# Grafi

Luca Tagliavini

April 26-March 17, 2021

## Contents

# 1 Introduzione

Un grafo e' un insieme di punti detti *nodi* collegati da ponti detti *archi*. I grafi possono essere sia **ciclici** che non, ma quelli non ciclici possono essere interpretati come alberi senza radice.

*Gli archi possono essere pesati* in modo che i collegamenti tra alcuni nodi possano avere un valore ad essi associato.
Gli archi possono oltretutto essere *orientati*, indicando che il collegamento da essi rappresentato e' solo in una data direzione (o maggiormente rilevante in una data direzione).

## 1.1 Problemi sui grafi

1. Visite
   - in ampiezza (BFS), usata per identificare il cammino di lunghezza minima da una singola sorgente
   - profondita' (DFS)
2. Alberi di copertura minima
3. Cammini minimi da una singola sorgente o tra tutte le coppie di vertici

### 1.1.1 Orientamento

I grafi possono essere **orientati** o **non orientati**.

- Un grafo *orientato* e' composto come una coppia $(V, E)$ dove $V$ e' l'insieme finito di tutti i vertici, ed $E$ e' l'insieme di archi **orientati** che collega gli elementi contenuti in $V$. Percio' se si vuole avere un collegamento a doppio senso tra due nodi $A$ e $B$ bisogna inserie le coppie $(A, B)$ e $(B, A)$ dentro a $E$.
- Un grafo *non orientato* e' rappresentato come una coppia $(V, E)$ in modo analogo a quello dei grafi orientati, ma per gli archi non venogno usate coppie ma bensi' insiemi i quali, a differenza delle coppie, ignorano l'ordine nel quale gli elementi appaiono in essi.

### 1.1.2 Incidenza e Adiacenza

Un *arco* si dice **incidente** su $v$ se collega un qualche nodo ad esso.
Un *arco* si dice **incidente** da $v$ se collega $v$ ad un qualche altro nodo.

Un *vertice* $w$ si dice **adiacente** a $v$ sse $(v, w) \in E$, ovvero esiste un collegamento diretto tra $v$ e $w$.

### 1.1.3 Operazioni sui grafi

- $n\_vertices() \rightarrow int$: ritorna il numero di vertici $\in V$.
- $n\_edges() \rightarrow int$: ritorna il numero di vertici $\in V$.
- $grade(vertex\_tv) \rightarrow int$: restituisce il numero di archi entranti o uscenti con esso, ossia tutti gli archi incidenti ad esso. Piu' avanti potremo anche definire *grade* come $in\_degree() + out\_degree()$ ossia la somma dei gradi in entrata e in uscita al nodo.
- $incident\_edges(vertex\_tv) \rightarrow edge\_t[]$: ritorna tutti gli archi incidenti al nodo
- $extremes(edge\_ta) \rightarrow vertex\_t[2]$: ritorna i nodi che l'arco in input collega
- $opposite(edge\_ta, vertex\_tv) \rightarrow vertex\_t$: restituisce l'altro nodo al quale e' collegato $v$ tramite l'argo $a$
- $adjacent(vertex\_ta, vertex\_tb) \rightarrow bool$: restituisce true sse i due vertici hanno un arco di collegamento diretto.
- $add\_vertex(vertex\_tv) \rightarrow void$: aggiunge un veritce alla lista di vertici $V$.
- $add\_edge(arx\_ta) \rightarrow void$: aggiunge un arco alla lista di archi $E$.
- $del\_vertex(vertex\_tv) \rightarrow void$: rimuove un veritce dalla lista di vertici $V$.
- $del\_edge(arx\_ta) \rightarrow void$: rimuove un arco dalla lista di archi $E$.

## 1.2 Possibili implementazioni

Tratteremo solo l'implementazione degli archi, in quanto i nodi saranno mantenuti in ogni caso in un vettore di nodi. Useremo i seguenti valori all'interno delle slide:

- $n$ = numero di vertici = $|V|$
- $m$ = numero di archi = $|E|$

1. Liste di archi (grafo non orientato): usiamo una struttura dati lista per memorizzare gli archi che collegano i vari nodi. Il costo in quantita' di spazio e' ottimale, $O(|E|)$, ovvero $O(m)$, ma rende estremamente costose alcune operazioni in quanto bisogna scorrere tutta la lista per prendere l'elemento in posizione $n$.
2. Liste di incidenza (grafo non orientato): si possono rappresentare gli archi inserendoli come informazioni aggiuntive ai singoli nodi in modo che i costi delle operazioni siano quasi tutti legati al grado dei nodi (numero di collegamenti del nodo). D'ora in avanti indicheremo il grado con il simbolo $\delta(n)$ dove $n \in V$.

3. Matrice di adiacenza (grafo non orientato): costruiremo una matrice atta a indicare quali nodi sono collegati e con chi, generano una matrice quadrata simmetrica $|v| \times |v|$ descritta dalla seguente equazione:

$$M(u,v) = \begin{cases} 1 & \text{se} \{u,v\} \in E \\ 0 & \text{altrimenti} \end{cases}$$

   Sfortunatamente non solo il costo di memoria sara' dell'ordine di $O(n^2)$, ma anche molte operazioni avranno tale costo a causa della possibile necessita' di scandagliare tutta la matrice o allargarla (copiando i valori vecchi in quella nuova).

4. Lista di adiacenza (grafo non orientato): si salvano i nodi in una lista come sottoliste. Ogni sottolista contiene il valore (nome del nodo) e un puntatore ad altri elementi di questa struttura a lista che contengono tutti i nodi ai quali il nostor $n \in N$ e' collegato. Questa tecnica, molto utilizzata, porta molti metodi a un costo computazionale $O(\delta(v))$ dove $v \in V$. **Rifrasando, teniamo i nodi in una lista, e ogni nodo viene rappresentato come una lista che contiene tutti i nodi ai quali esso e' collegato (ovvero i nodi adiacenti)**.

   Sara' il tipo d'implementazione usato maggiormente.

## 1.3 Grafi pesati

In alcuni grafi ogni arco ha un *peso* o costo associato. Il costo associato ad un arco puo' essere calcolato tramite una funzione $w : E \to \mathbb{R} \cup \{\infty\}$. Quando tra due nodi non esiste un arco il valore $w(\cdot) = \infty$.

Estendendo l'implementazione del grafo vista nel punto 3 sara' possibile scrive una matrice di adiacenza per grafi non orientati *pesati*:

$$M(u,v) = \begin{cases} w(u,v) & \text{se } \{u,v\} \in E \\ \infty & \text{altrimenti} \end{cases}$$

## 1.4 Cammini

I *cammini* sono una sequenza di archi che devo attraversare per arrivare da un vertice ad un altro. Matematicamente posso pensare al cammino $\langle w_0, w_1, \ldots, w_n \rangle$ tale che $\{w_i, w_i + 1\} \in E$ con $0 \leq i \leq n-1$. Il cammino $\langle w_0, w_1, \ldots, w_n \rangle$ contiene i vertici $w_0, w_1, \ldots, w_n$ ma anche i vertici usati per spostarsi nel cammino $\{w_0, w_1\}, \{w_1, w_2\}, \{w_n - 1, w_n\}$. La *lunghezza del cammino* e' il numero di archi che devo visitare per andare da $w_0$ a $w_n$.

I cammini sono detti *semplici* se non hanno vertici ripetuti al loro intero (i.e. $A \to B \to C$). I cammini sono detti *non semplici* quando al loro interno si hanno vertici ripetuti (i.e. $\underline{A} \to B \to \underline{A} \to C$).

### 1.4.1 Raggiungibilita'

Se esiste un cammino $c$ tra i vertici $v$ e $w$, si dice che $v$ *e' raggiungibile da* $w$ *tramite* $c$. Il viceversa $w \to v$ vale nei grafi non ordinati, ma non e' garantito in quelli ordinati.

## 1.5 Grafi connessi

Un grafo non orientato si dice *connesso* se $\forall v_1, v_2 \in V \quad w(v_1, v_2) \neq \infty$. In altre parole, si un grafo si dice connesso quando per ogni vertice esiste un cammino ad ongi altro vertice.

### 1.5.1 Grafo fortemente connesso

Se siamo in un grafo orientato si dice che esso e' *fortemente connesso* se esiste un cammino da ogni vertice ad ogni altro vertice.

## 1.6 Conversione di grafi

Prendendo un grafo *orientato* si puo' ottenere la sua versione *non orientata* ignorando i versi e rimuovendo i cappi.

Partendo invece da un grafo *non orientato* lo si puo' rendere *orientato* trasformando ogni arco in due archi direzionali che puntano in entrambe le direzioni.

### 1.6.1 Grafo debolmente connesso

Un grafo *ordinato* si dice debolmente connesso, se esso non e' fortemente connesso, e presa la sua rappresentazione come grafo *non ordinato* essa si puo' dire *connesso*.

## 1.7 Ciclo

Un *ciclo* in un grafo orientato e' un cammino $\langle w_0, w_1, \ldots, w_n \rangle$ di lunghezza $\geq 1$ tale che $w_0 = w_n$. Un ciclo si dice semplice se i nodi $w_1, \ldots, w_{n-1}$ *sono distinti*.

I grafi si dicono *aciclici* se non contengono cicli. Una terminologia molto usata sara' quella del *DAG*, ovvero Directed Acyclic Graph.

## 1.8 Grafo completo

Un *grafo non orientato completo* e' un grafo che ha un arco tra ogni coppia di nodi possibile. Quanti archi si avranno in questo modo? $\frac{n(n-1)}{2}$

## 1.9 Grafi come alberi

Un *albero libero* e' un grafo non ordinato connesso aciclico. Se un vertice viene denominato radice si ha un classico albero radicato.

# 2 Algoritmi sui grafi

## 2.1 Visite

Dato un grafo $G = (V, E)$ e un vertice sorgente $s \in V$ si visita ogni vertice raggiungibile da tale nodo sorgente *una sola volta*.

Come per le visite sugli alberi ci sono due tecniche per la visita: BFS che visita prima in ampiezza, analizzando ogni nodo direttamente collegato alla sorgente e allargandosi poi ricorisvamente, e DFS che visita in profondita' e visita i vertici in base alla loro distanza dalla sorgente.

### 2.1.1 Visita generica sui grafi

Possiamo usare gli algoritmi per la visita in ampiezza e profondita' sugli alberi, tramite una coda o una stack, ma dobbiamo curarci di segnare i vertici gia' visitati in quanto i grafi possono essere ciclici (o comunque per visitare i "figli" di un nodo si va a visitare inevitabilmente anche il padre) e il nostro algoritmo potrebbe visitare alcuni nodi piu' volte e andare in loop. Marcheremo percio' ogni nodo con tre diversi tipi di stato:

1. **inesplorato** il nodo non e' ancora stato visitato
2. **aperto** il nodo e' uno di quelli attualmente visitati
3. **chiuso** il nodo e' gia' stato esplorato

L'idea per la BFS sui grafi e' quella di creare un albero di visita $T$ radicato nella sorgente $s$ che contiene tutti i nodi raggiungibili da $s$ tramite gli archi che abbiamo poi effettivamente utilizzato per la ricerca. L'albero costituito da questi nodi e questi archi viene restituito dall'algoritmo.

Avremo poi una stack/queue $F$ chiamata anche frontiera, che contiene tutti i nodi gia' visitati ma che possono ancora darci qualche vertice tramite i propri adiacenti.

Esiste poi un algoritmo generico che ci consente di implementare sia una BFS che una DFS variando il tipo della frontierea $F$. Con una stack *lifo* avremo una ricerca DFS, mentre con una queue *fifo* avremo una ricerca BFS.

Eccone una implementazione generica in pseudocodice:

**Algorithm:** Generale DFS/BFS su grafi

```
Data: G = (V, E), V s dove s e' la radice
Result: T = traverse tree
begin
    foreach v ∈ V do
        v.marked ← false
    end
    t ← s
    f ← { s }
    while f ≠ ∅ do
        u ← f.extract()
        visit(u)
        foreach v ∈ adjacent(u) do
            if v.marked = false then
                v.marked ← true
                f.push(v)
                u.insert(v)
            end
        end
    end
    return t
end
```

Globalmente in *tutte le sue esecuzioni* il for each viene eseguito $k$ volte dove $k = 2|E| = m$ poiche' ogni arco viene traversato due volte (una padre-figlio, l'altra figlio-padre). Il ciclo while viene eseguito $k = |V| = n$ volte, e anche se essi sono nestati, abbiamo fatto il calcolo prima in modo globale, non pensandolo come ciclo nestato, il che ci da un costo complessivo di $O(n + m)$.

## 2.2 Visita in ampiezza BFS

La BFS traversa il grafo in ampiezza, visitando in successione tutti i nodi nello stesso "livello" del grafo, ossia tutti quelli alla stessa distanza dalla sorgente.

Vediamo lo pseudocodice:

**Algorithm:** Generale DFS/BFS su grafi

```
Data: G = (V, E), V s dove s e' la radice
Result: T = traverse tree
begin
    foreach v ∈ V do
        v.marked ← false
    end
    t ← s
    f ← new Queue(s)
    s.dist ← 0
    while f ≠ ∅ do
        u ← f.dequeue()
        visit(u)
        foreach v ∈ adjacent(u) do
            if v.marked = false then
                v.marked ← true
                v.dist ← u.dist + 1
                f.enqueue(v)
                u.insert(v)
            end
        end
    end
    return t
end
```

La BFS puo' essere sfruttata per trovare i percorsi piu' brevi tra due nodi (grazie al suo albero di visita), che vengono chiamati anche *cammini di lunghezza minima*.

## 2.3 In particolare: DFS su grafi

**Algorithm:** Funzione ausiliaria a DFS per grafi

```
dfs(G = (V, E), s)
    time ← time + 1
    /* discovery time - started exporing the node          */
    s.dt ← time
    foreach v ∈ adjacent(s) do
        if v.color = white then
            v.parent = s
            dfs(G, v)
        end
    end
    visit(s)
    time ← time + 1
    /* finish time - the node has been fully visited       */
    s.ft ← time
    s.color ← black
```

La funzione $dfs$ sara' poi chiamata dalla vera funzione di vista, che avra' questa forma:

**Algorithm:** Generale DFS/BFS su grafi

```
Data: G = (V, E), V[] V insieme di nodi da cui far partire la visita
Result: T = traverse tree
begin
    time leftarrow 0
    foreach v ∈ V do
        v.color ← white v.parent ← null
    end
    foreach u ∈ V do
        if u.color = white then
            dfs(u)
        end
    end
end
```

Come si puo' notare questo algoritmo $DFS$ esegue le sottochiamate ausiliare $dfs$ su ogni nodo dato in input. Questo comportamento e' desidierabile nel caso in cui si voglia far partire la ricerca da vari nodi poiche' alcuni non sono collegati tra di loro e ci daranno dunque accesso a "nuove" parti dell'albero (rispetto agli altri). Quello che verra' generato non sara' piu' un *albero di visita* ma bensi' un insieme di essi, denonminato *foresta di visita*.

Per ogni nodo manteniamo le variabili *discovery time* e *finish time* che rappresentano rispettivamente il "tempo" di scoperta e quello di fine dell'esplorazione di un determinato nodo. Questa proprieta' e l'algoritmo DFS ci consentiranno di risolvere problemi quali: determinare se un grafo orientato e' aciclico, trovare

la mappatura topologica di un grafo, ....

### 2.3.1 Teorema delle parentesi

Come in matematica e in programmazione vige l'implicita regola che ogni parentesi vada chiusa, possiamo identificare un simile pattern anche nelle visite DFS sui grafi, in particolare guardando alle variabili *dt* ed *ft*.

Poiche' la nostra DFS e' una funzione ricorsiva e le variabili *dt* e *ft* vengono assegnate prima e dopo le chiamate ricorsive, esse rispettano la proprieta' delle parentesi.

Per questo teorema, prendendo due vertici $u$ e $v$ dal nostor grafo visitato, varranno le seguenti proprieta':

1. Gli intervalli $[u.dt, u.ft]$ e $[v.df, v.ft]$ sono disgiunti
   i due vertici non sono legati da alcuna relazione di parentela
2. L'intervallo $[u.dt, u.ft]$ e' completamente contenuto in $[v.df, v.ft]$ o viceversa
   il nodo il cui intervallo e' contenuto e' figlio dell'altro
3. Gli intervalli $[u.dt, u.ft]$ e $[v.df, v.ft]$ sono parzialmente sovrapposti
   non puo' mai avvenire in una corretta visita *dfs*

### 2.3.2 Grafi orientati

Nei grafi orientati, preso un arco $(u, v)$ non presente negli alberi della foresta restituita da *dfs* si possono distinguere tre casi in base ai valore *df* e $ft$ dei vertici:

1. Se $v.dt < u.dt$ e $u.ft < v.ft$ l'arco $(u, v)$ e' detto all'indietro
2. Se $u.dt < v.dt$ e $v.ft < u.ft$ l'arco $(u, v)$ e' detto all'avanti
3. Se $v.ft < u.dt$ l'arco $(u, v)$ e' detto attraversamento a sinistra
   NOTA: l'opposto (attraversamento a destra) non si puo' mai verificare in quanto romperebbe la proprieta' di visita singola di ongi singolo vertice delle ricerche sui grafi.

### 2.3.3 Identificare grafi ciclici

Per identificare un ciclo in un grafo *orientato*, una volta svolta la *dfs* basta verificare se tra gli archi rimanenti (non usati dalla dfs) si ha un arco all'indietro. Un simile arco ci consentirebbe di tornare ad un nodo precedente e di conseguenza entrare in un ciclo.

### 2.3.4 Ordinamento topologico

In alcune situazioni, ad esempoio una lista di job da svolgere **tra loro dipendenti**, si hanno delle dipendenze causali tra i vari vertici. Una struttura di questo tipo viene solitamente rappresentata tramite un DAG (Direct, Acyclic Graph).

In un grafo di questo tipo si possono individuare ordinamenti, che vengon chiamati *topologici* sse preso un arco $(u, v)$ allora $u$ compare prima di $v$ nell'ordinamento. Si noti che un dag puo' avere svariati ordinamenti topologici. L'algoritmo per risolvere questo problema puo' essere implementato tramite i seguenti step:

1. Si effettua una DFS sul grafo dato
2. La funzione *visit* viene implementata in modo da aggiungere il nodo attualmente visitato in testa a una lista
3. La lista generata in questo modo viene restituita dall'algoritmo

## 2.4 Visite su componenti connesse (grafo non orientato)

Introduciamo la nozione di componente connessa: Due vertici $u$ e $v$ appartengono alla stessa componente connessa se $u$ e' raggiungibile da $v$. Questa e' una realzione di equivalenza con le proprieta' *riflessiva, simmetrica e transitiva.*

E' di nostro interesse eseguire delle visite sui vertici di queste componenti connesse.

> NOTA: in un grafo non ordinato tutti i vertici connessi, essendo collegati con archi in entrambe le direzioni, appartengono alla stessa componente connessa.

## 2.5 Componenti fortemente connesse (grafo orientato)

Ricordiamo: un grafo *orientato* $G$ e' fortemente connesso se ogni coppia di vertici e' connessa da un cammino.

Due vertici $u$ e $v$ appartengono alla stessa componente fortemente connessa sse esiste un cammino da $u$ a $v$ e viceversa. Anche *la relazione di connettivita' forte* gode delle proprieta' *riflessiva, simmetrica e transitiva.*

Due vertici $u$ e $v$ appartengono alla stessa componente fortemente connessa sse esistono i cammini $u \to v$ e $v \to u$. Per calcolare tutti i nodi raggiungili e dai quali si puo' tornare indietro basta fare l'intersezione tra i vertici discendenti $D(x)$ e i vertici antenti $A(x)$ dove $x$ e' il nodo di nostro interesse. Calcolare $D(x) \cap A(x)$ ci dara' dunque l'insieme di vertici fortemente connessi.

Identificare l'insieme $D(x)$ e' facile, basta seguire gli archi che partono da $x$, mentre un'idea furba per trovare $A(x)$ e' quella di invertire le direzioni di tutti gli archi e vedere ancora una volta gli archi che partono da (ma prima puntavano a) $x$.

## 2.6 Minimum Spanning Tree

Problema classico: Bisogna creare un circuito stampato dove svariati componenti hanno svariati pin che vanno collegati tra di loro, e desideriamo trovare i collegamenti piu' corti tra i pin.

Il problema e' risolvibile tramite un grafo *non orientato e connesso*, che ha anche una funzione peso $w : V \times V \to \mathbb{R}$ che assegna a ogni arco (filo nell'esempio pratico) un peso (lunghezza del filo). Il nostro obbiettivo sara' poi trovare il *sottografo* che connette tutti i vertici ed ha il minimo peso possibile. Possiamo indicare il grafo includendo anche la definizione della funzione peso in questo modo: $G = (V, E, w)$.

L'output del nostro problema sara' uno *spanning tree* (albero di copertura) $T = (V, E_T)$ tale che $V = V$ (ovvero contiene tutti i vertici iniziali), $E_T \subseteq E$ (ovvero i vertici sono collegati da un sottoinsieme degli archi iniziali), e tale che $T$ sia l'albero di copertura con **sommatoria dei pesi minima**, ovvero:

$$w(T) = \sum_{(u,v)\in T} w(u, v) \text{ sia minimo tra tutti i } T \text{ esistenti}$$

> NOTA: il MST potrebbe non essere unico, possono esserci infatti coperture equivalentemente buone come peso

## 2.7 Algoritmo generico (base di Kruskal e Prim)

Scriveremo un algoritmo *greedy* che cerca di ingrandire in modo incrementale un sottografo $T$ del grafo iniziale fino a raggiungere un sottografo con $n - 1$ archi ($n$ e' il numero di vertici) mantenendo sempre la proprieta' che $T$ **e' un sottoinsieme di qualche albero di copertura minimo**, ovvero e' composto interamente da archi *sicuri*, dove *essere sicuro* significa che un arco $(u, v) \cup T$ e' ancora un sottoinsieme di archi di un valido MST.

L'algoritmo generico puo' essere espresso in questo modo:

**Algorithm:** Generica costruzione di un MST

```
Data: G = (V, E, w)
Result: MST
begin
    tree ← ∅
    while !is_mst(tree) do
        vertex ← find_save_edge()
        tree ← tree ∪ vertex
    end
    return tree
end
```

### 2.7.1 Def: Taglio

- *Un taglio* e' una terna $(S, V, -S)$ di un grafo $G = (V, E)$ tale che il grafo viene diviso in due sottoinsiemi disgiunti.

- Un arco $(u, v)$ *attraversa un taglio* sse $u \in S \wedge v \in -S$.
- Un arco *rispetta un insieme di archi* $T$ se nessun arco di $T$ attraversa il taglio.
- Un arco che attraversa un taglio e' *leggero* se il suo peso e' minimo tra tutti i nodi che attraversano il taglio.

## 2.8 Regole

1. Regola del **taglio**: scegli un taglio $T$ che **non contiene archi blu**. Tra tutti gli archi del taglio seleziona una arco di peso minimo e coloralo blu.
2. Regola del **ciclo**: scegli un ciclo $G$ che **non contiene archi rossi**. Tra tutti gli archi del ciclo scegli l'arco di peso maggiore e coloralo rosso.

Soluzione *greedy*: applica ad ogni step una delle due regole, purche' si possa applicare, fino a generare un MST.

## 2.9 Algoritmo di Kruskal

L'algoritmo id Kruskal parte considerando tutti gli archi, in ordine non decrescente per peso dato dalla funzione $w$. Per ogni arco, se esso e' *l'arco che attraversa un taglio*, sara' sicuramente quello piu' *leggero* poiche' l'arco di peso minimo, quindi potremo colorarlo in blu (regola del taglio). Se esso non attraversa un taglio, ma bensi' fa parte di un ciclo dove e' l'arco piu' pesante esso viene colorato di rosso (regola del ciclo).

> NOTA: dire che non riusciamo a fare un taglio o che un arco appartiene ad un ciclo ed ha peso massimo sono la stessa cosa applicando questo algoritmo.

Una volta colorati tutti gli archi l'MST sara' dato da tutti gli archi colorati

di blu, o (ancora meglio) quando abbiamo colorato $n-1$ archi.

**Algorithm:** Algoritmo di Kruskal

```
Data: G = (V, E, w)
Result: MST
begin
    uf ← new UnionFind()
    tree ← ∅
    foreach v ∈ V do
        uf.make_set(v)
    end
    sort(E, w)
    foreach (u, v) ∈ E do
        id_u ← uf.find(u)
        id_v ← uf.find(v)
        if v ≠ v then
            tree ← tree ∪ (u, v)
            uf.merge(id_u, id_v)
        end
    end
    return tree
end
```

**Costo**: $O(2m \log_2 n + n \log_2 n)$, poiche' $m = O(n^2)$ si ha come costo massimo $O(m \log_2 n)$.

## 2.10 Algoritmo di Prim

L'idea e' quella di partire con un singolo vertice e far crescere l'albero ampliando lo spanning tree connettendo gli archi di costo minore al nodo gia' esplorato.

Ad esempio nella prima esecuzione del ciclo, avremo un solo vertice selezionato. In tal modo *creiamo un taglio* tra il singolo nodo e il resto del grafo, e scegliamo l'arco che attraversa il taglio prendendo quello *di costo minore*.

L'arco sara' ingrandito a ogni step ma si puo' applicare comunque questa

tecnica fino a completare il Minimum Spanning Tree.

**Algorithm:** Algoritmo di Prim

```
Data: G = (V, E, w), V s = radice da cui partire
Result: int[]
begin
    /* distance of the node from the tree                    */
    d ← double[len(V)]
    /* vettore dei padri, per ogni elemento indico qual'e' il
       padre (usato per ricostruire l'albero di Prim)        */
    p ← int[len(V)]
    b ← bool[len(V)]
    foreach v ∈ V do
        d[v] = ∞
        p[v] = -1
        b[v] = false
    end
    /* the first node is at 0 disance                        */
    d[s] ← 0
    q ← new ProprityQueue(s, d[s])
    while !q.is_empty() do
        u ← q.get_max()
        q.delete_max()
        b[u] ← true
        foreach v ∈ adjacent(u) and b[v] ≠ true do
            if d[v] = ∞ then
                d[v] ← w(u, v)
                p[v] ← u
                q.insert(v, d[v])
            else if w(u, v) ¡ d[v] then
                d[v] ← w(u, v)
                p[v] ← u
                q.update_key(v, d[v])
            end
        end
    end
    return p
end
```

Il costo dell'algorimto di Prim e' da: costo del while per costo del for $O(m)$. In ogni for vengono chiamate *insert* e *update* che hanno entrambe costo logaritmico. Si ha dunque $O(m) \cdot O(\log_2 n) = O(m \log_2 n)$. Notare che il while ha costo $O(n)$ poiche' visitera' tutti i vertici del grafo, ma sfruttiamo questa informazione per valutare il costo del for each, che complessivamente, in $n$ cicli, visitera' tutti gli archi, dandoci un costo di $O(m)$.

Anche *delete_min* ha costo logaritmico (essendo dentro al while avrebbe costo $O(n \log_2 n)$) ma puo' essere ignorata rispetto al costo di $O(m \log_2 n)$.

# 3 Cammini minimi

Dato un grafo *orientato e pesato* $G = (V, E, w)$ si definisce **il costo di un cammino** $\pi = (v_0, v_1, \ldots, v_k)$ che collega il nodo $v_0$ con il nodo $v_k$:

$$w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Dato una coppia di vertici $v_0, v_k$ vogliamo trovare il cammino tra tutti quelli esistenti con costo minimo. **Si noti che non e' garantito che questo cammino esista**.

Ci sono tre varianti del problema della ricerca del cammino minimo:

1. **cammino minimo tra** $u$ **e** $v$: trovare se esiste il cammino minimo tra tutti quelli del tipo $\pi_{u,v}$ tale che il suo costo sia minimo.
2. **single-source shortest path**: partendo da un vertice sorgente $s$ trovare tutti i cammini migliori per raggiungere ogni nodo raggiungibile da $s$.
3. **all-pairs shortest pairs**: determinare ogni cammino ottimale tra $\forall u \in V, \forall v \in V.(u, v)$.

   NOTA: non e' noto alcun problema in grado di risolvere il punto 1 senza risolvere anche il punto 2 nel caso peggiore.

## 3.1 Casi in cui non esiste un cammino minimo

I casi in cui non esiste un cammino minimo possono essere due:

- la destinazione non e' raggiungibile.
- quando ci sono dei **cicli di costo negativo**. In tal caso un qualunque percorso $a$ sarebbe comunuqe battibile da un percorso $b$ che fa la stessa strada di $a$ ma passa per una volta in piu' nel ciclo a costo negativo.

## 3.2 Proprieta' dei cammini minimi

- assumiamo che non ci siano cicli di costo negativo.
- tra ogni coppia di vertici connessi esiste un cammino *semplice* di costo minimo.
- preso un cammino minino, un suo sottocammino e' anch'esso un cammino minimo tra i vertici d'inizio e destinazione del sottocammino.
- per il punto precedente i nostir algoritmi punteranno a trovare i cammini minimi piu' piccoli per poi allargarli fino a raggiungere la destinazione cercata.

## 3.3 Risultati degli algoritmi sui cammini minimi

Tutti gli algoritmi per torvare dei cammini minimi resituiscolo un albero di copertura tale che ogni cammino nell'albero sia un cammino di costo minimio per andare dal nodo sorgente $s$ ad un qualunque altro nodo (che si trova nell'albero).

## 3.4 Algoritmo di Bellman-Ford

Dato un grafo $G = (V, E)$ definisco il costo del cammino minimo da $x$ a $y$ come:

$$d_{x,y} = \begin{cases} w(\pi_{x,y}) & \text{se esiste} \\ \infty & \text{altrimenti} \end{cases}$$

NOTA: $d_{v,v} = 0 \forall v \in V$
NOTA: vale la disuguaglianza triangolare:

$$d_{x,z} \leq d_{x,y} + d_{y,z} \quad \forall x, y, z \in V$$

### 3.4.1 Condizione di Bellman

Un particolare caso della disugualianza triangolare vale quando si considera un arco $(u, v)$ e una sorgente $s$:

$$d_{s,v} \leq d_{s,u} + w(u, v) \quad \forall u, v \in V$$

*Dimostrazione*:

- si parte della disugualianza triangolare:

$$d_{s,v} \leq d_{s,u} + d_{u,v} \quad \forall (u, v) \in E, s \in V$$

- ci si ricorda che vale

$$d_{u,v} \leq w(u, v)$$

  poiche' la distanza *minima* tra $u$ e $v$ non puo' essere maggiore del costo dell'arco che li collega

- allora ne segue:

$$d_{s,v} \leq d_{s,u} + w(u, v) \quad \forall (u, v) \in E, s \in V$$

Da questa disugualianza si puo' capire che l'arco $(u, v)$ fa parte del cammino minimo da $s$ a $v$ $\pi_{s,v}$ sse vale:

$$d_{s,v} = d_{s,u} + w(u, v) \quad \forall (u, v) \in E, s \in V$$

## 3.5 Tecnica del rilassamento

Supponiamo di mantenere una stima $D_{s,v} \geq d_{s,v}$ della lunghezza del cammino di costo minimo da $s$ a $v$. Effettuiamo poi dei passi di rilassamento (tecnica di raffinamento progressivo) tramite i quali miglioreremo la nostra stima fino ad avere $D_{s,v} = d_{s,v}$. In pseudocodice:

**Algorithm:** Condizione di raffinamento

```
if D_{s,u} + w(u,v) < D_{s,v} then
    D_{s,v} ← D_{s,u} + w(u,v)
end
```

## 3.6 Algoritmo di Bellman-Ford

**Algorithm:** Algoritmo di Bellman-Ford

```
Data: G = (V, E, w), V s = radice da cui partire
Result: double[]
begin
    int n ← len(V)
    /* predecessor for each node in the path from s to node
       */
    int pred[1..n]
    /* cost of each path from s to node                    */
    double D[1..n]
    /* set the cost of each path from s to any node to ∞   */
    for int v ← 1 to n do
        D[v] ← ∞
        pred[v] ← -1
    end
    /* the path from the source to itself has a cost of 0  */
    D[s] ← 0
    for int i ← 1 to n-1 do
        foreach (u, v) ∈ E do
            /* relaxing technique condition                */
            if D[u] + w(u,v) ¡ D[v] then
                D[v] ← D[u] + w(u,v) pred[v] ← u;
            end
        end
    end
    return D
end
```

Costo: $\Theta(n + (n-1) \cdot m) = O(n \cdot m)$

NOTA: puo' essere ottimizzato fermando l'esecuzione quando il **foreach** non esegue alcun cambiamento, il che significa che non potremo ulteriormente migliorare le nostre stime.

### 3.6.1 Dimostrazione per induzione

Un cammino di lunghezza $k$ viene scoperto al piu' in $k$ passaggi:

- **Caso base** ($k = 0$): per andare da $s$ a $v_{k_{|k=0}} = s$ ho costo 0:
  $D_{s,s} = d_{s,s} = 0$.
- **Caso ricorsivo** ($k > 0$): per andare da $s$ a $v_{k_{|k>0}}$ posso sfruttare il cammino per andare fino a $v_{k-1}$ e sommarvi anche il peso dell'arco $(v_{k-1}, v_k)$. Si ha quindi:
  $D_{s,v_k} = d_{s,v_k} = d_{s,v_{k-1}} + w(v_{k-1}, v_k)$

Si ha dunque che al $k$-esimo passagio potremo calcolare la sua lunghezza tramite i $k-1$ conti svolti prima, dunque svolgeremo al piu' $k$ passaggi semplici.

### 3.6.2 Controllo per cicli negativi

Una volta eseguito l'algoritmo di Bellman-Ford possono comunque esistere cammini migliori in caso ci siano dei cicli di peso negativo. Ecco un semplice loop da appendere in fondo all'algoritmo di BF per capire se siamo in un grafo con un ciclo di peso negativo.

**Algorithm:** Condizione per indetificare cicli negativi

```
foreach (u, v) ∈ E do
    if D[u] + w(u, v) ¡ D[v] then
        throw "ciclo negativo"
    end
end
```

## 3.7 Algoritmo di Dijkstra

**Algorithm:** Algoritmo di Dijkstra

```
Data: G = (V, E, w), V s = radice da cui partire
Result: double[]
begin
    int n ← len(V)
    /* predecessor for each node in the path from s to node
       */
    int pred[1..n], v, u
    /* cost of each path from s to node                    */
    double D[1..n]
    /* set the cost of each path from s to any node to ∞   */
    for int v ← 1 to n do
        D[v] ← ∞
        pred[v] ← -1
    end
    /* the path from the source to itself has a cost of 0  */
    D[s] ← 0
    q ← new PriorityQueue(s, D[s])
    while not q.is_empty() do
        u ← q.get_max()
        q.delete_max()
        foreach v ∈ adjacent(u) do
            if D[v] = ∞ then
                D[v] ← D[u] + w(u, v)
                q.insert(v, D[v])
                pred[v] = u
            else if D[u] + w(u, v) ¡ D[v] then
                D[v] ← D[u] + w(u, v)
                q.update_key(v, D[v])
                pred[v] = u
            end
        end
    end
    return D
end
```

## 3.8 Lemma di Dijkstra

Sia $G = (V, E, w)$ un grafo orientato con $w$ funzione peso $w : V \times V \to \mathbb{R}^+$. Sia $T$ una parte dell'*albero dei cammini di costo minimo*, per cui $T$ rappresenta cammini che partono da $s$ tutti di costo minimo. Si ha dunque che l'arco $(u, v)$ con $u \in vertices(T)$ e $v \notin vertices(T)$ che minimizza la quantita' $d_{s,u} + w(u, v)$ appartiene dunque all'albero $T$ nel cammino da $s$ a $v$.

### 3.8.1 Dimostrazione (per assurdo)

Supponiamo per assurdo che $(u, v)$ non appartenga al cammino di costo minimo da $s$ a $v$. Quindi $d_{s,u} + w(u, v) > d_{s,v}$ $(H_1)$.
Quindi deve esistere il cammino $\pi_{s,v}$ che porta da $s$ a $v$ ma non passa per $(u, v)$ con costo inferiore a $d_{s,u} + w(u, v)$.
Il cammino $\pi_{s,v}$ puo' essere spezzato in $\pi_{s,y}$ e $\pi_{y,v}$ dove $y$ e' il primo nodo che scegliamo al posto di $u$ per ottenere il cammino minimo.
Si ha dunque che $d_{s,v} = d_{s,x} + w(x, y) + d_{y,v}$ $(H_2)$.

Tuttavia, per ipotesi sapevamo $(u, v)$ e' l'arco che collega un vertice in $T$ con uno non ancora in $T$ tale che esso minimizza la somma $d_{s,u} + w(u, v)$.
In particolare: $d_{s,u} + w(u, v) \leq d_{s,x} + w(x, y)$ $(H_3)$.

$$\begin{aligned}
d_{s,u} + w(u,v) &> d_{s,v} \quad \text{da } H_1 \\
d_{s,u} + w(u,v) &> d_{s,x} + w(x,y) + d_{y,v} \quad \text{per } H_2 \\
&\geq d_{s,x} + w(x,y) \quad \text{da pesi } \geq 0 \\
&\geq d_{s,u} + w(u,v) \quad \text{da } H_3
\end{aligned}$$

Arriviamo ad avere che una cosa e' strettamente maggiore di se stessa, il che e' assurdo. □

## 3.9 Algoritmo di Floyd e Warshall

> Algoritmo volto a risolvere il problema **all-pair shortest paths**, ovvero torvare il cammino minimio $\forall x, y \in V$.

Questo algoritmo, scritto tramite la tecnica della programmazione dinamica, puo' essere applicato a grafi non rientati purche' non ci siano cicli negativi. Com'e' tipico di questa tecnica ridurremo il problema principale in sottoproblemi semplici, analizzando solo parti del nostro grafo alla volta, componendo poi le soluzioni per trvoare il risultato finale. Questo funziona poiche' siamo alla ricerca di cammini semplici, useremo quindi i vertici una volta sola.

Sia $D^k_{x,y}$ la distanza minima dal vertice $x$ sorgente a quello $y$ di arrivo, nell'ipotesi in cui eventuali nodi intermedi possano essere solo quelli $V[1..k]$, ossia i primi $1..k$ vertici (possiamo immaginare che ad ogni vertice sia associato un numero, un identificatore che va da 1 a $n$ dove $n$ e' il numero di vertici).

La soluzione al nostro problema sara' $D^n_{x,y}$ per ogni coppia di nodi $x$ e $y$, dove $x, y \in \{1, \ldots, n\}, k \in \{0, \ldots, n\}$.

### 3.9.1 Soluzioni per $x = y$ o $k = 0$

Per $k = 0$ abbiamo che dobbiamo raggiungere $y$ da $x$ usando nessun nodo, il che significa che basta controllare se esiste un arco diretto tra i due vertici $(x, y)$.

Per $x = y$ la risposta e' ovvia, i nodi coincidono quindi con 0 spostamenti abbiamo gia' risolto il problema.

### 3.9.2 Soluzioni dei casi non banali

Ora risolviamo il caso generale $D^k_{x,y}$ dove $x \neq \wedge k \geq 1$ assumendo (per la programmazione dinamica) di aver risolto gia' i problemi $D^k_{x',y'} - 1 \forall x', y' \in V$
Abbiamo due situazioni possibili:

1. $V[k]$ **non viene attraversato**: la soluzione e' equivalente a quella del sottoproblema semplice precendete $D^{k-1}_{x,y}$.

2. $V[k]$ **viene attraversato**: se $k$ viene attraversato sappiamo che nel nostro cammino esso compare, quindi si puo' spezzare il cammino in due pezzi, $D^k_{x,y} = D^{k-1}_{x,k} + D^{k-1}_{k,y}$

La soluzione al sottoproblema generale $D^k_{x,y}$ sara' uguale al minimo tra i costi dei due casi, ottenendo dunque:

$$D^k_{x,y} = \min\{D^{k-1}_{x,y}, D^{k-1}_{x,k} + D^{k-1}_{k,y}\}$$

### 3.9.3 Soluzione finale

Mettendo assieme le idee raccolte qui' sopra, possiamo formalizzare la soluzione in modo matematico. Partiamo dal caso in cui $k = 0$ o $x = y$.

$$D^{k=0}_{x,y} = \begin{cases} 0 & \text{se } x = y \\ w(x,y) & \text{se } (x,y) \in E \\ \infty & \text{se } (x,y) \notin E \end{cases}$$

Per il caso non banale invece possiamo usare la formula enunciata in precedenza:

$$D^{k\geq 1}_{x,y} = \min\{D^{k-1}_{x,y}, D^{k-1}_{x,k} + D^{k-1}_{k,y}\}$$

**Algorithm:** Algoritmo di Floyd-Warshall

```
Data: G = (V, E, w)
Result: double[]
begin
    int n ← len(V)
    /* cube matirx where all paths are stored as k increases
       */
    double D[1..n][1..n][0..n]
    for x ← 1 to n do
        for y ← 1 to n do
            /* base case                                          */
            if x = y then
                D[x][y][0] ← 0
            else if (x, y) ∈ E then
                D[x][y][0] ← w(x,y)
            else
                D[x][y][0] ← ∞
            end
        end
    end
    for k ← 1 to n do
        for x ← 1 to n do
            for y ← 1 to n do
                /* the body of the loop is the complex-case, an
                   algorithmic rewrite of the min function in
                   the bastract problem                           */
                D[x][y][k] ← D[x][y][k-1]
                if D[x][y][k] > D[x][k][k-1] + D[k][y][k-1] then
                    D[x][y][k] ← D[x][k][k-1] + D[k][y][k-1]
                end
            end
        end
    end
    /* return the last filled table                               */
    return D[1..n][1..n][n]
end
```

Si puo' notare poi come l'algoritmo possa essere ottimizzato dal punto dello spazio poiche' si possono fare modifiche in place, in quanto le modifiche applicate non cambiano i valori che possono tornarci utili nella tabella precedente $k-1$.

Eccone dunque una versione ottimizzata dal punto di vista dello spazio:

**Algorithm:** Algoritmo di Floyd-Warshall (ottimizzato)

```
Data: G = (V, E, w)
Result: double[]
begin
    int n ← len(V)
    /* cube matirx where all paths are stored (all variations
       in one as k increases)                                    */
    double D[1..n][1..n], x, y
    for x ← 1 to n do
        for y ← 1 to n do
            /* base case                                        */
            if x = y then
                D[x][y] ← 0
            else if (x, y) ∈ E then
                D[x][y] ← w(x,y)
            else
                D[x][y] ← ∞
            end
        end
    end
    for k ← 1 to n do
        for x ← 1 to n do
            for y ← 1 to n do
                /* the body of the loop is the complex-case, an
                   algorithmic rewrite of the min function in
                   the bastract problem                         */
                if D[x][y] > D[x][k] + D[k][y] then
                    D[x][y] ← D[x][k] + D[k][y]
                end
            end
        end
    end
    /* return the last filled table                             */
    return D[1..n][1..n]
end
```

### 3.9.4 Ricorstruzione dei cammini

Per ricorstruire i cammini si usera' una struttura dati aggiuntiva, un array $n \times n$, chiamata *next* che contiene in positione $next[x][y]$ *il secondo nodo* necessario per andare da $x$ a $y$ (il primo nodo e' $x$ e l'ultimo e' $y$). Ecco una ulteriore riscrittura

dell'algoritmo per tenere traccia del percorso.

**Algorithm:** Algoritmo di Floyd-Warshall (ottimizzato)

```
Data: G = (V, E, w)
Result: double[]
begin
    int n ← len(V)
    /* cube matirx where all paths are stored, and cube
       matrix for keeping track of each step in the paths  */
    double D[1..n][1..n], next[1..n][1..n] x, y
    for x ← 1 to n do
        for y ← 1 to n do
            /* base case                                    */
            if x = y then
                D[x][y] ← 0
                next[x][y] ← -1
            else if (x, y) ∈ E then
                D[x][y] ← w(x,y)
                next[x][y] ← y
            else
                D[x][y] ← ∞
                next[x][y] ← -1
            end
        end
    end
    for k ← 1 to n do
        for x ← 1 to n do
            for y ← 1 to n do
                /* the body of the loop is the complex-case, an
                   algorithmic rewrite of the min function in
                   the bastract problem                       */
                if D[x][y] > D[x][k] + D[k][y] then
                    D[x][y] ← D[x][k] + D[k][y]
                    next[x][y] ← next[x][k]
                end
            end
        end
    end
    /* return the last filled table                         */
    return D[1..n][1..n]
end
```