Programmazione dinamica (IV parte)

Progettazione di Algoritmi a.a. 2022-23

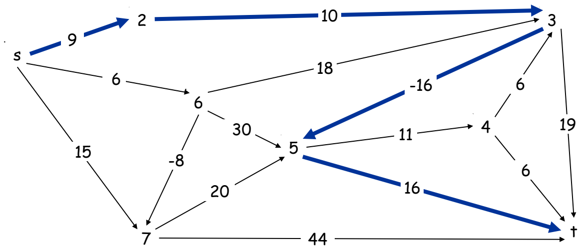
Matricole congrue a 1

Docente: Annalisa De Bonis

Cammini minimi

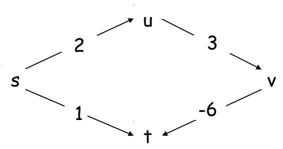
- Problema del percorso piu` corto. Dato un grafo direzionato $G = (V, E)$, con pesi degli archi $c_{vw}$, trovare il percorso piu` corto da s a t.

- Esempio. I nodi rappresentano agenti finanziari e c_{vw} e` il costo (eventualmente <0) di una transazione che consiste nel comprare dall'agente v e vendere immediatamente a w.

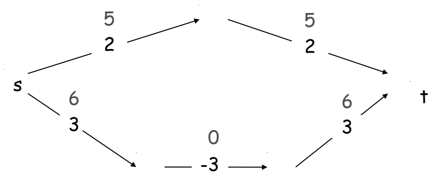


Cammini minimi in presenza di archi con costo negativo

- Dijkstra. Puo` fallire se ci sono archi di costo negativo

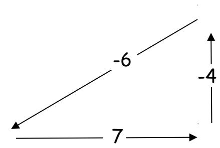


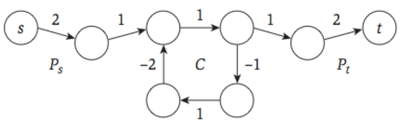
- Re-weighting. Aggiungere una costante positiva ai pesi degli archi potrebbe non funzionare.



Cammini minimi in presenza di archi con costo negativo

- Ciclo di costo negativo.





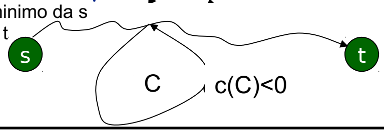
Cammini minimi in presenza di archi con costo negativo

Osservazione. Se qualche percorso da s a t contiene un ciclo di costo negativo allora non esiste un percorso minimo da s a t. In caso contrario esiste un percorso minimo da s a t che e` semplice (nessun nodo compare due volte sul percorso).

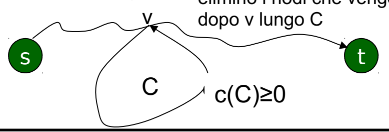
- Dim. Se esiste un percorso P da s a t con un ciclo C di costo negativo –c allora gni volta che attraversiamo il ciclo riduciamo il costo del percorso di un valore pari a c. Cio` rende impossibile definire il costo del percorso minimo perche` dato un percorso riusciamo sempre a trovarne uno di costo minore attraversando il ciclo C (osservazione questa che avevamo gia` fatto in precedenti lezioni).

Supponiamo ora che nessun percorso da s a t contenga cicli negativi e sia P un percorso minimo da s a t (ovviamente P e` privo di cicli di costo negativo). Supponiamo che un certo vertice v appaia almeno due volte in P. C'e` quindi in P un ciclo che contiene v e che per ipotesi deve avere costo **non negativo**. In questo caso potremmo rimuovere le porzioni di P tra due occorrenze consecutive di v in P senza far aumentare il costo del percorso che risulterebbe ancora minimo.

non esiste un percorso minimo da s a t



elimino i nodi che vengono dopo v lungo C



62

62

Cammini minimi: Programmazione dinamica

Sia t la destinazione a cui si vuole arrivare.

Def. OPT(i, v) = lunghezza del cammino piu` corto P per andare da v a t che consiste di al piu` i archi

Per computare OPT(i,v) quando i>0 e v≠t, osserviamo che

- il percorso ottimo P deve contenere almeno un arco (che ha come origine v).
- se (v, w) e` il primo arco di P allora P e` formato da (v, w) e d percorso piu` corto da w a t di al piu` i-1 archi.
- siccome non sappiamo quale sia il primo arco di P allora computiamo $OPT(i,w) = \min_{(v,w)\in E}\{ OPT(i-1,w) + c_{vw}\}$

La formula di ricorrenza è quindi:

$$\text{OPT(i,v)} = \begin{cases} 0 & \text{se v=t} \\ \infty & \text{se } i = 0 \text{ e } v \neq t \\ \min_{(v,w)\in E}\{OPT(i-1,w)+c_{vw}\} & \text{altrimenti} \end{cases}$$

Progettazione di Algoritmi A.A. 2022-23
A. De Bonis

63

63

Cammini minimi: Programmazione dinamica

$$\text{OPT(i,v)}=\begin{cases} 0 & \text{se v=t} \\ \infty & \text{se } i=0 \text{ e } v \neq t \\ \min_{(v,w)\in E}\{OPT(i-1,w)+c_{vw}\} & \text{altrimenti} \end{cases}$$

Notiamo che nel secondo caso usiamo ∞ per indicare che attraversando 0 archi non è possible raggiungere t.

Dove si usa l'osservazione di prima sul fatto che in assenza di cicli negativi il percorso minimo e` semplice?

Ecco dove...
Affermazione. Se non ci sono cicli di costo negativo allora OPT(n-1, v) = lunghezza del percorso piu` corto da v a t.

Dim. Dall'osservazione precedente se non ci sono cicli negativi allora esiste un percorso di costo minimo da v a t che e` semplice e di conseguenza contiene al piu` n-1 archi

$$\text{OPT(i,v)}=\begin{cases} 0 & \text{se v=t} \\ \infty & \text{se } i=0 \text{ e } v \neq t \\ \min_{(v,w)\in E}\{OPT(i-1,w)+c_{vw}\} & \text{altrimenti} \end{cases}$$

Algoritmo di Bellman-Ford per i cammini minimi

```
Shortest-Path(G, t) {
   foreach node v ∈ V
      M[0, v] ← ∞
      S[0, v] ← Ø // Ø  indica che non ci sono percorsi
                   //da v a t di al piu` 0 archi
   for i = 0 to n-1
      M[i, t] ← 0
      S[i, t] ← t  //t indica che non ci sono successori di t
                   //lungo il percorso ottimo da t a t
   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← ∞, S[i, v] ← Ø
         foreach edge (v, w) ∈ E
            if M[i-1, w] + cvw < M[i, v]
                M[i, v] ← M[i-1, w] + cvw
                S[i,v] ← w  //serve per ricostruire i
                            //percorsi minimi verso t
}
```

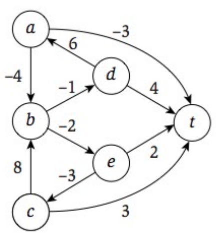
- Assumiamo che per ogni v esista un percorso da v a t → n=O(m)
- Analisi. Tempo Θ(mn), spazio Θ(n^2).
- S[i,v]: Memorizza il successore di v lungo il percoso minimo per andare da v a t attraversando al piu` i archi

Algoritmo di Bellman-Ford per i cammini minimi

Osservazione

- L'algoritmo di Bellman-Ford di fatto calcola le lunghezze dei cammini minimi da v a t per ogni v (risolve Single Destination Shortest Paths)
- Queste lunghezze sono contenute nella riga n-1
- L'algoritmo puo` essere scritto in modo che prenda in input un vertice sorgente s ed un vertice destinazione t ma il contenuto della tabella M dipende solo da t.
- In altri termini, una volta costruita la tabella per un certo t, possiamo ottenere la lunghezza del percorso piu` corto da un qualsiasi nodo v al nodo t andando a leggere l'entrata M[n-1,v]

Bellman-Ford: esempio



Come aggiorno la cella M[i,j] e la cella S[i,j], per i>1?
Inizialmente pongo M[i,j]=∞ e S[i,j]=∅.
Nella riga i-1 esamino tutte le entrate M[i-1,k] per cui esiste l'arco (j,k) e per ciascuna di queste entrate computo M[i-1,k]+c_{jk}. Se questo valore è piu` piccolo di M[i,j], pongo M[i,j]= M[i-1,k]+c_{jk} e S[i,j]=k.

M

	t	a	b	c	d	e
0	0	∞	∞	∞	∞	∞
1	0	-3	∞	3	4	2
2	0	-3	0	3	3	0
3	0	-4	-2	3	3	0
4	0	-6	-2	3	2	0
5	0	-6	-2	3	0	0

S

	t	a	b	c	d	e
0	t	∅	∅	∅	∅	∅
1	t	t	∅	t	t	t
2	t	t	e	t	a	c
3	t	b	e	t	a	c
4	t	b	e	t	a	c
5	t	b	e	t	a	c

Algoritmo che produce il cammino minimo

```
FindPath(i,v):
  if S[i,v]= Ø
      output "No path"
       return
  if v= t
      output t
      return
  output v
  FindPath(i-1,S[i,v])
```

prima volta invocato con i=n-1 e v uguale al nodo per il quale vogliamo computare il cammino minimo fino a t

tempo O(n) perche'
- se ignoriamo il tempo per la chiamata ricorsiva al suo interno, il tempo di ciascuna chiamata e` O(1)
- vengono effettuate al piu` n-1 chiamate

68

Bellman-Ford: esempio

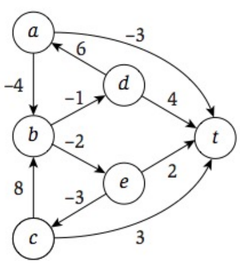
S

	t	a	b	c	d	e
0	t	Ø	Ø	Ø	Ø	Ø
1	t	t	Ø	t	t	t
2	t	t	e	t	a	c
3	t	b	e	t	a	c
4	t	b	e	t	a	c
5	t	b	e	t	a	c

Supponiamo di voler conoscere il percorso minimo tra a e t
Invoco FindPath(5,a)

output a e effettua ricorsione con i=4 e v=S[5,a]=b
output b e effettua ricorsione con i=3 e v= S[4,b]=e
output e e effettua ricorsione con i=2 e v= S[2,e]=c
output c e effettua ricorsione con i=1 e v= S[1,c]=t
output t ed esci

Il percorso minimo da a verso t e` a,b,e,c,t




```
FindPath(i,v):
  if S[i,v]= Ø
      output "No path"
       return
  if v= t
      output t
      return
  output v
  FindPath(i-1,S[i,v])
```

69

69

Implementazione efficiente di Bellman-Ford
Alcune osservazioni sull'algoritmo

- Risparmio in termini di spazio: puo` essere ottenuto osservandoche non e` necessario portarsi dietro tutta la matrice M perche' nell'algoritmo di fatto ogni volta che si riempe una nuova riga di M si fa uso solo dei valori della riga precedente
  - per riempire la riga i si usano solo i valori presenti della riga i-1
  - quindi perche' portarsi dietro anche le altre righe?

•Un primo immediato miglioramento lo si ottiene andando a modificare la prima versione dell'algoritmo in modo che
1. usi un array unidimensionale M
2. ad ogni iterazione del for piu` esterno vada ad aggiornare ciascun valore M[v] allo stesso modo in cui prima computava i valori M[i,v].
   - Per far questo invece di utilizzare i valori M[i-1,v] utilizzera` i valori M[v] computati all'iterazione precedente che saranno stati salvati in un array di appoggio

Con questa modifica usiamo 2 array unidimensionali per computare le lunghezze dei percorsi e un array S per tenere traccia dei successori-> spazio O(n)

Computazione del cammino minimo

- Per ogni vertice v memorizziamo in S[v] il successore di v, cioe` il primo nodo che segue v lungo il percorso da v a t di costo M[v].
- S[v] viene aggiornato ogni volta che M[v] viene aggiornato. Se M[v] viene posto uguale a c_{vw}+M[w] allora si pone S[v]=w.

Algoritmo di Bellman-Ford : I miglioramento

MA: array di appoggio

```
Improved-Shortest-Path_1(G, t) {
   foreach node v ∈ V
      M[v] ← ∞
       MA[v] ← ∞
      S[v] ← ∅ // ∅  indica che non ci sono percorsi
                   //da v a t di al piu` 0 archi
   M[t] ← 0
   MA[t]← 0
   S[t] ← t  //t indica che non ci sono successori
                 //lungo il percorso ottimo da t a t
   for i = 1 to n-1
    foreach node v ∈ V
         foreach edge (v, w) ∈ E
            if MA[w] + c_vw < M[v]
                M[v] ← MA[w] + c_vw
                S[v]← w  //serve per ricostruire i
                           //percorsi minimi verso t
    foreach node v ∈ V
        MA[v]=M[v] //salvo M[v] nell'array di appoggio
}
```


Implementazione efficiente di Bellman-Ford

Alcune osservazioni sull'algoritmo

- Possiamo apportare un ulteriore miglioramento basato sulla seguente. osservazione:
- Se in una certa iterazione i del for esterno il valore di MA[w] è lo stesso dell'iterazione precedente (M[w] non è stato aggiornato nel corso dell'iterazione i-1) allora i valori MA[w] + c_{vw} computati nell'iterazione i sono esattamente gli stessi computati nell'iterazione i-1.
- Questa osservazione dà l'idea per un secondo miglioramento dell'algoritmo: quando in una certa iterazione i del for esterno, l'algoritmo calcola M[v] va a considerare solo quei nodi w per cui esiste l'arco (v,w) e tali che M[w] e` stato modificato durante l'iterazione i-1.
- L'algoritmo nella slide successiva realizza questa idea in questo modo: scandisce ciascun nodo w del grafo e controlla se il valore di M[w] è cambiato nell'iterazione precedente e solo in questo caso esamina gli archi (v,w) entranti in v e per ciascuno di questi archi computa MA[w] + c_{vw}
- Cio` equivale a scandire tutti i nodi v e a controllare per ogni arco (v,w) uscente da v se M[w] e` cambiato nell'iterazione precedente prima di calcolare **MA[w] +** c_{vw}

Algoritmo di Bellman-Ford : II miglioramento

MA: array di appoggio

```
Improved-Shortest-Path_2(G, t) {
   foreach node v ∈ V
      M[v] ← ∞
      MA[v] ← ∞
      S[v] ← Ø // Ø  indica che non ci sono percorsi
                    //da v a t di al piu` 0 archi
   M[t] ← 0
   MA[t]← 0
   S[t] ← t  //t indica che non ci sono successori
                      //lungo il percorso ottimo da t a t
   for i = 1 to n-1
     foreach node w ∈ V
        if M[w] has been updated in iteration i-1
          foreach edge (v, w) ∈ E
             if MA[w] + cvw < M[v]
                M[v] ← MA[w] + cvw
                S[v]← w  //serve per ricostruire i
                          //percorsi minimi verso t
    foreach node v ∈ V
      MA[v]=M[v]//salvo M[v]nell'array di appoggio
}
```


Implementazione efficiente di Bellman-Ford

Alcune osservazioni sull'algoritmo

- Torniamo per un momento al fatto che un miglioramento dell'algoritmo consiste nell'usare un array unidimensionale M.
- Abbiamo detto che per far ciò l'algoritmo può usare un array di appoggio che memorizza i valori di M computati dall'iterazione precedente del for esterno.
- Domanda: cosa accade se non utilizziamo un array di appoggio?
- Consideriamo l'iterazione i del for esterno.
- Se non utilizziamo un array di appoggio, quando calcoliamo **M[w]** + c_{vw} , siamo costretti ad usare i valori M[w] presenti in M.

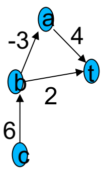
  •Quando calcoliamo **M[w]** + c_{vw}, il valore **M[w]** potrebbe essere uguale al valore computato nell'iterazione i-1 o potrebbe gia` essere stato aggiornato nell'iterazione i (anche piu` di una volta) .

  •Nel caso M[w] sia stato già modificato nell'iterazione i allora M[w] conterrà la lunghezza di un percorso piu` corto rispetto al valore di M[w] computato nell'iterazione precedente.

  - Di conseguenza M[v] potrebbe essere aggiornato con un valore piu` piccolo di quello che si sarebbe ottenuto utilizzando il valore di M[w] computato nell'iterazione precedente.

Implementazione efficiente di Bellman-Ford
Alcune osservazioni sull'algoritmo

- Conseguenze dell'osservazione nella slide precedente:
  - Dopo ogni iterazione i, M[v] potrebbe contenere la lunghezza di un percorso per andare da v a t formato da **piu` di i** archi.
  - La lunghezza di M[v] e` sicuramente non piu` grande della lunghezza del percorso piu` corto per andare da v a t formato da **al massimo** i archi.



- Esempio, Consideriamo il grafo qui di fianco.
- Iterazione i=1: supponiamo di esaminare i nodi w in questo ordine t,a,b,c. Quando esaminiamo w=t, poniamo M[a]=4 e M[b]=2. Quando si esamina w=a si ha M[a]=4 e di conseguenza M[b] diventa 1 (lunghezza del percorso b,a,t). Quando poi esaminiamo b, si ha che M[b]=1 e di conseguenza M[c] da ∞ che era, diventa 7 (lunghezza di c,b,a,t).
- Nell'implementazione con array di appoggio, alla fine della prima iterazione avremmo avuto M[b]=2 e M[c] = ∞.

•Il terzo e ultimo miglioramento consiste nel modificare **Improved-Shortest-Path_2** in modo che non usi l'array di appoggio. In questo modo si ottiene l'algoritmo **Push-Based-Shortest-Path.**

Implementazione efficiente di Bellman-Ford

```
Push-Based-Shortest-Path(G,  t) {
   foreach node v ∈ V {
      M[v] ← ∞
      S[v] ← ϕ   //nel libro si chiama first[v]
   }

   M[t] = 0 , S[t]=t
   for i = 1 to n-1 {
      foreach node w ∈ V {
      if (M[w] has been updated in previous iteration) {
         foreach node v such that (v, w) ∈ E {
            if (M[v] > M[w] + c_vw) {
               M[v] ← M[w] + c_vw
               S[v] ← w
            }
         }
      }
      if no M[v] value changed in this iteration i
        return M[]
   }
   return M[]
}
```

Le osservazioni viste nelle slide precedenti ci portano a questa versione dell'algoritmo

NB: in una certa iterazione del for esterno quando si calcola una distanza $M[w]+c_{vw}$ potrebbe accadere che M[w] sia stata aggiornata gia` in quella stessa iterazione.

Miglioramento dell'algoritmo

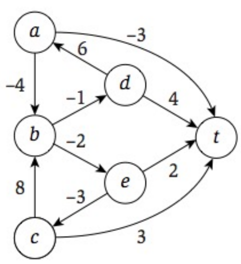
Teorema. Durante l'algoritmo **Push-Based-Shortest-Path**, M[v] e` la lunghezza di un certo percorso da v a t, e dopo i round di aggiornamenti (dopo i iterazioni del for esterno) il valore di M[v] **non è più grande della lunghezza del percorso minimo da v a t che usa al piu` i archi**

- Non usare un array di appoggio in pratica accelera i tempi per ottenere i percorsi piu` corti fino a t formati da al piu` n-1 archi (che sono quelli che ci interessa ottenere).
- **Nulla cambia per quanto riguarda l'analisi asintotica dell'algoritmo**

- Conseguenze sullo spazio usato da **Push-Based-Shortest-Path**
  - Memoria: O(n).

- Tempo:
- il tempo e` sempre O(nm) nel caso pessimo pero` in pratica l'algoritmo si comporta meglio.
  - Possiamo interrompere le iterazioni non appena accade che durante una certa iterazione i nessun valore M[v] cambia

Miglioramento dell'algoritmo: un esempio

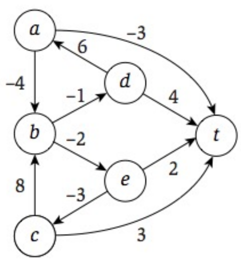


	t	a	b	c	d	e	
M	0	∞	∞	∞	∞	∞	inizializ-
S	t	ϕ	ϕ	ϕ	ϕ	ϕ	zazione
M	0	-3	∞	3	4	2	i=1
S	t	t	ϕ	t	t	t	w=t
M	0	-3	∞	3	3	2	i=1
S	t	t	ϕ	t	a	t	w=a
M	0	-3	∞	3	3	2	i=1
S	t	t	ϕ	t	a	t	w=b
M	0	-3	∞	3	3	0	i=1
S	t	t	ϕ	t	a	c	w=c
M	0	-3	2	3	3	0	i=1
S	t	t	d	t	a	c	w=d
M	0	-3	-2	3	3	0	i=1
S	t	t	e	t	a	c	w=e

per ogni nodo w esaminato nel secondo foreach le celle verdi sono quelle che cambiano valore

le celle arancioni sono quelle il cui valore non e` cambiato nell'i-esima iterazione

Miglioramento dell'algoritmo: un esempio

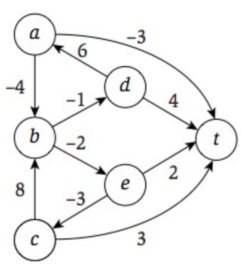


	t	a	b	c	d	e	
M	0	-3	-2	3	3	0	fine iteraz. i =1
S	t	t	e	t	a	c	
M	0	-3	-2	3	3	0	i=2 w=a
S	t	t	e	t	a	c	
M	0	-6	-2	3	3	0	i=2 w=b
S	t	b	e	t	a	c	
M	0	-6	-2	3	3	0	i=2 w=c
S	t	b	e	t	a	c	
M	0	-6	-2	3	3	0	i=2 w=d
S	t	b	e	t	a	c	
M	0	-6	-2	3	3	0	i=2 w=e
S	t	b	e	t	a	c	

per ogni nodo w esaminato nel secondo foreach le celle verdi sono quelle che cambiano valore

le celle arancioni sono quelle il cui valore non e` cambiato nell'i-esima iterazione

Miglioramento dell'algoritmo: un esempio

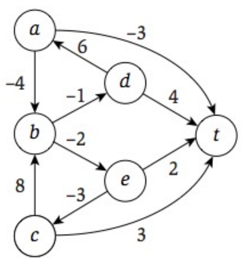


	t	a	b	c	d	e	
M	0	-6	-2	3	3	0	fine iteraz. i =2
S	t	b	e	t	a	c	
M	0	-6	-2	3	0	0	i=3 w=a
S	t	b	e	t	a	c	

per ogni nodo w esaminato nel secondo foreach le celle verdi sono quelle che cambiano valore

le celle arancioni sono quelle il cui valore non e` cambiato nell'i-esima iterazione

Miglioramento dell'algoritmo: un esempio



	t	a	b	c	d	e	
M	0	-6	-2	3	0	0	fine
S	t	b	e	t	a	c	iteraz. i=3
M	0	-6	-2	3	0	0	i=4
S	t	b	e	t	a	c	w=d

le celle arancioni sono quelle il cui valore non e` cambiato nell'i-esima iterazione