A Hybrid Dynamic Programming Algorithm for Solving the 0-1 Knapsack Problem

by

Feifan Xing
Bachelor of Electrical Engineering, Xi'an Jiaotong University, 2020

A Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

© Feifan Xing, 2022
University of Victoria

All rights reserved. This report may not be reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

Supervisory Committee

A Hybrid Dynamic Programming Algorithm for Solving the 0-1 Knapsack Problem

by

Feifan Xing
Bachelor of Electrical Engineering, Xi'an Jiaotong University, 2020

Supervisory Committee

Dr. T. Aaron Gulliver, (Department of Electrical and Computer Engineering)
Supervisor

Dr. Riham AlTawy, (Department of Electrical and Computer Engineering)
Departmental Member

Abstract

Supervisory Committee

Dr. T. Aaron Gulliver, (Department of Electrical and Computer Engineering)
Supervisor

Dr. Riham AlTawy, (Department of Electrical and Computer Engineering)
Departmental Member

The 0-1 knapsack problem is a classic non-deterministic polynomial complete problem and has a wide range of applications in many fields. It is a multistage decision problem and can be solved using dynamic programming. First, this report presents the concept, terminology, conditions, and steps of dynamic programming algorithms. Then, two algorithms are introduced to solve the 0-1 knapsack problem, namely Tradition Dynamic Programming (TDP) and Efficient Dynamic Programming (EDP). It is shown that the time complexity and space complexity of TDP are both $O(nW)$, and the time complexity of EDP is $O(nW)$ while the space complexity is only $O(W)$. However, EDP cannot provide any information about item selection, which limits its application. Next, a new hybrid algorithm that combines EDP and the divide-and-conquer method is proposed. This algorithm eliminates the backtracking process used in TDP for item selection but can still determine which items are in the optimal solution. Analysis shows that the time complexity of the hybrid algorithm is $O(nW)$ and the space complexity is $O(W + \log_2 n)$. Results are presented which show that compared with TDP, the new hybrid algorithm has significant advantages in solving large-scale knapsack problems.

Table of Contents

Supervisory Committee ii
Abstract iii
Table of Contents iv
List of Tables v
List of Figures vi
Acknowledgements vii
Chapter 1: Introduction 1
- 1.1 Background and Motivation 1
- 1.2 Related Work 2
- 1.3 Research Goals 3
- 1.4 Report Structure 3

Chapter 2: Dynamic Programming 4
- 2.1 Multistage Decision Problems 4
- 2.2 Dynamic Programming Terminology 5
- 2.3 Dynamic Programming Conditions 7
  - 2.3.1 Optimal Substructure 7
  - 2.3.2 Non-aftereffect 8
  - 2.3.3 Overlapping Subproblems 9
- 2.4 Dynamic Programming Steps 10

Chapter 3: Algorithms for Solving Knapsack Problems 11
- 3.1 Traditional Dynamic Programming 11
  - 3.1.1 Optimal Substructure 11
  - 3.1.2 Model Analysis 11
  - 3.1.3 Complexity Analysis 14
- 3.2 Efficient Dynamic Programming 15
  - 3.2.1 Model Analysis 15
  - 3.2.2 Complexity Analysis 17
- 3.3 Hybrid Dynamic Programming 17
  - 3.3.1 Divide-and-conquer 17
  - 3.3.2 Model Analysis 18
  - 3.3.3 Complexity Analysis 21

Chapter 4: Experiments and Results 22
- 4.1 Time Complexity Experiments 22
- 4.2 Space Complexity Experiments 29

Chapter 5: Conclusion and Future Work 34
- 5.1 Conclusion 34
- 5.2 Future Work 35

Bibliography 36

List of Tables

Table 1: Performance of three algorithms when $n = 1k$ and $W = 100$. 23
Table 2: Performance of three algorithms when $n = 10k$ and $W = 100$. 24
Table 3: Performance of three algorithms when $n = 100k$ and $W = 100$...................... 24
Table 4: Performance of three algorithms when $n = 1k$ and $W = 1k$. 27
Table 5: Performance of three algorithms when $n = 10k$ and $W = 1k$. 27
Table 6: Performance of three algorithms when $n = 100k$ and $W = 1k$........................ 28
Table 7: Space usage of three algorithms. 31

List of Figures

Figure 1: A shortest path problem. 4
Figure 2: A route planning problem.... 7
Figure 3: The 4×4 grid world problem.... 9
Figure 4: The TDP algorithm flowchart. 13
Figure 5: The backtracking flowchart for the TDP algorithm. 14
Figure 6: The EDP algorithm flowchart. 16
Figure 7: The DCDP algorithm flowchart. 20
Figure 8: Performance for uncorrelated problems when $W = 100$.... 25
Figure 9: Performance for weakly correlated problems when $W = 100$. 25
Figure 10: Performance for strongly correlated problems when $W = 100$. 26
Figure 11: Performance for uncorrelated problems when $W = 1k$.... 28
Figure 12: Performance for weakly correlated problems when $W = 1k$. 29
Figure 13: Performance for strongly correlated problems when $W = 1k$. 29
Figure 14: Memory analysis using JProfiler. 30
Figure 15: Space usage of the TDP and EDP algorithms when $n = 10$. 32
Figure 16: Space usage of the TDP and DCDP algorithms when $n = 10$. 32
Figure 17: Space usage of the TDP and EDP algorithms when $n = 100$. 33
Figure 18: Space usage of the TDP and DCDP algorithms when $n = 100$. 33

Acknowledgements

This is the biggest project I have completed in my education career, and each step has been a new challenge for me. In the beginning, I only had a basic idea. Through reading a large number of papers, I have learned about previous achievements and the corresponding problem analysis. The concept in my mind gradually became clearer, and eventually I was able to finish the project.

I would like to thank my parents first. They have fully supported my plan for further education in Canada and given me unlimited support emotionally and financially.

I would like to thank Dr. T. Aaron Gulliver for his generous help when I was in trouble and for his detailed and helpful guidance on my project. I am also thankful to Dr. Xiaodai Dong for her help and concern.

Finally, I would like to thank the University of Victoria for providing me with many useful resources during my Master's program. It's certainly a great place for students to study and conduct research.

Chapter 1: Introduction

1.1 Background and Motivation

The knapsack problem is a typical optimization problem in operations research [1]. The knapsack problem is described as follows. There are i, $i = 1, 2, \ldots, n$, items, each of which will consume a certain amount of resources and also provide a certain amount of benefit. It is required that the sum of the resources consumed by all items loaded into the knapsack does not exceed the capacity of the knapsack.

The knapsack problem can be derived from a series of related optimization problems such as finite knapsack problems (objects can have the same value and weight but a finite number), infinite knapsack problems (the number of objects with the same value and weight can be infinite), and multi-knapsack problems (loading objects into multiple knapsacks with different capacities). Unless otherwise specified, the knapsack problem in this report refers to the 0-1 knapsack problem.

The classic knapsack problem is described as follows. Consider a set of n items and a knapsack with capacity W, where item i has value v_i and weight w_i, $i = 1, 2, \ldots, n$. It is required to select a subset of the items such that their total value is maximized and the sum of their weights is less than or equal to the capacity of the knapsack W. To express the problem clearly, a binary variable x is introduced. If item i is selected, then $x_i = 1$, otherwise $x_i = 0$. Therefore, the mathematical formulation of the problem is to find the n-dimensional vector $[x_1, x_2, \ldots, x_n]$ that maximizes $V = \sum_{i=1}^{n} v_i x_i$ with constraints $\sum_{i=1}^{n} w_i x_i < W$, $x_i \in \{0, 1\}$.

The knapsack problem is a non-deterministic polynomial complete problem and has a wide range of applications in many fields [2]. Many practical problems can be transformed into knapsack problems such as storage space allocation and loan portfolio optimization. All integer linear programming problems can be transformed into 0-1 integer linear programming problems, while 0-1 integer linear programming problems can be transformed into knapsack problems [3]. The knapsack problem is important in applications such as information encryption, project investment, budget control, product purchase, cargo freight, and network security. Therefore, the research and optimization of

methods for solving the knapsack problem have huge significance in both theory and practice.

1.2 Related Work

Many researchers have made progress in finding solutions to the knapsack problem. Dantzig first carried out research in the mid-1950s and used a greedy algorithm to get a solution [4]. He also obtained an upper bound of the optimal solution to the knapsack problem. In 1974, Horowitz and Sahni used the branch-and-bound method to design an efficient algorithm to solve the knapsack problem [5]. They proposed the divisibility of the knapsack problem, indicating a new way to solve the problem. Later, Martello and Toth improved the Dantzig upper bound using integer constraints and the branch-and-bound method [6]. In 1980, Egon and Zemel proposed the idea of Core to solve the knapsack problem, which led to significant progress in the study of this problem [7]. In the late 1990s, Pisinger used balancing methods and the idea of kernel expansion to design a dynamic programming algorithm to solve the knapsack problem [8][9]. In [10], the minimum kernel algorithm based on kernel inflation was given, which is considered a new type of dynamic programming algorithm for solving the knapsack problem.

With the development of artificial intelligence, many evolutionary algorithms for solving knapsack problems have appeared such as the genetic, particle swarm, leapfrog, artificial bee colony, and artificial fish swarm algorithms. However, these algorithms can only provide approximate solutions. Szeto and Zhang introduced a new adaptive genetic algorithm using a mutation matrix and implemented it using the quasi-parallel time-sharing algorithm for solving the knapsack problem [11]. In addition to the traditional leapfrog algorithm to solve the knapsack problem, Gao et al. updated the global optimal solution in each local search iteration for use in subsequent iterations, thereby expanding the solution search space [12]. Cao and Yin proposed a modified artificial bee colony algorithm called the binary artificial bee colony algorithm with differential evolution to solve the knapsack problem [13]. Xue et al. proposed the binary fireworks algorithm to solve the knapsack problem and showed it has excellent search performance [14].

1.3 Research Goals

Although the knapsack problem is a non-deterministic polynomial problem, there are pseudo-polynomial algorithms for solving it. In [15], Pisinger proposed new test instances for the knapsack problem with characteristics that made them more challenging for the dynamic programming and branch-and-bound algorithms. This highlighted the influence of diversity in test instances to draw reliable conclusions about algorithm performance. Knapsack problems can be divided into six categories [15], three of which are relatively common and will be studied in this project. The research goals of this project are as follows.

(1) Summarize the concept, terminology, conditions, and steps of dynamic programming algorithms.

(2) Implement a traditional dynamic programming (TDP) algorithm to solve the knapsack problem and analyze its time and space complexity.

(3) Implement an efficient dynamic programming (EDP) algorithm to solve the knapsack problem. Based on EDP and the divide-and-conquer method, develop a hybrid algorithm to optimize the space complexity to meet the needs of large-scale knapsack problems.

(4) Explore the performance of the TDP, EDP, and hybrid algorithms for the three categories of knapsack problems.

1.4 Report Structure

This report is divided into 5 chapters. The first chapter presents the research background, motivation and research goals, as well as the structure of the report. The formulation of dynamic programming, including concept, terminology, conditions, and design is given in Chapter 2. Chapter 3 presents the TDP, EDP, and hybrid algorithms to solve the knapsack problem, and their time and space complexities are examined. Chapter 4 presents the performance of these three algorithms in solving knapsack problems. Chapter 5 provides a summary of the report and suggestions for future work.

Chapter 2: Dynamic Programming

Dynamic programming, a branch of operations research, is a mathematical method for the optimization of a decision process. In the early 1950s, Bellman and others proposed the principle of optimality to transform a multistage process into a series of stages and use the relationship between stages to solve the problem step by step. This resulted in a new method for solving this kind of optimization problem, namely dynamic programming.

2.1 Multistage Decision Problems

Figure 1 gives an example of the shortest path problem which is a multistage decision problem. The circles in the figure represent cities and the numbers represent the distances between cities. In the figure, the route from A to E can be divided into 4 stages: A to B (whether it is B_1 or B_2), B to C, C to D, and finally D to E. There are multiple routes to choose from at each stage. For example, there are two routes $A \rightarrow B_1$ and $A \rightarrow B_2$ from A to B, and five routes $B_1 \rightarrow C_1$, $B_1 \rightarrow C_2$, $B_1 \rightarrow C_3$, $B_2 \rightarrow C_1$, and $B_2 \rightarrow C_3$ from B to C. The goal is to find a route from A to E that minimizes the total distance, which is an optimization problem with multistage decisions.

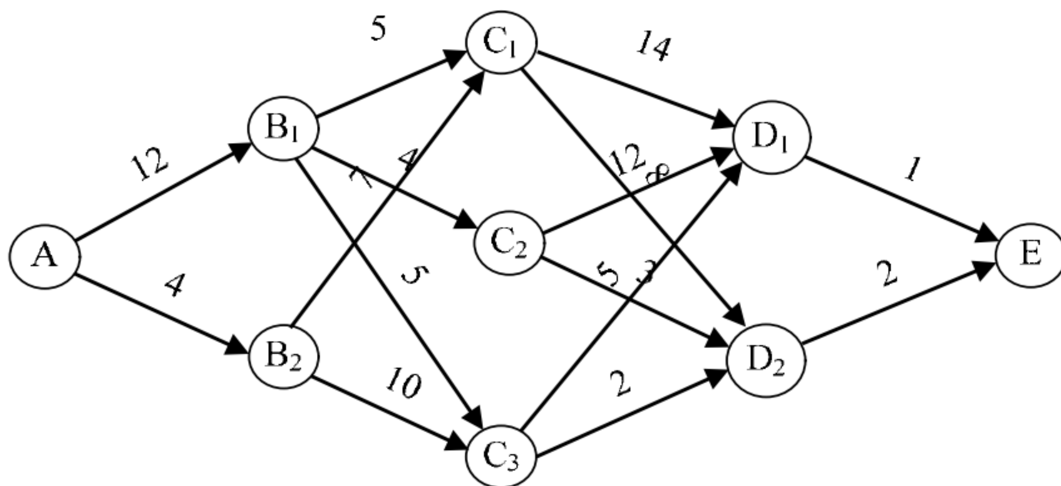

Figure 1: A shortest path problem.

A multistage decision problem is a problem that can be divided into multiple interconnected stages with a specific order. A decision needs to be made at each stage. The decision in the previous stage affects the state of the current stage, and the state of the latter

stage depends on the result in the current stage. The decisions of all stages form a decision sequence. In other words, solving a multistage decision optimization problem requires finding a decision sequence to obtain an optimal solution to the problem.

Both dynamic programming and the divide-and-conquer methods are commonly used to solve multistage decision problems. The key idea is to decompose the problem into several subproblems and solve each subproblem to obtain the solution to the original problem. However, different from the divide-and-conquer method, the subproblems are usually dependent on each other in dynamic programming. If the divide-and-conquer method is used to solve such problems, the number of subproblems may be too large, and some subproblems may be solved many times. If answers to the solved subproblems can be saved, the latter issues can be avoided.

2.2 Dynamic Programming Terminology

The basic terminology of dynamic programming is described below.

Stage: The stage is a property of the problem and is denoted by k, $k = 1, 2, \cdots$. Usually, a multistage decision problem can be divided into several interconnected stages according to the order of processing, so the problem can be solved sequentially in the order in which it is divided [16]. For example, the shortest path problem in Figure 1 consists of four stages: $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$, and $D \rightarrow E$. There are two characteristics of a stage. First, there is a sequence relationship between stages, and second, there is a dependency relationship between stages. These two characteristics indicate that given the solution of the previous stage (sequential relationship), the solution of the current stage can be calculated (dependency relationship). Dependencies between stages can be described by states and state transitions.

State: The state is a property of a stage that describes the situation. Each stage usually contains multiple states. A state variable f_k is used to represent the state in the kth stage, and the value of f_k is a member of the state set F_k, so $f_k \in F_k$. For example, in the shortest path problem in Figure 1, after the first stage, the state can be either B_1 or B_2. Therefore, the state set of the second stage is $F_2 = \{B_1, B_2\}$.

Decision: The choice of the state of the next stage from the state of the current stage is called a decision. In multistage decision optimization problems, the decision values are typically limited to a given range. In the kth stage, the decision when the state is f_k is

denoted by $v_k(f_k)$. Let $V_k(f_k)$ denote the set of allowed decisions when the state is f_k, so $v_k(f_k) \in V_k(f_k)$. For example, in Figure 1, starting from the state B_1 in the second stage, there are three decisions. These three decisions determine the state C_1, C_2 or C_3 in the third stage, so

$$k = 2,\ \ f_2 = B_1,\ \ V_2(f_2) = \{B_1 \to C_1, B_1 \to C_2, B_1 \to C_3\} \tag{2.1}$$

Policy: The policy is obtained by combining the decisions at each stage and is denoted by p. For the n-stage decision problem, the policy starting from the kth stage to the final stage can be expressed as

$$p_k = \{v_k(f_k), v_{k+1}(f_{k+1}), \cdots, v_n(f_n)\} \tag{2.2}$$

In the multistage decision problem, the available policies are typically limited to a given range, and all available policies starting from the kth stage to the final stage form a policy set denoted by P_k. The policy for the optimal solution is called the optimal policy.

State Transition: If f_k is the state in the kth stage and the decision is $v_k(f_k)$, then state f_{k+1} in the $(k+1)$th stage can be determined. Therefore, the state transition is related to both the state and the decision. The state transition is the process of determining state f_{k+1} in the $(k+1)$th stage from state f_k and decision $v_k(f_k)$ in the kth stage of the problem

$$f_{k+1} = T_k\big(f_k, v_k(f_k)\big) \tag{2.3}$$

Indicator Function: There are many different policies for a multistage decision problem, and each policy corresponds to a certain benefit. The indicator function is used to determine the benefit of a policy. Examples of indicator functions are time, space, output, path, and resource consumption. For example, in the shortest path problem, the indicator function gives the length of the route. The indicator function from state f_k in the kth stage to the final stage using policy p_k is $G_k(f_k, p_k)$.

The optimal indicator function is the value of the indicator function when the problem adopts the optimal policy, and the value of the optimal indicator function is the solution to the problem. The optimal indicator function $g_k(f_k)$ is used to indicate the result of the problem which starts from state f_k in the kth stage, and uses the optimal policy p_k^* to reach the final stage

$$g_k(f_k) = G_k(f_k, p_k^*) = \underset{p_k \in P_k}{\text{optimize}}\, G_k(f_k, p_k) \tag{2.4}$$

There are generally two ways to calculate the optimal indicator function. If the initial state of the problem is determined, it can be calculated by moving forward from the initial state to the target state. Conversely, if the target state of the problem is determined, the calculation can be performed by going backward from the target state [17].

2.3 Dynamic Programming Conditions

Not all decision optimization problems can be solved by dynamic programming. Decision optimization problems that can be solved by dynamic programming must satisfy the optimal substructure and non-aftereffect properties [18].

2.3.1 Optimal Substructure

A problem is said to have the optimal substructure property if its optimal solution can be constructed from optimal solutions of its subproblems. The route planning problem shown in Figure 2 can be solved using dynamic programming. Assuming that routes S_1 and S_2 form the shortest route from A to C, according to the optimal substructure property, the optimal solution to the problem is constructed from optimal solutions of its subproblems. Therefore, the shortest route from B to C of the subproblem must be S_2. This conclusion can be proved by contradiction. If there is another route S_2' which is the shortest route from B to C, that is, $S_2 > S_2'$, then $S_1 + S_2 > S_1 + S_2'$. This contradicts the statement that S_1 and S_2 form the shortest route from A to C, so S_2 must be the shortest route from B to C.

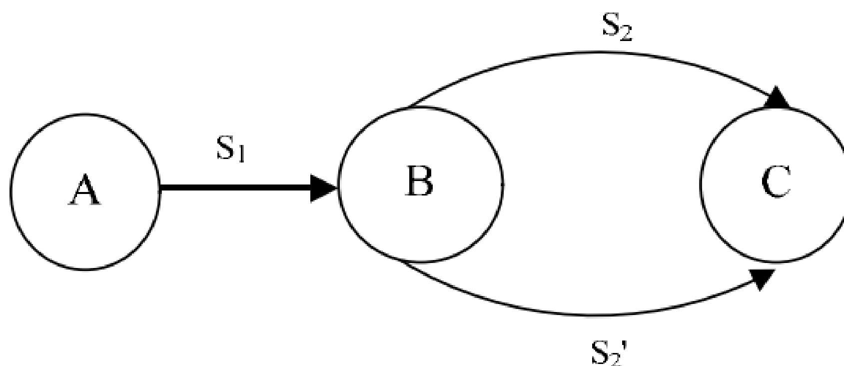

Figure 2: A route planning problem.

The optimal substructure property is the basis of dynamic programming algorithms. Since the optimal solution to the problem is composed of the optimal solutions to the

subproblems, it must be ensured that the subproblems used to construct the optimal solution are solved during the programming process.

2.3.2 Non-aftereffect

When a multistage decision problem is divided into stages, the states of previous stages cannot affect the decision in the current stage. The future state is only determined by the state and decision of the current stage, not the states of previous stages, which is called non-aftereffect.

The grid world problem shown in Figure 3 consists of a 4×4 grid. A player must go from the entrance in the upper left corner to the exit in the lower right corner. Assuming the rule is that they can only go down or right to an adjacent grid each step, then there are two options for moving from the entrance to the black grid $(2, 2)$, indicated by the blue and yellow arrows. Each grid can be considered as a state. After two stages the player reaches the black grid. Then when making decisions at this stage, how this state was reached does not need to be considered, which is an example of non-aftereffect.

If the rule changes so that a player can move up, down, left, or right to an adjacent grid, but cannot move to repeated grids, the situation will be different. When they reach the black grid and consider the next decision, previous decisions need to be considered, that is how the black grid was reached, because the player cannot move to a repeated grid. Thus, previous states affect the decision in the current state, which is an example of aftereffect.

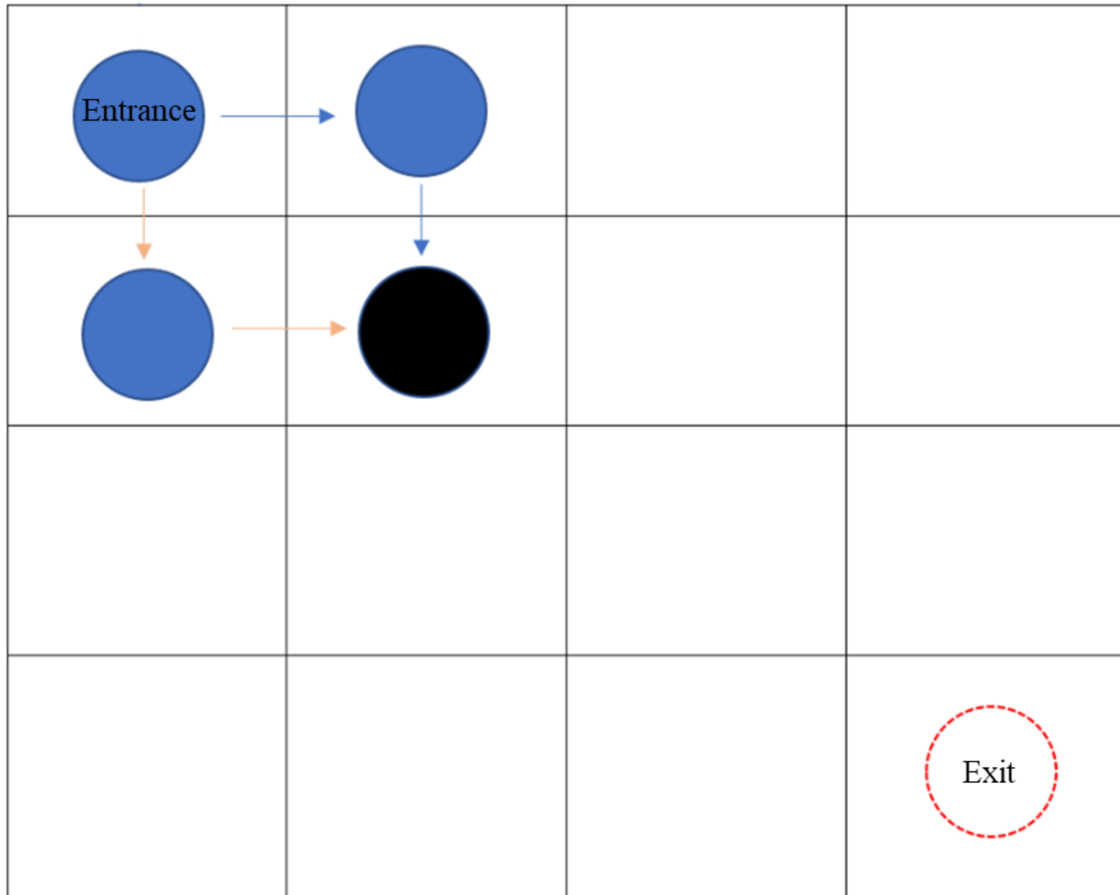

Figure 3: The 4 × 4 grid world problem.

A key condition for dynamic programming algorithms is that the states of the problem satisfy the non-aftereffect property. To determine this, an effective method is to use the states of each stage as vertexes, and the transition relationship between the stages as directed edges. Then construct a graph model and determine whether the graph can be topologically sorted. If topological sorting is impossible, it means there is a loop in the model, and the states of the problem have aftereffects, so the problem cannot be solved using dynamic programming.

2.3.3 Overlapping Subproblems

When an algorithm calculates the same subproblem many times, the subproblems are called overlapping. Dynamic programming solves this issue by exploiting the overlapping nature of the subproblems by computing each first-encountered subproblem and caching the solutions. Therefore, if a subproblem is encountered again, the result can be obtained

directly from the cache [18]. Thus, dynamic programming avoids repeated calculations of subproblems. The divide-and-conquer method does not cache the solutions to the subproblems and so must calculate them every time. Therefore, the subproblems should not overlap when using divide-and-conquer. The essence of dynamic programming is the exchange of space for time. The overlapping nature of subproblems is not a necessary condition for applying dynamic programming algorithms, but the time efficiency of dynamic programming algorithms depends on the degree of overlap of the subproblems.

2.4 Dynamic Programming Steps

The general steps of dynamic programming algorithms are as follows.

(1) Define the subproblems. The problem is divided into several subproblems according to its characteristics. There are sequential relationships among the subproblems, and the current subproblem can be solved after the solutions to the previous subproblems have been obtained.

(2) Select the states. The states represent the initial condition of the stages of a problem and must satisfy the non-aftereffect property.

(3) Determine the state transition equation. A state transition equation is the process of determining the state of the next stage from the state and decision of the current stage.

(4) Find the boundary conditions. Determine the initial and end conditions of the iterations of the state transition equation.

Note that there is no unified standard model for dynamic programming algorithms as different problems can have different models.

Chapter 3: Algorithms for Solving Knapsack Problems

3.1 Traditional Dynamic Programming

3.1.1 Optimal Substructure

The classic knapsack problem is described as follows. Consider a set of n items and a knapsack with capacity W where item i has value v_i and weight w_i, $i = 1, 2, \dots, n$. It is required to select a subset of the items such that their total value is maximized and the sum of their weights is less than or equal to the capacity of the knapsack W. The problem is to find the n-dimensional vector $[x_1, x_2, \dots, x_n]$ that maximizes $\sum_{i=1}^{n} v_i x_i$ with constraints $\sum_{i=1}^{n} w_i x_i < W$, $x_i \in \{0, 1\}$.

A characteristic of the knapsack problem is that there is only one item of each kind, and there are only two decisions for each item [19]. The knapsack problem is a multistage decision optimization problem. It satisfies the optimal substructure property, so if $[x_1, x_2, \dots, x_n]$ is the optimal solution to the knapsack problem [8]

$$\text{maximize } V = \sum_{i=1}^{n} v_i x_i \quad \left(\sum_{i=1}^{n} w_i x_i < W, x_i \in \{0, 1\} \right) \tag{3.1}$$

then $[x_1, x_2, \dots, x_{n-1}]$ is the optimal solution to the subproblem

$$\text{maximize } V = \sum_{i=1}^{n-1} v_i x_i \quad \left(\sum_{i=1}^{n-1} w_i x_i < W - w_n x_n, x_i \in \{0, 1\} \right) \tag{3.2}$$

3.1.2 Model Analysis

The model is established as follows. Subproblem $\langle i, j \rangle$ means that items from the first to the ith can be selected based on the capacity of the knapsack j. $f(i, j)$ is the maximum value from the subproblem and it represents the state. When the current state is determined, the decision is no longer affected by previous states and decisions, so the knapsack problem satisfies the non-aftereffect property. The state transition equation is [20]

$$f(i, j) = \begin{cases} \max(f(i-1, j), f(i-1, j-w_i) + v_i), & j \geq w_i \\ f(i-1, j), & 0 \leq j < w_i \end{cases} \tag{3.4}$$

and the initial condition is

$$f(0, j) = 0, \qquad j \geq 0 \tag{3.5}$$

A two-dimensional array is created to store the values of $f(i,j)$. The horizontal axis i ranges from 0 to the total number of items n, and the vertical axis j ranges from 0 to the capacity of the knapsack W. When the number of items is 0, the knapsack does not hold any items, so $f(0,j) = 0$. Therefore, the elements in the first row of the array are all 0.

The number of items that can be selected increases to n and the capacity of the knapsack increases to W. The weight of a new item and the capacity of the knapsack need to be compared each time. If new item i cannot be put into the knapsack because the weight of it is larger than the current capacity of the knapsack, then the value of the optimal indicator function is the same as when there is no new item, that is $f(i,j) = f(i-1,j)$. If the new item can be put into the knapsack, it must be determined whether the item should be put in. When the ith item is not put in, the value of the indicator function is the same as $f(i-1,j)$. When the item is put in, the value of the indicator function is equal to the value of the new item v_i plus the value of the optimal indicator function $f(i-1, j-w_i)$ for the subproblem that items from the first to the $(i-1)$th can be selected and the capacity of the knapsack is $j - w_i$. The results of these two choices need to be compared to obtain the optimal one. At the end of the iterations, the value of the optimal indicator function $f(n, W)$ for the original knapsack problem is obtained. The TDP algorithm flowchart is shown in Figure 4.

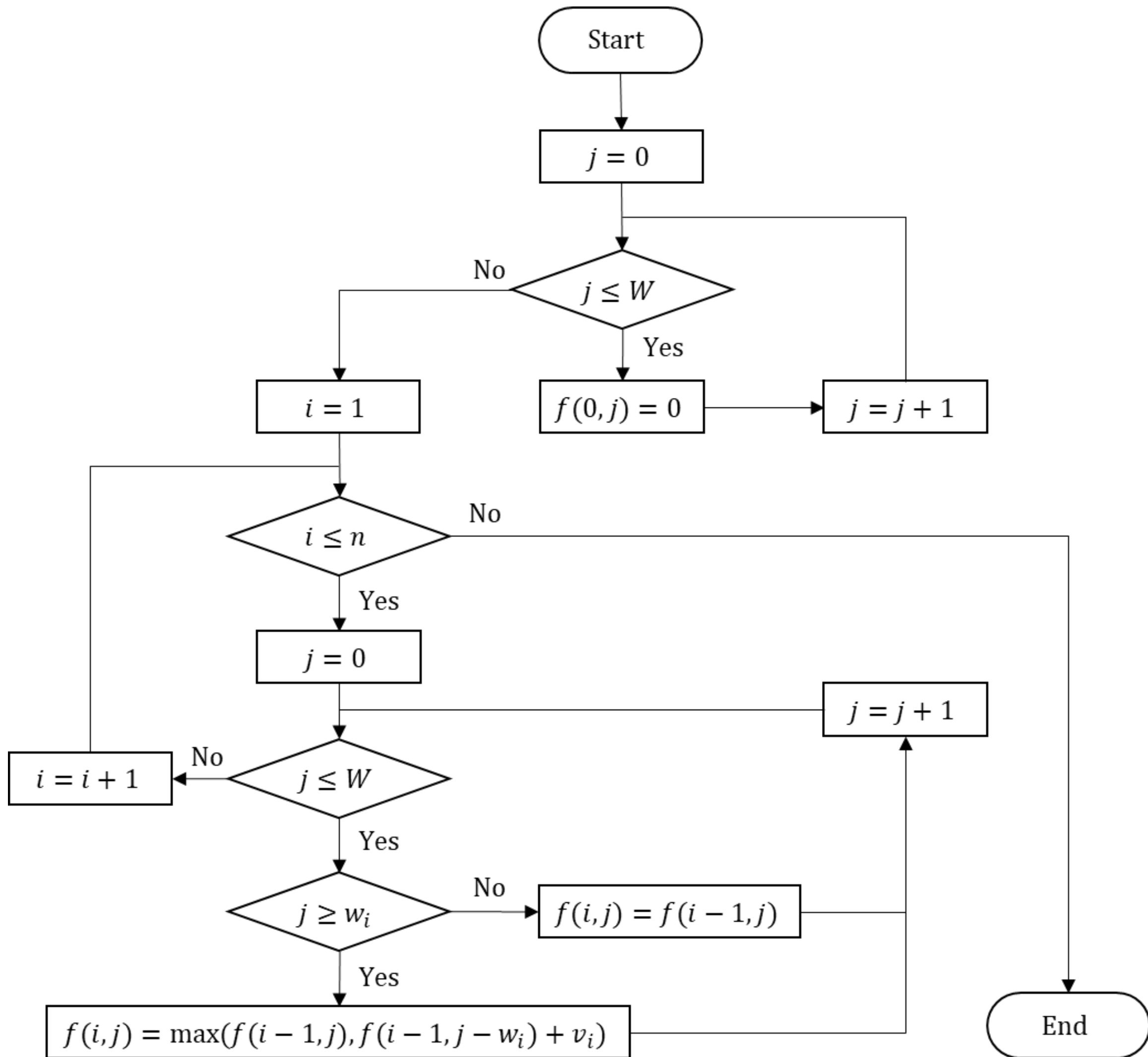

Figure 4: The TDP algorithm flowchart.

In order to determine which items are put into the knapsack and which items are not, the backtracking method is used. First, compare the final result $f(n, W)$ with $f(n-1, W)$. If $f(n, W) = f(n-1, W)$, then the nth item is not put into the knapsack in the solution, otherwise, the nth item is put into the knapsack. If the nth item is not put into the knapsack, then subproblem $\langle n-1, W \rangle$ is considered, i.e., compare $f(n-1, W)$ with $f(n-2, W)$ to determine whether the $(n-1)$th item is put into the knapsack. If the nth item is put into the knapsack, subproblem $\langle n-1, W-w_n \rangle$ is considered, i.e., $f(n-1, W-w_n)$ and $f(n-2, W-w_n)$ are compared. The backtracking continues until the number of items in

the current subproblem reaches zero, at which point the decision for the first item has been determined. The backtracking flowchart for the TDP algorithm is shown in Figure 5.

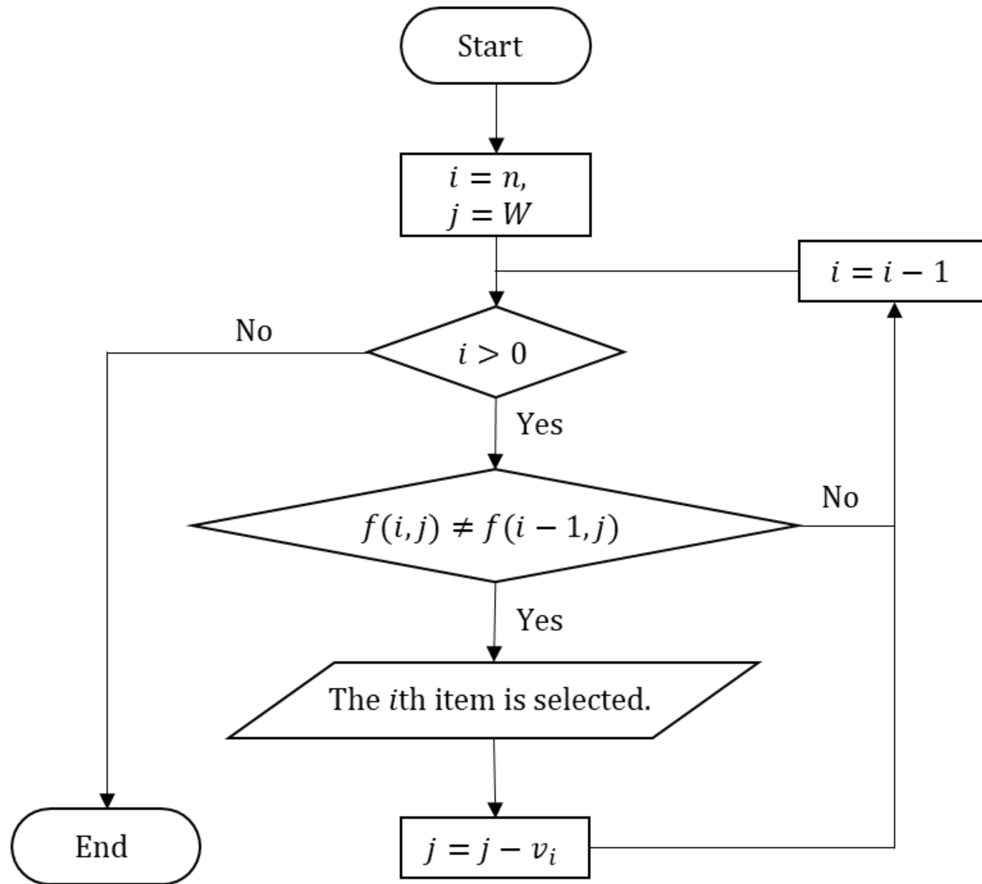

Figure 5: The backtracking flowchart for the TDP algorithm.

3.1.3 Complexity Analysis

From the above analysis, the core of the TDP algorithm is to calculate values in a two-dimensional array with W rows and n columns, which requires $n \times W$ iterations. The operations performed are relational, arithmetic, and assignments each iteration, and their time complexity is constant. In the backtracking, n iterations are performed to obtain the decisions for n items, and the time complexity of each iteration is also constant. Therefore, the total time complexity of the TDP algorithm is $O(nW)$.

The TDP algorithm appears to have polynomial time complexity but is actually a pseudo-polynomial algorithm [21]. Its time complexity is related to the capacity of the knapsack. When the capacity of the knapsack is a function of the number of items, e.g.,

$W = 2^n$, the time complexity is $O(n2^n)$, so the algorithm is not polynomial time. The result of the algorithm is stored in an $n \times W$ array, so the space complexity is $O(nW)$.

3.2 Efficient Dynamic Programming

3.2.1 Model Analysis

Dynamic programming algorithms can be implemented in two ways, forward chaining and backward chaining. Generally, if the initial state of the problem is deterministic but the end state is unknown, it can be realized using forward chaining. Conversely, if the end state is deterministic but the initial state is unknown, it can be realized using backward chaining. For the knapsack problem, both the initial and end states are deterministic, so either forward or backward chaining can be used.

In the TDP algorithm, both fewer items and a smaller knapsack are used to reduce the size of the problem. The solution to problem $\langle i, j \rangle$ depends on two subproblems in the previous iteration as the computation of $f(i,j)$ depends on $f(i-1,j)$ and $f(i-1,j-w_i)$. In the two-dimensional array that stores the solutions to all subproblems, $f(i-1,j)$ is directly above $f(i,j)$ and $f(i-1,j-w_i)$ is on the left of $f(i-1,j)$. Therefore, a one-dimensional state can be considered. State $f(j)$ represents the maximum value that can be obtained when the capacity of the knapsack is j, so the value of $f(j)$ depends on the values of $f(j)$ and $f(j-w_i)$ in the previous iteration. To ensure that $f(j-w_i)$ in the previous iteration is not overwritten by the updated $f(j-w_i)$ in the new iteration when calculating $f(j)$, $f(j)$ needs to be determined before $f(j-w_i)$ each iteration. Therefore, backward chaining is used in this case. The state transition equation is modified to

$$f(j) = \begin{cases} \max(f(j), f(j-w_i) + v_i), & j \geq w_i \\ f(j), & 0 \leq j < w_i \end{cases} \tag{3.6}$$

and the initial condition is

$$f(j) = 0, \qquad j \geq 0 \tag{3.7}$$

An array is created to store the values of $f(j)$. The length of the array is equal to the capacity of the knapsack W. As in TDP, in the first iteration, the number of items that can be selected is 0, so $f(j) = 0$. Therefore, all values in the array are initialized to 0. The number of items that can be selected increases each iteration, and the array is updated in reverse order. Different from TDP, when the weight of the new item is greater than the capacity of the knapsack, the maximum value that can be obtained is the same as when

there is no new item, so $f(j)$ does not need to be updated in this case. In TDP, the value of $f(i,j)$ needs to be updated to be the same as $f(i-1,j)$. This difference can save some running time. The new state transition equation is

$$f(j) = \max(f(j), f(j-w_i) + v_i), \qquad j \geq w_i \tag{3.8}$$

and the initial condition remains unchanged.

For each subproblem, the new item needs to be considered for inclusion in the knapsack. During this process, the decision with the larger value of the indicator function should be taken. In the nth iteration, the first to nth items can be selected. At this time, $f(W)$ stores the maximum value that can be obtained when the capacity of the knapsack is W, which is the solution to the original problem. The EDP algorithm flowchart is shown in Figure 6.

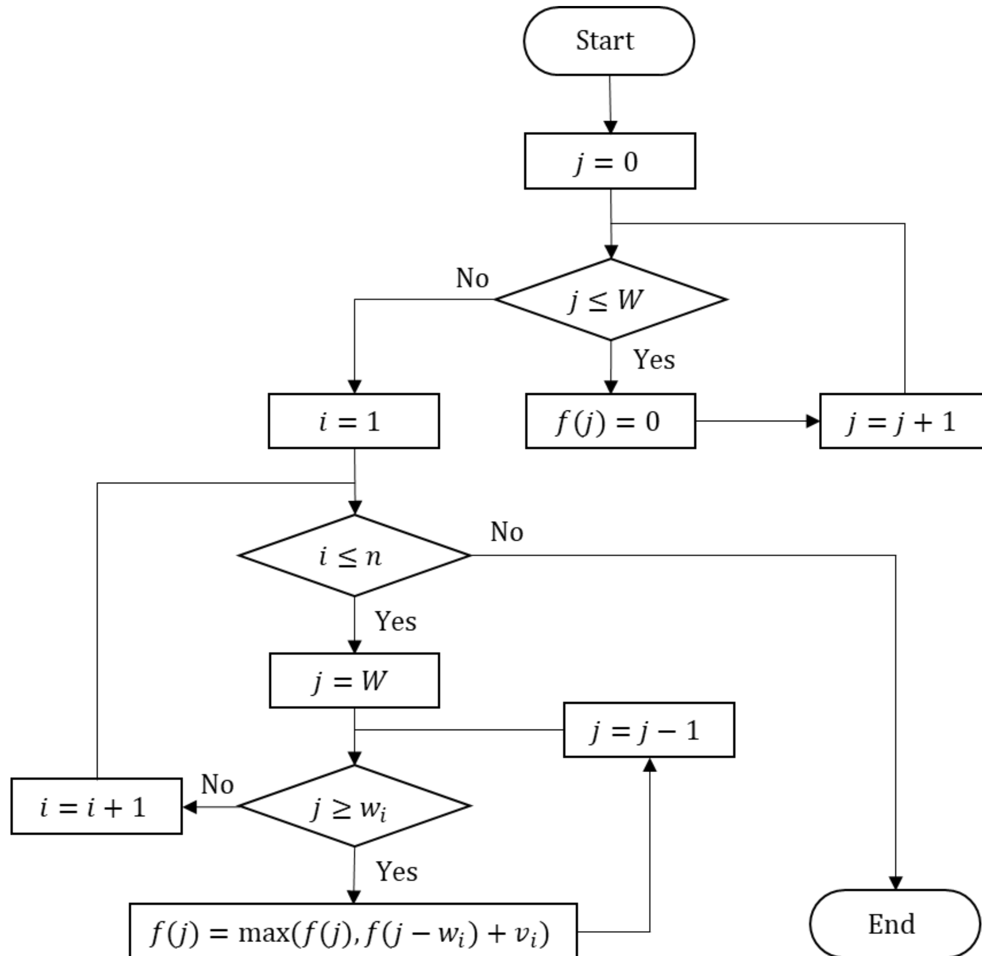

Figure 6: The EDP algorithm flowchart.

3.2.2 Complexity Analysis

From the above analysis, EDP updates the values of a one-dimensional array each iteration. The length of the array is W, and the number of iterations is n. Relational and arithmetic operations, and assignments are needed each iteration, and their time complexity is constant. Therefore, the total time complexity of the EDP algorithm is $O(nW)$. However, each iteration does not update the values of all elements in the array as EDP only updates the solutions to the subproblems in which the capacity of the knapsack is greater than or equal to the weight of the new item. For the ith iteration, the number of elements which are not updated is w_i. In comparison, the number of subproblems that need to be solved by TDP is nW, while the number of subproblems that need to be solved by EDP is $nW - \sum_{i=1}^{n} w_i$.

EDP only uses a one-dimensional array to store the results, so the space complexity is $O(W)$. Instead of keeping the results of all subproblems and the state transition relationships, EDP overwrites the results of old subproblems with the results of new subproblems during the iterations. Therefore, EDP cannot use backtracking to determine the decision for each item, which greatly restricts its applications.

3.3 Hybrid Dynamic Programming

3.3.1 Divide-and-conquer

The disadvantage of TDP is that to determine which items are put into the knapsack, it must keep the entire state transition table, so the space complexity is $O(nW)$. For large-scale problems, keeping the entire table is space-consuming. In this project, divide-and-conquer is used to eliminate the backtracking process to save space.

The divide-and-conquer algorithm divides a complex problem into multiple subproblems with the same form, and then keeps dividing the subproblems into smaller subproblems until the final subproblems can be solved simply and directly. The solution to the original problem is a combination of the solutions to the subproblems.

A problem of size n is divided into k smaller subproblems, $1 < k \leq n$. These subproblems should be independent and solvable, and solutions to these subproblems form the solution to the original problem. The subproblems are usually smaller forms of the original problem, which facilitates the use of recursion [18]. Then the repeated use of divide-and-conquer allows the subproblem to shrink in scale while maintaining the same

form as the original problem. Finally, the scale of the subproblem is reduced to the point where it is easy to find the solution directly.

Problems that can be solved by divide-and-conquer generally have the following characteristics.

(1) The problem becomes easier to solve when the scale is reduced. This characteristic is satisfied by most problems since the computational complexity of most problems decreases with a decrease in scale.

(2) The problem can be divided into similar smaller-scale subproblems. This is the premise of applying divide-and-conquer.

(3) The solutions to the subproblems can be used to obtain the solution to the original problem, i.e., the problem has the optimal substructure property. The use of divide-and-conquer depends on whether the problem satisfies this property. The knapsack problem has been shown to satisfy this property.

(4) The subproblems of the original problem are independent. This characteristic concerns the efficiency of the divide-and-conquer algorithm. If the subproblems are not independent, it will repeatedly solve the subproblems. In this case, dynamic programming is a better choice.

Consider a problem of size n divided into k subproblems of size $\frac{n}{m}$. Assume that the time complexity of dividing the original problem into k subproblems and combining the solutions of these subproblems into a solution to the original problem is $f(n)$. Then, the time required for the divide-and-conquer algorithm to solve a problem of size n is

$$T(n) = kT\left(\frac{n}{m}\right) + f(n) \tag{3.9}$$

3.3.2 Model Analysis

As in [22, 23], divide-and-conquer is used to solve the knapsack problem. This requires modification of the subproblem and state representations. In TDP and EDP, subproblem $\langle i, j \rangle$ is considered, and state $f(i, j)$ represents the maximum value that can be obtained from the first to ith items when the capacity of the knapsack is j. In the hybrid algorithm, subproblem $\langle i, j, k \rangle$ is considered, which means that the ith to $(j-1)$th items can be selected, and the capacity of the knapsack is k. $f(i, j, k)$ is the maximum value that can be obtained from subproblem $\langle i, j, k \rangle$. The idea of the hybrid algorithm is as follows.

First, for problem $\langle i, j, k \rangle$, the items are divided into two sets. It is better to divide the items into two sets of similar size as this will minimize the depth of the problem tree. Suppose the items are divided at mid. The first set contains items with indexes from i to $mid - 1$, and the second set contains items with indexes from mid to $j - 1$. At this point, the original problem is divided into two subproblems $\langle i, mid, k_1 \rangle$ and $\langle mid, j, k_2 \rangle$, $0 \leq k_1, k_2 \leq k$.

For the first subproblem $\langle i, mid, k_1 \rangle$, EDP is used to find the maximum value that can be obtained. This gives an array of k elements with the maximum values that can be obtained from the first item set when the capacity of the knapsack is from 0 to k. Then, EDP is used to find the maximum value that can be obtained for the second subproblem $\langle mid, j, k_2 \rangle$. Note that this subproblem is different from the previous because the items that can be selected no longer start with the first item. An array with the maximum values that can be obtained from the second item set is obtained when the capacity of the knapsack is from 0 to k.

The knapsack is then divided into two spaces, one with capacity k_1 for items from the first set, and the other with capacity $k - k_1$ for items from the second set. The best way to allocate the total capacity k to these two item sets needs to be found. That is, maximize the total value of the items in the knapsack

$$f(i, j, k) = \text{maximize } [f(i, mid, k_1) + f(mid, j, k - k_1)], \qquad 0 \leq k_1 \leq k \tag{3.10}$$

Therefore, consider the elements in the solution array of the first subproblem in forward order and the elements in the solution array of the second subproblem in reverse order, and add them together each time. The sums obtained are compared to find the optimal division point for the capacity of the knapsack. At this point, although the maximum value that can be obtained in problem $\langle i, j, k \rangle$ has been found, there is still no information about item selection.

In order to know which items are in the optimal solution, the subproblems need to be iteratively divided. For each subproblem of the form $f(i, j, k)$, divide it into two subproblems $\langle i, mid, k_1 \rangle$ and $\langle mid, j, k_2 \rangle$, $0 \leq k_1, k_2 \leq k$. Then, the optimal division of the capacity is found that maximizes the sum of the values that can be obtained from the two subproblems. The subproblems are divided until only one item is left in the sets of the subproblems. Then it is only necessary to compare the weight of the item with the capacity

allocated to the subproblem. If the allocated capacity is larger than the weight of the item, the item is put into the knapsack. Conversely, the item is not put into the knapsack. This way, decisions for each item are obtained without using backtracking. This new hybrid algorithm is called the divide-and-conquer and dynamic programming (DCDP) algorithm. The DCDP algorithm flowchart is shown in Figure 7.

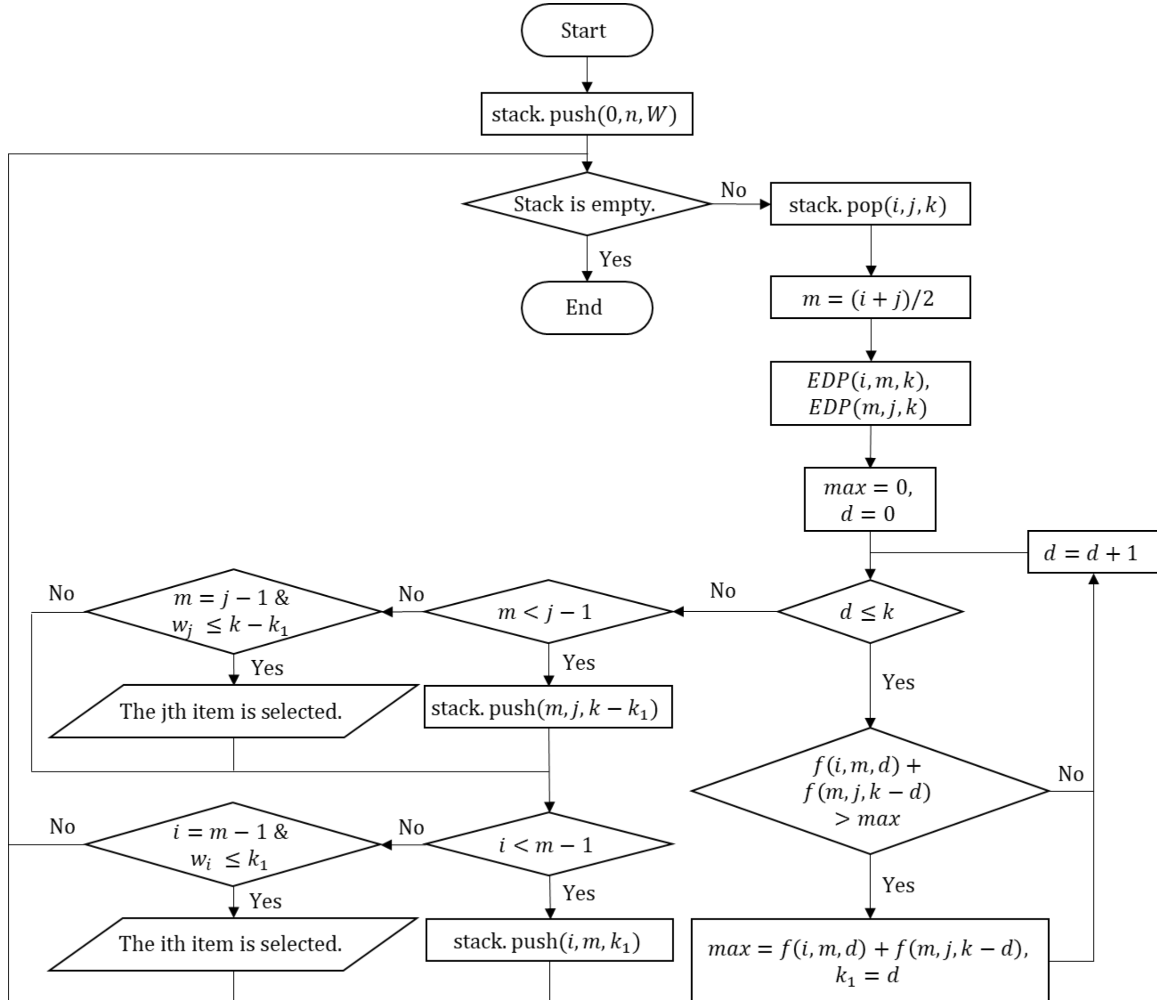

Figure 7: The DCDP algorithm flowchart.

A sketch of the proof that the DCDP algorithm provides the optimal solution is as follows. When the item set has only one item, the weight of the item and the allocated capacity are compared to determine whether the item is put into the knapsack or not. Assume that the algorithm works correctly when there are n_1 items in the item set. Then, when the item set has n $(n > n_1)$ items, the algorithm will divide the set into two sets of size n_1 and n_2, $n_1 \approx n_2$. The algorithm must allocate some capacity for items from set n_1

and the remaining capacity for items from set n_2 (the allocated capacity can be 0). Thus, the algorithm will traverse the results for the two sets and combine them into an optimal solution.

3.3.3 Complexity Analysis

For problem $\langle i, j, k \rangle$, the time complexity of executing EDP is $O[(j-i)k]$ where $(j-i)$ is the number of items and k is the capacity of the knapsack. In order to find the optimal division of the capacity of the knapsack, DCDP needs to traverse the EDP results, which has time complexity $O(j-i)$. The time complexity of determining whether an item is put into the knapsack is constant.

A problem is divided into two subproblems, creating a binary tree. If the original problem with n items has depth 0, then the depth of the binary tree is $\log_2 n$. Each problem of depth d has at most $\lceil n/2^d \rceil$ items, and the sum of the allocated capacity of all problems of the same depth is equal to the total capacity W. Therefore, the sum of the time complexities of the problems at depth d is $O(\frac{nW}{2^d})$. The total time complexity of DCDP is

$$\sum_{d=0}^{\log_2 n} O\left(\frac{nW}{2^d}\right) = O(nW) \tag{3.11}$$

The DCDP algorithm executes EDP each iteration, which requires $O(W)$ space to save the results. However, once the optimal allocation scheme of the current capacity to the two subproblems is determined, the results obtained by EDP are no longer needed, so this space can be reused. The algorithm uses a stack to store the capacities allocated to different subproblems. Since depth-first search is used for traversal, the number of values stored in the stack equals the depth of the binary tree, so $O(\log_2 n)$ space is required. Therefore, the total space complexity of DCDP is $O(W + \log_2 n)$.

Chapter 4: Experiments and Results

The difficulty of solving the knapsack problem is related to the correlation between the weights and values of the items. In [15], knapsack problems were divided into six categories. Current research is mainly focused on the following three categories.

I. Uncorrelated: The weights w and values v of the items are randomly distributed on $[1, R]$, $R \in N^+$.

II. Weakly Correlated: The weights w are randomly distributed on $[1, R]$, and the values v are randomly distributed on $\left[w - \frac{R}{10}, w + \frac{R}{10}\right]$.

III. Strongly Correlated: The weights w are randomly distributed on $[1, R]$, and the values are $v = w + \frac{R}{10}$.

In this project, experiments are conducted on the above three categories of knapsack problems. The operating system used for the experiments is Windows 10 Home. The processor is AMD Ryzen 5 5600H with Radeon Graphics and the memory size is 16 GB. The programming language used is Java and the programming platform is IntelliJ IDEA Ultimate.

4.1 Time Complexity Experiments

In order to test knapsack problems of different scales, the number of items n varies from $1k$ to $100k$, and the capacity of the knapsack W varies from 100 to $1k$. For each problem, tests are repeated three times and the average of the results is given.

The performance of the TDP, EDP, and DCDP algorithms when $n = 1k$ and $W = 100$, $n = 10k$ and $W = 100$, and $n = 100k$ and $W = 100$ is shown in Tables 1, 2, and 3, respectively. The performance of the three algorithms for uncorrelated, weakly correlated, and strongly correlated problems when $W = 100$ is given in Figures 8, 9, and 10, respectively. These results show that when the correlation between the weights and values of the items becomes greater, the time taken by the three algorithms to solve the problem decreases. For example, when $n = 1k$ and $W = 100$, the time taken by EDP to solve the uncorrelated problem is 2.43 ms. The time taken by EDP to solve the weakly correlated problem reduces to 0.72 ms. The time taken by EDP to solve the strongly correlated problem reduces to 0.20 ms. In addition, as the number of items increases, the time taken

by the three algorithms to solve the problem increases. For example, when n increases from $10k$ to $100k$, the time taken by TDP to solve the uncorrelated problem goes up from 11.41 ms to 52.22 ms. For the same problem, the time taken by DCDP is less than that of TDP but more than that of EDP. For example, when $n = 10k$ and $W = 100$, the time taken by TDP to solve the strongly correlated problem is 4.71 ms. The time taken by DCDP to solve this problem is 3.60 ms and the time taken by EDP is only 0.52 ms. These results show that DCDP has an advantage in solving weakly correlated problems compared to the other categories. For weakly correlated problems, the time taken by DCDP is close to that of EDP. For example, when $n = 100k$ and $W = 100$, the time taken by DCDP to solve the weakly correlated problem is 1.7 times that of EDP. In comparison, the time taken by TDP to solve this problem is 7.1 times that of EDP and 4.2 times that of DCDP.

Table 1: Performance of three algorithms when $n = 1k$ and $W = 100$.

Category	Algorithm	Worst (ms)	Best (ms)	Average (ms)
Uncorrelated	TDP	5.07	3.85	4.28
	EDP	2.65	2.24	2.43
	DCDP	4.52	4.05	4.32
Weakly Correlated	TDP	0.66	0.55	0.59
	EDP	0.88	0.82	0.85
	DCDP	0.81	0.63	0.72
Strongly Correlated	TDP	0.60	0.54	0.56
	EDP	0.21	0.19	0.20
	DCDP	0.59	0.39	0.51

Table 2: Performance of three algorithms when $n = 10k$ and $W = 100$.

Category	Algorithm	Worst (ms)	Best (ms)	Average (ms)
Uncorrelated	TDP	12.51	9.25	11.41
	EDP	5.64	4.64	4.99
	DCDP	12.92	9.91	10.96
Weakly Correlated	TDP	9.32	5.76	6.99
	EDP	0.88	0.55	0.72
	DCDP	4.87	3.95	4.28
Strongly Correlated	TDP	5.26	4.39	4.71
	EDP	0.65	0.44	0.52
	DCDP	3.98	3.40	3.60

Table 3: Performance of three algorithms when $n = 100k$ and $W = 100$.

Category	Algorithm	Worst (ms)	Best (ms)	Average (ms)
Uncorrelated	TDP	53.02	51.65	52.22
	EDP	9.62	8.98	9.39
	DCDP	32.29	28.43	30.05
Weakly Correlated	TDP	50.05	48.13	48.96
	EDP	7.16	6.68	6.87
	DCDP	13.37	10.78	11.73
Strongly Correlated	TDP	33.30	31.15	32.25
	EDP	5.87	4.36	4.88
	DCDP	17.61	16.82	17.09

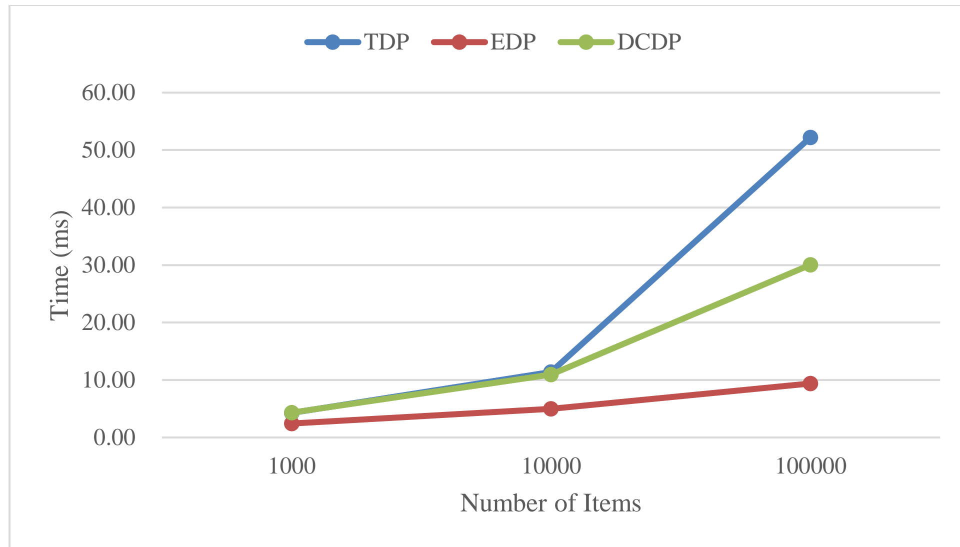

Figure 8: Performance for uncorrelated problems when $W = 100$.

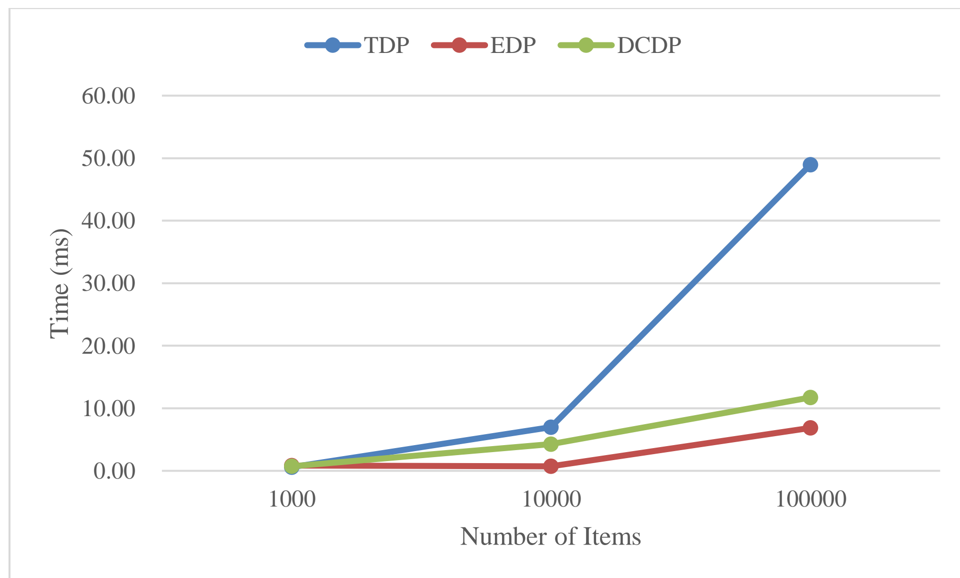

Figure 9: Performance for weakly correlated problems when $W = 100$.

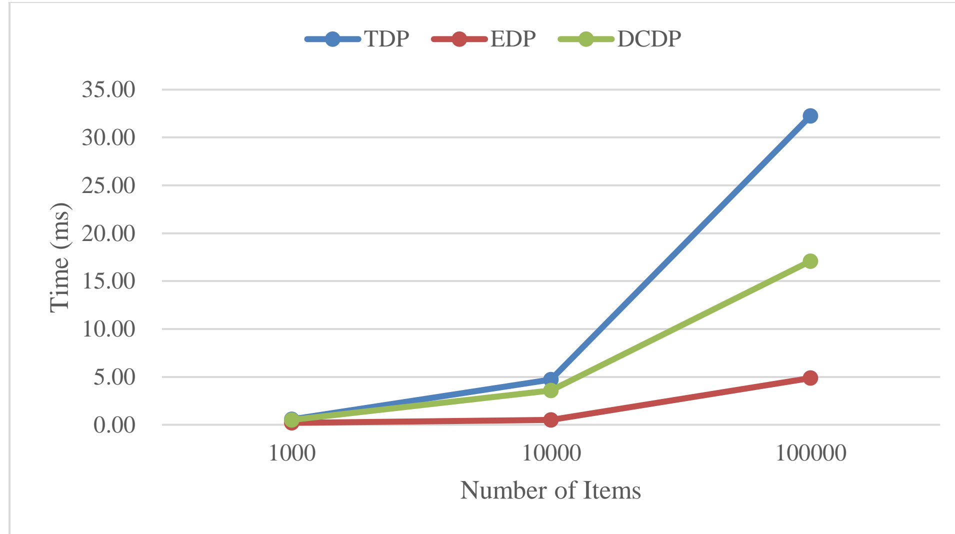

Figure 10: Performance for strongly correlated problems when $W = 100$.

The performance of the three algorithms when $n = 1k$ and $W = 1k$, $n = 10k$ and $W = 1k$, and $n = 100k$ and $W = 1k$ is shown in Tables 4, 5, and 6, respectively. The performance of the three algorithms for uncorrelated, weakly correlated, and strongly correlated problems when $W = 1k$ is given in Figures 11, 12, and 13, respectively. As before, when the correlation between the weights and values of the items becomes greater, the time taken to solve the problem decreases. Not only as the number of items grows, but also as the capacity of the knapsack grows, the time taken by the three algorithms increases. For example, when the capacity of the knapsack increases from 100 to $1k$ for the uncorrelated knapsack problem with $n = 10k$, the time taken by EDP increases from 4.99 ms to 13.48 ms. In addition, the time taken by the same algorithm to solve problems with a ten-fold increase in n or W is quite close, which confirms the previous analysis that the time complexities of the three algorithms are $O(nW)$. For example, when $n = 100k$ and $W = 100$, the time taken by TDP and DCDP to solve the uncorrelated problem is 52.22 and 30.05 ms, respectively. When $n = 10k$ and $W = 1k$, the time taken by TDP and DCDP is 55.70 and 27.59 ms, respectively. As before, the time taken by DCDP is less than that of TDP but more than that of EDP when solving the same problem. These results show that compared with TDP, DCDP has a significant advantage in time complexity, especially when solving weakly correlated problems.

Table 4: Performance of three algorithms when $n = 1k$ and $W = 1k$.

Category	Algorithm	Worst (ms)	Best (ms)	Average (ms)
Uncorrelated	TDP	11.02	10.91	10.95
	EDP	7.42	5.90	6.57
	DCDP	13.23	10.11	11.93
Weakly Correlated	TDP	8.66	6.00	6.98
	EDP	3.60	3.22	3.38
	DCDP	2.02	1.90	1.94
Strongly Correlated	TDP	4.42	4.03	4.21
	EDP	0.83	0.80	0.82
	DCDP	2.42	1.89	2.07

Table 5: Performance of three algorithms when $n = 10k$ and $W = 1k$.

Category	Algorithm	Worst (ms)	Best (ms)	Average (ms)
Uncorrelated	TDP	57.32	53.40	55.70
	EDP	14.29	12.94	13.48
	DCDP	27.99	27.04	27.59
Weakly Correlated	TDP	43.72	40.13	41.36
	EDP	10.39	9.69	9.94
	DCDP	17.34	17.02	17.18
Strongly Correlated	TDP	17.46	17.11	17.29
	EDP	7.80	7.10	7.42
	DCDP	19.36	14.10	16.73

Table 6: Performance of three algorithms when $n = 100k$ and $W = 1k$.

Category	Algorithm	Worst (ms)	Best (ms)	Average (ms)
Uncorrelated	TDP	430.0	390.6	404.5
	EDP	80.29	77.92	79.01
	DCDP	179.7	175.0	177.0
Weakly Correlated	TDP	457.7	435.4	443.3
	EDP	74.38	73.79	74.00
	DCDP	150.5	149.3	150.0
Strongly Correlated	TDP	231.7	220.9	224.9
	EDP	71.10	70.69	70.84
	DCDP	148.8	136.8	141.0

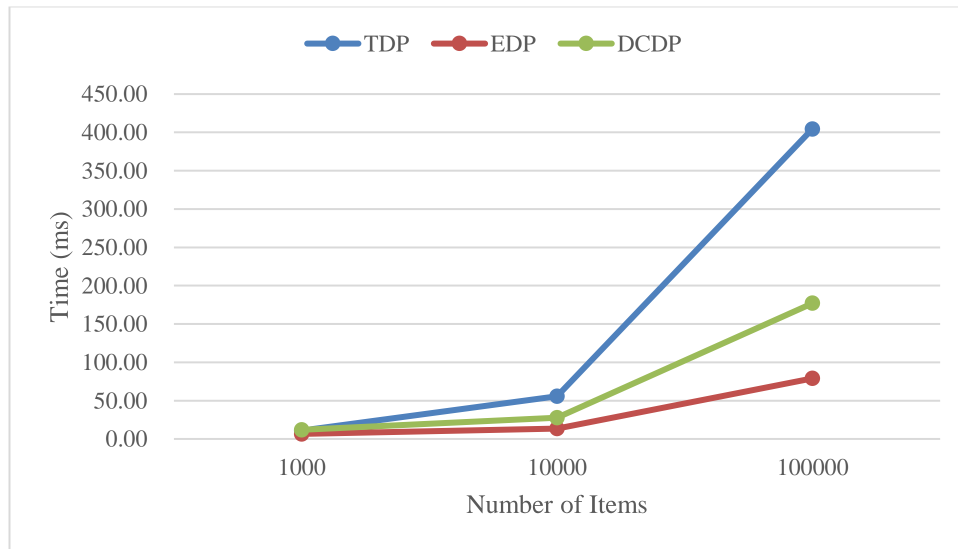

Figure 11: Performance for uncorrelated problems when $W = 1k$.

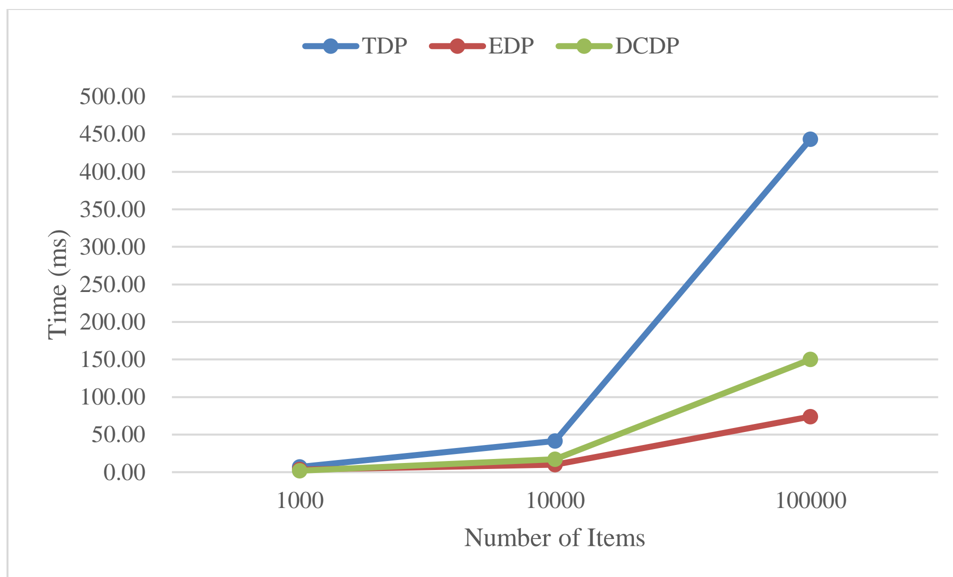

Figure 12: Performance for weakly correlated problems when $W = 1k$.

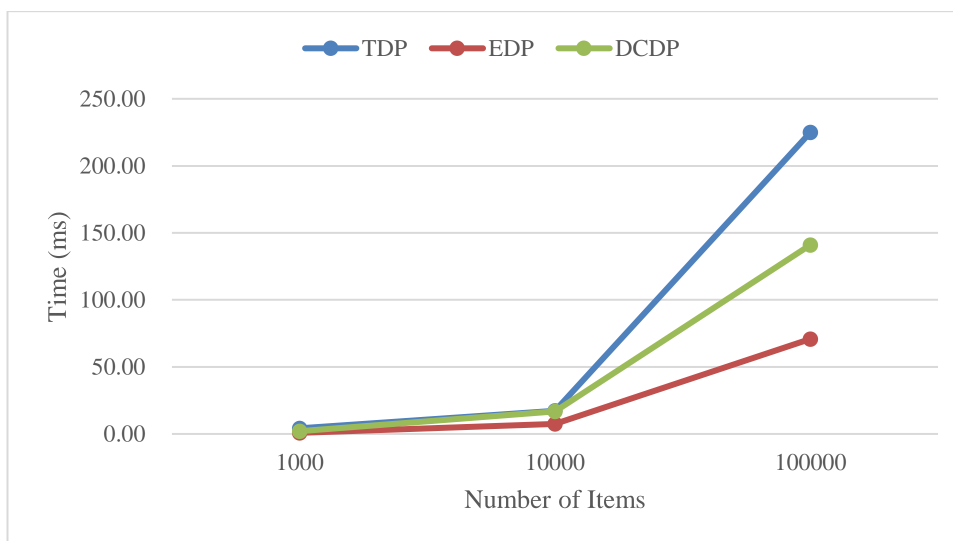

Figure 13: Performance for strongly correlated problems when $W = 1k$.

4.2 Space Complexity Experiments

JProfiler is used here to calculate the space usage of the algorithms. It is a leading profiler for profiling on the Java virtual machine. JProfiler helps resolve performance bottlenecks and identify memory leaks, and its CPU, Thread, and Memory analysis functions are particularly powerful. Memory analysis using JProfiler is shown as Figure 14. This indicates there are analysis options on the left side such as Memory, Classes, and Threads.

Memory analysis is used here to determine the allocated, used, and remaining space of the running program.

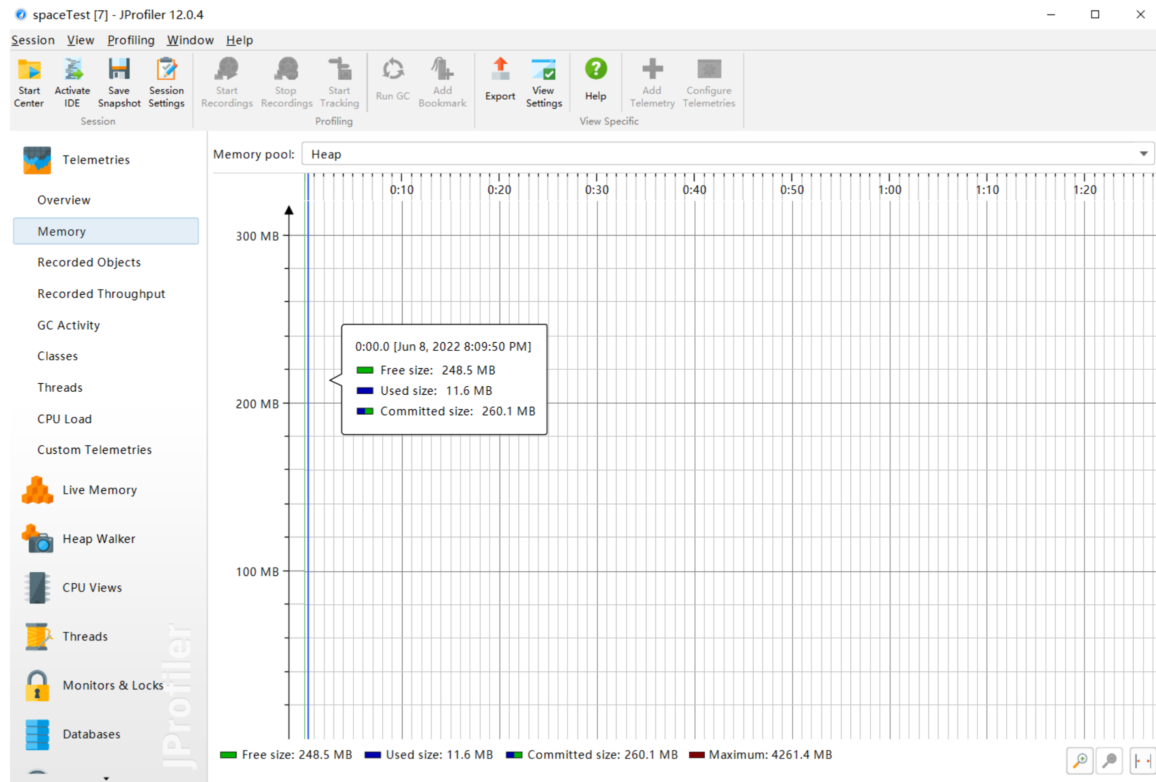

Figure 14: Memory analysis using JProfiler.

In order to determine the relationship between the space usage of an algorithm and the scale of the problem, the number of items n varies from 10 to 100, and the capacity of the knapsack W varies from 10^5 to 10^7. Since the space usage is independent of the knapsack problem category, only uncorrelated knapsack problems are considered in this section. The space usage of the TDP, EDP and DCDP algorithms is shown in Table 7. The space usage of the TDP, EDP and DCDP algorithms when $n = 10$ is shown in Figures 15 and 16. The space usage of the TDP, EDP and DCDP algorithms when $n = 100$ is shown in Figures 17 and 18. These results show that for the same problem, the space usage of EDP and DCDP is almost the same, and is much less than that of TDP. For example, when $n = 100$ and $W = 10^7$, both EDP and DCDP use 48.46 MB to solve the problem, while TDP uses 1765 MB. In addition, the space used by TDP increases as n or W increases. For example, when $W = 10^5$, the space used by TDP increases from 11.41 MB to 44.75 MB as n

increases from 10 to 100. When $n = 10$, the space used by TDP increases from 11.41 MB to 52.55 MB as W increases from 10^5 to 10^6. However, the space used by EDP and DCDP only increases with W, and their space consumption is independent of n. Further, when $n \times W$ for TDP is equal to W for EDP and DCDP, the space usage is similar. For example, when $n = 10$ and $W = 10^6$, the space used by TDP is 52.55 MB, and when $n = 100$ and $W = 10^5$, the space used is 44.75 MB. For $W = 10^7$, the space used by EDP and DCDP is about 48 MB for both values of n. This confirms the previous analysis that the space complexity of TDP is $O(nW)$ while that of EDP and DCDP is $O(W)$. These results show that compared with TDP, DCDP has a significant advantage in space complexity, especially when there are a large number of items in the problem.

Table 7: Space usage of three algorithms.

Scale \ Algorithm		TDP	EDP	DCDP
$n = 10$	$W = 10^5$	11.41 MB	7.89 MB	7.80 MB
	$W = 10^6$	52.55 MB	11.60 MB	11.88 MB
	$W = 10^7$	207.7 MB	48.01 MB	48.46 MB
$n = 100$	$W = 10^5$	44.75 MB	7.50 MB	7.85 MB
	$W = 10^6$	146.4 MB	11.60 MB	11.60 MB
	$W = 10^7$	1765 MB	48.46 MB	48.46 MB

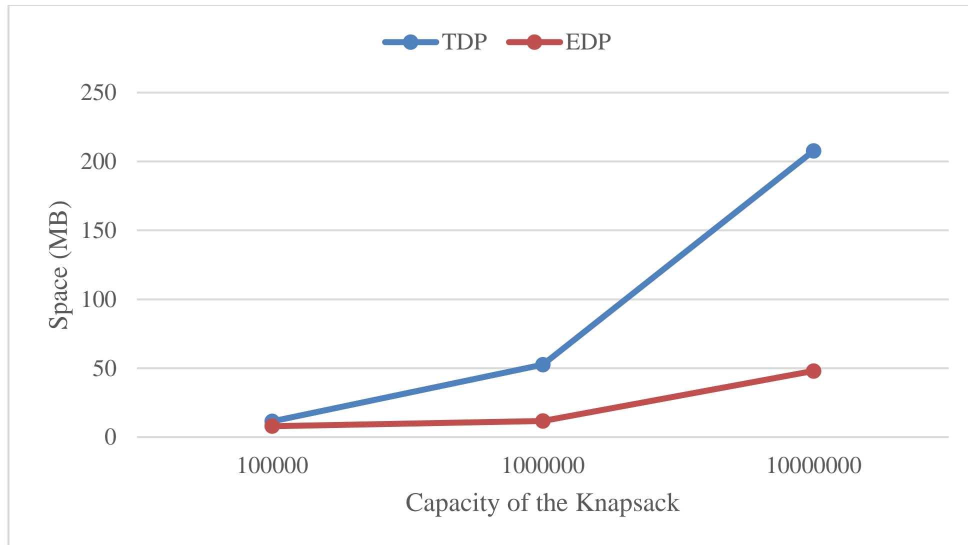

Figure 15: Space usage of the TDP and EDP algorithms when $n = 10$.

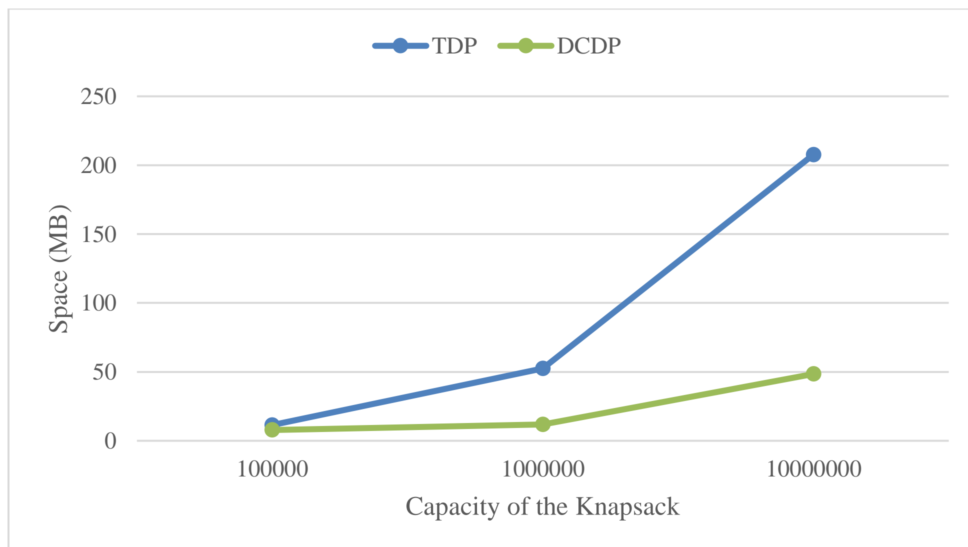

Figure 16: Space usage of the TDP and DCDP algorithms when $n = 10$.

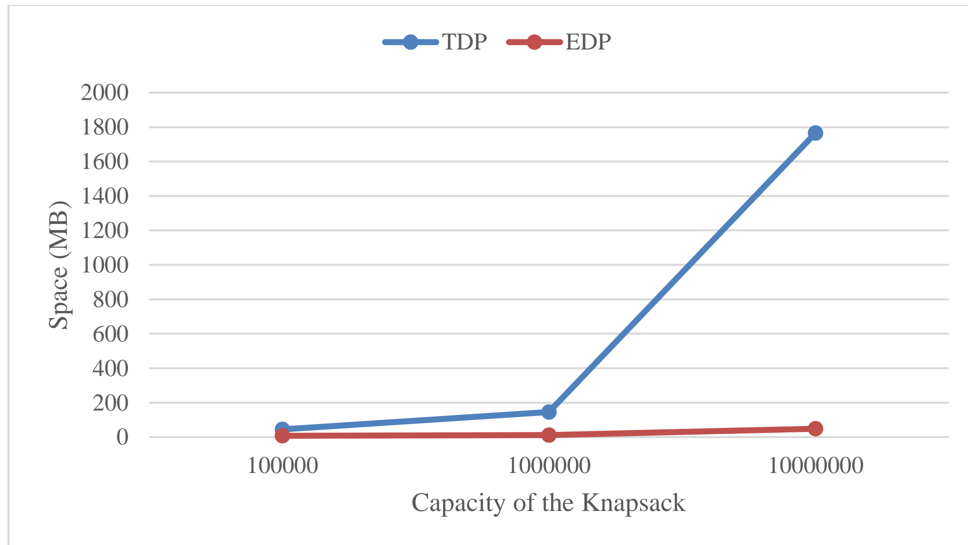

Figure 17: Space usage of the TDP and EDP algorithms when $n = 100$.

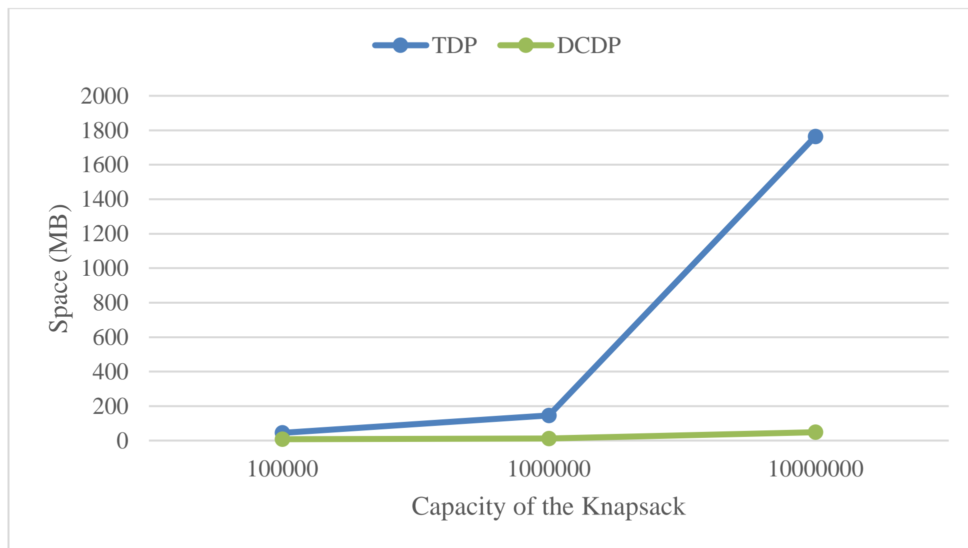

Figure 18: Space usage of the TDP and DCDP algorithms when $n = 100$.

Chapter 5: Conclusion and Future Work

5.1 Conclusion

The knapsack problem is a classic non-deterministic polynomial complete problem and has a wide range of applications in many fields [2]. Many practical problems can be transformed into knapsack problems such as storage space allocation and loan portfolio optimization. All integer linear programming problems can be transformed into 0-1 integer linear programming problems, while 0-1 integer linear programming problems can be transformed into knapsack problems [3]. The knapsack problem is important in applications such as information encryption, project investment, budget control, product purchase, cargo freight, and network security.

The knapsack problem is a multistage decision problem so it can be divided into subproblems. Traditional Dynamic Programming (TDP) and Efficient Dynamic Programming (EDP) are algorithms for solving knapsack problems. For a problem with n items and knapsack capacity W, TDP uses an $n \times W$ two-dimensional array to store the optimal results of the subproblems, and backtracking is used to determine which items are in the optimal solution. Its time complexity and space complexity are both $O(nW)$. EDP uses a one-dimensional array of length W to store the results, thus it only provides the optimal value and cannot provide information on item selection. The time complexity of EDP is also $O(nW)$ while the space complexity is $O(W)$.

A new hybrid algorithm called divide-and-conquer and dynamic programming (DCDP) was proposed. This algorithm divides the item set into two subsets of similar size and allocates the capacity of the knapsack to these subsets. For the subsets, EDP is used to find the maximum values that can be obtained when allocating different capacities to them. Then the results are compared to find the capacity allocation that maximizes the sum of the values from the two subsets. The problem is iteratively divided until there is only one item in the item set. Then, it is only necessary to compare the allocated capacity and the weight of the item to determine whether the item is put into the knapsack. In this way, the backtracking for item selection is eliminated, so the state transition table does not need to be stored. The time complexity of DCDP is $O(nW)$ and the space complexity is $O(W + \log_2 n)$.

Finally, experiments were conducted to evaluate the performance of TDP, EDP, and DCDP. In the time complexity experiments, n was varied from $1k$ to $100k$, and W from 100 to $1k$. Three categories of knapsack problems, namely uncorrelated, weakly correlated, and strongly correlated, were considered. The results obtained show that as n and W increase, the time taken by the three algorithms also increases. The time taken by DCDP is less than that of TDP but more than that of EDP. Compared with TDP, DCDP has a significant advantage in time complexity, especially when solving weakly correlated problems. In the space complexity experiments, n was varied from 10 to 100, and W from 10^5 to 10^7. The results obtained show that as W increases, the space usage of the three algorithms also increases. The space usage of TDP increases as n increases. However, the space usage of EDP and DCDP is independent of n. Thus, the space complexity of DCDP is much better than that of TDP, especially when solving problems with a large number of items.

5.2 Future Work

The trend of solving knapsack problems is to develop hybrid optimization algorithms that combine multiple algorithms, use artificial intelligence to obtain new optimization algorithms, and parallel processing of existing algorithms. In this project, a hybrid algorithm that combines the divide-and-conquer and dynamic programming algorithms was proposed. Compared with TDP, this algorithm reduces the time complexity and space complexity. However, improvements can still be made in the following areas.

(1) Use the branch-and-bound algorithm to improve the performance of the hybrid algorithm.

(2) Explore parallel processing of the hybrid algorithm to improve the running efficiency.

(3) There is a broad range of knapsack problems with different characteristics and requirements. Thus, the hybrid algorithm can be improved according to the needs of specific problems.

Bibliography

[1] Syslo M M, Deo N, Kowalik J S. Discrete Optimization Algorithms: With Pascal Programs. Dover Publications, Mineola, NY, USA, 1983.

[2] Garey M R, Johnson D S. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York, NY, USA, 1983.

[3] Chinneck J W. Practical Optimization: A Gentle Introduction. 2022.

[4] Dantzig G B. Discrete-Variable Extremum Problems. Operations Research, 1957, 5(2): 266-277.

[5] Horowitz E, Sahni S. Computing Partitions with Applications to the Knapsack Problem. Journal of the ACM, 1974, 21(2): 277-292.

[6] Martello S, Toth P. An Upper Bound for the 0-1 Knapsack Problem and a Branch and Bound Algorithm. European Journal of Operational Research, 1977, 1(3): 169-175.

[7] Balas E, Zemel E. An Algorithm for Large 0-1 Knapsack Problems. Operations Research, 1980, 28(5): 1130-1154.

[8] Pisinger D. Linear Time Algorithms for Knapsack Problems with Bounded Weights. Journal of Algorithms, 1999, 33(1): 1-14.

[9] Pisinger D. An Expanding-Core Algorithm for the Exact 0-1 Knapsack Problem. European Journal of Operational Research, 1995, 87(1): 175-187.

[10] Pisinger D. A Minimal Algorithm for the Multiple-Choice Knapsack Problem. European Journal of Operational Research, 1995, 83(2): 394-410.

[11] Szeto K Y, Zhang J. Adaptive Genetic Algorithm and Quasi-Parallel Genetic Algorithm: Application to Knapsack Problem. Springer, Berlin, Germany, 2005.

[12] Gao S, Xing Y, Xiao N, et al. A Greedy Leapfrog Algorithm for Solving the 0-1 Knapsack Problem. Computer Science, 2018, 45(7): 73-77.

[13] Cao J, Yin B, Lu X, et al. A Modified Artificial Bee Colony Approach for the 0-1 Knapsack Problem. Applied Intelligence, 2017, 48(6): 1582-1595.

[14] Xue J, Xiao J, Zhu J. Binary Fireworks Algorithm for 0-1 Knapsack Problem. International Conference on Artificial Intelligence and Advanced Manufacturing, 2019, 218-222.

[15] Pisinger D. Where Are the Hard Knapsack Problems?. Computers and Operations Research, 2005, 32(9): 2271-2284.

[16] Li D, Qian F, Li L, et al. Research on Dynamic Programming Problems. System Engineering - Theory and Practice, 2007, 27(8): 56-64.

[17] Brandimarte P. From Shortest Paths to Reinforcement Learning: A MATLAB-Based Tutorial on Dynamic Programming. Springer International, Cham, Switzerland, 2021.

[18] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to Algorithms. MIT Press, Cambridge, MA, USA, 2009.

[19] Toth P. Dynamic Programming Algorithms for the 0-1 Knapsack Problem. Computing, 1980, 25(1): 29-45.

[20] Bellman R E. Some Applications of the Theory of Dynamic Programming. Operations Research, 1954, 2(3): 275-288.

[21] Martello S, Pisinger D, Toth P. Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem. Management Science, 1999, 45(3): 414-424.

[22] Hirschberg D S. A Linear Space Algorithm for Computing Maximal Common Subsequences. Communications of the ACM, 1975, 18(6): 341-343.

[23] Xue J. A Unified Approach for Developing Efficient Algorithmic Programs. Journal of Computer Science and Technology, 1997, 12(4): 314-329.