O problema subset-sum

Esta página discute o problema *subset-sum*, também conhecido como *exact subset-sum*. Trata-se de um caso especial do problema da mochila booleana. Poderíamos chamar o problema de "soma de subconjunto", mas é melhor usar o nome tradicional em inglês.

O problema é mencionado nas seções 34.5.5 e 35.5 do CLRS. Veja também a seção 6.1 de KT. Veja ainda o verbete *Subset sum problem* na Wikipedia.

O problema

Dados números $p_1, p_2, \ldots, p_n$ e um subconjunto X de $\{1,2,\ldots,n\}$, denotaremos por $p(X)$ a soma $\sum_{i \in X} p_i$. Considere o seguinte problema:

> PROBLEMA SUBSET-SUM: Dados números naturais $p_1, p_2, \ldots, p_n$ e c, decidir se existe um subconjunto X de $\{1,2,\ldots,n\}$ tal que $p(X) = c$.

Diremos que os números $p_1,p_2,\ldots,p_n$ são os *pesos* e c é a *capacidade* do problema. (A ordem em que os pesos são dados é, obviamente, irrelevante.) O problema só admite duas soluções, "sim" e "não", que podemos representar por 1 e 0 respectivamente.

Alguns exemplos:

- A instância do problema definida por $p = (30, 40, 10, 15, 10, 60, 54)$ e $c = 55$ tem solução "sim" pois o conjunto $\{2,4\}$ tem a propriedade desejada. O conjunto $\{1,4,5\}$ também tem a propriedade desejada.
- O problema faz sentido mesmo quando n é 0. Uma instância com $n = 0$ tem solução "sim" se e somente se c vale 0.
- O problema dos cheques: Suponha que $p_1, p_2, \ldots, p_n$ são os valores dos cheques que você emitiu durante o mês. No fim do mês, o banco debita uma quantia c na sua conta. Você quer saber se algum conjunto de cheques corresponde ao valor debitado.

Exercícios

1. Problema: dados números naturais $p_1,p_2,\ldots,p_n$ e c, decidir se existem $i \neq j$ tais que $p_i + p_j = c$. Escreva um algoritmo que resolva o problema em $\Theta(n \lg n)$ unidades de tempo.
2. Discuta a relação entre o problema subset-sum e o seguinte problema: dada uma sequência $A[1..n]$ de números naturais e um número natural c, decidir se existe uma subsequência $B[1..k]$ de $A[1..n]$ tal que $B[1]+B[2]+\ldots+B[k] = c$.
3. Mostre que o objetivo do problema subset-sum pode ser formulado assim: encontrar números x_1,

… , x_n, todos em {0,1}, tais que $x_1 p_1 + \ldots + x_n p_n = c$.

4. Alguns livros formulam o problema assim: dado um número natural c e um conjunto A de números naturais, encontrar um subconjunto B de A tal que $\sum B = c$. Qual a diferença entre esta formulação e nossa formulação oficial? Qual é mais geral? Qual é mais fácil?
5. Considere o seguinte *problema da soma de segmento*: dados números naturais $p_1,\ldots,p_n$ e c, encontrar índices i e k tais que $1 \leq i \leq k \leq n$ e $p_i + \ldots + p_k = c$. Este problema é um caso especial do subset-sum?
6. [HEURÍSTICA GULOSA] Considere a seguinte heurística gulosa para o problema subset-sum. Para $i = 1,2,\ldots n$, se p_i "cabe no espaço disponível", então acrescente i à solução X; senão, ignore i. Descreva esta heurística em pseudocódigo. Mostre que a heurística não resolve o problema. Repita o exercício supondo que os pesos p_i estão em ordem crescente. Repita supondo que os pesos estão em ordem decrescente.
7. [REDUÇÃO ENTRE PROBLEMAS] Suponha que você recebeu de presente um programa muito rápido para o problema subset-sum. Como é possível usar esse programa (sem olhar o seu código) para encontrar um subconjunto X de $\{1,2,\ldots,n\}$ tal que $p(X) = c$? Quantas vezes é preciso executar o programa?

A estrutura recursiva do problema

Como mostraremos a seguir, o problema tem estrutura recursiva, ou seja, toda solução de uma instância é composta por soluções de suas subinstâncias.

Seja $(p_1,p_2,\ldots,p_n, c)$ uma instância do problema. Seja X um subconjunto de $\{1,2,\ldots,n\}$ tal que $p(X) = c$. Temos duas possibilidades: X contém n ou não contém n. No segundo caso, X é solução da subinstância

$$(p_1,p_2,\ldots,p_{n-1}, c)$$

No primeiro caso, $X-\{n\}$ é solução da subinstância

$$(p_1,p_2,\ldots,p_{n-1}, c-p_n).$$

(É claro que esta segunda alternativa só faz sentido se $p_n \leq c$.) Essa estrutura recursiva leva imediatamente ao seguinte algoritmo, que devolve 1 ou 0 conforme exista ou não um conjunto X tal que $p(X) = c$:

SUBSET-SUM-REC (p, n, c)
1 se $n = 0$
2 então se $c = 0$
3 então devolva 1
4 senão devolva 0
5 senão $s \leftarrow$ SUBSET-SUM-REC $(p, n-1, c)$
6 se $s = 0$ e $p_n \leq c$
7 então $s \leftarrow$ SUBSET-SUM-REC $(p, n-1, c-p_n)$
8 devolva s

O consumo de tempo do algoritmo é proporcional ao número de invocações de SUBSET-SUM-REC. (Alternativamente, poderíamos medir o consumo de tempo pelo número de comparações de c com um componente de p na linha 6.) Seja $T(n)$ esse número de invocações no pior caso. Então $T(0) = 0$ e

$$T(n) = 2 + T(n-1) + T(n-1)$$

se $n \geq 1$. A solução dessa recorrência é $T(n) = 2^{n+1}-2$. Assim, o consumo de tempo está em $O(2^n)$. O consumo de tempo também está em

$$\Omega(2^n)$$

no pior caso, o que torna o algoritmo inútil exceto para valores muito pequenos de n.

A cota inferior exponencial não surpreende pois, no pior caso, o algoritmo examina todos os 2^n subconjuntos de $\{1,\ldots,n\}$. Pode-se dizer que SUBSET-SUM-REC é um algoritmo de "enumeração implícita". (Note porém que o espaço "de trabalho" usado pelo algoritmo é apenas $O(n)$, pois a pilha de recursão nunca tem mais que n elementos.)

Exercícios

1. Prove que o problema de fato tem a estrutura recursiva descrita acima.
2. Mostre que a pilha de recursão do algoritmo SUBSET-SUM-REC nunca tem mais que n elementos.
3. Mostre que, no pior caso, o algoritmo SUBSET-SUM-REC examina *todos* os subconjuntos de $\{1,\ldots,n\}$. (Sugestão: Imagine uma instância com resposta "não". Mostre que, para cada subconjunto X de $\{1,\ldots,n\}$, o algoritmo é invocado com segundo argumento 0 e terceiro argumento $c-p(X)$. Por exemplo, quando $n = 4$ e $X = \{2,4\}$, alguma invocação do algoritmo terá argumentos $(p, 0, (c-p_4)-p_2)$.)
4. Mostre que o algoritmo SUBSET-SUM-REC pode recalcular várias vezes a solução de uma mesma instância do problema. (Sugestão: Suponha que $p_n = p_{n-1}$. Então as instâncias $(n-2, c-p_n)$ e $(n-2, c-p_{n-1})$ são idênticas.)
5. Escreva um algoritmo iterativo que resolva o problema subset-sum por enumeração explícita de todos os subconjuntos de $\{1,\ldots n\}$. (Sugestão: Represente cada subconjunto de $\{1,\ldots,n\}$ por um vetor de bits $(b_1,\ldots,b_n)$. Gere cada vetor de bits a partir do anterior como se estivesse somando 1 em notação binária.) Mostre que o seu algoritmo consome tempo $O(2^n)$ e espaço $O(n)$.
6. [ALGORITMO 1.415^n] Suponha que $p(X) = c$ para algum $X \subseteq \{1,\ldots,n\}$. Seja Y o conjunto $\{i \in X : i \leq n/2\}$ e Z o conjunto $\{i \in X : i > n/2\}$. Então $p(Y) = c - p(Z)$. Esta observação leva ao seguinte algoritmo:
   a. Seja m o piso de $n/2$. Seja A o conjunto $\{1,\ldots,m\}$ e B o conjunto $\{m+1,\ldots,n\}$.
   b. Sejam $A_1, A_2, \ldots, A_{2^m}$ todos os subconjuntos de A. Para $k = 1,\ldots,2^m$, seja $s[k]$ o número $c - p(A_k)$.
   c. Rearranje o vetor $s[1..2^m]$ em ordem crescente.
   d. Para cada subconjunto C de B, use busca binária para decidir se a soma $p(C)$ está em $s[1..2^m]$. Em caso afirmativo, devolva 1.
   e. Se nenhuma das somas da forma $p(C)$ estiver em $s[1..2^m]$, devolva 0.

   Descreva o algoritmo em pseudocódigo. Analise o consumo de tempo do algoritmo. Analise o consumo de espaço.

Algoritmo de programação dinâmica

"For every polynomial algorithm you have,
I have an exponential algorithm that I would rather run."
— *Alan Perlis*

O algoritmo Subset-Sum-Rec é ineficiente porque recalcula várias vezes, sem necessidade, a solução de algumas das instâncias do problema. O remédio é usar a técnica da programação dinâmica e guardar as soluções das (sub)instâncias em uma tabela. A tabela, digamos t, é definida assim:

$t[i,b]$ é a solução (0 ou 1)
da instância $(p_1,\ldots,p_i,\ b)$ do problema.

Os possíveis valores de i são $0,1,\ldots,n$ e os possíveis valores de b são $0,1,\ldots,c$ (pois $p_1,\ldots,p_n$ e c são números naturais). Para todo $i \geq 1$ teremos

$$t[i,b] \;=\; \begin{cases} t[i-1,b] & \text{se } p_i > b \quad \text{e} \\ t[i-1,b] \;\vee\; t[i-1,b-p_i] & \text{se } p_i \leq b \end{cases}$$

(onde $\vee$ é o operador lógico OU). Essa recorrência decorre da estrutura recursiva do problema. É fácil usar a recorrência para calcular a tabela t: basta preencher as casas da tabela de "da esquerda para a direita" (ou seja, para valores crescentes de i) e "de cima para baixo" (ou seja, para valores crescentes de b). Desse modo, as casas $(i-1,b)$ e $(i-1,b-p_i)$ já estarão preenchidas quando o algoritmo começar a cuidar da casa (i,b).

Subset-Sum-Prog-Din (p, n, c)
```
para i crescendo de 0 até n faça
    t[i,0] ← 1
para b crescendo de 1 até c faça
    t[0,b] ← 0
    para i crescendo de 1 até n faça
        s ← t[i−1,b]
        se s = 0 e p_i ≤ b
            então s ← t[i−1,b−p_i]
        t[i,b] ← s
devolva t[n,c]
```

Exemplo: Suponha que $n = 4$, $c = 5$ e o vetor p é dado pela figura esquerda. Então, a figura direita representa a tabela t. (A tabela tem um erro que você deve descobrir.)

	1	2	3	4
p	4	2	1	3

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

Desempenho do algoritmo

O consumo de tempo do algoritmo Subset-Sum-Prog-Din é proporcional ao tamanho da tabela t. Como a tabela tem n+1 linhas e c+1 colunas, o algoritmo consome $O(nc)$ unidades de tempo, mesmo no pior caso. (A estimativa é justa, pois o algoritmo consome $\Omega(nc)$ unidades de tempo em todos os casos.)

Para colocar em foco o efeito de c sobre o consumo de tempo, faça o seguinte experimento mental.

Imagine que os números $p_1,\ldots,p_n$ e c são multiplicados por 100. Esta operação é uma mera mudança de escala, e portanto a nova instância é conceitualmente idêntica à original. Apesar disso, o algoritmo consumirá 100 vezes mais tempo para resolver a nova instância.

O algoritmo Subset-Sum-Prog-Din não pode portanto ser considerado rápido. (Em linguagem técnica, o algoritmo é exponencial.) Infelizmente, não se descobriu ainda um algoritmo substancialmente melhor.

Exercícios

1. Se tivesse que programar Subset-Sum-Prog-Din pra valer, que comentários você escreveria junto com o código? [Solução]
2. O algoritmo Subset-Sum-Prog-Din preenche a tabela t "por colunas" (a primeira coluna, depois a segunda, etc.). Escreva uma variante do algoritmo que preencha a tabela "por linhas".
3. Discuta a seguinte ideia. Seja d o máximo divisor comum de $p_1,p_2,\ldots,p_n$ e c. Submeta ao algoritmo Subset-Sum-Prog-Din a instância $(p_1/d, p_2/d, \ldots, p_n/d, c/d)$ do problema.
4. [CLR 36.1-4, CLRS 34.1-4] O algoritmo Subset-Sum-Prog-Din é polinomial?
5. Escreva uma variante do algoritmo Subset-Sum-Prog-Din que devolva um conjunto X tal que $p(X) = c$. Se tal X não existe, o seu algoritmo deve parar sem nada devolver.
6. [Desafio] Descubra um algoritmo que resolva o problema subset-sum em tempo $O(n^4)$.
7. [Desafio] Descubra um algoritmo que resolva o problema subset-sum em tempo $O(d^3)$, sendo d o número total de dígitos decimais de $p_1,\ldots,p_n$ e c.

Problema subset-sum generalizado

Há uma generalização muito natural e útil do problema subset-sum:

> Problema subset-sum generalizado: Dados números naturais $p_1, p_2, \ldots, p_n$ e c, encontrar um subconjunto X de $\{1,2,\ldots,n\}$ que maximize $p(X)$ sob a restrição $p(X) \le c$.

(Imagine um armazém com uma grande quantidade de caixas de pesos variados. Para transportar essas caixas, tenho um caminhão com capacidade para c toneladas. Quero embarcar a maior carga possível no caminhão.)

É claro que qualquer algoritmo para o problema generalizado pode ser usado para resolver o nosso problema subset-sum básico.

Exercícios

1. Discuta a relação entre o problema subset-sum generalizado e o seguinte problema: Dados números naturais $p_1,p_2,\ldots,p_n$ e c, encontrar o maior número b em $\{0,1,\ldots,c\}$ tal que $p(X) = b$ para algum subconjunto X de $\{1,2,\ldots,n\}$.
2. Mostre como um algoritmo para o problema subset-sum generalizado pode ser usado para resolver o nosso problema subset-sum básico.

3. Modifique o algoritmo SUBSET-SUM-REC para resolver o problema subset-sum generalizado.
4. Modifique o algoritmo 1.415^n sugerido acima para resolver o problema subset-sum generalizado.
5. Modifique o algoritmo SUBSET-SUM-PROG-DIN para resolver o problema subset-sum generalizado.
6. Dados números naturais $p_1,\ldots,p_n$ e c, queremos encontrar números $x_1,\ldots,x_n$ em {0,1} que maximizem $x_1p_1+\ldots+x_np_n$ sob a restrição $x_1p_1+\ldots+x_np_n \le c$. Mostre que esse problema é idêntico ao problema subset-sum generalizado. Por que não usar um algoritmo de programação linear para resolver o problema?
7. [REDUÇÃO ENTRE PROBLEMAS] Suponha que você ganhou de presente um software muito rápido para o problema subset-sum. Eu gostaria de usar o software (sem olhar o seu código) para resolver o problema subset-sum generalizado. Como é possível fazer isso? Quantas vezes é preciso "rodar" o software?
8. Seja S um conjunto de números naturais e c um número natural. Queremos encontrar um subconjunto X de S que maximize $\sum X$ sob a restrição $\sum X \le c$. Discuta o seguinte algoritmo para o problema:

```
SUBSUMG (S, c)
  se c ≤ 1
     então ((use força bruta))
     senão  X ← SUBSUMG (S, ⌊c/2⌋)
            Y ← SUBSUMG (S, ⌈c/2⌉)
            devolva X ∪ Y
```

Veja minhas transparências, página 101.

---

http://www.ime.usp.br/~pf/analise_de_algoritmos/
Last modified: Mon Apr 13 07:08:06 BRT 2015

Paulo Feofiloff
Departamento de Ciência da Computação
Instituto de Matemática e Estatística da USP