Corso di laurea in Informatica
Progettazione d'algoritmi
Didattica blended

Tecnica Programmazione Dinamica 3

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Il problema dello zaino:

Abbiamo uno zaino di capacità C ed n oggetti, ognuno con un peso p_i e un valore v_i.
Vogliamo trovare un sottoinsieme degli oggetti la cui somma dei pesi non supera C e che massimizza il valore totale.

Esempio: Considera la seguente istanza con $C = 11$ e $n = 5$ oggetti con peso e valore riportati in tabella

oggetto	valore	peso
1	1	1
2	6	4
3	18	5
4	22	5
5	28	7

Il problema dello zaino: Abbiamo uno zaino di capacità C ed n oggetti, ognuno con un peso p_i e un valore v_i. Vogliamo trovare un sottoinsieme degli oggetti la cui somma dei pesi non supera C e che massimizza il valore totale.

Esempio: Considera la seguente istanza con $C = 11$ e $n = 5$ oggetti con peso e valore riportati in tabella

oggetto	valore	peso
1	1	1
2	6	4
3	18	5
4	22	5
5	28	7

- il sottoinsieme di oggetti $\{3, 5\}$ non è una soluzione ammissibile (perché di peso $12 > C$)
- il sottoinsieme di oggetti $\{1, 2, 4\}$ è una soluzione ammissibile (di peso 10 e valore 29) ma non ottima (perché ne esistono di valore maggiore).
- il sottoinsieme di oggetti $\{1, 3, 4\}$ è una soluzione ammissibile (di peso 11 e valore 41) si può dimostrare che è anche una soluzione ottima.

Concentriamoci preliminarmente sul calcolare il valore della soluzione ottima e definiamo una tabella bidimensionale T di dimensione $(n + 1) \times (C + 1)$:

$T[i][c]$ = massimo valore ottenibile dai primi i oggetti per uno zaino di capacità c

La soluzione al nostro problema sarà $T[n][C]$

Resta solo da definire la regola ricorsiva che permette di calcolare i vari valori della tabella.

$T[i][c]$ = massimo valore ottenibile dai primi i oggetti per uno zaino di capacità c

Ecco di seguito la relazione di ricorrenza che permette di calcolare le celle della tabella

$$T[i][c] = \begin{cases} 0 & \text{se } i = 0 \\ T[i-1][c] & \text{se } p_i > c \\ \max\{\ T[i-1][c],\ \ v_i + T[i-1][c-p_i]\ \} & \text{altrimenti} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- se non si hanno oggetti (vale a dire $i = 0$) il valore della soluzione sarà 0.
- se l'i-esimo oggetto ha un peso superiore alla capacità c dello zaino, allora non può esservi inserito e quindi il valore della soluzione dipenderà da quello che si può ricavare dagli altri $i-1$ oggetti (vale a dire $T[i-1,c]$)
- in generale, se nello zaino c'è possibilità di inserire l'oggetto i allora ci sono due possibilità:
  - *l'oggetto i non viene inserito nello zaino*. In questo caso il massimo valore della soluzione sarà $T[i-1,c]$.
  - l'oggetto i viene inserito nello zaino. In questo caso c'è un guadagno $V[i]$ e per i rimanenti $i-1$ oggetti resta disponibile una capacità residua di $c - P[i]$. Di conseguenza il valore massimo della soluzione sarà $V[i] + T[i-1,c-P[i]]$.

Pertanto il valore della soluzione sarà $\max(T[i-1,c], V[i] + T[i-1,c-P[i]])$

$$T[i][c] = \begin{cases} 0 & \text{se } i = 0 \\ T[i-1][c] & \text{se } p_i > c \\ \max\{\ T[i-1][c],\ \ v_i + T[i-1][c-p_i]\ \} & \text{altrimenti} \end{cases}$$

Implementazione:

```
def zaino(P,V,C):
    n=len(P)
    T=[[ 0 for _ in range(C+1)] for _ in range(n+1)]
    for i in range(n+1):
        for c in range(C+1):
            if i==0: T[i][c]=0
            elif P[i-1]>c: T[i][c]=T[i-1][c]
            else: T[i][c]=max( T[i-1][c], T[i-1][c-P[i-1]]+V[i-1])
    return T[n][C]


>>>P=[1, 4, 5, 5, 7]
>>>V=[1, 6, 18, 22, 28]
>>>zaino(P, V, 10)
40
```

Complessità $\Theta(n \cdot C)$

SCHEMI D'APPROSSIMAZIONE

Complessità $\Theta(n \cdot C)$

Nota che quello che abbiamo ottenuto è un algoritmo *pseudopolinomiale*.
Per il problema dello zaino non sono noti algoritmi efficienti in grado di risolverlo esattamente.
Il problema è però molto facile da approssimare. Per questo problema è infatti noto uno *schema d'approssimazione*.
Uno schema d'approssimazione per un problema è un algoritmo che prende in input l'istanza del problema ed un valore $0 < \epsilon \leq 1$ e nell'esecuzione si comporta come un algoritmo d'approssimazione di rapporto $1 + \epsilon$ il suo tempo di calcolo sarà funzione di n ed ϵ.

Ad esempio per il problema dello zaino si conosce uno schema d'approssimazione che garantisce rapporto $1 + \epsilon$ al costo di un tempo $O\left(\frac{n^3}{\epsilon}\right)$:

- con $\epsilon = 1$ si ha un algoritmo d'approssimazione che lavora in tempo $O(n^3)$ e garantisce un rapporto 2.
- con $\epsilon = \frac{1}{2}$ si ha un algoritmo d'approssimazione che lavora in tempo $O(2n^3)$ e garantisce un rapporto $1 + \frac{1}{2} = 1.5$
- con $\epsilon = \frac{1}{100}$ si ha un algoritmo d'approssimazione che lavora in tempo $O(100 \cdot n^3)$ e garantisce un rapporto $1 + \frac{1}{100} = 1.01$

Dal valore della soluzione alla soluzione:

- Una volta calcolati i valori della tabella T è facile ricavare l'insieme degli oggetti che producono la soluzione ottima: basta seguire a ritroso nella tabella T le scelte che hanno portato a calcolare il valore della soluzione.

$$T[i][c] = \begin{cases} 0 & \text{se } i = 0 \\ T[i-1][c] & \text{se } p_i > c \\ \max\{\ T[i-1][c],\ \ v_i + T[i-1][c-p_i]\ \} & \text{altrimenti} \end{cases}$$

- Partendo dalla cella $T[n,C]$ posso per muovermi tra le celle di T distinguendo due casi:
  1. se $T[i,c] = T[i-1,c]$ allora l'oggetto i non appartiene alla soluzione ottima e mi sposto alla cella $T[i-1,c]$
  2. se $T[i,c]! = T[i-1,c]$ allora l'oggetto appartiene alla soluzione ottima e mi sposto alla cella $T[i-1,c-P[i]]$.

```
def sol_zaino(P,V,C):
    n=len(P)
    T=[[ 0 for _ in range(C+1)] for _ in range(n+1)]
    for i in range(n+1):
        for c in range(C+1):
            if i==0: T[i][c]=0
            elif P[i-1]>c: T[i][c]=T[i-1][c]
            else: T[i][c]=max( T[i-1][c], T[i-1][c-P[i-1]]+V[i-1])
    #calcolo ora la soluzione ottima
    sol=set()
    i,c=n,C
    while i>0:
        if T[i][c]>T[i-1][c]:
            sol.add(i)
            c-=P[i-1]
        i-=1
    return T[n][C], sol


>>>P=[1, 4, 5, 5, 7]
>>>V=[1, 6, 18, 22, 28]
>>> sol_zaino(P,V,11)
(41, {1, 3, 4})
```

- inizializzare la tabella e riempirne le celle costa $\Theta(nC)$
- ricostruire la soluzione costa $\Theta(n)$

La complessità è $\Theta(nC)$

Esempio

Cammini minimi tra tutte le coppie di nodi.

Dato un grafo diretto e pesato G, vogliamo trovare i cammini minimi tra tutte le coppie di nodi del grafo.

Possiamo risolvere questo problema utilizzando algoritmi che già conosciamo:

- Se il grafo ha solamente pesi non negativi potremmo usare l'algoritmo di Dijkstra che, applicato a partire da ogni nodo, risolve il problema in tempo $O(n(n+m)\log n)$. Se il grafo è denso la complessità diventa $O(n^3 \log n)$.
- Se il grafo ha anche pesi negativi, possiamo usare l'algoritmo di Bellman-Ford che applicato a partire da ogni nodo richiede $O(n(n^2 + n \cdot m))$. Se il grafo è denso la complessità diventa $O(n^4)$.

Vedremo ora un algoritmo in grado di calcolare i cammini minimi tra tutte le coppie di nodi in presenza di archi di peso anche negativo in tempo $O(n^3)$.
Parliamo dell'**algoritmo di Floyd-Warshall** .
Come per Bellman-Ford anche quest'algoritmo utilizza la tecnica della programmazione dinamica.

Come al solito, per applicare la tecnica della programmazione dinamica bisogna individuare dei sotto-problemi.

- Per l'algoritmo di Bellman-Ford i sotto-problemi sono stati individuati limitando il numero di archi che si potevano percorrere lungo i cammini.
- Per l'algoritmo di Floyd-Warshall i sotto-problemi vengono individuati limitando l'indice dei nodi che possono essere attraversati dai cammini.

Assumiamo per iniziare che il grafo G non abbia cicli negativi , per cui sappiamo che i cammini minimi se esistono sono semplici (vale a dire in un cammino minimo non passo mai più volte da uno stesso nodo).
Utilizzeremo una tabella tridimensionale $n \times n \times n$ dove per i valori delle celle abbiamo:

$$T[k][u][v] = \begin{cases} \text{Minimo costo per cammino da nodo } u \text{ a nodo } v \text{ con nodi intermedi in } \{0,\ldots,k-1\} & \text{Se un tale cammino esiste.} \\ +\infty & \text{Altrimenti.} \end{cases}$$

- Chiaramente $T[n][u][v]$ è il costo minimo di un cammino da u a v.
- Resta da trovare la funzione ricorsiva che permette di calcolare le $\Theta(n^3)$ celle della matrice tridimensionale in funzione delle celle già calcolate.

$$T[k][u][v] = \begin{cases} \text{Minimo costo per cammino da nodo } u \text{ a nodo } v \text{ con nodi intermedi in } \{0,\ldots,k-1\} & \text{Se un tale cammino esiste.} \\ +\infty & \text{Altrimenti.} \end{cases}$$

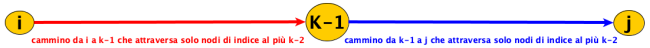
Per ottenere $T[k][i][j]$ possiamo usare la seguente relazione:

$$T[k][i][j] = \begin{cases} 0 & \text{Se } k=0 \text{ e } i=j \\ +\infty & \text{Se } k=0 \text{ e } (i,j) \notin E \\ costo(i,j) & \text{Se } k=0 \\ \min\{\ T[k-1][i][j],\ \ T[k-1][i][k-1]+T[[k-1][k-1][j]\ \} & \text{altrimenti.} \end{cases}$$

la ricorrenza viene fuori dal seguente ragionamento:

- Notate che se $k=0$ allora il cammino minimo da i a j non deve attraversare alcun nodo:
  - se i=j allora questo cammino costerà 0 (partenza ed arrivo coincidono ed è inutile spostarsi infatti un qualunque cammino che si allontani da i per poi tornarci compirebbe un ciclo il cui costo non può essere negativo
  - se $i!=j$ e (i,j) non è un un arco di G allora questo cammino costerà $+\infty$ perché non c'è possibilità di raggiungere j a partire da i senza eventualmente attraversare nodi intermedi.
  - se $i!=j$ e (i,j) è un un arco di G allora questo cammino costerà $costo(i,j)$ perché attraversare quell'arco è l'unico modo di raggiungere j da i senza toccare nodi intermedi.
- i cammini che vanno da i a j passando eventualmente per nodi dell'insieme $\{0,1\ldots k-1\}$ sono di due tipi:
  - non toccano il nodo $k-1$. In questo caso il loro costo minimo è $T[k-1,i,j]$
  - toccano il nodo $k-1$ allora il loro costo minimo è $T[k-1,\ i,\ k-1]+T[k-1,\ k-1,\ j]$ Nota infatti che questi cammini toccano il nodo $k-1$ una sola volta (in caso contrario nel cammino sarebbe presente un ciclo e in G non abbiamo cicli negativi)

il costo minimo per entrambi i tipi di cammini è dunque $\min\{T[k-1,\ i,\ j],\ T[k-1,\ i,\ k-1]+T[k-1,\ k-1,\ j]\}$



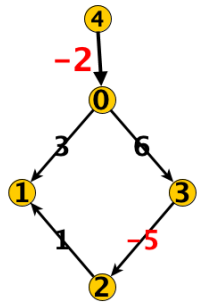
$$T[k][i][j] = \begin{cases} 0 & \text{Se } k=0 \text{ e } i=j \\ +\infty & \text{Se } k=0 \text{ e } (i,\ j) \notin E \\ costo(i,j) & \text{Se } k=0 \\ \min\{\ T[k-1][i][j],\ \ T[k-1][i][k-1]+T[[k-1][k-1][j]\ \} & \text{altrimenti.} \end{cases}$$

Implementazione:

```
def cammini(G):
    from math import inf
    n=len(G)
    T=[ [[inf for _ in range(n)] for _ in range(n)]for _ in range(n+1)]
    for i in G:
        for j,costo in G[i]: T[0][i][j]=costo
    for i in G: T[0][i][i]=0
    for k in range(1,n+1):
        for i in range(n):
            for j in range(n):
                T[k][i][j]=min(T[k-1][i][j],T[k-1][i][k-1]+T[k-1][k-1][j])
    return T[n]

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

>>> cammini(G)
[
[0,    2,   1,   6, inf],
[inf,  0, inf, inf, inf],
[inf,  1,   0, inf, inf],
[inf, -4,  -5,   0, inf],
[-2,   0,  -1,   4,   0]
]
```



- inizializzare la tabella costa $\Theta(n^3)$
- riempire le celle della tabella grazie alla formula di ricorrenza costa $\Theta(n^3)$

la complessità è $\Theta(n^3)$

OTTIMIZZAZIONI:

- Per memorizzare anche i cammini minimi oltre il loro valore è possibile utilizzare la matrice P dei predecessori di dimensione $n \times n$.
  - gli n^2 valori della matrice P vengono inizializzati come segue:
    $P[i][j] = \begin{cases} i & \text{se } i = j \\ -1 & \text{altrimenti} \end{cases}$
  - Durante l'esecuzione dell'algoritmo memorizziamo in P per ogni coppia i, j l'indice k dell'ultimo aggiornamento di $T[\cdot][i][j]$.
  - Al termine $P[i][j]$ conterrà il predecessore del nodo j nel cammino di costo minimo da i a j.
- Abbiamo assunto che il grafo su cui lavora l'algoritmo non contiene cicli negativi ma l'algoritmo di Floyd-Warshall permette di rivelarli. Infatti se non ci sono cicli, per ogni i, avremo che $T[n][i][i] = 0$. Mentre se c'è un ciclo di peso negativo allora per ogni nodo i che appartiene a un tale ciclo risulterà $T[n][i][i] < 0$, perché c'è un cammino da i a i di peso negativo.
- La memoria usata dall'algoritmo è $O(n^3)$, però possiamo risparliarne un bel pò. Considerando che il calcolo di $T[k][i][j]$ richiede solo i valori di $T[k-1][\cdot][\cdot]$ possiamo lavorare su di una matrice T di dimensione $O(n^2)$ (eliminando l'indice k).