Algoritmi e Strutture Dati

Algoritmi Elementari su Grafi

Maria Rita Di Berardini[2], Emanuela Merelli[1]

[1]Dipartimento di Matematica e Informatica
Università di Camerino

[2]Polo di Scienze
Università di Camerino ad Ascoli Piceno

Parte I

Visite su Grafi

Visità in Profondità (depth-first search, DFS)

La strategia adottata da questo schema di visita consiste nel visitare il grafo sempre più in "profondità" (fin quando è possibile)

Si comincia con il visitare un nodo qualsiasi v che viene marcato come "scoperto"

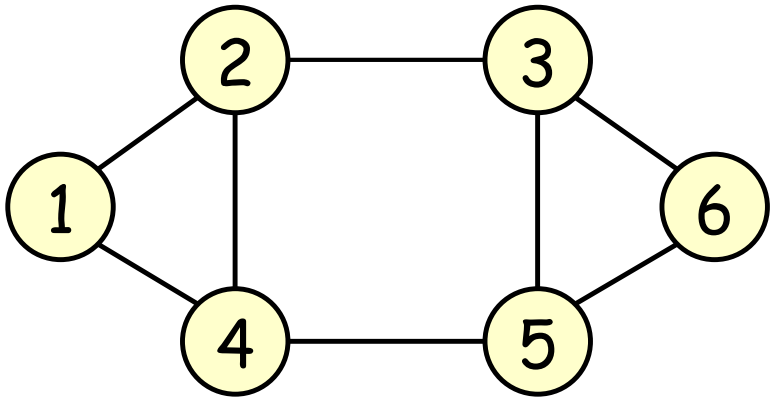
La visita prosegue ispezionando tutti gli archi uscenti dal nodo corrente v alla ricerca di un nodo w collegato ad v e non ancora visitato

Se tale nodo esiste, diventa il prossimo nodo corrente (viene marcato come "scoperto" e si ripente il passo precedente)

Prima o poi, si arriva ad un punto in cui tutti gli archi incidenti il nodo corrente v portano a nodi visitati

Visità in Profondità (depth-first search, DFS)

In questo caso siamo costretti a fare un passo indietro ("backtrack"), tornando al nodo u visitato prima di v; u diventa il nodo corrente e si ripetono i passi precedenti



Lo schema di visita in profondità (a partire dal nodo 1) visita, nell'ordine 1, 2, 3, 6 (backtrack) 5 (backtrack, due volte) 4

Visità in Profondità (depth-first search, DFS)

Proprio come la visita in ampiezza, colora i nodi per distinguere i nodi non ancora scoperti (WHITE), da quelli scoperti ma non espansi (GRAY) e da quelli espansi (BLACK)

Di nuovo ad ogni nodo v viene associato un precedessore denotato con $\pi[v]$, è il nodo che ci consente di raggiungere v durante la visita

Inoltre, ogni nodo v ha associate due informazioni temporali:

- il tempo di scoperta $d[v]$ registra il momento in cui il nodo viene scoperto (e diventa grigio)
- il tempo di completamento $f[v]$ registra il momento in cui la visita completa l'ispezione della lista di adiacenza di v (ed il nodo diventa nero)

Per ogni nodo v abbiamo che $d[v] < f[v]$; inoltre: v è WHITE prima di $d[u]$, è GRAY tra $d[u]$ e $f[u]$, e BLACK successivamente

Visità in Profondità (depth-first search, DFS)

DFS(G)

1. **foreach** nodo $u \in V[G]$
2. **do** $color[u] \leftarrow WHITE$
3. $\pi[u] \leftarrow NIL$
4. $time \leftarrow 0$

▷ time è una variabile globale usata per calcolare i tempi $d[u]$ e $f[u]$

5. **foreach** nodo $u \in V[G]$
6. **do if** $color[u] = WHITE$
7. **then DFS-visit**(u)

N.B: Il risultato della visita potrebbe dipendere dall'ordine con cui i nodi vengono ispezionati nella riga 5

In realtà, queste differenze nell'ordine non causano problemi, perchè tutti i possibili risultati della visita sono sostanzialmente equivalenti

Visità in Profondità (depth-first search, DFS)

DFS-visit(u)

1. $color[u] \leftarrow GRAY$
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. **foreach** nodo $v \in Adj[u]$
5. **do if** $color[v] = WHITE$ ▷ ispezioniamo l'arco (u, v)
6. **then** $\pi[v] \leftarrow u$
7. **DFS-visit**(u)
8. $color[u] \leftarrow BLACK$
9. $time \leftarrow time + 1$
10. $f[u] \leftarrow time$

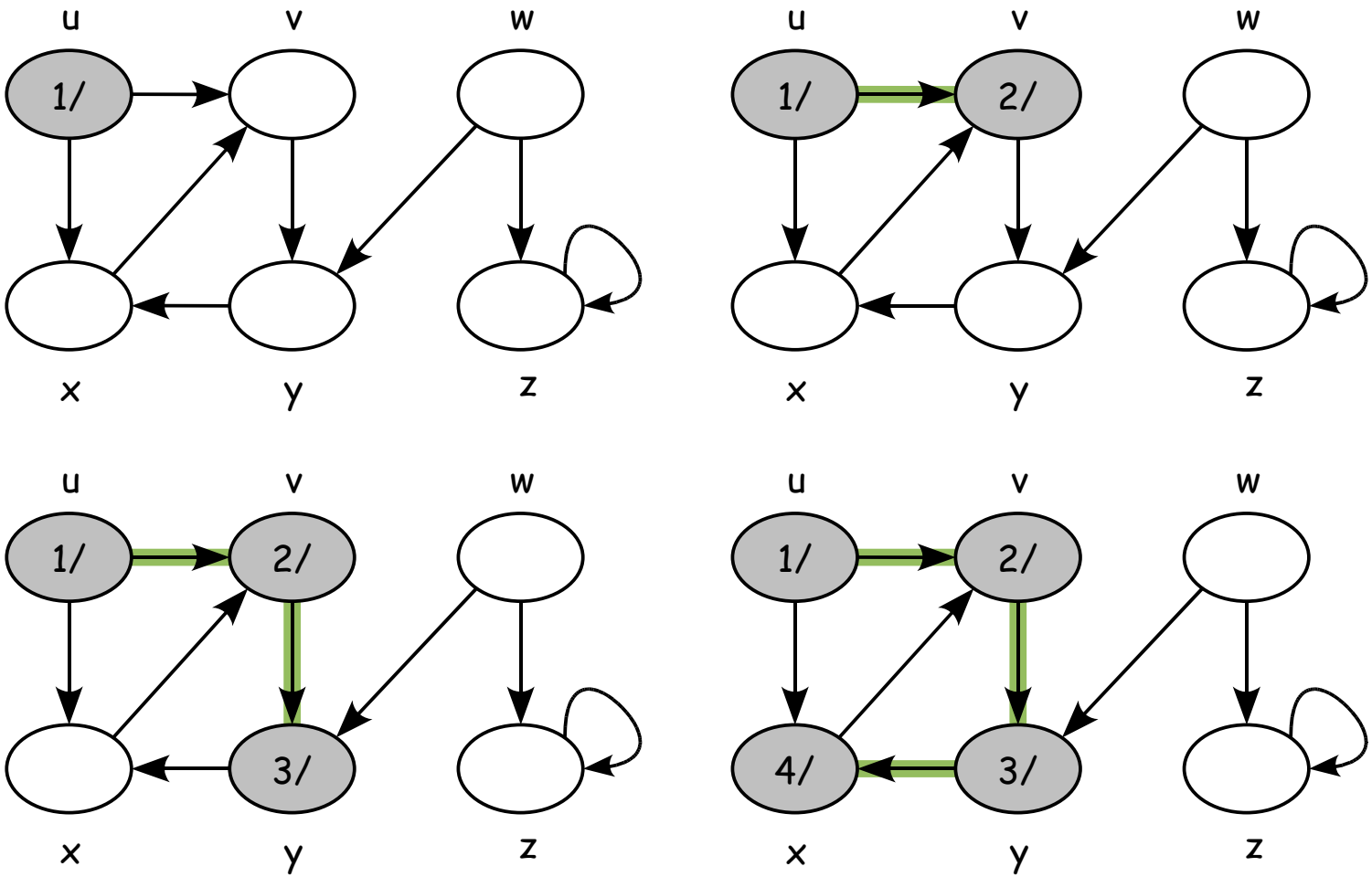
Visità in Profondità (depth-first search, DFS)

Nella fase di inizializzazione, tutti i nodi vengono colorati di bianco ed il loro predecessore viene settato a NIL; la variabile globale *time* viene settato a zero

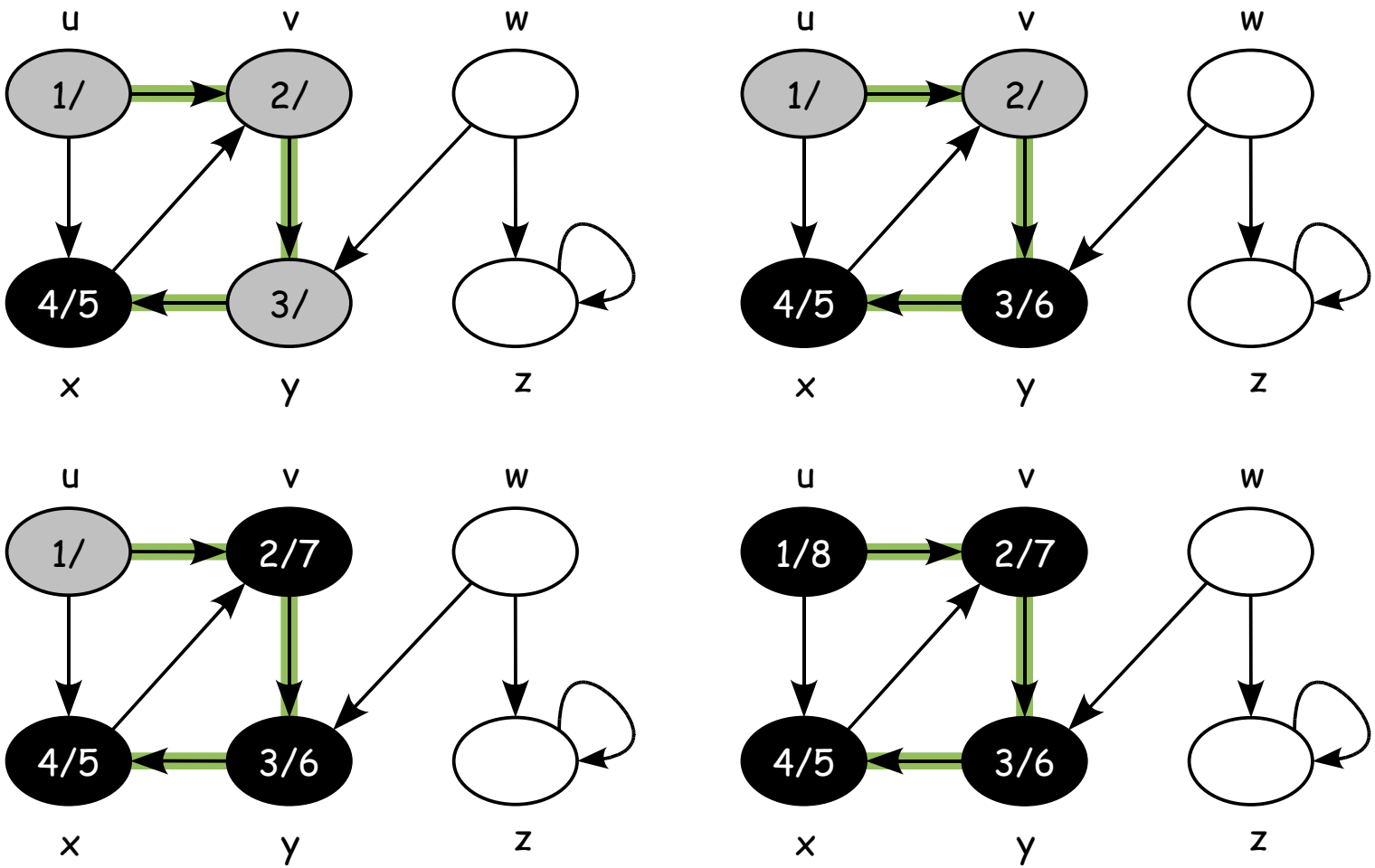
DFS-visit(u) visita (in profondità) tutti i nodi raggiungibili da u

Quando DFS finisce, ogni vertice u ha associato un tempo di scoperta $d[u]$ e un tempo di completamento $f[u]$

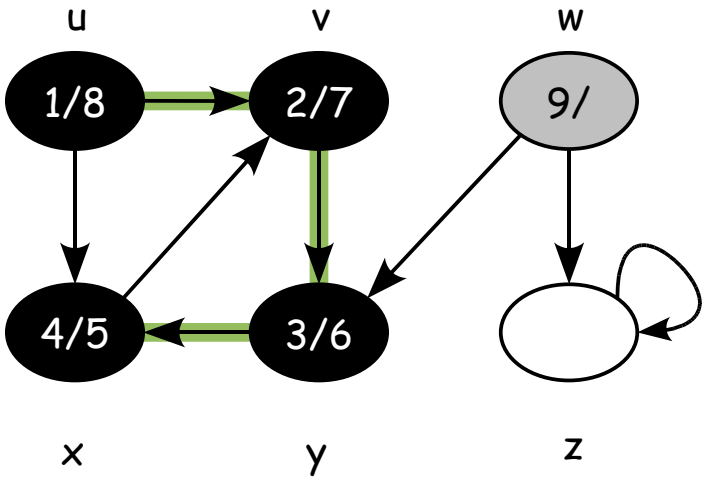
Visità in Profondità – un esempio

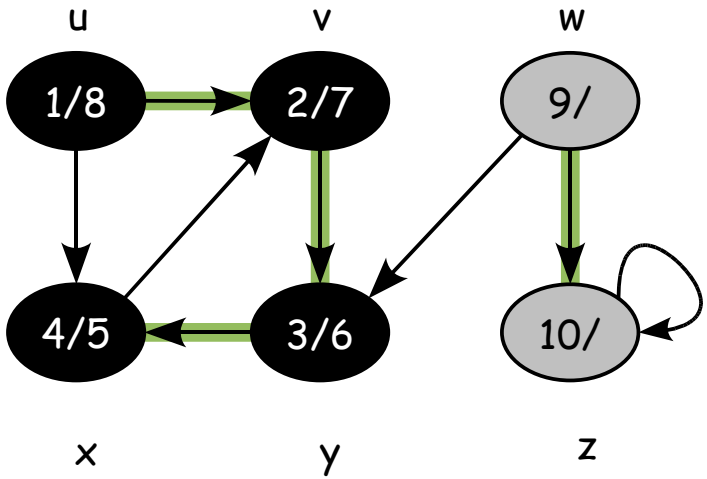


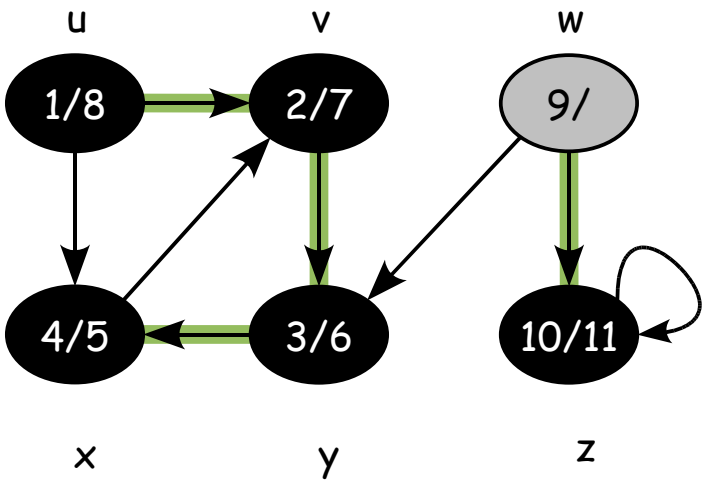
Visità in Profondità – un esempio

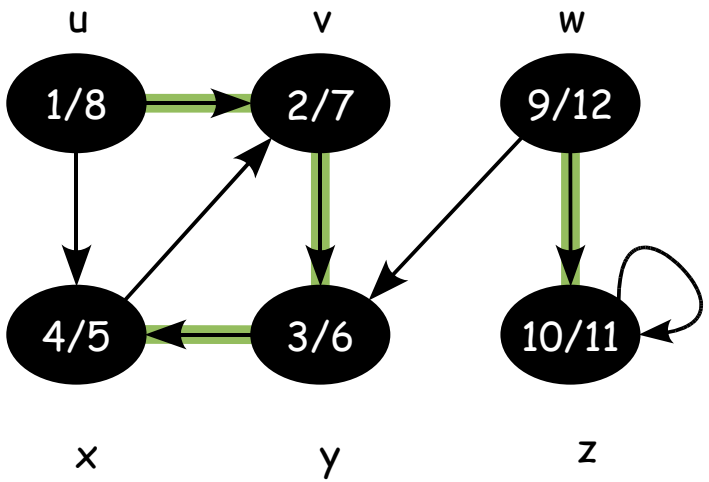


Visità in Profondità – un esempio









Visità in Profondità – complessità

Quale è il tempo di esecuzione di **DFS**? Il ciclo di righe 1-3 impiegano un tempo $O(n)$. Quanto ci costa eseguire ciascun chiamata di **DFS-Visit**?

La procedura **DFS-Visit** viene eseguita esattamente una volta per ogni vertice $v \in V$ (viene invoca solo se il colore è WHITE e la prima cosa che fa è cambiare il colore degli archi in GRAY)

Durante un'esecuzione di **DFS-Visit**, il ciclo delle righe 4-7 viene eseguito $|Adj[u]|$ volte. Poichè

$$\sum_{v \in V} |Adj[u]| = O(m)$$

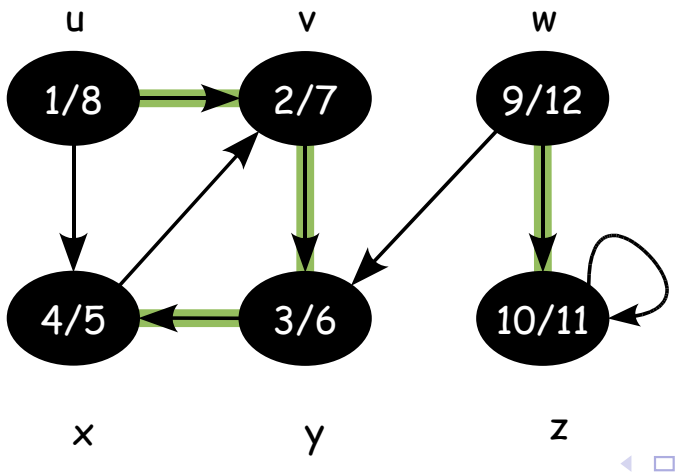
dove $m = |E|$, il costo totale per le chiamate di **DFS-Visit** è $O(m)$.
Quindi il costo totale dell'algoritmo è $O(n + m)$

Sottografo dei predecessori

Il sottografo dei predecessori è definito in modo leggermente diverso da BFS

$$\begin{aligned} G_\pi &= (V, E_\pi) \\ E_\pi &= \{(\pi[v], v) \ : \ v \in V \text{ and } \pi[v] \neq NIL\} \end{aligned}$$

G_π forma una **foresta** depth-first perchè, in generale, è composta da vari alberi depth-first



Parte II

Minimo Albero Ricoprente

Minimo albero ricoprente

Sia $G = (V, E)$ un grafo non orientato, connesso e pesato ($w : E \to \mathbb{R}$ è la funzione peso, dove $\mathbb{R}$ è l'insieme dei numeri reali)

Assumiamo che i nodi rappresentino delle città, un generico arco (u, v) un possibile collegamento tra le città u e u, e $w(u, v)$ il costo della costruzione di tale collegamento

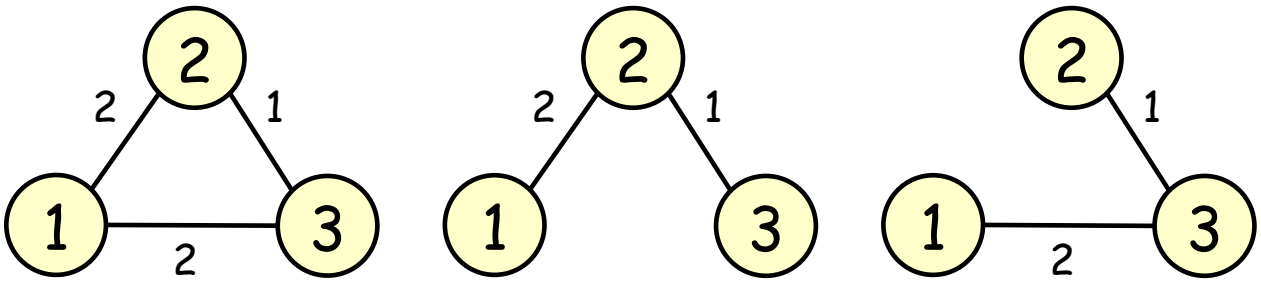
Problema: cercare un insieme T di strade/archi tale che ogni città sia raggiungibile da ogni altra (il sottografo $G' = (V, T)$ è quindi connesso) e tale da minimizzare la somma dei costi di costruzione (la somma dei costi associati agli archi)

Se i pesi associati agli archi sono positivi G' è senza cicli e quindi è un albero (G' è connesso e senza cicli)

Minimo albero ricoprente

Questo problema è noto come problema del **minimo albero ricoprente** (minimum spanning tree, MST)

Si noti che in generale, la soluzione di questo problema non è unica



Esamineremo due algoritmi golosi per risolvere il problema del minimo albero ricoprente: l'algoritmo di **Kruskal** e l'algoritmo di **Prim**

Algoritmo di Kruskal

Algoritmo di Kruskal

Ordina gli archi secondo costi crescenti e costruisce un insieme ottimo di archi T scegliendo di volta in volta un arco di peso minimo che non forma cicli con gli archi già scelti

Per far questo, gestisce una partizione $W = \{W_1, W_2, \ldots, W_k\}$ di V, insieme dei nodi del grafo, in cui ogni W_i rappresenta un insieme di nodi per cui è stato scelto un insieme di archi che li collega

Inizialmente, $T = \emptyset$ e $W = \{\{1\}, \{2\}, \ldots, \{n\}\}$, poichè nessun arco è stato scelto e, quindi, nessun nodo è stato collegato

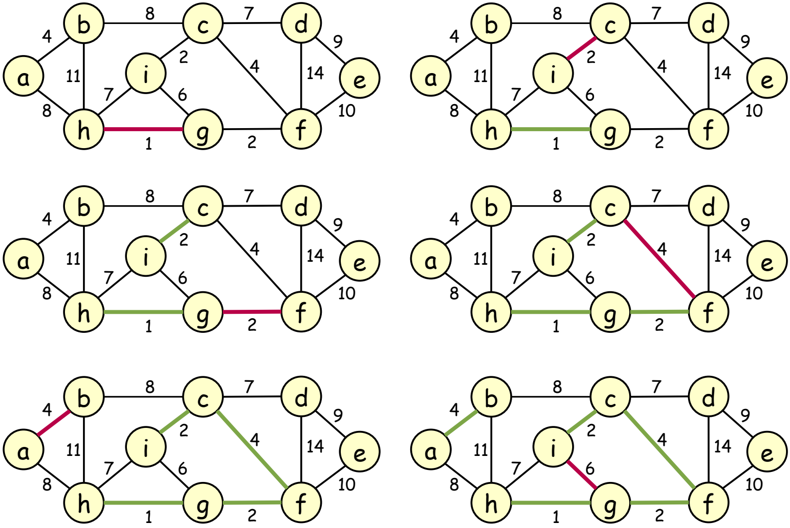
Alla prima iterazione viene scelto il nodo (u, v) di peso minimo; questo viene posto in T (T è vuoto e quindi l'inserimento di (u, v) non può formare cicli) e gli insiemi $\{u\}$ e $\{v\}$ vengono sostituiti con l'insieme $\{u, v\}$

Algoritmo di Kruskal

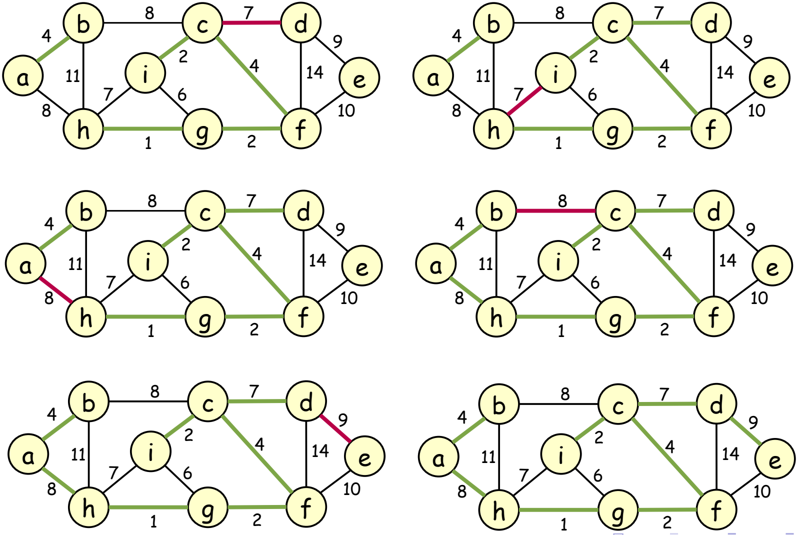
Alla generica iterazione i, esaminiamo l'arco (x, y) con i-esimo costo che viene aggiunto alla soluzione T solo se i nodi x e y non appartengono allo stesso insieme della partizione W (cioè se l'arco non forma cicli con gli archi inseriti in precedenza)

In questo caso, dopo aver inserito l'arco (x, y) in T, si sostituiscono nella partizione W gli insiemi (distinti) contenenti x e y con la loro unione

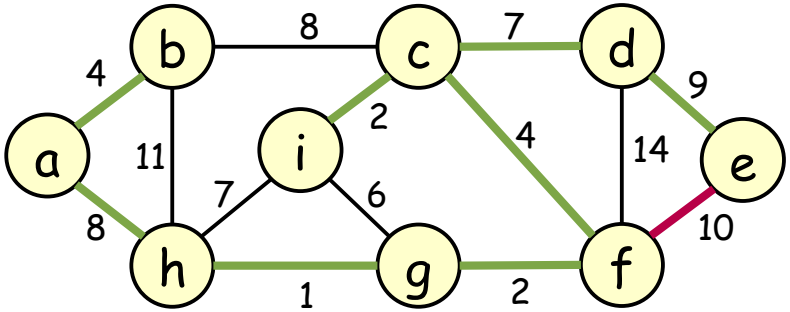
Algoritmo di Kruskal – un esempio

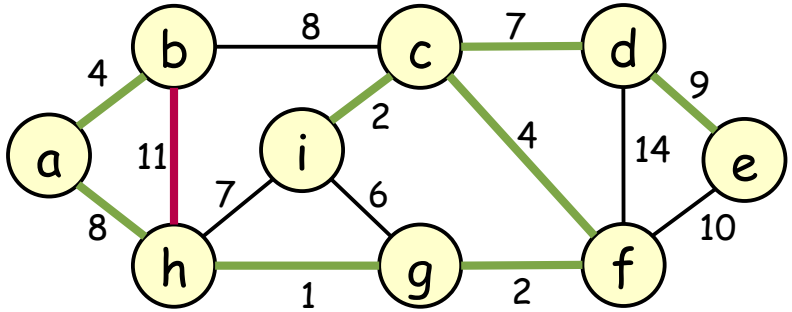


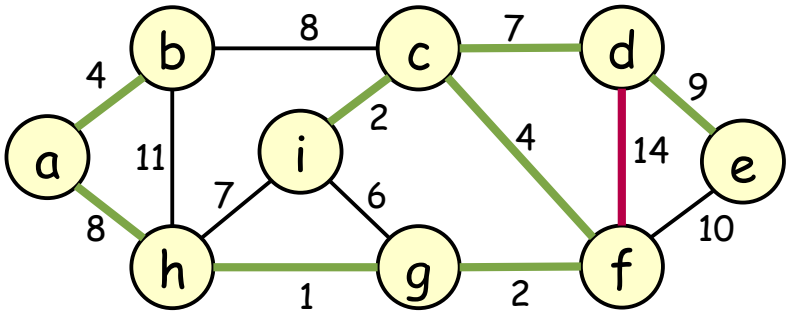
Algoritmo di Kruskal – un esempio



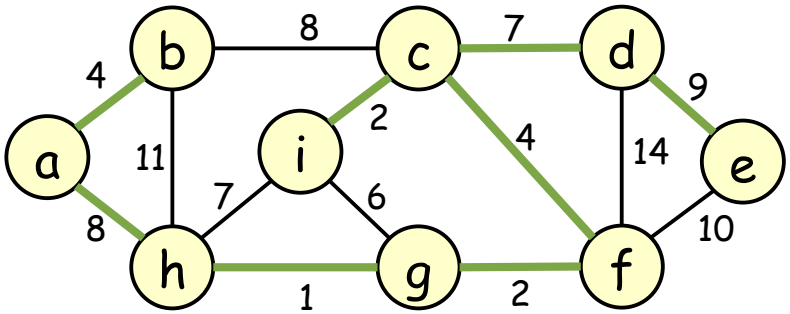
Algoritmo di Kruskal – un esempio







Soluzione ottima



Algoritmo di Kruskal

MST-Kruskal(G, w)

1. $T \leftarrow \emptyset$
2. **foreach** nodo $v \in V[G]$
3. **do Make-Set**(v) ▷ costruisce $W = \{\{1\}, \{2\}, \ldots, \{n\}\}$
4. ordina gli archi di E in senso non decrescente rispetto al peso w
5. **foreach** arco $(u, v) \in E[G]$ preso in ordine non decrescente di peso
6. **do if Find-Set**$(u) \neq$ **Find-Set**(v)
7. **then** $T \leftarrow T \cup \{(u, v)\}$
8. **Union-Set**(u, v)
9. **return** T

dove:

- **Find-Set**(x) restituisce, per ogni nodo x, l'indice di W contenente x
- **Union-Set**(u, v) costruisce l'unione di **Find-Set**(u) e **Find-Set**(v)

Algoritmo di Kruskal – complessità

L'ordinamento di $m = |E|$ archi richiede $O(m \log_2 m)$ passi. Il costo delle m iterazioni del ciclo di riga 5 dipende dal costo delle uniche operazioni non costanti, ossia **Find-Set** e **Union-Set**

In Hopcroft-Ullman viene presentata una struttura dati che rappresenta la partizione W come un insieme di alberi bilanciati

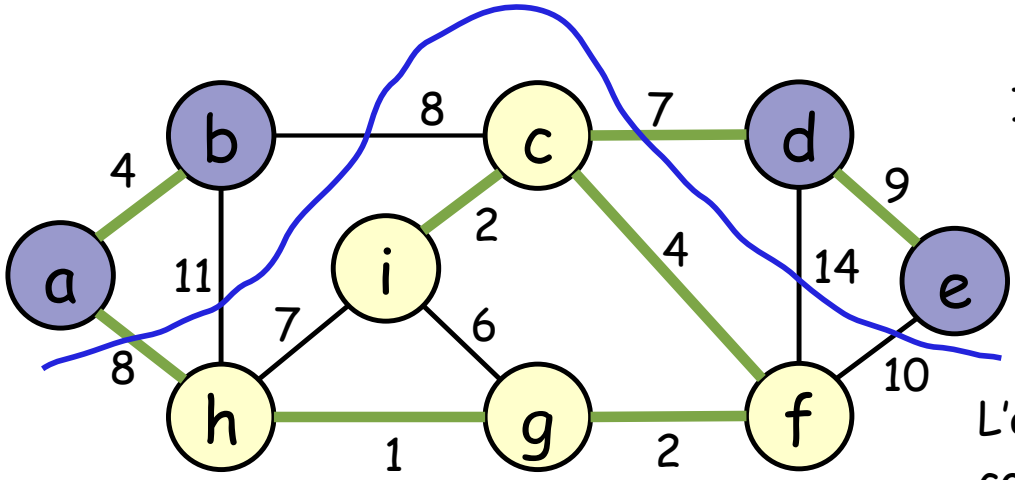
Questa struttura dati permette di eseguire una qualunque sequenza di cm (con c costante) operazioni di unione e ricerca su insiemi aventi complessivamente m elementi, in un tempo $O(m\, g(m))$ dove $g(m)$ è una funzioe tale che $g(m) = O(\log_2 m)$

Possiamo quindi concludere che il costo complessivo dell'algoritmo è $O(m \log_2 m) = O(m \log_2 n^2) = O(m \log_2 n)$ dove $n = |V|$ è il numero di nodi

Algoritmo di Kruskal – correttezza

Sia $G = (V, E)$ un grafo non orientato, non connesso e pesato:

1. un **taglio** di G è una partizione $(S, \overline{S} = V - S)$ dell'insieme V dei nodi
2. diciamo che un arco (u, v) **attraversa** un taglio $(S, \overline{S})$ se una delle due estremità si trova in S e l'altra in $\overline{S}$



Il taglio (S={a, b, d, e}, V-S) è attraversato dagli archi (a,h), (b, h), (b, c), (c, d), (f, d) e (f, e)

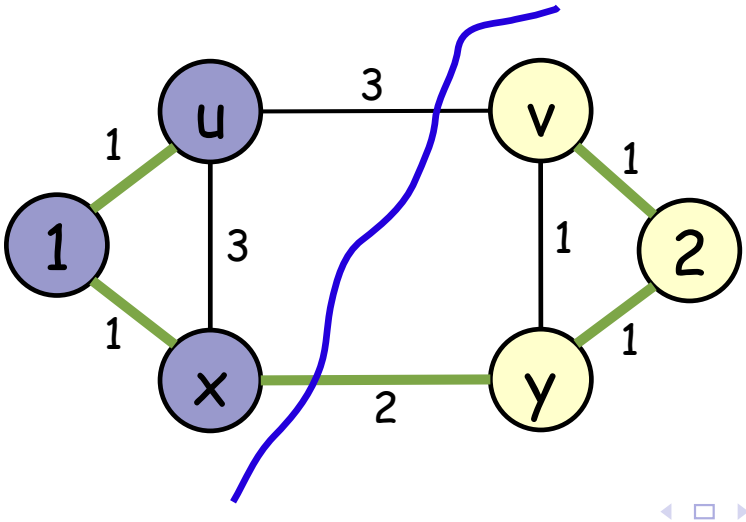
L'arco che attraversa il taglio di costo minimo – **arco leggero** – è l'arco (c,d)

Algoritmo di Kruskal – correttezza

Teorema 1: sia

- $G = (V, E)$ un grafo non orientato, non connesso e pesato;
- $(S, \overline{S})$ un taglio di G;
- T un'insieme di archi corrispondenti ad una soluzione ottima, ossia T è tale che $G' = (V, T)$ è un MST.

Un arco (u, v) di peso minino fra tutti quelli che attraversano $(S, \overline{S})$ (un arco leggero) fa parte della soluzione ottima T



Algoritmo di Kruskal – correttezza

Proviamo la correttezza dell'algoritmo di Kruskal mostrando come, ad ogni iterazione, questo algoritmo inserisce in T un "pezzo" di soluzione ottima

Sia T l'insieme di archi costruito da Kruskal appena prima di inserire l'arco (u, v). Allora:

- gli archi in T partizionano il grafo G in una o più componenti connesse

  ogni elemento W_i della partizione $W = \{W_1, W_2, \ldots, W_k\}$ è un insieme di nodi per cui è stato scelto un'insieme di archi che li collega, ossia, è una componente connessa del grafo
- inoltre, i nodi u e v appartengo a due componenti diverse e questo perchè **Find-Set**$(u) \neq$ **Find-Set**(v))

Consideriamo ora il taglio che $(W_i, \overline{W_i} = V - W_i)$ dove W_i è l'elemento di W tale che $u \in W_i$. L'arco (u, v) (ossia il prossimo arco scelto da Kruskal) è un arco di peso minimo che attraversa il taglio $(W_i, \overline{W_i})$. Per il Teorema 1, (u, v) è parte di una soluzione ottima

Algoritmo di Prim

Algoritmo di Prim

È essenzialmente un algoritmo di visita che, partendo da un nodo iniziale u (scelto arbitrariamente) esamina tutti i nodi del grafo

Analogamente agli algoritmi di visita, ad ogni iterazione, partendo da un nodo v, visita un nuovo nodo w (scelto secondo opportuni criteri) e pone l'arco (v, w) nell'insieme T che, al termine dell'esecuzione, conterrà una soluzione ottima

La differenza sostanziale rispetto ad un algoritmo di visita "standard", è che la scelta del prossimo nodo da visitare viene fatta introducendo un concetto di **priorità tra nodi** e che l'insieme Q dei nodi da visitare viene gestito come una **coda di priorità**

Se R è l'insieme dei nodi visitati, la priorità di un nodo $v \in Q$ è pari al minimo peso di un arco che collega v con un nodo in R

Algoritmo di Prim

Durante la generica iterazione, l'algoritmo estrae dalla coda Q dei nodi da visitare, il nodo v con priorità minima e inserisce in T (la soluzione corrente) l'arco (u, v) che risulta di peso minimo tra tutti quelli che collegano v con un nodo in R

L'iterazione termina aggiornando le priorità dei nodi non visitati collegati con v

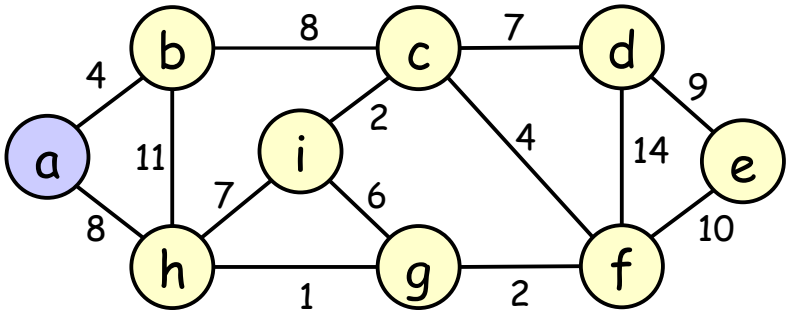
L'algoritmo di Prim

MST-Prim(G, w, r)

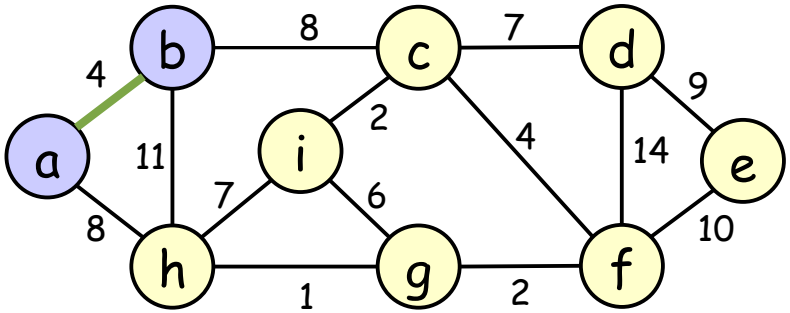
▷ w è la funzione peso ed r è il nodo da cui parte la visita

1. **foreach** nodo $v \in V[G]$
2. **do** $key[v] = \infty$ ▷ setta le priorità iniziali
3. $\pi[v] = NIL$ ▷ inizializza il vettore dei predecessori
4. $key[r] \leftarrow 0$
5. $Q \leftarrow V[G]$
6. **while** $Q \neq \emptyset$
7. **do** $u \leftarrow$ **Extract-Min**(Q)
8. **for each** $v \in Adj[u]$
9. **do if** $v \in Q$ **and** $w(u, v) < key[v]$
10. **then** $\pi[v] \leftarrow u$
11. $key[v] \leftarrow w(u, v)$

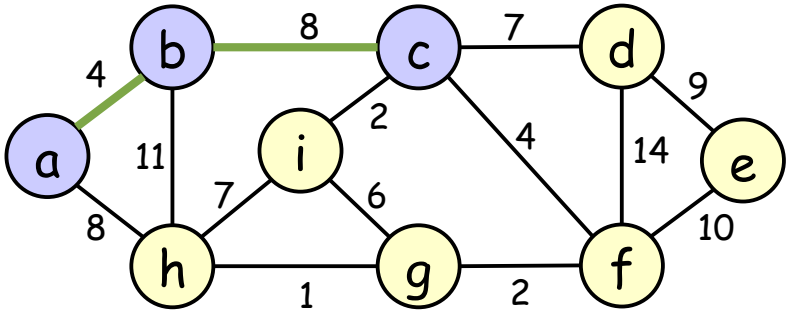
Algoritmo di Prim – un esempio



a	b	c	d	e	f	g	h	i
0	∞	∞	∞	∞	∞	∞	∞	∞
	4	∞	∞	∞	∞	∞	8	∞

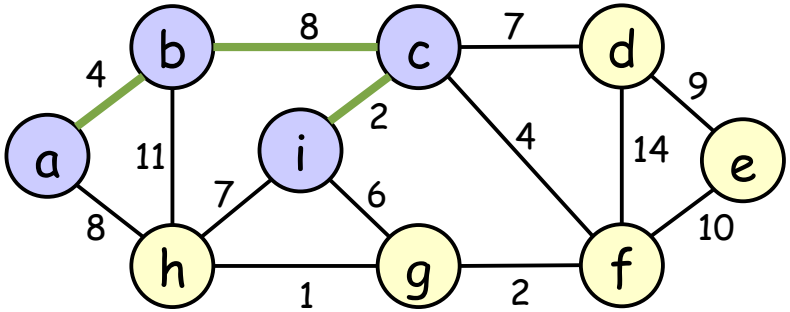


a	b	c	d	e	f	g	h	i
		8	∞	∞	∞	∞	8	∞

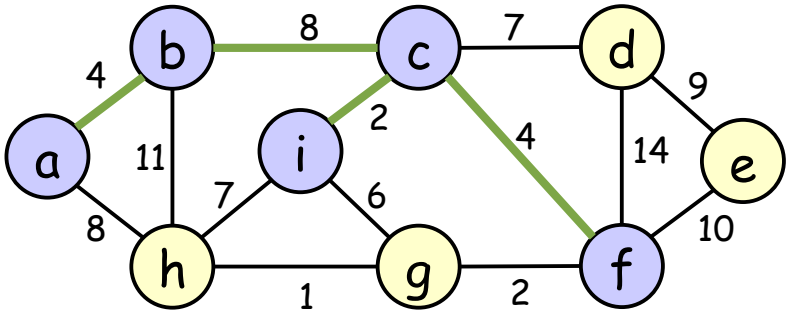


a	b	c	d	e	f	g	h	i
			7	∞	4	∞	8	2

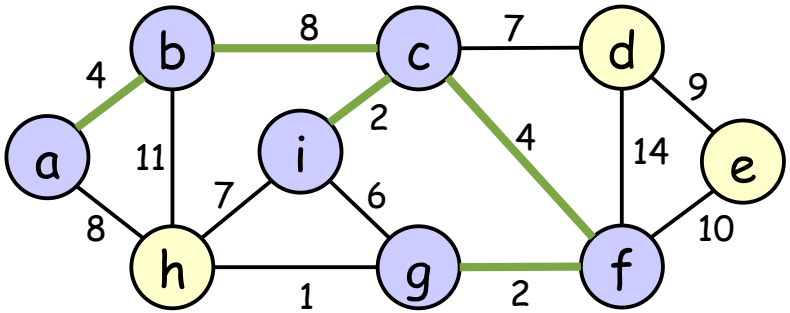
Algoritmo di Prim – un esempio



a	b	c	d	e	f	g	h	i
			7	∞	4	6	7	

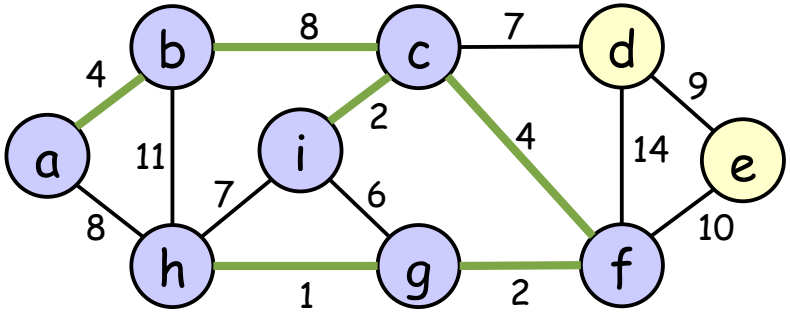


a	b	c	d	e	f	g	h	i
			7	10		2	7	

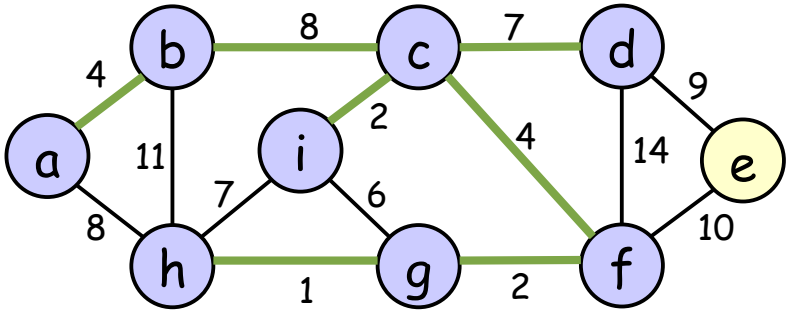


a	b	c	d	e	f	g	h	i
			7	10			7	

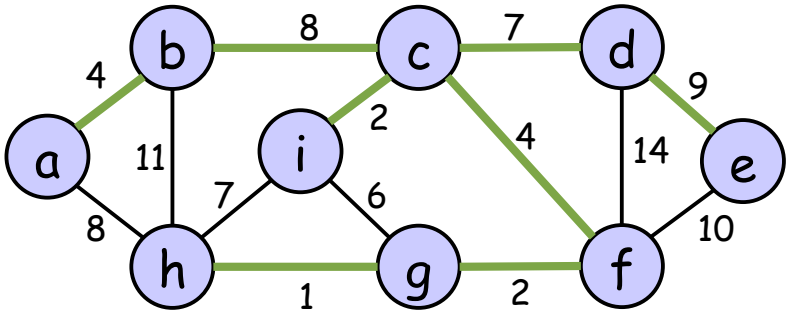
Algoritmo di Prim – un esempio



a	b	c	d	e	f	g	h	i
			7	10				



a	b	c	d	e	f	g	h	i
				9				



a	b	c	d	e	f	g	h	i

Algoritmo di Prim – Complessità

La fase di inizializzazione (righe 1-5) può essere eseguita in un tempo $O(n)$, dove $n = |V|$

Il costo del ciclo **while** (riga 6) (ciclo ripetuto n volte) è dato da:

- il costo di estrazione del minimo $\Rightarrow$ se Q è implementata come un min-heap questa operazione può essere implementata con un costo pari a $O(\log_2 n)$

  $+$

- costo del ciclo for (righe 8-11)

Algoritmo di Prim – Complessità

Per ogni $u \in Q$, il ciclo for (righe 8-11) viene eseguito α_u volte, dove con α_u abbiamo indicato la lunghezza della lista di adiacenza di u. Inoltre:

- Il test del di verifica di appartenenza a Q può essere fatto in tempo costante (mantenendo un bit per ogni vertice che dica se quel nodo appartiene o meno a Q – bit che viene aggiornato quando togliamo un nodo da Q)
- l'assegnamento di riga 11 richiede implicitamente un operazione **Decrease-Key** che può essere implementata (usando di nuovo un min-heap) in un tempo $O(log_2 n)$

Il costo di ogni iterazione ciclo for è $O(\alpha_u \log_2 n)$

Algoritmo di Prim – Complessità

La fase di inizializzazione (righe 1-5) può essere eseguita in un tempo $O(n)$, dove $n = |V|$

Il costo del ciclo **while** (riga 6) è dato da:

- il costo di estrazione del minimo $\Rightarrow$ se Q è implementata come un min-heap questa operazione può essere implementata con un costo pari a $O(\log_2 n)$

  $+$

- costo del ciclo for (righe 8-11) $= O(\alpha_u \log_2 n)$

Riassumendo, il costo del ciclo while $\leq \sum_{u \in V} O(\alpha_u \log_2 n + \log_2 n)$

Algoritmo di Prim – Complessità

Il costo del ciclo while

$$\sum_{u \in V} O(\alpha_u \log_2 n + \log_2 n) = O(\sum_{u \in V} \alpha_u \log_2 n + \log_2 n) =$$

$$O(\log_2 n \sum_{u \in V} (\alpha_u + 1)) = O\big(\log_2 n \, (\sum_{u \in V} \alpha_u + n)\big) =$$

$$O\big(\log_2 n \, (2m + n)\big) = O\big(\log_2 n \, (m + n)\big)$$

Il costo complessivo dell'algoritmo è

$$O\big(\log_2 n \, (m + n) + n\big) = O(\log_2 n \, (m + n)) = O(m \log_2 n)$$

Parte III

Cammini Minimi

Definizione e proprietà

Sia $G = (V, E)$ un grafo **orientato** e pesato, con una funzione di peso $w : E \to \mathbb{R}$, definiamo:

- il **peso di un cammino** $p = \langle v_0, v_1, \ldots, v_n \rangle$ come la somma dei pesi degli archi che lo compongono

$$w(p) = \sum_{i=1}^{n} w(v_{i-1}, v_i)$$

- il **peso di un cammino minino** da u a v come

$$\delta(u, v) = \begin{cases} \min\{w(p) \ : \ u \overset{p}{\rightsquigarrow} v\} & \text{se esiste un cammino da } u \text{ a } v \\ \infty & \text{altrimenti} \end{cases}$$

Esamineremo il problema dei cammini minimi da sorgente unica: trovare il cammino minimo da un nodo **sorgente** s a ciascun nodo $v \in V$

Varianti

Cammino minimo con destinazione unica: trovare un cammino minimo da ciascun nodo v ad un dato nodo **destinazione** t (può essere ricondotto al problema del cammino minimo da sorgente unica invertendo il senso degli archi)

Cammino minimo per una coppia di nodi: trovare un cammino minimo da u a v, noti i vertici u e v (basta risolvere il problema dei cammini minimi con sorgente u)

Cammino minimo per ogni coppia di nodi: trovare un cammino minimo da u a v per ogni coppia di nodi u e v (basta risolvere il problema dei cammini minimi per ogni nodo u del grafo)

Sottostruttura ottima del problema del cammino minimo

Dati un grafo **orientato** e pesato $G = (V, E)$, con una funzione di peso $w : E \to \mathbb{R}$, sia $p = \langle v_0, v_1, \ldots, v_k \rangle$ un cammino minimo tra i nodi v_1 e v_k

Per ogni $1 \leq i \leq j \leq k$, il sottocammino $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ è un **cammino minimo** da v_i a v_j

Proof: se scomponiamo il cammino p in $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ abbiamo che $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$

Assumiamo ora che esista un cammino p'_{ij} con peso $w(p'_{ij}) < w(p_{ij})$ (e quindi che $w(p_{ij})$ non sia ottimo)

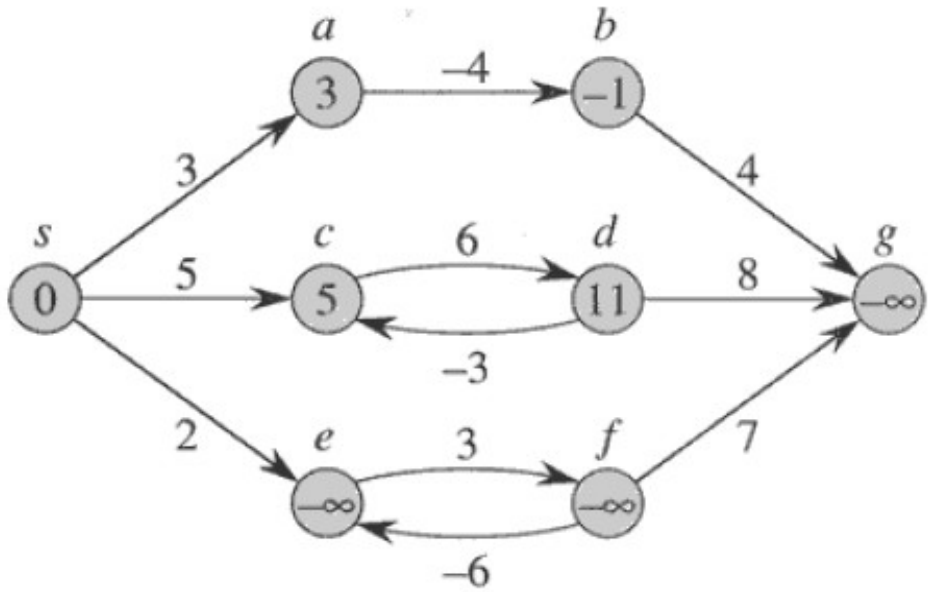
Allora il cammino $p' = v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ ha peso

$$w(p') = w(p_{1i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$$

il che contraddice il fatto che p è ottimo

Archi di peso negativo

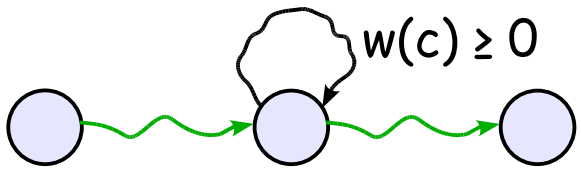
La funzione w può assegnare dei pesi negativi agli archi; tuttavia, nel caso in cui dalla sorgente s è possibile raggiungere un ciclo con peso negativo (il peso di un ciclo è la somma dei pesi dei suoi archi), il peso dei cammini minimi non è ben definito



Cammini minimi e cicli

Un cammino minimo può contenere dei cicli? Abbiamo appena visto che non può contenere cicli con peso negativi

Non può neanche contenere cicli con pesi positivi, altrimenti non sarebbe minimo



Infine è sempre possibile eliminare cicli di peso 0 senza alterare il peso del caamini, che se era minimo rimane minimo

Possiamo assumere che un cammino minimo non contenga cicli; quindi un cammino minimo in un grafo $G = (V, E)$ può avere al più $n - 1$ (dove $n = |V|$) archi

Rappresentazione dei cammini minimi

Spesso è utile calcolare non solo i cammini minimi ma anche i vertici che compongono tali cammini

Dato $G = (V, E)$, manteniamo per ogni vertice v un **predecessore** $\pi[v]$ che pò essere un altro nodo o NIL

L'**algoritmo di Dijkstra** (algoritmo per il calcolo dei cammini minimi da sorgente unica a che studieremo) gestisce π in maniera t.c. la catena dei predecessori che parte da v ripercorra all'indietro il cammino minimo da s a v

L'**albero dei predecessori** $G_\pi = (V_\pi, E_\pi)$ indotto da π è definito come :

$$V_\pi = \{v \in V \ : \ \pi[v] \neq NIL\} \cup \{s\}$$
$$E_\pi = \{(\pi[v], v) \in E \ : \ v \in V_\pi - \{s\}\}$$

G_π è tale che ogni cammino da s a v in G_π corrisponde ad un cammino minimo s a v in G

Rilassamento

L'algoritmo di Dijkstra usa una tecnica che prende il nome di tecnica del **rilassamento**

Questa tecnica consiste nel memorizzare, per ogni nodo v del grafo, un attributo $d[u]$ (**stima del cammino minimo**) che rappresenta un limite superiore al peso del cammino da s a v (limite che viene rilassato ogni volta che viene trovata una strada "migliore")

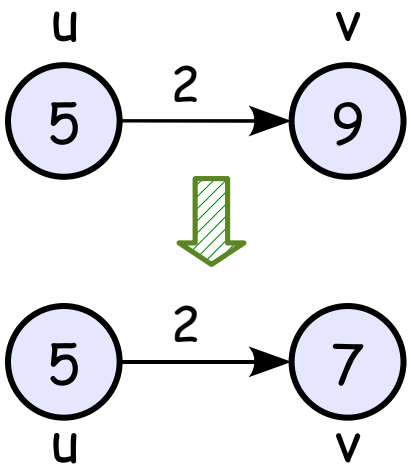
La seguente procedura inizializza le stime dei cammini minimi e il vettore dei predecessori in un tempo $O(n)$ (n è sempre il numero dei nodi)

Initialize-Single-Source(G, s)

1. **for each** $v \in V[G]$
2. **do** $d[u] \leftarrow \infty$
3. $\pi[v] \leftarrow NIL$
4. $d[s] \leftarrow 0$

Rilassamento

Il **processo di rilassamento** di un arco (u, v) consiste nel verificare se passando per u è possibile migliorare la stima del cammino minimo per v e in caso affermativo nell'aggiornare i valori di $d[u]$ e $\pi[u]$



Relax(u, v, w)

1. **if** $d[v] > d[u] + w(u, v)$
2. **then** $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$

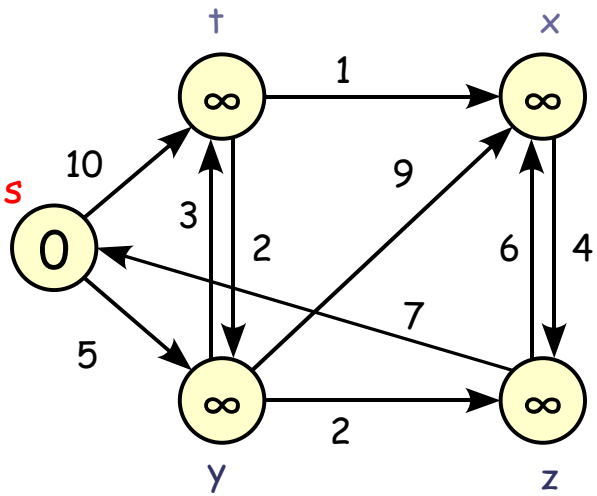
Algoritmo di Dijkstra

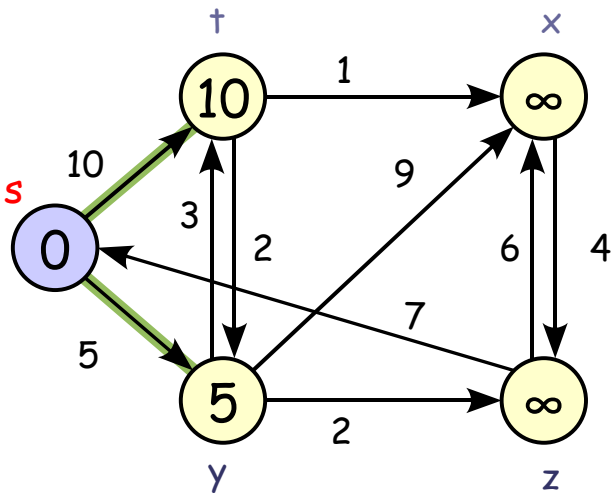
Risolve il problema dei cammini minimi a sorgente unica (indicata con s) in un $G = (V, E)$ orientato e con pesi sugli archi non negativi (per ogni arco $(u, v) \in E$, il peso $w(u, v) \geq 0$)

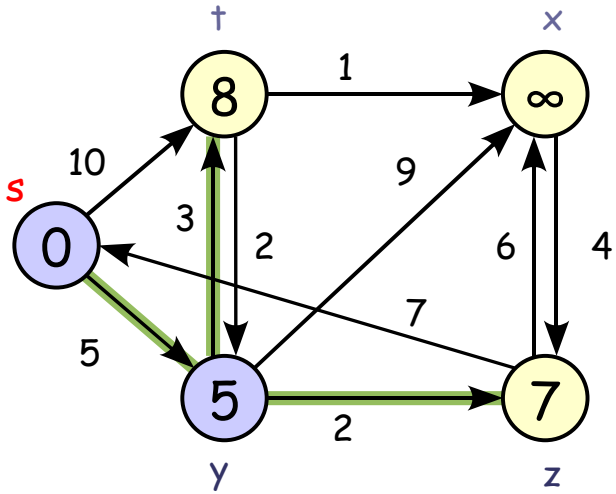
Mantiene un insieme di nodi S i cui pesi dei cammini minimi da s sono già stati determinati

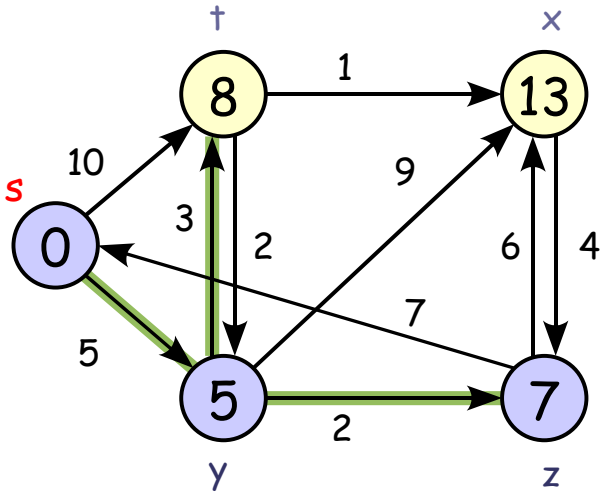
Seleziona ripetutamente il nodo in $u \in V - S$ con la minima stima del cammino minimo (Dijkstra è un algoritmo goloso) e rilassa tutti gli archi che escono da u

Algoritmo di Dijkstra – un esempio

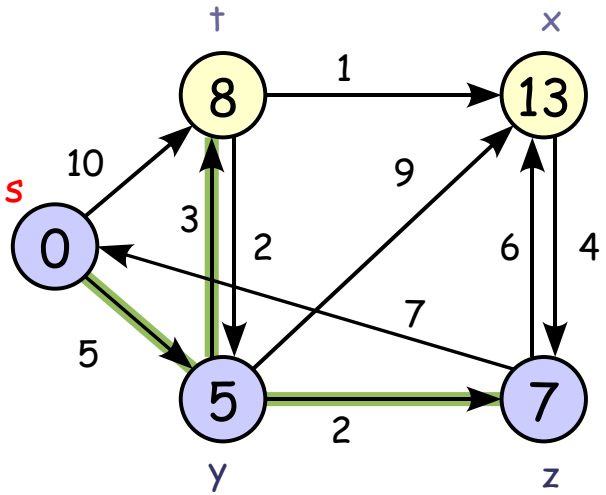


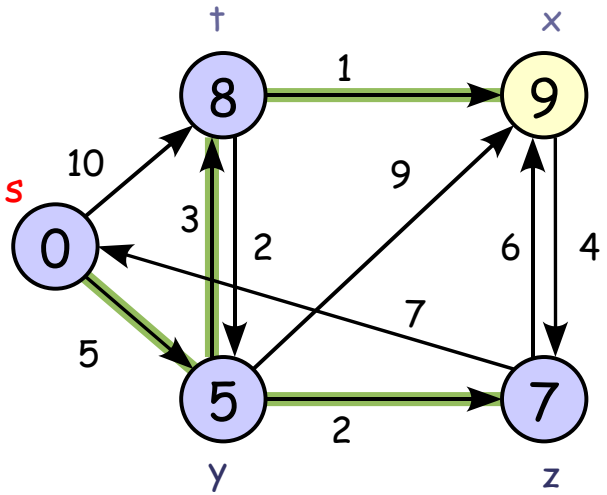


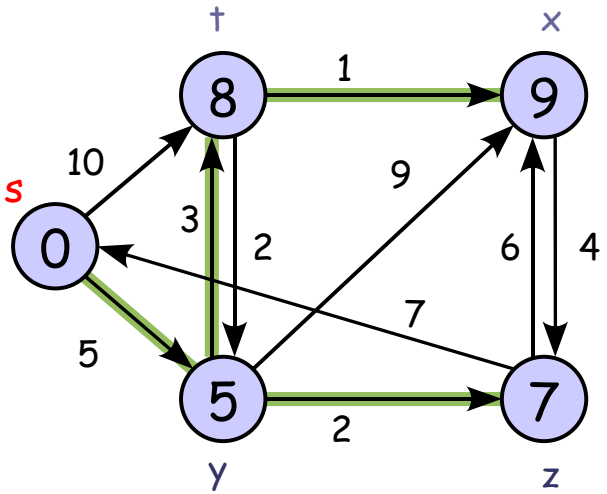




Algoritmo di Dijkstra – un esempio







Algoritmo di Dijkstra

Risolve il problema dei cammini minimi a sorgente unica (indicata con s) in un $G = (V, E)$ orientato e con pesi sugli archi non negativi (per ogni arco $(u, v) \in E$, il peso $w(u, v) \geq 0$)

Mantiene un insieme di nodi S i cui pesi dei cammini minimi da s sono già stati determinati

Seleziona ripetutamente il nodo in $u \in V - S$ con la minima stima del cammino minimo (Dijkstra è un algoritmo goloso) e rilassa tutti gli archi che escono da u

L'insieme Q dei nodi non in S ($Q = V - S$) viene gestito come una coda di min-priorità, utilizzando i valori di d come chiavi

Algoritmo di Dijkstra – complessità

Dijkstra(G, w, s)

1. **Initialize-Single-Source**(G, s)
2. $S \leftarrow \emptyset$
3. $Q \leftarrow V[G]$
4. **while** $Q \neq \emptyset$
5. **do** $u \leftarrow$ **Extract-Min**(Q)
6. $S \leftarrow S \cup \{u\}$
7. **foreach** $v \in Adj[u]$
8. **do Relax**(u, v, w)

Algoritmo di Dijkstra – complessità

Poichè l'insieme Q viene gestito come una coda di priorità, il costo di **Dijkstra** dipende da come implementiamo la coda e quindi dal costo delle operazioni **Insert** (implicita nella riga 3), **Extract-Min** (riga 5) e **Decrease-Key** (implicita in **Relax** di riga 8)

Assumiamo che la coda viene implementata semplicemente come un array di n elementi:

- **Insert** e **Decrease-Key** hanno un costo costante – $O(1)$
- **Extract-Min** ha un costo $O(n)$ – possiamo solo scandire l'array

Algoritmo di Dijkstra – complessità

Ora, osserviamo che il ciclo **while** di riga 4 viene eseguito esattamente n volte; inoltre:

- ciascuna iterazione ha un costo dato dalla chiamata di **Extract-Min** più il costo della scansione della lista di adiacenza del nodo u appena inserito in S
- poichè ogni nodo viene inserito in S solo una volta, ogni lista di adiacenza viene scandita esattamente una volta

Il costo del ciclo while è dato dal costo del n di **Extract-Min** più il costo della scansione delle liste, ossia $O(n^2 + m) = O(n^2)$

A questo costo va aggiunto il costo della fase di inizializzazione di riga 1; quindi il tempo richiesto dall'algoritmo è dell'ordine di

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