Exercise Report: Units 1–5

Joan Fernández Navarro

HEURISTIC ALGORITHMS
IN TRANSPORT AND FINANCE

MUCEIM

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Contents

List of Figures

List of Tables

List of Algorithms

The code developed to complete all the proposed exercises for this report is available in the online GitHub repository: https://github.com/JoanFer030/heuristic_exercises

1 Unit 1

1.1 01a. Optimization Modelling 101

1.1.1 Optimization Methods

Optimization methods are essential tools for finding the best solution to complex problems. These problems typically involve maximizing or minimizing an objective function. The choice of the appropriate method depends on the problem's nature, these methods can be categorized into three main families: Exact Methods, Metaheuristics, and Approximate Methods.

Exact methods
These algorithms are designed to find the global optimal solution to an optimization problem, guaranteeing its optimality. They are ideal for problems where optimality is critical, but can become computationally prohibitive for NP-hard problems.

- **Branch and X**
  - Branch and Bound: It divides the original problem into smaller subproblems (branching) and uses bounds (upper or lower limits on the objective function) to discard subproblems that cannot contain the optimal solution (pruning), thereby reducing the search space.
  - Branch and Cut: This method combines the Branch and Bound scheme with the addition of "cutting planes" (valid inequalities) to tighten the linear relaxation of the problem at each node, leading to better bounds and more efficient pruning.
  - Branch and Price: Used for problems with a vast number of variables (. It combines Branch and Bound with a pricing subproblem that identifies promising new variables to add to the model.
- **Constraint Programming.** Approach where relationships between variables are expressed as constraints. The algorithm searches for variable assignments that satisfy all constraints, using techniques like constraint propagation and backtracking to efficiently reduce the search space.
- **Dynamic Programming.** A method for solving complex problems by breaking them down into a sequence of simpler, overlapping subproblems. It solves each subproblem only once and stores its solution (memoization), thus avoiding recomputation. It is particularly effective for problems that exhibit the Bellman's Principle of Optimality, where an overall optimal solution contains within it optimal solutions to its subproblems (shortest path).

- **A* and IDA*.** They combine the actual cost from the start node with a heuristic estimate of the cost to the goal (like greedy best-first search). A* is complete and optimal if the heuristic is admissible. IDA* (Iterative Deepening A*) is a variant that uses less memory by performing an iterative deepening depth-first search.

Metaheuristics
These are general-purpose strategies that guide the search process. They do not guarantee finding the global optimum but are extremely useful for solving NP-hard problems or where the exact methods fail liki in highly complex, nonlinear, or non-convex problems.

- **Single-Solution Based Metaheuristics.** These methods start from a single solution and, through iterations, perturb or modify it to explore the surrounding search neighborhood. Example: GRASP which maintains and improves a single candidate solution in each iteration of its search.

- **Population-Based Metaheuristics.** Instead of a single solution, these methods maintain and improve a set (population) of candidate solutions over iterations, allowing for a broader exploration of the search space. Example: Genetic Algorithms based on natural evolution using selection, crossover, and mutation.

Approximate methods
These are algorithms designed to find "good enough" solutions in a reasonable amount of time, especially for NP-hard problems where finding the exact optimal solution would require exponential time.

- **Approximation Algorithms.** These algorithms are often designed for specific problem classes and provide theoretical guarantees on the quality of the found solution. For instance, an approximation algorithm can guarantee that its solution is within a known multiplicative factor of the optimal value.

- **Problem-Specific Heuristics.** These are customized, highly specialized algorithms that exploit the particular structure of a given problem to find good solutions very quickly. They are usually not generalizable to other problems but are highly effective for the ones they are designed for.

1.1.2 Task planning optimization (Multi-period DP)

The objective of this exercise was to implement a problem using Multi-period Dynamic Programming (DP). Inspired by the example presented in the lecture, we chose task scheduling optimization as the focus.
This problem is based on planning a project composed of multiple tasks with dependencies and execution times. The goal is to determine the necessary tasks to reach the final task; it is not required to complete all tasks. Instead, the focus is on identifying the tasks that minimize the total time to reach the final task. Therefore, the problem can be defined as determining the list of tasks that minimize the time needed to complete

the final task.
The algorithm, like the problem itself, aims to minimize the total time to complete the final task in a project with dependent tasks. The idea is to consider each task as a state within a multi-period decision process.

Algorithm Steps

1. **Definition.** Each task of the project is considered a state, and each period represents the moment when the task can start depending on the completion of its predecessor tasks.

2. **Initialization.** The final task has a time of 0, and structures are created to store the minimum times for each task and their optimal successors.

3. **Successor evaluation.** For each task, all its successor tasks are analyzed, and the total time to the goal is calculated if each successor is chosen.

4. **Optimal selection.** The successor that minimizes the total time is selected, storing both the minimum time and the next task in the optimal path.

5. **Backward recursion.** The process is applied recursively from the final task to the initial tasks, reusing previously computed results.

6. **Minimum path reconstruction.** Using the stored optimal successors, the path from the initial task to the final task that produces the minimum total time is reconstructed.

Mathematical Formalization
Let $f(T_i)$ be the minimum time from task i (T_i) to the final task (T_f).

$$f(T_i) = \begin{cases} 0 & \text{if } T_i = T_f\text{\$1em]} \min_{T_j \in \text{successors}(T_i)}(\text{duration}(T_i, T_j) + f(T_j)) \\ \text{otherwise} \end{cases}$$

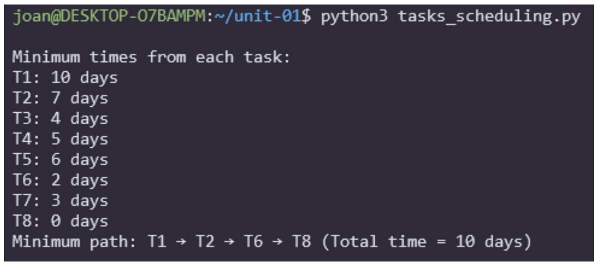
This definition allows us to apply backward recursive dynamic programming, efficiently optimizing the total time to complete the project.

Example: Task Scheduling Problem
As an example, we consider a project composed of 8 tasks (Table 1), where each task has its corresponding successor tasks, representing dependencies, along with their execution times. In this example, Task 1 represents the initial state, that is, the starting task, while Task 8 represents the final task, marking the completion of the project.

The execution of the problem-solving script generates the following output, providing the minimum times required to reach the final task from each task in the project. As observed in Figure 1, the minimum time to go from Task 1 to Task 8 is 10 days, with the minimum path being T1 → T2 → T6 → T8. This means that starting from Task 1, following Task 2 and Task 6 leads to the final task in the shortest possible time. The

Task	Successors	Duration (days)
T1	T2, T3, T4	3, 6, 7
T2	T5, T6	4, 5
T3	T6	2
T4	T6, T7	3, 4
T5	T8	6
T6	T8	2
T7	T8	3
T8	-	0

Table 1: Task scheduling example with dependencies and durations



```
joan@DESKTOP-O7BAMPM:~/unit-01$ python3 tasks_scheduling.py

Minimum times from each task:
T1: 10 days
T2: 7 days
T3: 4 days
T4: 5 days
T5: 6 days
T6: 2 days
T7: 3 days
T8: 0 days
Minimum path: T1 → T2 → T6 → T8 (Total time = 10 days)
```

Figure 1: Terminal output of the multi-period dynamic programming algorithm.

output also shows the minimum times for all other tasks, allowing a complete overview of the project's structure and dependencies.
Focusing on Task 4 as an example, there are two possible routes to reach the final task: T4 $\rightarrow$ T6 $\rightarrow$ T8 and T4 $\rightarrow$ T7 $\rightarrow$ T8. Calculating the total time for each route, we have T4 $\rightarrow$ T6 $\rightarrow T8 = 3 + 2 = 5$ days and T4 $\rightarrow$ T7 $\rightarrow T8 = 4 + 3 = 7$ days. The minimum of these two options is 5 days, so the shortest path from Task 4 is through Task 6, resulting in shortest(T4) $= 5$ days. This illustrates how the algorithm evaluates multiple successor options and selects the one that minimizes the total project duration.

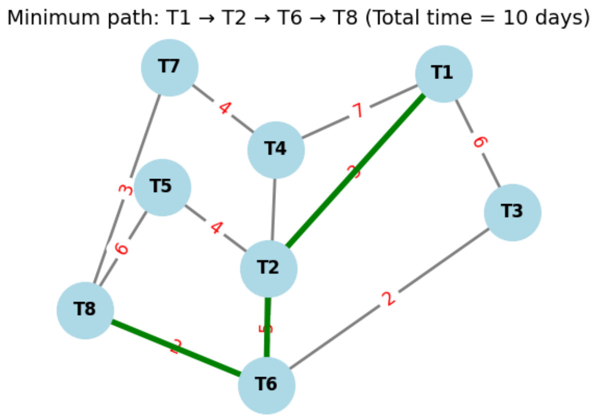


Figure 2: Graphical representation of the task scheduling problem.The minimum path is highlighted in green.

Finally, Figure 2 provides a graphical representation of the project, where each node displays a task. All dependencies are shown as gray edges with their corresponding execution times, and the minimum path is highlighted in green. This visualization clearly identifies the critical tasks, the sequence required to achieve the shortest total project duration, and alternative routes that do not contribute to the minimum time.

1.1.3 Diet Problem (LP)

The diet optimization problem is a classic in linear programming, it seeks to find the combination of foods that minimizes total cost while satisfying basic nutritional requirements.
The mathematical formulation balances economic cost with nutritional quality, considering multiple nutrients simultaneously: energy, proteins, fats, carbohydrates, and fiber. Each food contributes differently to these nutrients with varying costs, creating a multi-constraint optimization problem that only mathematical methods can solve efficiently.

Problem Data

Table 2: Food Nutritional Composition and Costs, expressed per kilogram.

Food	Cost	Energy	Proteins	Fats	Carbs	Fiber
Rice	1.20 €	3600 kcal	70 g	10 g	790 g	30 g
Chicken	5.80 €	1650 kcal	310 g	35 g	0 g	0 g
Lentils	2.10 €	3200 kcal	230 g	15 g	550 g	110 g
Apples	1.80 €	520 kcal	3 g	2 g	140 g	25 g
Yogurt	3.50 €	610 kcal	37 g	33 g	47 g	0 g

Table 3: Daily Nutritional Requirements

Nutrient	Minimum	Maximum
Energy (kcal)	2000	2500
Proteins (g)	50	-
Fats (g)	30	70
Carbohydrates (g)	250	400
Fiber (g)	25	-
Total weight (kg)	-	1.5

Mathematical Formalization

- **Decision Variables**
  Let x_i represent the kilograms of food i to be consumed daily, for $i = 1, 2, \ldots, 5$.

- **Objective Function**

$$\text{Min } Z = 1.20x_1 + 5.80x_2 + 2.10x_3 + 1.80x_4 + 3.50x_5$$

- **Nutritional Constraints**

$$\begin{aligned}
\text{Energy: } & 2000 \leq 3600x_1 + 1650x_2 + 3200x_3 + 520x_4 + 610x_5 \leq 2500 \\
\text{Proteins: } & 70x_1 + 310x_2 + 230x_3 + 3x_4 + 37x_5 \geq 50 \\
\text{Fats: } & 30 \leq 10x_1 + 35x_2 + 15x_3 + 2x_4 + 33x_5 \leq 70 \\
\text{Carbohydrates: } & 250 \leq 790x_1 + 0x_2 + 550x_3 + 140x_4 + 47x_5 \leq 400 \\
\text{Fiber: } & 30x_1 + 0x_2 + 110x_3 + 25x_4 + 0x_5 \geq 25 \\
\text{Total weight: } & x_1 + x_2 + x_3 + x_4 + x_5 \leq 1.5 \\
\text{Non-negativity: } & x_i \geq 0, \quad i = 1, 2, \ldots, 5
\end{aligned}$$

Results

Table 4: Optimal Diet Composition

Food	**Quantity**	**Subtotal Cost**
Rice	0.298 kg	0.358 €
Lentils	0.146 kg	0.306 €
Yogurt	0.752 kg	2.633 €
Chicken	0.000 kg	0.000 €
Apple	0.000 kg	0.000 €
Total	**1.197 kg**	**3.298 €**

Table 5: Nutritional Constraints Analysis

Nutrient	**Minimum**	**Maximum**	**Actual**
Energy (kcal)	2000	2500	2000
Proteins (g)	50	–	82.3
Fats (g)	30	70	30.0
Carbs (g)	250	400	351.3
Fiber (g)	25	–	25.0
Weight (kg)	–	1.5	1.197

The linear programming model successfully identified a cost-optimal diet composition that satisfies all nutritional requirements while minimizing total expenditure. The solution reveals several important patterns in constraint utilization and resource allocation. As shown in Table 4, the optimal solution utilizes only three of the five available foods: rice, lentils, and yogurt. The total cost of 3.298 € represents the minimum possible expenditure to meet all nutritional constraints. Notably, yogurt is the largest portion of the diet (0.752 kg, 62.8% of total weight) and contributes the majority of the cost (2.633 €, 79.8% of total cost), despite being the second most expensive food per kilogram. This counterintuitive result happens because yogurt provides a balanced combination of proteins, fats, and energy efficiently.

The complete exclusion of chicken and apples from the optimal solution, as evidenced by zero quantities, demonstrates the model's cost-driven selection process. Chicken is too expensive relative to its nutritional contribution, while apples are likely excluded due to their low energy density and limited protein content.

Table 5 reveals that three constraints are binding (active) in the optimal solution: energy, fats, and fiber. The energy constraint is precisely at its lower bound of 2000 kcal, indicating that increasing the minimum energy requirement would necessarily increase the diet cost. Similarly, the fat content is exactly at the minimum requirement of 30g, and fiber is precisely at 25g, its minimum threshold.
Several constraints show significant slack, particularly proteins (32.3g above minimum) and total weight (0.303kg below maximum). This slack indicates that these constraints are not limiting factors in the cost optimization. The carbohydrate constraint operates in the middle of its allowable range (351.3g vs. 250-400g range), suggesting it's naturally satisfied by the cost-optimal food combination rather than actively constraining the solution.
The status of the weight constraint (1.197 kg vs. 1.5 kg maximum) demonstrates that the model prioritizes nutrient density over volume, selecting foods that provide maximum nutritional value per kilogram rather than simply filling the weight allowance.

1.1.4 Production Planning with Setup Costs (IP)

The production planning problem with setup costs is a integer programming (IP) problem where we must decide which equipment to rent and how many units to produce of each product to maximize profit, subject to capacity and resource constraints.

Problem Data

Table 6: Product Data and Resource Requirements

Product	Labor	Cloth	Price	Var Cost	Fixed Cost	Effective Cap
Shirts	2.0	3.0	35 €	20 €	1,500 €	1,500 units
Shorts	1.0	2.5	40 €	10 €	1,200 €	2,250 units
Pants	6.0	4.0	65 €	25 €	1,600 €	667 units
Skirts	4.0	4.5	70 €	30 €	1,500 €	1,000 units
Jackets	8.0	5.5	110 €	35 €	1,600 €	500 units

Table 7: Resource Availability

Resource	Availability
Labor Hours	4,000 hours
Cloth	4,500 units

Mathematical Formulation

- **Decision Variables**
  Let x_p represent the number of units of product p to produce, and y_p indicate whether to rent equipment for product p, for $p \in \{\text{Shirts, Shorts, Pants, Skirts, Jackets}\}$.

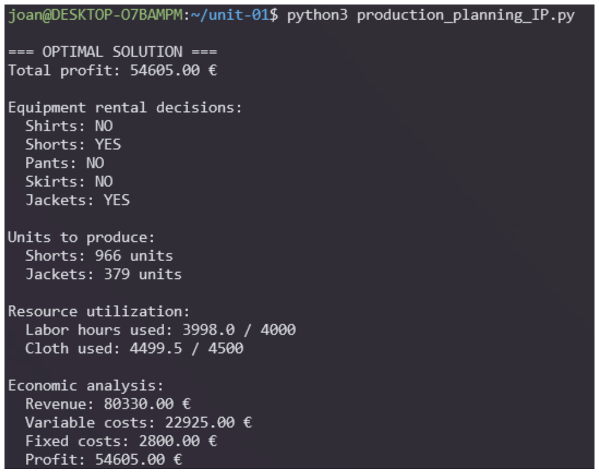
- **Objective Function**

$$\begin{aligned}\text{Max } Z = {} & (35-20)x_1 + (40-10)x_2 + (65-25)x_3 + (70-30)x_4 + (110-35)x_5 \\ & - 1500y_1 - 1200y_2 - 1600y_3 - 1500y_4 - 1600y_5\end{aligned}$$

- **Production Constraints**

$$\begin{aligned}\text{Capacity constraints: } & x_1 \leq 1500y_1 \\ & x_2 \leq 2250y_2 \\ & x_3 \leq 666.67y_3 \\ & x_4 \leq 1000y_4 \\ & x_5 \leq 500y_5 \\ \text{Labor constraint: } & 2.0x_1 + 1.0x_2 + 6.0x_3 + 4.0x_4 + 8.0x_5 \leq 4000 \\ \text{Cloth constraint: } & 3.0x_1 + 2.5x_2 + 4.0x_3 + 4.5x_4 + 5.5x_5 \leq 4500 \\ \text{Variable types: } & x_1, x_2, x_3, x_4, x_5 \geq 0 \text{ and integer} \\ & y_1, y_2, y_3, y_4, y_5 \in \{0, 1\}\end{aligned}$$

Results



```
joan@DESKTOP-O7BAMPM:~/unit-01$ python3 production_planning_IP.py

=== OPTIMAL SOLUTION ===
Total profit: 54605.00 €

Equipment rental decisions:
  Shirts: NO
  Shorts: YES
  Pants: NO
  Skirts: NO
  Jackets: YES

Units to produce:
  Shorts: 966 units
  Jackets: 379 units

Resource utilization:
  Labor hours used: 3998.0 / 4000
  Cloth used: 4499.5 / 4500

Economic analysis:
  Revenue: 80330.00 €
  Variable costs: 22925.00 €
  Fixed costs: 2800.00 €
  Profit: 54605.00 €
```

Figure 3: Terminal output of the production planning problem solved using PYOMO.

After solving the production planning problem with both methods, we observe a significant difference in the obtained results. While Pyomo identified an optimal solution of 54,605 € through a mixed strategy producing both Shorts and Jackets, OpenSolver (Table 9) converged to a suboptimal solution of 52,800 € based exclusively on Shorts production.

Table 8: Optimal Production Plan - OpenSolver Results

Product	**Rent Equipment**	**Units Produced**	**Contribution**
Shirts	No	0	0 €
Shorts	Yes	1,800	54,000 €
Pants	No	0	0 €
Skirts	No	0	0 €
Jackets	No	0	0 €
Total	**1 product**	**1,800 units**	**54,000 €**

Table 9: Economic Analysis - OpenSolver Results

Item	**Amount**
Revenue	72,000 €
Variable Costs	18,000 €
Fixed Costs	1,200 €%
Total Profit	**52,800 €**

This 1,805 € difference reveals fundamental limitations in OpenSolver's configuration for this type of mixed-integer programming problem with fixed costs. Despite extensive troubleshooting efforts including logical constraints with multiplication, testing multiple solver models, and attempting to force constraints through various formulations, OpenSolver consistently failed to identify the global optimum.

After unsuccessful attempts to obtain an optimal solution with OpenSolver, the analysis will focus exclusively on the results generated by Pyomo, which successfully identified the globally optimal production strategy.
As shown in the results (Figure 3), the optimal solution utilizes only two of the five available products: Shorts and Jackets. The total profit of 54,605 € represents the maximum achievable given the production constraints. Notably, the solution achieves near-perfect resource utilization with labor and cloth at 99% of available capacity, demonstrating exceptional optimization efficiency.
The complete exclusion of Shirts, Pants, and Skirts from the optimal production plan, as evidenced by zero production quantities, can be due to two reasons either unfavorable profit margins relative to resource consumption or capacity limitations that make them less attractive than the optimal product mix.
The near-binding status of both labor and cloth constraints indicates that these resources are the primary limiting factors in the production system. Any increase in either resource's availability would directly translate to increased profitability, while reductions would force trade-offs between the two product lines.

1.1.5 Non-Smooth Optimization

Juan et al., 2020 discusses how the use of Biased-Randomized Algorithms can be effective for solving non-smooth Optimization Problems (OPs). In this context, non-smooth OPs frequently arise when introducing soft constraints that allow certain restrictions to be violated in exchange for incurring a penalty cost. This penalization, which is not always linear or continuous, transforms the objective function into a non-smooth problem, posing significant challenges for traditional optimization methods. The authors demonstrate how these algorithms can be effectively integrated into practical domains such as logistics, transportation, and scheduling, where capacity, time, and resource constraints can be realistically relaxed.
Going in to detail about the practical cases, the firts domain is logistics. The article addresses the Facility Location Problem (FLP) and, instead of treating warehouse capacities as hard limits, introduces a non-smooth cost function that allows exceeding nominal capacity at the expense of a penalty. This approach mirrors real-world conditions, where additional storage can be subcontracted or direct shipments arranged once in-house capacity is saturated.
In the transportation domain, the same principle is applied to the Vehicle Routing Problem (VRP) and the Arc Routing Problem (ARP). The authors model situations in which delivery times or vehicle capacities can be slightly exceeded, with penalties proportional to the degree of violation. For instance, a 15-minute delivery delay may entail a smaller cost than redesigning the entire route, allowing for globally more efficient solutions.
For production scheduling, the paper introduces a failure-risk cost into the Flow-Shop Problem. Rather than merely minimizing the makespan, the objective function incorporates the cost associated with the risk of machine failure when operating continuously without maintenance. This yields a more realistic balance between productivity and operational sustainability, where sequences that appear less time-efficient may become optimal once accelerated equipment wear and associated risks are taken into account.
Finally, the results show that incorporating soft constraints through this approach leads to lower overall-cost solutions, as the benefits of strategically violating certain restrictions outweigh the associated penalty costs. This yields more flexible models that more accurately reflect real-world operational conditions.

The article successfully presents a valuable adaptation of classical optimization problems into more realistic formulations, effectively incorporating the notion of soft constraints through non-smooth cost functions. This approach more faithfully captures the trade-offs present in real operational contexts, where the rigidity of traditional constraints often proves limiting. Moreover, it convincingly demonstrates that the use of Biased-Randomized Algorithms not only enables the handling of the complexity associated with non-differentiable functions, but also achieves superior results compared to traditional methods, both in solution quality and computational efficiency. The proposed framework is particularly relevant for practical decision-making, where flexibility and adaptability are essential.

1.2 01b. Intro to X-Heuristic Optimization

1.2.1 Simheuristics

The video shows how simheuristics, which are the combination of metaheuristics and simulation, can improve the efficiency (optimization) of everyday activities. By integrating these two approaches, simheuristics make it possible to handle problems that involve uncertainty and variability, which are common in real-life operations. As the video illustrates, simheuristics can be used to enhance public transportation planning, goods distribution, or telecommunication networks, leading to more reliable and efficient systems. Overall, the video highlights how simheuristics provide practical tools to optimize daily processes in a more realistic and adaptive way.

Juan et al., 2018 presents a comprehensive review of the simheuristics concept and its potential applications within supply chain management. The authors explain that simheuristics combine metaheuristics with simulation layers to address optimization problems under uncertainty, where stochastic variables—such as random demands or travel times—affect system performance. The paper focuses primarily on describing this hybrid methodology and its possible implementations, without providing specific numerical or comparative results.
The review outlines several domains where simheuristics can be applied: in logistics, to tackle facility location problems with stochastic costs; in transportation, to solve vehicle routing problems with random demands, using simulation to assess the reliability of solutions; and in production planning, to address flow-shop scheduling problems with stochastic processing times. In each case, the article explains how the simulation layer enables the evaluation and refinement of candidate solutions generated by metaheuristics. However, it does not include quantitative data regarding the performance of these approaches.
The paper's main contribution is conceptual and methodological, focusing on establishing the potential value of simheuristics for managing uncertainty in complex optimization problems. Rather than presenting empirical validations or concrete application results, the authors aim to position simheuristics as a promising framework for integrating optimization and stochastic analysis in decision-making processes.measures.

1.2.2 Learnheuristics

Calvet et al., 2017 introduce the concept of learnheuristics, a hybrid methodology that combines metaheuristics with machine learning to address combinatorial optimization problems with dynamic inputs. In such problems, input parameters are not fixed but evolve predictably according to the structure of the solution being iteratively constructed by a heuristic. For instance, a customer's demand may decrease if they are visited later in a delivery route. The proposed learnheuristic framework leverages predictive models trained on historical data to dynamically update problem inputs during the heuristic solution-construction process.
The authors illustrate the potential applicability of this methodology across several domains. In transportation, customer demand or travel times may depend on the order

of visits within a route. In logistics, assigning customers to different distribution centers could influence their purchasing behavior. In production, processing times may vary depending on the job sequence assigned to a machine. In finance, the risk and return of assets in a portfolio can change depending on the current composition of the portfolio itself.
To validate the proposed approach, the paper presents a numerical experiment based on a vehicle routing problem (VRP) in which each customer's demand is dynamic and depends on its position within the route. The authors compare the classical Clarke and Wright heuristic (with fixed demands) to a learnheuristic version that employs a linear regression model to predict and update demands dynamically. The results show that the learnheuristic approach achieves a significantly lower total cost (791.26 vs. 896.86) and requires fewer routes (8 vs. 11), demonstrating the advantages of incorporating dynamic input behavior into the optimization process.
Overall, the article proposes an adaptation for real-world optimization problems, where dynamic inputs are common yet rarely modeled explicitly. The proposed learnheuristic framework effectively captures dependencies between decision variables and evolving system parameters, yielding higher-quality solutions. The methodology is presented in a generic and flexible way, allowing for adaptation across a wide range of applications.

1.2.3 Transportation

Reyes-Rubiano et al., 2019 address the problem of routing electric vehicles with stochastic travel times (EVRPST), a variant of the VRP that considers both the limited range of electric vehicles and uncertainty in travel times. The objective is to minimize the total expected time to complete the distribution, including possible penalties for route failures when a vehicle runs out of battery power prematurely. To solve this NP-hard problem, the authors propose a simheuristic algorithm that combines Monte Carlo simulation with a multi-start framework incorporating a biased randomization heuristic based on the Clarke & Wright savings algorithm. The methodology also introduces the concept of "energy safety stock," reserving a percentage of battery capacity to cover possible unforeseen delays. Computational tests on 27 benchmark instances demonstrated that:

- Deterministic solutions are suboptimal in stochastic environments.
- The simheuristic approach generates more reliable solutions with lower expected costs.
- Safety stock levels between 20% and 25% optimize the balance between reliability and cost, reducing the expected travel time by up to 11.44% compared to not using safety stock.

The study concludes that this approach allows for the design of more robust distribution plans for electric vehicle fleets in realistic urban environments with uncertainty.
Score: 9

1.2.4 Finance

Doering et al., 2022 analyze how biased randomization algorithms (BRA) and simheuristics can be applied to solve NP-hard combinatorial optimization problems in the financial and insurance fields, especially under conditions of uncertainty. BRA transform greedy constructive heuristics into probabilistic algorithms by using biased probability distributions (such as geometric), which allows for efficient exploration of the solution space and generates multiple high-quality solutions in reduced computational times, even when parallelizing executions. On the other hand, simheuristics combine metaheuristics with simulation to handle stochastic versions of these problems, where components such as asset returns or correlations are modeled as random variables. Successful applications include the optimization of rich and stochastic investment portfolios, asset and liability management (ALM) in uncertain environments, and the design of parametric catastrophe insurance. The main advantage of these methodologies is their ability to provide managers with a set of robust, high-quality solutions that take into account the uncertainty inherent in financial markets, overcoming the limitations of deterministic models and exact methods in large-scale problems with realistic constraints. The article concludes by highlighting the potential of these techniques to improve decision-making in the sector and suggests future lines of research, such as combining them with machine learning (learnheuristics) and fuzzy logic.
Score: 7

2 Unit 2. Random Search and CWS Heuristic

2.1 Random Search

Random Search (RS) is a simple stochastic optimization technique that explores the solution space by sampling candidate solutions using a random distribution. Random Search does not require derivative information, making it suitable for non-smooth, discontinuous, or black-box functions where the analytical form of the objective function is unknown or too complex. The main idea behind Random Search is to generate random solutions, evaluate their objective values, and keep track of the best one found so far. Despite its simplicity, it can achieve competitive results in low-dimensional problems or when the evaluation of solutions is computationally inexpensive.
Each solution is sampled independently of previous ones, typically using a uniform probability distribution across the search space. As can be seen in the Algorithm 1 the RS is a memoryless algorithm as it does not build a trajectory or keep record of previous solutions, but rather retains only the best solution observed so far. This makes the algorithm highly explorative, as it covers the entire space evenly, but at the cost of not exploiting promising regions in depth. Because of its stochastic nature, two executions of the algorithm may produce different results, and performance is generally measured through averages across multiple runs.

Algorithm 1 Random Search Algorithm

```
Require: Objective function $f(x)$, search domain $D$, number of iterations $N$
Ensure: Best solution found $x_{best}$
  $x_{best} \leftarrow$ random solution from $D$
  for $k = 1$ to $N$ do
      $x_{candidate} \leftarrow$ random solution from $D$
      if $f(x_{candidate}) > f(x_{best})$ then
          $x_{best} \leftarrow x_{candidate}$
      end if
  end for
```

2.2 Clarke & Wright Savings

The Clarke and Wright Savings (CWS) Heuristic, introduced in 1964, is one of the most widely used constructive heuristics for solving the Vehicle Routing Problem (VRP), an NP-hard optimization problem that seeks the most efficient set of routes for a fleet of vehicles delivering goods to a set of customers from a central depot.
The main idea behind the algorithm is to start with a naïve solution, where each customer is served individually by a vehicle returning directly to the depot, and then iteratively merge routes whenever doing so reduces the total cost. The term “savings” refers to the reduction in total travel cost achieved by combining two routes into one. For every pair of customers i and j, the savings value $s(i, j)$ is computed as:

$$s(i, j) = c(0, i) + c(0, j) - c(i, j)$$

where:

- $c(0,i)$ and $c(0,j)$ are he costs from the depot to customers i and j, respectively.
- $c(i,j)$ is the cost of traveling directly between customers i and j.

Algorithm 2 Clarke & Wright Savings Algorithm

```
Require: Distance matrix $C$, vehicle capacity $Q$, customer demands $d_i$
Ensure: Set of routes minimizing total distance
  $s(i,j) = c(0,i) + c(0,j) - c(i,j) \quad \forall i,j \quad i \neq j$        ▷ Compute all savings
  $savings_{sorted} \leftarrow sort((i,j), by = s(i,j))$    ▷ Prioritizing higher savings (descending)
  $R_i = (0,i,0),\ \forall i \in \mathcal{C}$                         ▷ Initialize a separate route
  for $(i,j)$ in $savings_{sorted}$ do
      if $can\_be\_merged(R_i, R_j)$ then          ▷ without violating capacity constraints
          $merge(R_i, R_j)$
      end if
  end for
  return final set of merged routes
```

As observed in Algorithm 2, the CWS heuristic begins by constructing an initial, trivial solution in which each customer is individually served by a separate vehicle returning directly to the depot. The algorithm then calculates a savings value for every possible pair of customers, representing the reduction in total distance achieved if those two customers are served on the same route rather than separately. Once all savings are computed, they are sorted in descending order, giving priority to the pairs that produce the greatest cost reduction.
The algorithm then proceeds iteratively: for each pair of customers, it checks whether their corresponding routes can be merged without violating vehicle capacity or route feasibility constraints. If merging is allowed, the two routes are combined into a single one, thereby reducing the total travel cost. This process continues until no further feasible merges can be made. The resulting set of routes forms the final solution, which is typically much more efficient than the initial configuration.

2.3 SR-1 Algorithm for CVRP

Faulin et al., 2008 presents SR-1, a simulation-based heuristic algorithm for solving the Capacity-Constrained Vehicle Routing Problem (CVRP). The proposal combines an initial solution (such as that of the Clarke & Wright algorithm) with Monte Carlo simulation to generate multiple random but oriented solutions. The method analyzes the initial solution to statistically model the distribution of distances between consecutive nodes on the routes. Then, in each iteration, SR-1 samples this distribution to generate a “desired distance” and selects the next customer whose distance is closest to that value, thus exploring a space of solutions that are structurally similar to known good solutions. A key advantage is that SR-1 provides a set of alternative solutions, allowing the decision maker to consider multiple criteria. In tests with instances of 20

and 50 customers, the algorithm improved on the results of the initial heuristic. The authors highlight its flexibility for real-world constraints but acknowledge the need for validation on more extensive benchmarks.
Score: 8

2.4 Basin Function Problem

The Basin Function Problem is a continuous optimization problem that is commonly used to evaluate the performance of optimization algorithms. It belongs to the family of unimodal benchmark functions, meaning it has a single global minimum and no local minima, making it ideal for analyzing convergence behavior and algorithmic efficiency in simple, smooth landscapes. The function is defined as:

$$f(x) = \sum_{i=1}^{n} x_i^2$$

The objective of the problem is to find the point in the search space that minimizes $f(x)$. Since the function is simply the sum of squared components, the minimum value occurs when each component $x_i = 0$.

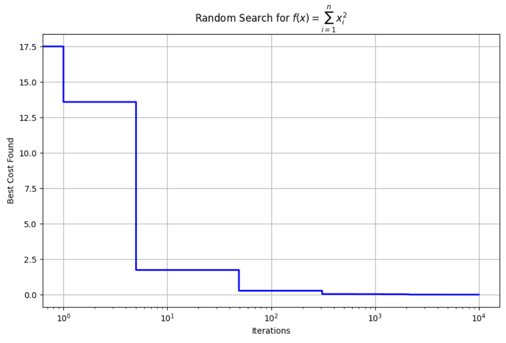
Results

For this example, we use the previously defined Basin Function, subject to the following constraints:

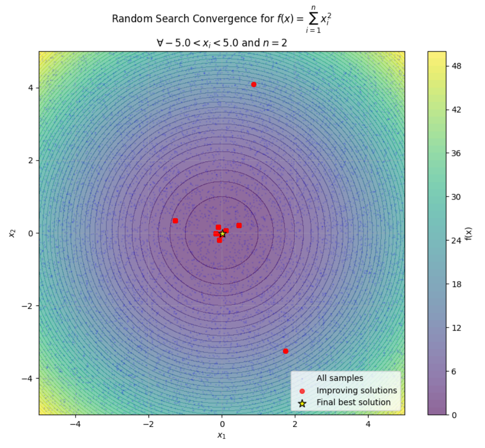
$$\forall - 5.0 < x_i < 5.0 \text{ and } n = 2$$

Since $n = 2$, the problem is defined in a two-dimensional search space, meaning that the input vector is $x = (x_1, x_2)$Consequently, the global minimum is located at $x^* = (0, 0)$ where the function reaches its minimum value $f(x^*) = 0$. To solve this optimization problem, we apply the Random Search algorithm with a maximum of $10,000$ iterations. As shown in Figure 4.a, the initial cost associated with the first random evaluation in the search space is approximately 17.5. As the algorithm progresses, the cost of the best solution gradually decreases, converging towards the global minimum around iteration $1,000$, after which it remains stable for the rest of the process.
In Figure 4.b, each red point represents a sampled solution within the search space. The plot illustrates how these points are initially scattered across the domain but progressively converge toward the center, approaching the optimal point as improvements occur. After 10,000 iterations, the best solution found was achieved after a total of eight improvements in the best-so-far value, with a total execution time of 1.325×10^{-2} seconds.

$$x^* = [0.010597993044246579, -0.014392368852528037]$$
$$f(x^*) = 0.00031945773775311181$$



(a) Convergence curve of the Random Search algorithm applied to the Basin Function.



(b) Visualization of the search process in the two-dimensional space. Red points represent sampled solutions.

Figure 4: Solution process for 2-dimensional Basin Function Problem using Random Search.

Table 10: Results of five independent runs of the Random Search algorithm applied to the Basin Function problem. Each run was performed with 10,000 iterations.

Run	Best Solution Found	Cost	Elapsed Time (s)
1	(-0.0222, -0.0359)	0.001786	0.01426
2	(-0.0085, 0.0055)	0.000102	0.01437
3	(-0.0511, 0.0455)	0.004678	0.01332
4	(0.0106, -0.0144)	0.000319	0.01325
Average	—	**0.001221**	**0.01380**

As summarized in Table 10, the results obtained across the five independent runs of the Random Search algorithm show highly consistent performance. The algorithm reliably converges to near-optimal solutions within a very short computational time—approximately 0.0138 seconds on average. The best costs achieved in each run range between 10^{-4} and 4.7×10^{-3}, confirming the algorithm's ability to locate solutions very close to the global minimum of the Basin Function.
The variability observed between runs is minimal and primarily due to the stochastic nature of Random Search, which samples the solution space differently each time. Despite this randomness, all runs successfully identify points near the true minimum at $x^* = (0, 0)$, with cost values approaching zero. These results demonstrate the robustness and reliability of Random Search in simple, convex landscapes, achieving consistent convergence behavior across repeated executions.

2.5 The Knapsack Problem

The Knapsack Problem (KP) is one of the most fundamental and well-studied problems in combinatorial optimization. The objective is to select a subset of items to include in a knapsack such that the total value (or profit) of the chosen items is maximized, while ensuring that the total weight does not exceed the knapsack's capacity.

In practical terms, the problem models situations where a limited resource such as space, weight, or budget must be allocated in the most profitable way possible. The decision is binary for each item, either it is included in the knapsack or it is not. This formulation defines the 0–1 Knapsack Problem, which is NP-hard, meaning that the computational effort required to find the exact optimal solution increases exponentially with the number of items.

Mathematical Formulation

- **Decision Variables**
  Let x_i be a binary variable that indicates whether item i is included in the knapsack:
  $$x_i = \begin{cases} 1 & \text{if item } i \text{ is selected, \$4pt] 0} \\ \text{otherwise,} & \end{cases} \qquad i \in \{1, \ldots, n\}.$$

- **Objective Function**
  $$\text{Max } Z \; = \sum_{i=1}^{n} v_i \, x_i$$
  where v_i denotes the value (profit) of item i.

- **Capacity Constraint**
  $$\sum_{i=1}^{n} w_i \, x_i \; \leq \; W$$
  where w_i is the weight (or resource consumption) of item i and W is the knapsack capacity.

- **Variable Domain**
  $$x_i \in \{0, 1\}, \qquad \forall i = 1, \ldots, n.$$

Greedy Baseline

To benchmark the performance of the Random Search algorithm, a Greedy Solution is also implemented as a comparison baseline. The Greedy approach is a simple constructive heuristic that selects items based on their value-to-weight ratio, in other words, the relative profit per unit of weight. The intuition behind this strategy is that items providing the highest economic return per unit of capacity should be prioritized for inclusion in the knapsack.
The algorithm begins by computing the ratio $r_i = \frac{v_i}{w_i}$ for each item, then sorts all items in descending order according to this ratio. Items are sequentially added to the knapsack while there remains available capacity. The process stops when no additional item can fit without exceeding the maximum weight limit. Although this heuristic does not always guarantee an optimal solution it generally provides high-quality solutions with very low computational effort, making it a standard baseline for comparison.

Random Instance Generation

To evaluate both methods under multiple conditions, random instances of the Knapsack Problem are generated. As can be seen in Algorithm 3, each random instance contains a selected number of items, in this case we will use 20 items, with random weights and values uniformly distributed between 1 and 100. Then, the total capacity of the knapsack is set to 20% of the total weight of all generated items, ensuring a moderately constrained optimization space.

Algorithm 3 Random Knapsack Instance Generation

```
Require: Number of items $n_{items}$, capacity ratio $capacity_{r}atio$, random seed $seed$
Ensure: Lists of weights, values, and total capacity
  Generate $weights_i \sim U(1, 100)$ for $i = 1, \ldots, n_{items}$
  Generate $values_i \sim U(1, 100)$ for $i = 1, \ldots, n_{items}$
  Compute total weight $W_{total} = \sum weights_i$
  Compute knapsack capacity $C = int(W_{total} \times capacity_{ratio})$
  return $(weights, values, C)$
```

Results

To evaluate both approaches, the Knapsack Problem was solved across several randomly generated instances using the Random Search algorithm (with a total of 100,000 iterations) and compared against the Greedy baseline.

Table 11: Comparison of Random Search and Greedy solutions for randomly generated Knapsack instances with 20 items. Each Random Search execution used 100,000 iterations.

Run	Greedy Profit	Random Search Profit	Execution Time (s)
1	352	353	1.745
2	603	603	1.821
3	511	484	1.820
4	577	532	1.867
5	473	459	1.958
Average	–	–	**1.842**

As summarized in Table 11, the results reveal that the Greedy algorithm consistently achieves slightly higher profits than the Random Search method, confirming the high effectiveness and efficiency of the Greedy approach for this optimization problem. The Greedy solution exploits the deterministic selection of items based on their value-to-weight ratio, which often leads to near-optimal solutions with negligible computational effort.

In contrast, the Random Search algorithm produces results that, while slightly inferior, remain surprisingly competitive given its stochastic and uninformed nature. The average profit obtained by Random Search is only about 5% lower than that achieved by the Greedy method, highlighting its ability to explore the solution space effectively even without any problem-specific heuristic guidance.

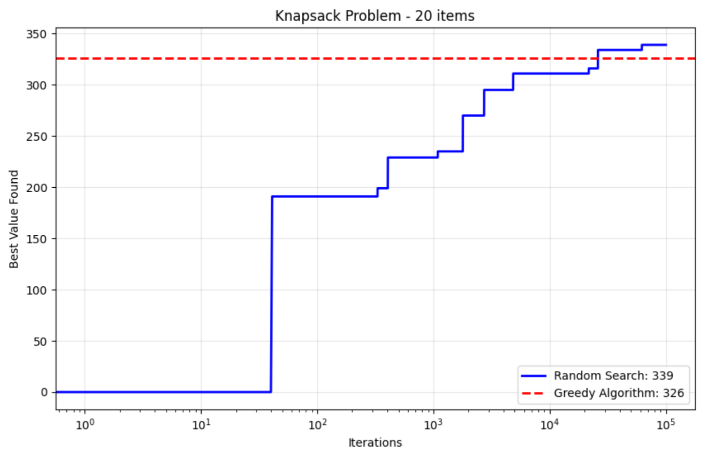


Figure 5: Convergence curve of the Random Search algorithm for a random instance of the Knapsack Problem.

The execution times for all runs are also very consistent, averaging around 1.842 seconds for 100,000 iterations, demonstrating that Random Search remains computationally lightweight despite the large number of evaluations.
Focusing on the new random execution, which is illustrated in Figure 5, we observe that the best solution achieved by Random Search reached a profit of 329 using 216 units of capacity, which is a slightly better solution that the one obtained with the greedy algorithm, reaching a profit of 326 using 211 units of capacity. The convergence curve shows a pattern of gradual and continuous improvements throughout the iterations. Initially, the profit increases rapidly as the algorithm discovers better configurations, followed by a slower refinement phase where improvements become more incremental. Notably, the curve shows long plain areas, indicating regions where no better solutions are found for several consecutive iterations. This behaviour reflects the random and memoryless nature of the algorithm, in other words, the improvements occur sporadically and depend purely on chance.
Despite this, the algorithm usually does not surpass the deterministic Greedy baseline, Random Search still proves capable of producing good-quality, feasible, and diverse solutions, making it a valuable reference for evaluating the performance of more advanced metaheuristic approaches.

2.6 Vehicle Routing Problem

The Vehicle Routing Problem (VRP) is a fundamental NP-hard optimization problem that generalizes the classical Traveling Salesman Problem (TSP) to multiple vehicles. The main objective of the VRP is to determine the optimal set of routes for a fleet of vehicles that must deliver goods or services to a set of geographically dispersed customers, starting and ending at a central depot. Each customer must be visited exactly once by one vehicle, and the total travel cost—typically distance or time—must be minimized. In addition to cost minimization, the VRP includes operational constraints, such as limited vehicle capacity, customer demand or maximum route duration.

Mathematical Formulation

- **Parameters and Sets**
  Let:

  $$N = \{1, 2, ..., n\} \text{ be the set of customers}$$
  $$V = \{0\} \cup N \text{ be the set of all nodes, where 0 represents the depot}$$

  Let $c_{i,j}$ denote the distance of traveling from node i to node j , and let d_i represent the demand of customer i.
  Each vehicle has a maximum capacity Q, and the number of available vehicles is K.

- **Decision Variables**
  Define the binary variable:

  $$x_{ij} = \begin{cases} 1 & \text{if a vehicle travels directly from node } i \text{ to node } j, \\ 0 & \text{otherwise.} \end{cases}$$

  Additionally, let y_i denote the cumulative load of the vehicle upon arriving at customer i.

- **Objective Function**
  The goal is to minimize the total travel cost across all vehicles:

  $$\text{Minimize } Z = \sum_{i \in V} \sum_{j \in V, j \neq i} c_{ij}\, x_{ij}$$

- **Constraints**

$$\begin{aligned}
&\sum_{j \in V, j \neq i} x_{ij} = 1, && \forall i \in N \quad \text{(each customer is visited once)} \\
&\sum_{j \in N} x_{0j} \leq K, && \text{(limit number of vehicles)} \\
&y_j \geq y_i + d_j - Q(1 - x_{ij}), && \forall i, j \in N, i \neq j \quad \text{(capacity constraint)} \\
&d_i \leq y_i \leq Q, && \forall i \in N \quad \text{(load bounds)} \\
&x_{ij} \in \{0, 1\}, \quad y_i \geq 0, && \forall i, j \in V
\end{aligned}$$

Implementations and Instances

To solve the Vehicle Routing Problem (VRP) we implemented a graph-based solution using the NetworkX library in Python. This approach allows for a highly flexible

representation of routes and a more efficient evaluation of possible merges between customer routes during the optimization process.
In this implementation, each node in the graph represents a customer, characterized by its corresponding demand, while node 0 represents the central depot. The edges between nodes are weighted by the Euclidean distance between the corresponding customer locations, which serves as the measure of travel cost. This structure enables efficient computation of route costs, adjacency relations, and feasible merges, which are essential for both the Random Search and Clarke & Wright Savings (CWS) heuristics.
To systematically evaluate the performance of both the Random Search and CWS implementations, we used a set of benchmark instances commonly employed. These instances are taken from the benchmark set proposed in the slides. These instances are designed where the first part contains a letter that represents the instances family (A, E, P, etc), that is the geographical distribution of demand points; the second part contains the number of customers; and the last part contains the size of the fleet, the number of vehicles. An additional parameter must be introduced that represents the capacity of the vehicle.
For example, instance A-n61-k9 includes 61 customers distributed across a large area and a fleet of 9 vehicles. These instances enable a fair comparison between the two methods, assessing not only their solution quality (measured in total cost) but also their computational efficiency and convergence behavior under identical conditions.

Results

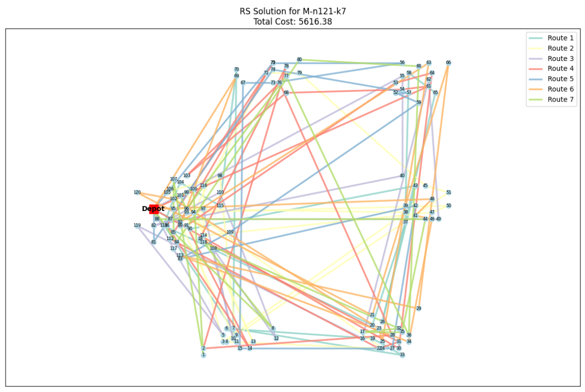
To evaluate both algorithms, several experiments were conducted across five different instances, each characterized by distinct geographical distributions, a varying number of customers (ranging from 50 to 121), and different fleet sizes. For the Random Search (RS) algorithm, the number of iterations was fixed at $100,000$ to ensure a comparable computational effort across all scenarios.

Table 12: Comparison of results between Dummy Solution, Random Search (RS), and Clarke & Wright Savings (CWS) heuristics for different VRP instances.

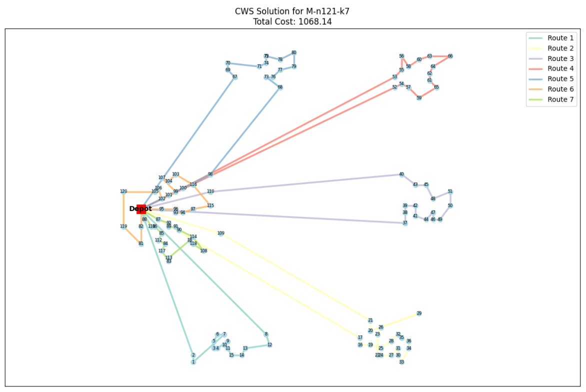
Instance	**Dummy**	**RS**	**Time (s)**	**CWS**	**Time (s)**
A-n80-k10	11153.81	4081.86	13.8585	1860.94	0.0135
B-n50-k7	3871.27	2049.93	8.4952	748.80	0.0172
E-n76-k14	3630.86	2210.07	14.3603	1054.60	0.0158
M-n121-k7	12239.31	5616.38	19.7814	1054.60	0.0158
P-n70-k10	3340.28	2011.30	12.2217	896.86	0.0071

As summarized in Table 12, the Clarke & Wright Savings (CWS) heuristic consistently demonstrates superior performance across all instances. In terms of computational efficiency, the contrast between both approaches is particularly striking: while CWS completes its execution in a matter of hundredths of a second, Random Search requires execution times ranging between 8 and 20 seconds, especially in instances with a larger number of customers.

When comparing solution quality, CWS again outperforms RS by a significant margin. On average, the total route costs obtained by CWS are approximately 2.5 times better than those achieved by Random Search. This highlights the strength of CWS heuristic, which efficiently exploit spatial and structural characteristics of the problem to generate high quality and geographically coherent solutions, while RS relies only on stochastic exploration. In addition, the design of the CWS heuristic converts this algrithm in a robust algorithm because it always generates the same solution for the same problem.



(a) Solution obtained for instance M-n121-k7 using Random Search.



(b) Solution obtained for instance M-n121-k7 using Clarke & Wright Savings.

Figure 6: Comparison of solutions obtained for instance M-n121-k7.

Focusing on a specific example, the M-n121-k7 instance, this difference becomes visually evident. As illustrated in Figure 6, the solution produced by CWS exhibits a clear geographical coherence: customers located in separate clusters are grouped into distinct routes that do not overlap, and customers close to the depot are efficiently divided into two balanced routes on either side of it. In contrast, the solution generated by Random Search shows no geographical structure, routes cossing the entire plane, connecting distant customers arbitrarily without forming coherent clusters. This chaotic pattern is expected, given the algorithm's random nature and the lack of problem-specific guidance in its search process.

3 Unit 3. Biased-Randomized Algorithms

3.1 Biased-Randomized Algorithms (BRA)

Biased-Randomized Algorithms (BRA) are constructive heuristics in which, at each step, the current solution is iteratively improved by adding a new element or performing a local modification. Since the decision rule follows a deterministic logic, typically based on a greedy criterion, the algorithm would normally generate the same solution in every execution. However, the introduction of randomness in the selection process allows the algorithm to produce different solutions across runs while maintaining its constructive nature.
In BRAs, instead of restricting the candidate list to the top-ranked options, a probability distribution is used to assign different probabilities of selection to each potential movement or candidate in the sorted list. Thus, the best-ranked elements are still more likely to be chosen, but every candidate maintains a non-zero probability of selection, promoting exploration of the solution space.

Algorithm 4 Biased-Randomized Algorithm (BRA)

```
Require: Candidate list $L$, sort criterion, probability distribution
Ensure: Constructed solution $S$
  $S \leftarrow \emptyset$                                  ▷ Initialize empty solution
  Create candidate list $L$ according to greedy ranking
  while termination condition not met do
      $c_i \leftarrow BRA\_choice(L, P(i))$                 ▷ Select candidate according to a $P(i)$
      if $c_i$ is feasible then
          $S \leftarrow S + c_i$                            ▷ Add candidate to the solution
          Update the candidate list $L$
      end if
  end while
  return $S$
```

As shown in Algorithm 4, the procedure begins by initialazyng an empty solution and generating a candidate list sorted according to a criterion. This criterion depends on the problem, for example, for the Knapsack Problem we will sort based on ratio between profit and weight. For each iteration, the algorithm samples one candidate from the list according to a biased probability distribution $P(i)$, instead of always taking the top-ranked candidate. As we will see, the shape and the specific parameters of $P(i)$ vary based on the distribution chosen.
If the selected candidate can be feasibly added to the current solution, it is incorporated, and the candidate list is updated accordingly. The process continues until the stopping criterion is met, which could be a complete feasible solution, a fixed number of iterations, or a computational time limit.
This controlled randomization introduces variability between runs while maintaining the greedy structure that ensures solution quality.

Probability Distributions

The key component of a Biased-Randomized Algorithm is the probability distribution that governs the selection of candidates from the sorted list. Several types of distributions can be used depending on the desired balance between exploration and exploitation.

- **Geometric Distribution.** This distribution allows the algorithm to favour elements with higher ranks in the sorted list while still maintaining a controlled level of randomness that enables exploration of alternative solutions.

  $$P(i) = (1 - \beta)^{i-1} \cdot \beta$$

  Where i represents the position on the sorted list and where $\beta \in (0, 1)$ is a shape parameter that controls the degree of greediness (determines the rate of decay of the probability curve):

  - When $\beta \to 1$, the algorithm behaves almost deterministically, favoring the top candidate.
  - When $\beta \to 0$, the selection becomes nearly uniform, resulting in a highly random search.

- **Discretized Triangular Distribution.** Assigns decreasing probabilities to lower-ranked candidates but without requiring an explicit control parameter. This method introduces a softer randomization effect and can be useful when a more flexible, adaptive selection mechanism is desired.

3.2 SR-GCWS Hybrid Algorithm

Juan et al., 2010 presents SR-GCWS, a hybrid algorithm for solving the Capacity-Constrained Vehicle Routing Problem (CVRP) that combines the classic Clarke & Wright heuristic with Monte Carlo simulation. Its methodology introduces controlled randomness through a quasi-geometric distribution to select edges during route construction, prioritizing those with the greatest savings but allowing diversified exploration without the need for prior parameter adjustment. The main advantage of the algorithm is its ability to generate, in reduced computational times, a set of high-quality alternative solutions that equal or exceed the best known solutions in benchmarks of up to 121 nodes, demonstrating an average gap of -0.21%. This flexibility allows the decision maker to consider additional criteria beyond cost minimization, making SR-GCWS a practical and competitive tool compared to more complex algorithms.
Score: 7

3.3 The Knapsack Problem enhanced with BR

To evaluate the Biased-Randomized Algorithm (BRA) in the Knapsack context (see Section 2.5 for the problem introduction and mathematical formulation), we perform a controlled experiment using the geometric selection distribution with shape parameter

$\beta = 0.3$. The aim is to measure the effect of biased randomization on solution quality, variability, and computational cost, and to compare the performance of BRA against the Greedy baseline and the results of the Random Search reported earlier.

Results

To ensure a fair and comparable evaluation, we build a solution using the Greedy, the RS and the BRA algorithm under equivalent experimental conditions. Specifically, we performed five independent runs, each using a newly generated random knapsack instance following the same configuration described in Section 2.5, but extended to 50 items per instance.

Table 13: Comparison of Greedy, Random Search, and BRA ($\beta = 0.3$) performance across five random knapsack instances with 50 items.

Run	Greedy	Time (s)	RS	Time (s)	BRA	Time (s)
1	1307	0.0000	632	4.1249	1307	14.8137
2	1132	0.0000	627	4.1441	1141	14.8160
3	1337	0.0000	925	4.1243	1337	14.4385
4	1240	0.0000	673	4.2416	1245	15.7696
5	1322	0.0000	792	4.1100	1322	15.3482
Average	–	**0.0000**	–	**4.1490**	–	**15.0372**

As shown in Table 13, the execution times obtained using the BRA are on average about three times higher than those achieved with Random Search (RS). This increase in computational cost is mainly due to the fact that the BRA procedure is repeatedly applied to the same problem over $100,000$ iterations, with the goal of continuously storing and updating the best-found solution.
However, when this repeated execution is removed (when BRA is executed only once per instance) the results remain very similar to those obtained through repeated runs, while the execution time drops dramatically to below 10^{-4} seconds. This demonstrates that the higher computational effort of BRA stems primarily from its iterative sampling process rather than from the complexity of each evaluation step.
Focusing on the quality of the obtained solutions, the performance of BRA is comparable to that of the greedy approach, which reaches the optimal or near-optimal solution in minimum time.
In contrast, when comparing the overall performance of the Random Search in comparison with the Greedy results we detect a different behaviour to those previously observed using RS with smaller instances of 20 items (as shown in Table 11), a clear degradation is observed. Whereas RS previously achieved solutions within approximately 5% of the greedy solution, it now fails to reach even 50% of that quality in some cases.
This behavior is directly related to the increase in the problem dimensionality, as the number of items grows, the search space expands exponentially, making it more challenging for pure random search to effectively explore and identify high-quality solutions. In contrast, the probabilistic bias introduced in BRA allows it to better guide the exploration process, maintaining high-quality results even as problem complexity increases.

3.4 Travelling Salesman Problem

The Traveling Salesman Problem (TSP) is a classical NP-hard combinatorial optimization problem that serves as the foundation for many routing and logistics problems. The main objective of the TSP is to determine the shortest possible route that allows a single salesman to visit each city exactly once and return to the starting city. The goal is to minimize the total travel cost while ensuring that every node in the network is visited exactly once.

Mathematical Formulation

- **Parameters and Sets**
  Let:

  $$N = 1, 2, ..., n \text{ be the set of cities to be visited}$$
  $$V = 0 \cup N \text{ be the set of all nodes, where 0 represents the starting city}$$

  Let c_{ij} denote the cost or distance of traveling from city i to city j, where $c_{ii} = 0$ for all i. The distance matrix is assumed to satisfy the triangle inequality in most cases, i.e., $c_{ij} + c_{jk} \geq c_{ik}$.

- **Decision Variables**
  Define the binary variable:

  $$x_{ij} = \begin{cases} 1 & \text{if the route goes directly from city } i \text{ to city } j, \\ 0 & \text{otherwise.} \end{cases}$$

- **Objective Function**
  The objective is to minimize the total travel distance of the tour:

  $$\text{Minimize } Z = \sum_{i \in V} \sum_{\substack{j \in V \\ j \neq i}} c_{ij} \, x_{ij}$$

- **Constraints**

$$\begin{aligned} &\sum_{\substack{j \in V \\ j \neq i}} x_{ij} = 1, && \forall i \in V \quad \text{(each city is departed once)} \\ &\sum_{\substack{i \in V \\ i \neq j}} x_{ij} = 1, && \forall j \in V \quad \text{(each city is entered once)} \\ &x_{ij} \in \{0, 1\}, && \forall i, j \in V \end{aligned}$$

Implementation and Instances

For our implementation of Biased-Randomized Algorithms (BRAs) applied to the Traveling Salesman Problem (TSP), we first precomputed the distance matrix between all pairs of nodes to optimize the construction process. This preprocessing step significantly reduces the computational cost during execution, as the algorithms no longer need to recompute pairwise distances at each iteration.
As a baseline for comparison, we implemented a greedy deterministic solution, whose procedure is described in Algorithm 5. This method iteratively selects, at each step, the nearest unvisited city until all cities have been visited. Although this approach is fast and deterministic, it may lead to suboptimal tours since decisions are made locally without considering future consequences.

Algorithm 5 Greedy Algorithm for the Traveling Salesman Problem

```
Require: Distance matrix $D$, starting node $s$
Ensure: Tour visiting all nodes exactly once
  $n \leftarrow$ number of nodes in $D$
  tour $\leftarrow [s]$
  unvisited $\leftarrow \{0, 1, ..., n-1\} - \{s\}$
  while unvisited $\neq \emptyset$ do
      $i \leftarrow$ last node in tour
      next $\leftarrow \arg\min_{j \in \text{unvisited}} D[i, j]$
      Append next to tour
      Remove next from unvisited
  end while
  return tour
```

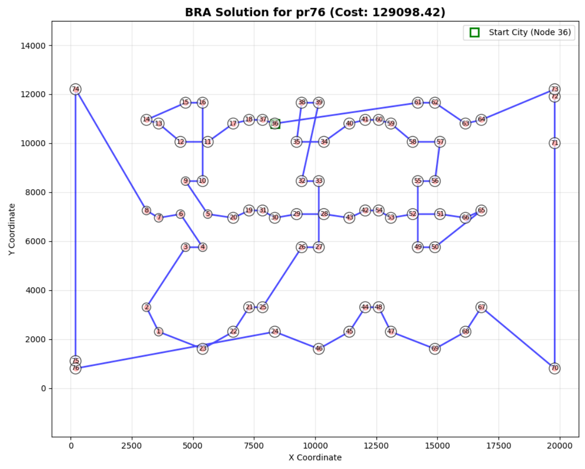
To evaluate the performance of the algorithm under diverse conditions, we employed several benchmark instances provided in class. These instances were selected to ensure sufficient variability in the number of nodes, spatial distribution, and inter-node distances, allowing a comprehensive analysis of the algorithm's robustness.
Finally, to further understand the behavior of the Biased-Randomized approach, we conducted a parameter sensitivity study. In this analysis, we examined the influence of the starting node, the β parameter (represented as $\alpha = 1 - \beta$ in our implementation), and various probability distributions used for randomization. This detailed exploration provides valuable insights into how these factors affect the convergence, stability, and overall quality of the obtained tours.

Results

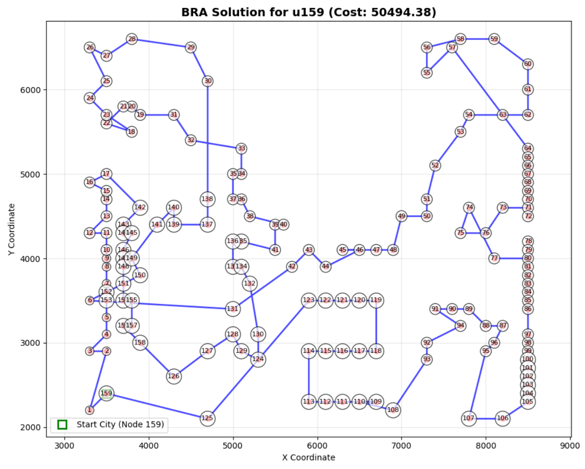
To evaluate the BRA algorithm, several experiments were conducted across five different instances, for all the executed instances an $\alpha = 0.1$ was used. Each instance is characterized by distinct geographical distributions and a varying number of nodes. Then the BRA was executed $1,000$ times to ensure a comparable computational effort across all scenarios.

Table 14: Comparison of Greedy and Biased-Randomized Algorithm (BRA) results for multiple TSP benchmark instances.

Instance	Greedy	Time (s)	BRA	Improvement (%)	Time (s)
ulysses16	104.73	0.0000	76.41	27.04	0.1255
berlin52	8980.92	0.0000	8102.00	9.79	0.8996
pr76	153,461.92	0.0000	129,098.42	15.88	1.7723
eil101	825.24	0.0000	741.40	10.16	3.4889
u159	54669.03	0.0030	50494.38	7.64	7.8297
Average	–	**0.0006**	–	**13.29**	**3.0890**



(a) Solution obtained for instance pr76.



(b) Solution obtained for instance u159.

Figure 7: Solution tours obtained for multiple instances using Biased-Randomized Algorithm.

As shown in Table 14, the Biased-Randomized Algorithm (BRA) consistently outperforms the Greedy solution across all tested Traveling Salesman Problem (TSP) benchmark instances. The greedy approach, while extremely fast, tends to produce suboptimal routes since it selects the nearest unvisited city at each step without considering global optimization effects. In contrast, the BRA introduces a controlled level of randomness in the selection process that enables the algorithm to escape local minimum and achieve shorter tour lengths.
The results demonstrate that the BRA achieves an average improvement of approximately 13.3% over the greedy baseline, with the most significant gain observed in the *ulysses16* instance, where the total tour length decreased by 27.0%. Even in larger and more complex instances such as *u159*, BRA still delivers notable improvements (around 7.6%) while maintaining reasonable computational times.
Regarding efficiency, execution times for BRA remain relatively low—ranging from 0.12 seconds for the smallest instance to under 8 seconds for the largest one. These results confirm that the algorithm can achieve substantial improvements in route quality with only a modest increase in computation time compared to the nearly instantaneous greedy heuristic.

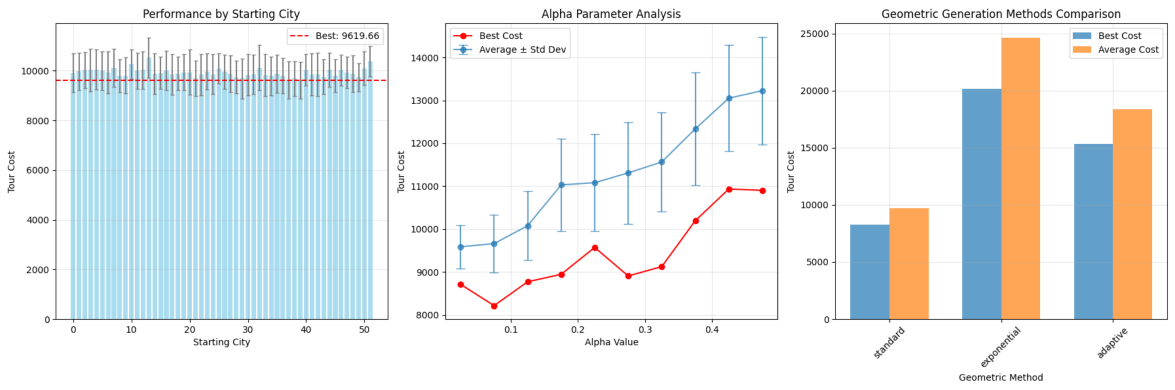


Figure 8: Caption

Focusing on Figure 8, we can perform a deeper analysis on berlin52 instance and observe several important patterns. First, when analyzing the impact of the starting city on the final tour, it becomes evident that this factor has virtually no significant influence on the outcome. As shown by the results of five independent runs starting from different cities, both the average tour lengths and their standard deviations remain very close to each other, indicating a stable and consistent performance across initial conditions.
On the other hand, when examining the effect of the α parameter, a clear trend emerges: lower values of α yield better overall performance, both in terms of average and best-case tour lengths. This behavior is expected, since a lower α implies a higher β (by definition, $\alpha = 1 - \beta$), leading the algorithm to adopt a more greedy behavior. In practical terms, this means that the algorithm assigns a higher probability to selecting the node that produces the smallest incremental cost, resulting in shorter tours on average.
Finally, when comparing different probability distributions, the standard geometric distribution consistently delivers the best performance, outperforming both the exponential and adaptive variants. The exponential distribution yields the poorest results, with an average tour cost of approximately 24,500, whereas the adaptive distribution performs moderately better with an average cost of around 18,000. The adaptive version, in particular, recalculates α at each iteration, progressively increasing the level of greediness. However, despite this adaptive adjustment, the fixed geometric distribution remains superior in terms of both stability and solution quality.

4 Unit 4. GRASP and PJS Heuristic

4.1 Greedy Randomized Adaptive Search (GRASP)

GRASP is a constructive metaheuristic similar in spirit to the Biased-Randomized Algorithm (BRA). Both aim to iteratively improve a solution through a combination of greedy logic and randomness. However, in GRASP, all elements within the Restricted Candidate List (RCL) are selected according to a uniform probability distribution. That is, once the list of promising candidates is built using a greedy criterion, every candidate inside the RCL has the same probability of being chosen. This approach ensures a balance between exploitation, by limiting the RCL to the best candidates, and exploration, by introducing randomness in the selection process.

Algorithm 6 General GRASP Algorithm

```
Require: Candidate set $C$, objective function $f$, greediness factor $\alpha$, number of iterations $N$
Ensure: Best solution found $S_{best}$
  $S_{best} \leftarrow \emptyset$
  for $k = 1$ to $N$ do
      $S \leftarrow$ ConstructionPhase($C$, $f$, $\alpha$)
      $S' \leftarrow$ LocalSearch($S$, $f$)
      if $f(S') < f(S_{best})$ or $S_{best}$ is empty then
          $S_{best} \leftarrow S'$
      end if
  end for
  return $S_{best}$
```

As shown in Algorithm 6, GRASP starts by initializing an empty best solution. Then, for each iteration until reaching the maximum number of iterations, it executes two main steps:

1. **Constructive Phase**, where a feasible initial solution is built by progressively adding elements based on greedy criteria and random selection from the RCL. Algorithm 7 shows that the construction phase begins by initializing an empty solution and iteratively adding new elements. In each iteration, all candidates are evaluated using a greedy function $g(c)$. The algorithm then determines the range between the best and worst greedy values (g_{max} and g_{min}) and constructs the Restricted Candidate List (RCL) by including only those candidates within the top α proportion of greedy quality.

   - When α is small, the RCL is narrow and the algorithm behaves greedily.
   - When α is large, the RCL expands, increasing randomness and exploration.

   A candidate is then selected uniformly at random from the RCL and added to the partial solution. The candidate list and feasibility conditions are updated at each step until a complete feasible solution is built.

2. **Local Search Phase**, where the constructed solution is refined by exploring its neighborhood until a local optimum is reached.

After both phases are completed, the improved solution is compared with the best one found so far. If it yields a lower cost, it replaces the current best solution.

Algorithm 7 GRASP Construction Phase

```
Require: Candidate set $C$, objective function $f$, greediness factor $\alpha$
Ensure: Constructed solution $S$
  $S \leftarrow \emptyset$                                   ▷ Initialize empty solution
  while termination condition not met do
      Evaluate all candidates $c \in C$ according to the greedy function $g(c)$
      Let $g_{max}$ and $g_{min}$ be the maximum and minimum values of $g(c)$
      Build RCL $= c \in C : g(c) \geq g_{max} - \alpha(g_{max} - g_{min})$
      Select one candidate $c_r$ from RCL uniformly at random
      $S \leftarrow S + c_r$
      Update candidate list $C$ and feasibility constraints
  end while
  return $S$
```

4.2 Panadero & Juan Savings (PJS)

The Panadero & Juan Savings (PJS) algorithm is a constructive heuristic inspired by the classical CWS algorithm, but specifically adapted to routing problems that include both a start and an end depot, such as the Team Orienteering Problem (TOP).
Unlike the standard CWS, where routes start and end at the same depot and only distance savings are considered, PJS introduces a bidirectional depot structure and combines both distance savings and reward maximization within a unified efficiency function.
In this approach, as shown in Algorithm 8, each customer is initially assigned to its own individual route connecting the start and end depots. The algorithm then computes the pairwise efficiencies between all customers, which represent a weighted combination of classical savings and accumulated rewards. At each step, the algorithm selects the pair of customers with the highest efficiency score $e_{i,j}$.Then, if merging them does not violate route feasibility or capacity constraints, their corresponding routes are joined into a single one. Once no further feasible merges are possible, all routes are sorted by their total collected reward, and only the top routes up to the fleet size limit are retained. This final filtering step ensures that the global solution maximizes total profit while respecting both the maximum route cost and the available number of vehicles.

4.3 GRASP related Paper

Ferone et al., 2019 present two key enhancements to the classical GRASP metaheuristic to improve its performance on both deterministic and stochastic combinatorial optimization problems. The first extension, Biased-Randomised GRASP (BR-GRASP),

Algorithm 8 Panadero & Juan Savings (PJS) Algorithm

```
Require: Instance $(V, E)$ with start and finish depots, distance matrix $D$, demands $d_i$, fleet size $F$, maximum route cost $C_{max}$, weighting parameter $\alpha$
Ensure: Set of feasible routes $S$
  Generate directed graph $G(V, E)$ including both start and finish depots
  for each pair of customers $(i, j)$ do
      Compute classical saving: $s_{ij} = c_{i,n+1} + c_{0,j} - c_{ij}$
      Compute combined efficiency: $e_{ij} = \alpha \cdot s_{ij} + (1 - \alpha) \cdot (d_i + d_j)$
      Store $(i, j, e_{ij})$ in the efficiency list
  end for
  Sort efficiency list in descending order of $e_{ij}$
  Initialize dummy solution: one route per customer $(0, i, n + 1)$
  while efficiency list not empty do
      Select top pair $(i, j)$ with highest $e_{ij}$
      if merging routes of $i$ and $j$ is feasible (cost $\leq C_{max}$) then
          Merge both routes and update total cost and demand
          Remove inverse pair $(j, i)$ from list
      end if
  end while
  Sort final routes by total reward and select top $F$ routes
  return $S$
```

replaces the traditional Restricted Candidate List used in the solution construction phase. Instead of a uniform selection from a subset of the best candidates, BR-GRASP assigns skewed probabilities to all feasible elements, using a distribution like the geometric to bias the selection towards the most promising options. This simple change, which requires no additional coding effort or complex parameters, significantly boosts solution quality without compromising the algorithm's speed or simplicity. The second extension, SimGRASP, integrates simulation into the framework to handle problems with stochastic elements, such as random demands or processing times. It operates as a simheuristic by first solving a deterministic version of the problem using BR-GRASP, then employing "short" simulations to quickly evaluate solutions under uncertainty and identify an elite set of promising candidates, which are finally re-evaluated with a more rigorous "long" simulation. To validate these approaches, the authors tested them on three NP-hard problems: the Permutation Flow Shop Problem, the Capacitated Vehicle Routing Problem, and the Uncapacitated Facility Location Problem. The results demonstrated that BR-GRASP consistently outperformed traditional GRASP on deterministic versions, achieving smaller gaps to best-known solutions. Furthermore, SimGRASP proved highly competitive against other state-of-the-art simheuristics, showing that these extensions successfully enhance GRASP's efficiency for deterministic problems and robustly extend its applicability to complex, real-world stochastic optimization scenarios.
Score: 7

4.4 Travelling Salesman Problem

To evaluate the GRASP heuristic in the Travelling Salesman Problem (see Section 3.4 for the problem introduction and mathematical formulation), we perform a controlled experiment using our own implementation and using several instances to be available to analyse the quality, variability, and computational cost of GRASP.

Implementation

The implementation of GRASP follows a similar design as the one used for the Biased-Randomized Algorithm (BRA), with the goal of maximizing computational efficiency. To achieve this, all pairwise distances between nodes are precomputed in advance, significantly reducing the number of calculations required during the iterative process. In our GRASP implementation three key parameters must be configured:

- **Maximum number of iterations.** Defines the total number of global iterations of the algorithm. Each iteration includes both the construction and the local search phases.
- **Maximum number of iterations without improvement.** This parameter applies to the local search phase and prevents the algorithm from entering an infinite loop when it reaches a local minimum. In our case, the local search is implemented using the 2-opt heuristic. The 2-opt algorithm iteratively removes two edges from the current tour and reconnects the resulting paths in the opposite order. If the new configuration yields a shorter route, it replaces the current one. This process continues until no further improvement is possible within the defined limit.
- **Alpha (α) parameter.** This parameter controls the level of greediness during the construction phase by determining which candidates are included in the Restricted Candidate List (RCL).
  - When α is close to 0, the construction process becomes more greedy, as only the elements with the lowest cost are included in the RCL.
  - When α approaches 1, the selection becomes more random, as the RCL widens to include a larger number of candidates.

Results

To evaluate the GRASP implementation, several experiments were conducted across five different instances, for all the executed instances an $\alpha = 0.3$ was used. Each instance is characterized by distinct geographical distributions and a varying number of nodes. Then the GRASP was executed with $1,000$ maximum iterations and 50 maximum iterations with no improvement to ensure a comparable computational effort across all scenarios.

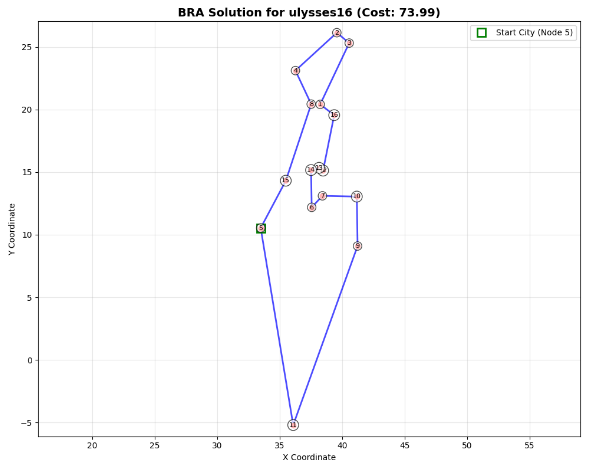
Table 15: Comparison of Greedy and Greedy Randomized Adaptive Search (GRASP) results for multiple TSP benchmark instances.

Instance	Greedy	Time (s)	GRASP	Improvement (%)	Time (s)
ulysses16	104.73	0.0000	73.99	28.85	1.6154
berlin52	8980.92	0.0000	9118.14	-1.52	8.6619
pr76	153,461.92	0.0000	167,524.19	-9.16	15.0865
eil101	825.24	0.0000	1064.89	-29.04	20.0288
u159	54,669.03	0.0030	111,311.46	-50.88	43.4295
Average	–	**0.0006**	–	**-12.35**	**17.7644**

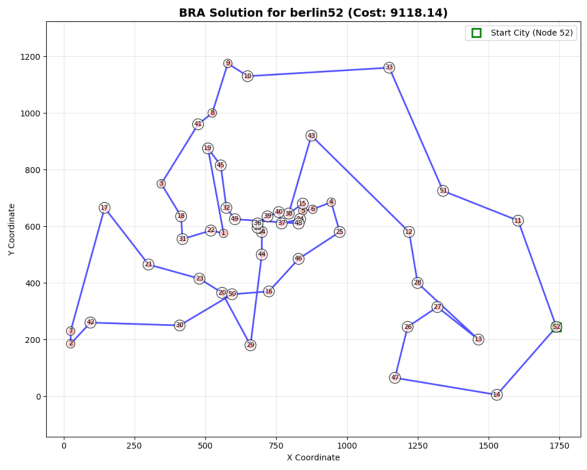
As shown in Table 15, the Greedy Randomized Adaptive Search Procedure (GRASP) was evaluated across the same set of Traveling Salesman Problem (TSP) benchmark instances previously used for the Biased-Randomized Algorithm (BRA).
The results, however, show that GRASP did not consistently outperform the purely constructive Greedy algorithm in this configuration. On average, GRASP achieved a negative improvement of approximately –12.35% compared to the greedy baseline, indicating that the routes obtained were longer in most cases. Only the smallest instance, *ulysses16*, exhibited a positive improvement of 28.85%, demonstrating the potential of GRASP when the problem scale is limited and the local search can efficiently explore the neighbourhood structure. In contrast, for medium and large instances such as *eil101* and *u159*, performance degraded substantially, with route costs increasing by up to 50.9% over the greedy reference.
This decline in performance can be attributed to several factors. First, GRASP's construction phase relies on the parameter α to control the balance between greediness and a larger α produces more diversified candidate selections, but this can also lead to less competitive initial solutions that require a strong local search to recover quality. In these experiments, the 2-opt local search was applied, but as problem size grows, the local improvement step becomes computationally expensive and may converge prematurely to poor local minima.
In terms of efficiency, GRASP shows considerably higher computational times than BRA. Execution times range from 1.6 s for the smallest instance to over 43 s for the largest one—an average of 17.76 s per run—compared to an average of 3.09 s for BRA. This reflects the additional computational time of performing a local search within each iteration, especially when applied to larger problem instances. In summary, the current GRASP implementation provides competitive results for small-scale TSP instances but struggles to maintain performance as the problem size increases.

4.5 Job Shop Scheduling Problem

The Job Shop Scheduling Problem (JSSP) is a classic and highly challenging combinatorial optimization problem within the field of production planning and scheduling. JSSP is known to be NP-hard, and even moderately sized instances can be extremely difficult to solve to optimality. The objective is to schedule a set of jobs on a set of machines, where each job consists of a fixed sequence of operations, each requiring a



(a) Solution obtained for instance ulysses16.



(b) Solution obtained for instance berlin52.

Figure 9: Solution tours obtained for multiple instances using Greedy Randomized Adaptive Search.

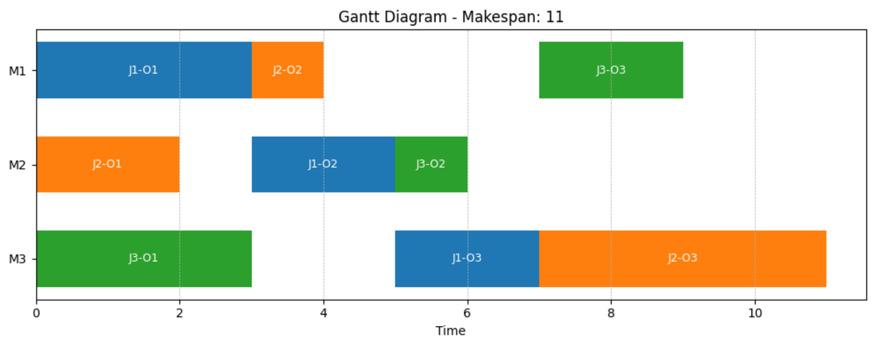


Figure 10: Optimal solution for a random instance of JSSP with 3 jobs and 3 machines.

specific machine and a predetermined processing time. The goal is to determine the start times of all operations such that the makespan (the total completion time of all jobs) is minimized, while respecting the following constraints:

- Each machine can process at most one operation at a time.
- Operations belonging to the same job must be executed in the predefined sequence.

Mathematical Formulation

- **Sets and Indices**
  Let:

$$\begin{aligned} J &= 1, 2, \ldots, n \quad \text{set of jobs} \\ M &= 1, 2, \ldots, m \quad \text{set of machines} \\ O_j &= 1, 2, \ldots, m \quad \text{sequence of operations for job } j \end{aligned}$$

Each operation (j, k) of job j must be processed on machine $\mu_{jk} \in M$ for p_{jk} time units.

- **Decision Variables**
  Let s_{jk} denote the start time of operation k of job j.

- **Objective Function**
  Minimize the makespan:

  $$\text{Minimize } C_{\max} = \max_{j \in J} (s_{j,m} + p_{j,m})$$

  where $s_{j,m} + p_{j,m}$ is the completion time of the last operation of job j.

- **Constraints**

  $$s_{jk} + p_{jk} \leq s_{j,k+1} \quad \forall j \in J, k = 1, \dots, m-1 \quad \text{(precedence within job)}$$
  $$s_{jk} + p_{jk} \leq s_{j'k'} \ \vee \ s_{j'k'} + p_{j'k'} \leq s_{jk} \quad \forall (j,k), (j',k') : \mu_{jk} = \mu_{j'k'} \quad \text{(machine non-overlap)}$$
  $$s_{jk} \geq 0 \quad \forall j \in J, k \in O_j$$

Greedy Baseline

To establish a performance baseline, a greedy constructive heuristic is implemented. This heuristic always selects the available operation with the shortest processing time. The available operations are stored in a list, sorted by increasing processing time. The shortest operation is scheduled at the maximum of its machine's available time and its job's available time. After scheduling, the machine and job availability times are updated, and the next operation of the job (if any) is added to the available list.
This greedy strategy is fast and simple, therefore, it often yields suboptimal solutions due to its dummy nature, ignoring future consequences of scheduling decisions. Nevertheless, it provides a useful benchmark for evaluating more sophisticated methods.

Implementation and Instances

The implementation of GRASP for the Job Shop Scheduling Problem follows a structured and modular design, with a focus on computational efficiency and solution quality. The algorithm is divided into two main phases: a randomized constructive phase and a local search improvement phase, which are iteratively applied to explore the solution space effectively.
In our GRASP implementation, several key parameters must be configured to balance exploration and exploitation:

- **Maximum number of iterations.** Defines the total number of global iterations of the algorithm. Each iteration includes both the construction phase and the local search phase.

- **Restricted Candidate List (RCL) size.** Determines the maximum number of candidate operations considered during the construction phase. A smaller RCL leads to more greedy behavior, while a larger RCL increases randomness.

- **Alpha (α) parameter.** This parameter controls the balance between greediness and randomness during the construction phase by determining which candidates are included in the Restricted Candidate List (RCL).
  - When α is close to 0, the construction process becomes more greedy, as only the elements with the lowest cost (earliest finish time) are included in the RCL.
  - When α approaches 1, the selection becomes more random, as the RCL widens to include a larger number of candidates with higher costs.
- **Local search iterations.** This parameter defines the number of local search steps applied to each constructed solution. The local search employs a neighborhood structure that swaps consecutive operations on critical machines to potentially improve the makespan.

The implementation uses object-oriented programming with two main classes: `JSSPInstance` for loading and storing problem data, and `JSSPSolution` for representing and evaluating candidate schedules. To enhance computational performance, the implementation maintains dynamic data structures tracking machine availability times, job next operations, and job available times. This avoids redundant calculations during both the construction and local search phases.
To evaluate the performance of our GRASP implementation, we use standardized benchmark instances from the JSSP Instances and Results Repository (Weise, 2020). This repository provides a comprehensive collection of JSSP instances with varying sizes and characteristics, along with their Best Known Solutions (BKS) for comparison.

Results

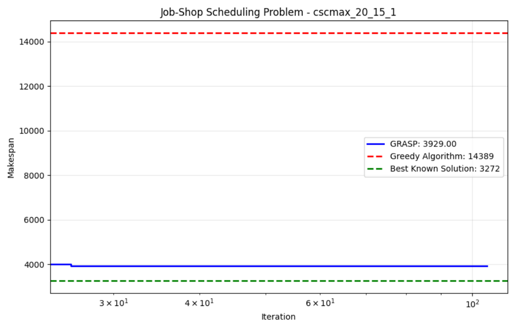
The GRASP algorithm was executed on several benchmark instances with parameter $\alpha = 0.3$ for the greedy construction phase, a maximum of 500 iterations for all the GRASP process, a restricted candidate list of maximum 3 elements and 50 iterations at maximum for the local search phase.Then, the results were compared against the greedy baseline solution and the best know solution (BKS) for each analysed instance.

Table 16: Comparison of Greedy and GRASP solutions for JSSP instances.

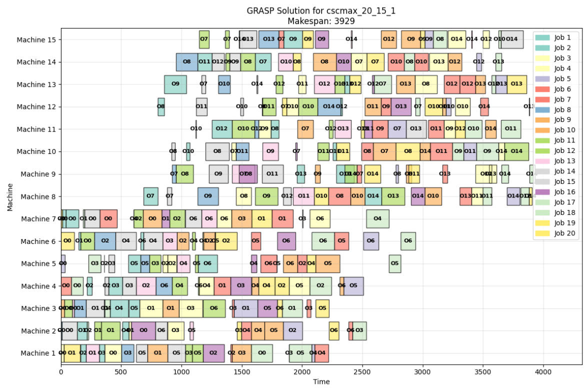
Instance	Greedy	GRASP	BKS	Time (s)
cscmax_20_15_1	14,389	3,991	3,272	30.12
rcmax_30_15_1	22,426	4,321	3,343	40.76
rcmax_30_20_10	34,104	4,888	3,814	58.62
rcmax_40_20_3	48,442	6,369	4,848	80.47
cscmax_50_15_4	32,050	8,267	6,196	73.86

As shown in Table 16, GRASP consistently outperforms the greedy heuristic by a large margin, reducing the makespan by an average of 81% across the tested instances. However, a gap of approximately 24% remains between GRASP solutions and the best known solution, indicating room for further improvement.

Focusing on the GRASP efficiency, the computational time remain stable accross all intances, but with a trend to increase with instance size, ranging from 30 seconds for smaller instances to over 80 seconds for larger ones. This is acceptable given the complexity of JSSP and the quality of solutions obtained.



(a) Convergence plot with 500 iterations.



(b) Gantt chart for the GRASP solution.

Figure 11: Solution generated using GRASP with 500 iterations and $\alpha = 0.3$ for the instance `cscmax_20_15_1`.

The convergence behavior, illustrated in Figure 11.a, reveals that the initial solution generated by GRASP remains almost identical to the final one. This occurs because the greedy constructive phase produces a solution that is already of high quality, leaving little room for significant improvement during the local search. However, after a few initial iterations, a slight improvement can be observed, reflecting the limited capacity of the local search to refine the already well-structured initial tour. Beyond this point, the convergence curve stabilizes completely, indicating that no further progress is achieved and that the algorithm has reached a local optimum. This behavior suggests that, for some problem instances, the greedy construction alone can generate near-optimal solutions, especially when the problem's landscape allows the heuristic to quickly capture the underlying structure of the optimal route.

The Gantt chart in Figure 11.b clearly highlights the non-overlapping execution of tasks across machines, ensuring that each machine processes only one operation at a time while maintaining the required job precedence constraints. Moreover, the schedule exhibits a high degree of balance, with relatively short idle periods and smooth transitions between consecutive operations. This structured arrangement reflects the algorithm's ability to build and refine feasible solutions efficiently, resulting in an overall schedule that minimizes idle time and maximizes machine utilization. The clarity of the resulting plan underscores the strength of GRASP in combining a greedy constructive approach with localized optimization to produce practical and coherent scheduling outcomes.

4.6 Team Orienteering Problem

The Team Orienteering Problem (TOP) is a combinatorial optimization problem within the field of vehicle routing and resource allocation. TOP is known to be NP-hard, and even moderately sized instances can be extremely difficult to solve to optimality. The objective is to design a set of routes for a fleet of vehicles to collect rewards (or scores) from a set of locations, where each location can be visited at most once. The goal is to maximize the total collected reward, while respecting the following constraints:

- Each route must start and end at specific depot nodes.
- The total cost of each route must not exceed a given limit.
- Each location (except depots) can be visited at most once.

Mathematical Formulation

- **Sets and Indices**
  Let:

$$\begin{aligned} V &= 0, 1, \ldots, n, n+1 \quad \text{set of nodes (0: start depot, } n+1\text{: end depot)} \\ K &= 1, 2, \ldots, m \quad \text{set of vehicles} \\ P &= 1, 2, \ldots, n \quad \text{set of customer nodes with rewards} \end{aligned}$$

  Each node $i \in P$ has a reward d_i, and each edge (i, j) has a travel cost c_{ij}.

- **Decision Variables**
  Let x_{ij}^k be a binary variable that equals 1 if vehicle k travels from node i to node j, and 0 otherwise.
  Let y_i be a binary variable that equals 1 if node i is visited, and 0 otherwise.

- **Objective Function**
  Maximize the total collected reward:

$$\text{Maximize } \sum_{i \in P} d_i \cdot y_i$$

- **Constraints**

$$\begin{aligned} &\sum_{k \in K} \sum_{j \in V \setminus \{0\}} x_{0j}^k \leq m \quad \text{(fleet size)} \\ &\sum_{i \in V \setminus \{n+1\}} x_{i,n+1}^k = 1 \quad \forall k \in K \quad \text{(end at finish depot)} \\ &\sum_{i \in V \setminus \{n+1\}} x_{ij}^k = \sum_{i \in V \setminus \{0\}} x_{ji}^k \quad \forall j \in P, k \in K \quad \text{(flow conservation)} \\ &\sum_{k \in K} \sum_{i \in V \setminus \{n+1\}} x_{ij}^k = y_j \quad \forall j \in P \quad \text{(node visited once)} \\ &\sum_{i \in V} \sum_{j \in V} c_{ij} \cdot x_{ij}^k \leq T_{\max} \quad \forall k \in K \quad \text{(route cost limit)} \\ &x_{ij}^k \in \{0, 1\}, \quad y_i \in \{0, 1\} \quad \forall i, j \in V, k \in K \end{aligned}$$

Implementation

The implementation of the PJS heuristic for the Team Orienteering Problem focus on computational efficiency and solution quality. Our implementation is divided into three main phases: graph generation, dummy solution construction, and iterative route merging.
The implementation uses a directed graph from the NetworkX library to efficiently manage nodes, edges, and route information. The solution is represented as a set of routes, each containing a sequence of edges, total cost, and total reward. The algorithm ensures that all constraints are respected, including the fleet size and maximum route cost.
To enhance computational performance, the implementation maintains dynamic data structures tracking which nodes are linked to the start and finish depots, and which route each node belongs to. This avoids redundant calculations during the merging phase and ensures efficient updates.

Results

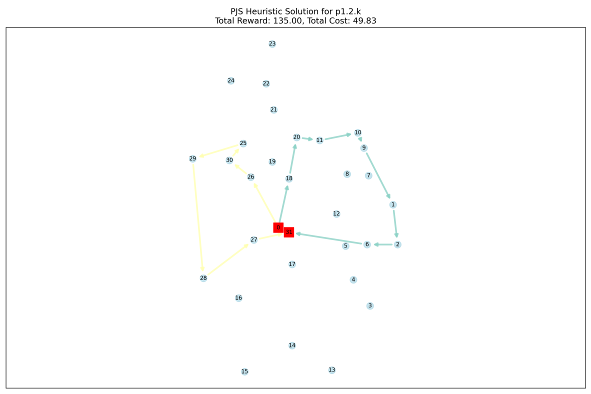
Table 17: Resultados del algoritmo PJS para diferentes instancias del Team Orienteering Problem.

Instance	Total Reward	Time (s)	Avg. Reward/Route
p5.3.q	975.00	0.03	325.00
p7.3.r	638.00	0.08	212.67
p6.4.d	156.00	0.03	39.00
p4.4.s	791.00	0.07	197.75
p2.2.f	200.00	0.00	100.00
Average	–	**0.042**	–

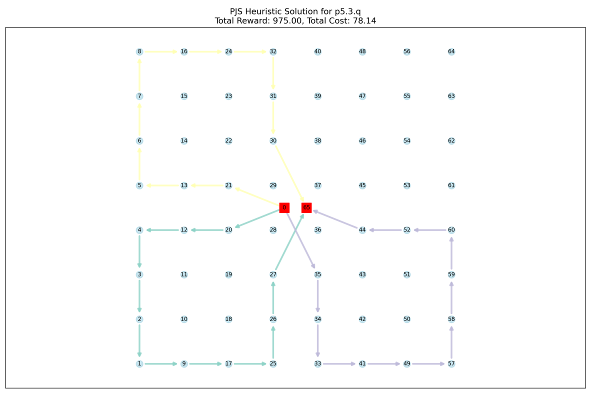
The PJS algorithm demonstrates outstanding temporal performance, maintaining remarkably low execution times regardless of the instance being solved. As shown in Table 17, the average computation time is only 0.042 seconds, ranging from instantaneous resolution (0.00s) for the simplest instance to a maximum of 0.08 seconds for the most complex instance. This temporal consistency is particularly notable considering the variability in instance size and complexity
Analyzing the graphical solutions presented in Figure 12, several patterns consistent with the nature of the Team Orienteering Problem are observed:

- **Selective Node Coverage.** As expected given the problem formulation, not all nodes are visited in the optimal solutions. The algorithm performs strategic selection of nodes that maximize total reward while respecting per-route cost constraints.

- **Geographical Coherence and Spatial Segregation.** The generated routes exhibit spatially coherent organization, forming patterns that follow the geographical distribution of nodes. It is clearly observed that the routes tend to cover specific regions without significant overlaps and there are no intersections between different routes, indicating efficient territory allocation to each vehicle.


PJS Heuristic Solution for p1.2.k
Total Reward: 135.00, Total Cost: 49.83

(a) Solution obtained for instance p1.2.k.


PJS Heuristic Solution for p5.3.q
Total Reward: 975.00, Total Cost: 78.14

(b) Solution obtained for instance p5.3.q.

Figure 12: Solution obtained for multiple instances using PJS Heurisitc.

5 Unit 5. ILS, TS and NEH Heuristic PFSP

5.1 Iterated Local Search (ILS)

Iterated Local Search (ILS) is a metaheuristic technique designed to escape from local minimum by iteratively applying perturbations and local search refinements to a base solution. Unlike pure local search algorithms that easily get trapped in a local minimum, ILS introduces controlled randomness to explore new regions of the solution space, while still exploiting promising areas through repeated local improvements.

Algorithm 9 Iterated Local Search (ILS)

```
Require: Objective function $f$, number of iterations $N$
Ensure: Best solution $S^*$
  $S_0 \leftarrow GenerateInitialSolution$
  $S \leftarrow LocalSearch(S_0)$                                ▷ Set the base solution
  $S^* \leftarrow S$
  for $k = 1$ to $N$ do
      $S' \leftarrow Perturbation(S, history)$
      $S'^* \leftarrow LocalSearch(S')$
      if $f(S'^*) < f(S^*)$ then
          $S^* \leftarrow S'^*$                                   ▷ Update best solution
      end if
      $S \leftarrow AcceptanceCriterion(S, S'^*, history)$
  end for
  return $S^*$
```

In Algorithm 9, the procedure starts from an initial solution, which is first improved using a local search method.Then, at each iteration, a two steps proceeding is a applied to the base solution:

1. **Perturbation.** Modifies the current solution to escape from a local minimum, introduces the randomness to the algorithm.
2. **Local Search** Refines the perturbed solution until a new local optimum is reached.

Then, using the objective function is checked if the new solution is better than the current best solution the best solution is updated. Finally, the acceptance criterion determines whether the new solution replaces the current base one. The simplest criterion accepts the new solution if it improves the best found so far. This cycle repeats for a fixed number of iterations or until a stopping criterion is met. The best solution found throughout the process is stored and returned as the final output.

5.2 Tabu Search (TS)

Tabu Search (TS) is a single-solution metaheuristic that enhances traditional local search by incorporating adaptive memory structures to avoid cycling and promote exploration. TS systematically explores the neighborhood of the current solution while

preventing the revisiting of recently explored configurations through a so-called tabu list.
The main characteristic of Tabu Search is that it allows non-improving moves to be selected, enabling the algorithm to explore a wider region of the solution space. To prevent cycling back to previously visited solutions, it incorporates a short-term, limited FIFO memory structure that stores recently visited solutions to temporarily forbid their selection.

Algorithm 10 Tabu Search (TS)

```
Require: Objective function $f$, tabu list size $L$, number of iterations $N$
Ensure: Best solution found $S^*$
  $S_0 \leftarrow GenerateInitialSolution$
  $S \leftarrow S_0$
  $S^* \leftarrow S_0$
  $T \leftarrow \{S_0\}$                                        ▷ Initialize tabu list
  for $k = 1$ to $N$ do
      $N_S \leftarrow GenerateNeighbourhood(S)$
      $N'_S \leftarrow RemoveTabuList(N_S)$        ▷ Remove solutions if they are in the tabu list
      $S' \leftarrow BestCandidate(N'_S, f)$
      if $f(S') < f(S^*)$ then
          $S^* \leftarrow S'$
      end if
      $T \leftarrow T \cup \{S'\}$                                 ▷ Add to tabu list
      if $|T| > L$ then
          Remove oldest element from $T$
      end if
      $S \leftarrow S'$
  end for
  return $S^*$
```

As shown in Algorithm 10, the procedure begins by generating an initial feasible solution, which is simultaneously stored as the current solution and the best solution. A tabu list is initialized to keep track of recently visited solutions, preventing the algorithm from returning to them in the short term.
During each iteration, the algorithm generates the neighbourhood of the current solution, which represents all candidate solutions reachable by applying an elementary modification. The solutions contained in the tabu list are removed from this neighbourhood, unless an aspiration criterion applies—typically, if a tabu solution would lead to a global improvement.
The best candidate from the remaining neighbourhood is then selected, regardless of whether it improves the current solution. If it improves the best known solution, it is stored as the new best. The selected solution is added to the tabu list to prevent immediate reversal, and if the tabu list exceeds its maximum size, the oldest element is removed, following a First-In-First-Out (FIFO) policy. This iterative process continues

until the maximum number of iterations is reached. The final output is the best solution obtained throughout the search.

5.3 Iterated Local Search for Solving the Flow-Shop Problem

Juan et al., 2014 present the ILS-ESP (Iterated Local Search – Efficient, Simple, Parallelizable) algorithm, an improved version of iterative local search designed to solve the permutation flow-shop problem (PFSP), aiming to minimize the makespan. The PFSP consists of sequencing a set of jobs that must be processed on several machines in the same order, seeking the sequence that minimizes the total completion time. Unlike other ILS methods that require careful parameter tuning, ILS-ESP stands out for its simplicity, use of natural parameters, and ability to run in parallel. Its main innovations include a perturbation mechanism based on an "enhanced exchange" that combines randomness with the NEH heuristic; a "Demon" acceptance criterion that allows controlled degradation of the solution to improve exploration; and a diversified generation of initial solutions through biased randomization over NEH, which facilitates parallelization. Implemented in Java and tested using Taillard's benchmarks, ILS-ESP matches or outperforms reference algorithms such as ILS and IG without requiring parameter adjustment. Moreover, parallel execution improves solution quality in reduced computational time, establishing ILS-ESP as a simple, efficient, and robust alternative for solving the PFSP.
Score: 7

5.4 Travelling Salesman Problem

To evaluate the ILS and TS heuristic in the Travelling Salesman Problem (see Section 3.4 for the problem introduction and mathematical formulation), we perform a controlled experiment using our own implementation and using several instances to be available to analyse the quality, variability, and computational cost of both heuristics.

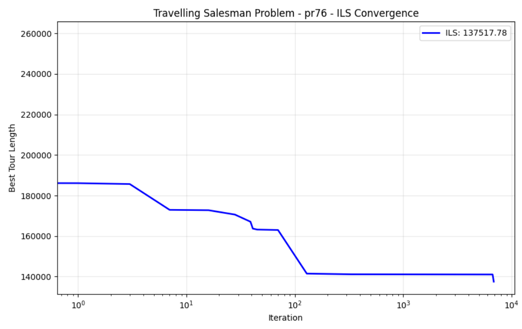
Results

Table 18: Comparison of Iterated Local Search (ILS) and Tabu Search (TS) for multiple TSP instances.

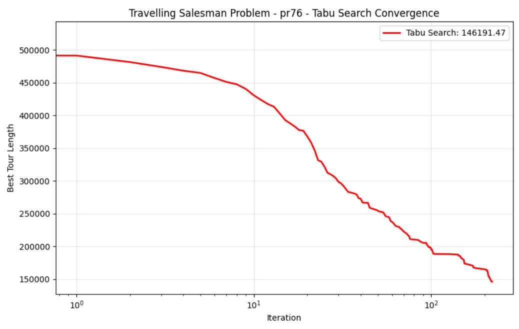
Instance	ILS	Time ILS	TS	Time TS	Improvement (%)
berlin52	8,281.23	21.03	8,479.10	4.21	-2.33
ulysses16	73.99	12.21	73.99	1.89	0.00
pr76	137,517.78	31.72	146,191.47	6.59	-6.31
eil101	765.93	29.59	914.83	7.33	-19.45
u159	79,077.72	49.62	89,898.78	10.54	-13.68
Average	–	**28.83**	–	**6.11**	**-8.35**

The results, Table 18, show that both algorithms achieve comparable results; however, we observe that the results of ILS are consistently superior to those of TS, with an average improvement of 8.35%. If we examine the average execution times, we see that

ILS presents an average execution time of nearly 29 seconds, compared to just over 6 seconds for TS. Looking at these times in detail, we see that for simpler instances, the differences are 1.9 seconds versus 12.2 seconds, with both heuristics obtaining the same results.



(a) Convergence plot for the Iterated Local Search Heuristic, converge in 6851 iterations.



(b) Convergence plot for the Tabu Search Heuristic, converge in 221 iterations.

Figure 13: Convergence plots for the instance pr76.

If we analyze a single instance in detail, observing Figure 13 with the pr76 instance, we see that while in TS improvements occur continuously across almost all iterations, obtaining the best solution in 221 iterations, with ILS, 6851 iterations are necessary to reach the best solution. We also observe that there are flat zones indicating that no better solution was generated for several iterations.

5.5 Permutated Flow-Shop Problem (NEH)

To evaluate the performance of the NEH heuristic in the Permutated Flow-Shop Problem (PFSP), we perform a controlled experiment using our own implementation across several Taillard benchmark instances. The objective is to analyse the quality of the solutions obtained (in terms of makespan), their improvement relative to a simple baseline heuristic, and the computational cost of the NEH algorithm.

Results

Table 19: Performance of the NEH heuristic on multiple PFSP instances.

Instance	Baseline	NEH	Improvement (%)	Time (s)
tai009_20_5	1502.00	1283.00	14.59	0.0010
tai027_20_20	2730.00	2437.00	10.73	0.0050
tai047_50_10	3616.00	3377.00	6.61	0.0170
tai078_100_10	6937.00	5815.00	16.17	0.0665
tai120_500_20	30243.00	27292.00	9.76	3.2432
Average	–	–	**11.57**	**0.67**

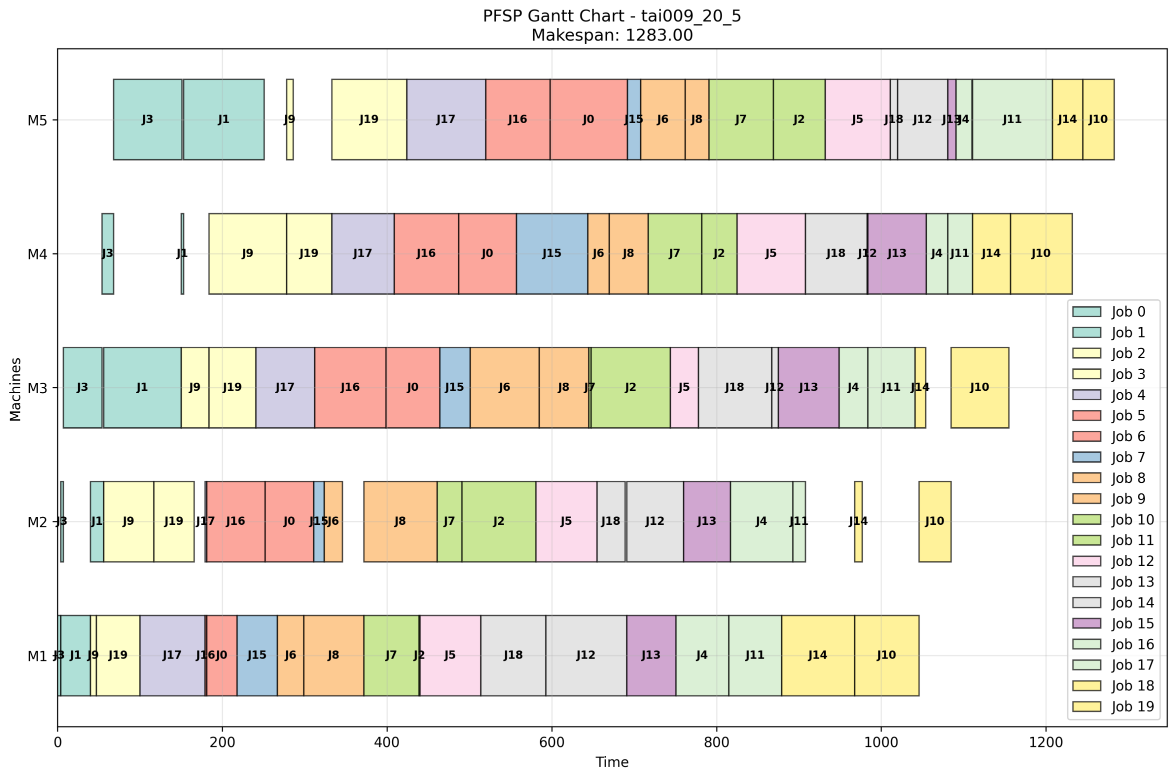


Figure 14: Gantt chart for the NEH heuristic applied to instance tai009_20_5.

The results, presented in Table 19, demonstrate that the NEH heuristic significantly improves the baseline solution in all tested instances, with an average improvement of 11.57% in terms of makespan. The improvement ranges from approximately 6.6% in medium-sized instances (50 jobs, 10 machines) to over 16% in larger instances with 100 jobs and 10 machines).

Regarding computational performance, NEH proves to be highly efficient, with execution times below 0.1 seconds for instances of up to 100 jobs and only around 3.2 seconds for the largest instance tested (500 jobs and 20 machines). This confirms the excellent scalability of the algorithm, which remains computationally feasible even for large-scale scheduling problems.

Figure 14 presents the Gantt chart corresponding to the schedule obtained by the NEH heuristic for the instance tai009_20_5. Each horizontal bar represents the sequence of operations of a job across the five machines. It can be observed that no jobs overlap on the same machine, as each machine processes a single job at a time, following the feasibility constraints of the flow-shop model. Likewise, each job strictly follows the operation order from Machine 1 to Machine 5, ensuring that every job is processed sequentially through all machines.

The chart highlights the balanced allocation of processing times and the overlapping of operations across different machines, which reduces idle periods and contributes to the overall makespan improvement. Compared with the baseline schedule, NEH produces a more compact and continuous workflow, confirming its efficiency in generating near-optimal sequences that fully utilize the available machine resources.

References

Juan, A. A., Corlu, C. G., Tordecilla, R. D., de la Torre, R., & Ferrer, A. (2020). On the use of biased-randomized algorithms for solving non-smooth optimization problems. *Algorithms*, *13*(1). https://doi.org/10.3390/a13010008

Juan, A. A., David Kelton, W., Currie, C. S., & Faulin, J. (2018). Simheuristics applications: Dealing with uncertainty in logistics, transportation, and other supply chain areas. *2018 Winter Simulation Conference (WSC)*, 3048–3059. https://doi.org/10.1109/WSC.2018.8632464

Calvet, L., de Armas, J., Masip, D., & Juan, A. A. (2017). Learnheuristics: Hybridizing metaheuristics with machine learning for optimization with dynamic inputs. *Open Mathematics*, *15*(1), 261–280. https://doi.org/doi:10.1515/math-2017-0029

Reyes-Rubiano, L., Ferone, D., Juan, A. A., & Faulin, J. (2019). A simheuristic for routing electric vehicles with limited driving ranges and stochastic travel times. *SORT-Statistics and Operations Research Transactions*, *43*(1), 3–24. https://raco.cat/index.php/SORT/article/view/356179

Doering, J., Nieto, A., Juan, A. A., & Perez-Bernabeu, E. (2022). Biased-randomized algorithms and simheuristics in finance & insurance [Available online]. *Boletín de Estadística e Investigación Operativa (BEIO)*, *38*(1), 1–16. https://www.seio.es/beio/biased-randomized-algorithms-and-simheuristics-in-finance-insurance/

Faulin, J., Gilibert, M., Ruiz, R., Juan, A. A., & Vilajosana, X. (2008). Sr-1: A simulation-based algorithm for the capacitated vehicle routing problem. *Proceedings of the 2008 Winter Simulation Conference*, 2579–2587. https://www.informs-sim.org/wsc08papers/341.pdf

Juan, A. A., Faulin, J., Ruiz, R., Barrios, B., & Caballé, S. (2010). The SR-GCWS hybrid algorithm for solving the capacitated vehicle routing problem. *Appl. Soft Comput.*, *10*(1), 215–224.

Ferone, D., Gruler, A., Festa, P., & Juan, A. A. (2019). Enhancing and extending the classical grasp framework with biased randomisation and simulation. *Journal of the Operational Research Society*, *70*(8), 1362–1375. https://doi.org/10.1080/01605682.2018.1494527

Weise, T. (2020). Jsspinstancesandresults: Results, data, and instances of the job shop scheduling problem. https://github.com/thomasWeise/jsspInstancesAndResults

Juan, A. A., Lourenço, H. R., Mateo, M., Luo, R., & Castella, Q. (2014). Using iterated local search for solving the flow-shop problem: Parallelization, parametrization, and randomization issues. *International Transactions in Operational Research*, *21*(1), 103–126. https://doi.org/https://doi.org/10.1111/itor.12028