Progettazione Algoritmi

Capitolo 2: Grafi

2.1 - Introduzione

Un grafo è una coppia di insiemi $G = (V, E)$, dove V è l'insieme dei nodi ed E è l'insieme degli archi. Gli archi possono aver associato un valore, chiamato peso. Un grafo può essere:

1. Direzionato
2. Non direzionato

Un grafo non direzionato può essere visto come un particolare grafo direzionato, dove per ogni coppia di nodi adiacenti (u, v) esistono due archi, uno che esce da u e entra in v e uno che esce da v e entra in u.

2.2 - Proprietà dei grafi

Consideriamo un grafo G con due nodi u e v connessi dall'arco $e = (u, v)$:

1. u e v sono adiacenti.
2. u e v sono le estremità dell'arco e.
3. l'arco e incide sui nodi u e v.
4. u è un nodo vicino di v.
5. v è un nodo vicino di u.

Dato un nodo u chiamano grado di u: il numero di archi incidenti su u e lo indichiamo con $deg(u)$. La somma di tutti i gradi dei nodi di G è $\sum_{u \in V} deg(u) = 2m$. Il numero m di archi in un grafo G non direzionato è al più $n(n-1)/2$, mentre in un grafo direzionato G il numero m di archi è al più n^2

2.3 - Percorsi, connettività e cicli

Un percorso in un grafo non direzionato $G = (V, E)$ è una sequenza P di nodi $v_1, v_2, \ldots, v_k$ con la proprietà che ciascuna coppia di vertici consecutivi v_i, v_{i+1} sono uniti da un arco. Un percorso si definisce semplice se tutti i nodi sono distinti.

Un grafo non direzionato si dice connesso se per ogni coppia di nodi u e v, esiste un percorso tra u e v.

Un ciclo è un percorso $v_1, v_2, \ldots, v_k$ in cui $v_1 = v_k$ con $k > 2$. Un ciclo si definisce semplice se i primi $k-1$ nodi del ciclo sono tutti distinti tra loro

2.4 - Breath First Search (visita in ampiezza)

Nella BFS partiamo da un nodo sorgente s ed esploriamo i nodi in ordine di distanza dalla sorgente, si dice che l'esplorazione avviene per livelli. L'esecuzione dell'algoritmo produce un albero chiamato albero BFS. L'albero BFS ha come radice il nodo sorgente s e come figli tutti i nodi raggiungibili da quest'ultimo. Nella costruzione dell'albero BFS l'algoritmo andrà

ad inserire solo gli archi, questo perchè gli archi danno informazioni di relazione PADRE-FIGLIO.

2.5 - Implementazione con lista di adiacenza

Implementazione con lista di adiacenza:

- Il grafo è rappresentato con liste di adiacenza;
- Ciascun livello L_i è rappresentato da una lista `L[i]`;
- Utilizziamo un array `Discovered[n]` di valori booleani per associare a ciascun nodo il valore vero o falso a seconda se sia stato scoperto o meno;
- Durante l'esecuzione costruiamo un albero BFS;

```
BFS(s){
    Poni Discovered[s]=true
    Poni Discovered[u]=false per tutti gli altri nodi u del grafo
    inizializza L[0] in modo che contenga solo s
    Poni il contatore dei livelli i = 0
    inizializza bfsTree come albero vuoto.
    while(i < n-2){
        inizializza L[i+1] come lista vuota
        foreach (nodo in L[i]){
            foreach (arco(u,v) incidente in u){
                if (Discovered[v]==false){
                    Discovered[v]=true
                    Aggiungi v alla lista L[i+1]
                    Aggiungi l'arco (u, v) all'albero bfsTree
                }
            }
        }
        i = i + 1
    }
}
```

Analisi

L'inizializzazione prende tempo costante tranne per quanto riguarda quella dell'array `Discovered[n]` che prende come tempo O(n) perchè dobbiamo "analizzare" tutti gli n nodi. il while viene eseguito in tempo O(n), mentre il foreach esterno, sul totale delle iterazioni del while, visiterà tutti i nodi, in quanto ogni nodo appartiene ad una sola lista L di grandezza n, quindi O(n). Infine il foreach più interno, verrà eseguito al più $\sum_{u \in V} deg(u) \leq 2m$, quindi O(m), perchè l'algoritmo andrà a visitare tutti gli archi incidenti su u. Totale O(n+m).

2.6 - Implementazione con coda FIFO

Implementazione con coda FIFO:

- Ogni volta che un nodo u viene scoperto, viene inserito in una coda di tipo FIFO;
- Nella coda possono esserci solo nodi che appartengono a livelli contigui;

- Durante l'esecuzione costruiamo un albero BFS;

```
BFS(s){
    Inizializza Q come coda vuota
    Inizializza bfsTree come albero vuoto.
    Poni Discovered[s]=true
    Poni Discovered[u]=false per tutti gli altri nodi u del grafo
    Inserisci s in coda a Q con una enqueue()
    while(Q non è vuota){
        estrai il front con una deque() e ponilo in u
        foreach (arco(u,v) incidente in u){
            if (Discovered[v]==false){
                    Discovered[v]=true
                    Aggiungi l'arco (u, v) all'albero bfsTree
                    Aggiungi v in coda a Q con una enqueue()
            }
        }
    }
}
```

Analisi

L'inizializzazione prende tempo costante tranne per quanto riguarda quella dell'array `Discovered[n]` che prende tempo O(n). Per quanto riguarda il while prende tempo O(n) in quanto nella coda transiteranno tutti i nodi del grafo, perchè ogni volta che scopriamo un nodo questo verrà inserito nella coda, mentre il foreach verrà eseguito al più

$\sum_{u \in V} deg(u) \leq 2m$, quindi O(m), perchè l'algoritmo andrà a visitare tutti gli archi incidenti su u.

Totale O(n+m).

2.7 - Depth First Search (visita in profondità)

La visita DFS parte dalla sorgente s e si spinge in profondità fino a che non è più possibile raggiungere nuovi nodi. La visita DFS simula il comportamento di esplorazione di un labirinto. L'algoritmo DFS produce un albero che ha come radice la sorgente s e come nodi tutti i nodi del grafo raggiungibili da s.

2.8 - Implementazione Ricorsiva

Implementazione Ricorsiva:

- Assumiamo G implementato con liste di adiacenza;
- R indica l'insieme dei vertici raggiunti/scoperti
- Lo stack viene utilizzato in maniera implicita andando ad utilizzare la ricorsione;

```
DFS(s){
    marchia u come "esplorato"
    aggiungi u ad R
    foreach( arco (u,v) incidente su u){
```

```
        if( v non è "esplorato")
            richiama ricorsivamente DFS(v);
    }
}
```

Analisi

Ciascuna visita ricorsiva richiede tempo O(1 + deg(u)):

- O(1) per marcare u e aggiungerlo ad R.
- O(deg(u)) per eseguire il foreach.

Se inizialmente invochiamo DFS su un nodo s, allora DFS viene invocata ricorsivamente su tutti i nodi raggiungibili a partire da s. Il costo totale è quindi al più:

$$\sum_{u \in V} O(1 + deg(u)) = \sum_{u \in V} O(1) + \sum_{u \in V} deg(u)) = O(n + m)$$

2.9 - Implementazione con Stack O(n+m)

Implementazione con Stack:

- Assumiamo G implementato con liste di adiacenza
- Lo stack in questo caso è usato in modo esplicito

```
DFS(s){
    poni Explored[s]=true e Explored[u]=false per tutti gli altri nodi u
    inizializza S con uno stack contenente la sorgente s
    Inizializza il DFStree come albero vuoto
    while(S non è vuoto){
        metti in u il nodo al top di S
        if (esiste un arco(u,v) incidente su u non ancora esplorato){
            if (Explored[v]==false){
                Explored[v]=true
                aggiungi l'arco (u, v) all'albero DFStree
                inserisci v al top di S
            }
        } else rimuovi il top di S
    }
}
```

Analisi

L'inizializzazione prende tempo costante tranne per quanto riguarda quella dell'array `Explored[n]` che impiega tempo O(n) pechè deve "marcare" tutti i nodi di G. Per quanto riguarda il while non è corretto ipotizzare che il suo tempo di esecuzione sia al più O(n) perchè potrebbe esserci più di un'interazione dove non verrà rimosso u dallo Stack S, ogni nodo verrà rimosso dal top dello stack se e solo se non ci sono altri archi incidenti su quel nodo da esplorare. Possiamo dire che ogni nodo all'interno dello stack è responsabile di un numero di iterazioni pari al suo grado. Risulta che in totale il while è iterato al più 2m volte, quindi O(m). Tempo O(n+m).

2.10 - Extra DFS:

Utilizzando una rappresentazione del grafo con liste adiacenti è possibile mantenere per ogni vertice u un puntatore al nodo della lista di adiacenza di u corrispondente al prossimo arco (u, v) da scandire. Questo ci porta a migliorare il tempo di esecuzione in quanto il while impiegherà tempo costante per ogni singola iterazione. Il tempo totale per tutte le iterazioni è quindi O(m). Questa versione simula il comportamento dell'algoritmo ricorsivo.

2.11 - Componente Connessa

Chiamiamo componente connessa un sottoinsieme di vertici tale che per ciascuna coppia di vertici u e v esiste un percorso tra u e v.

Teorema

Per ogni due nodi s e t di un grafo, le loro componenti connesse o sono uguali o disgiunte.

Dimostrazione

Banalmente se esiste un percorso tra s e t, ogni nodo u raggiungibile da s è anche raggiungibile da t e ogni nodo u raggiungibile da t è raggiungibile da s. Viceversa se non esiste un percorso tra s e t non può esserci un nodo che appartiene sia alla componente connessa di s che a quella di t.

2.12 - Pseudocodice Componente Connessa

La BFS va modificato in modo da lasciare la fase di inizializzazione di discovered a AllComponents.

```
AllComponents(G){
    per ogni nodo u in G Discovered[u] = false
    foreach (nodo u in G){
        BFS(u)
    }
}
```

Analisi

Indicando con n_i e m_i il numero di nodi e di archi dell'i-esima componente connessa il tempo necessario per ogni componente connessa sarà di O($n_i + m_i$) quindi il tempo totale sarà dato dalla $\sum_{i=0}^{k} O(n_i + m_i)$, ma siccome le componenti connesse sono a due a due disgiunte questa sommatoria sarà uguale a O(n+m). Risulta che il tempo totale dell'algoritmo è: O(n+m)

2.13 - Extra Componente Connessa

Algoritmo che trova le componenti connesse in un grafo, associando i nodi alla componente connessa di cui fanno parte:

- Utilizziamo un array monodimensionale `Component[n]` inizializzato a 0 per tutti i nodi u ∈ V.

- Modifichiamo la BFS in modo che accetti il numero della componente connessa che si sta "esplorando".
- Inizializziamo j a 1 (vogliamo trovare la prima componente connessa).
- Nella BFS ogni volta che viene scoperto un nuovo nodo v aggiorniamo Component[v] = j, AllCompConnessa restituisce l'array Component.
- Il tempo di esecuzione rimane O(n + m), in quanto le operazioni aggiunte non modificano il tempo di esecuzione

```
AllCompConnessa(G){
    per ogni nodo u in G Discovered[u] = false
    inizializza Component[u] = 0 per ogni u in G
    j = 1
    foreach (nodo u in G){
        BFS(u, j)
        j = j + 1
    }
    return Component
}

BFS(u, j){
    inizializza L[0] in modo che contenga solo u
    Poni il contatore dei livelli i = 0
    while(i<n-2){
        inizializza L[i+1] come lista vuota
        foreach (nodo in L[i]){
            foreach (arco(u,v) incidente in u){
                if (Discovered[v]==false){
                    Discovered[v]=true
                    Aggiungi v alla lista L[i+1]
                    Component[v] = j
                }
            }
        }
        i = i + 1
    }
}
```

2.14 - Grafo Bipartito

Un grafo bipartito è un grafo non direzionato tale che l'insieme dei nodi può essere partizionato in due sottoinsiemi X e Y tali che ciascun arco del grafo ha una delle due estremità in X e l'altra in Y.

Lemma

Se un grafo G è bipartito, non può contenere cicli dispari, cioè un ciclo formato da un numero di nodi dispari.

Dimostrazione

Graficamente in un ciclo dispari non è possibile colorare, con due colori, in modo alternato i nodi che lo compongono.

Osservazione: sia G un grafo connesso e siano $L_0, \ldots, L_k$ i livelli prodotti dall'esecuzione della visita BFS a partire dal nodo s:

1. Se nessun arco di G collega due nodi sullo stesso livello allora è un grafo bipartito.
2. Se un arco G collega due nodi sullo stesso livello allora e un Grafo non bipartito.

Caso 1 dimostrazione

In questo caso non ci sono archi tra nodi dello stesso livello, allora tutti gli archi del grafo collegano nodi in livelli successivi. Possiamo colorare di rosso i livelli di indice pari, in questo modo ho che le estremità di ogni arco sono di colore diverso.

Caso 2 dimostrazione

In questo caso ci basta dimostrare che esiste un ciclo dispari nel grafo. Supponiamo che esista un arco(x, y) $\in L_j$. Indichiamo con z l'antenato comune più prossimo a x e y. Sia L_i il livello dove si trova z. Possiamo ottenere un ciclo dispari del grafo prendendo il percorso d-z-a-x (j-i archi), quello da z a y (j-i archi) e l'arco (x, y), in totale il ciclo contiene 2(j - i) + 1 archi.

2.15 - Pseudocodice Componente Connessa

Algoritmo che determina se un grafo è Bipartito:

- Modifichiamo la BFS in modo da poter assegnare i due colori ed eseguiamo al termine della visita una scansione sugli archi
- Il grafo G è implementato con liste di adiacenza

```
BFS(s){
    Poni Discovered[s]=true
    Poni Discovered[u]=false per tutti gli altri nodi u del grafo
    inizializza L[0] in modo che contenga solo s
    Poni il contatore dei livelli i = 0
    inizializza bfsTree come albero vuoto.
    while(i<n-2){
        inizializza L[i+1]
        foreach (nodo u in L[i]){
            foreach (arco(u,v) incidente su v){
                if(Discovered[v] == false){
                    Discovered[v] == true;
                    Aggiungi v alla lista L[i+1]
                    Aggiungi l'arco (u, v) all'albero bfsTree
                    if((i+1) % 2 == 0){
                        Color[v] = rosso
                    }else{
                        Color[v] = blu
                    }
```

```
                }
            }
        }
        i=i+1
    }
    return IsBipart(Color, bfsTree)
}

IsBipart(Color, bfsTree){
    foreach (arco (u, v) in bfsTree){             //Tempo O(m)
        if(color[u] == color[v]) return false
    }

    return true
}
```

Analisi

L'aggiunta delle modifiche non cambia il tempo di esecuzione. L'assegnazione del colore impiega tempo costante O(1) per ogni nodo, quindi al più O(n). L'algoritmo che analizza tutti gli archi per controllare se due estremi hanno lo stesso colore impiega al più O(m). Risulta che il tempo totale è O(n+m)

2.16 - Grafi Direzionati

Due nodi u e v si dicono mutuamente raggiungibili se c'è un percorso da u a v e un percorso tra v e u. Un grafo in cui ogni coppia di nodi è mutuamente raggiungibile è definito fortemente connesso.

2.17 - Pseudocodice Grafo Fortemente Connesso

Dato un grafo G, si esegue la visita BFS su un nodo s di G, successivamente
si invertono tutti gli archi di G e si ripete la visita BFS su s, se i due array disc sono uguali allora G è fortemente connesso.

```
IsStrongConnect(s, G){
    discbfs_one = BFS(s, G)
    G_inv = inverti tutti gli archi del grafo G
    discbfs_two = BFS(s, G_inv)
    for(i=0, i < n; i++){
        if(discbfs_one[i] != discbfs_two[i]){
            return false
        }
    }
    return true
}
```

Il tempo di esecuzione risulta essere O(n+m).

2.18 - Grafi Direzionati Aciclici (DAG)

Un DAG è un grafo direzionato che non contiene cicli direzionati.

Ordinamento Topologico: preso un grafo G = (V, E), un ordinamento topologico è un ordinamento dei suoi nodi, tale che se c'è l'arco (u,v) in G, allora il nodo u precede il nodo v nell' ordinamento. In sostanza è un etichettatura dei nodi v_1, v_2, ..., v_n tale che se esiste l'arco(v_i, v_j) si ha che i < j. Tutti gli archi puntano in avanti nell'ordinamento. Un ordinamento topologico esiste se e solo se il grafo e aciclico.

Lemma 1
Se un grafo direzionato G ha un ordinamento topologico allora è un DAG.

Dimostrazione (per assurdo)
Supponiamo che G sia un grafo direzionato con un ordinamento v_1, v_2, ..., v_n, supponiamo per assurdo che G non sia un DAG e quindi abbia un ciclo direzionato C. Prendiamo il nodo $v_i \in C$ con indice più piccolo e prendiamo v_j quello precedente a v_i (all'interno del ciclo), questo implica che esiste un arco(v_j, v_i), quindi visto che esiste un ordinamento topologico per ipotesi, j deve essere minore di i. Ma questo è impossibile, perchè abbiamo preso i come il vertice di indice più piccolo nel ciclo.

Lemma 2
Se G è un DAG allora G ha un nodo senza archi entranti.

Dimostrazione (per assurdo)
Supponiamo che G sia un DAG e ogni nodo di G abbia almeno un arco entrante. Ora scorriamo tutti gli archi di G al ritroso per n o più volte. Così facendo attraversiamo almeno n archi i e n+1 vertici ciò vuol dire che incontriamo un nodo almeno due volte, questo implica che all'interno di G deve esserci un ciclo, quindi G non è un DAG per definizione, ma questo è assurdo perché per ipotesi abbiamo affermato che G è un DAG.

Lemma 3
Se G è un DAG, G ha un ordinamento topologico.

Dimostrazione (per induzione su n):
Passo Base: se n = 1 è banalmente vero (un nodo nell'ordinamento).
Passo induttivo: supponiamo che il lemma sia vero per n>=1. Preso un DAG con n+1>1 nodi, prendiamo un nodo senza archi entranti (sappiamo che esiste). Se cancelliamo v da grafo in modo da ottenere il grafo G-{v}, sappiamo che G-{v} rimane un DAG perché l'eliminazione di v non introduce cicli nel grafo. Se G-{v} è un DAG allora per ipotesi induttiva G-{v} ha un ordinamento topologico. Consideriamo di aggiungere v all'inizio dell'ordinamento aggiungendo tutti i G-{v} nodi nell'ordine in cui appaiono, sappiamo che v non ha archi entranti e i suoi archi uscenti ora puntano ai nodi di G-{v}. Quello che abbiamo ottenuto è un ordinamento topologico.

2.19 - Pseudocodice Ordinamento Topologico

Algoritmo permette di trovare ricorsivamente un ordinamento topologico di un DAG

```
TopologicalOrder(G){
    if (esiste un nodo u senza archi entranti)
        elimina v dal grafo G in modo da creare G-{v}
        L = TopologicalOrder(G-{v})
        aggiungi v all'inizio di L
        return L
    }else{
        return lista vuota
    }
}
```

Analisi

Trovare un nodo senza archi entranti, richiede O(n) se per ogni nodo memorizziamo il numero di archi entranti. Cancellare un nodo del grafo invece richiede tempo proporzionale al numero di archi del nodo quindi deg(u). Considerando tutte le chiamate ricorsive il tempo è $O(n^2)$ per l'if e O(m) per l'eliminazione. Totale: $O(n^2)$.

2.20 - Miglioramento con tempo O(n+m)

Algoritmo che ritorna l'ordinamento topologico di un DAG in tempo O(n+m), utilizza un array Count per contare il numero di archi entranti in ogni nodo e una lista S che contiene tutti i nodi senza archi entranti.

```
//Inizializzazione, questa fase è effettuata fuori dalla procedura
foreach (nodo u in G){
    foreach (arco(u, v) entrante in u){
        Count[u] = Count[u]+1
    }
    if (Count[u] == 0){
        S.push(u)
    }
}

//Procedura
TopologicalOrder(Count, S, G){
    if(S non è vuota){
        cancella u dalla lista S
        elimina u dal grafo G in modo da creare G-{u}

        foreach(arco(u, v) uscente da u){
            Count[v] = Count[v]-1
            if(Count[v] == 0){
```

```
                inserisci v nella lista S
            }
        }
        L = TopologicalOrder(Count, S, G)
        aggiungi u all'inizio di L
        return L
    }
    return lista vuota
}
```

Analisi

Per l'inizializzazione il tempo richiesto è O(n + m). I valori di `Count[n]` vengono inizializzati scandendo tutti gli archi e incrementando `Count[v]` per ogni arco entrante in v, richiede tempo O(m), se abbiamo informazioni sugli archi entranti per ogni nodo allora il tempo richiesto è O(n). Per inizializzare S occorre tempo O(n), in fase di inizializzazione tutti i nodi sono attivi per cui occorre effettuare una scansione per tutti gli n nodi e inserire solo quelli senza archi entranti.

Cancellare u da S richiede tempo costante O(1) e per eliminare u da G richiede tempo O(deg(u)). Ogni volta che troviamo un arco (u, v) decrementiamo `Count[v]` e se `Count[v]` == 0 allora inseriamo v in S. Queste istruzioni vengono ripetute per ogni nodo del grafo G, quindi il tempo richiesto dall'algoritmo è O(n) + O(m) = O(n + m)

G1 - Cammini Minimi

Definizione: per ogni percorso direzionato P, indichiamo con l(P) la somma delle lunghezze degli archi in P (Costo di P).

Input:

- Grafo direzionato G = (V, E)
- indichiamo con le il valore numerico(lunghezza o peso) associato ad ogni arco e.
- s = sorgente

Obiettivo: trovare i percorsi direzionati più corti da s verso tutti gli altri nodi di G.

Esistono diverse varianti del problema originale:

Single Source Shortest Paths:

- Determinare il cammino minimo da un dato nodo sorgente s ad ogni altro nodo

Single Destination Shortest Paths:

- Determinare i cammini minimi ad un dato nodo destinazione t da tutti gli altri nodi

Single-Pair Shortest Path:

- Per una data coppia di nodi u e v determinare un cammino minimo da u a v.

All Pairs Shortest Path:

- Per ogni coppia di nodi u e v, determinare un cammino minimo da u a v.

Se esiste un ciclo negativo lungo un percorso da s a v, allora non è possibile definire il cammino minimo da s a v. Il percorso avrà un costo dipendente dal numero di volte che il ciclo negativo è stato percorso.

Una possibile soluzione al problema dei cicli negativi potrebbe essere quella di utilizzare una certa costante c > 0, tale che ogni $I(e) = I(e) + c \geq 0$ l(e) = l(e) + c >= 0. Ma questo approccio non sempre funziona e in alcuni casi il percorso minimo trovato, nel grafo originale non corrisponde al cammino minimo.

G2 - Algoritmo di Dijkstra (1959)

Assumiamo lunghezze degli archi non negative per l'osservazione precedente fatta nel paragrafo precedente.

Funzionamento:

1. Ad ogni passo mantiene l'insieme S dei nodi esplorati, cioè di quei nodi per cui è stata già calcolata la distanza minima d(u) da s.
2. Inizializzazione di S = {s} e d(s) = 0.
3. Ad ogni passo, sceglie tra i nodi non ancora in S ma adiacenti a qualche nodo in S, quello che può essere raggiunto in modo più economico (scelta greedy).
4. In altra parole sceglie v che minimizza: d'(v) = $\min_{e=(u,v):u \in S}$(d(u) + le), aggiunge v ad S e pone d(v) = d'(v).

Questa versione dell'algoritmo assume che tutti i nodi siano raggiungibili da S

```
Dijkstra(G, s, l){
  Sia S l'insieme dei nodi esplorati
  foreach (nodo u in S){
    conserviamo la distanza d(u)
  }
  inizializza S = {s}
  d(s) = 0

  while(S != V){
    seleziona un nodo u ∉ S con almeno un vertice in S per cui
    d'(v) = min_e=(u,v):u∈S(d(u) + le) è il minimo possibile
    aggiungi v a S e definisci d(v) = d'(v)
  }
}
```

Analisi

Il while viene iterato n-1 volte. Se non usiamo nessuna struttura dati per il calcolo in modo efficiente di d'(v), dobbiamo scandire tutti gli archi che congiungono un vertice in S con un vertice non in S, questo prende tempo O(m) che moltiplicate alle iterazioni del while abbiamo O(nm).

G3 - Miglioramenti algoritmo di Dijkstra

Possiamo apportare diversi miglioramenti all'algoritmo proposto in precedenza:

1. Per ogni x ∉ S teniamo traccia dell'ultimo valore computato d'(x), se non è mai stato calcolato poniamo d'(x) = infinito;
2. Ad ogni iterazione del while: ∀ z ∉ S, ri-computare il valore di d'(z) se e solo se è stato aggiunto ad S un nodo u per cui esiste l'arco (u, z). Per calcolare il nuovo valore d'(z) basta calcolare il min{d'(z), d(u) + lz};
3. Se per ogni x ∈ S, memorizziamo (d'(x), x) in una coda a priorità, d'(v) può essere calcolato invocando la funzione Min oppure usando ExtractMin. L'aggiornamento di d'(z) al punto 1 può essere fatto con una ChangeKey;

G4 - Implementazione con coda a priorità

```
Dijkstra(G, s, l){

  Sia S l'insieme dei nodi esplorati
  Per ogni nodo  nodo u non in S, conserva la distanza d'(u)
  Per ogni nodo  nodo u in S, conserva la distanza d(u)
  Sia Q una coda a priorità della coppia (d'(u), u) tale che u ∉ S

  insert(Q, (0,s))
  foreach nodo u ≠ s
    insert(Q, (infinito, u))

  while(S != V){
    (d(v), v) = extractMin(Q)
    aggiungi v ad S
    foreach (arco e=(v, z)){
      if (z non in s) && (d(v) + le < d'(z)){
        changeKey(Q, z, d(v) + le)
      }
    }
  }
}
```

Analisi

Se la coda è implementata tramite una lista o un array non ordinato:

1. Inizializzazione: $O(n)$;
   While: $O(n^2)$ per le n extractMin e $O(m)$ per le m changeKey;
   Tempo totale: $O(n^2)$;
2. Se la coda è implementata mediante un heap binario:
   Inizializzazione: $O(n)$ con costruzione bottom up oppure $O(n \log n)$ con n inserimenti;

While: $O(n \log n)$ per le n extractMin e $O(m \log n)$ per le m changeKey;
Tempo totale: $O(n \log n + m \log n)$;

G5 - Dijkstra Heap Binario

Heap Binario: una struttura dati basata su un albero dove il nodo figlio sinistro contiene sempre un valore minore rispetto al padre, mentre il destro uno maggiore.

Implementazione per la creazione dell'albero dei cammini minimi:

```
Dijkstra(G, s, l){
  Sia s l'insieme dei nodi esplorati
  Per ogni nodo u non in S, conserva la distanza d'(u)
  Per ogni nodo u in S, conserva la distanza d(u)
  Sia Q una coda a priorità della coppia (d'(u),u) t.c. u non è in S

  Insert(Q,(0,s))
  Per ogni nodo u diverso da s insert(Q,(infinito,u))

  Inizializza dtree come albero vuoto
  Inserisci s come radice di dtree
  ln = s

  while(S != V){
    (d(v), v) = extractMin(Q)
    aggiungi v a S
    inserisci l'arco (ln, v) nell'albero dtree
    ln = v
    foreach (arco e=(v,z)){
      if (z non in s) && (d(v) + le < d'(z)){
        changeKey(Q, z, d(v) + le)
      }
    }
  }
  return dtree
}
```

G6 - Minimum Spanning Tree

Problema: sia dato un grafo non direzionato connesso G = (V, E) con costi C_e degli archi a valori reali. Un minimo albero ricoprente è un sottoinsieme di archi tale che T è un albero ricoprente di costo minimo.

Input: grafo non direzionato connesso G = (V, E). Ad ogni arco e è assegnato un costo che indichiamo con C_e.

Obiettivo: trovare il minimo spanning tree, ovvero quello con costo totale minimo.

Albero ricoprente (Spanning Tree)
Sia dato un grafo non direzionato connesso G = (V,E). Uno spanning tree di G è un sottoinsieme di archi T tali che $|T| = n - 1$ e gli archi di T non formano cicli.

Definizione costo di T
Sia dato un grafo non direzionato connesso G = (V, E) tale che ad ogni arco e di G è associato un costo C_e. Per ogni albero ricoprente T di G, definiamo il costo di $T = \sum_{e \in E} C_e$.

Teorema di Cayley
Ci sono n^{n-2} alberi ricoprenti del grafo completo K_n.

G7 - Taglio e Proprietà del Taglio

Taglio: un taglio è una partizione [S, V-S] dell'insieme dei vertici del grafo.
Gli archi che attraversano il taglio [S, V-S] sono definiti in un sottoinsieme D che hanno un'estremità in S e una in V-S.

Assumiamo per semplicità che i pesi degli archi siano a due a due disgiunti
Proprietà del taglio: Sia S un qualsiasi sottoinsieme di nodi e sia e l'arco di costo minimo che attraversa [S,V-S]. Ogni minimo albero ricoprente contiene e.

Proprietà del ciclo: sia C un qualsiasi ciclo e sia f l'arco di costo massimo in C. Nessun minimo albero ricoprente contiene f.

G8 - Algoritmo di Prim [Jarnik 1930, Prim 1957, Dijkstra 1959]

Comincia con un certo nodo s e costruisce un albero T avente s come radice. Ad ogni passo aggiunge a T l'arco di peso più basso tra quelli che hanno esattamente una delle due estremità in T. Se un arco ha entrambe le estremità in T, la sua introduzione in T creerebbe un ciclo.

Funzionamento:

- Ad ogni passo, T è un sottoinsieme di archi dello MST ed S è l'insieme di nodi di T.
- Inizializzazione: Pone in S un qualsiasi nodo u. Il nodo u sarà la radice dello MST.
- Ad ogni passo aggiunge a T l'arco (x, y) di costo minimo tra tutti quelli che congiungono un nodo x in S ad un nodo y in V-S (Scelta Greedy).
- Termina quando S = V.

Implementazione con coda a priorità:

```
Prim(G, c){
    Sia S l'insieme dei nodi esplorati
    Per ogni nodo u non in S, conserva il costo c[u]
    Sia Q la coda a priorità di coppie (c[u], u) t.c. u non è in S
```

```
    foreach (nodo u in V){
        insert(Q, INFINITO, u)
    }

    while(Q non è vuota){
        (c[u], u) <-- ExtractMin(Q)
        aggiungi u ad S
        foreach(arco e = (u, v)){
            if ((v non in S) AND (Ce < c[v])){
                changeKey(Q, v, Ce)
            }
        }
    }
}
```

Analisi

L'analisi è simile all'analisi dell'algoritmo di Dijkstra:

- $O(n^2)$ con array o lista non ordinati.
- $O(n \log n + m \log n)$ con heap. Siccome nel problema il grafo è connesso allora $m \geq n - 1$ allora $O(n \log n + m \log n) = O(m \log n)$;

Modifica che restituisce l'insieme dei nodi appartenenti allo MST:

```
Prim(G, c){
    Sia S l'insieme dei nodi esplorati
    Per ogni nodo u non in S, conserva il costo c[u]
    Sia Q la coda a priorità di coppie (c[u],u) t.c. u non è in S
    Sia T l'insieme dei nodi appartenenti allo MST

    foreach (nodo u in V){
        insert(Q,infinito,u)
    }

    while(Q non è vuota){
        (c[u],u)<--ExtractMin(Q)
        add u to S
        add u to T
        foreach(arco e=(u,v)){
            if ((v non in S)and (Ce<c[v])){
                changeKey(Q,v,Ce)
            }
        }
    }
    return T
}
```

G12 - Algoritmo di Kruskal

Comincia con T = Ø. Considera gli archi in ordine non decrescente di costo.
Inserisce un arco e in T se e solo se il suo inserimento non genera la creazione di un ciclo in T.

Funzionamento:
Considera ciascun arco in ordine non decrescente di peso, possiamo avere due situazioni:

1. Se e crea un ciclo allora scarta e.
2. Se e non crea un ciclo inserisci e in T.

Durante l'esecuzione di Kruskal su G =(V, E), l'insieme di vertici V e l'insieme di archi in T formano una foresta composta da uno o più alberi, cioè le componenti connesse del grafo (V, T) sono alberi. (in pratica si generano dei mini alberi che poi si uniranno in uno solo).

```
Kruskal(G, c){
    Ordina gli archi in ordine non decrescente di peso, t.c. c1c2...cn
    Inizializza T come vuoto

    foreach (nodo u in V){
        crea un singleton che contiene solo u
    }

    for(i = 1 to m){
        (u, v) = ei
        if (u e v sono in due insiemi differenti){
            T = T U {ei}
            unisci gli insiemi che contengono u e v
        }
    }
    return T
}
```

Per rappresentare gli insiemi di vertici disgiunti possiamo utilizzare una struttura dati chiamata Union-Find

G9 - Implementazione con Union Find

All'interno dell'algoritmo ciascun albero della foresta è rappresentato dal suo insieme di vertici. Per rappresentare questi vertici, si utilizza la struttura dati Union-Find per la rappresentazione di insiemi disgiunti.

Istruzioni supportate dall'Union-Find:

- MakeUnionFind(S): Crea una collezione di insiemi ognuno dei quali contiene un elemento di S. Viene chiamata nella fase di inizializzazione dell'algoritmo: 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 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 insieme X e Y. Se l'arco(u, v) non crea un ciclo in T allora l'algoritmo invoca Union(Find(u), Find(v)) per unire le componenti connesse di u e v in un'unica componente connessa.

Gli elementi sono etichettati con interi consecutivi da 1 a n e ad ogni elemento è associato una cella dell'array S che contiene il nome del su insieme di appartenenza.

```
Kruskal(G,c){
    Ordina gli archi in ordine non decrescente di peso, tale che c1 >=
c2 <= ... <= cn
    inizializza T come vuoto

    MakeUnionFind(V)

    for(i = 1 to m){
        (u, v) = ei
        if (Find(u) != Find(v)){
            T = T U {ei}
            Union(Find(u), Find(v))
        }
    }
    return T
}
```

Analisi

- Inizializzazione = $O(n) + O(m\ log\ m)$ = [ATTENZIONE] $O(m\ log\ n^2) = O(m\ log\ n)$
- Per ogni arco esaminato:
    - O(1) per le due Find()
    - In totale il ciclo for viene eseguito m volte quindi $2 * O(1) + O(m) = O(m)$
- Per ogni arco raggiunto $O(n)$ per le union:
    - In totale $O(n^2)$ per le n-1 union [ATTENZIONE 2]

Tempo totale di esecuzione: $O(n^2) + O(m\ log\ n)$

ATTENZIONE $O(n) < O(m\ log\ m) \rightarrow\ m \leq n^2\ \rightarrow\ m \leq n(n-1)/2\ = O(n^2)$ gli archi di un grafo non direzionato sono coppie non ordinate $O(logn^2)\ =\ O(2\ log\ n)\ =\ O(log\ n)$
ATTENZIONE 2 Dopo aver inserito n-1 archi in T, l'algoritmo non inserirà più alcun arco m >= n-1

G10 - Implementazione con Union Find con array union-by-size

Stessa implementazione della slide precedente, ma si usa anche un altro array A per mantenere traccia della cardinalità di ciascun insieme. L'array ha n celle perché inizialmente ci sono n insiemi:

- Find(x) = O(1)
- MakeUnionFind(S) = O(n), inizializza tutte le celle di S e di A
- Union(X, Y) = viene confrontata la cardinalità, per l'insieme minore si aggiornano le sue celle in S. In queste celle si inserisce il nome dell'insieme più grande. La cella in A che rappresenta l'insieme più piccolo viene impostata a 0, quella dell'insieme grande viene aggiornata. Tempo O(n)

Affermazione: una qualsiasi sequenza di unioni richiede O(n log n). Per ogni elemento viene svolto un lavoro in tempo O(log n). Dato che per la sequenza di union il tempo è O(n). il tempo richiesto totale è O(n log n)

1. Inizializzazione = O(n) + O(m log m) = O(m log m).
    a. O(n) creare la struttura Union-Find e
    b. O(m log m) ordinare gli archi
2. In totale il numero di find è 2m che in totale richiedono O(m)
3. Una qualsiasi sequenza di unioni richiede O(n log n)

Nuovo tempo totale = O(m log m + n log n) = O(m log m) = O(m log n^2) = O(m log n)

G11 - Implementazione basata su struttura a puntatori

Gli insiemi vengono rappresentati da strutture a puntatori. Ogni nodo ha due campi:

- Un campo per l'elemento
- Un campo che contiene un puntatore ad un altro nodo dello stesso insieme

In ogni insieme esiste un nodo che punta a se stesso in questo modo si specifica il nome dell'insieme. inizialmente ogni insieme è costituito da un solo nodo che ne specifica il nome.

- Find(x) = Si segue il percorso che va dal nodo contenente l'elemento x fino al nodo che rappresenta il nome dell'insieme. Il tempo dipende dal numero di puntatori attraversati, nel caso pessimo il tempo è O(n). Per migliorare la find(x) usiamo la union-by-size, il tempo richiesto è O(log n)

- MakeUnionFind(S) = O(n), inizializza tutte le celle di S e di A

- Union(A, B) = per eseguire la union di due insiemi A e B si pone nel campo puntatore dell'elemento x in A il nodo dell'elemento y in B. Tempo O(1)

* 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

Tempo totale = O(m log n)

G12 - Path Compression

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.

Idea: ogni volta che eseguiamo la Find con in input un elemento x di un certo insieme possiamo fare del lavoro in più che ci fa risparmiare sulle successive operazioni di Find. Questo lavoro in più non fa comunque aumentare il tempo di esecuzione asintotico della singola Find.

Questa euristica insieme alla union-by-size permettono ad una sequenza di operazioni union-find di venir eseguite in tempo O(n α(n)). α(n) è l'inversa della funzione di Ackermann [α(n) <= 4].