文章编号: 1001- 2486(2004) 06- 0103- 04

子集和问题的分治求解*

姜新文, 彭立宏
(国防科技大学计算机学院, 湖南 长沙 410073)

摘 要: 介绍了求解子集和问题的一个分治算法。设给定的 n 个正整数为$A(1), A(2), \cdots, A(n-1), A(n)$, 给定的子集和为正整数 M, 算法的时间复杂性为 $O(n^{\log_2(M+1)+1})$, 空间复杂性为 $O(n)$。当 M 较小时, 算法复杂性优于二表算法的复杂性。

关键词: 子集和问题; NP 完全问题; 分治策略; 算法
中图分类号:TP301.6 文献标识码: B

A New Algorithm for Subset Sum Problem with Time Complexity

JIANG Xin-wen, PENG Li-hong
(College of Computer, National Univ. of Defense Technology, Changsha 410073, China)

Abstract: We introduce a new algorithm for subset sum problem. For any given problem with $A(1), A(2), \cdots, A(n-1), A(n)$, and M, the time complexity of the algorithm is $O(n^{\log_2(M+1)+1})$ and the space complexity is $O(n)$, where $A(1), A(2), \cdots, A(n-1), A(n)$, and M are all positive integers. This result is better than that of the two-list algorithm when M is relatively small.

Key words: subset sum problem; NP complete problem; divide and conquer; algorithm

子集和问题描述如下: 给定 n 个正整数$A(i)$ $(1 \leqslant i \leqslant n)$ 以及正整数 M, 问$\{A(1), A(2), \cdots, A(n-1), A(n)\}$中是否存在这样一个子集, 使得子集中各元素之和等于 M。子集和问题属于NP 完全问题[1]。由于解空间由 2^n 个不同的可能解构成, 直接的穷举搜索可能需要遍历问题的解空间, 最坏情况下的时间复杂性为 $O(2^n)$。

Brickell[2] 和 Lagarias、Odlyzko[3] 分别提出了两个算法, 这两个算法仅仅依赖于子集和问题的性质而不是依赖于特定方法的构造, 指出绝大多数低密度的子集和问题可在多项式时间解决。这里, 密度定义为:

$$d = \frac{n}{\log_2 \max\limits_{1 \leqslant i \leqslant n} A(i)}$$

基于上述算法的思想, 人们开展了一系列研究[4-6]。该算法不涉及密度的概念。

Horowitz 和 Sahni 提出了求解子集和问题的二表算法[7], 该算法的时间和空间复杂性分别降至 $O(N \lg N)$ 和 $O(N)$, 其中, $N = 2^{n/2}$。这是至今求解子集和问题的最有效算法[8]。二表算法采用分治策略。文献[8]对二表算法进行研究, 利用最优并行归并算法, 提出了一种基于 CREW-SIMD 共享存储计算模型的并行算法, 其时间复杂性和空间复杂性都是 $O(N)$。这里, N 的取值与前面一样。

将 n 个元素分成两组以后, 二表算法便直接求解这两个规模为 $n/2$ 的子集和问题, 产生所有可能的部分解并搜索问题的解。我们的算法与此类似, 采用分治策略, 提出一种不断分解, 直至规模等于 1 再直接求解的算法。

1 算法的基本思想

设给定的集合 $S = \{A(1), A(2), \cdots, A(n-1), A(n)\}$, 子集和为 M, M 以及$A(1), A(2), \cdots, A(n-$

* 收稿日期: 2004- 03- 05
作者简介: 姜新文(1962—), 男, 教授。

1), $A(n)$ 都是正整数。如果给定的问题有解, M 应该是 $A(1), A(2), \cdots, A(n-1), A(n)$ 中间部分元素的和。

将 $A(1), A(2), \cdots, A(n-1), A(n)$ 分成两个相等的部分 R_1 和 R_2, 问题的解当然也会因此分成两个部分, 一部分在 R_1 中, 一部分在 R_2 中。设问题的解在 R_1 中的部分的和为 M_1, 那么问题的解在 R_2 中的其余部分的和为 $M-M_1$(见图 1)。

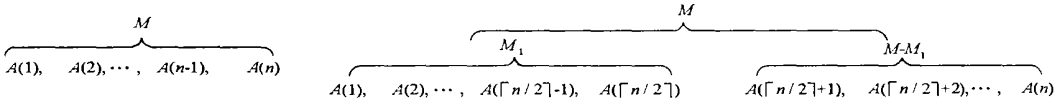
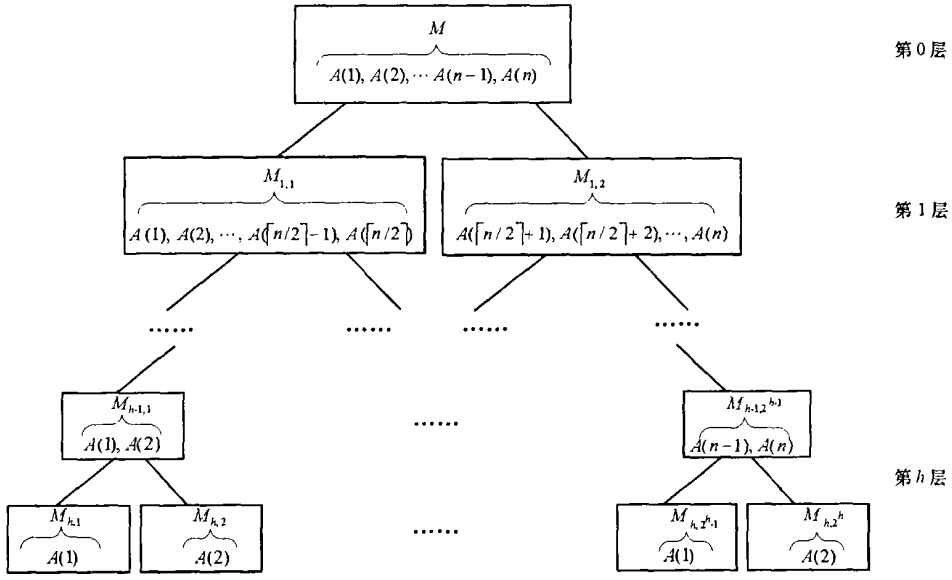


图 1 子集和问题分割示意图

Fig. 1 The dividing strategy for subset sum problem

显然, 在上面的划分中, M_1 可能为 0, $M-M_1$ 也可能为 0, 这两种情况分别对应问题的解全部在 R_2 中间或者全部在 R_1 中的情况。由于事先不知道 M_1 应该取多大的值, 因此这一级划分有 $(M+1)$ 种不同情况, 分别对应 R_1 中部分和等于 0、R_2 中的部分和等于 M, R_1 中部分和等于 1、R_2 中的部分和等于 $M-1$, ……, R_1 中部分和等于 M、R_2 中的部分和等于 0 的情况。

以 M_1 以及 $M-M_1$ 为子集和的规模为 $n/2$ 的子集和问题可以同样划分下去, 直至问题的规模等于 1。此时, 可以直接判定给定的元素对于给定的子集和值是否相等(见图 2)。



图中, $h = \lceil \log_2 n \rceil - 1$。对任意层 k, 有 $\sum_{i=1}^{2^k} M_{k,j} = M$, 其中 $0 < k \leqslant h$。

图 2 子集和问题分解树

Fig. 2 The dividing tree for subset sum problem

2 分治求解算法

现在按照图 3 所示结构, 给出子集和问题的分治求解算法的描述。算法中各参数意义如下: A, M 用来定义问题输入, i、j 用来确定子问题的范围。M_1 是将问题分割成两个部分后, 左半部分的子集和值。l_part 和 r_part 是指明哪个部分有解的标志。算法在问题有解时返回 true, 无解时返回 false。显然, 当左半部分的子集和值为 M_1 时, 右半部分的子集和值为 $M-M_1$。

对于给定的问题, 只需调用 SUBSUM(A, M, 1, n) 即可。

```
Procedure SUBSUM(A, M, i, j)
    begin
      if (j - i= 0) then  //对规模为 1 的子集和问题直接求解//
        if A(i)= M then return true;
        else return false
      else  //对规模大于 1 的问题先分解, 再分别求解 //
        for M1= 0 to M
        begin
          m= (i+ j- 1)/2;
          l_part = SUBSUM(A, M1, i, m);
          r_part = SUBSUM(A, M- M1, m+ 1, j);
          if l_part and r_part then return true;
              // 问题的解分属于 R1, R2 两个部分, 结束调用, 返回上一层//
          if l_part and M1= M then return true;// 问题的解只属于 R1, 返回//
          if r_part and M1= 0 then return true;// 问题的解只属于 R2, 返回 //
          any other case: return false;
        end
      end if;
    end.
```

定理　对任意给定的 n 个正数$A(i)$ $(1\leqslant i\leqslant n)$ 以及正数 M, 算法 SUBSUM 可以在 $O(n^{\log_2(M+1)+1})$ 时间内判定子集和问题有解或者无解。

证明　首先证明算法的正确性。对元素个数 n 进行归纳证明。

$n=1$ 时, 算法直接判断 $A(1)$ 是否等于 M。如果相等返回 true, 否则返回 false。结论显然正确。

假设对于 $n<k$ 结论仍然成立。当 $n=k$ 时, 按照算法, 包含 k 个元素的集合被等分成两个包含元素个数小于k 的集合。如果 $n=k$ 时问题有解, 只有可能出现下面三种情况之一:

(1) $M_1\neq 0$ 且 $M_1\neq M$。此时问题的解分属于 R_1 和 R_2。由于 R_1 和 R_2 的规模都小于 k, 按照归纳假设, 算法能够判定 R_1 中有否子集和等于 M_1 的子集, 同时 R_2 中有否子集和等于 $M-M_1$ 的子集。

(2) $M_1=M$。此时问题的解全部集中在 R_1 中。由于 R_1 的规模小于 k, 按照归纳假设, 算法能够判定 R_1 中有否子集和等于 $M_1=M$ 的子集。

(3) $M_1=0$。此时问题的解全部集中在 R_2 中间。由于 R_2 的规模小于 k, 按照归纳假设, 算法 SUBSUM 能够判定 R_2 中有否子集和等于 $M-M_1=M$ 的子集。

三种情况分别对应算法中的三个 if 语句。这三种情况下算法都能从子问题的解得到原问题的解。

如果 $n=k$ 时问题没有解, 上面三种情况都不成立, 于是原问题无解。

下面分析算法的时间复杂性。以 M 为循环控制变量的循环内部, 要求解两个规模为 $n/2$ 的子问题, 随后要执行一系列判断确定问题有解或是无解。在这个循环内部, 完成这些判断所需要的时间独立于 n 和M。因此, 设算法的时间复杂性为 $T(n)$, $n=2^k$, 则 $T(n)$ 满足如下递归式:

$$T(n)\leqslant\begin{cases}B, & n=1\\ 2(M+1)T(n/2)+2(M+1)B, & n>1\end{cases}$$

其中, B 是一个常数。

令 $L=M+1$, 可以得到:

$$\begin{aligned}T(n)&\leqslant 2LT(n/2)+2LB\\&\leqslant (2L)^2T(n/4)+(2L)^2B+2LB\\&\leqslant\cdots\end{aligned}$$

$$\leqslant (2L)^k T(n/2^k) + (2L)^k B + \cdots + 2LB$$
$$\leqslant (2L) B \frac{2(2L)^k - 1}{2L - 1}$$
$$= (2L - 1 + 1) B \frac{2(2L)^k - 1}{2L - 1}$$
$$= B(2(2L)^k - 1 + \frac{2(2L)^k - 1}{2L - 1}$$
$$\leqslant 4B(2L)^k - 1 < 4B(2L)^k$$
$$= 4Bn^{\log_2(M+1)+1}$$

于是得到 $T(n) = O(n^{\log_2(M+1)+1})$。

算法的空间复杂性主要是数组 A 所占空间, 因此 $S(n) = O(n)$。

3 讨 论

本文提出的算法并没有从根本上解决子集和问题, 但是算法实现了依据 M 对子集和问题进行划分。按照我们的算法, M 越大, 求解越困难。

对于给定的数不是正整数的情形可以转化成我们讨论的情形。例如, 所有的数可以通过增加一个固定的量来消除负数, 等等; 也可以通过修改算法, 以便适应给定的问题中存在负数的情形。例如, M_1 的取值可以从 $\min\{A(1), A(2), \cdots, A(n)\}$ 到 $\max\{A(1), A(2), \cdots, A(n)\}$, 等等。

虽然很难精确分析, 然而一些措施仍然可以用来改进算法。对于给定的 n 个数 $A(1)$, $A(2)$, $\cdots$, $A(n)$ 和 M, 子集和问题的解当中, 最多包含一个大于 $M/2$ 的数, 因此, 如果按照大于 $M/2$ 和小于等于 $M/2$, 将 $A(1)$, $A(2)$, $\cdots$, $A(n)$ 划分成 B_1 和 B_2 两个集合, B_1 最多有一个元素是问题的解。这种发现启示我们, 可能还存在着其它的划分方式也可以求解子集和问题。将这种发现用于我们提出的算法, 可以减少 M_1 作为循环控制变量遍历的数量。当然, 这样做的时候, 将 $A(1)$, $A(2)$, $\cdots$, $A(n)$ 预先排成递减序或者递增序会使得算法处理过程简单而且有效一些。

参 考 文 献:

[1] Garey M R, Johnson D S. Computers and Intractability: A Guide to the Theory of NP-completeness[M]. W. H. Freeman and Company, 1979.

[2] Brickell E F. Solving Low Density Knapsacks Advances in Cryptology[A]. Proceedings of Crypto' 83, Plenum Press, 1984: 25- 37.

[3] Lagarias J C, Odlyzko A M. Solving Low-density Subset Sum Problems[J]. J. Assoc. Comp. Mach., 1985, 32(1): 229- 246.

[4] Coster M J, LaMacchia B A, Odlyzko A M, et al. An Improved Low-density Subset Sum Algorithm[A]. In Advances in Cryptology: EUROCRYPT' 91, 1992: 54- 67.

[5] Coster M J, Joux A, LaMacchia B A, et al. Improved Low-density Subset Sum Algorithms[J]. Comput. Complex. 1992, (2): 111- 128.

[6] Radziszowski S, Kreher D. Solving Subset Sum Problems with the L3 Algorithm[J]. J. Combin, Math. Combin. Comput., 1988, (3): 49- 63.

[7] Schroeppel R, Shamir A. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ Algorithm for Certain NP-complete Problems[J]. SIAM Journal of Computing, 1981, 10(3): 456- 464.

[8] 李庆华, 李肯立, 蒋盛益, 等. 背包问题的最优并行算法[J]. 软件学报, 2003, 14(5): 891- 896.