Different Types of Knapsack Problems And Their Comparative Analysis

Algorithm Analysis and Data Structures

CSCI-7432

Team Members

Name: Kerem Dogan

Eagle ID: 901438964

Email: kd20511@georgiasouthern.edu

Name: Noushin Gauhar

Eagle ID: 901410562

Email: ng06503@georgiasouthern.edu

Name: Rushmila Shabneen

Eagle ID: 901380789

Email: rs24222@georgiasouthern.edu

Table of Contents

Abstract

For our project, we have delved into the world of combinatorial problems. As a combinatorial problem we selected knapsack problems. We have studied the different types of knapsack problems. We have explored and studied the different types of algorithms that have been used to solve different knapsack problems. For our project, we have explored 0/1 knapsack problem, fractional knapsack problem and bounded knapsack problem. The project explores brute force, greedy method and dynamic programming for each variant. We have provided pseudocodes for each algorithm for each variant of the knapsack problem. We have also done a time and space complexity analysis. Later in the paper, we did a comparative analysis on which algorithm performed better for which variant.

Introduction

The Knapsack problem is an example of the combinatorial optimization problem. This problem is also commonly known as the "Rucksack Problem". The name of the problem is defined from the maximization problem as mentioned: Given a bag with maximum weight capacity of W and a set of items, each having a weight and a value associated with it. Decide the number of each item to take in a collection such that the total weight is less than the capacity and the total value is maximized. The knapsack problem can be classified into the following types **[3]**:

1. 0/1 Knapsack Problem

Given a set of n items, where each item i has a weight wi and a value vi and knapsack with a maximum capacity W, determine the maximum total value that can be achieved by selecting a subset of items, ensuring the total weight does not exceed W.

Let, $xi \in \{0, 1\}$: A binary variable indicating whether item i is selected ($x_i = 1$) or not ($x_i = 0$). The problem can be expressed as:

Maximize $\sum_{i=1}^{n} vi \cdot xi$ Subject to, $\sum_{i=1}^{n} wi \cdot xi \leq W, xi \in \{0, 1\}, \forall i \in \{1, 2, \ldots, n\}$

2. Fractional Knapsack Problem

Given the weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack. Mathematically the problem can be expressed as:

Maximum value = $\sum_{i=1}^{N} v_i x_i$; x_i: fraction of item i to include in the knapsack, where $0 \leq x_i \leq 1$

weight constraint: $\sum_{i=1}^{N} w_i x_i \leq W$

3. Bounded Knapsack Problem

Given N items, each item having a given weight wi and a value vi, the task is to maximize the value by selecting a maximum of K items adding up to a maximum weight W. Mathematically, it can be expressed:

$$\text{Maximum value} = \sum_{i=1}^{N} v_i x_i \text{ subject to } \sum_{i=1}^{N} w_i x_i \leq W \text{ and } x_i \in \{0, 1, \ldots, K\}$$

4. Unbounded Knapsack Problem

Given a knapsack weight W and a set of N items with certain value vi and weight wi, we need to calculate the maximum amount that could make up this quantity exactly. This is different from the 0/1 Knapsack problem, here we can use an unlimited number of instances of an item.

Background

A classic optimization issue in operations research and computer science is the 0/1 Knapsack problem. It falls within the area of combinatorial optimization problems and finds extensive use in financial modeling, decision-making, and resource allocation. Despite its NP-completeness, a number of algorithms that behave admirably in the typical case have been presented. This part of the paper provides an introduction to the topic, a list of problems that are directly connected to it, and a proof that the problem belongs to the NP-complete class. Combinatorial optimization problems are a broad category of problems that includes the 0–1 Knapsack Problem. We attempt to "maximize" (or "minimize") a certain "quantity" while adhering to certain limitations in these kinds of situations. The Knapsack problem, for instance, is to maximize the profit made while staying within the knapsack's capacity **[1]**.

Diverging from the discrete nature of the 0/1 Knapsack, the Fractional Knapsack introduces a continuous dimension to the problem, facilitating more flexible solutions. Fractional Knapsack finds applications in contexts where items can be utilized in fractional amounts, such as in resource allocation or inventory management **[5]**.

The Bounded Knapsack Problem is a classic dynamic programming problem **[3]**. Departing from the assumption of an unlimited supply of each item, the Bounded Knapsack Problem introduces constraints on the quantity of each item available. This variant reflects scenarios where resources are limited and must be carefully managed, making it applicable in contexts such as production planning or budget allocation **[4]**.

Algorithms

For our project, we have decided to use the brute force method, greedy method and dynamic programming for each of the knapsack variant problems. The following section discusses the algorithm, pseudocode and time space complexity for each method.

0/1 Knapsack Problem

Brute Force

By thoroughly listing every potential solution and assessing each one to determine the best outcome, the brute force approach is a simple technique for problem-solving. It is a straightforward but computationally costly method that is frequently employed when there is no better solution or when the problem is minor. The brute force method for the 0/1 Knapsack Problem is creating every conceivable subset of items and figuring out the total weight and value of each subset. The ideal answer is the subset with the highest value that stays within the knapsack's capacity. There will be a second set of possible item combinations for the knapsack if there are n options. Either two items are selected or not. A bit string of length n made up of 0s and 1s is produced. The *i*th item is not selected if the bit string's *i*th symbol is 0, and it is selected if it is 1 **[2]**. There are some steps that we should follow while using brute force approach for the 0/1 Knapsack Problem.

1. **Generate all subsets of items**: Each subset corresponds to a potential selection of items, represented as binary decisions ($xi = 0$ or $xi = 1$). For n items, there are 2^n possible subsets.
2. **Evaluate each subset**: Calculate the total weight and value of the items in the subset. Check if the total weight exceeds the knapsack's capacity W. If it does, discard the subset.
3. **Keep track of the best solution**: If the subset's total weight is within the capacity and its value is greater than the current maximum value, update the optimal solution.
4. **Return the best solution**: Output the subset with the highest value that satisfies the weight constraint.

While solving the problem, we can follow this algorithm:

- Input: An array of weights (*w*), values (*v*), number of items (*n*), and knapsack capacity (*W*).
- Initialize maxValue = 0 and bestSubset = [].
- Iterate over all possible subsets (using recursion or a bitmask):
    - For each subset, calculate the total weight and value.
    - If the total weight is $\leq W$ and the value is greater than maxValue, update maxValue and bestSubset.
- Output maxValue and bestSubset.

Runtime Analysis

1. **Subset Generation**:
    - 2^n subsets need to be generated.
2. **Subset Evaluation**:
    - For each subset, the algorithm checks all n items to compute the total weight and value. O(n)
3. **Overall Time Complexity**: $O(2^n \cdot n)$
    - The exponential growth comes from the 2^n subsets, while the linear factor n is for processing each subset.

Space Complexity

1. **Input Storage**:
    - Array of items: O(n).
2. **Auxiliary Variables**:
    - Best subset array: O(n).
    - Current subset array: O(n).
3. **Total Space Complexity**: O(n)
    - The space complexity is linear since no additional structures grow with the size of the subset space.

Greedy Method

The greedy algorithm is based on selecting items in a way that seems locally optimal at each step. For the knapsack problem, this involves sorting items based on a specific criterion, such as:

- **Value**: Items with higher value are selected first.
- **Weight**: Items with lower weight are prioritized.
- **Value-to-Weight Ratio**: Items with a higher value-to-weight ratio (*vi* / *wi*) are selected first.

There are some possible greedy strategies to the 0/1 Knapsack problem **[2]**:

1. To raise the knapsack's value as soon as possible, select the item with the highest value among the others.
2. To fit more items in the knapsack, pick the lightest item from the remaining items that takes up capacity as slowly as feasible.
3. Pick the products that have the highest value relative to their weight.

While applying the greedy algorithm, we can follow these steps:
1. **Calculate Ratios**: Compute the value-to-weight ratio (*vi* / *wi*) for each item.
2. **Sort Items**: Sort the items in descending order of the chosen criterion (e.g., value-to-weight ratio).
3. **Select Items**:
    - Iterate through the sorted items.
    - Include an item if it fits within the remaining capacity of the knapsack.
    - Stop when no more items can fit.

This pseudo-code can be used while creating the algorithm:
```
function GreedyKnapsack(values, weights, capacity):
        n = length(values)
        items = [(values[i], weights[i], values[i] / weights[i]) for i in range(n)]
        // Sort items by value-to-weight ratio in descending order
        items.sort(by=lambda x: x[2], descending=True)

        totalValue = 0
        totalWeight = 0
```

```
for (value, weight, ratio) in items:
if totalWeight + weight <= capacity:
totalWeight += weight
totalValue += value
// Cannot add more items if weight exceeds capacity
return totalValue
```

Runtime Analysis

1. **Sorting**:
    - Sorting the items by value-to-weight ratio takes O(n logn).
2. **Greedy Selection**:
    - Iterating through the sorted list of items takes O(n).
3. **Overall Time Complexity**: O(n logn)

Space Complexity

1. **Input Storage**:
    - Array of items: O(n).
2. **Total Space Complexity**: O(n)

Dynamic Programming

The process of solving problems whose solutions meet recurrence relations with overlapping subproblems is known as dynamic programming. Instead of repeatedly solving overlapping subproblems, dynamic programming solves each smaller subproblem only once and logs the outcomes in a table. The original problem's answer is then obtained using the table **[1]**.
The following concept serves as the foundation for the first dynamic programming strategy we will discuss here. For example, *xn* may have a value of 0 or 1. The best profit that can be made from the remaining *n-1* objects with knapsack capacity *W* is what happens if *xn* = 0. The profit *pn* (already chosen) plus the best profit made by the remaining *n-1* objects with knapsack capacity *W* - *wn* (in this example, *wn* ≤ *W*) is the maximum profit that could be made if *xn* = 1. Consequently, the maximum of these two "best" profits will be the ideal profit.
The following recurrence relation results from the aforementioned concept, where P(i, m), *i* = 1, ..., n, *m* = 0, ..., W, is the maximum profit that can be made from the objects 1, …, i with knapsack capacity m:

$$\boxed{P(i,m) = \begin{cases} P(i-1,m) & ,w_i > m \\ max\left\{\begin{smallmatrix} P(i-1,m) \\ P(i-1,m-w_i)+p_i \end{smallmatrix}\right\} & ,w_i \leq m \end{cases}}$$

with initial conditions

$$\boxed{P(1,m) = \begin{cases} 0 & , w_1 > m \\ p_1 & , w_1 \le m \end{cases}}$$

So, the optimal profit is P(n, W) **[2]**.
While solving the problem, the algorithm we have created is working like this:

1. **Input**: The user specifies the number of items, their weights and values, and the knapsack's capacity.
2. **Dynamic Programming Table**: Create a (n + 1) × (*capacity* + 1) table *dp* where *dp[i][w]* represents the maximum value achievable with the first *i* items and a capacity of *w*.
3. **Recurrence Relation**: If the weight of the current item *i* is less than or equal to *w*:
   - *dp[i][w] = max(dp[i − 1] [w], dp[i − 1] [w − weight[i − 1]] + value[i − 1])*

   Otherwise:
   - *dp[i][w] = dp[i − 1] [w]*
4. **Reconstruction**: Trace back through the *dp* table to determine which items were included in the optimal subset.

Runtime Analysis

1. **Table Filling**:
   - The algorithm iterates over nn items and W capacities.
   - Time Complexity: O(n ⋅ W).
2. **Reconstruction**:
   - The reconstruction process traces back through n rows.
   - Time Complexity: O(n).
3. **Total Time Complexity**: O(n ⋅ W)

Space Complexity

1. **Dynamic Programming Table:**
   - A (n + 1) × (capacity + 1) table is used.
   - Space Complexity: O(n ⋅ W).
2. **Auxiliary Variables:**
   - Constant space for variables like *w*, *i*, and loop counters.
3. **Total Space Complexity:** O(n ⋅ W)

Fractional Knapsack Problem

For the Fractional Knapsack Problem, we have also explored the brute force method, greedy method and dynamic programming. Each of the methods for this problem is discussed below.

Brute Force

The Brute force pseudocode for the fractional knapsack problem is given below-

Pseudocode

```
fractional_knapsack_bf(n, weights, values, capacity, fraction_levels=10):
1.    best_value = 0
2.    Initialize an empty list best_combination
3.    Initialize an empty list fraction_range
4.    for i = 0 to fraction_levels
5.        Add i / fraction_levels to fraction_range
6.    for each combination in CartesianProduct(fraction_range, repeat=n):
7.        Initialize total_weight = 0
8.        Initialize total_value = 0
9.        for i = 0 to n - 1
10.           total_weight = total_weight + (combination[i] * weights[i])
11.           total_value = total_value + (combination[i] * values[i])
12.       if total_weight <= capacity AND total_value > best_value
13.           best_value = total_value
14.           best_combination = combination
15.   return best_value, best_combination
```

Time Complexity and Space Complexity

Assuming the total length of fraction_range is k. Since the outer loop (line 6) generates Cartesian product of k fractions for n items and compares the value of the combinations to get the best value n times. Therefore, the total time complexity is $O(n.k^n)$

As the total length of fraction_range is k . Hence, the space requirement for the fraction_range list is O(k). On the other hand, the cartesian product does not store all the combinations simultaneously rather generates combinations one at a time so the space required for storing the current combination is O(n) for n number of items, as it holds one fraction value for each item. Total space complexity is O(n+k)

Greedy Method

The Greedy method pseudocode for the fractional knapsack problem is as follows-

Pseudocode

```
fractional_knapsack_greedy(n, w, v, W):
1.    Sort the items according to v[i] / w[i] so that v1/w1 ≥ v2/w2 ≥ ... ≥ vn/wn
2.    Let L[1..n] be a new array to store the fractions of items to take
3.    total_value = 0
4.    load = 0
5.    i = 1
6.    while load < W and i ≤ n:
7.        if w[i] + load <= W
```

```
8.                  L[i] = 1
9.                  total_value += v[i]
10.                 load += w[i]
11.             else
12.                 fraction = (W - load) / w[i]
13.                 L[i] = fraction
14.                 total_value += v[i] * fraction
15.                 load = W
16.             i = i + 1
17.         return total_value
```

Time Complexity and Space Complexity

The first step is to sort the items by v_i/w_i and the time complexity of sorting a list of n items is O(*nlogn*). Then the algorithm iterates through the sorted list of n items. The time complexity is O(n). Therefore, total time complexity is O(*nlogn*)

Sorting the items requires O(n) for the sorted list and the array L requires O(n) space. Total space complexity is O(n)

Dynamic Programming

The dynamic programming pseudocode is giving in the following -

```
fractional_knapsack_dp(n, weights, values, W)
1.      for i 0 to W
2.          dp[i]  = 0
3.      for each item i from 1 to n
4.          for each capacity w from W to 1
5.          if weights[i] <= w
6.              dp[w] = max(dp[w], dp[w - weights[i]] + values[i])
7.          else
8.              Remaining capacity = W - load
9.              Fraction = Remaining capacity / weights[i]
10.             dp[w] = max(dp[w], dp[w - Remaining capacity] + Fraction * values[i])
11.             load = load + Fraction * weights[i]
12.     Return dp[W]
```

Time Complexity and Space Complexity

The time complexity of the outer loop is O(n) and the time complexity of the inner loop is O(n * W). So the total time complexity is O(n * W).

O(W) space for the DP array of size W+1. Since no additional data structures with size dependent on n (the number of items) are used. Hence, the space complexity is O(W).

Bounded Knapsack Problem

The brute force method, greedy method and dynamic programming are again explored for the bounded Knapsack Problem,. Each of the methods for this problem is discussed below-

Brute Force Pseudocode

The pseudocode for the bounded knapsack problem is given below.

```
FUNCTION bounded_knapsack_brute_force(values, weights, quantities, capacity):
        n ← LENGTH(values)            // Number of items
        max_value ← 0                 // Initialize maximum value as 0
        // Step 1: Generate all combinations of quantities
        all_combinations ← CARTESIANPRODUCT of ranges (0 to quantities[i]) for each item i
        // Step 2: Iterate over all combinations
        FOR combination IN all_combinations:
        total_weight ← 0
        total_value ← 0
        // Compute the total weight and value of the current combination
        FOR i FROM 0 TO n - 1:
        total_weight ← total_weight + combination[i] * weights[i]
        total_value ← total_value + combination[i] * values[i]
        // Step 3: Check if the weight is within capacity
        IF total_weight ≤ capacity:
        max_value ← MAX(max_value, total_value)
        // Step 4: Return the maximum value
        RETURN max_value
```

Time Complexity and Space Complexity

For n items, where each item's quantity can range from 0 to quantities[i], the total number of combinations is: $\prod_{i=1}^{n}(quantities[i] + 1)$. In the worst case (all items have q_{max} maximum quantity), the number of combinations is:$O(q_{max}^{n})$. For each combination, the algorithm computes total_weight and total_value by iterating over the n items. This requires *O(n)* operations per combination. The total time complexity is $O(q_{max}^{n}. n)$. The algorithm uses space complexity of *O*(*n*) space for variables.

Greedy Algorithm Pseudocode

The pseudocode for the bounded knapsack problem is given below.

```
FUNCTION greedy(values, weights, quantities, capacity):
        n ← LENGTH(values)            // Number of items
```

```
    total_value ← 0          // Initialize the total value to 0
    // Step 1: Calculate value-to-weight ratios and prepare items list
    items ← EMPTY LIST
    FOR i FROM 0 TO n - 1:
        ratio ← values[i] / weights[i]
        APPEND (ratio, values[i], weights[i], quantities[i]) TO items
    // Step 2: Sort items by value-to-weight ratio in descending order
    SORT items BY ratio IN DESCENDING ORDER
    // Step 3: Select items greedily
    FOR EACH (ratio, value, weight, quantity) IN items:
        IF capacity == 0:
            BREAK
        // Step 3.1: Determine the maximum number of items we can take
        max_items ← MIN(quantity, FLOOR(capacity / weight))
        // Step 3.2: Update the total value and capacity
        total_value ← total_value + (max_items * value)
        capacity ← capacity - (max_items * weight)
    // Step 4: Return the total value
    RETURN total_value
```

Time Complexity and Space Complexity

Computing the value-to-weight ratio requires iterating over the n items, which takes $O(n)$ time. Sorting the n items by their ratios in descending order takes: $O(n\log n)$ time. The algorithm iterates over the sorted list of n items. Thus, iterating over n items takes: $O(n)$. The total time complexity is the sum of the steps required $O(n) + O(n\log n) + O(n)$. So the total Time Complexity is $O(n\log n)$. The algorithm creates a list of tuples, where each tuple contains: ratio, value, weight, and quantity for each item. This requires $O(n)$ space. Variables require constant space $O(1)$. The total Space Complexity: $O(n)$.

Dynamic Programming Pseudocode

The pseudocode and its time and space complexity are discussed below.

```
FUNCTION bounded_knapsack_dp(values, weights, quantities, capacity):
    n ← LENGTH(values)  // Number of items
    // Step 1: Initialize the DP table
    dp ← 2D array of size (n+1) × (capacity+1), filled with 0
    // Step 2: Fill the DP table
    FOR i FROM 1 TO n:            // Iterate over items
        FOR w FROM 0 TO capacity:  // Iterate over all capacities
            // Option 1: Don't take the current item
            dp[i][w] ← dp[i-1][w]
            // Option 2: Consider taking the current item (up to its quantity limit)
            max_quantity ← quantities[i-1]
            FOR qty FROM 1 TO max_quantity:
                total_weight ← qty × weights[i-1]
                total_value ← qty × values[i-1]
                IF w ≥ total_weight:  // If capacity permits this quantity
                dp[i][w] ← MAX(dp[i][w], dp[i-1][w - total_weight] + total_value)
        // Step 3: Return the maximum value for all items and full capacity
        RETURN dp[n][capacity]
```

Time Complexity and Space Complexity

The outer loop runs $O(n)$ as it iterates over n items. It iterates over capacities which are O(capacity). The innermost loop runs $O(q\text{max})$. For each combination of item (n) and capacity (capacity), we perform $O(q\text{max})$ iterations in the innermost loop. So the total Time Complexity is $O(n \cdot \text{capacity} \cdot q\text{max})$. A 2D DP table of size (n+1) × (capacity+1) is used. Space required for the DP table is O(n · capacity). The total space complexity is O(n · capacity).

Experiments

Fractional Knapsack Problem

The experimental results of the fractional factorial knapsack problem are shown below-

	Input size = 5		Input size = 10		Input size = 50	
	Output	Time	Output	Time	Output	Time
BruteForce	96	243.80 ms	–	2hrs>	–	4hrs>
Greedy	96	0.14 ms	89.58	0.12 ms	1230.77	0.16 ms
DP	96	0.89 ms	89.58	0.40 ms	1215.8	3.73 ms

The brute force algorithm for solving the fractional knapsack problem is slower for input sizes like n=10 and n=50 than the other algorithms due to the exponential growth in the number of possible combinations of item fractions. For n=10, the total number of combinations is 10^{10}, and for n=50, it grows to 10^{50}, resulting in an extremely large number of combinations to evaluate. Furthermore, each combination requires iterating over the items to calculate the total weight and value even when the total weight exceeds the Knapsack weight capacity which includes more computational cost. Dynamic Programming may not provide optimal solutions for fractional knapsack problems because the DP approach relies on integer decisions.

0/1 Knapsack Problem

The experimental results of the bounded knapsack problem are presented in the table below.

	Input size = 5		Input size = 10		Input size = 50	
	Output	Time	Output	Time	Output	Time
BruteForce	44	22.00 ms	127	20.10 ms	–	4hrs>
Greedy	44	1.41 ms	126	1.71 ms	1206	2.31 ms
DP	44	18.87 ms	127	23.04 ms	1211	20.71 ms

We can infer from the table that brute force is inappropriate for this problem. Even at moderate sizes, the number of viable options grows exponentially with increasing input and becomes infeasible. Even when many of the combinations are invalid because they exceed the knapsack's capacity, it nonetheless evaluates them all. Dynamic programming produces more accurate results since it looks for all potential optimal solutions, but the greedy approach may perform faster as input size increases because it stops searching after reaching local optimum.

Bounded Knapsack Problem

The experimental results of the bounded knapsack problem are presented in the table below.

	Input size = 5		Input size = 10		Input size = 50	
	Output	Time	Output	Time	Output	Time
BruteForce	144	5.55 ms	149	3 min 3 sec	–	5hr>
Greedy	144	0.08 ms	149	0.10 ms	2866	0.21 ms
DP	144	0.21 ms	149	1.23 ms	2879	28.64 ms

From the table, we can conclude that brute force is not suitable for this problem. As input increases, the possible combinations grow exponentially and become infeasible even for moderate sizes. It evaluates all combinations, even when many of them exceed the knapsack's capacity and are invalid. As input size increases, the greedy method may perform faster as it stops searching after achieving local optimum but dynamic programming gives more accurate results as it searches for all possible optimal solutions.

Conclusions

While the brute force approach guarantees the optimal solution, its exponential time complexity makes it impractical for larger input sizes. Therefore, brute force approach is not desirable in cases of knapsack problems. Dynamic programming has proven to provide optimal solutions for the knapsack problems except fractional knapsack problems. The greedy method is the fast one among all the algorithms analyzed in this project but does not always guarantee an optimal solution. Therefore, when we have larger inputs and we need to be fast, we can opt for greedy methods. When we want to acquire the optimal solution, dynamic programming is the best solution.

References

[1] Lagoudakis, M. G. (1996). The 0–1 knapsack problem: An introductory survey. *The Center for Advanced Computer Studies, University of Southwestern Louisiana.*

[2] Hristakeva, M., & Shrestha, D. (2005). Different approaches to solve the 0/1 knapsack problem. *Computer Science Department, Simpson College, Indianola, IA.*

[3] https://medium.com/@florian_algo/unveiling-the-bounded-knapsack-problem-737d71c4146b

[4] Angad and Leena Jain, A Study on Different Techniques to Solve Knapsack Problem, International Journal of Research in Mathematics and Computation (IJRMC), 10(1), 2024, pp. 7-14.

[5] https://iaeme.com/Home/article_id/IJRMC_10_01_002