UNIVERSITÀ DEGLI STUDI DI TRIESTE

MODULO 2: Algoritmo di Dijkstra

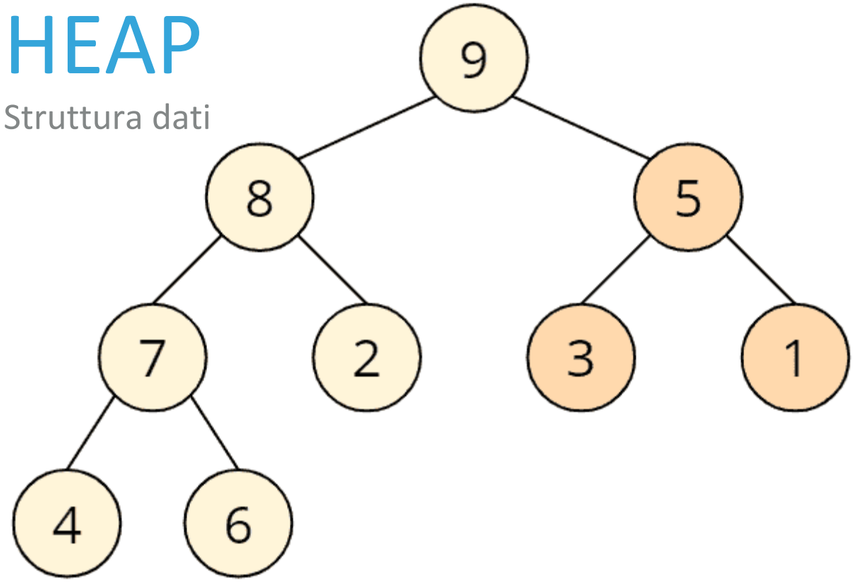
Prof.ssa Giulia Cisotto

giulia.cisotto@units.it

Trieste, 28 maggio 2025

HEAP

Struttura dati

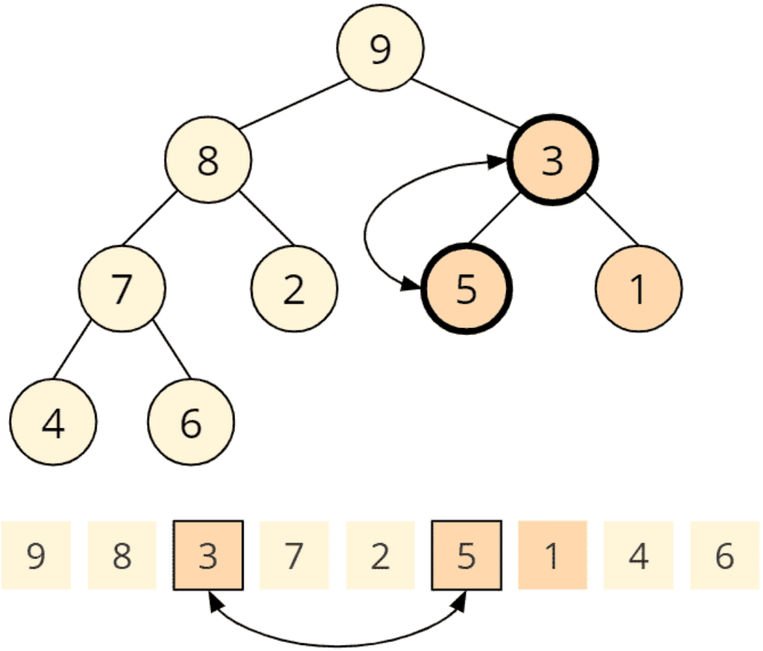


HEAPSORT

Algoritmo di ordinamento:

1. Estraggo il massimo
2. Max-heapify()

$$\boldsymbol{T(n) = O(nlogn)}$$



Dato un grafo G = (V, E) con n vertici ed m archi, si **vuole trovare il percorso di costo minimo** tra uno o più vertici di partenza s e uno o più altri vertici di destinazione d.

COSTO di un percorso $(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$ è la somma dei pesi di tutti gli archi: $\sum_{i=1}^{n} w_{v_i, v_j}$

Anni ’50-’60: Dijkstra, Bellman-Ford-*Moore*, Floyd-Warshall.

Anni **’70–’90:** efficienza, prime euristiche, grafi sparsi, pesi negativi (es. A*)

Dopo **2000:** algoritmi **euristici** e **metaeuristici** per problemi realistici più complessi e vincolati (AI, robotica, logistica).

Anche AI (es. reinforcement learning) si può usare, ma per problemi standard gli algoritmi classici sono ancora i più efficienti ed affidabili.

PUNTO DI PARTENZA:

- Il grafo può essere orientato o non orientato
- *Non sono ammessi cicli negativi*
- I pesi possono essere negativi: **algoritmo di Bellman-Ford**
- I pesi sono solo non negativi: **algoritmo di Dijkstra**

Output dell'algoritmo di Bellman-Ford:

1. Distanze minime da una sorgente a tutti gli altri nodi
2. Cammino minimo per qualsiasi nodo (da predecessori)
3. Rilevamento cicli negativi (se dopo |V|-1 iterazioni ci sono ancora rilassamenti)

Algoritmo di Bellman-Ford

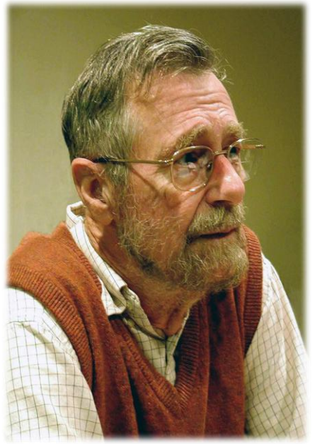
Complessità

$T(V, E) = O(VE)$

```
Parametri: grafo G, nodo sorgente s
# inizialmente impostiamo distanza e predecessore di tutti i nodi
for all v ∈ V:
    distanza[v] = +∞
    predecessore[v] = None
# e poi impostiamo il nodo s come sorgente a distanza 0
distanza[s] = 0
for i in range(0, |V| − 1)
    for all (u,v) ∈ E # effettuiamo il rilassamento di tutti gli archi
        RELAX(u, v, w_{u,v})
```

esplora tutti gli archi

ALGORITMO DI DIJKSTRA



Edsger W. Dijkstra
(1930 – 2002)
fisico olandese

- L’algoritmo di Dijkstra è applicabile nel caso più ristretto in cui tutti i pesi sono non negativi
- Questo caso si presenta spesso in casi pratici: tempi di percorrenza, distanze in km, ecc
- Rispetto all’algoritmo di Bellman-Ford, l’algoritmo di Dijkstra riesce ad essere più efficiente **sfruttando una coda di priorità**: una struttura dati che permette di inserire elementi e rimuovere l’elemento di valore minimo con costo O(log n)

IDEA GENERALE: ALGORITMO DI DIJKSTRA (BASE)

1. Teniamo una **lista/array** di nodi non visitati (all'inizio ci metto tutti i nodi)
2. Definiamo un array/dizionario con distanza minima da sorgente ad ogni nodo
3. Il nodo sorgente ha distanza 0 e tutti gli altri nodi hanno distanza infinita
4. Cerchiamo il nodo v con distanza minima tra quelli non visitati:
   - Per ogni vicino aggiorniamo la distanza vedendo se possiamo raggiungerlo più velocemente passando da v
   - Rimuoviamo v dalla lista dei nodi non visitati
5. Continuiamo finché non ci sono più nodi da visitare

IDEA GENERALE: ALGORITMO DI DIJKSTRA (CON HEAP)

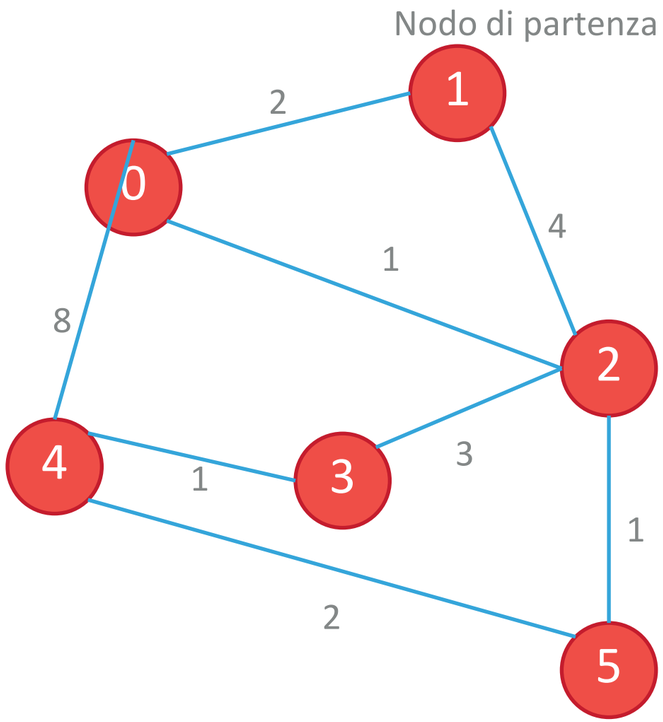
1. Definiamo un array/dizionario con distanza da sorgente ad ogni nodo
2. Inizializziamo il dizionario con nodo sorgente che ha distanza 0 e tutti gli altri con distanza infinita
3. Inizializzo **un min-heap** solo con nodo sorgente come radice
4. Estraggo il nodo v con distanza minima (radice dell'heap). Nota: al primo step tale nodo sarà il nodo sorgente.
5. Dalla matrice/liste di adiacenza, trovo i vicini del nodo v. Per ogni vicino:
   - Aggiorniamo il dizionario/array nel caso in cui rilassiamo l'arco corrispondente (ovvero se costa meno arrivare al vicino, dalla sorgente, passando per il nodo v)
   - Se è così, allora inseriamo il nodo nello heap* e aggiorniamo localmente l'heap (decrease_key()) 6.
6. Ripetiamo punti 4-6 finché l'heap non ha più nodi

PSEUDOCODICE

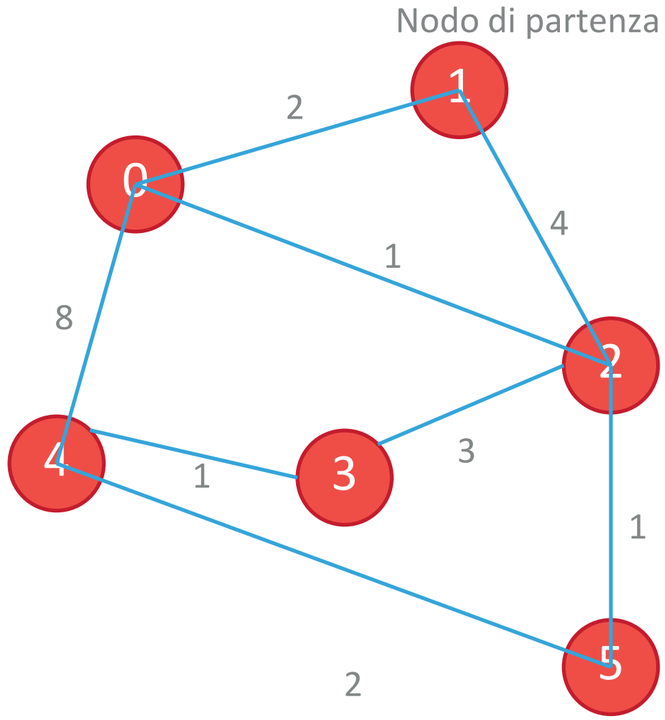
```
Parametri: grafo G, nodo sorgente s
 // inizialmente impostiamo distanza e predecessore di tutti i nodi
for all v ∈ V:
    distanza[v] = +∞
    predecessore[v] = None
distanza[s] = 0 // e poi impostiamo il nodo s come sorgente a distanza 0
S = {}  // insieme dei nodi già visitati (inizialmente vuoto)
Q = coda_di_priorità(V) // coda di priorità con tutti i vertici già inseriti
while Q is non-empty
    u = estrai_minimo(Q)      // estraiamo radice e sistemiamo la heap con heapify_down()
    S = S U {u}               // lo inseriamo nella lista dei visitati
    for all v adiacenti a u   // troviamo i vicini non ancora visitati
        RELAX(u, v, w_{u,v})
         if arco rilassato
            predecessore[v] = u
            DECREASE_KEY(Q,v,dist[v]) // va anche sistemata la heap con heapify_up()
```

esplora solo i vicini del nodo u

ESEMPIO DI ESECUZIONE



ESEMPIO DI ESECUZIONE



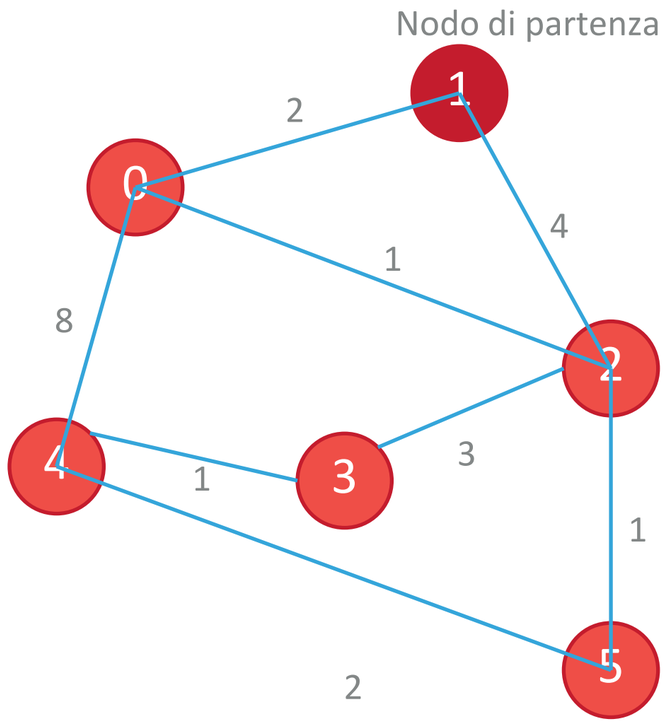
Lista dei nodi non visitati

Vertice	Peso
0	∞
1	0
2	∞
3	∞
4	∞
5	∞

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 1

Lista dei nodi non visitati



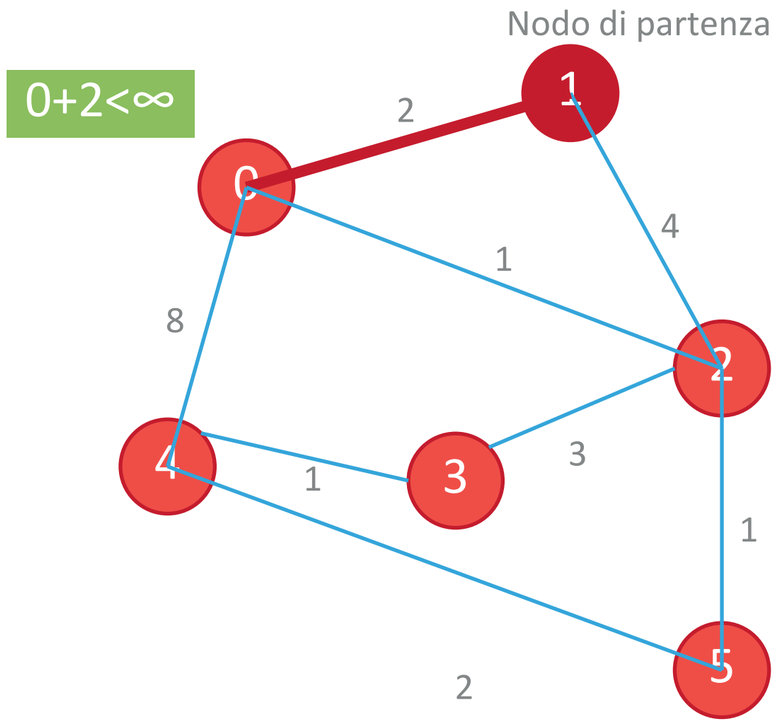
Vertice	Peso
0	∞
1	0
2	∞
3	∞
4	∞
5	∞

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 1

Lista dei nodi non visitati

0	2	3	4	5



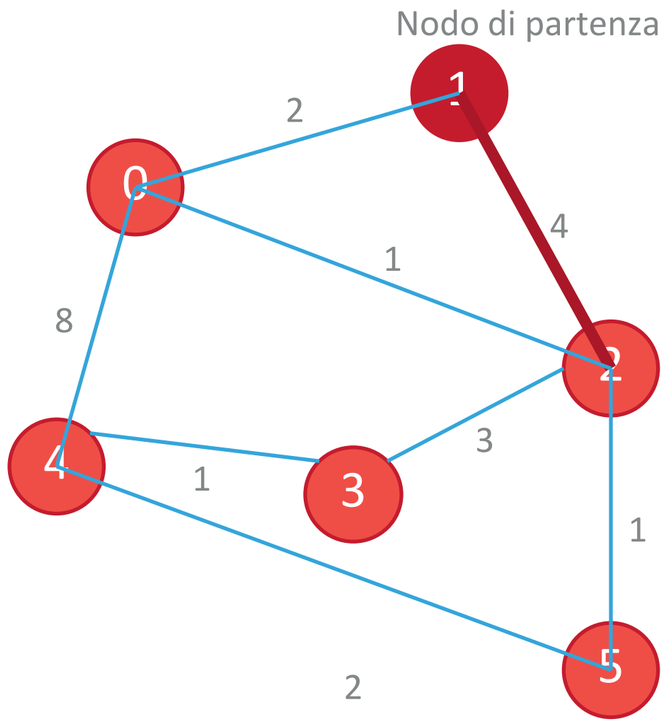
Vertice	Peso
0	2
1	0
2	∞
3	∞
4	∞
5	∞

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 1

Lista dei nodi non visitati

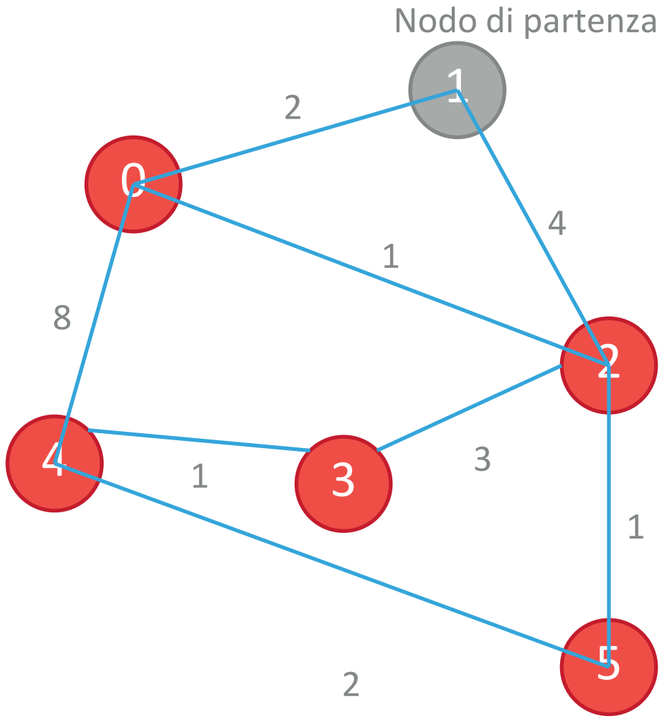
0	2	3	4	5



0+4<∞

Vertice	Peso
0	2
1	0
2	4
3	∞
4	∞
5	∞

ESEMPIO DI ESECUZIONE



Lista dei nodi non visitati

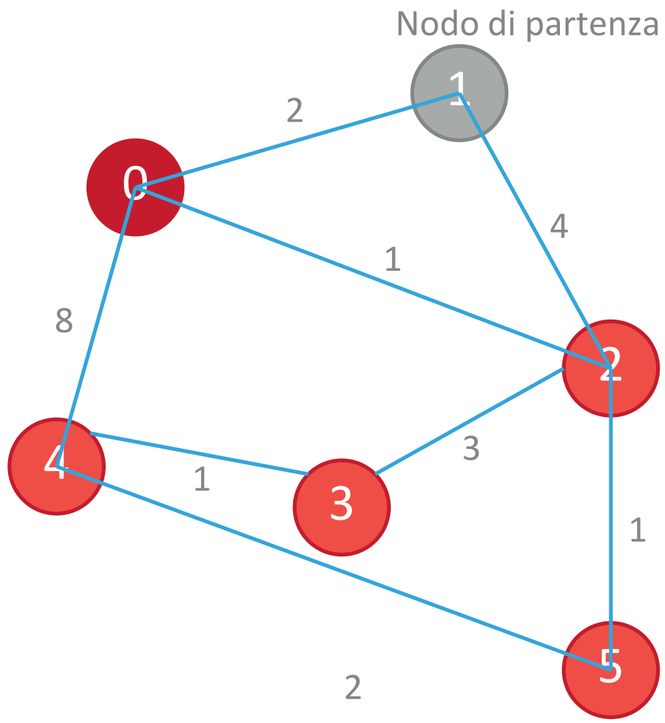
Vertice	Peso
0	2
1	0
2	4
3	∞
4	∞
5	∞

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 0

Lista dei nodi non visitati

2 3 4 5



Vertice	Peso
0	2
1	0
2	4
3	∞
4	∞
5	∞

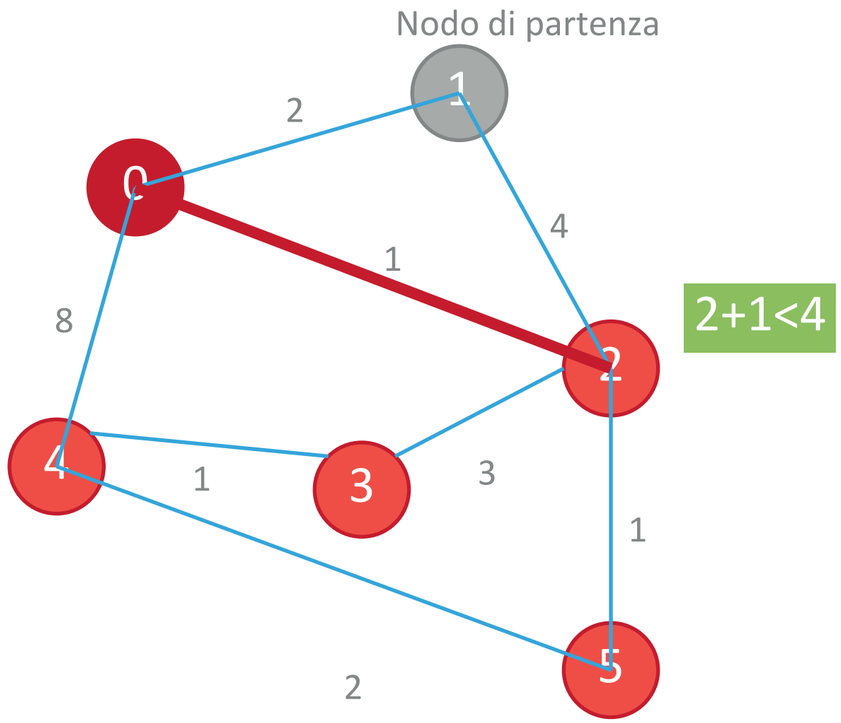
Nota. Scegliamo 0 perché il nodo con distanza minore.

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 0

Lista dei nodi non visitati

2 3 4 5



2+1<4

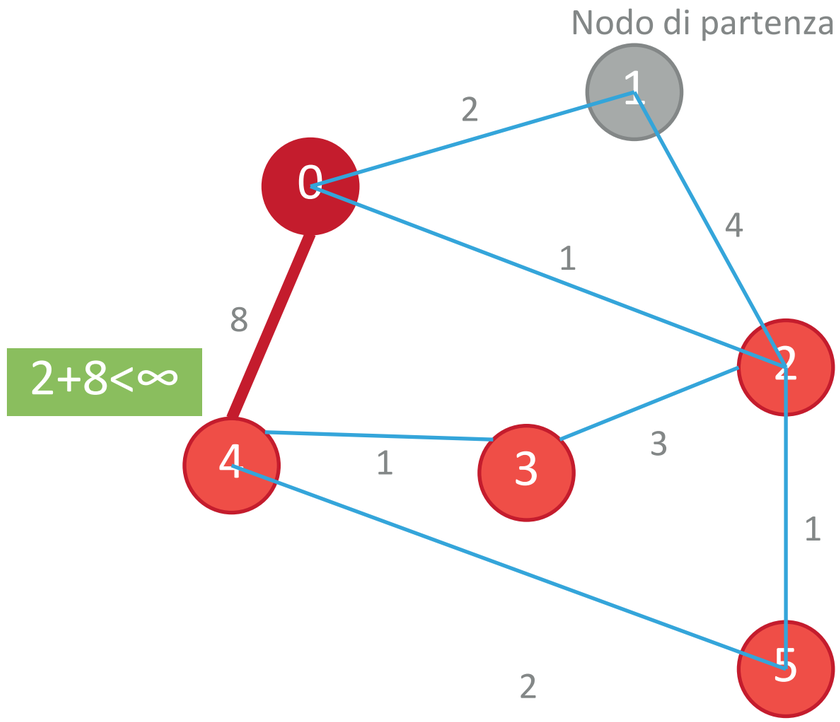
Vertice	Peso
0	2
1	0
2	3
3	∞
4	∞
5	∞

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 0

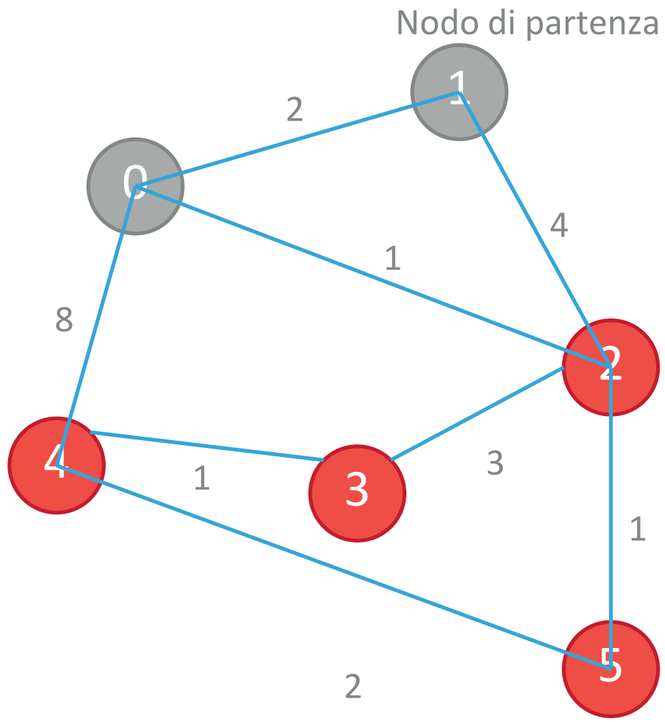
Lista dei nodi non visitati

2 3 4 5



Vertice	Peso
0	2
1	0
2	3
3	∞
4	10
5	∞

ESEMPIO DI ESECUZIONE



Lista dei nodi non visitati

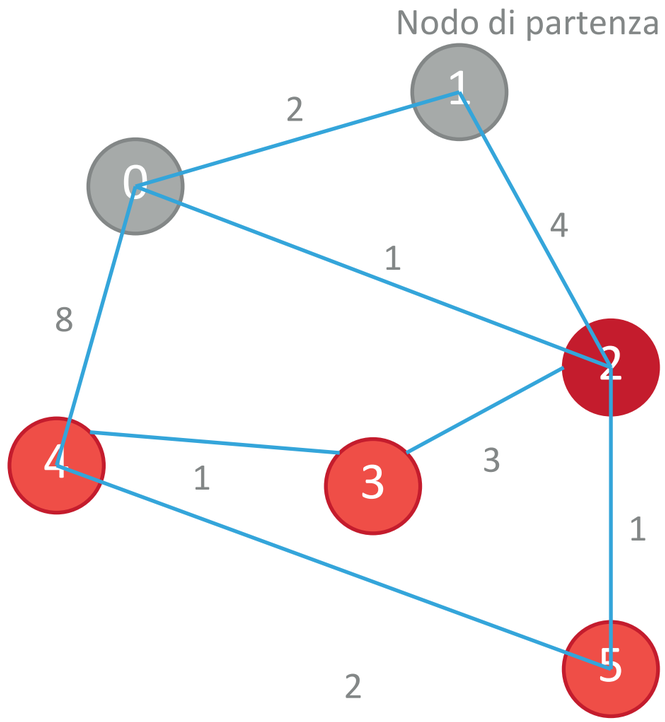
Vertice	Peso
0	2
1	0
2	3
3	∞
4	10
5	∞

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 2

Lista dei nodi non visitati

3 4 5



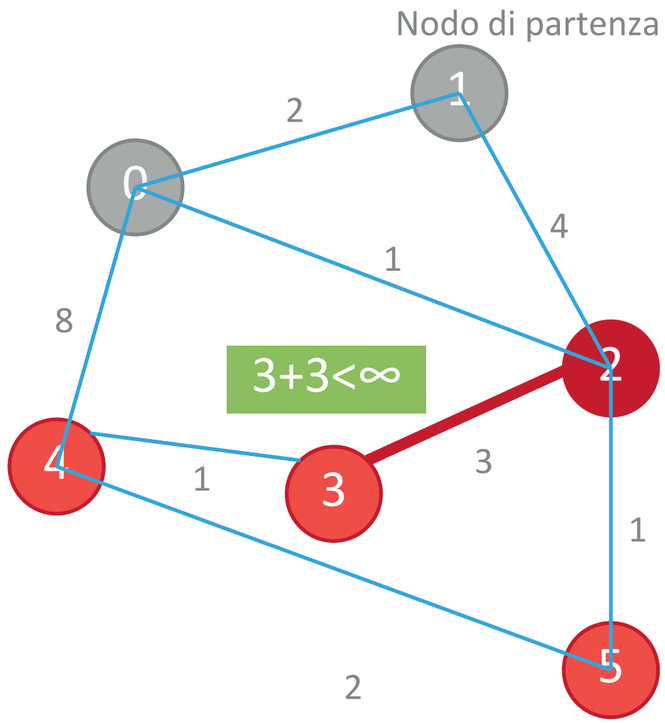
Vertice	Peso
0	2
1	0
2	3
3	∞
4	10
5	∞

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 2

Lista dei nodi non visitati

3 4 5



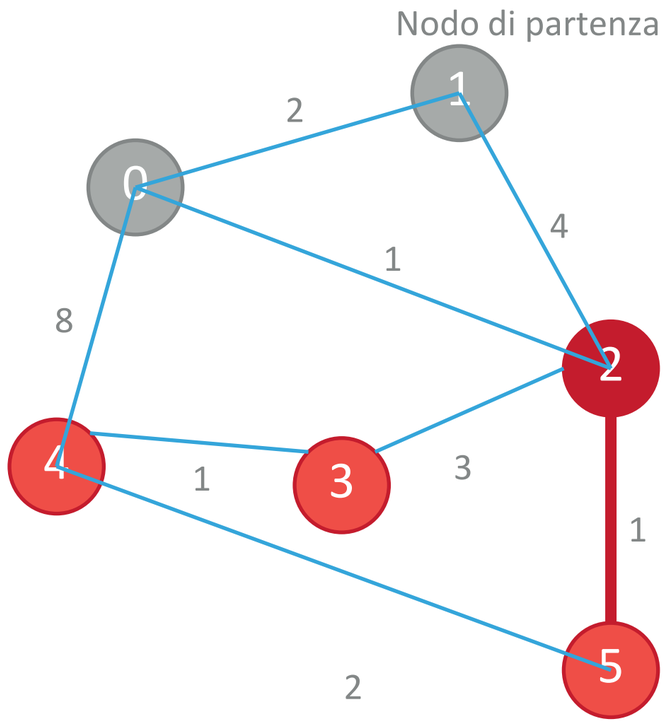
Vertice	Peso
0	2
1	0
2	3
3	6
4	10
5	∞

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 2

Lista dei nodi non visitati

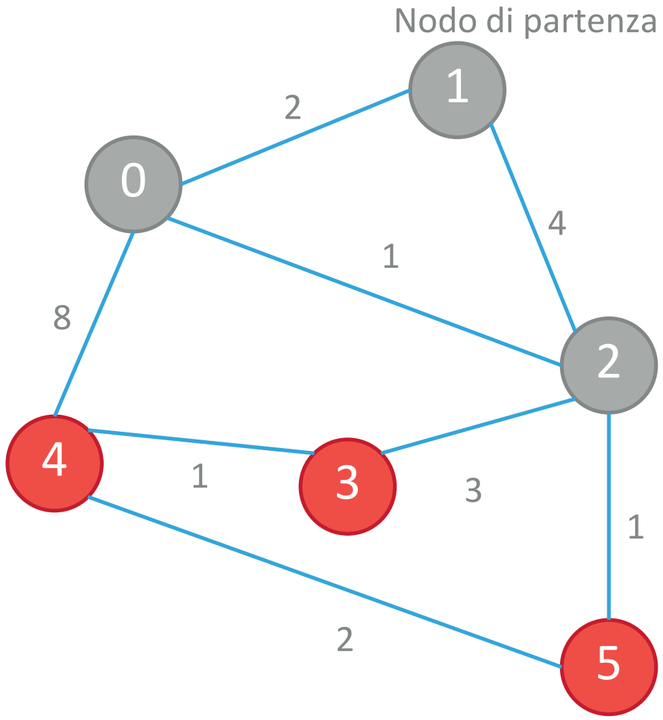
3 4 5



3+1<∞

Vertice	Peso
0	2
1	0
2	3
3	6
4	10
5	4

ESEMPIO DI ESECUZIONE



Lista dei nodi non visitati

3 4 5

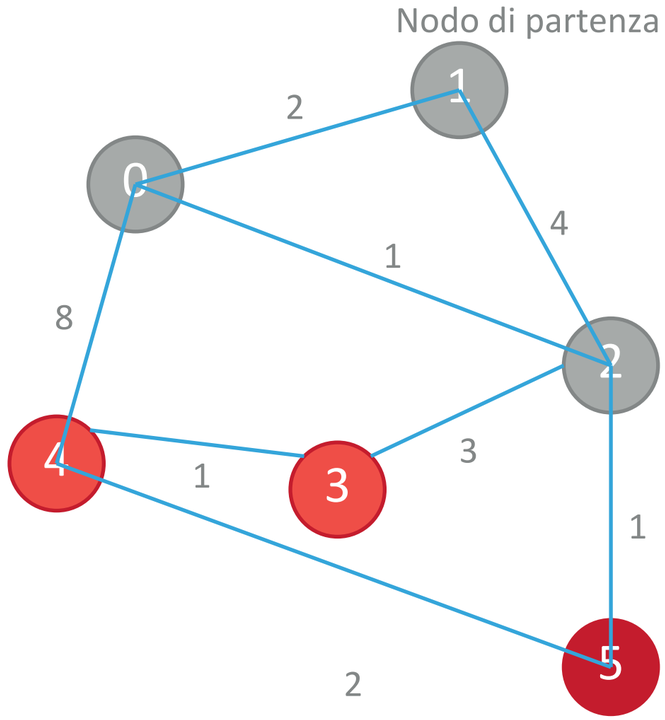
Vertice	Peso
0	2
1	0
2	3
3	6
4	10
5	4

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 5

Lista dei nodi non visitati

3 4



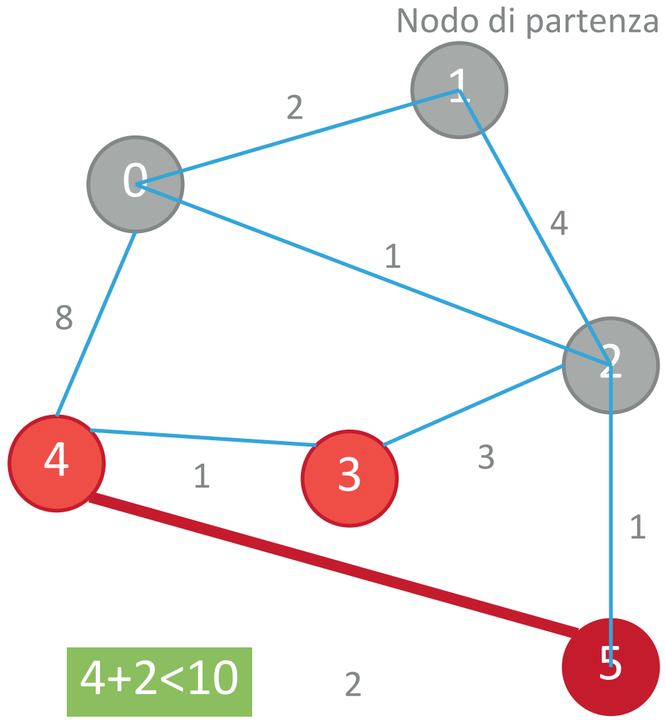
Vertice	Peso
0	2
1	0
2	3
3	6
4	10
5	4

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 5

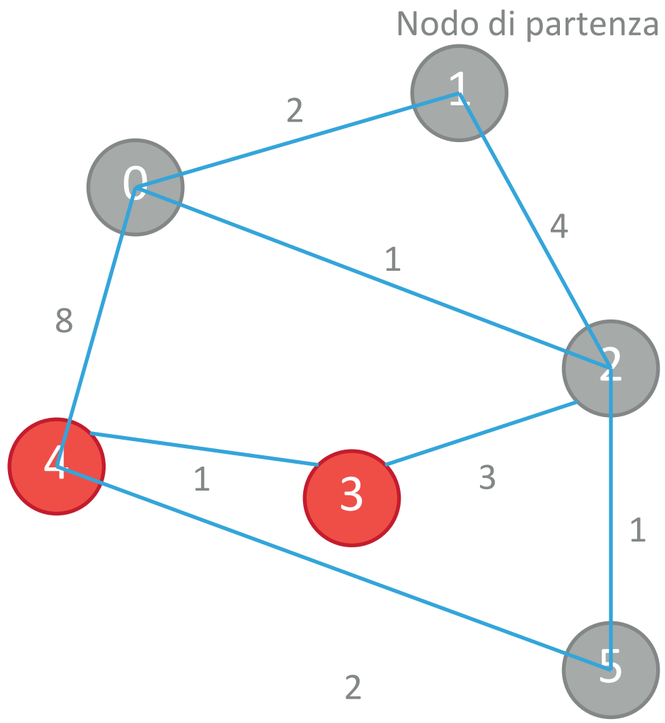
Lista dei nodi non visitati

3 4



Vertice	Peso
0	2
1	0
2	3
3	6
4	6
5	4

ESEMPIO DI ESECUZIONE



Lista dei nodi non visitati

3 4

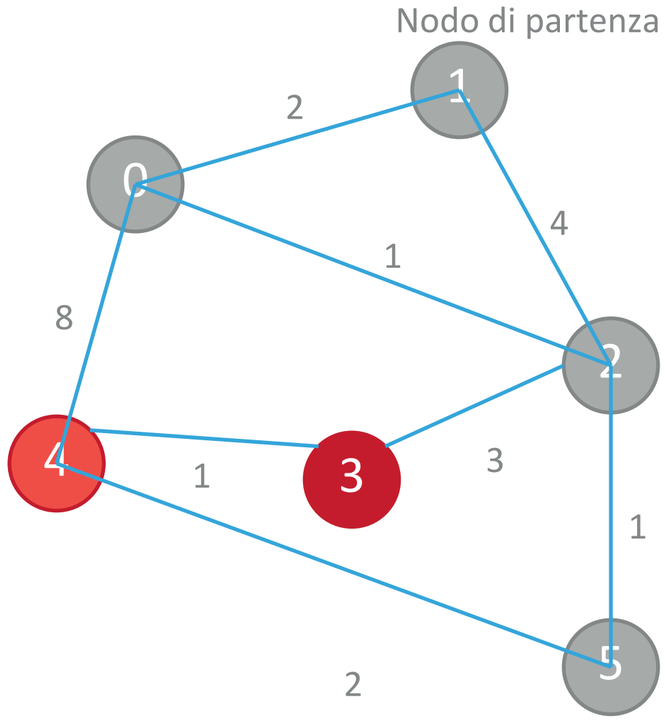
Vertice	Peso
0	2
1	0
2	3
3	6
4	6
5	4

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 3

Lista dei nodi non visitati

4



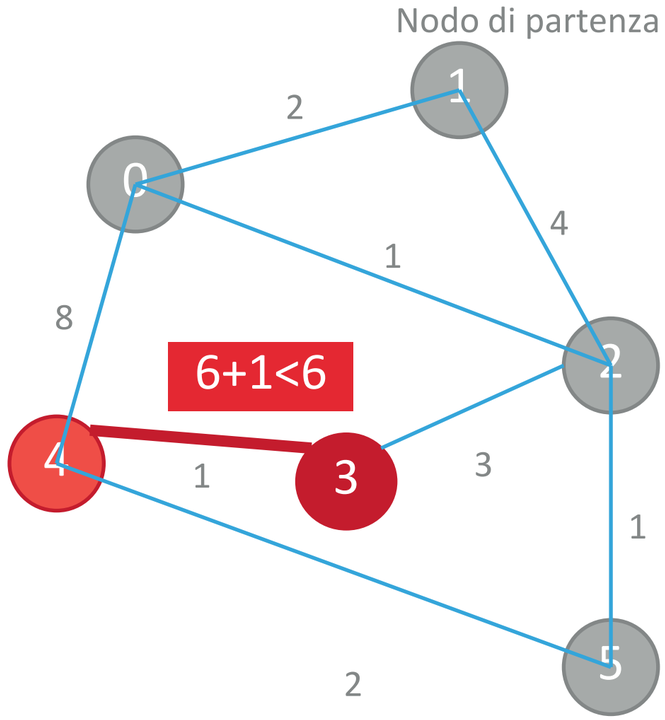
Vertice	Peso
0	2
1	0
2	3
3	6
4	6
5	4

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 3

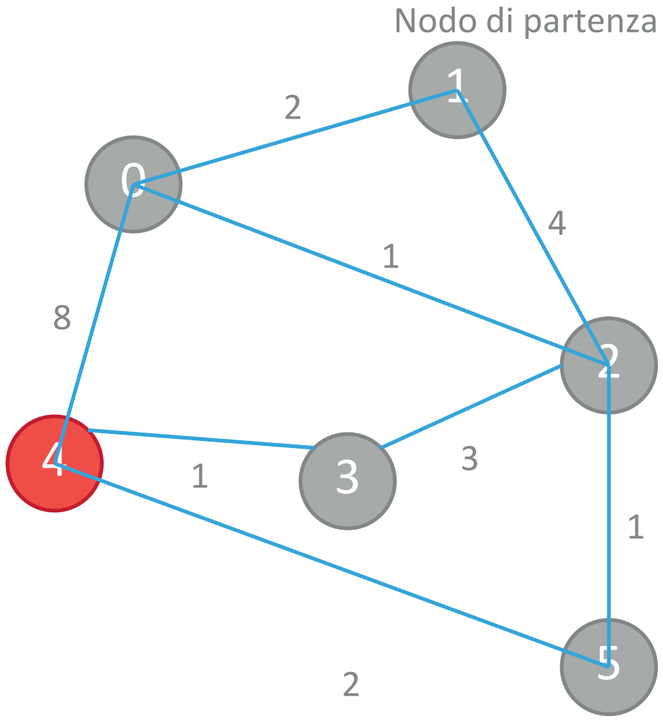
Lista dei nodi non visitati

4



Vertice	Peso
0	2
1	0
2	3
3	6
4	6
5	4

ESEMPIO DI ESECUZIONE



Lista dei nodi non visitati

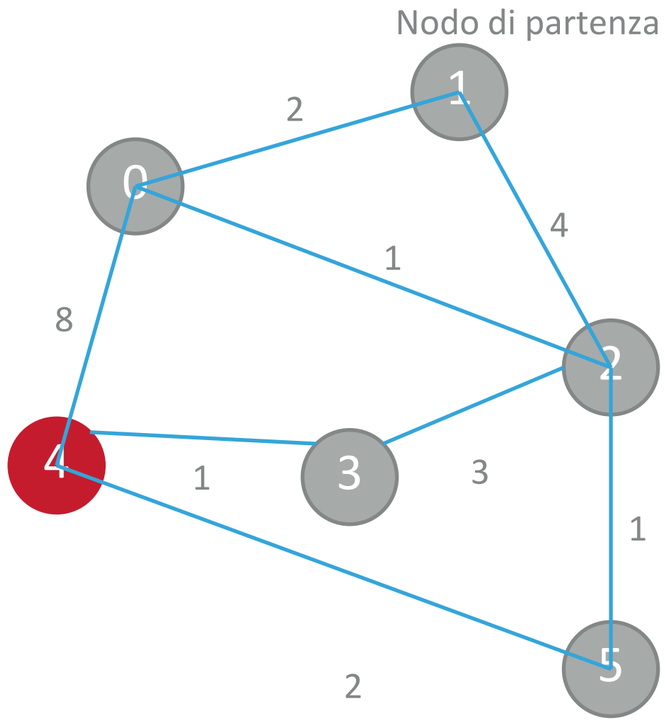
4

Vertice	Peso
0	2
1	0
2	3
3	6
4	6
5	4

ESEMPIO DI ESECUZIONE

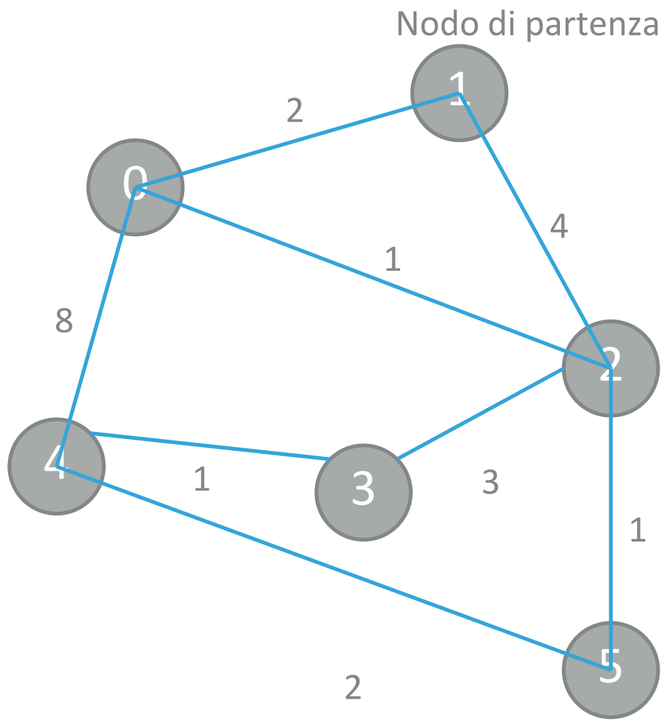
Nodo con distanza minima: 4

Lista dei nodi non visitati



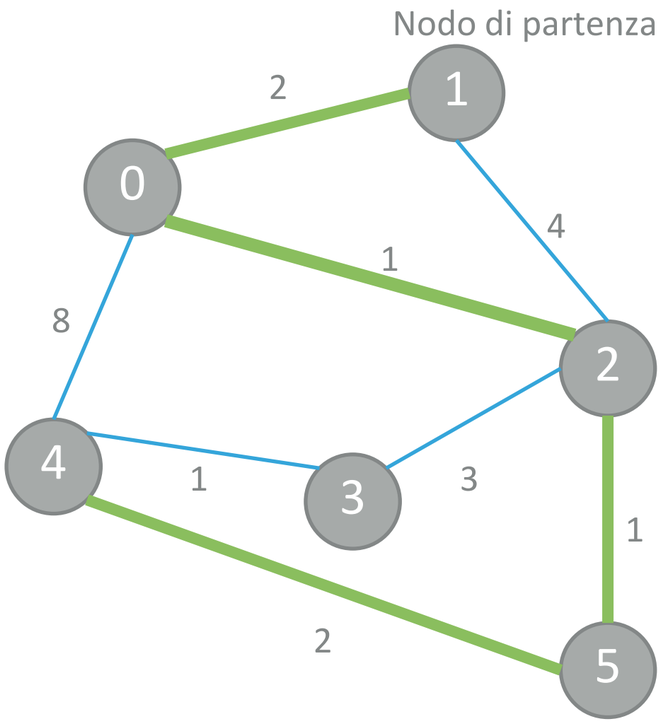
Vertice	Peso
0	2
1	0
2	3
3	6
4	6
5	4

ESEMPIO DI ESECUZIONE



Vertice	Peso
0	2
1	0
2	3
3	6
4	6
5	4

ESEMPIO DI ESECUZIONE



Percorso di lunghezza minima
per andare dal nodo (1) al nodo (4)

Vertice	Peso
0	2
1	0
2	3
3	6
4	6
5	4

CONSIDERAZIONI SUL TEMPO DI CALCOLO

- Visitiamo ogni nodo al più una volta:
  quando lo estraiamo dalla lista
- Visitiamo ogni arco al più una volta:
  la prima volta che incontriamo una delle sue estremità
- Il tempo di calcolo quindi dipende da questi due valori e dalle operazioni di:
  - Trovare il minimo tra i nodi non visitati
  - Aggiornare la distanza dei nodi

Dipendono da come
rappresentiamo il grafo
e le altre strutture
che ci servono

CONSIDERAZIONI SUL TEMPO DI CALCOLO (CON ARRAY)

- Se usiamo un array per mantenere le distanze di ogni nodo:
  - Ogni aggiornamento di distanza richiede $O(1)$, dato che è sufficiente cambiare il valore
  - Trovare il minimo richiede tempo $O(V)$ perché l'array va scandito
  - Otteniamo quindi **un costo totale di $\boldsymbol{O(V^2 + E) = O(V^2)}$** perché per ogni nodo dobbiamo estrarre il minimo e per ogni arco aggiornare un peso

CONSIDERAZIONI SUL TEMPO DI CALCOLO (CON HEAP)

Ogni nodo viene estratto almeno una volta dalla heap (`V` estrazioni)

Ogni estrazione/rimozione del minimo (`extract_min()`) costa $O(\log V)$

Ogni arco può essere potenzialmente aggiornato (`E` aggiornamenti)

Ogni aggiornamento è una modifica della heap che va risistemata (`decrease_key()`): $O(\log V)$

Il costo totale è quindi: $\boldsymbol{O(VlogV + ElogV) = O((V + E)logV)}$

Osservazioni:

- meglio dell'implementazione con array quando $E = o(V^2/\log V)$ archi, ovvero *nel caso di grafi sparsi*
- Fa meglio anche di Bellman-Ford ($\boldsymbol{O(VE)}$)
- Limitato a grafi con pesi non negativi

ALTRI ALGORITMI

Algoritmo	Sorgente → Destinazione	Pesi negativi	Complessità (tempo)	Note principali
Dijkstra	1 → Tutti	❌ No	O((V + E)·log V) (con heap)	Solo per pesi ≥ 0. Molto efficiente su grafi sparsi.
Bellman-Ford	1 → Tutti	✅ Sì	O(V·E)	Gestisce pesi negativi e rileva cicli negativi.
Floyd-Warshall	Tutti → Tutti	✅ Sì	$O(V^3)$	Algoritmo dinamico. Efficiente solo per grafi piccoli.
Johnson	Tutti → Tutti	✅ Sì	$O(V^2 \cdot \log V + V \cdot E)$	Combina Bellman-Ford + Dijkstra. Buono su grafi sparsi.
BFS (su grafi non pesati)	1 → Tutti	❌ No (solo peso = 1)	O(V + E)	Solo per grafi non pesati. Restituisce cammino più corto in passi.
A*	1 → 1	❌ No	Dipende da euristica	Usa funzione euristica. Ottimo in spazi di ricerca noti (es. mappe).
Bidirectional Dijkstra	1 → 1	❌ No	O((V + E)·log V) (in pratica più veloce)	Espande da sorgente e destinazione contemporaneamente.
Ant Colony Optimization	1 → 1 (o più)	✅ (in teoria)	Dipende da iterazioni e formicae	Metaeuristica ispirata al comportamento delle formiche; ottimo in spazi complessi.
Q-learning / RL	1 → 1 (o policy globale)	✅	Dipende da episodi, stato/azione	Impara una politica ottimale tramite esperienza; utile in ambienti dinamici.
Genetic Algorithms	1 → 1 (o k-cammini)	✅	Dipende da popolazione e generazioni	Ottimo per problemi combinatori; non garantisce ottimo globale.

UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

Prossimo laboratorio: 29 ***maggio, h.9:00, aula 4C***