MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

V.N. Karazin Kharkiv National University

School of Mathematics and Computer Science

Department of Theoretical and Applied Informatics

Master’s Thesis

Solving the Knapsack Problem in sorting algorithms

Author:
Final year Master’s Program student,
group MCS-64
specialty - Computer Sciences and
Information Technologies,
educational program: "Informatics"
Wan Shuangquan
Supervisor: Iryna Zaretska
Reviewer: Kyryl Korobchynskyi
Adviser: Anna Zozulia

Kharkiv, 2024

// Анотація

Метою цієї статті є дослідження та порівняння застосування та ефективності трьох різних алгоритмів - жадібного алгоритму, алгоритму динамічного програмування та методу зворотного ходу - при розв'язанні задачі про рюкзак 0-1. Задача про рюкзак 0-1 є класичною задачею комбінаторної оптимізації, яка полягає у виборі предметів з множини предметів таким чином, щоб максимізувати загальну вартість предметів у рюкзаку, не перевищуючи при цьому його місткість. вибору предметів з набору предметів таким чином, щоб максимізувати загальну вартість предметів у рюкзаку. Задача має широкий спектр застосувань у різних галузях, таких як розподіл ресурсів та завантаження складів.

Жадібні алгоритми добре відомі своєю простотою та низькою часовою складністю, але вони можуть не давати оптимального розв'язку при розв'язанні задачі про рюкзак 0-1, оскільки їх локально оптимальний вибір може призвести до глобально неоптимального розв'язку. Алгоритм динамічного програмування забезпечує точний розв'язок задачі про рюкзак 0-1. Алгоритм відстежує оптимальні розв'язки на різних етапах, будуючи таблицю простору розв'язків. Алгоритм динамічного програмування гарантовано знаходить оптимальний розв'язок. Зворотний хід - це алгоритм, який знаходить

розв'язок задачі методом проб і помилок. Він намагається побудувати розв'язок поетапно і повертається назад, коли виявляється, що поточний шлях навряд чи призведе до оптимального розв'язку.

У цій статті спочатку представлено базову модель та характеристики задачі про рюкзак 0-1, а потім детально описано основні принципи та кроки реалізації жадібного алгоритму, алгоритму динамічного програмування та методу зворотного відстеження. За допомогою теоретичного аналізу та експериментальних порівнянь оцінюється ефективність цих трьох алгоритмів на різних розмірах задач про рюкзаки, включаючи якість часу розв'язання та якість розв'язку. Наведено теоретичні основи та практичні рекомендації для вибору відповідних алгоритмів у реальних задачах.

Ключові слова: задача про 0-1 рюкзак; жадібний алгоритм; динамічне програмування; метод зворотного ходу

ABSTRACT

The purpose of this paper is to explore and compare the application and efficiency of three different algorithms-the greedy algorithm, the dynamic programming algorithm, and the backtracking method-in solving the 0-1 knapsack problem.The 0-1 knapsack problem is a classical combinatorial optimization problem involving the selection of items from a set of items in order to maximize the total value of the items in the knapsack without exceeding the knapsack's capacity, selecting items from a set of items to maximize the total value of the items in the backpack. The problem has a wide range of applications in various fields such as resource allocation and warehouse loading.

Greedy algorithms are well known for their simplicity and low time complexity, but they may not yield the optimal solution in solving the 0-1 knapsack problem because their local optimal choices may lead to a global non-optimal solution. The dynamic programming algorithm provides an exact solution to the 0-1 knapsack problem. The algorithm keeps track of the optimal solutions at different stages by constructing a table of the solution space.The dynamic programming algorithm is guaranteed to find the optimal solution.Backtracking is an algorithm that finds a solution to the problem by trial and error. It tries to construct the solution in stages and backtracks when it is found that the current path is unlikely to produce an optimal solution.

This paper first introduces the basic model and characteristics of the 0-1 knapsack problem, and then elaborates the basic principles and implementation steps of the greedy algorithm, dynamic programming algorithm and backtracking method. Through theoretical analysis and experimental comparison, the performance of these three algorithms on different sizes of knapsack problems, including the quality of solution

time and solution, is evaluated. The theoretical basis and practical guidance are provided for choosing suitable algorithms in practical applications.

Keywords:0-1 knapsack problem; greedy algorithm; dynamic programming; backtracking

Table of Contents

1 Introduction 1

1.1 Background and significance of the study 1

1.1.1 Background of the study 1

1.1.2 Research significance 2

1.2 0-1 Overview of the Backpack Problem 3

2 Algorithm Analysis Based On Greedy Algorithm 6

2.1 Greedy algorithm 6

2.2 Algorithmic principle 7

2.3 Analysis of the 0-1 Backpacking Problem by the Greedy Algorithm 9

2.4 Time complexity calculation 10

2.5 Code Implementation Process 11

2.6 Advantages and disadvantages analysis 12

3 Analysis Of Algorithms Based On Dynamic Programming 16

3.1 Recursion 17

3.1.1 Recursion principle 17

3.1.2 Recursive code implementation 18

3.1.3 Time complexity calculation 19

3.1.4 Advantages and disadvantages analysis 20

3.2 Recursive with memo 23

3.2.1 Recursion principle with memo 23

3.2.2 Time complexity calculation 25

3.2.3 Code implementation 26

3.2.4 Advantages and disadvantages analysis 26

3.3 Dynamic programming 29

3.3.1 Principles of Dynamic Programming 29

3.3.2 0-1 Backpack Analysis Process and Calculation of Time Complexity 30

3.3.3 Calculation of time complexity 32

3.3.4 Code implementation 32

3.3.4 Advantages and disadvantages analysis 33

4 Algorithm Analysis Based On Backtracking Method 37

4.1 Overview of the retrospective approach 37

4.2 Algorithm analysis 38

4.3 Case Study 40

4.3Algorithm design and code implementation 43

4.4 Computation of time complexity 45

4.5 Advantages and disadvantages analysis 46

5 Conclusion 48

References 50

1 Introduction

1.1 Background and significance of the study

1.1.1 Background of the study

In actual production and life, many important issues need to be decided from a large number of options for an optimal solution, under certain existing conditions, through the decision-making program to improve production efficiency and production revenue, such problems can be categorized as optimization problems. With the development of science and technology and the progress of construction and production, the optimization problem is almost all over the people's life and production in all aspects, playing an increasingly important role in various fields have a wide range of applications, optimization problem is gradually becoming one of the important theoretical basis of modern science. The backpack problem is a typical optimization problem with simple structure and easy to extend, which is widely used in freight loading, encrypted communication, investment decision, energy saving and emission reduction, etc. In 1978, the backpack problem was firstly proposed by M. Ralph and H. Martin, based on which the 0-1 backpack problem has been gradually developed, and many solutions to the 0-1 backpack problem have been put forward[1] . In 2001, Ma Liang et al. used ant optimization algorithm to solve the 0-1 backpack problem and

achieved satisfactory results after experimental validation [2].In 2012, R. Mahajan et al. analyzed the methods for solving the 0-1 backpack problem [3].J. Zavala-Diaz et al. proposed a bifurcation delimitation algorithm for solving the 0-1 backpack problem, and the experimental results showed the effectiveness of this solving method [4]. A. Rong proposed a strategy to decompose the discounted 0-1 backpack problem into several easier subproblems, solving the subproblems using a dynamic programming approach, and confirmed the rationality of the idea experimentally [5].

1.1.2 Research significance

The goal of the 0-1 knapsack problem is to select items to maximize the total value without exceeding the maximum weight of the knapsack. Since this problem is NP-hard, finding the exact solution is very time consuming for large scale problems. It becomes especially important to study different solution algorithms. The solution algorithms can be categorized into exact algorithms and heuristic algorithms.

Exact algorithms: such as dynamic programming, branch-and-bound method, etc. These algorithms can guarantee to find the optimal solution, but with the increase of the problem size, the time complexity of the computation will also increase dramatically.

Heuristic algorithms: such as greedy algorithms, the solution found is not necessarily the optimal solution, but can return a feasible solution,

in practical applications can often quickly find a near-optimal solution.

By studying and applying these algorithms, it is possible to find a balance in real problems, i.e., to find a good enough solution in an acceptable time. This is why the study of different solution algorithms is crucial for solving the 0-1 knapsack problem.

1.2 0-1 Overview of the Backpack Problem

As a typical NP-hard problem, the knapsack problem has a wide range of applications in many real-life domains, such as encryption design, freight loading, investment decision, distributed systems, and unmanned aerial vehicle task allocation. Since the backpack problem was proposed in the 1950s, with the progress of science and technology, the expression forms of the backpack problem have been gradually increasing [6]. Several common knapsack problems are: classical 0-1 knapsack problem, multi-objective knapsack problem, multiple knapsack problem, multi-dimensional knapsack problem, quadratic knapsack problem, and so on.

The 0-1 backpack problem consists of the following three main sets of mathematical forms, which represent three sets of objective functions, constraints and variable constraints, the objective function Z is to maximize the total value of all the selected items in the backpack, and the classical 0-1 backpack problem requires that, the total value of the items to be loaded into the backpack is as large as possible. The second

constraint implies that the total weight of all selected items cannot exceed the capacity of the backpack W. This constraint ensures that the backpack is limited in the amount of weight it can carry. The variable x, which can only take the value 0 or 1, is restricted. This is why it is called the "0-1" knapsack problem, because for each item, there are only two choices: to take it or not to take it.

The symbolic meanings and their mathematical expressions are as follows:

Table 1-1 Meaning of symbols and their mathematical expressions

sign	meaning
n	Total number of items
v_i	Value of item i
w_i	Weight of item i
W	Maximum capacity of the backpack
x_i	Selection variable: x_i= 1 means select item i, x_i= 0 means do not select item i.
Z	An objective function that represents the total value of the selected items

$$Maximize \quad Z = \sum_{i=1}^{n} v_i x_i$$

$$Subject\ to \quad \sum_{i=1}^{n} w_i x_i \le W$$

$$x_i \in 0,1 \qquad i = 1,2,\ldots,n$$

The main problem with the 0-1 knapsack problem is the complexity of its solution space. For example, for a given set of items, each item has two choices - to take or not to take - and thus the solution space grows exponentially (2^n possibilities), making the exhaustive method infeasible when dealing with large-scale problems. There is usually a complex nonlinear relationship between the weight and value of the items,

and how to choose an optimal combination (which maximizes the total value) under the capacity constraint is the core challenge of the problem, which requires precise weighing of the advantages and disadvantages of multiple choices.

2 Algorithm Analysis Based On Greedy Algorithm

2.1 Greedy algorithm

A greedy algorithm is an algorithmic strategy that takes the best or optimal choice in the current state at each step of the selection process, thus hoping that the final result is globally optimal. This algorithm is simple, intuitive, easy to implement, and can find optimal solutions in some problems, such as the minimum spanning tree problem in graph theory. However, in practice greedy algorithms do not always result in a globally optimal solution, especially in problems where global information is considered to make the best decision, the solution given by the greedy is not considered as a whole, and the result of the computation is a locally optimal solution, which ultimately results in a solution for the whole, which may be an optimal solution or an approximate solution [7]. Early applications of greedy algorithms can be traced back to the 1950s and 1960s, when computer science and algorithm theory were in their infancy. During this period, greedy algorithms were used to solve some simple optimization problems such as the coin change problem. By the 1970s and 1980s, with the development of computer science, greedy algorithms began to be applied to more complex optimization problems, such as Huffman coding and minimum spanning tree problems [14]. These applications demonstrated the effectiveness of greedy algorithms in

solving specific types of problems. Into the 1990s, with the rise of the Internet and big data, greedy algorithms were widely used in areas such as network routing, load balancing, and data compression [8]. During this period, the research of greedy algorithms began to involve more complex optimization problems, such as the knapsack problem and the traveler's problem.

2.2 Algorithmic principle

The core idea of the greedy method is to take the optimal choice in the current state at each step of the problem solving process in the hope that such locally optimal choices can be accumulated into a globally optimal solution. This method is simple and intuitive, easy to implement, and can give optimal solutions to some specific problems.

In some of the backpack problems, because the items can be split, the algorithmic principle of the greedy method can be described as follows:

1. Value Density Calculation: the value density of each item is calculated, i.e. the ratio of the item's value to its weight. The higher this ratio is, the more value the item contributes per unit weight.

2. Sorting: Sort all items according to their value density from highest to lowest. This step is to ensure that items with high value density are always considered first in the greedy selection process.

3. Selection process: Starting from the sorted list of items, each item

is considered in turn. For each item, determine whether adding it to the backpack will exceed the backpack's capacity limit. If it will not exceed, add it to the backpack and update the remaining capacity of the backpack; if it will exceed, skip the item.

4. Iterate until completion: Repeat the above selection process until all items have been considered or the backpack is full.

5. Output result: finally, the set of items in the backpack is the solution to the greedy algorithm and its total value is the maximum value sought.

The specific algorithm is as follows

```
def fractional_knapsack(values, weights, W):
    unit_values = [(val / wt, val, wt) for val, wt in zip(values, weights)]
    unit_values.sort(reverse=True)
    total_value = 0
    for uv, val, wt in unit_values:
        if W >= wt:
            W -= wt
            total_value += val
        else:
            total_value += uv * W
            break
    return total_value
```

Figure 2-1 The greedy algorithm solves the knapsack code

The key to the greedy algorithm is to make locally optimal choices at each step, i.e., to choose the item with the highest value density in the current situation and that can fit in the backpack, which is reachable to the optimal solution. The advantage of this method is that it is simple and

fast, but it does not guarantee that a globally optimal solution can always be found, because it does not have only related to the current state. So it is important to consider whether the current problem being solved has no backwardness. In the 0-1 knapsack problem, the greedy algorithm usually fails to obtain an optimal solution because it does not take into account all possible combinations of items.

2.3 Analysis of the 0-1 Backpacking Problem by the Greedy Algorithm

The greedy algorithm can get an optimal solution to the backpack problem where items are detachable, but it does not work for 0-1 backpacks, such as the following items: item 1, weight 10, value 24; item 2, weight 4, value 9; item 3, weight 4, value 9; item 4, weight 5, value 10; The volume of the current backpack is 13, the item cannot be split, the greedy algorithm is solved as follows.

1. Calculate the value for money of each item;

2. Sort the items according to their value for money;

3. determine whether the volume of the selected item exceeds the volume of the backpack;

4.Repeat the above steps until the items are finished, or the backpack volume is filled.

Based on the above steps the following table can be obtained:

Table 2-1 Item Information Sheet

Product	Price	Volume	Cost performance
Item 1	24	10	2.40
Item 2	9	4	2.25
Item 3	9	4	2.25
Item 4	10	5	2.00

According to the greedy strategy, based on the calculation of the cost-effectiveness of each item, the final result is that item 1 is selected, the current volume of the backpack is 10, the remaining volume is 3, and the total value is 24; however, the current optimal solution should be 28. In this case the greedy algorithm will not be able to find the optimal solution, and therefore the greedy algorithm tends to find an approximate solution for 0-1 backpacks.

2.4 Time complexity calculation

1. Sorting: First, all items need to be sorted based on the unit weight value of each item. Using an efficient sorting algorithm (e.g., quick sort, subsumption sort, etc.), the time complexity is O(nlogn), where n is the number of items.

2. Iterate through the items: Once the items have been sorted, the next step is to try to put the items into the backpack in the order in which they were sorted. This step involves traversing the sorted list of items

once and checking whether the current item can be put into the backpack. Therefore, the time complexity of this phase is O(n).

2.5 Code Implementation Process

1. Calculate the unit value:

For each item, calculate the ratio of its value to its weight, i.e., the value per unit weight (value/weight).

2.Sorting:

Sort all items from highest to lowest unit value. The sorting is to ensure that we consider the items with the highest unit value first, which is the basis for greedy selection.

3. Initialize variable:

Initialize a variable to store the total_value and total_weight of the current backpack.

These two variables will keep track of the cumulative value and weight of the items we put into the backpack.

4. Select items:

Iterate through the sorted list of items, and for each item, check if adding it to the backpack will exceed the backpack's capacity limit.

If the current item can fit completely in the backpack (i.e., total_weight + weight <= W), then add its value and weight to total_value and total_weight.

5. Handle remaining capacity:

If the current item can't fit completely into the backpack, check if the remaining capacity of the backpack allows partial items to be put in.

If it can, calculate the value of the largest partial item that can be put in, and update total_value.

6. Termination conditions:

The algorithm terminates when all items are traversed or the backpack is full.

7. return result:

Returns total_value as the maximum value that can fit in the backpack.

```
def greedy_knapsack(values, weights, W):
    n = len(values)
    unit_values = [(v / w, w) for v, w in zip(values, weights)]
    unit_values.sort(reverse=True)
    total_value = 0
    total_weight = 0
    for uv in unit_values:
        value, weight = uv
        if total_weight + weight <= W:
            total_value += value
            total_weight += weight
        else:
            remaining_capacity = W - total_weight
            total_value += value * (remaining_capacity / weight)
            break
    return total_value
```

Figure 2-2 The Code implementation procedure

2.6 Advantages and disadvantages analysis

The greedy algorithm for solving the 0-1 knapsack problem has its distinct advantages and disadvantages. These advantages and

disadvantages are analyzed below:

Pros:

1. Simple and easy to implement:

The logic of the greedy algorithm is simple, easy to understand and implement, and does not require complex data structures or complicated Recursive Logic.

2. Time efficiency:

The time complexity of greedy algorithms is usually low because it avoids complex recursion and backtracking and is suitable for large-scale data processing.

3.Space efficiency:

Greedy algorithms do not need to store all possible solutions and therefore have low space complexity.

4.Suitable for approximate solutions:

In some cases, if you need to get an approximate solution quickly instead of the exact optimal solution, greedy algorithm is a good choice.

5.Heuristics:

Greedy algorithms can be used as part of heuristics for solving more complex problems, such as in some metaheuristics.

Disadvantages:

1. optimal solution is not guaranteed:

The biggest disadvantage of the greedy algorithm is that it does not

always yield an optimal solution to the 0-1 knapsack problem. It only considers local optimal choices and ignores the global optimal solution.

2. limited applicability:

The greedy algorithm is only applicable to those problems that have the nature of greedy selection, i.e., problems where the local optimal choice can lead to a globally optimal solution.

3.May miss better solutions:

Since the greedy algorithm makes greedy choices at each step, it may miss better combinations, resulting in a final result that is not optimal.

4.Restrictions on the problem:

The greedy algorithm requires the problem to have some kind of greedy choice property, which limits its application to other types of problems.

5. Lack of flexibility:

The greedy algorithm does not backtrack to consider other possibilities once it has identified a choice, which may lead to missing out on a better solution in some cases.

6.Sensitivity to input data:

The results of a greedy algorithm may be very sensitive to the input data and small changes in the data may lead to significant changes in the results.

The greedy algorithm for solving the 0-1 knapsack problem is suitable for those scenarios where time efficiency and space efficiency are required and approximate solutions are acceptable. However, if the problem requires an exact optimal solution or if the greedy selection nature of the problem is not obvious, then it may be necessary to consider using other algorithms such as dynamic programming or branch-and-bound methods.

3 Analysis Of Algorithms Based On Dynamic Programming

The birth of dynamic programming algorithms can be traced back to the 1950s, by the American mathematician Richard Bellman.Bellman proposed the concept of dynamic programming while studying the optimization problem of multi-stage decision-making processes [9]. In the 1960s, dynamic programming began to be applied in the fields of operations research and control theory. During this period, dynamic programming was mainly used to solve some problems with the nature of overlapping subproblems and optimal substructures, such as the knapsack problem, the longest common subsequence, and so on. In the 1970s, the theory of dynamic programming was gradually improved. Scholars began to study the mathematical properties of dynamic programming in depth, such as optimal substructure, no posteriority, etc[10]. These properties provide important theoretical support for the application of dynamic programming. From the 1980s to the present, dynamic programming has achieved remarkable results in the fields of artificial intelligence and bioinformatics. With the rapid development of computer science, the applications of dynamic programming have gradually expanded [11].

The content of this chapter starts from recursion, then from recursive analysis with memoization, and finally to dynamic programming algorithm solution, to carry out multi-faceted, multi-

dimensional analysis and solution of 0-1 backpacks.

3.1 Recursion

3.1.1 Recursion principle

Recursion is an algorithmic design method that solves problems by calling functions on themselves, and its core idea is to decompose a large problem into smaller, similarly structured subproblems until the subproblems are simple enough to be solved directly. The key elements of recursion include recursive exit and recursive call[21]. The recursive base is the most basic form of the problem, which defines the condition under which the recursion stops, and once that condition is satisfied, the recursive process terminates and returns the result. Recursive calls, on the other hand, progressively transform the current problem into smaller subproblems by downsizing it into smaller subproblems and solving these subproblems through constant function calls [12].

The core idea of the recursive algorithm for solving the 0/1 Backpack Problem is to solve each subproblem recursively by breaking the problem into subproblems. For each item i, there are two choices: leave it out of the backpack and recursively solve for the maximum value of the remaining items, or put it in the backpack and recursively solve for the maximum value of the remaining items, minus the weight of the current item. The recursive state transfer equation is:

knapsack(i,W)=max(knapsack(i−1,W),v[i−1]+knapsack(i−1,W−w[i

−1]))

Suppose the number of items is n, the volume of the backpack is c, and the commodity volumes are all v. Then a recursive tree can be constructed.

The process of constructing a recursive tree is shown below:

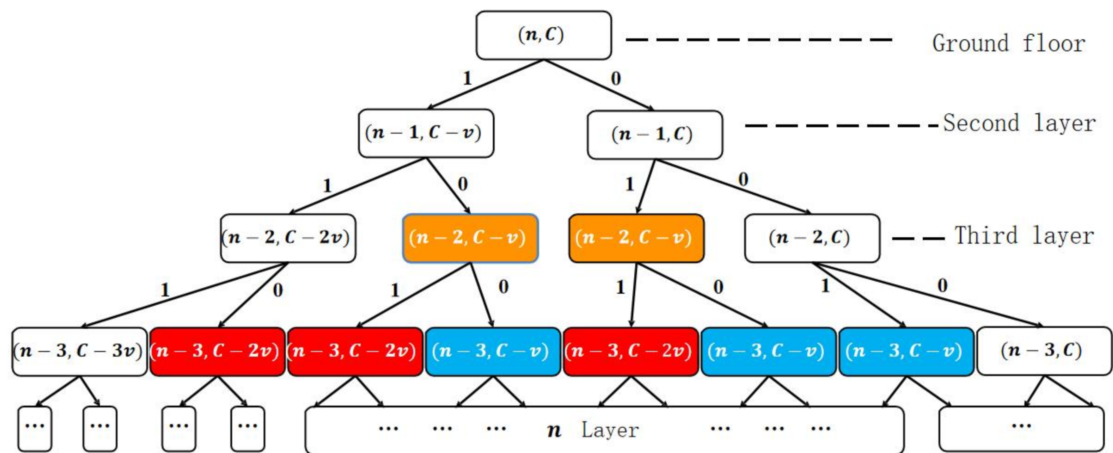

Figure 3-1 Recursive tree decomposition

3.1.2 Recursive code implementation

1. Define a recursive function:

Define a recursive function, e.g. Backtrack(t), where t denotes the index of the item currently under consideration.

2. Recursive termination condition:

When t exceeds the total number of items n, the recursion terminates. At this point, all items have been considered, and you can check whether the current solution is better than the known optimal solution.

3. Check the current solution:

If the total value of the current solution is greater than the known

optimal solution, update the optimal solution.

4. return the optimal solution:

At the end of the recursion.

```
def knapsack(n, W, weights, values):
    if n == 0 or W == 0:
        return 0
    if weights[n-1] > W:
        return knapsack(n-1, W, weights, values)
    else:
        include_item = values[n-1] + knapsack(n-1, W-weights[n-1], weights, values)
        exclude_item = knapsack(n-1, W, weights, values)
        return max(include_item, exclude_item)
```

Figure 3-2 Recursive code implementation

3.1.3 Time complexity calculation

There are many ways to compute the time complexity of recursion, we kind of use the Master Theorem method to solve the time complexity, the Master Theorem (Master Theorem) is a powerful tool for solving the time complexity of recursive algorithms with the nature of partitioning. It is able to derive the time complexity of an algorithm directly from the recursive relations. The Master Theorem provides three basic scenarios for solving this type of recursive relations and derives the time complexity according to the different scenarios. The following is a detailed analysis of the main theorem:

Assume a recursive relationship:

$$T(n) = aT(\frac{n}{b}) + O(n^d)$$

Where $a \geq 1, b > 1, d \geq 0$。

（1）Case 1: if $a > b^d$，Then the time complexity is:

$$T(n) = O(n^{\log_b a})$$

This case indicates that the number of subproblems increases faster than the amount of work involved in decomposing each of them, so the total time complexity of the recursion is determined by the number of decomposed subproblems.

（2）Case 2：if$a = b^d$，Then the time complexity is:

$$T(n) = O(n^d \cdot \log\ n)$$

This case indicates that the number of subproblems and the amount of work per subproblem are balanced between the layers of the recursion, with a depth of recursion, and therefore additional time is needed to process the work at each layer.

（3）Case 3：if $a < b^d$，Then the time complexity is:

$$T(n) = O(n^d)$$

The Main Theorem provides a concise and efficient method for solving recursive time complexity. By analyzing the parameters a, b and d in the recurrence relation equation, the time complexity can be quickly determined. It is particularly applicable to recursive algorithms for partitioning methods and can efficiently derive the time complexity of common recurrence relations.

3.1.4 Advantages and disadvantages analysis

Recursive algorithms, as a powerful tool for algorithm design, have many advantages, but they also have some disadvantages that cannot be

ignored.

Advantages of recursive algorithms:

(1) Concise and easy to understand code: recursion usually expresses complex problems in a very concise way, especially for algorithms with a partition nature (e.g., tree traversal, sorting, depth-first search of graphs, etc.). Recursion makes the structure of the code clearer and the logic easy to understand, especially if the problem itself is recursive in nature.

(2) Natural Expression Partitioning: recursion is well suited for solving partitioning type problems, which are able to decompose a large problem into multiple similar smaller problems. Many classic partitioning algorithms, such as subsumption sort, quick sort, and binary lookup, are realized by recursion.

(3) Easy to solve complex problems: for some structurally complex problems (such as tree traversal, graph traversal, etc.), recursion provides a very intuitive and natural way to solve them. Recursion can directly reflect the hierarchical structure of the problem, avoiding the complexity of manually managing state and iteration variables.

(4) Reduced code redundancy: recursion avoids the use of a large number of loops and iteration variables, reducing code redundancy. For example, recursion can automatically manage the iteration state while traversing the data structure, simplifying the problem solving process.

Disadvantages of recursive algorithms:

(1) Large space consumption: recursion presses the execution state of the current function into the stack every time the function is called, resulting in a large amount of stack space being consumed by too large a recursion depth. Especially in the case of a large recursion depth, it may cause the stack to overflow and not be able to complete the calculation. The space complexity of a recursive algorithm is usually proportional to the recursion depth.

(2) Lower performance (possible duplication of computation): In some cases, recursion can lead to duplication of computation, especially in the absence of optimization. For example, when solving Fibonacci series, recursion will compute the same sub-problem multiple times, leading to a sharp drop in performance. Although such problems can be optimized by dynamic programming, unoptimized recursion is usually less efficient.

(3) Debugging Difficulty: Debugging recursive functions is usually more complicated than iteration, especially when the recursion depth is large, and each level of calls in the stack may contain different states, resulting in a cumbersome debugging process. Errors in recursion may lead to stack overflow or infinite recursion, making debugging more difficult.

(4) May lead to overcalling: recursive calls may be very frequent,

especially without proper termination conditions, and can easily fall into a state of infinite recursion, leading to program crashes. Especially when there is no strict control on the termination conditions of recursion, the efficiency and stability of recursion may be affected.

(5) Efficiency problems: Recursion may lead to efficiency problems in some problems, especially when there is no tail recursion optimization. Some programming languages do not support tail recursion optimization, which means that each recursion takes up a new stack frame, increasing the time and space overhead of the program.

3.2 Recursive with memo

3.2.1 Recursion principle with memo

Recursion with memoization is a method of optimizing recursive algorithms to improve efficiency by caching the results of subproblems that have already been computed and avoiding repeated computations. It consists of two main directions: first, decompose the problem from top to bottom, compute and store the solution of each sub-problem when the recursion is called; then, solve the sub-problems from bottom to top, and use the already computed solutions directly when needed.

1. Decompose from top to bottom: The basic idea of recursive methods is to decompose a large problem into several smaller problems and solve them step by step. Recursive calls usually start at the top and recursively break the problem into smaller subproblems. For each

subproblem, first determine if it has already been computed. If it has already been computed, the cached result is returned directly (to avoid double-counting). Otherwise, compute the value for that subproblem and store the result for subsequent use.

2. Solving subproblems from the bottom up: When the recursive call encounters a subproblem that has not yet been computed, it continues to recursively decompose the problem until it reaches the most basic subproblem, i.e., it reaches the boundary conditions. During the recursive return, the solution to each subproblem is computed and cached. As the recursion moves back level by level, the computed subproblem solutions can be utilized to solve higher level subproblems.

The key step of recursion with memoization is to avoid repeated computation by caching the solution of the subproblems using a data structure; then, during the recursion, it checks whether the current subproblem has already been computed, and if it has been computed, it returns the cached result directly; if it has not been computed, it performs the computation and stores the result in the cache for subsequent use. This method effectively improves the efficiency of the recursive algorithm and avoids repeated calculations.

Process diagram with memoized recursive record of repeating subproblems:

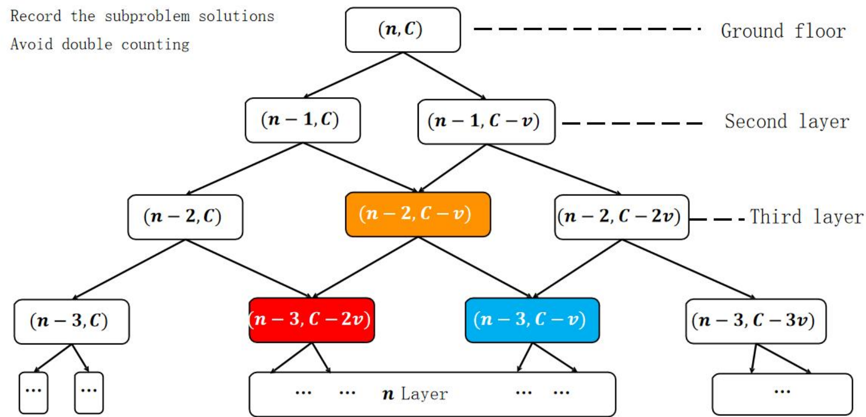

Figure 3-3 Top-to-bottom decomposition with memo recursion

The solution sequence diagram with memoized recursion is shown below:

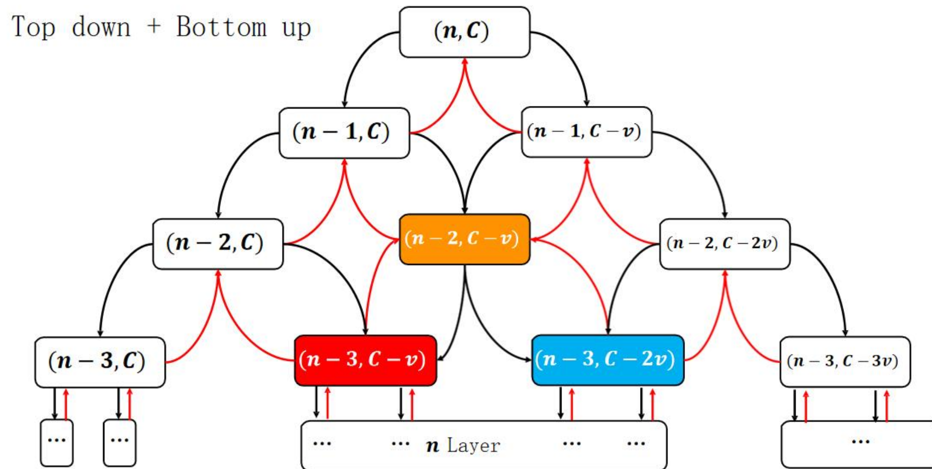

Figure 3-4 Recursive solving diagram with memos

3.2.2 Time complexity calculation

By caching intermediate results, memoized recursion ensures that each subproblem is computed only once. Therefore, the time complexity can be calculated in the following way:

Number of subproblems: assume that the problem size is n and the recursive decomposition produces n subproblems. Each subproblem is

computed only once, resulting in a total number of subproblems of O(n).

Computation cost per subproblem: The operation of computing a subproblem at each recursive call is typically constant time O(1), except for checking the cache. So the computational cost per subproblem is O(1).

The time complexity of recursion with memoization is O(n), where n is the size of the problem and denotes the number of subproblems. In each recursion, the cost of computing each subproblem is constant time, so the total time complexity is O(n).

3.2.3 Code implementation

```
def knapsack_memo(n, W, weights, values):
    memo = [[-1 for _ in range(W + 1)] for _ in range(n + 1)]
    def knapsack_helper(n, W):
        if memo[n][W] != -1:
            return memo[n][W]
        if n == 0 or W == 0:
            result = 0
        else:
            if weights[n - 1] > W:
                result = knapsack_helper(n - 1, W)
            else:
                include_item = values[n - 1] + knapsack_helper(n - 1, W - weights[n - 1])
                exclude_item = knapsack_helper(n - 1, W)
                result = max(include_item, exclude_item)
        memo[n][W] = result
        return result
    return knapsack_helper(n, W)
```

Figure 3-5 Recursive code implementation with memo

3.2.4 Advantages and disadvantages analysis

Recursion with memoization improves efficiency by caching the results of computed subproblems and avoiding repeated computations in recursion. Following are the advantages and disadvantages of recursion with memoization:

Advantages:

(1) Increased efficiency: memoized recursion significantly reduces repetitive computations, especially for recursive problems with overlapping subproblems (e.g., Fibonacci series, knapsack problem, shortest path problem, etc.). By caching the results of each subproblem, each subproblem is computed only once, thus reducing the time complexity from exponential to linear or polynomial level.

(2) Simple to implement: memoized recursion usually only requires the addition of a cache to store intermediate results. It is easier to implement than other optimization methods (e.g., dynamic programming), especially for problems that already have a recursive structure.

(3) Reduced space complexity: For some problems, memoized recursion can effectively reduce space complexity. For example, Fibonacci series recursion can reduce unnecessary space overhead because it caches only the desired results without keeping the entire recursion tree.

(4) Simple and intuitive code: When using memoized recursion, the code usually still maintains the recursive structure, which is more simple and intuitive than the iterative approach of dynamic programming. It is suitable for problems that have a natural recursive structure and do not require complex iteration methods.

Disadvantages:

(1) Space overhead: memoized recursion requires additional storage space to cache the computed results for each subproblem, so the space complexity may be high. For example, for Fibonacci series, the cache stores all the computed values, resulting in a space complexity of O(n). For larger scale problems, the cache may occupy a large amount of memory.

(2) Limited applicability: memoized recursion is best suited for recursive problems with overlapping subproblems. If the problem has no overlapping subproblems (i.e., each subproblem occurs only once), then memoized recursion does not provide significant optimization, and the efficiency may even be reduced due to additional caching operations.

(3) Recursion Depth Limit: There is a limit to the recursion depth (usually 1000 recursions by default). When the recursion depth is too large, it may result in a stack overflow error. In this case, with memo recursion may also face the problem of recursion depth, especially when dealing with very large problems, you need to consider other optimization methods.

(4) Increased complexity: For simple problems, the use of memoized recursion may seem too complex. Compared to direct recursion or iteration, memoized recursion may increase the complexity of the implementation, especially if a cache needs to be managed. If the number of subproblems is very large, or the management of the cache becomes

complex, additional code may be required to manage and clean the cache.

3.3 Dynamic programming

3.3.1 Principles of Dynamic Programming

Dynamic programming is a method of solving complex problems by splitting them into simple sub-problems and solving these sub-problems, which is widely used in the solution of optimization problems. The core idea of dynamic programming is to avoid repeated computation of the same subproblem and to reduce the computational complexity by preserving intermediate results. Dynamic programming is usually solved using a bottom-up solution, which starts from the smallest subproblem and gradually solves larger subproblems by iterating, and finally obtains the solution of the whole problem [13]. Dynamic programming is suitable for problems with the following characteristics:

1. optimal substructure: the optimal solution of the problem can be constructed from the optimal solutions of the subproblems.

2. overlapping subproblems: in the recursive solution of the problem, there are multiple identical subproblems, and traditional recursion causes these subproblems to be computed repeatedly, thus increasing the computational complexity. Dynamic programming avoids such repeated computations by memorization.

3.3.2 0-1 Backpack Analysis Process and Calculation of Time Complexity

In the 0-1 backpack problem, the goal of dynamic programming is to solve the optimal solution step-by-step by populating a two-dimensional table (or array) to eventually obtain the maximum value that the backpack can hold. Suppose there is a backpack with volume c, number of items i, weight of the items w, and corresponding value of the items p. For each item i, there are two choices, to put it in the backpack or not to put it in the backpack. By not putting it into the backpack, the maximum value is the same as the maximum value of the previous i-1 items when the backpack capacity is W , i.e. $dp[i][W] = dp[i-1][W]$; into the backpack, then the remaining capacity of the backpack becomes , and the maximum value is the value of the current item plus the maximum value of the remaining items, i.e. $dp[i][W] = v[i-1] + dp[i-1][W - w[i-1]]$， The premise is that the weight of the current item does not exceed the remaining capacity. The final state transfer equation obtained is:$dp[i][W] = max(dp[i-1][W], v[i-1] + dp[i-1][W - w[i-1]])$

Assuming that the volume of the backpack c,each item corresponds to a value p, the final simplification by constructing a two-dimensional table yields the equation expression as: $P[i, c] = max(P[i-1, c], P[i-1, c - v[i] + p[i])$

Now assume that the volume of the backpack c = 13 and there are

now 5 items, each corresponding to a weight of Vi and a value of Pi, as shown in the following table.

Table 3-1 List of volume and value of items

vi	10	3	4	5	4
pi	24	2	9	10	9

By means of the transfer equation one obtains

$P[i-1, c-v_i]$ and $P[i-1, c]$ in $P\ i, c$ left, so fill the two-dimensional table can get the optimal solution

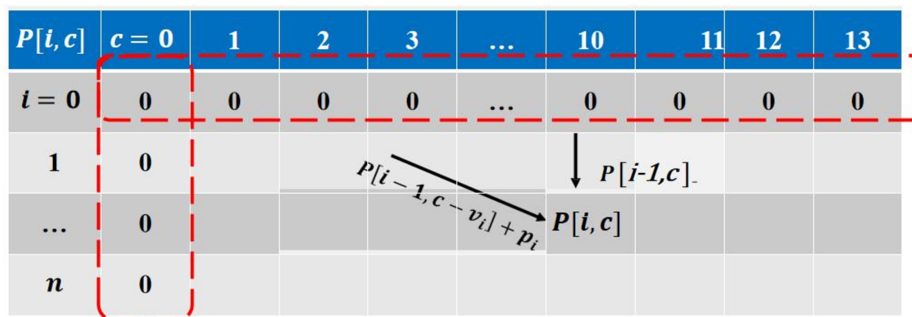

Figure 3-6 Dynamic Planning Two Dimensional Table Diagrams

Based on the above transfer equations for dynamic programming and the related table construction, the following table is finally obtained by continuously filling in the table.

Table 3-2 Dynamic Planning Two Dimensional Table

$P[i, c]$	$c = 0$	1	2	3	4	5	6	7	8	9	10	11	12	13
$i = 0$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	24	24	24	24
2	0	0	0	2	2	2	2	2	2	2	24	24	24	26
3	0	0	0	2	9	9	9	11	11	11	24	24	24	26
4	0	0	0	2	9	10	10	11	12	19	24	24	24	26
5	0	0	0	2	9	10	10	11	18	19	24	24	24	28

3.3.3 Calculation of time complexity

(1) Determine the number of subproblems:

In a knapsack problem, the number of subproblems is usually the product of the number of items and the knapsack capacity , where n is the number of items and W is the knapsack capacity.

(2) Computational time for each subproblem:

In general, the computation of each subproblem of dynamic programming is constant time O(1) or small linear time. In the backpack problem, for each subproblem, the computation is simply comparing two values and hence the time complexity is O(1).

(3) Total time complexity:

The computational time complexity is obtained by multiplying the number of subproblems by the computational time for each subproblem. The final time complexity is:

Total time complexity = number of subproblems × computation time for each subproblem

3.3.4 Code implementation

1. define the state: let dp[i][j] denote the maximum value of the first i items that can fit into a backpack of capacity j.

2. state transfer equation.

3. initialization state: when there are no items (i=0), the value is 0 regardless of the capacity of the backpack, i.e., dp[0][j] = 0. When the

capacity of the backpack is 0 (j=0), the value is 0 regardless of the number of items, i.e., dp[i][0] = 0.

4. Populate the 2D table: Populate the dynamic planning table dp in the order of items (from 1 to n) and in the order of backpack capacity (from 1 to W). For each item i and each capacity j, calculate the value of dp[i][j] according to the state transfer equation.

5. Get the result: the last item of the dynamic planning table dp[n][W] is the maximum value sought, where n is the total number of items and W is the total capacity of the backpack.

```
def knapsack_dp(n, W, weights, values):
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, W + 1):
            if weights[i - 1] > j:
                dp[i][j] = dp[i - 1][j]
            else:
                include_item = values[i - 1] + dp[i - 1][j - weights[i - 1]]
                exclude_item = dp[i - 1][j]
                dp[i][j] = max(include_item, exclude_item)
    return dp[n][W]
```

Figure 3-7 Dynamic programming code implementation

3.3.4 Advantages and disadvantages analysis

Advantages:

(1) Avoid repeated computation: dynamic programming avoids repeated computation of the same subproblem in recursion by storing the solution of the subproblem. This approach drastically reduces the computation time and is especially suitable for problems with overlapping subproblems, such as the knapsack problem, shortest path

problem, Fibonacci series, etc.

(2) Ensure optimal solution: Dynamic programming is suitable for problems with optimal sub-structures, and can construct the optimal solution of the overall problem by solving the sub-problems. Therefore, dynamic programming can guarantee to find the optimal solution, not just an approximate solution or local optimal solution.

(3) Widely used: Dynamic programming is applied to many classical computational problems, such as shortest path, longest common subsequence, backpack problem, sequence alignment, matrix chain multiplication and so on. It can deal with a wide range of problems and is applicable to many combinatorial optimization problems and decision-making problems.

(4) An effective method for solving complex problems: dynamic programming is particularly suitable for solving complex optimization problems, especially when the problem can be decomposed into overlapping subproblems. By decomposing the problem into small, simple subproblems and solving them step by step, dynamic programming is able to construct the final solution over many iterations.

(5) Ability to reduce time complexity: Dynamic programming greatly reduces the time complexity of recursive algorithms from exponential to polynomial. For example, the time complexity of the Fibonacci series is reduced from to O(n) and the knapsack problem is

reduced from to O(nW).

Disadvantages:

(1) Higher space complexity: dynamic programming requires extra space to store intermediate results. For example, the state table of the 0/1 backpack problem needs to store the solutions of the subproblems, where n is the number of items and W is the backpack capacity. This may lead to high space consumption, especially when the number of subproblems is large.

(2) Higher implementation complexity: The implementation of dynamic programming is usually more complex than simple recursive methods. Appropriate state representations, state transfer equations, and structures for storing intermediate results need to be designed. This makes the solution of dynamic programming for some problems more difficult and the code structure may be more complex.

(3) Not Applicable to All Problems: Dynamic programming is applicable to problems with optimal substructures and overlapping subproblems, but not all problems meet these two conditions. If the problem does not have overlapping subproblems, or there is no optimal substructure, then the application of dynamic programming will seem inappropriate, and may not even be advantageous.

(4) Over-optimization Problems: Dynamic programming is used to solve problems by splitting them into smaller sub-problems, but in some

cases the problem itself is simpler, and the complexity of using dynamic programming may exceed the needs of the problem itself. For example, for some small, simple problems, recursive or greedy algorithms may be simpler and more efficient, and using dynamic programming may result in over-optimization, leading to unnecessary time and space overhead.

(5) Difficult to Adapt and Scale: The state transfer equations and state representations of dynamic programming may need to be modified considerably once the size and conditions of the problem change. This makes dynamic programming less flexible than other algorithms in some scenarios.

Dynamic programming is suitable for solving complex optimization problems with optimal substructures and overlapping subproblems. While it offers significant performance gains, its use requires careful trade-offs between problem size and algorithmic complexity due to its higher space requirements and more complex implementation.

4 Algorithm Analysis Based On Backtracking Method

4.1 Overview of the retrospective approach

The Backtracking Method, also known as the Exploration and Backtracking Method or the Trial Method, is a general-purpose algorithm used to find all (or some) solutions to certain computational problems. The Backtracking Method is a powerful general-purpose algorithm for solving a wide variety of constraint satisfaction problems and combinatorial optimization problems. It has gone through several stages of development from origin to generalization and has been widely used in several fields[15] By constructing candidate solutions step by step and checking the constraints, the backtracking method is able to find all satisfying solutions or determine the problem to be solution-free[16]. Although the backtracking method may have a high worst-case time complexity (e.g., exponential time), it is still a viable solution in many practical applications and the time complexity of the algorithm can be reduced by effective pruning operations[17]. The term “backtrack” (“backtrack”) was first coined by the American mathematician D.H. Lehmer in the 1950s.R.J. Walker in his 1960 paper “An enumerative technique for a class of combinatorial problems” gave a fairly general exposition of backtracking, which was the first attempt to comprehensively describe the scope and methodology of backtracking

programming [18].In 1965, S. Golamb and L. Baumert modified backtracking programming and successfully applied it to the chunking solution of a variety of combinatorial problems.In 1976 Stallman, R.M. and Sussman, G.J. proposed the correlation backtracking method[19].In 1993, the dynamic backtracking method was proposed as a way to move back points deeper in the search space, using polynomial space, while still providing useful control information and retaining the integrity of the earlier methods[20].

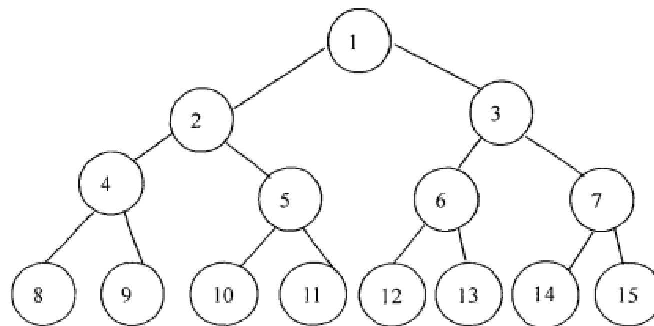

Figure 4-1 Binary Tree Diagram

4.2 Algorithm analysis

The 0-1 knapsack problem is a classical combinatorial optimization problem described as follows: given n items and a knapsack, the weight of item i is Wi, the value of item i is Vi, and the capacity of the knapsack is c. It is asked how the items that are to be loaded into the knapsack should be chosen so that the total value of the items loaded into the knapsack is maximized.

The basic idea of the backtracking method for solving the 0-1

backpack problem is that, starting with the first item, there are two choices for each item - to fit in the backpack or not to fit in the backpack. For each choice, recursively process the remaining items and the remaining backpack capacity until all items have been processed or the backpack capacity is full. During the recursion, the current maximum value is recorded and this maximum is updated on backtracking.

1. Select a number of items to put into the backpack to maximize the value, which can be expressed as follows:

$$\max \sum_{i=1}^{n} v_i \ x_i$$

$$\text{s.t.} \begin{cases} \sum_{i=1} w_i \, x_i \ \le \ W \\ x_i \ \in \{0,1\}, 1 \le i \le n \end{cases}$$

2. Determine the organization of the solution space

The solution space of the problem corresponds to 2n possible solutions and can be represented by a subset tree:

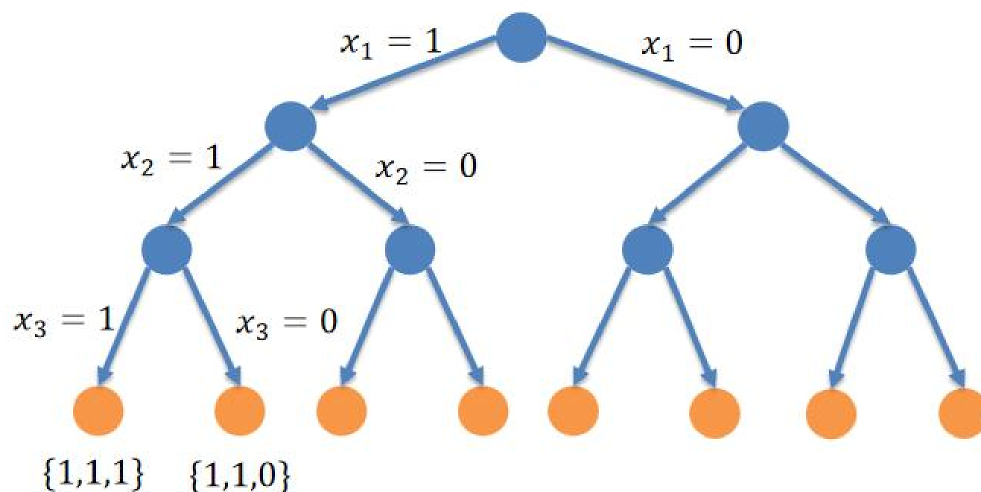

Figure 4-2 Subset tree diagram

4.3 Case Study

Steps to solve the 0-1 backpack problem:

1. define a solution space where each solution is a binary vector of length n indicating whether each item is selected or not (1 means selected, 0 means unselected).

2. traverse the solution space using depth-first search (DFS), and for each solution, compute the total value and total weight.

3. Judge the constraints and use the constraint function when searching in the left branch, called left pruning, and judge the limiting conditions when searching in the right branch, called right pruning.

4. Continue the search until all possible solutions have been tried, i.e. all nodes become dead.

Suppose now there are 4 items with weight (2,5,4,2) and value (6,3,5,4) and the capacity of the backpack is 10.

1. initialization, the total weight of the items sumw = 13, the total value sumv = 18, the initial weight of the items put into the shopping cart cw = 0, the value of the items put into the shopping cart cp = 0, the current optimal value bestp = 0

2. Search the first layer (t=1), expand node 1, first left branch judgment, $cw+w[1]=2<W$, meet the constraints, expand the left branch, so that x[1]=1,cw=2,cp=cp+v[1]=6, to generate the node 2.

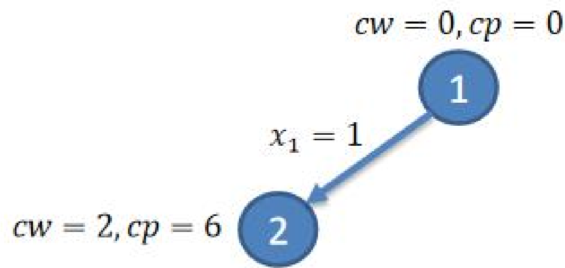

Figure 4-3 Illustration of the backtracking method solution process1

3. Expand node 2 (t=2), first left branch judgment, cw+w[2]=2+5<W, satisfy the constraints, expand the left branch, so that x[2]=1,cw=7,cp=cp+v[2]=9, to generate node 3.

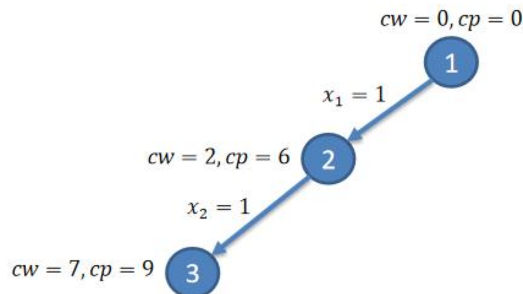

Figure 4-4 Illustration of the backtracking method solution process2

4. Expand node 3 (t = 3), first left branch judgment, cw + w [3] = 7 + 4 > W, does not meet the constraints; turn to judge the right branch, at this time, cp = 9, rp = 4 (residual value), bestp = 0, to meet the cp + rp > bestp, to the right branch to extend the branch, so that x [3] = 0, generating the 4th node.

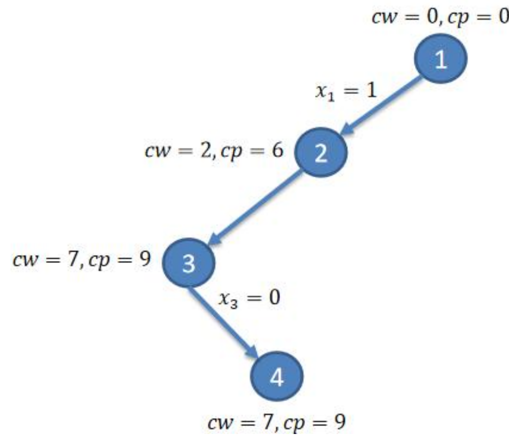

Figure 4-5 Illustration of the backtracking method solution process3

5. Extend node 4, first left branch judgment, cw+w[4]=7+2<W, satisfy the constraints, so branch to the left to extend, generate node 5, node 5 becomes a live node, extend node 5 (t=5),cw=9,cp=9+cp[4]=13, bestp=13, indicating the current optimal solution; and because node 5 has no child nodes (t>n), it becomes a dead node, and a feasible solution bestp=13 is obtained.

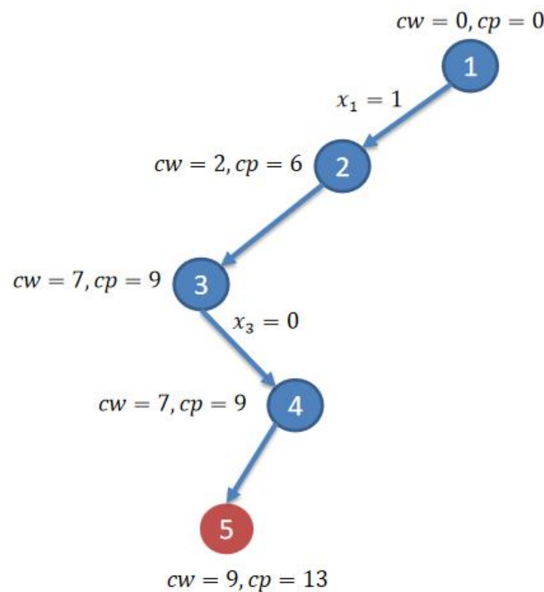

Figure 4-6 Illustration of the backtracking method solution process4

6. backtrack to node 4 (t=4), backtracking, how to add, how to return, cw=cw-w[4]=7,cp=cp-v[4]=9; node 4 right child tree has not yet been generated, examining the limiting conditions, at this time there is no item can be taken, rp=0, cp + rp=9<bestp, cut off, node 4 becomes a dead node, and then backtracked up to node 3 (t = 3), the left and right children of node 3 have been examined, continue back to node 2, the right child of node 2 has not been examined yet, and examine the limit condition. According to this left constraint, the right limit of the way, finally can use backtracking method of obtaining a subset of the tree results of the optimal solution bestp = 15.

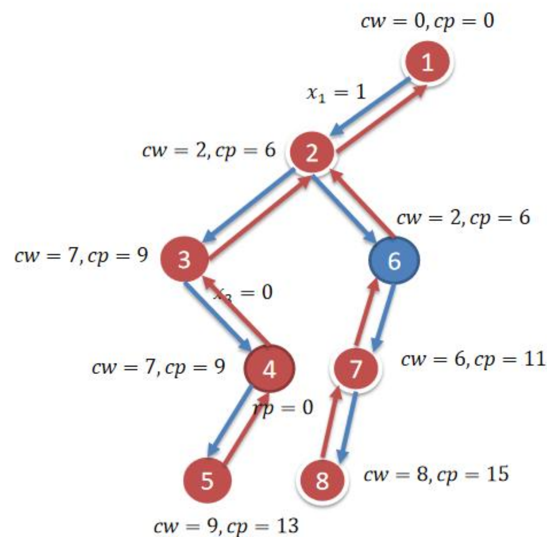

Figure 4-7 Illustration of the backtracking method solution process5

4.3Algorithm design and code implementation

1. Calculating the upper bound

The upper bound is the sum of the value of the loaded items, cp, and the total value of the remaining items, rp. If you are not sure which items

are left to be loaded and assume that they are all loaded, i.e., you calculate the maximum value (the total value of the remaining items), then you get rp, which is the upper bound on the value of the items that can be loaded:

```
def bound(i, cp, v, n):
    rp = 0
    while i <= n:
        rp += v[i]
        i += 1
    return cp + rp
```

Figure 4-8 Computational upper bound

2. Search by constraints and limiting conditions, using the data structures w[], v[], x[], bestp, and the variables; t: the number of layers where the current extension node is located, used to determine whether to reach the leaf, cw: the weight of the current item that has been put into the item, and cp: the value of the current item that has been put into the item.

```
def Backtrack(t):
    if t > n:  # Reach leaf node
        for j in range(1, n+1):
            bestx[j] = x[j]
        bestp = cp
        return
    if cw + w[t] <= W:  # If the constraints are met, extend to the left
        x[t] = 1
        cw += w[t]
        cp += v[t]
        Backtrack(t+1)
        cw -= w[t]
        cp -= v[t]
    if Bound(t+1) > bestp:  # If the limit condition is met, extend to the right
        x[t] = 0
        Backtrack(t+1)
```

Figure 4-9 Code implementation

4.4 Computation of time complexity

1. number of decision tree nodes:

Each item corresponds to one node of the decision tree. For each node (i.e., each item), we have two child nodes: one to indicate that the item is put in, and the other to indicate that the item is not put in. The total number of nodes in the decision tree is 2n (including the root node and all leaf nodes).

2. processing time at each node:

At each node, it is necessary to check whether the currently selected combination of items satisfies the total weight limit of the backpack. This usually involves calculating the total weight of the currently selected items and comparing it with the capacity W of the backpack. The time complexity of this checking process is O(1).

3. total time complexity:

Since the decision tree has 2n nodes and the processing time for each node is O(1), the total time complexity is O(2n*1). After calculation, the total time complexity is O(2n).

The time complexity of the 0-1 backpack problem using the backtracking algorithm is O(2n), where n is the number of items. This complexity is determined by the number of all possible nodes in the decision tree, where each node represents a choice of an item (to put or not to put in the backpack).

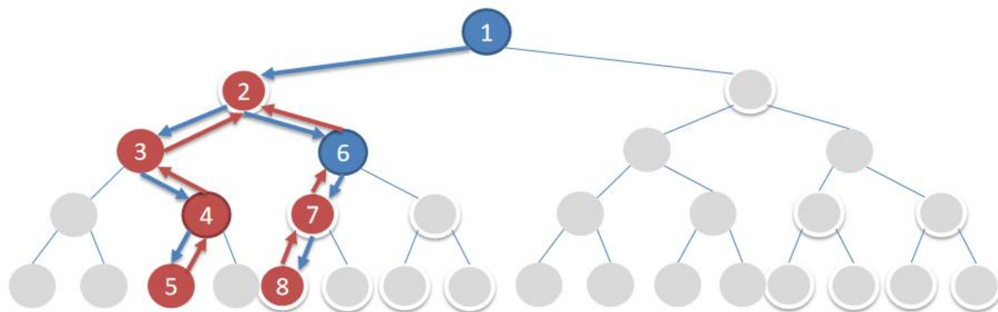

Figure 4-10 Backtracking method solution diagram

4.5 Advantages and disadvantages analysis

Advantages:

1. Systematic: the backtracking method can systematically explore all possible solutions to the problem, ensuring that no solution is missed.

2. Flexibility: the backtracking method can be flexibly adjusted according to the specific needs of the problem, and is suitable for a variety of combinatorial optimization problems.

3. easy to understand: the idea of backtracking method is relatively simple and intuitive, easy to understand and implement.

Disadvantages:

1. High time complexity: for large-scale problems, the time complexity of the backtracking method is too high, which may lead to a long calculation time.

2. High space complexity: in the recursive process, a large number of intermediate states need to be saved, resulting in high space complexity as well.

3. Dependence on specific problems: the efficiency of the backtracking method depends on the specific size and structure of the problem.

5 Conclusion

In solving the 0-1 knapsack problem, the greedy algorithm, backtracking algorithm and branch-and-bound method each show different advantages and limitations. The greedy algorithm, known for its simplicity and speed, fills the backpack by selecting the item with the highest unit value, which does not guarantee to find the global optimal solution, but has obvious time advantages when dealing with large-scale data. However, this algorithm ignores the possibility of a globally optimal solution and only makes locally optimal choices, and thus may be biased in its solution results.

The backtracking algorithm searches for the optimal solution by exploring all possible combinations of items, and can guarantee to find the global optimal solution. This method performs well on small-scale problems, but its time complexity grows exponentially as the problem size increases, leading to a sharp decrease in efficiency. Therefore, the application of backtracking algorithms to large-scale problems is limited.

The branch-and-bound method provides a compromise by constructing a search tree to explore all possible solutions and using bounds to prune and avoid unnecessary computations. This method can significantly reduce the number of solutions to be examined while finding the optimal solution and is suitable for medium-sized problems. The

efficiency of the branch-and-bound method depends on the design of the limit function, which can effectively reduce the search space and improve the efficiency of the algorithm if the limit function is properly designed.

The 0-1 knapsack problem, as a classic problem in algorithm design and analysis, not only has a wide range of practical application background, but also occupies an important position in the field of algorithm research. With the development of technology and in-depth research, the algorithmic solution of the 0-1 knapsack problem is also progressing and evolving, and the future research of the 0-1 knapsack problem will not only be limited to the optimization and improvement of the algorithm, but also involve the universality, adaptability, and cross-disciplinary integration of the algorithm. With the advancement of technology and the development of new theories, we can expect more innovations and breakthroughs in the research of the 0-1 knapsack problem.

References

[1] M. Ralph, H. Martin. Hiding information and signatures in trapdoor knapsacks [J]. IEEETransaction on Information Theory, 1978, 24(5): 525-530.

[2]Ma Liang, Wang Longde. Ant optimization algorithm for the backpack problem [J]. Computer Applications, 2001, 21(8): 4-5.

[3] R. Mahajan, S. Chopra. Analysis of 0-1 knapsack problem using deterministicandprobabilistic techniques[C]. 2012 Second International Conference on. Kyoto, Japan, 2012, pp. 150-155.

[4 J. Zavala-Diaz, J. Ruiz-Vanoye , O. Diaz-Parra et al. A solution to the strongly correlated0-1 knapsack problem by a binary branch and bound algorithm[C]. 2012FifthInternational Joint Conference on Computational Sciences and Optimization(CSO). Washington, USA , 2012, pp. 237-241.

[5] A. Rong, J. Figueira, K. Klamroth. Dynamic programming based algorithms for thediscounted {0-1} knapsack problem [J]. Applied Mathematics and Computation, 2012, 218(12): 6921-6933.

[6] Wei Qian, Ji Bin. A multi-objective discrete combinatorial optimization algorithm incorporating decomposition and adaptive neighborhood[J]. Journal of Frontiers of Computer Science & Technology, 2024, 18(7).

[7] Tropp J A. Greed is good: Algorithmic results for sparse approximation[J]. IEEE Transactions on Information theory, 2004, 50(10):

2231-2242.

[8] Tewari M, Vaisla K S. Optimized hybrid ant colony and greedy algorithm technique based load balancing for energy conservation in WSN[J]. International Journal of Computer Applications, 2014, 104(17).

[9] Bellman R. Dynamic programming: A reluctant theory[C]//New Methods of Thought and Procedure: Contributions to the Symposium on Methodologies. Berlin, Heidelberg: Springer Berlin Heidelberg, 1967: 99-122.

[10] Miller J. Ancestral mounds: vitality and volatility of Native America[M]. U of Nebraska Press, 2015.

[11] Liu D, Xue S, Zhao B, et al. Adaptive dynamic programming for control: A survey and recent advances[J]. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2020, 51(1): 142-160.

[12] Etheve M. Solving repeated optimization problems by Machine Learning[D]. HESAM Université, 2021.

[13] Shi Chengxiang. Application of dynamic planning algorithm in logistics distribution and containerization problems [J]. Logistics Technology,2013,32(13):297-299.

[14] Chen Zhen, Zhong Yiwen, Lin Juan. A hybrid greedy genetic algorithm for solving the 0-1 backpack problem [J]. Computer Applications, 2021,41(01): 87-94.

[15] Ginsberg M L. Dynamic backtracking[J]. Journal of artificial

intelligence research, 1993, 1: 25-46.

[16] De Sanctis A E, Shang F, Uber J G. Real-time identification of possible contamination sources using network backtracking methods[J]. Journal of Water Resources Planning and Management, 2010, 136(4): 444-453.

[17] Van Beek P. Backtracking search algorithms[M]//Foundations of artificial intelligence. Elsevier, 2006, 2: 85-134.

[18] Walker R J. An enumerative technique for a class of combinatorial problems[J]. Combinatorial analysis, 1960: 91-94.

[19] Stallman R M, Sussman G J. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis[J]. Artificial intelligence, 1977, 9(2): 135-196.

[20] Wallace M. Principles and Practice of Constraint Programming-CP 2004[C]//10th International Conference, CP. 2004: 162.

[21] Hamouda S. Enhancing Learning of Recursion[D]. Doctoral dissertation, Virginia polytechnic Institute and State University, 2015.