Programmazione dinamica (II parte)

Progettazione di Algoritmi a.a. 2022-23

Matricole congrue a 1

Docente: Annalisa De Bonis

Subset sums

Input
- n job 1,2,…,n
  - il job i richiede tempo $w_i > 0$.
- Un limite W al tempo per il quale il processore puo` essere utilizzato

- Obiettivo: selezionare un sottoinsieme S degli n job tale che $\sum_{i\in S} w_i$ sia quanto piu` grande e` possibile, con il vincolo $\sum_{i\in S} w_i \leq W$

Greedy 1: ad ogni passo inserisce in S il job con peso piu` alto in modo che la durata complessiva dei job in S non superi W
Esempio: Input una volta ordinato [W/2+1,W/2,W/2]. L'algoritmo greedy seleziona solo il primo mentre la soluzione ottima e` formata dagli ultimi due.
Greedy 2: ad ogni passo inserisce in S il job con peso piu` basso in modo che la duranta complessiva dei job in S non superi W
Esempio: Input [1,W/2,W/2] una volta ordinato . L'algoritmo greedy seleziona i primi due per un peso complessivo di 1+W/2. mentre la soluzione ottima e` formata dagli ultimi due di peso complessivo W.

Programmazione dinamica: falsa partenza

Def. OPT(i) = valore della soluzione ottima per {1, …, i}.

- Caso 1: La soluzione ottima per {1, …, i} non include i.
  - La soluzione ottima per {1, …, i} e` la soluzione ottima per { 1, 2, …, i-1 }
- Caso 2: La soluzione ottima per {1, …, i} include i.
  - Prendere i non implica immediatamente l'esclusione di altri elementi.
  - Cio` che sappiamo e` che se viene eseguito i allora rimane un tempo complessivo per eseguire i restanti job pari al tempo che avevo prima meno w_i
  - Il parametro i non e` sufficiente a descrivere la sottostruttura ottima del problema
- Conclusione. Approccio sbagliato!

Programmazione dinamica: approccio corretto

- Per esprimere il valore della soluzione ottima per un certo i in termini dei valori delle soluzioni ottime per input piu` piccoli di i, dobbiamo introdurre un limite al tempo totale da dedicare all'esecuzione dei job che precedono i.
- Per ciascun j, consideriamo il valore della soluzione ottima per i job 1, …, j con il vincolo che il tempo necessario per eseguire i job nella soluzione non superi un certo w.
- Def. OPT(i, w) = valore della soluzione ottima per i job 1, …, i con limite w sul tempo di utilizzo del processore.

Programmazione dinamica: approccio corretto

Def. OPT(i,w) = valore della soluzione ottima per i job 1, …, i con limite w sul tempo di utilizzo del processore.

- Caso 1: La soluzione ottima per i primi i job, con limite di utilizzo w non include il job i.
  - La soluzione ottima e` in questo caso la soluzione ottima per { 1, 2, …, i-1 } con limite di utilizzo w → in questo caso OPT(i, w) = OPT(i-1, w)
- Caso 2: La soluzione ottima per i primi i job, con limite di utilizzo w include il job i.
  - La soluzione ottima include la soluzione ottima per { 1, 2, …, i–1 } con limite di utilizzo w– w_i → in questo caso OPT(i, w) = w_i +OPT(i-1, w-w_i)

La soluzione ottima per i primi i job con limite di utilizzo w va ricercata tra le soluzioni ottime per i due casi. Questo pero` se i>0 e $w_i \leq$ w.

Se i=0, banalmente si ha OPT(i,w)=0. Se w_i >w, e` possibile solo il caso 1 perche' il job i non puo` far parte della soluzione in quanto richiede piu` tempo di quello a disposizione.

$$OPT(i,w) = \begin{cases} 0 & \text{if } i=0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\left\{ OPT(i-1,w),\ \ w_i + OPT(i-1, w-w_i) \right\} & \text{otherwise} \end{cases}$$

Subset sums: algoritmo

- Versione iterativa in cui si computa la soluzione in modo bottom-up
- Si riempie un array bidimensionale nxW a partire dalle locazioni di indice di riga i piu` piccolo

SubsetSums(n,w_1,…,w_n,W)

```
for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}

return M[n, W]
```

Subset sums: esempio di esecuzione dell'algoritmo

Limite W = 6, durate job $w_1 = 2$, $w_2 = 2$, $w_3 = 3$

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1							
2							
3							

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2
2							
3							

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2
2	0	0	2	2	4	4	4
3							

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2
2	0	0	2	2	4	4	4
3	0	0	2	3	4	5	5

Subset sums: correttezza algoritmo

Induzione sui i. Dimostriamo che dopo i iterazioni del for piu` esterno ogni riga di M con indice r compreso tra 0 e i contiene i valori OPT(r,0),OPT(r,1),….,OPT(r, W)

- Base induzione. i=0: La riga 0 ha correttamente tutte le entrate uguali a 0. Per cui M[0,w]=0=OPT(0,w) per ogni w.
- Passo induttivo:
  - Ipotesi induttiva. Supponiamo che all'iterazione i-1 ≥0, ciascuna riga con indice r compreso tra 0 e i-1 contenga correttamente i valori OPT(r,0),OPT(r,1),….,OPT(r, W).
  - L'ipotesi induttiva implica M[i-1, w]=OPT(i-1,w) ed M[i-1, w-w_i]=OPT(i-1,w-w_i)
  - Vediamo cosa succede all'i-esima iterazione. All'i-esima iterazione, l'algoritmo setta M[i,w] come segue:

  se w_i > w, M[i, w] =M[i-1,w] che per ipotesi induttiva e` OPT(i-1,w),
  altrimenti, M[i, w]= max {M[i-1, w], w_i + M[i-1, w-w_i]}
  che per ipotesi induttiva e`max{OPT(i-1,w),w_i +OPT(i-1,w-w_i)}

  Quindi anche per i, M[i,w] e` uguale al valore fornito dalla relazione di ricorrenza di OPT(i,w). Per cui M[i,w] = OPT(i,w)

Subset sums: tempo di esecuzione algoritmo

- Tempo di esecuzione. $\Theta(n\ W)$.

- Non e` polinomiale nella dimensione dell'input!
- "Pseudo-polinomiale": L'algoritmo e` efficiente quando W ha un valore ragionevolmente piccolo.

31

Algoritmo ricorsivo per subset sums

```
for i=0 to n
   for w=0 to W
    M[i,w]=empty //M array globale

SubsetSums(i,w):
   if i=0
     M[i,w]=0
   if M[i,w] ≠ empty
     return M[i,w]
   if w_i > w
      M[i,w]=SubsetSums(i-1,w)
   else
      M[i,w]= max{SubsetSums(i-1,w), w_i+ SubsetSums(i-1, w-w_i)}
   return M[i,w]
```

La prima volta invocata con i=n e w=W

32

Algoritmo ricorsivo che stampa la soluzione di subset sums

Supponiamo di aver gia` invocato SubsetSums (una delle due versioni)

```
FindSubset(i,w):
   if i=0
      return
   if M[i,w] =M[i-1,w]
      FindSubset(i-1,w)
   else
     print i
     FindSubset(i-1, w-w_i)
```

prima volta invocata con i=n e w=W

tempo O(n)

perche'?

33

Problema dello zaino

- Input
  - n oggetti ed uno zaino
  - L'oggetto i pesa $w_i > 0$ chili e ha valore $v_i > 0$.
  - Lo zaino puo` trasportare fino a W chili.
- Obiettivo: riempire lo zaino in modo da massimizzare il valore totale degli oggetti inseriti senza eccedere il limite W.
- Esempio: { 3, 4 } ha valore 40.

W = 11

Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: seleziona ad ogni passo l'oggetto con il rapporto v_i/w_i piu` grande in modo che il peso totale dei pesi selezionati non superi w

Esempio: soluzione greedy { 5, 2, 1 } ha valore = 35 $\Rightarrow$ greedy non e` ottimo

34

Problema dello zaino

Input

- n oggetti: l'oggetto i pesa w_i > 0 chili e ha valore v_i > 0
- limite W

Obiettivo: selezionare un sottoinsieme S degli n oggetti in modo

- da rispettare il vincolo $\sum_{i \in S} w_i \le W$, cioe` che la somma dei pesi degli oggetti selezionati sia minore di W
- e da massimizzare $\sum_{i \in S} v_i$

Corrisponde al problema subset sums quanto v_i=w_i per ogni i.

Problema dello zaino: estensione approccio usato per Subset Sums

Def. OPT(i, w) = valore della soluzione ottima per gli oggetti 1, …, i con limite di peso totale w.

- Caso 1: La soluzione ottima per i primi i oggetti, con limite di utilizzo w non include l'oggetto i.
  - La soluzione ottima e` in questo caso la soluzione ottima per { 1, 2, …, i-1 } con limite di utilizzo w → in questo caso OPT(i, w) = OPT(i-1, w)
- Caso 2: La soluzione ottima per i primi i oggetti, con limite di utilizzo w include l'oggetto i.
  - La soluzione ottima include la soluzione ottima per { 1, 2, …, i-1 } con limite di utilizzo w– w_i → in questo caso OPT(i, w) = v_i +OPT(i-1, w-w_i)

La soluzione ottima per i primi i oggetti con limite di utilizzo w va ricercata tra le soluzioni ottime per i due casi. Questo pero` se i>0 e w_i <w.

Se i=0, banalmente si ha OPT(i,w)=0. Se w_i >w, e` possibile solo il caso 1 perche' i non puo` far parte della soluzione in quanto ha peso maggiore del peso trasportabile.

$$OPT(i,w) = \begin{cases} 0 & \text{if } i=0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i) \} & \text{otherwise} \end{cases}$$

Problema dello zaino: algoritmo

Knapsack ($n, w_1, \ldots, w_n, v_1, \ldots, v_n, W$)

```
for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```


Algoritmo per il problema della zaino: esempio

W →

n ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(5,11) = OPT(4,11) = v_4+OPT(3,11-w_4) = v_4+OPT(3,5) = v_4+v_3+OPT(2,0) = v_4+v_3+OPT(1,0)
= v_4+v_3+OPT(0,0) = 22+18+0 = 40

Soluzione ottima : { 4, 3 }
Valore soluzione ottima = 22 + 18 = 40

W = 11

Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Problema dello zaino: tempo di esecuzione algoritmo

- Tempo di esecuzione. $\Theta(n\ W)$.

- Non e` polinomiale nella dimensione dell'input!
- "Pseudo-polinomiale": L'algoritmo e` efficiente quando W ha un valore ragionevolmente piccolo.

- Se volessimo produrre la soluzione ottima, potremmo scrivere un algoritmo simile a quelli visti prima in cui la soluzione ottima si ricostruisce andando a ritroso nella matrice M. Tempo $O(n)$.

- Esercizio: Scrivere lo pseudocodice dell'algoritmo che produce la soluzione ottima per un'istanza del problema dello zaino.

- Esercizio: Scrivere la versione ricorsiva dell'algoritmo di programmazione dinamica per il problema dello zaino.