `dist[v]`

[4] grafi con $m \approx n^2$

Algoritmo di Prim

Algoritmo di Prim

- scelgo un nodo come radice dell'mst
- per $n - 1$ volte, aggiungo alla soluzione l'arco minimo che collega un nodo dell'mst a un nodo non ancora nell'mst
  - si implementa tenendo un array `dist` dove `dist[v]` è la distanza di un nodo dall'mst
  - ad ogni iterazione aggiungo il nodo `v` che minimizza `dist[v]`

L'algoritmo di Prim classico si implementa con una coda di priorità con una coda di priorità in $O(m \log n)$.

Per grafi densi[4] è preferibile l'implementazione in $O(n^2)$, più veloce anche di Kruskal ($O(n^2)$ contro $O(n^2 \log n^2)$).

[4] grafi con $m \approx n^2$

Domande?

- Dijkstra
  - Problema
  - Implementazione
- Disjoint Set Union
  - Problema
  - Implementazione
  - Ottimizzazioni
- Minimo albero ricoprente
  - Problema
  - Algoritmo di Kruskal
  - Algoritmo di Prim

Problemi

https://cses.fi/problemset/task/1666

https://cses.fi/problemset/task/1671

https://cses.fi/problemset/task/1195

https://training.olinfo.it/#/task/mincammino/statement

https://training.olinfo.it/#/task/ois_railroad/statement

https://training.olinfo.it/#/task/ois_rainstorm/statement

https://training.olinfo.it/#/task/tai_mle/statement

https://training.olinfo.it/#/task/ois_dna/statement