Grafi pesati e l'algoritmo di Dijkstra

Algoritmi 2 – Lezione 8

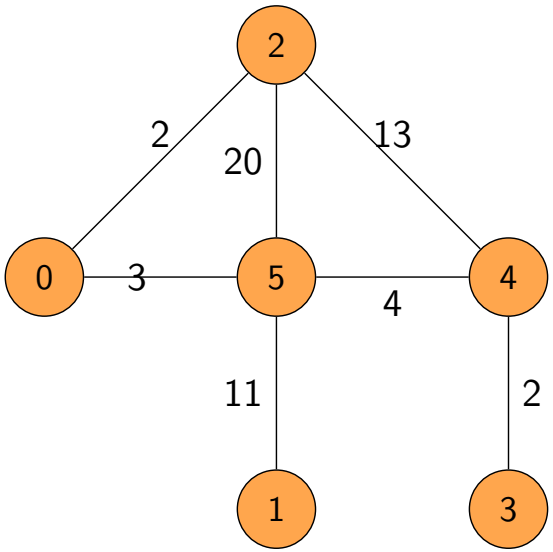
A. Monti
Sapienza Università di Roma

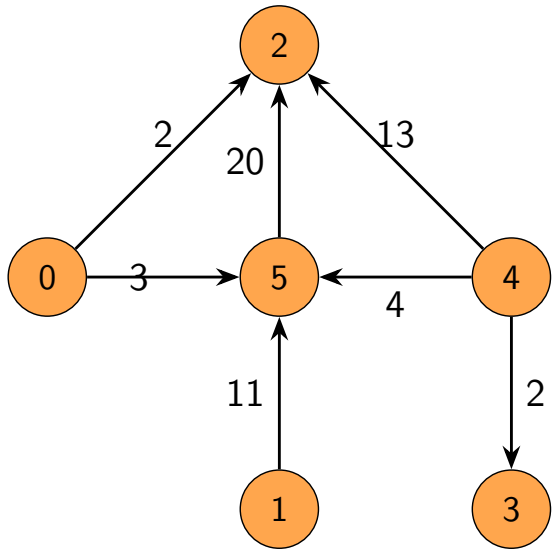
Corso: Algoritmi 2

Indice

Grafi pesati: la Definizione

- I grafi pesati sono una struttura fondamentale per modellare relazioni dove ogni arco ha un valore numerico associato, come un costo, una distanza o un tempo.

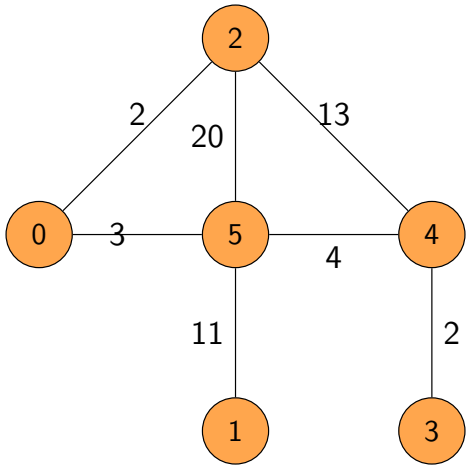


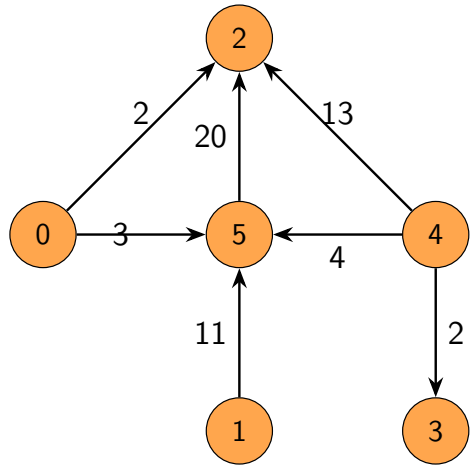


- Esempi includono la ricerca del percorso più breve su una mappa o l'instradamento di pacchetti in una rete.

Rappresentazione in Python

La rappresentazione tramite liste di adiacenza memorizza per ogni nodo una coppia (v, p) dove v è il nodo adiacente e p è il peso dell'arco che lo collega.






```
G = [
    [(2,2), (5,3)],
    [(5,11)],
    [(0,2), (5,20), (4,13)],
    [(4,2)],
    [(2,13), (3,2), (5,4)],
    [(0,3), (2,20), (4,4), (1,11)]
  ]
```

```
G = [
    [(2,2), (5,3)],
    [(5,11)],
    [],
    [],
    [(2,13), (5,4)],
    [ (2,20)]
  ]
```

Il problema dei cammini minimi

- In un grafo pesato, il **costo di un cammino** è la somma dei pesi dei suoi archi.
- Il nostro obiettivo è trovare il cammino con il **minore costo totale**.
- Esistono diverse varianti del problema:
  - **Cammino minimo da sorgente singola:** trovare i cammini minimi da un nodo s a tutti gli altri nodi.
  - **Cammini minimi tra ogni coppia di nodi:** trovare i cammini minimi tra ogni possibile coppia di nodi.

L'algoritmo di Dijkstra, descrizione generale

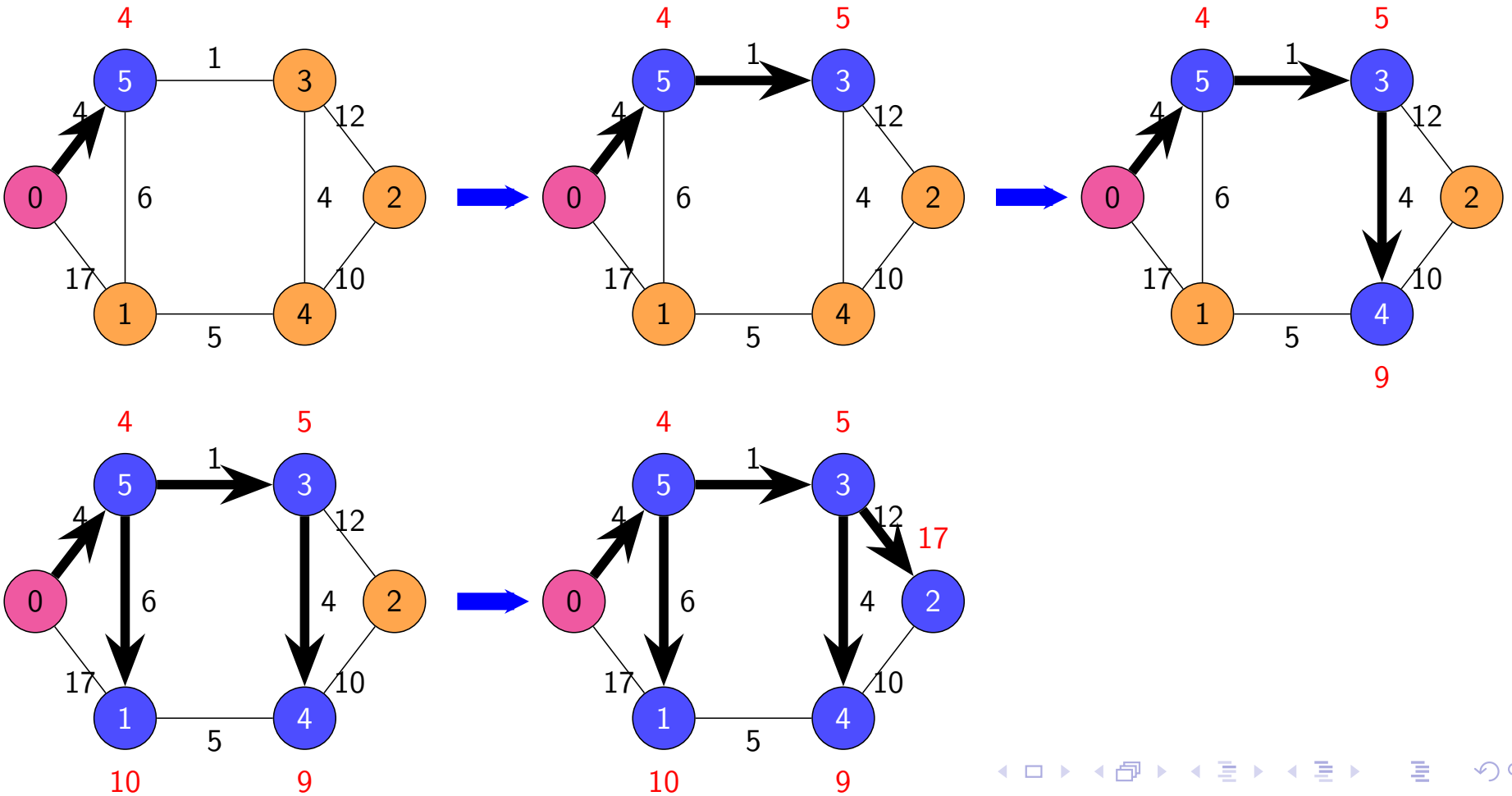
- L'algoritmo risolve il problema del cammino minimo da sorgente singola in grafi con **pesi non-negativi**.
- Adotta una strategia "greedy", costruendo progressivamente un albero dei cammini minimi, aggiungendo un arco alla volta.
- A ogni passo, seleziona il nodo non ancora visitato con la distanza provvisoria minima dalla sorgente, rendendola definitiva.
- Due i principi fondamentali:
  - **Decisioni irrevocabili:** una volta stabilita, la distanza di un nodo non viene più modificata.
  - **Criterio locale:** si sceglie localmente l'opzione migliore, che conduce alla soluzione ottima globale.

Fasi operative dell'algoritmo

1. **Inizializzazione:** Si imposta $D[s] = 0$ e $D[v] = +\infty$ per ogni altro nodo v.
2. **Espansione:** Si seleziona il nodo con distanza provvisoria minima e si esplorano i suoi vicini.
3. **Aggiornamento:** Per ogni vicino v, si aggiorna la distanza $D[v]$ se si trova un cammino più breve.
4. **Terminazione:** L'algoritmo termina quando tutti i nodi sono stati visitati o non restano nodi raggiungibili.

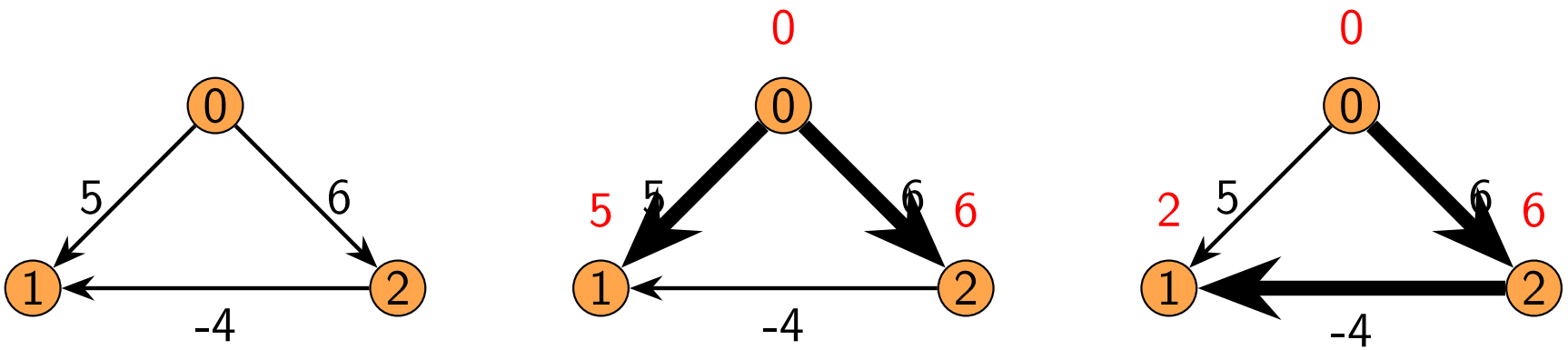
Esempio illustrato di esecuzione

La figura mostra l'esecuzione dell'algoritmo di Dijkstra in cinque passaggi su un grafo con sei nodi. In **grassetto** gli archi inclusi nell'albero dei cammini minimi; in rosso le distanze provvisorie assegnate ai nodi.



Limiti: grafi con pesi negativi

- L'algoritmo di Dijkstra può produrre risultati errati in presenza di **pesi negativi**, come mostrato nell'esempio seguente.



A sinistra, il grafo originale G; al centro, l'albero prodotto da Dijkstra; a destra, il vero albero dei cammini minimi. I risultati non coincidono.

Correttezza dell'algoritmo (1/2)

Teorema: L'algoritmo di Dijkstra restituisce la distanza minima da s a ogni nodo raggiungibile, se tutti i pesi sono non negativi.

Dimostrazione (per induzione):

- Sia S_i l'insieme dei nodi per cui Dijkstra ha già fissato la distanza minima dopo i iterazioni.
- **Caso base:** inizialmente $S_0 = \{s\}$, con $d(s) = 0$ che è la distanza corretta.
- **Passo induttivo:** supponiamo che per ogni nodo $u \in S_i$ la distanza $d(u)$ sia corretta.

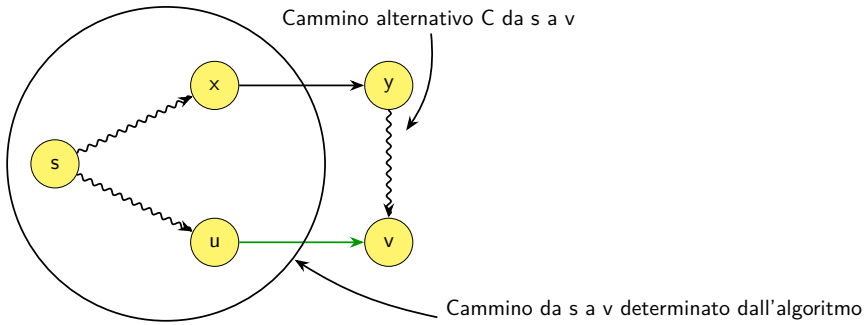
Dijkstra seleziona il nodo $v \notin S_i$ con distanza provvisoria minima:

$$d(v) = \min_{u \in S_i} \{d(u) + p(u, v)\}$$

Supponiamo per assurdo che esista un cammino più breve da s a v che passa per un nodo $y \notin S_i$ prima di raggiungere v.

Sia C tale cammino e (x, y) il primo arco che attraversa da S_i a fuori da S_i. Allora $x \in S_i$, $y \notin S_i$.

Correttezza dell'algoritmo (2/2)



Osservazione: Poiché tutti i pesi sono non negativi, ogni prefisso del cammino C ha costo almeno quanto la somma dei pesi degli archi percorsi.

In particolare: $\text{costo}(C) \geq d(x) + p(x, y)$.
Ma Dijkstra ha scelto v proprio perché:

$$d(v) = \min_{u \in S_i} \{d(u) + p(u, v)\} \leq d(x) + p(x, y)$$

Quindi: $\text{costo}(C) \geq d(v)$

Conclusione: nessun cammino alternativo può essere più breve. Quindi $d(v)$ è la distanza minima da s a v.

Per induzione: al termine dell'algoritmo, tutte le distanze sono corrette.

Idea dell'implementazione

- L'algoritmo utilizza una struttura lineare (**array/lista**) per gestire i nodi.
- Manteniamo tre strutture:
  - **D**: distanza definitiva dalla sorgente
  - **P**: predecessore nel cammino minimo
  - **Lista**: stato dei nodi
- In **Lista**, per ogni nodo v:
  - se v è **raggiunto ma non ancora fissato**, memorizziamo (predecessore, distanza provvisoria)
  - se v è **non ancora raggiunto** oppure **già fissato**, il valore è **None**
- Un nodo è:
  - **candidato** (raggiunto ma non fissato) se `Lista[v]` $\neq$ `None`
  - **fissato** se `Lista[v] = None` e la distanza è stata salvata in **D**

Passi dell'algoritmo

- Inizializzazione:
  - $D[s] = 0$
  - i vicini di s vengono inseriti in Lista con distanza provvisoria
- Ad ogni iterazione:
  1. si seleziona tra i nodi con `Lista[v]` $\neq$ `None` quello con distanza minima
  2. il nodo viene **fissato**: si aggiornano D e P e si pone `Lista[v] = None`
  3. si aggiornano le distanze dei vicini non fissati (**rilassamento**)
- L'algoritmo termina quando non esistono più nodi con `Lista[v]` $\neq$ `None`.

Dijkstra con Array: implementazione in Python

```
def dijkstra_array(s, G):
    '''
    Restituisce il vettore delle distanze D ed il vettore dei padri P che si ottengono
    applicando l'algoritmo di Dijkstra al grafo G con sorgente s
    '''
    n = len(G)
    # lista in cui inserire (predecessore, distanza)
    Lista = [ None for _ in range(n)]
    P = [ - 1 for _ in range(n)]
    D = [ float('inf') for _ in range(n)]
    P[s], D[s], Lista[s] = s, 0, None
    # Inizializza le distanze dei vicini di s
    for y, costo in G[s]:
        Lista[y] = (s, costo)
    while True:
        nodo, costo_minimo = -1, float('inf')
        # Ricerca lineare del nodo  con distanza minima
        for i, valore in enumerate(Lista):
            if valore is not None and valore[1] < costo_minimo:
                nodo, costo_minimo = i, valore[1]
        if nodo == -1:
            # non ci sono più nodi raggiungibili e l'algoritmo termina
            break
        # Fissa nodo
        pred, dist = Lista[nodo]
        P[nodo], D[nodo], Lista[nodo]  = pred, dist, None
        # Rilassamento degli archi
        for y, costo in G[nodo]:
            nuovo_costo = D[nodo] + costo
            if P[y] == -1:
                # il nodo y non è stato ancora fissato
                if Lista[y] is None or nuovo_costo < Lista[y][1]:
                    Lista[y] = (nodo, nuovo_costo)
    return D, P
```

Complessità

- La ricerca del nodo con distanza minima da fissare avviene tramite **scansione lineare** di Lista:
$$O(n)$$
- Questa operazione viene ripetuta al più n volte:
$$O(n^2)$$
- Il rilassamento degli archi richiede:
$$O(m)$$
- Complessità totale:
$$O(n^2 + m) = O(n^2)$$
- Questa implementazione è adatta soprattutto per grafi densi dove
$$m \approx n^2.$$

Possiamo fare meglio?

Implementazione con array

Ricerca del minimo tra i nodi non fissati

$$O(n)$$

Ripetuta per n iterazioni

$$O(n^2)$$

Possiamo evitare la scansione lineare dei nodi?

Idea: usare un *heap minimo*

Dijkstra con Min-Heap

Nel min-heap memorizziamo coppie

$$(d, v)$$

dove

- v è un vertice
- d è la stima corrente della distanza dalla sorgente

L'heap permette di estrarre rapidamente il vertice con distanza minima.

Dijkstra con Min-Heap: implementazione in Python

```
import heapq

def dijkstra_heap(G, s):
    '''
    Restituisce il vettore delle distanze D
    ed il vettore dei padri P  che si ottengono
    applicando l'algoritmo di Dijkstra al grafo
    G con sorgente s
    '''
    n = len(G)
    D = [float('inf')] * n
    D[s] = 0
    P = [-1] * n
    heap = [(0, s)]
    while heap:
        costo, u = heapq.heappop(heap)
        if costo > D[u]:
            continue
        for v, peso in G[u]:
            if D[u] + peso < D[v]:
                D[v] = D[u] + peso
                P[v] = u
                heapq.heappush(heap, (D[v], v))
    return D, P
```

Un dettaglio importante dell'implementazione

Quando *rilassiamo* un arco (u, v):

$$\text{se } D[u] + p(u, v) < D[v]$$

aggiorniamo

$$D[v] = D[u] + p(u, v)$$

e inseriamo nell'heap

$$(D[v], v)$$

Non rimuoviamo la vecchia copia di v dall'heap.

Questo significa che nell'heap possono esistere **più copie dello stesso vertice** con distanze diverse.

Gestione delle copie obsolete

Quando estraiamo una coppia (d, v) dall'heap:

- se $d = D[v]$ la distanza è corretta e processiamo v
- se $d > D[v]$ la coppia è **obsoleta** e la ignoriamo

Le copie obsolete nascono perché durante l'esecuzione possiamo aver trovato una distanza migliore per lo stesso vertice.

Questo non compromette la correttezza dell'algoritmo: semplicemente alcune estrazioni non producono lavoro utile.

Analisi del tempo di esecuzione

Operazioni sull'heap

- inseriamo nell'heap un elemento ogni volta che una distanza migliora
- ogni arco può causare al più un miglioramento

quindi eseguiamo al più $O(m)$ inserimenti.

Inoltre eseguiamo al più $O(n)$ estrazioni utili.

Il costo di ciascuna di queste operazioni è

$$O(\log m) = O(\log n) \qquad \text{(poiché } m \leq n^2\text{)}.$$

La complessità totale è quindi

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

Quando conviene usare l'heap

La complessità dell'algoritmo è

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

Grafi sparsi ($m \approx n$)

$$O(n \log n)$$

molto migliore della versione con array ($O(n^2)$).

Grafi densi ($m \approx n^2$)

$$O(n^2 \log n)$$

che può essere peggiore della versione con array che ora vedremo.

ESERCIZI

Suggerimento: Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

Esercizi

1. Si considerino tre contenitori di capacità a, b e c litri, con $1 \leq a < b < c \leq 20$. Inizialmente, i contenitori da a e b litri sono pieni d'acqua, mentre quello da c litri è vuoto.
   L'unica operazione consentita consiste nel versare acqua da un contenitore ad un altro, interrompendo il trasferimento quando il contenitore di partenza diventa vuoto o quello di destinazione si riempie.
   Si richiede di sviluppare i seguenti algoritmi e di valutarne la complessità:
   1. **Sequenza di versamenti per ottenere d litri**
      Dati a, b, c e un valore d con $d \leq a$, determinare una sequenza di versamenti che termini lasciando esattamente d litri nel primo o nel secondo contenitore. Se una tale sequenza non esiste, l'algoritmo deve restituire `None`.
   2. **Conteggio delle sequenze valide**
      Dati a, b, c e un valore d con $d \leq a$, restituire il numero di sequenze di versamenti che terminano lasciando esattamente d litri nel primo o nel secondo contenitore.
   3. **Numero minimo di versamenti**
      Dati a, b, c e un valore d con $d \leq a$, determinare il numero minimo di versamenti necessari per ottenere esattamente d litri nel primo o nel secondo contenitore. Se non è possibile ottenere tale quantità, l'algoritmo deve restituire `None`.
   4. **Stato raggiungibile con la sequenza più lunga**
      Dati a, b e c, determinare la terna (a', b', c') che può essere ottenuta a partire dallo stato iniziale ma che richiede il numero massimo di versamenti. Se esistono più terne che richiedono il numero massimo di versamenti va restituita quella che precede lessicograficamente.
   5. **Costo minimo di una sequenza di versamenti**
      Il costo di una sequenza di versamenti è definito come il numero totale di litri trasferiti nei vari passaggi. Dati a, b, c e un valore d con $d \leq a$, determinare il costo minimo di una sequenza di versamenti che porti a lasciare esattamente d litri nel primo o nel secondo contenitore. Se tale sequenza non esiste, l'algoritmo deve restituire `None`.

Esercizi

2 Abbiamo un grafo connesso G di n nodi, i nodi sono di due tipi: "pericolosi" o meno. Un vettore binario P di n componenti permette di individuare i nodi pericolosi ($P[i] = 1$ se e solo se i è pericoloso).

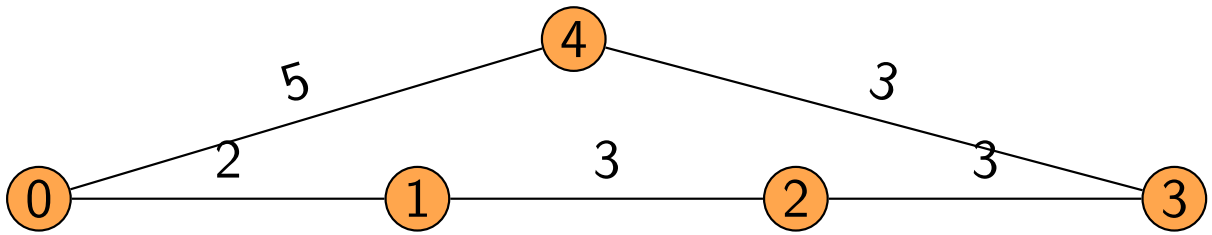
- Progettare un algoritmo che, dato G, un suo nodo s ed il vettore P, restituisce in tempo $O(n + m)$ una lista con tutti i nodi irraggiungibili da i senza dover passare per nodi pericolosi.
- Progettare un algoritmo che, dato G, un suo nodo s ed il vettore P, genera in tempo $O(n + m)$ un albero dei cammini minimi radicato in s dove il costo di un cammino è dato dal numero di nodi pericolosi toccati lungo il cammino.

Esercizi

3 Dato un grafo pesato G, un cammino in G da un nodo u ad un nodo v si dice **super-minimo** se ha peso minimo tra tutti i cammini da u a v in G e inoltre tra tutti i cammini di peso minimo da u a v ha il minimo numero di archi.

Ad esempio nel grafo G in figura tra i cammini che vanno da 0 a 3:

- Il cammino $0 \to 1 \to 2 \to 3$ è minimo (ma non super-minimo),
- Il cammino $0 \to 4 \to 3$ è super-minimo.



Sviluppare un algoritmo che, dato il grafo pesato G, con pesi interi positivi, ed un nodo s, trovi i cammini super-minimi che partono da s.

Suggerimento: Mostrare come modificare i pesi del grafo G in modo tale che applicando l'algoritmo di Dijkstra al grafo coi nuovi pesi e al nodo s si ottenga l'albero dei cammini super-minimi di G.