Report Title: Visualization of the 0/1 Knapsack Problem using Dynamic Programming

Course: Algorithm Design & Analysis

Date: 12 April, 2025

Submitted By

Name: Sumaya Akther Rose ID: 232-115-038

Name: Sadia Sharmin Tumpa ID: 232-115=039

Batch: CSE 59th Sec: A

Submitted To

Nasif Istiak Remon

Senior Lecturer

Metropolitan University, Sylhet

1. Abstract

This report details the implementation of a C++ program designed to solve the classic 0/1 Knapsack problem using a dynamic programming approach. The primary feature of this implementation is the step-by-step visualization of the dynamic programming table's construction. By clearing the console and reprinting the table after each cell calculation, along with a short delay, the program allows users to observe how the optimal solution is built incrementally. This serves as an educational tool to better understand the dynamic programming paradigm applied to this specific problem.

2. Introduction

2.1. The 0/1 Knapsack Problem: The 0/1 Knapsack problem is a fundamental problem in combinatorial optimization. It is defined as follows: Given a set of items, each with a specific weight and value, determine the number of each item to include in a collection (the "knapsack") so that the total weight is less than or equal to a given limit [1] (the knapsack's capacity), and the total value is maximized. The "0/1" constraint signifies that for each item, we can either choose to include it entirely (1) or not include it at all (0); we cannot take fractions of items.

2.2. Significance and Applications: The knapsack problem appears in various real-world scenarios, including resource allocation, cargo loading, investment selection, and cutting stock problems. Its study is crucial in understanding optimization techniques and complexity theory.

2.3. Solution Approaches: While a brute-force approach (checking all possible subsets of items) is possible, its exponential time complexity (O(2^n)) makes it infeasible for larger inputs. Dynamic Programming (DP) provides a more efficient pseudo-polynomial time solution (O(nW), where n is the number of items and W is the capacity).

2.4. Project Goal: The goal of this project is to implement the dynamic programming solution for the 0/1 Knapsack problem in C++ and, more importantly, to provide a clear, step-by-step visualization of how the DP table is filled. This visualization aims to enhance the understanding of the algorithm's mechanics.

3. Methodology: Dynamic Programming Approach

3.1. DP State Definition: We use a 2D DP table (matrix), denoted `dp[i][w]`, where:

- `i` represents considering the first `i` items (from item 1 to item `i`).
- `w` represents the current maximum allowed knapsack capacity.
- `dp[i][w]` stores the maximum value that can be achieved using a subset of the first `i` items with a total weight not exceeding `w`.

The dimensions of the table are `(n+1) x (W+1)`, where `n` is the total number of items and `W` is the maximum knapsack capacity. The extra row and column (index 0) represent the base cases (0 items or 0 capacity).

3.2. Recurrence Relation: The DP table is filled iteratively, usually row by row or column by column. For each cell `dp[i][w]`, we consider the `i`-th item (which has weight `weights[i-1]` and value `values[i-1]` in a 0-indexed array context):

1. **If the `i`-th item's weight (`weights[i-1]`) is greater than the current capacity `w`:** We cannot include the `i`-th item. The maximum value is the same as the maximum value achievable using the first `i-1` items with capacity `w`. `dp[i][w] = dp[i-1][w]`
2. **If the `i`-th item's weight (`weights[i-1]`) is less than or equal to the current capacity `w`:** We have two choices:

- **Exclude the `i`-th item:** The maximum value is `dp[i-1][w]`.
- **Include the `i`-th item:** The maximum value is the value of the `i`-th item (`values[i-1]`) plus the maximum value achievable using the first `i-1` items with the remaining capacity (`w - weights[i-1]`). This is `values[i-1] + dp[i-1][w - weights[i-1]]`. We take the maximum of these two choices: `dp[i][w] = max(dp[i-1][w], values[i-1] + dp[i-1][w - weights[i-1]])`

3.3. Base Cases

- `dp[0][w] = 0` for all `w` (no items, no value).
- `dp[i][0] = 0` for all `i` (no capacity, no value). These are implicitly handled by initializing the DP table with zeros.

3.4. Final Solution The maximum value that can be achieved using all `n` items with the full capacity `W` is found in the bottom-right cell of the table: `dp[n][W]`.

4. Implementation Details

4.1. Programming Language: C++

4.2. Key Libraries Used:

- `<iostream>`: For console input/output.
- `<vector>`: For using dynamic arrays (vectors) to store weights, values, and the DP table.
- `<iomanip>`: For formatting the output (specifically `std::setw` to align columns in the printed DP table).

- `<chrono>` & `<thread>`: To introduce delays (`std::this_thread::sleep_for`) between calculation steps for visualization purposes.
- `<cstdlib>`: For the `system()` call used to clear the console screen.

4.3. Core Functions:

- `clearScreen()`: Clears the console window. It uses preprocessor directives (`#ifdef _WIN32`) to select the appropriate command (`cls` for Windows, `clear` for Linux/macOS).
- `printDPMatrix(const vector<vector<int>>& dp, int n, int W)`: Takes the current DP table, number of items, and capacity as input. It clears the screen and then prints the entire table with appropriate headers and formatting for readability.
- `knapsack(int W, const vector<int>& weights, const vector<int>& values, int n)`: This is the main function implementing the DP algorithm.
    - Initializes the `dp` table of size `(n+1) x (W+1)` with zeros.
    - Iterates through items `i` from 1 to `n`.
    - Iterates through weights `w` from 1 to `W`.
    - Inside the inner loop:
        - It calculates the value for `dp[i][w]` based on the recurrence relation.
        - Crucially, *after* calculating `dp[i][w]`, it calls `printDPMatrix` to display the updated table.
        - It prints a message indicating which cell (`dp[i][w]`) was just computed.
        - It pauses execution for 500 milliseconds using `std::this_thread::sleep_for`.
    - Returns the final result `dp[n][W]`.
- `main()`: Handles user input (maximum weight `W`, number of items `n`, and the weight/value for each item) and calls the

`knapsack` function. Finally, it prints the computed maximum value.

4.4. Visualization Mechanism The visualization is achieved through a cycle within the nested loops of the `knapsack` function:

1. Calculate a single cell `dp[i][w]`.
2. Clear the console screen (`clearScreen()`).
3. Print the entire DP table in its current state (`printDPMatrix()`).
4. Print the status message (e.g., "Computing dp[2][5]").
5. Pause execution briefly (`sleep_for`).

This loop repeats for every cell `dp[i][w]` being computed, creating an animation-like effect where the user can watch the table fill up.

5. How to Compile and Run

1. **Save:** Save the code as a `.cpp` file.
2. **Compile:** Use a C++ compiler.
3. **Run:** Execute the compiled program.
4. **Input:** The program will prompt the user to enter:

   i.The maximum weight capacity of the knapsack (`W`).
   ii.The total number of items (`n`).
   iii.The weight and value for each of the `n` items.

6. Results and Output

During execution, the program repeatedly clears the console and displays the DP table. Each display shows the table with one more cell filled compared to the previous step. Below the table, a message indicates which cell (`dp[i][w]`) was just computed. This allows the user to follow the algorithm's progression cell by cell.

After the loops complete, the final DP table is shown one last time, followed by the final output line:

```
Maximum value in Knapsack = [calculated maximum value]
```

7. Discussion

7.1. Strengths:

- **Educational Value:** The step-by-step visualization is highly effective for understanding how the DP table is constructed and how the optimal solution is derived from subproblems.
- **Correctness:** Implements the standard DP solution for the 0/1 Knapsack problem accurately.
- **Clarity:** The `printDPMatrix` function uses formatting (`setw`) to present the table clearly.

7.2. Limitations:

- **Performance:** The visualization significantly slows down the computation. The `sleep_for(500ms)` and repeated `system(CLEAR)` calls add substantial overhead. This makes the visualization practical only for small values of `n` and `W`. For larger inputs, the visualization would be too slow.
- **Console Dependency:** The visualization relies on console-specific features (clearing the screen). The `system(CLEAR)` call is somewhat crude and might cause flickering. A graphical user

interface (GUI) would provide a smoother visualization experience.

- **Platform Dependence:** The `clearScreen` function uses conditional compilation, but relies on specific system commands (`cls`, `clear`) being available.

7.3. Complexity:

i. Time Complexity: The underlying DP algorithm has a time complexity of O(nW), as it fills an n x W table, and each cell takes constant time to compute. The visualization adds a constant but significant overhead *per cell*, so the *wall-clock time* is much higher, but the *asymptotic time complexity* remains O(nW).

ii. Space Complexity: The space complexity is O(nW) due to the storage required for the DP table `dp[n+1][W+1]`. (Note: Space optimization to O(W) is possible for the standard knapsack problem if only the final value is needed, but this would hinder the full table visualization).

8. Conclusion

This project successfully implements the dynamic programming solution for the 0/1 Knapsack problem with an integrated step-by-step visualization. The program serves as a valuable educational tool, clearly demonstrating the process of building the DP table. While the visualization method limits the practical input size due to performance overhead, it effectively achieves the goal of enhancing understanding of this classic algorithm. Future improvements could involve migrating the visualization to a GUI framework for a smoother experience and potentially visualizing the backtracking process to find the actual items included in the optimal solution.