Corso di laurea in Informatica
Progettazione d'algoritmi
Didattica blended

Tecnica Programmazione Dinamica 2

Angelo Monti

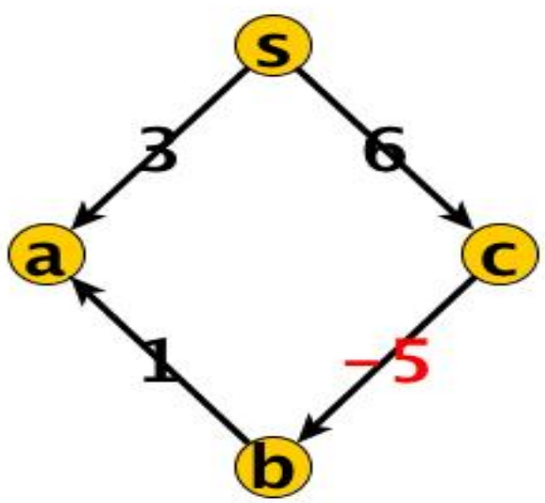
SAPIENZA
UNIVERSITÀ DI ROMA

Algoritmi per la ricerca di cammini su grafi con pesi anche negativi

Problema: dato un grafo diretto e pesato G con archi i cui pesi possono essere anche negativi e fissato un suo nodo s, vogliamo calcolare il costo minimo dei cammini che portano da s agli altri nodi del grafo (il costo è $+\infty$ se non c'è cammino)

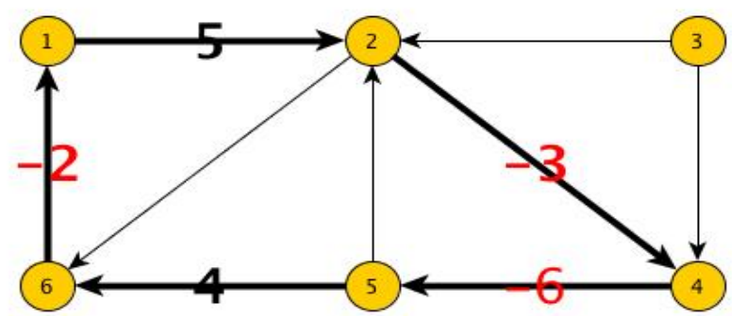
- È un problema che abbiamo già affrontato nel caso in cui i pesi del grafo G sono tutti non negativi (algoritmo di Dijkstra)

L'algoritmo di Dijkstra può sbagliare nel caso siano presenti in G anche archi di peso negativo:



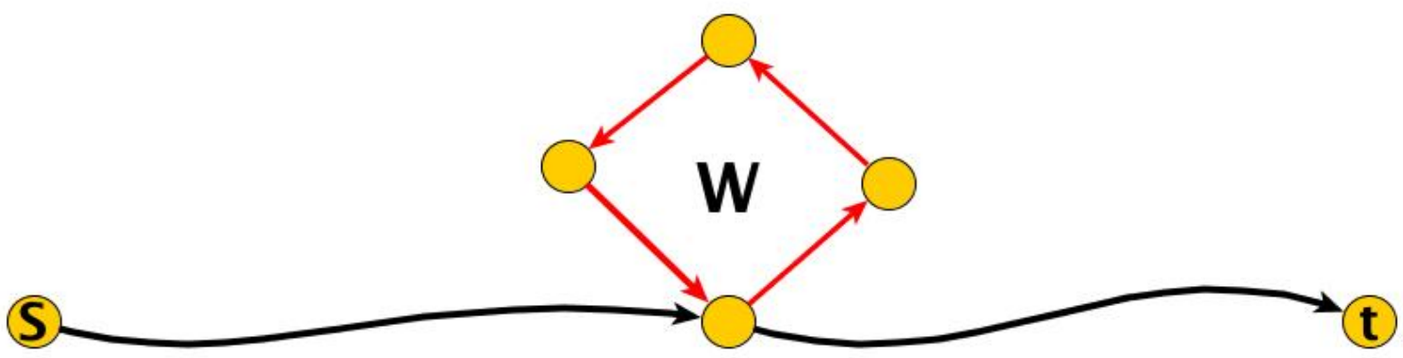
L'algoritmo di Dijkstra sceglie come prima cosa il cammino composto dal solo arco **(s,a)** di costo **3**, ma il cammino da **s** ad **a** che passa per **c** e **b** costa 6 - 5 +1= **2**.

Definizione: Un ciclo negativo è un ciclo diretto formato da archi con pesi la cui somma è negativa.



Il ciclo evidenziato in figura è negativo perché ha costo $5 - 3 - 6 + 4 - 2 =$ - 2

Se in un cammino tra i nodi s e t è presente un nodo che appartiene ad un ciclo negativo allora tra s e t non esiste il cammino di costo minimo.



- se per il ciclo W si ha $costo(W) < 0$, ripassando più volte attraverso il ciclo W possiamo abbassare arbitrariamente il costo del cammino da s a t.

Alla luce di quanto appena visto il problema non era ben formulato. Ecco di seguito la giusta formulazione:

Problema: dato un grafo diretto e pesato G con archi i cui pesi possono essere anche negativi ma che non contiene cicli negativi e fissato un suo nodo s, vogliamo calcolare il costo minimo dei cammini che portano da s agli altri nodi del grafo (il costo è $+\infty$ se non c'è cammino).

Vedremo ora un algoritmo basato sulla tecnica della programmazione dinamica che risolve il problema in tempo $O(n^2 + m \cdot n)$.
L'algoritmo è noto in letteratura come algoritmo di Bellman-Ford.

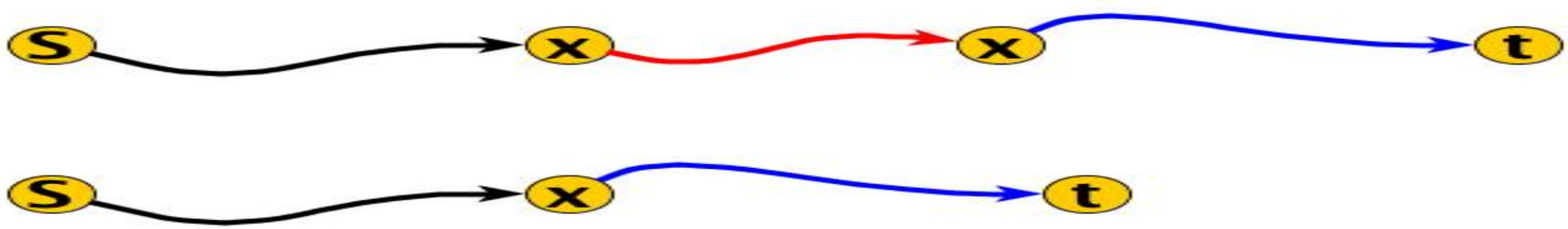
Se G non ha cicli negativi allora per ogni nodo t raggiungibile da s esiste un cammino di costo minimo da s a t che ha lunghezza al più $n-1$.

Prova:

- consideriamo il più corto cammino minimo e assumiamo per assurdo che abbia lunghezza superiore ad $n-1$.

- c'è allora un nodo x toccato più volte dal cammino. Di conseguenza il cammino contiene un ciclo.

- possiamo eliminare dal cammino il ciclo ed ottenere un cammino di lunghezza inferiore che è ancora minimo (ricorda che il ciclo eliminato non può avere costo negativo).



Per quanto appena visto: restringersi a cammini di lunghezza al più $n-1$ non fa perdere di generalità.
Tutto questo suggerisce di considerare i sottoproblemi che si ottengono limitando la lunghezza dei cammini.
Definiamo così la seguente tabella di dimensione $n \times n$:

$$T[k,j] = \begin{cases} \text{costo di un cammino minimo da } s \text{ a } j \text{ di lunghezza al più } k & \text{se esiste} \\ +\infty & \text{altrimenti} \end{cases}$$

Ovviamente la soluzione al nostro problema sarà data dagli n valori che troviamo nell'ultima riga della tabella:

$$T[n-1,\ 0],\ \ T[n-1,\ 1],\ \ T[n-1,\ 2] \ldots\ldots T[n-1,\ n-1]$$

Infatti il costo minimo per andare da s al generico nodo v sarà $T[n-1][v]$

La tabella verrà calcolata per righe, resta dunque da definire la generica cella $T[k,j]$ in funzione delle celle già calcolate presenti nella riga $k-1$ della tabella T.

$$T[k,j] = \begin{cases} \text{costo di un cammino minimo da } s \text{ a } j \text{ di lunghezza al più } k & \text{se esiste} \\ +\infty & \text{altrimenti} \end{cases}$$

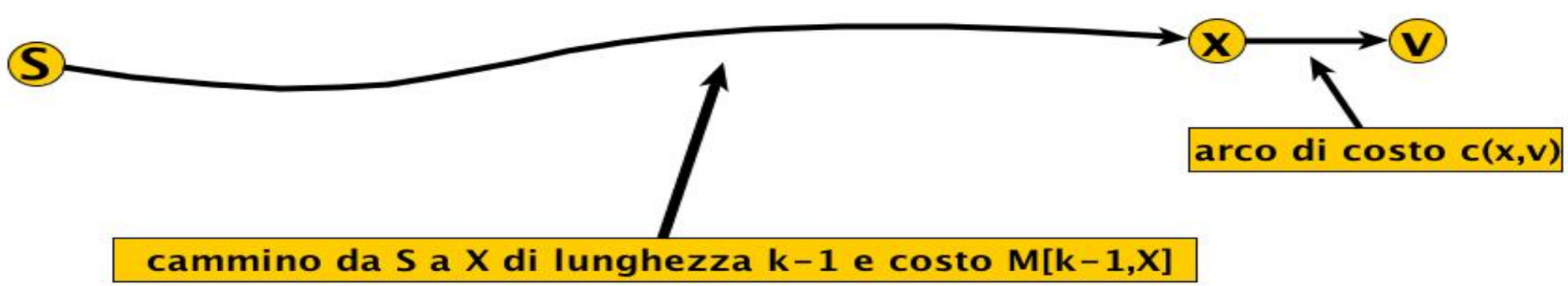
Vale:

$$T[k,\ v] = \min(T[k-1,\ v],\quad min_{(x,v)\in E}\{T[k-1,\ x] + costo(x,v)\})$$

Infatti:

Per calcolare $T[k,v]$ con $k > 0$ osserviamo che:

- se il cammino di costo minimo ha lunghezza inferiore a k allora si ha $T[k,v] = T[k-1,v]$,
- se al contrario il cammino di costo minimo ha lunghezza esattamente k allora il nodo v è preceduto nel cammino da un qualche nodo x da cui parte un arco verso v ed il cammino di costo minimo da s a x ha lunghezza esattamente $k-1$ per cui $T[k,v] = T[k-1,x] + costo(x,v)$.



Possiamo dunque calcolare le n^2 celle della tabella T con la seguente ricorrenza:

$$T[k,\ v] = \begin{cases} 0 & \text{se } k = 0 \text{ e } v = s \\ +\infty & \text{se } k = 0 \\ \min(T[k-1,\ v],\ \ min_{(x,v)\in E}\{T[k-1,\ x] + costo(x,v)\}) & \text{altrimenti} \end{cases}$$

Implementazione:

```
def costo_cammini(G,s):
    from math import inf
    n=len(G)
    T=[ [inf for _ in range(n)] for _ in range(n)]
    T[0][s]=0
    GT={v:[] for v in G}
    for v in G:
        for x,costo in G[v]: GT[x].append((v,costo))
    for k in range(1,n):
        for v in range(n):
            T[k][v]=T[k-1][v]
            for x,costo in GT[v]:
                T[k][v]=min(T[k][v],T[k-1][x]+costo)
    return T[n-1]

>>>G={
0:[(1,3),(3,6)],
1:[],
2:[(1,1)],
3:[(2,-5)],
4:[(0,-2)]
}

>>> costo_cammini(G,4)
[-2, 0, -1, 4, 0]
>>> costo_cammini(G,0)
[0, 2, 1, 6, inf]
```

- l'inizializzazione della tabella T richiede tempo $\Theta(n^2)$.
- la costruzione del grafo trasposto GT richiede tempo $O(n+m)$.
  Nota che: grazie a questo pre-processing avremo accesso efficiente all'insieme degli archi entranti al nodo v (il grafo G ci dava invece accesso efficiente all'insieme degli archi uscenti nel nodo v.
- Per i tre FOR annidati (for k in $range(1,n)$:... for v in $range(n)$... for $x, costo$ in $GT[v]$: ...) è ovvio il limite superiore $O(n^3)$. Tuttavia un'analisi più attenta permette di ottenere il limite superiore più stretto $\Theta(n \cdot m)$:
  - I due FOR più interni (for v in $range(n)$... for $x, costo$ in GT: ...) hanno costo totale $\Theta(m)$.
    Sostanzialmente il tempo richiesto è quello di scorrere tutte le liste di adiacenza del grafo GT che hanno lunghezza totale $\Theta(m)$

La complessità della procedura è $\Theta(n^2 + nm)$.

Dal costo ai cammini:

Per ritrovare anche i cammini, oltre al loro costo, bisogna calcolare con la tabella T anche l'albero P dei cammini minimi. Questo si può fare facilmente mantenendo per ogni nodo v il suo predecessore, cioè il nodo u che precede v nel cammino minimo. Il valore di $P[v]$ andrà aggiornato ogni volta che il valore di $T[k][v]$ cambia (ovvero diminuisce) in quanto abbiamo trovato un cammino migliore.

```
def costo_cammini(G, s) :
    from math import inf
    n = len(G)
    T = [[inf for _ in range(n)] for _ in range(n)]
    P = [-1 for _ in range(n)]
    P[s] = s
    GT = {v : [ ] for v in G}
    for v in G :
        for x, costo in G[v] : GT[x].append((v, costo))
    for k in range(1, n) :
        for v in range(n) :
            T[k][v] = T[k - 1][v]
            for x, costo in GT[v] :
                if T[k - 1][x] + costo < T[k][v]:
                    T[k][v] = T[k - 1][x] + costo
                    P[v] = x
return T[n - 1],  P
```

Con questa implementazione al termine dell'algoritmo

- $T[n-1][v] \neq +\infty$ indica che v è raggiungibile da s, in questo caso $P[v]$ conterrà il nodo che precede v nel cammino minimo da s a v
- $T[n-1][v] = +\infty$ indica che v non è raggiungibile da s, in questo caso $P[v]$ conterrà il valore -1.

Ottimizzazioni:

- il contenuto di una cella della riga k dipende dal contenuto delle celle alla riga $k-1$, quindi:
  - Se la riga k della tabella T è identica alla riga $k-1$ anche le righe seguenti non varieranno, tanto vale allora terminare l'algoritmo senza aver calcolato le righe restanti della tabella. Questo accorgimento non modifica la complessità asintotica dell'algoritmo ma in pratica può contare molto.
  - Non serve memorizzare l'intera tabella T bastano le ultime due righe. Perciò l'algoritmo può essere facilmente modificato in modo da utilizzare memoria $O(n)$ anziché $O(n^2)$

L'algoritmo appena descritto è conosciuto con il nome di Algoritmo di Bellman-Ford.

A priori non sappiamo se il grafo su cui applichiamo l'algoritmo ha cicli negativi (e quindi se i cammini restituiti sono effettivamente i cammini minimi o semplicemente i cammini minimi con lunghezza al più $n-1$).

Una piccola modifica alla versione dell'algoritmo di Bellman-Ford appena descritto permette di scoprire se il grafo contiene cicli negativi raggiungibili da s o meno:

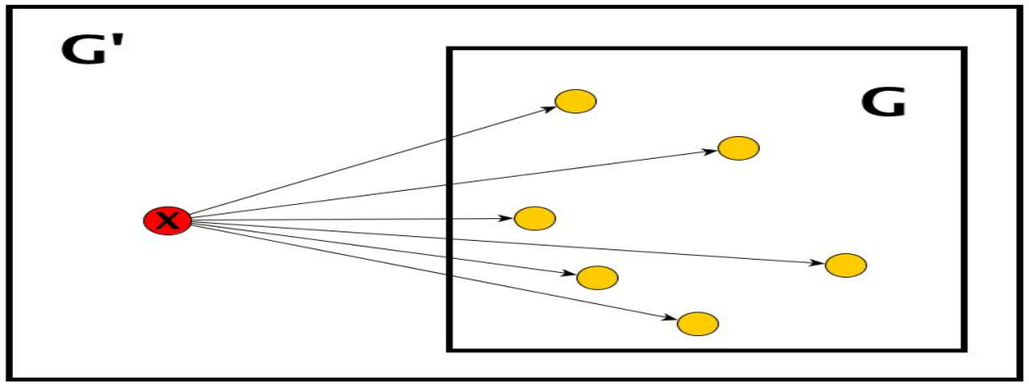
- calcola una riga in più della tabella (vale a dire la riga n) con il costo dei cammini minimi di lunghezza al più n.
- Le righe n ed $n-1$ della tabella risultano identiche se e solo se nel grafo non ci sono cicli negativi raggiungibili da s.

Implementare questo test ovviamente non cambia l'asintotica dell'algoritmo.

ESERCIZIO d'esame: progettare un algoritmo che dato un grafo G diretto e pesato con pesi anche negativi scopre se il grafo contiene cicli negativi.

L'algoritmo proposto deve avere complessità $O(n^2 + n \cdot m)$

ESERCIZIO d'esame: progettare un algoritmo che dato un grafo G diretto e pesato con pesi anche negativi in tempo $O(n^2 + n \cdot m)$ scopre se il grafo contiene cicli negativi.



Sia G il grafo di cui vuoi sapere se contiene cicli negativi.

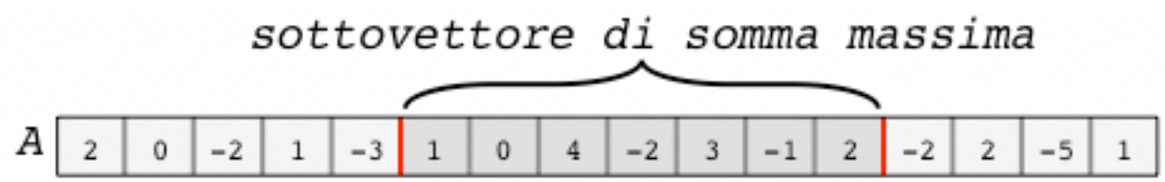
- Costruisci un nuovo grafo G' a partire da G aggiungendo un nuovo nodo x con n archi diretti (di costo 0) verso gli n nodi del grafo originario.
- ovviamente il grafo G' gode di queste due proprietà
  - ha gli stessi cicli di G
  - tutti i suoi cicli sono raggiungibili dal nodo x
- per sapere se G contiene cicli negativi esegui l'algoritmo di Bellman-Ford sul grafo G' scegliendo x come radice.

Poiché la costruzione di G' richiede tempo $O(n + m)$ abbiamo dimostrato che:

scoprire se in un grafo G sono presenti cicli negativi richiede tempo $O(n^2 + nm)$.

ESEMPIO:

Il problema del massimo sottovettore: Data una lista A di n interi, vogliamo trovare una sottolista (una sequenza di elementi consecutivi della lista) la somma dei cui elementi è massima.



Abbiamo già parlato di questo problema come esempio di applicazione della tecnica del *divide et impera*, **lo riprendiamo ora per tentare un approccio basato sulla programmazione dinamica. Come è tipico di questa tecnica ci concentriamo prima sul calcolare il valore della soluzione:**

Cominciamo con l'individuare i sottoproblemi dalla composizione delle cui soluzioni sarà poi possibile risolvere il problema originario.

- una scelta che ci porta a definire n sottoproblemi è la seguente:

  $T[i]$ = massima somma possibile per le sottoliste di A che terminano nella posizione i

- Poiché la sottolista a valore massimo deve terminare in una qualche posizione il valore della soluzione che cerchiamo sarà poi dato da

$$\max_{0 \leq i < n} T[i]$$

- $T[i]$ = massima somma possibile per le sottoliste di A che terminano nella posizione i

vediamo ora come risolvere in modo efficiente gli n sottoproblemi:

- Il caso base è facile: $T[0] = A[0]$.
- dobbiamo ora poter calcolare $T[i]$, con $i > 0$, sfruttando la conoscenza di $T[i-1]$:

$$T[i] = \max(A[i], A[i] + T[i-1])$$

  Infatti la sottolista di valore massimo che termina con $A[i]$ può essere di due soli tipi:

  - consiste del solo elemento A[i]
  - ha lunghezza superiore ad 1. In questo caso vale $A[i] + S$ dove S è la massima somma di un sottovettore che termina in $i-1$ (il che significa che $S = T[i-1]$)

Implementazione:

```
def MS( lista):
    T=[0 for _ in range(len(lista))]
    T[0]=lista[0]
    for i in range(1, len(lista)):
        T[i]=max(lista[i],lista[i]+T[i-1])
    return max(T)

>>> A = [ 2, 0, -2, 1, -3, 1, 0, 4, -2, 3, -1, 2, -2, 2, -5, 1]
>>> MS(A)
7
```

Complessità $O(n)$

Ottimizzazioni:

- possiamo evitare di dover calcolare al termine il massimo in T, basta una variabile m che tiene traccia del massimo per il valore delle celle di T che via via calcoliamo.

- possiamo evitare di tenere in memoria l'intera tabella T visto che per calcolare $T[i]$ basta disporre della sola cella $T[i-1]$ (e del valore $A[i]$) (in questo modo la complessità di spazio passa da $O(n)$ ad $O(1)$).

Implementazione 2:

```
def MS1( lista):
    m=T=lista[0]
    for i in range(1, len(lista)):
        T=max(lista[i],lista[i]+T)
        m=max(m,T)
    return m

>>> A = [ 2, 0, -2, 1, -3, 1, 0, 4, -2, 3, -1, 2, -2, 2, -5, 1]
>>> MS1(A)
7
```

Complessità $O(n)$

Come è tipico della *programmazione dinamica*, una volta trovato come produrre il valore della soluzione non è difficile modificare l'algoritmo, senza alterarne la complessità, in modo tale che oltre al valore della soluzione venga prodotta anche la soluzione.

In questo caso basterà tener traccia in una variabile di dove termina la sottolista ottima e poi partendo dalla fine della sottolista muoversi verso sinistra fino ad arrivare alla posizione per cui la somma dei valori compresi nella sottolista sia proprio la somma ottima:

```
def MS2( lista):
    b=m=T=lista[0]
    for i in range(1, len(lista)):
        T=max(lista[i],lista[i]+T)
        if m<T:
            m,b=T,i
    a,tot=b,m
    while tot-lista[a]!=0:
        tot -=lista[a]
        a-=1
    return m , (a,b)
```

Complessità $O(n)$