Ricerca Operativa
Modulo 2
Teoria dei Grafi: Parte 1

Marco A. Boschetti

Università degli Studi di Bologna
Dipartimento di Matematica
marco.boschetti@unibo.it

Outline

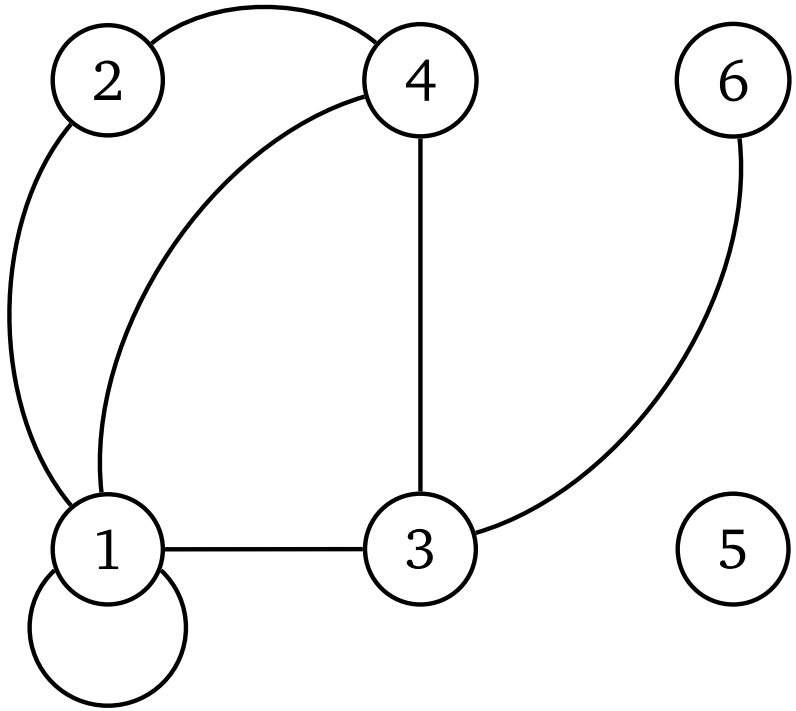
Outline (2)

Grafi non orientati e orientati

- Un grafo non orientato, rappresentato come $G = (V, E)$, è definito dall'insieme dei *vertici* (o *nodi*) e dall'insieme dei *lati* che congiungono coppie non ordinate di vertici:
  - $V = \{1, 2, \ldots, n\}$: insieme dei vertici (o nodi);
  - $E = \{e_1, e_2, \ldots, e_m\}$: insieme dei lati, che corrispondono a coppie *non ordinate* di vertici di V che sono *collegati*, i.e., un lato $e_k = \{i, j\}$ *collega* i vertici i e j.

- Un grafo orientato (o grafo diretto) $G = (V, A)$ si differenzia da un grafo non orientato per la sostituzione dell'insieme dei lati con l'insieme degli *archi*, che sono coppie *ordinate* di vertici:
  - $V = \{1, 2, \ldots, n\}$: insieme dei vertici (o nodi);
  - $A = \{a_1, a_2, \ldots, a_m\}$: insieme degli archi, che corrisponde a coppie *ordinate* di vertici di V, i.e., l'arco $a_k = (i, j)$ indica che il vertice i è collegato al vertice j.

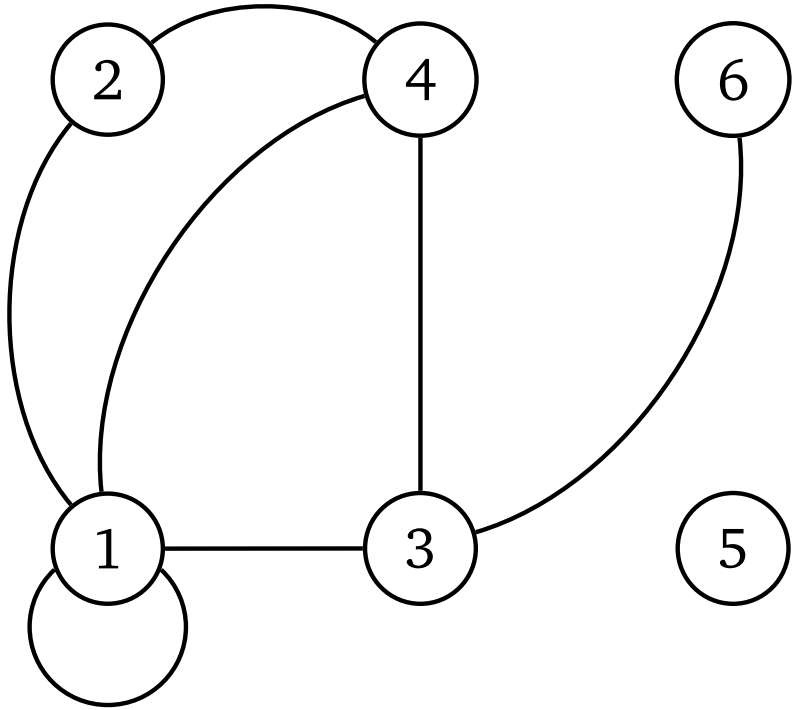
Esempio di grafo non orientato



$$V = \{1,2,3,4,5,6\}$$
$$E = \{\{1,1\},\{1,2\},\{1,4\},\{1,3\},\{2,4\},\{3,4\},\{3,6\}\}$$

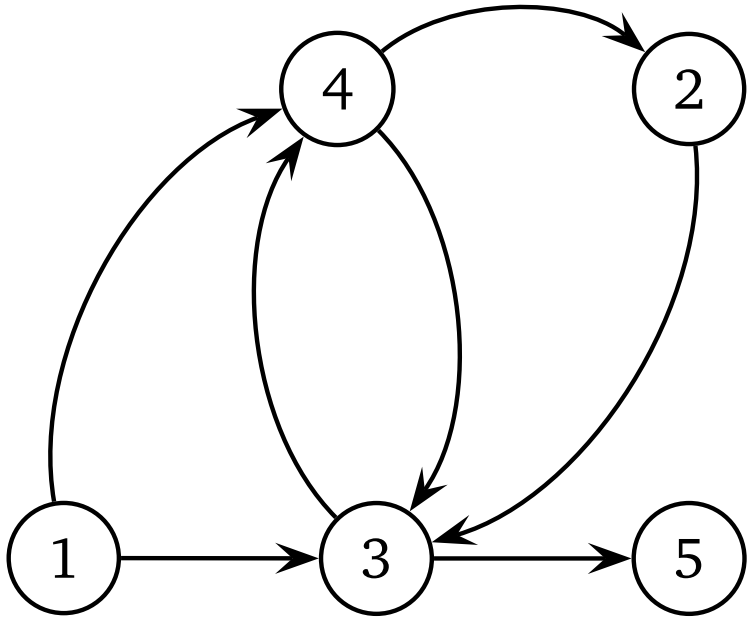
- Il lato $\{i,j\}$ *collega* i e j. Due vertici sono *adiacenti* se esiste il lato che li collega. Due lati sono *consecutivi* se hanno un vertice in comune.
- Il grafo ha un *loop* (lato $\{1,1\}$), anche detto *autoanello* o *cappio*.

Esempio di grafo non orientato



- Si denota con $E(S)$ l'insieme dei lati con entrambi gli estremi nel sottoinsieme di vertici $S \subseteq V$ e $\Gamma(i)$ insieme dei vertici collegati a i.
- Se $S = \{1, 2, 4\}$ allora $E(S) = \{\{1, 1\}, \{1, 2\}, \{1, 4\}, \{2, 4\}\}$.
- $\Gamma(2) = \{1, 4\}$, $\Gamma(4) = \{1, 2, 3\}$, $\Gamma(5) = \emptyset$.

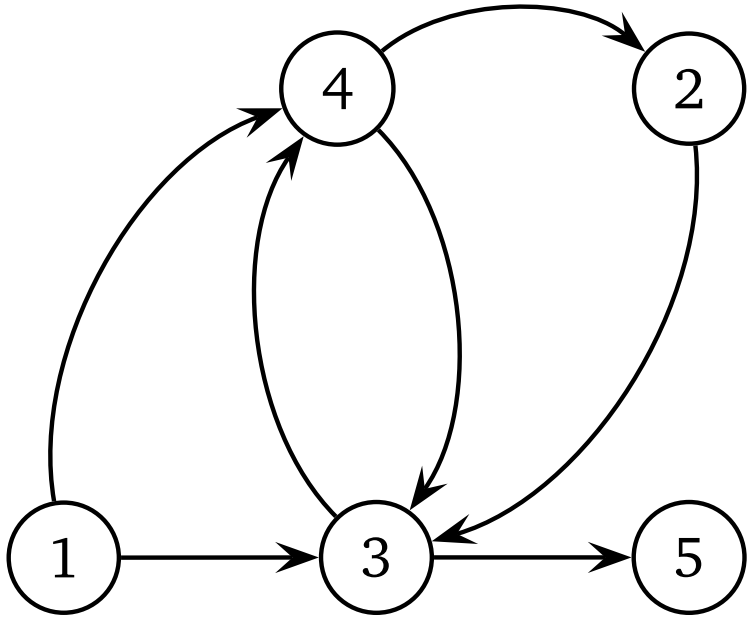
Esempio grafo orientato



$V = \{1,2,3,4,5\} \quad A = \{(1,4),(1,3),(3,4),(4,3),(4,2),(2,3),(3,5)\}$

- L'arco $(1,4)$ *esce* dal vertice 1 e *entra* nel vertice 4.
- Dato l'arco (i,j) il vertice i è detto *vertice iniziale* (*coda* oppure *tail*) e j è detto *vertice terminale* (*testa* oppure *head*). Il vertice j è anche detto *successore* di i mentre i è detto *predecessore* di j.

Esempio grafo orientato

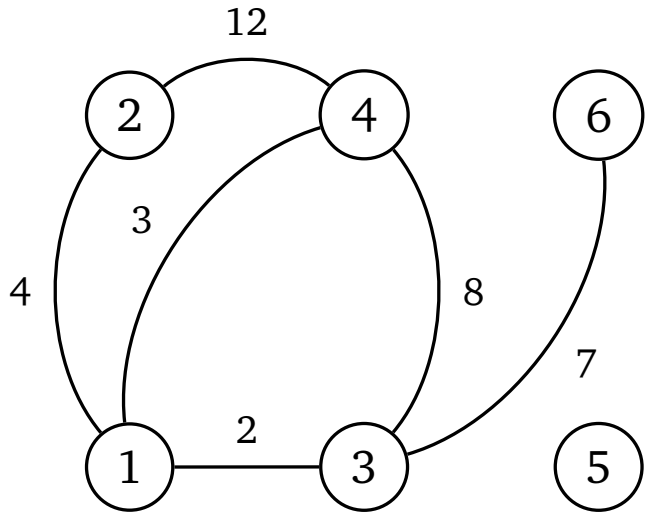


- Si denota con $A(S)$ l'insieme degli archi con entrambi gli estremi (vertice iniziale e finale) nel sottoinsieme di vertici $S \subseteq V$ e con $\Gamma^+(i)$ e $\Gamma^-(i)$ gli insiemi dei successori e dei predecessori di i.
- Se $S = \{1,3,4\}$, allora $A(S) = \{(1,4),(1,3),(3,4),(4,3)\}$.
- $\Gamma^+(1) = \{3,4\}$, $\Gamma^+(4) = \{2,3\}$, $\Gamma^+(5) = \emptyset$, mentre $\Gamma^-(1) = \emptyset$, $\Gamma^-(4) = \{1,3\}$, $\Gamma^-(5) = \{3\}$.

Grafi pesati (non orientati e orientati)

- Il grafo G non orientato (orientato) è pesato sui lati (archi) se esiste una funzione $c : E \to R$ ($c : A \to R$) che associa un valore (o *peso*) ad ogni lato (arco).
- Il grafo G è pesato sui vertici se esiste una funzione $w : V \to R$ che associa un valore (*peso*) ad ogni vertice.

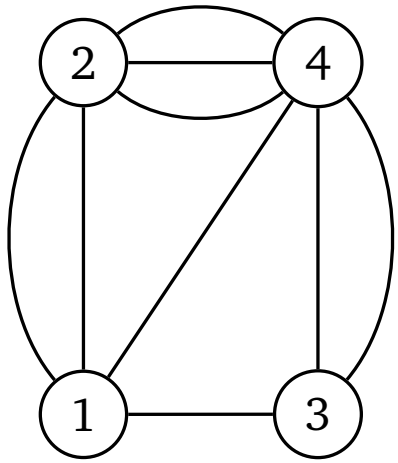
Esempio: grafo non orientato pesato



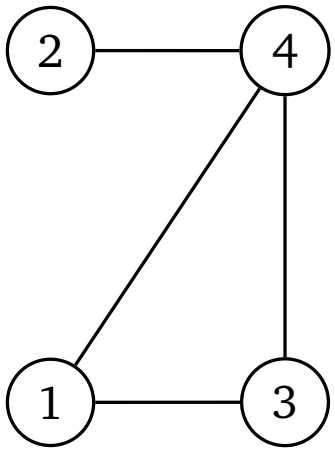
Grafi multipli, semplici e completi

- Un grafo è *multiplo* se può avere più di un lato per la stessa coppia di vertici.
- Un grafo è *semplice* se non comprende loop e lati multipli.
- Generalmente considereremo solo grafi semplici.
- Un grafo è *completo* se per ogni coppia di vertici esiste un lato.

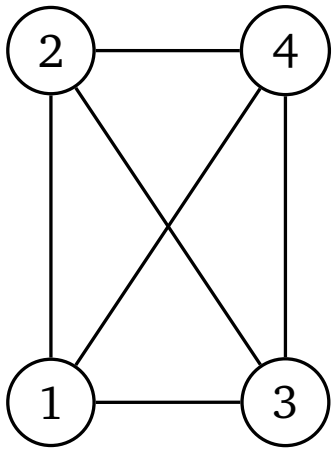
Esempi



(a) Grafo multiplo



(b) Grafo semplice



(c) Grafo completo

Grafi: Applicazioni

Tra gli argomenti più noti nell'ambito della teoria dei grafi possiamo citare ad esempio:

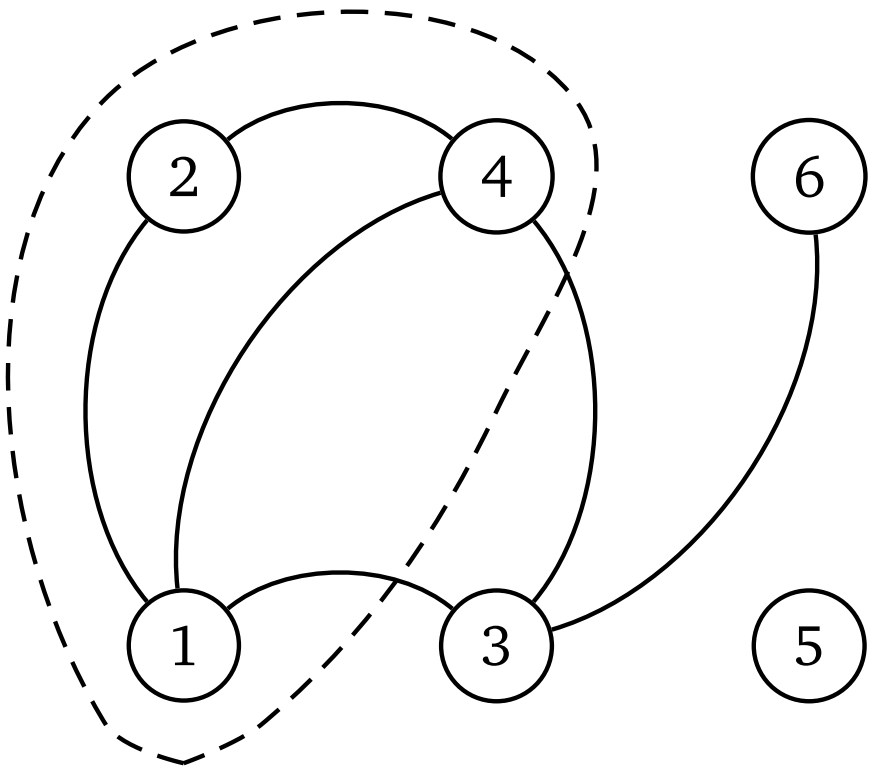
- Cammini Euleriani: originato dal problema posto da Eulero, per determinare un percorso che, partendo da una qualsiasi delle quattro zone della città di Könisberg, attraversasse tutti i sette ponti una ed una sola volta ritornando al punto di partenza.
- Colorazione dei grafi: dove un esempio di applicazione e la colorazione delle mappe per garantire di non usare lo stesso colore per nazioni confinanti.
- Problema della clique (cricca): per esempio per calcolare la clique (i.e., sottografo completo) di cardinalità massima.

Grafi: Applicazioni Reali (Reti Fisiche)

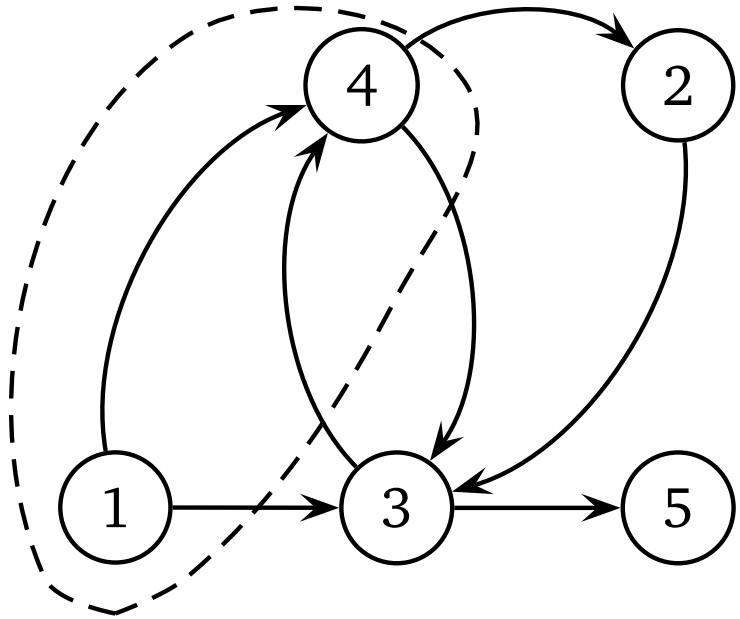
Applications	Physical Analog of Nodes	Physical Analog of Arcs	Flow
Communication systems	Telephone exchanges, computers, transmission facilities, satellites	Cables, fiber optic links, microwave relay links	Voice messages, data, video transmissions
Hydraulic systems	Pumping stations, reservoirs, lakes	Pipelines	Water, gas, oil, hydraulic fluids
Integrated computer circuits	Gates, registers, processors	Wires	Electrical current
Mechanical systems	Joints	Rods, beams, springs	Heat, energy
Transportation systems	Intersections, airports, rail yards	Highways, railbeds, airline routes	Passengers, freight, vehicles, operators

Taglio di un grafo

- Dato un sottoinsieme S di vertici, si dice *taglio* l'insieme dei lati (o archi) che congiungono i vertici in S con quelli in $V \setminus S$.



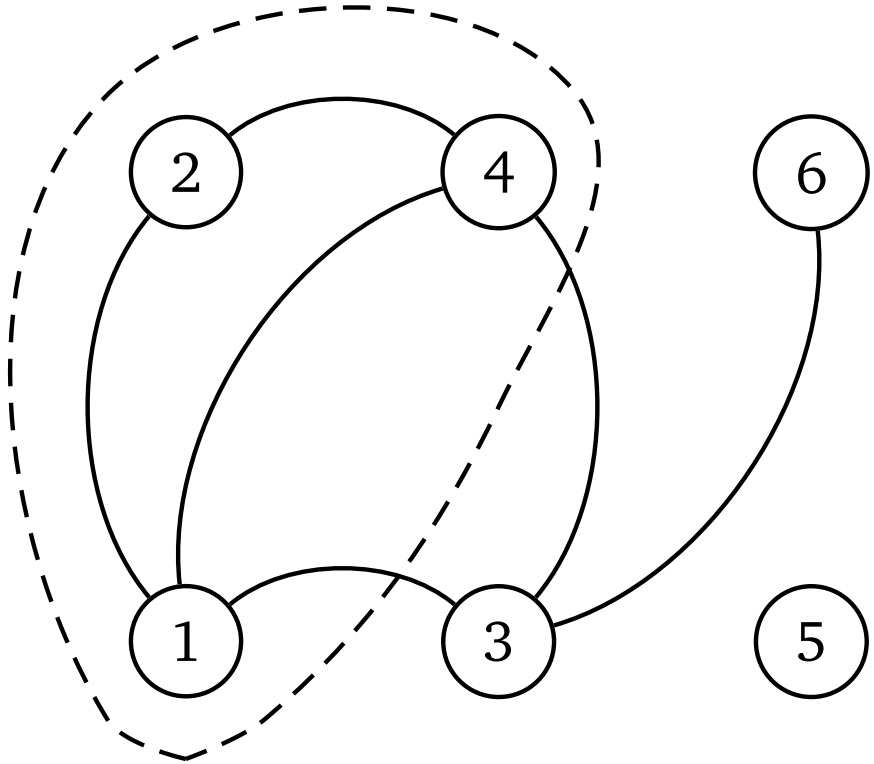
(a) Grafo non orientato



(b) Grafo orientato

Taglio di un grafo non orientato

- Per i grafi non orientati:
  $\delta_G(S) = \{\{i,j\} \in E : i \in S, j \in V \setminus S \text{ oppure } j \in S, i \in V \setminus S\}$.

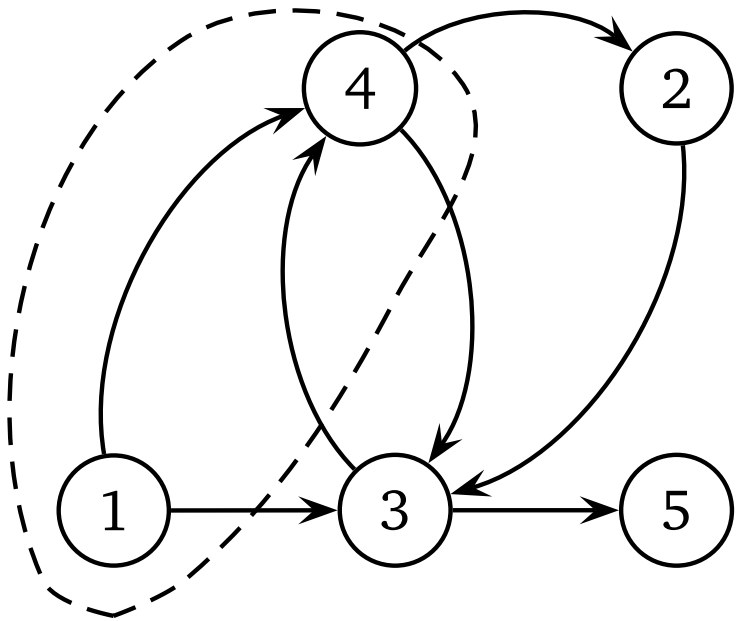


Nell'esempio $\delta_G(\{1,2,4\}) = \{(1,3),(3,4)\}$.

Taglio di un grafo orientato

- Nei grafi orientati distinguiamo tra archi uscenti ed entranti in $S \subset V$:
  - $\delta_G^+(S) = \{(i,j) \in A : i \in S, j \notin S\}$;
  - $\delta_G^-(S) = \{(i,j) \in A : j \in S, i \notin S\}$.

  Si noti che $\delta_G^+(S) \equiv \delta_G^-(V \setminus S)$.



Nell'esempio $\delta_G^+(\{1,4\}) = \{(1,3),(4,2),(4,3)\}$ e $\delta_G^-(\{1,4\}) = \{(3,4)\}$.

Cammini

- Un cammino è una sequenza di vertici $v_1, v_2, \ldots, v_k \in V$ tale che per ogni coppia di vertici consecutivi (v_i, v_{i+1}) esiste il corrispondente lato (grafo non orientato) o arco (grafo orientato).
- Un cammino P si può rappresentare sia come una sequenza di vertici:

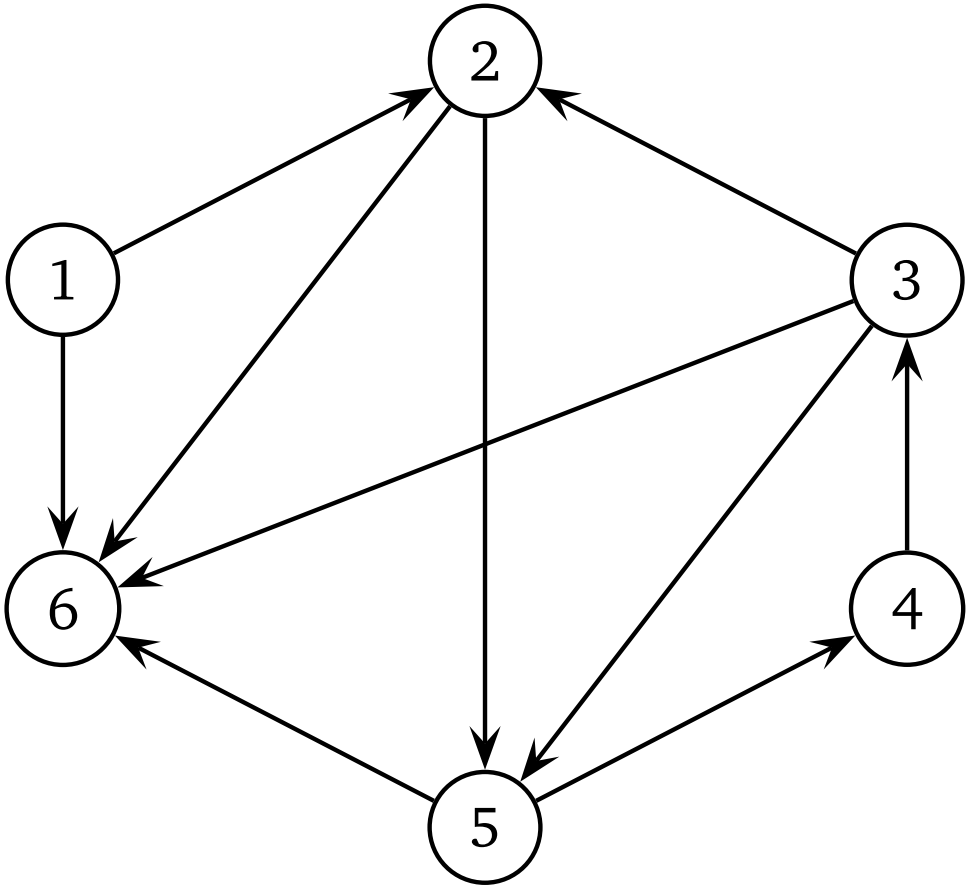
$$P = (v_1, v_2, v_3, \ldots, v_k)$$

- Un cammino può essere rappresentato anche come una sequenza archi (o lati):

$$P = ((v_1, v_2), (v_2, v_3), (v_3, v_4), \ldots, (v_{k-1}, v_k))$$

- In generale non ci sono vincoli che impediscono di visitare più volte alcuni vertici o percorrere più volte alcuni archi (o lati).

Cammini: Esempi



Esempi di cammini:

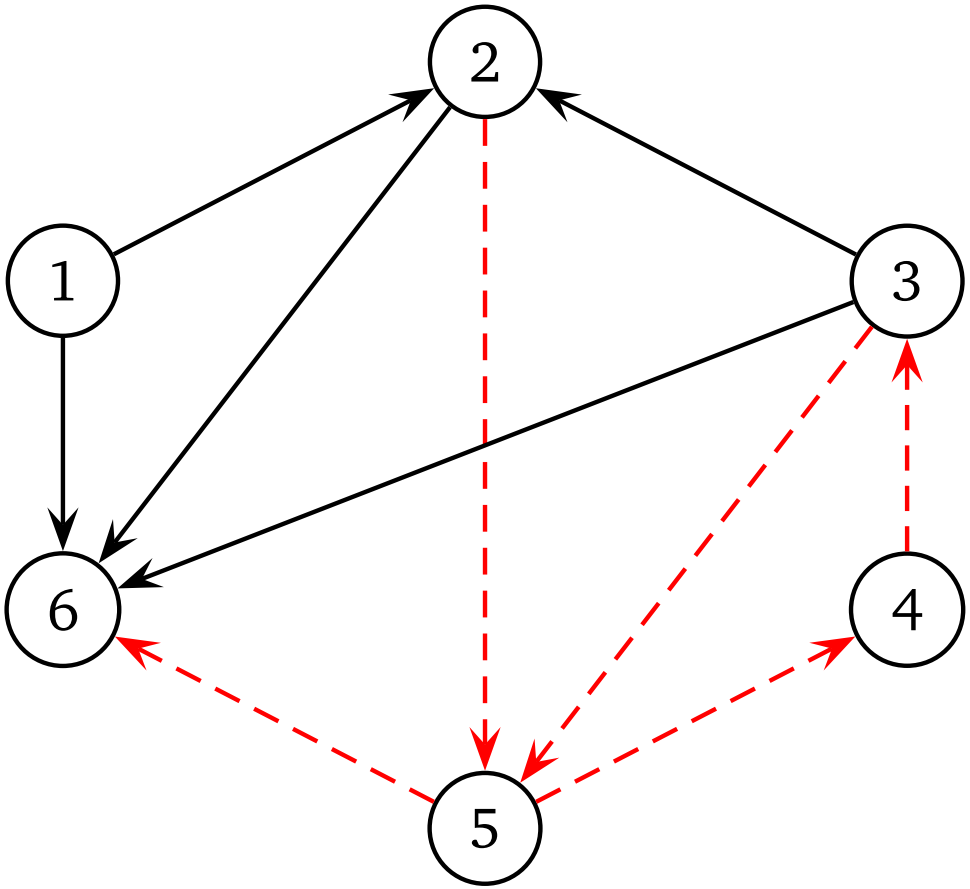
$P_1 = (2, 5, 4, 3, 5, 6)$

$P_2 = (1, 2, 5, 4, 3)$

$P_3 = (1, 2, 5, 4, 3, 2, 5)$

$P_4 = (3, 5, 4, 3)$

Cammini: Esempi



Esempi di cammini:

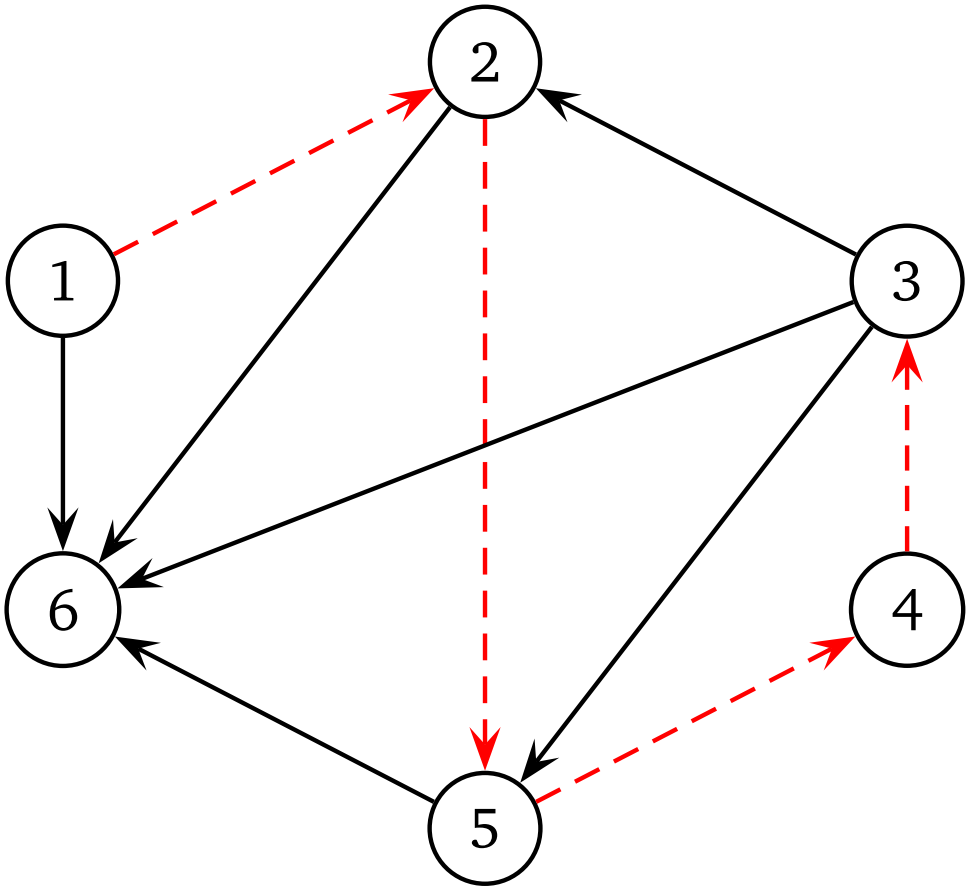
$P_1 = (2, 5, 4, 3, 5, 6)$

$P_2 = (1, 2, 5, 4, 3)$

$P_3 = (1, 2, 5, 4, 3, 2, 5)$

$P_4 = (3, 5, 4, 3)$

Cammini: Esempi



Esempi di cammini:

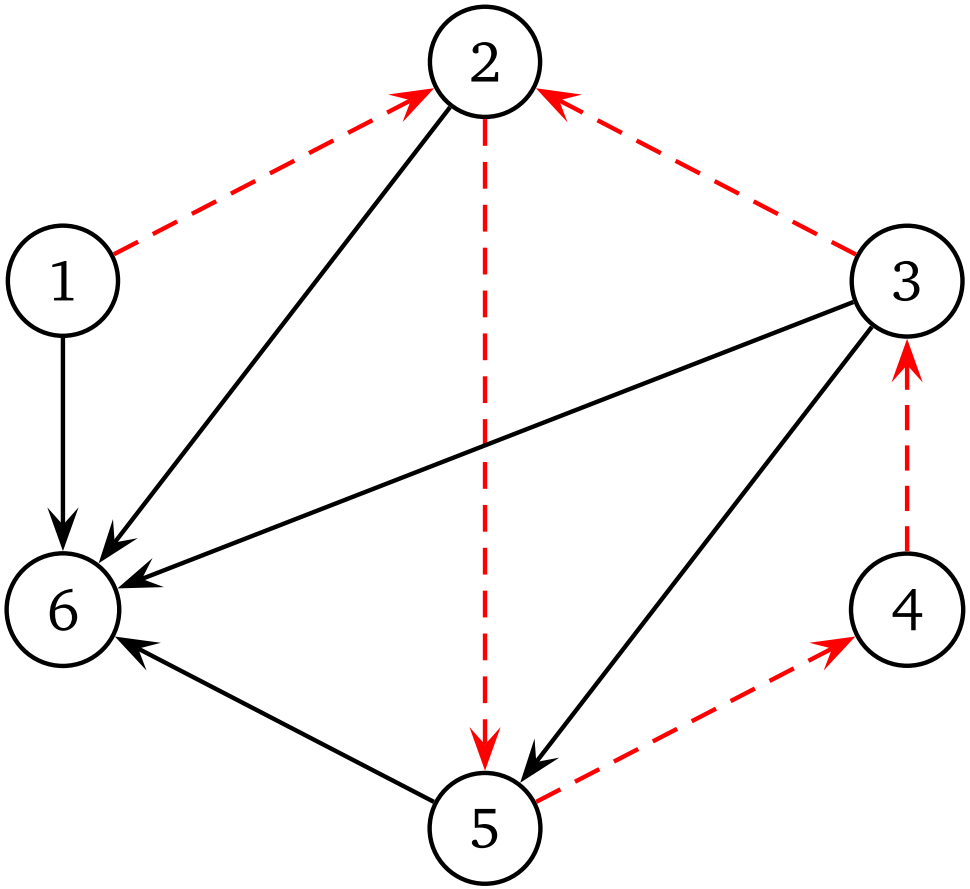
$P_1 = (2, 5, 4, 3, 5, 6)$

$P_2 = (1, 2, 5, 4, 3)$

$P_3 = (1, 2, 5, 4, 3, 2, 5)$

$P_4 = (3, 5, 4, 3)$

Cammini: Esempi



Esempi di cammini:

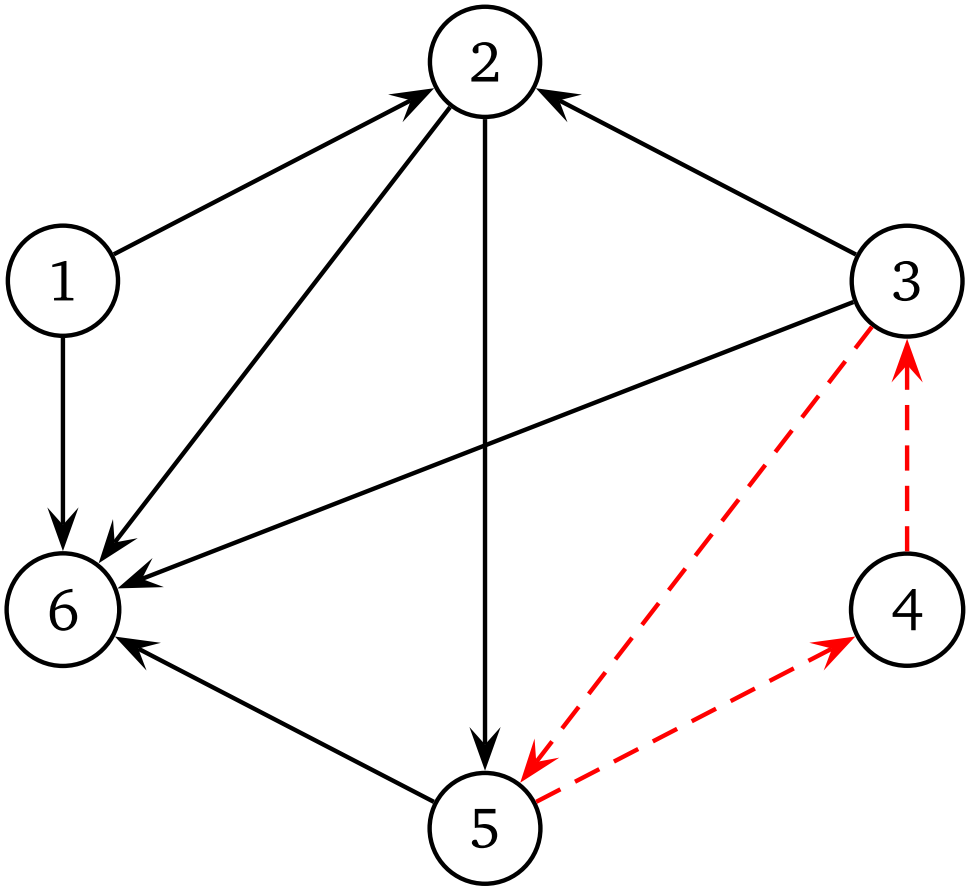
$P_1 = (2, 5, 4, 3, 5, 6)$

$P_2 = (1, 2, 5, 4, 3)$

$P_3 = (1, 2, 5, 4, 3, 2, 5)$

$P_4 = (3, 5, 4, 3)$

Cammini:Esempi



Esempi di cammini:

$P_1 = (2, 5, 4, 3, 5, 6)$

$P_2 = (1, 2, 5, 4, 3)$

$P_3 = (1, 2, 5, 4, 3, 2, 5)$

$P_4 = (3, 5, 4, 3)$

Costo di un cammino

- Dato un cammino $P = (v_1, v_2, v_3, \ldots, v_k)$ il suo *costo* $c(P)$ è dato da:

$$c(P) = \sum_{i=1}^{k-1} c_{v_i v_{i+1}}$$

  dove c_{ij} è il costo dell'arco (i,j).

- Il costo di un cammino, a seconda del contesto e dell'applicazione, è anche detto *lunghezza*, *peso*, etc.

- Per esempio, se il costo di ciascun arco (i,j) corrisponde al tempo necessario per spostarsi dalla località i alla località j, allora il costo del cammino corrisponde al tempo necessario per visitare le località v_i, $i = 1, \ldots, k$, nell'ordine indicato dal cammino.

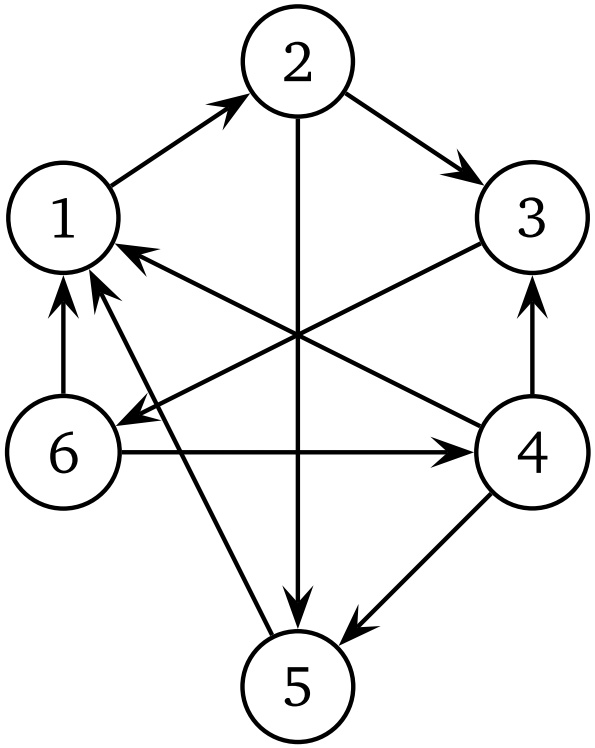
Cammini, circuiti e cicli

- Cammino semplice: non usa più di una volta lo stesso arco/lato. (P_1, P_2 e P_4 sono semplici; P_3 no)
- Cammino elementare: non passa più di una volta per lo stesso vertice. (P_2 è elementare; P_1, P_3 e P_4 no)
- Cammino hamiltoniano: usa una ed una sola volta tutti i vertici del grafo; quindi deve visitare tutti vertici del grafo.
- Cammino euleriano: usa una ed una sola volta tutti gli archi/lati del grafo.
- Circuito: in un grafo orientato è un cammino in cui il vertice iniziale coincide con il vertice terminale.
- Ciclo: controparte non orientata di un circuito.

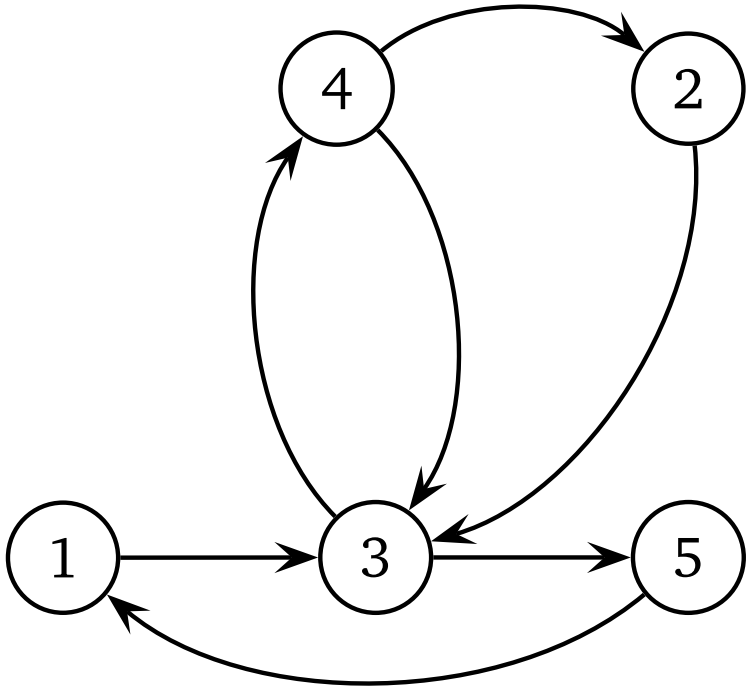
Cammini, circuiti e cicli (2)

- Circuito elementare: è un circuito che, a parte il primo e l'ultimo vertice (che coincidono), non passa più di una volta per lo stesso vertice.
- Circuito hamiltoniano: è un circuito elementare che passa attraverso ogni vertice del grafo. Oppure, equivalentemente, è un cammino hamiltoniano chiuso (i.e., con un arco che collega l'ultimo vertice con il primo del cammino).
- Circuito euleriano: è un circuito elementare che passa attraverso ogni arco del grafo. Oppure, equivalentemente è un cammino euleriano chiuso.
- I grafi che possiedono almeno un circuito/ciclo hamiltoniano sono detti grafi hamiltoniani. Invece, i grafi che possiedono almeno un circuito/ciclo euleriano sono detti grafi euleriani.

Esempio di Circuiti



(a) Grafo hamiltoniano

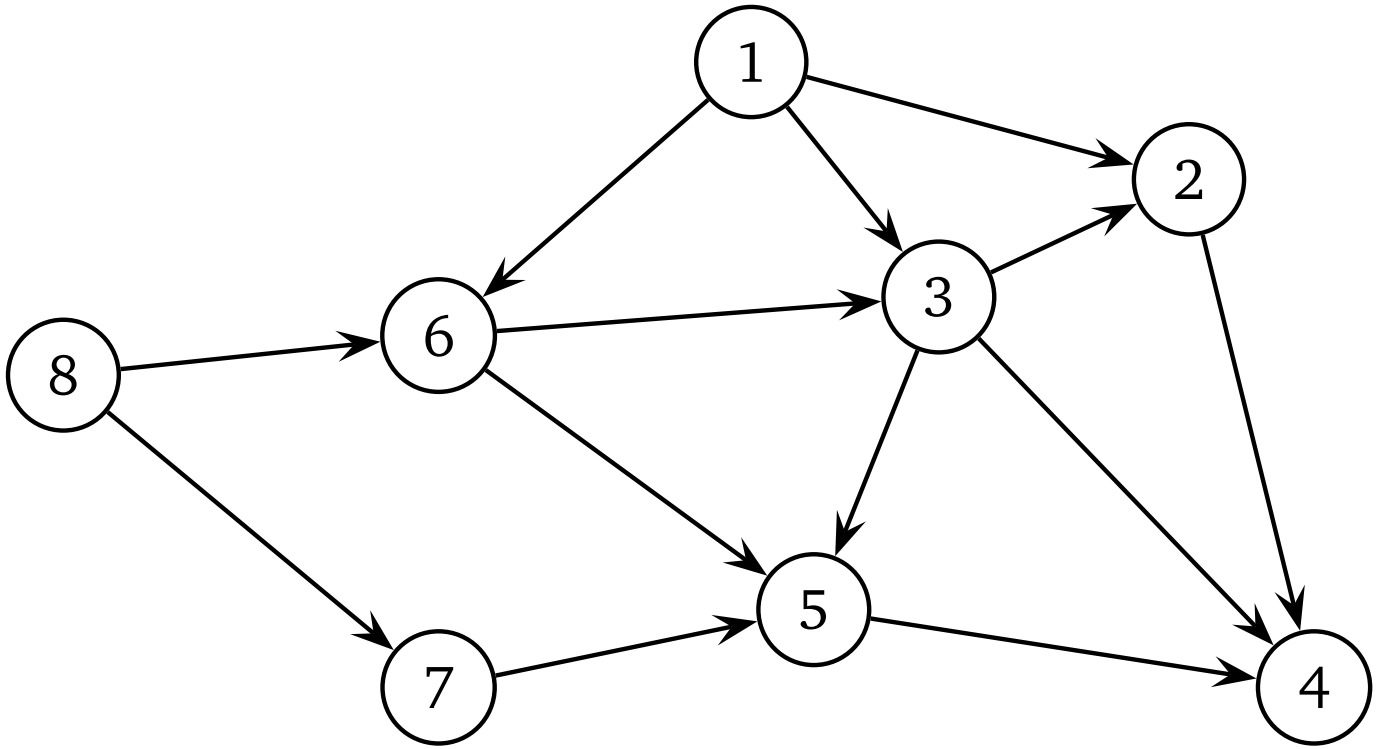


(b) Grafo non hamiltoniano

Circuito elementare (a) $C_1 = (1, 2, 3, 6, 1)$, (b) $C_2 = (3, 4, 2, 3)$.
Circuito hamiltoniano (a) $C_3 = (1, 2, 3, 6, 4, 5, 1)$, (b) non ne possiede.

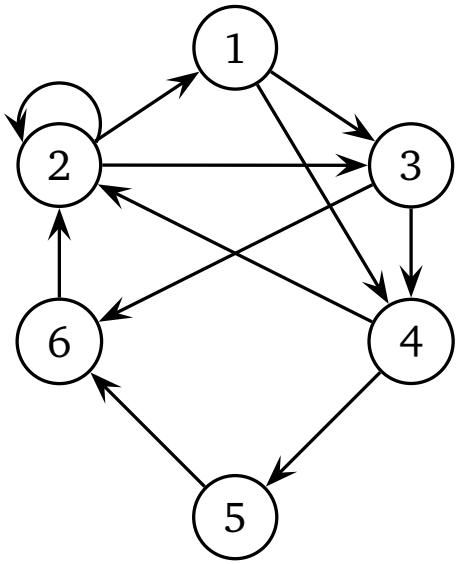
Grafi Aciclici

- Grafo aciclico: è un grafo che non contiene circuiti (cicli).

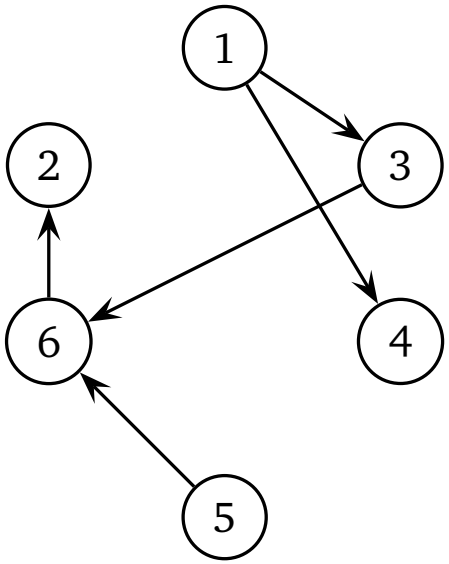


Grafi Parziali e Sottografi

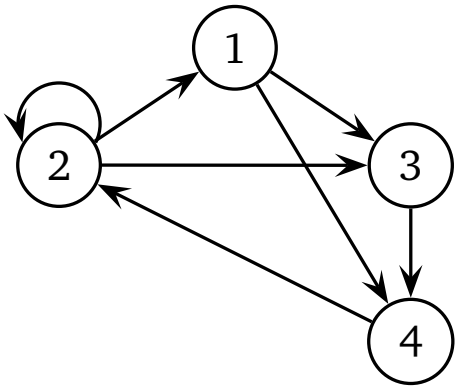
- Grafo parziale di $G = (V, A)$: è il grafo $G' = (V, A')$ dove $A' \subset A$.
- Sottografo di $G = (V, A)$: è il grafo $G' = (V', A')$ dove $V' \subseteq V$ e $A' \subseteq A$.



(a) Grafo orientato



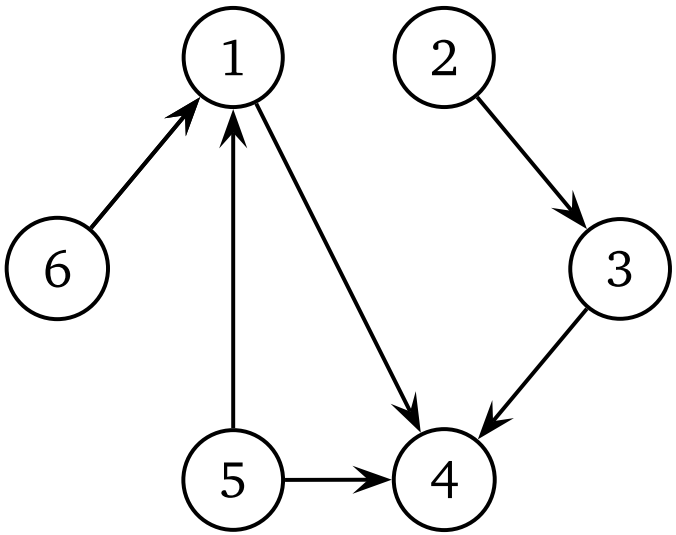
(b) Grafo parziale



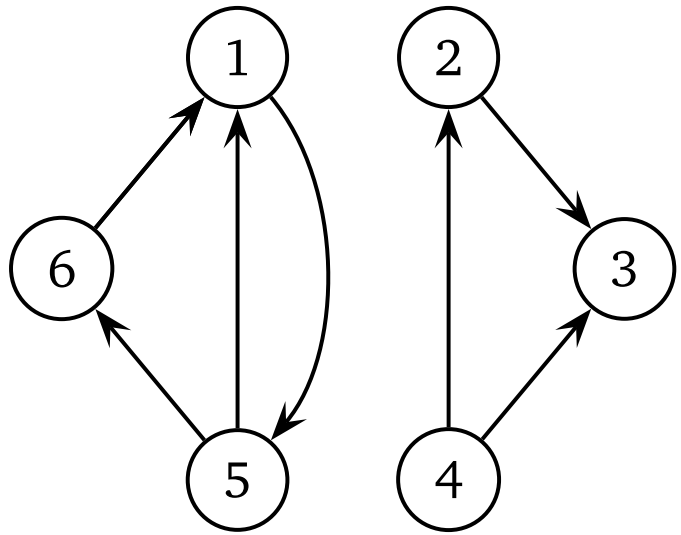
(c) Sottografo

Connessioni e Componenti di un Grafo Orientato

- Grafo connesso: se il grafo *non orientato* relativo al grafo orientato ha almeno un cammino che congiunge ogni coppia di vertici.
- Se tale cammino non esiste allora il grafo viene detto disconnesso.



(a) Grafo connesso



(b) Grafo disconnesso

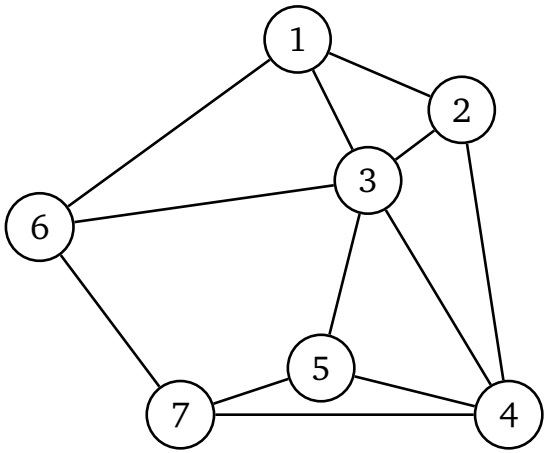
- Grafo fortemente connesso: se nel grafo esiste almeno un cammino orientato che congiunge ogni coppia di vertici.

Alberi

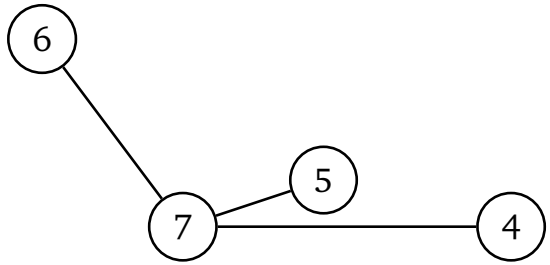
- Un grafo G_a non orientato di n vertici è un albero se rispetta le seguenti condizioni, che sono equivalenti:
  - G_a è connesso e aciclico;
  - G_a è aciclico e si crea un ciclo semplice se si aggiunge un lato al grafo G_a;
  - G_a è connesso, ma diventa non connesso non appena si elimina un solo lato di G_a;
  - G_a è connesso a ha $n-1$ lati;
  - G_a non ha cicli semplici e ha $n-1$ lati.
- In letteratura esistono anche altre condizioni equivalenti. Ognuna di queste definizioni equivalenti può essere utile a "identificare" e "utilizzare" gli alberi.

Alberi (2)

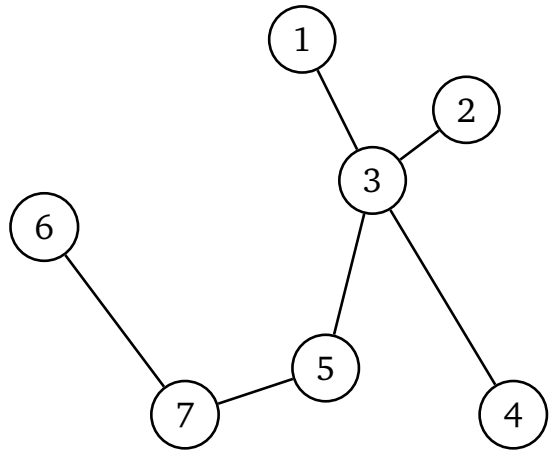
- Dato un grafo G, possono essere definiti dei sottografi di G che sono alberi. Tra questi, si definisce albero completo di G (detto anche spanning tree) un grafo parziale di G (i.e., "copre" tutti i vertici) che è un albero.
- Ogni grafo connesso ha almeno uno spanning tree.



(a) Grafo G



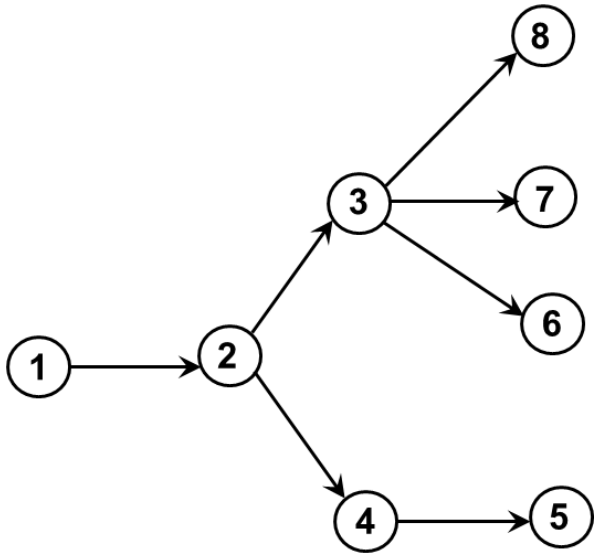
(b) Albero



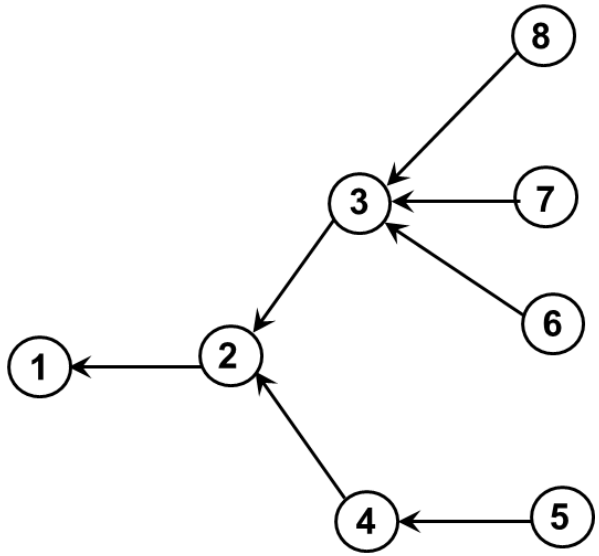
(c) Albero completo

Alberi (3)

- Directed-out-tree: Albero in cui l'unico cammino dal nodo s a tutti gli altri nodi è diretto.
- Directed-in-tree: Albero in cui l'unico cammino da un qualsiasi altro nodo al nodo s è diretto.



(a) Directed-Out-Tree



(b) Directed-In-Tree

Rappresentazione dei Grafi

- Il ruolo delle strutture dati è cruciale nello sviluppo di algoritmi efficienti.
- Il modo in cui sono salvati i dati del grafo (*rete*) nella memoria del calcolatore determina le performance degli algoritmi che operano su tali dati.
- Alcune delle operazioni che devono essere svolte dagli algoritmi sono le seguenti:
  - Accedere alle informazioni dei vertici;
  - Accedere alle informazioni degli archi;
  - Determinare tutti gli archi che partono da un vertice i;
  - Determinare tutti gli archi che arrivano a un vertice i;
  - Determinare tutti gli archi che incidono su un vertice i.

Rappresentazione dei Grafi (2)

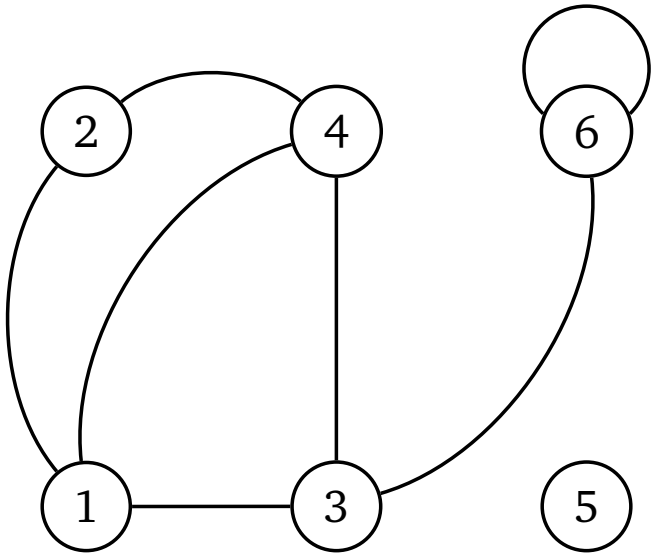
- In letteratura sono presentate numerose proposte. Alcune permettono un efficiente accesso ai dati, ma sono dispendiose dal punto di vista dell'occupazione di memoria, altre forniscono efficaci compromessi.
- La scelta della struttura dati più opportuna dipende principalmente dall'algoritmo che si deve implementare e dalle "risorse" a disposizione.

Rappresentazione dei Grafi: Matrice di Adiacenza

Definizione. La matrice di adiacenza Q di un grafo non orientato semplice $G = (V, E)$ è la matrice simmetrica $|V| \times |V|$ con elementi:

$$q_{ij} = \begin{cases} 1 & \text{se } \{i,j\} \in E; \\ 0 & \text{altrimenti.} \end{cases}$$

Esempio



(a) Grafo G

$$Q = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \end{array} \begin{array}{l} \\ \to \Gamma(1) \\ \to \Gamma(2) \\ \to \Gamma(3) \\ \to \Gamma(4) \\ \to \Gamma(5) \\ \to \Gamma(6) \end{array}$$

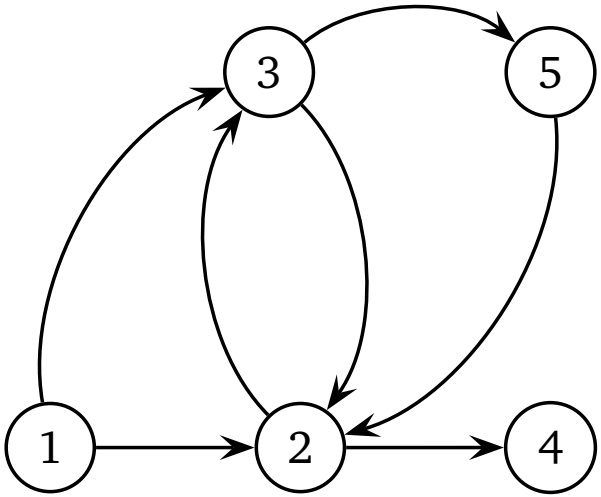
(b) Matrice di adiacenza

Rappresentazione dei Grafi: Matrice di Adiacenza

Definizione. La matrice di adiacenza Q di un grafo orientato semplice $G = (V, A)$ è la matrice $|V| \times |V|$ con elementi:

$$q_{ij} = \begin{cases} 1 & \text{se } (i,j) \in A; \\ 0 & \text{altrimenti.} \end{cases}$$

Esempio



(a) Grafo G

$$Q = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \underbrace{\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \left[\begin{array}{ccccc} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array}}_{\downarrow\ \Gamma^-(i)} \left.\begin{array}{l} \\ \\ \\ \\ \\ \end{array}\right\} \rightarrow \Gamma^+(i)$$

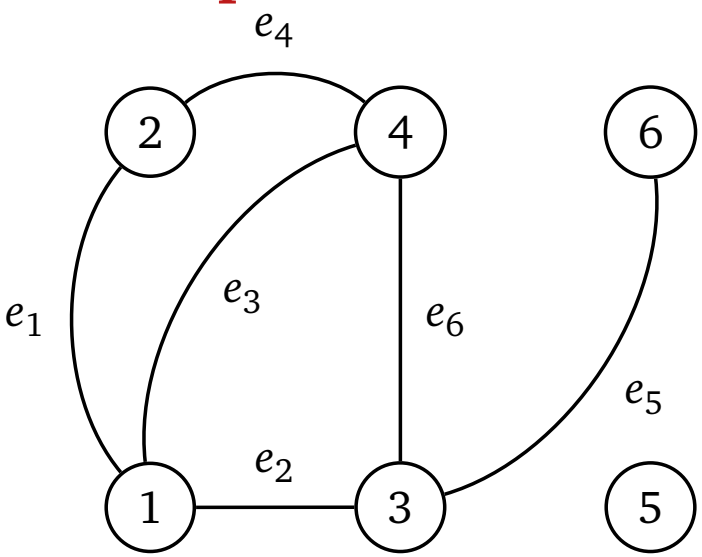
(b) Matrice di adiacenza

Rappresentazione dei Grafi: Matrice di Incidenza

Definizione. La matrice di incidenza nodi-lati D di un grafo non orientato $G = (V, E)$ è la matrice $|V| \times |E|$ con elementi:

$$d_{ik} = \begin{cases} 1 & \text{se il } k\text{-esimo lato è incidente nel vertice } i \text{ (i.e., } e_k = \{i, j\}); \\ 0 & \text{altrimenti.} \end{cases}$$

Esempio



(a) Grafo G

$$D = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

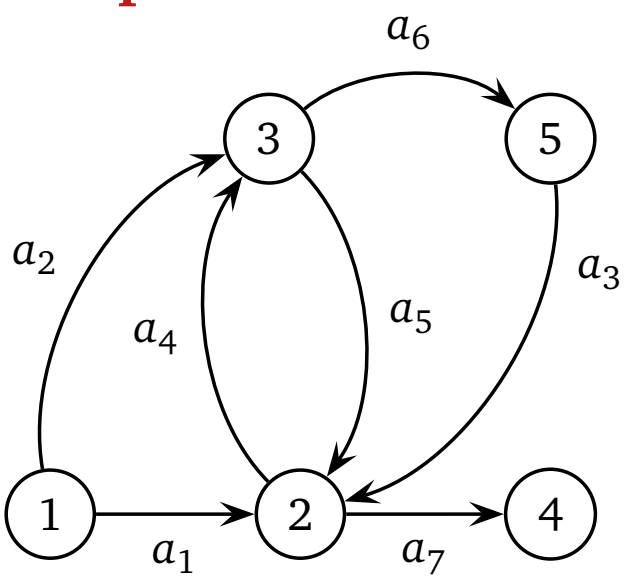
(b) Matrice di incidenza

Rappresentazione dei Grafi: Matrice di Incidenza

Definizione. La matrice di incidenza nodi-archi D di un grafo orientato $G = (V, A)$ è la matrice $|V| \times |A|$ con elementi:

$$d_{ik} = \begin{cases} 1 & \text{se il } k\text{-esimo arco esce dal vertice } i \text{ (i.e., } a_k = (i,j)); \\ -1 & \text{se il } k\text{-esimo arco entra nel vertice } i \text{ (i.e., } a_k = (j,i)); \\ 0 & \text{se } i \text{ non è vertice terminale di } a_k. \end{cases}$$

Esempio



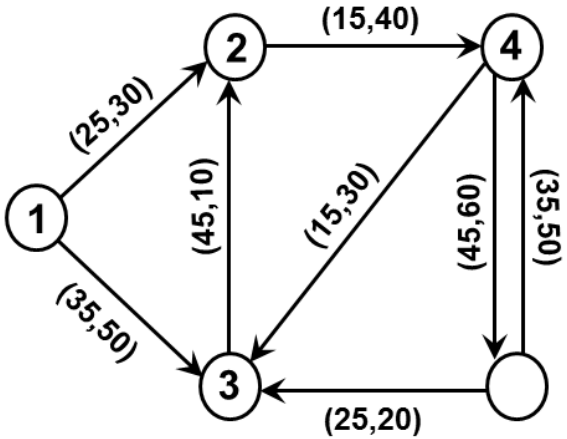
(a) Grafo G

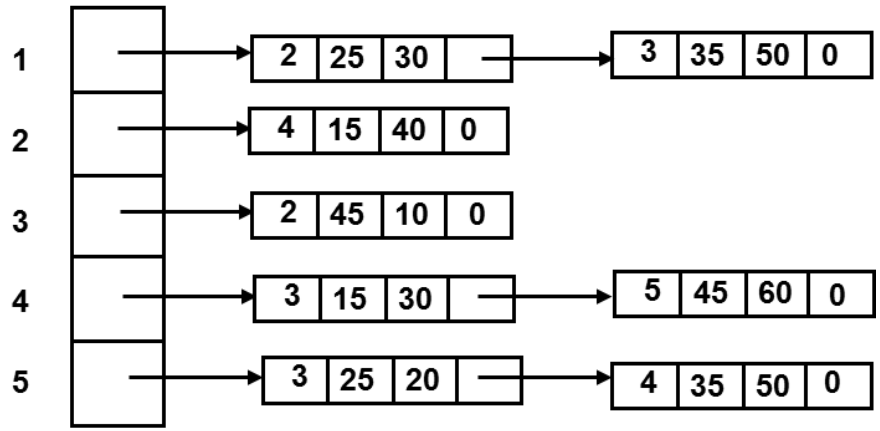
$$D = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccccc} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & -1 & 1 & -1 & 0 & 1 \\ 0 & -1 & 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{bmatrix} \end{array}$$

(b) Matrice di incidenza

Rappresentazione dei Grafi: Liste di Adiacenza

- Le liste di adiacenza conservano per ogni vertice i la lista $A(i)$ degli archi che partono da esso.
- Per cui è necessario un vettore n-dimensionale ($n = |V|$) *first*, dove *first*(i) memorizza il puntatore al primo elemento della lista $A(i)$.



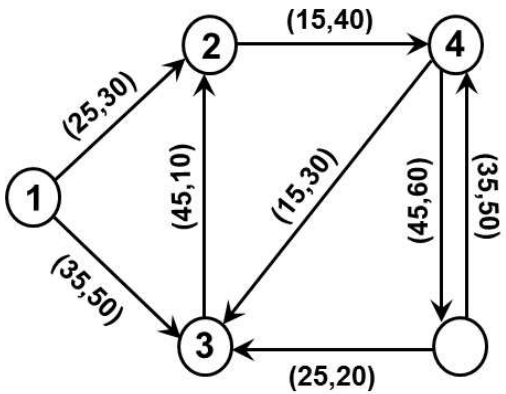


- Impiegando le liste di adiacenza si risparmia tempo calcolo e spazio di memoria. Però richiedono una "gestione" più complessa.

Rappresentazione dei Grafi: Forward Star

- La rappresentazione forward star richiede di salvare le informazioni degli archi in un vettore m-dimensionale ($m = |A|$). Gli archi devono essere ordinati per indice del *vertice iniziale* (*tail node*) crescente.

- Un vettore n-dimensionale ($n = |V|$) di puntatori *point* memorizza l'indice in cui sono salvate le informazioni del primo arco che *parte* dal vertice i nel corrispondente vettore. Gli archi che *partono* dal vertice i sono posizionati da $point(i)$ fino a $point(i+1)-1$.



	point
1	1
2	3
3	4
4	5
5	7
6	9

	tail	head	cost	capacity
1	1	2	25	30
2	1	3	35	50
3	2	4	14	40
4	3	2	45	10
5	4	3	15	30
6	4	5	45	60
7	5	3	25	20
8	5	4	35	50

Rappresentazione dei Grafi: Forward e Backward Star

- La rappresentazione forward star consente di accedere in modo efficiente agli archi che partono da un determinato vertice i.
- Nel caso sia necessario accedere agli archi che arrivano a un vertice i, la forward star non permette un'equivalente performance.
- Nel caso l'algoritmo necessiti di accedere agli archi che arrivano a un determinato vertice i è necessario utilizzare la backward star.
- La rappresentazione backward star è analoga alla forward star, ma gli archi sono ordinati per indice del *vertice finale* (*head node*) crescente;
- Inoltre, il vettore $point(i)$ memorizza l'indice in cui sono salvate le informazioni del primo arco che *arriva* al vertice i nel corrispondente vettore. Gli archi che arrivano al vertice i sono posizionati da $point(i)$ fino a $point(i+1)-1$.

Cammini di Costo Minimo

- Sia $G = (V, A)$ un grafo orientato con $n = |V|$ vertici e $m = |A|$ archi. Sia c_{ij} il costo associato ad ogni arco $(i, j) \in A$.
- Il costo di un cammino da $s \in V$ a $t \in V$ è pari alla somma dei costi degli archi che lo compongono.
- Il cammino di costo minimo (*cammino minimo*) da s a t è quello che, fra tutti i cammini da s a t, ha il costo più piccolo.
- Se $c_{ij} \geq 0$, $\forall (i, j) \in A$, il cammino minimo è elementare.
- Se alcuni dei costi c_{ij} sono negativi allora il grafo G può contenere circuiti di costo negativo. In questo caso il circuito di costo negativo può essere usato un numero infinito di volte per ridurre il costo.

Cammini di Costo Minimo (2)

- Nel caso si voglia calcolare il *cammino minimo elementare* in un grafo in cui alcuni dei costi c_{ij} sono negativi è necessario imporre esplicitamente la restrizione che il cammino passi attraverso ciascun vertice al massimo una sola volta.
- Purtroppo in presenza di cicli di costo negativo il problema è NP-Hard.
- Esistono tuttavia casi particolari in cui non esistono sicuramente cicli di costo negativo e che possono essere risolti in tempo polinomiale, fra i quali:
  - grafi aciclici;
  - grafi con costi positivi.

Formulazione Matematica

- Per ogni arco $(i,j) \in A$ si consideri la variabile decisionale:

$$x_{ij} = \begin{cases} 1 & \text{se } (i,j) \text{ viene scelto nel cammino;} \\ 0 & \text{altrimenti.} \end{cases}$$

- Il problema del cammino minimo elementare da s a t ($s \neq t$) può essere formulato come segue:

$$Min \underbrace{\sum_{(i,j)\in A} c_{ij}x_{ij}}_{\text{costo cammino}}$$

$$\underbrace{\sum_{j\in\Gamma^+(i)} x_{ij}}_{\text{n. archi uscenti}} - \underbrace{\sum_{j\in\Gamma^-(i)} x_{ji}}_{\text{n. archi entranti}} = \begin{cases} 1 & \text{se } i = s \\ -1 & \text{se } i = t \\ 0 & \forall i \in V \setminus \{s,t\} \end{cases}$$

$$x_{ij} \in \{0,1\}, \quad \forall (i,j) \in A$$

Formulazione Matematica (2)

- Nel caso possano esserci cicli di costo negativo è necessario aggiungere i seguenti vincoli:

$$\underbrace{\sum_{(i,j)\in A(S)} x_{ij}}_{\text{n. archi in } S} \leq |S|-1, \quad \forall S \subseteq V, S \neq \emptyset \qquad (*)$$

  dove $A(S)$, $S \subseteq V$, è l'insieme degli archi con entrambi gli estremi in S, i.e., $A(S) = \{(i,j) \in A : i \in S, j \in S\}$.

- Siccome dobbiamo definire i vincoli $(*)$ per ogni sottoinsieme di V, i vincoli sono comlessivamente $2^n - 1$.

- I vincoli $(*)$ impediscono il formarsi di cicli di costo negativo, per cui sono anche noti come vincoli di *subtour elimination*.

Assunzioni

- Tutti i costi degli archi c_{ij} sono interi e con C denotiamo il costo più alto, i.e., $C = \max\{c_{ij} : (i,j) \in A\}$.

  Si noti che in linea di principio tutti i costi "razionali" possono essere convertiti in interi, mentre i costi "irrazionali" (e.g., $\sqrt{2}, \pi, \ldots$) non possono essere gestiti come interi.

- La rete (grafo) contiene un cammino diretto dal nodo s a ogni altro nodo.

  Per soddisfare questa assunzione possiamo aggiungere degli archi artificiali con un costo "sufficientemente" grande.

Assunzioni (2)

- Per alcuni algoritmi assumiamo che non esistano cicli di costo negativo.

  Nel caso vi siano dei cicli di costo negativo, la soluzione ottima del problema sarebbe illimitata.

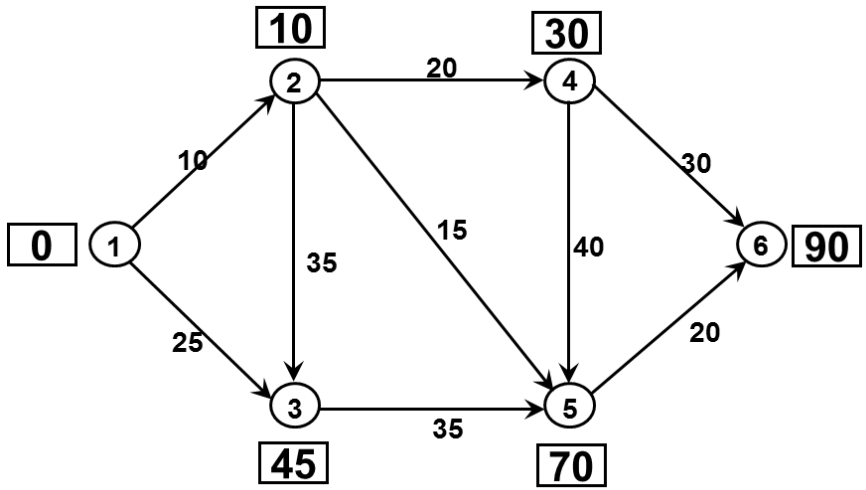
  Questi algoritmi non possono essere utilizzati per grafi in cui vi sono cicli di costo negativo, perché non garantirebbero la soluzione e/o il corretto funzionamento.

- Il grafo è orientato.

  Per soddisfare questa assunzione possiamo sostituire ogni arco non orientato (lato) $\{i,j\}$ di costo c_{ij} con due archi diretti (i,j) e (j,i) entrambi di costo c_{ij}.

Distance Label

- Diversi algoritmi per calcolare i cammini minimi impiegano il vettore delle distance label. Per ogni vertice è definita una label $d(i)$.
- La distance label $d(i)$ rappresenta il costo di un qualche cammino diretto dal vertice sorgente s al nodo i.

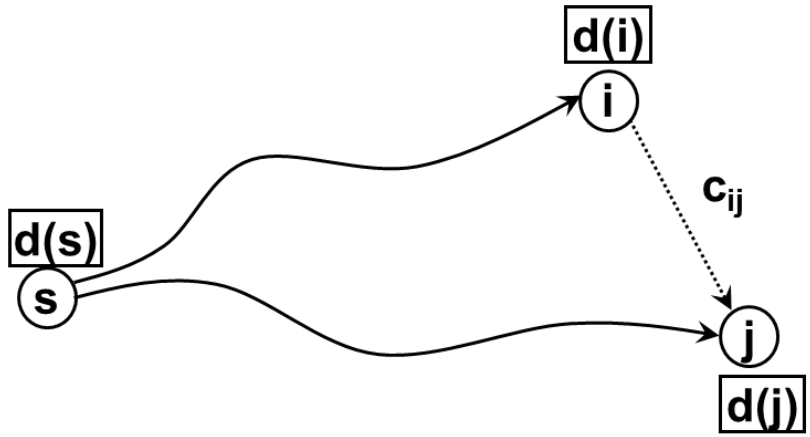


- Le distance label sono un upper bound al costo del cammino minimo dal vertice sorgente s al nodo i.

Condizioni di Ottimalità

Lemma. Se le distance label $d(k)$, per ogni $k \in V$, rappresentano il costo del cammino minimo da s a k, allora devono soddisfare le condizioni:

$$d(j) \leq d(i) + c_{ij} \quad \text{per ogni arco } (i,j) \in A.$$



Dimostrazione. Se per qualche arco $(i,j) \in A$ dovesse accadere che $d(j) > d(i) + c_{ij}$, allora la distance label $d(j)$ non rappresenterebbe il costo del cammino di costo minimo dal vertice s al vertice j perché esisterebbe un cammino meno costoso che arriva dal vertice i con l'arco (i,j).

Condizioni di Ottimalità (2)

Teorema. Le distance label $d(k)$, per ogni $k \in V$, rappresentano il costo del cammino minimo da s a k se e solo se:

$$\bar{c}_{ij} = c_{ij} + d(i) - d(j) \geq 0 \quad \text{per ogni arco } (i,j) \in A$$

Dimostrazione. Per il Lemma se le distance label $d(i)$ rappresentano il costo del cammino minimo, allora $\bar{c}_{ij} = c_{ij} + d(i) - d(j) \geq 0$ per ogni arco $(i,j) \in A$.

Ora si vuole dimostrare che se $\bar{c}_{ij} = c_{ij} + d(i) - d(j) \geq 0$ per ogni arco $(i,j) \in A$, allora le distance label $d(i)$ rappresentano il costo del cammino minimo.

Dato un qualsiasi cammino P diretto dal nodo s al nodo k.

$$\sum_{(i,j)\in P} \bar{c}_{ij} = \sum_{(i,j)\in P} \big(c_{ij} + d(i) - d(j)\big)$$

Condizioni di Ottimalità (3)

Se si semplificano le distance label si ha:

$$\sum_{(i,j)\in P} \bar{c}_{ij} = \left(\sum_{(i,j)\in P} c_{ij} \right) + d(s) - d(k)$$

Siccome $d(s) = 0$ e $\bar{c}_{ij} \geq 0$ per ogni arco $(i,j) \in P$, allora abbiamo:

$$d(k) \leq \sum_{(i,j)\in P} c_{ij} \tag{1}$$

Per cui $d(k)$ è senz'altro un lower bound al costo di ogni cammino dal nodo s al nodo k.

Dato che $d(k)$ è anche la lunghezza di un qualche cammino da s a k, allora deve essere il costo del cammino minimo.

Algoritmo Label Correcting

Algoritmo Label Correcting (Generico)

Require: Grafo orientato connesso e senza cicli di costo negativo;
Ensure: Cammini minimi da s a $V \setminus \{s\}$ definiti da $pred[j]$, $\forall j \in V \setminus \{s\}$;

```
// Inizializzazione
for j = 1 to n do
    d[j] = ∞;
    pred[j] = −1;
end for
d[s] = 0;
// Ripete finché c'è una condizione violata
while (∃(i,j) ∈ A : d[j] > d[i] + c_ij) do
    d[j] = d[i] + c_ij;
    pred[j] = i;
end while
```

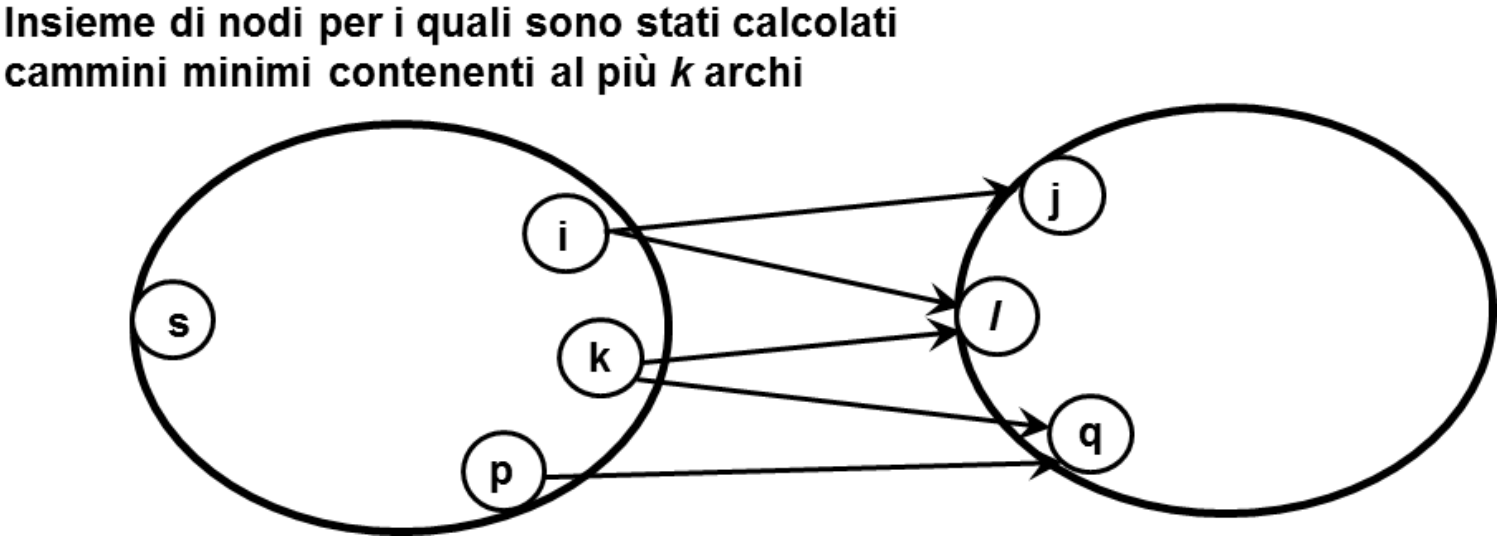
Algoritmo Label-Correcting: Complessità

- Ad ogni iterazione l'algoritmo deve considerare tutti gli archi con una complessità pari a $O(m)$.
- Il numero di iterazioni è $O(2nC)$ perché:
  - All'inizio $d(s) = 0$ e $d(j) = \infty$ per ogni $j \in V \setminus \{s\}$;
  - Ogni distance label finita $d(i)$ è limitata superiormente dal valore nC inferiormente dal valore $-nC$;
  - Ad ogni iterazione una distance label diminuisce di almeno un'unità;
  - Nessuna distance label aumenta.
- La complessità computazionale complessiva è pari a $O(2nmC)$ (pseudo-polinomiale).
- La complessità può essere diminuita a $O(nm)$.

Algoritmo Label-Correcting: Teorema

Teorema. Se ad ogni iterazione si esaminano tutti gli archi uno alla volta, verificando le condizioni di ottimalità e aggiornando le distance label quando necessario, allora dopo k iterazioni saranno determinati tutti i cammini minimi contenenti al più k archi.

Dimostrazione. Si dimostra per induzione rispetto a k.



Algoritmo di Bellman-Ford

Algoritmo di Bellman-Ford

Require: Grafo orientato;
Ensure: Cammini minimi da s a $V \setminus \{s\}$ definiti fa $pred[j]$, $\forall j \in V \setminus \{s\}$;

```
// Inizializzazione
for j = 1 to n do
    d[j] = ∞; pred[j] = −1;
end for
d[s] = 0;
// Controlla gli archi per n − 1 iterazioni
for k = 1 to n − 1 do
    for (i, j) ∈ A do
        if (d[j] > d[i] + c_ij) then
            d[j] = d[i] + c_ij; pred[j] = i;
        end if
    end for
end for
```

Algoritmo di Bellman-Ford (2)

// Controlla se ci sono cicli di costo negativo
for $(i,j) \in A$ **do**
  if $(d[j] > d[i] + c_{ij})$ **then**
    Il grafo contiene cicli di costo negativo;
  end if
end for

Osservazioni:

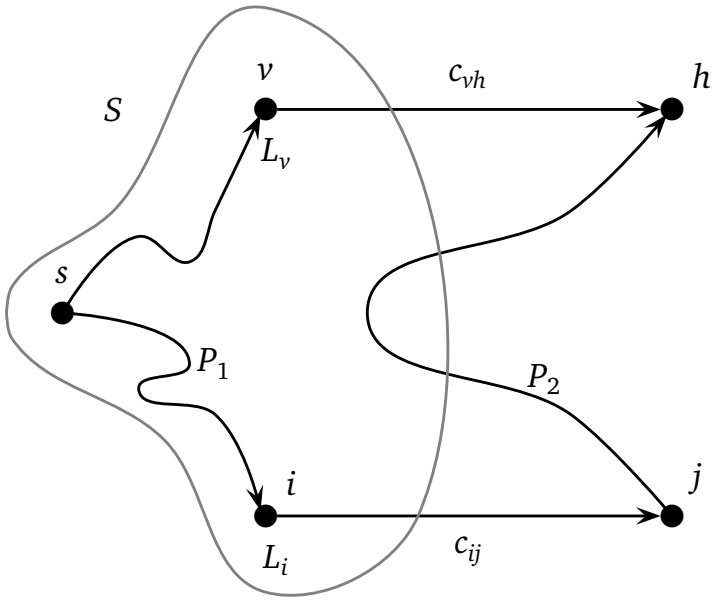
- Se nel ciclo principale non vengono trovate condizioni di ottimalità violate, allora nessuna distance label $d(i)$ verrà aggiornata.
- Se nessuna distance label verrà modificata, allora anche nell'iterazione successiva nessuna condizione di ottimalità sarà violata.
- L'algoritmo può essere ulteriormente migliorato modificando il ciclo principale per permettere un'uscita anticipata quando nessuna distance label potrà essere ulteriormente modificata.

Algoritmo di Bellman-Ford (3)

...
//Controlla gli archi per $n-1$ iterazioni
for $k = 1$ to $n-1$ **do**
    $update = False$;
    for $(i,j) \in A$ **do**
        if $(d[j] > d[i] + c_{ij})$ **then**
            $d[j] = d[i] + c_{ij}$;
            $pred[j] = i$;
            $update = True$;
        end if
    end for
    if $(update = False)$ **then**
        Esci dal ciclo;
    end if
end for
...

Algoritmo di Dijkstra: Teorema

- L'algoritmo di Dijkstra calcola i cammini minimi da $s \in V$ a ogni $t \in V$ solo se i costi degli archi sono non negativi ($c_{ij} \geq 0$, $\forall (i,j) \in A$).
- **Teorema.** Dato un sottoinsieme $S \subseteq V$ che include s (i.e., $s \in S$), sia L_i il costo del cammino minimo da s al vertice i, per ogni vertice $i \in S$. Se $(v,h) = argmin\{L_i + c_{ij} : (i,j) \in \delta^+(S)\}$, allora $L_v + c_{vh}$ rappresenta il costo del cammino minimo da s ad h.

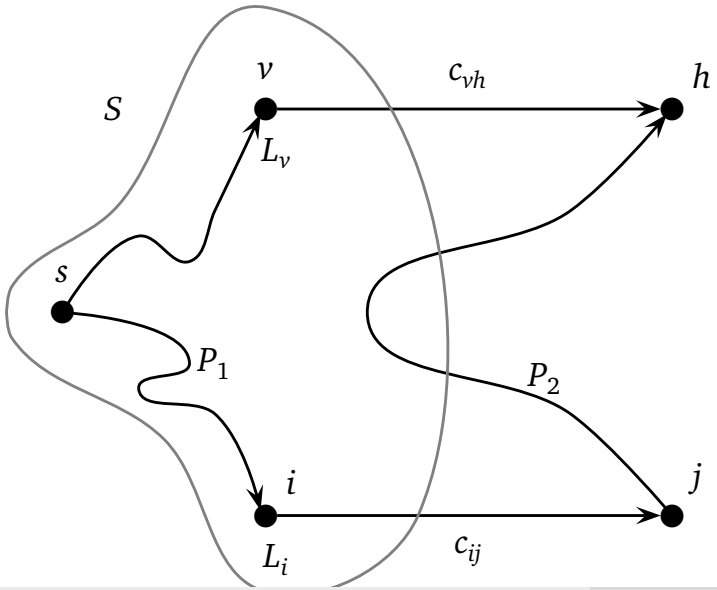


Algoritmo di Dijkstra: Teorema (2)

Dimostrazione. $L_v + c_{vh}$ rappresenta il costo di un cammino da s ad h. Si consideri un altro cammino P che termina in h. Sia $(i,j) \in P \cap \delta^+(S)$ e si partizioni P in $P_1 \cup \{(i,j)\} \cup P_2$, dove P_1 e P_2 sono due cammini da s ad i e da j ad h, rispettivamente. Si ha:

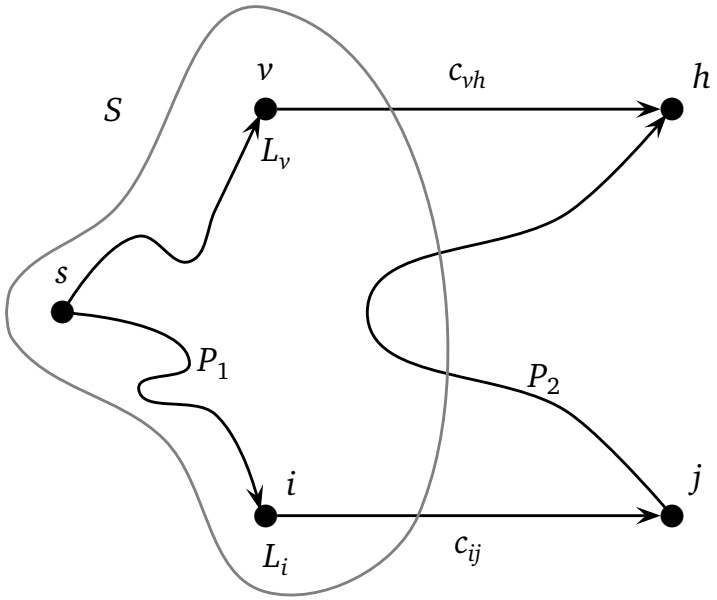
$$C(P) = \underbrace{c(P_1)}_{\geq L_i} + c_{ij} + \underbrace{C(P_2)}_{\geq 0} \geq L_i + c_{ij} \geq L_v + c_{vh}.$$

Per cui $L_v + c_{vh}$ rappresenta il costo del cammino di costo minimo da s ad h.



Algoritmo di Dijkstra: Prima Versione

- Il teorema precedente suggerisce il seguente algoritmo iterativo per la determinazione dei cammini minimi da $s \in V$ ad ogni $t \in V$.
- L'insieme S può essere interpretato come l'insieme dei vertici *permanenti* le cui label rappresentano i costi del cammino di costo minimo.
- Il vertice h dato da $(v,h) = argmin\{L_i + c_{ij} : (i,j) \in \delta^+(S)\}$ rappresenta il nuovo vertice che entra nell'insieme S dei vertici permanenti.



Algoritmo di Dijkstra: Prima Versione (2)

Algoritmo di Dijkstra (1^a versione)

Require: Grafo orientato con costi $\{c_{ij}\}$ non-negativi;
Ensure: Cammini minimi da s a $V \setminus \{s\}$ definiti da $pred[j]$, $\forall j \in V \setminus \{s\}$;

$S = \{s\}$;
$L[s] = 0$;
$pred[s] = s$;
while $(|S| \neq n)$ **do**
  if $(\delta^+(S) \neq \emptyset)$ **then**
    $(v,h) = argmin\{L[i] + c_{ij} : (i,j) \in \delta^+(S)\}$;
    $L[h] = L[v] + c_{vh}$;
    $pred[h] = v$;
    $S = S \cup \{h\}$;
  else
    Grafo G disconnesso; STOP;
  end if
end while

Algoritmo di Dijkstra: Prima Versione (3)

- La complessità dell'algoritmo è pari a $O(nm)$.
- E' possibile ottenere una complessità $O(n^2)$ se ad ogni iterazione si sfruttano opportunamente le informazioni già acquisite nelle iterazioni precedenti disponibili nelle seguenti strutture dati definite per ogni $j \in V$:
  - $flag[j] = \begin{cases} 1 & \text{se } j \in S \\ 0 & \text{altrimenti} \end{cases}$
  - $L[j] = \begin{cases} \text{costo del cammino da } s \text{ a } j, & \text{se } j \in S; \\ min\{L[i] + c_{ij} : i \in S\}, & \text{se } j \notin S; \end{cases}$
  - $pred[j] = \begin{cases} \text{predecessore di } j \text{ nel cammino da } s \text{ a } j, & \text{se } j \in S \\ argmin\{L[i] + c_{ij} : i \in S\}, & \text{se } j \notin S \end{cases}$

Algoritmo di Dijkstra: Versione Migliorata

Algoritmo di Dijkstra (versione $O(n^2)$)

Require: Grafo orientato connesso con costi $\{c_{ij}\}$ non-negativi;
Ensure: Cammini minimi da s a $V \setminus \{s\}$ definiti da $pred[j]$, $\forall j \in V \setminus \{s\}$;

```
// Inizializzazione
for j = 1 to n do
    flag[j] = 0; pred[j] = s; L[j] = c_sj;
end for
flag[s] = 1; L[s] = 0;
for k = 1 to n − 1 do
    // Individua h = argmin{L[j] : j ∉ S}
    min = +∞;
    for j = 1 to n do
        if (flag[j] = 0) and (L[j] < min) then
            min = L[j]; h = j;
        end if
    end for
```

Algoritmo di Dijkstra: Versione Migliorata (2)

```
    // Aggiorna S = S ∪ {h}
    flag[h] = 1;
    // Aggiorna L[j] e pred[j] per ogni j ∉ S
    for j = 1 to n do
        if (flag[j] = 0) and (L[h] + c_hj < L[j]) then
            L[j] = L[h] + c_hj;
            pred[j] = h;
        end if
    end for
end for
```

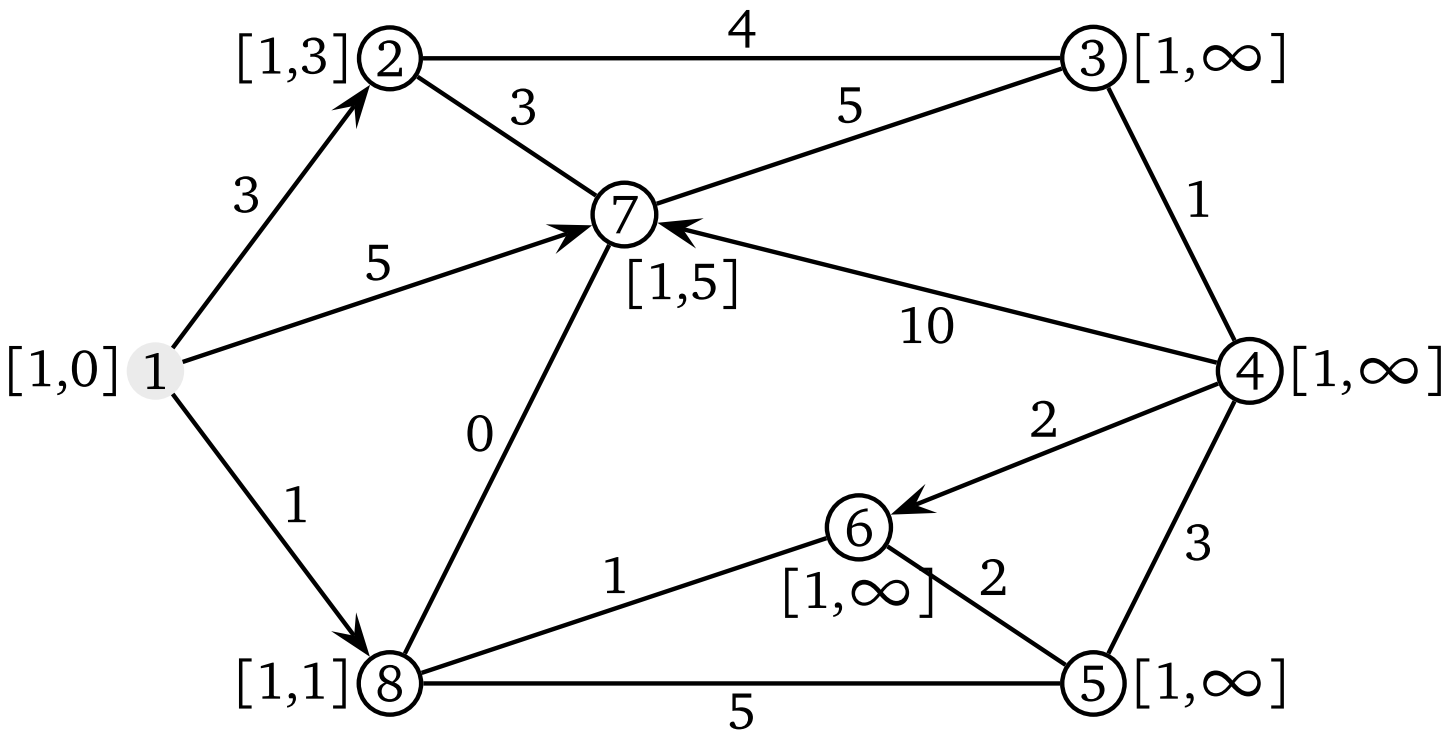
NOTA: Se si è interessati a calcolare i cammini minimi dal vertice s a un sottoinsieme di vertici $T \subseteq V$ (eventualmente contenente un solo vertice t, i.e., $T = \{t\}$), possiamo fermare l'Algoritmo di Dijkstra non appena le etichette di tutti i vertici di T sono in S, quindi permanenti.

Algoritmo di Dijkstra: Ulteriori Miglioramenti

- La complessità dell'Algoritmo di Dijkstra può essere ridotta utilizzando strutture dati per mantenere *ordinate* le label dei vertici non ancora inseriti nell'insieme dei permanenti S.
- L'obiettivo è quello di rendere più efficiente la selezione del nodo che diventerà permanente e sarà espanso.
- Tra le diverse opzioni vi è l'impiego di una *heap*.
- Se si utilizza una *Fibonacci Heap*, l'Algoritmo di Dijkstra avrà complessità $O(m + n \log n)$.
- Un'altra opzione molto interessante è l'utilizzo dell'*approccio di Dial* che permette di avere una complessità $O(m+nC)$, che nonostante sia pseudopolinomiale raramente raggiunge il caso peggiore. Inoltre, per grafi in cui C è piccolo anche il caso peggiore risulta competitivo.

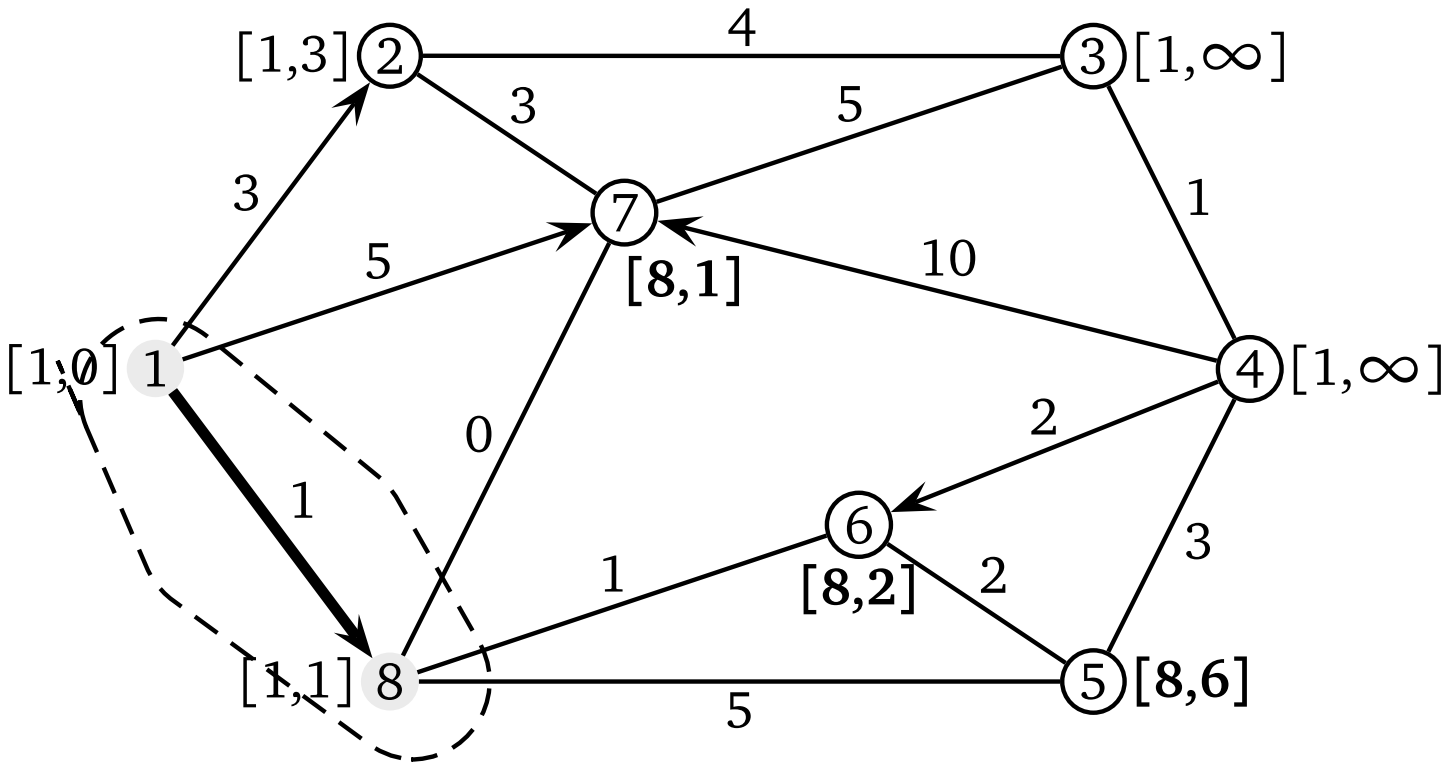
Algoritmo di Dijkstra: Esempio

Inizializzazione $s = 1$ (etichette $[pred[j], L[j]]$ sui vertici)



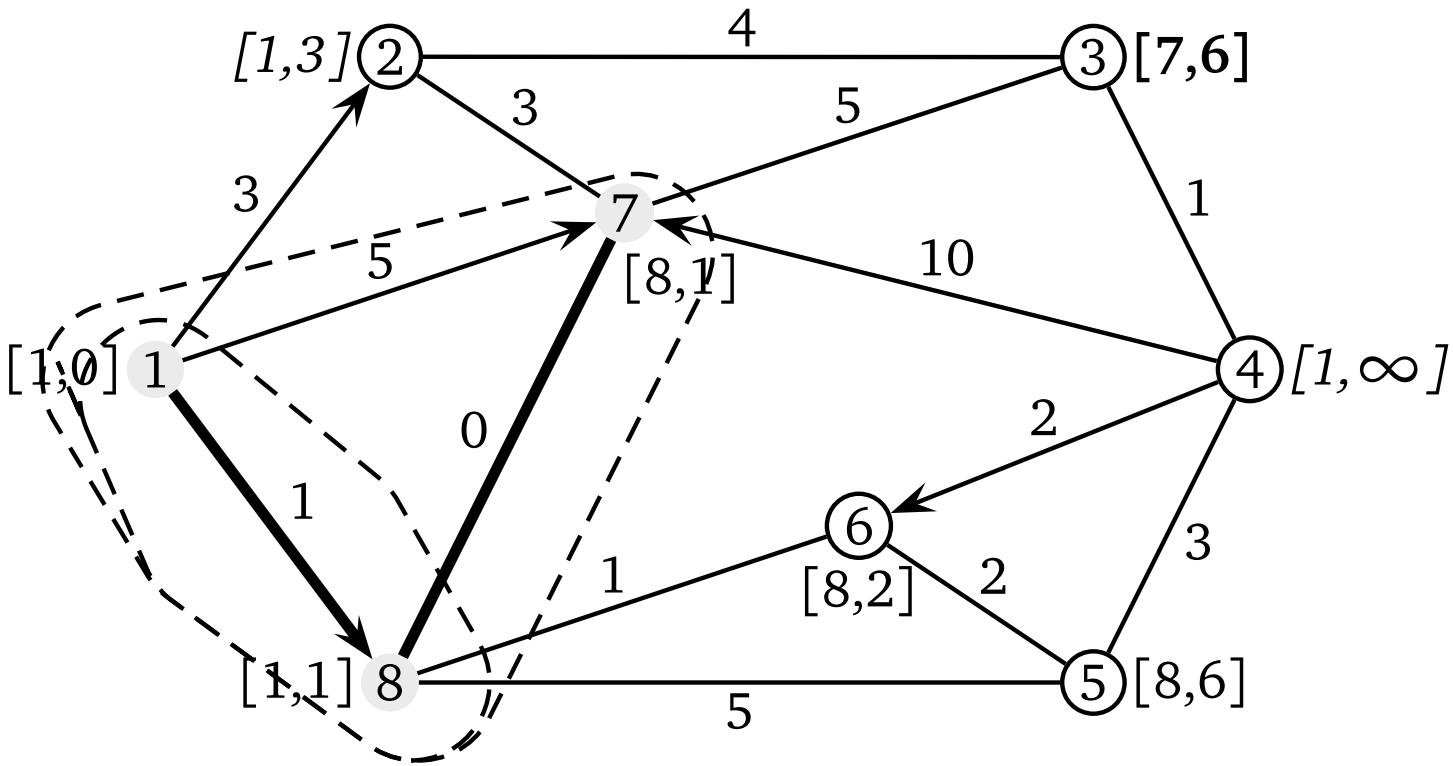
Algoritmo di Dijkstra: Esempio (2)

Vertice scelto 8



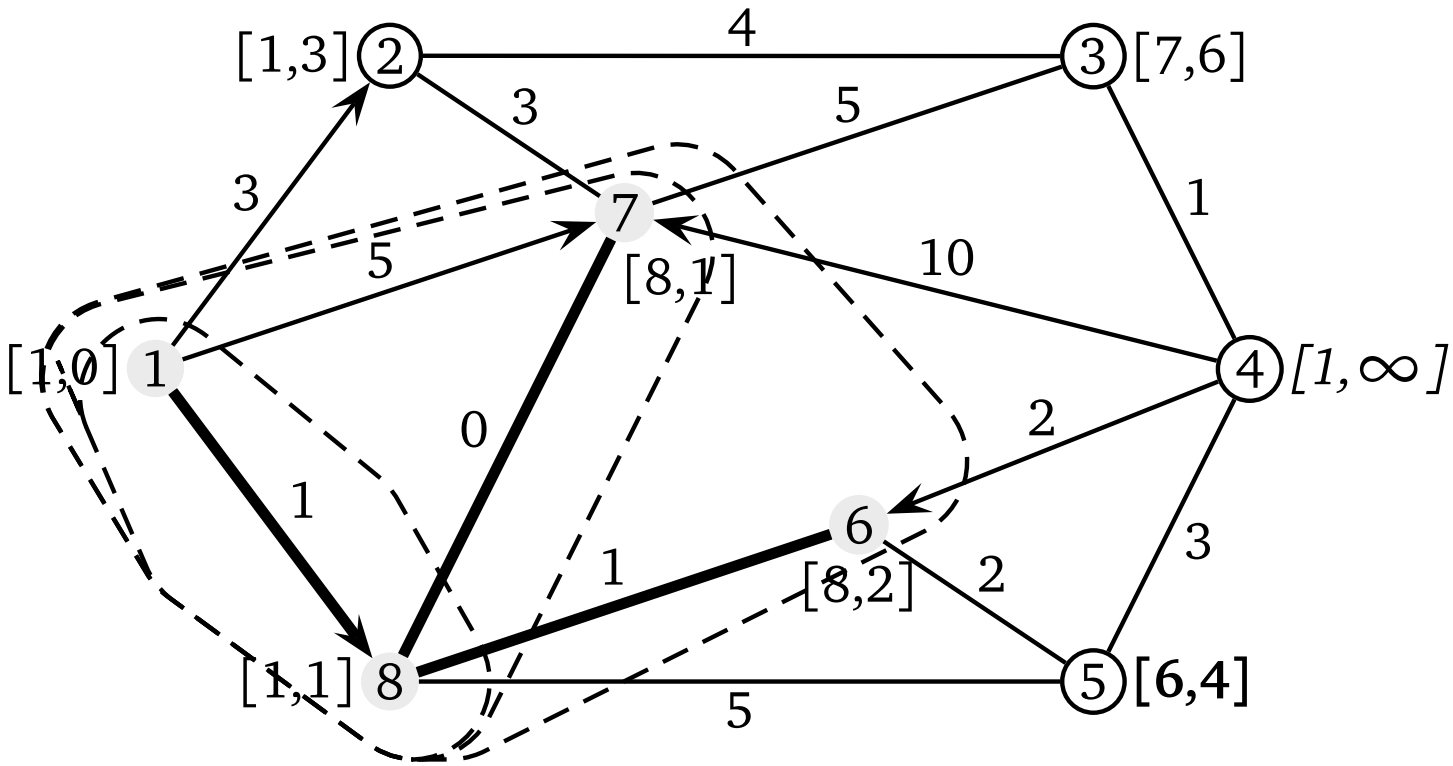
Algoritmo di Dijkstra: Esempio (3)

Vertice scelto 7



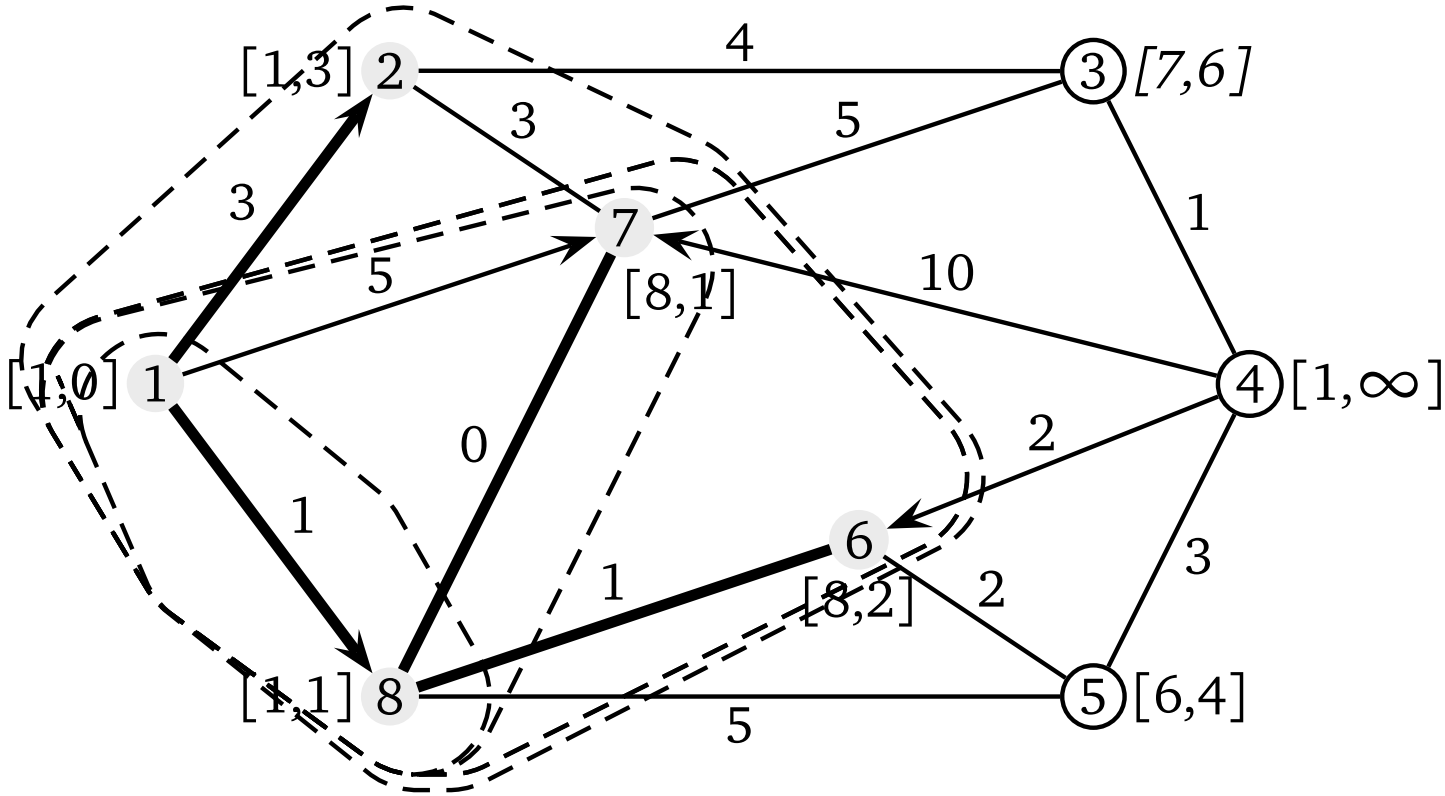
Algoritmo di Dijkstra: Esempio (4)

Vertice scelto 6



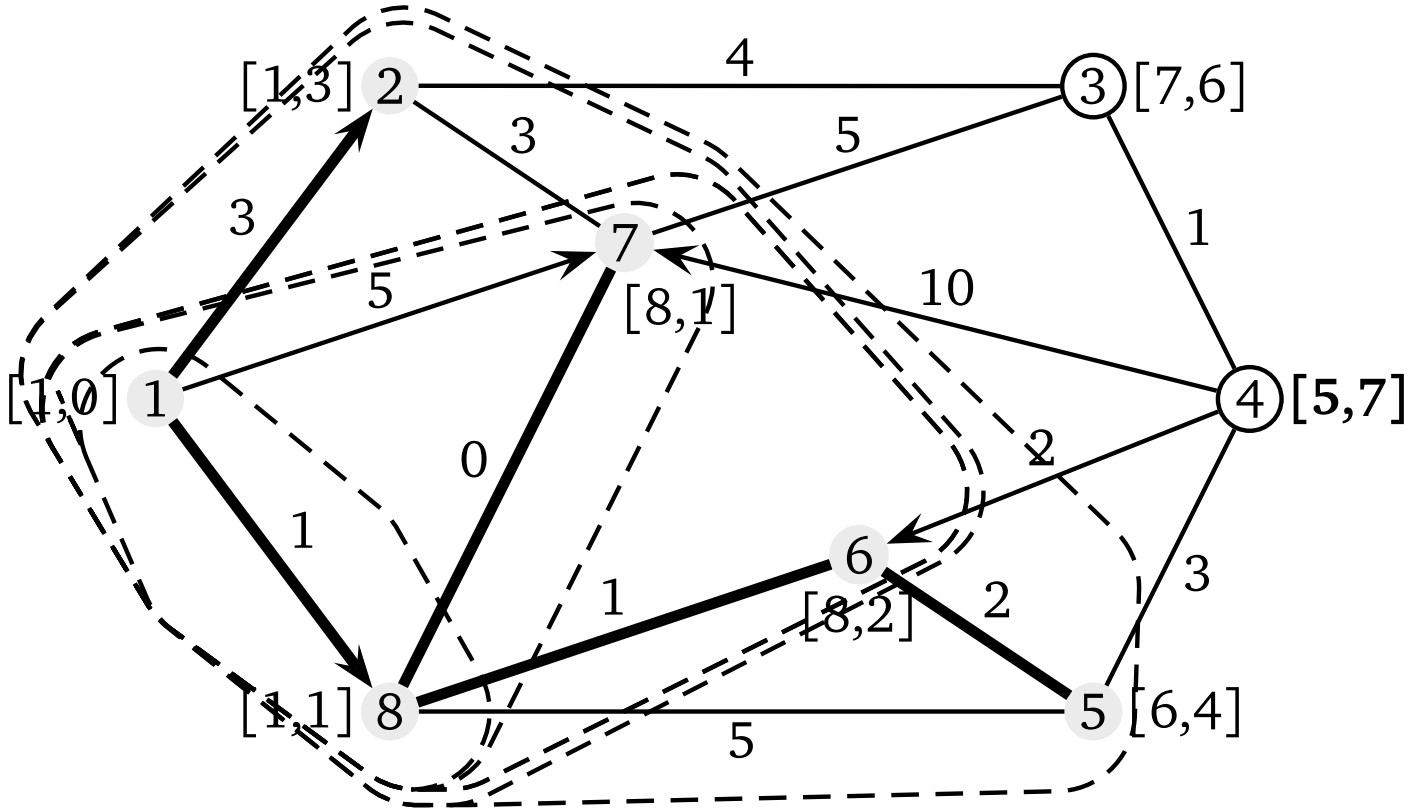
Algoritmo di Dijkstra: Esempio (5)

Vertice scelto 2



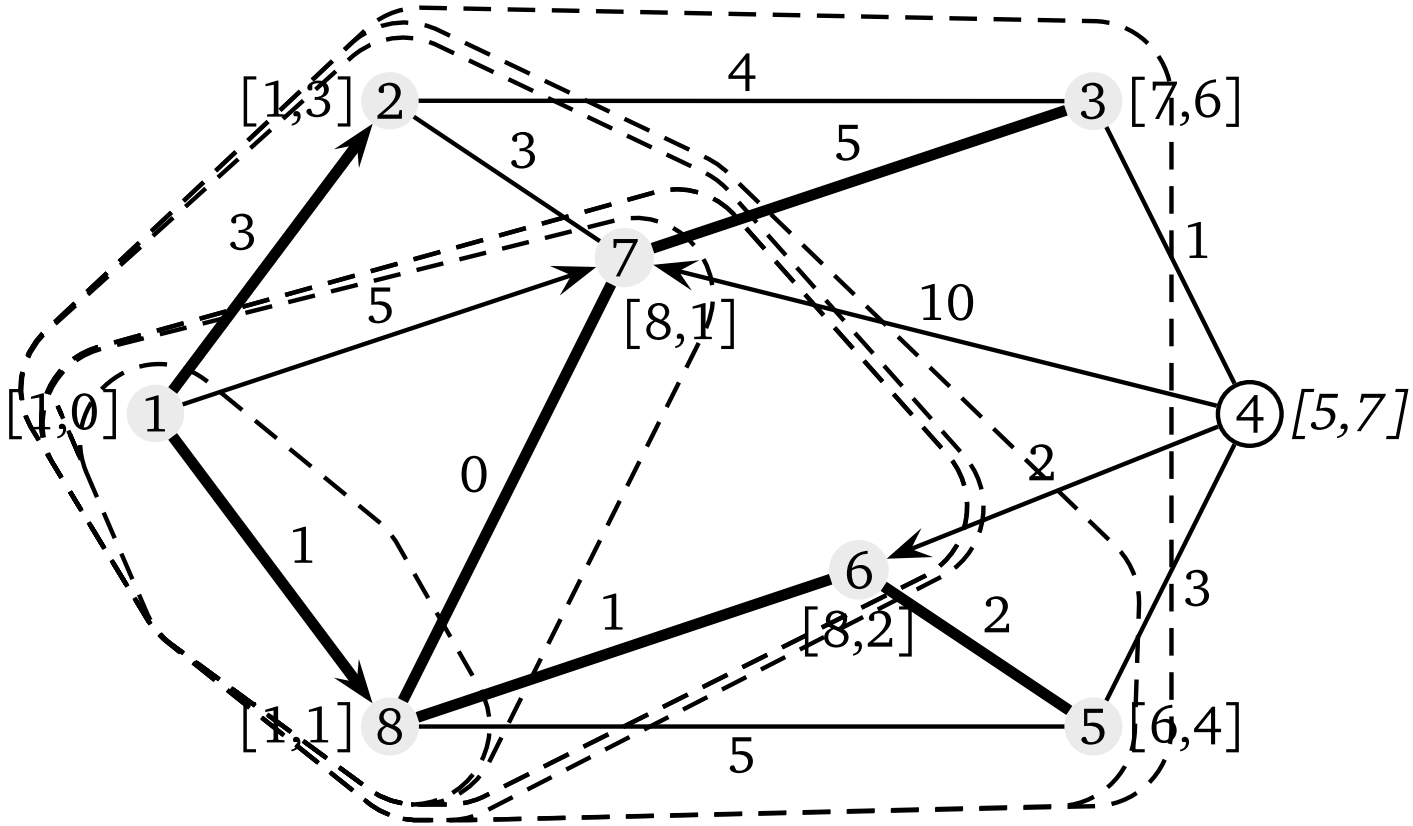
Algoritmo di Dijkstra: Esempio (6)

Vertice scelto 5



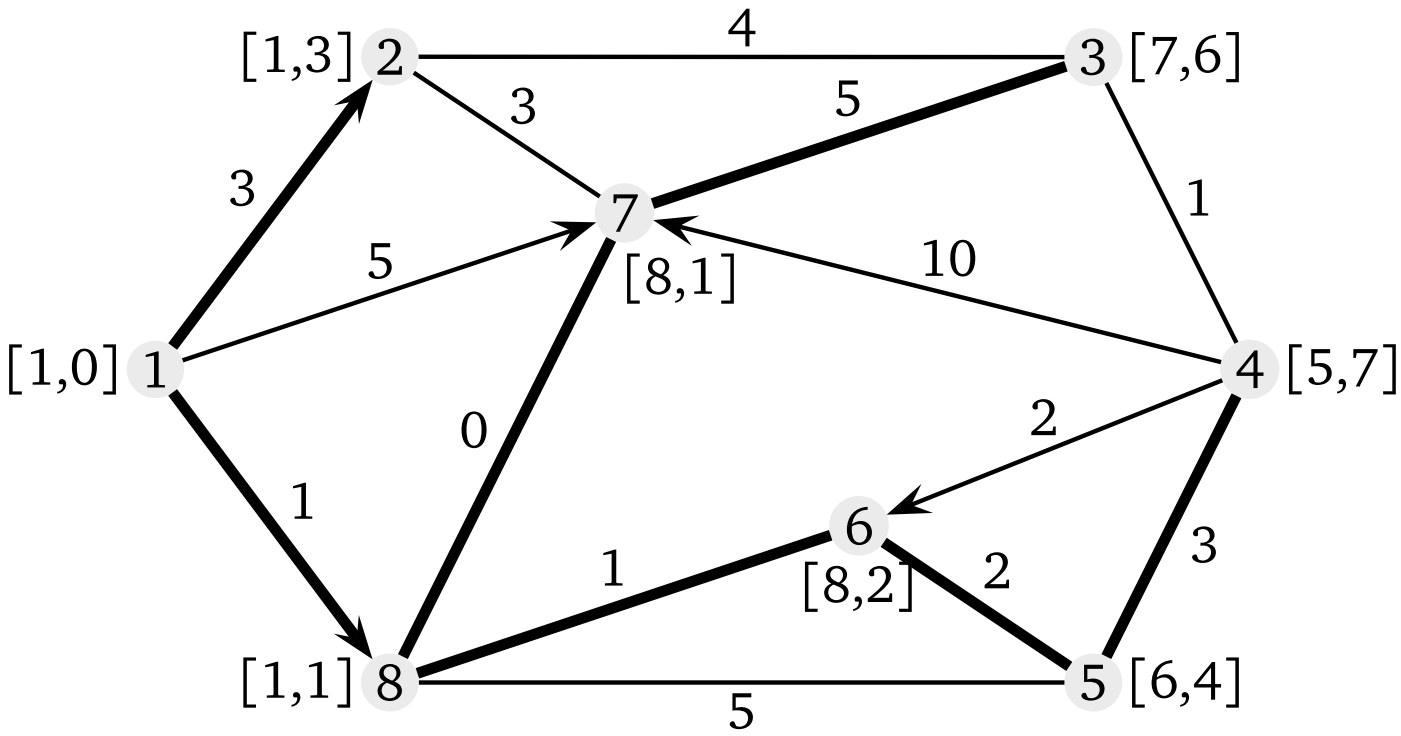
Algoritmo di Dijkstra: Esempio (7)

Vertice scelto 3

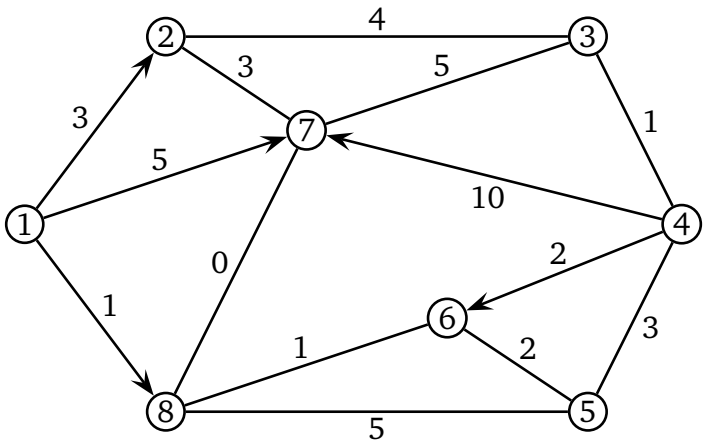


Algoritmo di Dijkstra: Esempio (8)

Vertice scelto 4



Formato Tabellare dell'Algoritmo di Dijkstra



(a) Grafo G

$$[c_{ij}] = \begin{bmatrix} & 3 & & & & & 5 & 1 \\ & & 4 & & & & 3 & \\ & 4 & & 1 & & & 5 & \\ & & 1 & & 3 & 2 & 10 & \\ & & & 3 & & 2 & & 5 \\ & & & & 2 & & & 1 \\ & 3 & 5 & & & & & 0 \\ & & & & 5 & 1 & 0 & \end{bmatrix}$$

(b) Matrice dei Costi

S	$L[j]$							$pred[j]$						
	2	3	4	5	6	7	8	2	3	4	5	6	7	8
{1}	3	∞	∞	∞	∞	5	**1**	1	-	-	-	-	1	1
{1,8}	3	∞	∞	6	2	**1**	1	1	-	-	8	8	8	1
{1,8,7}	3	6	∞	6	**2**	1	1	1	7	-	8	8	8	1
{1,8,7,6}	**3**	6	∞	4	2	1	1	1	7	-	6	8	8	1
{1,8,7,6,2}	3	6	∞	**4**	2	1	1	1	7	-	6	8	8	1
{1,8,7,6,2,5}	3	**6**	7	4	2	1	1	1	7	5	6	8	8	1
{1,8,7,6,2,5,3}	3	6	**7**	4	2	1	1	1	7	5	6	8	8	1
{1,8,7,6,2,5,3,4}	3	6	7	4	2	1	1	1	7	5	6	8	8	1

Cammini Minimi fra Tutte le Coppie di Vertici

- I cammini di costo minimo tra tutte le coppie di vertici possono essere calcolati eseguendo n volte l'algoritmo di Dijkstra utilizzando ad ogni esecuzione come vertice iniziale s uno degli n vertici del grafo.
- La complessità dell'algoritmo risultante è $O(n^3)$.
- Per applicare l'Algoritmo di Dijkstra i costi degli archi devono essere non-negativi.
- Un diverso metodo è quello dell'algoritmo di Floyd-Warshall:
  - ha complessità $O(n^3)$;
  - si applica a grafi con costi qualunque ed è in grado di riconoscere circuiti di costo negativo.

Cammini Minimi fra Tutte le Coppie di Vertici (2)

- L'algoritmo si applica ad un grafo orientato definito dalla matrice $n \times n$ dei costi $[c_{ij}]$, dove si assume che $c_{ij} = \infty$ se $(i,j) \notin A$ e $c_{ii} = 0$ per ogni $i \in V$.
- L'implementazione dell'algoritmo di Floyd-Warshall richiede:
  - una matrice U di ordine $n \times n$ per memorizzare i costi dei cammini di costo minimo;
  - una matrice $Pred$ di ordine $n \times n$ per ricostruire i cammini di costo minimo.
- Al termine dell'algoritmo, per ogni $i,j \in V$, u_{ij} rappresenta il costo del cammino minimo da i a j mentre $pred[i,j]$ rappresenta il predecessore di j nel cammino minimo da i a j.

Cammini Minimi fra Tutte le Coppie di Vertici (3)

- Se $u_{ii} < 0$ allora esiste un circuito negativo (ricostruibile a partire da $pred[i,i]$).

- Il meccanismo di funzionamento dell'Algoritmo di Floyd-Warshall si basa sul seguente teorema.

  Teorema. Per ogni coppia di vertici i e j del grafo $G(V,A)$, u_{ij} sia il costo di un qualche cammino da i a j.
  I costi $[u_{ij}]$ rappresentano i cammini di costo minimo tra tutte le coppie di vertici del grafo G se e solo se soddisfano la seguente condizione di ottimalità:

$$u_{ij} \leq u_{ik} + u_{kj}, \quad \text{per tutti i vertici } i,j \text{ e } k.$$

Algoritmo di Floyd-Warshall

Algoritmo di Floyd-Warshall

Require: Grafo orientato definito dalla matrice dei costi $[c_{ij}]$;
Ensure: Matrici $[u_{ij}]$ e $[pred[i,j]]$;

```
// Inizializzazione
for i = 1 to n do
    for j = 1 to n do
        u_ij = c_ij; pred[i,j] = i;
    end for
end for
// Operazione triangolare su k
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if (u_ik + u_kj < u_ij) then
                u_ij = u_ik + u_kj; pred[i,j] = pred[k,j];
            end if
```

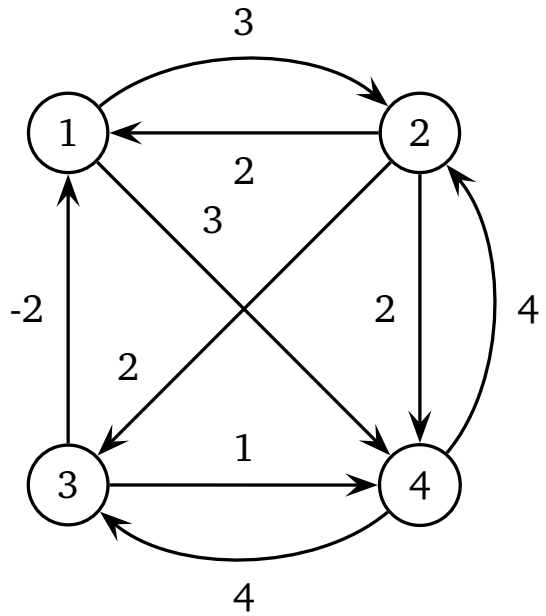
Algoritmo di Floyd-Warshall (2)

end for
end for
for $i = 1$ to n **do**
if $(u_{ii} < 0)$ **then**
STOP, *circuiti negativi*;
end if
end for
end for

Nota: Ciascun valore u_{ij} calcolato all'iterazione k-esima dell'Algoritmo di Floyd-Warshall rappresenta il costo del cammino di costo minimo dal vertice i al vertice j usando come vertici *interni* al cammino i vertici dell'insieme $\{1, 2, \ldots, k\}$.

Algoritmo di Floyd-Warshall: Esempio

Consideriamo il grafo seguente:



$$[c_{ij}] = \begin{bmatrix} 0 & 3 & \infty & 3 \\ 2 & 0 & 2 & 2 \\ -2 & \infty & 0 & 1 \\ \infty & 4 & 4 & 0 \end{bmatrix}$$

Inizializzazione:

u_{ij}

0	3	∞	3
2	0	2	2
-2	∞	0	1
∞	4	4	0

$pred[i,j]$

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Algoritmo di Floyd-Warshall: Esempio (2)

Alla prima iterazione con $k = 1$ si ha:

$$u_{ij} = \text{Min}\,\{u_{ij}, (u_{i1} + u_{1j})\}.$$

u_{ij}

0	3	∞	3
2	0	2	2
-2	∞	0	1
∞	4	4	0

$\Rightarrow$

u_{ij}

0	3	∞	3
2	0	2	2
-2	1	0	1
∞	4	4	0

$pred[i,j]$

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

$\Rightarrow$

$pred[i,j]$

1	1	1	1
2	2	2	2
3	1	3	3
4	4	4	4

Ad esempio:

$$u_{32} = \text{Min}\,\{u_{32}, (u_{31} + u_{12})\} = \text{Min}\,\{\infty, (-2 + 3)\} = 1.$$

Algoritmo di Floyd-Warshall: Esempio (3)

Alla seconda iterazione con $k = 2$ si ha:

$$u_{ij} = \text{Min}\,\{u_{ij}, (u_{i2} + u_{2j})\}.$$

u_{ij}

0	3	∞	3
2	0	2	2
-2	1	0	1
∞	4	4	0

$\Rightarrow$

u_{ij}

0	3	5	3
2	0	2	2
-2	1	0	1
6	4	4	0

$pred[i,j]$

1	1	1	1
2	2	2	2
3	1	3	3
4	4	4	4

$\Rightarrow$

$pred[i,j]$

1	1	2	1
2	2	2	2
3	1	3	3
2	4	4	4

Ad esempio:

$$u_{13} = \text{Min}\,\{u_{13}, (u_{12} + u_{23})\} = \text{Min}\,\{\infty, (3 + 2)\} = 5.$$

Algoritmo di Floyd-Warshall: Esempio (4)

Alla terza iterazione con $k = 3$ si ha:

$$u_{ij} = \text{Min}\,\{u_{ij}, (u_{i3} + u_{3j})\}.$$

u_{ij}

0	3	5	3
2	0	2	2
-2	1	0	1
6	4	4	0

$\Rightarrow$

u_{ij}

0	3	5	3
0	0	2	2
-2	1	0	1
2	4	4	0

$pred[i,j]$

1	1	2	1
2	2	2	2
3	1	3	3
2	4	4	4

$\Rightarrow$

$pred[i,j]$

1	1	2	1
3	2	2	2
3	1	3	3
3	4	4	4

Ad esempio:

$$u_{21} = \text{Min}\,\{u_{21}, (u_{23} + u_{31})\} = \text{Min}\,\{2, (2-2)\} = 0.$$

Algoritmo di Floyd-Warshall: Esempio (5)

Alla quarta e ultima iterazione con $k = 4$ si ha:

$$u_{ij} = \text{Min}\,\{u_{ij}, (u_{i4} + u_{4j})\}.$$

u_{ij}

0	3	5	3
0	0	2	2
-2	1	0	1
2	4	4	0

$\Rightarrow$

u_{ij}

0	3	5	3
0	0	2	2
-2	1	0	1
2	4	4	0

$pred[i,j]$

1	1	2	1
3	2	2	2
3	1	3	3
3	4	4	4

$\Rightarrow$

$pred[i,j]$

1	1	2	1
3	2	2	2
3	1	3	3
3	4	4	4

Ad esempio:

$$u_{23} = \text{Min}\,\{u_{23}, (u_{24} + u_{43})\} = \text{Min}\,\{2, (2+4)\} = 2.$$

Descrizione del Problema

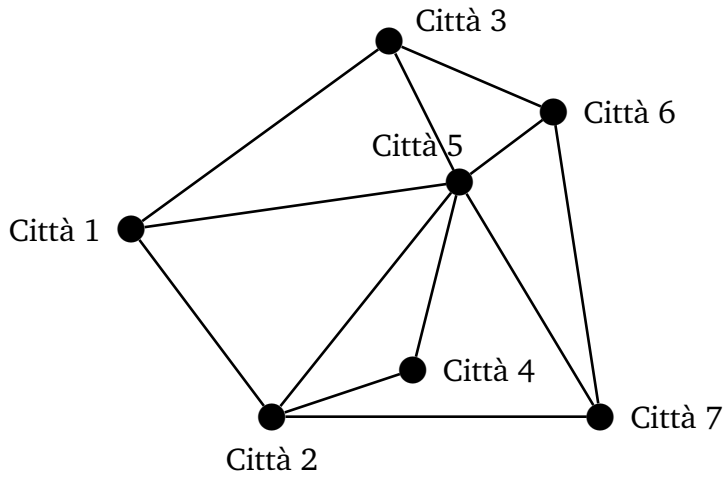
- Sia $G = (V, E)$ un grafo non orientato connesso con n vertici e m lati.
- Sia c_e o c_{ij} il costo associato ad ogni lato $e = \{i, j\} \in E$.
- Dato un albero completo di G definito dai vertici di V e dal sottoinsieme di lati $T \subseteq E$, il costo $c(T)$ dell'albero $G_T(V, T)$ è dato dalla somma dei costi degli lati che lo compongono:

$$c(T) = \sum_{e \in T} c_e$$

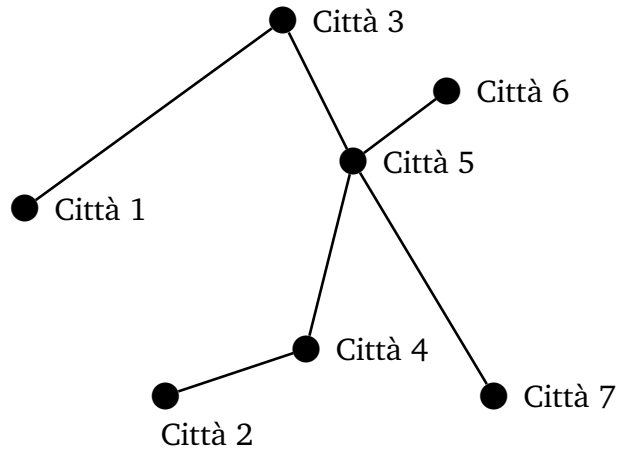
- Il *Problema dell'Albero di Copertura (o di Supporto o Completo o Coprente) di Costo Minimo (Shortest Spanning Tree, SST)* consiste nell'individuare, fra i diversi alberi completi di G, un albero $G_T = (V, T)$ di costo $c(T)$ minimo.

Applicazione: Disegno di una Rete di Comunicazione

- Dato un grafo non orientato connesso G che rappresenta una rete dei possibili collegamenti fra n città, si vuole determinate la rete di costo minimo per connettere le n città in modo tale che:
  - ogni città sia collegata direttamente con almeno un'altra città;
  - deve esistere un cammino che collega ogni coppia di città.



(a) Rete dei possibili collegamenti



(b) Rete di comunicazione

Formulazione Matematica

- Dato un grafo non orientato $G = (V, E)$ con n vertici ed m lati, un suo albero completo $G' = (V, A')$ è definito da una delle seguenti definizioni equivalenti:
  - G' è connesso e non ha cicli;
  - G' ha esattamente $n-1$ lati e non ha cicli;
  - G' è connesso e contiene esattamente $n-1$ lati.
- Per ogni lato $e \in E$ si definisce la variabile decisionale:

$$x_e = \begin{cases} 1 & \text{se il lato } e \text{ viene scelto nell'albero minimo;} \\ 0 & \text{altrimenti.} \end{cases}$$

Formulazione Matematica

- Una formulazione matematica per il problema è la seguente:

$$(SST) \qquad z_{SST} = \min \sum_{e \in E} c_e x_e \tag{2}$$

$$s.t. \sum_{e \in E} x_e = n - 1, \tag{3}$$

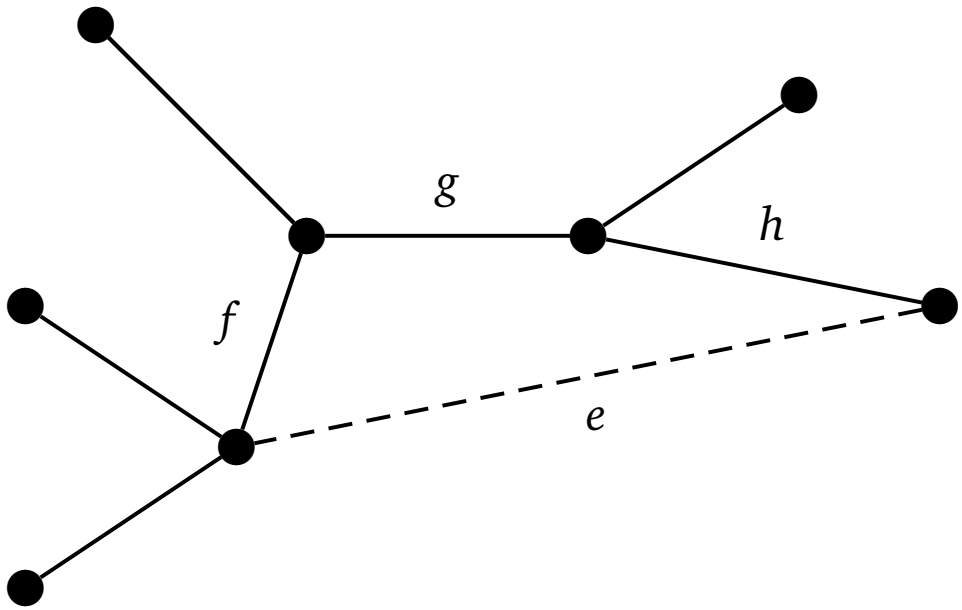
$$\sum_{e \in E(S)} x_e \leq |S| - 1, \quad \forall S \subseteq V, |S| \geq 3 \tag{4}$$

$$x_e \in \{0, 1\}, \qquad \forall e \in E \tag{5}$$

- La funzione obiettivo minimizza il costo dell'albero di copertura.
- Il vincolo (3) impone che siano scelti esattamente $n-1$ lati.
- Il numero dei vincoli (4) è $O(2^n)$ e garantiscono l'assenza di cicli nel grafo parziale $G_T = (V, T)$, dove $T = \{e \in E : x_e = 1\}$.

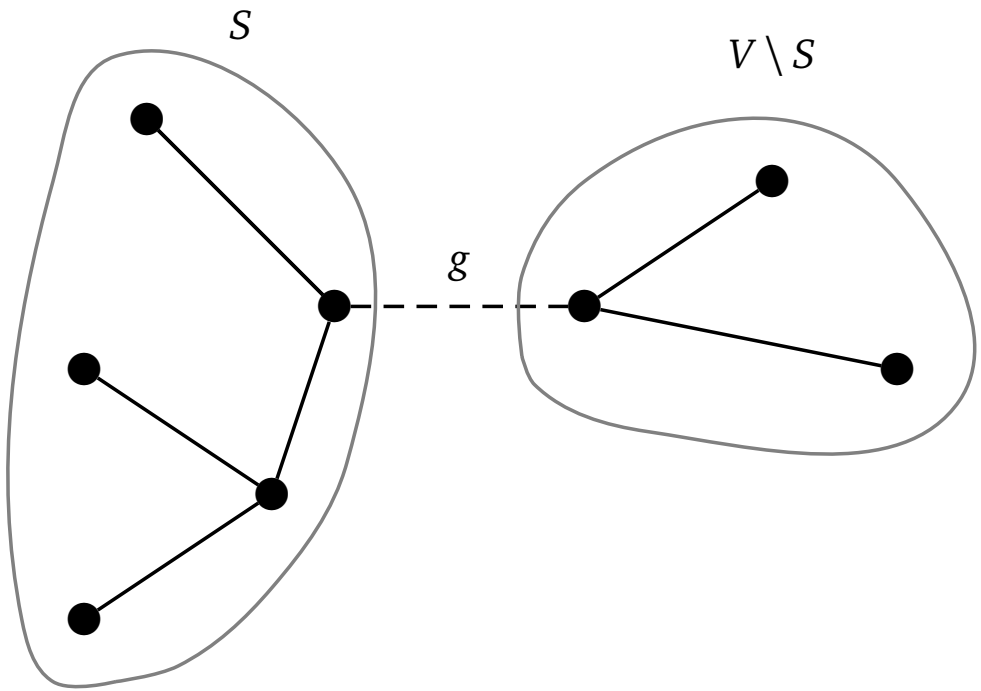
Condizioni di Ottimalità

- Dato un albero $G_T = (V, T)$ ed un lato $e \notin T$, se aggiungiamo il lato e all'albero otteniamo un ciclo. Si denota con $C(T, e)$ l'insieme dei lati del ciclo contenuto in $T \cup \{e\}$ (nell'esempio $C(T, e) = (e, f, g, h)$).



Condizioni di Ottimalità (2)

- Dato un albero $G_T = (V, T)$ ed un suo lato $g \in T$, se eliminiamo il lato g dall'albero otteniamo due alberi, i cui vertici appartengono all'insieme S e all'insieme $V \setminus S$. Gli insiemi S e $V \setminus S$ determinano un taglio $\delta(S) = \{\{i,j\} \in E : i \in S, j \in V \setminus S \text{ oppure } j \in S, i \in V \setminus S\}$.



Condizioni di Ottimalità (3)

Teorema (Cut Optimality Conditions). L'albero di copertura $G_T = (V, T)$ è di costo minimo se e solo se soddisfa le seguenti condizioni di ottimalità: per ogni lato appartenente all'albero $e \in T$, $c_e \leq c_g$ per ogni lato g appartenente al taglio S e $V \setminus S$ generato dall'eliminazione del lato e.

Dimostrazione

(a) Se l'albero di copertura è di costo minimo, allora *deve* soddisfare le condizioni di ottialità.

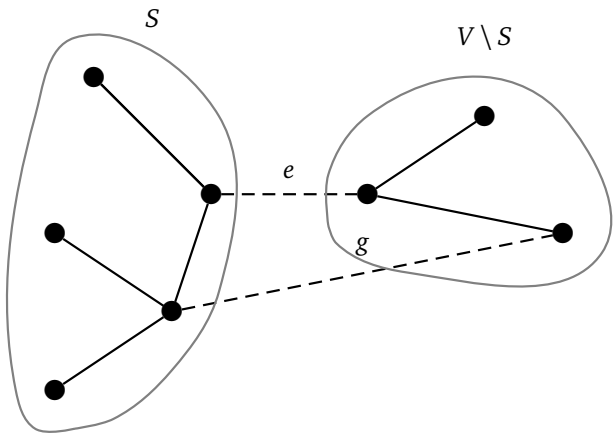
Se per un lato $e \in T$ accadesse che $c_e > c_g$ per un lato g appartenente al taglio S e $V \setminus S$ generato dall'eliminazione del lato e, allora basterebbe sostituire il lato e con g in T per ottenere un albero di copertura di costo più basso contraddicendo l'ipotesi iniziale.

Condizioni di Ottimalità (4)

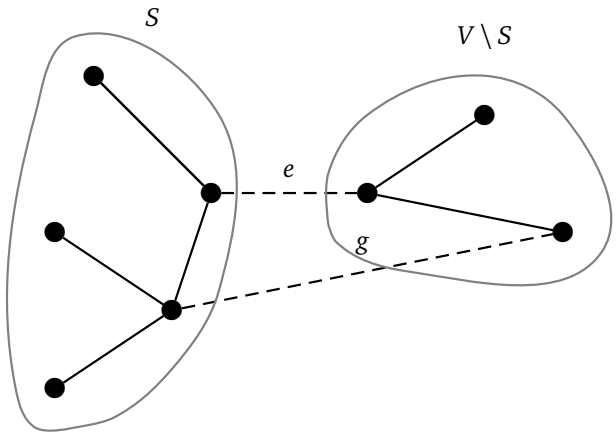
(b) Se l'albero di copertura rispetta le condizioni di ottimalità, allora è un albero di copertura di costo minimo.

Supponiamo che l'albero di copertura T non sia quello di costo minimo e che quello ottimo sia T^*. Quindi, esiste un lato $e \in T$ che non è nell'albero di copertura di costo minimo, i.e., $e \notin T^*$.

Eliminare il lato e da T crea un taglio corrispondente ai due alberi così ottenuti, mentre aggiungere il lato e all'albero ottimo T^* crea un ciclo.



Condizioni di Ottimalità (5)



L'albero T soddisfa le condizioni di ottimalità, quindi $c_e \leq c_g$ per tutti i lati del taglio compreso il lato g che assieme al lato e crea un ciclo nell'albero ottimo T^*.

Però, se T^* è l'albero ottimo $c_e \geq c_g$, altrimenti potremo trovare un albero con costo più basso scambiando i lati.

Per cui, $c_e = c_g$. Se ripetiamo, questo processo per tutti i lati di T che non sono in T^* otterremo lo stesso risultato. Quindi, l'albero di copertura T è di costo minimo.

Condizioni di Ottimalità (6)

Teorema (Path Optimality Conditions). L'albero di copertura $G_T = (V, T)$ è di costo minimo se e solo se soddisfa le seguenti condizioni di ottimalità: per ogni lato g non appartenente all'albero (i.e., $g \notin T$), $c_g \geq c_e$ per ogni lato e contenuto nel path in T che collega gli estremi del lato g.

Dimostrazione

(a) Se l'albero di copertura è di costo minimo, allora *deve* soddisfare le condizioni di ottialità.

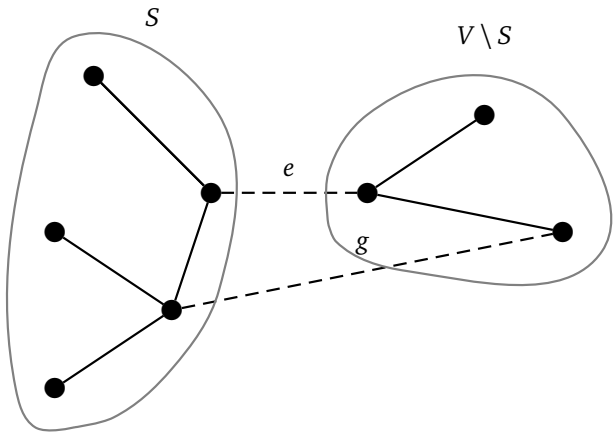
Se e è un lato del path in T che collega gli estremi del lato $g \notin T$ e $c_e > c_g$, potremo sostituire il lato e con g ottenendo un albero di copertura con un costo più basso contraddicendo l'ipotesi iniziale.

Condizioni di Ottimalità (7)

(b) Se l'albero di copertura rispetta le condizioni di ottimalità, allora è un alberto di copertura di costo minimo.

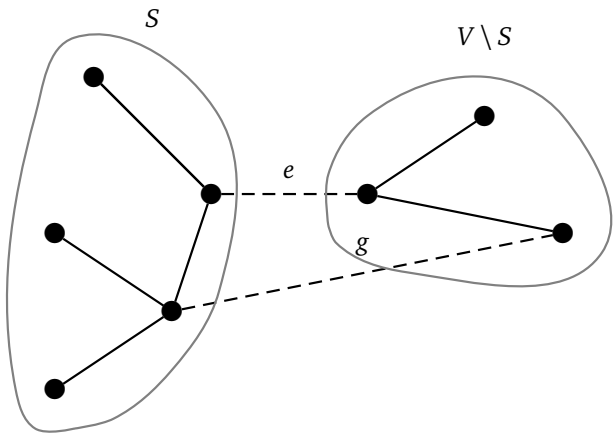
Dimostriamo che un albero T che soddisfa le *path optimality conditions* soddisfa anche le *cut optimality conditions*, così, per il teorema precedente, T è l'albero di copertura di costo minimo.

Sia e un qualsiasi lato appartenente all'albero T, e sia S e $V \setminus S$ il taglio generato dalla sua eliminazione. Per ogni lato g del taglio, nell'albero T ci sarà un unico path P che collega gli estremi di g.



Condizioni di Ottimalità (8)

Siccome e era l'unico lato del taglio che apparteneva a T (per costruzione), deve appartenere al path P.



Per ipotesi le condizioni di ottimalità sono valide, quindi $c_g \geq c_e$ per ogni lato e contenuto nel path in T che collega gli estremi del lato g.

Siccome questa condizione deve essere valida per tutti i lati non appartenenti a T del taglio ottenuto eliminando un qualsiasi lato di T, allora deve soddisfare le *cut optimality conditions*.

Condizioni di Ottimalità (9)

Quindi l'albero T è quello di costo minimo.

NOTA: La dimostrazione del teorema dimostra anche l'equivalenza tra le *cut optimality conditions* e le *path optimality conditions*.

Algoritmo di Prim-Dijkstra

- Le cut optimality conditions suggeriscono un algoritmo iterativo per la determinazione dell'albero di copertura di costo minimo di un grafo non orientato $G(V,E)$.
- L'algoritmo definisce un insieme iniziale S contenente un solo vertice del grafo, mentre l'albero iniziale T è vuoto. Per esempio, $S = \{1\}$ e $T = \emptyset$.
- Ad ogni iterazione dell'algoritmo si espande la componente connessa S determinando il lato $\{i,j\}$, con $i \in S$ e $j \in V \setminus S$, di costo minimo.
- L'algoritmo termina quando l'albero T ha $n-1$ archi e, quindi, copre tutti i vertici V.
- Avendo costruito l'albero nel rispetto delle cut optimality conditions, T è l'albero di copertura di costo minimo.

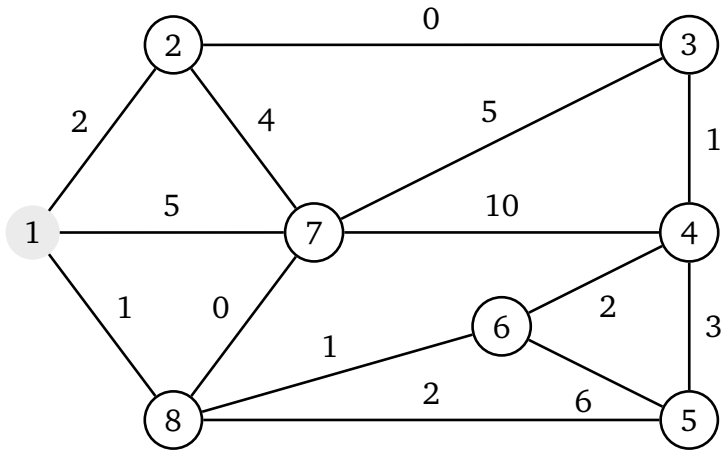
Algoritmo di Prim-Dijkstra (2)

Algoritmo di Prim-Dijkstra (1^a versione)

Require: Grafo non orientato $G(V,E)$ pesato con costi c_e, $\forall e \in E$;
Ensure: Albero di copertura di costo minimo T^*;
// Inizializzazione
$T^* = \emptyset$;
$S = \{1\}$;
// Costruisci l'albero di copertura
while $|T^*| \neq n-1$ **do**
Individua il lato $\{i,j\} \in \delta(S)$ di costo minimo, con $i \in S$ e $j \notin S$;
$T^* = T^* \cup \{\{i,j\}\}$;
$S = S \cup \{j\}$;
end while

L'algoritmo richiede $O(m)$ confronti ad ogni iterazione, quindi la sua complessità è pari perciò a $O(nm)$.

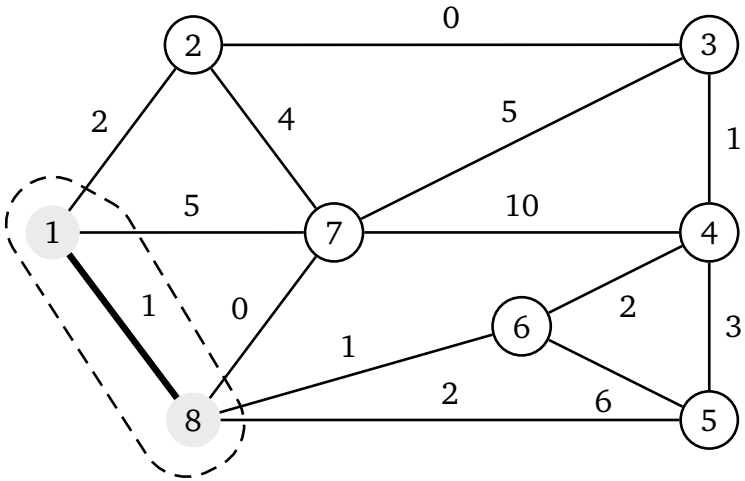
Algoritmo di Prim-Dijkstra (1^a versione): Esempio



Inizializzazione:

$T^* = \emptyset$

$S = \{1\}$

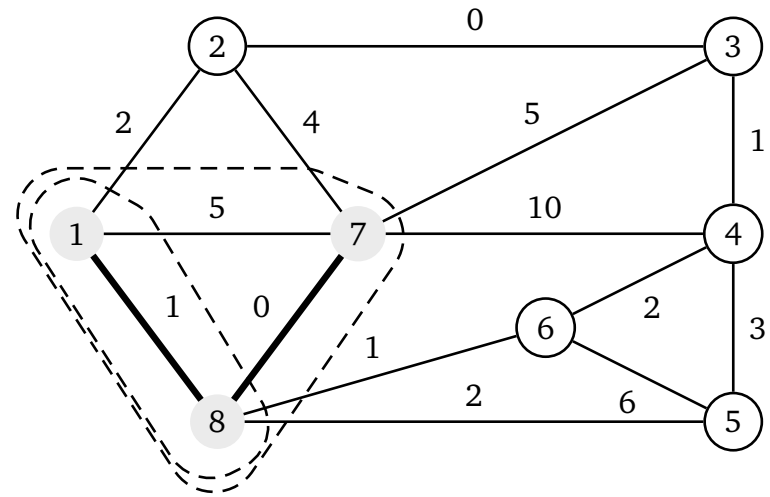


Iterazione 1:

$T^* = T^* \cup \{\{1, 8\}\}$

$S = \{1, 8\}$

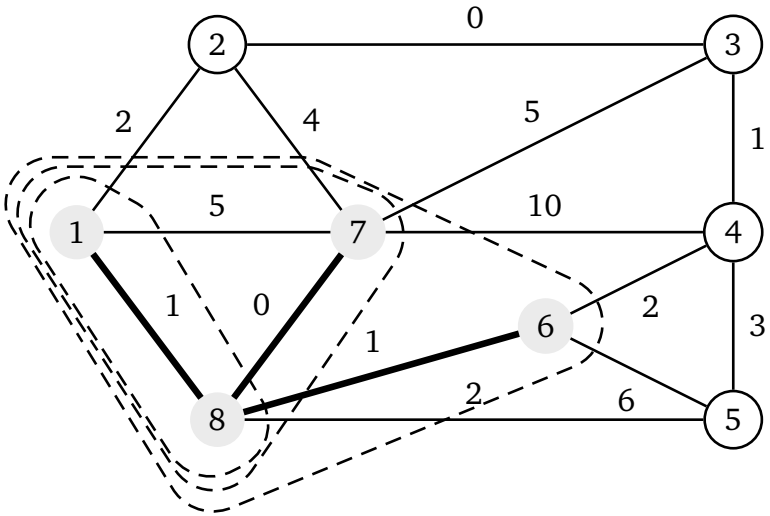
Algoritmo di Prim-Dijkstra (1^a versione): Esempio (2)



Iterazione 2:

$T^* = T^* \cup \{\{8,7\}\}$

$S = \{1,8,7\}$

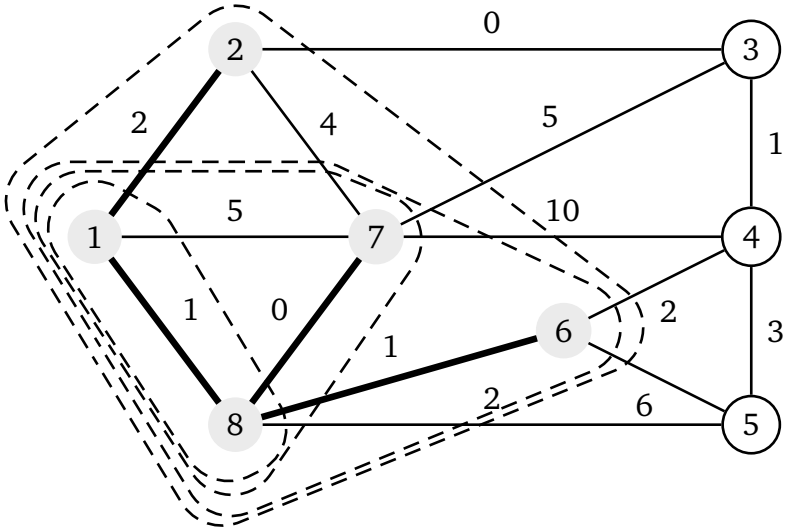


Iterazione 3:

$T^* = T^* \cup \{\{8,6\}\}$

$S = \{1,8,7,6\}$

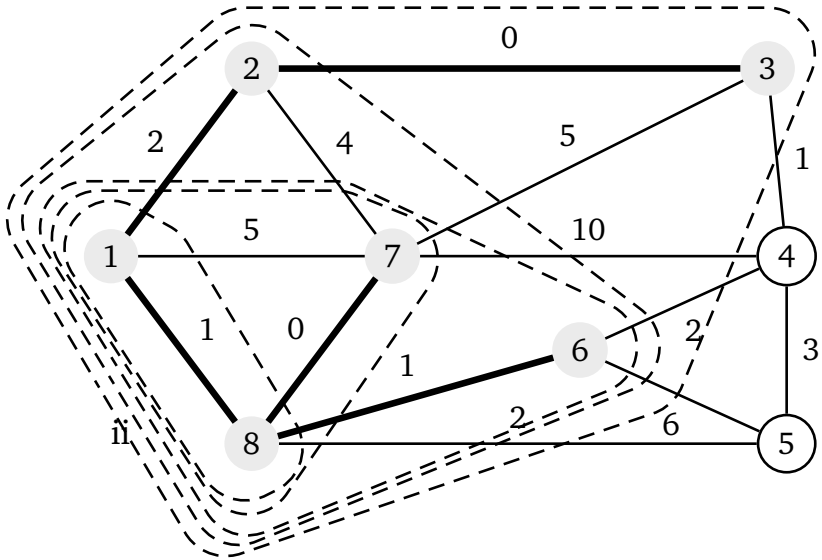
Algoritmo di Prim-Dijkstra (1^a versione): Esempio (3)



Iterazione 4:

$T^* = T^* \cup \{\{1,2\}\}$

$S = \{1,8,7,6,2\}$

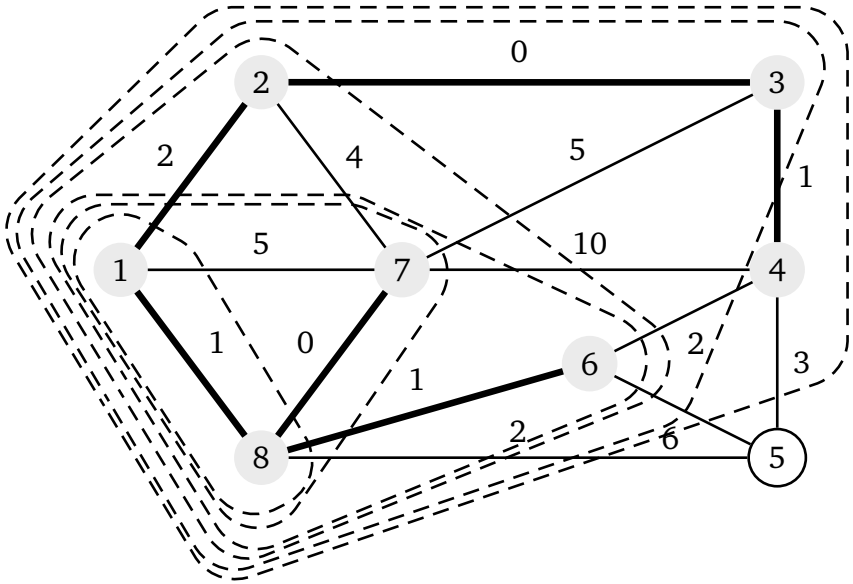


Iterazione 5:

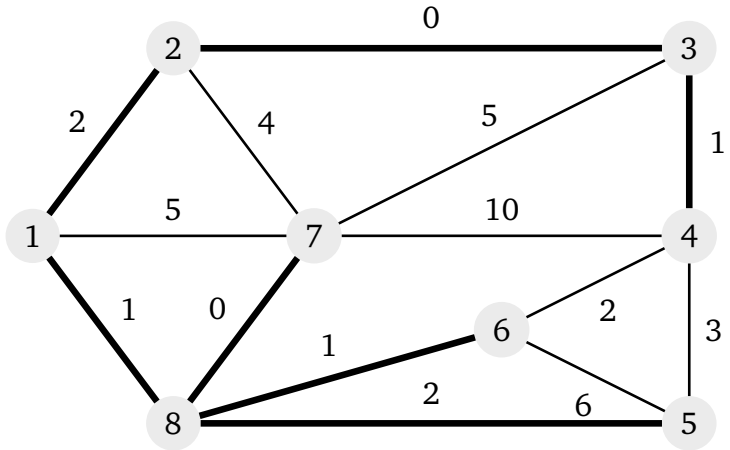
$T^* = T^* \cup \{\{2,3\}\}$

$S = \{1,8,7,6,2,3\}$

Algoritmo di Prim-Dijkstra (1^a versione): Esempio (4)



Iterazione 6:
$T^* = T^* \cup \{\{3,4\}\}$
$S = \{1,8,7,6,2,3,4\}$

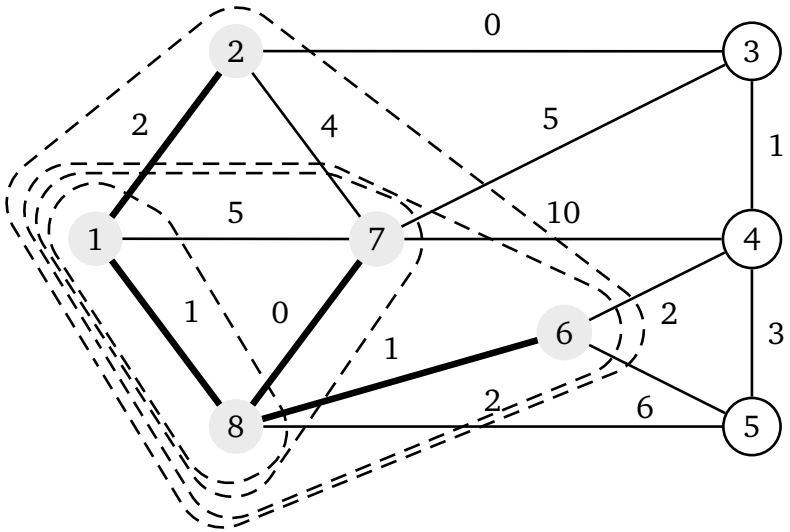


Iterazione 7:
$T^* = T^* \cup \{\{8,5\}\}$
$S = \{1,8,7,6,2,3,4,5\}$
Il costo è 7

NOTA: In questo grafo ci sono due alberi di costo minimo diversi tra loro.

Algoritmo di Prim-Dijkstra (2^a versione)

- Si consideri l'esecuzione dell'iterazione 4 nel seguente caso:

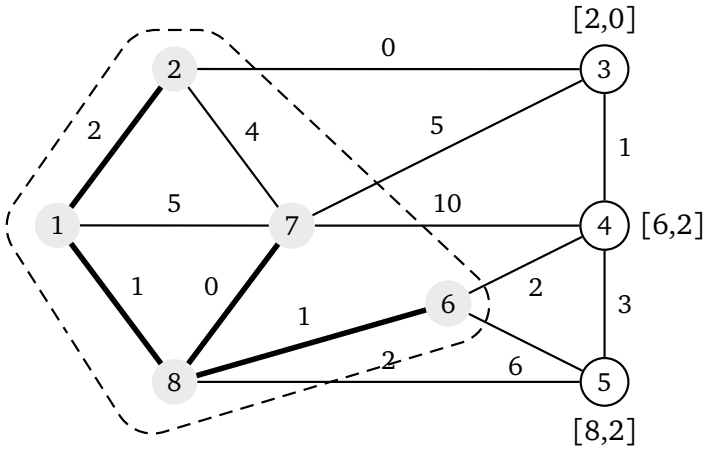


$$S = \{1,2,6,7,8\}$$
$$T^* = \{\{1,2\},\{1,8\},\{8,7\},\{8,6\}\}$$

- Per non esaminare tutti i lati $\{i,j\} \in E$ tali che $i \in S$ e $j \in V \setminus S$ (i.e., $(i,j) \in \delta(S)$) si suppone di conoscere per ogni $j \in V \setminus S$:
  - $pred[j]$: vertice di S più vicino a j, i.e., $c_{\{pred[j],j\}} = \min_{i \in S} \{c_{\{i,j\}}\}$;
  - $L[j]$: costo del lato $\{pred[j],j\}$, i.e., $L[j] = c_{\{pred[j],j\}}$.

Algoritmo di Prim-Dijkstra (2^a versione) (2)

- Quindi per l'iterazione 4 si avrà:



Etichette
$[pred[j], L[j]]$

- Il lato di costo minimo $\{i^*, j^*\} \in E$, con $i^* \in S$ e $j^* \in V \setminus S$, può essere calcolato determinando $j^* \in V \setminus S$ tale che:

$$L[j^*] = \min_{j \in V \setminus S} \{L[j]\}$$

e ponendo quindi $i^* = pred[j^*]$.

- Questo metodo per calcolare, all'iterazione 4, il lato (i^*, j^*) richiede $|V - S| \leq n$ confronti, quindi la complessità è $O(n)$.

Algoritmo di Prim-Dijkstra (2^a versione) (3)

- Le etichette $[pred[j], L[j]]$, $\forall j \in V \setminus S$, devono essere opportunamente aggiornate ad ogni iterazione dell'algoritmo; ciò può essere fatto con complessità $O(n)$.
- L'algoritmo risultante ha perciò una complessità pari a $O(n^2)$.
- L'algoritmo per calcolare l'albero di costo minimo T utilizza le seguenti strutture dati:
  - $flag[i] = \begin{cases} 1, & \text{se } i \in S \\ 0, & \text{se } i \in V \setminus S \end{cases}$
  - $L[j] = min\{c_{ij} : i \in S\}$, per ogni $j \notin S$;
  - $pred[j] = \begin{cases} argmin\{c_{ij} : i \in S\}, & \text{se } j \notin S \\ predecessore \text{ di } j \text{ in } T, & \text{se } j \in S \end{cases}$

Algoritmo di Prim-Dijkstra (2^a versione)

Algoritmo di Prim-Dijkstra (2^a versione)

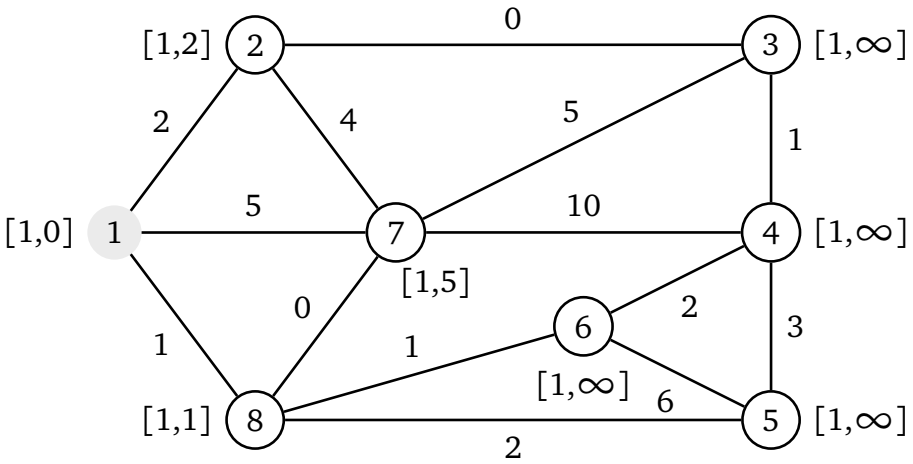
Require: Grafo non orientato $G(V,E)$ pesato con costi c_e, $\forall e \in E$;
Ensure: Albero di copertura di costo minimo T^* definito da $pred[j], j \in V$;

```
// Inizializza le strutture dati partendo da S = {1}
flag[1] = 1;
pred[1] = 1;
for j = 2 to n do
    flag[j] = 0;
    L[j] = c_1j;
    pred[j] = 1;
end for
// Seleziona gli n − 1 lati dell'albero
for k = 1 to n − 1 do
    // Sceglie il lato minimo in δ(S)
```

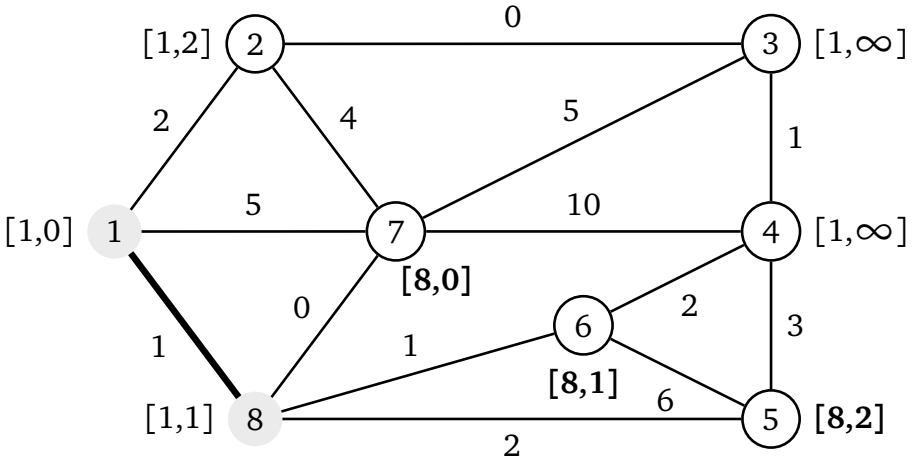
Algoritmo di Prim-Dijkstra (2^a versione) (2)

$min = +\infty$;
for $j = 2$ to n **do**
  if $(flag[j] = 0)$ **and** $(L[j] < min)$ **then**
    $min = L[j]$; $h = j$;
  end if
end for
// Include il vertice h in S e aggiorna $L[j]$ e $pred[j]$ per ogni $j \notin S$
$flag[h] = 1$;
for $j = 2$ to n **do**
  if $(flag[j] = 0)$ **and** $(c_{hj} < L[j])$ **then**
    $L[j] = c_{hj}$; $pred[j] = h$;
  end if
end for
end for

Algoritmo di Prim-Dijkstra (2^a versione): Esempio

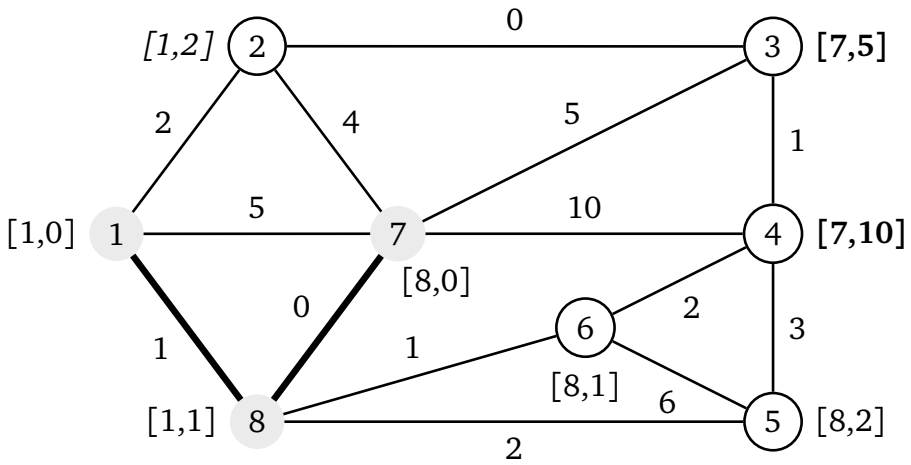


Inizializzazione, etichette $[pred[j], L[j]]$ sui vertici

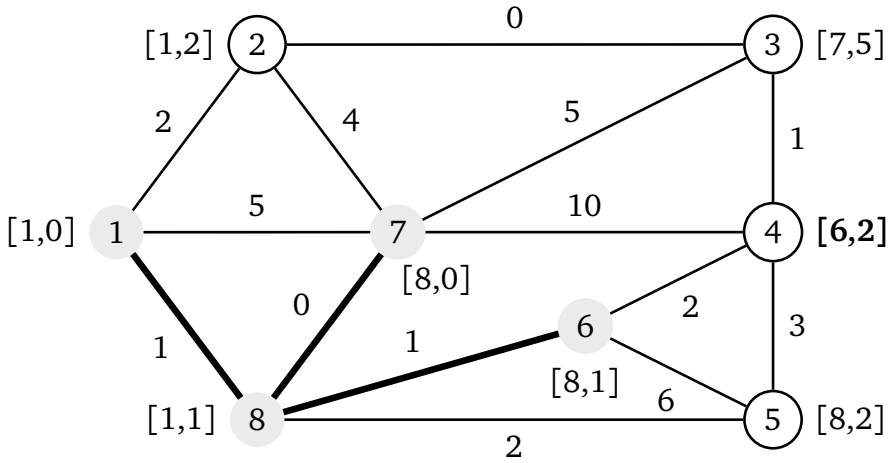


Vertice scelto 8

Algoritmo di Prim-Dijkstra (2^a versione): Esempio (2)

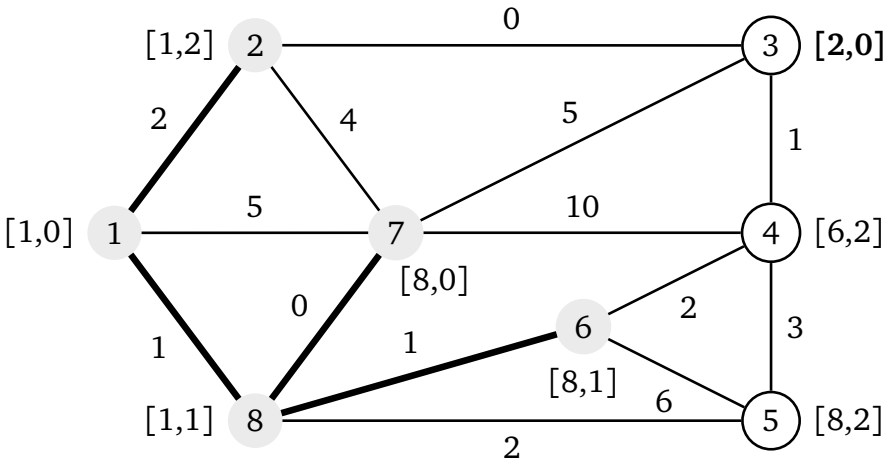


Vertice scelto 7

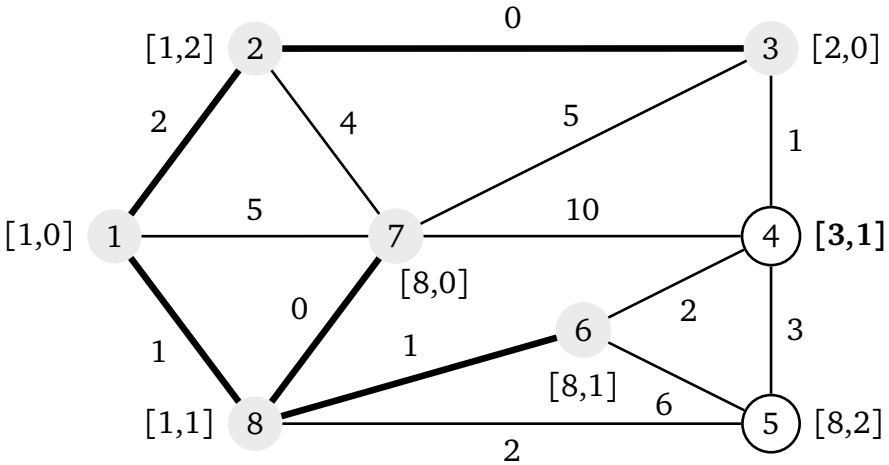


Vertice scelto 6

Algoritmo di Prim-Dijkstra (2^a versione): Esempio (3)

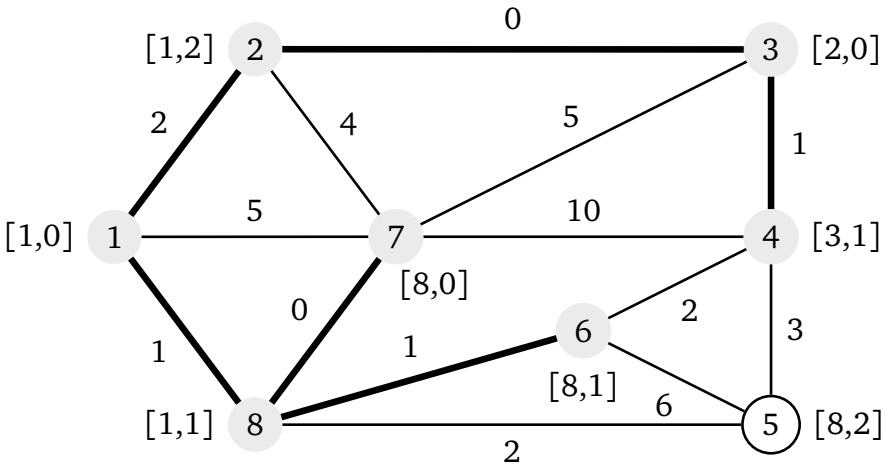


Vertice scelto 2

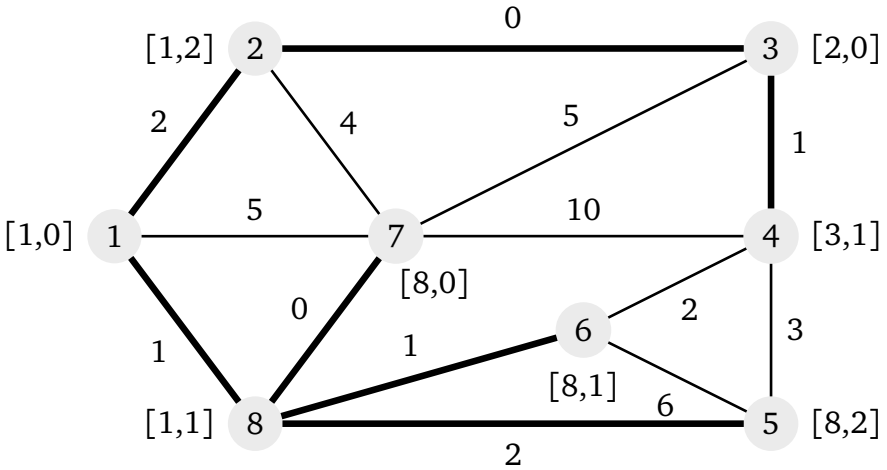


Vertice scelto 3

Algoritmo di Prim-Dijkstra (2^a versione): Esempio (4)



Vertice scelto 4



Vertice scelto 5
Il costo dello SST è 7

Formato Tabellare dell'Algoritmo di Prim-Dijkstra

S	$L[j]$						
	2	3	4	5	6	7	8
$\{1\}$	2	∞	∞	∞	∞	5	**1**
$\{1,8\}$	2	∞	∞	2	1	**0**	1
$\{1,8,7\}$	2	5	10	2	**1**	0	1
$\{1,8,7,6\}$	**2**	5	2	2	1	0	1
$\{1,8,7,6,2\}$	2	**0**	2	2	1	0	1
$\{1,8,7,6,2,3\}$	2	0	**1**	2	1	0	1
$\{1,8,7,6,2,3,4\}$	2	0	1	**2**	1	0	1
$\{1,8,7,6,2,3,4,5\}$	2	0	1	2	1	0	1

Formato Tabellare dell'Algoritmo di Prim-Dijkstra (2)

$pred[j]$						
2	3	4	5	6	7	8
1	-	-	-	-	1	1
1	-	-	8	8	8	1
1	7	7	8	8	8	1
1	7	6	8	8	8	1
1	2	6	8	8	8	1
1	2	3	8	8	8	1
1	2	3	8	8	8	1
1	2	3	8	8	8	1

Algoritmo di Kruskal

- Le path optimality conditions suggeriscono un nuovo algoritmo iterativo per la determinazione dell'albero di copertura T di costo minimo di un grafo non orientato $G(V,E)$.
- Il principio di funzionamento dell'algoritmo è piuttosto semplice:
  - I lati sono inseriti nell'albero T per costi crescenti, quindi devono essere ordinati.
  - Ciascun lato (considerato in ordine di costo crescente) è inserito in T se non forma un ciclo con i lati già in T, altrimenti viene scartato.
  - Se un lato $e = \{i,j\}$ non viene inserito perché forma un ciclo, sicuramente non costerà di meno di quelli già inseriti nel path da i a j, quindi rispetta le *path optimality conditions*.

Algoritmo di Kruskal (2)

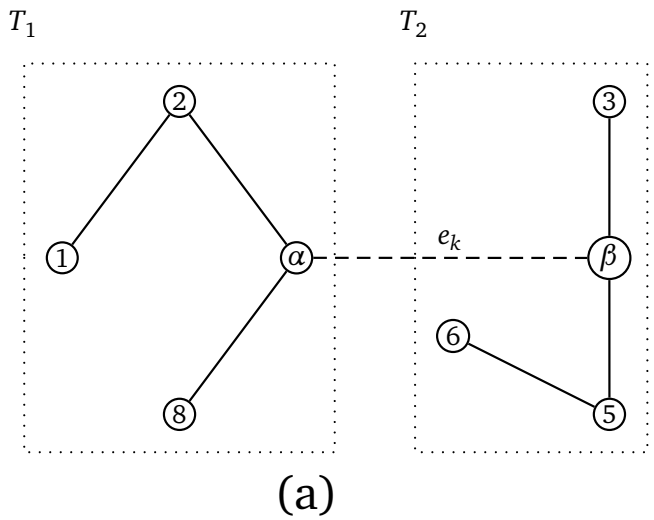
Algoritmo di Kruskal (1^a versione)

Require: Grafo non orientato $G(V,E)$ pesato con costi c_e, $\forall e \in E$;
Ensure: Albero di copertura di costo minimo T^*;
  // Inizializza strutture dati e ordina i lati
  $T^* = \emptyset$;
  Ordina i lati per costi crescenti ($E = \{e_1, \ldots, e_m\}$: $c(e_i) \leq c(e_{i+1})$);
  // Costruisci l'albero di copertura di costo minimo
  $h = 1$;
  while ($|T^*| < n-1$) **and** ($h < m$) **do**
    Se il lato e_h non chiude un ciclo in T^*, allora $T^* = T^* \cup \{e_h\}$;
    $h = h + 1$;
  end while

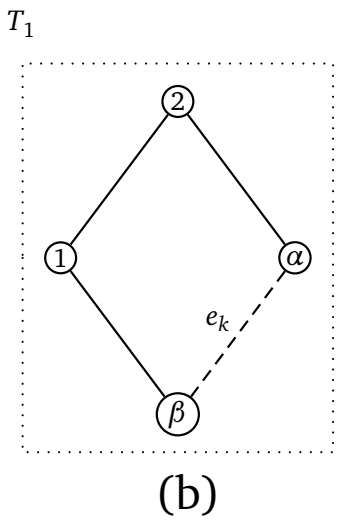
NOTA: Il costo computazionale maggiore è dovuto all'ordinamento degli m lati, quindi la complessità dell'algoritmo è $O(m \log m)$.

Algoritmo di Kruskal (3)

- Per definire una modalità efficiente per decidere se aggiungere a T^* il lato $e_k = (\alpha, \beta)$, si hanno i seguenti casi:
  - (a) α e β fanno parte di due sottoalberi distinti;
  - (b) α e β appartengono al medesimo sottoalbero, quindi l'introduzione di e_k chiude un ciclo.



(a)

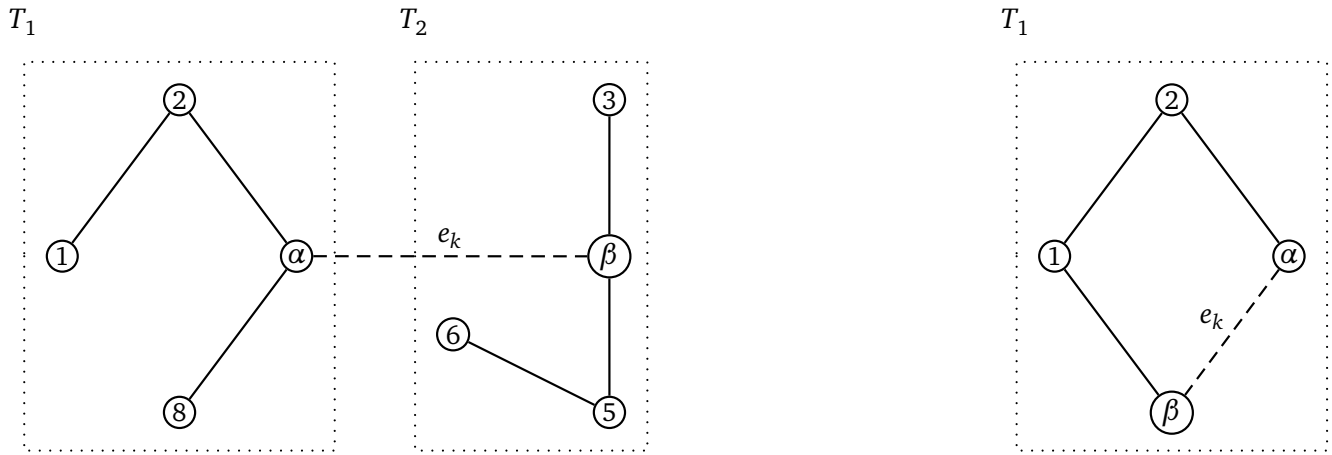


(b)

Algoritmo di Kruskal (4)

- Per verificare se per il lato $e_k = (\alpha, \beta))$ si ha il caso (a) o (b) si associa ad ogni vertice $i \in V$ una etichetta $comp[i]$ dove:

$$comp[i]: \text{ indice del sottoalbero a cui } i \text{ appartiene.}$$



- Nell'esempio (a): $comp[1] = comp[2] = comp[8] = comp[\alpha] = 1$, mentre $comp[3] = comp[5] = comp[6] = comp[\beta] = 2$.

- Nell'esempio (b): $comp[1] = comp[2] = comp[\alpha] = comp[\beta] = 1$.

Algoritmo di Kruskal (5)

- Perciò è semplice verificare che:
  - Se $comp[\alpha] \neq comp[\beta]$ allora si ha il caso (a), ovvero α e β fanno parte di due sottoalberi distinti.
  - Se $comp[\alpha] = comp[\beta]$ allora si ha il caso (b), ovvero α e β fanno parte del medesimo sottoalbero.
- Nel caso (a) si provvederà a inserire il lato $e_k = (\alpha, \beta))$ nell'albero T_1 e a unire l'albero T_1 a T_2.
- Per unire gli alberi T_1 e T_2 si dovrà aggiornare l'etichetta $comp[i]$ dei vertici in T_2 con quella dei vertici in T_1.

Algoritmo di Kruskal (6)

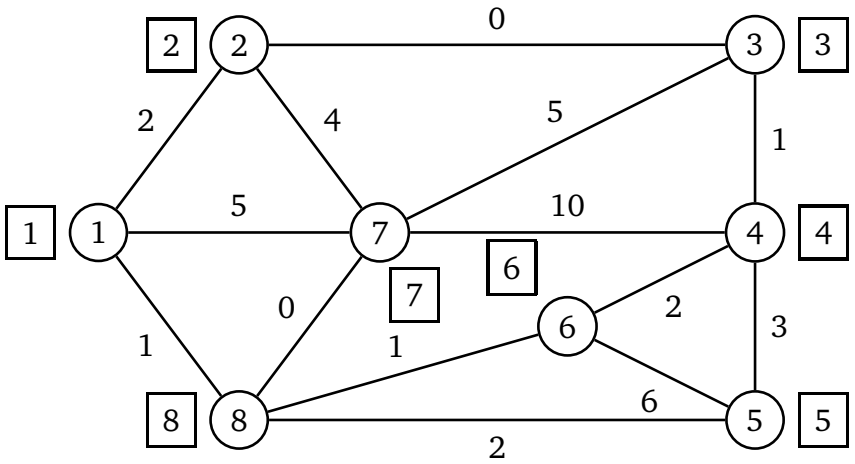
Algoritmo di Kruskal (2^a versione)

Require: Grafo non orientato $G(V,E)$ pesato con costi c_e, $\forall e \in E$;
Ensure: Albero di copertura di costo minimo T^*;
// Inizializza le strutture dati e ordina i lati
Ordina i lati per costi crescenti ($E = \{e_1, \ldots, e_m\} : c(e_i) \leq c(e_{i+1})$);
$k = 0$; $h = 0$;
// Ogni vertice rappresenta una componente distinta
for $i = 1$ to n **do**
  $comp[i] = i$;
end for
// Costruisci l'albero di copertura di costo minimo
while $(k < n-1)$ **and** $(h < m)$ **do**
  Seleziona il lato $e_h = \{i,j\}$;
  $C1 = comp[i]$; $C2 = comp[j]$;

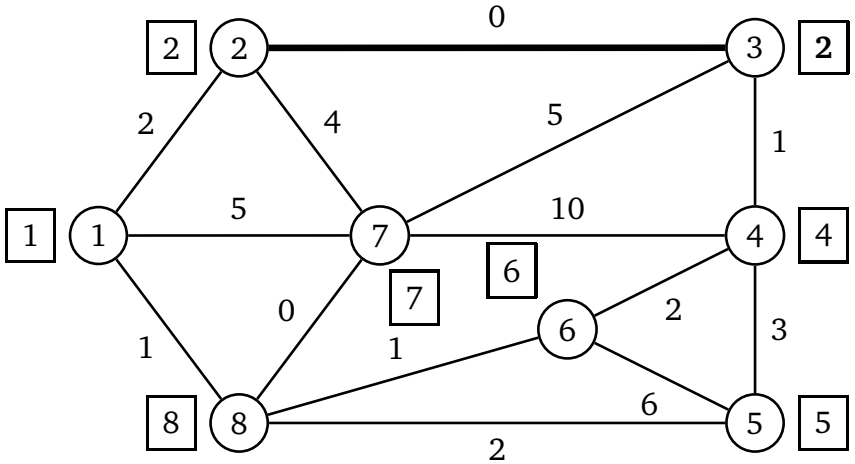
Algoritmo di Kruskal (7)

if $(C1 \neq C2)$ **then**
  Inserisci e_h nell'albero T^*; k=k+1;
  // Unisci le componenti $C1$ e $C2$
  for $q = 1$ to n **do**
    if $(comp[q] = C2)$ **then**
      $comp[q] = C1$;
    end if
  end for
end if
$h = h + 1$;
end while
if $(k \neq n - 1)$ **then**
  il grafo G è disconnesso;
end if

Algoritmo di Kruskal: Esempio

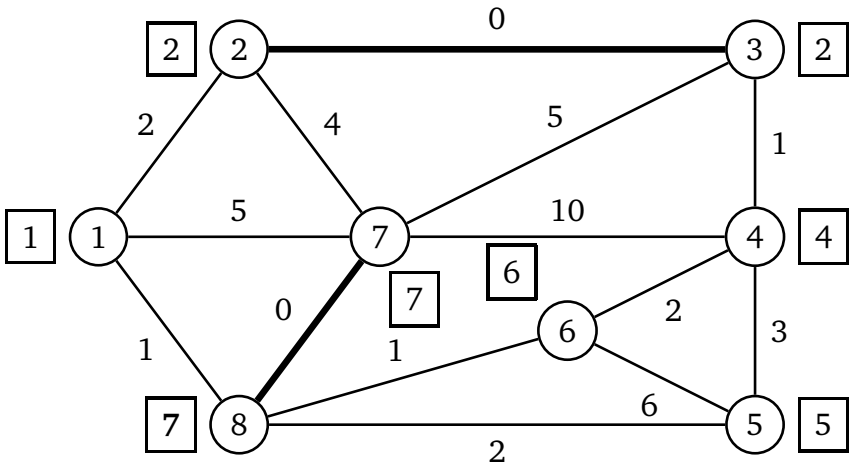


Inizializzazione delle etichette $comp[i]$

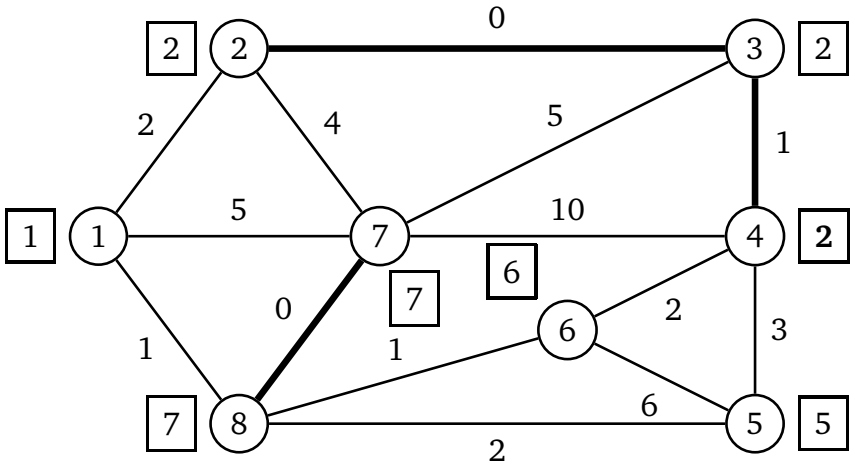


Iterazione 1: inserimento lato $\{2, 3\}$

Algoritmo di Kruskal: Esempio (2)

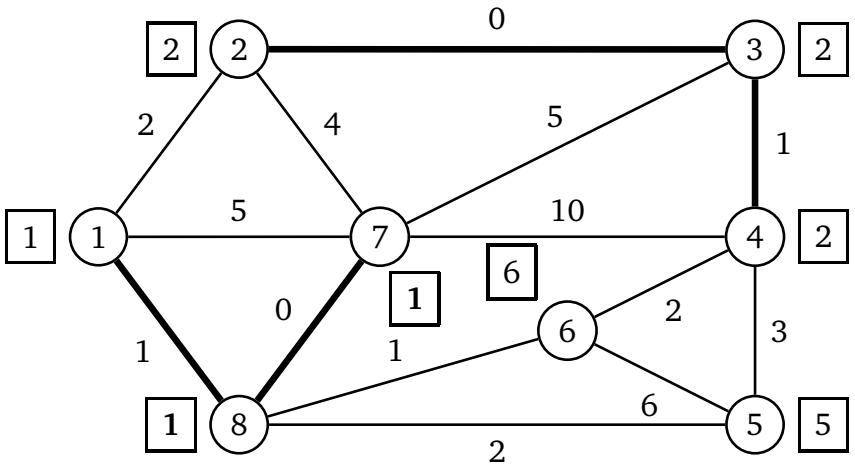


Iterazione 2:
inserimento lato $\{7, 8\}$

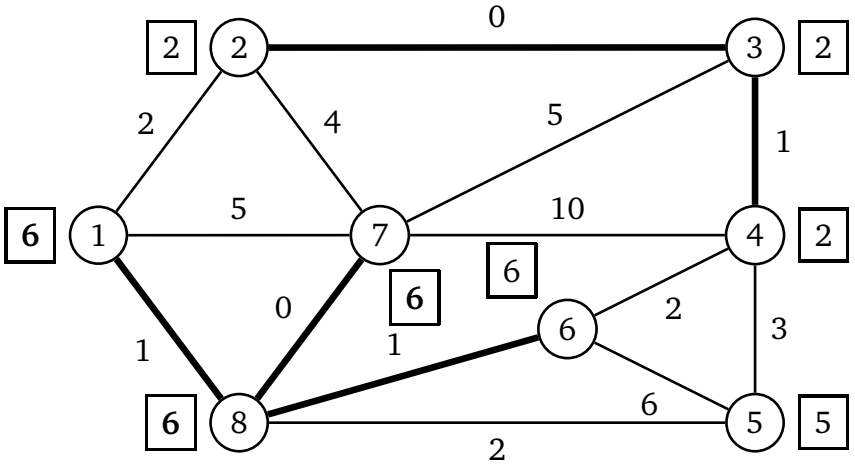


Iterazione 3:
inserimento lato $\{3, 4\}$

Algoritmo di Kruskal: Esempio (3)

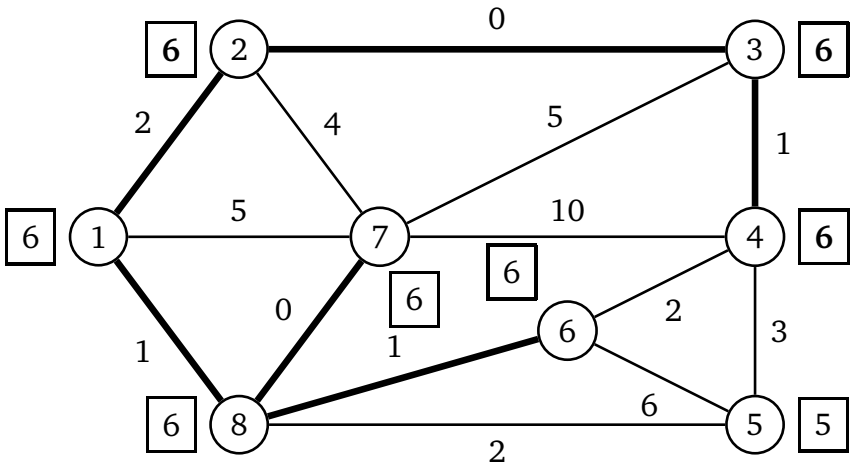


Iterazione 4:
inserimento lato $\{1, 8\}$

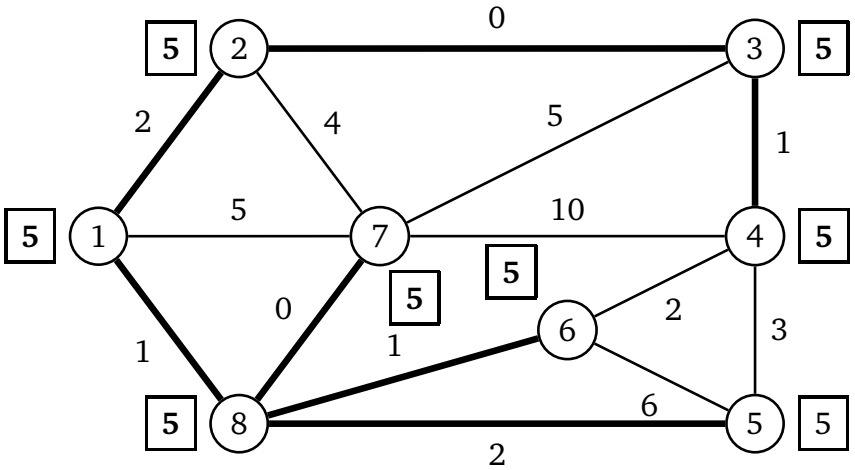


Iterazione 5:
inserimento lato $\{6, 8\}$

Algoritmo di Kruskal: Esempio (4)



Iterazione 6: il lato $\{6,4\}$ chiude un ciclo, inserimento lato $\{1,2\}$



Iterazione 7: inserimento lato $\{5,8\}$