FACULTY OF ENGINEERING TECHNOLOGY

**KU LEUVEN - BRUGES**

# Applied AI: Academic Perspectives

Combinatorial optimization

**Thijs Alens**
**Emin Hilali**
**Nicolaas Danneels**

Professor: Haspeslagh S.

Academic year 2025-2026

## TABLE OF CONTENTS

# 1 INTRODUCTION

The problem that was presented to us to solve is the Team Orienteering Problem with Time Windows (TOPTW). This problem tries to create K routes for K vehicles with a capacity of Q to serve n customers. Each customer has a time window in which they need to be served and a profit linked to it. A focus was placed only on hard time windows (no soft ones) to get clearer comparisons of search parameters. The goal is to maximize the profit while minimizing the driving distance (since fuel costs as well).

Below is an overview of the parameters that are used in the problem.

| Symbol | Explanation |
|---|---|
| **K** | Number of vehicles / routes |
| **Q** | The capacity of a single vehicle |
| **N** | The number of customers in a problem |
| **$T_{max}$** | The maximum time the vehicles have to serve all the customers along their route |

The problem was solved using two algorithms: Deterministic Hill-climbing Local Search and Adaptive Large Neighbourhood Search (ALNS). Both of these algorithms require an initial solution and some way to evaluate a solution. These things will be discussed first, before going in depth on the algorithms themselves and the tuning of them.

## 1.1 Initial solution

A greedy algorithm was used to create an initial solution. The algorithm starts by creating K "empty" (from depot at time 0 to depot at time $T_{max}$) routes. It tries to insert the most promising customer into any route. The most promising customer is decided by maximizing the "profit density", which is the profit-to-cost ratio, and placing them in the cheapest available slot. If it does not fit in any of the routes, it is discarded, otherwise it is added. This approach was kept simple rather than using a more complex method like regret-2. It is simpler, gives a worse solution, but it is a bit faster.

## 1.2 Objective function

The goal (the objective function) is to maximize total profit minus distance cost (times a weight alpha) minus vehicle cost (as shown in the formula below). However, the fixed cost for vehicles was set to zero. This will not impact the comparison between solutions since every solution has K set to 3. This was done because the benchmarks require it and so we could focus purely on finding high-quality routes, not on minimizing the number of vehicles.

$$F(solution) = MAX\left[\sum_i S_i - \alpha \cdot \sum_{i,j} d_{i,j} - \beta \cdot K\right]$$

# 2 Method 1: Deterministic Hill-climbing Local Search

## 2.1 The algorithm

The local search algorithm starts from an initial solution and tries to improve it by doing one of four things (we used four operators, there are more operators out there). The four we used are explained in the table below.

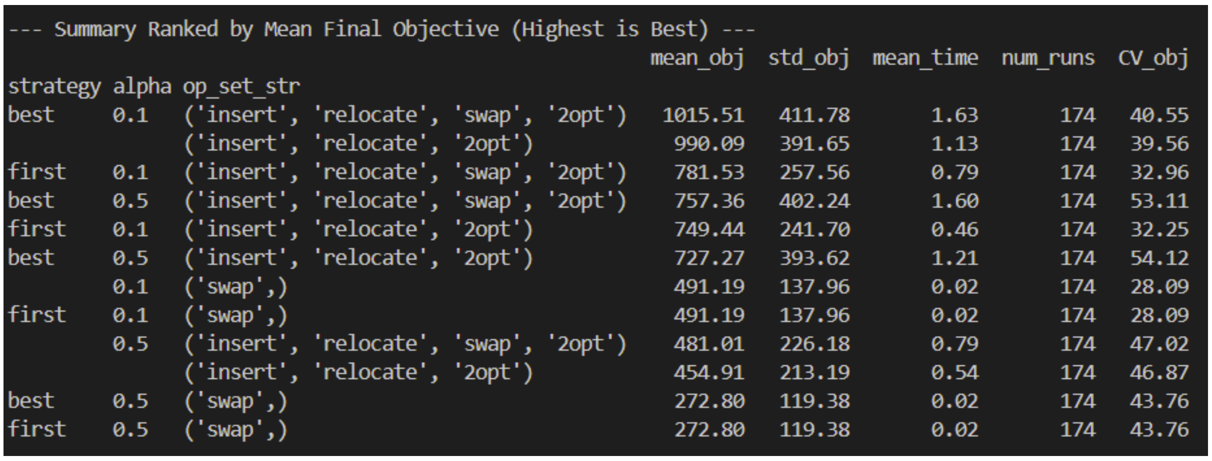
| Operator | Meaning |
|---|---|
| **Insert** | Takes a new customer and finds the best feasible spot to add them into an existing route |
| **Relocate** | Takes one customer who is already in a route and moves them to a new position, either within the same route or in a different route |
| **Swap** | Takes two customers from two different routes and swaps them |
| **2opt** | It takes a route, reverses a segment of it, and reconnects the ends |

After each alteration of the solution, the new solution is evaluated. If the objective function has not gone up in value, the solution is discarded. If a solution has gone up in value, one of two things can happen (there are other possibilities, but we have implemented two). If First-Improvement (FI) is used, the new solution gets accepted and used in the following iteration as base. If Best-Improvement (BI) is used, the new solution gets compared to all the other solutions generated by this iteration. When all the mutations of the original solution are done, the best new solution is picked as a base for the next iteration.

## 2.2 Tuning and results

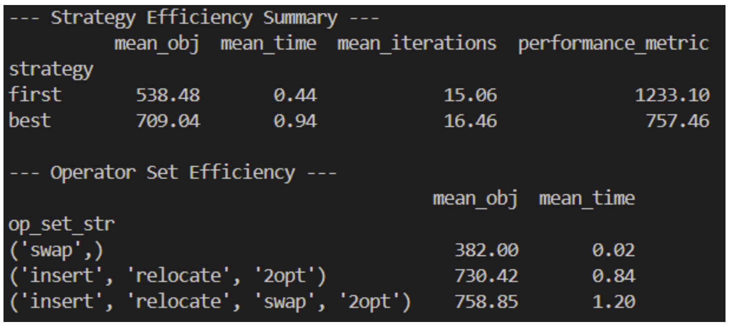
**Best-Improvement vs First-Improvement**

- Best-Improvement (BI) with all operators ("Max Coverage") found the highest-profit solutions, but it was also the slowest.
- The top First-Improvement (FI) setup was nearly as good (ranking 3rd for profit) but finished in less than half the time.



```
--- Summary Ranked by Mean Final Objective (Highest is Best) ---
                                                   mean_obj  std_obj  mean_time  num_runs  CV_obj
strategy alpha op_set_str
best     0.1   ('insert', 'relocate', 'swap', '2opt')  1015.51   411.78       1.63       174   40.55
               ('insert', 'relocate', '2opt')          990.09   391.65       1.13       174   39.56
first    0.1   ('insert', 'relocate', 'swap', '2opt')   781.53   257.56       0.79       174   32.96
best     0.5   ('insert', 'relocate', 'swap', '2opt')   757.36   402.24       1.60       174   53.11
first    0.1   ('insert', 'relocate', '2opt')          749.44   241.70       0.46       174   32.25
best     0.5   ('insert', 'relocate', '2opt')          727.27   393.62       1.21       174   54.12
         0.1   ('swap',)                               491.19   137.96       0.02       174   28.09
first    0.1   ('swap',)                               491.19   137.96       0.02       174   28.09
         0.5   ('insert', 'relocate', 'swap', '2opt')   481.01   226.18       0.79       174   47.02
               ('insert', 'relocate', '2opt')          454.91   213.19       0.54       174   46.87
best     0.5   ('swap',)                               272.80   119.38       0.02       174   43.76
first    0.5   ('swap',)                               272.80   119.38       0.02       174   43.76
```

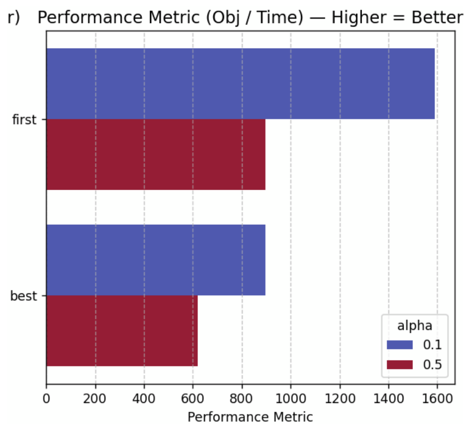
When we used a metric that balanced both profit and runtime, First-Improvement (FI) was the clear winner. This suggests its speed is a major advantage.



```
--- Strategy Efficiency Summary ---
            mean_obj  mean_time  mean_iterations  performance_metric
strategy
first         538.48       0.44            15.06             1233.10
best          709.04       0.94            16.46              757.46

--- Operator Set Efficiency ---
                                        mean_obj  mean_time
op_set_str
('swap',)                                 382.00       0.02
('insert', 'relocate', '2opt')            730.42       0.84
('insert', 'relocate', 'swap', '2opt')    758.85       1.20
```

**Operator Sets**

The "Balanced" operator set (insert, relocate, 2opt) gave the best all-around performance, hitting a good sweet spot between profit and speed.



**Distance Weights (alphas)**

The results were consistent across both alpha values. As expected, the lower alpha (profit-focused) led to higher profits. More importantly, FI had the better performance metric regardless of the weight, which confirms our main finding.

# 3 METHOD 2: SIMULATED ANNEALING

## 3.1 The algorithm

Simulated Annealing (SA) is a search algorithm that improves on the Local Search algorithm. It does this by running the LS to alter the solution.

After the change, the new solution's cost is evaluated.

- If the new solution is better than the old one, it is always accepted.
- If the new solution is worse, it might still be accepted based on a probability.

This probability is controlled by a temperature (T). The algorithm starts with a high temperature, where it accepts almost any move, even bad ones. This allows it to explore many different parts of the solution space and avoid getting stuck on the first good-but-not-great solution it finds.

The algorithm then slowly "cools down" by gradually lowering the temperature. As T gets lower, the algorithm becomes much less likely to accept a worse move (it converges). By the end, the temperature is near zero, and the algorithm will only accept moves that improve the solution, allowing it to fine-tune the best solution it found.

## 3.2 Tuning and results

The parameters to tune and their meaning are presented in the table below.

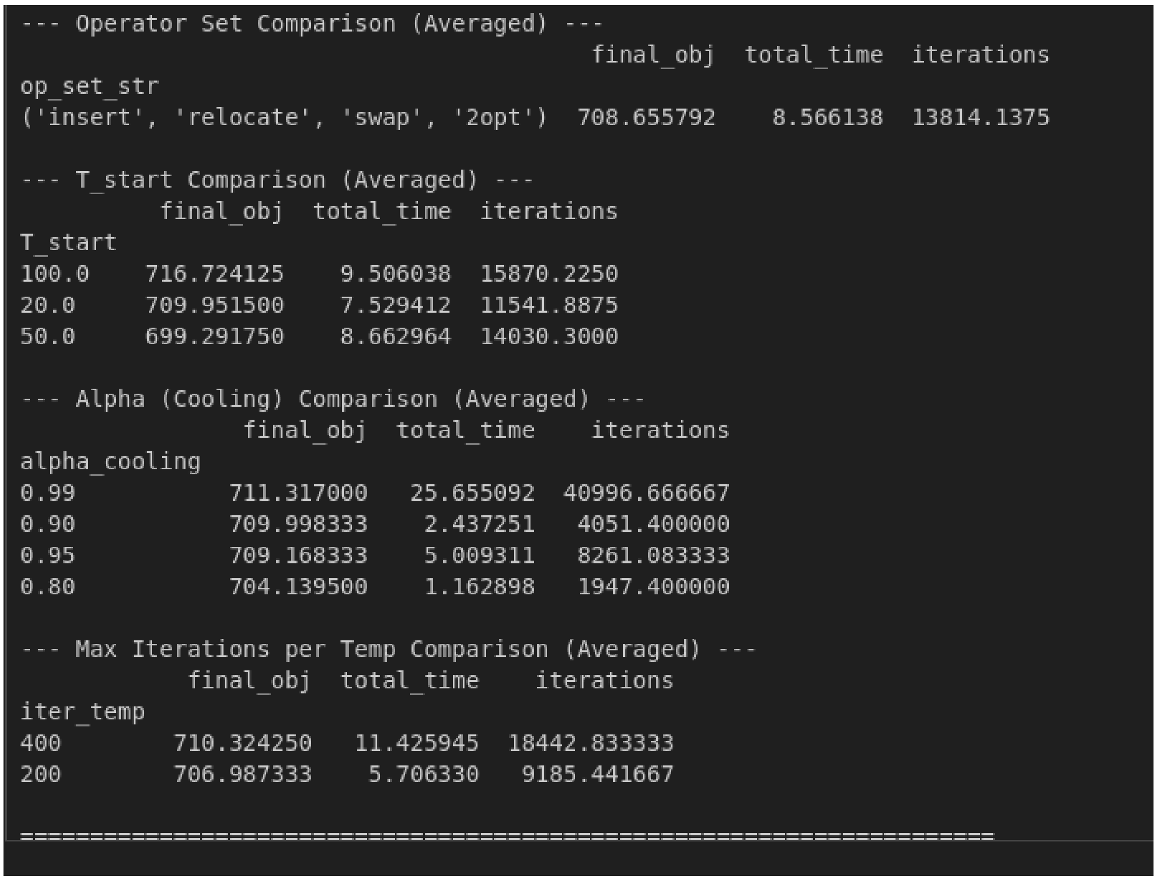
| Parameter | Meaning |
|---|---|
| **$T_0$** | The starting temperature |
| **α** | The rate at which the temperature drops |
| **L** | The number of iterations executed at a fixed temperature |
| **Operators** | These are the operators used in the local search part of the algorithm. The best performing (highest obj. value) combination, according to the test in method 1, was used. |

As in the first method, different tuning combinations were used to get a feel for what parameters work best. In contrast to the first tests, only a subset of the benchmarks were used, this due to time constraints.

The results show that the best algorithm, according to the objective function, was the one with the highest $T_0$, the largest α, and a high L. The problem is that this also took the longest to calculate. The high $T_0$ and α make it so the algorithm explores a lot of neighbourhoods, especially in the beginning, this is a good thing, but takes a long time.

Interesting to see is the fact that for a $T_0$ = 20 the objective value is higher then for 50. This 20 was calculated using the 80% acceptance for worsening moves at the start.

By decreasing α, and thus making it cool down faster, we can force the algorithm to converge faster. This results in a slightly lower objective value, but a much faster calculation. The drop in objective value is almost negligible, so there is no need to keep α that high.

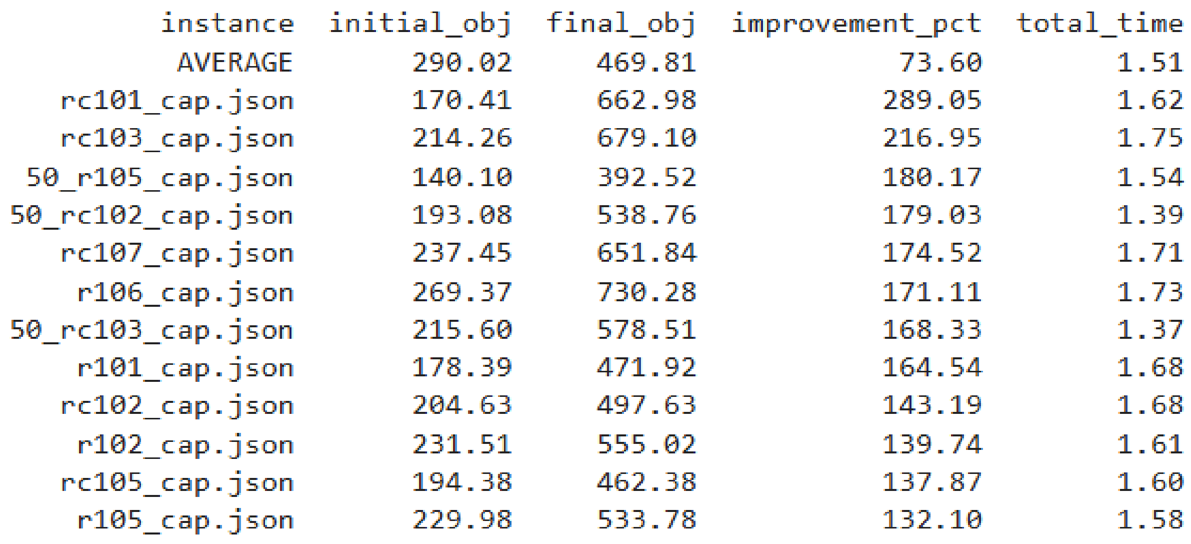


```
--- Operator Set Comparison (Averaged) ---
                                          final_obj  total_time  iterations
op_set_str
('insert', 'relocate', 'swap', '2opt')  708.655792    8.566138  13814.1375

--- T_start Comparison (Averaged) ---
          final_obj  total_time  iterations
T_start
100.0    716.724125    9.506038  15870.2250
20.0     709.951500    7.529412  11541.8875
50.0     699.291750    8.662964  14030.3000

--- Alpha (Cooling) Comparison (Averaged) ---
                final_obj  total_time    iterations
alpha_cooling
0.99           711.317000   25.655092  40996.666667
0.90           709.998333    2.437251   4051.400000
0.95           709.168333    5.009311   8261.083333
0.80           704.139500    1.162898   1947.400000

--- Max Iterations per Temp Comparison (Averaged) ---
            final_obj  total_time    iterations
iter_temp
400        710.324250   11.425945  18442.833333
200        706.987333    5.706330   9185.441667

======================================================================
```

When looking at L, the solution improves when it can explore more neighbourhoods per temperature step. This diversifies the search, making it (slightly) better, but making it twice as long to run, so it is not really worth it.

As a final run, using our findings and optimal tuning for time vs objective value, on all the benchmark data, there were a few extra interesting things to point out. The algorithm performed better on more complex benchmarks (the rc-benchmarks are found at the top). This is probably because the initial solution was quite weak, and thus had a lot of things to improve. The simpler benchmarks already had quite good initial solutions, and thus did not need much improvements. The average time spent on a single SA search was also not that long (around 2 seconds), this is a result of the tuning, possibly not getting the maximum out of the solution.

A part of the results is shown below, for the full output, consult the notebook.



```
          instance  initial_obj  final_obj  improvement_pct  total_time
           AVERAGE       290.02     469.81            73.60        1.51
    rc101_cap.json       170.41     662.98           289.05        1.62
    rc103_cap.json       214.26     679.10           216.95        1.75
  50_r105_cap.json       140.10     392.52           180.17        1.54
 50_rc102_cap.json       193.08     538.76           179.03        1.39
    rc107_cap.json       237.45     651.84           174.52        1.71
     r106_cap.json       269.37     730.28           171.11        1.73
 50_rc103_cap.json       215.60     578.51           168.33        1.37
     r101_cap.json       178.39     471.92           164.54        1.68
    rc102_cap.json       204.63     497.63           143.19        1.68
     r102_cap.json       231.51     555.02           139.74        1.61
    rc105_cap.json       194.38     462.38           137.87        1.60
     r105_cap.json       229.98     533.78           132.10        1.58
```

## 4 FUTURE WORK

The algorithms we have implemented keep the number of vehicles available at 3 (K=3). However, this is something to explore further, since it can be used to optimize the algorithm even more. It would result in a more realistic problem, which is harder to solve, but definitely something to look into. It can be easily tested using our code, as only K should be adapted.

It can also be interesting to look into a more realistic objective function. The objective function used, is tuned with a factor alpha to make the factors a bit more balanced. However, it does not really represent the real world. By gathering some data about the fuel usage of the vehicles plus the cost of a driver, a more accurate objective function could be created, making the results of the algorithms more profound.