

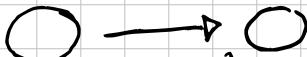
# GRAFI

Un grafo è definito come una coppia  $[V, E]$  in cui

- $V$  è un insieme di nodi, detti vertici
- $E$  è una collezione di coppie di vertici detti archi

Ci sono 2 tipi di archi:

- Diretti: vi è una coppia ordinata di vertici, in cui vi è un origine ed una fine



- Indiretti: NON c'è ordinamento di vertice, non c'è un origine né fine. Possono essere "percorsi" da entrambi le parti.



Applicazioni:

Mapppe: es. google maps o anche mappe di rete

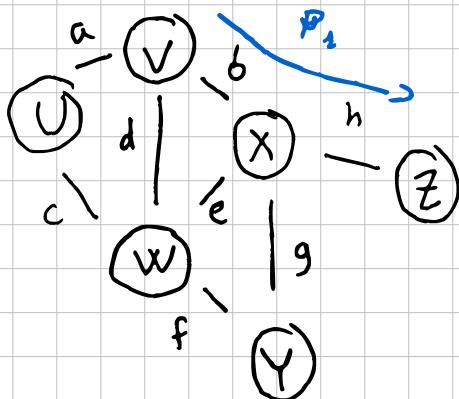
Un CAMMINO è

Una sequenza alternata di vertici ed archi  
in cui si inizia e finisce con un vertice.

È un cammino SEMPLICE se tutti i vertici  
e gli archi vengono "visitati" una sola volta

$$P_i = (V, b, X, h, Z)$$

è un cammino  
semplice



il grado di un vertice è pari al numero di archi  
toccanti lo stesso.

ES

$$\deg(X) = h$$

# Proprietà

1)  $\sum \deg(v) = 2m$

i gradi dei vertici  $v$  sono sempre  $2m$ , in quanto ogni arco si conta 2 volte

2) in un grafo non diretto senza capi e archi paralleli:

$$m \leq n(n-1)/2$$

in quanto ogni vertice ha grado al più  $n-1$

=

Un grafo è una collezione di vertici e archi.  
Saranno 3 tipi di dato

Vertex

Edge

Graph

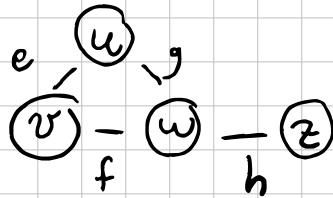
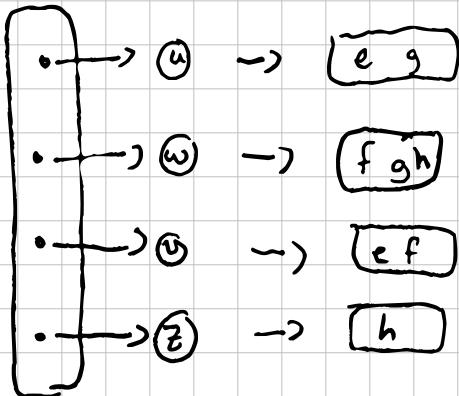
Un vertice (Vertex) è un oggetto che memorizza un certo dato scelto dall'utente

Un arco (Edge) memorizza un oggetto associato fornito col metodo element()

## Strutture dati

- Struttura a lista di dati (edge list) lista non ordinata che contiene tutti i lati, intesi come coppia di vertici, non ordinata.  
INEFFICIENTE
- Liste di Adiacenza, in cui per ogni vertice si aggiunge una lista con tutti i vertici adiacenti.  
Utile per trovare tutti i vertici collegati ad un determinato vertice.

# Liste di adiacenza



ogni nodo della lista punta ad una lista con tutti gli archi collegati al vertice

prestazioni:

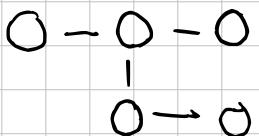
	Lista di archi	Lista d. adiacenza
Spazio		
inArchi( $V$ )	$m$	$\deg(v)$
SonoAdiacenti( $v, w$ )	$m$	$\min(\deg(v), \deg(w))$
InserisciVertice( $o$ )	1	1
InserisciArco( $v, w, o$ )	1	1
rimuoviVertice( $v$ )	$m$	$\deg(v)$
rimuoviArco( $e$ )	1	1

## Albero

Un albero è un grafo  $T$  tale che

- $T$  è connesso (ovvero c'è un cammino fra ogni coppia di vertici)
- $T$  non ha cicli (ergo non ha sequenze circolari)

es.

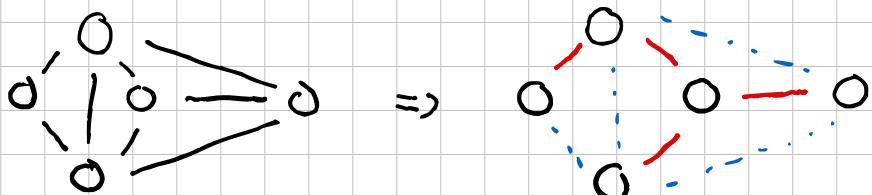


## Forest

Una foresta è invece un grafo NON diretto senza cicli

## Alberi Ricoprenti

Dato un grafo, un albero ricoprente del grafo fa sì che tutti i vertici siano visitabili.



come per visitare, amplizzare e BFS

## Attraversamento in profondità DFS

Il primo algoritmo che viene usato è il DPS, viene usato anche per verificare l'esistenza di un percorso tra 2 nodi.

Si parte da un vertice d. partenza  $s_1$ , che si presuppone non esplorato.

In generale se  $s_i$  indica con  $u$  il vertice attuale, si attraversa  $G$  utilizzando un lato qualsiasi  $(u, v)$  incidente.

Se il vertice  $v$  è stato già visitato si ignora il lato, altrimenti si va su  $v$  e lo si marca come visitato, e lo si rende il vertice attuale.

Si raggiungerà prima o poi un vicolo cieco. Si tornerà indietro fino a trovare un lato non visitato, e lo si visita.

↳ Algoritmo Ricorsivo

Dfs(Grafo  $G$ , Vertice  $v$ )

for (vertice adj:  $v$ .adjacent)

if (!adj.isVisited)

adj = visiting

dfs( $G$ , adj)

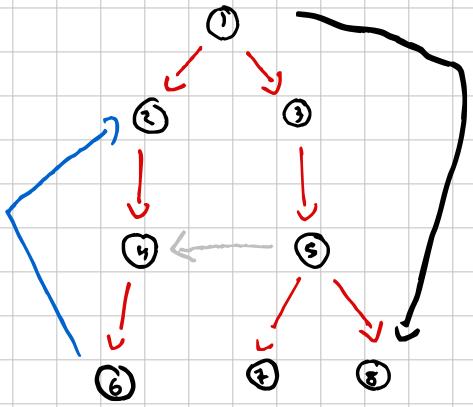
adj = visited

3 categorie

• lato back: connettono vertice ad antenato

- lato forward: connettono vertice a discendente,

• lato cross: connette vertici non app. ad Albero DFS



forward:  
arco descendente  
ma non fa  
parte dell'  
albero DF

discovery

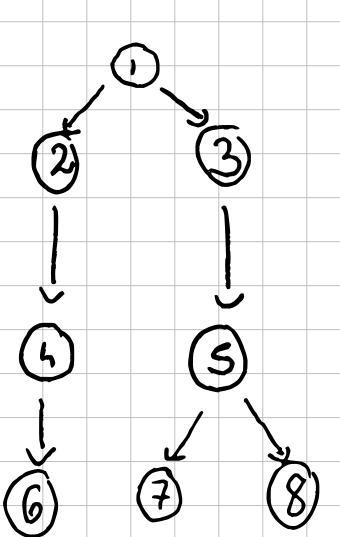
back

[porta a ciclo.  
arco porta in  
nodo DPS NT]

CROSS

[nodo che  
porta a DFS  
terminata  
(no ciclo)]

albero discovery equivalente



↳ formato da soli  
archi discovery

Un arco di tipo back indica che vi è un ciclo, in quanto durante un attraversamento d' alberi un arco porta ad un nodo con etichetta "EXPLORING" dello stesso ramo.

Un arco di tipo forward invece capita quando viene connesso il nodo ad un suo discendente nell'albero, già di tipo "EXPLORED"

## Proprietà DPS

- i) Se  $G$  è un grafo NON diretto in cui si è fatto l'attraversamento DPS e  $S$  il vertice iniziale, allora i lati discovery formano un arco ricoprente del componente连通的.
- ii) Se  $\tilde{G}$  è un grafo diretto in cui DFS dal vertice  $S$ , allora l'albero ha visitato tutti i vertici raggiungibili da  $S$ .
- iii) Sia  $G$  un grafo NON orientato (non diretto) con  $n$  vertici e  $m$  lati. L'attraversamento di  $G$  può essere eseguito in tempo  $O(n+m)$ , e con lo stesso tempo si può:
  - costruire un percorso tra 2 vertici in  $G$ , se esiste
  - verificare se  $G$  è连通的
  - costruire albero ricoprente di  $G$
  - individuare cicli in  $G$  (esistenza nodi back)

L'ALGORITMO DFS visita tutti i nodi del grafo visitabili a partire da  $s$

DIMOSTRAZIONE =

Si indicano con  $V$  il sottoinsieme di vertici di  $G$  visitati da  $s$ .

Si supponga, per a

# Visita in ampiezza

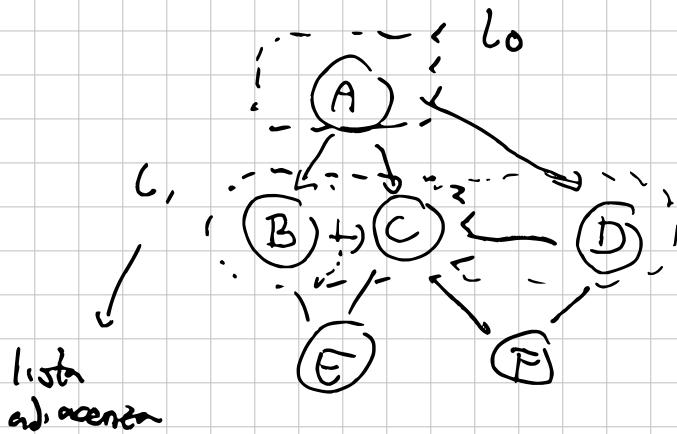
# BFS

Un attraversamento in ampiezza come la BFS divide i vertici in livelli, che indicano la "distanza" dal nodo di partenza S, che c'è nel livello  $L_0$ .

Viene usata la Coda (meccanismo First In - First Out). e l'algoritmo termina quando, esaminando un livello, non vi si trovano più vertici.

## BFS(Grafo g, Vertice S)

```
Coda coda = new Coda();
coda.add(S)
while (!coda.isEmpty()){
    Vertice v = coda.pop(0);
    for (vertice u : v.adiacenti) {
        if (u.status = UNEXPLORED) {
            u.add(u)
        }
    }
}
```



es. alla fine del while (coda.isEmpty()) dopo aver processato A, ci sarà BCD

Level	A	→	Level	BCD
Next	BCD		Next	EP

:

Ottimo algoritmo path finding! (cammini non pesati)

anch'esso ha costo  $O(m+n)$

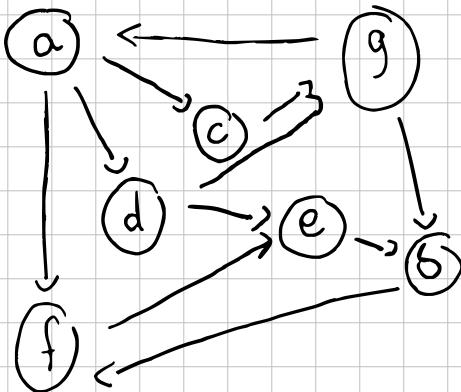
## Proprietà BFS

I) Se  $G$  è un grafo in cui è stato effettuato l'attraversamento BFS, a partire dal vertice  $s$ , allora:

- L'attraversamento ha visitato tutti i vertici di  $G$  che sono raggiungibili da  $s$ .
- $\forall v \in V$ : il percorso che, nell'albero BFS  $T$ , va da  $s$  a  $v$  ha un numero d, lati uguali a:
- se  $(u, v)$  è un arco che non appartiene all'albero, allora il livello di  $v$  può essere al massimo superiore di un'unità del livello di  $u$ .

# CONNETTIVITÀ FORTE

oda ogn. vertice è possibile ogni altro vertice.  
[catene di Markov - IRREDUCIBILITÀ]

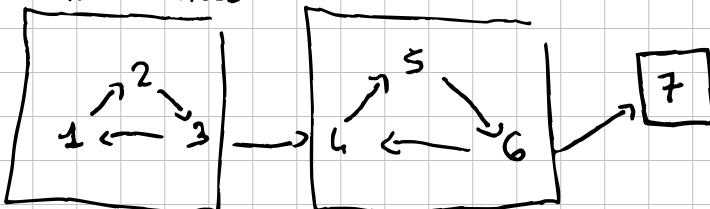


Componenti, fortemente connesse

- si possono cercare SOTTOGRAFI MASSIMALI che sono f. connessi, in cui ogni nodo può raggiungere quals. altro. costituiscono classi d: EQUIVALENZA.

2 vertici sono in relazione se sono MUTUALMENTE RAGGIUNG.

3 componenti connesse fortemente



$$L_1 = \{1, 2, 3\}$$

$$L_2 = \{4, 5, 6\}$$

$$L_3 = \{7\}$$

trovare componente connesse

- 1) Visita in avanti
- 2) Visita in trasposto
- 3) Intersezione delle 2 visite porta a comp. connesse

Necessità di usare transposed DFS

## Chiusura transitiva

Dato un grafo diretto, si aggiungono archi ogni volta da un nodo c'è possibile arrivare ad un altro

↓

La chiusura transitiva di un grafo  $G$  diretto è un grafo  $G^*$  che ha stessi vertici di  $G$ , e ha archi da  $(u, v)$  solo se esiste un percorso che da  $u$  va a  $v$ .

per  $i = 1 \dots n$  il grafo orientato  $G_i$  contiene l'arco  $(V_i, V_j)$  se c'è solo se il grafo orientato  $G$  contiene un percorso da  $V_i$  a  $V_j$ , i cui i vertici intermedi, se esistono sono  $\{V_1, \dots, V_k\}$ .

In particolare  $G_n$  coincide con  $G^*$ , che è la chiusura transitiva di  $G$

Algoritmo Floyd-Warshall  $\rightarrow O(n^3)$

Sia  $v_1, v_2, \dots, v_n$  una numerazione di vertici di  $G$

$G_0 = G$

for  $k$  in range( $1, n$ ) do

$G_k = G_{k-1}$

for Coppia  $i, j$ ,  $i+j+k \leq n$   $\{1, \dots, n\}$

# GRAFI Pesati

Un grafo pesato è un particolare grafo dotato di etichette numeriche associate ad ogni arco, chiamato PESO, weight.

## Shortest Path

Sia  $G$  un grafo pesato, la lunghezza di un percorso  $P$  è definito come la somma degli archi, che collegano gli estremi del percorso.

Se  $P = \{(v_0, v_1), (v_1, v_2), \dots, (v_{n-1}, v_n)\}$  allora  
la lunghezza di  $P$  è

$$w(P) = \sum_{i=0}^{n-1} w(v_i, v_{i+1})$$

La distanza di un vertice da un altro in  $G$  è detta  
il percorso più breve

$$d[v] \leq w(P) \quad \forall P(u, v)$$

## Algoritmo d. Dijkstra

Partendo dal fatto che, ora, ogni nodo avrà associato un parametro per salvare la distanza da un generico nodo, e che partendo da  $v_0$ , nodo  $u$  tutti gli altri avranno  $d[v] = \infty \quad \forall v \neq u \in G$

L'algoritmo consiste nel far crescere una "nuvola di nodi attorno ad  $S$ ", ovvero il nodo di partenza.

Tutti i nodi sull'interno saranno già cammini minimi a partire da  $S$ , e volta per volta si aggiungerà, al nodo più "vicino" ad  $S$ .

## Rilassamento dei lati

Come si è detto, ogni volta che si visita un nodo al di fuori della nuvola  $S$ , si aggiornerà la sua distanza, ANCHE SE non lo si aggiunge alla nuvola stessa.

Questo è necessario per validare se magari, avendo a disposizione più nodi, esiste un percorso migliore per il nodo trovato. L'aggiornamento è chiamato **RILASSAMENTO**.

Questo porta ad una condizione di **OTTIMALITÀ**

$d[u]$  è la distanza minima da  $S$  per ogni  $u$  se e solo se la seguente condizione è soddisfatta.

$$\forall u \in V(u,v) \in E \quad d[u] \leq d[v] + w(u,v)$$

ne seguirà l'algoritmo di rilassamento:

$$\begin{aligned} & \text{if } d[u] + w(u,v) < d[v] \\ & \quad d[v] = d[u] + w(u,v) \end{aligned}$$

Dovendo scegliere volta per volta l'arco il cui peso è maggiore, è utile utilizzare una code prioritaria, aggiornandole ogni volta i pesi cambiano.

## Dijkstra:

shortest Path ( $G, s$ )

input: Graph  $G$  e nodo  $s$  d<sub>i</sub> partenza

output: lunghezza distanza  $(s, v) \forall v \in G$

inizializza  $d[s] = 0 \quad d[v] = \infty$

crea coda prioritaria con nodi e relativi pesi

while ( $!Q.\text{isEmpty}$ ) do

{addVertexNuova}

$u = Q.\text{removeMin}$

for ogni nodo  $(u, v), v \in Q$  do  
{relaxing d<sub>i</sub> u, v}

: if  $d[u] + w(u, v) < d[v]$

$d[v] = d[u] + w(u, v)$

replace<sub>i</sub>(v)

return  $d[v] \forall v \in G$ .

## COSTO

inizializzazione  $O(n) + O(n \log n)$

ciclo while

# iterazioni =  $O(n)$  //  $n$  numero nodi in  $G$

1 iterazione :

- removeMin:  $O(\log n)$

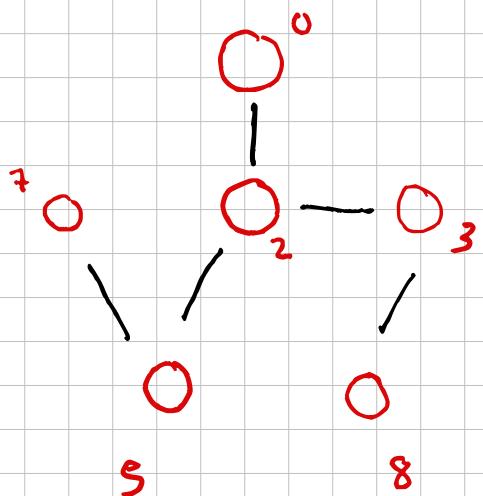
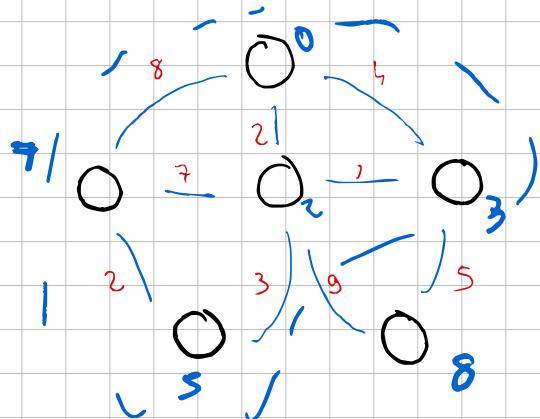
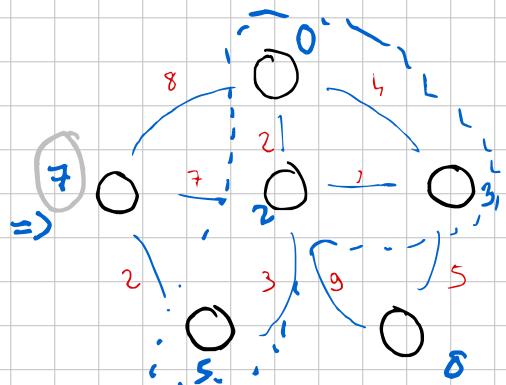
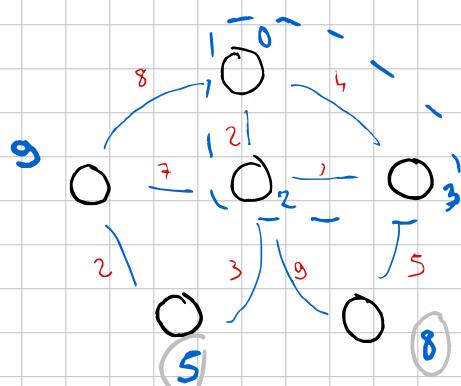
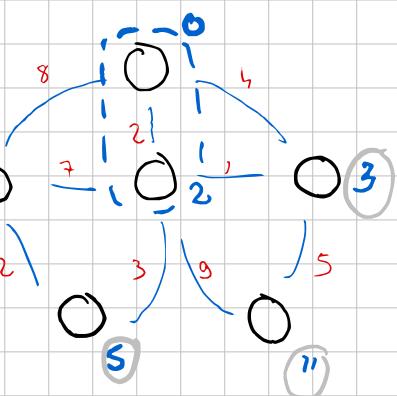
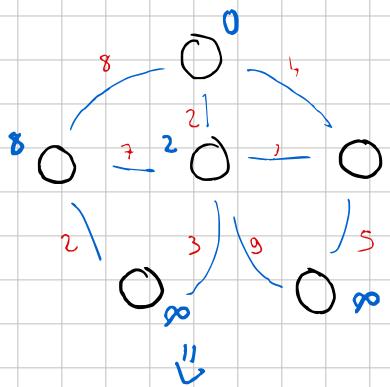
- costo  $O(\deg(v))$

- cambio label  $O(\deg(v) \cdot \log n)$

!!

$O((m+n) \log n)$

$\sum \deg(v) = m$



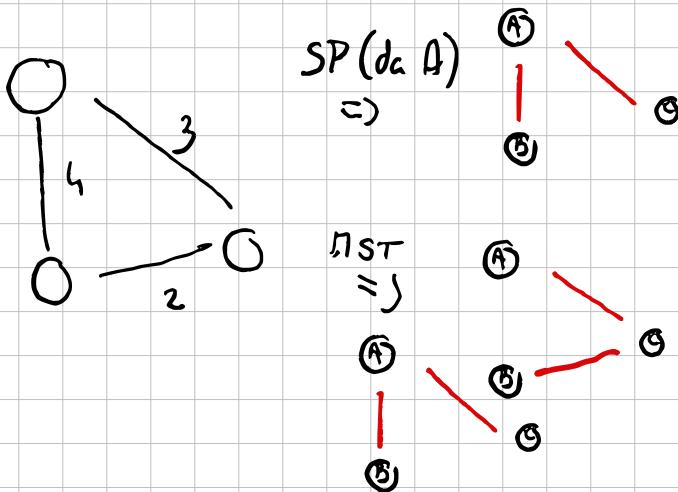
Albero minimo ricoprente

# ALBERI MINIMI RICOPRENTI

Si considera un grafo, diretto e non. Il suo albero minimo ricoprente è un albero (insieme di nodi e archi senza cicli) la cui somma d. pesi è la miglior possibile.

Esso sarà anche unico se tutti gli archi hanno tra loro pesi diversi.

Alberi minimi ricoprenti e alberi Shortest Path sono DIFFERENTI!!! (SP parte da un nodo, MST no)



## PROPRIETÀ:

Ciclo :

dato  $T$  un MST d.  $G$ , e un arco d.  $G \setminus T$   
allora, per ogni arco  $f$  di  $C$ , ovvero un ciclo passante  
in  $T \cup \{e\}$

$$w(f) \leq w(e)$$

## TAGLIO:

S. definisce il taglio come una PARTIZIONE dell'insieme dei vertici  $V$  in  $V_1$  e  $V_2$ .

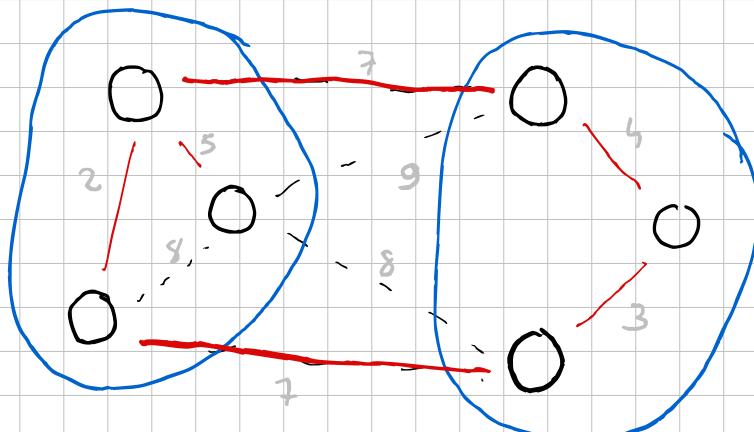
Un arco del taglio è qualsiasi arco che collega vertici da  $V_1$  a  $V_2$

- Sia  $e$  un arco di peso minimo del taglio  $(V_1, V_2)$

Allora ESISTE un MST che contiene  $e$

- vorrà usato nell'algoritmo di Prim-Jarník, simile a quello di Dijkstra -

2 possibili:  
anche MST,  
in questo  
caso se  
ne sceglie  
uno a  
caso



# ALGORITMO DI PRIM-JARNIK

Si comincia con un vertice  $s$  di  $G$  qualsiasi, e da lì si inizia a costruire la "NUVOLA DI VERTICI".  
Po:, per la proprietà di taglio, si sceglie l'arco di taglio e il relativo vertice  $v$ . Il vertice  $v$  viene puntato all'interno della nuvola e il processo si ripete finché non include tutti i vertici raggiungibili da  $s$ .

Algoritmo PrimJarnik( $G$ )

Input: un grafo  $G$  connesso non orientato e pesato, con  $n$  vertici e  $m$  lati.

Output: MST per  $G$

S, scegli  $s \in d[s] = 0$

for  $v$  in vertex( $G$ ) do  
 $d[v] = \infty$

Initialize priorityqueue with  $v$  in vertex( $G$ ) with  $d[v]$  as key.

Init,frasiize T

while Q is notEmpty do

$v = Q.pop$

connect  $v$  to  $T$  with arco e

for each edge  $e^* = (u, v)$  such that  $v \in Q$  do

if  $w(u, v) < d[v]$  do

$d[v] = w(u, v)$

change key of vertex  $v$  to  $d[v]$

return  $T$ .

## COSTO

-inizializzazione  $O(n \log n)$

-costo 1 iterazione while:  $O(\deg(u) \log n) \rightarrow$  aggiornamento di  $Q$  per ogni vicino di  $u$

-costo totale while:

$O((m+n) \log n)$

