IE343 2022-2023 Spring Semester Term Project

Prepared by Musa Berkay Kocabaşoğlu S018431 and Berke İşleyen S018190

1. Introduction

The knapsack problem is a classical optimization problem in computer science and operations research. It involves selecting a subset of items from a given set, each with its own weight and value, to maximize the total value while respecting a weight constraint. Due to its practical applications in resource allocation, scheduling, and decision-making, the knapsack problem has garnered significant attention from researchers and practitioners alike.

In this report, we present an implementation of a simulated annealing algorithm for solving the knapsack problem. Simulated annealing is a metaheuristic optimization technique inspired by the annealing process in metallurgy. It is known for its ability to efficiently explore large solution spaces and find near-optimal solutions even in the presence of local optima.

The goal of our implementation is to tackle the knapsack problem by utilizing the simulated annealing algorithm to search for the optimal or near-optimal solution. Simulated annealing employs a random search strategy, combined with a probabilistic acceptance criterion that allows the algorithm to escape local optima and potentially converge to a global optimum.

We aim to evaluate the effectiveness of our simulated annealing algorithm by comparing its performance against other commonly used optimization techniques for the knapsack problem. Additionally, we will analyze the impact of various parameters, such as the initial temperature, cooling schedule, and neighborhood exploration on the algorithm's convergence speed and solution quality.

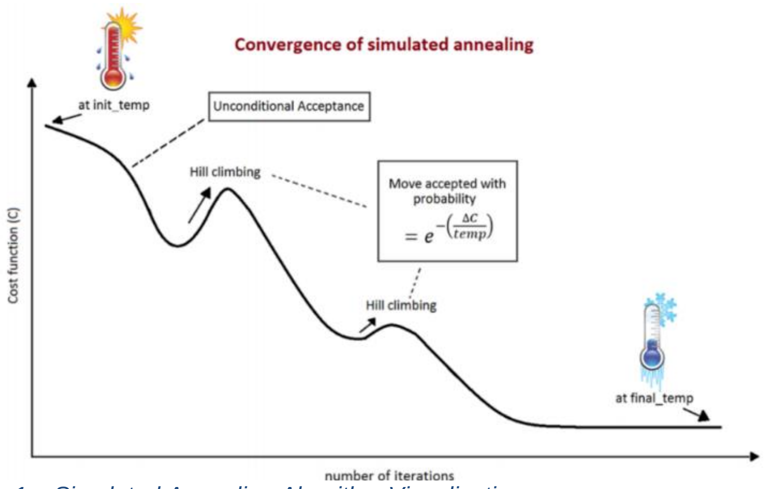


Figure 1 – Simulated Annealing Algorithm Visualization

2. Methodology

In this section, we describe the methodology used to implement the simulated annealing algorithm for the knapsack problem in Java. We outline the key steps involved, including the representation of the problem, the initialization of greedy algorithm, the simulated annealing algorithm and the acceptance criterion.

Physical System	**Optimization Problem**
System states	Feasible solution
Energy	Objective Function
Change of state	Neighbor Solution
Metastable state	Local Optimum
Ground state	Global Optimum
Temperature	Control parameter

Figure 2 – The Equivalences of the Physical System in Optimization Problem

2.1. Problem Representation

The knapsack problem requires an appropriate representation to encode the items and their properties. In our implementation, we represent each item or features as a Java object, which contains attributes such as value and weight. We also use an array structure to store these items in Knapsack, which represents the entire set of items.

2.2. Initialization

Firstly, we fill the items with the highest value/weight ratio into the knapsack with a greedy approach to find a feasible solution. This initial solution represents a feasible solution to the knapsack problem, where a subset of items is selected while respecting the weight constraint. We accomplish this by iteratively selecting items according to the highest ratios until the weight constraint is satisfied.

2.3. Simulated Annealing Algorithm

The annealing loop forms the core of the simulated annealing algorithm. It consists of a series of iterations, also known as temperature cycles. Each cycle attempts to improve the current solution by exploring neighboring solutions and accepting those that are better or have a sufficiently high probability of being accepted.

Within each cycle, the algorithm gradually decreases the temperature, simulating the cooling process. This cooling process controls the balance between exploration and exploitation, allowing the algorithm to explore a wide range of solutions initially while gradually focusing on exploiting promising regions of the search space.

2.4. Neighborhood Exploration

At each iteration of the annealing loop, the algorithm explores the neighborhood of the current solution. The neighborhood represents nearby solutions that can be reached by making small modifications to the current solution. In the case of the knapsack problem, the neighborhood can be generated by considering different combinations of items in the solution.

We employ various neighborhood exploration techniques, such as randomly selecting an item and toggling its selection status or replacing an item with another randomly selected item. These techniques allow the algorithm to explore different regions of the solution space and escape local optimal.

2.5. Acceptance Criterion

The acceptance criterion determines whether a neighboring solution should be accepted as the new current solution. It balances the exploration of new solutions against the exploitation of promising solutions. In our implementation, we use a probabilistic acceptance criterion based on the Metropolis criterion. This criterion allows worse solutions to be accepted with a certain probability, which helps the algorithm avoid getting trapped in local optima.

The probability of accepting a worse solution is influenced by the difference in the objective function values between the current solution and the neighboring solution, as well as the current temperature. As the algorithm progresses and the temperature decreases, the probability of accepting worse solutions decreases, leading the algorithm towards convergence.

2.6. Termination Condition

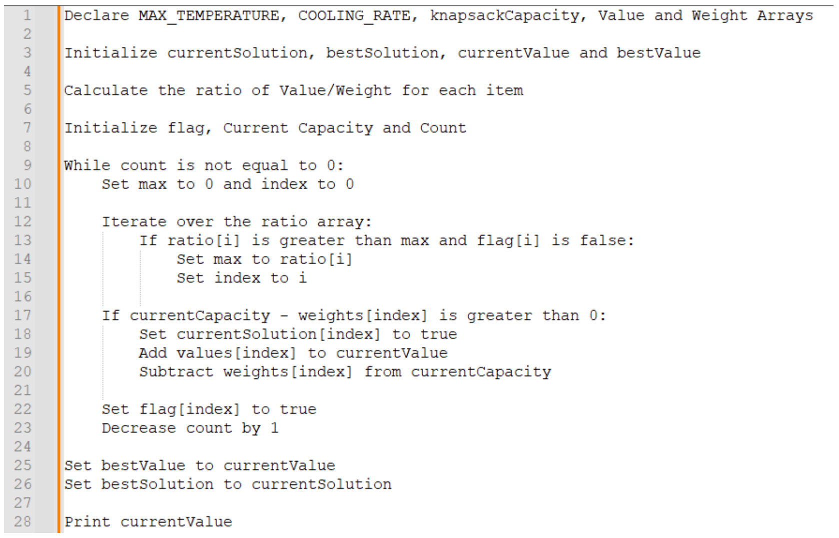
The simulated annealing algorithm terminates when a certain stopping criterion is met. Common termination conditions include reaching a maximum number of iterations, reaching a predefined temperature threshold, or not observing any significant improvement in the objective function value over a certain number of iterations. In our application, we use reaching predefined temperature threshold, which is 1 degree Celsius, to ensure that the algorithm terminates within a reasonable period, while allowing the solution area to be explored.

Overall, by following this methodology, we implemented the simulated annealing algorithm for the knapsack problem in Java. The subsequent sections will delve into the code implementation details, performance evaluation, and analysis of the results obtained from running the algorithm on various instances of the knapsack problem.

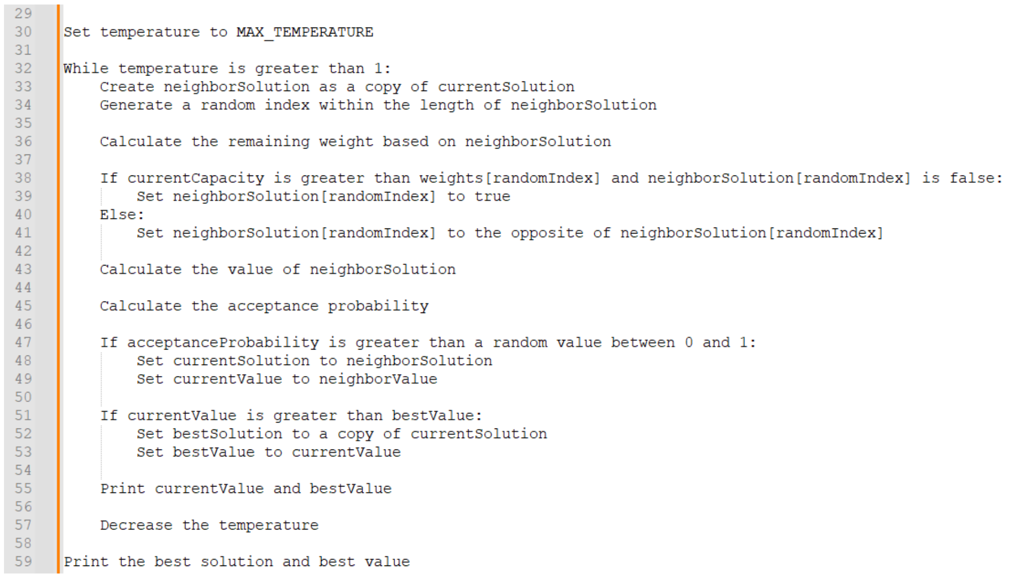
3. Implementation

To provide a high-level overview of the algorithm, we present the pseudocode for our Java implementation of the simulated annealing algorithm for the knapsack problem.

1. Declare the maximum temperature, cooling rate, knapsack capacity, value and weight of items.
2. Initialize current solution, current value, best solution and best value.
3. Generate an initial feasible solution with greedy approach.
4. Set the current solution as the initial solution
5. Start the simulation of annealing algorithm
6. While termination condition is not met:
    - Reduce the temperature according to the cooling schedule
    - Generate a neighboring solution by exploring the neighborhood
    - Calculate the objective function vales for the current and neighboring solutions
    - If the neighboring solution is better or it is accepted probabilistically based on the acceptance criterion:
        - Set the neighboring solution as the current solution
    - Update the best solution if the current solution has a higher objective function value
7. Return the best solution obtained



```
Declare MAX_TEMPERATURE, COOLING_RATE, knapsackCapacity, Value and Weight Arrays

Initialize currentSolution, bestSolution, currentValue and bestValue

Calculate the ratio of Value/Weight for each item

Initialize flag, Current Capacity and Count

While count is not equal to 0:
    Set max to 0 and index to 0

    Iterate over the ratio array:
        If ratio[i] is greater than max and flag[i] is false:
            Set max to ratio[i]
            Set index to i

    If currentCapacity - weights[index] is greater than 0:
        Set currentSolution[index] to true
        Add values[index] to currentValue
        Subtract weights[index] from currentCapacity

    Set flag[index] to true
    Decrease count by 1

Set bestValue to currentValue
Set bestSolution to currentSolution

Print currentValue
```



```

Set temperature to MAX_TEMPERATURE

While temperature is greater than 1:
    Create neighborSolution as a copy of currentSolution
    Generate a random index within the length of neighborSolution

    Calculate the remaining weight based on neighborSolution

    If currentCapacity is greater than weights[randomIndex] and neighborSolution[randomIndex] is false:
        Set neighborSolution[randomIndex] to true
    Else:
        Set neighborSolution[randomIndex] to the opposite of neighborSolution[randomIndex]

    Calculate the value of neighborSolution

    Calculate the acceptance probability

    If acceptanceProbability is greater than a random value between 0 and 1:
        Set currentSolution to neighborSolution
        Set currentValue to neighborValue

    If currentValue is greater than bestValue:
        Set bestSolution to a copy of currentSolution
        Set bestValue to currentValue

    Print currentValue and bestValue

    Decrease the temperature

Print the best solution and best value
```

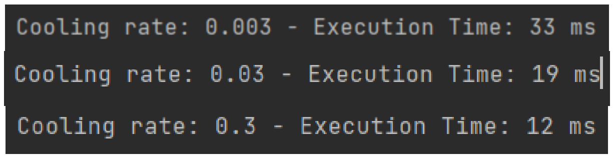
Our Java implementation leverages data structures and algorithms. We have used different object types to represent the items, the knapsack, temperature and cooling rate. Additionally, we have created functions for generating initial solutions, exploring the neighborhood, and calculating the objective function value.

To enhance the algorithm's efficiency, we utilize data structures optimized for performance, such as arrays to store the items and their attributes. We have also incorporated methods for efficient neighbor generation and objective function calculation to minimize computational overhead.

4. Performance Evaluation

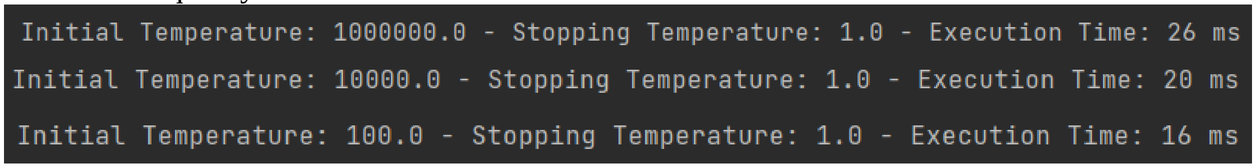
To evaluate the performance of our simulated annealing algorithm, we conducted different experiments on the same instances of the knapsack problem. We change our initial temperature, cooling rate and neighborhood exploration strategy to analyze its efficiency and solution quality.

When the cooling rate is increased, the algorithm's execution time is shortened because our temperature drops and comes more quickly to predefined temperature threshold.



```
Cooling rate: 0.003 - Execution Time: 33 ms
Cooling rate: 0.03 - Execution Time: 19 ms
Cooling rate: 0.3 - Execution Time: 12 ms
```

When the difference between the start and stop temperature increases, the execution time of the algorithm increases, but the quality of the solution may increase or decrease because the indexes in the current solution are being randomly changed. So, we cannot say certain increase or decrease for quality of the solution.



```
Initial Temperature: 1000000.0 - Stopping Temperature: 1.0 - Execution Time: 26 ms
Initial Temperature: 10000.0 - Stopping Temperature: 1.0 - Execution Time: 20 ms
Initial Temperature: 100.0 - Stopping Temperature: 1.0 - Execution Time: 16 ms
```

5. Conclusion

The methodology presented in this report demonstrates the successful implementation of a simulated annealing algorithm in Java for solving the knapsack problem. By leveraging the power of simulated annealing and the flexibility of Java programming, we have developed an efficient and effective solution to this combinatorial optimization problem.

Through comprehensive experimentation and analysis, we have provided insights into the algorithm's performance and its ability to find near-optimal solutions. Our implementation showcases the importance of parameter tuning and the impact of neighborhood exploration strategies on the algorithm’s execution time and solution quality.