Improved Dynamic Programming Algorithms for the 0-1 Knapsack Problem

Xiaohua Meng¹,², Yue-an Zhu², Xiaoming Wu³
¹School of Computer Science and Engineering, South China University of Technology, Guangzhou 510632, P. R. China
²School of Information Science and Technology, Jinan University, Guangzhou 510632, P. R. China
³School of Biological Science and Engineering, South China University of Technology, Guangzhou, P. R. China,
xhmeng@163.com

Abstract—Based on the classic dynamic programming solution to solve the 0-1 knapsack problem, we give an improved algorithm called IKP. Further, in order to decrease the space complexity of IKP, we combine divided-and-conquered strategy with IKP to obtain a new algorithm DKP. Our Analysis shows that DKP has a great advantage over IKP in running time and resource cost. Moreover, DKP has a better time complexity than some known algorithms for the 0-1 knapsack problem, and it has high parallel, in which way DKP can relief the tension of memory cost.

Keywords-0-1 knapsack problem; dynamic programming; divided-and-conquer; algorithm complexity

I. INTRODUCTION

The classic 0-1 knapsack problem (KP) is described like this: given a container with capacity c and a set of n items, each kind of item j having profit $p_j$ and weight $w_j$ ($1 \leq j \leq n$), select a subset of the items so that the weight of the subset is no greater than c and their total profit is maximized. By introducing a binary variable $x_j$ for every item $j$ such that $x_j = 1$ if the item is selected otherwise $x_j = 0$, we can formulate mathematically the 0-1-knapsack problem as follows:

$$\begin{align*}
\text{maximize} & \quad z = \sum_{j=1}^{n} p_j x_j \\
\text{subject to} & \quad \sum_{j=1}^{n} w_j x_j \leq c
\end{align*}$$

Here comes the result:

$$\begin{align*}
\text{maximize} & \quad z' = \sum_{i=2}^{n} p_i x_i \\
\text{subject to} & \quad \sum_{j=2}^{n} w_j x_j \leq c - w_1 x_1 \\
& \quad x_i \in \{0,1\}, 2 \leq i \leq n
\end{align*}$$

The above expressions imply $(x_1, x_2, ..., x_n)$ is an optimal solution to the 0-1 KP, and $(x_2, ..., x_n)$ not. Thus, we have

$$\begin{align*}
\sum_{j=2}^{n} x_j p_j < \sum_{i=2}^{n} z_i p_i \\
w_1 x_1 + \sum_{i=2}^{n} w_i z_i \leq c
\end{align*}$$

II. DYNAMIC PROGRAMMING FOR 0-1 KP

A. optimal substructure

0-1 KP has optimal substructures. Supposed $(x_1, x_2, ..., x_n)$ is an optimal solution for the 0-1 KP, and then $(x_2, ..., x_n)$ is an optimal solution for the corresponding sub-problem below:

$$\begin{align*}
\text{maximize} & \quad z' = \sum_{i=2}^{n} p_i x_i \\
\text{subject to} & \quad \sum_{j=2}^{n} w_j x_j \leq c - w_1 x_1 \\
& \quad x_i \in \{0,1\}, 2 \leq i \leq n
\end{align*}$$

Otherwise, we let $(z_2, z_3, ..., z_n)$ be an optimal solution to the sub-problem above, and $(x_2, ..., x_n)$ not. Thus, we have

$$\begin{align*}
\sum_{i=2}^{n} x_i p_i < \sum_{i=2}^{n} z_i p_i \\
w_1 x_1 + \sum_{i=2}^{n} w_i z_i \leq c
\end{align*}$$

Here comes the result:

$$\begin{align*}
p_1 x_1 + \sum_{i=2}^{n} p_i z_i > \sum_{i=1}^{n} p_i x_i \\
w_1 x_1 + \sum_{i=2}^{n} w_i z_i \leq c
\end{align*}$$

The above expressions imply $(x_1, z_2, ..., z_n)$ is an optimal solution to the 0-1 KP, and $(x_1, x_2, ..., x_n)$ not. This leads a contradictory conclusion. So, 0-1 KP has optimal sub-structure property.

B. Recursion function

Let us think the following sub-problem of the 0-1 KP

$$\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{n} p_i x_i
\end{align*}$$
Subject to
\[
\sum_{i=1}^{n} w_i x_i \leq j
\]
(11)
\[x_i \in \{0, 1\}, i \leq t \leq n\]

Let the optimal value of this sub-problem be \(f(i, j)\), i.e., \(f(i, j)\) is the optimal value for the container of capacity \(j\) and items through \(i\) to \(n\). Because of the optimal sub-structure property, we can formulate the recursion function as follows:
\[
f(i, j) = \begin{cases} 
\max \{ f(i+1, j), f(i+1, j-w_i) + p_i \} & j \geq w_i \\
 f(i+1, j) & 0 \leq j < w_i
\end{cases}
\]
(12)

Before using formula (12) to compute, we must initialize the function like this:
\[
f(n, j) = \begin{cases} 
\sum_{i=1}^{n} w_i & j \geq w_n \\
0 & 0 \leq j < w_n
\end{cases}
\]
(13)

C. Time and space complexity analysis

When using the above recursion function to guide programming, it requires the weights of items must be integers. Under this condition, we can know that the time complexity is \(O(nc)\) and the space complexity is \(O(nc)\) as well. By carefully programming we can reduce the space complexity to \(O(c)\). This algorithm is seemingly polynomial, but as a fact it is pseudo-polynomial [4]. When the capacity of container is large, the cost of computing time is high. For example, when \(c > 2^n\), this algorithm has a time complexity of \(\Omega(n^2)\). Martello, Pisinger and Toth [5] give an algorithm, called Combo, which has a good performance. The Combo combines dynamic programming with tight bounds for the 0-1 KP. Although the time complexity of Combo is also \(O(nc)\), on average, it avoids worst cases appearing as a result of tight bounds and most of cases can be solved in an acceptable time. In next sections, we will make some improvements over the classic dynamic programming algorithm for 0-1 KP according to the drawback above, which has no concern with tight bounds.

III. FIRST IMPROVED VERSION

As a matter of fact, we notice that variable \(j\) in recursion function \(f(i, j)\) is a continuous variable, i.e., even if the capacity of container is real number, the recursion function also works. The function \(f(i, j)\) is non-decreasing about variable \(j\), i.e., for a limited \(0 = j_1 < j_2 < \ldots < j_k\), the function fulfill the relationship:
\[
f(i, j_1) < f(i, j_2) < \ldots < f(i, j_k)
\]
and
\[
f(i, x) = \infty \text{ for } x < j_1, f(i, x) = f(i, j_k)(j_k \leq x)\]
and
\[
f(i, x) = f(i, j_1)(j_1 \leq x < j_{i-1})\]

The Characteristics of this type function are jumping points. In general, function \(f(i, j)\) is fixed by all its jumping points, illustrated by Figure 1.

We use ordered set \(G(i) = \{(j, f(i, j)) | 0 \leq t \leq k\}\) denoting \(f(i, j)\) where every element is a 2-tuple \((X, Y)\) with \(X = j, Y = f(i, j)\) and elements are ordered by \(X\).

From the definition of \(G(i)\), we have equation \(G(n+1) = \{(0, 0)\}\). In order to get \(G(i)\) from \(G(i+1)\), we must first compute \(H(i+1)\). We give the definition of \(H(i+1)\) as follows:
\[
H(i+1) = \{(X, Y) | (X - w_i, Y - p_i) \in G(i+1)\}
\]

Now, we can merge ordered 2-tuple set \(G(i+1)\) and \(H(i+1)\) to form \(G(i)\). If \(G(i+1)\) contains two 2-tuple elements \((X_1, Y_1)\) and \((X_2, Y_2)\) such that \(Y_1 \leq Y_2\), and \(X_1 \geq X_2\), then according to non-decreasing feature of \(f(i, j)\), we can delete 2-tuple \((X_1, Y_1)\) and we name this 2-tuple controlled-jumping-point (CJP). Here we give an example to see how this improved method works.

Let \(n=3\), the weight vector of items \(w = (2, 3, 4)\), corresponding price vector of items \(p = (1, 2, 5)\) and the capacity of the container \(c = 6\). According to the above computing way, we have:
\[
\begin{align*}
G(4) &= \{(0, 0)\}; H(4) = \{(2, 1)\}, \\
G(3) &= \{(0, 0), (2, 1)\}; H(3) = \{(3, 2), (5, 3)\}, \\
G(2) &= \{(0, 0), (2, 1), (3, 2), (5, 3)\}; H(2) = \{(4, 5), (6, 6), (7, 7), (9, 8)\}, \\
G(1) &= \{(0, 0), (2, 1), (3, 2), (4, 5), (6, 6)\}.
\end{align*}
\]

Note that we can delete CJP and those 2-tuples whose \(X\) is greater than container capacity. So, the answer is in 2-tuples whose \(X\) are the largest in \(G(1)\). By backtracking, we can know which item was selected. We use pseudo-code to describe our above algorithm.
Algorithm IKP \((p, w, n, c)\)

1. Define the set \(G(i)\) for \(i = 1, 2, \ldots, n\) as follows:
   \[
   G(n + 1) = \{(0, 0)\};
   \]
   \[\text{for } (i = n; i >= 2; i--) \{\]
   \[G(i) = \text{MergeSet}(G(i + 1), H(i + 1));\]
   \[\}
   \]

2. Define the set \(H(i)\) for \(i = 1, 2, \ldots, n\) as follows:
   \[
   H(i + 1) = \{(X, Y) | (X - w_i, Y - p_i) \in G(i + 1)\}
   \]
   and \(X \leq c\).

3. Define the set \(G(i)\) for \(i = 1, 2, \ldots, n\) as follows:
   \[
   G(i) = \text{MergeSet}(G(i + 1), H(i + 1));\]

4. The main computation of IKP is to produce ordered 2-tuple set \(G(i)\) \((1 \leq i \leq n)\). Before going on, we give a definition.

   Definition 1: \((1)\) \((j, k)\)

   \[
   H(i + 1) = G(i + 1) \cup (w_j, p_j) \quad \text{if } \exists (X', Y') \in G(2), \quad \text{search reversely in } G(2), \quad \text{pick up an pair } (X, Y).
   \]

   (2) \((1)\)

   \[H(i) = \{1, 2, 3, 4\}; \quad (w_j, p_j) = (1, 3); \quad H(i) = G(i + 1) \cup (w_j, p_j).\]

   By definition 1, we have

   \[
   H(i) = \{(1, 2) + (1, 3), (3, 1) + (1, 3)\} = \{(2), (4, 7)\}
   \]

   We obtain \(H(i)\) by \(G(i + 1) \cup (w_j, p_j)\). The main computation to produce \(G(i)\). Because of \(H(i + 1) = G(i + 1) \cup (w_j, p_j)\) and \(G(i) = G(i + 1) \cup H(i + 1)\), computing \(H(i + 1)\) needs \(\mathcal{O}(|G(i + 1)|)\) time. Deleting CIP and merging \(G(i + 1)\) and \(H(i + 1)\) also need \(\mathcal{O}(|G(i + 1)|)\) time.

   From the definition of \(G(i)\), \(G(i)\) indicates all possible states of \((x_1, \ldots, x_n)\) by \(2^{n-i} + 1\) times decision. 2-tuple \((X, Y)\) is used to present the state, where \(X\) is weight of combinatorial items and \(Y\) is respective value. In order to get \(G(i-1)\), \(X_{i-1}\)

   can be set to 1 or 0. If \(X_{i-1}\) equals to 0, the state of \(G(i-1)\) is the same as \(G(i)\). If \(X_{i-1}\) equals to 1, the state of \(G(i-1)\) is every state of \(G(i)\) plus \((w_{i-1}, p_{i-1})\), i.e., \(H(i) = G(i) \cup (w_{i-1}, p_{i-1})\). So, by merging \(G(i)\) and \(H(i)\), we can get \(G(i-1)\). Obviously, \(|H(i)| \leq |G(i)|\) and, in worst case no 2-tuple can be deleted. In this case, \(|G(i-1)| \leq 2|G(i)|\).

   So, the time and space complexities of the improved algorithm are both \(\mathcal{O}(2^n)\). When the weights of items are integers, we have inequality \(|G(i)| \leq \min\{c, \sum_{j=1}^{n} w_j\} + 1, (1 \leq i \leq n)\).

   When the profits of items are integers, we have \(|G(i)| \leq \sum_{j=1}^{n} p_j + 1, (1 \leq i \leq n)\). When both of them are integers, both of the time and space complexities are \(\mathcal{O}(\min\{nc, n \sum_{j=1}^{n} p_j, 2^n\})\). In fact, when both profits and weights are integers, we have a stronger bound \(8\), i.e., using \(\min\{c, \sum_{j=1}^{n} w_j\} / \gcd(w_1, w_2, \ldots, w_n, c)\) to replace \(\sum_{j=1}^{n} p_j\). Now, the time and space complexities of IKP are both \(\mathcal{O}(\min\{nc, n \sum_{j=1}^{n} p_j / \gcd(p_1, p_2, \ldots, p_n), 2^n\})\) when both profits and weights of items are integers.

### IV. Second Improved Version

When \(n\) is large enough, IKP algorithm need too much memory and lack of practicality. But in reality, weight \(w\) and price \(p\) are usually integers and \(c\) is much less than \(2^n\), so most problems will be solved in an acceptable time. More over, CJD will be deleted in running time. The method we propound below will enable IKP algorithm to run fast and require less memory. We call this algorithm DKP.

In reference \[6\], the authors adopted divided-and-conquered strategy to reduce memory cost when solving the shortest path problem. Here, we adopt the same strategy, i.e., in DKP, we combine IKP algorithm and divided-and-conquered strategy. Next, first we give the framework of algorithm DKP, and then analyze its performance.
We give an example to see how DKP works.

Example 2: $n = 5, c = 11, w = \{2, 2, 6, 5, 4\}, p = \{6, 3, 5, 4, 6\}$. The algorithm divides the set of items into two disjoint subsets whose have a difference of at most 1 in cardinality.

1. $n_1 = 2, c = 11, w_1 = \{2, 2\}, p_1 = \{6, 3\}$
2. $n_2 = 3, c = 11, w_2 = \{6, 5, 4\}, p_2 = \{5, 4, 6\}$

IKP algorithm runs on two subsets separately. The result is then stored in 2-tuple set $S$ and $T$. So, we get

$S(1) = \{(0, 0), (2, 6), (4, 9)\}$
$T(1) = \{(0, 0), (4, 6), (9, 10)\}$

Finally, by orderly traversing the set $S_1$ and reversely traversing the set $T_1$, we get the solution max $p = 16$. Then, we can backtrack to see which item was selected.

A. time and space complexity analysis for DKP

Based on the analysis of IKP algorithm, we have $|S| = 2^{\lceil n/2 \rceil}$, $|T| = 2^{\lceil n/2 \rceil}$. In worst case, the algorithm did not produce CJD, and equality $|S(1)| = |T(1)| = 2^{n-1}$ hold. So both of the time and space complexities of DKP are $O(\min\{2^{n/2}, nc\})$. When the weights and profits of items are both integers, we have $O(\min\{nc, n \sum_{j=1}^{n} p_j \gcd(p_1, p_2, \ldots, p_n), 2^{n/2}\})$.

V. CONCLUSIONS

Dynamic programming is one powerful tool to solve some kinds of combinatorial problem. Usually, by studying the problem carefully, a good improved method will come up. Divided-and-conquered strategy divides the problem into small sub-problems, solve them separately, and then combine the results to form a solution to the original problem. Divided-and-conquered strategy usually plays an important part in reducing time and space complexities. In this paper, we first recall the dynamic programming algorithm for 0-1 KP, and then according to some drawback of the algorithm, propose an improved algorithm called IKP. Finally as far as memory cost is concerned, we combine divided-and-conquered strategy with IKP algorithm to obtain algorithm DKP, whose space and time complexities are reduced to the square root of ones of IKP algorithm, and this time complexity is better than some known algorithms for 0-1 KP such as Combo [5]. Furthermore, DKP has high parallel, and so it can relief the tension of memory cost. In future work, we can take tighter bounds into account as in reference [5] in our DKP to continue the optimization of 0-1 KP.

ACKNOWLEDGEMENT

this work is motivated by Ellis Horowitz and Sartaj Sahni[7, 8].

REFERENCES