Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Lab Report

on

"Solving Knapsack problem using different algorithm design strategies"

[Code No. : COMP 314]

Submitted by:

Prafful Raj Thapa (59)

Submitted to:

Dr. Rajni Chulyadyo

Department of Computer Science and Engineering

January 24, 2025

Table of Contents

Chapter 1 : Knapsack Problem

The Knapsack Problem is an optimization problem where you must choose a subset of items with given weights and values to maximize the total value, while ensuring that the total weight doesn't exceed a given capacity.

Key Types:

1. **0/1 Knapsack:** Each item can either be included or excluded.
2. **Fractional Knapsack:** Items can be taken in fractional amounts.

Common Approaches:

- **Brute Force:** Check all subsets (inefficient).
- **Dynamic Programming (DP):** Efficient exact solution with O(nW)O(nW)O(nW) time complexity.
- **Greedy Algorithm:** For Fractional Knapsack, take items in order of value-to-weight ratio (faster, but only for fractional items).
- **Backtracking & Branch and Bound:** Used for more efficient search by pruning infeasible solutions.

Applications:

- Resource allocation, cargo loading, investment decisions, cryptography, etc.

The 0/1 Knapsack Problem is NP-complete, meaning it's computationally hard to solve for large inputs.

Chapter 2 : Brute Force Method

2.1 Brute Force

Brute Force is a problem-solving technique where all possible solutions or combinations are exhaustively checked until the correct one is found. It's a simple approach but often inefficient for large datasets.

Steps of a Brute Force Algorithm

1. **Define the Problem**: Identify the input and the desired output.
2. **Generate All Possible Solutions**: List all potential solutions or combinations to be tested.
3. **Evaluate Each Solution**: Check each solution to see if it satisfies the problem's requirements.
4. **Return the Solution**: Once the correct solution is found, return it. If no solution is found after testing all possibilities, return an indication that the solution doesn't exist.

2.2 Algorithm of Brute-force in Fractional Knapsack Problem

Brute Force Algorithm for Fractional Knapsack Problem (Steps)

1. **Input:**

- A list of n items, each with a weight and a value.
- A knapsack with a weight capacity W.

2. **Generate all possible fractional combinations**
   - For each item, consider all possible fractions from 0 to 1 (e.g., 0%, 10%, 20%, ..., 100%).

3. **Evaluate each combination:**
   - For each combination of fractions, calculate:
     - The total weight of the selected items.
     - The total value of the selected items.

4. **Check if the combination is valid:**
   - If the total weight of the selected items is less than or equal to the knapsack's capacity W, proceed to the next step.

5. **Update the maximum value:**
   - Track the highest value found among all valid combinations (those that don't exceed the weight limit).

6. **Return the maximum value:**
   - The maximum value represents the best selection of items that can be taken without exceeding the weight limit.

2.3 Algorithm of Brute-force in 0/1 Knapsack Problem

1. **Input**:
   - A list of `n` items, each with a weight and a value.
   - A knapsack with a weight capacity `W`.
2. **Generate all possible subsets of items**:
   - Each item can either be **included** or **excluded** from the knapsack.
   - This results in 2n2^n2n possible subsets (where `n` is the number of items).
3. **Evaluate each subset**:
   - For each subset, calculate:
     - The total weight of the items in the subset.
     - The total value of the items in the subset.
4. **Check if the subset is valid**:
   - If the total weight of the subset is less than or equal to the knapsack's capacity `W`, proceed to the next step.
5. **Update the maximum value**:
   - Track the highest value found among all valid subsets (those that don't exceed the weight limit).
6. **Return the maximum value**:
   - The maximum value represents the best selection of items that can be taken without exceeding the weight limit.

2.4 Analysis Result

The result of the **0/1 Knapsack** brute-force solution will provide the exact optimal value that can be achieved within the given weight capacity. Since the algorithm explores all possible combinations of items, it guarantees the best selection of items, ensuring the maximum possible value without exceeding the knapsack's weight limit. However, due to the exponential time complexity, this approach becomes impractical for large numbers of items, and the solution may take a significant amount of time for datasets where the number of items exceeds 20 or so.

In contrast, the **Fractional Knapsack** brute-force approach will yield an approximate solution by considering discrete fractions of items, stepping in small increments (e.g., 0.01). While this approach runs much faster than the 0/1 Knapsack brute-force method due to its linear time complexity, the result may not be perfectly optimal because the solution is based on a finite set of fractional choices. The value obtained will be close to the true optimal solution but may slightly deviate due to the limitations of discrete approximation. This method is more practical for larger inputs but sacrifices the precision of the exact fractional solution.

Chapter 3 : Greedy Method

3.1 Greedy Method

The Greedy Method is a problem-solving approach that makes a series of choices, each of which looks best at the moment, with the aim of finding an optimal global solution. It selects the most promising option at each step, based on a specific criterion, and does not reconsider previous decisions. While simple and efficient, the greedy method does not always guarantee the best solution for every problem.

Steps of a Greedy Algorithm

1. **Define the Problem**: Identify the input and determine what needs to be optimized (e.g., maximizing value or minimizing cost).
2. **Select the Best Option**: At each step, choose the solution that appears to offer the best immediate benefit, typically based on a heuristic or value-to-weight ratio.
3. **Update the State**: Modify the problem state based on the selected choice and move forward.
4. **Repeat**: Continue the process until a solution is reached or no further choices can be made.
5. **Return the Solution**: After completing the process, return the final solution, which is assumed to be optimal, but may not always be the best for all types of problems.

3.2 Algorithm of Greedy Method in Fractional Knapsack Problem

1. **Input**:
    - List of n items, where each item has a weight and a value.
    - A knapsack with a weight capacity W.
2. **Calculate value-to-weight ratio**:
    - For each item, calculate the **value-to-weight ratio**: Value-to-weight ratio={value of item} / {weight of item}
    - Store these ratios along with their corresponding items.
3. **Sort items based on value-to-weight ratio**:
    - Sort the items in **descending order** based on their value-to-weight ratios, so that the item with the highest ratio is considered first.
4. **Select items for the knapsack**:
    - Start with an empty knapsack.
    - Iterate through the sorted items:
        - If the current item can be fully added to the knapsack without exceeding the weight capacity, add it entirely.

    - If the current item cannot be fully added, take the maximum possible fraction of the item (i.e., fill the knapsack with the remaining capacity).
- Repeat this process until the knapsack is full or all items have been considered.

5. **Return the maximum value**:
    - The algorithm finishes when the knapsack is filled, or no further items can be added. Return the total value accumulated in the knapsack.

3.3 Analysis Result

The result of the **Greedy Fractional Knapsack** solution will provide an optimal value within the given weight capacity by selecting items based on their value-to-weight ratio. The algorithm efficiently chooses the items with the highest value per unit of weight first, ensuring that the most valuable items are included in the knapsack until it reaches capacity. This approach guarantees an optimal solution for the **Fractional Knapsack Problem**, as it leverages the greedy choice property, where local optimal decisions lead to a globally optimal outcome. The time complexity is O(nlogn)O(n \log n) due to the sorting step, making it suitable for larger datasets compared to exhaustive methods.

However, the **Greedy Algorithm** assumes that items can be taken in fractional quantities, which may not be ideal in scenarios where only whole items are allowed. In cases where fractional items are not possible or the problem requires exact whole-item solutions, this method may not be applicable. While the greedy approach provides an optimal solution for the fractional version, it may not be suitable for 0/1 knapsack problems, where the items must be fully included or excluded. Despite this, the algorithm's efficiency and the fact that it guarantees optimal results for the fractional knapsack problem make it a practical and widely used solution.

Chapter 4 : Dynamic Programming Method

4.1 Dynamic Programming

Dynamic Programming (DP) is a problem-solving technique that breaks down a complex problem into smaller, simpler subproblems and solves each subproblem only once, storing the results for reuse. This approach is particularly effective for optimization problems that exhibit overlapping subproblems and optimal substructure. By solving each subproblem only once and saving its result, dynamic programming avoids redundant calculations, making it more efficient than brute-force methods for many problems.

Dynamic Programming ensures an optimal solution by solving smaller subproblems in a bottom-up or top-down manner, combining these solutions to solve larger subproblems. The key is to avoid recalculating solutions for the same subproblem multiple times, which improves both time and space efficiency.

Steps of a Dynamic Programming Algorithm

1. **Define the Problem**: Break the problem into smaller subproblems. Identify what needs to be optimized, such as maximizing profit or minimizing cost.
2. **Characterize the Structure of an Optimal Solution**: Understand how the optimal solution can be constructed from solutions to smaller subproblems.
3. **Define the State and Recurrence Relation**: Set up a state representation (typically in a table or array) that holds the solution to each subproblem. Establish a recurrence relation that describes how to compute the solution for each subproblem based on smaller subproblems.
4. **Compute the Solution to the Subproblems**: Solve each subproblem by filling in the table or array, typically starting from the base cases and working upwards (bottom-up) or using recursion with memoization (top-down).
5. **Reconstruct the Solution**: Once the table is filled, use the information in the table to construct the solution to the original problem.
6. **Return the Final Solution**: After filling in the table and constructing the solution, return the final optimized result.

4.2 Algorithm of Dynamic Programming in 0/1 Knapsack Problem

Input:

- A list of n items, where each item has a weight w[i] and a value v[i].
- A knapsack with a weight capacity W.

Steps:

1. **Define the DP table:**
    - Create a 2D table dp[i][w], where i represents the number of items considered, and w represents the current weight capacity. Each cell dp[i][w] will store the maximum value achievable using the first i items and a knapsack capacity of w.
2. **Initialize base cases:**
    - If there are no items or if the knapsack capacity is zero, the maximum value is zero:
        - dp [0] [w] = 0 for all w
        - dp [i] [0] = 0 for all i
3. **Fill the DP table:**
    - Iterate over each item i and each weight capacity w:
        - If the weight of the current item (w[i-1]) is less than or equal to the current capacity w, we have two choices:
            - Exclude the item: The value is the same as the value when excluding this item:

              dp [i-1] [w].

            - Include the item: The value is the value of the current item plus the maximum value for the remaining capacity:

              v[i-1] + dp[i-1] [w - w[i-1]].

        - Take the maximum of these two options:

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

        - If the current item cannot be included (i.e., its weight exceeds w), simply carry forward the value from the previous row:
            a. dp[i][w]=dp[i−1][w]
4. **Return the maximum value:**
    - The maximum value achievable with the given capacity W and all n items will be stored in dp[n][W].

4.3 Analysis Result

The result of the Dynamic Programming (DP) solution for the **0/1 Knapsack Problem** will provide the exact optimal value that can be achieved within the given weight capacity. By systematically considering each item and each possible capacity, the DP algorithm ensures that the best selection of items is made. It evaluates both including and excluding each item to maximize the total value while respecting the knapsack's weight limit. The time complexity of this approach is **O(n × W)**, where `n` is the number of items and `W` is the knapsack capacity. This makes it an efficient solution for moderate-sized problems, where `n` and `W` are not too large. The DP approach guarantees an optimal solution, as it explores all possibilities in a methodical manner.

However, despite its optimality, the Dynamic Programming approach comes with significant computational costs, especially when the number of items (`n`) and the knapsack capacity (`W`) are large. The **O(n × W)** time and space complexity can become prohibitive for larger datasets, making the algorithm less practical in cases where the problem size grows exponentially. Additionally, this solution requires **O(n × W)** space, which can be a concern for memory constraints in large-scale problems. While this DP solution is guaranteed to find the optimal result, its inefficiency for very large inputs is a trade-off that should be considered, especially when compared to other approaches like greedy algorithms or approximations that may provide near-optimal solutions with reduced computational overhead.