GRAFI: CAMMINI MINIMI E ALGORITMO DI DIJKSTRA

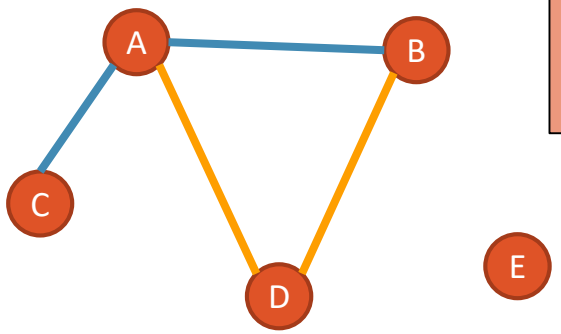
[Deme, seconda edizione] cap. 14

Fino a Sezione 14.1.1 esclusa

Sezione 14.5

(dall'introduzione)
Cammino minimo o distanza

Il cammino minimo tra due nodi v e w in cui w è raggiungibile da v si dice **distanza**. Se w non è raggiungibile da v si dice che la loro distanza è infinita.



$\langle C, A, D, B \rangle$ ha lunghezza = 3
$\langle C, A, B \rangle$ ha lunghezza 2 e non esistono cammini più corti tra C e B, quindi la distanza tra di loro è 2.
La distanza tra C ed E è $+\infty$.

(dalla visita in ampiezza)
Teorema 3 (distanza nell'albero BFS)

Al termine dell'esecuzione di BFS-VISITA si ha d[v] = δ(s,v) per tutti i vertici v ∈ V.

Dimostrazione.

Se v non è raggiungibile da s allora d[v] rimane ∞ = δ(s,v).

Altrimenti v è nero e il teorema vale per il lemma precedente.

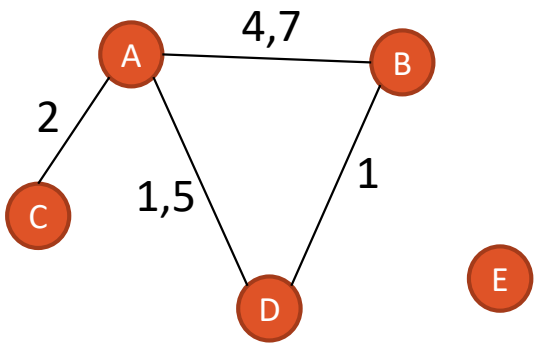
- per ogni vertice v raggiungibile da s, il cammino da s a v sull'albero ottenuto con la visita è un cammino minimo.
- Il livello di un vertice nell'albero è indipendente dall'ordine in cui sono memorizzati i vertici nelle liste di adiacenza.

(dall'introduzione ai grafi) Grafo pesato

Definiamo una funzione peso $\boldsymbol{W: E \to \mathbb{R}}$ (dove $\mathbb{R}$ è l'insieme dei numeri reali).

Per ogni arco $(\mathrm{v}, \mathrm{w}) \in E$, W definisce il **peso** di (v, w).

La coppia (G, W) è definita **grafo pesato**.



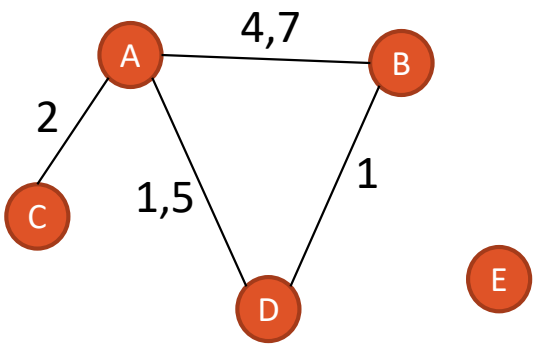
$$W(C, A) = 2$$
$$W(A, B) = 4{,}7$$
$$W(A, D) = 1{,}5$$
$$W(D, B) = 1$$

Definizioni (ovvie)

peso o costo di un cammino da un nodo s a un nodo v (in un grafo pesato): è la **somma** dei **pesi** o costi degli archi che compongono il cammino

cammino minimo da un nodo s a un nodo v (in un grafo pesato): un cammino da s a v si dice minimo se è un **cammino di peso minimo fra tutti i cammini da s a v**; naturalmente possono esistere cammini minimi distinti da s a v, aventi lo stesso peso

distanza di v da s, o da s a v, indicata δ(s, v) (in un grafo pesato): è il **peso** di un **cammino** di peso **minimo** da s a v



Qual è ora la distanza tra C e B?
Qual è il cammino minimo?
Non possiamo più contare gli archi, dobbiamo sommare i loro pesi.

$\langle C, A, D, B \rangle$ ha peso = 4,5
$\langle C, A, B \rangle$ ha peso = 6,7.
La distanza tra C e B è quindi 4,5 ed il cammino minimo è $\langle C, A, D, B \rangle$.

Il problema della distanza

In molte applicazioni, abbiamo bisogno di trovare la **distanza** tra **due nodi** in un **grafo pesato**. Ad esempio:

- Trovare **l'itinerario più corto, o più veloce**, da un luogo ad un altro su una **mappa stradale**.
- Trovare il **percorso di durata minima** fra due stazioni di una rete ferroviaria o di una rete di **trasporto pubblico urbano**.
- Protocollo di routing OSPF (Open Shortest Path First) usato in internet per trovare la **migliore connessione** da ogni nodo della rete a tutte le **possibili destinazioni**.

Il teorema della distanza nell'albero BFS visto per i grafi non pesati **non vale** per un grafo pesato, quindi **$d[v] \neq \delta(s,v)$** se consideriamo un grafo pesato e la nozione di lunghezza di un cammino data per i grafi pesati.

Varianti del Problema

- Cammino minimo per **una coppia di vertici** (s,v): trovare un cammino di lunghezza δ(s,v) tra s e v.

(OUTPUT: una **sequenza ordinata** di archi/vertici)

- Cammini minimi **a sorgente unica** s: trovare un cammino minimo da un vertice sorgente s ad ogni altro vertice $v \in V$

(OUTPUT: un insieme di cammini…vedremo che è rappresentabile con un **albero**)

- Cammini minimi **a destinazione unica** t: trovare un cammino minimo da ogni vertice $u \in V$ a t.

(OUTPUT: un insieme di cammini)

- Cammini minimi tra **tutte le coppie di vertici**: trovare un cammino minimo per ogni coppia di vertici $(u, v) | \, u, v \in V$.

(OUTPUT: un insieme di cammini)

Un approccio ingenuo per grafi pesati: esaminare tutti i cammini

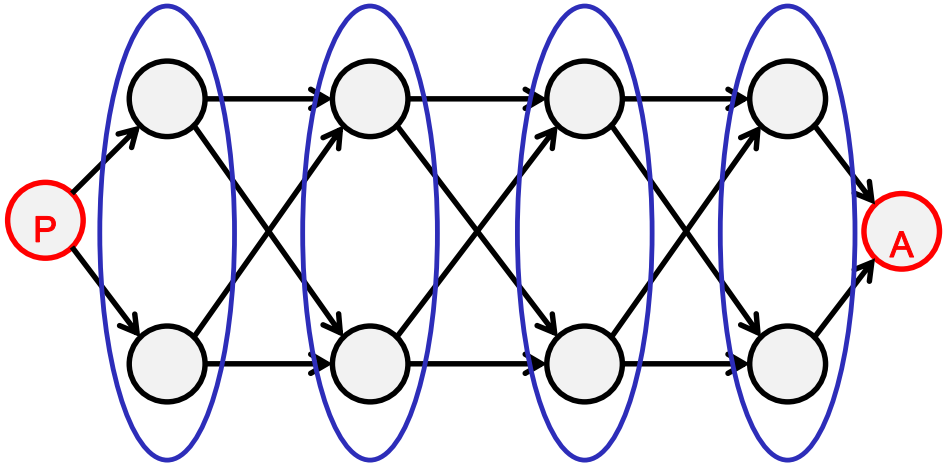
La definizione definisce che cosa è un cammino minimo, non **come trovarlo**.

Come molte definizioni, essa specifica implicitamente un metodo **ingenuo** per trovare ciò che viene definito:

Si esaminano tutti i cammini fra i due nodi calcolandone le rispettive lunghezze, e di essi si prende il minimo (cioè quello di lunghezza minima).

Tale metodo, perfettamente corretto, ha tuttavia una complessità che, per grafi generici, è **esponenziale** (perché tale può essere il numero dei cammini, diversamente dal numero degli archi che è limitato da n^2).

Approccio ingenuo, esempio



Quanti sono i possibili percorsi **da P ad A**?

da P ci sono **2 archi uscenti**, che arrivano rispettivamente a due nodi;

da ognuno dei 2 nodi così raggiunti, ci sono 2 archi uscenti, quindi in tutto $\mathbf{2 \times 2 = 2^2 = 4}$ percorsi per arrivare alla seconda coppia "verticale" di nodi;

per ognuno di tali 4 percorsi si può proseguire in 2 modi, quindi in tutto $\mathbf{4 \times 2 = 2^3 = 8}$ percorsi fino alla la terza coppia;

infine, per ognuno di essi si può proseguire in 2 modi, quindi in tutto ci sono $\mathbf{8 \times 2 = 2^4 = 16}$ percorsi distinti da P ad A.

Approccio ingenuo, fallimento

In generale, per n coppie di nodi con archi disposti come nella figura precedente, più un nodo di partenza e uno di arrivo, abbiamo **2n + 2** nodi ma **2^n percorsi distinti**.

Al crescere di n, il numero di nodi cresce linearmente, cioè si mantiene circa proporzionale a n, ma il numero di percorsi cresce **esponenzialmente**.

Per **n = 50**, si hanno **102 nodi**, ma più di **10^{15} percorsi distinti, cioè più di un milione di miliardi di percorsi distinti**.

Un computer che esamini un miliardo di cammini al secondo impiegherebbe, per trovare il percorso minimo, un milione di secondi, cioè più di **11 giorni**.

Per n = 100, si hanno 202 nodi, ma più di 10^{30} percorsi distinti, il computer ci metterebbe più di **30000 miliardi di anni**!

Una rete stradale è tipicamente costituita da migliaia di nodi…

Obiettivo

È abbastanza facile immaginare che l'approccio ingenuo non possa essere il meglio che possiamo fare.

Il nostro obiettivo è quindi quello di capire se è possibile formulare metodi che trovino i cammini minimi **senza analizzare tutti i possibili cammini**.

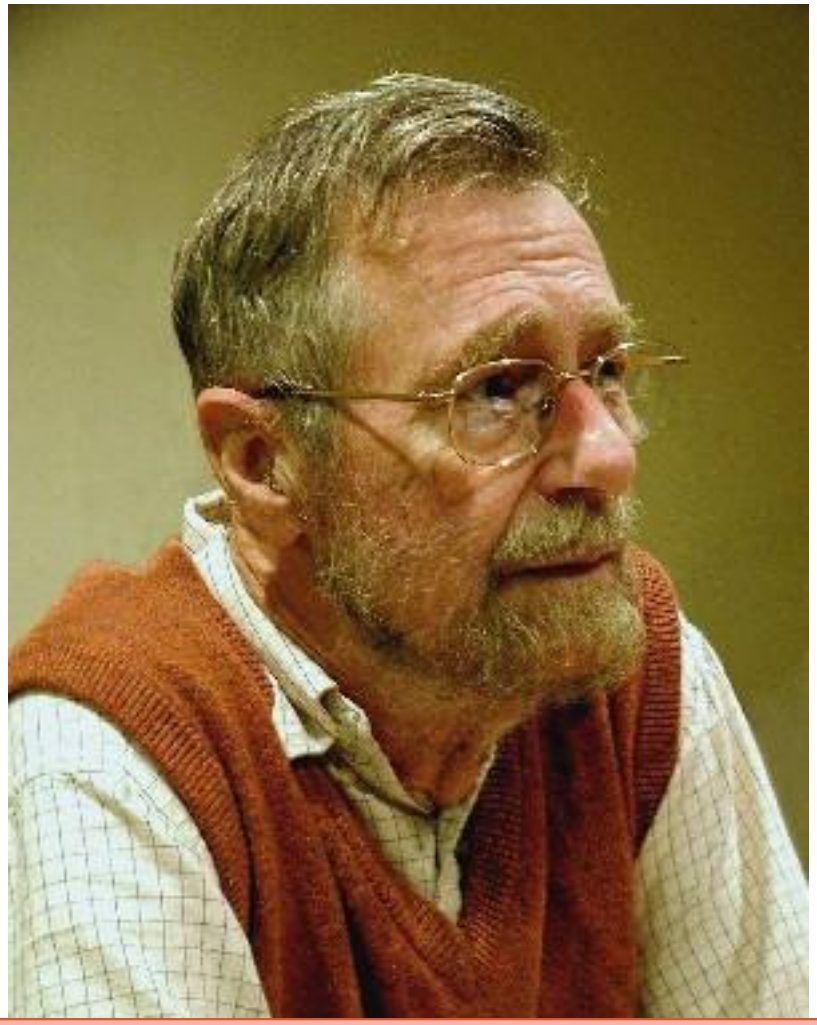
In realtà, ci ha già pensato Edsger Wybe **Dijkstra** nel 1956.

"Dijkstra pensò il suo algoritmo in una mattina di sole del 1956, senza carta né matita, seduto a bere il caffè con la moglie in una terrazza di Amsterdam. Per trovare **il percorso più breve tra due nodi**, studiò il problema più generale di **determinare simultaneamente i cammini minimi da un nodo preso come sorgente verso tutti gli altri**."

da "L'informatica invisibile", a cura di Ausiello e Petreschi, Mondadori Università, 2010, cap. 4, "La ricerca della via più breve", di Demetrescu e Italiano.

E. W. Dijkstra

(pronuncia “deikstra” con la “e” larga, suona quasi “daikstra”)



«L'informatica non riguarda i computer più di quanto l'astronomia riguardi i telescopi».

E. W. Dijkstra 1930-2002

Edsger Wybe Dijkstra was one of the most influential members of computing science's founding generation. Among the domains in which his scientific contributions are fundamental are

- algorithm design
- programming languages
- program design
- operating systems
- distributed processing
- formal specification and verification
- design of mathematical arguments

In addition, Dijkstra was intensely interested in teaching, and in the relationships between academic computing science and the software industry.

During his forty-plus years as a computing scientist, which included positions in both academia and industry, Dijkstra's contributions brought him many prizes and awards, including computing science's highest honor, the ACM Turing Award. (fonte **wikipedia**)

Proprietà 1 dei sottocammini (minimi) di un cammino minimo

PROPRIETÀ: Un **sottocammino** di un **cammino minimo è un cammino minimo**. (**SOTTOSTRUTTURA OTTIMA**)

DIMOSTRAZIONE:

Sia u ⇝ v un **sottocammino** di un **cammino minimo** da s a t:

s ⇝ **u** ⇝ **v** ⇝ t

Se u ⇝ v non fosse minimo, ci sarebbe **un altro cammino da u a v di costo inferiore**.

Ma allora sostituendo tale cammino nel cammino da s a t, si otterrebbe un **cammino da s a t di costo inferiore** rispetto al cammino minimo.

Esempio: se il percorso più breve fra Torino e Firenze comprende un tratto fra Alessandria e La Spezia, questo sarà evidentemente il percorso più breve fra Alessandria e La Spezia!

L’idea

Costruiamo **un albero (di visita)** che contenga tutti i vertici raggiungibili da un nodo di partenza s.

Partendo da un albero che contiene solo s, ad ogni iterazione della visita scegliamo un vertice u e lo aggiungiamo all’albero, aggiungendo anche un arco (del grafo) che lo collega all’albero.

Alla fine della visita, l’albero conterrà **tutti i vertici raggiungibili da s ed i cammini minimi tra s ed i vertici**.

Come scegliamo u e l’arco da aggiungere? Che tipo di visita adottiamo?

L’algoritmo di Dijkstra è una **visita** («BFS») in cui la struttura dati D è gestita come in un **algoritmo greedy con appetibilità modificabili**. Il vertice u viene scelto in base alla sua **appetibilità greedy** e ogni volta che aggiungo u all’albero **aggiorno l’appetibilità dei vertici v ad esso adiacenti**.

L’idea più nel concreto

L’appetibilità di un vertice u è data da una **stima della distanza tra s ed u** che l’algoritmo ha in un determinato istante (che indicheremo con d[u]).

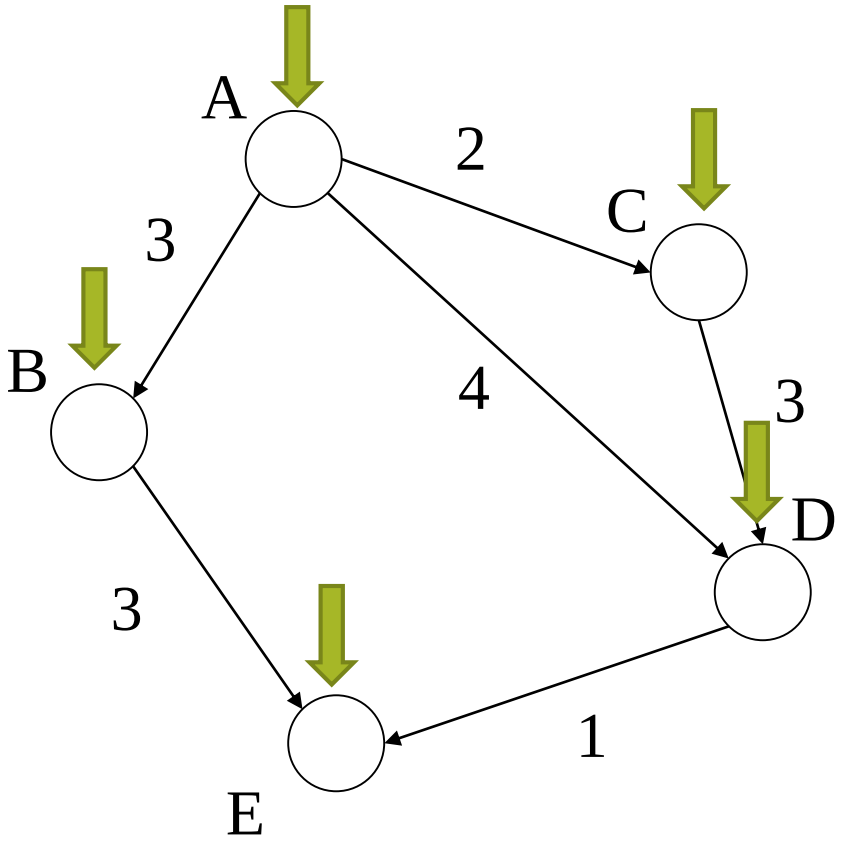
Inizialmente, tutti i vertici sono stimati a **distanza ∞** da s, tranne s stesso che (banalmente) ha distanza d[s] = 0 da se stesso.

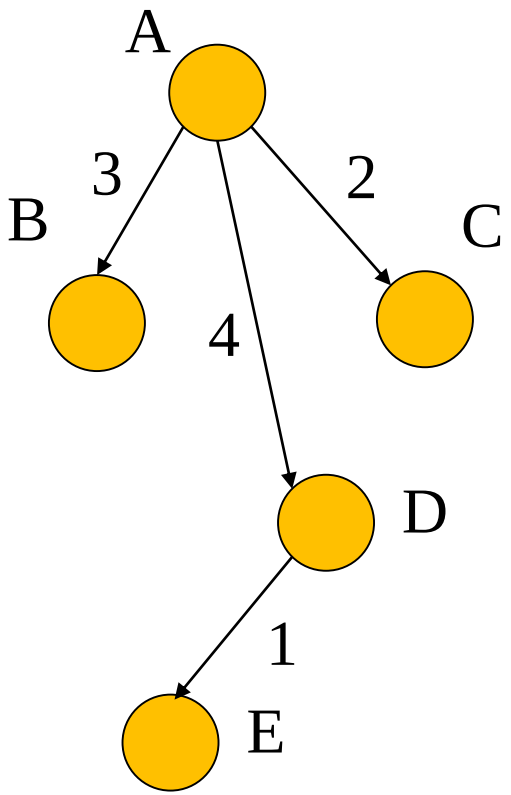
Ad ogni ciclo (di una visita), scelgo un vertice u da aggiungere all’albero, tra quelli non ancora inseriti ma raggiunti dalla ricerca (cioè grigi), e scelgo quello con **distanza da s stimata minima**.

Come aggiorno le distanze stimate/l’appetibilità dei vertici non neri?

Se il cammino tra s e v (un nodo adiacente ad u) che passa da u (il nodo appena aggiunto all’albero) è di lunghezza minore a quello finora stimato (cioè, se **d[u] + W(u,v) < d[v]**) ho trovato una nuova distanza stimata (ed un cammino) tra s e v, **migliore** della precedente.

Esempio





d

0	3	2	4	5
A	B	C	D	E

π

\	A	A	A	D
A	B	C	D	E

Algoritmo di Dijkstra, prima versione

***Dijkstra* (G, W, s)**
INIZIALIZZA (G)
color [s] <- gray
d[s] <- 0
while esistono vertici grigi **do begin**
u <- vertice grigio con d[u] minore
S <- S ∪ {u}//aggiungo u all'albero definitivamente
for ogni v adj ad u **then**
if color[v] ≠ black **then**
color [v] <- gray
if d[v] > d[u] + W(u,v) then
π[v] <- u
d[v] <- d[u] + W(u,v)
end for
color [u] <- black
end

***INIZIALIZZA* (G)**
…
S = ∅
for ogni vertice u ∈ V[G] **do**
color [u] <- white
π[u] <- NULL
d[u] <- +∞

Gestione dei nodi grigi

Non abbiamo considerato la struttura dati D per gestire i nodi grigi.

Abbiamo bisogno di una struttura che ci permetta di inserire i nodi, ed **estrarli secondo la loro «appetibilità»** d[u].

Conosciamo qualcosa del genere?

Sì, una **coda con priorità**.

Per questo l'algoritmo Dijkstra è una specie di visita BFS.

Algoritmo di Dijkstra, con priority queue

```
Dijkstra (G, W, s)
   INIZIALIZZA (G)
   D <- empty_priority_queue()
   color [s] <- gray
   d[s] <- 0
   enqueue(D,s,d[s])
   while NotEmpty(D) do begin
      u <- dequeue_min(D)
      S <- S ∪ {u}//aggiungo u all'albero definitivamente
      for ogni v adj ad u then
         if color[v] ≠ black then
            if color[v] = white then
               color [v] <- gray
               enqueue(D,v,d[v])
            if d[v] > d[u] + W(u,v) then
               π[v] <- u
               d[v] <- d[u] + W(u,v)
               decrease_key(D,v,d[v])
      end for
      color [u] <- black
   end
```

Ulteriori miglioramenti

Notiamo che è possibile **non distinguere nodi bianchi da grigi**, ed **inserire tutti nella coda fin dall'inizio**, assegnando loro distanza **infinita** da s.

Chiamiamo **definitivi** i nodi neri, e **non definitivi** i nodi grigi e bianchi.

Gestiamo definitivi/non definitivi con un vettore di booleani **def**.

Algoritmo di Dijkstra - II

```
Dijkstra (G, W, s)
   INIZIALIZZA (G)
   D <- empty_priority_queue()
   d[s] <- 0
   for ogni v in V[G]
      enqueue(D,v,d[v])
      def[v] <- false
   while NotEmpty(D) do begin
      u <- dequeue_min(D)
      S <- S ∪ {u} //aggiungo u all'albero definitivamente
      def[u] <- true
       for ogni v adj ad u then
         if def[v] = false and d[v] > d[u] + W(u,v) then
               π[v] <- u
               d[v] <- d[u] + W(u,v)
               decrease_key(D,v,d[v])
      end for
   end
```

Ulteriori miglioramenti

Notiamo che è possibile **non distinguere nodi bianchi da grigi**, ed **inserire tutti nella coda fin dall'inizio**, assegnando loro distanza **infinita** da s.

Inoltre, **non** è nemmeno **necessario distinguere** i **nodi neri**, poiché saranno quelli che non sono in coda, inoltre non sarà mai possibile dover aggiornare d[u] per u nero (perché è già minimo).

Da queste osservazioni deriva la seguente versione, in cui **i colori non sono considerati** e la sola coda con priorità gestisce i nodi.

Algoritmo di Dijkstra

```
Dijkstra (G, W, s)
   INIZIALIZZA (G)
   D <- empty_priority_queue()
   d[s] <- 0
   for ogni v in V[G]
      enqueue(D,v,d[v])
   while NotEmpty(D) do begin
      u <- dequeue_min(D)
      S <- S ∪ {u} //aggiungo u all'albero definitivamente
      for ogni v adj ad u then
            if d[v] > d[u] + W(u,v) then
               π[v] <- u
               d[v] <- d[u] + W(u,v)
               decrease_key(D,v,d[v])
      end for
   end
```

Complessità

Chiamiamo:

t_c il tempo impiegato dalla **costruzione della coda**;

t_e il tempo impiegato da una **estrazione del minimo**;

t_d il tempo impiegato da una **decrease key**.

Ricordando (ancora una volta) che una visita ha complessità **O(n+m)**

Ad **ogni ciclo** della visita devo **estrarre il minimo** dalla coda

Per **ogni arco** trovato potrei dover **decrementare la chiave** di un vertice

Il tempo totale è allora:

$$O(t_c + nt_e + mt_d)$$

La complessità totale dell'algoritmo dipende dunque (anche) dalle **complessità delle operazioni sulla coda**.

Complessità - II

Coda con priorità realizzata come sequenza non ordinata:

t_c è **O(n)**

t_e è **O(n)**

t_d è **O(1)** **(perché?)**

TOT **$O(n + n^2 + m) = O(n^2 + m)$**

Coda con priorità realizzata come sequenza ordinata:

t_c è **O(n)**

t_e è **O(1)**

t_d è **O(n)**

TOT **$O(n + n + n*m) = O(n + n*m)$**

Versione di Johnson

Johnson ha introdotto nel 1977 l'uso di una coda con priorità realizzata come **heap**. La versione dell'algoritmo di Dijkstra implementata con uno heap è talvolta chiamata **algoritmo di Johnson**.

Complessità algoritmo di Johnson:

t_c è **O(n)**

t_e è **O(log n)**

t_d è **O(log n)**

TOT **O((m+n) log n)**

Attenzione – Grafi Densi

Ricorderete sicuramente che un grafo denso è un grafo in cui

$$|E| \cong |V|^2$$

Quindi in cui $m = \Theta(n^2)$.

Di conseguenza, per grafi densi, la complessità è

Coda con priorità realizzata come **sequenza non ordinata**:

$O(n^2 + m) = O(n^2 + n^2) = O(n^2)$

Coda con priorità realizzata come **sequenza ordinata**:

$O(n + n*m) = O(n + n*n^2) = O(n^3)$

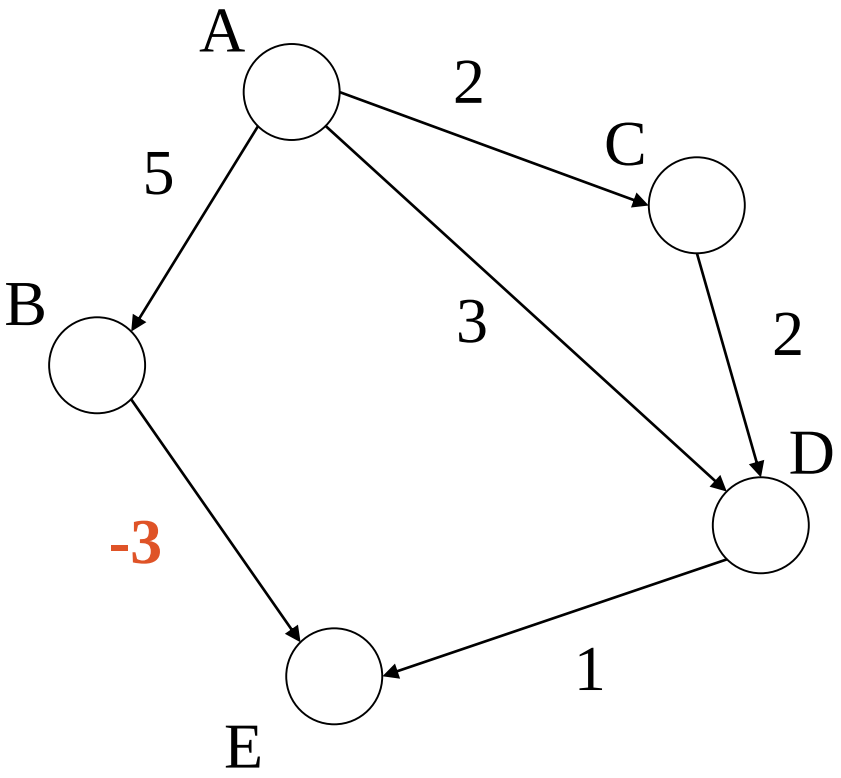
Algoritmo di Johnson

$O((m+n) \log n) = O((n^2+n) \log n) = O(n^2 \log n)$

Questo è un tipico caso in cui conoscere qualcosa in più sul grafo può aiutare…

Fallimento di Dijkstra

Proviamo ad applicare l'algoritmo al seguente grafo



Alla fine dell'algoritmo, **d[E] = 4**.

Ma **δ(A,E) = 2**.

L'algoritmo di Dijkstra funziona **solo** con **archi non negativi**(?).

Proprietà 1 dei sottocammini (minimi) di un cammino minimo

PROPRIETÀ: Un **sottocammino** di un **cammino minimo è un cammino minimo**. (**SOTTOSTRUTTURA OTTIMA**)

DIMOSTRAZIONE:

Sia u ⇝ v un **sottocammino** di un **cammino minimo** da s a t:

s ⇝ **u** ⇝ **v** ⇝ t

Se u ⇝ v non fosse minimo, ci sarebbe **un altro cammino da u a v di costo inferiore**.

Ma allora sostituendo tale cammino nel cammino da s a t, si otterrebbe un **cammino da s a t di costo inferiore** rispetto al cammino minimo.

Esempio: se il percorso più breve fra Torino e Firenze comprende un tratto fra Alessandria e La Spezia, questo sarà evidentemente il percorso più breve fra Alessandria e La Spezia!

Proprietà 2

Siano **S** l'insieme dei vertici già considerati dalla visita (cioè aggiunti definitivamente all'albero dei cammini minimi) e **D** l'insieme dei vertici ancora da considerare. Le seguenti asserzioni

2.1 $\forall v \in V: v \in S \Rightarrow d[v]$ **non viene (più) modificata**

2.2 $\forall v \in D: \pi[v] \neq NULL \Rightarrow \pi[v] \in S$

2.3 $\forall v \in V - \{s\}: d[v] \neq \infty \Leftrightarrow \pi[v] \neq NULL$

2.4 $\forall v \in V - \{s\}: d[v] \neq \infty \Rightarrow d[v] = d[\pi[v]] + W(\pi[v], v)$

Sono **invarianti** del ciclo.

Dimostrazione ovvia dal corpo del ciclo.

Proprietà 3

$\forall v \in S : d[v] \neq \infty \Leftrightarrow$ esiste un cammino da s a v in G

=>

Dimostriamo che, se per il vertice u estratto dalla coda **$d[u] \neq \infty$**, il **cammino esiste**.

Ipotesi induttiva: **esiste un cammino da s a $\pi[u]$**.

Allora **il cammino da s a $\pi[u]$** più l'arco **$(\pi[u], u)$** costituisce un **cammino** da s a u.

<=

Dimostriamo che, se **esiste un cammino da s a v** (con $v \in S$) in G, allora **$d[v] \neq \infty$**

Supponiamo, per **assurdo**, che tra s e u vi sia almeno un cammino $s \equiv v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{k-1} \rightarrow v_k \equiv u$ e che u venga estratto da D con $d[u] = \infty$.

Allora, v_{k-1} è già stato estratto (pro. 2.2) e quindi $d[u] = d[v_k] = d[v_{k-1}] + W(v_{k-1}, v_k)$

Ma $W(v_{k-1}, v_k) < \infty$ e $d[u] = \infty$. Quindi $d[u] = \infty \Rightarrow$ **$d[v_{k-1}] = \infty$**. Questo può essere iterato per **ciascun vertice** del cammino.

Ma anche s appartiene al cammino, quindi, iterando fino ad s, **$d[s] = \infty$**. Questo è ovviamente **assurdo** ($d[s] = 0$).

Correttezza

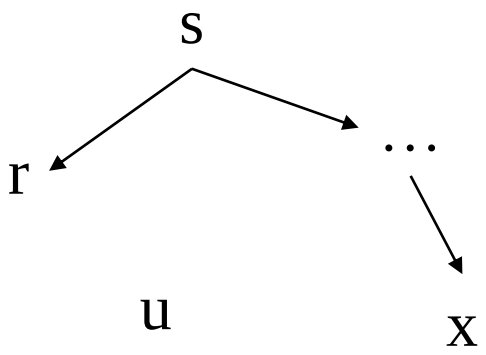
Dimostriamo ora (per induzione) che il predicato

$\forall t \in S: d[t] = \delta(s, t)$ è un invariante del ciclo while.

BASE: Il predicato è vero all'inizio poiché S è vuoto.

PASSO: Supponiamo che sia vero quando l'albero è stato costruito parzialmente, e dimostriamo che **per il nuovo vertice u** estratto da D, **$d[u] = \delta(s, u)$**.

Per ipotesi induttiva **$\forall t \in S: d[t] = \delta(s, t)$**.



Sia u il vertice estratto da D. Abbiamo 2 casi.

CASO 1: d[u] ≠ ∞

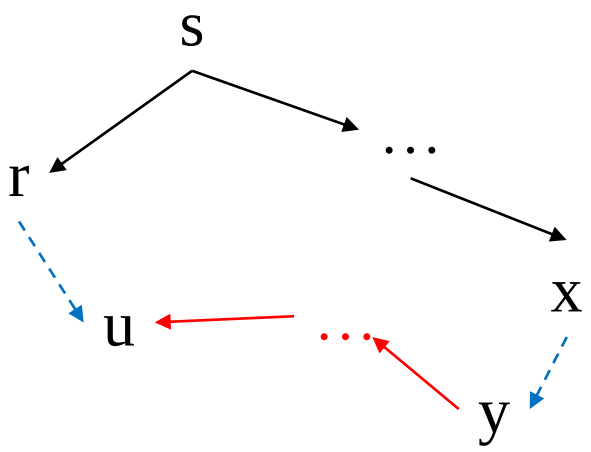
Sia **π[u] = r ≠ NULL** (proprietà 2.3).

Sappiamo allora (proprietà 2.2) che **r ∈ S**, cioè all'albero dei cammini minimi

e che (proprietà 2.4) **d[u] = d[r] + W(r, u)**.

Supponiamo (per **assurdo**) che tra s e u esista un cammino di peso minore di d[u]

esso deve contenere un arco tra un vertice in S (nodi «neri») e uno in D (nodi «grigi»). Poniamo siano x il vertice in S e y quello in D.



Questo cammino tra s e u può allora essere visto come la concatenazione di tre cammini

s ⇝ x ⇝ y ⇝ u

Se s ⇝ x ⇝ y ⇝ u è minimo, anche s ⇝ x ⇝ y è minimo (per la proprietà dei sottocammini minimi)

quindi d[y], che è il suo peso, è uguale a δ(s, y).

W(s ⇝ x ⇝ y ⇝ u) = W(s ⇝ x ⇝ y) + W(y ⇝ u)

= d[y] + W(y ⇝ u)

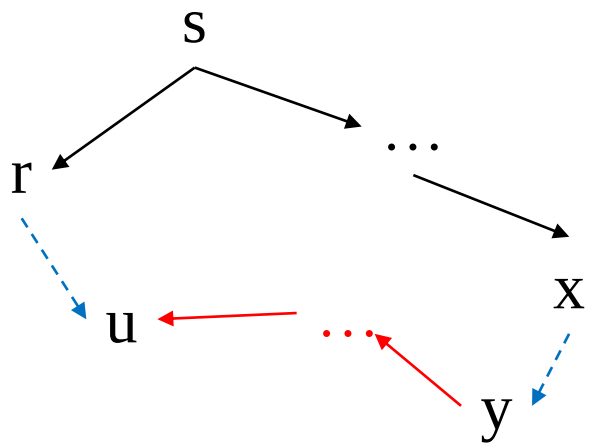
ma d[y] + W(y ⇝ u) ≥ d[y] **(perché W(y ⇝ u) ≥ 0)**

e d[y] **≥ d[u]** **(perché u è stato estratto e ha d[u] minimo)**

Quindi W(s, … x, … y, … u) = d[y] + W(y ⇝ u)

non può essere minore di d[u],

Quindi **d[u] = δ(s, u)** **CVD.**



CASO 2: d[u] = ∞

Se u è estratto con d[u] = ∞, allora (per la proprietà 3) non esiste nessun cammino tra s e u, cioè **d[u] = ∞ = δ(s, u)**

Di conseguenza **d[u] = δ(s, u) è un invariante di ciclo ed è vero alla fine del ciclo.**

Ricerca di un cammino minimo singolo

A volte possiamo aver bisogno di trovare solamente un **cammino minimo tra 2 nodi**, **s** e **t**. In questo caso, possono essere possibili diverse ottimizzazioni:

OTT 1: ogni volta che si estrae un nodo u da D, si può controllare se **coincide con t**. In questo caso, si può terminare l'algoritmo.

OTT 2: nota: poiché l'algoritmo esegue una visita in ampiezza, si vanno a calcolare prima tutti i cammini di lunghezza inferiore a δ(s, u).

Se δ(s, u) è lungo, si devono calcolare parecchi cammini. Sarebbe quindi preferibile «**indirizzare**» la ricerca per cercare di **costruire meno cammini**.

In Intelligenza Artificiale si usano le **euristiche**, funzioni che individuano i nodi più promettenti.

Algoritmo di Dijkstra bidirezionale (OTT 3)

Dati il nodo di partenza s e il nodo di arrivo t, per restringere il numero di nodi esaminati si può eseguire l'algoritmo di Dijkstra contemporaneamente

- sul grafo **G a partire da s** e
- sul grafo **trasposto G^T, a partire da t**

alternando i passi.

Quando si ottengono un cammino (minimo) da s e uno da t ad uno **stesso nodo v**, il cammino s, … v, … t è un cammino da s a t in G.

Tuttavia, esso **non è necessariamente un cammino minimo da s a t** (il cammino minimo da Torino a Tortona seguito dal cammino minimo da Tortona a Milano non è un cammino minimo da Torino a Milano).

Algoritmo di Dijkstra bidirezionale

Teniamo in memoria la **lunghezza d del miglior cammino** così trovato.

Inizialmente, **$d = \infty$**.

Ogni volta che (nell'algoritmo «in avanti») si percorre un arco (u, v) con u nodo nero (nell'algoritmo «in avanti») e v nodo nero (nell'algoritmo «all'indietro»),

se **$\delta(s, u) + W(u, v) + \delta(v, t) < d$** o meglio

$d[u] + W(u, v) + d^T[v]$ si **aggiorna d**.

Quando si estraggono (in sequenza) un nodo x (nell'algoritmo «in avanti») e un nodo y (nell'algoritmo «all'indietro») tali che

$\delta(s, x) + \delta(y, t) > d$

Allora il cammino trovato di peso d è **il cammino minimo**.

Algoritmo di Dijkstra bidirezionale

Infatti, Dijkstra trova cammini minimi di pesi via via **crescenti**, quindi ogni altro arco (x', y') con x' nodo nero da s, e y' nodo nero da t, sarà tale che

d(s, x') + W (x', y') + d(y', t) > d

e quindi il valore di d **non potrà essere ulteriormente diminuito**

Cosa devo aver capito fino ad ora

- Il problema dei cammini minimi
- I cammini minimi in un grafo pesato. Cosa cambia?
- Varianti del problema
- Approccio ingenuo (perché non funziona?)
- Algoritmo di Dijkstra
  - Relazione con visita di grafo e relazione con algoritmi greedy
  - Implementazione dell'algoritmo
  - Algoritmo di Johnson
- Proprietà dell'albero dei cammini minimi ottenuto con Dijkstra
- Correttezza Dijkstra
- Ricerca di cammini singoli con Dijkstra
- Algoritmo di Dijkstra bidirezionale

…se non ho capito qualcosa

- Alzo la mano e chiedo
- Ripasso sul libro
- Chiedo aiuto sul forum
- Chiedo o mando una mail al docente