Comparison and Analysis of Algorithms for the 0/1 Knapsack Problem

Comparison and Analysis of Algorithms for the 0/1 Knapsack Problem

Xiaohui Pan[1] **and Tao Zhang**[2,3]

1 Computer Course Teaching Department, Shanghai University of Political Science and Law, Shanghai, China 201701
2 Shanghai Institute for Advanced Communication and Data Science, Shanghai, China 200444
3 Department of Computer Engineering and Science, Shanghai University, Shanghai, China 200444
Email: panxiaohui@shupl.edu.cn; taozhang@shu.edu.cn

Abstract. The 0/1 knapsack problem is a typical problem in the field of operational research and combinatorial optimization, and it belongs to the NP problem. Research on the solutions of the 0/1 knapsack problem algorithm has very important practical value. This paper first described the 0/1 knapsack problem, and then presented the algorithm analysis, design and implementation of the 0/1 knapsack problem using the brute force algorithm, the greedy algorithm, the genetic algorithm and the dynamic programming algorithm, and compared the four algorithms in terms of algorithm complexity and accuracy. On this basis, the future research directions of the 0/1 knapsack problem were analysed, and we proposed to use the rough set theory to solve the 0/1 knapsack problem. This paper can provide insight for solving the 0/1 knapsack problem and applications.

1. Introduction

Using intelligent optimization algorithms to solve the optimization problems is a hot topic and has many practical applications. The optimization problems can be divided into (1) solving function optimization problems where the function value is the smallest value of a function, and (2) optimizing the optimization solution to minimize the value of the objective function in a solution space. In some recent research, PSO-GA algorithm [1], artificial bee colony algorithm [2] and GA-GSA algorithm [3] are use respectively to solve constrained optimization problems. The knapsack problem is a typical combinatorial optimization problem. There are more than 40 years of research history so far. The problem can be described mathematically as follows. There are n items, each with a value v_i and a weight w_i, where v_i and w_i are both integers. The knapsack can hold at most weight W, where W is also an integer. The target for this problem is to determine which items to put into the knapsack such that the total value is maximized [4]. This is called the 0-1 knapsack problem because each item can either be taken or left behind; it cannot be taken a fraction of it or multiple or it. An example is shown in Fig. 1.

The knapsack problem can be widely applied in debris collection, resource allocation, job scheduling, capital budgeting, investment decisions, project selection, cargo packing and other fields. For this problem, its solution methods can be divided into two categories: precision algorithms (such as exhaustive search, dynamic programming method, branch and bound method, etc.) and approximation algorithms (such as greedy method, genetic algorithm, ant algorithm, etc.) [5]. Because the knapsack problem belongs to the NP-C (Non-deterministic Polynomial Completeness) problem [6], its computational complexity is $O(2^n)$. In this paper, the brute force exhaustive search algorithm [7], the greedy algorithm [8] [9], the genetic algorithm [10] [11] and the dynamic programming algorithm [12] [13] are respectively used and to solve the 0-1 knapsack problem. The main contributions of this paper are as follows. First, we introduced the 0/1 knapsack problem and analysed four algorithms and their design. Second, we implemented and compared the four algorithms in terms of algorithm complexity and accuracy. Third, we discussed the future research directions of the 0/1 knapsack problem and proposed to use the rough set theory to solve this problem.

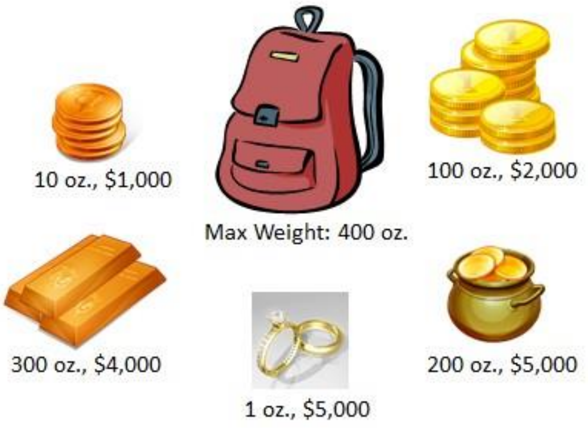

Figure 1. 0-1 knapsack problem.

2. The NP-completeness of the Knapsack Problem

The decision version of the knapsack problem can be formulated mathematically as:

Instance: a finite set of items U, a weight $w(u) \in Z+$ and a value $v(u) \in Z+$ for each $u \in U$, along with a knapsack capacity $W \in Z+$, a desired value $V \in Z+$.

Question: Is there a subset $U' \subseteq U$ such that:

$$\sum_{u \in U'} w(u) \leq W \text{ } and \text{ } \sum_{u \in U'} v(u) \geq V \quad (1)$$

Theorem 1: the knapsack problem is NP-complete [6].

Proof: to show that the knapsack problem is in NP, assume we are given a certificate that consists of a capacity W, a target value V and a set of items $U^{'}$, it is easy to sum up the weight and value of each item in $U^{'}$ (in $O(n)$ time) and verify that their total weight is at most W and their total value is at least V (in $O(1)$ time) in polynomial time.

Next the knapsack problem can be proved as NP-hard by showing that SUBSET-SUM $\leq$ pKNAPSACK. The SUBSET-SUM is a known NP-complete problem, as proved in chapter 34 of [14]. We can show the transformation is indeed a reduction: a set of positive integers A has a subset $A^{'}$ whose sum is t iff a set of items U has a subset $U^{'}$ whose weight is at most W and value is at least V.

Suppose a set of n positive integers $A = a_1, ..., a_n$ has a subset $A^{'}$ whose sum is t. That is, $< A, t > \in$ subset – sum. Define an instance of knapsack problem with a set of n items $U = u_1, ..., u_n$, and $w(u_i) = v(u_i) = a_i$. That is, the weight and the value of item u_i equal positive integer a_i. Finally, set $t = W = V$. By choosing the items that correspond to the subset $A^{'}$ into $U^{'}$, the total weight $\leq W$ and the total value $\geq V$. That is, $< A, t > \in subset - sum \Rightarrow < U, W, V > \in knapsack$.

Conversely, suppose a set of n items U above has a subset $U^{'}$ whose total weight $\leq W$ and the total value $\geq V$. Since $t = W = V$, we can pick out those integers correspond to the weight or value of items ($w(u_i) = v(u_i) = a_i$) in $U^{'}$ to form a subset $A^{'}$, whose sum is exactly t. That is, $< U, W, V > \in knapsack \Rightarrow < A, t > \in subset - sum$. As the reduction can be done in polynomial time, knapsack problem is NP-complete $\diamond$.

3. The Analysis and Implementation of Algorithms

3.1. The Brute Force Algorithm
The 0-1 knapsack problem can be solved using a brute force algorithm [7]:

- Since there are *n* items, there are 2^n possible combinations of items.
- The algorithm just goes through all combinations and finds the one with the maximum value and with total weight less or equal to the capacity *W* of the knapsack.

3.2. The Greedy Algorithm
The greedy algorithm can be used to solve the 0-1 knapsack problem, but it doesn't guarantee an optimal solution [8] [9]. The algorithm works as following:

- Step1: divide each item's value $v(i)$ by its weight $w(i)$ to get its unit value $v'(i)$.
- Step2: Sort the items in decreasing order by their $v'(i)$.
- Step3: put the items with the highest $v'(i)$ into the knapsack (the greedy choice) until the knapsack is full (cannot hold another item).

3.2.1 Why this Algorithm Fails. To see that this greedy algorithm potentially arrives at non-optimal solution for the 0-1 knapsack problem, let's examine an example from [14].

Table 1. Example of a 0-1 knapsack problem.

Property	Item 1	Item 2	Item 3
Value	$60	$100	$120
Weight	10	20	30
Unit Value	6	5	4

As shown in Table 1, there are three items that have unit value of 6, 5, and 4 respectively. The capacity of the knapsack is 50. According to the greedy algorithm, the item1 and item2 will be selected into the knapsack since they have the highest unit value. Since now the knapsack can't hold the next item with the highest unit value (item3), and there are no more items, the algorithm terminates with a total value of $160 in the knapsack. However, as we can see, the optimal solution should be that selects item2 and item3 into the knapsack such that the total value is maximized ($220).

3.3. The Genetic Algorithm
Genetic Algorithm (GA) is a search algorithm used to solve optimization problems in computational mathematics [10] and is a kind of evolutionary algorithm. The evolutionary algorithm was originally developed from phenomena in Darwin's evolutionary biology, such as heredity, mutation, natural selection, and hybridization.

Genetic Algorithms are usually implemented as a kind of computer simulation. For an optimization problem, a certain number of candidate solutions (known as individuals) can be abstracted as chromosomes, allowing the population to evolve to a better solution. Traditionally, a binary representation (that is, a string of zeros and ones) can also be expressed in other ways. Evolution starts with a completely random population of individuals, and then it happens from generation to generation. To evaluate the adaptability of an entire population in each generation, to randomly select multiple individuals (based on their fitness) from the current population and to generate new populations of life through natural selection and mutation, the population is the current population in the next iteration of the algorithm.

The basic genetic algorithms start with the initial population, through natural selection, crossover, and mutation operations to generate new populations, which are constantly updated until optimal solutions are found [11]. For the knapsack problem, the calculation steps are as follows:

1. Coding: Based on the model of the knapsack problem, I have designed a chromosome encoding method for it. Total *n* objects are encoded into a *n*-bit binary string x. $x[i] = 0$ means that object *i* is not placed in the knapsack, and $x[i] = 1$ indicates that the object *i* is placed in the knapsack. For example,

1010 represents a solution, which means that objects 1 and 3 are placed in a knapsack, while the other objects are not put in.

2. Generating an initial population: in this case, the size of the population is taken to *n*, that is, the population A consists of *n* individuals, and A[i] can be generated by a random method.
3. Population fitness calculation: The individual fitness in a population is calculated according to the following formula:

$$\mathrm{T} = \sum w[i]x[i] \tag{2}$$

$$\text{Fitness} = \begin{cases} \sum \mathrm{v[i]x[i]},\ \text{if } \mathrm{T} \leq \mathrm{W} \\ \sum \mathrm{v[i]x[i]} - \alpha * (\mathrm{T} - \mathrm{W}),\ \text{ifT} > \mathrm{W} \end{cases} \tag{3}$$

The equation (3) is the punishment function for fitness, where the parameter $\alpha > 1.0$. In this example, we take $\alpha = 2$.

4. Choice: I use a probability proportional to fitness to determine the number of individuals in the next generation. The process is as follows:
   · Calculate the sum of the fitness of all individuals in the population $\sum f(A[i]), i = 1,2,...n$.
   · Secondly, the relative fitness of each individual is calculated as $p(A[i] = f(A[i]) / \sum f(A[i]), i = 1, 2, ...n$. It is the probability of each individual being passed on to the next generation;
   · The sum of the total probabilistic values is 1 for each probabilistic value;
   · Finally, a random number between 0 and 1 is generated, and the number of times an individual is selected is determined by the number of random numbers that appear in the above probabilistic regions;
5. Intersections: A single-point cross-cutting method is adopted. Groups are randomly matched first. Then the position of the crossing point is set randomly. And then they exchange some of the genes that go between the paired chromosomes.
6. Mutation: I use the fundamental mutation method to perform the mutation operation. Firstly, the position of gene variation of each individual was determined. Then the original genetic value of the mutation point is reversed according to a certain probability.

3.4. The Dynamic Programming Algorithm

The dynamic programming algorithm [12] can be used to solve the 0-1 knapsack problem and guarantee an optimal solution.

First, this problem has optimal substructure. Let *i* be the highest numbered item in an optimal solution S for knapsack capacity W and items 1..*n*, Then $S^{'} = S - \{i\}$ is an optimal solution for the subproblem of capacity $W - w(i)$ and items 1 through i-1. Otherwise, if there were a better solution P to the subproblems, i.e. P has a higher value, we could use P to combine with item i to get a overall better solution than S, contradicting the assumed optimality of the overall solution.

This problem also has overlapped subproblems [13]. All possible combination of items will be tried so there will be many overlapped subproblems. Specifically, every solution that selects item i as the last item into the knapsack would have to consider the combination of items 1 through i-1 on remaining capacity $W - w(i)$.

Let $n = |U|$ to be the total number of items, and define $V[n, W]$ to be the aggregate value of the solution for items 1 to *n*. Then

$$\mathrm{V[n, W]} = \begin{cases} 0, \quad if\ n = 0\ or\ W = 0 \\ V[n-1, w], if\ w(n) > W \\ \max(v[n] + V[n-1, W - w(n)], V[n-1, W],\ if\ n > 0\ and\ W(n) \leq W \end{cases} \tag{4}$$

The second case means that if item *n* does not fit into the knapsack with capacity *W*, then we discard it. The third case means that when item *n* fits into the knapsack with capacity *W*, then we either take it, adding value $v(n)$ to total value and leaving $W - w(n)$ capacity for remaining items, or we do not take it, keeping total value unchanged and leaving W capacity for remaining items. The whole value is defined recursively.

4. Results and Analysis

4.1. Time Complexity Analysis of Four Algorithms

For the brute force algorithm, since there are 2^n possible combinations to be checked, the running time of this algorithm will be $\Theta(2^n)$. Therefore, this algorithm is not applicable when n is large.

In the greedy algorithm, for n items, the first step takes $O(n)$ time. The second step takes $O(nlgn)$ time to sort. Finally, the third step takes $O(n)$ time to pick the items. Overall, the algorithm takes $O(nlgn)$ time [9].

The run time of the genetic algorithm depends on many factors including the number of iterations to converge. Normally, existing research shows that the time complexity of this algorithm is $O(n^2)$. In most of the times, the genetic algorithm can find the optimal solution in polynomial time.

The time consumed by the dynamic programming algorithm can be divided into two parts. First, it takes $\Theta(nW)$ to construct the table for n items and W capacity. Each cell in table takes $\Theta(1)$ time. Secondly, this algorithm takes $\Theta(n)$ time to trace back and find out an optimal solution from table $V[n, W]$. As a whole, the algorithm takes $(\Theta(nW) + \Theta(n))$ time, which is $\Theta(nW)$.

4.2. Experiment Results

We generated the testing data with 50 objects:
Size={ 80, 82, 85, 70, 72, 70, 66, 50, 55, 25, 50, 55, 40, 48, 50, 32, 22, 60, 30, 32, 40, 38, 35, 32, 25, 28, 30, 22, 50, 30, 45, 30, 60, 50, 20, 65, 20, 25, 30, 10, 20, 25, 15, 10, 10, 10, 4, 4, 2, 1}, Value={220, 208, 198, 192, 180, 180, 165, 162, 160, 158, 155, 130, 125, 122, 120, 118,115, 110, 105, 101, 100, 100, 98, 96, 95, 90, 88, 82, 80, 77, 75, 73, 72, 70, 69, 66, 65, 63, 60, 58, 56, 50, 30, 20, 15, 10, 8, 5, 3, 1}, and $W = 1000$.

For comparison, we implemented the brute-force algorithm, greedy algorithm, genetic algorithm and dynamic programming algorithm respectively. The experiment results are as following:

- The Brute Force Algorithm: total value is 3103, total weight is 1000;
- The Greedy Algorithm: total value is 3077, total weight is 999;
- The Genetic Algorithm: total value is 3103, total weight is 1000, maximum 75 iterations to converge;
- The Dynamic Programming Algorithm: total value is 3103, total weight is 1000.

In the experiments, the brute force algorithm, the genetic algorithm and the dynamic programming algorithm all achieve the optimal solution of value 3103 and weight 1000.

However, the greedy algorithm reaches a non-optimal solution $\{item1, item3, item4, item5\}$ whose aggregate weight is 84 and value is 55. Therefore, greedy algorithm cannot ensure an optimal solution for this 0-1 knapsack optimization problem. Please refer to the attached program for the detail of implementation.

5. Future Research Directions

Algorithms for the knapsack problem have evolved from the earliest solution to solve the knapsack problem, to the subsequent approximation algorithm, and to some hybrid algorithms. These methods are highly effective, but there are still many deficiencies. One of the future directions of the knapsack problem is to introduce the knowledge discovery algorithm into the traditional algorithm and use the knowledge discovery algorithm to mine the main information in the knapsack problem. In knowledge discovery algorithms, rough set theory is often used to solve some inaccurate problems, which is suitable for knowledge classification and acquisition. At the same time, knowledge reduction capabilities of rough set theory can be used to simplify knowledge systems, reduce attributes and attribute values and therefore reduce the complexity of the attribute project.

The possibility of using rough set theory to solve the 0/1 knapsack problem exists. The rough set theory can be used to find the main object in the 0/1 knapsack problem, and then it is iteratively calculated using the genetic algorithm. Genetic algorithm will generate a large amount of data when solving the 0-1 knapsack problem. Rough set theory is used to simplify, filter, and analyse the data. This can reduce the complexity and dimensionality of the data. It will be easier to use refined and summarized data. The discovery of important genetic bits in the genetic algorithm can avoid the genetic algorithm falling into the local optimal solution and to some extent improve the search efficiency of genetic algorithms.

6. Conclusion

The knapsack problem occurs in the decision-making process in the real world in a variety of fields such as finding the least wasteful way to cut raw materials. This problem has been proved to be an NP-complete problem which is hard to achieve an optimal solution. This paper introduces this problem, its solving algorithms and the applications. Four algorithms including the brute force algorithm, the greedy algorithm, the genetic algorithm and the dynamic programming algorithm are analysed and implemented to solve this problem. Experiment results show that the genetic algorithm and the dynamic programming algorithm all achieve the optimal solution. However, the genetic algorithm is not stable and cannot guarantee an optimal solution. The brute force algorithm conducts exhaustive search to get an optimal solution, which takes expensive time cost. The dynamic programming algorithm is the best choice both in running time and accuracy of the solution. We also discussed the future directions and proposed to use the rough set theory to solve the 0/1 knapsack problem.

7. Acknowledgements

The authors would like to thank anonymous reviewers for their fruitful feedback and comments that have helped them improve the quality of this work. This research is partially supported by Natural Science Foundation of Shanghai under grant No. 17ZR1409800.

8. References

[1] Harish G 2016 A hybrid PSO – GA algorithm for constrained optimization problems Applied Mathematics and Computation (Atlanta, USA: Elsevier) 274 292 – 305
[2] Harish G 2014 Solving structural Engineering Design Optimization Problems using an artificial bee colony algorithm J.l of Industrial and Management Optimization 10 (3) 777 – 794
[3] Harish G 2014 A hybrid GA-GSA algorithm for optimizing the performance of an industrial system by utilizing uncertain data In Handbook of Research on Artificial Intelligence Techniques and Algorithms ed P Vasant (USA: IGI Global) Chapter 20 625 – 659
[4] Jaszkiewicz A 2002 On the performance of multiple-objective genetic local search on the 0/1 knapsack problem-a comparative experiment IEEE Transactions on Evolutionary Computation 6 402–412
[5] Simoes A, Costa E 2001 An evolutionary approach to the zero/one knapsack problem: Testing ideas from biology Artificial Neural Nets and Genetic Algorithms (Verlag Berlin, Heidelberg: Springer) 236–239
[6] Kellerer H, Pferschy U, Pisinger D 2004 Introduction to NP-Completeness of knapsack problems In Knapsack problems (Verlag Berlin, Heidelberg: Springer) 483–493
[7] Mohammad A, Saleh O, Abdeen R A 2006 Occurrences Algorithm for String Searching Based on Brute-force Algorithm J. of Computer Science 2 82–85
[8] Goyal A, Lu W, Lakshmanan L V 2011 Celf++: optimizing the greedy algorithm for influence maximization in social networks Proc. of the 20th int. conf. companion on World wide web (New York: ACM) pp 47–48
[9] Wang Y, Cong G, Song G, Xie K 2010 Community-based greedy algorithm for mining top-k influential nodes in mobile social networks Proc. of the 16th ACM SIGKDD int. conf. on knowledge discovery and data mining (New York: ACM) pp 1039–1048
[10] Hassan R, Cohanim B, De Weck O, Venter G 2005 a comparison of particle swarm optimization and the genetic algorithm 46th AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conf. p 1897
[11] Ter Braak, C J 2006 A markov chain monte carlo version of the genetic algorithm differential

evolution: easy bayesian computing for real parameter spaces Statistics and Computing 16 239–249
[12] Godfrey G A, Powell W B 2002 An adaptive dynamic programming algorithm for dynamic fleet management, II: multiperiod travel times Transportation Science 36 21–39
[13] Liu D, Wei Q 2014 Policy iteration adaptive dynamic programming algorithm for discrete-time nonlinear systems IEEE Transactions on Neural Networks and Learning Systems 25 621–634
[14] Cormen T H 2009 Introduction to algorithms (third edition) (MIT press)