Algorithm Assignment2 Report

Department: RVIS, ID: 41127B041, Name: Liu, Shang-En

1 INTRODUCTION

The 0-1 Knapsack Problem is a classic problem in combinatorial optimization, which appears frequently in real-world applications such as resource allocation, portfolio optimization, and logistics. The problem involves selecting a subset of items, each with a given weight and value, to include in a knapsack such that the total weight does not exceed a specified capacity and the total value is maximized.

This report presents a comparative study of three algorithms for solving the 0-1 Knapsack Problem:

- Top-Down Dynamic Programming (Memoization)
- Bottom-Up Dynamic Programming (Tabulation)
- Greedy Algorithm (based on value-to-weight ratio)

The goal is to evaluate and analyze the performance of these methods in terms of optimality and efficiency under different input conditions.

2 ALGORITHM DESIGN

2.1 TOP-DOWN DYNAMIC PROGRAMMING

The top-down approach uses recursion and memoization to store intermediate results. For each item, the algorithm decides whether to include it or not, and recursively computes the maximum value.

Time Complexity: $O(nW)$, where n is the number of items and W is the knapsack capacity.

Clarification on Time Complexity:
It is important to distinguish between the naive recursive approach and the top-down dynamic programming approach with memoization. While a naive recursive implementation of the 0-1 Knapsack problem exhibits exponential time complexity—specifically $O(2^n)$, due to solving overlapping subproblems repeatedly—the use of memoization in the top-down dynamic programming method eliminates redundant computations. By storing intermediate results in a lookup table, each unique subproblem is computed only once. This optimization reduces the overall time complexity to O(nW), matching the efficiency of the bottom-up approach. Thus, although the algorithm uses recursion in its structure, its performance characteristics align with those of iterative dynamic programming.

2.2 Bottom-Up Dynamic Programming

This method constructs a table dp[i][w], where each entry represents the maximum value achievable using the first i items with a total weight limit w. The final answer resides in dp[n][W].

Time Complexity: $O(nW)$

2.3 Greedy Algorithm

The greedy method sorts items by value-to-weight ratio and selects items starting from the highest ratio, adding as many as possible until the capacity is filled. While fast, it does not guarantee an optimal solution in the 0-1 version of the knapsack problem.

Time Complexity: $O(n \log n)$ due to sorting.

3 Experiment Design

To evaluate the performance and effectiveness of the three implemented algorithms—Top-Down Dynamic Programming, Bottom-Up Dynamic Programming, and the Greedy approach—we conducted a series of tests under consistent experimental conditions. The goal was to observe and compare the solution accuracy and execution time across various input sizes.

We prepared three benchmark datasets:

- test_small.txt: A small-scale instance with 4 items and a knapsack capacity of 10.
- test_medium.txt: A medium-sized instance with 10 items and a knapsack capacity of 50.
- test_large.txt: A large-scale instance with 100 items and a knapsack capacity of 1000.

All algorithms were executed using the same compiled C++ executable, and each test was repeated to ensure result consistency. The execution times were recorded using Python's time module, and output values were extracted from the standard output stream. The system used for testing included:

- **Processor**: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- **RAM**: 8.00 GB
- **OS**: Windows 11 64-bit
- **Compiler**: MinGW with g++ version 8.1.0
- **Python Version**: 3.10.6 with pandas and matplotlib for post-processing

The results were stored in result.csv and visualized through plot_results.py to compare the trade-offs between optimality and performance for each algorithm.

4 Data Analysis

The experimental results reveal clear performance differences among the three algorithms.

To evaluate the effectiveness and efficiency of the implemented algorithms, we conducted experiments on three distinct datasets: test_small.txt, test_medium.txt, and test_large.txt, each varying in the number of items and knapsack capacity. For each dataset, we recorded the maximum total value obtained and the execution time for the Top-Down (DP), Bottom-Up (DP), and Greedy algorithms.

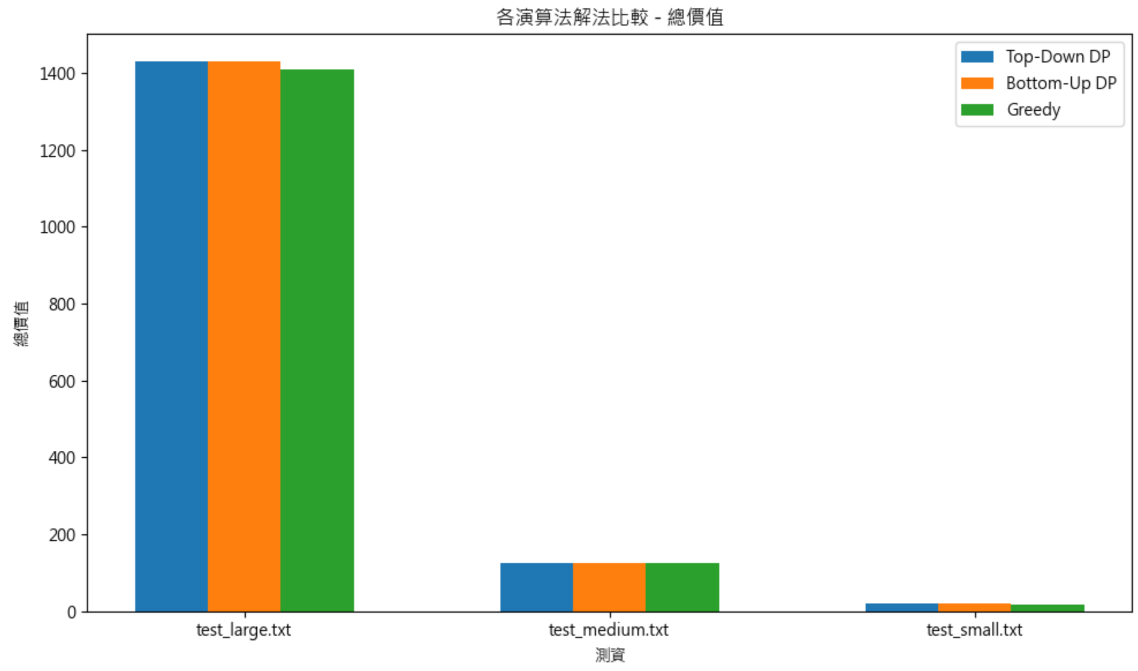

FIGURE 1. TOTAL VALUE ACHIEVED BY EACH ALGORITHM ACROSS DATASETS.

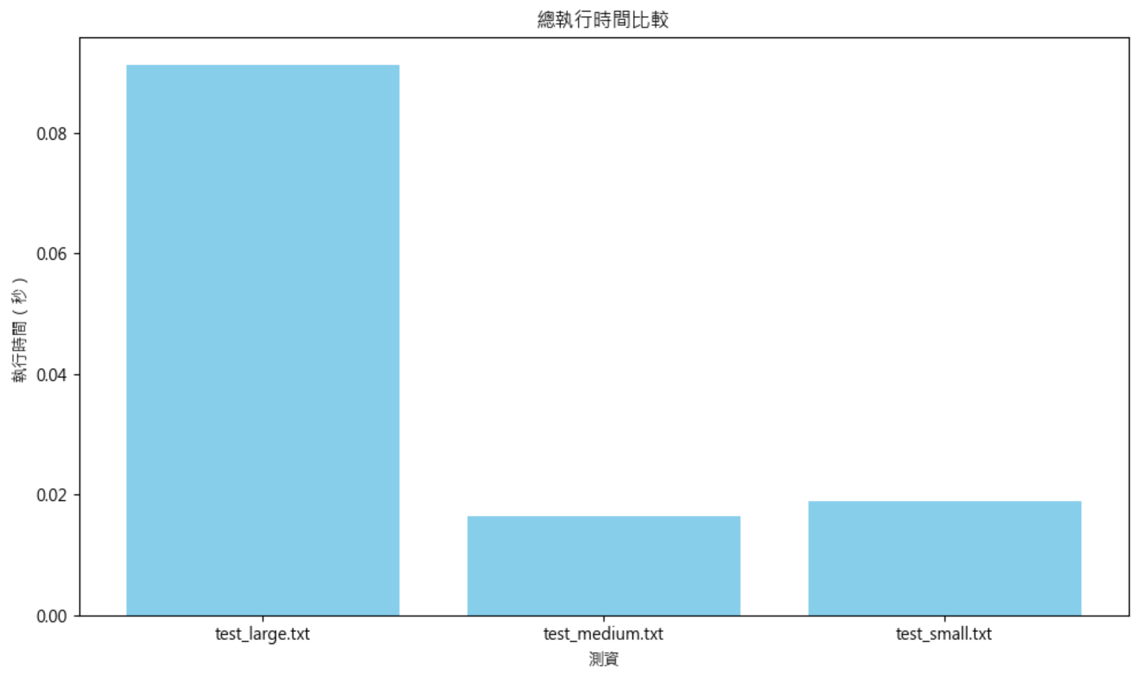

FIGURE 2. EXECUTION TIME COMPARISON ACROSS ALGORITHMS.

4.1 OPTIMALITY OF SOLUTIONS

Across all test cases, both Top-Down and Bottom-Up Dynamic Programming approaches consistently produced optimal solutions. For instance, in the test_large.txt dataset, both methods returned a value of **1430**, while the Greedy algorithm returned **1410**. Similarly, in the test_small.txt case, the greedy method yielded **17** compared to the optimal value of **19**. This confirms that the Greedy algorithm, although faster, may sacrifice solution quality, particularly in larger or more complex item distributions.

4.2 EXECUTION TIME

In terms of execution time, the Greedy algorithm outperformed the two DP methods, especially on larger datasets. For test_large.txt, the greedy method completed execution in a noticeably shorter duration compared to both DP implementations. However, for smaller datasets like test_small.txt, the execution times of all three algorithms were relatively close, making them all viable for small-scale problems.

4.3 VISUAL SUMMARY

The differences in both execution time and result quality are visualized in the figure below. The chart demonstrates that while dynamic programming methods guarantee optimal results, their time complexity is more sensitive to input size. In contrast, the greedy method scales well but may compromise on solution quality.

5 CONCLUSION

This experiment compared three approaches to solving the 0/1 Knapsack problem: Top-Down Dynamic Programming, Bottom-Up Dynamic Programming, and a Greedy heuristic.
Our findings can be summarized as follows:

- **Dynamic Programming** methods (both Top-Down and Bottom-Up) consistently yield optimal solutions. These approaches are ideal when accuracy is essential, though they incur higher time complexity.
- The **Greedy algorithm**, while significantly faster, often produces suboptimal outcomes—particularly when item values and weights vary significantly. Nevertheless, its efficiency makes it suitable for large-scale problems where approximate results are acceptable.
- The **trade-off between accuracy and efficiency** is clearly illustrated: dynamic programming ensures reliable results at the cost of computational time, whereas the greedy method favors speed but compromises on precision.

In real-world scenarios, the choice between these algorithms should be guided by the specific application requirements—whether optimality or performance is the overriding priority.