

Algoritmi e Strutture Dati

Capitolo 13

Cammini minimi:

Algoritmo di Dijkstra (*)

(ACM in **grafi diretti e non diretti senza archi di peso negativo**)

Punto della situazione

- **Algoritmo basato sull'ordinamento topologico:** albero dei cammini minimi in grafi diretti **aciclici**. Complessità $\Theta(n+m)$ (grafo rappresentato con liste di adiacenza).
- **Algoritmo di Bellman&Ford:** albero dei cammini minimi in grafi diretti che **non contengono cicli negativi**. Complessità $\Theta(n \cdot m)$ (grafo rappresentato con liste di adiacenza).
- **Algoritmo di Floyd e Warshall:** cammini minimi tra tutte le coppie di nodi in grafi diretti che **non contengono cicli negativi**. Complessità $\Theta(n^3)$ (sia con liste di adiacenza che con matrice di adiacenza)

Algoritmo di Dijkstra (1959)

(albero dei cammini minimi
in grafi (sia diretti che non diretti) con **pesi
non negativi**)

Richiamo sulla notazione

$w(u,v)$: peso dell'arco (u,v) del grafo pesato (diretto o non diretto)

$$G=(V,E,w)$$

$w(\pi)$: lunghezza di un cammino π nel grafo, ovvero somma dei pesi di tutti gli archi del cammino

π_{uv} : cammino **minimo** nel grafo tra i nodi u e v

d_{uv} : **distanza** nel grafo (ovvero lunghezza del cammino minimo) tra i nodi u e v

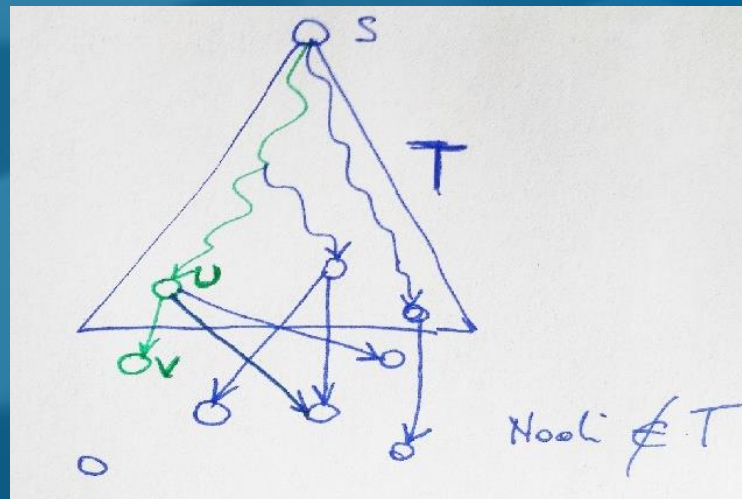
D_{uv} : **sovrastima** della distanza nel grafo tra i nodi u e v (cioè $D_{uv} \geq d_{uv}$)

Estendere l'albero dei cammini minimi

Lemma di Dijkstra (1959): Sia $G=(V,E,w)$ (diretto o non diretto) con pesi **non negativi**, e sia T un **sottoalbero** dell'albero dei cammini minimi radicato in s che include s ma non include tutti i vertici raggiungibili da s . Sia:

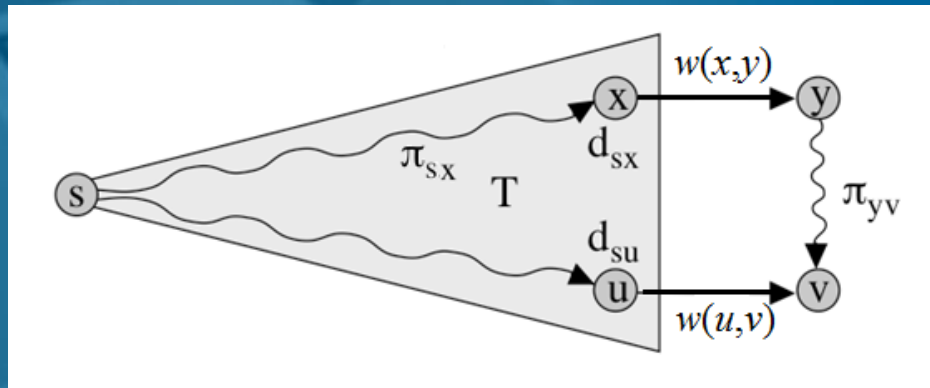
$$(u,v) = \arg \min \{ d_{st} + w(t,z) \mid (t,z) \in E, t \in T \text{ e } z \notin T \}$$

(in altre parole, v è il nodo **non appartenente** a T più vicino ad s). Allora, (u,v) appartiene a un cammino minimo da s a v .



Prova del lemma di Dijkstra (1/2)

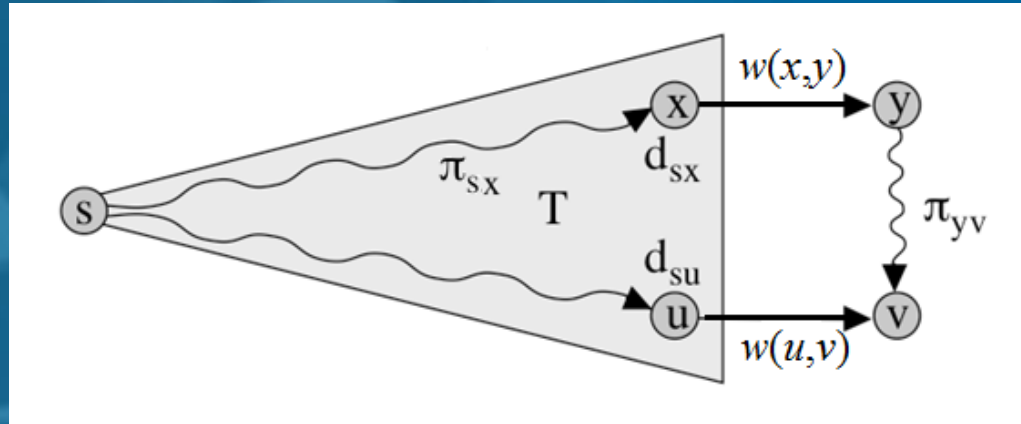
Dim.: Supponiamo per assurdo che (u,v) non appartenga ad un cammino minimo da s a v , e quindi che $d_{sv} < d_{su} + w(u,v)$. Allora, d_{sv} è la lunghezza di un cammino minimo da s a v che non passa per (u,v) . Tale cammino, per uscire da T , passerà allora per un qualche arco $(x,y) \neq (u,v)$, con $x \in T$ e $y \notin T$. Sia quindi $\pi_{sv} = \langle s, \dots, x, y, \dots, v \rangle$.



Per la **minimalità dei sottocammini di un cammino minimo**, il cammino da s a x deve essere minimo e avere quindi lunghezza d_{sx}

$$w(\pi_{sv}) = w(\pi_{sx}) + w(x,y) + w(\pi_{yv}) = d_{sx} + w(x,y) + w(\pi_{yv}).$$

Prova del lemma di Dijkstra (2/2)



Ma poiché (u,v) minimizza $d_{st} + w(t,z)$ per ogni $t \in T$ e $z \notin T$, allora:

$$d_{sx} + w(x,y) \geq d_{su} + w(u,v)$$

e quindi:

$$w(\pi_{sv}) \geq d_{su} + w(u,v) + w(\pi_{yv})$$

e poiché $w(\pi_{yv}) \geq 0$, ne segue $d_{sv} \equiv w(\pi_{sv}) \geq d_{su} + w(u,v)$, assurdo (avevamo supposto $d_{sv} < d_{su} + w(u,v)$). ■

Approccio di Dijkstra

Sia T il sottoalbero dei cammini minimi costruito sinora (all'inizio T contiene solo il nodo sorgente s). Supponiamo quindi che l'algoritmo abbia già trovato d_{su} per ogni $u \in T$, e abbia invece una sovrastima D_{sv} per ogni $v \notin T$. Al passo successivo, seleziona il nodo $v \notin T$ che minimizza la quantità $D_{sv} := d_{su} + w(u, v)$ (si noti che D_{sv} coincide con d_{sv} per il lemma di Dijkstra), con $u \in T$; quindi, aggiungi v a T ed effettua il passo di rilassamento su ogni nodo $y \notin T$ adiacente a v (per il quale cioè $(v, y) \in E$).

I nodi da aggiungere progressivamente a T sono mantenuti in una **coda di priorità**, associati ad **un unico arco** che li connette a T . Se y è in coda con arco (x, y) associato, e se dopo aver aggiunto v a T troviamo un arco (v, y) tale che $D_{sv} + w(v, y) < D_{sx} + w(x, y)$, allora **rimpiazziamo** (x, y) con (v, y) , ed aggiorniamo D_{sy} .

Pseudocodice

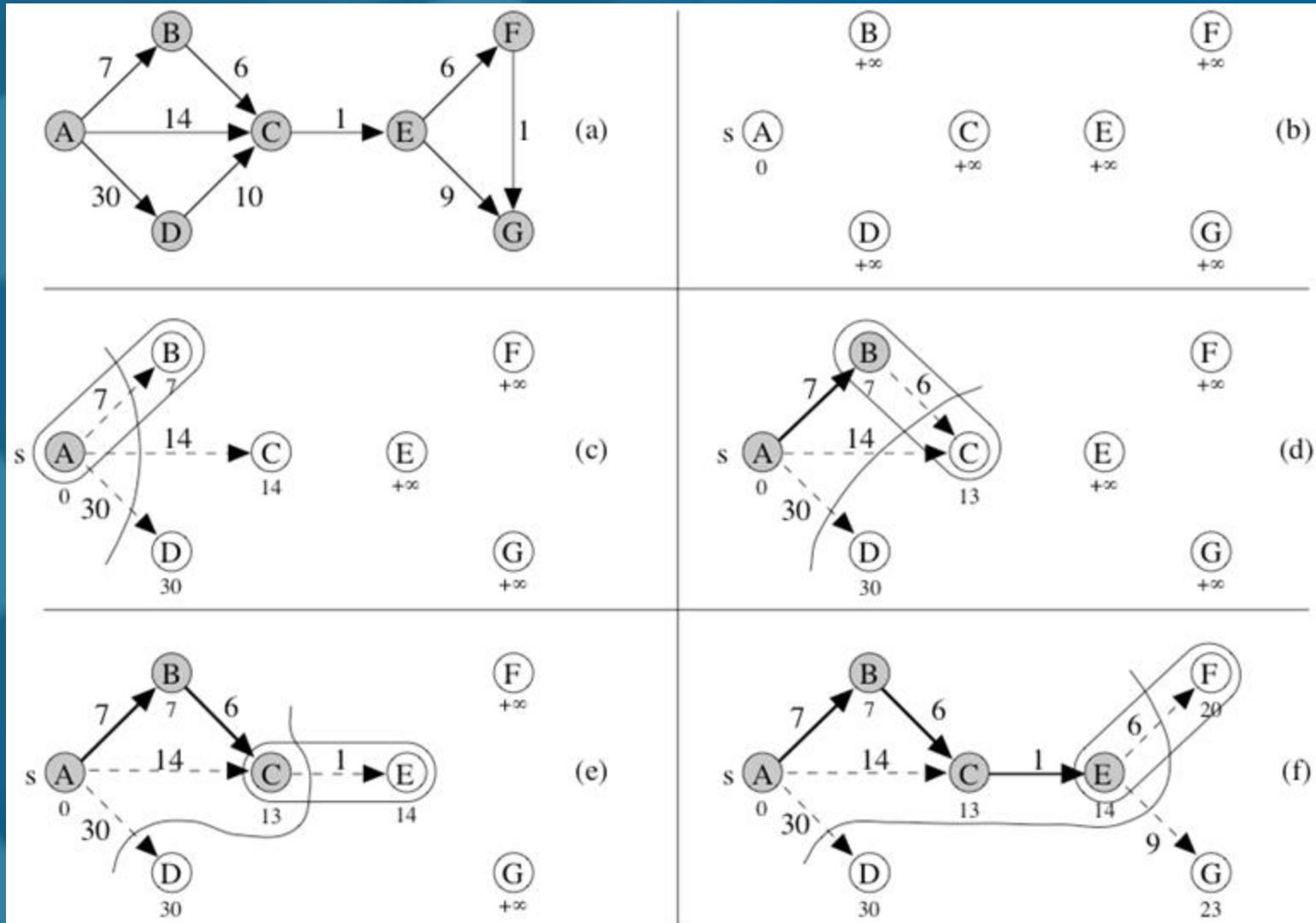
```

algoritmo Dijkstra(grafo  $G$ , vertice  $s$ )  $\rightarrow$  albero
  for each ( vertice  $u$  in  $G$  ) do  $D_{su} \leftarrow +\infty$ 
   $\hat{T} \leftarrow$  albero formato dal solo nodo  $s$ 
  CodaPriorita  $S$ 
   $D_{ss} \leftarrow 0$ 
   $S.insert(s, 0)$ 
  while ( not  $S.isEmpty()$  ) do
     $v \leftarrow S.deleteMin()$ 
    for each ( arco  $(v, y)$  in  $G$  ) do
      if ( $D_{sy} = +\infty$ ) then
         $S.insert(y, D_{sv} + w(v, y))$ 
         $D_{sy} \leftarrow D_{sv} + w(v, y)$ 
        rendi  $v$  padre di  $y$  in  $\hat{T}$ 
      else if ( $D_{sv} + w(v, y) < D_{sy}$ ) then
         $S.decreaseKey(y, D_{sv} + w(v, y))$ 
         $D_{sy} \leftarrow D_{sv} + w(v, y)$ 
        rendi  $v$  nuovo padre di  $y$  in  $\hat{T}$ 
  return  $\hat{T}$ 

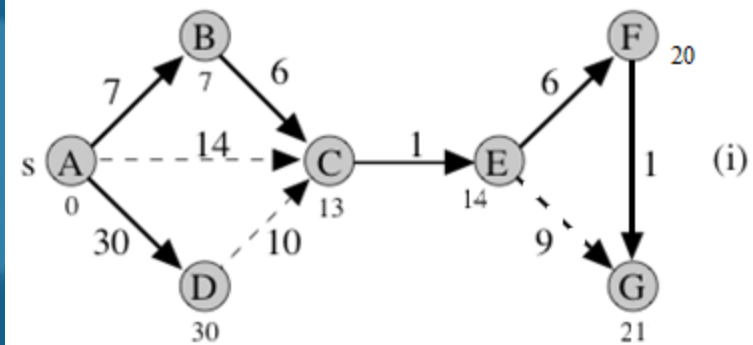
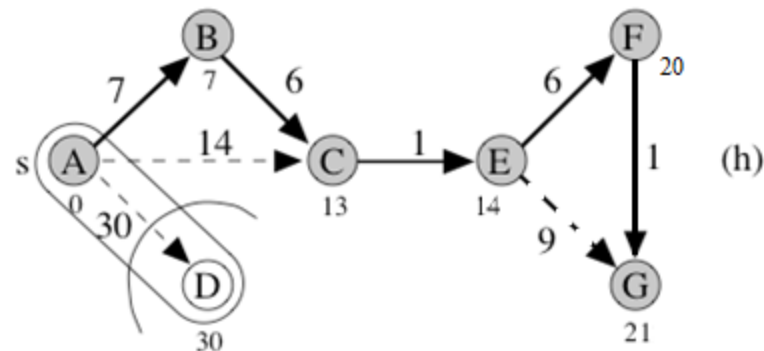
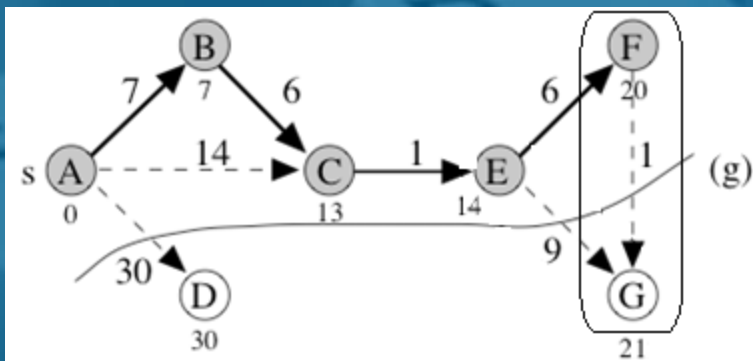
```

Nota: \hat{T} è un albero che contiene tutti i nodi già aggiunti alla soluzione (ovvero quelli in T secondo la notazione del lemma di Dijkstra, per i quali è già stato trovato il cammino minimo da s), **più** i nodi correntemente contenuti nella coda di priorità, cioè “appesi” a T , ciascuno connesso al rispettivo nodo padre.

Esempio (1/2)



Esempio (2/2)



Tempo di esecuzione: implementazioni elementari

Supponendo che il grafo $G=(V,E,w)$ sia rappresentato tramite liste di adiacenza e che $|V|=n$ ed $|E|=m$, e supponendo che G sia fortemente connesso rispetto ad s (e quindi $m \geq n-1$), avremo n `insert`, n `deleteMin` e al più m `decreaseKey` nella coda di priorità S , al costo di:

	Insert	DelMin	DecKey
Array non ord.	$O(1)$	$\Theta(S)=O(n)$	$O(1)$
Array ordinato	$O(S)=O(n)$	$O(1)$	$O(S)=O(n)$
Lista non ord.	$O(1)$	$\Theta(S)=O(n)$	$O(1)$
Lista ordinata	$O(S)=O(n)$	$O(1)$	$O(S)=O(n)$

- $n \cdot O(1) + n \cdot O(n) + O(m) \cdot O(1) = O(n^2)$ con array non ordinati
- $n \cdot O(n) + n \cdot O(1) + O(m) \cdot O(n) = O(m \cdot n)$ con array ordinati
- $n \cdot O(1) + n \cdot O(n) + O(m) \cdot O(1) = O(n^2)$ con liste non ordinate
- $n \cdot O(n) + n \cdot O(1) + O(m) \cdot O(n) = O(m \cdot n)$ con liste ordinate

Osservazione sulla **decreaseKey**

- Ricordiamo che le complessità computazionali esposte per la **decreaseKey** sono valide supponendo di avere un **puntatore diretto** all'elemento su cui eseguire l'operazione. Come possiamo garantire tale condizione?
- Semplicemente inizializzando un puntatore tra il nodo **v** nell'array dei nodi della lista di adiacenza del grafo e l'elemento nella coda di priorità associato al nodo **v**; tale puntatore viene inizializzato nella fase di **inserimento** di quest'ultimo all'interno della coda.

Tempo di esecuzione: implementazioni efficienti

Supponendo che il grafo $G=(V,E,w)$ sia rappresentato tramite liste di adiacenza e che $|V|=n$ ed $|E|=m$, e supponendo che G sia fortemente connesso rispetto ad s (e quindi $m \geq n-1$), avremo n `insert`, n `deleteMin` e al più m `decreaseKey` nella coda di priorità, al costo di:

	Insert	DelMin	DecKey
Heap binario	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap Binom.	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap Fibon.	$O(1)$	$O(\log n)^*$ (ammortizzata)	$O(1)^*$ (ammortizzata)

- $n \cdot O(\log n) + n \cdot O(\log n) + O(m) \cdot O(\log n) = O(m \cdot \log n)$ utilizzando heap binari o binomiali
- $n \cdot O(1) + n \cdot O(\log n)^* + O(m) \cdot O(1)^* = O(m + n \cdot \log n)$ utilizzando heap di Fibonacci (si dimostri che questa è l'implementazione più efficiente)

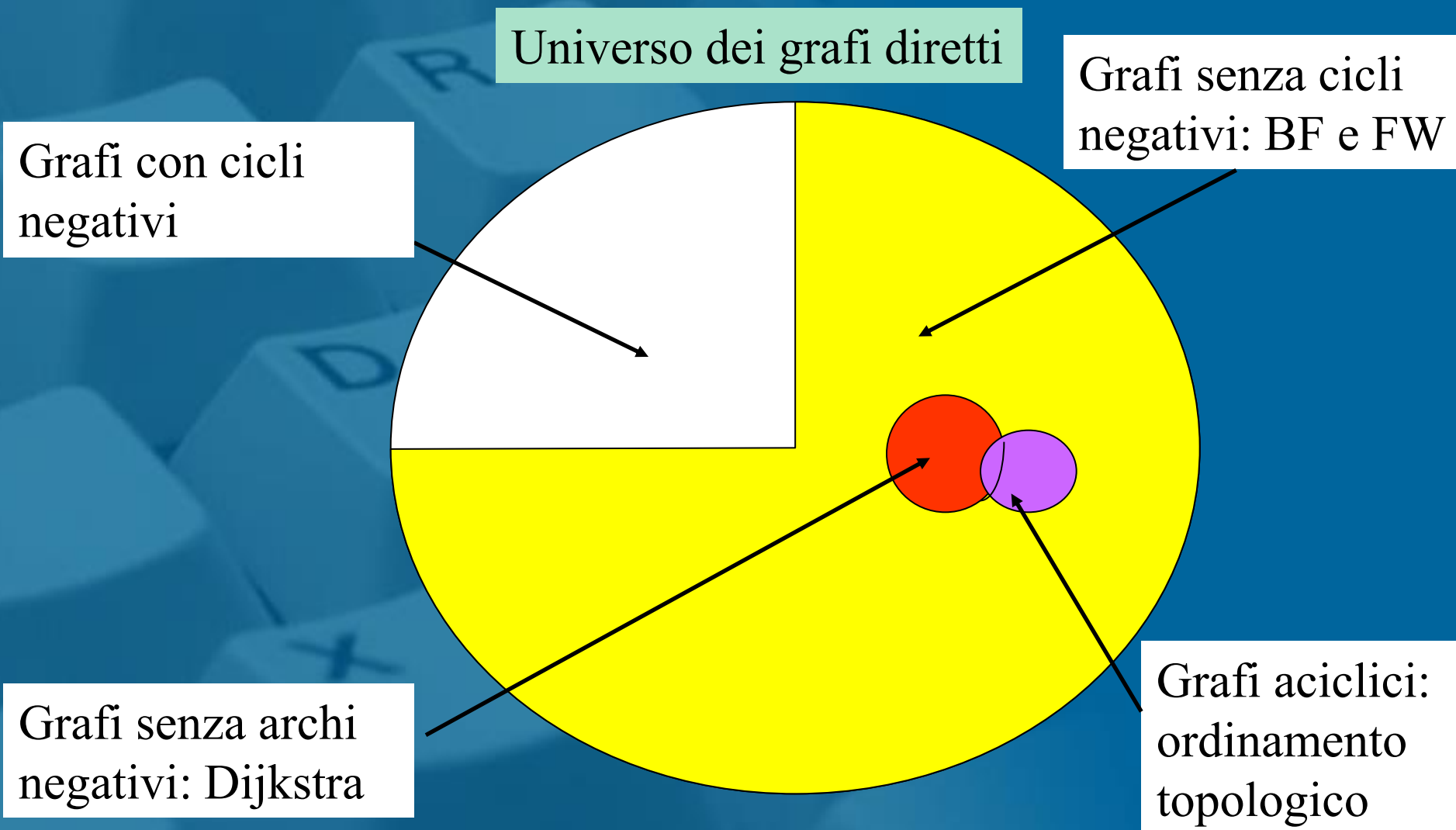
Confronto con le altre implementazioni

- L'implementazione di Dijkstra con heap di Fibonacci è la più efficiente delle implementazioni presentate, poiché:
 - $m + n \log n = O(n^2)$ (liste/array non ord.), in quanto $m = O(n^2)$ e $n \log n = o(n^2)$
 - $m + n \log n = o(n m)$ (liste/array ord.), in quanto $m = o(n m)$ e $n \log n = o(n m)$
 - $m + n \log n = O(m \log n)$ (heap binari/binomiali), in quanto $m = o(m \log n)$ e $n \log n = O(m \log n)$

Domanda

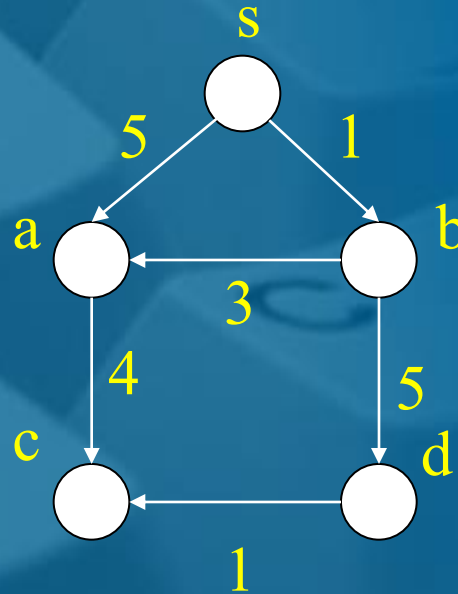
- Quanto costano le varie implementazioni di Dijkstra se il grafo è rappresentato mediante una **matrice di adiacenza** (pensateci, lo chiederò all'orale)?
- Algoritmo di **Dijkstra**:
 - liste/array non ordinate: $\Theta(n^2)$
 - liste/array ordinate: $O(n m)$
 - heap binari/binomiali: $O(n^2 + m \log n)$
 - heap di Fibonacci: $\Theta(n^2)$
- Ricordiamo che per l'algoritmo basato sull'**ordinamento topologico**, tale complessità era pari a $\Theta(n^2)$, mentre per l'algoritmo di **Bellman&Ford**, era pari a $\Theta(n^3)$. Qual è la miglior soluzione?

Riepilogo grafico (per grafi diretti)



Approfondimento

Applicare l'algoritmo di Dijkstra con sorgente **s** sul seguente grafo:



Soluzione esercizio di approfondimento

Applicare l'algoritmo di Dijkstra con sorgente s sul seguente grafo:

