Corso di laurea in Informatica
Progettazione d'algoritmi

Algoritmo di Bellman-Ford

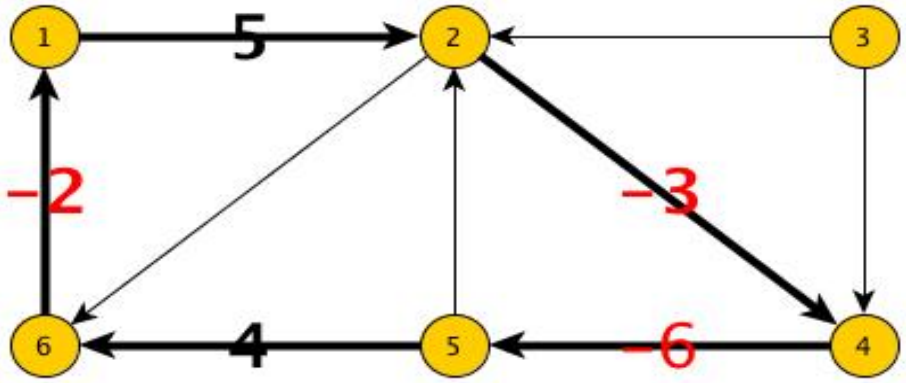
Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Algoritmo per la ricerca di cammini su grafi con pesi anche negativi.

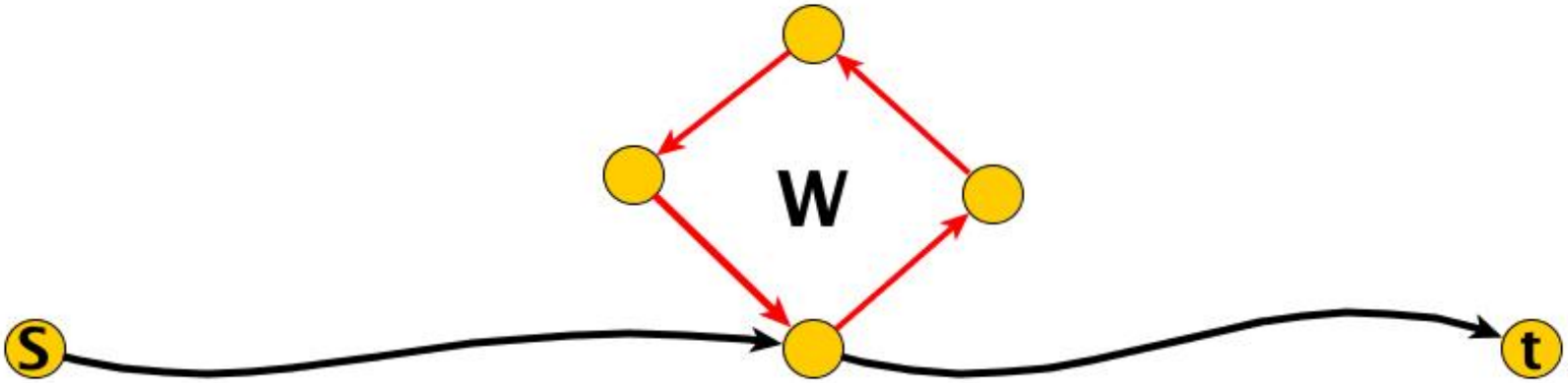
Problema: dato un grafo diretto e pesato G in cui i pesi degli archi possono essere anche negativi e fissato un suo nodo s, vogliamo determinare il costo minimo dei cammini che conducono da s a tutti gli altri nodi del grafo. Se non esiste un cammino verso un determinato nodo il costo sarà considerato infinito.

Definizione: Un ciclo negativo è un ciclo diretto in un grafo in cui la somma dei pesi degli archi che lo compongono è negativa.



Il ciclo evidenziato in figura è negativo di 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 non esiste il cammino di costo minimo tra s e t.



- 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.

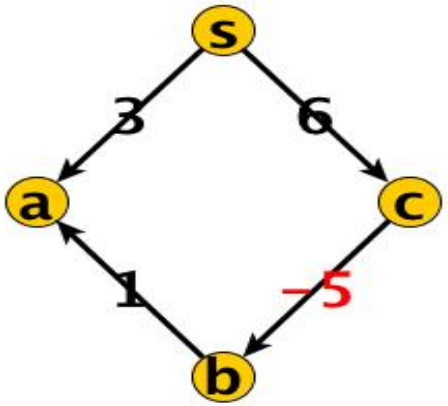
Alla luce di quanto appena detto sui cicli negativi, la formulazione del problema non era del tutto corretta. Ecco di seguito la versione corretta:

Problema: dato un grafo diretto e pesato G in cui i pesi degli archi possono essere anche negativi **ma che non contiene cicli negativi,** e fissato un suo nodo s, vogliamo determinare il costo minimo dei cammini che conducono da s a tutti gli altri del grafo. Se non esiste un cammino verso un determinato nodo il costo sarà considerato infinito.

E' un problema che abbiamo già affrontato nel caso in cui i pesi del grafo G siano tutti non negativi utilizzando l'algoritmo di Dijkstra.

Tuttavia in presenza di archi con peso negativo, l'algoritmo di Dijkstra può produrre risultati errati poiché assume che, una volta visitato un nodo con costo minimo, tale costo non possa più essere migliorato. Questa assunzione non è valida se esistono archi di peso negativo.

Considera ad esempio il grafo:



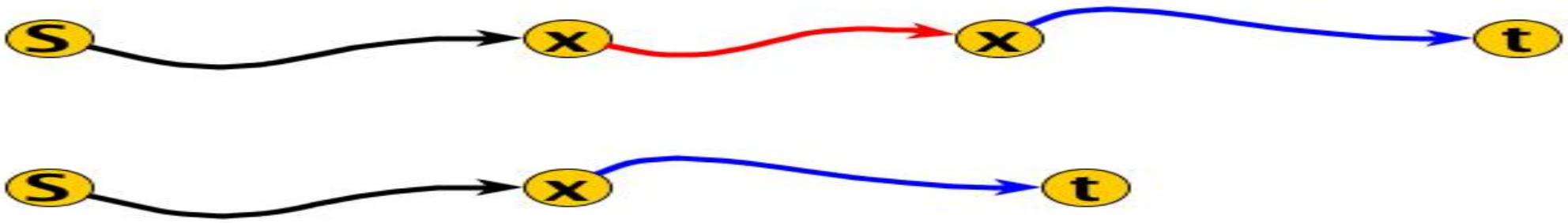
L'algoritmo di Dijkstra sceglie come prima cosa il cammino composto dal solo arco **(s,a)** di costo **3**, ma il cammino da **s** ad **a** che passa per **c** e **b** costa 6 - 5 +1= **2**.

Vedremo ora un algoritmo più adatto 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.

Proprietà: se il grafo G non contiene cicli negativi, allora per ogni nodo t raggiungibile dalla sorgente s esiste un cammino di costo minimo che attraversa al più $n-1$ archi.

Infatti, se un cammino avesse più di $n-1$ archi, allora almeno un nodo verrebbe ripetuto, formando un ciclo. Poiché il grafo non ha cicli negativi, rimuovere eventuali cicli dal cammino **non aumenta** il suo costo complessivo. Di conseguenza esiste sempre un cammino ottimale di lunghezza $n-1$.
Questo garantisce che il costo minimo può essere calcolato considerando solo cammini di questa lunghezza.



Per quanto appena visto: nella ricerca dei cammini minimi possiamo restringerci a cammini di lunghezza al più $n-1$.
Tutto questo suggerisce di considerare sottoproblemi che si ottengono limitando la lunghezza dei cammini.

Definiamo così la seguente tabella di dimensione $n \times n$:

$T[i][j] =$ **costo di un cammino minimo da** s **al nodo** j **di lunghezza al più** i

Calcoleremo la soluzione al nostro problema determinando valori della tabella pe righe.
La soluzione al nostro problema sarà data dagli n valori che troviamo nell'ultima riga:

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

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

I valori della prima riga della tabella T sono ovviamente tutti $+\infty$ tranne $T[0][s]$ che vale zero. Inoltre $T[i][s] = 0$ per ogni $i > 0$.

Resta da definire la regola che permette di calcolare i valori delle celle $T[i][j]$ con $j \neq s$ della riga $i > 0$ in funzione delle celle già calcolate della riga i-1:
Distinguiamo due casi a seconda che il cammino di lunghezza al più i da s a j abbia lunghezza inferiore a i o esattamente i.
Nel primo caso ovviamente si ha
$T[i][j] = T[i-1][j]$.
Nel secondo caso deve invece esistere un cammino minimo di lunghezza al più $i-1$ ad un nodo x e poi un arco che da x mi porta a j. In altre parole:

$$T[i][j] = \min_{(x,\ j) \in E} \left(T[i-1][x] + costo(x, j) \right)$$

cammino da s a x di al più I -1 archi e costo T[I-1][x] | arco di costo c(<x,j>)

Non sapendo in quale dei due casi siamo la formula giusta è:

$$T[i][j] = \min \left(T[i-1][j],\ \min_{(x,\ j) \in E} \left(T[i-1][x] + costo(x, j) \right) \right)$$

Riassumendo le celle della tabella possono essere riempite per righe in base a questa regola:

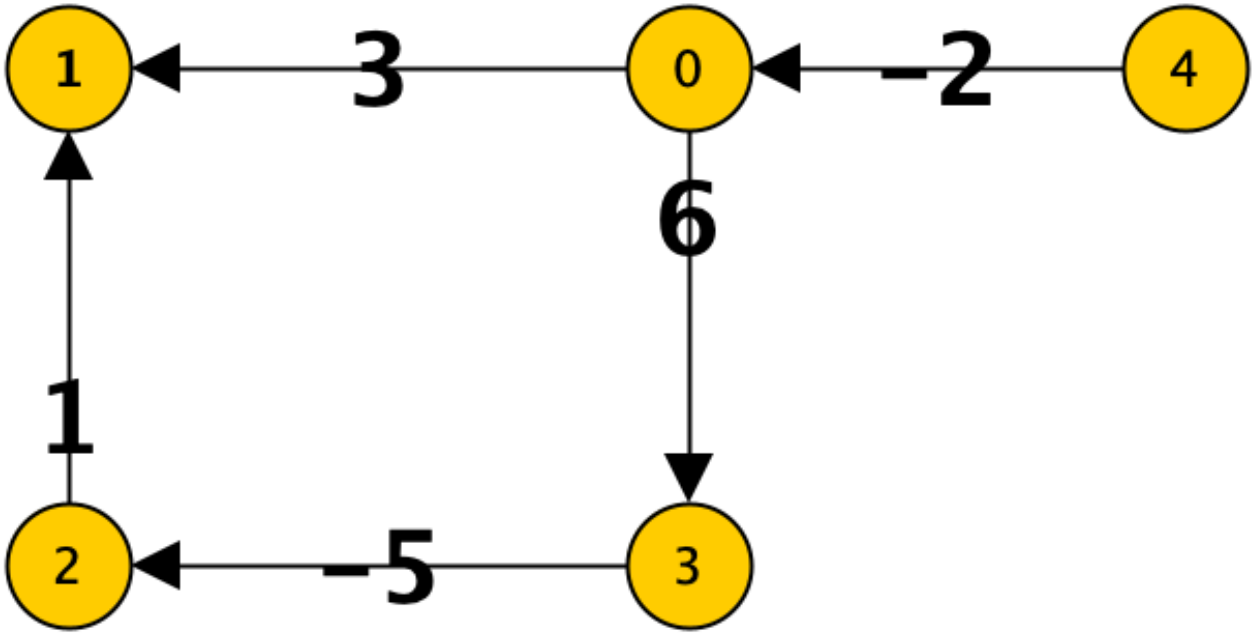
$$T[i][j] = \begin{cases} 0 & \text{se } j = s \\ +\infty & \text{se } i = 0 \\ \min\Big(T[i-1][j],\ \min_{(x,j)\in E}\Big(T[i-1][x] + costo(x,j)\Big)\Big) & \text{altrimenti} \end{cases}$$

NOTA: per un'implementazione efficiente, poiché nel calcolo della formula è necessario più volte conoscere gli archi entranti nel generico nodo j, conviene precalcolare il grafo trasposto GT di G. In questo modo, in $GT[j]$ avremo l'elenco di tutti i nodi x tali che in G esiste un arco da x a j. Questo permette di accedere rapidamente agli archi entranti di un nodo, migliorando l'efficienza.

IMPLEMENTAZIONE:

```
def CostoCammini(G, s):
    n = len(G)
    inf= float('inf')
    T =[ [inf]*n for _ in range(n) ]
    T[0][s] = 0
    GT = Trasposto(G)
    for i in range(1,n):
        for j in range(n):
            T[i][j]=T[i-1][j]
            for x,costo in GT[j]:
                T[i][j] = min( T[i][j], T[i-1][x] + costo )
    return T[n-1]


def Trasposto(G):
    n=len(G)
    GT = [ [] for _ in G ]
    for i in range(n):
        for j,costo in G[i]:
            GT[j].append( (i,costo) )
    return GT
```




```
G=[
    [(1,3),(3,6)],
    [],
    [(1,1)],
    [(2,-5)],
    [(0,-2)]
  ]
```

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

- 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 un accesso efficiente ai nodi che hanno un arco diretto in j.
- per i tre for annidati (for k in $range(1,n)$ ... for j in $range(n)$ ... for $< x,\ costo >$ in $GT[j]$) è ovvio il limite superiore $O(n^3)$. Tuttavia un'analisi più attenta permette di dare un limite più stretto:
  - I due for più interni (for j in $range(n)$ ... for $< x,\ costo >$ in $GT[j]$) 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 m.

Complessità:

```
def CostoCammini(G, s):
    n = len(G)
    inf= float('inf')
    T =[ [inf]*n for _ in range(n) ]
    T[0][s] = 0
    GT = Trasposto(G)
    for i in range(1,n):
        for j in range(n):
            T[i][j]=T[i-1][j]
            for x,costo in GT[j]:
                T[i][j] = min( T[i][j], T[i-1][x] + costo )
    return T[n-1]
```

- 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 un accesso efficiente ai nodi che hanno un arco diretto in j.
- per i tre for annidati (for i in $range(1,n)$ ...for j in $range(n)$ ...for x, $costo$ in $GT[j]$) è ovvio il limite superiore $O(n^3)$. Tuttavia un'analisi più attenta permette di dare un limite più stretto:
  - I due for più interni (for j in $range(n)$ ...for x, $costo$ in $GT[j]$) 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 m.

La complessità complessiva è $O(n^2+mn)$

Dal costo ai cammini:

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

```
def CostoCammini1(G, s):
    n = len(G)
    inf= float('inf')
    T =[ [inf]*n for _ in range(n) ]
    P = [-1]* n # inizializzo il vettore dei padri
    GT = Trasposto(G)
    P[s]=s # s è la radice dell'albero dei cammini
    T[0][s] = 0
    for i in range(1,n):
        for j in range(n):
            T[i][j] = T[i-1][j]
            for x, costo in GT[j]:
                if T[i-1][x] + costo < T[i][j]:
                    T[i][j] =  T[i-1][x] + costo
                    # il cammino attuale minimo che da s
                    # arriva a j ci arriva tramite x
                    P[j] = x
    return T[n-1], P # restituisco le distanze ed il vettore dei padri
```

```python
def CostoCammini1(G, s):
    n = len(G)
    inf= float('inf')
    T =[ [inf]*n for _ in range(n) ]
    P = [-1]* n
    GT = Trasposto(G)
    P[s]=s
    T[0][s] = 0
    for i in range(1,n):
        for j in range(n):
            T[i][j] = T[i-1][j]
            for x, costo in GT[j]:
                if T[i-1][x] + costo < T[i][j]:
                    T[i][j] =  T[i-1][x] + costo
                    P[j] = x
    return T[n-1], P
```

Con questa implementazione al termine dell'algoritmo

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