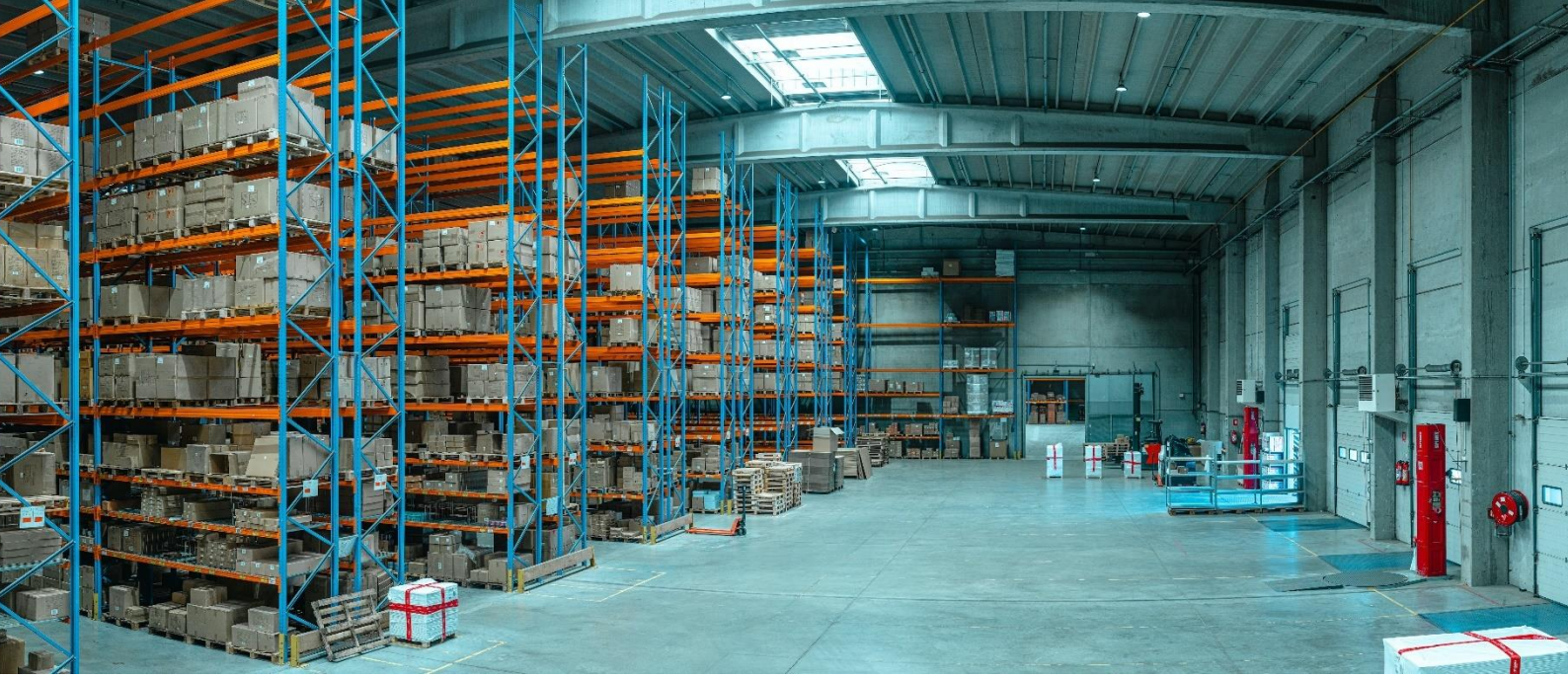


Optimization Algorithms Applied to Line Production Problems

Justification of Research and Implementation

Marit van Megesen, Karina Salanți

Fontys University of Applied Sciences

Department of Applied Mathematics

AMAOR5-B

Supervisor: Roy Willemen

November 13th, 2023

Table of Contents

Introduction

Addressing Line Production's complex scheduling challenges, this paper focuses on creating feasible solutions to optimize production schedules and minimize late delivery penalties. The project involves designing and implementing three distinct solution methods—constructive heuristics, improving search heuristics, and meta-heuristics like simulated annealing—to provide Line Production with a robust, adaptable solution capable of consistently delivering high-quality schedules for diverse datasets. The methodology includes a generic mathematical optimization model, Python implementation, and experimentation on September 2023 and December 2023 datasets.

Math Program

Sets & Indices

$L \triangleq \{1, 2, \dots, 7\}$ $\quad l \triangleq production\ line\ l \in L$

$P \triangleq \{1, 2, \dots, 40\}$ $\quad p \triangleq product\ p \in P$

Decision Variable

$$x_{l,p} \triangleq \begin{cases} 1, & if\ product\ p \in P\ goes\ on\ line\ l \in L \\ 0, & else \end{cases}$$

$h_p \triangleq time\ that\ product\ p \in P\ starts\ processing\ in\ hours$

Parameters

$t_{l,p} \triangleq time\ it\ takes\ to\ process\ product\ p \in P\ on\ line\ l \in L\ in\ hours$

$d_p \triangleq deadline\ of\ product\ p \in P\ \ in\ hours$

$m_p \triangleq penalty\ points\ of\ product\ p \in P\ \ per\ hour$

Math Program

Objective $\quad \min \sum_{p \in P} \sum_{l \in L} x_{l,p} \cdot m_p \cdot (max\left(([h_p + t_{l,p}] - d_p), 0\right))$

s.t. $\quad \sum_{l \in L} x_{l,p} = 1 \quad \forall p \in P \quad$ "every product is processed at one line"

$\sum_{p \in P} x_{l,p} \leq 1 \quad \forall l \in L \quad$ "Every line is processing at most one product at a time"

Research Approach

Literature:

- Rardin, R. L. (2019). *Optimization in operations research*. Pearson India.

External Sources:

- Use of ChatGPT 3.5 for help to implement the Improving Search and Simulated Annealing algorithms and for documentation about dictionary-specific functions in Python (code snippets).
- Use of https://www.geeksforgeeks.org/ for Python documentation.
- Use of https://pandas.pydata.org/docs/ for Pandas DataFrame handling.

- Use of resources provided by supervisor in Canvas (Discrete Improving Search Lectures and Tutorials, ChatGPT provisory codes, Rardin Pseudocode for various algorithms)

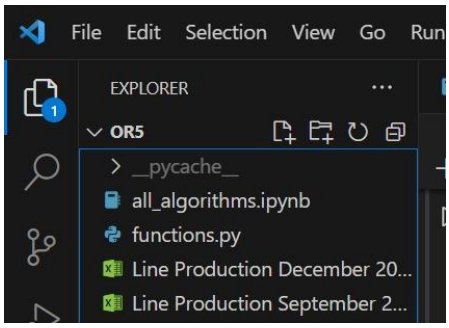
Organization of Code

Our code is thoughtfully structured, emphasizing modularity and clarity. Each heuristic method—constructive heuristic ideal and random, improving search, and simulated annealing—is encapsulated within its function, contributing to a well-organized and easily navigable codebase.

Moreover, detailed documentation practices are evident throughout the code. Comprehensive docstrings are added to every function to explain the input arguments and output. The docstrings serve as a helpful reference for others interacting with the code, encouraging readability and understanding.

Comments have been placed throughout the codebase, offering insights into the logic behind the lines of code. This practice is essential for understanding complex algorithms such as simulated annealing or improving search. The comments guide other programmers, offering insights into decision-making and facilitating easier collaboration and code review.

The code is distributed across two files in the same folder in Visual Studio Code. The `.py` file contains all the functions of the heuristic algorithms; it includes the core logic and main code of the functions used to calculate the schedules. The second file, the `.ipynb` notebook, is a user-friendly interface with markdowns that introduce each heuristic method, emphasizing clarity and ease of understanding. Importantly, this notebook imports the functions from the `.py` file, establishing a seamless connection between the documented explanations and the underlying code. This separation of code and documentation ensures an efficient workflow, promoting collaboration and ease of use for both programmers and end-users.



Constructive Search Algorithms

Ideal Line Algorithm

Design

- This algorithm only uses the DataFrame as a starting point
- Products sorted by a proportion of deadline/penalty cost – this decision prioritizes the products with a low deadline and a high penalty cost to avoid high lateness fees
- Starting with a blank Python dictionary, we fix a free variable with each iteration (one at a time), to get to a feasible solution, without going back for changes
- Decision – we assign each product (in the specified order) to the line where its processing time is the lowest (the 'ideal' line)
- The decision of assigning the 'ideal' line to each product makes a greedy constructive search algorithm – it serves the best purpose to minimize the objective value (total penalty cost)
- The algorithm works instantaneously in Python for the given schedules (September and December)
- The time complexity of this algorithm is O(n), with n=number of products
- The code minimizes hard-coded data for flexibility and future use

Algorithm

Algorithm 12G in Rardin (2014) - Rudimentary Constructive Search

ALGORITHM 12G: RUDIMENTARY CONSTRUCTIVE SEARCH

Step 0: Initialization. Start with all-free initial partial solution $\mathbf{x}^{(0)} = (\#, \ldots, \#)$ and set solution index $t \leftarrow 0$.

Step 1: Stopping. If all components of current solution $\mathbf{x}^{(t)}$ are fixed, stop and output $\hat{\mathbf{x}} \leftarrow \mathbf{x}^{(t)}$ as an approximate optimum.

Step 2: Step. Choose a free component x_p of partial solution $\mathbf{x}^{(t)}$ and a value for it that plausibly leads to good feasible completions. Then, advance to partial solution $\mathbf{x}^{(t+1)}$ identical to $\mathbf{x}^{(t)}$ except that x_p is fixed at the chosen value.

Step 3: Increment. Increment $t \leftarrow t + 1$, and return to Step 1.

Experiment

This code only uses the Dataframe as a starting point and has a greedy decision of sorting and using the 'ideal' line for each product, so for the same dataset, it generates one unique solution. There is no fine parameter tuning involved in this type of algorithm. This algorithm is not robust – it works best with September data, since for December data, L1 is 'ideal' for every product. It is worth noting that this algorithm gives a feasible solution every time (based on how the problem is described). This algorithm has been tested for 27 and 40 products. Below are the solutions for September (left) and December (right, all products on L1).

```
Final Best Solution:
L1: P26, P6, P9, P24
L2: P7, P16, P27, P15, P10
L3: P13, P25
L4: P4, P8
L5: P21, P18, P3
L6: P11, P5, P23, P19, P20, P2, P14
L7: P22, P17, P1, P12
Final Best Penalty Cost: 3653
```

```
Final Best Solution:
L1: P1, P11, P2, P5, P23, P21, P17, P18, P25, P27, P4,
L2:
L3:
L4:
L5:
L6:
L7:
Final Best Penalty Cost: 176744
```

Python Implementation

The function can be found in the file *functions.py.* For help, run the code below:

```
help(constructive_heuristic_ideal)
```

Pseudocode

Calculate the priority for all products (deadline/penalty cost).

Sort products by ascending proportion value.

Prepare the DataFrame with the following information: ideal line, time on ideal line.

Initialize the solution with an empty dictionary:

Repeat**:** (until all products have lines assigned)

For each line, initialize the time (time=0)

For each product: Assign the ideal line for the free product with the highest priority.

Calculate the current time on the assigned line.

Initialize total penalty score = 0.

Repeat: (until the penalty cost for all products has been calculated)

If product deadline < current time:

Then calculate tardiness (deadline – current time) and multiply it by the penalty cost per hour. Update total penalty cost.

Return the obtained feasible schedule and its penalty cost.

Random Line Algorithm

Design

- This algorithm only uses the DataFrame as a starting point
- Products sorted by a proportion of deadline/penalty cost – this decision prioritizes the products with a low deadline and a high penalty cost to avoid high lateness fees
- Starting with a blank Python dictionary, we fix a free variable with each iteration (one at a time), to get to a feasible solution, without going back for changes
- Decision – we assign each product (in the specified order) to a random line from L1 to L7
- The decision of assigning a random line to each product makes a greedy constructive search algorithm – we fix every element of the solution one at a time, while retaining feasibility
- The algorithm works instantaneously in Python for the given schedules (September and December)
- The time complexity of this algorithm is O(n), with n=number of products
- The code minimizes hard-coded data for flexibility and future use

Algorithm

Algorithm 12G in Rardin (2014) - Rudimentary Constructive Search

ALGORITHM 12G: RUDIMENTARY CONSTRUCTIVE SEARCH

Step 0: Initialization. Start with all-free initial partial solution $\mathbf{x}^{(0)} = (\#, \ldots, \#)$ and set solution index $t \leftarrow 0$.

Step 1: Stopping. If all components of current solution $\mathbf{x}^{(t)}$ are fixed, stop and output $\hat{\mathbf{x}} \leftarrow \mathbf{x}^{(t)}$ as an approximate optimum.

Step 2: Step. Choose a free component x_p of partial solution $\mathbf{x}^{(t)}$ and a value for it that plausibly leads to good feasible completions. Then, advance to partial solution $\mathbf{x}^{(t+1)}$ identical to $\mathbf{x}^{(t)}$ except that x_p is fixed at the chosen value.

Step 3: Increment. Increment $t \leftarrow t + 1$, and return to Step 1.

Experiment

This code only uses the Dataframe as a starting point, but uses the greedy decision of assigning a random line to each product, based on their priority. There is no fine parameter tuning involved in this type of algorithm. This algorithm is not very robust – there might be instances where the solution is of bad quality. However, we have tested this algorithm with multiple random seeds (up to 35 experiments) and have decided to use seed(23), since it gives the most optimal solution for the upcoming algorithms, too. It is worth noting that this algorithm gives a feasible solution every time (based on how the problem is described). This algorithm has been tested for 27 and 40 products. Below is a solution for September (left) and the chosen solution for December (right).

```
Final Best Solution:                    Final Best Solution:
L1: P26, P16, P19, P2, P3               L1: P5, P23, P9, P14, P26, P34, P13, P39, P8
L2: P6, P8, P20                         L2: P6, P32, P10, P3
L3: P13, P4, P21                        L3: P11, P17, P4, P31, P30, P37
L4: P11, P17, P25, P1, P23, P10, P14    L4: P18, P25, P40, P19, P29, P28, P24
L5: P7, P5, P24                         L5: P21, P27, P7, P38, P36
L6: P12                                 L6: P33, P12, P15, P20, P22
L7: P22, P27, P9, P15, P18              L7: P1, P2, P35, P16
Final Best Penalty Cost: 46829          Final Best Penalty Cost: 20841
```

Python Implementation

The function can be found in the file *functions.py.* For help, run the code below:
`help(constructive_heuristic_random)`

Pseudocode

Calculate the priority for all products (deadline/penalty cost).

Sort products by ascending proportion value.

Prepare the DataFrame with the following information: chosen random line, time on chosen line.

Initialize the solution with an empty dictionary:

Repeat: (until all products have lines assigned)

For each line, initialize the time (time=0)

> For each product: Assign the random line for the free product with the highest priority.
>
> Calculate the current time on the assigned line.

Initialize total penalty score = 0.

Repeat: (until the penalty cost for all products has been calculated)

> If product deadline < current time:
>
> > Then calculate tardiness (deadline – current time) and multiply it by the penalty cost per hour. Update total penalty cost.

Return the obtained feasible schedule and its penalty cost.

Improving Search Algorithm

Design

- This algorithm uses an initial solution as a starting point. To retain the flow of the optimization project, we use our random constructive heuristic solution to start the improving search algorithm.
- Decision – we swap two products with each other, calculate the new penalty cost and check to see if an improvement has been made from the initial cost.
- This decision of swapping products to check for improvement makes an improving search algorithm – with every swap a calculation is make and if it did make an improvement it takes the current solution to be the best solution, if it did not make an improvement the best solution stays unchanged.
- This algorithm ensures reaching a local optimum but terminates once it achieves this state, raising uncertainty about whether the obtained solution is a global or local optimum

- The algorithm works within a minute in Python for the given schedules (September and December)
- The time complexity of this algorithm is: iterations * O(n^2), with n=number of products and iterations = the amount of iterations the loop takes with swapping
- The code minimizes hard-coded data for flexibility and future use

Algorithm

Algorithm 12C in Rardin (2014) – Discrete Improving Search

ALGORITHM 12C: DISCRETE IMPROVING SEARCH

Step 0: Initialization. Choose any starting feasible solution $\mathbf{x}^{(0)}$, and set solution index $t \leftarrow 0$.
Step 1: Local Optimum. If no move $\Delta\mathbf{x}$ in move set $\mathcal{M}$ is both improving and feasible at current solution $\mathbf{x}^{(t)}$, stop. Point $\mathbf{x}^{(t)}$ is a local optimum.
Step 2: Move. Choose some improving feasible move $\Delta\mathbf{x} \in \mathcal{M}$ as $\Delta\mathbf{x}^{(t+1)}$.
Step 3: Step. Update

$$\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} + \Delta\mathbf{x}^{(t+1)}$$

Step 4: Increment. Increment $t \leftarrow t + 1$, and return to Step 1.

Experiment

The code initializes its process with an initial solution, which can take various forms as long as it is feasible. Two other essential arguments are the data containing all product-related information and the specified number of iterations (swaps). We conducted testing using initial solutions generated by both the constructive heuristic random and ideal methods and observed their robustness across different datasets. Through experimentation with varying iteration counts (20, 40, and 200), we observed that the algorithm generally converges to a local optimum within the first 20 swaps for most solutions. The image on the right shows the best solution of the September data, and the image on the left shows the best solution for the December data.

```
Final Best Solution:
L1: P26, P6, P24, P15
L2: P7, P16, P10, P19
L3: P13, P25, P9, P14
L4: P4, P8, P11, P2
L5: P21, P18, P3, P27
L6: P5, P23, P20
L7: P22, P17, P1, P12
Final Best Penalty Cost: 2305
```

```
Final Best Solution:
L1: P23, P13, P39, P8, P32, P17, P40, P29, P27
L2: P6, P10, P3, P5, P34, P31, P35
L3: P11, P4, P30, P37
L4: P18, P19, P24, P2
L5: P21, P7, P38, P36
L6: P33, P12, P15, P20, P22, P9, P14, P26, P25, P28
L7: P1, P16
Final Best Penalty Cost: 10894
```

Python Implementation

The function can be found in the file *functions.py.* For help, run the code below:
`help(improving_search)`

Pseudocode

Repeat: (amount of iterations)

Create copies of the solution and calculate penalty cost of initial solution

Set improvement to be false to check later if it changed

Repeat: (get every line and their products from the current best solution)

Repeat: (get every product in the products in the current line of the current solution)

Repeat: (get different product in different line of current solution)

Repeat: (check if current line is not the same as the other line)

Create a copy of the best solution, call it new solution

Repeat: (check if product is in the new solution line)

Remove product from line 1 and add to line2

Add them to the new solution

Calculate the new solution penalty cost

Repeat: (check if penalty cost is better)

Create new best solution with copy

Create new best penalty cost

Set improvement to true

Repeat: (is improvement true?):

Set starting solution to best solution

If it is not true then break the whole loop because it's a t local optimum (and improvement is false)

Print initial solution and the improved solution

Return the improved solution

Simulated Annealing Algorithm

Design

- Simulated Annealing uses an initial solution as a starting point. To retain the flow of the optimization project, we use our discrete improving search solution to start the simulated annealing algorithm. Additionally, it uses the DataFrame to retrieve information about processing times.
- Additional arguments: initial temperature, cooling rate (between 0 and 1), iterations per temperature
- The decision to use the Simulated Annealing algorithm was made because of its ability to accept non-improving solutions, wider search space, implementation simplicity and directness, ability to work with discrete optimization problems with non-linear objective functions
- This code applies 2-exchange neighborhood – swapping elements of the initial solution and accepting them based on defined criteria
- This algorithm accepts non-improving solutions based on the acceptance factor e^((-Δ)/T) - as the temperature cools, it is less likely to accept non-improving solutions
- This algorithm guarantees a high-quality schedule (a local optimum), but the outcome depends on the chosen initial solution (thus we use improving search solution – another local optimum)

- The algorithm works within 10 minutes in Python for the given schedules (September and December)
- The time complexity of this algorithm is: $\log_{10}$(cooling_rate)*iterations_per_temp*initial_temperature*O(n^2)
- The code minimizes hard-coded data for flexibility and future use

Algorithm

Algorithm 12E in Rardin (2014) – Simulated Annealing Search

ALGORITHM 12E: SIMULATED ANNEALING SEARCH

Step 0: Initialization. Choose any starting feasible solution $\mathbf{x}^{(0)}$, an iteration limit $t_{\max}$, and a relatively large initial temperature $q > 0$. Then set incumbent solution $\hat{\mathbf{x}} \leftarrow \mathbf{x}^{(0)}$ and solution index $t \leftarrow 0$.

Step 1: Stopping. If no move $\Delta\mathbf{x}$ in move set $\mathcal{M}$ leads to a feasible neighbor of current solution $\mathbf{x}^{(t)}$, or if $t = t_{\max}$, then stop. Incumbent solution $\hat{\mathbf{x}}$ is an approximate optimum.

Step 2: Provisional Move. Randomly choose a feasible move $\Delta\mathbf{x} \in \mathcal{M}$ as a provisional $\Delta\mathbf{x}^{(t+1)}$, and compute the (possibly negative) net objective function improvement Δobj for moving from $\mathbf{x}^{(t)}$ to $\left(\mathbf{x}^{(t)} + \Delta\mathbf{x}^{(t+1)}\right)$ (increase for a maximize, decrease for a minimize).

Step 3: Acceptance. If $\Delta\mathbf{x}^{(t+1)}$ improves, or with probability $e^{\Delta \text{obj}/q}$ if Δobj ≤ 0, accept $\Delta\mathbf{x}^{(t+1)}$ and update

$$\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} + \Delta\mathbf{x}^{(t+1)}$$

Otherwise, return to Step 2.

Step 4: Incumbent Solution. If the objective function value of $\mathbf{x}^{(t+1)}$ is superior to that of incumbent solution $\hat{\mathbf{x}}$, replace $\hat{\mathbf{x}} \leftarrow \mathbf{x}^{(t+1)}$.

Step 5: Temperature Reduction. If a sufficient number of iterations have passed since the last temperature change, reduce temperature q.

Step 6: Increment. Increment $t \leftarrow t + 1$, and return to Step 1.

Experiment

This code uses a feasible initial solution and its DataFrame. Since this function has multiple arguments, we performed some fine parameter tuning – changing the initial temperature and the iterations per temperature, and found that, most of the time initial temperature of 100 and 10 iterations per temperature were a good compromise between a quality solution and a time efficient code. This way, the code works within 2 minutes. This algorithm is very robust and always finds a high-quality schedule, given enough iterations and a high initial temperature. We have tested this algorithm with 27 and 40 products. Below are the results of a high-quality September schedule (left) and the best quality December schedule (right).

```
Final Best Solution:
L1: P26, P6, P24, P15
L2: P7, P16, P10, P27
L3: P13, P11, P12, P14
L4: P4, P23, P25, P8
L5: P21, P18, P9, P2
L6: P19, P5, P20
L7: P22, P17, P1, P3
Final Best Penalty Cost: 634
```

```
Final Best Solution:
L1: P1, P12, P13, P20, P10, P17, P31, P40, P37
L2: P3, P8, P6, P5, P22, P24, P30
L3: P11, P35, P27, P38
L4: P23, P39, P28, P36
L5: P18, P16, P15, P33
L6: P4, P2, P7, P29, P25, P19, P9, P14, P26, P32
L7: P21, P34
Final Best Penalty Cost: 2365
```

Python Implementation

The function can be found in the file *functions.py.* For help, run the code below:

```
help(simulated_annealing)
```

Pseudocode

Check if the cooling rate is between 0 and 1 (otherwise break the code).

Initialize current schedule and best yet schedule as the initial schedule.

Initialize the best cost as the current cost.

While: temperature > 0.1

    Repeat: (amount of iterations per temperature)

    Generate a random sample of two products and swap them to get a neighbor schedule.

    Calculate the neighbor schedule cost and compare it to the current cost (delta).

    If: delta < 0 or randomly generated possibility (between 0 and 1)< e**(-delta / temperature)

        Then accept the neighbor solution as the next best solution

Cool the temperature according to the cooling rate.

Return the generated high-quality schedule.

Additional Function

The calculate_penalty_cost function takes two arguments: a DataFrame (df) containing production times, deadlines, and costs for each product and a schedule dictionary (schedule) representing a feasible solution. It calculates the total penalty cost of the given schedule. The function iterates through the schedule, considering processing times, deadlines, and penalty costs for each product on its allocated production line. It computes the lateness of each product, calculates the associated penalty cost if it is late, and accumulates the penalty costs for each production line. The function returns the total penalty cost of the given schedule.

The solution_excel function also takes two arguments: a DataFrame (df) containing production times, deadlines, and costs for each product and a schedule dictionary (schedule) representing a feasible solution. The function generates a pandas DataFrame containing the solution's data with nine columns: Product, Line, Start, Process Time, End, Deadline, Tardiness, Penalty Cost per Hour, and Total Penalty Cost. It iterates through the solution dictionary, extracting relevant data from the DataFrame for each product and calculating time-related variables. The information for each product is stored in a list of dictionaries (result_data). This list is then converted into a DataFrame (result_df), providing a comprehensive overview of the schedule and associated data.

Test Data and Results

Algorithm:	Constructive Heuristic	Improving Search	Simulated Annealing
September 2023 Data (27 Products)			
Penalty Cost	3653	2305	634
Computation Time (seconds)	0s	28s	65s
December 2023 Data (40 products)			
Penalty Cost	20841	10894	2365
Computation Time (seconds)	0s	49s	92s

Even though the simulated annealing algorithm is the most time consuming heuristic, we advise to use this method for optimal results.

Realized Depth in Operations Research

We implemented the following Operations Research techniques:

- Constructive Heuristic Ideal Algorithm
- Constructive Heuristic Random Algorithm
- Discrete Improving Search Algorithm
- Simulated Annealing Algorithm
- Compare heuristics

Throughout the project, we focused on writing efficient code with user-friendly functions. For this purpose, we used docstrings (for documentation) and comments, but also created a user-friendly, simple interface where the code can be tested and implemented for any dataset.