Exploring the Knapsack Problem

Kiera Conway
Dakota State University
Seattle, Washington
Kiera.Conway@trojans.dsu.edu

***Abstract*—The Knapsack Problem (KP) has been a subject of fascination since its appearance in 1895, when Mathews' first laid the groundwork for combinatorial optimization problems like the KP. The primary goal of the KP focuses on optimizing resource allocation by selecting the most valuable combination of items to fit within a knapsack's weight capacity. Over time, the KP has not only inspired the exploration of related problems like the Traveling Salesman Problem and the Graph Coloring Problem, but it has also exploded into a vast field of KP variants that address the practical complexities of real world uses. Notable problem variations include the 0/1 KP, Bounded Knapsack Problem, and the Unbounded Knapsack Problem. This report focuses on two prominent solution variations for the 1/0 KP: the Extended Greedy Algorithm and Dynamic Programming Algorithm. As the KP remains a compelling and universal problem, researchers and mathematicians continue to delve into new problem variations and discover additional solution variants; the significance and impact of the Knapsack Problem continues to command attention and drive innovation.**

Keywords—Knapsack Problem, Resource Allocation, Unbounded, Bounded, Greedy Algorithm, Dynamic Programming, 0/1, Combinatorial Optimization

I. Introduction

Imagine a daring thief who has found himself amidst a legendary treasure trove, equipped with just a worn knapsack on his back. As his eyes dart across the most enticing items in the room, he is sorely reminded of the limitations of his bag. He finds himself forced to make a strategic decision about which treasures to take and which to leave behind as each item contains distinct weights and values. The imposition of his knapsack capacity has placed him in the precise predicament that has challenged some of the greatest minds for generations.

This modified version of the classic Thief example from Bhargava demonstrates an intriguing optimization challenge, known as the Knapsack Problem (KP), that has tested mathematicians, computer scientists, and even fictional thieves throughout history [2]. At its core, the KP focuses on optimal resource allocation by framing it into one basic question: How can we maximize the value of the items in the knapsack without exceeding a weight limit?

II. Problem Statement

The classic KP challenge pertains to the efficient packing of items into a knapsack with a limited weight capacity. The problem includes a knapsack with a predetermined weight capacity and a variety of items, each of which is characterized by a corresponding weight and value. It is assumed that the weight capacity of the knapsack is at least as large as the heaviest item, and allows inclusion of as many items as possible without exceeding the knapsack's weight limit. As the challenge's primary focus is on distribution of the items, real world constraints such as dimensions and size are ignored [1].

While there are many variations of the KP, one of the most discussed and researched is known as the 0/1 KP. Unlike some KP variations which allow for inclusion of fractional items to maximize value, the 0/1 KP asserts that each item can be either be entirely included in the knapsack or entirely excluded. It is important to note that while we often allude to items going in a knapsack, mathematically, this problem simplifies down to just a "combination of binary decisions" [6]. In other words, for each item in the knapsack, it is either 'in the bag' (1) or 'not in the bag' (0). Throughout this report, we will primarily focus on the 0/1 KP, henceforth referred to as KP, unless explicitly specified otherwise.

A. Mathematical Representation

In the formal representation of the KP, a combination of a given set of items, characterized by their corresponding weight (w_j) and value (p_j), are selected to place in the knapsack, such that the value is maximized while the weight does not exceed the knapsack's capacity (c). This representation can be expressed as follows:

$$\text{maximize} \sum_{j=1}^{n} (p_j \times x_j)$$

$$\text{subject to} \sum_{j=1}^{n} w_j \times x_j \leq c$$

$$x_j \in \{0,1\} \text{ for } j = 1, \dots, n \qquad (1)$$

where

- p_j represents the profit/value of item j
- w_j corresponds to the weight of item j
- c denotes the knapsack's maximum weight capacity
- x_j is a binary decision variable that takes the value 1 if item j is included in the knapsack and 0 otherwise.

In equation (1), the function $\sum_{j=1}^{n}(p_j \times x_j)$ represents the total value/profit achieved by selecting items with their corresponding profits (p_j) .The constraint $\sum_{j=1}^{n} w_j \times x_j \leq c$ ensures that the total weight of the selected items does not exceed the knapsack's capacity (c) by restricting the combination of items that can be placed in the knapsack. Since this problem pertains to the 1/0 KP variation of the knapsack problem, the binary variable $x_j \in \{0,1\}$ is used to represent the selection ($x_j = 1$) or exclusion ($x_j = 0$) of items. This representation enables the optimization process to find the optimal combination of items that maximizes the total profit while adhering to the weight constraint [6].

B. History

The KP has a long history, dating back centuries to its first appearance in 1895 [6]. Although it lacked an official name at the time, Mathews' work on Partition Theory introduced the

underlying connections to combinatorial optimization problems like the Knapsack problem. Partition Theory's focus on dividing or representing a number as a sum of positive integers introduced the concept of dividing the knapsack's capacity into parts (weights) to maximize the overall value[6][4]. Combinatorial optimization is a branch of mathematics which deals with finding the best solution from a limited selection and a finite set of possible combinations. However, the KP is just one of many problems in this category; other well-known combinatorial optimization problems include the Traveling Salesman Problem, the Vehicle Routing Problem, and the Graph Coloring Problem, to name a few [11]. This optimization concept set the stage for future developments in the KP.

In 1980, the introduction of the 0/1 variant of the KP by Gallo, Hammer, and Simeone further expanded the problem's scope. With this variant came the inclusion of decision variables (x_j), which brought an additional layer of complexity to the KP and opened new avenues for research and exploration [9]. Since then, researchers have been working to improve the algorithm efficiency to various instances of the KP and its variants, resulting in significant advancements that can now solve "nearly all standard instances [of the KP] found in the literature" [7]. These advancements have played an important role in advancing the study and application of the KP.

C. Importance

The KP serves as a powerful tool across the fields of mathematics and computer science, enabling the exploration algorithmic workings and efficiency [1]. Computer scientists can use the KP to help in program partitioning and task allocation, where it is essential to prioritize critical tasks while optimizing computational resources. Furthermore, the KP demonstrates versatility and utility for solving diverse challenges in day-to-day scenarios. The KP can be used to help efficiently pack food for survival situations, with a goal to maximize nutrition while minimizing overall weight or help assist investors in maximizing their profits while adhering to their budget constraints [5]. It is also invaluable to the lumber industry, where the KP contributes to the optimization of log cutting processes by maximizing the value obtained from each log while minimizing waste. In fact, while KP is often considered a 'packing' problem, it also commonly referred to as a 'cutting problem [6].

Whether efficiently packing food for survival, optimizing investment portfolios, or even cutting logs, the versatility of the Knapsack is evident in many real-world applications. However, the real world often presents additional complexities that the basic KP does not account for. Factors including nutritional sustainability, investment risk tolerance, time constraints, market price variations, and other practical factors, require a different approach. As such, there appeared a substantial "need for [an] extension of the basic knapsack model," which led to the development of "various extensions and variations," with each variant offering a solution to tackle a specific scenario [6]. These extensions further enhance the KP's applicability and make it invaluable across various industries and applications.

D. Variants

There have been numerous variations created to tackle the 0/1 KP. Among the earliest and most well-known variants are the Unbounded Knapsack Problem (UNP) and the Bounded Knapsack Problem (BNP), both of which focus on the limitation of item repetition. In the UNP, items can be included in the knapsack an unlimited number of times. This allows greater flexibility in item selection and the potential to achieve a higher total value, but increases the number of possible solutions. Alternatively, the BNP limits the number of times each item can be placed inside the bag. Since each item appears an equal number of times, the primary focus is on the specific combination of the items. As a result, the solutions tend to be simpler and more realistic, but the number of possible solutions is limited [6].

Amidst the variations in the BNP, two popular solution variants for solving the 0/1 KP are the Greedy and Dynamic Programming algorithms. The Greedy approach focuses on locally optimal choices but may disregard the overall consequences of these choices, resulting in efficient but approximate solutions. In contrast, the Dynamic Programming variant breaks the problem into smaller subproblems and systematically calculates optimal solutions, ensuring a more accurate solution but with a potentially less-efficient running time.

1) Greedy Algorithm: The Greedy Algorithm is characterized by its prioritization of immediate choices over evaluating multiple options to find the most optimal result [8]. While often considered naïve and heuristic due to its focus on local gains rather than long-term consequences, it stands out as an intuitive method for solving the KP. This is due to the seemingly obvious approach: it assigns a value-to-weight ratio for each item and fills the knapsack with items having the largest ratio until it reaches as close to the maximum weight as possible [6]. This simplicity makes it more efficient and easier to understand and implement compared to other methods like dynamic programming [8].

However, the Greedy Algorithm has its limitations. It may not always yield the most optimal solution compared to dynamic programming, as it does not consider all possible combinations of items [8]. As such, it should only be used for scenarios where "good enough" solutions are acceptable, such as finding the least amount of coins needed to return change or solving certain variants of the shortest path problem [6][8]. While it may not be "as broadly applicable as other algorithm design approaches [like] dynamic programming", the Greedy Algorithm remains a valuable tool for solving specific instances of the Knapsack Problem efficiently and intuitively [8].

2) Dynamic Programming Algorithm: The Dynamic Programming (DP) emerges as one of the best techniques for solving the challenging 0/1 KP [7]. It involves breaking a larger problem down into smaller subproblems and solving them iteratively. Instead of dealing with an entire problem at once, DP starts with the simplest cases and gradually builds up to the original problem. For the KP, DP operates in two main phases: the forward phase and the backtracking phase.

In the forward phase, DP employs the Bellman Recursion to compute optimal solution values for knapsack subproblems [10][6]. This process involves creating a grid of items and knapsack weights, starting from the smallest weight capacity and progressively moving through each capacity and item, until it reaches the capacity of the original knapsack [2]. By iteratively calculating the optimal solutions for smaller

subproblems, DP efficiently determines the maximum value achievable at each weight capacity.

In the backtracking phase, DP starts from the final cell of the table generated during the forward phase and identifies the items that were included in each knapsack to achieve the maximum value. This step ensures that the algorithm provides not only the maximum value, but also which specific combination of items achieves that value.

The DP approach is known for its efficiency and ability to find the optimal solution for the 0/1 KP, even for large and complex instances of the problem, with a guaranteed worst-case running time [7]. It is considered one of the most effective techniques for solving the KP, providing a reliable and accurate solution that accounts for all possible item combinations.

III. Algorithms

To better understand the previously discussed algorithm variations and how they work, we will analyze the pseudocode of both the Greedy Algorithm and the Dynamic Algorithm. By analyzing the step-by-step procedures of each approach, we can visualize their methods for solving the Knapsack Problem. This analysis will provide valuable insights into the strengths and limitations of each technique, with the goal to highlight the distinct features and performance characteristics of each algorithm, showcasing their significance in the realm of solving the KP.

A. Extended Greedy Algorithm

Before observing the pseudocode of the Greedy Algorithm for KP, it is essential to address a special case which may prevent the standard Greedy Algorithm from returning an optimal solution [6]. Consider following values for our Knapsack instance:

Table I. Knapsack Instance

Number of items		$n = 2$
Knapsack capacity		$c = 2$
Item 1	*weight*	$w_1 = 1$
	profit	$p_1 = 2$
	efficiency	$e_1 = \frac{p_1}{w_1} = \frac{2}{1} = 2$
Item 2	*weight*	$w_2 = 3$
	profit	$p_2 = 3$
	efficiency	$e_2 = \frac{p_2}{w_2} = \frac{3}{3} = 1$

Given the values in Table 1, the Greedy Algorithm selects Item 1 due to its higher efficiency ($e_1 = 2$) compared to Item 2 ($e_2 = 1$). It then places Item 1 into the knapsack, leaving a remaining capacity of 2. As no additional items fit within this space, the function concludes that the knapsack is full. However, the optimal solution in this case would have been to pack Item 2, which has a value of 3, resulting in a higher profit.

To address this limitation, I have implemented Kellerer's "Extended Greedy" algorithm. This enhanced version incorporates an additional step to account for the possibility of selecting a single item with the highest profit value. This is accomplished by comparing the solution value obtained from the standard Greedy Algorithm to the largest individual profit value, of which the larger value is added to the knapsack [6]. This modification will improve the algorithm's performance and ensure a more reliable and accurate approximation for the Knapsack problem.

The pseudocode for the Extended Greedy Algorithm is as follows:

Table II. Extended Greedy Algorithm Pseudocode

```
1. Initialize total weight and profit variables to 0
2. Sort items by efficiency in decreasing order (highest first).
3. Iterate through the sorted items, and for each item:
    - If adding item does not exceed the weight capacity:
        - Include the item
        - Update the total weight
        - Update the total profit
    - Else:
        - skip adding the item to knapsack
```

With the implementation of this special case, the Extended Greedy Algorithm aims to provide a promising alternative to the standard Greedy approach by offering improved results for certain instances of the KP.

B. Dynamic Programming Algorithm

As previously mentioned, the Dynamic Programming approach solves the 0/1 Knapsack Problem by breaking it down into smaller subproblems and iteratively finding each optimal solution. To illustrate the steps of the Dynamic Programming algorithm variant, consider the following pseudocode:

Table III. Dynamic Programming Algorithm Pseudocode

```
1. Initialization
    - Initialize total weight and profit variables to 0
    - Create DP array[n+1, capacity+1] and initialize all cells
      to 0
      Note: The DP array will contain the Bellman Recursion
      Table

2. Bellman Recursion:
    - Iterate from 1 to n, For each item:
        - Iterate from 0 to capacity, For each sub-capacity:
            - If Adding Current Item Exceeds Capacity
                - Set cell to the previous best solution (cell
                above)
            - Else
                - Set current cell to max(cell above current
                cell, best value for remaining capacity)

3. Reconstruction:
    - Initialize empty list to store knapsack items
    - While able to reverse iterate through DP table:
        - If current cell is different from above cell:
            - Include item in bag
            - Update knapsack weight
        - Continue to next item
```

The Dynamic Programming algorithm considers all possible combinations of items to achieve the maximum profit

while adhering to the knapsack's weight capacity. Through its iterative nature and efficient subproblem solving, DP guarantees the most optimal solution, making it a powerful algorithm for solving the 0/1 Knapsack Problem [6].

IV. Time Complexity

A. Extended Greedy Algorithm

The time complexity of the Extended Greedy Algorithm depends on whether the sorting of items takes place inside the function or not. This item sort requires $O(n \log n)$, where n pertains to the number of items. If the items are sorted before calling the algorithm, the iteration through the sorted items takes linear time, $O(n)$ [6]. During which, all subsequent lines, to verify if an item can be included in the knapsack and to update the total weight and profit variables, can be done in constant time, $O(1)$. As such, the overall time complexity of the Extended Greedy Algorithm is $O(n)$ if the sort takes place prior to the algorithm call, and $O(n \log n)$ if the sorting happens within the call. It is important to note that these complexities directly relate to the algorithm only; no matter where the sorting takes place, the time complexity of the program will be dominated by the sorting step, resulting in a time complexity of $O(n \log n)$, unless other code within the program becomes the dominant factor.

B. Dynamic Programming

Similar to the Extended Greedy Algorithm, the time efficiency of the Dynamic Programming (DP) Algorithm can vary depending on the presence of the Reconstruction section. While the example DP pseudocode in Table III includes the Reconstruction section, it is worth noting that this part is optional. If the primary objective of the DP algorithm is to determine the most optimal value for the KP, it may return after completing the Bellman Recursion [6][7].

The Bellman Recursion section of the Dynamic Programming Algorithm involves iterating through each item and each sub-capacity of the knapsack and filling a table to calculate their corresponding optimal value solutions. Each iteration runs in linear time, with the outer loop executing once for each item from 1 to $n + 1$, resulting in a time complexity of $O(n)$. The inner loop runs once for each sub-capacity from 0 to c, leading to a time complexity of $O(c)$. Therefore, if the algorithm were to return after the Bellman Recursion, the time complexity will be dominated by the second for-loop and have a time complexity of $O(nc)$[6].

Alternatively, if the algorithm also aims to identify the corresponding items for the optimal value, it can invoke the Reconstruction section to backtrack through the DP table and retrieve the items within the knapsack. This would result in additional iterations, backwards through the table, significantly increasing the time complexity to $O(n^2 c)$ [6].

V. Code

A. Script Startup

To analyze the runtime of each algorithm, you must first ensure that the required libraries are installed. Both programs are written using python and utilize the `prettytable` library for displaying data in formatted table format. If not already installed, this library can be installed using pip:

```
> pip install prettytable
```

Once you have ensured that both scripts, "kp_extended_greedy.py" and "kp_dynamic.py", are saved an easily accessible directory, they can be ran using the following format:

```
> python3 <program_name>.py
```

For example, to run "kp_dynamic.py", you would type the following into your command prompt.

```
> python3 kp_dynamic.py
```

By default, both scripts generate random items and their corresponding weight and profit values. However, you can specify the number of items using the flag '`-n <items>`'. For example, the command shown below would execute the DP algorithm with 15 items.

```
> python3 kp_dynamic.py -n 15
```

Additionally, if you want to run the scripts with a constant set of items, you can execute the DEBUG flag. This flag can be invoked typing either '`-d`' or '`--debug`'. For example:

```
> python3 kp_dynamic.py -d
> python3 kp_dynamic.py --debug
```

Running this script will use a set of predefined items to enable performance comparison on the same set of items and the ability to observe the difference in their respective runtimes. This flag defaults to 10 items with the following properties:

Table IV. Script Default Items

Item	*Weight*	*Value*
1	2	10
2	3	15
3	5	8
4	4	12
5	1	6
6	6	20
7	2	14
8	7	30
9	3	25
10	4	18

B. Script Insight

In this section, we will compare the "`kp_extended_greedy.py`" script, which uses the Extended Greedy algorithm with the "`kp_dynamic.py`" script, which employs the Dynamic Programming algorithm. Both algorithms aim to solve the 0/1 Knapsack Problem as efficiently as possible, given their respective time complexities.

```python
def ext_greedy_knapsack(item_list):
    # Initialize Variables
    current_weight = 0
    current_value = 0

    # Iterate List of Items
    for each in item_list:

        # If Adding Current Item Does Not Exceed Capacity
        if each.weight + current_weight <= max_capacity:

            # Add Item to Knapsack
            knapsack.append(each)
            each.update_knap_status(True)

            # Update Knapsack Values
            current_weight += each.weight
            current_value += each.value

    # Return Total Knapsack Values
    return current_weight, current_value
```

Fig I. Extended Greedy Algorithm in Python

The "kp_extended_greedy.py" script shown in Fig I implements the Extended Greedy algorithm, which is a variant of the Greedy Algorithm with an additional step to handle the special case discussed in Section III.

Prior to entering this function, a crucial step is performed where the items are sorted in descending order based on their efficiency $\left(\frac{value}{weight}\right)$. This sorting is essential as it enables the algorithm to start its iteration from the top of the list and work its way down, optimizing the process for maximum efficiency.

Once inside the algorithm, it begins by initializing the variables for the current total weight and value of items in the knapsack (lines 70-71). Then the algorithm iterates through the list of items, and determines if adding the current item to the knapsack would exceed its capacity (line 77). If the addition of the item is permissible, it is added to the knapsack (lines 80-81). Subsequently, the algorithm updates the current total weight and value to reflect the inclusion of the new item, ensuring accurate tracking of the knapsack's contents and their cumulative values (lines 84-85).

```python
""" Bellman Recursion: Fill in DP table """
for each_item in range(1, n+1):
    for each_capacity in range(max_capacity+1):
        current_weight = item_list[each_item - 1].weight
        current_value = item_list[each_item - 1].value

        # If Adding Current Item Exceeds Capacity
        if each_capacity < current_weight:
            # Set cell to the previous best solution
            DP[each_item][each_capacity] = DP[each_item - 1][each_capacity]     # current cell = above cell

        else:
            # Set cell to max(cell above current cell, best value for the remaining capacity)
            DP[each_item][each_capacity] = max(DP[each_item - 1][each_capacity],
                                               DP[each_item - 1][each_capacity - current_weight] + current_value)
```

Fig II. DP Algorithm in Python - Bellman Recursion

The "kp_dynamic.py" script leverages the functionality of Dynamic Programming to break down the KP into smaller subproblems and iteratively find the optimal solutions. The DP computes these optimal solution values for each subproblem using the Bellman Recursion technique, shown in Fig. II. This recursion iterates through each item (line 69) and each sub-capacity (line 70) of the knapsack to calculate their corresponding values. If adding the current item exceeds the capacity (line 75), the cell in the DP table is set to the previous best solution (line 77). Otherwise, it sets the cell to the maximum value between the cell above it and the best value for the remaining capacity (line 81).

```python
""" Reconstruction: Determine items in the knapsack """
current_item = n                    # set current item to last cell
current_capacity = max_capacity     # set current capacity to last cell

# Iterate backwards through table
while current_item > 0 and current_capacity > 0:

    # If current cell is different from cell above
    if DP[current_item][current_capacity] != DP[current_item-1][current_capacity]:

        # included current_item in knapsack
        knapsack.append(item_list[current_item - 1])            # append item to knapsack list
        knap_weight += item_list[current_item - 1].weight    # update weight
        knap_value += item_list[current_item - 1].value      # update value
        current_capacity -= item_list[current_item - 1].weight  # update capacity

    current_item -= 1   # decrement cell

""" Return optimal solution value """
return DP[n][max_capacity]
```

Fig III. DP Algorithm in Python - Reconstruction

Upon completion of the Bellman Recursion, the DP algorithm proceeds with the Reconstruction phase, as shown in Fig III, to determine the items in the knapsack. The reconstruction process iterates backward through the DP table (line 88), examining the differences between the current cell and the cell above it. If they differ (line 92), the current item is included in the knapsack, and its weight, value, and capacity are updated accordingly (lines 95-98).

C. *Debug Runtime Comparison*

As shown in Fig II, when the Greedy Algorithm script is run using the DEBUG flag, 7 of the 10 items are added to the Knapsack, utilizing the entire capacity of 22, for a total value of 118. For this scenario, the algorithm completes execution in 0.000003 seconds while the script completes execution in 0.000717 seconds.

```
There were 7 items added to the Knapsack:
+-------------------------------------+
| Item   Weight   Value   Efficiency |
+-------------------------------------+
|  0       2        10        5      |
|  1       3        15        5      |
|  4       1        6         6      |
|  6       2        14        7      |
|  7       7        30        4      |
|  8       3        25        8      |
|  9       4        18        4      |
+-------------------------------------+

Greedy Knapsack Summary:
+----------------------+
| Total Value      118 |
| Total Weight     22  |
| Total Capacity   22  |
+----------------------+

Algorithm runtime: 0.000003 seconds
Script runtime: 0.000717 seconds
```

Fig. IV:Extended Greedy Debug Output

As shown in Fig V, when the DP algorithm script is run using the DEBUG flag, the same 7 items are added to the Knapsack as the Greedy Algorithm. For this scenario, the DP algorithm completes execution in 0.000051 seconds, while the entire script completes execution in 0.000641 seconds.

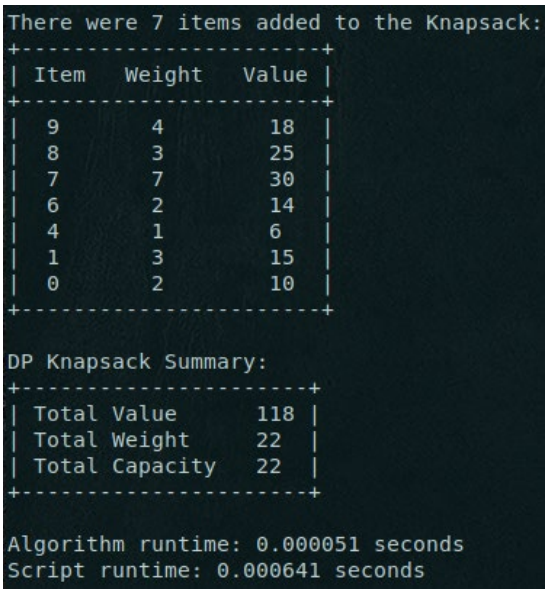

Fig V. Extended Greedy Debug Output

It is evident that the Greedy Algorithm excels in terms of runtime efficiency compared to the DP algorithm for this specific scenario. The Greedy Algorithm's fast execution can be attributed to its linear time complexity of $O(n)$, as discussed in Section IV: Time Complexity portion of the report. On the other hand, the DP algorithm's time complexity of $O(n^2c)$ results in a relatively longer runtime.

D. High-Volume Stress-Test Runtime Comparison

To further test each algorithm's performance at a higher threshold, I conducted an analysis using 2000 items by executing the flag '`-n 2000`'. As these items were randomly generated, it's important to note that the specific knapsack contents cannot be directly compared between algorithm. However, focusing on the runtime efficiency, the results are quite notable.

Fig VI. shows the runtime results for the Greedy algorithm when $n = 2000$. As shown, it completed its execution in an impressive 0.000261 seconds. Additionally, the entire script, including sorting, took 0.086104 seconds to complete.

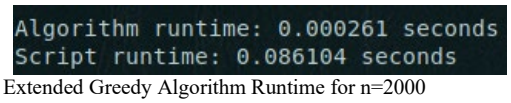

Fig VI. Extended Greedy Algorithm Runtime for n=2000

Fig VII. shows the runtime results for the equivalent stress test when ran on the DP. As shown, it took over three seconds to complete its execution, at 3.196040 seconds. Additionally, the entire script, including setup and output generation, took 3.279251 seconds to complete.

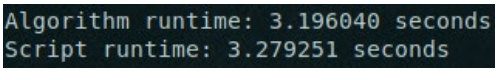

Figure VII. DP Algorithm Runtime for n=2000

This comparison highlights the Greedy Algorithm's strength in handling larger datasets efficiently, making it a compelling choice for scenarios where fast runtime is a priority. However, it's important to note that we did not analyze the selection in terms of optimal value selection. While the Greedy algorithm is more efficient, the DP algorithm spends more time analyzing the items to provide an optimal solution to the KP, ensuring the highest possible value in the knapsack for any given set of items and capacity.

In conclusion, the runtime comparison of the Greedy and DP algorithms at different thresholds reveal valuable insights into their respective strengths and weaknesses. The choice between the two algorithms ultimately depends on the specific requirements of the problem and the desired balance between runtime efficiency and optimality of the solution.

VI. Conclusion

In conclusion, the Knapsack Problem (KP) has been a subject of interest since its first appearance in 1895, rooted in Mathews' work on combinatorial optimization. Over the years, it has paved the way for related problems like the Traveling Salesman Problem and the Graph Coloring Problem, becoming a worthy challenge in resource allocation. The KP's practicality has made it applicable in various real-world scenarios, leading to the exploration of variants and solutions such as the Bounded Knapsack Problem, Unbounded Knapsack Problem, 0/1 Knapsack Problem, Extended Greedy Algorithm, and Dynamic Programming Algorithm. The KP's versatility has made it applicable in diverse real-world scenarios, ranging from program partitioning, task allocation, and to investment portfolio optimization. As researchers and mathematicians continue to study new variations and algorithmic improvements, the KP's significance and popularity will continue to spread.

VII. References

[1] R. Hurbans, Grokking Artificial Intelligence Algorithms, Shelter Island, NY: Manning Publications Co, 2020.

[2] A. Bhargava, Grokking Algorithms, Shelter Island, NY: Manning Publications Co, 2016.

[3] G. Mathews, "On the Partition of Numbers," *Proceedings of the London Mathematical Society,* June 1897.

[4] M. La Rocca, Advanced Algorithms and Data Structures, Shelter Island, NY: Manning Publications Co, 2021.

[5] D. Pisinger, "Where are the hard knapsack problems?," Copenhagen, 2003.

[6] H. Kellerer, U. Pferschy and D. Pisinger, Knapsack Problems, Berlin: Springer, 2004.

[7] D. Zingaro, Algorithmic Thinking, San Francisco, CA: No Starch Press, 2020.

[8] A. Billionnet and F. Calmels, "Linear programming for the 0–1 quadratic knapsack problem," *European Journal of Operational Research,* vol. 92, no. 2, pp. 310-325, 19 July 1996.

[9] R. Andonov, V. Poirriez and S. Rajopadhye, "Unbounded knapsack problem: Dynamic programming revisited," *European Journal of Operational Research,* vol. 123, no. 2, pp. 394-407, June 2000.

[10] Q. Wang, K. H. Lai and C. Tang, "Solving combinatorial optimization problems over graphs with BERT-Based Deep Reinforcement Learning," *Information Sciences,* vol. 619, pp. 930-946, January 2023.