Principi della programmazione dinamica

Sapienza Università di Roma

Indice

Perché non enumerare tutte le soluzioni?

Dividere, conquistare, combinare

Proprietà del problema richieste dagli algoritmi di PD

Passi per progettare un algoritmo di PD

Gli algoritmi di PD di questo modulo

Errori tipici

In sintesi

Obiettivi

Alla fine di questo capitolo saremo in grado di:

- spiegare perché l'enumerazione di tutte le soluzioni ammissibili non funziona, e che cosa fa invece un algoritmo di PD;
- enunciare i principi di divisione, conquista e combinazione, e la differenza fra sottoproblemi indipendenti e sottoproblemi sovrapposti;
- verificare su un problema la sottostruttura ottima e i sottoproblemi sovrapposti;
- elencare i passi di un algoritmo di PD: ricorrenza, casi base, tabella di memorizzazione e ricostruzione di una soluzione ottima.

Prerequisiti

- Dal modulo *Problemi di ottimizzazione combinatoria*, il capitolo «Ottimizzazione combinatoria: definizioni»: i problemi di ottimizzazione combinatoria, le loro soluzioni ammissibili e il fatto che il loro numero cresce in modo esponenziale.
- Dal modulo *Problemi di ottimizzazione combinatoria*, il capitolo «Problemi di zaino»: il problema dello zaino 0/1 (KP), l'esempio guida degli algoritmi di PD.
- Dal modulo *Algoritmi*: pseudocodice (Capitolo 2), ricorsione (Capitolo 14) e la crescita del tempo di esecuzione di un algoritmo (Capitolo 10 «Crescita asintotica»).

Perché non enumerare tutte le soluzioni?

Introduzione: perché non enumerare tutte le soluzioni?

- L'algoritmo esatto più semplice elenca tutte le soluzioni ammissibili e tiene la migliore
- Vediamo quanto cresce in fretta il suo tempo di esecuzione con la dimensione dell'istanza
- Lo confrontiamo con algoritmi il cui numero di passi cresce come n e come n^2
- Da qui nasce un metodo esatto più furbo: gli algoritmi di PD di questa parte del corso

Algoritmi a forza bruta

- Un **algoritmo a forza bruta** elenca tutte le soluzioni ammissibili di un problema di ottimizzazione combinatoria.
- Il loro numero cresce in modo esponenziale: per il KP ci sono 2^n sottoinsiemi di oggetti.
- La tabella seguente mostra come cresce il tempo di esecuzione di un algoritmo con $\mathcal{O}(n)$, $\mathcal{O}(n^2)$ e $\mathcal{O}(2^n)$ passi al crescere della dimensione n dell'istanza, a $1\,\mu$s per passo.

Tempi di esecuzione a 1 μs per passo

	Dimensione n					
Algoritmo	10	20	30	40	50	60
$\mathcal{O}(n)$	0,00001 secondi	0,00002 secondi	0,00003 secondi	0,00004 secondi	0,00005 secondi	0,00006 secondi
$\mathcal{O}(n^2)$	0,0001 secondi	0,0004 secondi	0,0009 secondi	0,0016 secondi	0,0025 secondi	0,0036 secondi
$\mathcal{O}(2^n)$	0,001 secondi	1,0 secondo	17,9 minuti	12,7 giorni	35,7 anni	366 secoli

- Gli algoritmi polinomiali restano sotto il centesimo di secondo.
- Quello esponenziale va da un millisecondo a secoli.

La tecnologia non aiuterà

- La tecnologia non aiuterà.[1]
- Con una CPU 1000 volte più veloce, in un'ora:
  - un algoritmo $\mathcal{O}(n)$ risolve un'istanza 1000 volte più grande;
  - un algoritmo $\mathcal{O}(2^n)$ risolve un'istanza più grande di sole 10 unità (perché $2^{10} = 1024$).

[1] M. Garey, D. Johnson, *Computers and Intractability: A Guide to NP-Completeness*.

La forza bruta non funziona

Gli algoritmi a forza bruta non funzionano:

un algoritmo di PD evita l'enumerazione
riusando i valori ottimi dei sottoproblemi.

Riepilogo

Forza bruta:

- Elenca tutte le soluzioni ammissibili: 2^n sottoinsiemi di oggetti per il KP
- Con $\mathcal{O}(2^n)$ passi il tempo di esecuzione esplode: secoli per $n = 60$
- Una CPU 1000 volte più veloce guadagna solo 10 unità di dimensione

La via d'uscita:

- Un algoritmo di PD riusa i valori ottimi dei sottoproblemi invece di enumerare

Dividere, conquistare, combinare

Introduzione: dividere, conquistare, combinare

- In questa sezione definiamo che cos'è un algoritmo di PD
- Vediamo i tre principi del paradigma su cui si basa: dividere, conquistare e combinare
- Incontriamo l'idea chiave che rende efficiente la PD: il valore ottimo di ogni sottoproblema si memorizza in una tabella di memorizzazione e non si ricalcola mai

Algoritmi di PD

Definizione 1: Algoritmo di PD

*Un **algoritmo di programmazione dinamica (PD)** è un algoritmo esatto per trovare una soluzione ottima di una classe di problemi di ottimizzazione con proprietà specifiche (sezione successiva). Decompone il problema originale in sottoproblemi più piccoli dello stesso tipo e riusa l'informazione calcolata risolvendoli.*

Forza bruta e PD

- Un algoritmo a forza bruta elenca tutte le soluzioni ammissibili, e il loro numero cresce in modo esponenziale (2^n sottoinsiemi di oggetti per il KP).
- Un algoritmo di PD evita questa enumerazione riusando i valori ottimi dei sottoproblemi.

Come decompone un problema un algoritmo di PD?
Segue un paradigma classico: **dividere, conquistare e combinare**.

Il paradigma divide et impera

Un algoritmo di PD sfrutta il ***paradigma divide et impera*** (dividere, conquistare e combinare), i cui principi sono:

1. **Principio di divisione** $\rightsquigarrow$ decomporre il problema in un certo numero di sottoproblemi, che sono problemi più piccoli dello stesso tipo;
2. **Principio di conquista** $\rightsquigarrow$ risolvere i sottoproblemi in modo ricorsivo. Se le dimensioni dei sottoproblemi sono abbastanza piccole, risolverli direttamente;
3. **Principio di combinazione** $\rightsquigarrow$ combinare soluzioni ottime dei sottoproblemi per ottenere una soluzione ottima del problema originale.

Sottoproblemi indipendenti

- Gli algoritmi di PD dividono il problema in ***sottoproblemi indipendenti***, li risolvono ricorsivamente e poi ne combinano le soluzioni per risolvere il problema originale.

- Qui **indipendenti** significa che ogni sottoproblema generato dalla divisione si può risolvere senza influire sugli altri.

La tabella di memorizzazione

- In un algoritmo di PD lo stesso sottoproblema può essere generato più volte durante il processo ricorsivo di soluzione.
- Per evitare calcoli ridondanti, il valore ottimo di ogni sottoproblema si memorizza in una ***tabella di memorizzazione***.
- Ogni volta che lo stesso sottoproblema viene generato di nuovo, il suo valore ottimo si legge dalla tabella di memorizzazione invece di essere ricalcolato.

Riepilogo

Algoritmo di PD:

- Un algoritmo esatto che decompone un problema in sottoproblemi più piccoli dello stesso tipo e riusa l'informazione calcolata risolvendoli

Dividere, conquistare, combinare:

- Dividere in sottoproblemi più piccoli dello stesso tipo
- Conquistarli ricorsivamente (direttamente quando sono abbastanza piccoli)
- Combinarne le soluzioni ottime in una soluzione ottima del problema

Tabella di memorizzazione:

- Il valore ottimo di ogni sottoproblema si calcola una volta e poi si legge

Proprietà del problema richieste dagli algoritmi di PD

Introduzione: proprietà del problema richieste dagli algoritmi di PD

- La PD non funziona per ogni problema: la decomposizione ricorsiva deve avere due proprietà
- Per ciascuna proprietà vediamo l'idea e i punti da controllare in pratica
- Verificarle è la prima cosa da fare prima di progettare un algoritmo di PD

Due proprietà

Gli algoritmi di PD funzionano solo per problemi di ottimizzazione combinatoria che ammettono un processo di decomposizione ricorsiva con le seguenti proprietà:

1. ***Sottostruttura ottima***;

2. ***Sottoproblemi sovrapposti***.

Sottostruttura ottima

Sottostruttura ottima $\rightsquigarrow$ l'idea è che, quando un problema è decomposto in sottoproblemi più piccoli, una soluzione ottima del problema si può costruire a partire da soluzioni ottime dei sottoproblemi.

In pratica, per verificare questa proprietà si controllano di solito i seguenti punti:

- il problema si può decomporre in sottoproblemi più piccoli dello stesso tipo;
- i sottoproblemi si possono risolvere in modo indipendente;
- le soluzioni ottime dei sottoproblemi si possono combinare per ottenere una soluzione ottima del problema originale;
- si può stabilire una **relazione di ricorrenza** per calcolare il valore ottimo del problema.

Sottoproblemi sovrapposti

Sottoproblemi sovrapposti $\rightsquigarrow$ l'idea è che, quando un problema è decomposto ricorsivamente in sottoproblemi più piccoli, gli stessi sottoproblemi possono essere generati più volte.

In pratica, per verificare questa proprietà si controllano di solito i seguenti punti:

- il processo di decomposizione ricorsiva genera lo stesso sottoproblema più di una volta;
- è possibile memorizzare i valori ottimi dei sottoproblemi in una tabella di memorizzazione; allora, ogni volta che lo stesso sottoproblema serve di nuovo, il suo valore ottimo si legge dalla tabella invece di essere ricalcolato.

Quando si possono usare gli algoritmi di PD?

Gli algoritmi di PD si possono usare quando un problema ha sia

la sottostruttura ottima

sia

i sottoproblemi sovrapposti.

Riepilogo

Sottostruttura ottima:

- Una soluzione ottima del problema si costruisce da soluzioni ottime dei sottoproblemi
- Controllo: stesso tipo, sottoproblemi indipendenti, combinazione, relazione di ricorrenza

Sottoproblemi sovrapposti:

- La decomposizione ricorsiva genera lo stesso sottoproblema più volte
- Controllo: sottoproblemi ripetuti, valori ottimi memorizzati in una tabella di memorizzazione

La PD ha bisogno di entrambe le proprietà

Passi per progettare un algoritmo di PD

Introduzione: passi per progettare un algoritmo di PD

- Mettiamo insieme i principi in una ricetta per progettare un algoritmo di PD
- Prima la decomposizione ricorsiva e il controllo delle due proprietà, poi due passi principali
- Ogni algoritmo di PD che vedremo più avanti nel corso segue questa ricetta

Il processo di decomposizione ricorsiva

Identificare il ***processo di decomposizione ricorsiva*** che, a partire da un problema dato, genera i suoi sottoproblemi, che sono problemi più piccoli dello stesso tipo, e verificare le due proprietà seguenti:

1. la proprietà di sottostruttura ottima, controllando che una soluzione ottima del problema si possa ottenere combinando le soluzioni ottime dei suoi sottoproblemi;
2. la proprietà dei sottoproblemi sovrapposti, controllando se il processo di decomposizione ricorsiva genera più volte gli stessi sottoproblemi.

I due passi principali

La costruzione di un algoritmo di PD segue di solito due passi principali:

1. calcolo del **valore ottimo**;
2. costruzione di una **soluzione ottima**.

Vediamo ora i due passi uno alla volta.

Passo 1: calcolo del valore ottimo

- definire una ***relazione di ricorrenza*** che esprime il valore ottimo di un problema in funzione dei valori ottimi dei suoi sottoproblemi;
- specificare i ***casi base***, ossia i sottoproblemi la cui soluzione ottima è immediata (il valore ottimo ne segue direttamente);
- determinare un algoritmo di PD che calcoli i valori ottimi di tutti i sottoproblemi rilevanti, di solito ***dal basso verso l'alto*** (bottom-up), memorizzando ogni valore in una tabella di memorizzazione, in modo che sia calcolato una sola volta e letto ogni volta che serve di nuovo.

Passo 2: costruzione di una soluzione ottima

- specificare l'informazione aggiuntiva da memorizzare per ricostruire una soluzione ottima del problema;
- identificare il modo di costruire una soluzione ottima dall'informazione memorizzata.

Riepilogo

Prima di progettare:

- Identificare il processo di decomposizione ricorsiva e verificare le due proprietà

Passo 1, il valore ottimo:

- Relazione di ricorrenza, casi base, calcolo bottom-up con una tabella di memorizzazione

Passo 2, una soluzione ottima:

- Informazione aggiuntiva memorizzata, e il modo di costruirne la soluzione

Gli algoritmi di PD di questo modulo

Introduzione: gli algoritmi di PD di questo modulo

- Una mappa degli algoritmi di PD che vedremo più avanti nel corso
- Per ogni problema: lo stato della tabella, su che cosa corre la ricorrenza e il numero di passi
- Vediamo perché questi tempi di esecuzione si chiamano pseudo-polinomiali

Due passi per ogni algoritmo

- Ogni algoritmo di PD di questo modulo segue i due passi precedenti: una ricorrenza con i suoi casi base dà il valore ottimo, una tabella aggiuntiva dà una soluzione ottima.
- Nelle prossime slide: i problemi e, per ciascuno, lo stato della tabella, su che cosa corre la ricorrenza e il numero di passi.

I problemi

Sigla	Problema
KP	problema dello zaino 0/1
UKP	problema dello zaino illimitato
BKP	problema dello zaino limitato
SSP	problema del subset sum
MCKP	problema dello zaino a scelta multipla
MDKP	problema dello zaino multidimensionale
KPS	problema dello zaino con setup
MKP	problema degli zaini multipli
RCP	problema del taglio delle aste
CCP	problema del cambio monete
ULS	problema del lot sizing senza capacità
CLS	problema del lot sizing capacitato

Stati, ricorrenze e passi (1/2)

Problema	Stato	Ricorrenza su
KP	$v(j,k)$	oggetti e capacità
UKP	$v(k)$	capacità
BKP	$v(j,k)$	oggetti e capacità
SSP	$r(j,k)$	oggetti e valori da raggiungere
MCKP	$v(h,k)$	classi e capacità
MDKP	$v(j,k_1,\ldots,k_m)$	oggetti e vettori di capacità
KPS	$v(h,k)$, $u(h,i,k)$	classi, oggetti e capacità
MKP	$v(j,k_1,\ldots,k_m)$	oggetti e capacità residue
RCP	$v(\ell)$	lunghezze dell'asta
CCP	$v(k)$	importi del resto
ULS	$v(t)$	periodi
CLS	$v(t,k)$	periodi e scorte

Stati, ricorrenze e passi (2/2)

Problema	Stato	Passi
KP	$v(j,k)$	$\Theta(nc)$
UKP	$v(k)$	$\Theta(nc)$
BKP	$v(j,k)$	$\Theta(nc\, b_{\max})$
SSP	$r(j,k)$	$\Theta(nc)$
MCKP	$v(h,k)$	$\Theta(nc)$
MDKP	$v(j,k_1,\ldots,k_m)$	$\Theta(m\,n\prod_d(c_d+1))$
KPS	$v(h,k),\ u(h,i,k)$	$\Theta(nc)$
MKP	$v(j,k_1,\ldots,k_m)$	$\Theta(m\,n\prod_i(c_i+1))$
RCP	$v(\ell)$	$\Theta(n^2)$
CCP	$v(k)$	$\Theta(nc)$
ULS	$v(t)$	$\Theta(m^2)$
CLS	$v(t,k)$	$O(mDC)$

Tempi di esecuzione pseudo-polinomiali

- In tutti gli esempi numerici i dati sono interi: il numero di stati è finito, e ogni stato è una cella della tabella.
- I tempi di esecuzione $\Theta(nc)$, $\Theta(nc\, b_{\max})$ e $O(mDC)$ contengono numeri dell'input (una capacità, un bound, una domanda), non solo la sua dimensione: sono **pseudo-polinomiali**.
- Con un vettore di capacità (MDKP, MKP) la tabella ha una dimensione per ogni capacità: il numero di passi è pseudo-polinomiale solo per un numero fissato m di dimensioni o di zaini.

Riepilogo

Gli algoritmi di PD di questo modulo:

- Seguono tutti i due passi: ricorrenza e casi base per il valore ottimo, una tabella aggiuntiva per una soluzione ottima
- Problemi di zaino, taglio delle aste, cambio monete e lot sizing

Tempi di esecuzione:

- Una cella della tabella per ogni stato
- Numeri dell'input nel tempo di esecuzione: pseudo-polinomiale
- Vettori di capacità: pseudo-polinomiale solo per m fissato

Errori tipici

Introduzione: errori tipici

- Elenchiamo gli errori che si commettono più spesso usando la PD
- Riguardano le due proprietà, i casi base e la differenza fra il valore ottimo e una soluzione ottima
- Conoscerli aiuta a progettare algoritmi di PD corretti nei capitoli che seguono

Osservazione 1: Errori tipici (1/2)

- ***Usare la PD senza la sottostruttura ottima.*** *Se una soluzione ottima del problema non si può costruire da soluzioni ottime dei sottoproblemi, la ricorrenza è sbagliata.*
- ***Usare la PD senza sottoproblemi sovrapposti.*** *Se ogni sottoproblema compare una sola volta, memorizzare i valori in una tabella non porta alcun vantaggio: l'algoritmo di PD è una semplice ricorsione.*

Osservazione: Errori tipici (2/2)

- ***Dimenticare i casi base.*** *La ricorrenza ha bisogno dei sottoproblemi il cui valore ottimo è immediato, altrimenti la ricorsione non termina.*
- ***Confondere il valore ottimo con una soluzione ottima.*** *La tabella dei valori ottimi non dà la soluzione: serve una tabella aggiuntiva delle decisioni.*

Riepilogo

Errori tipici:

- Senza sottostruttura ottima: la ricorrenza è sbagliata
- Senza sottoproblemi sovrapposti: la tabella di memorizzazione non serve
- Senza casi base: la ricorsione non termina
- La tabella dei valori ottimi non è una soluzione ottima: si memorizzano anche le decisioni

In sintesi

Introduzione: in sintesi

- Raccogliamo le idee chiave del capitolo
- Sono la base comune di tutti gli algoritmi di PD che seguono nel corso

Riepilogo

- Un algoritmo di PD è un algoritmo esatto che divide un problema in sottoproblemi dello stesso tipo, li risolve ricorsivamente e combina le soluzioni ottime.
- Funziona quando il problema ha la sottostruttura ottima e sottoproblemi sovrapposti; i valori ottimi si memorizzano in una tabella di memorizzazione.
- La progettazione ha due passi: il valore ottimo (ricorrenza, casi base, tabella bottom-up) e una soluzione ottima (decisioni memorizzate e lette a ritroso).
- La forza bruta richiede tempo esponenziale; la PD richiede un tempo polinomiale nel numero di stati, e pseudo-polinomiale quando un numero dell'input è una dimensione della tabella.