CMSC142 - Design and Analysis of Algorithms Machine Problem 1: An Analysis on the Efficiency of the Exhaustive Search Algorithm

Katrina Ang [*] and Elaine Pajarillo [*]

Department of Mathematics and Computer Science, University of the Philippines - Baguio
[*]Corresponding author: knang@up.edu.ph
[*]Corresponding author: etpajarillo@up.edu.ph

Abstract

In this paper, we investigate the computational efficiency of an exhaustive search algorithm for solving the 0-1 Knapsack Problem, a classic combinatorial optimization challenge involving the selection of items to maximize the total value within a fixed weight capacity. This experiment aims to empirically evaluate how execution time scales with increasing item counts, highlighting the limitations of the brute-force approach. Using a Python-based Gray code algorithm, we systematically generated all possible subsets of items, measuring the time to determine the optimal solution for the number of items from $n = 10$ to $n = 50$. Random item weights of 50 to 100 and values of 100 to 500 were assigned within a fixed knapsack capacity of 1000. The experimental findings reveal that execution times remain manageable for smaller item sizes but escalate sharply beyond $n = 25$, reaching values over 2000 seconds at $n = 50$. This exponential growth pattern, as visualized through polynomial-fitted curves, confirms the theoretical $O(2^n)$ time complexity associated with exhaustive search. These results emphasize that while exhaustive search reliably finds optimal solutions, its computational demands make it impractical for large data sets. These insights support the need for more efficient approaches, such as dynamic programming or greedy algorithms, for scalable solutions to complex optimization problems like the Knapsack Problem.

1 Introduction

Optimization problems are fundamental in different fields of computer science and mathematics that involve logically finding the "best" solution to a situation from a set of feasible options based on certain specifications. Most such problems involve finding an element which maximizes or minimizes a desired characteristic, and typically arise in situations involving permutations, combinations, and subsets of a set (Levitin, 2012). Such problems are important not only theoretically, but the consideration of processes that optimize the allocation of limited resources and improve decision making processes can be applied in various real-world situations such as in designing travel efficient and cost-effective public transit systems, determining the most efficient network and data storage methods, etc.

A systematic and finite approach to these problems is the exhaustive search method, which is described by Levitin (2012) as a brute-force approach to combinatorial optimization problems. It proposes creating every possible solution, choosing those that satisfy the predefined constraints of the problem, and then determining the best option among the feasible options. The consideration and analysis of this method are key to understanding the performance and limitations of brute-force methods. Throughout this paper, we aim to understand the general performance and efficacy of such algorithms to solve problems in terms of the computational cost that is required to explore every possible solution, even for large sets of data.

Among these, one of the most well-known optimization problems is an NP-hard optimization problem involving resource allocation called the Knapsack Problem. More specifically, the '0-1' knapsack problem, in which we are given a knapsack with capacity W, and n items with weights $w_1, w_2, ..., w_n$ and values $v_1, v_2, ..., v_n$ the goal is to select a subset of the items $x_1, x_2, ..., x_n$ in such a way that the total weight does not exceed W while the profit margin of the values in the subset is maximized. This is formally described as

$$\begin{aligned} &\text{maximize} \quad \sum_{i=1}^{n} v_i x_i \\ &\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq W \\ &x_i \in \{0,1\}, \quad \forall i \in \{1,2,\ldots,n\} \end{aligned} \tag{1}$$

Note that in the case of the '0-1' knapsack problem, the values of x are limited to 0 or 1 only, with 0 indicating you leave the entire item or take the entire item inside the knapsack, as there is no option to take only a fraction of the item in the case of this problem (Jooken et al., 2023). The exhaustive-search approach to this problem would be to generate all subsets of the set of n items, compute the total weight w of each subset and identify the feasible subsets in context of the set capacity W of the knapsack, and finally identify the subset containing the largest value among them, deeming this as the "best solution"(Levitin, 2012).

Through the course of this paper, the students aim to investigate the efficiency of the exhaustive search algorithm by solving the 0-1 Knapsack problem. In particular, we aim to perform an empirical analysis through running an exhaustive search-type algorithm on different input sizes and measuring the time taken to find the best solution. For the purpose of this analysis, the knapsack capacity W is fixed to 1000, the weight w of the items are random positive integers from 50 to 100, the values v of the items are random positive integers from 100 to 500, and the number of items n to choose from is $n = 10$ to $n = 50$. The next portion of the paper will discuss the exact specifications, methods, and processes used by the students to achieve this objective based on the general definition provided. This includes the algorithm used, experiment setup, generation of inputs, and the tools and methods utilized in measuring runtime that the students put to use in achieving their results.

2 Methodology

This section details the approach taken to empirically analyze the time efficiency of an exhaustive search algorithm for solving the 0-1 Knapsack problem with the given initial parameters.

2.1 Algorithm Overview

The algorithm used to solve the 0-1 Knapsack problem is a brute force, exhaustive search algorithm that uses a Gray code-based approach to generate all possible subsets of items as well as to maximize the faster computation of the subset's total value. In this algorithm, the solution is derived by evaluating every combination of the items in the set to find the best subset of items that can fit within the knapsack's fixed capacity while maximizing the total value.

The algorithm operates by iterating through the integers from 0 to $2^n - 1$, with n as the number of items. The minimal change algorithm, the binary-reflected Gray code representation of each number, is utilized to reduce the number of alterations needed to move from one subset to the next making it easier to traverse all possible combinations as the gray code ensures that only one element changes at a time (Levitin, 2012). The algorithm calculates the total weight and value for each subset, comparing them to find the optimal solution. The running time it took to find the best subset during this process was then returned as the output.

Furthermore, the students decided to use Python version 2.6 and above as the programming language based on the ease of use for data fitting and analysis due to the large number of built-in libraries and methods such as `NumPy` that allow for more simple and readable code (UCD Professional Academy, n.d.). The specific Python code utilized by the students in solving the problem can be seen as follows

Python Code for the 0-1 Knapsack Problem Algorithm

```
import random
import time
import numpy as np
import matplotlib.pyplot as plt

# Define the Knapsack problem parameters
capacity = 1000
n = 50  # Total number of items
min_weight = 50
max_weight = 100
min_value = 100
max_value = 500

# Generate a set of 50 random items (weights and values)
items = [(random.randint(min_weight, max_weight),
random.randint(min_value, max_value)) for _ in range(n)]

# Function to convert an integer to its binary reflected Gray code equivalent
def gray_code_convert(num):
    return num ^ (num >> 1)

# Exhaustive search algorithm to solve the 0-1 Knapsack problem using Gray code
def knapsack_exhaustive_search(capacity, items):
    n = len(items)
    best_value = 0
    best_subset_indices = []
    best_total_weight = 0

    for subset in range(1 << n):
        gray_subset = gray_code_convert(subset)
        total_weight = 0
        total_value = 0
```

Python Code for the 0-1 Knapsack Problem Algorithm (continued)

```
        current_subset_indices = []

        for i in range(n):
            if gray_subset & (1 << i):
                total_weight += items[i][0]
                total_value += items[i][1]
                current_subset_indices.append(i)

        if total_weight <= capacity and total_value > best_value:
            best_value = total_value
            best_subset_indices = current_subset_indices
            best_total_weight = total_weight

    best_subset = [(items[i][0], items[i][1]) for i in best_subset_indices]
    return best_value, best_total_weight, best_subset, best_subset_indices

# Test runtimes for n = 10, n = 15, and n = 20
for test_n in [10, 15, 20]:
    test_items = items[:test_n]
    start_time = time.time()
    knapsack_exhaustive_search(capacity, test_items)
    end_time = time.time()
    print(f"Execution time for {test_n} items: {end_time - start_time:.4f} seconds")

# EXTRAPOLATION METHOD
# Gather execution times for smaller test sizes (e.g., from 10 to 25 items)
test_sizes = range(10, 26)
execution_times = []

for test_size in test_sizes:  # Iterate over values in test_sizes range
    test_items = items[:test_size]  # Create new list containing the first x items
    start_time = time.time()  # Record current time as start time
    knapsack_exhaustive_search(capacity, test_items)
    end_time = time.time()
    execution_times.append(end_time - start_time)

# Fit a polynomial to the data
coefficients = np.polyfit(test_sizes, execution_times, deg=2)
# Fit a polynomial of degree 2
polynomial = np.poly1d(coefficients)

# Predict execution times for 25, 30, 35, 40, 45, and  50 items
test_sizes_to_predict = [25, 30, 35, 40, 45, 50]
predicted_times = polynomial(test_sizes_to_predict)

for size, time in zip(test_sizes_to_predict, predicted_times):
    print(f"Predicted execution time for {size} items: {time:.4f} seconds")
```

Python Code for the 0-1 Knapsack Problem Algorithm (continued)

```
# Plot the data and the fitted curve
plt.scatter(test_sizes, execution_times, color='red', label='Actual data')
plt.plot(range(10, 51), polynomial(range(10, 51)), label='Fitted polynomial',
color='blue')
plt.scatter(test_sizes_to_predict, predicted_times, color='green', label=
'Predicted data')
plt.xlabel('Number of Items')
plt.ylabel('Execution Time (seconds)')
plt.title('Extrapolated Execution Time for Knapsack Problem')
plt.legend()
plt.show()
```

2.2 Experiment Setup

The experimental setup involved establishing the parameters for the knapsack problem and configuring the conditions for running the algorithm. The knapsack capacity was fixed at 1000 which represents the maximum capacity it could hold, the weight of each item was a random positive integer from 50 to 100, the values of the items were also random positive integers from 100 to 500, and the number of items n to choose from was $n = 10$ to $n = 50$. A total of 50-item set were generated for testing, with each item having its weight and value randomly assigned within the specified ranges.

The experiment's test trials were performed initially by running the algorithm using Python version 2.6 and above on a MacBook Air M1 with M1 chip processor, 512 SSD storage space, 8 GB RAM, and MacOS Sequoia 15.0.1 operating system. However, given the substantial computational demands and overall execution time in a local compiler environment within the specified laptop, we have opted to run the code on Google Colab, an online compiler hosted by the Jupyter Notebook service. This online service allows a significant execution speed for the given code above, as it provides more powerful cloud-based hardware with higher computing capabilities than previously specified locally (Google Colaboration, n.d.). In specific, the Google Colab runtime environment ran the algorithm using Python 3 with system RAM identified as 12.7 GB and a 107.7 GB disk storage.

The experiment aimed to test the algorithm's run-time performance using different input sizes, specifically selecting subsets of items ranging from 10 to 50. This approach allows for a detailed analysis of how the execution time scales with the number of items considered in the knapsack problem.

2.3 Generation of Inputs

To generate the inputs of the experiment, a list of 50 random items was created, each item was represented as a tuple containing its weight and value, both of which were randomly selected from the specified setup mentioned above. The generation of random inputs was achieved using Python's built-in `random.randint()` function.

The code snippet below illustrates the declaration of initial parameters and the generation of the inputs.

```
# Define the Knapsack problem parameters
capacity = 1000
n = 50
min_weight = 50
max_weight = 100
min_value = 100
max_value = 500

# Generate a set of 50 random items (weights and values)
items = [(random.randint(min_weight, max_weight), random.randint(min_value, max_value)) for _ in range(n)]
```

2.4 Tools & Methods in Measuring Runtime

The execution time of the algorithm was measured using the `time` module in Python, which provides a simple way to track how long the algorithm takes to run. The process involved capturing the time immediately before starting the execution of the algorithm and then capturing it again immediately after the algorithm completed its execution. The difference between these two timestamps gives the code's total execution time in seconds.

This approach was executed using these specific lines of code.

```
start_time = time.time()
end_time = time.time()

execution_time = end_time - start_time
```

Given the tendency of an exhaustive search algorithm to grow exponentially, we have adapted an extrapolation method to get the predicted execution time of the program for larger test cases. We used the built-in `polyfit` function in Python, provided by the `numpy` library to perform polynomial regression. The `polyfit` function fits a polynomial of a specified degree to a set of data points, effectively finding the best-fit line (or curve) that minimizes the difference between the data points and the polynomial model. The `polyfit` function takes three inputs: an array of x-coordinates (in this case, the test sizes), an array of y-coordinates (the execution times), and the degree of the polynomial (2 for this problem). It returns the coefficients for the best-fit polynomial. These coefficients will then be passed to another built-in `poly1d` function in Python, also part of the `numpy` library, and be used to create a polynomial object. This object was then treated as a function that the students used to evaluate various test cases, specifically, 25, 30, 35, 40, 45, and 50 test cases, and get the predicted execution time for each. Below is the code snippet of how the `polyfit` and `poly1d` were coded to fit the problem to be solved.

```
# Fit a polynomial to the data
coefficients = np.polyfit(test_sizes, execution_times, deg=2)  # Fit a polynomial of degree 2
polynomial = np.poly1d(coefficients)

# Predict execution times for 25, 30, 35, 40, 45, and 50 items
test_sizes_to_predict = [25, 30, 35, 40, 45, 50]
predicted_times = polynomial(test_sizes_to_predict)
```

In addition to these methods, the `matplotlib` Python library was used in plotting both the data points and the fitted curve for the trials and averages of the trials. In particular, the `pyplot` sub module was imported and served this function under the `plt` alias in order to create the graphs and their plotting areas. This function also enables the legends and labeling of all graphs seen under section 3.1 of the results and discussion (3).

The portion of the exhaustive search algorithm for the experiment is identified below.

```
# Plot the data and the fitted curve
plt.scatter(test_sizes, execution_times, color='red', label='Actual data')
plt.plot(range(10, 51), polynomial(range(10, 51)), label='Fitted polynomial',
color='blue')
plt.scatter(test_sizes_to_predict, predicted_times, color='green', label=
'Predicted data')
plt.xlabel('Number of Items')
plt.ylabel('Execution Time (seconds)')
plt.title('Extrapolated Execution Time for Knapsack Problem')
plt.legend()
plt.show()
```

Additionally, the Python code used to plot and visualize the average run times in relation to the number of items is displayed as follows. Note that the code to graph the average run times of the trials are separated from the main algorithm in order to accommodate the large runtime needed to provide the necessary information for the trial test cases, given the limited GPU and CPU capabilities of the compiler used.

Python Code to Graph the Average Running Time between 3 Trials

```
import matplotlib.pyplot as plt
#Number of items in the knapsack
n_val = [10, 15, 20, 25, 30, 35, 40, 45, 50]
#Averages of runtime from 3 trials
avg_time = [0.002766666667, 0.1197, 5.472866667, 144.7934, 354.6318, 649.5407,
1029.520233, 1494.5703, 2044.6909]

#Plotting the graph
plt.figure(figsize = (9, 5))
plt.plot(n_val, avg_time, marker = 'o', color = 'b', linestyle = '-',
linewidth = 1, markersize = 4)
plt.xlabel("Number of Items (n)")
plt.ylabel("Average Running Time (seconds)")

#Legends
plt.xlabel("Number of Items (n)")
plt.ylabel("Average Running Time (seconds)")
plt.title("Average Running Time vs Number of Items")
plt.grid(True)

plt.show()
```

2.5 Recording of Results

The results from each test case were recorded for analysis. The experiment was performed for 10, 15, 20, 25, 30, 35, 40, 45, and 50 test cases, and the algorithm was run three times in each test case to account for variability in execution time due to external factors such as system load, random number generation, and other hidden processes in the system.

After running the algorithm the run time it took for each test case was collected. These results were tabulated for each test size and each repetition, allowing for an average execution time to be calculated to provide a clearer picture of how the algorithm's performance scales with increasing input sizes.

The nature of the results was divided depending on the method used to arrive at these values. The first was the **actual execution time** of the algorithm outputted after running the program, performed for items $n = 10, 15, 20, 25$, and the second was the **predicted execution time** that was extrapolated using the polynomial regression through the built-in `polyfit` of Python that generated of the results for items $n = 30, 35, 40, 45, 50$.

Furthermore, the method of extrapolation will start after the test for the smaller test sizes has terminated, we have set the maximum number of items to be $n_{max} = 25$ as the runtime for items greater than 25 grows exponentially and takes a great deal of processing time, a point which will be made clearer towards the result and discussion portion of the paper.

3 Results and Discussion

This chapter presents the analysis of the data. All the gathered data from the experiment were presented and interpreted in visual forms, particularly using tables and graphs. Note that in analyzing the runtime of the exhaustive search algorithm, several potential sources of error may or may not impact the accuracy of the experiment's results.

One possible source of error could be the hardware and compilation tool used to obtain the results. Differences in the system and online Google Colab tool performance may impact the runtime measurements. Given the online nature of the compiler used, there is a chance that any sudden update or automatic change in the runtime environment at the mercy of the online compiler may introduce instability and variability to the results. Aside from this, the limited n_{max} value used ($n = 25$) in comparison to the item values it extrapolates ($n = 30$ to $n = 50$) may affect the accuracy of the true behavior of the runtime for the larger values of n. That is, it may over or underestimate the runtime predictions in regions far from the actual measured data, which could alter the understanding of the algorithm's behavior for larger item sizes. Another possible error in regard to the extraction of data could be in consideration of floating-point precision in the cases where there are small differences in the recorded values per item count. If such floating point errors pile on, there would be a possibility of inaccuracy within recorded run times.

3.1 Tables and Figures

That being said, the following section is devoted to describing the exact results extracted from the students' test cases.

n	Trial 1	Trial 2	Trial 3	Average
10	0.0026	0.0026	0.0031	0.002766666667
15	0.1191	0.1199	0.1201	0.1197
20	6.1641	5.0189	5.2356	5.472866667
25	142.9149	142.8725	148.5928	144.7934
30	350.216	349.52	364.1594	354.6318
35	641.5865	639.8796	667.156	649.5407
40	1017.0266	1013.9514	1057.5827	1029.520233
45	1476.5361	1471.7353	1535.4395	1494.5703
50	2020.1151	2013.2313	2100.7263	2044.6909

Table 1: Time in Seconds s of Exhaustive Search Method Runtime for Solving 0-1 Knapsack Problem

Table 1 describes the recorded runtime in seconds s of running the aforementioned exhaustive search algorithm for solving the 0-1 Knapsack Problem from $n = 10$ to $n = 50$, iterating by 5 for every value n. This range of values was chosen in order to provide a sufficient amount of n values for proper data plotting in consideration to the polynomial extrapolation method mentioned in the Methodology.

The following figures are the graphs taken from the data produced in each trial. These following graphs illustrate the execution time for solving the 0-1 Knapsack Problem using the students' exhaustive search algorithm as the number of n items increase. The x-axis represents the number of items from the range of $n = 10$ to $n = 50$ and the y-axis illustrates the execution time measured in seconds. As seen from the legend of these graphs, we also identify that the red dots represent the "actual data" or the data of which the items n were not extrapolated. That is, the the actual recorded execution times for the number of items $n = 10$ to $n = 20$. The green dots then depict the the predicted data, or the extrapolated execution times for the larger item counts $n = 25$ to $n = 50$, while the straight blue line portrays the fitted polynomial curve that approximates the execution time as a function of n.

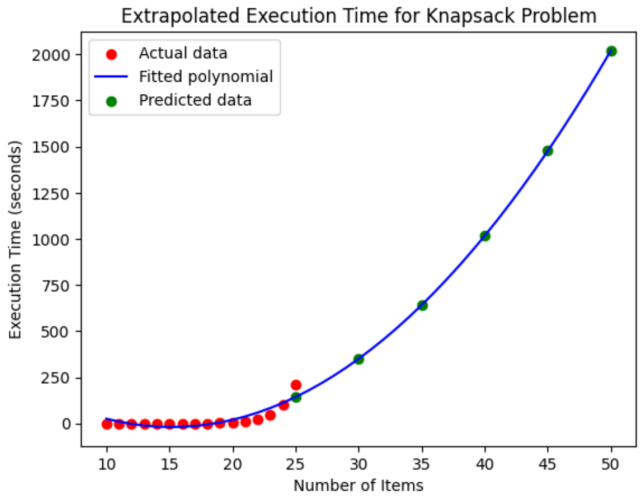

Figure 1: Trial 1 Graph for the Execution Time of Exhaustive Search Method for Solving the Knapsack Problem

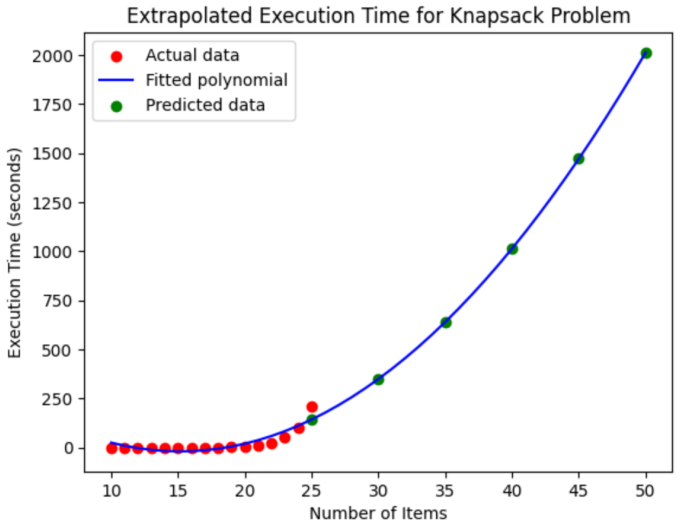

Figure 2: Trial 2 Graph for the Execution Time of Exhaustive Search Method for Solving the Knapsack Problem

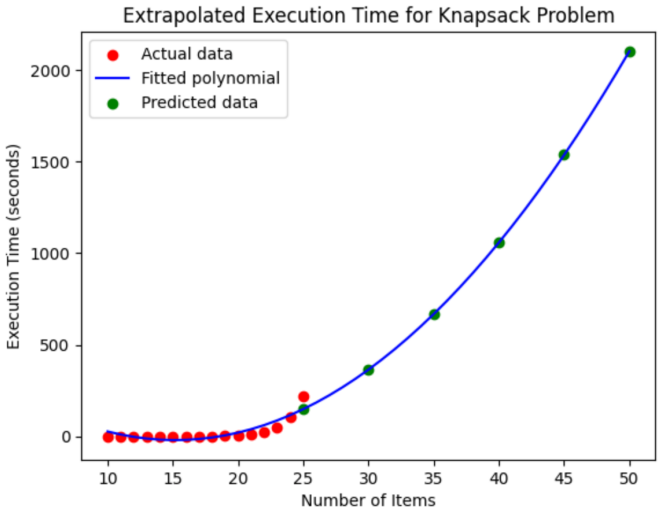

Figure 3: Trial 3 Graph for the Execution Time of Exhaustive Search Method for Solving the Knapsack Problem

The following graph represents the averages of the running times per each of the three trials in respect to the number of items in the knapsack.

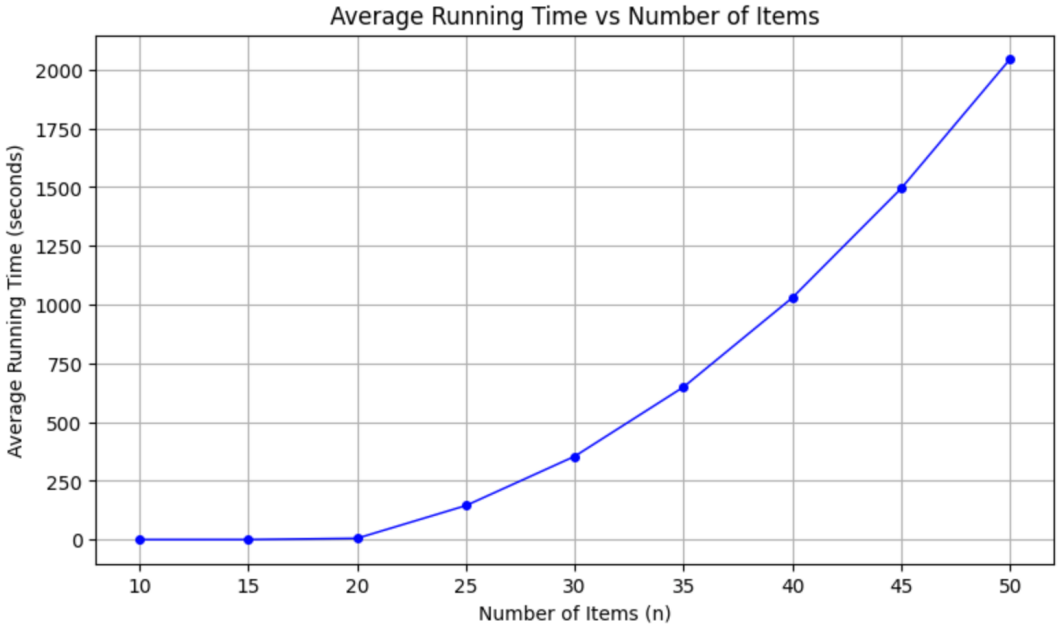

Figure 4: Graph of the Average Running Times From 3 Trials

As seen in the trials and visualized through figures 1 to 3, and further reiterated with the graph of the averages among the trials in figure 4, there is a clear trend in regards to the running time of the exhaustive search algorithm implemented. The graphs of the trials indicate a marked increase in the execution or computation time necessary to process items as item size n grows larger. The relationship between the actual data points, the extrapolated data points, and the fitted polynomial captures the initial approximation of the growth pattern as an upward increase. That is, the curve indicates that the execution time begins on a slow increase as knapsack items grow, but escalates rapidly after surpassing a sufficiently large n value of above 25 items.

This observation is further strengthened by considering the graph of the average running time taken from all 3 trials (Fig. 4). The graph shows that for every trial done for the exhaustive search algorithm in solving the Knapsack Problem, the average running time remains relatively small, ranging from around 0.0027 seconds to 5.47 seconds for the 10 to 20 items but as the number of items increases, there is an observable inflection point occurring around the n value $n = 25$. This leads to a sharp increase in running time from around item values 25 to 50, in which the running time may reach values above 2000 seconds, nearly doubling the time it takes to run the algorithm with every interval past 25.

Such a rapid and increasing curve is generally observed within every run of the algorithm on random values and weights, where even a small increase in the number of items in the knapsack results in a disproportionately large increase in running time can be observed as mimicking an exponential-like growth pattern. Such a pattern could lead to implications that as the number of items in the knapsack increases, the application of the algorithm becomes significantly more intensive computation-wise, which could potentially lead to complications in manageability and scalability for even larger data sets and sufficiently large values of n.

Further implications of the findings present within the test case results recovered during experimentation are discussed more in-depth in the next chapter, alongside the students' recommendations in improvements to alternatives and analysis to solving the 0-1 Knapsack problem algorithmically.

4 Conclusion

This paper has explored the computational performance of the exhaustive search algorithm in solving the 0-1 Knapsack Problem, focusing on the correlation between item count and execution time. Through the analysis of different data sets, we demonstrated that while the exhaustive search algorithm reliably identifies the optimal solution, its computational cost escalates exponentially as the number of items grows. This result aligns with theoretical expectations, confirming the expensive runtime of an exhaustive search for larger data sets.

The experiments, conducted with varying item sizes from $n = 10$ to $n = 50$, revealed that execution times increase sharply once the item count exceeds approximately 20. At smaller instances, average runtimes ranged from 0.0027 to 5.47 seconds, but for larger item counts, the algorithm's runtime increased drastically, with values exceeding 2000 seconds at n=50. This exponential growth, visualized through polynomial-fitted curves, highlights the impracticality of exhaustive search for large instances of NP-hard problems like the Knapsack Problem.

Considering the theoretical nature of brute-force algorithms such as the exhaustive search, which aims to examine all possible subsets of each item in finding the optimal solution, the algorithm described would lead to a time complexity of $O(2^n)$ given that for each item in the knapsack, the algorithm must decide whether or not to include it. This behavior was mirrored exactly in the graphs and results extracted from the experiment on random item values and weights, as runtime values increase exponentially as item size increases. As such, we were able to combine our findings from the experimentation with the conceptual theory that such algorithms may perform well enough for small values of n, but prove to be extremely impractical for larger values as computation needs will increase rapidly beyond a sufficiently large value of n.

From this, we conclude that the use of such brute-force algorithms to solve combinatorial problems such as the 0-1 Knapsack Problem becomes inefficient beyond a certain size threshold. This logic can be applied in emphasizing the limitations of exhaustive search algorithms for real-world applications of combinatorial problems. Building upon this conclusion, the students recommend implementing the algorithm only for smaller values and exploring other algorithm types, such as dynamic programming or greedy algorithms, to solve problems of this nature more effectively for larger values. These methods, while potentially less precise, could offer significant reductions in computation time and still yield near-optimal solutions, which are suitable for real-world applications where time constraints are critical.

5 References

[1] AlgoMonster. (2021). *Gray Code - In-Depth Explanation.* AlgoMonster. https://algo.monster/liteproblems/89

[2] Anany Levitin. (2012). *Introduction to the design and analysis of algorithms* (3rd ed., pp. 115–147). Pearson

[3] Digital Ocean. (n.d.). Python Bitwise Operators. https://www.digitalocean.com/community/tutorials/python-bitwise-operators

[4] Google. (n.d.). *Colaboratory – Google.* Research.google.com; Google. Retrieved October 31, 2024, from https://research.google.com/colaboratory/faq.html

[5] Jooken, J., Leyman, P., & De Causmaecker, P. (2023). Features for the 0-1 knapsack problem based on inclusionwise maximal solutions. *European Journal of Operational Research.* https://doi.org/10.1016/j.ejor.2023.04.023

[6] Kumar, A. (2022, July 25). What is XOR in Python? Scaler Topics. https://www.scaler.com/topics/xor-in-python/

[7] UCD Professional Academy. (2024). *Why Do Data Analysts Use Python?.* https://www.ucd.ie/professionalacademy/resources/why-do-data-analysts-use-python/

[8] Whealton, S. (2004). The Reflected Binary Gray Code Transform. Mathematical Connections in Art, Music, and Science, 149–156. Bridges Proceedings. https://doi.org/20010-2605