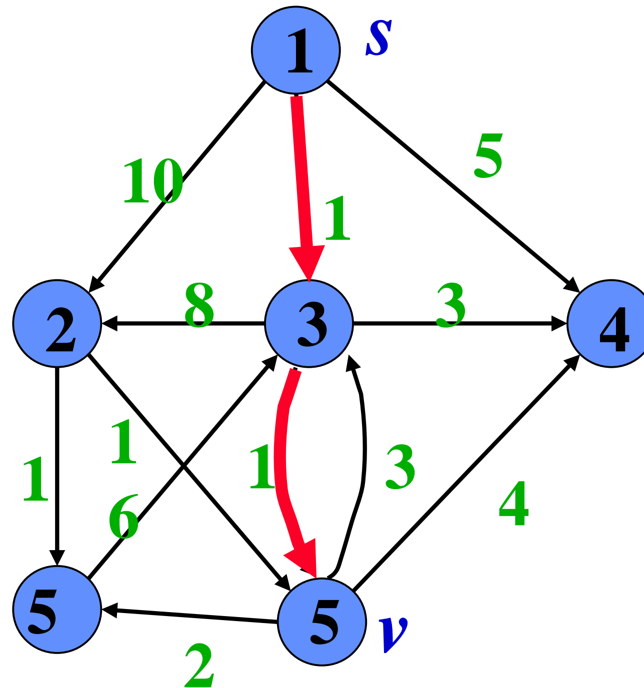
Algoritmi e Strutture dati Mod B

Grafi

Percorsi Minimi: algoritmi esatti e algoritmi euristici (A*)

Grafi: Percorsi minimi

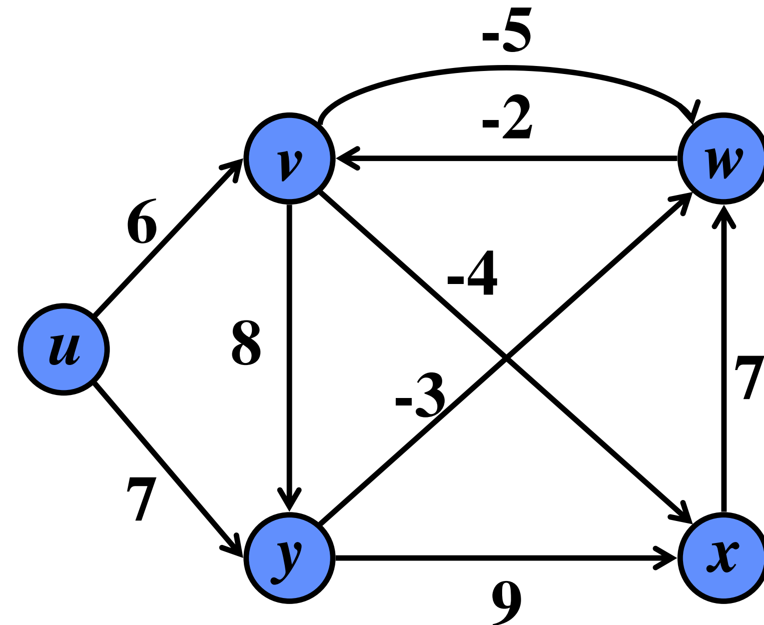
Un *percorso minimo* in un *grafo G=<V,E>* grafo *pesato* orientato, con funzione di peso *w*: $E \rightarrow \Re$ che mappa archi in pesi a valori reali tra due vertici *s* e *v*, è un percorso da *s* a *v* tale che la *somma dei pesi degli archi* che formano il percorso sia *minima*.



Percorsi minimi: pesi negativi

Qual è il *percorso minimo* tra *u* e *x* nel grafo sottostante?

Percorso	Peso
<*u*,*v*,*x*>	2
<*u*,*v*,*w*,*v*,*x*>	-5
<*u*,*v*,*w*,*v*,*w*,*v*,*x*>	-12
<*u*,*v*,*w*,*v*,*w*,*v*,*w*,*v*,*x*>	-19
…	…
…	…



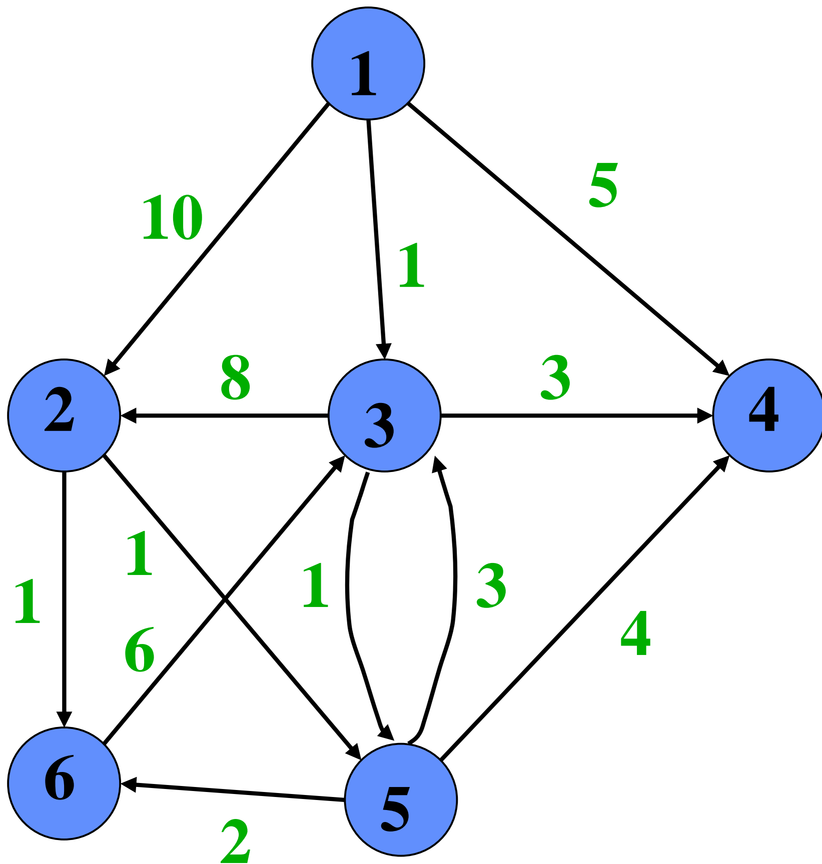
Non **esiste alcun *percorso minimo* tra *u* e *x*!**

Grafi: Percorsi minimi

- **Peso unitario**
  - **Breadth First Search**
- **Pesi non negativi**
  - **Algoritmo di Dijkstra**
- **Pesi negativi con cicli non negativi**
  - **Algoritmo di Bellman-Ford**
- **Cicli negativi**
  - **Nessuna soluzione**

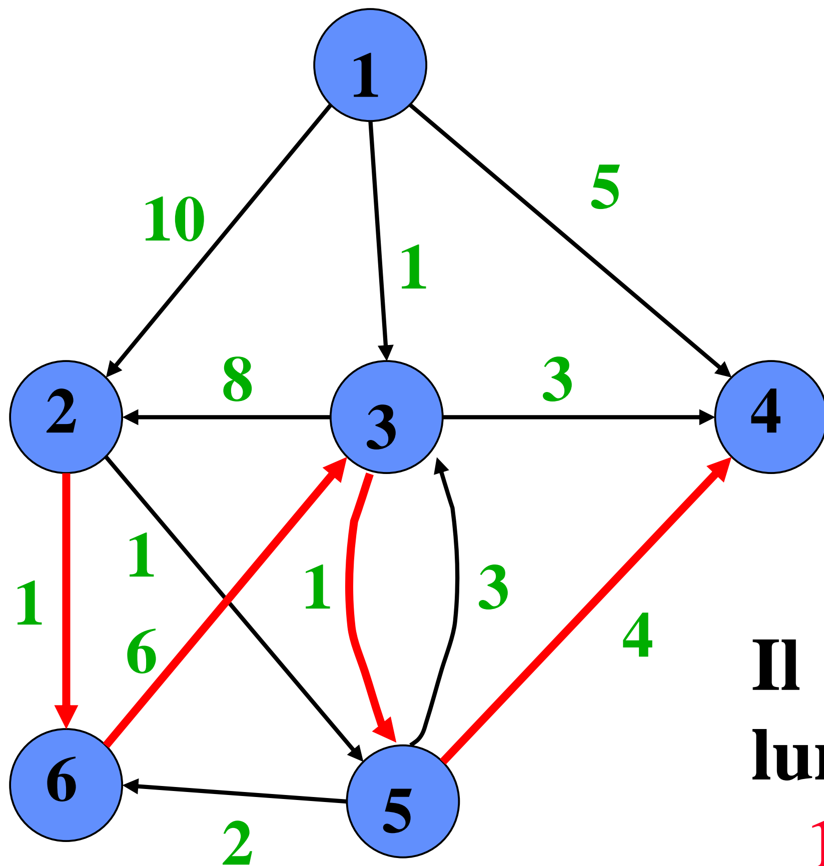
Grafi: Percorsi minimi (Dijkstra)

Sia dato un grafo pesato orientato $G = (V, E)$, con funzione di peso $w: E \rightarrow \Re$ che mappa archi in pesi a valori reali.



Grafi: Percorsi minimi (Dijkstra)

Sia dato un grafo pesato orientato $G = (V, E)$, con funzione di peso $w: E \rightarrow \Re$ che mappa archi in pesi a valori reali.



Il *peso* di un percorso

$$p = (v_1, v_2, \ldots, v_k)$$

è

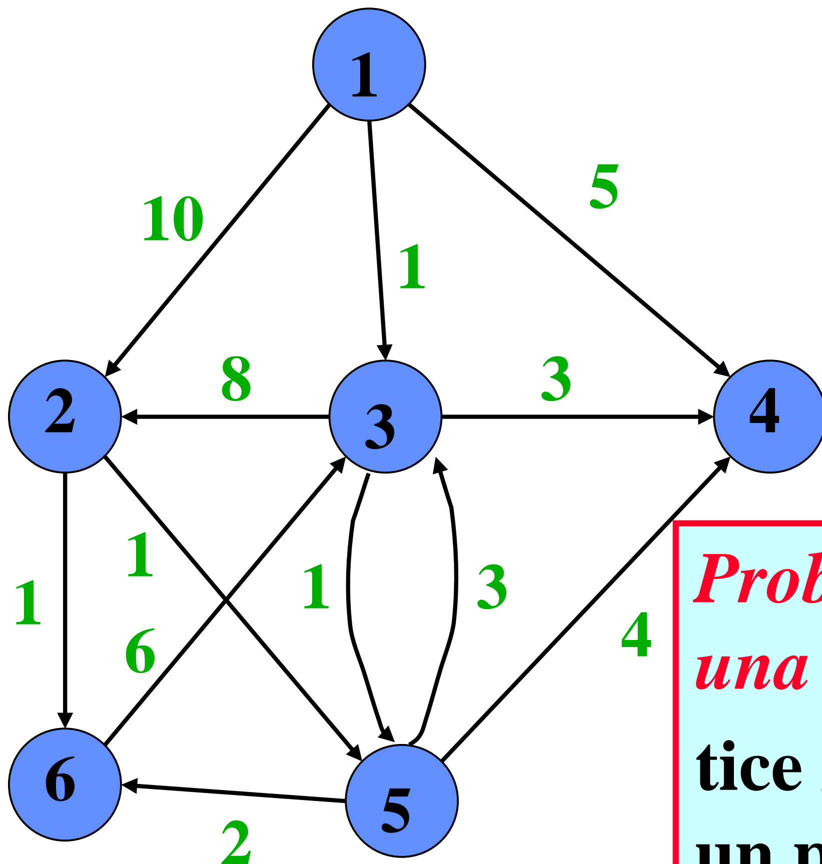
$$\sum_{i=1}^{k-1} w(v_i, v_{i+1}).$$

Il *peso* del percorso lungo gli *archi rossi* è

$$1 + 6 + 1 + 4 = 12.$$

Grafi: Percorsi minimi (Dijkstra)

Sia dato un grafo pesato orientato $G = (V, E)$, con funzione di peso $w: E \rightarrow \Re$ che mappa archi in pesi a valori reali.



Il *peso* di un percorso

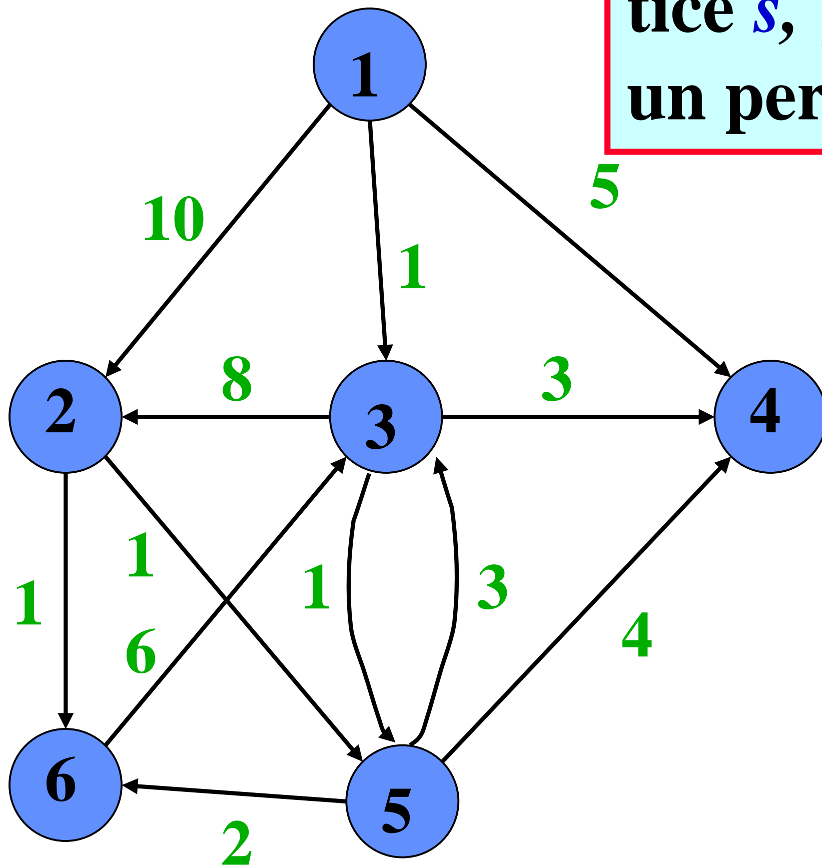
$$p = (v_1, v_2, \ldots, v_k)$$

è

$$\sum_{i=1}^{k-1} w(v_i, v_{i+1}).$$

***Problema del percorso minimo da una singola sorgente*: dato un vertice s, per ogni vertice $v \in V$ trovare un percorso minimo da s a v.**

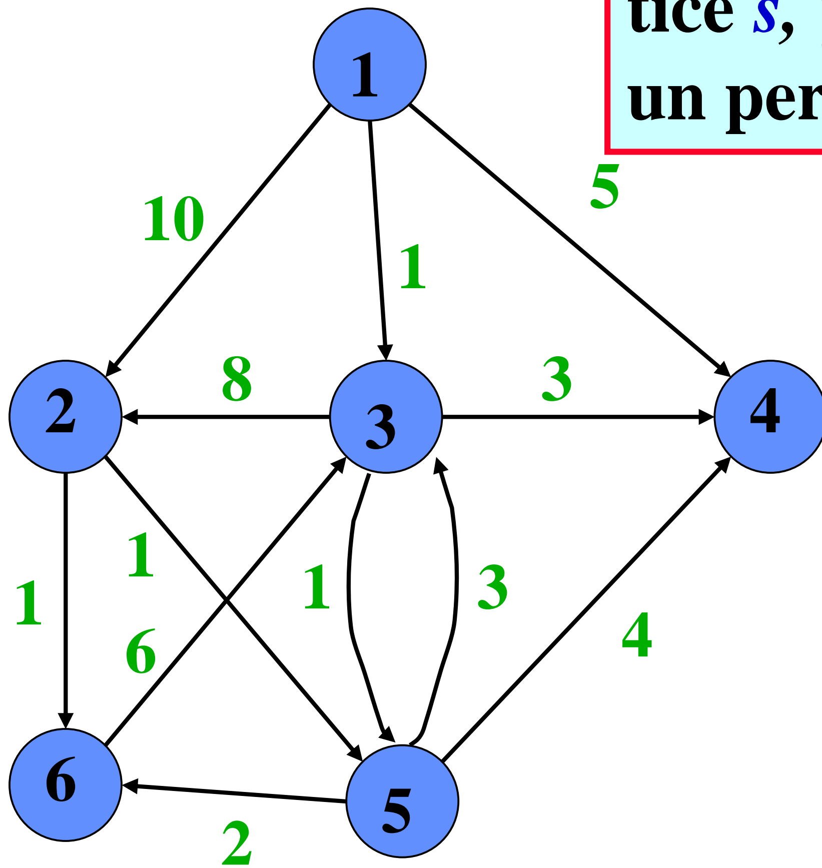
Grafi: Percorsi minimi (Dijkstra)



Problema del percorso minimo da una singola sorgente: **dato un vertice** s**, per ogni vertice** $v \in V$ **trovare un percorso minimo da** s **a** v**.**

L'algoritmo di Dijkstra **risolve il problema in modo efficiente nel caso in cui** ***tutti i pesi siano non-negativi*****, come nell'esempio del grafo.**

Grafi: Percorsi minimi (Dijkstra)



***Problema del percorso minimo da una singola sorgente*: dato un vertice *s*, per ogni vertice $v \in V$ trovare un percorso minimo da *s* a *v*.**

***L'algoritmo di Dijkstra* risolve il problema in modo efficiente nel caso in cui *tutti i pesi siano non-negativi*, come nell'esempio del grafo.**

- Utilizza un campo *d*[*v*] per la stima della *distanza minima*

- Utilizza un campo *p*[*v*] per il *nodo predecessore* di *v*

Sottografo dei predecessori (Dijkstra)

- L'*algoritmo di Dijkstra* sul grafo $G=<V,E>$ costruisce in $p[]$ il *sottografo dei predecessori* denotato con $G_p=<V_p,E_p>$, dove:

$$V_p = \{ v \in V : p[v] \neq \text{Nil} \} \cup \{s\}$$

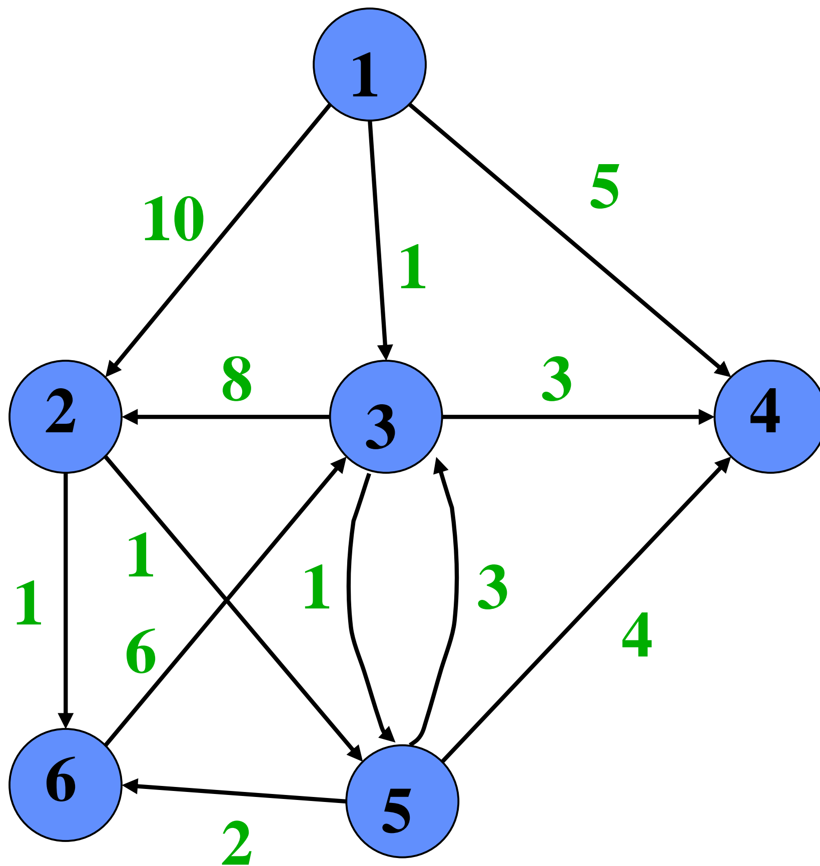
$$E_p = \{ (p[v],v) \in E : v \in V_p - \{s\} \}$$

Il *sottografo dei predecessori* è definito come per *BFS*
La differenza è che ora verrà costruito in modo che i *percorsi che individua* siano quelli con *peso minimo* (*non* col numero minimo di archi)

Si dimostra che il *sottografo dei predecessori* costruito dall'*algoritmo di Dijkstra* è un *albero dei percorsi minimi*

Grafi: Percorsi minimi (Dijkstra)

Inizializzazione del grafo



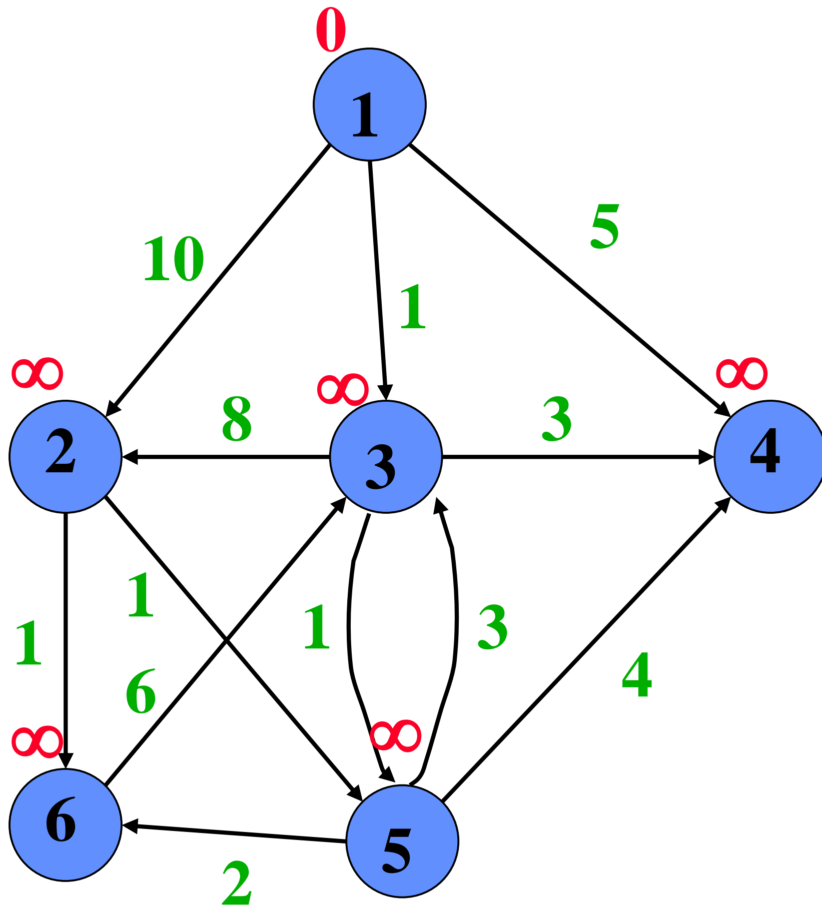

```
Inizializza(G,s)
```

1 *La stima della distanza* $d[s]$ *viene posta a* **0**

2 *Tutte le altre stime delle disanze* $d[v]$ *sono poste a* ∞

Grafi: Percorsi minimi (Dijkstra)

Inizializzazione del grafo



`Inizializza(G,s)`

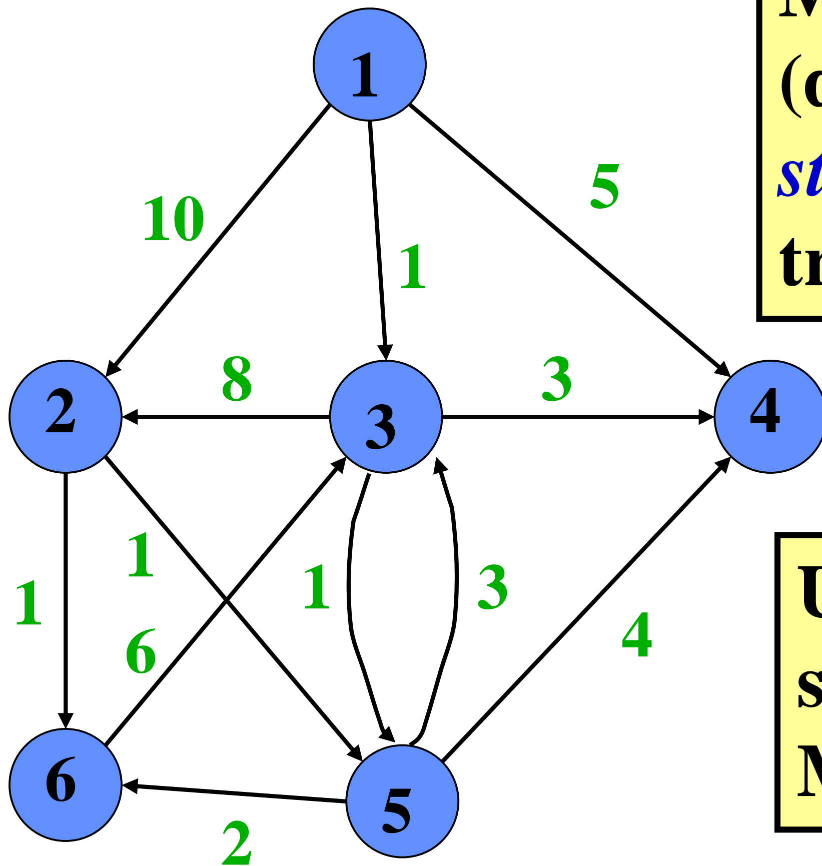
1 La stima della distanza $d[s]$ viene posta a 0

2 Tutte le altre stime delle disanze $d[v]$ sono poste a ∞

3 I predecessori $p[v]$ sono posti a Nil

Grafi: Percorsi minimi (Dijkstra)

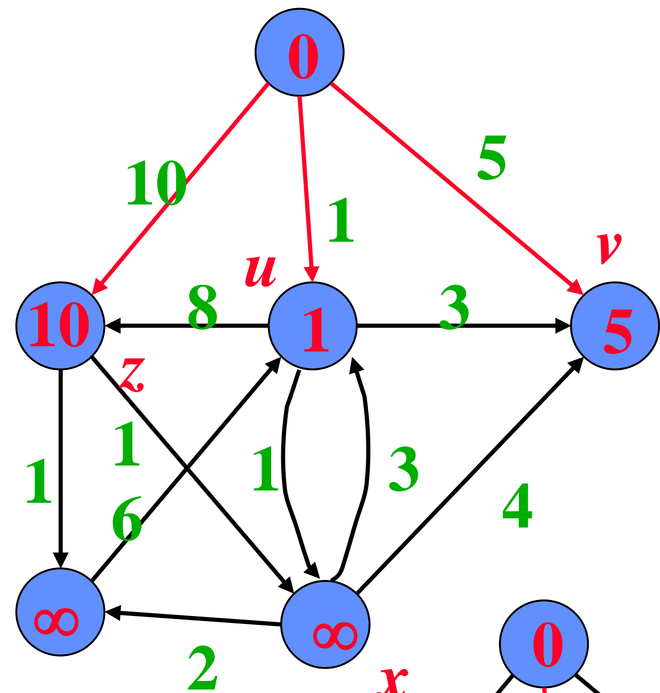
Rilassamento degli archi

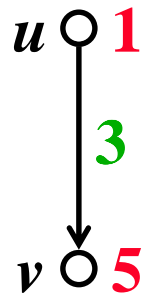


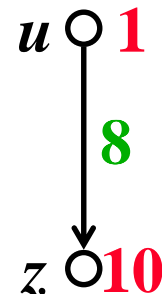
Meccanismo di *aggiustamento* (diminuzione) progressivo *delle stime d*[*v*] delle *distanze minime* tra *s* e gli altri nodi *v*.

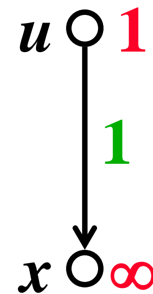
Utilizza la funzione di peso *w* e si applica agli *archi del grafo*. Modifica sia *d*[*v*] che *p*[*v*].

Rilassamento

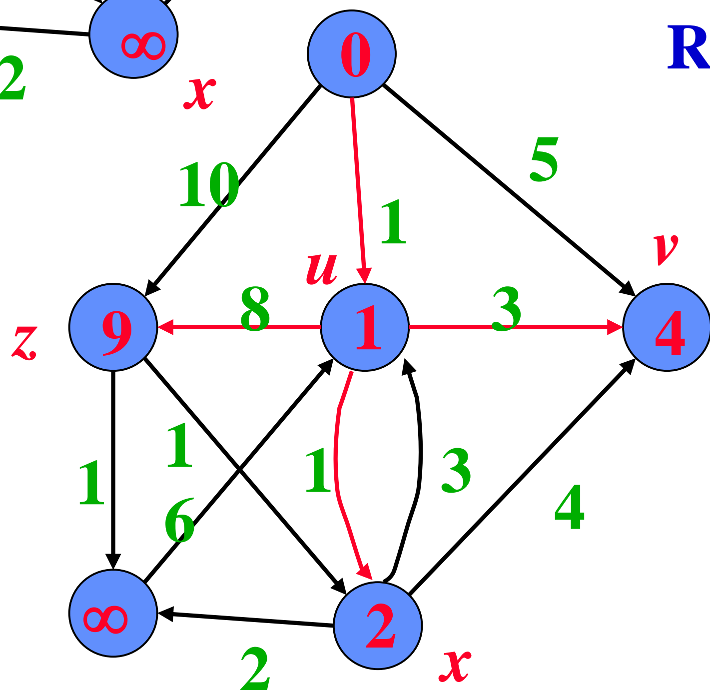


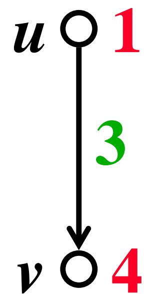


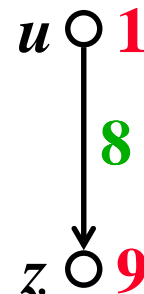


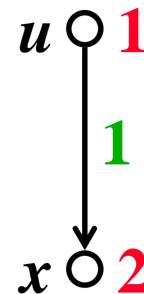


Relax(u,v,w) Relax(u,z,w) Relax(u,x,w)

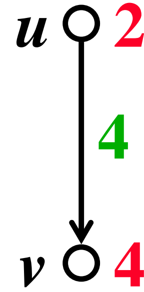


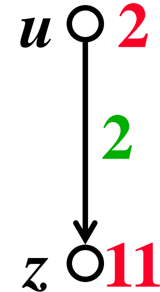




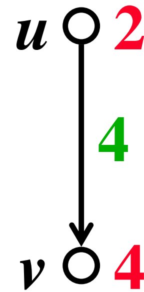


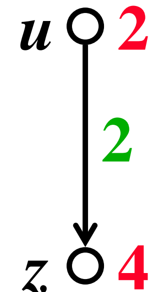
Rilassamento





Relax(u,v,w) **Relax**(u,z,w)





Rilassamento

Verifica **se è possibile *ottenere un percorso migliore* tra *s* e *v* passando per il vertice *u***

- ***d*[*v*]: *estremo superiore* della lunghezza del *percorso minimo* tra *s* a *v* (*s* è la *sorgente*)**
- ***p*[*v*]: il *vertice predecessore* di *v* nel *percorso minimo corrente* tra *s* e *v* (padre di *v*)**
- ***w*(*u*,*v*): *peso* dell'arco (*u*,*v*)**

```
Relax(u,v,w)
   if d[v] > d[u] + w(u,v)
      then d[v] = d[u] + w(u,v)
           p[v] = u
```

Rilassamento: proprietà

Lemma: **Sia dato un grafo pesato orientato $G = (V, E)$, con funzione di peso $w: E \rightarrow \Re$. Sia s la sorgente e il grafo sia inizializzato con una chiamata a `Inizializza(G,s)`.**

Allora, vale $d[v] \geq \delta(s,v)$ per ogni vertice v di G e tale *invariante* viene mantenuto da ogni sequenza di operazioni di *rilassamento*.

Inoltre, appena $d[v] = \delta(s,v)$, $d[v]$ *non cambia più*.

Rilassamento: proprietà

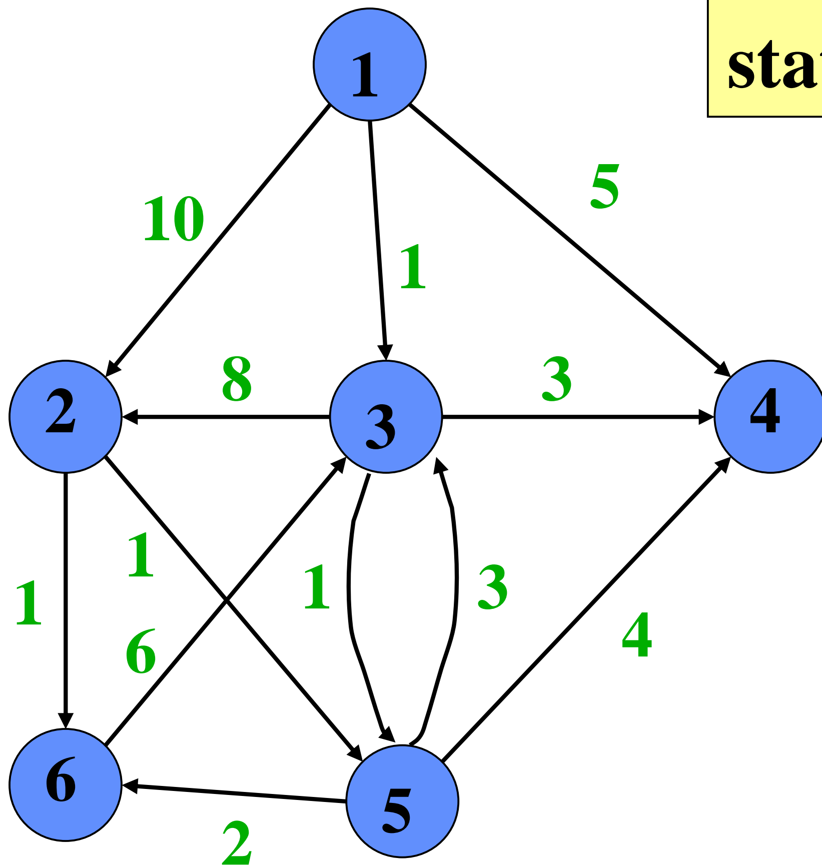
Lemma: **Sia dato un grafo pesato orientato $G = (V, E)$, con funzione di peso $w: E \rightarrow \Re$.**

Sia s la sorgente e $s \xrightarrow{p'} u \rightarrow v$ sia un *percorso minimo*, per $u,v \in V$.

Il grafo sia inizializzato con una chiamata a `Inizializza(G,s)` e venga applicata una sequenza di operazioni di *rilassamento* che includa `Relax(u,v,w)`.

Se $d[u] = \delta(s,u)$ in qualunque momento *prima della chiamata* `Relax(u,v,w)`, allora vale sicuramente $d[v] = \delta(s,v)$ *dopo la chiamata*.

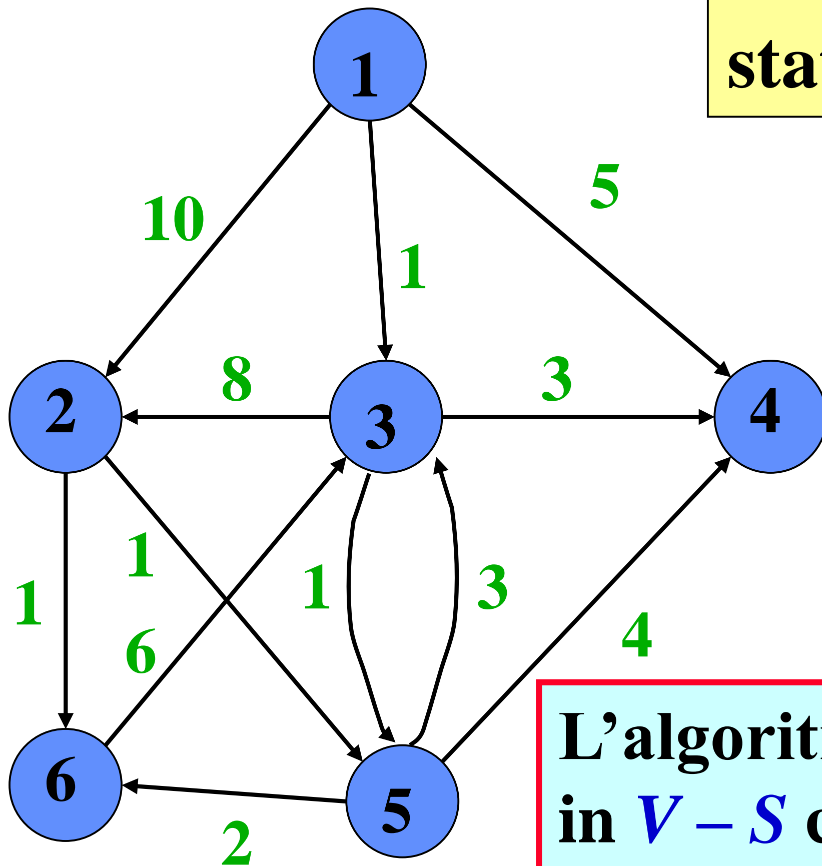
Grafi: Percorsi minimi (Dijkstra)



L'*algoritmo di Dijkstra* utilizza un insieme *S* di vertici i cui pesi del percorso minimo sono già stati determinati.

Utilizza anche, per ogni vertice *v* non in *S*, un campo *d*[*v*] contenente ad ogni passo l'*estremo superiore* del peso del percorso minimo da *s* a *v*.

Grafi: Percorsi minimi (Dijkstra)

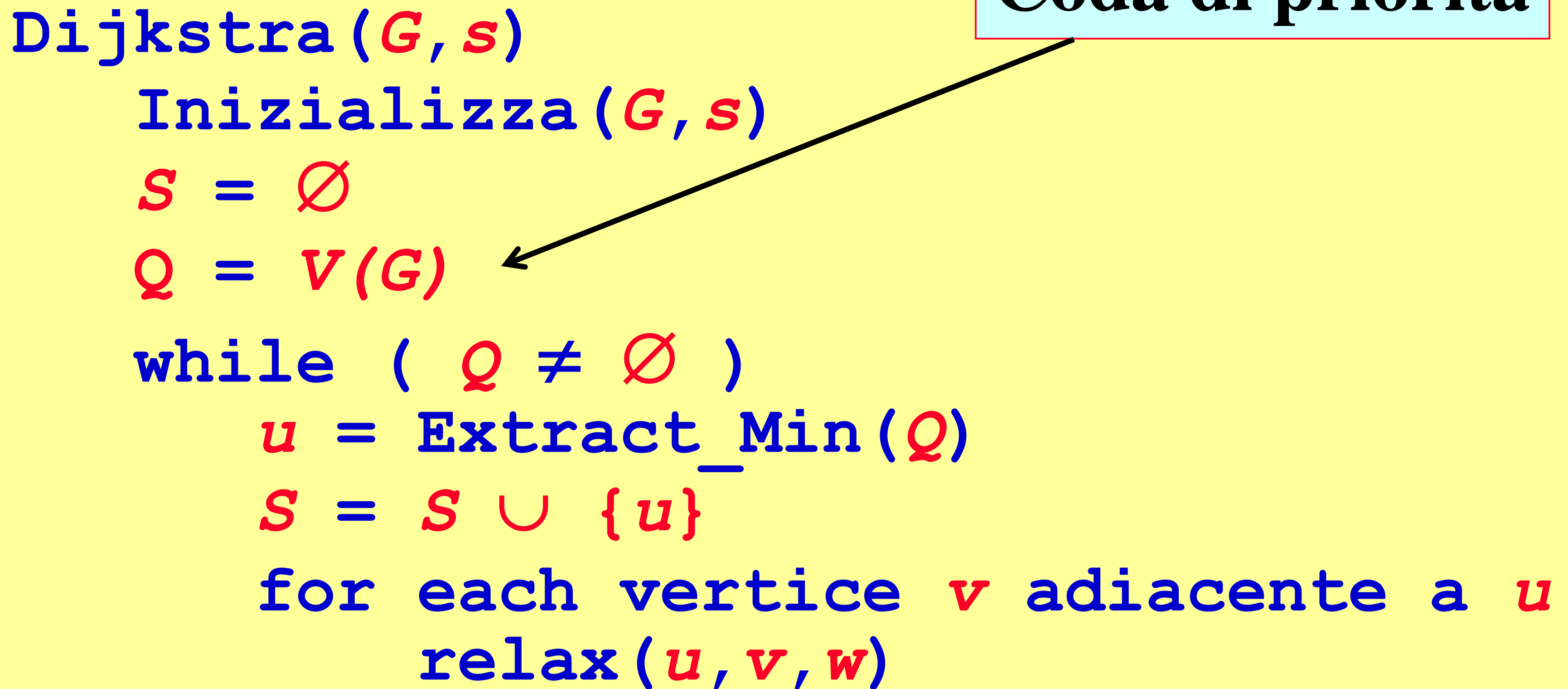


L'*algoritmo di Dijkstra* utilizza un insieme *S* di vertici i cui pesi del percorso minimo sono già stati determinati.

Utilizza anche, per ogni vertice *v* non in *S*, un campo *d*[*v*] contenente ad ogni passo l'*estremo superiore* del peso del percorso minimo da *s* a *v*.

L'algoritmo seleziona a turno il vertice *u* in *V* – *S* col *minimo valore* *d*[*u*], inserisce *u* in *S*, e *rilassa* tutti gli archi uscenti da *u*.

L'algoritmo di Dijkstra



L'algoritmo di Dijkstra

Coda di priorità

```
Dijkstra(G,s)
    Inizializza(G,s,d)
    S = ∅
    Q = V(G)
    while ( Q ≠ ∅ )
        u = Extract_Min(Q)
        S = S ∪ {u}
        for each vertice v adiacente a u
            relax(u,v,w)
```

Operazione riduzione del valore di un elemento

```
Relax(u,v,w)
    if d[v] > d[u] + w(u,v)
      then Decrase_key(d[v],d[u]+w(u,v))
           p[v] = u
```

Tempo di esecuzione: Dijkstra

***Tempo di esecuzione*: verifichiamo il tempo in relazione a *differenti implementazioni* della *coda di priorità*.**

Differenti implementazioni danno *differenti costi* per le *operazioni sulla coda*.

- **`Extract_Min` quante volte viene eseguita?**

Tempo di esecuzione: Dijkstra

***Tempo di esecuzione*: verifichiamo il tempo in relazione a *differenti implementazioni* della *coda di priorità*.**

Differenti implementazioni danno *differenti costi* per le *operazioni sulla coda*.

- **`Extract_Min` viene eseguita $O(|V|)$ volte.**

Tempo di esecuzione: Dijkstra

Tempo di esecuzione**: verifichiamo il tempo in relazione a *****differenti implementazioni***** della *****coda di priorità*****.**

Differenti implementazioni danno *****differenti costi*** per le *****operazioni sulla coda*****.**

- **`Extract_Min` viene eseguita $O(|V|)$ volte.**
- **`Decrease_key` viene eseguita $O(|E|)$ volte.**

Tempo di esecuzione: Dijkstra

Tempo di esecuzione: **verifichiamo il tempo in relazione a** ***differenti implementazioni*** **della** ***coda di priorità*.**
Differenti implementazioni danno ***differenti costi*** **per le** ***operazioni sulla coda*.**

- **`Extract_Min`** **viene eseguita $O(|V|)$ volte.**
- **`Decrease_key`** **viene eseguita $O(|E|)$ volte.**
- *Tempo totale* $= |V|\, T_{\text{Delete_min}} + |E|\, T_{\text{Decrease_key}}$

Tempo di esecuzione: Dijkstra

Tempo di esecuzione: **verifichiamo il tempo in relazione a** ***differenti implementazioni*** **della** ***coda di priorità*****.**
Differenti implementazioni danno ***differenti costi*** **per le** ***operazioni sulla coda*****.**

- **`Extract_Min`** **viene eseguita** $O(|V|)$ **volte.**
- **`Decrease_key`** **viene eseguita** $O(|E|)$ **volte.**
- *Tempo totale* $= |V|\, T_{\text{Delete_min}} + |E|\, T_{\text{Decrease_key}}$

Coda a priorità	$T_{\text{Delete_min}}$	$T_{\text{Decrease_key}}$	*Tempo Totale*
Array non ordinato			
Heap binario			

Tempo di esecuzione: Dijkstra

Tempo di esecuzione: **verifichiamo il tempo in relazione a** ***differenti implementazioni*** **della** ***coda di priorità*****.**

Differenti implementazioni danno ***differenti costi*** **per le** ***operazioni sulla coda*****.**

- `Extract_Min` **viene eseguita** $O(|V|)$ **volte.**
- `Decrease_key` **viene eseguita** $O(|E|)$ **volte.**
- *Tempo totale* $= |V|\, T_{\text{Delete_min}} + |E|\, T_{\text{Decrease_key}}$

Coda a priorità	$T_{\text{Delete_min}}$	$T_{\text{Decrease_key}}$	*Tempo Totale*
Array **non ordinato**	$O(\|V\|)$	$O(1)$	
Heap binario			

Tempo di esecuzione: Dijkstra

Tempo di esecuzione: verifichiamo il tempo in relazione a *differenti implementazioni* della *coda di priorità*. Differenti implementazioni danno *differenti costi* per le *operazioni sulla coda*.

- `Extract_Min` viene eseguita $O(|V|)$ volte.
- `Decrease_key` viene eseguita $O(|E|)$ volte.
- *Tempo totale* = $|V|\ T_{\text{Delete_min}} + |E|\ T_{\text{Decrease_key}}$

Coda a priorità	$T_{\text{Delete_min}}$	$T_{\text{Decrease_key}}$	*Tempo Totale*
Array non ordinato	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Heap binario	$O(\log \|V\|)$	$O(\log \|V\|)$	

Tempo di esecuzione: Dijkstra

Tempo di esecuzione: **verifichiamo il tempo in relazione a *differenti implementazioni* della *coda di priorità*.**

Differenti implementazioni danno *differenti costi* per le *operazioni sulla coda*.

- `Extract_Min` **viene eseguita $O(|V|)$ volte.**
- `Decrease_key` **viene eseguita $O(|E|)$ volte.**
- ***Tempo totale*** $= |V|\, T_{\text{Delete_min}} + |E|\, T_{\text{Decrease_key}}$

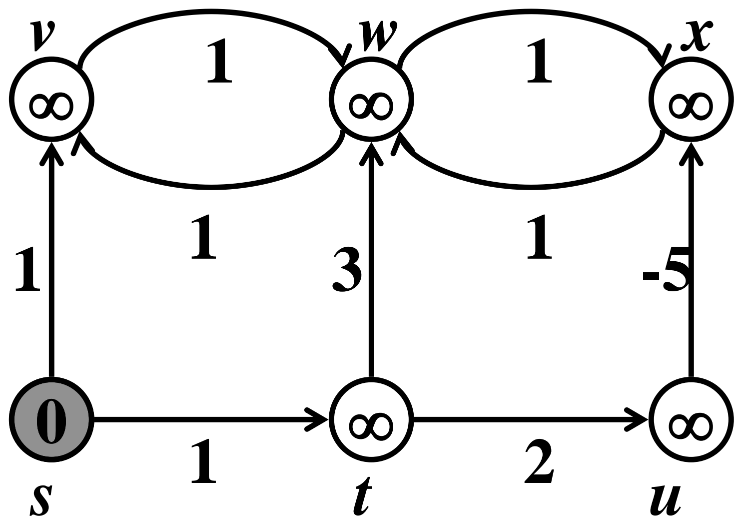
Coda a priorità	$T_{\text{Delete_min}}$	$T_{\text{Decrease_key}}$	*Tempo Totale*
Array non ordinato	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Heap binario	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(E \log \|V\|)$

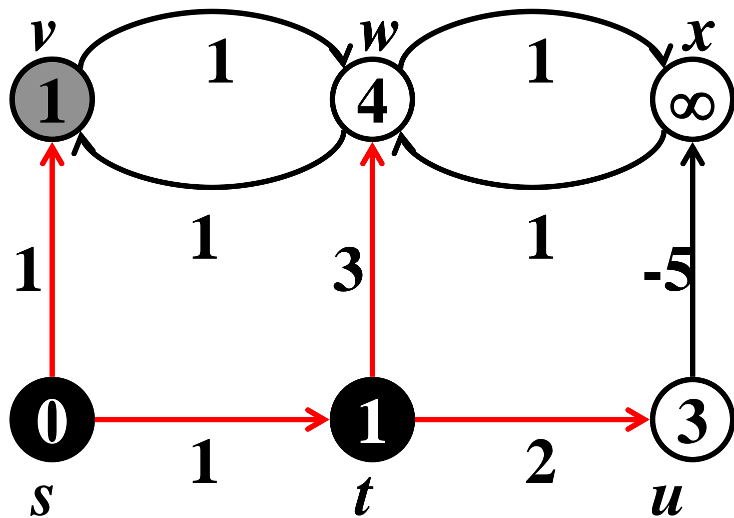
Algoritmo di Dijkstra: correttezza

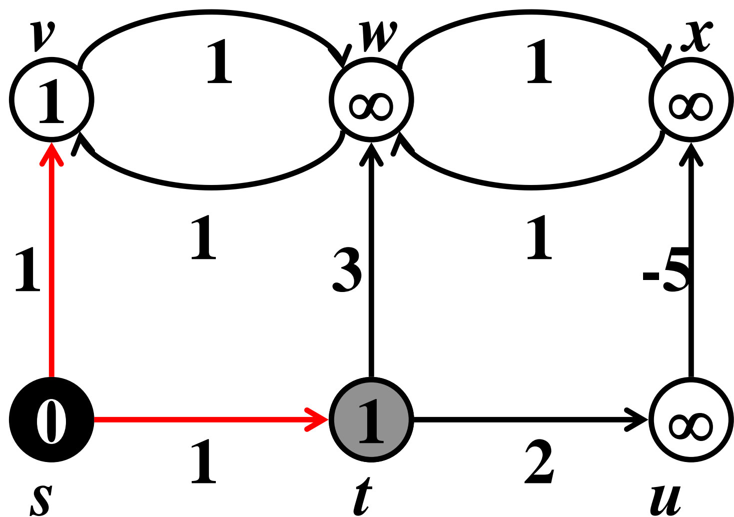
Teorema: Se eseguiamo l'*algoritmo di Dijkstra* su un grafo pesato orientato $G = (V, E)$, con funzione di peso $w: E \rightarrow \Re$ che mappa archi in pesi a valori reali *non-negativi*, e un vertice sorgente s, allora, alla terminazione, $d[u] = \delta(s,u)$ per tutti i vertici u in V.

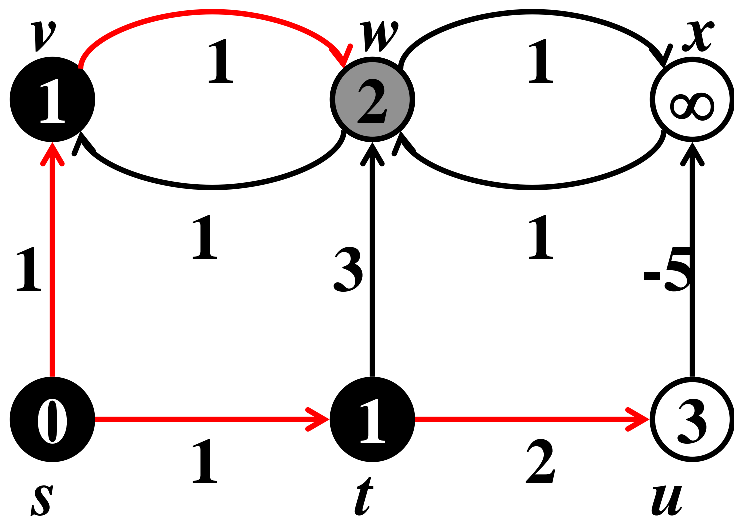
Algoritmo di Dijkstra: correttezza 2

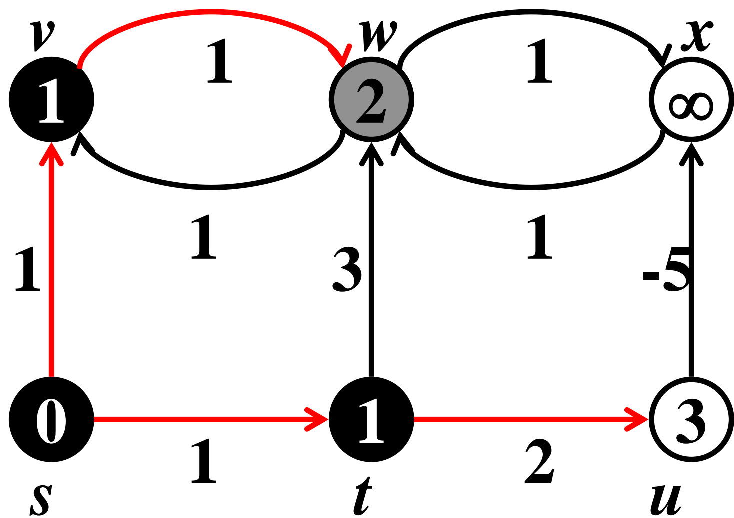
Corollario: **Se eseguiamo l'*algoritmo di Dijkstra* su un grafo pesato orientato $G = (V, E)$, con funzione di peso $w: E \rightarrow \Re$ che mappa archi in pesi a valori reali *non-negativi*, e un vertice sorgente s, allora, alla terminazione il *sottografo dei predecessori* G_p corrisponde all'*albero dei percorsi minimi* con *radice* s.**

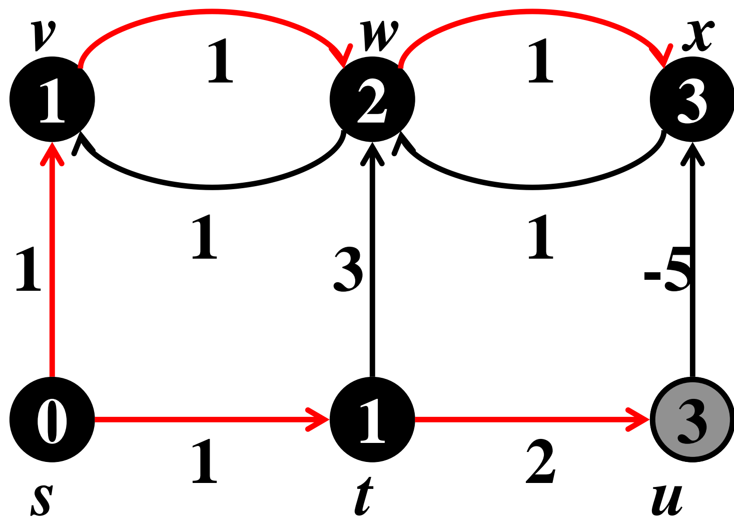

v
w
x
1
1
∞
∞
∞
1
1
1
3
-5
0
∞
∞
1
2
s
t
u

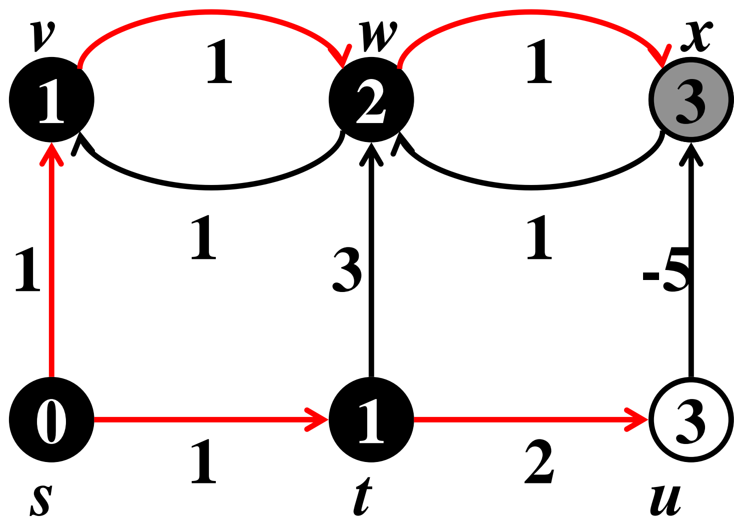

v
w
x
1
1
1
4
∞
1
1
1
3
-5
0
1
3
1
2
s
t
u

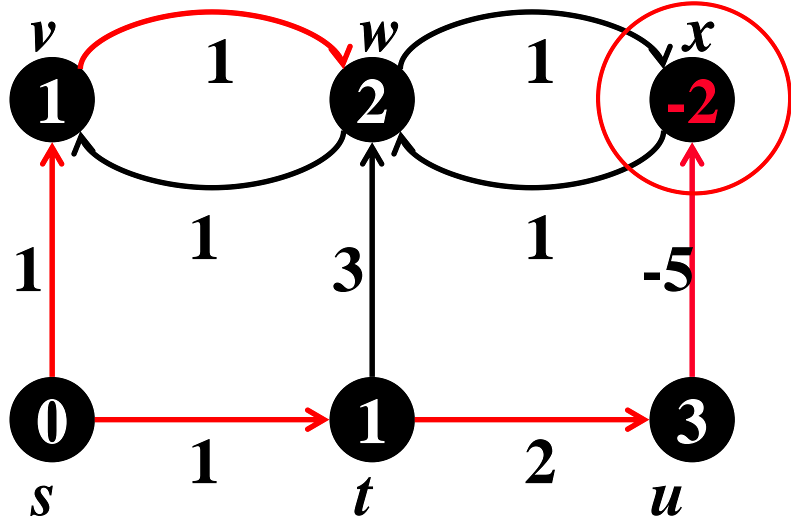

v
w
x
1
1
1
∞
∞
1
1
1
3
-5
0
1
∞
1
2
s
t
u


v
w
x
1
1
1
2
∞
1
1
1
3
-5
0
1
3
1
2
s
t
u

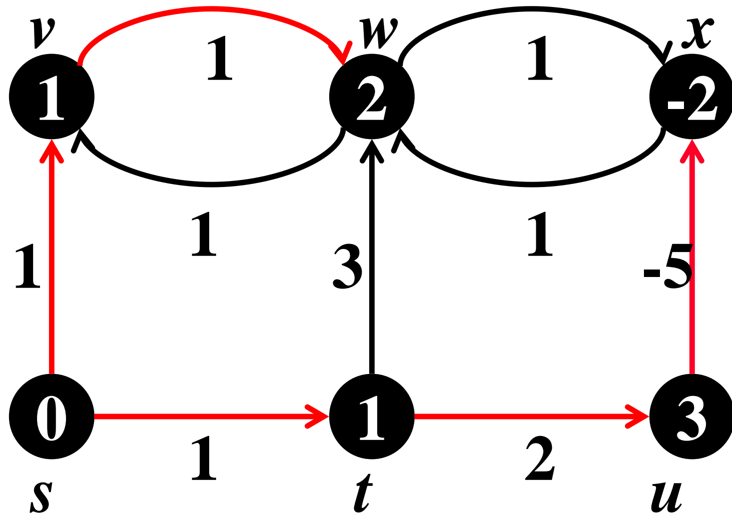

v
w
x
1
2
∞
1
1
1
1
1
3
-5
0
1
3
1
2
s
t
u

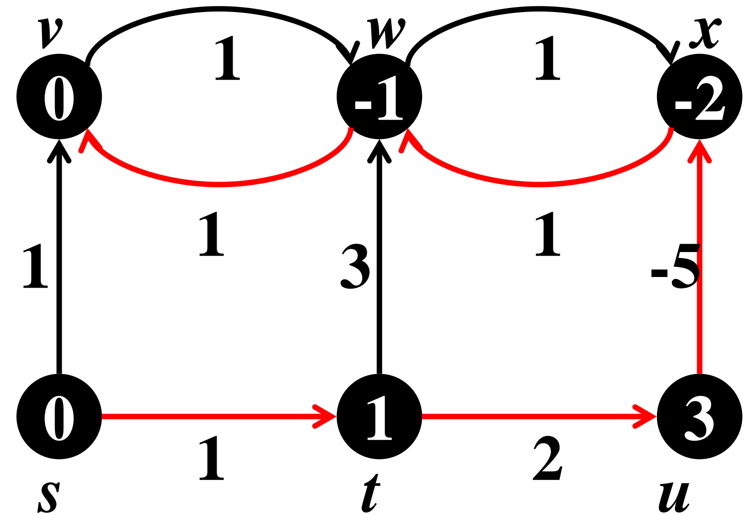

v
w
x
1
2
3
1
1
1
1
1
3
-5
0
1
3
1
2
s
t
u


v
w
x
1
2
3
1
1
1
1
1
3
-5
0
1
3
1
2
s
t
u


v
w
x
1
2
-2
1
1
1
1
1
3
-5
0
1
3
1
2
s
t
u

Problemi con l'Algoritmo di Dijkstra



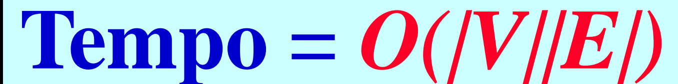


Soluzione di Dijkstra

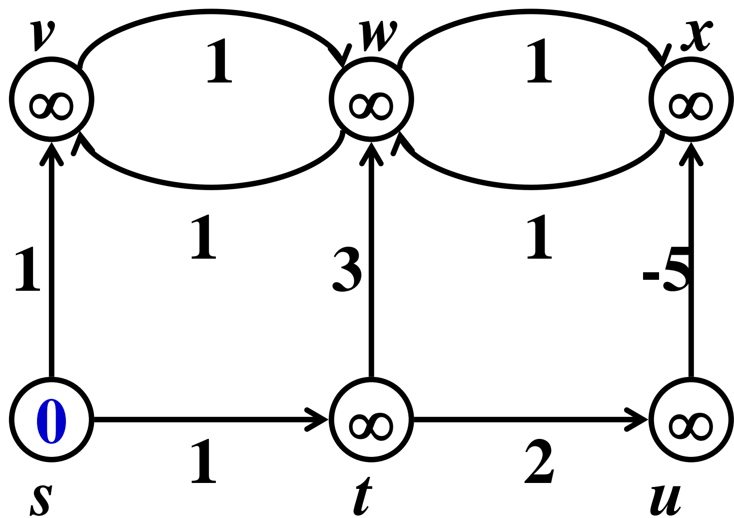
Soluzione Ottimale

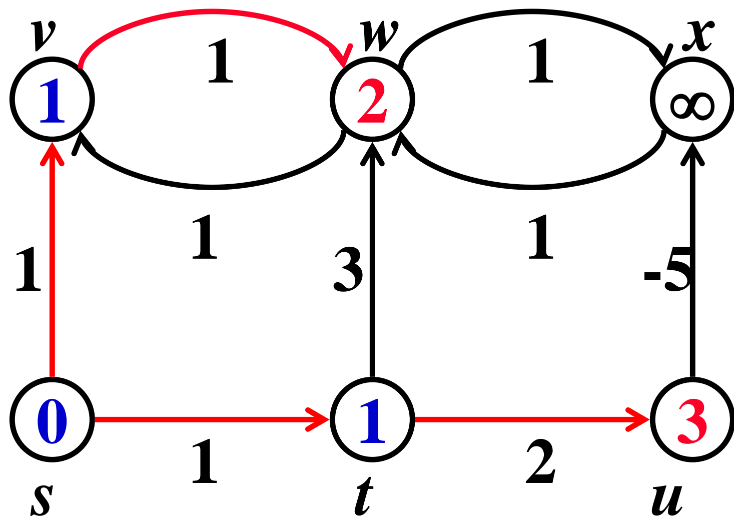
Algoritmo di Bellman-Ford

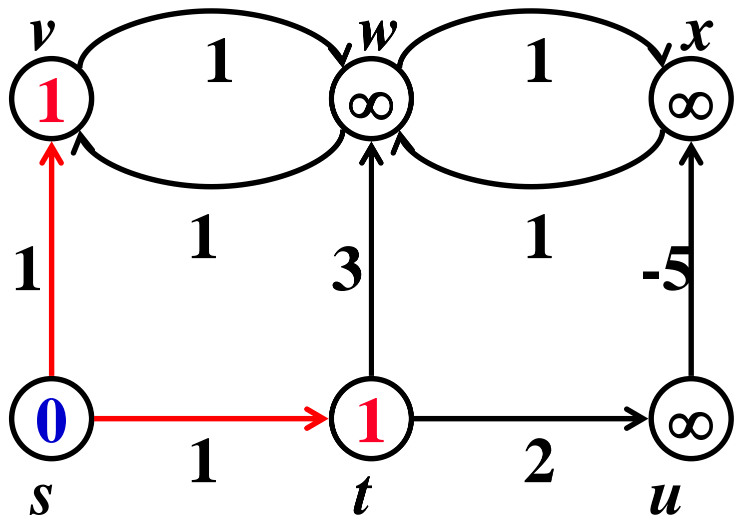
- Inizalmente, $d[s] = 0$ e $d[u] = \infty$ per $u \neq s$
- Vengono fatte $|V| - 1$ passate
- Ad ogni passata, si applica il rilassamento *a ogni arco*

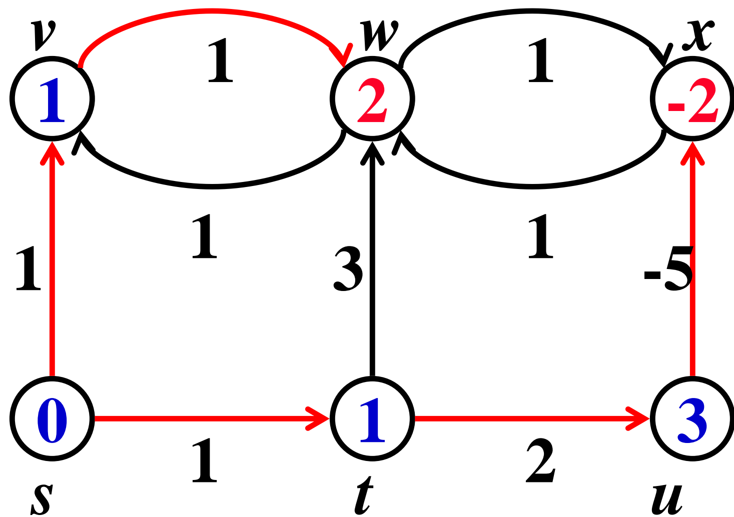


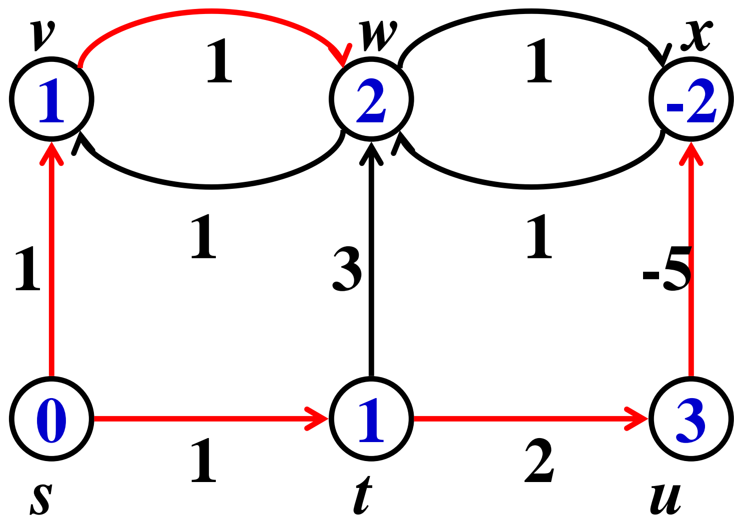

```
Bellman_Ford(G,s)
  Inizializza(G,s,d)
  for i = 1 to |V|-1 do
    for each arco (u,v) in E do
         relax(u,v,w)
  for each arco (u,v) in E do
      if d[v] > d[u] + w(u,v) then
         return FALSE
  return TRUE
```


v
w
x
1
1
∞
∞
∞
1
1
1
3
-5
0
∞
∞
1
2
s
t
u

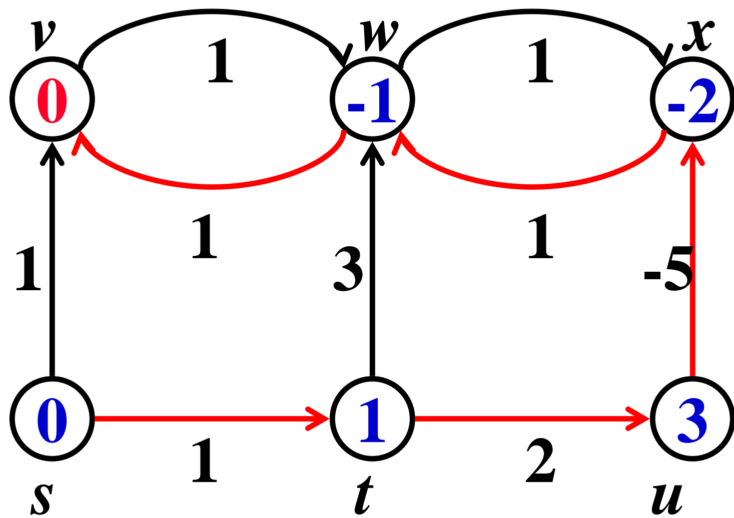

v
w
x
1
1
1
2
∞
1
1
1
3
-5
0
1
3
1
2
s
t
u


v
w
x
1
1
1
∞
∞
1
1
1
3
-5
0
1
∞
1
2
s
t
u

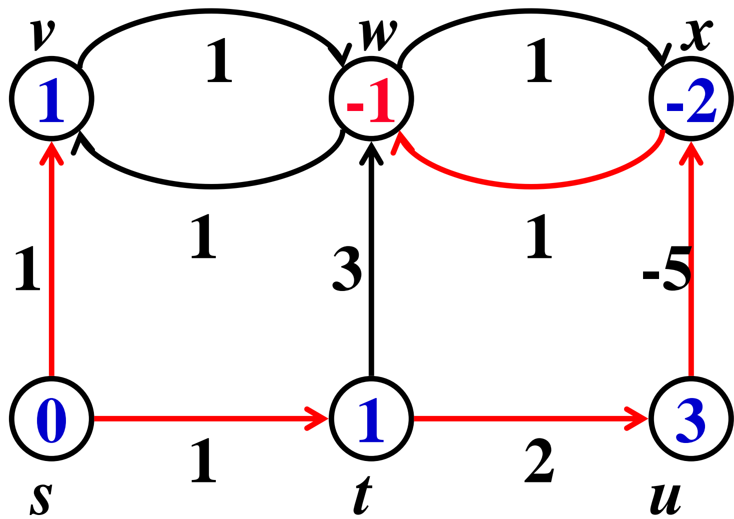

v
w
x
1
1
1
2
-2
1
1
1
3
-5
0
1
3
1
2
s
t
u



v
w
x
1
1
1
2
-2
1
1
1
3
-5
0
1
3
1
2
s
t
u



v
w
x
1
1
0
-1
-2
1
1
1
3
-5
0
1
3
1
2
s
t
u



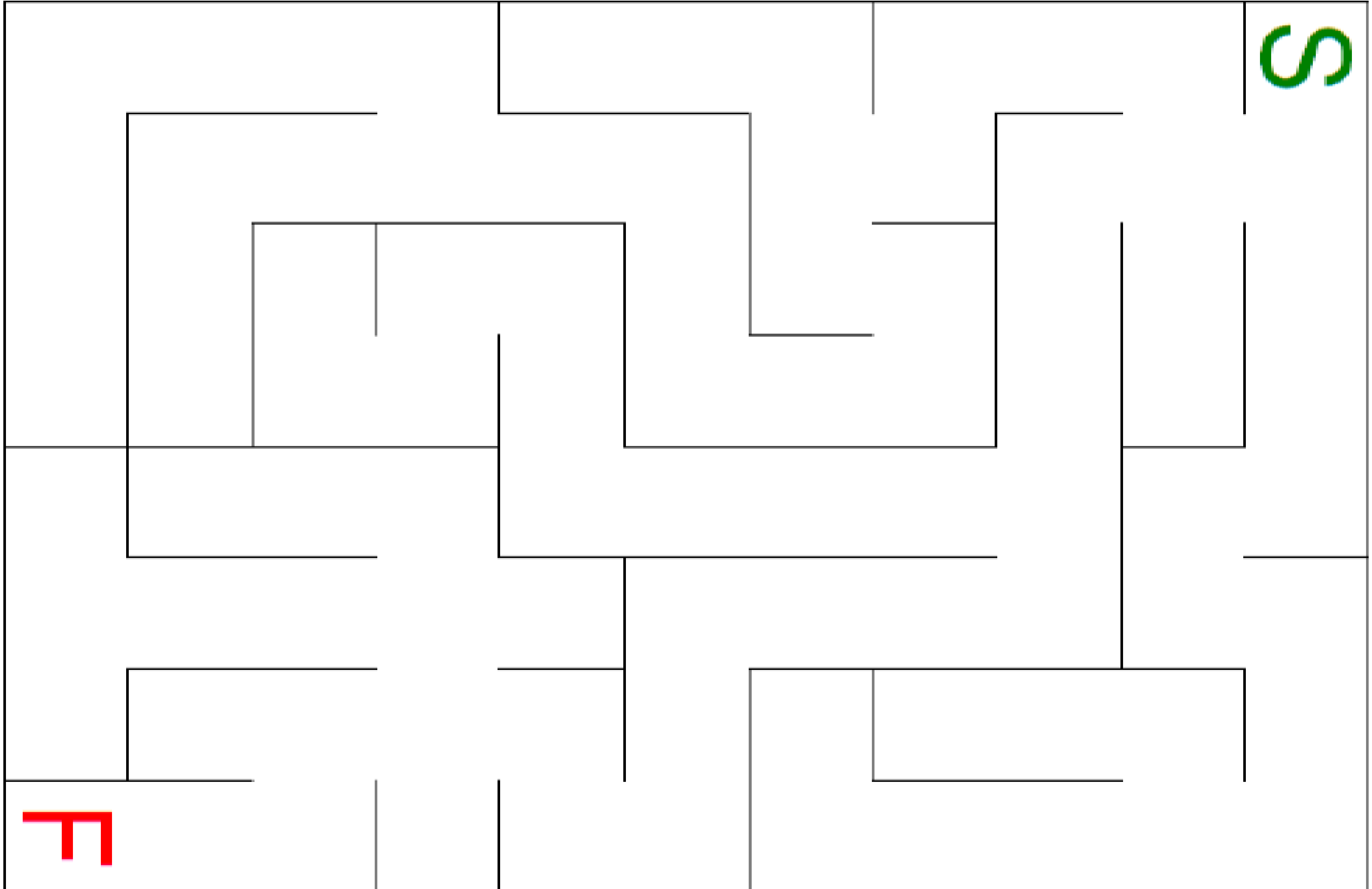
v
w
x
1
1
1
-1
-2
1
1
1
3
-5
0
1
3
1
2
s
t
u

Bellman-Ford: correttezza

Teorema: **Si esegua l'*algoritmo di Bellman-Ford* su un grafo pesato orientato $G = (V, E)$, con funzione di peso $w: E \rightarrow \Re$, e un vertice sorgente s:**

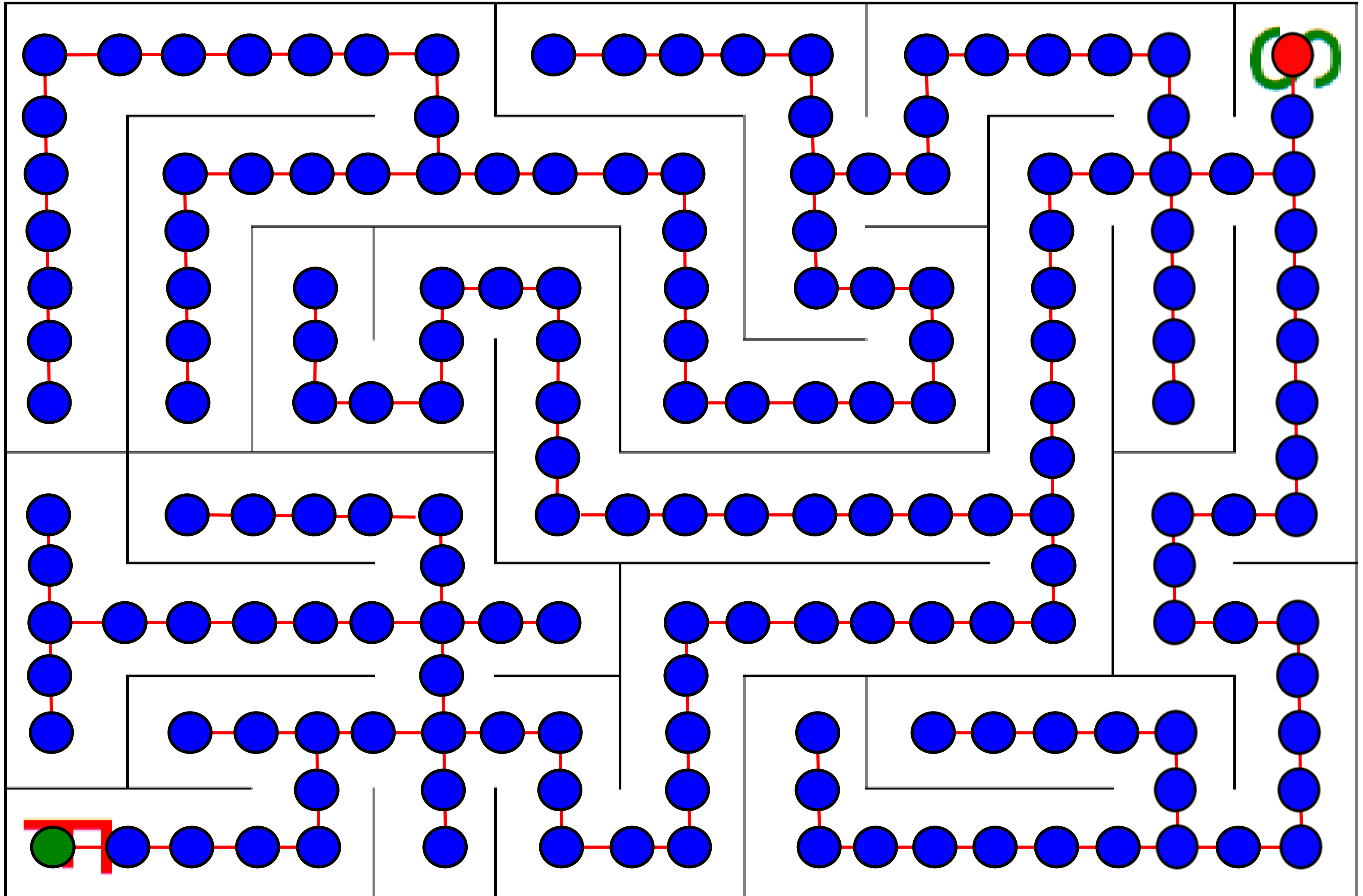
- ***se non esistono cicli negativi raggiungibili da s, allora*** **l'algoritmo ritorna `TRUE`, $d[v] = \delta(s,v)$, $\forall v \in V$, inoltre il *grafo dei predecessori* G_p è l'*albero dei percorsi minimi*;**
- ***se esistono cicli negativi raggiungibili da s, allora*** **l'algoritmo ritorna `FALSE`.**

Labirinto

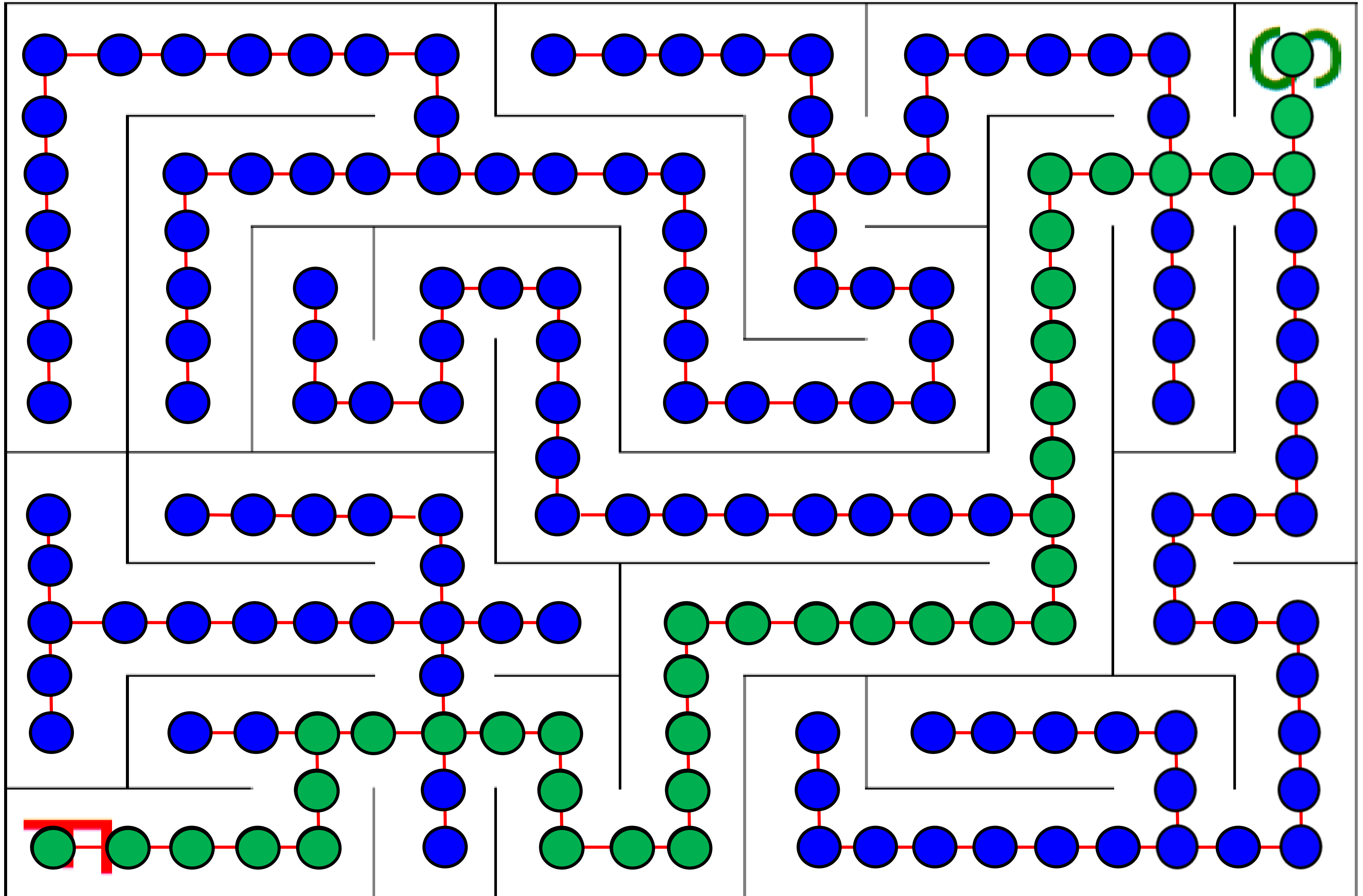


Labirinto

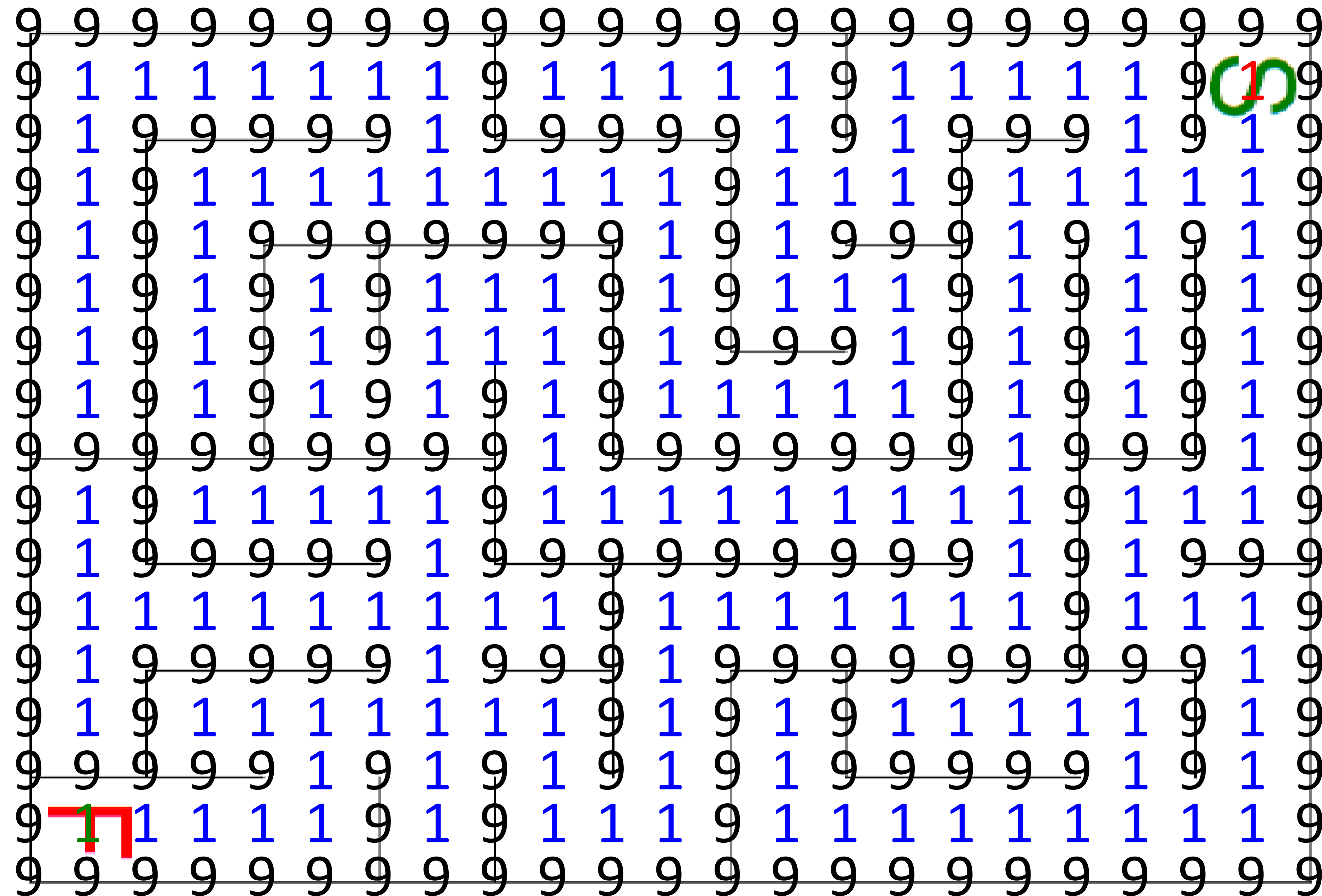
Labirinto: rappresentazione esplicita



Labirinto: rappresentazione esplicita



Labirinto: rappresentazione implicita



Labirinto: rappresentazione implicita

9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
9	1	1	1	1	1	1	1	9	1	1	1	1	1	9	1	1	1	1	1	9	1	9
9	1	9	9	9	9	9	1	9	9	9	9	9	1	9	1	9	9	9	1	9	1	9
9	1	9	1	1	1	1	1	1	1	1	1	9	1	1	1	9	1	1	1	1	1	9
9	1	9	1	9	9	9	9	9	9	9	1	9	1	9	9	9	1	9	1	9	1	9
9	1	9	1	9	1	9	1	1	1	9	1	9	1	1	1	9	1	9	1	9	1	9
9	1	9	1	9	1	9	1	1	1	9	1	9	9	9	1	9	1	9	1	9	1	9
9	1	9	1	9	1	9	1	9	1	9	1	1	1	1	1	9	1	9	1	9	1	9
9	9	9	9	9	9	9	9	9	1	9	9	9	9	9	9	9	1	9	9	9	1	9
9	1	9	1	1	1	1	1	9	1	1	1	1	1	1	1	1	1	9	1	1	1	9
9	1	9	9	9	9	9	1	9	9	9	9	9	9	9	9	9	1	9	1	9	9	9
9	1	1	1	1	1	1	1	1	1	9	1	1	1	1	1	1	1	9	1	1	1	9
9	1	9	9	9	9	9	1	9	9	9	1	9	9	9	9	9	9	9	9	9	1	9
9	1	9	1	1	1	1	1	1	1	9	1	9	1	9	1	1	1	1	1	9	1	9
9	9	9	9	9	1	9	1	9	1	9	1	9	1	9	9	9	9	9	1	9	1	9
9	1	1	1	1	1	9	1	9	1	1	1	9	1	1	1	1	1	1	1	1	1	9
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9

Ricerca del percorso ottimo

La nozione di ***percorso ottimo*** dipende dal problema.

Se ogni passo (arco) lungo un percorso ha lo stesso costo, essa collassa con la nozione di ***percorso di lunghezza minima***.

Può essere risolto tramite un algoritmo di ricerca (visita) del grafo sottostante.

Ad esempio, con una semplice ***visita in ampiezza*** (**BFS**) si può trovare il percorso di lunghezza minima. Non risolve correttamente il problema per generici grafi pesati!

Per grafi di dimensioni medio-grandi la visita in ampiezza risente di scarsa efficienza.

Ricerca euristica

- Variante della ricerca in ampiezza: ad ogni nodo ***v*** visitato è associato un ***valore f(v) che stima la bontà*** del percorso ***completo*** fino all'obiettivo.
- la ***stima*** è ottenuta per via ***empirica***, basata su esperienza o conoscenza ***specifica del dominio***.
- la ricerca del percorso ottimo procede scegliendo di "***estendere***" il percorso parziale che termina nel nodo con stima migliore.
- intuitivamente, ad ogni passo si estende il percorso ***localmente più promettente***.
- si sfrutta una ***funzione euristica*** che ***stima*** la distanza di un nodo da nodo obiettivo.

Stima euristica

Ad ogni nodo visitato ***v*** si associa un valore ***f(v)*** calcolato tramite composizione di due funzioni:

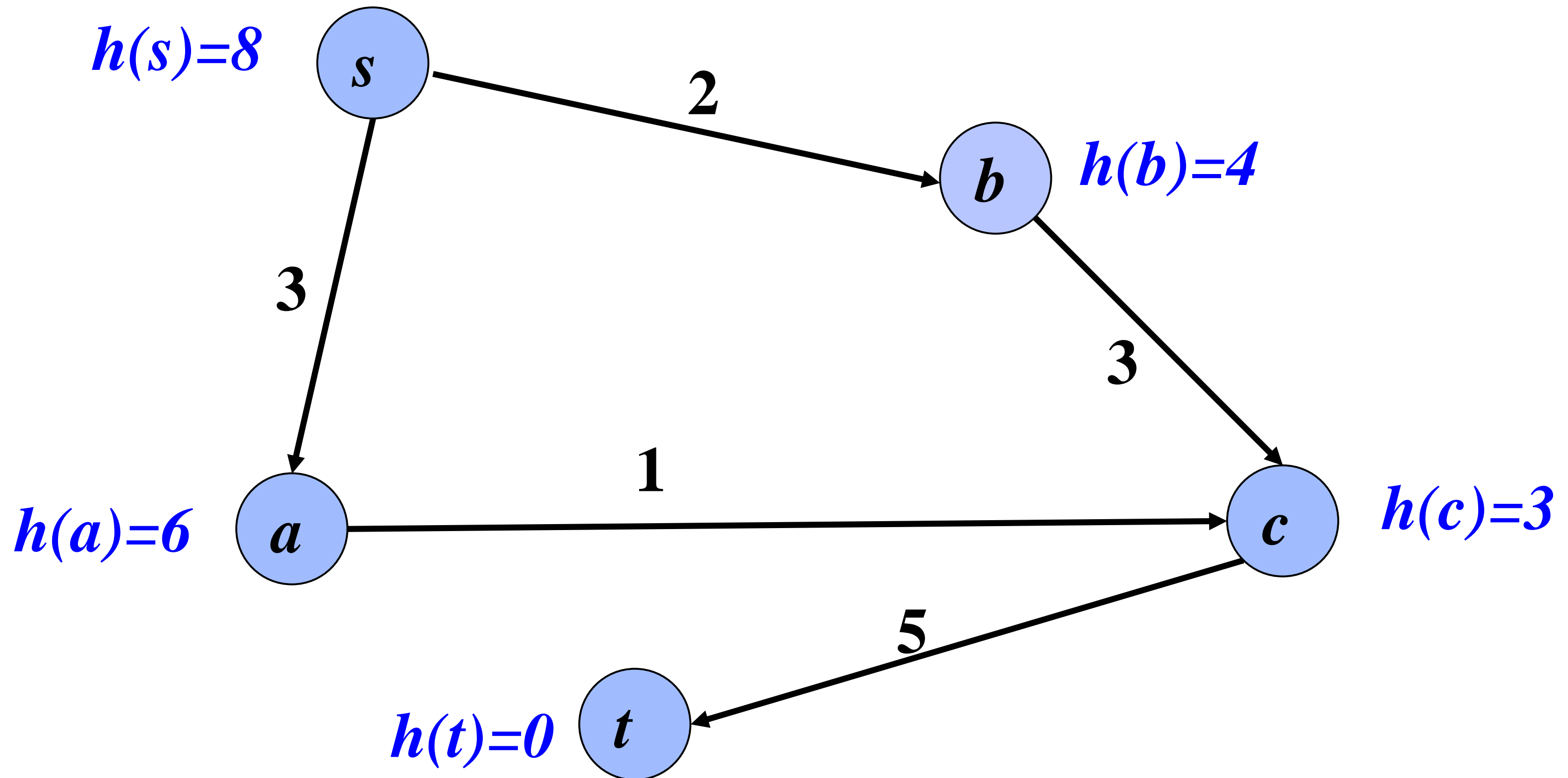
$$f(v) = g(v) + h(v)$$

dove

- **g(v)** è il costo esatto del miglior percorso finora trovato dal nodo di partenza **s** al nodo corrente ***v*** (corrisponde a **d[v]** in ***Dijkstra***);
- **h(v)** è una stima (*previsione*) del costo per raggiungere il **target *t*** a partire dal nodo **v**.

h() si dice ***ammissibile*** se ***h(u) ≤ h*(u,t),*** per ogni nodo ***u*** e target ***t***, con ***h*(u,t)*** il costo ***esatto*** del percorso minimo da ***u*** a ***t***.

Euristica ***ammissibile***



…

Stima euristica

g(v) potrebbe, ad esempio, misurare la lunghezza del percorso migliore trovato finora dalla sorgente s al nodo corrente ***v***;

h(v) dipende fortemente dal dominio applicativo. Nel caso di un ***labirinto bidimensionale*** potrebbe misurare la distanza, misurata sulla griglia ignorando i muri, tra $v = (x_v, y_v)$ e il nodo destinazione $t = (x_t, y_t)$.

Esempi:

- **Distanza Manhattan**: $h(v) = \left(|x_t - x_v| + |y_t - y_v|\right)$
- **Distanza Euclidea**: $h(v) = \sqrt{(x_t - x_v)^2 + (y_t - y_v)^2}$

Algoritmo A*

Popolare algoritmo di ricerca euristica su alberi e su grafi che deriva dalla **BFS**.

Come la **BFS**, distingue tre insiemi di vertici:

- Vertici non raggiunti
- **Open-set**: vertici raggiunti ma non ancora espansi
- **Closed-set**: vertici già espansi

Open-set è l'analogo della coda di vertici grigi (scoperti) della BFS.

Closed-set è l'analogo dell'insieme dei vertici neri (terminati) della BFS.

Algoritmo A*

L'algoritmo è inizializzato con **Closed-set** vuoto e **Open-set** contente solo la sorgente ***s***.

A ogni iterazione sceglie per l'espansione il vertice ***x*** *più promettente* (quello col minor valore della funzione ***f(.)***) da **Open-set**.

Genera i suoi adiacenti (***y***) e per ciascuno di essi:

1. Calcola il valore di ***costo*** ottenibile passando per ***x***;
2. Se ***y*** è presente in **Open-set** aggiorna, se necessario, la stima ***f(y)*** in **Open-set** col valore ***costo***;
3. Se ***y*** non è in **Open-set** ma è in **Closed-set** e il ***costo*** è migliore della stima ***f(y)***, ***y*** viene tolto da **Closed-set** e inserito in **Open-set**, aggiornandone la stima;
4. Altrimenti, pone ***f(y) = costo*** e lo inserisce in **Open-set**.

Algoritmo A*

Dalla descrizione si nota che **A***

- non espande tutti i nodi grigi (in **Open-Set**) ad ogni passo;
- può dover *riconsiderare più volte lo stesso vertice* (passo 3). Nel caso peggiore, tante volte quanti sono i percorso dalla sorgente al vertice;
- se il percorso ottimo consiste di ***k*** archi, non necessariamente tutti i vertici a distanza ***k*** verranno visitati.

Nella pratica è mediamente assai più efficiente della **BFS**.

```
Algoritmo A*(G,source,target)
    f(source) = h(source,target)
    Closed = ∅; found = false
    Open = {source}
    WHILE Open ≠ ∅ AND not found DO
        x = extract-min(Open)
        IF x = target THEN found = true
        ELSE FOR each y ∈ Adj(x) DO
              cost = g(y) + h(y,target)
              IF y ∈ Open AND cost < f(y) THEN
                  Aggiorna_Stima(Open,y,cost) /* f(y)=cost */
              ELSE IF y ∈ Closed AND cost < f(y) THEN
                  Delete(Closed,y)
                  Insert(Open,y, cost)
              ELSE IF y ∉ Closed AND y ∉ Open THEN
                  Insert(Open,y,cost)
              Insert(Closed,x)
    IF found THEN path = Generate-path(source,target)
    ELSE path = NIL
    return path
```

```
Algoritmo A*(G,source,target)
    f(source) = h(source,target)
    Closed = ∅; found = false
    Open = {source}
    WHILE Open ≠ ∅ OR not found DO
        x = extract-min(Open)
        IF x = target THEN found = true
        ELSE FOR each y ∈ Adj(x) DO
                cost = g(y) + h(y,target)
                IF y ∈ Open AND cost < f(y) THEN
                    Aggiorna_Stima(Open,y,cost) /* f(y)=cost */
                ELSE IF y ∈ Closed AND cost < f(y) THEN
                    Delete(Closed,y)
                    Insert(Open,y, cost)
                ELSE IF y ∉ Closed AND y ∉ Open THEN
                    Insert(Open,y,cost)
                Insert(Closed,x)
    IF found THEN path = Generate-path(source,target)
    ELSE path = NIL
    return path
```

Euristica consistente (monotona)

La funzione ***h()*** di stima si dice ***consistente*** (o ***monotona***) se:

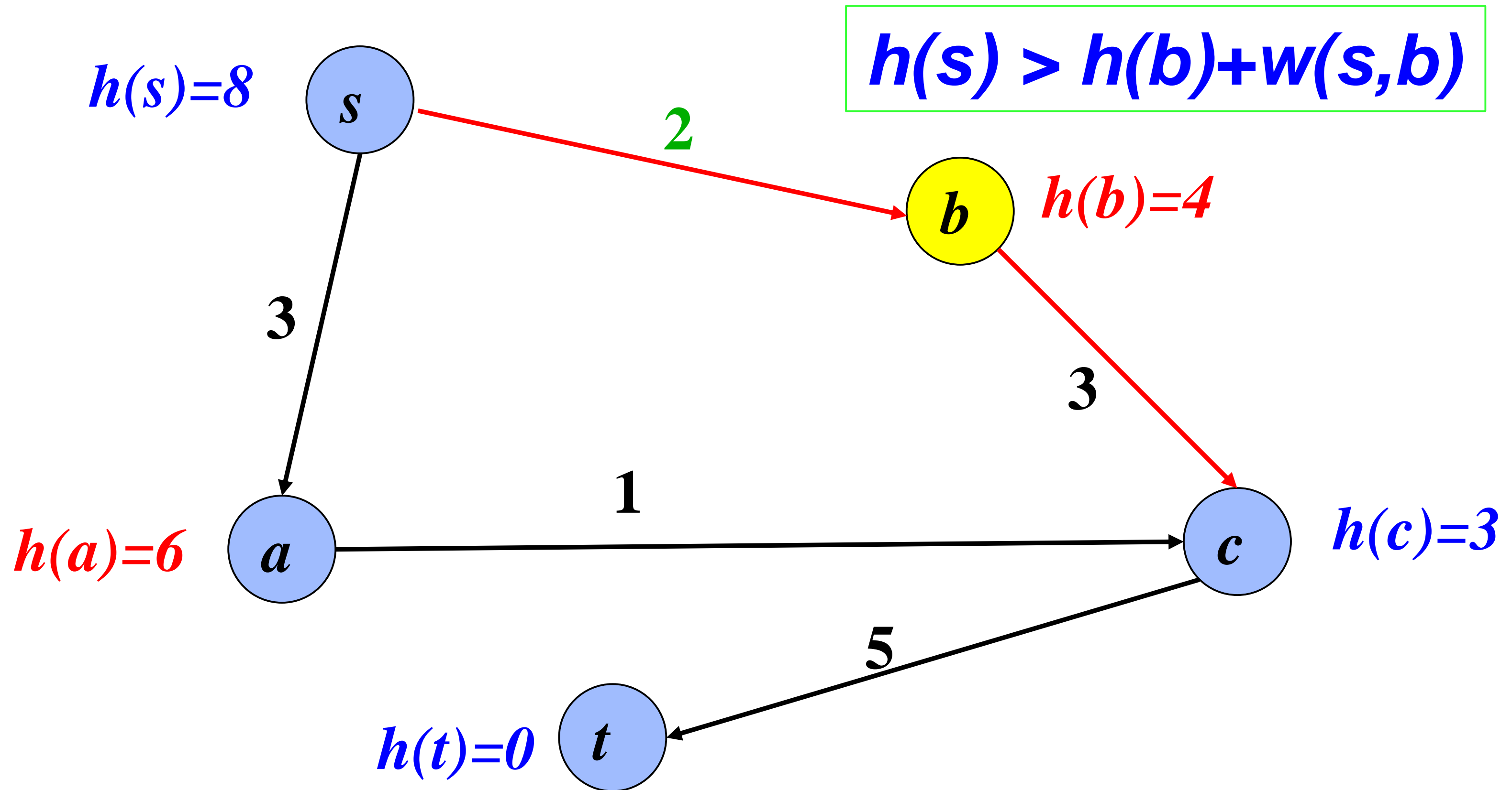
- per ogni arco (***u,v***), il costo stimato ***h(u)*** da ***u*** al target ***t*** non è maggiore del costo dell'arco ***w(u,v)*** più la stima ***h(v)*** da ***v*** a ***t***. Cioè se

$$h(u) \leq h(v) + w(u,v)$$
$$h(t) = 0$$

Ogni ***euristica consistente*** è ammissibile. Inoltre, nodi espansi (**Closed-set**) non dovranno ***mai essere riconsiderati*** (***riespansi***).

Euristica consistente (monotona)

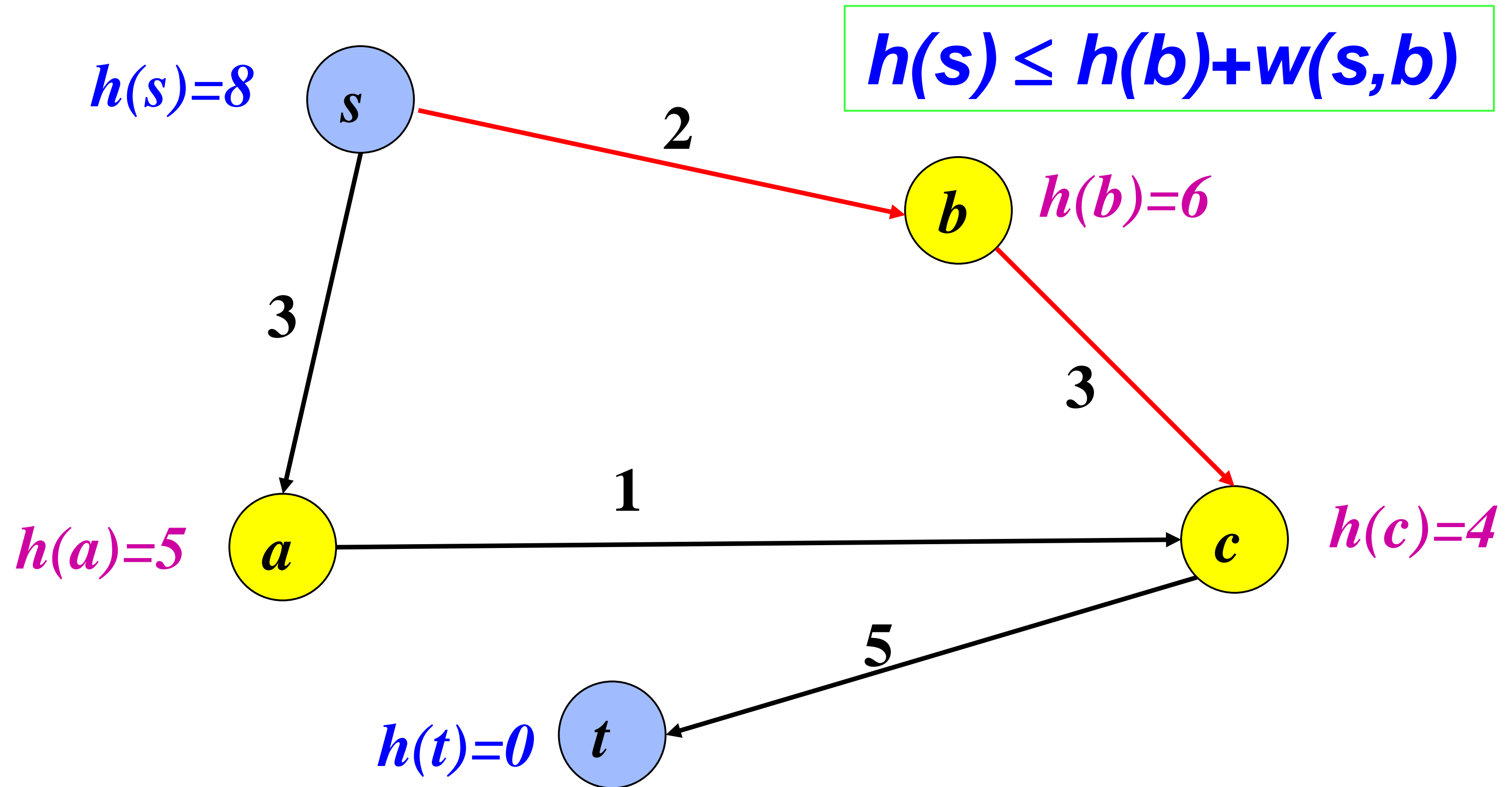
Euristica ***ammissibile*** ma non ***consistente***



Il nodo ***c*** verrà prima espanso lungo il **percorso rosso**
Ma lungo il **percorso verde** la sua stima migliora.
Dovrà, quindi, essere ***riespanso***.

Euristica consistente (monotona)

Euristica ***ammissibile*** e ***consistente***



Ora il nodo ***a*** verrà espanso prima del nodo ***c*** (che sta nella Open) e non dovrà mai essere ***espanso*** una seconda volta.

```
Algoritmo A*(G,source,target)
    f(source) = h(source,target)
    Closed = ∅; found = false
    Open = {source}
    WHILE Open ≠ ∅ OR not found DO
        x = extract-min(Open)
        IF x = target THEN found = true
        ELSE FOR each y ∈ Adj(x) DO
                cost = g(y) + h(y,target)
                IF y ∈ Open AND cost < f(y) THEN
                    Aggiorna_Stima(Open,y,cost) /* f(y)=cost */
                // se h(.) è consistente nessun nodo in
                // Closed verrà mai riconsiderato.
                ELSE IF y ∉ Closed AND y ∉ Open THEN
                    Insert(Open,y,cost)
                Insert(Closed,x)
    IF found THEN path = Generate-path(source,target)
    ELSE path = NIL
    return path
```

Algoritmo A*

Differenze con l'algoritmo di **BFS:**

- **A*** non espande ad ogni passo tutti i nodi grigi (in **Open-Set**), come invece fa **BFS**;
- se ***h(.)*** è ***non consistente***, può *riconsiderare più volte lo stesso vertice* (passo 3). Nel caso peggiore, tante volte quanti sono i percorsi dalla sorgente al vertice (può, quindi, impiegare tempo esponenziale);
- non visita necessariamente tutti i vertici ad una certa distanza ***k***, come invece accade con **BFS**.

Implementare **Open-Set** come ***Coda a Priorità***, per **operazioni di estrazione del minimo e di aggiornamento delle stime efficienti.**

Se la stima è ammissibile (i.e. $h(v) \leq h^*(v)$**)** (con $h^*(v)$ è la distanza reale di v da t), allora **A* è garantito trovare sempre il percorso ottimo.**

Sviluppare una Libreria per Labirinti

Input da File:

1. una descrizione del labirinto come griglia di dimensioni $N \times M$, utilizzando simboli differenti per i muri (es. "**-**" e "**|**") e per i corridoi (es. spazi " ").

2. Una locazione della griglia $\mathbf{s=(x_s,y_s)}$ che rappresenta la posizione di partenza;

3. Una locazione della griglia $\mathbf{t=(x_t,y_t)}$ che rappresenta la posizione di arrivo.

Rappresentare il labirinto:

- come grafo esplicito (matrice e liste di adiacenza);
- come matrice di posizioni (grafo implicito).

Libreria Labirinti

Implementare la ricerca del percorso ottimo per raggiungere una meta ***t*** da una sorgente ***s***, utilizzando:

- Ricerca in ampiezza **BFS**;
- Algoritmo di **Dijkstra**;
- Ricerca euristica con l'algoritmo **A***.

Con entrambe le rappresentazioni del labirinto (grafo esplicito o griglia).

Potete, inoltre, prevedere anche una funzionalità di visualizzazione a terminale del labirinto (usando i caratteri) e del percorso ottimo risultante.