UNIVERSITÀ DEGLI STUDI DI TRIESTE

MODULO 2: Algoritmo di Dijkstra

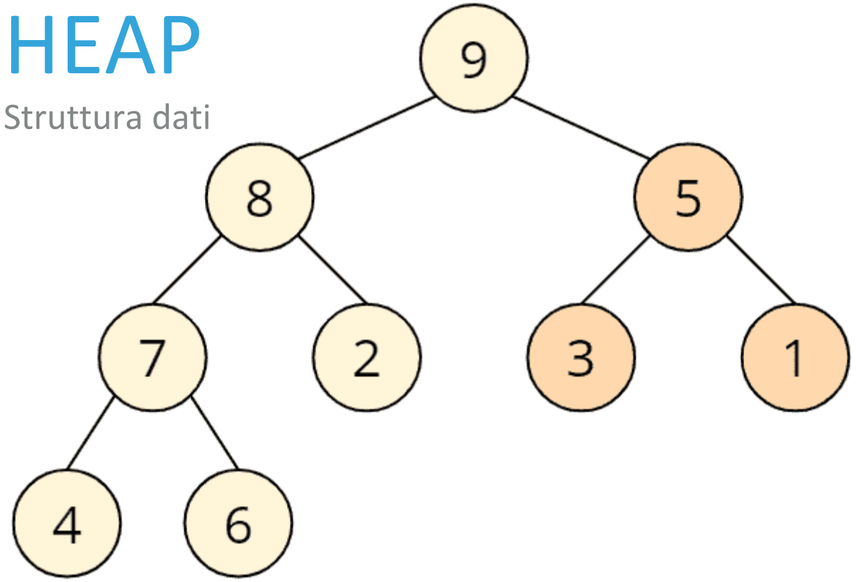
Prof.ssa Giulia Cisotto

giulia.cisotto@units.it

Trieste, 14 maggio 2026

HEAP

Struttura dati

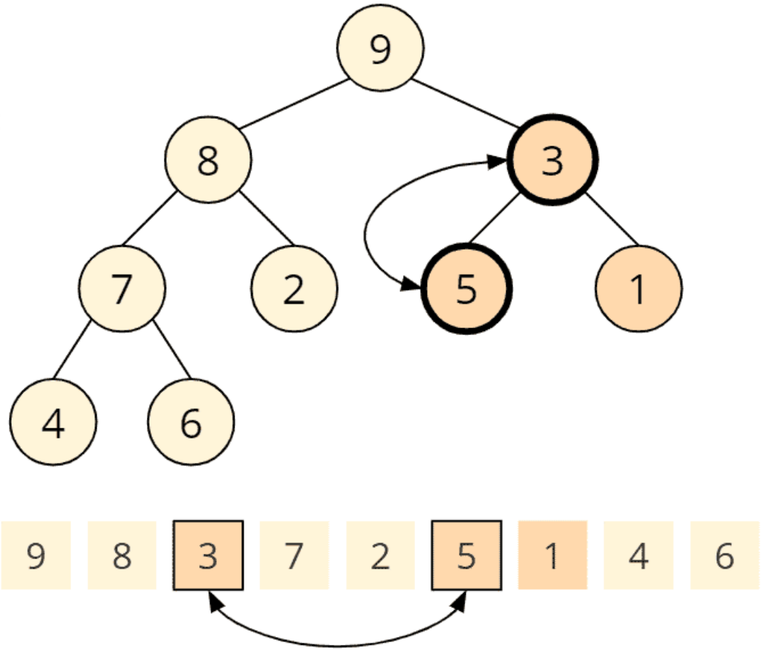


HEAPSORT

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: $\sum_{i=1}^{n} \boldsymbol{w_{v_i,v_j}}$

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

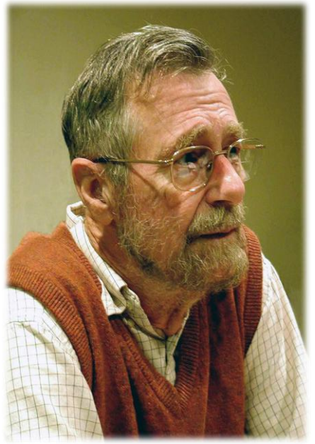
Complessità

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

```
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)

*Una **coda di priorità** in cui l'elemento con massima priorità è quello di valore più piccolo si può implementare con un **min-heap** (un array).

IDEA GENERALE: ALGORITMO DI DIJKSTRA (CON HEAP)

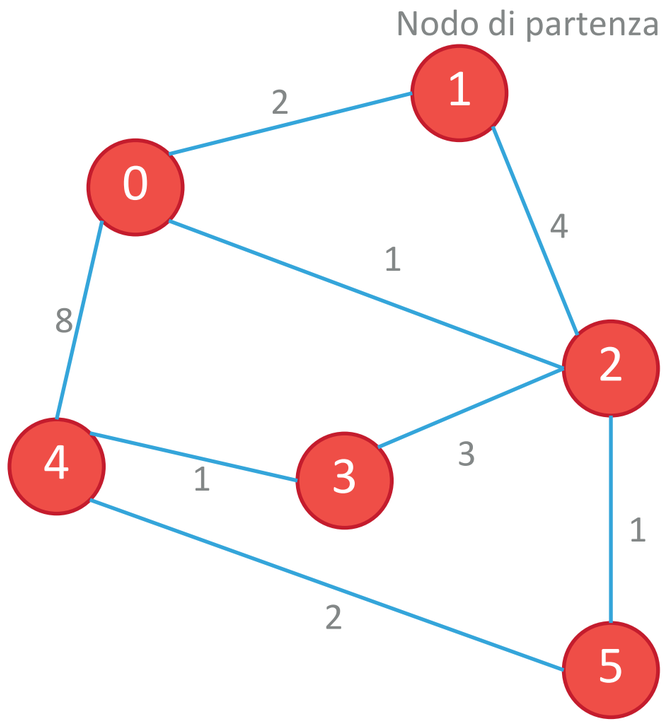
1. Definiamo un array con le distanze dalla sorgente ad ogni nodo
2. Inizializziamo l'array con distanza 0 per il nodo sorgente e +∞ per tutti gli altri
3. Inizializzo **un min-heap** con il nodo sorgente come radice. La coda di priorità conterrà poi i nodi "da esplorare", ordinati per distanza provvisoria minima dalla sorgente.
4. **Estraggo la radice dell'heap** (ovvero il nodo u con distanza minima dalla sorgente). *Nota: al primo step tale nodo sarà il nodo sorgente.*
5. Dalla matrice/liste di adiacenza, identifico i **vicini del nodo u**. Per ogni vicino v:
   - Se serve operiamo il rilassamento (ovvero nel caso distanza[v] > distanza[u] + $w_{u,v}$)
   - Se è così, allora aggiorniamo la priorià del nodo v nello heap e aggiorniamo localmente l'heap (decrease_key())
6. Ripetiamo punti 4-6 finché l'heap non ha più nodi

PSEUDOCODICE

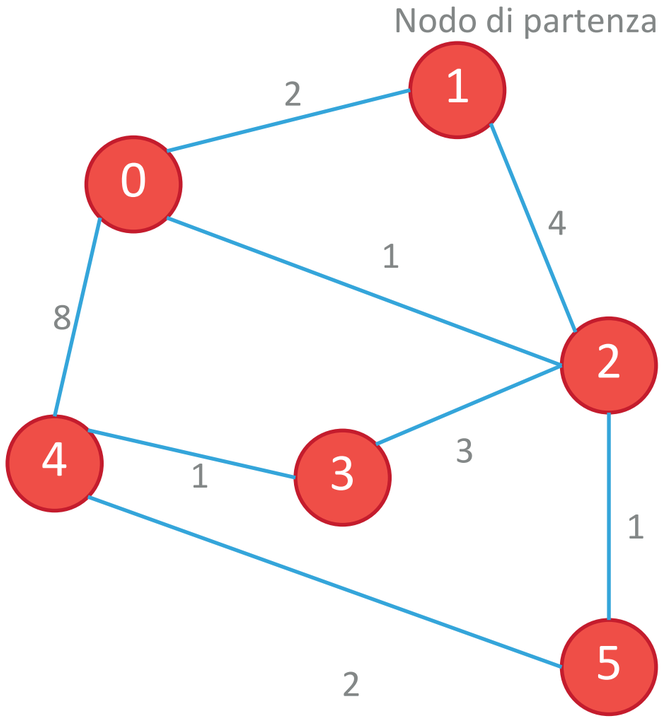
```
Parametri: grafo G, nodo sorgente s
 // inizialmente impostiamo distanza e predecessore di tutti i nodi
for all v ∈ V:
    distanza[v] = +∞
    predecessore[v] = None
distanza[s] = 0 // e poi impostiamo il nodo s come sorgente a distanza 0
Q = coda_di_priorità(V) // min-heap con tutti i vertici(radice=sorgente)
while Q is non-empty
    u = estrai minimo(Q)      // estraiamo radice e sistemiamo min-heap
    for all v adiacenti a u & v ∈ Q // vicini (archi uscenti da v) non ancora visitati
        if distanza[v] > distanza[u] + w_{u,v}   // rilasso
            RELAX(u, v, w_{u,v})
            predecessore[v] = u
            DECREASE_KEY(Q,v,distanza[v]) // sistemo la coda di priorità (min-heap)
```

esplora solo i vicini del nodo u

ESEMPIO DI ESECUZIONE



ESEMPIO DI ESECUZIONE



Coda di priorità (min-heap) dei nodi non visitati

1	0	2	3	4	5

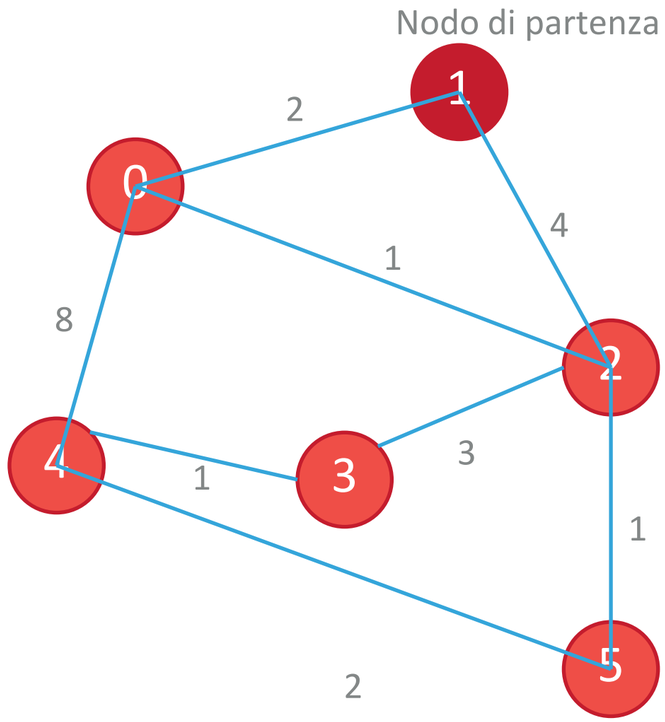
Vertice	Peso	Pred
0	∞	-
1	0	-
2	∞	-
3	∞	-
4	∞	-
5	∞	-

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 1

Coda di priorità (min-heap) dei nodi non visitati

0 2 3 4 5



Vertice	Peso	Pred
0	∞	-
1	0	-
2	∞	-
3	∞	-
4	∞	-
5	∞	-

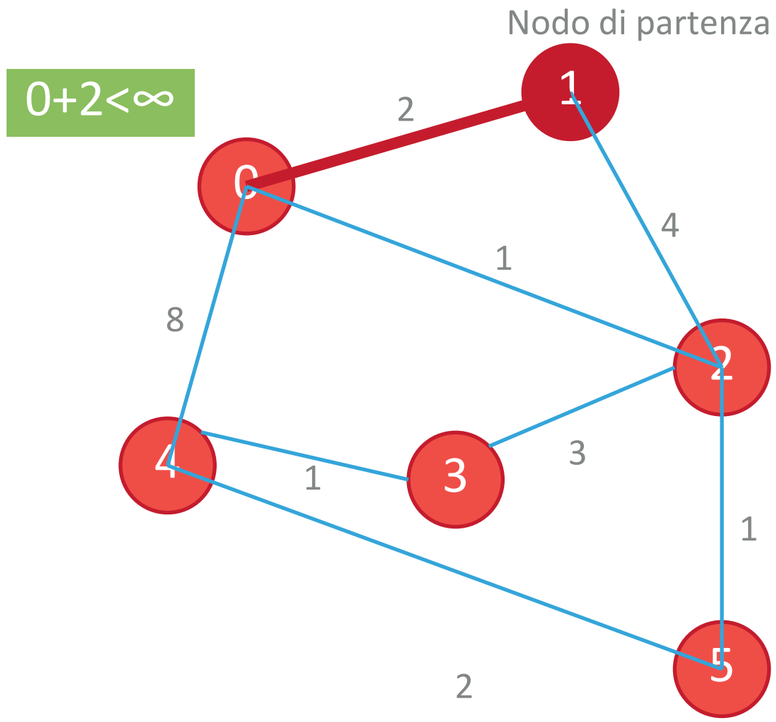
ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 1

Coda di priorità (min-heap) dei nodi non visitati

0 2 3 4 5

La coda è già un min-heap!



Vertice	Peso	Pred
0	2	1
1	0	-
2	∞	-
3	∞	-
4	∞	-
5	∞	-

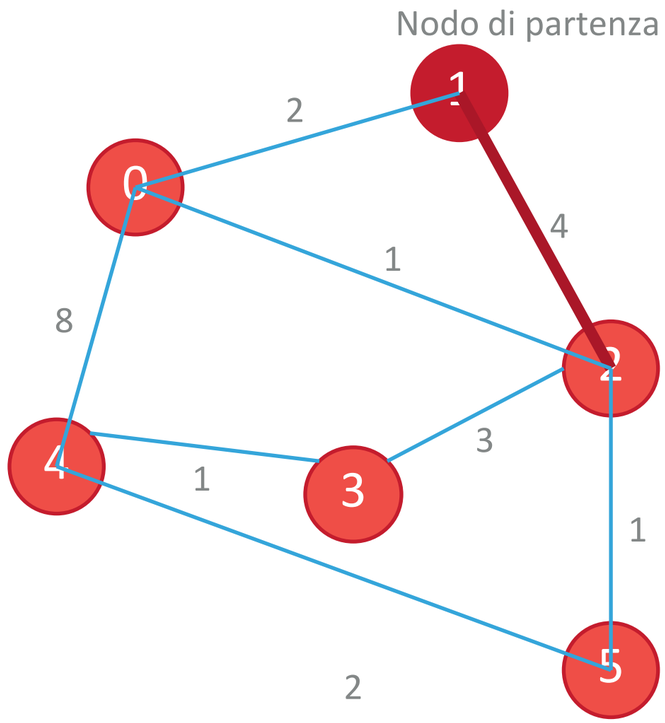
ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 1

Coda di priorità (min-heap) dei nodi non visitati

0 2 3 4 5

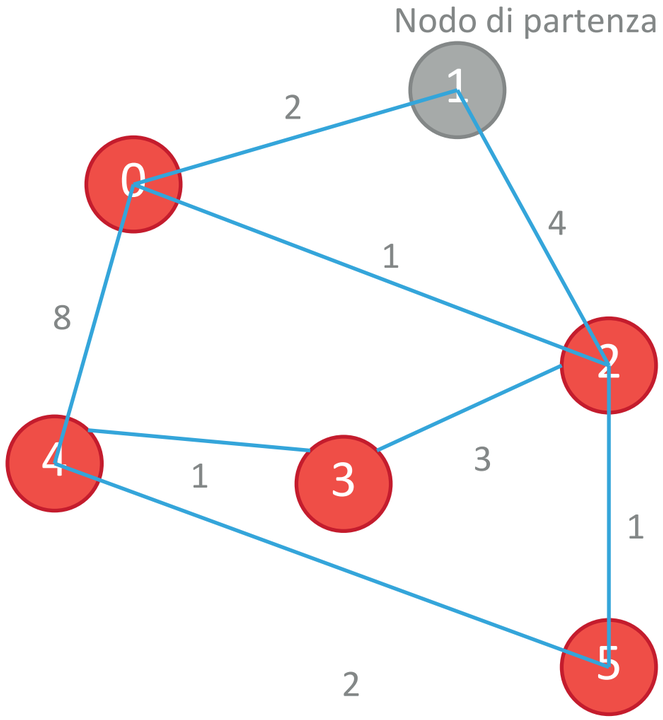
La coda è già un min-heap!



0+4<∞

Vertice	Peso	Pred
0	2	1
1	0	-
2	4	1
3	∞	-
4	∞	-
5	∞	-

ESEMPIO DI ESECUZIONE



Coda di priorità (min-heap) dei nodi non visitati

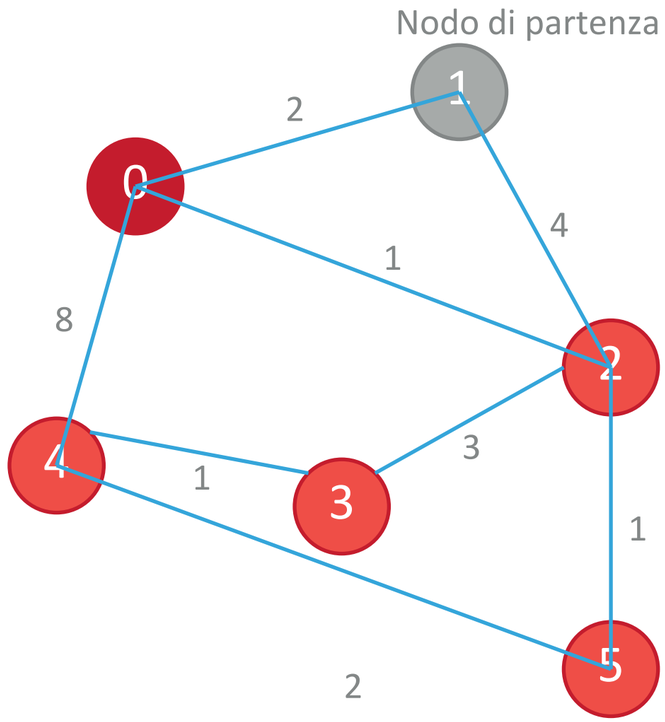
Vertice	Peso	Pred
0	2	1
1	0	-
2	4	1
3	∞	-
4	∞	-
5	∞	-

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 0

Coda di priorità (min-heap) dei nodi non visitati

2 3 4 5



Vertice	Peso	Pred
0	2	1
1	0	-
2	4	1
3	∞	-
4	∞	-
5	∞	-

Nota. Scegliamo 0 perché il nodo con distanza minore.

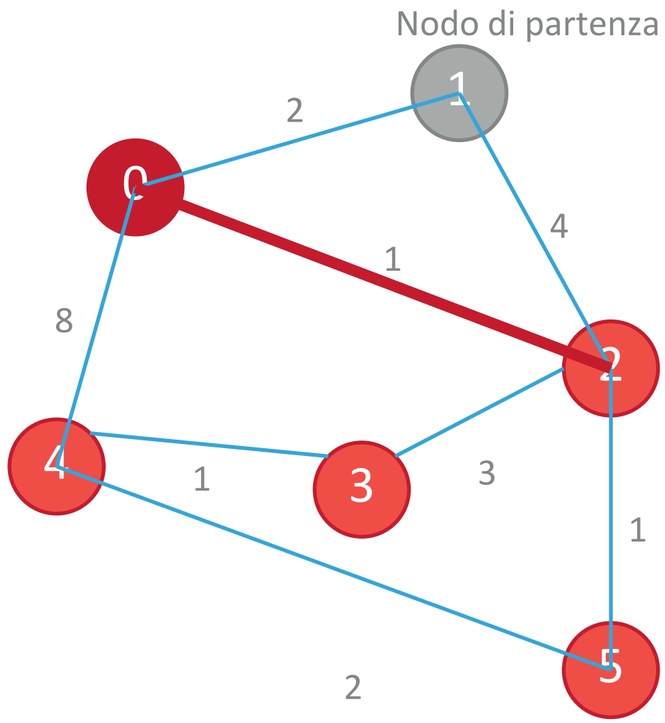
ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 0

Coda di priorità (min-heap) dei nodi non visitati

2 3 4 5

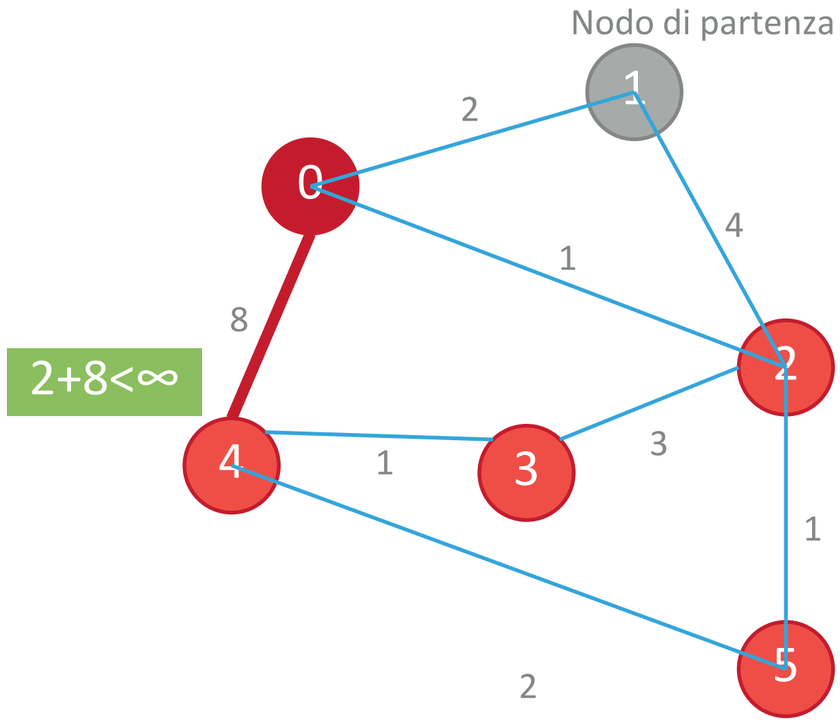
La coda è già un min-heap!



2+1<4

Vertice	Peso	Pred
0	2	1
1	0	-
2	3	0
3	∞	-
4	∞	-
5	∞	-

ESEMPIO DI ESECUZIONE



Nodo con distanza minima: 0

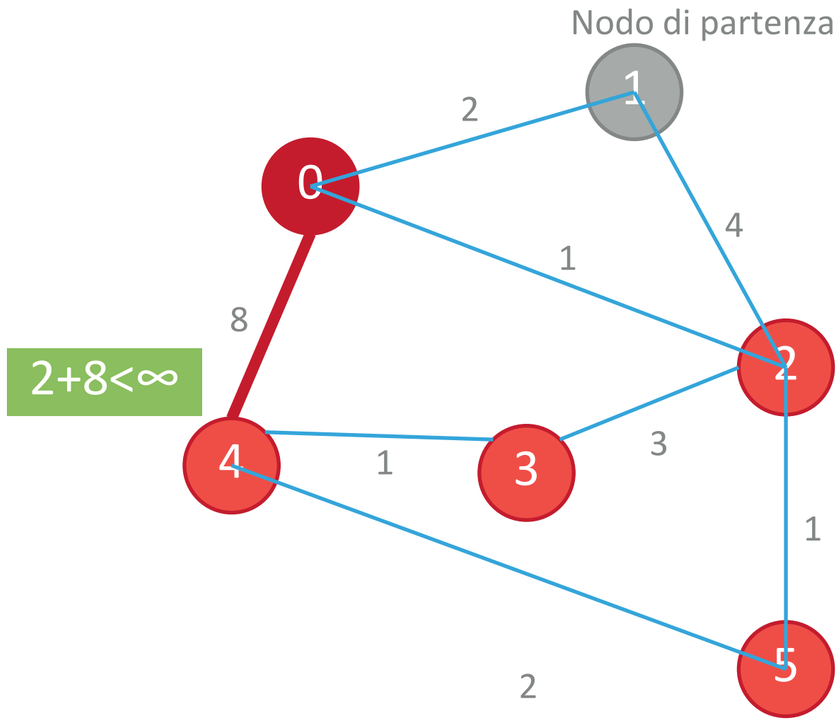
Coda di priorità (min-heap) dei nodi non visitati

2 3 4 5

La coda non è un min-heap!

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

ESEMPIO DI ESECUZIONE



Nodo con distanza minima: 0

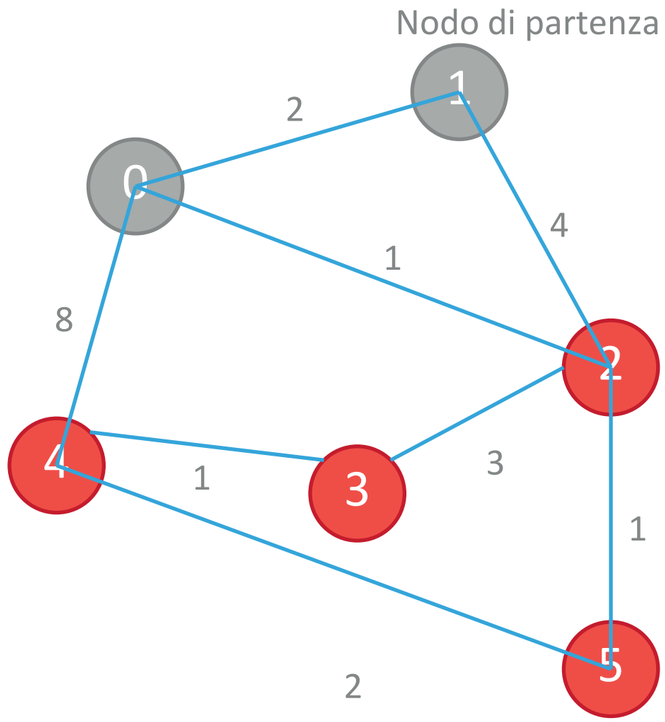
Coda di priorità (min-heap) dei nodi non visitati

2 4 3 5

Adesso la coda è un min-heap!

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

ESEMPIO DI ESECUZIONE



Coda di priorità (min-heap) dei nodi non visitati

2 4 3 5

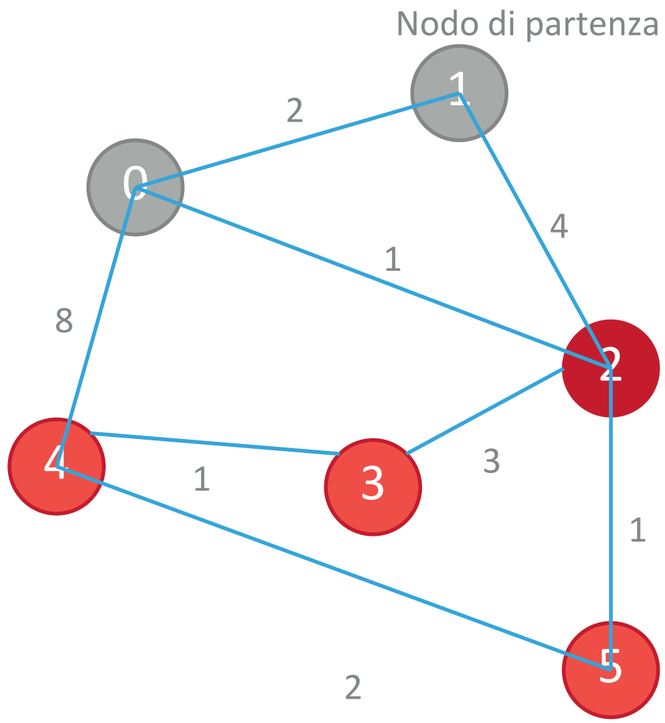
Vertice	Peso	Pred
0	2	1
1	0	-
2	3	0
3	∞	-
4	10	0
5	∞	-

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 2

Coda di priorità (min-heap) dei nodi non visitati

4 3 5



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

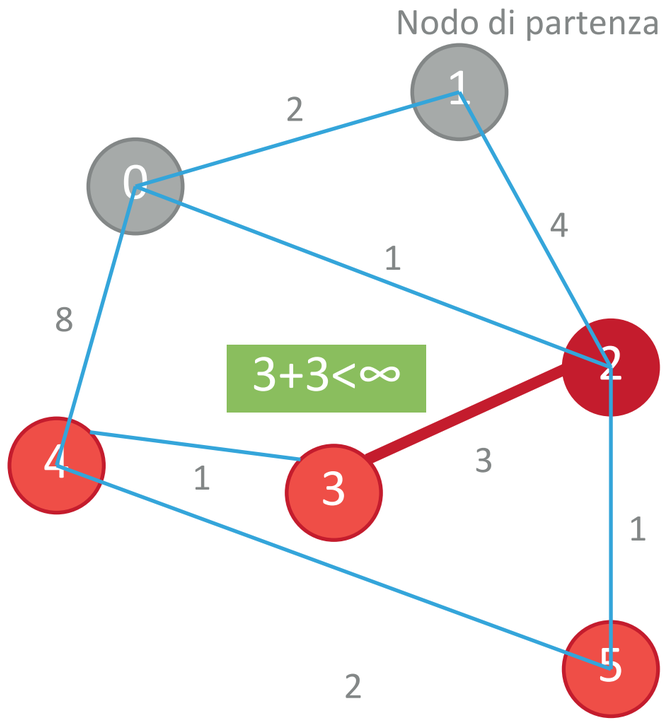
ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 2

Coda di priorità (min-heap) dei nodi non visitati

4 3 5

La coda non è un min-heap!



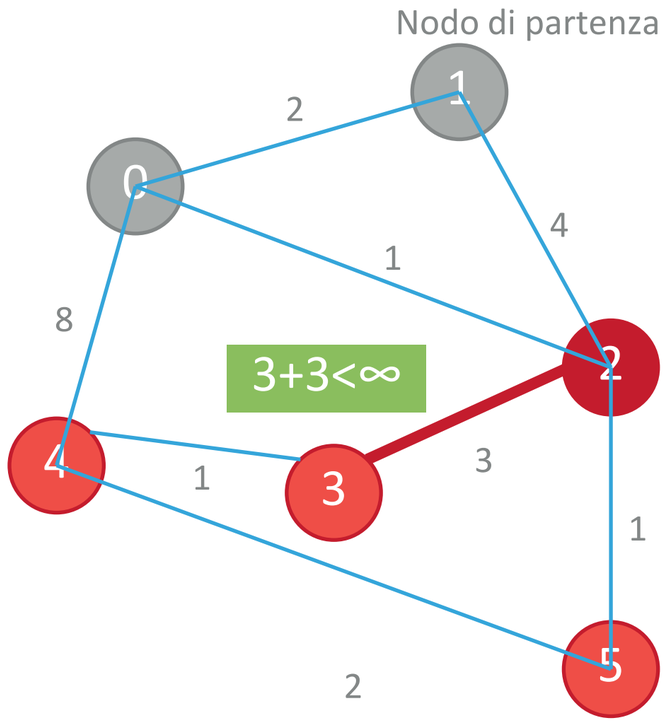
Vertice	Peso	Pred
0	2	1
1	0	-
2	3	0
3	6	2
4	10	0
5	∞	-

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 2

Coda di priorità (min-heap) dei nodi non visitati

3 4 5



Adesso la coda è un min-heap!

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

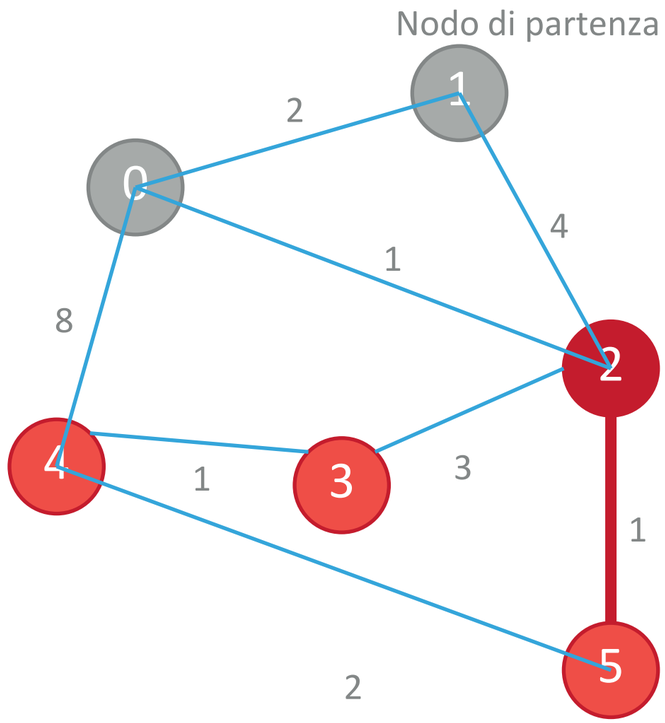
ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 2

Coda di priorità (min-heap) dei nodi non visitati

3 4 5

La coda non è un min-heap!



3+1<∞

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

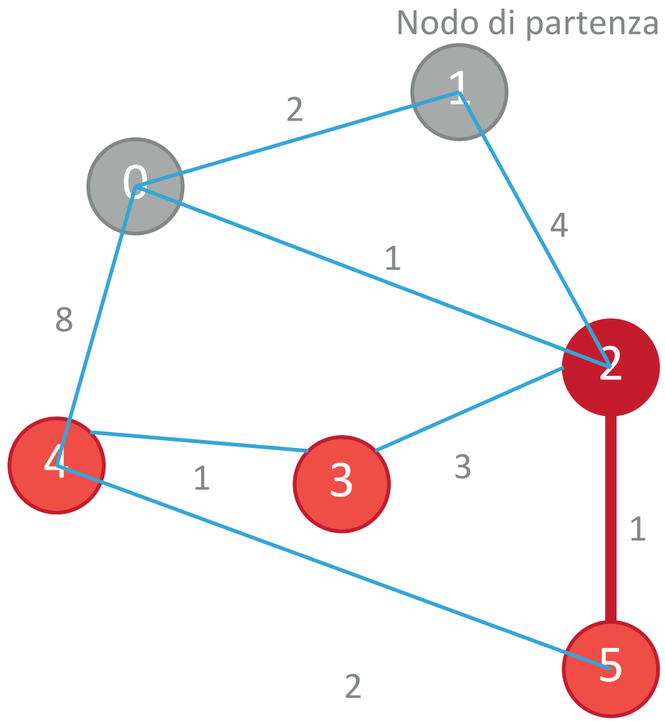
ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 2

Coda di priorità (min-heap) dei nodi non visitati

5 3 4

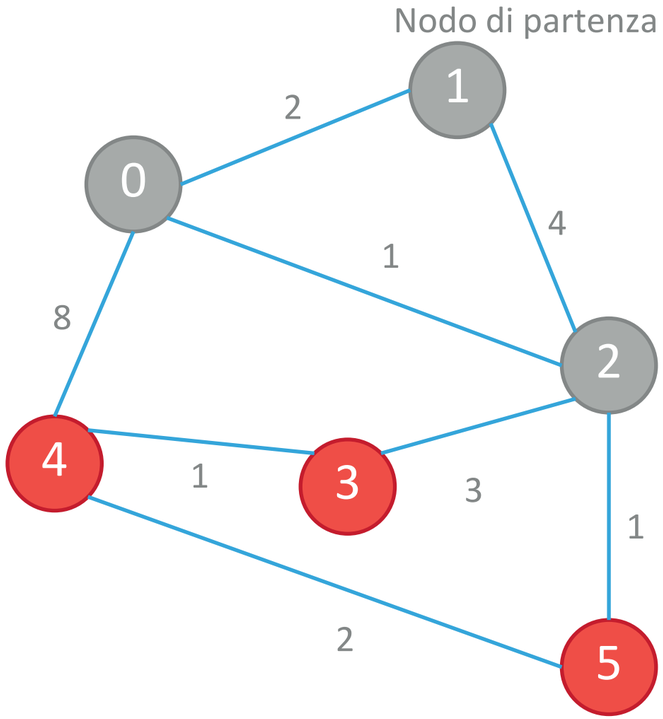
Adesso la coda è un min-heap!



3+1<∞

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

ESEMPIO DI ESECUZIONE



Coda di priorità (min-heap) dei nodi non visitati

5 3 4

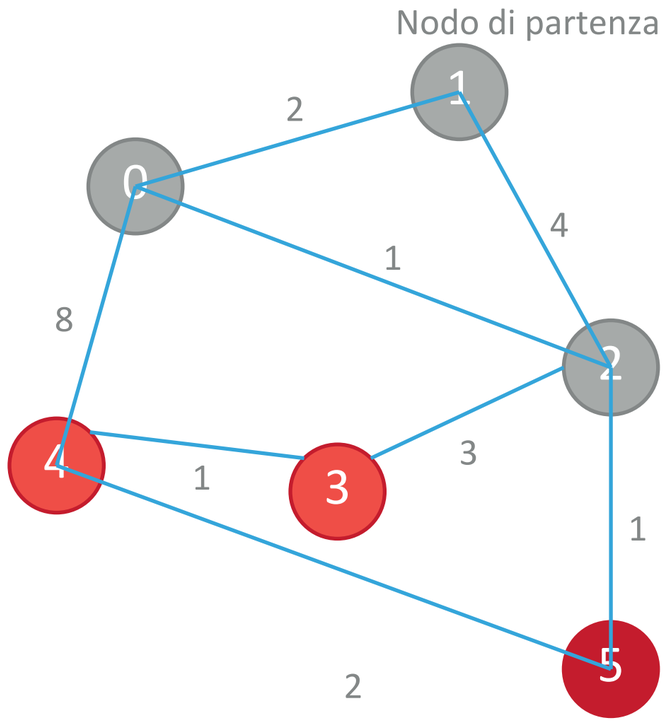
Vertice	Peso	Pred
0	2	1
1	0	-
2	3	0
3	6	2
4	10	0
5	4	2

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 5

Coda di priorità (min-heap) dei nodi non visitati

3 4



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

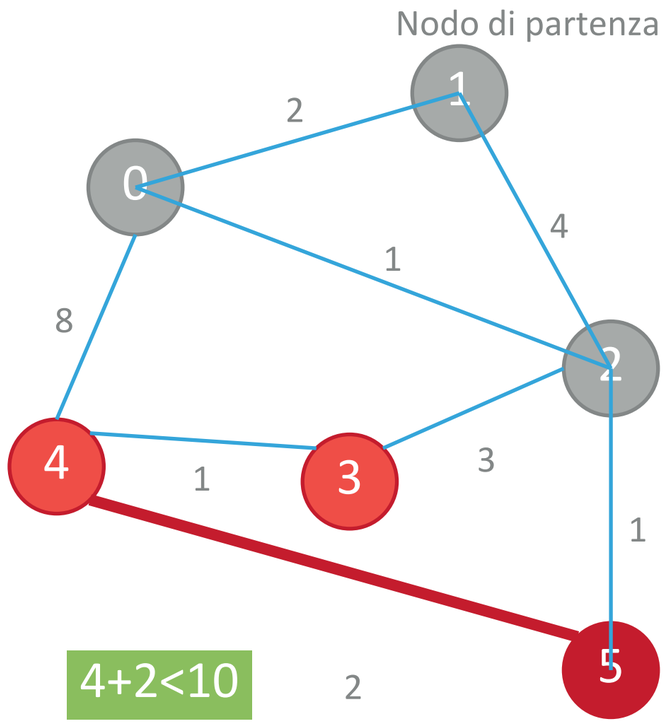
ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 5

Coda di priorità (min-heap) dei nodi non visitati

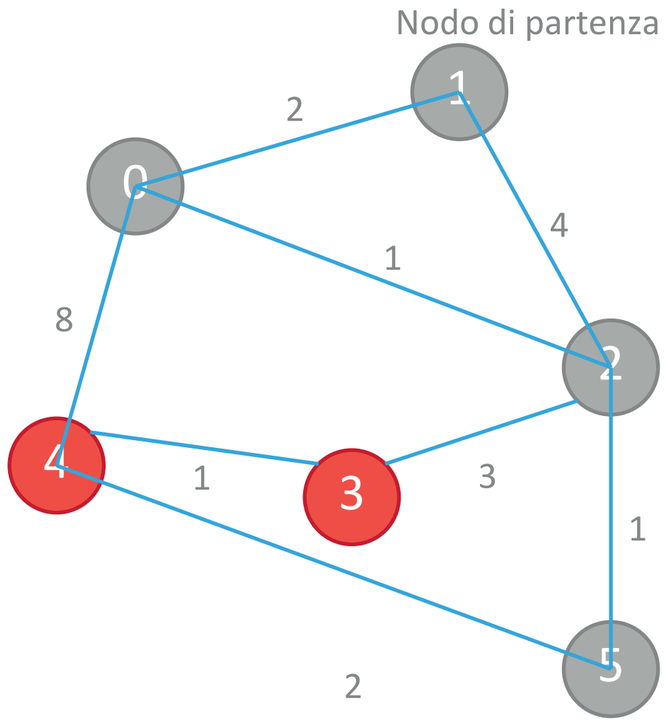
3 4

La coda è ancora un min-heap!



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

ESEMPIO DI ESECUZIONE



Coda di priorità (min-heap) dei nodi non visitati

3 4

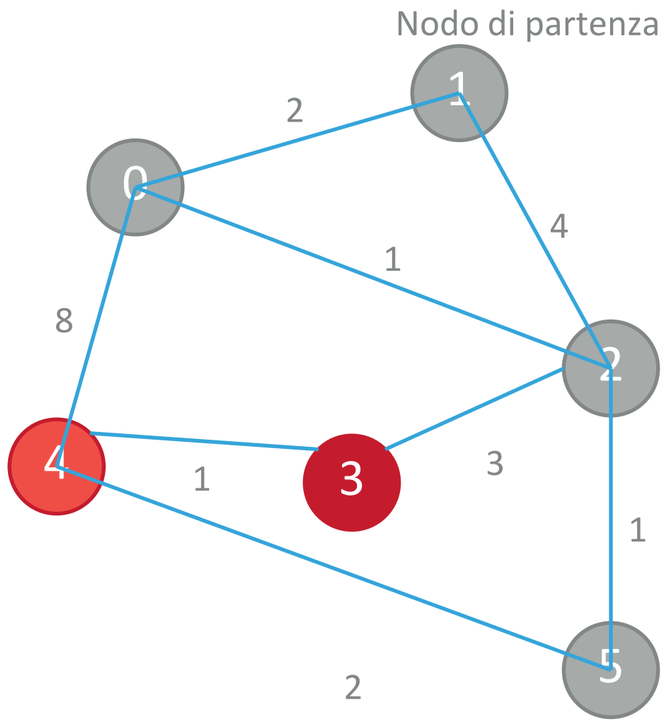
Vertice	Peso	Pred
0	2	1
1	0	-
2	3	0
3	6	2
4	6	5
5	4	2

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 3

Coda di priorità (min-heap) dei nodi non visitati

4



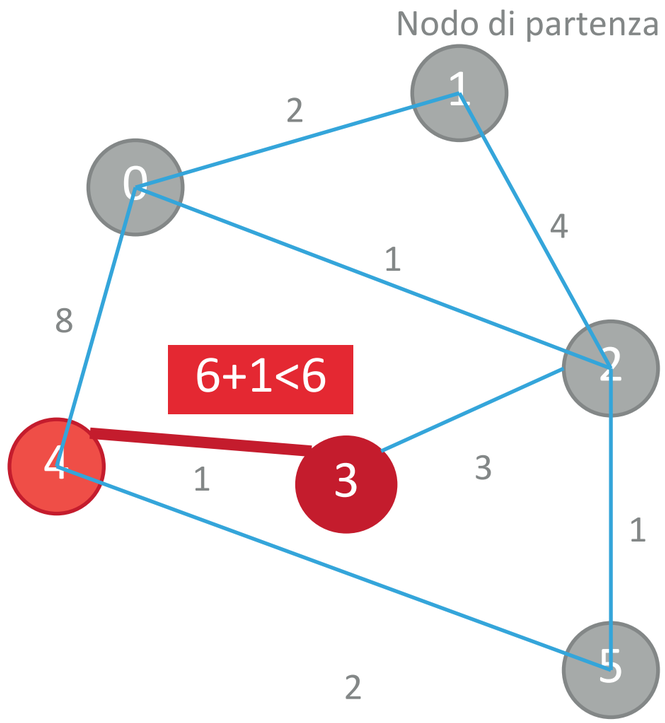
Vertice	Peso	Pred
0	2	1
1	0	-
2	3	0
3	6	2
4	6	5
5	4	2

ESEMPIO DI ESECUZIONE

Nodo con distanza minima: 3

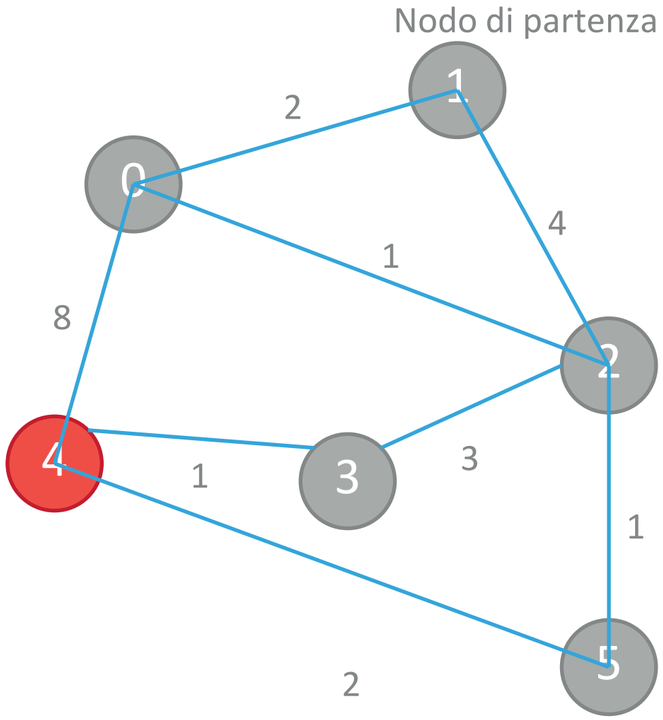
Coda di priorità (min-heap) dei nodi non visitati

4



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

ESEMPIO DI ESECUZIONE



Coda di priorità (min-heap) dei nodi non visitati

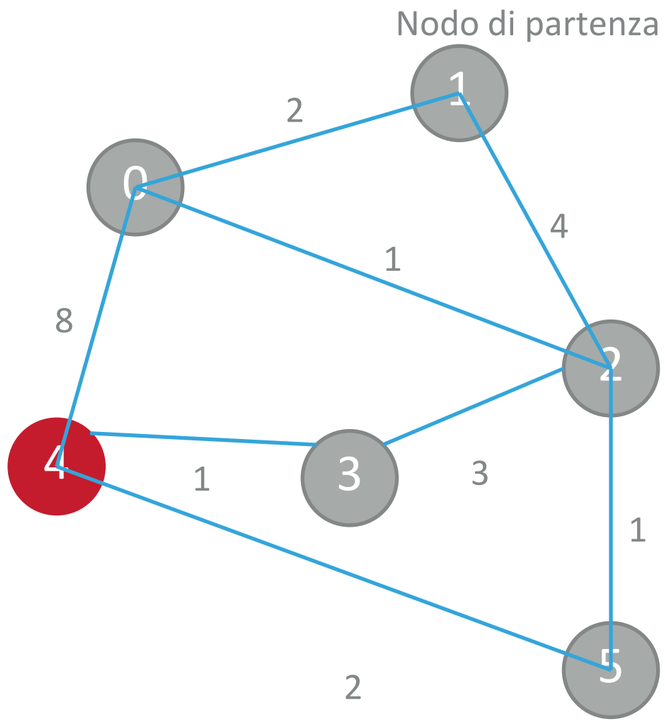
4

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

ESEMPIO DI ESECUZIONE

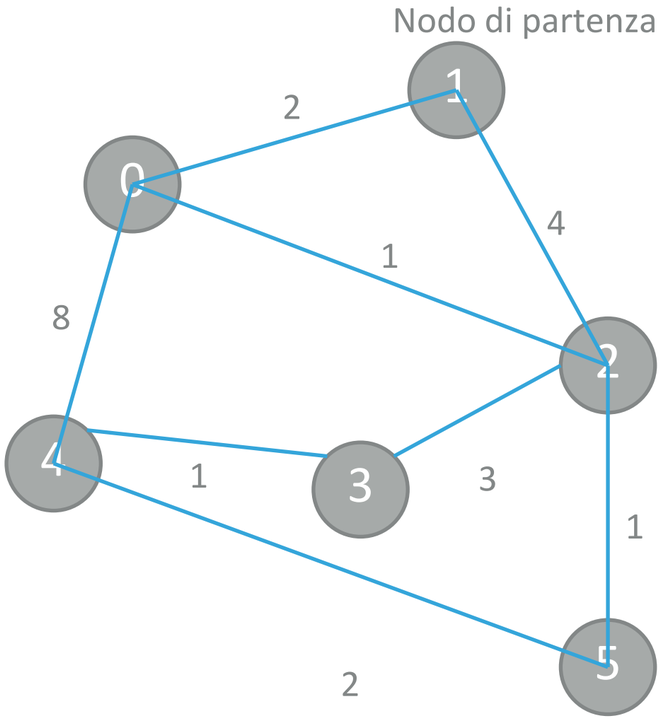
Nodo con distanza minima: 4

Coda di priorità (min-heap) dei nodi non visitati



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

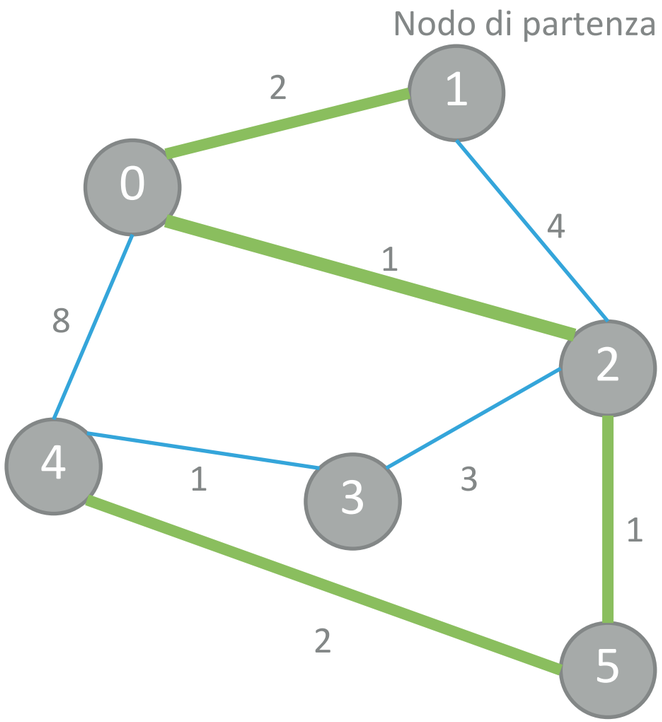
ESEMPIO DI ESECUZIONE



Abbiamo finito!

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

ESEMPIO DI ESECUZIONE



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

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

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 HEAP)

Ogni nodo viene estratto al massimo 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: $\boldsymbol{O(VlogV + ElogV) = O((V + E)logV)}$

- **Se avessimo usato 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)

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