VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MATHEMATICAL MODELING (CO2011)

Assignment

"Cutting stock problem"

Team 3

Instructor(s): Mai Xuân Toàn

No.	Member	ID	Effort
1	Nguyễn Thanh Phú	2014140	35%
2	Trần Tiến Khải	2211560	5%
3	Lê Phú Cường	2210425	55%
4	Trần Đình Quang	2011901	0%
5	Ngô Anh Hoàn	2311022	5%

Table of Contents

Acknowledge

∗ *I would like to express my sincere gratitude to* ***Mai Xuân Toàn****, my advisor, for their continuous support, patience, and invaluable guidance throughout the course of this project.*
∗ *I also wish to thank my team member, whose expertise and suggestions were instrumental in shaping the direction and outcome of this research.*

1 Introduction

The Cut Stock Problem (CSP) is a significant optimization challenge within the domains of Operations Research and Industrial Engineering. It involves determining the most efficient way to cut a given large stock length into smaller required lengths, minimizing waste while fulfilling demand exactly. This problem is prevalent in industries such as paper, steel, and textiles, where materials are produced in standard sizes but must be cut to meet specific customer requirements.The complexity of the CSP increases with the diversity of order sizes and the need to satisfy all demands precisely. A critical issue faced by companies across these industries is how to efficiently utilize material resources, which directly impacts both operational costs and sustainability efforts. By minimizing material waste, companies can reduce production costs, improve profitability, and contribute to environmental sustainability. Over the years, various solution methodologies have been developed to tackle the CSP. These include dynamic programming, which optimizes decisions based on stages and states; column generation, which iteratively adds variables to the problem to improve the solution; and heuristic methods, which provide practical, often near-optimal solutions in a shorter time. Each of these approaches offers unique advantages and is suited to different scenarios depending on problem size, complexity, and industry-specific constraints.

Steel Industry:
In the steel industry, manufacturers often deal with high costs and environmental impacts due to material waste. A steel plant might produce large steel rolls that need to be cut into various lengths to meet customer specifications for different applications. Here, the one-dimensional CSP is applied to minimize leftover material, optimize production efficiency, and reduce costs. By implementing algorithms like column generation, manufacturers can determine the optimal cutting patterns to fulfill demand while minimizing waste. This leads to significant cost savings and enhances sustainability by reducing the amount of unused steel.

Paper Industry:
Paper mills face similar challenges as they cut large rolls of paper into smaller sizes for products such as notebooks, brochures, and packaging materials. The CSP in this context involves maximizing the yield from each paper roll while ensuring that all customer orders are met. By employing dynamic programming techniques, paper manufacturers can calculate the best way to slice rolls into the desired dimensions, thereby minimizing waste and improving operational efficiency. This process not only optimizes material usage but also helps the company reduce its environmental footprint.

Textile Industry:
In the textile industry, large fabric rolls are cut into various patterns to create garments and other products. Here, the CSP involves determining the most efficient layout of cuts to produce different sizes and shapes required for clothing production. Heuristic methods are often employed in this context, providing practical solutions that are quick and sufficiently close to optimal. By reducing fabric waste, textile companies can lower their production costs and improve their sustainability efforts by minimizing the environmental impact associated with textile manufacturing.

Aims:
This report aims to provide a comprehensive analysis of the CSP, starting with a literature review that highlights its historical context, industrial significance, and mathematical modeling approaches. It will explore different algorithms and heuristics developed for solving the CSP,

examining their strengths and weaknesses. The report will also feature case studies that illustrate real-world applications and solutions implemented in industries facing CSP challenges. Furthermore, the development of a tool or software to solve CSP will be discussed, along with performance evaluations comparing it to existing solutions. Finally, the report will identify current research gaps and propose directions for future work to advance the field of CSP.

2 Modeling and Case study

2.1 Case Study

In this assignment, we choose a specfific industrial such as steel industry. In the steel industry, the Cut Stock Problem (CSP) is particularly relevant when large steel rolls or billets need to be cut into smaller segments to meet customer orders for various applications. Modeling this problem as a linear programming (LP) problem involves defining decision variables, an objective function, and constraints tailored to the steel production process. The primary objective is to minimize material waste while fulfilling all customer demands.

Problem formulation:
Suppose a manufacturing plant needs to cut steel bars from a large steel bar with a length as follows:

Stocks	Length	Cost
Type 1	80 inches	90
Type 2	100 inches	110
Type 3	120 inches	130

The plant needs to cut the steel bars into various sizes and quantities as follows:

Order	Order Length	Demand
S	15 inches	20
M	30 inches	10
L	34 inches	15
XL	47 inches	5

A single steel bar, whether it is 100 inches, 80 inches, or 120 inches in length, can be cut into one or more of the order lengths. For instance, a 100-inch bar could be cut into two 47-inch pieces with a 6-inch piece of scrap. Alternatively, it could be cut into a 47-inch piece, a 34-inch piece, and a 15-inch piece with a 4-inch piece of scrap. Similarly, the 80-inch and 120-inch bars can also be cut into various combinations of order lengths. Each possible way of cutting a bar is referred to as a pattern. In this example, there are numerous different patterns for each bar length. Determining how many of each pattern to cut across all three types of steel bars to meet customer orders while minimizing scrap is too complex to do by hand. Instead, this problem can be formulated as an optimization problem, specifically as an integer linear program.

Here's an illustration of a specific pattern for a 100-inch steel bar:

Order Lengths:

$$[15 \text{ inches},\ 30 \text{ inches},\ 34 \text{ inches},\ 47 \text{ inches}]$$

Pattern:

$$[0,\ 0,\ 0,\ 2]$$

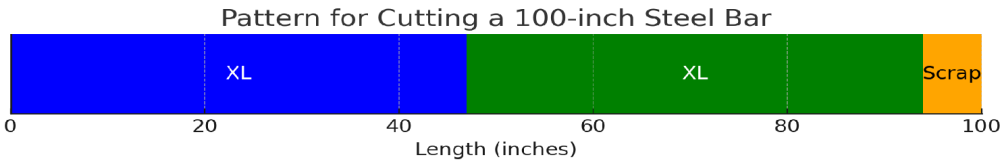


Figure 1: Simulaiton of a pattern

This pattern means that we cut the 100-inch bar into 2 pieces of 47-inch lengths, with no cuts of the other lengths.

Total Length Used: 94 inches - 2 pieces of 47 inches each (XL)
Scrap: 6 inches (remaning length after the cuts)
Etc.

2.2 Modeling

To solve the Cutting Stock Problem (CSP) using linear programming, we need to model it with variables, an objective function, and constraints. Here's a step-by-step guide to model the problem:

Sets:

$$I = \{1, 2, \ldots, I\} \quad \text{Set of all piece sizes or order sizes.}$$

$$K = \{1, 2, \ldots, K\} \quad \text{Set of all steel bar types.}$$

$$J_k = \{1, 2, \ldots, J_k\} \quad \text{Set of all cutting patterns for steel bar type } k.$$

Parameters:

$a_{i,j,k}$ Number of pieces of size i obtained from cutting pattern j on steel bar type k.

d_i Demand for pieces of size i.

l_i Length of a piece of size i.

L_k Length of steel bar type k.

$\text{Cost}(k)$ Cost associated with using one unit of steel bar type k.

Decision Variables:

$x_{j,k}$ The number of times pattern j is used from steel bar type k.

Objective Function:

$$\text{Minimize: } \sum_{k=1}^{K} \sum_{j=1}^{J_k} \text{Cost}(k) \cdot x_{j,k}$$

Constraints:

- Demand Constraints:

$$\sum_{k=1}^{K}\sum_{j=1}^{J_k} a_{i,j,k} \cdot x_{j,k} \geq d_i \quad \text{for all } i \in I$$

- Stock Length Constraints:

$$\sum_{i=1}^{I} l_i \cdot a_{i,j,k} \leq L_k \quad \text{for all } j \in J_k, k \in K$$

- Integer Constraints:

$$x_{j,k} \in \mathbb{Z}^+ \quad \text{for all } j \in J_k, k \in K$$

Now, we are applying it to our data:

- **Steel Bars:**
  - Type 1: 80 inches, Cost = 90
  - Type 2: 100 inches, Cost = 110
  - Type 3: 120 inches, Cost = 130
- **Orders:**
  - S: 15 inches, Demand = 20
  - M: 30 inches, Demand = 10
  - L: 34 inches, Demand = 15
  - XL: 47 inches, Demand = 5

Some examples of possible patterns:

- **Pattern 1 - Type 1 - 80 inches:**
  - 1 S (15 inches) and 2 M (30 inches each) → Total length = 75 inches (Remaining = 5 inches)
- **Pattern 2 - Type 1 - 80 inches:**
  - 3 S (15 inches each) and 1 M (30 inches) → Total length = 75 inches (Remaining = 5 inches)
- **Pattern 1 - Type 2 - 100 inches:**
  - 2 L (34 inches) and 1 M (30 inches) → Total length = 98 inches (Remaining = 2 inches)
- **Pattern 2 - Type 2 - 100 inches:**
  - 3 M (30 inches each) → Total length = 90 inches (Remaining = 10 inches)
- **Pattern 1 - Type 3 - 120 inches:**
  - 1 XL (47 inches) and 2 L (34 inches) → Total length = 115 inches (Remaining = 5 inches)

- **Pattern 2 - Type 3 - 120 inches:**
  - 1 XL (47 inches) and 1 M (30 inches) and 1 L (34 inches) → Total length = 111 inches (Remaining = 9 inches)
- Etc.

Decision Variables:

- $x_{j,1}$: Number of times pattern j is used from steel bar type 1 (80 inches).
- $x_{j,2}$: Number of times pattern j is used from steel bar type 2 (100 inches).
- $x_{j,3}$: Number of times pattern j is used from steel bar type 3 (120 inches).

Objective Function:
Minimize the total cost:

$$\text{Minimize} \quad 90\sum_{j=1}^{J_1} x_{j,1} + 110\sum_{j=1}^{J_2} x_{j,2} + 130\sum_{j=1}^{J_3} x_{j,3}$$

Constraints:

- **Demand Constraints:**

$$\sum_{k=1}^{K}\sum_{j=1}^{J_k} a_{i,j,k} \cdot x_{j,k} \geq d_i \quad \text{for all } i \in I$$

- $a_{i,j,k}$ represents the quantity of order i in pattern j for steel bar type k.

$$\begin{aligned} 1 \cdot x_{1,1} + 3 \cdot x_{2,1} &\geq 20 \quad \text{(for S)} \\ 2 \cdot x_{1,1} + 1 \cdot x_{2,1} + 1 \cdot x_{1,2} + 3 \cdot x_{2,2} + +1 \cdot x_{3,3} &\geq 10 \quad \text{(for M)} \\ 2 \cdot x_{1,2} + 2 \cdot x_{1,3} + 1 \cdot x_{2,3} &\geq 15 \quad \text{(for L)} \\ 1 \cdot x_{1,3} + 1 \cdot x_{2,3} &\geq 5 \quad \text{(for XL)} \end{aligned}$$

- **Stock Length Constraints:**

$$\sum_{i=1}^{I} l_i \cdot a_{i,j,k} \leq L_k \quad \text{for all } j \in J_k, k \in K$$

$$\begin{aligned} 15 \cdot 1 + 30 \cdot 2 &\leq 80 \quad \text{(for Stock Type 1 - Pattern 1 - 80 inches)} \\ 15 \cdot 3 + 30 \cdot 1 &\leq 80 \quad \text{(for Stock Type 1 - Pattern 2 - 80 inches)} \\ 34 \cdot 12 + 30 \cdot 1 &\leq 100 \quad \text{(for Stock Type 2 - Pattern 1 - 100 inches)} \\ 30 \cdot 3 &\leq 100 \quad \text{(for Stock Type 2 - Pattern 2 - 100 inches)} \\ 47 \cdot 1 + 34 \cdot 2 &\leq 120 \quad \text{(for Stock Type 3 - Pattern 1 - 120 inches)} \\ 47 \cdot 1 + 30 \cdot 1 + 34 \cdot 1 &\leq 120 \quad \text{(for Stock Type 3 - Pattern 2 - 120 inches)} \end{aligned}$$

- **Integer Constraints:**

$$x_{j,k} \geq 0 \text{ and integer}, \quad \forall j, \forall k$$

We have modeled the Cut Stock Problem (CSP) as a linear or integer programming problem. Solving the Cut Stock Problem (CSP) manually can be highly complex and time-consuming due to the following reasons:

- **Large Solution Space:** The number of possible patterns and their combinations can grow exponentially with the number of steel bar types and order sizes. This results in a vast solution space that is impractical to explore manually.
- **Integer Constraints:** The requirement for integer solutions adds a layer of complexity, as the solutions need to be exact without fractional values. This further complicates the manual solution process.
- **Demand Fulfillment:** Ensuring that all demands are met while minimizing waste requires careful balancing of patterns and quantities, which can be tedious and error-prone when done by hand.
- **Pattern Feasibility:** Each pattern must fit within the length of the steel bar, and managing these constraints manually for multiple patterns across various steel bar types adds significant complexity.

Approaches for Efficient Solution:

- **Optimization Algorithms:** Employing optimization algorithms such as Linear Programming (LP) or Integer Linear Programming (ILP) can efficiently handle the CSP. Software tools like CPLEX, Gurobi, or open-source solvers such as GLPK can be used to solve the ILP models.
- **Heuristic Methods:** For large-scale problems, heuristic methods such as Greedy Algorithms, Simulated Annealing, or Tabu Search can provide near-optimal solutions with reduced computational effort. These methods are particularly useful when exact solutions are computationally expensive.
- **Metaheuristic Approaches:** Advanced metaheuristic approaches like Genetic Algorithms or Particle Swarm Optimization can also be applied to find effective solutions for CSPs, especially when dealing with complex or large datasets.
- **Software Tools:** Utilizing dedicated software tools and programming languages (such as Python with libraries like PuLP or SciPy) can automate the solution process, providing accurate and efficient results without manual calculations.

In summary, while solving the Cut Stock Problem manually is impractical due to its complexity, leveraging optimization algorithms, heuristic methods, and software tools can greatly enhance efficiency and accuracy in finding solutions.

We need to prepare possible patterns. Here is the Python code to create the patterns of our data:

```python
from itertools import product

# Define stock information with their lengths and costs
stocks = {
    "Type 1": {"length": 80, "cost": 90},
    "Type 2": {"length": 100, "cost": 110},
    "Type 3": {"length": 120, "cost": 130},
}

# Define order requirements with their lengths and demands
order = {
    "S": {"length": 15, "demand": 20},
    "M": {"length": 30, "demand": 10},
    "L": {"length": 34, "demand": 15},
    "XL": {"length": 47, "demand": 5},
}

def is_valid_pattern(pattern, order, stock_length):
    """
    Check if the pattern is valid, i.e., total length of the cuts does not exceed the stock length,
    and the remaining length does not exceed any of the required sizes.
    """
    total_length = sum(order[f]["length"] * count for f, count in pattern.items())
    remaining_length = stock_length - total_length

    # Check if remaining length is not larger than any required size
    if remaining_length > 0 and remaining_length >= min(order[f]["length"] for f in order):
        return False

    return total_length <= stock_length

def generate_patterns(stock_length, order):
    """
    Generate all feasible patterns for a given stock length.
    """
    max_cuts = [stock_length // order[f]["length"] for f in order]
    feasible_patterns = []

    # Generate all possible combinations of cut counts
    for pattern in product(*(range(m + 1) for m in max_cuts)):
        pattern_dict = dict(zip(order.keys(), pattern))
        if is_valid_pattern(pattern_dict, order, stock_length):
            feasible_patterns.append(pattern_dict)

    return feasible_patterns

# Generate patterns for each stock
all_patterns = {}
for stock_id, stock_info in stocks.items():
    stock_length = stock_info["length"]
    patterns = generate_patterns(stock_length, order)
    all_patterns[stock_id] = patterns
```

Print result:

```python
# Display patterns
for stock_id, patterns in all_patterns.items():
    print(f"\nPatterns for stock {stock_id} (length {stocks[stock_id]['length']}):")
    for i, pattern in enumerate(patterns):
```

```
        total_length = sum(order[f]["length"] * count for f, count in pattern.
    items())
        remaining_length = stocks[stock_id]['length'] - total_length
        print(f"Pattern {i+1}: {pattern}, Total length: {total_length}, Remaining
    length: {remaining_length}")

# Optionally, save patterns to a JSON file
import json
with open('patterns.json', 'w') as f:
    json.dump(all_patterns, f)

print("\nPatterns have been saved to patterns.json.")
```

```
Patterns for stock Type 1 (length 80):
Pattern 1: {'S': 0, 'M': 0, 'L': 2, 'XL': 0}, Total length: 68, Remaining length: 12
Pattern 2: {'S': 0, 'M': 1, 'L': 0, 'XL': 1}, Total length: 77, Remaining length: 3
Pattern 3: {'S': 1, 'M': 1, 'L': 1, 'XL': 0}, Total length: 79, Remaining length: 1
Pattern 4: {'S': 1, 'M': 2, 'L': 0, 'XL': 0}, Total length: 75, Remaining length: 5
Pattern 5: {'S': 2, 'M': 0, 'L': 0, 'XL': 1}, Total length: 77, Remaining length: 3
Pattern 6: {'S': 3, 'M': 0, 'L': 1, 'XL': 0}, Total length: 79, Remaining length: 1
Pattern 7: {'S': 3, 'M': 1, 'L': 0, 'XL': 0}, Total length: 75, Remaining length: 5
Pattern 8: {'S': 5, 'M': 0, 'L': 0, 'XL': 0}, Total length: 75, Remaining length: 5

Patterns for stock Type 2 (length 100):
Pattern 1: {'S': 0, 'M': 0, 'L': 0, 'XL': 2}, Total length: 94, Remaining length: 6
Pattern 2: {'S': 0, 'M': 1, 'L': 2, 'XL': 0}, Total length: 98, Remaining length: 2
Pattern 3: {'S': 0, 'M': 2, 'L': 1, 'XL': 0}, Total length: 94, Remaining length: 6
Pattern 4: {'S': 0, 'M': 3, 'L': 0, 'XL': 0}, Total length: 90, Remaining length: 10
Pattern 5: {'S': 1, 'M': 0, 'L': 1, 'XL': 1}, Total length: 96, Remaining length: 4
Pattern 6: {'S': 1, 'M': 1, 'L': 0, 'XL': 1}, Total length: 92, Remaining length: 8
Pattern 7: {'S': 2, 'M': 0, 'L': 2, 'XL': 0}, Total length: 98, Remaining length: 2
Pattern 8: {'S': 2, 'M': 1, 'L': 1, 'XL': 0}, Total length: 94, Remaining length: 6
Pattern 9: {'S': 2, 'M': 2, 'L': 0, 'XL': 0}, Total length: 90, Remaining length: 10
Pattern 10: {'S': 3, 'M': 0, 'L': 0, 'XL': 1}, Total length: 92, Remaining length: 8
Pattern 11: {'S': 4, 'M': 0, 'L': 1, 'XL': 0}, Total length: 94, Remaining length: 6
Pattern 12: {'S': 4, 'M': 1, 'L': 0, 'XL': 0}, Total length: 90, Remaining length: 10
Pattern 13: {'S': 6, 'M': 0, 'L': 0, 'XL': 0}, Total length: 90, Remaining length: 10

Patterns for stock Type 3 (length 120):
Pattern 1: {'S': 0, 'M': 0, 'L': 2, 'XL': 1}, Total length: 115, Remaining length: 5
Pattern 2: {'S': 0, 'M': 1, 'L': 1, 'XL': 1}, Total length: 111, Remaining length: 9
Pattern 3: {'S': 0, 'M': 2, 'L': 0, 'XL': 1}, Total length: 107, Remaining length: 13
Pattern 4: {'S': 0, 'M': 4, 'L': 0, 'XL': 0}, Total length: 120, Remaining length: 0
Pattern 5: {'S': 1, 'M': 0, 'L': 0, 'XL': 2}, Total length: 109, Remaining length: 11
Pattern 6: {'S': 1, 'M': 0, 'L': 3, 'XL': 0}, Total length: 117, Remaining length: 3
Pattern 7: {'S': 1, 'M': 1, 'L': 2, 'XL': 0}, Total length: 113, Remaining length: 7
Pattern 8: {'S': 1, 'M': 2, 'L': 1, 'XL': 0}, Total length: 109, Remaining length: 11
Pattern 9: {'S': 2, 'M': 0, 'L': 1, 'XL': 1}, Total length: 111, Remaining length: 9
Pattern 10: {'S': 2, 'M': 1, 'L': 0, 'XL': 1}, Total length: 107, Remaining length: 13
Pattern 11: {'S': 2, 'M': 3, 'L': 0, 'XL': 0}, Total length: 120, Remaining length: 0
Pattern 12: {'S': 3, 'M': 0, 'L': 2, 'XL': 0}, Total length: 113, Remaining length: 7
Pattern 13: {'S': 3, 'M': 1, 'L': 1, 'XL': 0}, Total length: 109, Remaining length: 11
Pattern 14: {'S': 4, 'M': 0, 'L': 0, 'XL': 1}, Total length: 107, Remaining length: 13
Pattern 15: {'S': 4, 'M': 2, 'L': 0, 'XL': 0}, Total length: 120, Remaining length: 0
Pattern 16: {'S': 5, 'M': 0, 'L': 1, 'XL': 0}, Total length: 109, Remaining length: 11
Pattern 17: {'S': 6, 'M': 1, 'L': 0, 'XL': 0}, Total length: 120, Remaining length: 0
Pattern 18: {'S': 8, 'M': 0, 'L': 0, 'XL': 0}, Total length: 120, Remaining length: 0
```

Figure 2: All posible patterns of each Stock Type

In figure 2: It displays the possible cutting patterns, including the specific number of orders for each type, the total length used, and the remaining length.

Draw a lot of some pattern:

```
    def plot_pattern(stock_length, pattern, order, ax, stock_label):
    """
    Plot the cutting pattern on a chart.
    """
    x_offset = 0
    colors = {'S': 'red', 'M': 'blue', 'L': 'green', 'XL': 'orange'}
```

```
7
8       # Add cutting pieces to the chart
9       for size, count in pattern.items():
10          piece_length = order[size]['length']
11          for _ in range(count):
12              # Create a rectangle for each piece and add it to the plot
13              rect = patches.Rectangle((x_offset, 0), piece_length, 1, linewidth=1,
    edgecolor='black', facecolor=colors[size])
14              ax.add_patch(rect)
15              x_offset += piece_length
16
17      # Draw the remaining part of the stock bar if there is any leftover
18      if x_offset < stock_length:
19          remaining_length = stock_length - x_offset
20          # Create a rectangle for the remaining part with a dashed line style
21          rect = patches.Rectangle((x_offset, 0), remaining_length, 1, linewidth=1,
    edgecolor='black', facecolor='gray', linestyle='--')
22          ax.add_patch(rect)
23
24      # Set the x-axis limits and ticks
25      ax.set_xlim(0, stock_length)
26      ax.set_ylim(0, 1.5)
27      ax.set_yticks([])
28      ax.set_xticks([x for x in range(0, stock_length + 1, 10)])
29      ax.set_xticklabels([str(x) for x in range(0, stock_length + 1, 10)])
30      ax.set_title(stock_label)
31
32      # Add a legend to the chart
33      patches_list = [patches.Patch(color=color, label=label) for label, color in
    colors.items()]
34      ax.legend(handles=patches_list, loc='upper left')
35
36  # Create subplots for each stock type
37  fig, axs = plt.subplots(len(stocks), figsize=(12, 3 * len(stocks)))
38  for ax, (stock_id, stock_info) in zip(axs, stocks.items()):
39      stock_length = stock_info['length']
40      patterns = all_patterns[stock_id]
41      for i, pattern in enumerate(patterns[:3]):  # Only plot the first 3 patterns
    for each stock type
42          plot_pattern(stock_length, pattern, order, ax, f"{stock_id} - Pattern {i
    +1}")
43
44  plt.tight_layout()
45  plt.show()
```

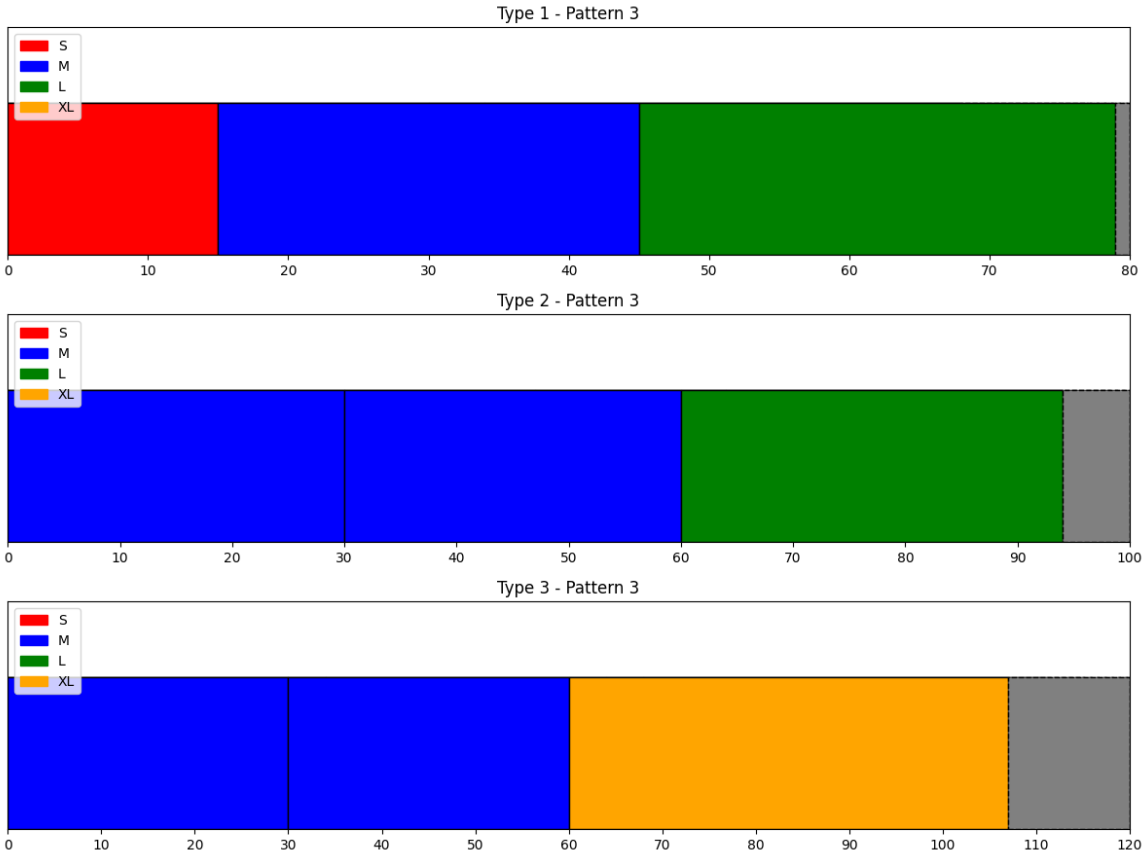


Figure 3: Visualize

In figure 3: Stock Type 1 - Pattern 3 cut into 1xS, 1xM, 1xL, 0xXL. Stock Type 2 - Pattern 3 cut into 0xS, 2xM, 1xL, 0xXL. Stock Type 3 - Pattern 3 cut into 0xS, 2xM, 0xL, 1xXL. And the remaining length is gray.

Next, we will analyze the algorithms to solve this problem.

3 Algorithm Analysis:

Choosing the Right Algorithm:

Size and Complexity: Exact algorithms are preferable for smaller problems or when an optimal solution is essential. For larger problems, heuristic or metaheuristic algorithms may be more practical.
Solution Quality vs. Computational Resources: Balance the trade-off between solution quality and computational resources.
Each algorithm has its strengths and weaknesses, so the choice depends on the specific requirements and constraints of our Cut Stock Problem.

There are many easy and available algorithms like Integer Linear Programming

(ILP) can efficiently handle the CSP. Software tools like CPLEX, Gurobi, or open-source solvers such as GLPK can be used to solve the ILP models. We are interested in heuristic methods to solve the CSP problem and compare them with exact algorithms as well as compare heuristic algorithms.

3.1 Greedy Heuristic

A greedy heuristic is an algorithmic approach used to find an approximate solution to optimization problems by making a series of locally optimal choices. The idea behind a greedy heuristic is simple: at each step of the algorithm, we choose the option that seems the best or most promising at that moment, without necessarily considering the global context or future consequences.

Key Characteristics of Greedy Heuristic:

- **Local Optimality:** The algorithm makes a choice that appears to be the best at each step based on the current situation. This choice is locally optimal, meaning it's the best immediate choice available.
- **No Backtracking:** Once a choice is made, the algorithm does not reconsider it. There's no backtracking or revisiting previous decisions, which often makes the algorithm faster but can also lead to suboptimal solutions.
- **Simplicity:** Greedy algorithms are typically easy to implement and understand. Their straightforward approach makes them suitable for problems where an approximate solution is acceptable.
- **Efficiency:** Because they do not involve exhaustive searches or backtracking, greedy algorithms are usually more time-efficient and can handle larger problem instances quickly.

In the CSP, we have several types of steel bars with different lengths and costs, and customer demands for specific sizes of steel. The goal of the problem is to find a way to cut these steel bars so that the total cost is minimized while fully meeting the customer demands. The greedy algorithm approaches this problem by performing the following steps:

- **Step 1 - Sorting steel bars by cost per unit length:** Steel bars are sorted by their cost per unit length in ascending order to prioritize using the cheaper bars.
- **Step 2 - Generating valid cutting patterns:** For each type of steel bar, the algorithm generates all valid cutting patterns, which are different ways to cut a steel bar without exceeding its length.
- **Step 3 - Selecting the best cutting pattern using a greedy approach:** At each step, the algorithm selects the most optimal cutting pattern based on cost and the number of steel bars needed to meet the remaining customer demand.
- **Step 4 - Updating demand:** After selecting a cutting pattern, the customer demand is updated to reflect the number of pieces that have been fulfilled.
- **Step 5 - Repeating the process:** The algorithm continues this process until all customer demands are met.

Implement Greedy Heuristic in Python:

```python
from itertools import product
import time
# Define stock information with their lengths and costs
stocks = {
    "Type 1": {"length": 80, "cost": 90},
    "Type 2": {"length": 100, "cost": 110},
    "Type 3": {"length": 120, "cost": 130},
}

# Define order requirements with their lengths and demands
order = {
    "S": {"length": 15, "demand": 20},
    "M": {"length": 30, "demand": 10},
    "L": {"length": 34, "demand": 15},
    "XL": {"length": 47, "demand": 5},
}
def greedy_cutting(order, stocks):
    """Perform the greedy cutting based on cost minimization and print the summary."""

    # Sort stocks by the ratio of cost per unit length in ascending order
    sorted_stocks = sorted(stocks.items(), key=lambda x: x[1]['cost'] / x[1]['length'])

    # Initialize remaining demand for each item and track stock usage
    remaining_demand = {k: v['demand'] for k, v in order.items()}
    stock_usage = {stock_id: {} for stock_id in stocks}  # Track the usage of each stock
    cut_counts = {demand: 0 for demand in order}  # Track how many times each demand is cut
    total_cost = 0  # Initialize total cost

    # Continue cutting until all demands are met
    while any(remaining_demand[f] > 0 for f in remaining_demand):
        for stock_id, stock_info in sorted_stocks:
            stock_length = stock_info["length"]
            cost = stock_info["cost"]

            # Generate all possible cutting patterns for the current stock length
            patterns = generate_patterns(stock_length, order)

            # Sort patterns based on the total pieces cut in descending order
            patterns = sorted(patterns, key=lambda p: sum(p.values()), reverse=True)

            best_pattern = None
            best_pattern_cost = float('inf')

            # Find the best pattern that meets the current remaining demand and has the lowest cost
            for pattern in patterns:
                if all(remaining_demand[f] >= pattern[f] for f in pattern):
                    if cost < best_pattern_cost:
                        best_pattern = pattern
                        best_pattern_cost = cost

            # If no suitable pattern is found, continue to the next stock
            if not best_pattern:
                continue

            # Track the usage of the best pattern for the current stock
            pattern_tuple = tuple(sorted(best_pattern.items()))
```

```
            if pattern_tuple in stock_usage[stock_id]:
                stock_usage[stock_id][pattern_tuple] += 1
            else:
                stock_usage[stock_id][pattern_tuple] = 1

            # Update remaining demand and cut counts based on the selected pattern
            for item in best_pattern:
                cut_counts[item] += best_pattern[item]
                remaining_demand[item] -= best_pattern[item]

            # Update total cost with the cost of the selected pattern
            total_cost += best_pattern_cost

            # If all demands are met, break out of the loop
            if all(remaining_demand[f] <= 0 for f in remaining_demand):
                break

    # Return the stock usage summary, total cost, and the cut counts for each demand
    return stock_usage, total_cost, cut_counts

# Perform greedy cutting
# Start timing
start_time = time.time()
stock_usage, total_cost, cut_counts = greedy_cutting(order, stocks)
# End timing
end_time = time.time()
# Calculate execution time
execution_time = end_time - start_time
# Print the summary with patterns as vectors and costs
print("Summary of Steel Bars Usage and Demand Fulfillment:\n")

total_cost_by_stock = {stock_id: 0 for stock_id in stocks}  # Initialize costs for each stock type

for stock_id, patterns in stock_usage.items():
    print(f"Stock {stock_id} (Length: {stocks[stock_id]['length']}):")
    pattern_index = 1
    for pattern_tuple, count in patterns.items():
        pattern_dict = dict(pattern_tuple)
        vector = [pattern_dict.get(demand, 0) for demand in order.keys()]
        pattern_cost = stocks[stock_id]['cost']
        total_cost_by_stock[stock_id] += pattern_cost * count  # Accumulate cost for the current pattern
        print(f"  Pattern {pattern_index}: {vector} x{count} (Cost: ${pattern_cost} each)")
        pattern_index += 1
```

Print result:

```
print("\nDemand Fulfillment:")
for demand in order:
    print(f"  {demand}: {cut_counts[demand]}/{order[demand]['demand']} pieces cut")

print("\nTotal Cost by Stock Type:")
for stock_id, cost in total_cost_by_stock.items():
    print(f"  {stock_id}: ${cost}")

print(f"\nTotal Cost: ${total_cost}\n")
print(f"Execution Time: {execution_time:.4f} seconds")
```

Result:

```
Stock Type 1 (Length: 80):
  Pattern 1: [5, 0, 0, 0] x1 (Cost: $90 each)
  Pattern 2: [1, 1, 1, 0] x1 (Cost: $90 each)
  Pattern 3: [0, 0, 2, 0] x5 (Cost: $90 each)
Stock Type 2 (Length: 100):
  Pattern 1: [6, 0, 0, 0] x1 (Cost: $110 each)
  Pattern 2: [0, 1, 2, 0] x1 (Cost: $110 each)
  Pattern 3: [0, 0, 0, 2] x2 (Cost: $110 each)
Stock Type 3 (Length: 120):
  Pattern 1: [8, 0, 0, 0] x1 (Cost: $130 each)
  Pattern 2: [0, 4, 0, 0] x2 (Cost: $130 each)
  Pattern 3: [0, 0, 2, 1] x1 (Cost: $130 each)

Demand Fulfillment:
  S: 20/20 pieces cut
  M: 10/10 pieces cut
  L: 15/15 pieces cut
  XL: 5/5 pieces cut

Total Cost by Stock Type:
  Type 1: $630
  Type 2: $440
  Type 3: $520

Total Cost: $1590

Execution Time: 0.0079 seconds
```

Figure 4: Results of running the greedy algorithm.

Analyze results:

The figure displays a summary of the results after running the algorithm. Specifically, we interpret the model results. Stock Type 1, we use patterns $1\text{x}[5,0,0,0]$, $1\text{x}[1,1,1,0]$, $5\text{x}[0,0,2,0]$. $x_{1,1} = 1$ (stock type 1 using pattern 1 are used 1 time) with cost $= 90$ and $a_{S,1,1} = 5$ (the number of order S in pattern 1 for stock 1). Same for the other stocks.

Ojective Function:

$$\textbf{Minimize} \sum_{k=1}^{K} \sum_{j=1}^{J_k} \text{Cost}(k) \cdot x_{j,k}$$

$$= Cost(1)(x_{1,1} + x_{2,1} + x_{3,1}) + Cost(2)(x_{1,2} + x_{2,2} + x_{3,2}) + Cost(3)(x_{1,3} + x_{2,3} + x_{3,3})$$

$$= 90(1+1+5) + 110(1+1+2) + 120(1+2+1) = 630 + 440 + 520 = \mathbf{1590}$$

Constraints:

$$\sum_{k=1}^{K} \sum_{j=1}^{J_k} a_{i,j,k} \cdot x_{j,k} \geq d_i, \quad \forall i$$

for demand S: $a_{S,1,1}x_{1,1} ... + a_{s,3,3}x_{3,3} = 5 + 1 + 0 + 6 + 0 + 0 + 8 + 0 + 0 = 20 >= d_S = \mathbf{20}$
for demand M: $a_{M,1,1}x_{1,1} ... + a_{s,3,3}x_{3,3} = 0 + 1 + 0 + 0 + 1 + 0 + 0 + 4 \cdot 2 + 0 = 10 >= d_M = \mathbf{10}$

for demand L: $a_{L,1,1}x_{1,1}... + a_{s,3,3}x_{3,3} = 0+1+2\cdot 5+0+2+0+0+0+2 = 15 >= d_L = \mathbf{15}$
for demand XL: $a_{XL,1,1}x_{1,1}... + a_{s,3,3}x_{3,3} = 0+0+0+0+0+2\cdot 2+0+0+1 = 5 >= d_{XL} = \mathbf{5}$

$$x_{j,k} \geq 0 \text{ and integer}, \quad \forall j, \forall k$$

Thus, that is the feasible solution (maybe optimal solution) to our problem using the greedy algorithm was determined above with how to cut stock and total cost ís **1590**. The Execution Time took **0.0079** seconds. We need to compare with the exact result of the problem.

We can easily get the optimal solution of the problem by using the available python library (ILP):

```python
import pulp
from itertools import product
# Create an optimization problem to minimize costs
prob = pulp.LpProblem("Cut_Stock_Problem", pulp.LpMinimize)

# Create decision variables for each stock and pattern combination
x = {}
for stock_id, patterns in all_patterns.items():
    for j, pattern in enumerate(patterns):
        x[(stock_id, j)] = pulp.LpVariable(f"x_{stock_id}_{j}", lowBound=0, cat='Integer')

# Objective function: Minimize the total cost
prob += pulp.lpSum(x[(stock_id, j)] * stocks[stock_id]['cost'] for stock_id in stocks for j in range(len(all_patterns[stock_id])))

# Constraint: Meet the demand for each size
for f in order:
    prob += pulp.lpSum(pattern[f] * x[(stock_id, j)]
                       for stock_id in stocks
                       for j, pattern in enumerate(all_patterns[stock_id])) >= order[f]['demand'], f"Demand_{f}"

# Solve the optimization problem
prob.solve()
```

Print result:

```python
# Display the results
print(f"Status: {pulp.LpStatus[prob.status]}")
for stock_id, patterns in all_patterns.items():
    for j, pattern in enumerate(patterns):
        if pulp.value(x[(stock_id, j)]) > 0:
            print(f"Use {pulp.value(x[(stock_id, j)])} of stock {stock_id} with pattern {pattern}")
```

```
Result - Optimal solution found

Objective value:                1500.00000000
Enumerated nodes:               0
Total iterations:               18
Time (CPU seconds):             0.01
Time (Wallclock seconds):       0.01

Option for printingOptions changed from normal to all
Total time (CPU seconds):       0.01   (Wallclock seconds):       0.01

Status: Optimal
Use 3.0 of stock Type 2 with pattern {'S': 2, 'M': 0, 'L': 2, 'XL': 0}
Use 4.0 of stock Type 3 with pattern {'S': 0, 'M': 0, 'L': 2, 'XL': 1}
Use 1.0 of stock Type 3 with pattern {'S': 0, 'M': 1, 'L': 1, 'XL': 1}
Use 1.0 of stock Type 3 with pattern {'S': 2, 'M': 3, 'L': 0, 'XL': 0}
Use 3.0 of stock Type 3 with pattern {'S': 4, 'M': 2, 'L': 0, 'XL': 0}
```

Figure 5: Optimal solution for this problem.

The result of the Greedy algorithm is **1590**, which is higher than **1500** - the optimal solution, indicating that Greedy did not find the optimal solution. But Greedy took **0.0079** seconds to find a solution and the optimal solution took **0.01** seconds. That shows that greedy in this case gives a near-optimal solution but it saves more time, which makes sense in complex problems that need calculation speed.

Test Data 1: The optimal solution is 220, The solution of Greedy algorithm is also 220

```
 # Define stock information with their lengths and costs
stocks = {
    "Type 1": {"length": 80, "cost": 90},
    "Type 2": {"length": 100, "cost": 110},
}

# Define the order requirements with their lengths and demands
order = {
    "A": {"length": 20, "demand": 5},
    "B": {"length": 30, "demand": 3},
}
```

```
Summary of Steel Bars Usage and Demand Fulfillment:

Stock Type 1 (Length: 80):
Stock Type 2 (Length: 100):
  Pattern 1: [5, 0] x1 (Cost: $110 each)
  Pattern 2: [0, 3] x1 (Cost: $110 each)

Demand Fulfillment:
  A: 5/5 pieces cut
  B: 3/3 pieces cut

Total Cost by Stock Type:
  Type 1: $0
  Type 2: $220

Total Cost: $220

Execution Time: 0.0000 seconds
```

Figure 6: Total Cost = 220, Time = 0.0000.

Test Data 2: The optimal solution is 960, The solution of Greedy algorithm is 970

```python
# Steel stock data with more types
stocks = {
    "Type 1": {"length": 80, "cost": 90},
    "Type 2": {"length": 100, "cost": 110},
    "Type 3": {"length": 120, "cost": 130},
}

# Detailed order requirements
order = {
    "A": {"length": 20, "demand": 10},
    "B": {"length": 30, "demand": 8},
    "C": {"length": 40, "demand": 6},
    "D": {"length": 50, "demand": 4},
}
```

```
Summary of Steel Bars Usage and Demand Fulfillment:

Stock Type 1 (Length: 80):
  Pattern 1: [1, 2, 0, 0] x1 (Cost: $90 each)
  Pattern 2: [0, 0, 2, 0] x2 (Cost: $90 each)
Stock Type 2 (Length: 100):
  Pattern 1: [2, 2, 0, 0] x1 (Cost: $110 each)
  Pattern 2: [1, 0, 2, 0] x1 (Cost: $110 each)
  Pattern 3: [0, 0, 0, 2] x2 (Cost: $110 each)
Stock Type 3 (Length: 120):
  Pattern 1: [6, 0, 0, 0] x1 (Cost: $130 each)
  Pattern 2: [0, 4, 0, 0] x1 (Cost: $130 each)

Demand Fulfillment:
  A: 10/10 pieces cut
  B: 8/8 pieces cut
  C: 6/6 pieces cut
  D: 4/4 pieces cut

Total Cost by Stock Type:
  Type 1: $270
  Type 2: $440
  Type 3: $260

Total Cost: $970

Execution Time: 0.0040 seconds
```

Figure 7: Total Cost = 970, Time = 0.0040.

Test Data 3: The optimal solution is 2650, The solution of Greedy algorithm is 3150

```python
 # Define stock information with their lengths and costs
stocks = {
    "Type 1": {"length": 80, "cost": 100},
    "Type 2": {"length": 100, "cost": 150},
    "Type 3": {"length": 120, "cost": 200},
}
# Define steel stock data with various types
order = {
    "S": {"length": 10, "demand": 50},
    "M": {"length": 20, "demand": 30},
    "L": {"length": 30, "demand": 20},
    "XL": {"length": 40, "demand": 10},
}
```

```
  Pattern 2: [2, 1, 0, 1] x1 (Cost: $100 each)
  Pattern 3: [1, 0, 1, 1] x1 (Cost: $100 each)
  Pattern 4: [1, 2, 1, 0] x1 (Cost: $100 each)
  Pattern 5: [0, 4, 0, 0] x2 (Cost: $100 each)
  Pattern 6: [3, 1, 1, 0] x1 (Cost: $100 each)
  Pattern 7: [0, 1, 2, 0] x1 (Cost: $100 each)
Stock Type 2 (Length: 100):
  Pattern 1: [0, 2, 2, 0] x1 (Cost: $150 each)
  Pattern 2: [2, 1, 2, 0] x2 (Cost: $150 each)
  Pattern 3: [3, 2, 1, 0] x1 (Cost: $150 each)
  Pattern 4: [0, 0, 2, 1] x2 (Cost: $150 each)
  Pattern 5: [2, 4, 0, 0] x1 (Cost: $150 each)
  Pattern 6: [6, 2, 0, 0] x1 (Cost: $150 each)
  Pattern 7: [0, 1, 0, 2] x1 (Cost: $150 each)
Stock Type 3 (Length: 120):
  Pattern 1: [9, 0, 1, 0] x1 (Cost: $200 each)
  Pattern 2: [4, 0, 0, 2] x1 (Cost: $200 each)
  Pattern 3: [1, 0, 1, 2] x1 (Cost: $200 each)
  Pattern 1: [8, 0, 0, 0] x2 (Cost: $100 each)
  Pattern 2: [0, 4, 0, 0] x2 (Cost: $100 each)
  Pattern 3: [0, 1, 2, 0] x1 (Cost: $100 each)
  Pattern 4: [0, 0, 0, 2] x2 (Cost: $100 each)
Stock Type 2 (Length: 100):
  Pattern 1: [10, 0, 0, 0] x2 (Cost: $150 each)
  Pattern 2: [0, 5, 0, 0] x2 (Cost: $150 each)
  Pattern 3: [0, 0, 2, 1] x3 (Cost: $150 each)
Stock Type 3 (Length: 120):
  Pattern 1: [12, 0, 0, 0] x1 (Cost: $200 each)
  Pattern 2: [2, 5, 0, 0] x1 (Cost: $200 each)
  Pattern 3: [0, 6, 0, 0] x1 (Cost: $200 each)
  Pattern 4: [0, 0, 4, 0] x3 (Cost: $200 each)
  Pattern 5: [0, 0, 0, 3] x1 (Cost: $200 each)

Demand Fulfillment:
  S: 50/50 pieces cut
  M: 30/30 pieces cut
  L: 20/20 pieces cut
  XL: 10/10 pieces cut

Total Cost by Stock Type:
  Type 1: $700
  Type 2: $1050
  Type 3: $1400

Total Cost: $3150

Execution Time: 0.0245 seconds
```

Figure 8: Total Cost = 3150, Time = 0.0245.

Discuss about Greedy Heuristic
Advantages:

- **Speed:** Greedy algorithms are often faster and more computationally efficient compared to exhaustive search algorithms like dynamic programming.
- **Simplicity:** The greedy approach is very intuitive and usually requires fewer lines of code, making it easier to implement.

Disadvantages:

- **Non-Optimal Solutions:** The main drawback is that the greedy approach does not always guarantee a globally optimal solution. It may lead to solutions that are only locally optimal, especially for complex problems where the best immediate choice may not lead to the best overall outcome.
- **Problem-Specific:** Greedy algorithms are highly dependent on the specific problem. They work well for certain problems but not for others, and determining when a greedy approach is appropriate requires careful analysis.

Impact on the Manufacturer:
Benefits:

- **Cost Efficiency:** If the results from the greedy algorithm significantly reduce cutting costs compared to standard methods, the manufacturer could achieve considerable savings.
- **Implementation Feasibility:** The greedy method is easy to implement and understand, making it a practical choice for manufacturers without advanced optimization tools.

Drawbacks:

- **Potential Cost Increase:** If the greedy algorithm results in higher overall costs compared to the optimal solution, the manufacturer may face increased operational expenses.
- **Limited Flexibility:** This method may not be flexible enough to adapt to changing demands or new types of steel bars without rerunning the algorithm.

For a manufacturer, using the greedy algorithm can provide a quick and feasible solution, but it should be validated against optimal solutions or used in conjunction with other methods to ensure it effectively meets the manufacturer's cost and demand requirements.

3.2 Simulated Annealing (SA)

Simulated Annealing (SA) is a probabilistic optimization algorithm inspired by the annealing process of metals. It is designed to find optimal solutions for complex optimization problems by exploring a large search space. SA is particularly useful when traditional exact methods may be inefficient or infeasible. SA begins with an initial solution and a high "temperature," allowing it to explore the search space widely and avoid becoming trapped in local optima. As the algorithm progresses, the temperature is gradually reduced according to a predefined cooling schedule, which decreases the probability of accepting worse solutions over time. This cooling process allows SA to focus on refining the current solution to find the global optimum.
Operating Principle
Simulated Annealing operates on the principle of allowing the acceptance of worse solutions under certain conditions to avoid getting trapped in local minima and improve the chances of finding a global optimum. The algorithm works as follows:

- **Initialization:** Start with a random initial solution and a high temperature. The high temperature allows the acceptance of worse solutions, helping to explore the search space more broadly.
- **Iteration:** In each iteration:
  - Generate a new solution by making a small random change to the current solution.
  - Compute the change in cost (or objective function) between the new solution and the current solution.
  - If the new solution is better, accept it as the current solution.
  - If the new solution is worse, accept it with a probability depending on the temperature and the cost change. This probability decreases as the temperature decreases.
- **Cooling:** Gradually decrease the temperature using a cooling rate ($0 < \alpha < 1$). This cooling schedule allows the algorithm to shift from exploration to exploitation as it converges.
- **Termination:** The algorithm stops when the temperature drops below a certain threshold or after a specified number of iterations.

Simulated annealing algorithm

```
Select the best solution vector x0 to be optimized
Initialize the parameters: temperature T, Boltzmann's constant k, reduction factor c
while termination criterion is not satisfied do
    for number of new solution
        Select a new solution: x0+Δx
            if f(x0+Δx) > f(x0) then
                fnew = f(x0 + Δx);  x0 = x0 + Δx
                else
                    Δf = f(x0 + Δx) − f(x0)
                    random r(0, 1)
                        if r > exp(−Δf/kT) then
                            fnew = f(x0 + Δx),  x0 = x0 + Δx
                            else
                            fnew = f(x0),
                        end if
            end if
        f = fnew
        Decrease the temperature periodically: T = c x T
    end for
end while
```

Figure 9: Pseudocode for Simulated Annealing.

Implement Simulated Annealing in Python:

```python
    def cost_of_pattern(pattern, stock_cost):
    """Calculate the cost of using a specific pattern."""
    return stock_cost

def evaluate_solution(solution, stocks, order):
    """Calculate the total cost of a given solution and check if it meets the
    demand."""
    total_cost = 0
```

```
    total_cut = {demand: 0 for demand in order}

    for stock_id, patterns in solution.items():
        stock_cost = stocks[stock_id]['cost']
        total_cost += stock_cost * sum(patterns.values())
        for pattern_tuple, count in patterns.items():
            pattern_dict = dict(pattern_tuple)
            for item, qty in pattern_dict.items():
                total_cut[item] += qty * count

    # Check if total cuts meet or exceed the demand
    demand_met = all(total_cut[d] >= order[d]['demand'] for d in order)
    return total_cost, demand_met

def create_initial_solution(stocks, order):
    """Create an initial random solution."""
    solution = {stock_id: {} for stock_id in stocks}
    remaining_demand = {k: v['demand'] for k, v in order.items()}

    while any(remaining_demand[f] > 0 for f in remaining_demand):
        stock_id = random.choice(list(stocks.keys()))
        stock_length = stocks[stock_id]["length"]
        patterns = generate_patterns(stock_length, order)
        if patterns:
            pattern = random.choice(patterns)
            pattern_tuple = tuple(sorted(pattern.items()))
            if pattern_tuple not in solution[stock_id]:
                solution[stock_id][pattern_tuple] = 0
            solution[stock_id][pattern_tuple] += 1
            for item in pattern:
                remaining_demand[item] -= pattern[item]

    return solution

def simulated_annealing(order, stocks, initial_temperature=1000, cooling_rate
    =0.995, max_iterations=500):
    """Perform Simulated Annealing to find the optimal cutting solution."""

    # Initialize the current solution using a method that creates an initial
    solution
    current_solution = create_initial_solution(stocks, order)

    # Evaluate the initial solution to get the cost and check if the demand is met
    current_cost, current_demand_met = evaluate_solution(current_solution, stocks,
     order)

    # Set the best solution as the current solution initially
    best_solution = current_solution
    best_cost = current_cost
    best_demand_met = current_demand_met

    # Initialize the temperature
    temperature = initial_temperature

    # Iterate through the maximum number of allowed iterations
    for iteration in range(max_iterations):
        # Generate a new potential solution
        new_solution = create_initial_solution(stocks, order)

        # Evaluate the new solution to get its cost and check if the demand is met
        new_cost, new_demand_met = evaluate_solution(new_solution, stocks, order)

```

```
        # Calculate the cost difference between the new solution and the current solution
        cost_diff = new_cost - current_cost

        # Determine if the new solution should replace the current one
        # Replace if the new solution is better (lower cost), or probabilistically based on temperature
        if new_demand_met and (cost_diff < 0 or random.random() < pow(2.718, -cost_diff / temperature)):
            current_solution = new_solution
            current_cost = new_cost
            current_demand_met = new_demand_met

            # Update the best solution found if the new solution is the best so far
            if new_demand_met and new_cost < best_cost:
                best_solution = new_solution
                best_cost = new_cost
                best_demand_met = new_demand_met

        # Cool down the temperature
        temperature *= cooling_rate

    # Return the best solution found, its cost, and whether the demand was fully met
    return best_solution, best_cost, best_demand_met
# Perform Simulated Annealing
start_time = time.time()
best_solution, best_cost, best_demand_met = simulated_annealing(order, stocks)
end_time = time.time()
execution_time = end_time - start_time
```

Print result:

```
# Print the summary with patterns as vectors and costs
print("Summary of Steel Bars Usage and Demand Fulfillment:\n")

if best_demand_met:
    total_cost_by_stock = {stock_id: 0 for stock_id in stocks}

    for stock_id, patterns in best_solution.items():
        print(f"Stock {stock_id} (Length: {stocks[stock_id]['length']}):")
        pattern_index = 1
        for pattern_tuple, count in patterns.items():
            pattern_dict = dict(pattern_tuple)
            vector = [pattern_dict.get(demand, 0) for demand in order.keys()]
            pattern_cost = stocks[stock_id]['cost']
            total_cost_by_stock[stock_id] += pattern_cost * count
            print(f"  Pattern {pattern_index}: {vector} x{count} (Cost: ${pattern_cost} each)")
            pattern_index += 1

    print("\nTotal Cost by Stock Type:")
    for stock_id, cost in total_cost_by_stock.items():
        print(f"  {stock_id}: ${cost}")

    print(f"\nTotal Cost: ${best_cost}\n")
else:
    print("No feasible solution meets the demand.")

print(f"Execution Time: {execution_time:.4f} seconds")
```

Because the algorithm will produce different results each time the program is executed, we need to run it many times to find the best result.

```
Summary of Steel Bars Usage and Demand Fulfillment:

Stock Type 1 (Length: 80):
  Pattern 1: [0, 0, 2, 0] x1 (Cost: $90 each)
  Pattern 2: [2, 0, 0, 1] x1 (Cost: $90 each)
  Pattern 3: [1, 1, 1, 0] x1 (Cost: $90 each)
Stock Type 2 (Length: 100):
  Pattern 1: [0, 1, 2, 0] x1 (Cost: $110 each)
  Pattern 2: [1, 0, 1, 1] x1 (Cost: $110 each)
  Pattern 3: [0, 0, 0, 2] x1 (Cost: $110 each)
  Pattern 4: [2, 1, 1, 0] x1 (Cost: $110 each)
  Pattern 5: [2, 0, 2, 0] x1 (Cost: $110 each)
  Pattern 6: [1, 1, 0, 1] x1 (Cost: $110 each)
  Pattern 7: [4, 1, 0, 0] x1 (Cost: $110 each)
Stock Type 3 (Length: 120):
  Pattern 1: [2, 3, 0, 0] x1 (Cost: $130 each)
  Pattern 2: [1, 1, 2, 0] x1 (Cost: $130 each)
  Pattern 3: [0, 1, 1, 1] x1 (Cost: $130 each)
  Pattern 4: [3, 0, 2, 0] x1 (Cost: $130 each)
  Pattern 5: [2, 0, 1, 1] x1 (Cost: $130 each)

Total Cost by Stock Type:
  Type 1: $270
  Type 2: $770
  Type 3: $650

Total Cost: $1690

Execution Time: 8.6213 seconds

=== Code Execution Successful ===
```

Figure 10: Runs Simulated Annealing (1).

In figure 10: In the first run, the total cost is **1690** and the Execution Time is **8.6213** seconds.

```
Summary of Steel Bars Usage and Demand Fulfillment:

Stock Type 1 (Length: 80):
  Pattern 1: [1, 2, 0, 0] x1 (Cost: $90 each)
  Pattern 2: [0, 0, 2, 0] x2 (Cost: $90 each)
  Pattern 3: [3, 1, 0, 0] x1 (Cost: $90 each)
  Pattern 4: [3, 0, 1, 0] x1 (Cost: $90 each)
Stock Type 2 (Length: 100):
  Pattern 1: [1, 0, 1, 1] x1 (Cost: $110 each)
  Pattern 2: [0, 0, 0, 2] x1 (Cost: $110 each)
Stock Type 3 (Length: 120):
  Pattern 1: [3, 1, 1, 0] x2 (Cost: $130 each)
  Pattern 2: [1, 1, 2, 0] x1 (Cost: $130 each)
  Pattern 3: [1, 0, 3, 0] x1 (Cost: $130 each)
  Pattern 4: [2, 0, 1, 1] x2 (Cost: $130 each)
  Pattern 5: [0, 4, 0, 0] x1 (Cost: $130 each)

Total Cost by Stock Type:
  Type 1: $450
  Type 2: $220
  Type 3: $910

Total Cost: $1580

Execution Time: 6.0528 seconds

=== Code Execution Successful ===
```

Figure 11: Runs Simulated Annealing (2).

In figure 11: In the second run, the total cost is **1580** and the Execution Time is **6.0528** seconds.

```
Summary of Steel Bars Usage and Demand Fulfillment:

Stock Type 1 (Length: 80):
  Pattern 1: [0, 0, 2, 0] x2 (Cost: $90 each)
  Pattern 2: [1, 1, 1, 0] x1 (Cost: $90 each)
  Pattern 3: [5, 0, 0, 0] x1 (Cost: $90 each)
Stock Type 2 (Length: 100):
  Pattern 1: [0, 1, 2, 0] x1 (Cost: $110 each)
  Pattern 2: [2, 0, 2, 0] x1 (Cost: $110 each)
  Pattern 3: [1, 0, 1, 1] x2 (Cost: $110 each)
  Pattern 4: [3, 0, 0, 1] x1 (Cost: $110 each)
Stock Type 3 (Length: 120):
  Pattern 1: [2, 3, 0, 0] x2 (Cost: $130 each)
  Pattern 2: [3, 1, 1, 0] x1 (Cost: $130 each)
  Pattern 3: [0, 0, 2, 1] x1 (Cost: $130 each)
  Pattern 4: [0, 1, 1, 1] x1 (Cost: $130 each)

Total Cost by Stock Type:
  Type 1: $360
  Type 2: $550
  Type 3: $650

Total Cost: $1560

Execution Time: 6.5687 seconds

=== Code Execution Successful ===
```

Figure 12: Runs Simulated Annealing (3).

In figure 12: In the third run, the total cost is **1560** and the Execution Time is **6.5687** seconds. This may be a good solution that the SA algorithm produces.

Analyze results: (similar to Greedy)
Ojective Function:

$$\textbf{Minimize} \sum_{k=1}^{K} \sum_{j=1}^{J_k} \text{Cost}(k) \cdot x_{j,k}$$

$$= Cost(1)(x_{1,1}+x_{2,1}+x_{3,1})+Cost(2)(x_{1,2}+x_{2,2}+x_{3,2}+x_{4,2})+Cost(3)(x_{1,3}+x_{2,3}+x_{3,3}+x_{4.3})$$
$$= 90(2+1+1)+110(1+1+2+1)+120(2+1+1+1) = 360+550+650 = \textbf{1560}$$

Constraints:

$$\sum_{k=1}^{K} \sum_{j=1}^{J_k} a_{i,j,k} \cdot x_{j,k} \geq d_i, \quad \forall i$$

for demand S: $a_{S,1,1}x_{1,1}...+a_{s,4,3}x_{4,3} = 0+1+5+0+2+1\cdot2+3+2\cdot2+3+0+0 = 20 >= d_S = \mathbf{20}$
for demand M: $a_{M,1,1}x_{1,1}...+a_{s,4,3}x_{4,3} = 0+1+0+1+0+0+0+3\cdot2+1+0+1 = 10 >= d_M = \mathbf{10}$
for demand L: $a_{L,1,1}x_{1,1}...+a_{s,4,3}x_{4,3} = 2\cdot2+1+0+2+2+1\cdot2+0+0+1+2+1 = 15 >= d_L = \mathbf{15}$
for demand XL: $a_{XL,1,1}x_{1,1}...+a_{s,4,3}x_{4,3} = 0+0+0+0+0+1\cdot2+1+0+0+1+1 = 5 >= d_{XL} = \mathbf{5}$

$$x_{j,k} \geq 0 \text{ and integer}, \quad \forall j, \forall k$$

Indeed, in this case the SA algorithm produces a more optimal solution **(1560)** than the greedy algorithm **(1590)**, but it is not the most optimal **(1500)** and it takes a lot of execution time **(6.5687)**.

Test Data 1: The optimal solution is 220, The solution of Simulated Annealing algorithm is also 220

```python
 # Define stock information with their lengths and costs
stocks = {
    "Type 1": {"length": 80, "cost": 90},
    "Type 2": {"length": 100, "cost": 110},
}

# Define the order requirements with their lengths and demands
order = {
    "A": {"length": 20, "demand": 5},
    "B": {"length": 30, "demand": 3},
}
```

```
Summary of Steel Bars Usage and Demand Fulfillment:

Stock Type 1 (Length: 80):
Stock Type 2 (Length: 100):
  Pattern 1: [0, 3] x1 (Cost: $110 each)
  Pattern 2: [5, 0] x1 (Cost: $110 each)

Total Cost by Stock Type:
  Type 1: $0
  Type 2: $220

Total Cost: $220

Execution Time: 0.0477 seconds
```

Figure 13: Total Cost = 220, Time = 0.0477.

Test Data 2: The optimal solution is 960, The solution of Simulated Annealing algorithm is 1000

```python
# Steel stock data with more types
stocks = {
    "Type 1": {"length": 80, "cost": 90},
    "Type 2": {"length": 100, "cost": 110},
    "Type 3": {"length": 120, "cost": 130},
}

# Detailed order requirements
order = {
    "A": {"length": 20, "demand": 10},
    "B": {"length": 30, "demand": 8},
```

```
    "C": {"length": 40, "demand": 6},
    "D": {"length": 50, "demand": 4},
}
```

```
Summary of Steel Bars Usage and Demand Fulfillment:

Stock Type 1 (Length: 80):
  Pattern 1: [0, 1, 0, 1] x2 (Cost: $90 each)
  Pattern 2: [4, 0, 0, 0] x1 (Cost: $90 each)
  Pattern 3: [0, 0, 2, 0] x1 (Cost: $90 each)
  Pattern 4: [0, 1, 1, 0] x1 (Cost: $90 each)
  Pattern 5: [1, 2, 0, 0] x1 (Cost: $90 each)
Stock Type 2 (Length: 100):
  Pattern 1: [0, 2, 1, 0] x1 (Cost: $110 each)
  Pattern 2: [1, 1, 0, 1] x1 (Cost: $110 each)
  Pattern 3: [3, 0, 1, 0] x1 (Cost: $110 each)
Stock Type 3 (Length: 120):
  Pattern 1: [1, 0, 1, 1] x1 (Cost: $130 each)

Total Cost by Stock Type:
  Type 1: $540
  Type 2: $330
  Type 3: $130

Total Cost: $1000

Execution Time: 2.3014 seconds
```

Figure 14: Total Cost = 1000, Time = 2.3014.

Test Data 3: The optimal solution 2650, The solution of Simulated Annealing algorithm is 3100

```
 # Define stock information with their lengths and costs
stocks = {
    "Type 1": {"length": 80, "cost": 100},
    "Type 2": {"length": 100, "cost": 150},
    "Type 3": {"length": 120, "cost": 200},
}
# Define steel stock data with various types
order = {
    "S": {"length": 10, "demand": 50},
    "M": {"length": 20, "demand": 30},
    "L": {"length": 30, "demand": 20},
    "XL": {"length": 40, "demand": 10},
}
```

```
Total Cost by Stock Type:
  Pattern 5: [0, 1, 2, 0] x2 (Cost: $100 each)
  Pattern 6: [5, 0, 1, 0] x1 (Cost: $100 each)
  Pattern 7: [2, 1, 0, 1] x1 (Cost: $100 each)
  Pattern 8: [2, 0, 2, 0] x1 (Cost: $100 each)
  Pattern 9: [6, 1, 0, 0] x1 (Cost: $100 each)
  Pattern 10: [4, 0, 0, 1] x1 (Cost: $100 each)
  Pattern 11: [0, 0, 0, 2] x1 (Cost: $100 each)
Stock Type 2 (Length: 100):
  Pattern 1: [1, 0, 3, 0] x1 (Cost: $150 each)
  Pattern 2: [2, 0, 0, 2] x1 (Cost: $150 each)
  Pattern 3: [0, 5, 0, 0] x1 (Cost: $150 each)
  Pattern 4: [2, 4, 0, 0] x1 (Cost: $150 each)
  Pattern 5: [3, 2, 1, 0] x1 (Cost: $150 each)
  Pattern 6: [0, 3, 0, 1] x1 (Cost: $150 each)
Stock Type 3 (Length: 120):
  Pattern 1: [3, 0, 3, 0] x1 (Cost: $200 each)
  Pattern 2: [0, 6, 0, 0] x1 (Cost: $200 each)
  Pattern 3: [0, 0, 4, 0] x1 (Cost: $200 each)
  Pattern 4: [2, 3, 0, 1] x1 (Cost: $200 each)
  Pattern 5: [12, 0, 0, 0] x1 (Cost: $200 each)

Total Cost by Stock Type:
  Type 1: $1200
  Type 2: $900
  Type 3: $1000

Total Cost: $3100

Execution Time: 23.1411 seconds
```

Figure 15: Total Cost = 3100, Time = 23.1411.

Indeed, the Simulated Annealing (SA) algorithm can exhibit significant variability in finding an optimal solution and often requires a lot of time, heavily depending on the specific problem at hand.

Commentary on Result Variability in Simulated Annealing:
Simulated Annealing (SA) can produce different results in each run due to its inherent randomness. This variability arises because the algorithm generates new solutions by making random changes to the current solution and accepts worse solutions with a probability that depends on the temperature. This randomness means that each run of SA may explore different regions of the solution space. **Therefore, running the algorithm multiple times is crucial to finding the best possible solution**. By performing multiple runs, we can average out the randomness and improve the likelihood of finding an optimal or near-optimal solution. This approach helps in enhancing the reliability and quality of the final result while minimizing the risk of getting trapped in local minima.

Advantages:

- **Ability to Avoid Local Minima:** SA can accept worse solutions with a certain probability, which helps it avoid getting trapped in local minima and allows it to explore a broader

solution space.

- **Simplicity and Flexibility:** The algorithm is relatively simple to implement and can be applied to a wide range of optimization problems without needing specific constraints or detailed problem information. It only requires a cost function and a method to generate new solutions.
- **Comprehensive Exploration:** With its cooling mechanism, SA explores the solution space thoroughly, increasing the chances of finding a global optimum or a near-optimal solution.
- **No Need for Problem-Specific Knowledge:** SA does not require specific information about the structure of the solution space or the relationships between variables, making it highly adaptable.

Disadvantages:

- **Computationally Expensive:** SA may require many iterations and substantial computational time, especially for complex problems with large solution spaces. The running time can become impractical for very large problems.
- **Parameter Sensitivity:** The performance of SA is highly sensitive to parameters such as the initial temperature, cooling rate, and number of iterations. Choosing the appropriate parameters can be challenging and may require multiple trials.
- **Randomness:** Due to its probabilistic nature in accepting worse solutions and generating new solutions, the results of SA can vary between runs. Multiple runs are often necessary to ensure finding the best possible solution.
- **No Guarantee of Optimal Solution:** While SA can find good solutions, there is no guarantee that it will find the global optimum. The results depend heavily on the cooling schedule and exploration strategy.

3.3 FFD heuristic

FFD (First-Fit Decreasing) Heuristic consists in putting the largest item in the pattern the highest possible number of times without excess in the demand. If the selected item does not fit the pattern anymore, the second largest item is selected and so forth. The complete algorithm is presented below:

BEGIN

P.1. Sort the items in decreasing order of size.

Suppose $l_1 \geq l_2 \geq \cdots \geq l_m$

P.2. Be r_i the residual demand of the item $i \in I$.

$I = \{1, \ldots, m\}$. {set of indexes of the items}

At first: $r_i = d_i, \forall i \in I$.

Do $k = 1$ {First cutting pattern }

STOP=False {logic variable tht indicates a non-null demand}

While STOP=False

P.3. Do: $Rest = L$ and $\alpha_{ik} = 0, \forall i \in I$;

Be $i = 1$ {start by putting the first item in the pattern }

While ($i \leq m$ and $Rest \geq l_i$) Do:

$\alpha_{ik} = \min \left\{ \left\lfloor \frac{Rest}{l_i} \right\rfloor, r_i \right\}$

(α_{ik} is the quantity of items type i in pattern k)

Do: $Rest = Rest - (\alpha_{ik} l_i)$

$r_i = r_i - \alpha_{ik}; i = i + 1$

End of While

P.4. Determine the frequency of pattern k: $x_k = \left\{ \begin{array}{l} \min \left\lfloor \frac{d_i}{\alpha_{ik}} \right\rfloor, \\ \forall i \in I; \alpha_{ik} > 0 \end{array} \right\}$

P.5. (Stop Criterion)

If $r_i = 0, \ \forall i \in I$ **then** STOP = Truth.

Otherwise Do $k = k + 1$ and come back to **P.3**

End of While.

END

The FFD algorithm

Implement FFD Heuristic in python:

```python
import math
import time

# Stock and order information (provided)
stocks = {
    "Type 1": {"length": 80, "cost": 90},
    "Type 2": {"length": 100, "cost": 110},
    "Type 3": {"length": 120, "cost": 130},
}

order = {
    "S": {"length": 15, "demand": 20},
    "M": {"length": 30, "demand": 10},
```

```
    "L": {"length": 34, "demand": 15},
    "XL": {"length": 47, "demand": 5},
}

def ffd_heuristic(stocks, order):
    """
    Applies the First-Fit Decreasing heuristic to the cutting stock problem.

    Args:
        stocks: A dictionary containing stock types, their lengths, and costs.
        order: A dictionary containing order types, their lengths, and demands.

    Returns:
        A list of cutting patterns, where each pattern is a dictionary
        indicating how many of each order type are cut from a stock type.
    """

    # Sort order items by decreasing length
    order_items = sorted(order.items(), key=lambda x: x[1]["length"], reverse=True
    )

    # Initialize residual demands
    residual_demands = {item: details["demand"] for item, details in order.items()
    }

    # Initialize patterns list
    patterns = []

    while any(residual_demands.values()):  # Continue until all demands are met
        for stock_type, stock_details in stocks.items():
            stock_length = stock_details["length"]
            remaining_length = stock_length
            pattern = {item: 0 for item in order}  # Initialize an empty pattern

            for item, _ in order_items:
                item_length = order[item]["length"]
                while remaining_length >= item_length and residual_demands[item] >
     0:
                    pattern[item] += 1
                    remaining_length -= item_length
                    residual_demands[item] -= 1

            if any(pattern.values()):  # Add pattern only if it's not empty
                patterns.append({"stock_type": stock_type, "cuts": pattern})

    return patterns
```

Print Result

```
Summary of Steel Bars Usage And Demand Fulfillment:

Stock Type 1 (Length: 80):
  Pattern 1: [0, 1, 0, 1] x1 (Cost: $90 each)
  Pattern 2: [0, 0, 2, 0] x1 (Cost: $90 each)
  Pattern 3: [0, 0, 2, 0] x1 (Cost: $90 each)
  Pattern 4: [1, 1, 1, 0] x1 (Cost: $90 each)
  Pattern 5: [5, 0, 0, 0] x1 (Cost: $90 each)
  Total Cost For Type 1: $450

Stock Type 2 (Length: 100):
  Pattern 1: [0, 0, 0, 2] x1 (Cost: $110 each)
  Pattern 2: [0, 1, 2, 0] x1 (Cost: $110 each)
  Pattern 3: [0, 1, 2, 0] x1 (Cost: $110 each)
```

```
  Pattern 4: [0, 3, 0, 0] x1 (Cost: $110 each)
  Pattern 5: [6, 0, 0, 0] x1 (Cost: $110 each)
  Total Cost For Type 2: $550

Stock Type 3 (Length: 120):
  Pattern 1: [1, 0, 0, 2] x1 (Cost: $130 each)
  Pattern 2: [1, 0, 3, 0] x1 (Cost: $130 each)
  Pattern 3: [1, 0, 3, 0] x1 (Cost: $130 each)
  Pattern 4: [2, 3, 0, 0] x1 (Cost: $130 each)
  Pattern 5: [3, 0, 0, 0] x1 (Cost: $130 each)
  Total Cost For Type 3: $650

Demand Fulfillment:
  S: 20/20 pieces cut
  M: 10/10 pieces cut
  L: 15/15 pieces cut
  XL: 5/5 pieces cut


Total Cost by Stock Type:
  Type 1: $450
  Type 2: $550
  Type 3: $650

Total Cost: $1650
Execution Time: 0.0001 seconds
```

Ojective Function:

$$\textbf{Minimize} \sum_{k=1}^{K} \sum_{j=1}^{J_k} \text{Cost}(k) \cdot x_{j,k}$$

$= Cost(1)(x_{1,1}+x_{2,1}+x_{3,1})+x_{4,1}+x_{5,1})+Cost(2)(x_{1,2}+x_{2,2}+x_{3,2}+x_{4,2}+x_{5,2})+Cost(3)(x_{1,3}+x_{2,3}+x_{3,3}+x_{4.3}+x_{5.3})$

$= 90(1+1+1+1+1)+110(1+1+1+1+1)+130(1+1+1+1+1) = 450+550+650 = \mathbf{1650}$

Constraints:

$$\sum_{k=1}^{K} \sum_{j=1}^{J_k} a_{i,j,k} \cdot x_{j,k} \geq d_i, \quad \forall i$$

for S: $a_{S,1,1}x_{1,1}...+a_{s,4,3}x_{4,3} = 0+0+0+1+5+0+0+0+0+6+1+1+1+2+3 = 20 >= d_S = \mathbf{20}$
for M: $a_{M,1,1}x_{1,1}...+a_{s,4,3}x_{4,3} = 1+0+0+1+0+0+1+1+3+0+0+0+0+3+0 = 10 >= d_M = \mathbf{10}$
for L: $a_{L,1,1}x_{1,1}...+a_{s,4,3}x_{4,3} = 0+2+2+1+0+0+2+2+0+0+0+3+3+0+0 = 15 >= d_L = \mathbf{15}$
for XL: $a_{XL,1,1}x_{1,1}...+a_{s,4,3}x_{4,3} = 1+0+0+0+0+2+0+0+0+0+2+0+0+0+0 = 5 >= d_{XL} = \mathbf{5}$

$$x_{j,k} \geq 0 \text{ and integer}, \quad \forall j, \forall k$$

Indeed, in this case the FFD algorithm produces a less optimal solution **(1650)** than the greedy algorithm **(1590)** and SA algorithm **(1560)**, but it takes small excution time **0.0001** seconds.

Test Data 1: The optimal solution is 220, The solution of FFD algorithm is 290

```
 # Define stock information with their lengths and costs
stocks = {
    "Type 1": {"length": 80, "cost": 90},
    "Type 2": {"length": 100, "cost": 110},
}

# Define the order requirements with their lengths and demands
```

```
order = {
    "A": {"length": 20, "demand": 5},
    "B": {"length": 30, "demand": 3},
}
```

Result:

```
Summary of Steel Bars Usage And Demand Fulfillment:

Stock Type 1 (Length: 80):
  Pattern 1: [1, 2] x1 (Cost: $90 each)
  Pattern 2: [1, 0] x1 (Cost: $90 each)
  Total Cost For Type 1: $180

Stock Type 2 (Length: 100):
  Pattern 1: [3, 1] x1 (Cost: $110 each)
  Total Cost For Type 2: $110

Demand Fulfillment:
  A: 5/5 pieces cut
  B: 3/3 pieces cut


Total Cost by Stock Type:
  Type 1: $180
  Type 2: $110

Total Cost: $290
Execution Time: 0.0001 seconds
```

Test Data 2: The optimal solution is 960, The solution of FFD algorithm is 990

```
# Steel stock data with more types
stocks = {
    "Type 1": {"length": 80, "cost": 90},
    "Type 2": {"length": 100, "cost": 110},
    "Type 3": {"length": 120, "cost": 130},
}

# Detailed order requirements
order = {
    "A": {"length": 20, "demand": 10},
    "B": {"length": 30, "demand": 8},
    "C": {"length": 40, "demand": 6},
    "D": {"length": 50, "demand": 4},
}
```

Result:

```
Summary of Steel Bars Usage And Demand Fulfillment:

Stock Type 1 (Length: 80):
  Pattern 1: [0, 1, 0, 1] x1 (Cost: $90 each)
  Pattern 2: [0, 0, 2, 0] x1 (Cost: $90 each)
  Pattern 3: [1, 2, 0, 0] x1 (Cost: $90 each)
  Total Cost For Type 1: $270

Stock Type 2 (Length: 100):
  Pattern 1: [0, 0, 0, 2] x1 (Cost: $110 each)
  Pattern 2: [1, 0, 2, 0] x1 (Cost: $110 each)
  Pattern 3: [2, 2, 0, 0] x1 (Cost: $110 each)
  Total Cost For Type 2: $330

Stock Type 3 (Length: 120):
```

```
  Pattern 1: [0, 1, 1, 1] x1 (Cost: $130 each)
  Pattern 2: [1, 2, 1, 0] x1 (Cost: $130 each)
  Pattern 3: [5, 0, 0, 0] x1 (Cost: $130 each)
  Total Cost For Type 3: $390

Demand Fulfillment:
  A: 10/10 pieces cut
  B: 8/8 pieces cut
  C: 6/6 pieces cut
  D: 4/4 pieces cut


Total Cost by Stock Type:
  Type 1: $270
  Type 2: $330
  Type 3: $390

Total Cost: $990
Execution Time: 0.0002 seconds
```

Test Data 3: The optimal solution 2650, The solution of FFD algorithm is 3150

```python
 # Define stock information with their lengths costs
stocks = {
    "Type 1": {"length": 80, "cost": 100},
    "Type 2": {"length": 100, "cost": 150},
    "Type 3": {"length": 120, "cost": 200},
}
# Define steel stock data with various types
order = {
    "S": {"length": 10, "demand": 50},
    "M": {"length": 20, "demand": 30},
    "L": {"length": 30, "demand": 20},
    "XL": {"length": 40, "demand": 10},
}
```

Result:

```
Summary of Steel Bars Usage And Demand Fulfillment:

Stock Type 1 (Length: 80):
  Pattern 1: [0, 0, 0, 2] x1 (Cost: $100 each)
  Pattern 2: [0, 0, 0, 2] x1 (Cost: $100 each)
  Pattern 3: [0, 1, 2, 0] x1 (Cost: $100 each)
  Pattern 4: [0, 1, 2, 0] x1 (Cost: $100 each)
  Pattern 5: [0, 4, 0, 0] x1 (Cost: $100 each)
  Pattern 6: [0, 4, 0, 0] x1 (Cost: $100 each)
  Pattern 7: [8, 0, 0, 0] x1 (Cost: $100 each)
  Total Cost For Type 1: $700

Stock Type 2 (Length: 100):
  Pattern 1: [0, 1, 0, 2] x1 (Cost: $150 each)
  Pattern 2: [0, 0, 2, 1] x1 (Cost: $150 each)
  Pattern 3: [1, 0, 3, 0] x1 (Cost: $150 each)
  Pattern 4: [1, 0, 3, 0] x1 (Cost: $150 each)
  Pattern 5: [0, 5, 0, 0] x1 (Cost: $150 each)
  Pattern 6: [6, 2, 0, 0] x1 (Cost: $150 each)
  Pattern 7: [10, 0, 0, 0] x1 (Cost: $150 each)
  Total Cost For Type 2: $1050

Stock Type 3 (Length: 120):
  Pattern 1: [0, 0, 0, 3] x1 (Cost: $200 each)
  Pattern 2: [0, 0, 4, 0] x1 (Cost: $200 each)
```

```
  Pattern 3: [0, 0, 4, 0] x1 (Cost: $200 each)
  Pattern 4: [0, 6, 0, 0] x1 (Cost: $200 each)
  Pattern 5: [0, 6, 0, 0] x1 (Cost: $200 each)
  Pattern 6: [12, 0, 0, 0] x1 (Cost: $200 each)
  Pattern 7: [12, 0, 0, 0] x1 (Cost: $200 each)
  Total Cost For Type 3: $1400

Demand Fulfillment:
  S: 50/50 pieces cut
  M: 30/30 pieces cut
  L: 20/20 pieces cut
  XL: 10/10 pieces cut


Total Cost by Stock Type:
  Type 1: $700
  Type 2: $1050
  Type 3: $1400

Total Cost: $3150
Execution Time: 0.0001 seconds
```

Discuss about FFD Heuristic
Advantages:

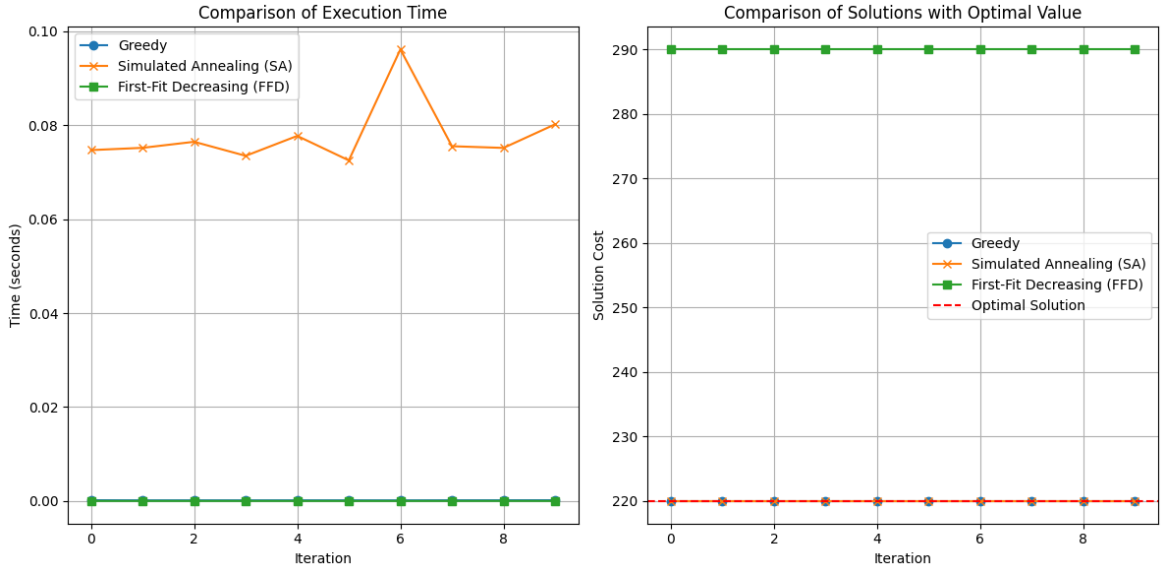
- **Simplicity**: The FFD algorithm is relatively simple to understand and implement.
- **Efficiency:** It generally provides reasonably good solutions in a relatively short amount of time.
- **Worst-Case Performance Guarantee:** In the context of bin packing, FFD has a proven worst-case performance bound, ensuring that the number of bins used is not significantly larger than the optimal solution.

Disadvantages:

- **Suboptimality:** FFD is a heuristic, so it doesn't guarantee finding the absolute optimal solution. There might be cases where other algorithms or approaches could produce better results.
- **Limited Lookahead:** The "first-fit" strategy means that the algorithm doesn't consider future items when making placement decisions, which can sometimes lead to less efficient packing or cutting.

3.4 Performance Evaluation

Data 1:




```
Average Greedy Time: 0.000166 seconds
Average Simulated Annealing Time: 0.079689 seconds
Average FFD Time: 0.000035 seconds
```

→ In **Data 1** (mentioned earlier) and executing **10** runs, both the greedy and simulated annealing (SA) algorithms produce an optimal result of **220** (shown on the right), the FFD produce the resulf of 290. However, SA takes significantly more time than Greedy and FFD (shown on the left).

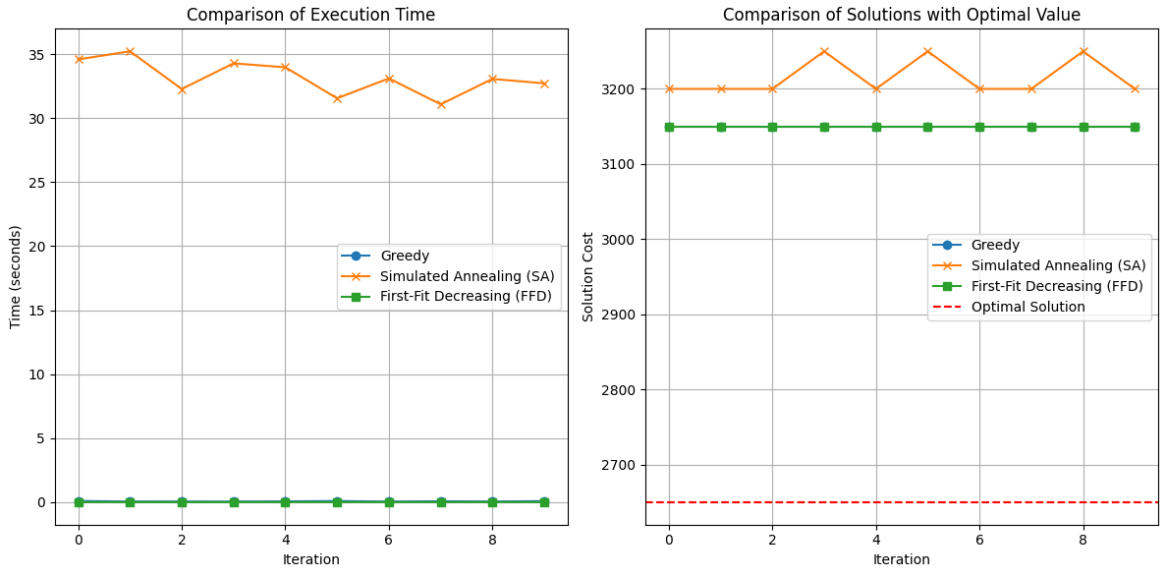
Data 2:


```
Average Greedy Time: 0.006256 seconds
Average Simulated Annealing Time: 4.033568 seconds
```

```
Average FFD Time: 0.000071 seconds
```

→ In **Data 2** and **10** runs, while the greedy and FFD algorithm is very stable, SA exhibits a high degree of variability and its results are not as good as those of the greedy and FFD algorithm in these **10** runs. Additionally, neither algorithm produces an optimal result. SA takes significantly more time than Greedy.

Data 3:




```
Average Greedy Time: 0.055850 seconds
Average Simulated Annealing Time: 33.192665 seconds
Average FFD Time: 0.000157 seconds
```

→ In **Data 3**, Three algorithms produce results that are far from the optimal outcome. In this case, the greedy algorithm still outperforms SA in both result quality and execution time.

In the realm of cutting stock optimization, the Simulated Annealing (SA), Greedy, and First-Fit Decreasing (FFD) algorithms each showcase distinct strengths and trade-offs. SA, while capable of exploring a vast solution space and potentially uncovering near-optimal solutions, exhibits variability due to its probabilistic nature. The Greedy algorithm, prioritizing immediate gains, offers stability and consistent results but might be susceptible to local optima. FFD, with its focus on efficiently allocating larger items first, strikes a balance between solution quality and computational speed.

In our specific test case, Greedy demonstrated superior performance compared to SA, highlighting its effectiveness for this particular problem. However, SA's flexibility to escape local optima remains valuable for complex solution spaces. FFD, while not the absolute best performer, showcased its efficiency by producing good solutions in a shorter time frame, making it attractive for scenarios where computational resources are limited.

Ultimately, the choice between these algorithms hinges on the specific characteristics of the cutting stock problem, including its complexity, data structure, and computational constraints. A comprehensive evaluation across various scenarios is crucial to pinpoint the most suitable ap-

proach, ensuring optimal utilization of resources and achieving high-quality cutting patterns.

Summary:

Criteria	Greedy Heuristic	Simulated Annealing	FFD Heuristic
Search Strategy	Always selects the best option at each step based on current information.	Starts with a random solution and gradually improves, accepting worse solutions with decreasing probability.	Sorts items in decreasing order of size and places them in the first bin that can accommodate them.
Global Optimality	Low, as Greedy is prone to getting stuck in local minima by only choosing the optimal solution at each step.	Higher, due to the ability to escape local minima by temporarily accepting worse solutions.	Moderate; generally provides good solutions but may not find the absolute optimal solution due to its heuristic nature.
Search Time	Typically faster since it only involves a straightforward sequence of decisions.	Generally slower due to the need to evaluate multiple solutions and the potential to revisit previous solutions.	Relatively fast, as it involves a simple placement strategy.
Complexity of Search Space	Suitable for problems with small, simple search spaces.	Suitable for problems with large and complex search spaces, where there are many local minima.	Can handle moderately complex search spaces efficiently.
Effectiveness in Large Search Spaces	Less effective when the search space is large or complex, prone to getting stuck in local minima.	More effective in large and complex search spaces, due to the broad exploration capability.	Can be less effective in very large or complex spaces as it doesn't explore all possibilities.
Escape from Local Minima	Lacks the ability to escape local minima once stuck.	Can escape local minima by temporarily accepting worse solutions.	Limited ability to escape local minima; depends on the initial sorting and bin packing order.
Multi-Objective Optimization	Difficult to apply to multi-objective optimization, as it only focuses on optimizing at each step.	More flexible in multi-objective optimization, thanks to the ability to explore multiple solutions.	Can be adapted for multi-objective optimization with modifications, but primarily focuses on a single objective (e.g., minimizing the number of bins).
Adjustability	Little adjustability after implementation, as the algorithm is simple and direct by nature.	Can adjust various parameters like initial temperature, cooling rate, allowing customization of the search process to improve results.	Some adjustability through different sorting or bin packing strategies.

It is clear that selecting the appropriate algorithm for real-world problems is crucial and highly context-dependent. The choice between Greedy, Simulated Annealing, and the First-Fit

Decreasing (FFD) algorithm depends on various factors such as the specific needs of the problem, the complexity of the solution space, and the available resources. Greedy algorithms, known for their simplicity and efficiency, are often suitable for problems where quick, near-optimal solutions are acceptable and where computational resources are limited. On the other hand, Simulated Annealing, with its ability to explore a broader solution space and escape local minima, is beneficial for more complex problems where higher-quality solutions are needed, albeit at the cost of increased computational time. FFD offers a balance, providing good solutions relatively quickly, making it attractive for scenarios where a compromise between solution quality and computational efficiency is desired. Therefore, companies must carefully assess their problem requirements and resource constraints to determine which algorithm aligns best with their goals and operational capabilities.

4 Conclusion

In this report, we addressed the Cut Stock Problem (CSP) using real-world data and modeled the problem as an Integer Linear Programming (ILP) optimization problem. By applying three different algorithms—Greedy Heuristic, Simulated Annealing, FFD heuristic, we had the opportunity to compare the effectiveness of these methods in finding optimal solutions. The results show that these algorithms can provide reasonable solutions to the CSP; however, each has its own strengths and weaknesses. Greedy Heuristic is efficient and easy to implement but can get trapped in local optima and may fail to find better solutions. Simulated Annealing can escape local optima and search for solutions closer to the global optimum, but it requires longer computation time and depends on parameter selection. FFD , strikes a balance between efficiency and solution quality by prioritizing the placement of larger items, making it a practical choice for scenarios where computational resources are limited.

Regarding scalability, the ILP model and these algorithms have significant potential for application to more complex variants of CSP and other optimization problems in various industries.

- **Multiple Material Types:** The ILP model can be extended to handle multiple types of raw materials with different dimensions and costs. This extension would allow the model to optimize not only the cutting process but also the selection of materials, aiming to minimize overall costs while meeting production requirements.
- **Production and Storage Constraints:** The model can also incorporate more complex constraints related to production capacity, storage limitations, and inventory management. For instance, a steel manufacturing company might need to consider the limited availability of certain cutting machines or the costs associated with storing leftover materials for future use. By including these constraints, the model would provide more realistic and applicable solutions.
- **Variable Costs and Uncertain Demand:** Another extension could involve integrating variable cost factors, such as fluctuating material prices or changes in demand over time. The model could be adapted to include scenarios where demand is uncertain, allowing for the creation of robust solutions that perform well under different market conditions.

For the algorithms:

- **Hybrid Approaches:** The Greedy Heuristic and First-Fit Decreasing could be combined with other optimization techniques, such as dynamic programming or branch and bound, to

overcome its limitations in finding global optimal. This hybrid approach could help explore a broader search space, leading to better overall solutions.

- **Advanced Metaheuristics:** Simulated Annealing could be enhanced by integrating it with other metaheuristic algorithms like Tabu Search, which uses memory structures to avoid revisiting the same solutions, or Genetic Algorithm, which employs evolutionary techniques to explore multiple solutions simultaneously. These combinations could help achieve faster convergence to high-quality solutions in large-scale problems.
- **Parallel Computing:** To address the challenges of large-scale problems, both Greedy Heuristic and Simulated Annealing could be parallelized to run on high-performance computing systems. This approach would significantly reduce computation time and enable the solving of much larger and more complex instances of the CSP.

By implementing these specific extensions, the ILP model and algorithms can be adapted to a wider range of real-world applications, making them more versatile and effective in optimizing resource utilization across various industries. However, for large-scale problems, optimizing computation time and memory becomes a significant challenge. Therefore, further research into complexity reduction methods and parallelizing algorithms to run on high-performance computing systems could be promising directions to expand the applicability of this model and these algorithms.

5 Future Works

1. Improving Algorithmic Efficiency:

Enhanced Metaheuristics: Current algorithms such as Simulated Annealing and Genetic Algorithms could be further refined. Research could focus on hybrid approaches that combine the strengths of multiple metaheuristics (e.g., combining Genetic Algorithms with Simulated Annealing) to achieve better performance.

Parallel and Distributed Computing: Develop parallel versions of algorithms to handle large-scale instances more efficiently. This involves implementing distributed computing techniques to leverage multiple processors or machines.

2. Advanced Mathematical Models:

Multi-Objective Optimization: Extend the model to handle multiple objectives simultaneously, such as minimizing both costs and waste or balancing between cost and delivery time. This could involve the development of multi-objective optimization algorithms and techniques.

Stochastic Models: Incorporate uncertainty and variability in demand and supply. Develop stochastic models that can handle random fluctuations in demand or supply chain disruptions.

3. Algorithm Development:

Algorithmic Innovations: Investigate novel algorithms specifically tailored for large and complex instances of the Cut Stock Problem. This includes developing new heuristic or exact

algorithms that outperform current methods in both accuracy and computational efficiency.

Dynamic Algorithms: Explore algorithms that adapt to changes in demand or stock availability over time, rather than assuming static input data.

4. Integration with Other Optimization Problems:

Combined Problems: Study the integration of the Cut Stock Problem with other related optimization problems, such as scheduling and inventory management. Develop algorithms that can simultaneously solve these interconnected problems to optimize overall operations.

5. Application Areas:

New Industry Applications: Explore the application of Cut Stock Problem solutions in new industries or contexts, such as paper cutting, textile manufacturing, or even digital content management.

Real-Time Applications: Investigate real-time optimization for dynamic environments where demands and stock levels change rapidly. This could involve developing online algorithms or adaptive strategies that respond to real-time data.

6. Data-Driven Approaches:

Machine Learning Integration: Integrate machine learning techniques to predict demand patterns and optimize cutting patterns based on historical data. Develop models that can learn from past data to improve decision-making in the Cut Stock Problem.

Big Data: Handle large-scale datasets using advanced data processing techniques. Research could focus on efficient algorithms for processing and analyzing big data related to cutting patterns and demands.

7. Robustness and Scalability:

Robust Optimization: Develop algorithms that are robust to data inaccuracies or unexpected changes in demand. This involves creating models that are resilient to variations and uncertainties in real-world scenarios.

Scalable Solutions: Focus on algorithms that scale efficiently with problem size. Research could address challenges related to memory usage and computational complexity for very large instances.

By addressing these gaps and exploring these new directions, the research and application of the Cut Stock Problem can be significantly advanced, leading to more efficient and adaptable solutions in a variety of contexts.

6 References

References

[1] G. R. Cerqueira, S. S. Aguiar, and M. Marques, "Modified greedy heuristic for the one-dimensional cutting stock problem," Journal of Combinatorial Optimization, vol. 42, no. 3, pp. 657–674, 2021.

[2] The MO Book Group, "Extra Material: Cutting Stock" https://ampl.com/mo-book/notebooks/05/cutting-stock.html#optimal-cutting-using-known-patterns

[3] SicenseDirect, "Carousel greedy: A generalized greedy algorithm with applications in optimization" https://www.sciencedirect.com/science/article/abs/pii/S0305054817300801

[4] Baeldung, "Simulated Annealing Explained" https://www.baeldung.com/cs/simulated-annealing

[5] ResearchGate, "Pseudo-code for Simulated Annealing algorithm" https://www.researchgate.net/figure/Pseudo-code-for-Simulated-Annealing-algorithm_fig1_288496956