Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

January 24, 2025

Comparative Analysis of Algorithmic Strategies for the 0/1 Knapsack Problem

Dynamic Programming, Heuristics, Metaheuristics, and ILP Models for Combinatorial Optimization

Valentino Belotcaci

Abstract

This thesis evaluates different algorithmic approaches for solving the 0/1 Knapsack Problem, a combinatorial optimization problem. The project involves implementing and analyzing dynamic programming, greedy, branch-and-bound, Martello-Toth Algorithm, simulated annealing, Ant Colony Optimization, and Integer Linear Programming (ILP) models. These methods are evaluated for their efficiency, scalability, and performance on small, medium and large problem instances. The evaluation uses a set of metrics, including mean and median execution time, memory usage, the 1st and 3rd quartiles of the obtained profit, the variability of profits and execution times as measured by their standard deviation, deviation from the optimal solution (when available), and scalability trends as the problem size increases. The study is conducted on 5000 benchmarks created specifically for this project and 3000 additional benchmarks sourced from a research paper. This statistical comparison emphasize the strengths, limitations, and suitability of these approaches for different problem scenarios.

Advisor
Prof. Antonio Carzaniga

Advisor's approval (Prof. Antonio Carzaniga): Date:

Contents

1 Introduction

The Knapsack Problem is a combinatorial optimization problem with different practical applications in areas like resource allocation and logistics. It involves selecting a subset of items, each with a specific weight and value, to maximize the total value (profit) without exceeding a given capacity constraint imposed by the knapsack. This problem is classified as NP-hard because there isn't a known algorithm that can solve every instance of the problem optimally within a time complexity that scales polynomially with the size of the input.

1.1 Historical Background and Practical Scenarios

The Knapsack Problem derives its name and essence from a scenario faced by a mountain climber according to [3] who must decide which items to pack into a knapsack. Each item has a specific weight and profit value, and the climber's goal is to maximize the profit while ensuring the total weight does not exceed the knapsack's capacity. This dilemma is not just theoretical but also reflects real-world optimization challenges.

For example consider a data storage problem where n data files of different sizes (weights) and computing importance (values) need to be stored on a device with limited capacity W. The goal is to select a subset of files that maximizes the total computing utility while ensuring the total size of selected files does not exceed the total storage. Such scenarios underline the relevance of the Knapsack Problem in areas such as cloud storage management, logistics planning, and resource allocation.

The Knapsack Problem has given rise to several variations, including:

- **Multiple Choice Knapsack Problem:** Items are grouped into disjoint sets, and the task is to choose one item from each set while respecting the capacity constraint.
- **Multiple Knapsack Problem:** There are multiple knapsacks that need to be filled at the same time, optimizing the profit of all the knapsacks.

1.2 Mathematical Formulation of the 0/1 Knapsack Problem

The 0/1 Knapsack Problem can be defined as follows according to [4]:

- Let n denote the number of items available.
- Each item i has a value $v_i > 0$ and a weight $w_i > 0$.
- The capacity of the knapsack is W.
- Define x_i as a binary decision variable where:

$$x_i = \begin{cases} 1 & \text{if item } i \text{ is included in the knapsack,} \\ 0 & \text{otherwise.} \end{cases}$$

The objective is to maximize the total value of items included in the knapsack, subject to the weight constraint:

$$\text{Maximize } Z = \sum_{i=1}^{n} v_i x_i$$

$$\text{Subject to: } \sum_{i=1}^{n} w_i x_i \leq W, \quad x_i \in \{0, 1\}, \forall i \in \{1, 2, \dots, n\}.$$

1.3 Purpose of the Study and motivation

This study focuses on implementing and comparing various algorithmic approaches to solving the 0/1 Knapsack Problem. The Knapsack problem has many important applications in various areas in engineering and logistics. However, given the NP-hard nature of the Knapsack Problem, finding efficient and scalable solutions is likely impossible in general. We therefore have to resort to various heuristics, that may be effective for some classes of problems. Comparing these different algorithms provides information of their performance, helping identify the most suitable methods for different scenarios. This is important when dealing with large-scale benchmarks and the need for a statistical evaluation of algorithmic efficiency.

2 Implementation Details

The project was implemented using two programming languages, each serving a specific purpose: The core algorithms were implemented in C. C was chosen for its performance efficiency and control over low-level operations, which is really important when executing computationally intensive algorithms efficiently.

On the other hand, we used Python for the creation of benchmarks (problem instances) as well as for data processing. In particular, python was used for:

- Generating and managing benchmark instances for the Knapsack Problem.
- Analyzing and visualizing the results using statistical libraries such as NumPy and pandas.
- Producing plots to evaluate algorithm performance.

This combination of languages allowed for an optimal balance between performance (C) and flexibility in data analysis and visualization (Python). The benchmarks created in Python were used as input for the C implementations, and the output data was then processed and examined using Python tools to extract the desired information.

3 Benchmark Creation and Results

The goal of generating these benchmarks was to explore different data samples and study aspects of algorithm performance: in detail we wanted to evaluate how the algorithms perform as the problem size increases or as the capacity increases, measure computational time and memory usage for different problem complexities and determining the ability to find almost optimal or optimal solutions on different instances.
By changing the size, capacity, and item characteristics, the benchmarks provide a complete dataset for evaluating the pros and cons of each algorithm in realistic scenarios.
To evaluate the performance of various algorithms for solving the 0/1 Knapsack Problem, a set of benchmark instances was generated using Python. The benchmarks were designed to include a wide range of problem complexities, enabling a detailed analysis of algorithmic behavior and performance. This section describes the way these benchmarks were created.
One high-level parameter was the *size* of the problem instance. The benchmarks were generated to cover different sizes, ranging from small-scale problems to large-scale, computationally intensive ones. Specifically, the selected sizes were: 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, and 10000. This range ensures that the algorithms were tested on different sized instances, providing information into their efficiency.
It is also crucial to test the algorithms with problem instances characterized by different levels of resources. The main resource in the Knapsack problem is the *capacity* of the knapsack. We therefore divided each size into five groups, each with a distinct knapsack capacity. The capacities were randomly chosen within a range of 10 to 1000. We decided to have an upper bound of 1000 of capacity for this set of generated benchmarks because the other online benchmarks (taken from an external source) will cover bigger sizes.
For a given capacity and size, what then defines the problem instance is the distribution of weights and profits of the item. Each benchmark instance consisted of items with the following attributes: For the weights, we generated random values between 1 and half the knapsack capacity. As for the profits, we randomly generated values between 1 and 100. The randomization of weights and profits was done in order to follow realistic scenarios where item characteristics are not predictable.
The input format also requires a unique identifier for each item. The chosen format is the one defined for the external, online benchmarks. We use the same format for the sake of uniformity in our experimentation process.
Another high-level goal of our experimentation was to obtain statistically significant results. We therefore generated and experimented with a large number of problem instances. In particular, we created 100 benchmarks for each size, resulting in a robust dataset for performance analysis.
In practice, we organized the generated benchmarks in a directory tree that also follows the online benchmarks' folder structure: each benchmark was stored in a folder named `benchmark_<number>` into the main folder `structured_benchmarks`, subsequently the data for each benchmark was saved in a file named `test.in`, formatted as follows:

- **First Line:** The number of items.
- **Next *n* Lines:** Item details in the format `<ID> <Profit> <Weight>`.
- **Last Line:** The knapsack capacity.

Results format

The output of the algorithms is presented in a tabular format, as shown below:

Algorithm	Folder	Size	Capacity	Time (s)	Memory (kB)	Best Value
Greedy Algorithm	benchmark_4998	10000	287	1.373325	2304	9833
Greedy Algorithm	benchmark_4999	10000	287	1.436566	2304	10325
Greedy Algorithm	benchmark_5000	10000	287	1.224091	2304	9519
...						
...						
Total Time (s): 793.902756						

Table 1. Output format for algorithm results.

Special Cases

- **Timeout:** If the algorithm exceeds the given time limit (e.g., 180 seconds), the output for *Best Value* is set to **-1**. - **Full Memory:** If the algorithm runs out of memory, the output for *Best Value* is set to **-1**.
This format ensures that results are easy to interpret, with chosen indicators for exceptional cases.

4 Implementation Issues

4.1 Code Implementation

Problem: At the beginning the main problem we faced was that we were creating the algorithm's implementations in separate files and then porform a function call 2 different main files, one for the benchmarks we created and one for the benchmark found on the internet, this happened because we didn't create the benchmarks in the same way that the online benchmarks were structured. Another weakness of using a main is that retrieving the data from the benchmarks and then running each algorithm for each file, was giving some major problems, for example each new algorithm needed to be added a call on the main function in order to run it, we were running all the algorithms for each file, this results in some problems because some algorithms may not be fully implemented and we also had to wait all the algorithms to run all the benchmark files in order to get the results.

Solution: We decided to remove all the work we did on the 2 different main files, recreate all the benchmark files in order to make it structured like the online benchmarks and create a script file in order to run each algorithm. This results in an increased time performance because the script opens the benchmark folder once and iterate through all the files of the benchmark and run the algorithms. We then decided to create only one script file for each algorithm (and not create one script that runs all the algorithms simultaneously) in order to see which algorithms work faster. This gave me more power and control over the algorithms because I was able to run each algorithm and see which one could have some problems, since some algorithms are not really intuitive, we then had to run them multiple times and correct the errors, until we reached a well working set of algorithms.

4.2 Online Benchmark Results

Problem: At the beginning we decided to run the online benchmarks but we were getting full memory usage for example for the dynamic programming (when the capacity is really big), or insanely large running times for example for the Branch and Bound algorithm since the running time increases as the values of the benchmark increase.
Solution: We decided to concentrate more on how the benchmark's names are structured and we noticed that the capacity and the size of the benchmark were written in the name of each file. In order to get some results from the different algorithms runs, I had to select the benchmarks by capacity (for the dynamic programming) and by size (for the Branch and Bound); we will explain how we selected the different benchmarks for each algorithm in another section. By running the easiest benchmarks we were able to decide if continue to run the algorithm for the other harder benchmarks or to stop the algorithm, if we were not getting any result (timeout reached, or full memory).

This strategy significantly reduced running time, as running each benchmark without producing any value would have taken more than 9000 minutes per algorithm.(3 minutes timeout for each benchmark file).

5 Algorithms implemented

5.1 Dynamic Programming

Description

Dynamic programming involves solving a problem in multiple stages by breaking it into smaller subproblems and using previously computed results to avoid useless calculations. For the Knapsack Problem, dynamic programming constructs a table that represents the optimal solution for subproblems of different capacities and numbers of items. The algorithm is based on the principle of optimality: the optimal solution to a problem of size n can be constructed from the optimal solutions of its subproblems. Specifically, for each stage $m > 1$, the algorithm calculates the values $f_m(\hat{c})$, where $\hat{c}$ represents the capacity (ranging from 0 to c). These values are computed using a classical recursion formula (Bellman, 1954, 1957; Dantzig, 1957) according to [5]:

$$f_m(\hat{c}) = \begin{cases} f_{m-1}(\hat{c}) & \text{if } w_m > \hat{c}, \\ \max(f_{m-1}(\hat{c}), v_m + f_{m-1}(\hat{c} - w_m)) & \text{if } w_m \leq \hat{c}. \end{cases}$$

Here:

- $f_m(\hat{c})$ represents the maximum value pbtained using the first m items and a knapsack capacity of $\hat{c}$,
- w_m and v_m are the weight and value of the m-th item,
- $f_{m-1}(\hat{c})$ is the maximum value obtained without including the m-th item,
- $v_m + f_{m-1}(\hat{c} - w_m)$ is the maximum value obtained by including the m-th item.

The algorithm starts with a base corresponding to "no items are included" ($f_0(\hat{c}) = 0$ for all capacities $\hat{c}$), and iteratively fills the table until the solution for all items and the full capacity is computed.
I implemented this algorithm iteratively, as opposed to the traditional recursive approach, based on the explanation in the KP book. The iterative method involves filling a 2D table, row by row, where each row corresponds to an item, and each column corresponds to a knapsack capacity.

Pseudocode

```
Function knapSackRec(W, wt[], val[], index, dp):
    If index < 0:
        Return 0
    If dp[index][W] is not -1:
        Return dp[index][W]
    If wt[index] > W:
        dp[index][W] = knapSackRec(W, wt, val, index - 1, dp)
    Else:
        include = val[index] + knapSackRec(W - wt[index], wt, val, index - 1, dp)
        exclude = knapSackRec(W, wt, val, index - 1, dp)
        dp[index][W] = max(include, exclude)
    Return dp[index][W]

Function knapSack_dynamic(W, wt[], val[], n):
    Initialize dp as a 2D array of size [n][W+1], filled with -1
    Return knapSackRec(W, wt, val, n - 1, dp)
```

Time Complexity Analysis

The time complexity of the recursive procedure KnapsackRec is $O(s)$, where s represents the number of states explored. Since s is bounded by $\min(2^n - 1, c)$, the time complexity of the dynamic programming procedure DP2 is $O(\min(2^n, nc))$.
It is important to note that the procedure DP2 does not require any specific ordering of the items.

Pros and Cons

- Pros: Guarantees an optimal solution. Efficient for small to medium-sized inputs.
- Cons: Inefficient for large capacities cause of high space complexity and it uses huge amount of memory.

Algorithm Performance Results

Metric	Time Results (s)	Memory Results (KB)	Error Results (%)
Total	166.3890	21.83 GB	-
Mean	0.033278	4578	0
Median	0.005076	2304	0
Standard Deviation	0.054489	4523	0
Minimum	0.000012	2176	0
Maximum	0.312257	25088	0
25th Percentile	0.000386	-	0
75th Percentile	0.041565	-	0

Table 2. Performance Statistics for Dynamic Programming Algorithm

5.2 Greedy Algorithm

Description

The greedy algorithm selects items based on their value-to-weight ratio in descending order according to [1]. It fills the knapsack until it is full or no more items can be added. The implementation also ensures the sorting step is clearly defined.

Pseudocode

```
1. Define an array of structures for items, including profit, weight, and ratio.
2. Calculate value-to-weight ratio for each item.
3. Sort items by ratio in descending order using a comparator function.
4. Initialize total_value = 0 and current_capacity = W (knapsack capacity).
5. For each item (in sorted order):
       If weight <= current_capacity:
           Add item to knapsack.
           Update total_value and current_capacity.
6. Return total_value.
```

Time and Space Complexity

- Time Complexity: $O(n \log n)$ in this implementation (since `qsort` C function was used for sorting).
- Space Complexity: $O(n)$ (for the arrays to store profits, weights, and ratios).

Performance Analysis

The performance of the greedy algorithm is influenced by the structure of the problem:

- The solution value z' is often close to the optimal solution z, but it may fail in edge cases where high-profit items are skipped.

- Sorting by value-to-weight ratio ensures that items with higher efficiency are prioritized.

Pros and Cons

- Pros: Simple to implement and efficient for larger input sizes with $O(n \log n)$ sorting.
- Cons: Does not guarantee an optimal solution for the 0/1 Knapsack Problem.

Curiosity

It was decided to implement two versions of the greedy algorithm to observe the impact of sorting complexity on performance. The $O(n^2)$ implementation took **793 seconds**, while the $O(n \log n)$ implementation took **6 seconds** to run the same set of 50000 generated benchmarks, highlighting the efficiency improvement with the optimized sorting approach.

Algorithm Performance Results

Metric	Time Results (s)	Memory Results (KB)	Error Results (%)
Total	5.101102	10.37 GB	-
Mean	0.001020	2176	0.54
Median	0.000127	2176	0.011
Standard Deviation	0.001905	2.56	1.42
Minimum	0.000005	2048	0
Maximum	0.009781	2176	23.27
25th Percentile	0.000025	-	0
75th Percentile	0.000774	-	0.34258

Table 3. Performance Statistics for Greedy Algorithm

5.3 Branch and Bound

Description

Branch and Bound explores all combinations of items in a depth-first (DFS) way according to [6], pruning unlikely branches based on an upper bound. At each step, the algorithm decides whether to include or exclude an item, and it computes a fractional *bound* (using the ratio-sorted items) to determine if further exploration could bring to a better solution than the current one.

Pseudocode (Depth-First)

```
1. Sort items in descending order of value to weight ratio.
2. Initialize global bestProfit = 0.
3. Define a recursive function branchAndBound(level, currentWeight, currentValue):
     3.1. If timed out or level == n:
              update bestProfit if currentValue is better (and weight <= capacity).
              return.
     3.2. Compute fractional upper bound for items from "level" onward.
          If bound <= bestProfit, prune (return).
     3.3. Include the item at position "level" (if it fits):
              newWeight = currentWeight + item[level].weight
              newValue  = currentValue + item[level].value
              branchAndBound(level + 1, newWeight, newValue)
     3.4. Exclude the item at position "level":
              branchAndBound(level + 1, currentWeight, currentValue)
4. Call branchAndBound(0, 0, 0) to begin from level=0 with empty knapsack.
5. Return bestProfit at the end.
```

Time and Space Complexity

- **Time Complexity:** Exponential in the worst case ($O(2^n)$). Pruning typically reduces the average runtime significantly.
- **Space Complexity:** $O(n)$ for the depth-first recursion stack and item storage.

Performance Analysis

The depth-first Branch and Bound variant ensures that when it expands a node, it computes a fractional bound to prune unpractical branches. It is guaranteed to find the optimal solution (matching dynamic programming) if it does not time out. However, for large n, the branching factor may lead to high runtime because of pruning.

- **Pros:** Guarantees optimality; can prune large portions of the search space.
- **Cons:** Potentially expensive for large problem sizes (exponential in worst-case).

Implementation Details

- Items are first sorted in descending order by $\frac{\text{value}}{\text{weight}}$.
- A recursive `branchAndBound` function explores including or excluding each item.
- The `computeBound` function calculates the fractional bound from the current state to prune nodes when no better solution is possible.
- A global `bestProfit` tracks the current optimal.
- Timeouts are handled by checking a global `timeout_flag` set by a timer signal.

Algorithm Performance Results

Metric	Time Results (s)	Memory Results (KB)	Error Results (%)
Total	0.362834	10.38 GB	-
Mean	0.000072	2177	0
Median	0.000034	2177	0
Standard Deviation	0.000107	14	0
Minimum	0.000001	2048	0
Maximum	0.001034	2816	0
25th Percentile	0.000001	-	0
75th Percentile	0.000087	-	0

Table 4. Performance Statistics for Branch and Bound Algorithm

5.4 Martello-Toth Algorithm

Description

The Martello-Toth algorithm is a variant of the Branch and Bound algorithm designed for solving the 0/1 Knapsack Problem according to [7]. It uses a breadth-first search (BFS) strategy with a priority queue for handling nodes. The algorithm calculates upper bounds using fractional knapsack solutions and prunes branches that cannot bring to a better solution than the current one.

Key features of the algorithm include:

- Presort items by their value to weight ratio to improve efficiency.
- Use of a queue-based implementation for managing nodes during the branching process.
- Tight bounds use fractional knapsack solutions to prune suboptimal branches.

The algorithm differs from a standard Branch and Bound approach by making use of implemented queue management and incorporating specific optimizations for the knapsack problem.

Pseudocode

```
1. Input: Number of items (n), item values and weights, knapsack capacity (W).
2. Preprocess:
   a. Calculate value-to-weight ratios for items.
   b. Sort items by descending ratio.
3. Initialize:
   a. Create a root node with level = -1, profit = 0, weight = 0, bound = upper bound.
   b. Enqueue the root node into a priority queue.
   c. Set maxProfit = 0.
4. While the queue is not empty:
   a. Dequeue a node u.
   b. If u’s level is -1, set v.level = 0; otherwise, set v.level = u.level + 1.
   c. For the next item:
      i. Include the item in v if it fits.
         - Update v’s weight and profit.
         - If v’s profit > maxProfit, update maxProfit.
         - Calculate v’s bound. If bound > maxProfit, enqueue v.
      ii. Exclude the item from v.
         - Keep u’s weight and profit.
         - Calculate bound for v. If bound > maxProfit, enqueue v.
5. Output: maxProfit.
```

Time and Space Complexity

- Time Complexity: $O(2^n)$ in the worst case, but effective pruning reduces the number of explored nodes significantly.
- Space Complexity: $O(n)$ for storing item data and the priority queue.

Pros:

- Guarantees an optimal solution.
- Aggressive pruning reduces computation time compared to basic Branch and Bound.
- Efficient memory management since we use a priority queue.

Cons:

- Computationally expensive for very large problem sizes due to exponential growth in the search space.
- Requires sorting of items, which adds an $O(n \log n)$ overhead.

Differences from Standard Branch and Bound

- **Search Strategy:** The new Branch and Bound uses a depth-first search recursion, while Martello–Toth implements more complex branching rules and may include heuristics or transformations before the main search.
- **Preprocessing and Reductions:** Martello-Toth includes preprocessing steps (like fixing certain items in or out of the solution) and problem specific reductions that further cut down the search space before or during the bounding process.
- **Bound Tightening:** Both approaches use fractional knapsack solutions to compute upper bounds, but Martello-Toth typically refines these bounds with additional heuristics and item ordering strategies, achieving higher aggressive pruning.
- **Efficiency:** Martello–Toth tends to outperform a normal depth-first B&B on larger instances, thanks to its stronger pruning and specialized data structures. However, the basic depth-first method remains simpler to implement and guarantees optimal solutions for 0/1 Knapsack if given enough time.

Algorithm Performance Results

Metric	Time Results (s)	Memory Results (KB)	Error Results (%)
Total	1.456095	10.47 GB	-
Mean	0.000291	2196	0
Median	0.000062	2176	0
Standard Deviation	0.000502	61.36	0
Minimum	0.000001	2048	0
Maximum	0.004166	2392	0
25th Percentile	0.000012	-	0
75th Percentile	0.000309	-	0

Table 5. Performance Statistics for MT Algorithm

5.5 Simulated Annealing

Description

Simulated Annealing uses a probabilistic approach to explore the solution space according to [8]. It accepts sub-optimal solutions to escape local optima and converge to the global optimum. The algorithm starts with an initial solution and a high temperature, which gradually decreases according to a cooling schedule. At each iteration, a neighboring solution is generated by flipping a randomly chosen item's inclusion in the knapsack. If the new solution improves the profit, it gets accepted, otherwise it is accepted with a probability determined by the change in profit and the current temperature. This way helps explore the solution space. The process continues until the temperature goes below a chosen minimum.
The algorithm balances exploration and exploitation by allowing worse solutions to be accepted early on, when the temperature is high, and focusing on better solutions when the temperature decreases.

Pseudocode

```
1. Initialize the solution using a greedy heuristic and temperature T.
2. While T > stopping_temperature:
       For a fixed number of neighbors:
           Generate a neighbor solution by flipping one item's inclusion.
           Calculate the change in profit (P).
           If P > 0 or exp(P / T) > random(0, 1):
               Accept the neighbor solution as the current solution.
               Update the best solution if the current solution improves it.
       Decrease temperature using a cooling rate.
3. Return the best solution found.
```

Time and Space Complexity

- Time Complexity: $O(n \cdot \text{neighbor\_count} \cdot \log(\text{INITIAL\_TEMPERATURE}/\text{MIN\_TEMPERATURE}))$, where n is the number of items.
- Space Complexity: $O(n)$ (for the solution arrays).

Pros:

- Effective for escaping local optima.
- Goof for large and complex search spaces.
- Easy to implement with flexible parameter tuning.

Cons:

- Computationally expensive due to the iterative process.
- Convergence depends heavily on the cooling schedule and parameters like temperature, neighbor count, and the stopping criteria.

Algorithm Performance Results

Metric	Time Results (s)	Memory Results (KB)	Error Results (%)
Total	1037.043	10.75 GB	-
Mean	0.207409	2255	0.299
Median	0.061017	2176	0
Standard Deviation	0.310660	107	0.77
Minimum	0.002504	2176	0
Maximum	1.182992	2560	13.837
25th Percentile	0.015228	-	0
75th Percentile	0.279299	-	0.218

Table 6. Performance Statistics for Simulated Annealing Algorithm

5.6 Ant Colony Optimization

Description

Ant Colony Optimization (ACO) is inspired by the searching food behavior of ants. In the context of the 0/1 Knapsack Problem, ACO employs virtual pheromones to guide the selection of items into the knapsack according to [9]. The algorithm initializes pheromone levels for all items, and a group of "ants" iteratively builds solutions by probabilistically selecting items based on the pheromone intensity and heuristic information (e.g., value to weight ratio). After evaluating the solutions, the pheromones get updated: pheromone levels of items in good solutions are increased, while pheromone levels of others evaporate over time. This process balances exploration and exploitation, enabling the algorithm to converge toward good solutions over several iterations.

The key steps of ACO for the knapsack problem include:

- extbfInitialization: Pheromone levels and heuristic information (e.g., value-to-weight ratio) are initialized. Parameters like the number of ants, iterations, and evaporation rate are adjusted based on the problem size.
- extbfSolution Construction: Each ant constructs a solution by selecting items probabilistically based on a combination of pheromone intensity and heuristic value. The probability of selecting an item i is given by:

$$P(\text{choose item } i) = \frac{(\text{pheromone}[i])^{\alpha} \cdot (\text{heuristic}[i])^{\beta}}{\sum_j (\text{pheromone}[j])^{\alpha} \cdot (\text{heuristic}[j])^{\beta}}$$

  Here:

  - pheromone[i]: The pheromone intensity associated with item i.
  - heuristic[i]: The heuristic value of item i (e.g., value-to-weight ratio).
  - α: The importance of pheromone intensity.
  - β: The importance of heuristic information.
- extbfEvaluation: Each ant's solution is evaluated based on its total profit and weight. Solutions exceeding the capacity are penalized by setting their fitness to zero.
- extbfPheromone Update: Pheromone levels are updated as follows:
  - extbfEvaporation: Pheromones are reduced by a fixed rate to diminish the influence of past solutions.
  - extbfReinforcement: Pheromones are increased for items included in the best solutions, proportional to their contribution to the solution quality.
- extbfConvergence: The best solution is tracked, and the algorithm terminates when the stopping criterion (e.g., maximum iterations or timeout) is met.

Pseudocode

```
1. Initialize parameters (number of ants, iterations, alpha, beta, evaporation rate, etc.).
2. Initialize pheromone levels and heuristic information.
3. While stopping condition not met:
       For each ant:
           a. Construct a solution probabilistically using:
                  P(choose item i) = (pheromone[i]^alpha * heuristic[i]^beta) / sum for all items.
           b. Evaluate the solution's fitness:
                  - Calculate total profit and weight.
                  - Penalize solutions exceeding capacity by setting fitness to 0.
       c. Update pheromone levels:
           - Evaporate pheromones:
                  pheromone[i] *= (1 - evaporation_rate).
           - Reinforce pheromones for items in the best solutions:
                  pheromone[i] += Q * (best_fitness / capacity).
4. Return the best solution found.
```

Time and Space Complexity

- Time Complexity: $O(A \cdot I \cdot n)$, A is the number of ants, I is the number of iterations and n is the number of items. - Space Complexity: $O(n)$ for pheromone levels.

Pros:

- Effective for exploring large solution spaces.
- Can escape local optima through probabilistic selection and pheromone updates.
- Well-suited for combinatorial optimization problems.

Cons:

- Convergence depends mainly on parameter tuning (e.g., evaporation rate, number of ants).
- Slower than simpler heuristics due to its iterative nature and relies on multiple agents.

Algorithm Performance Results

Metric	**Time Results (s)**	**Memory Results (KB)**	**Error Results (%)**
Total	1458.873	11.87 GB	-
Mean	0.291774	2488	68
Median	0.015248	2304	75
Standard Deviation	0.603989	517	21
Minimum	0.000306	2176	0
Maximum	2.237581	4076	100
25th Percentile	0.003044	-	55
75th Percentile	0.163514	-	85

Table 7. Performance Statistics for Ant Colony Optimization Algorithm

5.7 Integer Linear Programming (ILP)

Description

The Knapsack Problem can be formulated as an Integer Linear Programming (ILP) problem and solved using an ILP solver such as GLPK according to [2]. The objective function is to maximize the total profit under the constraint that the total weight does not exceed the knapsack's capacity. The decision variables are binary $(0/1)$, indicating whether each item is included in the knapsack. The solver uses optimization techniques to find the exact solution by exploring the feasible region defined by the constraints.

The process includes:

- Formulating the objective function to maximize the total profit.
- Defining constraints to ensure the total weight is within the knapsack's capacity and variables are binary.
- Using an ILP solver to optimize the objective function.
- Returning the optimal solution.

Pseudocode

```
1. Read the number of items (n), weights, values, and knapsack capacity from input.
2. Create a GLPK problem instance and set it to maximize the objective function.
3. Add one row (constraint):
       a. Total weight of selected items <= capacity.
4. Add n columns (decision variables):
       a. Set bounds for each variable as 0 <= x[i] <= 1.
       b. Set the variable type to binary (0/1).
       c. Set the profit of each item as the objective coefficient.
5. Define the constraint matrix:
       a. Each item's weight contributes to the weight constraint.
6. Load the constraint matrix into the GLPK problem.
7. Run the simplex method to solve the relaxed problem.
8. Run the integer optimizer to solve the ILP.
9. If no timeout occurs, retrieve the optimal solution and profit.
10. Return the best solution and profit.
```

Time and Space Complexity

- Time Complexity: Depends on the solver and problem size but is typically exponential in the worst case due to the combinatorial nature of ILP.
- Space Complexity: Depends on the solver's implementation and the size of the problem.

Pros:

- Guarantees an optimal solution.
- Suitable for small to medium instances.
- Can handle complex constraints in an efficient way.

Cons:

- Computationally expensive for large instances.
- Requires a specialized solver such as GLPK.

Algorithm Performance Results

Metric	Time Results (s)	Memory Results (KB)	Error Results (%)
Total	68.26624	19.36 GB	-
Mean	0.013653	4060	0
Median	0.001806	3200	0
Standard Deviation	0.029941	1633	0
Minimum	0.000177	2944	0
Maximum	0.534275	10388	0
25th Percentile	0.000469	-	0
75th Percentile	0.012177	-	0

Table 8. Performance Statistics for Integer Linear Programming Algorithm

6 Data Analysis

The data analysis for this project involved the collection and computation of different metrics for multiple algorithms used to solve the Knapsack Problem. The objective was to compare these algorithms based on their performance metrics such as execution time, memory usage, and error percentage with respect to the optimal solution obtained by running the Dynamic Programming algorithm.
The results for each algorithm were saved in different files, metrics such as time (in seconds), memory usage (in kilobytes), and the best solution found. These files were read into separate data frames for processing. Columns were renamed across all data frames then, irrelevant columns were dropped such as the `Algorithm` column. All the data frames were merged based on common columns such as `Folder`, `Size`, and `Capacity`. For algorithms other than Dynamic Programming, the error percentage was calculated using the following formula:

$$\text{Error \%} = 100 \cdot \frac{\text{DP\_BestValue} - \text{Algorithm\_BestValue}}{\text{DP\_BestValue}}$$

A special case was handled where `DP_BestValue` was zero, ensuring no division by zero.

Extended Statistics: For each algorithm, the following metrics were computed:

- **Mean and Median:** The mean (average) provides an overall measure of performance of all problem instances, while the median highlights the middle value.
- **Standard Deviation:** Measures how the data spreads around the mean. A smaller standard deviation indicates that the algorithm's performance is more consistent, while a larger standard deviation highlights variability. This metric is important for detecting how stable the algorithms are on different problem instances.
- **Minimum and Maximum:** Provide the range of performance metrics, helping identify the best and worst-case scenarios for each algorithm.
- **25th Percentile (Q1):** Represents the value below which 25% of the data falls. This is particularly useful to understand how the algorithm behaves in smaller or simpler instances.
- **75th Percentile (Q3):** Represents the value below which 75% of the data falls. This helps to evaluate the typical performance for the majority of instances, excluding edge cases.
- **Total Metrics:** The total execution time and memory usage on all the instances were computed for each algorithm to provide an clear measure of resource usage.

These statistical measures were computed for execution time, memory usage, and error percentage.
In the end we displayed the distribution of execution times for each algorithm and showed the relationship between problem size (number of items) and execution time for each algorithm. Correlation coefficients were computed between problem size and execution time for each algorithm. If the coefficient approaches 1, it means that the algorithm's execution time increases as the problem size increases.

7 Scatter Plots: Algorithm vs. Problem Size

The following scatter plots illustrate the relationship between the problem size (number of items) and the execution time for each algorithm. These visualizations provide insights into how the algorithms scale as the size of the problem increases.

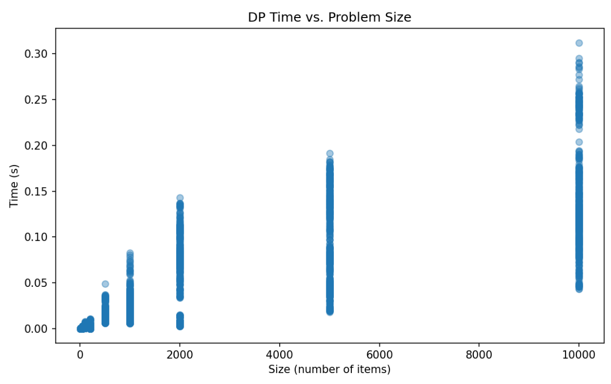

Figure 1. DP: Time vs. Problem Size

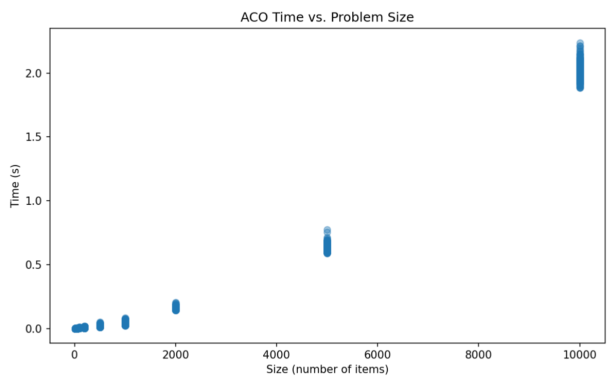

Figure 2. ACO: Time vs. Problem Size

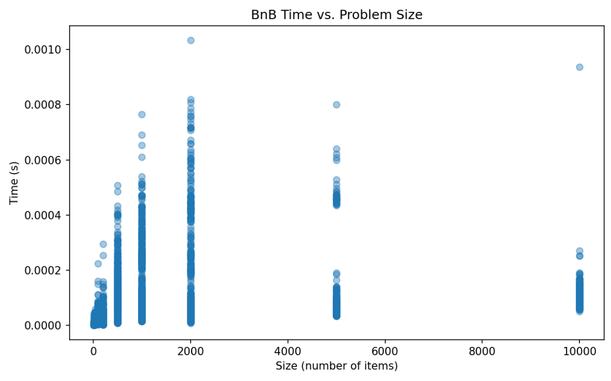

Figure 3. BnB: Time vs. Problem Size

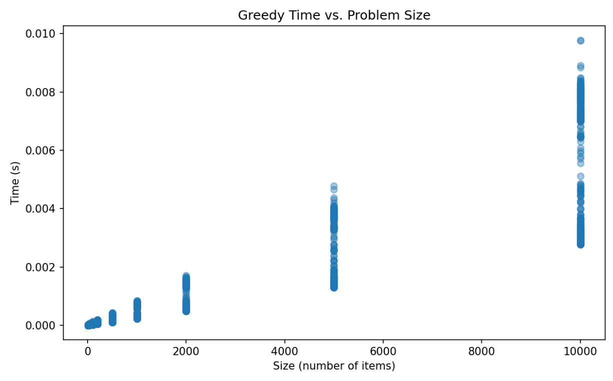

Figure 4. Greedy: Time vs. Problem Size

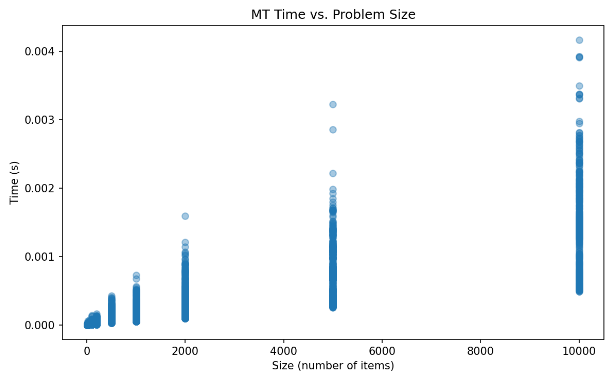

Figure 5. MT: Time vs. Problem Size

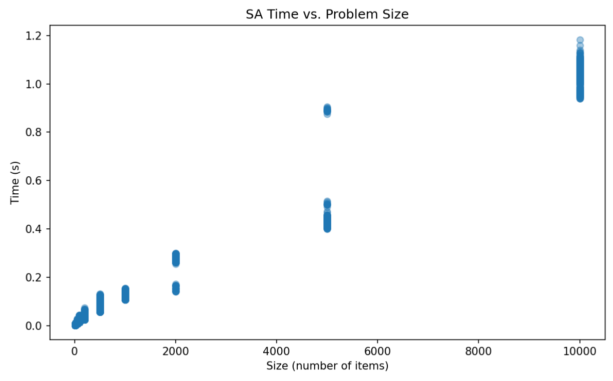

Figure 6. SA: Time vs. Problem Size

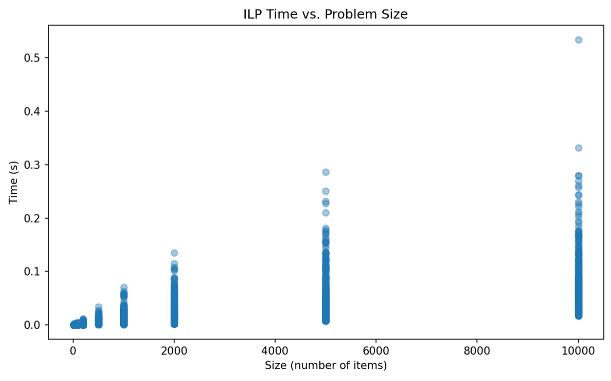

Figure 7. ILP: Time vs. Problem Size

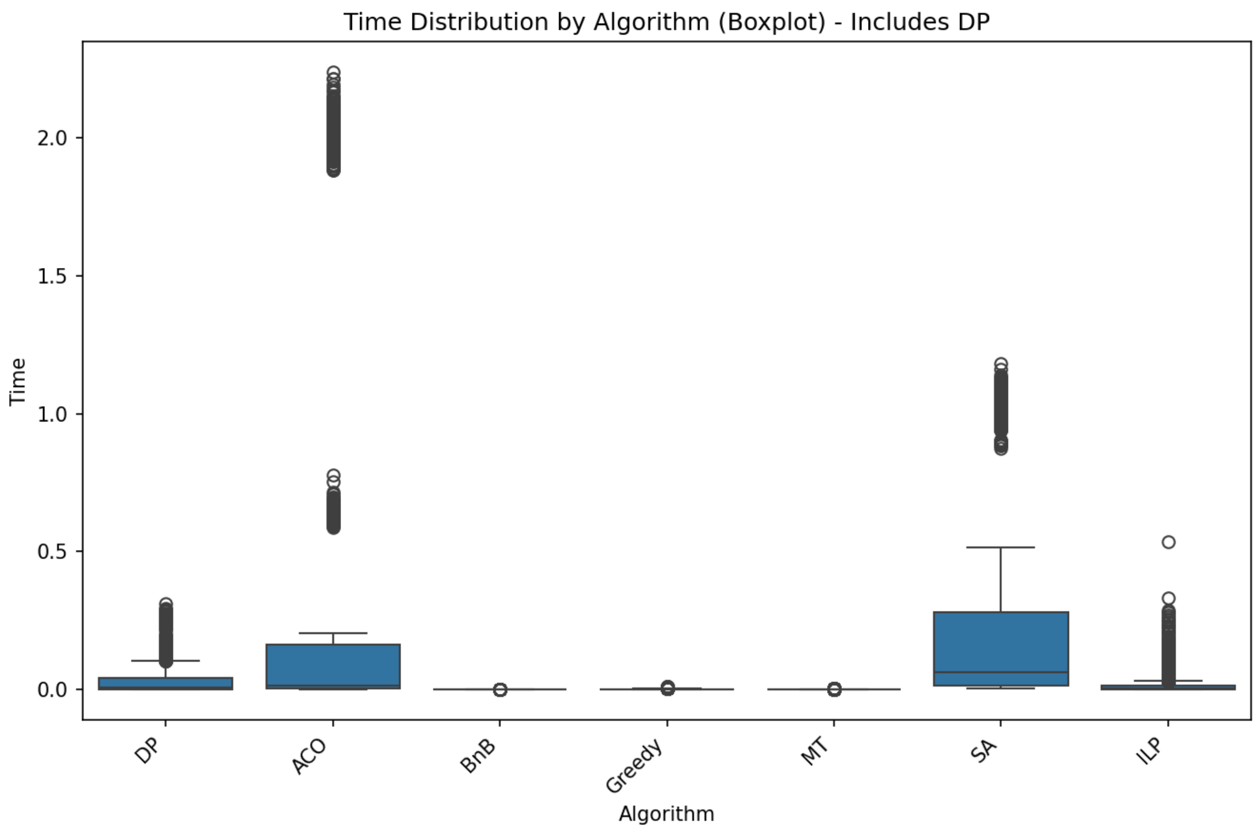

Figure 8. Time distribution by algorithm

Algorithm	Correlation (Size vs. Time)
DP	0.8432
ACO	0.9799
BnB	0.2250
Greedy	0.9055
MT	0.8616
SA	0.9897
ILP	0.6522

Table 9. Correlation Between Problem Size and Execution Time for Each Algorithm

Data analysis

As we can see from the plots above and the table representing the correlation, most of the algorithms got affected by the problem size, so the execution time is strictly related to the problem size. Only 2 algorithms do not follow this trend, BnB actually performed better when the size of the benchmarks was 10,000, but is evident that the execution time increases a lot faster than the other algorithms, when the size is between 10 to 2000. The fact that BnB performed better when the size got near 10,000 is something really strange, because as we know BnB should take more time as the size of the benchmarks increase, but this didn't happen in this case. On the other hand dinamyc programming should finish the memory as the capacity increases. ILP increased slower with problem size, showing a more relaxed increasing trend after the problem size got bigger than 5000. ACO and Simulated annealing algorithms showed a more compact time taken for each problem size, which is really unique compared to the other ones. As it can be seen from the results of ACO, it performed really bad with a mean error of 68%, this can be due to the implementation of the algorithm, we still decided to include the results obtained, but we want to underline the fact that such bad performance could be affected by the incorrect way of implementing the algorithm, even if a dynamic parameter tuning was implemented.
Dynamic Programming (DP) and Integer Linear Programming (ILP) are observed to use significantly more memory compared to algorithms like Ant Colony Optimization (ACO), Branch and Bound (BnB), and Simulated Annealing (SA). This higher memory consumption is linked to the characteristics of these algorithms.

- **Dynamic Programming (DP)**: DP uses memoization to store results of computations, requiring a multidimensional array with dimensions that are proportional to the problem size (e.g., item count and capacity in knapsack problems). This approach brings to a space complexity of $O(nW)$, which increases memory use significantly as n (number of items) and W (capacity) grow.
- **Integer Linear Programming (ILP)**: ILP involves creating large matrices to handle variables and constraits representing the problem. The size of these matrices and the overhead from ILP solvers, which use additional structures for the optimization processes, result in higher memory usage.

8 Online Benchmarks Results

Dynamic Programming Algorithm

Metric	Time Results (s)	Memory Results (KB)	Error Results (%)
Total	20691	3220 GB	-
Mean	19.176	3129328	0
Median	14.756	3127936	0
Standard Deviation	15.072	1105378	0
Minimum	1.815	1564416	0
Maximum	81.999	4691200	0
25th Percentile	6.531	-	0
75th Percentile	29.088	-	0

Table 10. Comprehensive Performance Statistics for Dynamic Programming Algorithm

The following describes the process taken to extract performance results from running the Dynamic Programming (DP) algorithm on benchmarks with smaller capacities. Initially, to identify the group of smaller capacities, we executed a command line in the terminal to list and categorize the problem instances based on their capacity. The command used was:

```
problemInstances$ ls -1 | cut -f 4 -d _ | sort -n | uniq -c
```

This command outputs the frequency of each unique capacity value, as shown below:

```
1079 1000000
1080 100000000
1079 10000000000
```

It is clear that the smallest capacity group is 1 million. Therefore, we started our analysis by focusing on benchmarks with this capacity. We created a new script to run the DP algorithm only on these benchmarks.

We need tom underline the memory consumption of running 1,100 benchmarks which used 3,220 GB of memory which is a very large number compared to the memory used on the created benchmarks.

After obtaining results from the 1 million capacity benchmarks, we tried to run the DP algorithm on the next size capacity (100 million). Unfortunately, in this case, the algorithm was not able to find any solutions and encountered issues related to full memory utilization.

Greedy Algorithm

Metric	Time Results (s)	Memory Results (KB)	Error Results (%)
Total	0.627559	6.18 GB	-
Mean	0.000211	2175.96	0.854
Median	0.000194	2176	0.0072
Standard Deviation	0.000101	2.347	2.048
Minimum	0.000082	2048	0
Maximum	0.001026	2176	11.32
25th Percentile	0.000146	-	0.00018
75th Percentile	0.000258	-	0.2515

Table 11. Comprehensive Performance Statistics for Greedy Algorithm

To run the greedy algorithm we just ran it on all the benchmarks in less time than the created benchmaks, this shows that dp is more affected by the size of the knapsack rather than the capacity. The algorithm actually showed a mean error close to 1% which is really good if we think that the reference research paper we took the benchmark from used 810 hours of cpu time on a super computer to found the optimum of each benchmark. The algorithm also used normal memory amount which is also really good.

Other Algorithms

None of the other algorithms managed to find any solution when tested on the benchmarks. The algorithms that are sensitive to the knapsack's capacity were specifically run on benchmarks with the lowest capacities, while those influenced by the problem size, like the Branch and Bound (BnB) algorithm, were tested on differently sized benchmarks.

Benchmark Size Analysis

To see all the sizes of the benchmarks, the following command was executed in the terminal:

```
problemInstances$ ls -1 | cut -f 2 -d _ | sort -n | uniq -c
```

This command provided a count of benchmarks categorized by their size:

```
647 400
648 600
647 800
648 1000
648 1200
```

The smallest benchmark size identified was 400. However, even with these considerations, no algorithm found a solution. When the BnB algorithm was ran to structured benchmarks of size 10000, it found a solution, showing that BnB's performance is influenced not only by the size but also by the capacity of the knapsack.
Below the correlation between problem size and time is shown.

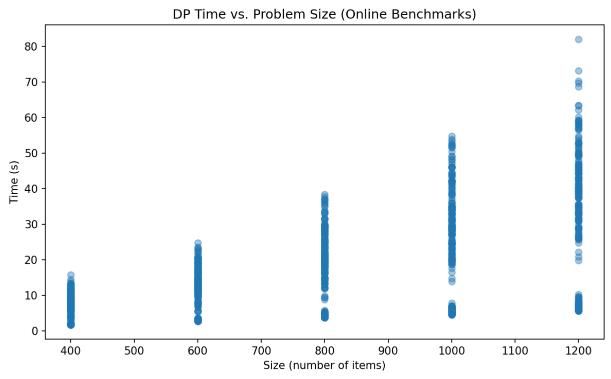

Figure 9. DP: Time vs. Problem Size(online)

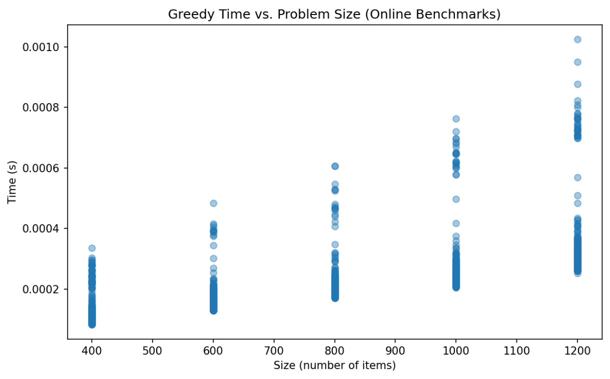

Figure 10. Greedy: Time vs. Problem Size(online)

As we can see the 2 algorithms show a similar increasing trend but the time used for each of the two algorithms is completely different.

References

[1] M. I. R. A. T. H. N. Devano Fernando Boes, Kenrick Panca Dewanto. Analyzing the most effective algorithm for knapsack problems. IEEE, 2024.

[2] R. S. Federico Della Croce, Fabio Salassa. An exact approach for the 0–1 knapsack problem with setups. In *Computers Operations Research*, pages 61–67,, 2017.

[3] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley-Interscience, 1990. Referenced on page 1.

[4] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley-Interscience, 1990. Referenced on page 13.

[5] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley-Interscience, 1990. Referenced on pages 36–38.

[6] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley-Interscience, 1990. Referenced on page 29-30.

[7] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley-Interscience, 1990. Referenced on page 32-36.

[8] N. Moradi, V. Kayvanfar, and M. Rafiee. An efficient population-based simulated annealing algorithm for 0–1 knapsack problem. *Engineering with Computers*, 38:2771–2790, 2022.

[9] X. Z. Peiyi Zhao, Peixin Zhao. A new ant colony optimization for the knapsack problem. In *2006 7th International Conference on Computer-Aided Industrial Design and Conceptual Design*, Hangzhou, China, Nov 2006. IEEE. CD ISBN: 1-4244-0684-6.