Exact and metaheuristic methods for the symmetric Travelling Salesman Problem

Alberto Schiabel ✉, **1236598**

Methods and Models for Combinatorial Optimization • February 22, 2021
Supervisor: **Prof. Luigi de Giovanni**
Department of Mathematics, University of Padova

Abstract

In this report, we discuss two different implementations of the symmetric Travelling Salesman Problem in the context of minimizing the time taken by a board drilling machine. The first implementation is an exact Mixed Integer Linear Programming model based on a compact network flow formulation using `IBM CPLEX`. The second approach uses a metaheuristic method based on Genetic Algorithms, Local Search, and the Farthest Insertion heuristic. Multiple crossover and mutation operators as well as several strategies for evolving the genetic pool are examined. We test our implementations on both real-world datasets and simulated (but hopefully realistic) instances, with sizes ranging from 10 to 1665 nodes. The metaheuristic method is calibrated on a subset of the simulated datasets. Finally, we present a detailed comparison between the exact and metaheuristic implementations in terms of running times and gaps with respect to the optimal solutions, also confronting the two algorithms with a random baseline method. We perform a benchmark of our implementations running the same TSP instances 11 times with multiple timeout limits and collecting relevant about the fluctuations of the given solutions. We finally observe that our metaheuristic implementation, although less stable and precise for smaller datasets, should be preferred in many practical cases, especially when a solution should be computed in a matter of seconds.

This document has been written in partial fulfillment of the requirements for the *Methods and Models for Combinatorial Optimization* class, supervised by Professor Luigi de Giovanni. The source code for the TSP solvers and scripts documented in this report is available at https://github.com/jkomyno/combinatorial-optimization-tsp. We used `C++17` for the implementing the TSP methods, and `Python3` for collecting statistics about our programs, calibrating the metaheuristic method, and creating many of the tables and visualizations included in this report.

Keywords TSP, IBM CPLEX, Metaheuristics, Genetic Algorithm, Local Search

1 Introduction

This report discusses the implementation of the two laboratory exercise for the *Methods and Models for Combinatorial Optimization* class. **Exercise 1**[1] introduces the board drilling machine context for which we need to implement an exact symmetric TSP implementation using a compact MILP network flow formulation. **Exercise 2**[2] asks to implement an ad-hoc optimization algorithm any metaheuristic or alternative approaches related to the content of the course, and finally to compare the performances of the two algorithms on several realistic problem instances. In particular, we chose to implement a Genetic Algorithm studying several possible crossover and mutation operators, as well as multiple neighborhood exploration strategies, aiming to help the algorithm escape local minima.
The main questions that want to address are:

- Which implementation is better for a particular problem size?
- What is the gap with respect to the optimal solution?
- How well does the metaheuristic implementation escape local minima to reach the optimal solution?
- Are the time-limited `CPLEX` and metaheuristic approaches better than a simple time-limited random search of the TSP solution?

[1] https://www.math.unipd.it/ luigi/courses/metmodoc/z01.eserc.lab.01.en.pdf
[2] https://www.math.unipd.it/ luigi/courses/metmodoc/z02.eserc.lab.01.en.pdf

Outline

We implemented three TSP solvers: `ex1-cplex`, `ex2-metaheuristic`, and `random-baseline`, each of which is tested with several timeout limits (100ms, 1000ms, 10000ms, and 60000ms). We decided to count the execution time only after computing the distance matrix of the TSP instance file read in input, to avoid biases in the comparison of the algorithms.

The rest of the document is structured as follows. Section 2 presents the TSP instance files used for our experiments and explains how we simulated the majority of them. Some of these datasets are dedicated to the benchmark phase, whereas a different partition is used only for calibrating the metaheuristic TSP solver. Section 3 describes the MILP implementation of the symmetric TSP using `IBM CPLEX`, as required by the first exercise of the homework. Section 4 dives deep into the several strategies we implemented for the metaheuristic algorithm, the tradeoffs between them, and some relevant implementation details. Some of the strategies we implemented for the second homework exercise are novel, i.e., we did not find them in any research paper. Section 5 briefly introduces a random TSP solver used as a baseline to compare the exact and the metaheuristic implementations. Section 6 discusses our approach to calibrating a number of parameters of the metaheuristic model, and how we allowed the user to update the default value of those parameters after the calibration step, without recompiling the `C++` code. Section 7 presents the benchmark phase, which involves multiple runs of the same algorithms to take into account the variability of random components. Section 8 gathers the evidences collected by our experiments on the drawbacks of the solvers we implemented. Section 9 concludes the main part of the report, tries to answer to the questions we addressed in the introduction, and mentions some possible improvement.

We also included a comprehensive appendix. Appendix A presents the tables for the size and optimal solutions of each dataset we used. Appendix B shows the tables of statistics and solution gaps on interrupted and non-interrupted instances with varying time limits and algorithms. Appendix C describes a synthetic summary of the overall code structure of the submitted project. Appendix D mentions the very few third-party dependencies we used, even though they are mainly needed for the data collection and analysis tasks in `Python3`. Finally, Appendix E provides a tutorial on how to build the submitted software and reproduce our experiments.

2 Datasets

The first exercise mentions the following about the the context in which the Travelling Salesman Problem should be solved:

> A company produces boards with holes used to build electric panels. Boards are positioned over a machine and a drill moves over the board, stops at the desired positions and makes the holes. Once a board is drilled, a new board is positioned and the process is iterated many times. Given the position of the holes on the board, the company asks us to determine the hole sequence that minimizes the total drilling time, taking into account that the time needed for making an hole is the same and constant for all the holes.

There exist multiple datasets containing instances for several variations of the TSP, some of which represent cities or maps. These representations, however, are not coherent with the context presented by the first exercise, and are thus inappropriate for our purposes. There are, however, some real-world instances of board-drilling contexts available in the `TSPLIB` library that we can use. We also decided to generate a number of small, synthetic instances with 10 to 200 nodes.

2.1 `TSPLIB` instances

`TSPLIB` [Rei91] is a library of sample instances for testing several variations of the Traveling Salesman Problem. We selected 5 datasets for the STSP problem from this library. We chose only instances whose distance metric was `EUC_2D` (2D-Euclidean distance) (rather than geodesic distance) and whose description was "circuit board-drilling". For computing the distance matrix required by our algorithms, we used the approximation formula of the Euclidean distance described in[3]. This was needed to obtain objective function's values comparable to the known solutions.

The 5 instances we selected have sizes $n \in \{198, 493, 657, 1291, 1655\}$, with known exact solutions.

2.2 Synthetic Instances

We created some pseudo-random semi-regular datasets that could realistically represent real-world board-drilling instances. Our purposes were:

(a) (*Mandatory*) Avoid sampling points from a uniform distribution;
(b) (*Mandatory*) Guarantee a minimum gap distance between any two points;
(c) (*Desirable*) Create semi-regular geometric grid-like shapes, while still embedding a random sampling component.
(d) (*Desirable*) Create two families of datasets with the same instance size (i.e. the same number of TSP nodes), but with a slightly different structure, so that one family can be used for calibrating the metaheuristic algorithm and the other one can be used for testing its quality metrics.

At first, we tried oversampling points and eliminating the points that are "too close" to each other with a KD-Tree data structure [Ben75]. We even tried creating random blobs, applying a K-Means clustering algorithm to those blobs and returning the centroid points. However, the results we initially obtained weren't particularly satisfying, and the generated points didn't present noticeable regularities.

Random Semi-regular Grids

We opted for another strategy that plots points on a 2-dimensional limited-size surface starting from the bottom-left origin. The idea is generating many rectangular sub-grids of different dimensions and distances, but preserving the same sizes and distances every two consecutive rows of points. After the 2D surface is filled with these regular points, we prune some of them at random with uniform probability until we reach the desired instance size, obtaining semi-regular structures.

More in detail, the generation method is the following:

1. Choose a 2-dimensional domain, such as $(width = 1600, height = 900)$.
2. Select the discrete intervals for the starting point in terms of left and bottom padding, such as $\{50, 100\}$.
3. Select the discrete intervals for the width and height of the sub-grids, as well as the vertical and horizontal spacing between the sub-grids.
4. The sub-grid height and vertical spacing are chosen at random from the given discrete intervals.
5. The list of y coordinates is generated iteratively, from bottom to top, using a range that alternates the selected sub-grid height to the selected vertical spacing between sub-grids, until the domain height is reached.

[3] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf

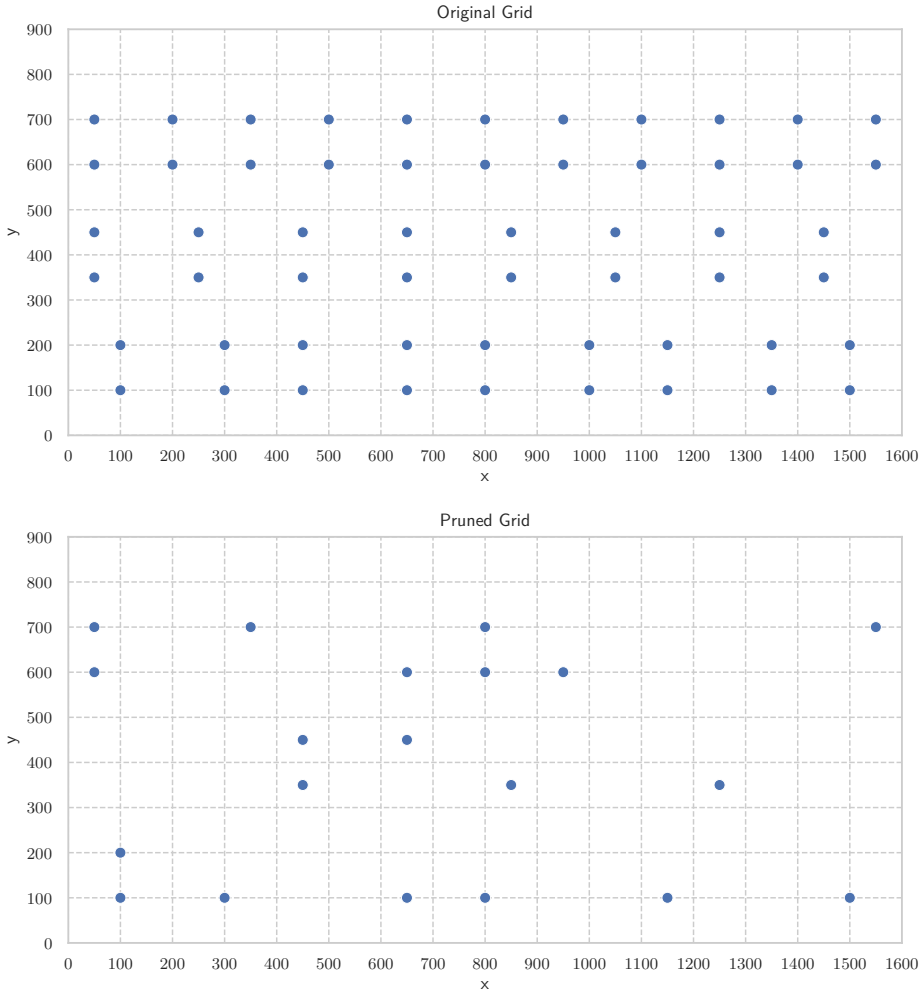


Figure 1 Comparison between a regular grid and the relative semi-regular grid with $|N| = 20$ nodes obtained by pruning the original grid.

6. The list of x coordinates is generated from left to right. For every 2 distinct and consecutive y coordinates (y_a, y_b), the sub-grid width, horizontal spacing and the left padding are chosen at random from the given discrete intervals. The x coordinates for (y_a, y_b) are generated using a range that alternates the selected sub-grid width to the selected horizontal spacing between sub-grids, until the domain width is reached.
7. Finally, the 2-dimensional points obtained are pruned at random to reach the desired TSP instance size.

We generated two random semi-regular datasets for each of each size in $\{10, 15, \ldots, 95, 100, 200\}$, i.e. considering each multiple of 5 between 10 and 100 and then considering size 200. Half of these datasets, which we refer to as `family A`, are used exclusively for the calibration step of the metaheuristic algorithm. The other half is conjunction with the `TSPLIB` instances for benchmarking purposes. We named these family of instances as `sim-grid-[F]-[n].tsp`, where `F` indicates the family type (`A` or `B`), and `n` is the instance size. For instance, the simulated semi-regular datasets of size 50 for the calibration and benchmarking phases are saved, respectively, in the `sim-grid-A-50.tsp` and `sim-grid-B-50.tsp` files.

The `.tsp` extension indicates that we use the same `TSPLIB` format used by the datasets presented in Section 2.1, for which we wrote a custom parser in `C++17`.

We thus satisfied every constraint and desirable requirement listed in Section 2.2. The simulated datasets we generated are 50: 25 of family `A` and 25 of family `B`. We computed their optimal solution using the exact `MILP` implementation presented in Section 3 with no timeout limits. For nodes with 100 nodes or else, we discovered the optimum in less than 30 minutes, whereas for the biggest simulated instance (200 nodes) the solution discovery process took more than 3 hours.

3 Exercise 1: Exact Method

3.1 Network Flow Model

The symmetric TSP can be formulated as a network flow model on a weighted complete graph $G = (N, A)$. This formulation requires only a polynomial number of variables and constraints. Let 0 be an arbitrary starting node, and let $|N| - 1$ be the amount of its outgoing flow. The idea is to push this amount of flow towards the remaining nodes such that:

(i) Each node (different than 0) receives 1 unit of flow;
(ii) Each node is visited once;
(iii) The sum of c_{ij} over all the arcs shipping some flow is minimum.

Sets:

- N = graph nodes, representing the holes;
- A = arcs $(i, j), \forall\, i, j \in N$, representing the trajectory covered by the drill to move from hole i to hole j.

Parameters:

- c_{ij} = time taken by the drill to move from i to j, $\forall\ (i, j) \in A$;
- 0 = arbitrarily selected starting node, $0 \in N$.

Decision variables:

- x_{ij} = amount of the flow shipped from i to j, $\forall\ (i, j) \in A$;
- $y_{ij} = 1$ if arc (i, j) ships some flow, 0 otherwise, $\forall\ (i, j) \in A$.

3.2 Mixed Integer Linear Programming Formulation

This is the MILP model formulation implemented using the CPLEX API, inspired by [GG78]:

$$\min \quad \sum_{i,j:(i,j)\in A} c_{ij} y_{ij} \tag{1}$$

$$\text{s.t.} \quad \sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j),j\neq 0} x_{kj} = 1 \qquad \forall\ k \in N \setminus \{0\} \tag{2}$$

$$\sum_{j:(i,j)\in A} y_{ij} = 1 \qquad \forall\ i \in N \tag{3}$$

$$\sum_{i:(i,j)\in A} y_{ij} = 1 \qquad \forall\ j \in N \tag{4}$$

$$x_{ij} \leq (|N| - 1)\, y_{ij} \qquad \forall\ (i, j) \in A, j \neq 0 \tag{5}$$

$$x_{ij} \in \mathbb{R}_+ \qquad \forall\ (i, j) \in A, j \neq 0 \tag{6}$$

$$y_{ik} \in \{0, 1\} \qquad \forall\ (i, j) \in A \tag{7}$$

▶ Note. Rather than leaving constraint (6) as it is, we restricted the x_{ij} variable bounding it to the $[0, |N| - 1]$ domain. The resulting model is still valid, as it is compatible with constraint (5).

▶ Note. We avoided inserting any loop ($(i, j),\ i = j$) in the implemented model. The resulting model is still valid, as each loop have cost equal to 0. In fact, loops do not contribute to the summations in constraints (2), (3), (4).

▶ Note. The graph that models each TSP instance is complete, so the graph is connected. This implies that a solution always exists.

3.3 Model Implementation

We implemented the first exercise in the `ex1-cplex` folder of the submitted project. The `CPLEXModel` class exposes a façade that abstracts the use of the IBM CPLEX `C` APIs from `C++`. We included the `cpx_macro.h` header file presented in the laboratory classes.

The constructor of `CPLEXModel` receives a reference to a distance matrix and the maximum number of milliseconds (`timeout_ms`) allotted for solving the linear problem. The private method `setup_lp` sets up the linear problem environment by defining the decision variables and the constraints. We first declared the decision variables, followed by the constraints. The x_{ij} decision variables are $(|N| - 1)^2$, whereas the y_{ij} decision variables are $|N| \cdot (|N| - 1)$. The helper method `add_column` is responsible for inserting a new variable in `CPLEX`'s environment using the `CPXnewcols` command. The variable names use the pattern `[v]_i_j`, where `[v]` is a `char` value that can be either `"x"` or `"y"`. Constraints are added using the `CPXaddrows` command. Only a single row or column is added at a time.

In order to allow referencing these decision variables in the constraint definitions, we saved their internal `CPLEX` position in two matrices, `x_variable_mat` and `y_variable_mat`. For instance, variable x_{ij} can be referenced as `x_variable_mat.at(i, j)`.
We modelled the coefficients of constraints (2), (3), (4) using a `std::vector<T>` data structure, extracting the raw `T*` data and passing it to the `CPXaddrows` command. In the case of constraint (5), we put both the coefficients and the two decision variables x_{ij} and y_{ij} in a stack-allocated array of size 2 before passing the pointers to the beginning of those arrays as an option to the `CPXaddrows` command.

▶ Note. Due to `CPLEX` API's restrictions, we had to transform constraint (5) to the equivalent $x_{ij} - (|N| - 1)y_{ij} \leq 0\ \forall\ (i, j)\ \in A,\ j \neq 0$ before inserting it into the linear problem.

3.4 Time Limit

If we assume that the time and memory at disposal is unbounded, the algorithm always returns the exact solution. For practical reasons, however, we need to limit the maximum time at disposal. We achieved this by calling `CPLEX`'s `CPXsetdblparam` function passing the `CPXPARAM_TimeLimit` enum identifier, which refers to the `CPLEX` solver's time limit expressed in seconds[4].

If the time limit is too low compared to the size of the problem, two different scenarios may arise:

(i) `CPLEX` computes a feasible, but approximated solution;
(ii) `CPLEX` is not even able to provide a feasible solution.

In order to handle the second case, we wrapped the result of the `CPLEXModel::get_solution()` method in a `std::optional`[5] data type.

[4] https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/TiLim.html
[5] https://en.cppreference.com/w/cpp/utility/optional

3.5 Parallelism

We leverage some level of parallelism by instructing `CPLEX` to use every available CPU thread with an *opportunistic* strategy, which is non-deterministic. According to the official documentation[6], "the opportunistic setting entails less synchronization between threads and consequently may provide better performance. [...] However, in opportunistic mode, the actual optimization may differ from run to run, including the solution time itself and the path traveled in the search".

4 Exercise 2: Metaheuristic Method

We implemented a genetic algorithm that extends the basic structure presented in [ES15], using the Farthest Insertion constructive heuristic to generate the initial solution and trying to combine multiple local search methods. Given our personal interest in genetic algorithms, we invested time into researching and implementing several strategies for selecting and evolving pools of encoded TSP solutions, mutating them and merging two solutions to generate a new offspring. Even though we selected a subset of the studied strategies for the final metaheuristic calibration phase, we give the expert user the possibility to choose among several strategies at compile time by modifying the template parameters of the `TSPSolver` class. There is no run time penalty due to repeated invocations of branch instructions because we used `C++17`'s constant conditionals using the `if constexpr` keywords. The metaheuristic method can be found in the `ex2-metaheuristic` folder.

We first compared the various implemented strategies *by hand*, trying a subset of model hyperparameters that would allow us to perform quick development iterations. After we empirically identified the most promising genetic algorithm strategies, we set those strategies as default and we fine-tuned their parameters during the *calibration* phase (see Section 6). A summary of the final model that gives the overall structure of the metaheuristic implementation is given in Section 4.12. In our genetic algorithm implementation, the population pool has a fixed even size μ, and the offspring pool has a fixed even size λ, with $\lambda > \mu$. If two paths of the population pool are selected for mating (*crossover*), they will always generate two offsprings. The new offsprings may be affected by mutations.

4.1 Solution Representation

The first stage of building any evolutionary algorithm is to decide the genetic representation of a candidate solution to the problem. This kind of representation is commonly referred to as *chromosome*. We use the **permutation representation**: a permutation of a fixed set of n integer values is used to represent admissible solutions. We numbere instance nodes from 0 to $|N| - 1$. The TSP problem is based on adjacency (and not on order), so e.g. if $|N| = 4$, both $[0, 1, 2, 3]$ and $[1, 2, 3, 0]$ represent the same solution. This permutation representation is implemented using a `std::vector<size_t>` container, which allows $\mathcal{O}(1)$ random access lookup and is CPU cache-friendly, since the `C++` standard guarantees that the data in `std::vector` is stored contiguously[7]. Two nodes in a path can be swapped in $\mathcal{O}(1)$ with no memory copies. As a convenience, we use the node labelled as 0 as depot node, i.e., it is the first and last node of the Hamiltonian circuit solution.

4.1.1 Feasibility Criterion

We decided that every permutation path representation should always represent feasible solutions. We initially considered accepting non-feasible representations with a probability p_a if the solution was admissible, possibly *fixing* the solution after if an unfeasible solution was accepted. The fixing

[6] https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/ParallelMode.html
[7] https://github.com/cplusplus/draft/blob/master/papers/n4835.pdf, Paragraph 22.3.11.1

process would take an unfeasible solution that violates the permutation representation property, and substitute every repeated node in $\{0, \ldots, n-1\}$ with a missing index in ascending order, restoring the admissibility of the circuit. However, we decided to get rid of this approach because we noticed much worse run-time performance without obtaining better TSP results in our initial experiments.

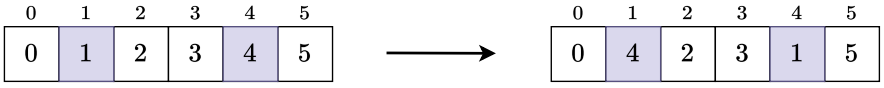
▶ Note. Since our metaheuristic algorithm only deals with feasible solutions, we only focused on mutation operators that preserve the permutation representation property.

4.2 Population Initialization

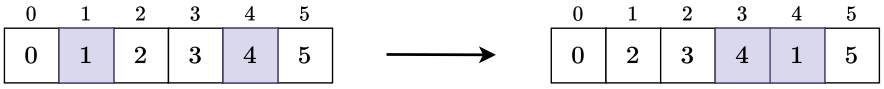
▶ **Definition 1** (Population pool). *We define the* population pool *of our genetic algorithm as a* `std::vector` *of permutation paths with fixed size* μ.

We initialize the population pool as follows:

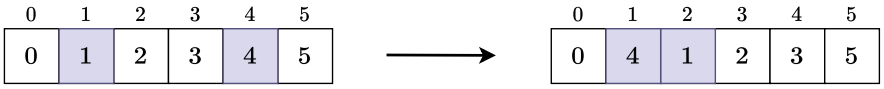
(i) We select an initial solution s_0 using the **Farthest Insertion** constructive heuristic strategy;
(ii) We add s_0 to the population pool.
(iii) We perform $\mu - 1$ random permutations of a copy of the initial solution s_0 and we add the resulting paths to the population pool until it reaches size μ.



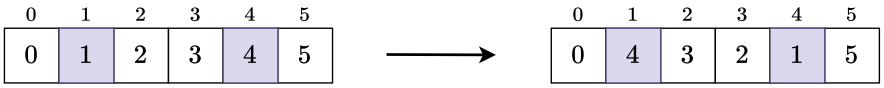
(a) Swap mutation.



(b) Left rotation mutation.



(c) Right rotation mutation.



(d) Insertion mutation.

Figure 2 Results of different mutation operators on an example circuit path $[0, 1, 2, 3, 4, 5]$ with two selected indexes $i = 1,\ j = 4$.

4.3 Mutation Operators

Mutation is the name given to variation operators that use a single parent a create one offspring by performing a randomised perturbation to change the representation. Each mutation operator we implemented requires two nodes i, j to be selected, which mark the solution area affected by the mutation. Mutation is regulated by a model parameter p_m, the *mutation probability*. For a path with n nodes, we generate $n - 1$ uniform probabilities, one for each node (except the *depot* node), and we select the pair of nodes whose probability is $\leq p_m$, proceeding to the mutation. We repeat the process for every path in the offspring pool.

We implemented multiple mutation operators, which are presented in the following paragraphs.

Swap Mutation

Two distinct positions $i,\ j$ (called *genes*) in a given chromosome are randomly chosen and their values are swapped. An example of **Swap** mutation is presented in Figure 2a.

Left Rotation Mutation

Two distinct positions $i,\ j$ are selected at random, and the first position is placed immediately after the second one, shifting the other positions originally in $[i, j]$ to the left. A practical example of the **Left Rotation** operator can be seen is Figure 2b. This is the mutation operator we included in the final model.

Right Rotation Mutation

The **Right Rotation** mutation operator is the symmetric of the Left Rotation operator. Two distinct positions $i,\ j$ are selected at random, and the second position is placed immediately before the first one, shifting the other positions originally in $[i, j]$ to the right. An example of it is presented in Figure 2c.

Inversion Mutation

Two distinct positions $i,\ j$ in the chromosome are selected and the order in which the values appear in the $[x, y]$ range is reversed. The original circuit is thus teared into three partitions, with all edges inside a partition being preserved, and only the edges between the two partitions being broken. This type of mutation is thus the smallest change that can be performed on an adjacency-based problems, and it is the basic move behind the 2-opt search heuristic for TSP [LK73]. The **Inversion** mutation's effects are shown in Figure 2d.

4.4 Crossover Operators

Implementing crossover operators for permutation representations presents some difficulties, because a generic substring exchange between parents does not necessarily maintain the permutation property. We focused on a specialized operator specifically designed for permutation representations: the **Order** crossover, which we implemented in two different variants.

If two parents are selected, they always generate two offsprings. We do not check whether two paths are actually different before mating them: we rely on the diversification that happens in the selection and mutation phases.

Order Crossover (`OX`)

The Order crossover operator was first designed by Davis in [Dav91]. It has the purpose of sharing information to the first offspring about the relative order from the second parent, and to the second offspring about the relative order from the first parent. The steps of the crossover operators are similar to those of the more famous *Partially Mapped Crossover* (`PMX`) operator, and are presented as follows (only the generation of the first offsprings considered, the second offspring can be generated in a specular fashion).

- **(i)** Two distinct crossover points i, j are chosen, and the segment in $[i, j]$ in the first parent p_0 is copied into the first child c_0.
- **(ii)** Starting from the j^{th} point of the second parent p_1, the remaining unused numbers are copied into c_0 in the order that they appear in p_1, wrapping around at the end of the list.
- **(iii)** The second offspring c_1 can be created in a similar manner, with the parents' role reversed.

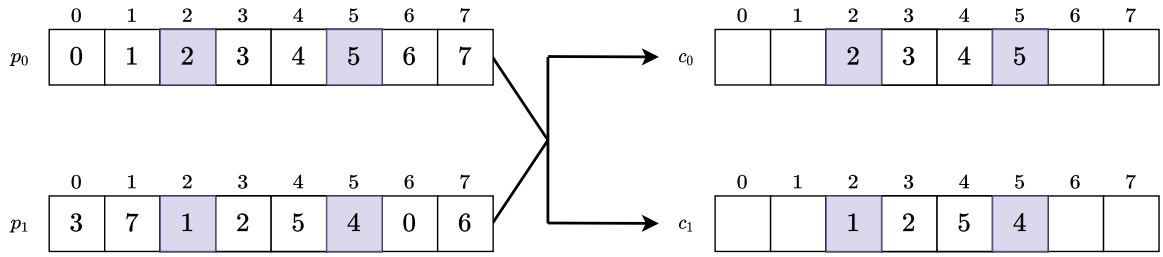
A practical example of the `OX` crossover operator is shown in Section 4.4.

Alternative Order Crossover (`OX-Alt`)

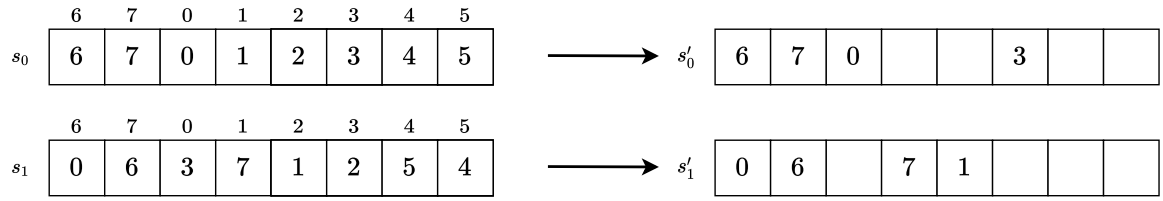
A limitation of the original `OX` crossover is that the same cut points are used for both parents, potentially limiting the expressiveness and the offspring generation possibilities. We propose an alternative variation (called `OX-Alt`) in which two pairs of cut points are used. Not only the cut points are different, but also the size of the subpaths between the cut points in both parents are in general not equal. The other implementation steps are similar to the `OX` operator's, and are presented in an example in Figure 4. `OX-Alt` is the crossover operator we implemented in the final model.

4.5 Parent Selection

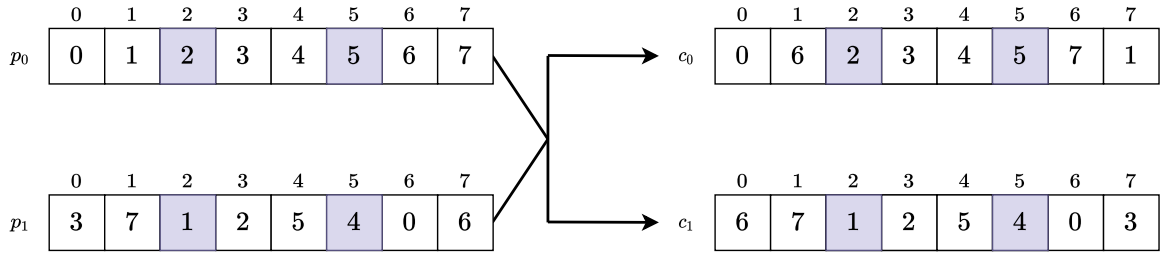
We decided to avoid implementing Fitness Proportional Selection (FPS) [Hol92] because past experiments suggest that outstanding individuals take over the entire population very quickly, leading to **premature convergence**. Instead, we tried to alternative approaches: **Ranking Selection** and **Tournament Selection**.



(a) The segment between the two crossover points $i = 2, j = 5$ from p_0 is copied into c_0, and from p_1 it is copied to c_1.



(b) s_0 shows the sequence of cities of p_0 starting from immediately after the second crossover point j. s_1 shows the analogous shifted sequence of cities of p_1. s'_0 is obtained from s_0 by removing the cities in the segment that had been copied to p_1: $[1, 2, 5, 4]$. Analogously, s'_1 is s_1 without the segment that had been copied to p_0, $[2, 3, 4, 5]$.



(c) The cities in s'_0 are copied in order of appearance to c_1, starting from the beginning of the list. The same happens for s'_1 and c_0.

Figure 3 Recombination of two parents $p_0 = [0, 1, 2, 3, 4, 5, 6, 7], p_1 = [3, 7, 1, 2, 5, 4, 0, 6]$ using the `OX` crossover operator with two selected indexes $i = 2$, $j = 5$. The recombination process generates two offspring children: c_0 and c_1.

Ranking Selection

Ranking selection is a selection method proposed by Baker in [Bak87] after observing the drawbacks of FPS. It preserves a constant selection pressure on the population pool w.r.t fitness, allocating selection probabilities to chromosomes according to their rank, rather than their absolute fitness values.

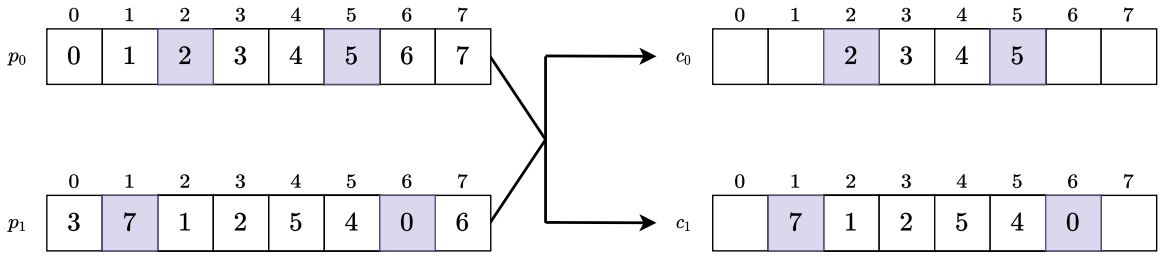
In particular, we implemented an **Exponential Ranking** scheme, where the selection probability for an individual of rank i is:

$$P_{\text{rank}}(i) = \frac{1 - e^{-i}}{c} \tag{8}$$

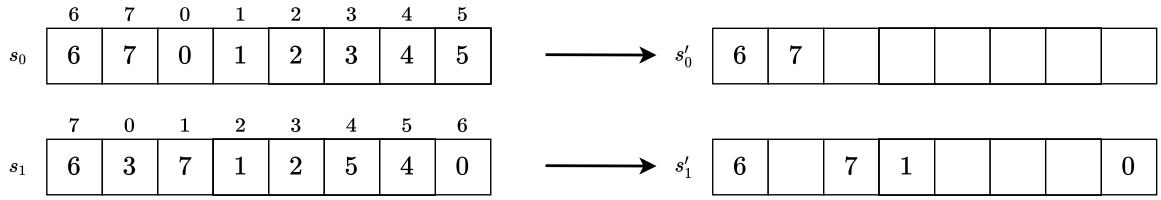
where c is a normalization factor chosen such that the sum of probabilities for each rank in $0 \dots n-1$ equals to 1. Rather than preserving constant **selection pressure** like the *Linear Ranking* strategy, the exponential ranking implies a higher selection pressure, with more emphasis on selecting individuals of above-average fitness.

The basic steps of the Exponential Ranking Selection algorithm are:

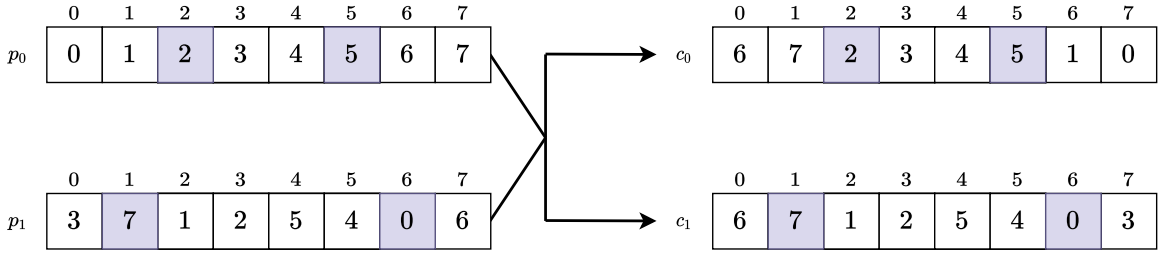
(i) Sort the population pool of size μ according to the paths' costs, such that the path with the



(a) The segment between the two crossover points $i_0 = 2, j_0 = 5$ from p_0 is copied into c_0. Conversely, the segment between the two points $i_1 = 1, j_1 = 6$ from p_1 is copied into c_1.

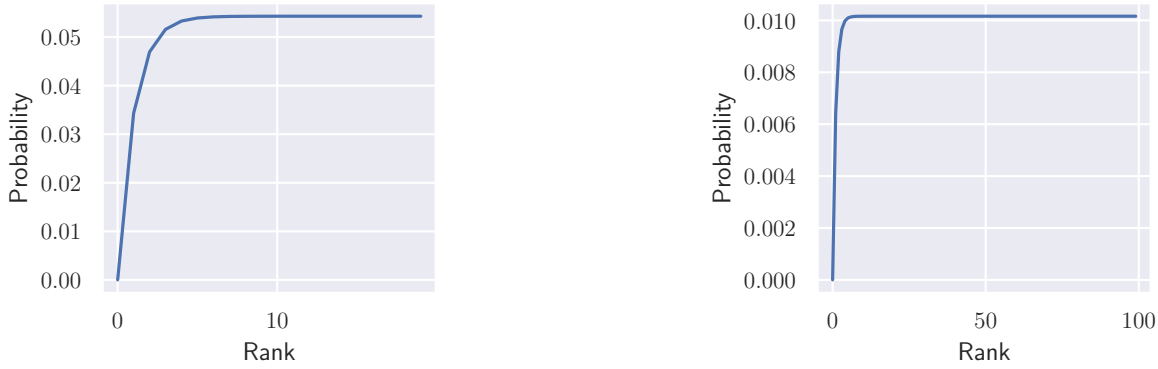


(b) s_0 shows the sequence of cities of p_0 starting from immediately after point j_0. s_1 shows the analogous shifted sequence of cities of p_1 starting from after j_1. s'_0 is obtained from s_0 by removing the cities in the segment that had been copied to p_1: $[7, 1, 2, 5, 4, 0]$. Analogously, s'_1 is s_1 without the segment that had been copied to p_0, $[2, 3, 4, 5]$.



(c) The cities in s'_0 are copied in order of appearance to c_1, starting from the beginning of the list. The same happens for s'_1 and c_0.

Figure 4 Recombination of two parents $p_0 = [0, 1, 2, 3, 4, 5, 6, 7], p_1 = [3, 7, 1, 2, 5, 4, 0, 6]$ using the `OX-Alt` crossover operator. Two selected pairs of indexes are used: $(i_0 = 2,\ j_0 = 5)$ for p_0, and $(i_0 = 1,\ j_0 = 6)$ for p_1. The recombination process generates two offspring children: c_0 and c_1.



(a) $|N| = 20$.

(b) $|N| = 100$.

Figure 5 Exponential ranking probabilities trend with $|N| = 20$ (left) and $|N| = 100$ (right).

highest cost (worst fitness) is moved to the front and the path with the lowest cost (best fitness) is moved to the end of the population pool.

(ii) Assign the rank number in ascendant order to the sorted population pool, starting from 0: the worst path will have rank 0, the second-to-worst path will have rank 1, and the best path will have rank $n - 1$.

(iii) Assign to each rank the probability of being selected using the formula in Equation (8).

(iv) Sample with replacement λ candidate solutions from the population pool.

▶ Note. With the Exponential Ranking selection, the worst solution is never selected because $P_{\text{rank}}(0) = (1 - e^{-0})/c = 0$.

Tournament Selection

The Tournament selection operator has two interesting properties:

(i) It does not require any global knowledge of the population;

(ii) It does not require a quantifiable measure of quality.

Tournament selection relies on a complete ordering relation that can compare and order any two chromosomes in a population pool. λ members are selected from a pool of μ individuals. In our implementation, the sampling happens with replacement.

The basic steps of the algorithm are the following:

(i) Initialize an empty mating pool.

(ii) Pick k individuals randomly with replacement from the population pool.

(iii) Compare these k individuals and select the best of them with probability p or the second best with probability $1 - p$.

(iv) Add the selected individual to the mating pool.

(v) Iterate from (ii) until the mating pool reaches size λ.

▶ Note. The rank of an individual in a population pool is estimated without needing to sort the entire population.

▶ Note. When sampling with replacement, it's possible that even the second-to-least-fit member of the population is selected, since with probability $\frac{1}{\mu^k} \geq 0$ all tournament candidates will be copies of that member.

▶ Note. The greater the value of k, the higher the selection pressure.

▶ Note. The lower the value of p, the lower the selection pressure.

In the final model, we opted for the *deterministic* tournament selection, where p is set to 1 (the best element is selected every time).

4.6 Recombination

In the recombination step, λ offsprings are generated from λ parents. Thus, the number of parents selected for mating is equal to the size of the new offspring pool. Once we gather the parents using Tournament Selection, we select pairs of parents (p_0, p_1) to generate pairs of children (c_0, c_1) with a fixed crossover operator (`Order-Alt`).

We implemented two pair-picking strategies for parents: **Sequential recombination** and **Random recombination**. In both cases, all the parents present in the mating pool are used only once. In the final model, we used the Sequential recombination strategy. The recombination might fail with probability $1 - p_c$, where p_c is the *crossover rate*.

Sequential Recombination

The **Sequential recombination** is the simplest case: the λ paths in the mating pool are sequentially divided into $\frac{\lambda}{2}$ partitions (the first 2 elements p_0 and p_1 fall into the first partition, the next two elements p_2 and p_3 fall into the second partition, and so on). We then apply the crossover operator to the parents in the same partition, and return the generated offsprings for all partitions.
For consistency reasons, we require both λ and μ to be even numbers.

Random Recombination

The **Random recombination** follows the same steps are the Sequential Recombination, but the $\frac{\lambda}{2}$ partitions are created at random. The idea is that choosing parents at random from a pre-selected mating pool should help escape plateaus more effectively than a deterministic choice of parent pairs.

4.7 Survivor Selection

The **Survivor selection** mechanism is the process responsible of reducing the memory of a genetic algorithm from a multiset of μ parents and λ offsprings to a multiset of μ individuals that compose the next generation. We used the **generational model**: all the parents are discarded, and the whole population pool is replaced with a selection of μ offsprings. Our hope is that discarding all the parents helps the algorithm escape local optima more effectively.

▶ Note. Except when *elitism* is active, no path is kept for more than one generation, but *copies* of it might exist in any generation.

We adopted the so-called (μ, λ) selection: after generating $\lambda > \mu$ parents, we generated λ children, of which we selected μ chromosomes to put in the population pool, discarding the previous generation entirely.
For each offspring, the probability of being dropped without being added to the newest generation is computed using the **weighted random sampling** technique, with longer solutions dropped more probably than shorter ones. Please refer to Section 4.10 for more details.

By default, we also adopted the **elitism** strategy, which is used to prevent the loss of the current fittest member of the population. If none of the offsprings inserted into the population has cost lower than the shortest parent, one of the offsprings is randomly discarded and the best parent is kept in the new generation's population pool.

4.8 Local Improvement

There are three cases in which we manipulate the genetic algorithm's solution pool to try to escape plateaus or local minima:

1. At the beginning of the metaheuristic algorithm, exploring possible better solutions by perturbing the initial solution pool;
2. Repeatedly, after a number of genetic algorithm generations;
3. After the last iteration is performed / when the timeout is over.

Alternating *intensification* and *diversification* throughout the metaheuristic algorithm hopefully gives us better final solutions. These local strategies rely on the concept of *neighborhood* of a permutation path.

4.8.1 Neighborhood of a Permutation Path

For each given candidate permutation path solution P, we determine its neighbours $N(P) = \{n_0, n_1, n_2\}$ as the result of applying the four mutation operators we mentioned in Section 4.3:

(i) Swap Operator;
(ii) Left Rotation Operator;
(iii) Right Rotation Operator;
(iv) Inversion Operator.

Each of these mutation operators require two distinct indexes $i, j \in 0, 1, \ldots, n-1, (i < j)$. The best neighbor is the mutated path with the minimum total distance. Variable Neighborhood Search (VNS) is a metaheuristic where systematic changes of neighborhood both during descent steps and perturbation steps are performed to find local optimum and escape plateaus. We took inspiration from [HM05] and implemented 3 different VNS strategies.

Greedy Bounded Variable Neighborhood Descent

For enhancing the initial generation's pool, we apply a technique called *Bounded Variable Neighborhood Descent*. For each permutation path `path` in the population pool, we:

(a) Select a pair of distinct indexes i, j;
(b) We apply the four mutations to `path` (regardless of p_m) and we select the best neighbor;
(c) If the best neighbor is no better than `path`, we increment a counter, otherwise we reset the counter to 0, assigning the best neighbor to `path`;
(d) The process repeats itself until a predefined counter value is reached (for our experiments, the maximum counter is 10).

This greedy approach is memory efficient, but it does not have memory of the previous solutions in the neighborhood. In addition, it is incredibly fast, because only 4 mutations between 2 indexes are tested for each path.

Complete Sliding Variable Neighborhood Descent

This is a technique we implemented but did not use in the final metaheuristic model. In practice, all pairs of indexes i, j, $i < j$ are selected for mutation, and the best path is repeatedly updated every time a neighbor has a total distance that is lower than the previous `path`'s distance. I.e., the `path` *slides* over the neighborhood while the complete combination of indexes is tested. We discarded this technique because it is too slow in practice.

Random Windowed Variable Neighborhood Descent

Every 20 generations, we perturb the solution pool with a technique we called Random Windowed Variable Neighborhood Descent. The same technique is used for refining the last solution pool. It proceeds as follows:

(a) For each path, a window of indexes is randomly chosen. A window is a range $[lb, ub]$ with the lowest and highest indexes of a path's nodes eligible for mutations.
(b) We then apply the same algorithm as Complete Sliding Variable Neighborhood Descent, but rather than changing all pairs of indexes, we restrict ourselves on indexes such that $lb \leq i < j \leq ub$.

The most important part of this technique is how we generate the initial window. We want the window to be as large as possible when the size of the path is small (to give more possibility of finding the global optimum on small instances) and we want it to decrease exponentially as the size of each path grows, to keep the total computational time approximately invariant to the size of the problem.

We start by defining a logarithm threshold t:

$$t = \log_2(n \cdot \frac{2}{5}),$$

where n is the size of each path. We then define Δ_{min} and Δ_{max}:

$$\Delta_{min} = \lfloor t \rfloor, \qquad \Delta_{max} = \lfloor 3.5 \cdot t \rfloor,$$

where Δ_{min} is the minimum distance between the two extreme points of the window and Δ_{max}, likewise, is the maximum distance. Perhaps surprisingly, we are able to sample each window in $\mathcal{O}(1)$ time.

4.9 Stopping Criterion

The metaheuristic algorithm uses a combination of commonly used techniques to stop the execution of the algorithms:

- Time limit;
- Maximum number of non-improving generations; (`max_gen_no_improvement`) reached;
- Maximum number of generations (`max_gen`) reached.

4.10 Implementation Details

Finding the two farthest nodes in a distance matrix

The initial population generation method that we used starts from the candidate solution path found applying the *Farthest Insertion* strategy. The trivial way of finding the maximum distance and the two corresponding farthest nodes in a distance matrix would require a full scan, so $\mathcal{O}(n)$ time. We have instead exploited the fact that the distance matrix is symmetric: the upper triangle is equal to the transposed of the bottom triangle of the distance matrix, and the main diagonal is the null vector (0^n). We thus created a custom iterator `upper_triangular_iterator` that traverses only the upper triangular part of the matrix, and used it via the `std::max_element` function. The two farthest cities are then found in $\mathcal{O}(\frac{n}{2})$.

▶ Note. If we decided to process the `TSPLIB` complete graph coordinates directly rather than computing a distance matrix first, we could have determined the two farthest cities in a path in $\mathcal{O}(n \cdot logn)$ by computing the convex hull H of the cities first with Chan's method [Cha96], and then apply the $\mathcal{O}(n)$ rotating calipers method [Tou00] for finding the two endpoints of the diameter of H. This would lower the run-time of Farthest Insertion from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2 \cdot logn)$, but since the heuristic solution is computed only once, we decided to use a simpler, less error-prone approach.

Memoization of Permutation Path's cost

Each TSP circuit is modelled with a `std::vector<size_t>` container of size n. For any path p, its *circuit distance* must be computed multiple times throughout a single generation phase. This computation takes time $\mathcal{O}(n)$ and might significantly slow down the run-time performances even with just $n = 100$ cities. We thus embedded the `std::vector<size_t>` container in a `PermutationPath<T>` class, where `T` is the generic type of the distance between any two cities (e.g. `double`). The distance is computed lazily: it is initialized to $+\infty$, and when the distance of a path needs to be computed (via the `cost()` method), the distance is computed and memorized for future reference. Thus, the first `cost()` invocation requires $\mathcal{O}(n)$ time, but all future invocations are just $\mathcal{O}(1)$.

▶ Note. Offsprings' cost must be computed from scratch, thus requiring $\mathcal{O}(n)$ time.

▶ Remark. The circuit distance of a path $[0, 1, \ldots, n-1]$ is defined as $d(0,1) + d(1,2) + \cdots + d(n-2, n-1) + d(n-1, 0)$, where $d(\cdot,\cdot)$ is a 2-dimensional distance measure. In our case, $d(\cdot,\cdot)$ is L_2 approximated to an integer according to `TSPLIB`'s formula.

Efficient sampling of k distinct values

The crossover operators we implemented require to select $k = 2$ or $k = 4$ distinct positions to mutate in a chromosome.
The trivial way of sampling k distinct values from a pool of size n can be problematic because the higher the value of k, the more likely it is that the trivial algorithms degrates to a $\mathcal{O}(n^2)$ run-time. If the random generator is not truly random (such as `rand` in `C`), it may even never terminate when k is very close to n. We decided to implement Robert Floyd's algorithm [BF87], which is surprisingly easy to write, and it takes only $\mathcal{O}(k)$ time.

Weighted random sampling in (μ, λ) selection

Chromosomes with a lower fitness (paths with a higher cost) have a higher probability of being discarded without becoming part of the new generation's pool. The generational (μ, λ) strategy used to choose μ of the $\lambda > \mu$ offsprings for the next generation's pool uses the weighted random sampling with a reservoir presented in [ES06]. It allows to draw a weighted random sample of size μ from a population of λ weighted items, and it is easily adaptable to a streaming setup for a more powerful genetic algorithm.

4.11 Failed Experiments

As we mentioned earlier, we spent a considerable percentage of the total amount of time dedicated to this project experimenting several strategies for the metaheuristic implementation. We remind that we discarded these experiments before the calibration phase, as the total amount of time for testing every parameter combination would probably sum up to months of work.

Alternative Population Initialization

We initially attempted a two-way initialization of the population pool:

1. Initialize the initial solution s using the Farthest Insertion constructive heuristic;
2. Assign random permutations of s to the first $\frac{\mu}{2}$ slots of the population pool vector, including the heuristic solution s itself;
3. Fill the remaining $\frac{\mu}{2}$ slots with random permutations of the sorted path of nodes $[0, 1, \ldots, n-1]$;
4. Shuffle the population pool before returning.

We ended up dropping support for this kind of initialization because we observed solutions too divergent from the optimal value even after hundreds of generations.

4.11.1 Adaptive Parameter Control

In its simplest version, a genetic algorithm is influenced by static parameters that are set once and remain fixed during the run. It is an intrinsically dynamic and stochastic process, so the usage of static parameters is in contrast with the nature of a genetic algorithm.

▷ Claim. Different parameter values might be optimal at different stages of the metaheuristic search process.

For example, larger mutation steps might be better for the first generations to further explore the search space, while smaller mutation steps are probably more suitable for late generations, for fine-tuning candidate solutions.

Crossover Rate

Initially, the crossover rate p_c was 1. However, that parameter value was seriously hindering the performance of the genetic algorithm pipeline, as it did not allow the metaheuristic to determine any solution better than the initial solution computed using Farthest Insertion.

▶ Remark 2. Two parents are selected and two offspring are created via recombination of the two parents with probability p_c, or by simply copying the parents, with probability $1 - p_c$.

We also tried both logarithmically increasing and decreasing rates for p_c. Even though increasing crossover rates gave us better results than $p_c = 1$ for each generation, this approach was both slower and slightly worse than the fixed crossover rate we determined via calibration ($p_c = 0.87548$).

Mutation Rate

Similarly to the crossover rate, we tried decreasing mutation rates. With an exponentially decaying rate p_m from 0.8 to 0.001 we obtained results comparable to the using the fixed mutation rate determined by the calibration step ($p_m = 0.01832$), but we got much worse time performances, so we did not include decaying rates in the final revision of the project.

4.12 Summary of Metaheuristic Model

Now that we explained the most important metaheuristic strategies we implemented, we proceed to summarize the structure of `ex2-metaheuristic`.

- Feasible path solutions are stored using a permutation representation whose distance cost is memoize;
- The initial solution is computed via the Farthest Insertion constructive heuristic;
- The initial population pool of size μ includes the initial heuristic solution and random permutations of it;
- The initial population pool is improved via a local search step, using the *Greedy Bounded Variable Neighborhood Descent* strategy;
- Until the time limit, the maximum number of generations, or the maximum number of generations without improvement are reached, we perform one iteration of the genetic algorithm over the current generation of solutions.
- λ paths are selected for the mating pool using the Tournament Selection strategy.
- Pairs of the mating pool are selected with probability to either generate two new offsprings via the `Order-Alt` crossover operator, or to simply copy themselves. The crossover operators are applied sequentially;
- Paths of the offspring pool are mutated with probability using the Left Rotation operator;

OS	Ubuntu 20.04
Arch	x64
CPU	Intel(R) Core(TM) i7-8550U @1.80GHz
RAM	16GB

Table 1 Specifications of the laptop used for the experiments.

	Constraint	**Bounds**	**Calibrated Value**
p_m	$0 \leq p_m \leq 1$	$[0.01,\ 0.02]$	0.01832
p_c	$0 \leq p_c \leq 1$	$[0.8,\ 1]$	0.87548
μ	$\mu < \lambda$; μ even	$[26,\ 50]$	40
λ	μ odd	$[30,\ 74]$	60
k	$k < \lambda$	$[10,\ 14]$	13
`max_gen_no_improvement`	`max_gen_no_improvement` $\leq$ `max_gen`	$[50,\ 200]$	177
`max_gen`	-	$[100,\ 500]$	408

Table 2 Summary of the tuning parameters for the Metaheuristic Algorithm. p_m is the mutation rate. p_c is the crossover rate. μ is the population size. λ is the size of the offspring pool. k is the tournament size for the parent selection strategy. The bounds were initially looser, but we iteratively tightened them during the development.

- The new generation of μ paths is choosen from the λ elements of the offspring pool using the (μ, λ) selection strategy with elitism;
- The number of generations is incremented;
- After 20 iterations, before updating the new best solution, the Random Windowed Variable Neighborhood Descent local search strategy is run over the population pool;
- After the timeout is expired, another quick run of the Random Windowed Variable Neighborhood Descent local search strategy is performed on the last population pool;
- The best solution found among all generations is returned.

5 Random Baseline

We want to make sure that both `ex1-cplex` and `ex2-metaheuristic` are much better than a random TSP solution, i.e. we want their solutions to be meaningful. Thus, we implemented a completely random TSP solver (which can be found in the `random-baseline` folder) whose behavior is the following:

1. Select $[0, 1, \ldots, n-1]$ as the initial path to optimize;
2. Until the time limit expires, shuffle the path and overwrite it only when a new, better solution is found;
3. Return the best permutation obtained.

6 Calibration

The purpose of the calibration step is identifying the best hyperparameter values of the metaheuristic algorithm. The calibration is itself an optimization problem: we try to discover the combination of parameters of `ex2-metaheuristic` that returns both small and stable solutions on sample datasets. More formally, we use a black-box linear multi-optimization algorithm provided by `Pymoo` to execute the Genetic Algorithm on all simulated datasets of family `A`, multiple times and with a fixed timeout, aggregating the solution results and minimizing both the average and the mean standard deviation of the solutions.

We tried to keep the number of parameters to calibrate sufficiently low. We only performed the calibration step after fixing the mutation and crossover operators, as well as other strategies specified in Section 4.12.

▶ Remark. The higher the number of parameters to tune, the more complex the calibration process is, and good parameter combinations are less likely to be found.

The calibration happens in two phases. Table 2 shows the tuned hyperparameters, their range of selection, and the value selected by the calibration.

Phase 1

In 25 iterations, each of which consisting of 30 executions with a timeout of 2000 milliseconds (2 seconds), the following 5 variables are tuned:

- **Mutation probability** (p_m);
- **Crossover rate** (p_c);
- **Size of the population pool** (μ);
- **Size of the offspring pool before pruning** (λ);
- **The size of the tournament** (k).

The maximum number of generations `max_gen_no_improvement` and the maximum total amount of generations `max_gen` are initially fixed to 200 and 500, respectively.

▶ Note. Since we require μ and λ to be both even numbers, the simplest way we found to express this constraint is simply asking `Pymoo` to find the optimal values of $\frac{\mu}{2}$ and $\frac{\lambda}{2}$, and then multiply them by 2.

Phase 2

We fix every hyperparameter value determined by the first phase and we look for the best values of `max_gen_no_improvement` and `max_gen`. We run 50 executions for an unbounded number of iterations, with an execution timeout of 60000 milliseconds (1 minute), stopping the second calibration phase after 6 hours. The final obtained values are then saved to a `CSV` file for future reference. The `CSV` file is then parsed by the `./scripts/ex2-metaheuristic.sh` script defined in Appendix E, so that the user is always able to use the most recent calibrated model values without compiling the source code of `ex2-metaheuristic` again.

▶ Note. Even though `Pymoo` offers the opportunity to leverage the known optimal solutions of problems for determining the optimal Pareto front and set, we decided not to use this feature. In a real-world context, we probably have at disposal plenty of examples of TSP instances, but we would not likely known their optimal solutions.

7 Benchmark

The benchmark phase assesses the runtime performance of the algorithms we implemented and the deviation of their solutions from the known optimum value. We decided to run every algorithm 11 times on the same instance, for all 25 TSP instances dedicated to the benchmark phase (the 5 `TSPLIB` datasets and the 20 simulated semi-regular datasets of family `B`). Running the same algorithm several times allows us to gather an estimate of the generated solutions even though the method uses randomized components. We used the same approach for the `MILP` implementation in `CPLEX` because the parallelism strategy we used is not deterministic, and because, when the timeout is too tight, that method is not able to provide exact solutions.

Each experiment is repeated with four different timeout values, expressed in milliseconds: 100, 1000, 10000, 60000. Since we performed this experiments on three programs (`ex1-cplex`, `ex2-metaheuristic`, and `random-baseline`), this gives us an estimate of

$$11 \cdot 25 \cdot (100 + 1000 + 10000 + 60000) \cdot 3 = 58657500\text{ms},$$

i.e. about 16 hours to complete the benchmark. The logic of the benchmark phase is delegated to the `benchmark` `Python` module.

7.1 Solution Gap

The *solution gap* is a measure of the divergence of the solution obtained by an algorithm with respect to a known optimum o_s. Given a TSP dataset and a TSP solver, we first compute the mean solution m_s over 11 repeated run. We then define the solution gap as the relative error:

$$100 \cdot \frac{|m_s - o_s|}{o_s}$$

8 Results

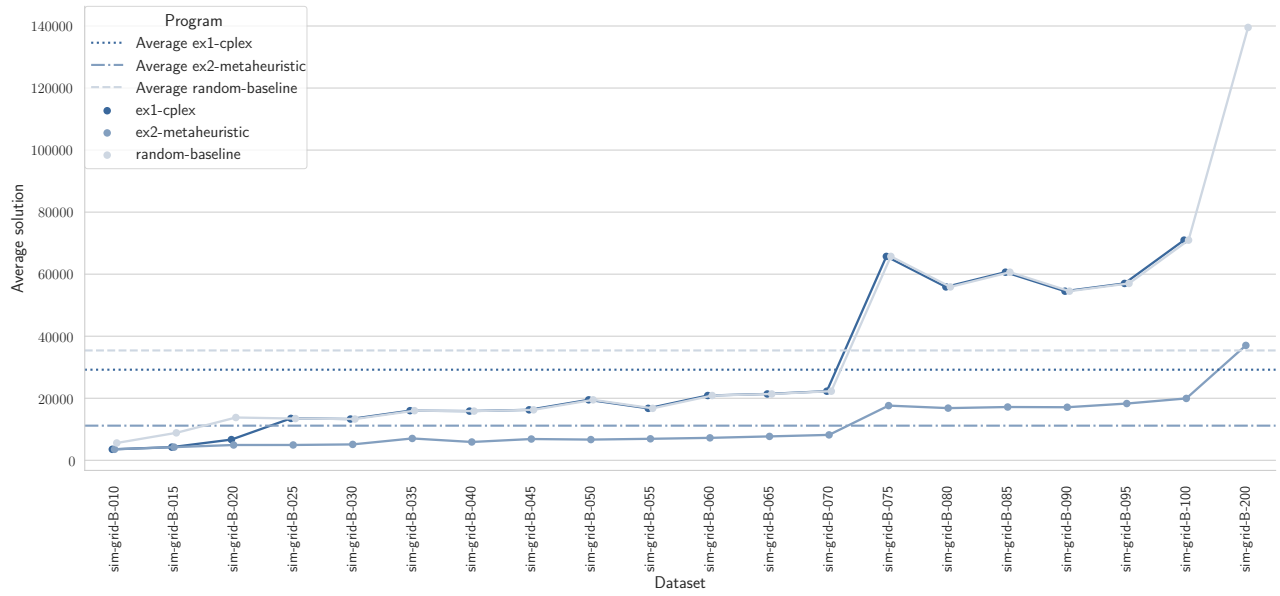
The MILP implementation provided by `ex1-cplex` is definitely the most stable and reliable solver for relatively small instances, if waiting some seconds before getting an answer is acceptable. On simulated datasets, `CPLEX` is able to find the exact solution in less than 10 seconds on instances with up to 70 nodes, but the solution gap increases enourmously after that point, culminating in the same error as the random baseline on the `sim-grid-B-200` dataset even with a timeout of 1 minute, with analogous errors for all real-world datasets except the smallest `d198` for the same 1 minute timeout. In many cases, the MILP model is not even able to find a feasible solution, as it happens with `d1291` with a timeout of 10000ms.

The solutions provided by `CPLEX` take a variable amount of time that tends to grow exponentially as the size of the problem grows, but that is difficult to foresee precisely. For instance, given a timeout of 1 minute, on average, `ex1-cplex` takes 15 seconds to solve `sim-grid-B-085`, but it takes 40 seconds to solve the smaller `sim-grid-B-080` instance.

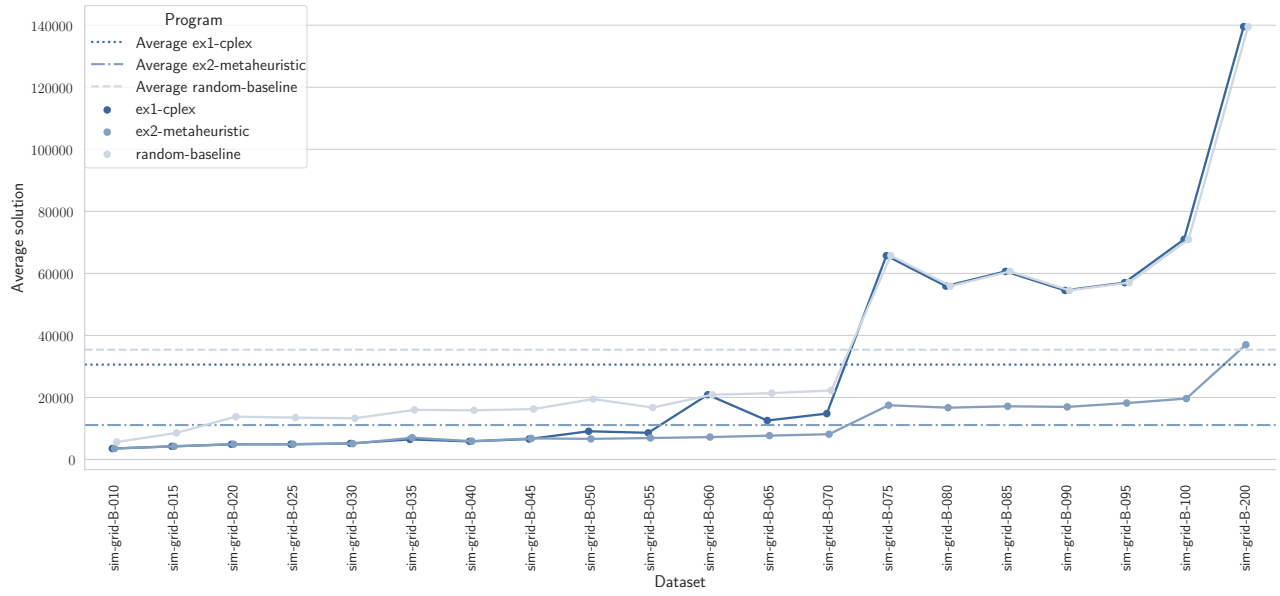
The metaheuristic implementation, even though it suffers of a certain instability, is consistently the best method to use with bigger instances. In some cases, it can even reach very good solutions in just 100ms, as it happens with `sim-grid-B-060` (with an average gap of 0.56 versus the 190.4 gap of `ex1-cplex`) or with `d198`, with an average gap of 2.97 (with `CPLEX` not even able to compute a feasible solution).

The multiple experiments we conducted seem to reveal that the improvement of the first solution of `ex2-metaheuristic` is useless in most of the cases, as it does not find better solutions that the one provided by the Farthest Insertion constructive heuristic. This may be due either to the excessive simplicity of the local optimization strategy (Greedy Bounded Variable Neighborhood Descent), which is too greedy, or simply because Farthest Insertion is particularly suitable for the symmetric TSP in board-drilling contexts.

`ex2-metaheuristic` is extremely fast. With a timeout of 10 seconds, all simulated datasets are solved without interruptions, i.e. the algorithm stops either because of stalling solutions for an excessive amount of generations with no improvement, or because the maximum total limit of generations is reached. And even on the interrupted real-world datasets, the maximum average gap is less than 22 units. `ex1-cplex` tends to have much higher average solution gaps on interrupted instances, akin to `random-baseline`.



(a) Mean solutions for the simulated datasets of family `B` with timeout = 100ms.

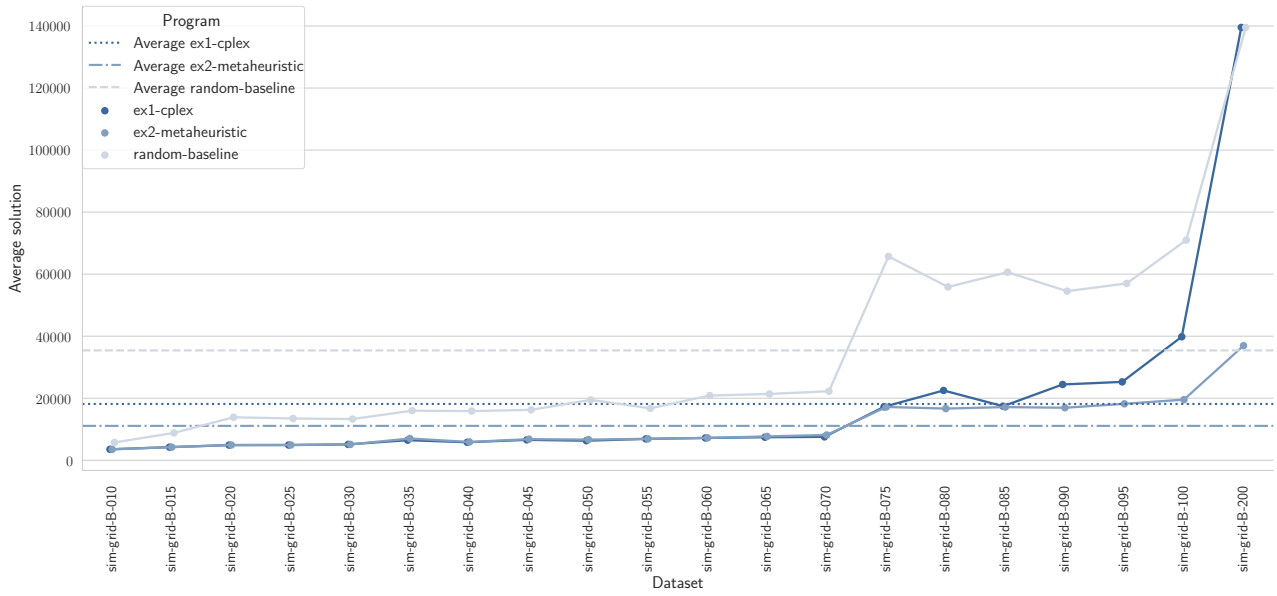


(b) Mean solutions for the simulated datasets of family `B` with timeout = 1000ms.

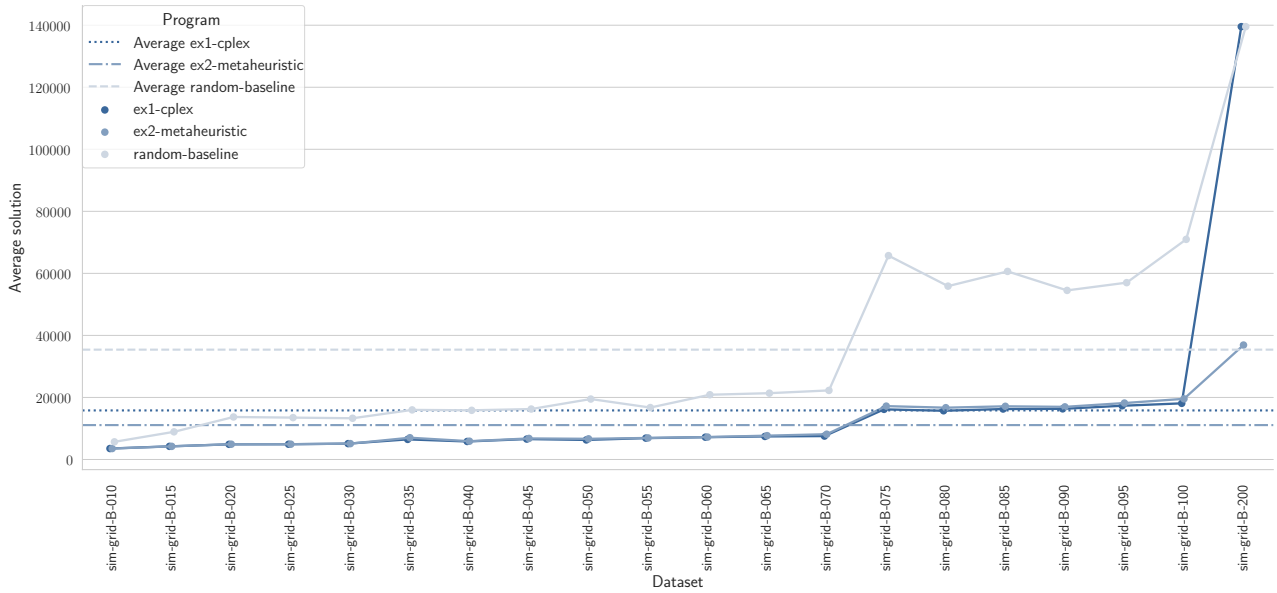
Figure 6 Mean solutions for the simulated datasets of family `B` with timeout set to 100ms and 1000ms. The dotted lines indicate the global average solutions among all simulated datasets. Clearly, `ex2-metaheuristic` is much better than `ex1-cplex` on average in these cases, whereas `ex1-cplex` agrees with the solutions provided by `random-baseline` for $|N| > 70$.

9 Conclusions

In this report we examined and compared two different implementations for the Traveling Salesman Problem in the symmetric and complete-graph case: one exact implementation based on a Integer Linear Programming model, and one based on Genetic Algorithms with Local Search strategies. We



(a) Mean solutions for the simulated datasets of family `B` with timeout = 10000ms.



(b) Mean solutions for the simulated datasets of family `B` with timeout = 60000ms.

Figure 7 Mean solutions for the simulated datasets of family `B` with timeout set to 10000ms and 60000ms. The dotted lines indicate the global average solutions among all simulated datasets. Again, `ex2-metaheuristic` is better than `ex1-cplex` on average in these cases, but not as much as it was for lower timeouts.

managed to achieve solutions of significant qualities with the metaheuristic approach, even for large problem instances. We relied on some datasets provided by the TSPLIB library (with known optimum solutions), and we even created some pseudo-random instances that could be more realistic for the board drilling context. Moreover, we gained experience in the difficulties of calibrating a complex algorithm like `ex2-metaheuristic`.

In the board drilling context, if the size of the instances isn't excessive (i.e. $|N| < 200$) and the same drilling pattern must be applied to several plate batches, it might be worth it to let `CPLEX` run for some hours, obtain the exact TSP solution and then use that solution for the future drilling of those batches. Otherwise, if the problem instance is massive or the number of drilling patterns is considerable, the genetic algorithm can be a good choice.

The source code for this project is hosted in a Github repository:

https://github.com/jkomyno/combinatorial-optimization-tsp

9.1 Questions & Answers

We now proceed to reply to the questions we addressed in Section 1.

- Which implementation is better for a particular problem size?

It mainly depends by the time at disposal. When the problem size is low (i.e. $|N| \leq 70$), `CPLEX` always finds the optimal solution in less than a minute, thus obtaining better quality than the metaheuristic implementation. However, if a solution is needed in less than a second, and the test instances have more than 30 nodes, `ex2-metaheuristic` should be preferred. For medium to big instances (i.e. $|N| \geq 200$), `ex1-cplex` should be avoided because it becomes as bad as a random guesser. As expected, the exact algorithm follows an exponential time trend. On the other hand, the genetic algorithm run-time depends on the given maximum allotted time limit, but tends to increase as the size of the TSP dataset it is tested on grows.

- What is the gap with respect to the optimal solution?

Section 7.1 gives the definition of *solution gap*, and tables in Appendix B summarize the solution gap for every relevant experiment we ran.

- How well does the metaheuristic implementation escape local minima to reach the optimal solution?

In general, the initial heuristic solution is almost always the worse than the final solution returned by `ex2-metaheuristic`. However, we noticed that the genetic algorithm tends to stall too often on local

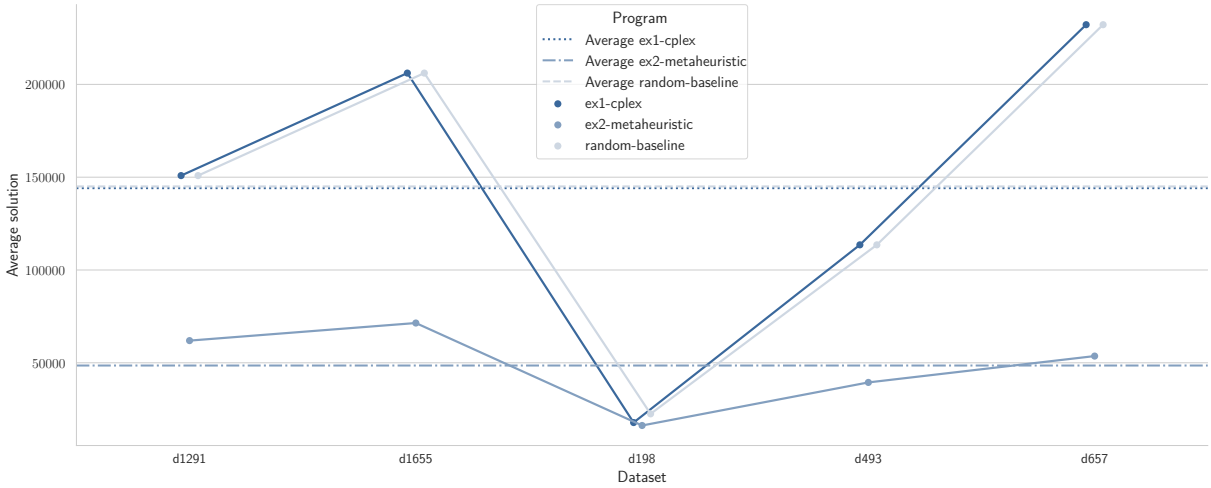
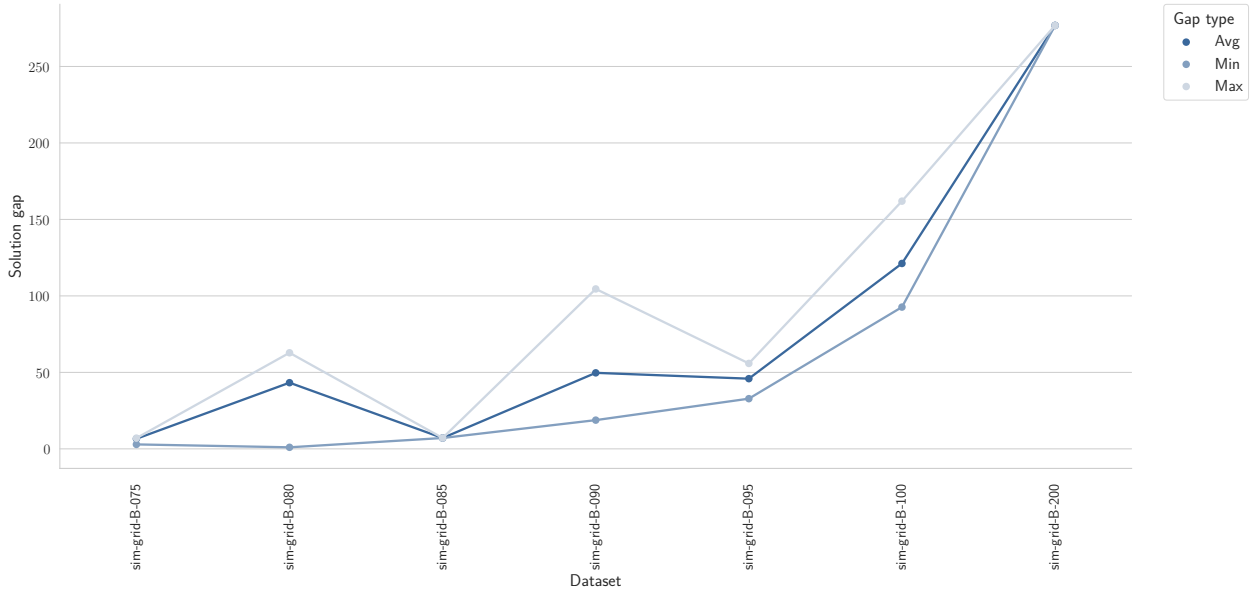
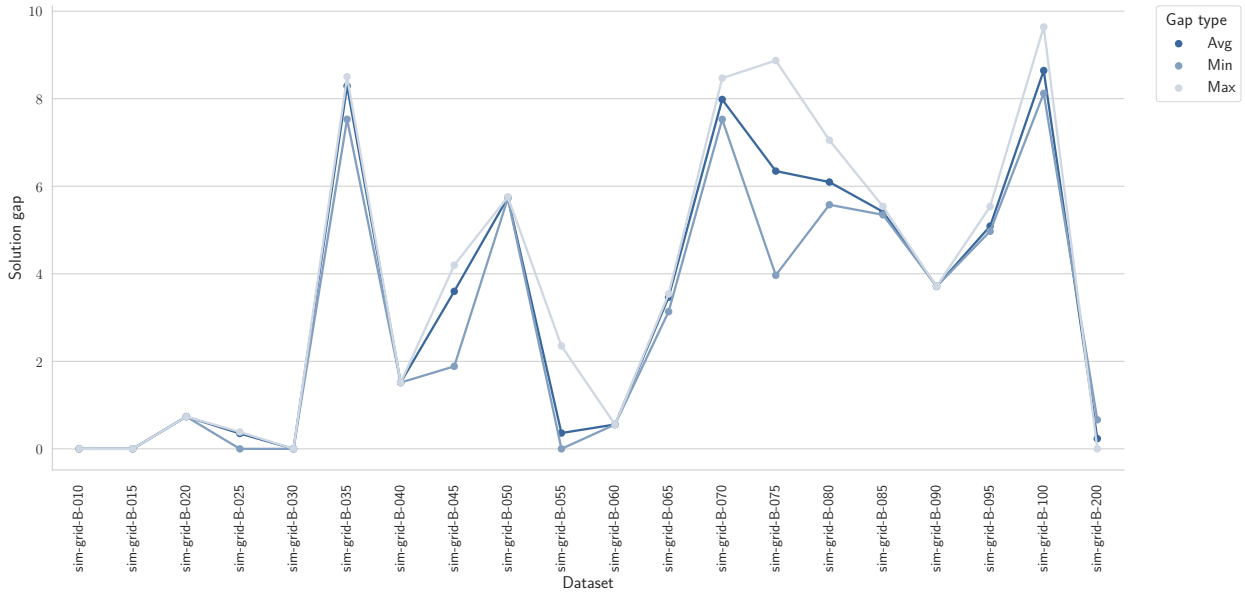


Figure 8 Mean solutions for the board-drilling `TSPLIB` datasets with timeout = 60000ms. We can notice how, on instances with more than 200 nodes, the solutions provided by the random baseline are as good as the one computed by CPLEX in the MILP model.



(a) Solution gaps of the `ex1-cplex` solver on interrupted instances of simulated datasets of family `B`, with timeout = 10000ms.



(b) Solution gaps of the `ex2-metaheuristic` solver on non-interrupted instances of simulated datasets of family `B`, with timeout = 10000ms.

Figure 9 Solution gaps of the `ex1-cplex` and `ex2-metaheuristic` solvers on simulated datasets of family `B`, with timeout = 10000ms. For `ex1-cplex`, only interrupted simulated instances are considered. For `ex2-metaheuristic`, only non-interrupted simulated instances are considered (there are no interrupted simulated instances). We can notice that the MILP implementation consistently returns the exact solutions for the smallest simulated datasets, as expected. From $|N| > 75$, however, the solution gap increases almost exponentially. The Genetic Algorithm, on the other hand, produces less stable solutions that are slightly worse than the optimal ones, but is consistently has a much tighter gap with respect to the known solution. For $|N| = 200$, the metaheuristic approach even manages to find the global optimum.

minima, interrupting its execution before the timeout because the maximum number of generations without solution improvement has been reached. Improving the initial solution pool is almost always useless, but the repeated Random Windowed Variable Neighborhood Descent strategy seems to be more than effective, and also scalable to larger instances.

The metaheuristic implementation is very flexible, because it virtually allows to generate a solution for any instance size, provided that it is given enough time to compute at least the first heuristic solution. It will generally find good - but imperfect - solutions that might be sufficient in some realistic applications.

- Are the time-limited `CPLEX` and metaheuristic approaches better than a simple time-limited random search of the TSP solution?

Yes, they are generally better, especially `ex2-metaheuristic`. However, the solutions provided by `ex1-cplex` are no longer meaningful when considering medium and large problem instances, even with timeouts up to some minutes.

9.2 Possible Improvements

In the work we just presented, we mainly aimed to explore metaheuristic strategies, in particular the combination of genetic algorithms and local search algorithms, in a *depth-first* fashion, exploring even strategies we did not include in our final model. This leaves room to some possible improvements to the `ex2-metaheuristic` implementation, of which we list a selection that is relevant to the topics of the course:

1. Currently, after applying a mutation operator to a permutation path, we recompute the total path cost from scratch. There are strategies for reducing the time needed to compute the updated cost that involve a constant number of sums and differences.
2. The timeout limit is not very precise. Currently, we check whether the time limit is expired once per generation. This creates some bias where experiments on bigger instances are executed for more time than they should be allowed to. We should have polled the timeout status multiple times for each generation. However, our timeout implementation seems to more precise than the one provided by `CPLEX`.
3. We could have implemented multiple restarts. We avoided this because it would have introduced too much complexity, especially for the calibration phase.
4. We could have chosen a different initial heuristic. Currently, Farthest Insertion generates solutions that in many cases are exceptionally close to the final solution of the metaheuristic algorithm. Maybe, a less optimized constructive heuristic like Nearest Neighbor could have limited the times in which the algorithm gets stuck in a plateau.
5. In our implementation, we do not have any concept of memory of visited solutions. We are confident that if there was an easy way to tell two different solutions (implemented as vectors) apart, we could exploit this fact diversify the explored solutions even more, possibly escaping local minima more easily. Hash maps do not work well with vectors of numbers, especially with potentially big ones (the biggest dataset we considered has 1655 nodes). Perhaps, a Trie data structure could be used instead.
6. We could have run multiple independent instances of `ex2-metaheuristic` in parallel, with the same timeout, and return the best solution obtained. While this is very easy to implement in a high-level language, we feel that the equivalent code in `C++17` would have been too obscure and error-prone.

9.3 A Note on Enterprise Costs

A major drawback of the exact implementation is that the exact implementation relies on `IBM CPLEX`, which costs about 199$ per authorized user per month, plus taxes[8]. For those companies who do not wish to depend on recurring subscription costs, our personal suggestions is using Google's Glop linear solver[9], which is available in multiple programming languages and available in the OR-tools software suite for optimization[10].

The metaheuristic algorithm, on the other hand, does not depend on any third-party dependency, and would then be a much cheaper option.

References

Bak87 J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *ICGA*, 1987.

BD20 J. Blank and K. Deb. Pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

Ben75 Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

BF87 Jon Bentley and Bob Floyd. Programming pearls. *Communications of the ACM*, 30(9):754–757, September 1987.

Cha96 T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(4):361–368, April 1996.

CRH20 K. Jarrod Millman *et al* Charles R. Harris. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

Dav91 L Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

ES06 Pavlos S. Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, March 2006.

ES15 A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer Berlin Heidelberg, 2015.

GG78 B. Gavish and S. Graves. The travelling salesman problem and related problems. volume GR-078-78. Massachusetts Institute of Technology, 1978.

HM05 Pierre Hansen and Nenad Mladenović. *Variable Neighborhood Search*, pages 211–238. Springer US, Boston, MA, 2005.

Hol92 John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA., 1992.

LK73 S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, April 1973.

pdt20 The pandas development team. pandas-dev/pandas: Pandas, February 2020.

PG07 Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

Rei91 Gerhard Reinelt. TSPLIB—a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, November 1991.

Tou00 Godfried Toussaint. Solving geometric problems with the rotating calipers. *In Proceedings of IEEE MELECON'83*, 83, 02 2000.

Wtsdt20 Michael Waskom and the seaborn development team. mwaskom/seaborn, September 2020.

[8] https://www.ibm.com/products/ilog-cplex-optimization-studio/pricing
[9] https://developers.google.com/optimization/lp/glop
[10] https://developers.google.com/optimization

A Tables: Datasets

Dataset	N	Optimal solution	Purpose
`d198`	198	15780	Benchmark
`d493`	493	35002	Benchmark
`d657`	657	48912	Benchmark
`d1291`	1291	50801	Benchmark
`d1655`	d1655	62128	Benchmark

Table 3 Summary of the TSPLIB datasets, their size, and known optimal solution. Every one of these datasets is just used to benchmark the performances of our algorithms.

Dataset	N	Optimal solution	Purpose
`sim-grid-A-010`	10	3876	Calibration
`sim-grid-A-015`	15	3764	Calibration
`sim-grid-A-020`	20	5374	Calibration
`sim-grid-A-025`	25	5317	Calibration
`sim-grid-A-030`	30	5459	Calibration
`sim-grid-A-035`	35	5845	Calibration
`sim-grid-A-040`	40	6098	Calibration
`sim-grid-A-045`	45	5833	Calibration
`sim-grid-A-050`	50	5959	Calibration
`sim-grid-A-055`	55	6561	Calibration
`sim-grid-A-060`	60	7220	Calibration
`sim-grid-A-065`	65	7821	Calibration
`sim-grid-A-070`	70	7623	Calibration
`sim-grid-A-075`	75	15235	Calibration
`sim-grid-A-080`	80	16966	Calibration
`sim-grid-A-085`	85	16587	Calibration
`sim-grid-A-090`	90	16681	Calibration
`sim-grid-A-095`	95	17625	Calibration
`sim-grid-A-100`	100	18109	Calibration
`sim-grid-A-200`	200	38488	Calibration

Table 4 Summary of the simulated semi-regular datasets of family `A`, their size, and known optimal solution. Every one of these datasets is just used to calibrate the hyperparameters of our metaheuristic implementation.

Dataset	N	Optimal solution	Purpose
sim-grid-B-010	10	3533	Benchmark
sim-grid-B-015	15	4235	Benchmark
sim-grid-B-020	20	4888	Benchmark
sim-grid-B-025	25	4916	Benchmark
sim-grid-B-030	30	5115	Benchmark
sim-grid-B-035	35	6481	Benchmark
sim-grid-B-040	40	5808	Benchmark
sim-grid-B-045	45	6579	Benchmark
sim-grid-B-050	50	6307	Benchmark
sim-grid-B-055	55	6888	Benchmark
sim-grid-B-060	60	7188	Benchmark
sim-grid-B-065	65	7433	Benchmark
sim-grid-B-070	70	7545	Benchmark
sim-grid-B-075	75	16163	Benchmark
sim-grid-B-080	80	15709	Benchmark
sim-grid-B-085	85	16270	Benchmark
sim-grid-B-090	90	16326	Benchmark
sim-grid-B-095	95	17321	Benchmark
sim-grid-B-100	100	18001	Benchmark
sim-grid-B-200	200	37030	Benchmark

Table 5 Summary of the simulated semi-regular datasets of family 'B', their size, and known optimal solution. Every one of these datasets is just used to benchmark the performances of our algorithms.

B Tables: Solution Gaps

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-010	57.55	3.53	3533	0	0
sim-grid-B-015	93	1.41	4235	0	0

Table 6 `ex1-cplex`: Statistics and solution gaps on non-interrupted TSP instances with timeout = 100ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-020	111.27	0.79	6667	0	36.40
sim-grid-B-025	112.64	5.64	13500	0	174.61
sim-grid-B-030	111.55	2.84	13294	0	159.90
sim-grid-B-035	126.64	4.72	15989	0	146.71
sim-grid-B-040	125.09	1.58	15852	0	172.93
sim-grid-B-045	130.36	15.60	16265	0	147.23
sim-grid-B-050	124.91	1.92	19486	0	208.96
sim-grid-B-055	125.64	1.36	16744	0	143.09
sim-grid-B-060	128.18	7.28	20874	0	190.40
sim-grid-B-065	138.27	3.04	21379	0	187.62
sim-grid-B-070	143.55	5.75	22268	0	195.14
sim-grid-B-075	153.27	5.20	65721	0	306.61
sim-grid-B-080	148.00	9.01	55887	0	255.76
sim-grid-B-085	156.55	5.61	60635	0	272.68
sim-grid-B-090	174.82	3.19	54511	0	233.89
sim-grid-B-095	162.73	17.38	57009	0	229.13
sim-grid-B-100	167.73	1.85	70965	0	294.23
sim-grid-B-200	344.73	7.14	-	-	-
d198	353.73	26.20	-	-	-
d493	1732.64	107.41	-	-	-
d657	3119.09	316.99	-	-	-
d1291	12506.36	274.02	-	-	-
d1655	21763.55	2082.50	-	-	-

Table 7 ex1-cplex: Statistics and solution gaps on interrupted TSP instances with timeout = 100ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-010	56.36	1.36	3533	0	0
sim-grid-B-015	92.91	0.94	4235	0	0
sim-grid-B-020	206.64	1.43	4888	0	0
sim-grid-B-025	219.27	2.10	4916	0	0
sim-grid-B-035	393.82	3.37	6481	0	0

Table 8 ex1-cplex: Statistics and solution gaps on non-interrupted TSP instances with timeout = 1000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-030	860.00	64.37	5115	0	0
sim-grid-B-030	1025.67	4.16	5344.33	166.08	4.48
sim-grid-B-040	1028.27	4.73	5835.64	61.49	0.48
sim-grid-B-045	962.00	16.97	6579	0	0
sim-grid-B-045	1027.33	8.11	6595.67	36.74	0.25
sim-grid-B-050	1037.91	7.31	9096.82	324.43	44.23
sim-grid-B-055	1046.73	4.20	8603.91	507.75	24.91
sim-grid-B-060	1045.91	1.45	20874	0	190.40
sim-grid-B-065	1055.55	6.36	12568	0	69.08
sim-grid-B-070	1060.64	0.92	14789.82	2480.23	96.02
sim-grid-B-075	1064.00	8.05	65721	0	306.61
sim-grid-B-080	1063.45	6.30	55887	0	255.76
sim-grid-B-085	1078.00	6.72	60635	0	272.68
sim-grid-B-090	1081.27	8.03	54511	0	233.89
sim-grid-B-095	1088.55	4.55	57009	0	229.13
sim-grid-B-100	1089.45	0.52	70965	0	294.23
sim-grid-B-200	1306.82	25.83	139539	0	276.83
d198	1296.00	24.27	22498.00	0.00	42.57
d493	2924.45	131.71	-	-	-
d657	4059.00	191.40	-	-	-
d1291	13524.00	38.94	-	-	-
d1655	21803.18	1371.90	-	-	-

Table 9 `ex1-cplex`: Statistics and solution gaps on interrupted TSP instances with timeout = 1000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-010	56.55	1.13	3533	0	0
sim-grid-B-015	92.55	1.04	4235	0	0
sim-grid-B-020	206.18	1.08	4888	0	0
sim-grid-B-025	219.55	1.75	4916	0	0
sim-grid-B-030	950.91	169.45	5115	0	0
sim-grid-B-035	408	25.31	6481	0	0
sim-grid-B-040	1310	161.54	5808	0	0
sim-grid-B-045	1066.73	132.96	6579	0	0
sim-grid-B-050	2963.45	239.23	6307	0	0
sim-grid-B-055	6602.82	650.35	6888	0	0
sim-grid-B-060	4136.64	498.94	7188	0	0
sim-grid-B-065	5641.64	114.18	7433	0	0
sim-grid-B-070	2058	269.39	7545	0	0

Table 10 `ex1-cplex`: Statistics and solution gaps on non-interrupted TSP instances with timeout = 10000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-075	10060.91	6.41	17230	198.91	6.6
sim-grid-B-080	10072.18	3.31	22505.64	4373.23	43.27
sim-grid-B-085	10076.73	6.75	17425	0	7.10
sim-grid-B-090	10090.45	5.77	24440	6318.95	49.7
sim-grid-B-095	10092.82	8.32	25270.09	1982.97	45.89
sim-grid-B-100	10105.73	9.14	39819.73	4977.50	121.21
sim-grid-B-200	10293.64	25.65	139539	0	276.83
d198	10281.09	21.52	19461	0	23.33
d493	11822.36	27.45	113549	0	224.41
d657	13077.91	56.95	232159	0	374.65
d1291	23901.64	85.19	-	-	-
d1655	30999.09	61.18	-	-	-

Table 11 `ex1-cplex`: Statistics and solution gaps on interrupted TSP instances with timeout = 10000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-010	56.45	0.93	3533	0	0
sim-grid-B-015	92.09	0.94	4235	0	0
sim-grid-B-020	206.82	0.98	4888	0	0
sim-grid-B-025	218.55	1.51	4916	0	0
sim-grid-B-030	894.45	87.74	5115	0	0
sim-grid-B-035	394.91	2.63	6481	0	0
sim-grid-B-040	1228.27	82.85	5808	0	0
sim-grid-B-045	1023.55	58.04	6579	0	0
sim-grid-B-050	2953.09	333.04	6307	0	0
sim-grid-B-055	6759.27	659.74	6888	0	0
sim-grid-B-060	4074.73	582.01	7188	0	0
sim-grid-B-065	5633.18	151.84	7433	0	0
sim-grid-B-070	1937.55	89.83	7545	0	0
sim-grid-B-075	27189.09	4676.21	16163	0	0
sim-grid-B-080	39651.09	8108.04	15709	0	0
sim-grid-B-085	15501.73	823.92	16270	0	0
sim-grid-B-090	29080.27	2709.84	16326	0	0

Table 12 `ex1-cplex`: Statistics and solution gaps on non-interrupted TSP instances with timeout = 60000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-095	50271.86	4494.67	17321	0	0
sim-grid-B-095	60098.75	2.5	17326	10.00	0.03
sim-grid-B-100	50110	4640.79	18001	0	0
sim-grid-B-100	60088.8	33.55	18212.8	250.46	1.18
sim-grid-B-200	60289.36	23.88	139539	0	276.83
d198	60355.18	4.87	17802.82	13.58	12.82
d493	61749.73	6.08	113549	0	224.41
d657	63472.36	199.07	232159	0	374.65
d1291	80819.82	912.35	150852	0	196.95
d1655	87934.18	1984.21	206087	0	231.71

Table 13 ex1-cplex: Statistics and solution gaps on interrupted TSP instances with timeout = 60000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-010	59.73	0.79	3533	0	0

Table 14 `ex2-metaheuristic`: Statistics and solution gaps on non-interrupted TSP instances with timeout = 100ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-015	96	3	4235	0	0
sim-grid-B-015	101.5	0.53	4235	0	0
sim-grid-B-020	102	0.77	4924	0	0.74
sim-grid-B-025	102.91	0.83	4931.55	7.69	0.32
sim-grid-B-030	103	0.77	5115	0	0
sim-grid-B-035	103.45	0.93	7034.36	16.95	8.54
sim-grid-B-040	104.09	0.83	5898.18	4.85	1.55
sim-grid-B-045	103.73	0.65	6851.36	12.06	4.14
sim-grid-B-050	104.82	0.87	6669.45	0.93	5.75
sim-grid-B-055	105.00	1.18	6923.27	78.73	0.51
sim-grid-B-060	105.36	1.29	7228	0	0.56
sim-grid-B-065	106.82	1.08	7696	0	3.54
sim-grid-B-070	106.27	1.01	8183.09	19.38	8.46
sim-grid-B-075	107.09	1.45	17622.09	87.41	9.03
sim-grid-B-080	108	1.26	16814.18	26.16	7.04
sim-grid-B-085	108.45	1.21	17171.00	0	5.54
sim-grid-B-090	108.91	1.45	17096.82	81.5	4.72
sim-grid-B-095	110.36	1.43	18289.09	29.73	5.59
sim-grid-B-100	110.64	1.63	19936.91	20.2	10.75
sim-grid-B-200	132.82	6.10	37030	0	0
d198	129.36	3.35	16248.27	9.57	2.97
d493	323.82	3.68	39481.91	8.53	12.8
d657	539.18	9.66	53805.45	36.69	10
d1291	3625.82	88.3	61958.82	0.6	21.96
d1655	10096.18	234.84	71604.91	5.94	15.25

Table 15 `ex2-metaheuristic`: Statistics and solution gaps on interrupted TSP instances with timeout = 100ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-010	61.82	8.05	3533	0	0
sim-grid-B-015	117.09	24.67	4235	0	0
sim-grid-B-020	137.27	1.10	4924	0	0.74
sim-grid-B-025	188.09	33.09	4931.55	7.69	0.32
sim-grid-B-030	217.09	3.30	5115	0	0
sim-grid-B-035	452.73	90.87	7020.45	1.21	8.32
sim-grid-B-040	399.64	92.75	5897.09	3.62	1.53
sim-grid-B-045	553.18	161.19	6794.91	53.02	3.28
sim-grid-B-050	583.64	123.03	6647.09	47.52	5.39
sim-grid-B-055	533.82	91.85	6917.45	65.53	0.43
sim-grid-B-060	523.64	5.46	7228	0	0.56
sim-grid-B-065	558.45	4.11	7696	0	3.54

Table 16 `ex2-metaheuristic`: Statistics and solution gaps on non-interrupted TSP instances with timeout = 1000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-070	719.78	108.84	8151	15	8.03
sim-grid-B-070	1007.5	0.71	8148	2.83	7.99
sim-grid-B-075	740.5	92.63	17629.5	55.86	9.07
sim-grid-B-075	1007.22	1.39	17458.56	130.91	8.02
sim-grid-B-080	814.33	63.04	16775	36.59	6.79
sim-grid-B-080	1008.88	1.64	16678	39.05	6.17
sim-grid-B-085	813.5	79.69	17152.4	16.01	5.42
sim-grid-B-085	1012	-	17140	-	5.35
sim-grid-B-090	928.4	47.19	16972.2	89.89	3.96
sim-grid-B-090	1009.83	0.75	16932	0	3.71
sim-grid-B-095	1010.64	1.75	18190.18	7.83	5.02
sim-grid-B-100	1010.55	1.92	19640	118.55	9.11
sim-grid-B-200	1033.27	4.2	36990.73	69.12	0.11
d198	1031.55	4.95	16232	34.58	2.86
d493	1198.73	9.83	39439.45	18.3	12.68
d657	1437.91	27.02	53739.73	38.72	9.87
d1291	4458.18	117.79	61958	1.61	21.96
d1655	10776.45	204.06	71594.73	22.31	15.24

Table 17 `ex2-metaheuristic`: Statistics and solution gaps on interrupted TSP instances with timeout = 1000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-010	58.91	0.83	3533	0	0
sim-grid-B-015	112.27	13.83	4235	0	0
sim-grid-B-020	139.45	2.02	4924	0	0.74
sim-grid-B-025	181.55	18.8	4933.27	5.73	0.35
sim-grid-B-030	216	1.67	5115	0	0
sim-grid-B-035	392.82	108.04	7018.27	16.92	8.29
sim-grid-B-040	398.09	41.83	5896	0	1.52
sim-grid-B-045	441.27	136.27	6815.73	41.64	3.6
sim-grid-B-050	534.27	150	6669.09	1.04	5.74
sim-grid-B-055	570.82	121.11	6912.91	56.54	0.36
sim-grid-B-060	525.36	5.14	7228	0.00	0.56
sim-grid-B-065	568.55	27.06	7690.55	12.14	3.46
sim-grid-B-070	825.55	220.15	8147.27	16.11	7.98
sim-grid-B-075	1403.64	259.48	17188.73	198.12	6.35
sim-grid-B-080	1488.91	292.66	16666.64	66.57	6.1
sim-grid-B-085	947.55	244.57	17151.27	15.64	5.42
sim-grid-B-090	1029	119.36	16932	0	3.71
sim-grid-B-095	1584.27	432.24	18202.36	31.78	5.09
sim-grid-B-100	1844.55	305.33	19556.82	96.63	8.64
sim-grid-B-200	2834.27	734.19	36943.45	102.69	0.23
d198	4048.91	1205.97	16196.64	67.66	2.64

Table 18 `ex2-metaheuristic`: Statistics and solution gaps on non-interrupted TSP instances with timeout = 10000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
d493	10203.25	9.95	39423	0	12.63
d657	10431.73	15.7	53626.91	17.21	9.64
d1291	13394.27	83.21	61953.82	1.83	21.95
d1655	19681.36	254.01	71550.55	46.89	15.17

Table 19 `ex2-metaheuristic`: Statistics and solution gaps on interrupted TSP instances with timeout = 10000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
sim-grid-B-010	58.91	1.04	3533	0	0
sim-grid-B-015	120.82	21.33	4235	0	0
sim-grid-B-020	137.82	1.17	4924	0	0.74
sim-grid-B-025	190.82	26.32	4929.82	8.87	0.28
sim-grid-B-030	215.91	2.43	5115	0	0
sim-grid-B-035	363.82	41.57	7023	4.45	8.36
sim-grid-B-040	398.73	74.91	5896	0	1.52
sim-grid-B-045	507.45	158.04	6789.55	55.17	3.2
sim-grid-B-050	529.73	99.36	6668.73	1.01	5.74
sim-grid-B-055	510.36	80.62	6946.91	81.73	0.86
sim-grid-B-060	526.64	6.71	7228	0	0.56
sim-grid-B-065	679.55	191.23	7685.09	15.14	3.39
sim-grid-B-070	837.18	128.93	8146.73	1.62	7.98
sim-grid-B-075	1469.91	146.96	17206.36	209.08	6.46
sim-grid-B-080	1253.82	371.15	16701.82	85.80	6.32
sim-grid-B-085	992.91	249.78	17148.45	14.48	5.4
sim-grid-B-090	989.09	94.57	16968.55	81.31	3.94
sim-grid-B-095	1388.82	407.02	18197	25.98	5.06
sim-grid-B-100	1986.82	241.57	19555.55	130.57	8.64
sim-grid-B-200	2918.91	845.61	36894.18	162.6	0.37
d198	3238.45	1091.41	16221.73	25.3	2.8
d493	9266.73	1945.81	39422.73	0.9	12.63
d657	17523.45	2499.97	53618.73	4.1	9.62
d1291	40697.64	8420.90	61952.18	0.6	21.95

Table 20 `ex2-metaheuristic`: Statistics and solution gaps on non-interrupted TSP instances with timeout = 60000ms.

Dataset	Avg ms	Std ms	Avg solution	Std solution	Avg gap
d1655	69725.45	229.84	71428.00	38.85	14.97

Table 21 `ex2-metaheuristic`: Statistics and solution gaps on interrupted TSP instances with timeout = 60000ms.

Dataset	Program	Avg Gap	Solution		
			Avg	Min	Max
sim-grid-B-010	ex1-cplex	0	3533	3533	3533
	ex2-metaheuristic	0	3533	3533	3533
	random-baseline	57.94	5580.09	4578	5919
sim-grid-B-015	ex1-cplex	0	4235	4235	4235
	ex2-metaheuristic	0	4235	4235	4235
	random-baseline	108.81	8842.91	7136	9243
sim-grid-B-020	ex1-cplex	36.4	6667	6667	6667
	ex2-metaheuristic	0.74	4924	4924	4924
	random-baseline	182.04	13785.91	13372	13865
sim-grid-B-025	ex1-cplex	174.61	13500	13500	13500
	ex2-metaheuristic	0.32	4931.55	4916	4935
	random-baseline	173.51	13445.82	12904	13500
sim-grid-B-030	ex1-cplex	159.9	13294	13294	13294
	ex2-metaheuristic	0	5115	5115	5115
	random-baseline	159.9	13294	13294	13294
sim-grid-B-035	ex2-metaheuristic	8.54	7034.36	7021	7067
	random-baseline	146.71	15989	15989	15989
	ex1-cplex	146.71	15989	15989	15989
sim-grid-B-040	random-baseline	172.93	15852	15852	15852
	ex1-cplex	172.93	15852	15852	15852
	ex2-metaheuristic	1.55	5898.18	5896	5908
sim-grid-B-045	random-baseline	147.23	16265	16265	16265
	ex1-cplex	147.23	16265	16265	16265
	ex2-metaheuristic	4.14	6851.36	6815	6855
sim-grid-B-050	ex2-metaheuristic	5.75	6669.45	6668	6670
	random-baseline	208.96	19486	19486	19486
	ex1-cplex	208.96	19486	19486	19486
sim-grid-B-055	random-baseline	143.09	16744	16744	16744
	ex2-metaheuristic	0.51	6923.27	6888	7096
	ex1-cplex	143.09	16744	16744	16744
sim-grid-B-060	ex2-metaheuristic	0.56	7228	7228	7228
	random-baseline	190.4	20874	20874	20874
	ex1-cplex	190.4	20874	20874	20874
sim-grid-B-065	ex2-metaheuristic	3.54	7696	7696	7696
	random-baseline	187.62	21379	21379	21379
	ex1-cplex	187.62	21379	21379	21379
sim-grid-B-070	ex2-metaheuristic	8.46	8183.09	8146	8196
	ex1-cplex	195.14	22268	22268	22268
	random-baseline	195.14	22268	22268	22268
sim-grid-B-075	random-baseline	306.61	65721	65721	65721
	ex2-metaheuristic	9.03	17622.09	17384	17669
	ex1-cplex	306.61	65721	65721	65721
sim-grid-B-080	random-baseline	255.76	55887	55887	55887
	ex1-cplex	255.76	55887	55887	55887
	ex2-metaheuristic	7.04	16814.18	16746	16829
sim-grid-B-085	random-baseline	272.68	60635	60635	60635
	ex2-metaheuristic	5.54	17171	17171	17171
	ex1-cplex	272.68	60635	60635	60635
sim-grid-B-090	ex1-cplex	233.89	54511	54511	54511
	random-baseline	233.89	54511	54511	54511
	ex2-metaheuristic	4.72	17096.82	16932	17137
sim-grid-B-095	random-baseline	229.13	57009	57009	57009
	ex2-metaheuristic	5.59	18289.09	18230	18305
	ex1-cplex	229.13	57009	57009	57009
sim-grid-B-100	random-baseline	294.23	70965	70965	70965
	ex1-cplex	294.23	70965	70965	70965
	ex2-metaheuristic	10.75	19936.91	19876	19943
sim-grid-B-200	ex2-metaheuristic	0	37030	37030	37030
	ex1-cplex	-	-	-	-
	random-baseline	276.83	139539	139539	139539

Table 22 Solutions and average gaps returned by every solver on all simulated datasets of family B, with timeout = 100ms.

Dataset	Program	Avg Gap	Solution		
			Avg	Min	Max
sim-grid-B-010	ex1-cplex	0	3533	3533	3533
	ex2-metaheuristic	0	3533	3533	3533
	random-baseline	58.97	5616.45	4837	5919
sim-grid-B-015	ex1-cplex	0	4235	4235	4235
	ex2-metaheuristic	0	4235	4235	4235
	random-baseline	102.11	8559.36	7154	9243
sim-grid-B-020	ex1-cplex	0	4888	4888	4888
	ex2-metaheuristic	0.74	4924	4924	4924
	random-baseline	182.38	13802.91	13182	13865
sim-grid-B-025	ex1-cplex	0	4916	4916	4916
	ex2-metaheuristic	0.32	4931.55	4916	4935
	random-baseline	174.61	13500	13500	13500
sim-grid-B-030	ex1-cplex	1.22	5177.55	5115	5491
	ex2-metaheuristic	0	5115	5115	5115
	random-baseline	159.9	13294	13294	13294
sim-grid-B-035	ex2-metaheuristic	8.32	7020.45	7018	7021
	random-baseline	146.71	15989	15989	15989
	ex1-cplex	0	6481	6481	6481
sim-grid-B-040	random-baseline	172.93	15852	15852	15852
	ex1-cplex	0.48	5835.64	5808	5960
	ex2-metaheuristic	1.53	5897.09	5896	5908
sim-grid-B-045	random-baseline	147.23	16265	16265	16265
	ex1-cplex	0.21	6592.64	6579	6686
	ex2-metaheuristic	3.28	6794.91	6703	6855
sim-grid-B-050	ex2-metaheuristic	5.39	6647.09	6551	6670
	random-baseline	208.96	19486	19486	19486
	ex1-cplex	44.23	9096.82	8999	10075
sim-grid-B-055	random-baseline	143.09	16744	16744	16744
	ex2-metaheuristic	0.43	6917.45	6888	7050
	ex1-cplex	24.91	8603.91	7073	8757
sim-grid-B-060	ex2-metaheuristic	0.56	7228	7228	7228
	random-baseline	190.4	20874	20874	20874
	ex1-cplex	190.4	20874	20874	20874
sim-grid-B-065	ex2-metaheuristic	3.54	7696	7696	7696
	random-baseline	187.62	21379	21379	21379
	ex1-cplex	69.08	12568	12568	12568
sim-grid-B-070	ex2-metaheuristic	8.02	8150.45	8146	8191
	ex1-cplex	96.02	14789.82	14042	22268
	random-baseline	195.14	22268	22268	22268
sim-grid-B-075	random-baseline	306.61	65721	65721	65721
	ex2-metaheuristic	8.21	17489.64	17235	17669
	ex1-cplex	306.61	65721	65721	65721
sim-grid-B-080	random-baseline	255.76	55887	55887	55887
	ex1-cplex	255.76	55887	55887	55887
	ex2-metaheuristic	6.34	16704.45	16613	16817
sim-grid-B-085	random-baseline	272.68	60635	60635	60635
	ex2-metaheuristic	5.42	17151.27	17140	17171
	ex1-cplex	272.68	60635	60635	60635
sim-grid-B-090	ex1-cplex	233.89	54511	54511	54511
	random-baseline	233.89	54511	54511	54511
	ex2-metaheuristic	3.82	16950.27	16932	17133
sim-grid-B-095	random-baseline	229.13	57009	57009	57009
	ex2-metaheuristic	5.02	18190.18	18182	18197
	ex1-cplex	229.13	57009	57009	57009
sim-grid-B-100	random-baseline	294.23	70965	70965	70965
	ex1-cplex	294.23	70965	70965	70965
	ex2-metaheuristic	9.11	19640	19463	19837
sim-grid-B-200	ex2-metaheuristic	0.11	36990.73	36846	37030
	ex1-cplex	276.83	139539	139539	139539
	random-baseline	276.83	139539	139539	139539

Table 23 Solutions and average gaps returned by every solver on all simulated datasets of family `B`, with timeout = 1000ms.

Dataset	Program	Avg Gap	Solution		
			Avg	Min	Max
sim-grid-B-010	ex1-cplex	0	3533	3533	3533
	ex2-metaheuristic	0	3533	3533	3533
	random-baseline	62.46	5739.55	5008	5919
sim-grid-B-015	ex1-cplex	0	4235	4235	4235
	ex2-metaheuristic	0	4235	4235	4235
	random-baseline	108.4	8825.91	7855	9243
sim-grid-B-020	ex1-cplex	0	4888	4888	4888
	ex2-metaheuristic	0.74	4924	4924	4924
	random-baseline	183.65	13865	13865	13865
sim-grid-B-025	ex1-cplex	0	4916	4916	4916
	ex2-metaheuristic	0.35	4933.27	4916	4935
	random-baseline	173.92	13465.91	13125	13500
sim-grid-B-030	ex1-cplex	0	5115	5115	5115
	ex2-metaheuristic	0	5115	5115	5115
	random-baseline	159.9	13294	13294	13294
sim-grid-B-035	ex2-metaheuristic	8.29	7018.27	6969	7032
	random-baseline	146.71	15989	15989	15989
	ex1-cplex	0	6481	6481	6481
sim-grid-B-040	random-baseline	172.93	15852	15852	15852
	ex1-cplex	0	5808	5808	5808
	ex2-metaheuristic	1.52	5896	5896	5896
sim-grid-B-045	random-baseline	147.23	16265	16265	16265
	ex1-cplex	0	6579	6579	6579
	ex2-metaheuristic	3.6	6815.73	6703	6855
sim-grid-B-050	ex2-metaheuristic	5.74	6669.09	6668	6670
	random-baseline	208.96	19486	19486	19486
	ex1-cplex	0	6307	6307	6307
sim-grid-B-055	random-baseline	143.09	16744	16744	16744
	ex2-metaheuristic	0.36	6912.91	6888	7050
	ex1-cplex	0	6888	6888	6888
sim-grid-B-060	ex2-metaheuristic	0.56	7228	7228	7228
	random-baseline	190.4	20874	20874	20874
	ex1-cplex	0	7188	7188	7188
sim-grid-B-065	ex2-metaheuristic	3.46	7690.55	7666	7696
	random-baseline	187.62	21379	21379	21379
	ex1-cplex	0	7433	7433	7433
sim-grid-B-070	ex2-metaheuristic	7.98	8147.27	8113	8184
	ex1-cplex	0	7545	7545	7545
	random-baseline	195.14	22268	22268	22268
sim-grid-B-075	random-baseline	306.61	65721	65721	65721
	ex2-metaheuristic	6.35	17188.73	16804	17597
	ex1-cplex	6.6	17230	16634	17297
sim-grid-B-080	random-baseline	255.76	55887	55887	55887
	ex1-cplex	43.27	22505.64	15868	25577
	ex2-metaheuristic	6.1	16666.64	16585	16817
sim-grid-B-085	random-baseline	272.68	60635	60635	60635
	ex2-metaheuristic	5.42	17151.27	17140	17171
	ex1-cplex	7.1	17425	17425	17425
sim-grid-B-090	ex1-cplex	49.7	24440	19393	33396
	random-baseline	233.89	54511	54511	54511
	ex2-metaheuristic	3.71	16932	16932	16932
sim-grid-B-095	random-baseline	229.13	57009	57009	57009
	ex2-metaheuristic	5.09	18202.36	18182	18280
	ex1-cplex	45.89	25270.09	23005	26995
sim-grid-B-100	random-baseline	294.23	70965	70965	70965
	ex1-cplex	121.21	39819.73	34686	47154
	ex2-metaheuristic	8.64	19556.82	19463	19736
sim-grid-B-200	ex2-metaheuristic	0.23	36943.45	36784	37030
	ex1-cplex	276.83	139539	139539	139539
	random-baseline	276.83	139539	139539	139539

Table 24 Solutions and average gaps returned by every solver on all simulated datasets of family B, with timeout = 10000ms.

Dataset	Program	Avg Gap	Solution		
			Avg	Min	Max
sim-grid-B-010	ex1-cplex	0	3533	3533	3533
	ex2-metaheuristic	0	3533	3533	3533
	random-baseline	61.17	5694.18	4963	5919
sim-grid-B-015	ex1-cplex	0	4235	4235	4235
	ex2-metaheuristic	0	4235	4235	4235
	random-baseline	110.57	8917.55	7381	9243
sim-grid-B-020	ex1-cplex	0	4888	4888	4888
	ex2-metaheuristic	0.74	4924	4924	4924
	random-baseline	180.28	13700	12300	13865
sim-grid-B-025	ex1-cplex	0	4916	4916	4916
	ex2-metaheuristic	0.28	4929.82	4916	4935
	random-baseline	174.61	13500	13500	13500
sim-grid-B-030	ex1-cplex	0	5115	5115	5115
	ex2-metaheuristic	0	5115	5115	5115
	random-baseline	159.9	13294	13294	13294
sim-grid-B-035	ex2-metaheuristic	8.36	7023	7021	7032
	random-baseline	146.71	15989	15989	15989
	ex1-cplex	0	6481	6481	6481
sim-grid-B-040	random-baseline	172.93	15852	15852	15852
	ex1-cplex	0	5808	5808	5808
	ex2-metaheuristic	1.52	5896	5896	5896
sim-grid-B-045	random-baseline	147.23	16265	16265	16265
	ex1-cplex	0	6579	6579	6579
	ex2-metaheuristic	3.2	6789.55	6703	6855
sim-grid-B-050	ex2-metaheuristic	5.74	6668.73	6668	6670
	random-baseline	208.96	19486	19486	19486
	ex1-cplex	0	6307	6307	6307
sim-grid-B-055	random-baseline	143.09	16744	16744	16744
	ex2-metaheuristic	0.86	6946.91	6888	7050
	ex1-cplex	0	6888	6888	6888
sim-grid-B-060	ex2-metaheuristic	0.56	7228	7228	7228
	random-baseline	190.4	20874	20874	20874
	ex1-cplex	0	7188	7188	7188
sim-grid-B-065	ex2-metaheuristic	3.39	7685.09	7666	7696
	random-baseline	187.62	21379	21379	21379
	ex1-cplex	0	7433	7433	7433
sim-grid-B-070	ex2-metaheuristic	7.98	8146.73	8146	8150
	ex1-cplex	0	7545	7545	7545
	random-baseline	195.14	22268	22268	22268
sim-grid-B-075	random-baseline	306.61	65721	65721	65721
	ex2-metaheuristic	6.46	17206.36	16813	17520
	ex1-cplex	0	16163	16163	16163
sim-grid-B-080	random-baseline	255.76	55887	55887	55887
	ex1-cplex	0	15709	15709	15709
	ex2-metaheuristic	6.32	16701.82	16568	16829
sim-grid-B-085	random-baseline	272.68	60635	60635	60635
	ex2-metaheuristic	5.4	17148.45	17140	17171
	ex1-cplex	0	16270	16270	16270
sim-grid-B-090	ex1-cplex	0	16326	16326	16326
	random-baseline	233.89	54511	54511	54511
	ex2-metaheuristic	3.94	16968.55	16932	17133
sim-grid-B-095	random-baseline	229.13	57009	57009	57009
	ex2-metaheuristic	5.06	18197	18182	18272
	ex1-cplex	0.01	17322.82	17321	17341
sim-grid-B-100	random-baseline	294.23	70965	70965	70965
	ex1-cplex	0.53	18097.27	18001	18484
	ex2-metaheuristic	8.64	19555.55	19440	19860
sim-grid-B-200	ex2-metaheuristic	0.37	36894.18	36652	37030
	ex1-cplex	276.83	139539	139539	139539
	random-baseline	276.83	139539	139539	139539

Table 25 Solutions and average gaps returned by every solver on all simulated datasets of family `B`, with timeout = 60000ms.

Dataset	Program	Avg Gap	Solution		
			Avg	Min	Max
d198	ex1-cplex	-	-	-	-
	ex2-metaheuristic	2.97	16248.27	16225	16256
	random-baseline	42.57	22498	22498	22498
d493	ex1-cplex	-	-	-	-
	ex2-metaheuristic	12.8	39481.91	39469	39494
	random-baseline	224.41	113549	113549	113549
d657	ex1-cplex	-	-	-	-
	ex2-metaheuristic	10	53805.45	53716	53825
	random-baseline	374.65	232159	232159	232159
d1291	ex1-cplex	-	-	-	-
	ex2-metaheuristic	21.96	61958.82	61957	61959
	random-baseline	196.95	150852	150852	150852
d1655	ex1-cplex	-	-	-	-
	ex2-metaheuristic	15.25	71604.91	71591	71608
	random-baseline	231.71	206087	206087	206087

Table 26 Solutions and average gaps returned by every solver on all board-drilling `TSPLIB` datasets, with timeout = 100ms.

Dataset	Program	Avg Gap	Solution		
			Avg	Min	Max
d198	ex1-cplex	42.57	22498	22498	22498
	ex2-metaheuristic	2.86	16232	16143	16252
	random-baseline	42.57	22498	22498	22498
d493	ex1-cplex	-	-	-	-
	ex2-metaheuristic	12.68	39439.45	39423	39473
	random-baseline	224.41	113549	113549	113549
d657	ex1-cplex	-	-	-	-
	ex2-metaheuristic	9.87	53739.73	53680	53808
	random-baseline	374.65	232159	232159	232159
d1291	ex1-cplex	-	-	-	-
	ex2-metaheuristic	21.96	61958	61954	61959
	random-baseline	196.95	150852	150852	150852
d1655	ex1-cplex	-	-	-	-
	ex2-metaheuristic	15.24	71594.73	71560	71608
	random-baseline	231.71	206087	206087	206087

Table 27 Solutions and average gaps returned by every solver on all board-drilling `TSPLIB` datasets, with timeout = 1000ms.

Dataset	Program	Avg Gap	Solution		
			Avg	Min	Max
d198	ex1-cplex	23.33	19461	19461	19461
	ex2-metaheuristic	2.64	16196.64	16061	16252
	random-baseline	42.57	22498	22498	22498
d493	ex1-cplex	224.41	113549	113549	113549
	ex2-metaheuristic	12.63	39422.73	39420	39423
	random-baseline	224.41	113549	113549	113549
d657	ex1-cplex	374.65	232159	232159	232159
	ex2-metaheuristic	9.64	53626.91	53617	53672
	random-baseline	374.65	232159	232159	232159
d1291	ex1-cplex	-	-	-	-
	ex2-metaheuristic	21.95	61953.82	61952	61957
	random-baseline	196.95	150852	150852	150852
d1655	ex1-cplex	-	-	-	-
	ex2-metaheuristic	15.17	71550.55	71461	71608
	random-baseline	231.71	206087	206087	206087

Table 28 Solutions and average gaps returned by every solver on all board-drilling `TSPLIB` datasets, with timeout = 10000ms.

Dataset	Program	Avg Gap	Solution		
			Avg	Min	Max
d198	ex1-cplex	12.82	17802.82	17791	17817
	ex2-metaheuristic	2.8	16221.73	16179	16252
	random-baseline	42.57	22498	22498	22498
d493	ex1-cplex	224.41	113549	113549	113549
	ex2-metaheuristic	12.63	39422.73	39420	39423
	random-baseline	224.41	113549	113549	113549
d657	ex1-cplex	374.65	232159	232159	232159
	ex2-metaheuristic	9.62	53618.73	53617	53631
	random-baseline	374.65	232159	232159	232159
d1291	ex1-cplex	196.95	150852	150852	150852
	ex2-metaheuristic	21.95	61952.18	61952	61954
	random-baseline	196.95	150852	150852	150852
d1655	ex1-cplex	231.71	206087	206087	206087
	ex2-metaheuristic	14.97	71428	71386	71479
	random-baseline	231.71	206087	206087	206087

Table 29 Solutions and average gaps returned by every solver on all board-drilling `TSPLIB` datasets, with timeout = 60000ms.

C Code Structure

In this section, we present the overall structure of the submitted project. The `C++17` software required by the two homework exercises is in the `ex1-cplex` and `ex2-metaheuristic` folders. The random TSP solver we used as a baseline for comparing the two main solvers is in the `random-baseline` folder.

C.1 Exact TSP Solver: `ex1-cplex`

```
/ex1-cplex/
├── cli.h ..... Command-line arguments definition
├── CPLEXModel.h ..... Wrapper class for the exact CPLEX implementation
├── cpx_macro.h ..... Macro utilities for CPLEX
└── main.cpp ..... Executable entrypoint
```

C.2 Metaheuristic TSP Solver: `ex2-metaheuristic`

```
/ex2-metaheuristic/
├── cli.h ..... Command-line arguments definition
├── crossover.h ..... Definition of crossover operators
├── farthest_insertion.h ..... Farthest Insertion heuristic implementation
├── local_search.h ..... High-level methods for neighborhood functions
├── main.cpp ..... Executable entrypoint
├── mating.h ..... Definition of recombination strategies
├── MetaHeuristicsParams.h ..... Struct for the initial parameters
├── mutation.h ..... Definition of mutation operators
├── neighborhood.h ..... Low-level definitions for local neighborhood exploration
├── population.h ..... Functions for generating the initial solution
├── sampling.h ..... Abstractions for random sampling methods
├── selection.h ..... Parents and children selection strategies
├── Solver.h ..... Abstract base class for a genetic algorithm
├── SolverTablePrinter.h ..... Utility for printing tabulated info about the genetic algorithm
├── TSPSolver.h ..... Concrete metaheuristic solver implementation
├── statistics.h ..... Utilities to compute the average solution cost for each generation
└── utils.h ..... General-purpose utilities
```

C.3 `Python` scripts: `python`

```
/python/
├── benchmark/ ..... Module for benchmarking the TSP solvers
├── calibrate/ ..... Module for calibrating the ex2-metaheuristic solver
└── requirements.txt ..... Index of the third-party Python libraries used
```

C.4 Random TSP Solver: `random-baseline`

```
/random-solver/
├── cli.h ..... Command-line arguments definition
├── main.cpp ..... Executable entrypoint
└── RandomSolver.h ..... Wrapper class for the random baseline implementation
```

C.5 Shared `C++` Utilities: `shared`

```
/shared/shared/
└── path_utils/ ..... Contains PermutationPath.h and some other path-related utilities
```

```
├── read_tsp_utils/ ..... Utilities required by read_tsp_file.h
├── DistanceMatrix.h ..... Class that computes the distance matrix of a path
├── Matrix.h ..... Generic matrix implementation
├── read_tsp_file.h ..... Parser for TSP instances in TSPLIB format
└── stopwatch.h ..... Abstraction to track the CPU time
```

C.6 TSP Instances: `tsp-datasets`

```
/tsp-datasets/
├── benchmark/ ..... Datasets used for benchmarking all solvers
└── calibration/ ..... Datasets used for calibrating ex2-metaheuristic
```

D Third-party Libraries

Even though we built most of this project from scratch, we relied on some third-party libraries for purposes outside the content of the course, or for automating some tasks. For instance, we used third-party dependencies for parsing command arguments from the command line, manipulating statistics in a tabular format, and performing a black-box metaoptimization in the calibration step of the Genetic Algorithm. The main dependencies we used are cited in the following paragraphs.

Argparse

Argparse[11] is a `C++17` modern command-line argument parser. We used it to specify the value of the configuration options (metaheuristic hyperparameters, maximum alotted timeout, TSP instance file) for the programs we implemented.

PriorityQueue

PriorityQueue[12] is a `C++` priority queue data-structure implementation based on Binary Heaps and K-Heaps. We adopted it to efficiently sample path indexes based on probabilities in the (μ, λ) selection implementation discussed in Section 4.7. We are the authors of this library.

Pymoo

Pymoo[13] is a `Python` multi-objective optimization suite developed at the *Computational Optimization and Innovation Laboratory* of the *Michigan State University* [BD20]. We used it as the basis for the calibration step of our metaheuristic implementation.

Scientific `Python` libraries

In order to aggregate the experimental data, create plots, and simulate TSP datasets, we relied on some very common `Python` libraries, whose details will be omitted:

- **Pandas** [pdt20];
- **Numpy** [CRH20];
- **Seaborn** [Wtsdt20];
- **IPython** [PG07].

[11] https://github.com/morrisfranken/argparse
[12] https://github.com/jkomyno/priority-queue
[13] https://github.com/msu-coinlab/pymoo

E Scripts Execution

The following paragraphs describe how to run the software we developed. We assume that the reader has downloaded our project repository, and that is using a computer running any `x86/x64` Linux distribution with `make`, `g++ 7.5.0` (or superior), and `IBM CPLEX 12.8.0` installed. Moreover, for running the benchmark and calibration scripts, we assume that `Python 3.6.9` (or superior) is installed.

E.1 Compiling the Solvers

We defined a `Makefile` to include all third-party dependencies and link the required dynamic libraries for our programs written in `C++17`. To create the `ex1-cplex.out`, `ex2-metaheuristic.out`, and `random-baseline.out` files in the `./build` folder, please run:

```
make all
```

E.2 Running the Solvers

Every solver shares some common command-line options:

- `-f, --filename [FILE]`: Name of the input TSP instance file;
- `-t, --timeout-ms [TIMEOUT]`: Timeout expressed in milliseconds;
- `-s, --show-path`: If specified, it shows the Hamiltonian circuit found by the solver;
- `--help`: If specified, it shows the available command-line options.

By default, the timeout is set to 1000 milliseconds (1 second) and the actual solution path is not shown (only the cost of the solution is reported).

We also provided some convenient scripts for running the solvers. Command-line arguments are propagated via the $@ bash expansion.

Running `ex1-cplex`

```
./scripts/ex1-cplex.sh
```

Running `ex2-metaheuristic`

```
./scripts/e2-metaheuristic.sh
```

`ex2-metaheuristic` supports some other command-line options:

- `-m, --mutation-probability [PROBABILITY]`: Probability that a mutation occurs in the genetic algorithm;
- `-c, --crossover-rate [RATE]`: Probability that two selected solutions are mated to create a new offspring;
- `--mu [SIZE]`: Size of the population pool;
- `--lambda [SIZE]`: Size of the offspring pool before being pruned;
- `-k [SIZE]`: Size of the tournament selection;
- `-N, --max-gen-no-improvement` [NUMBER]: Maximum number of generations without solution improvement;
- `-M, --max-gen` [NUMBER]: Maximum number of generations.

For each of these custom arguments, we set a default value equal to the result of the metaheuristic calibration phase. Even if the calibration phase is repeated in the future, the default arguments are updated dynamically without having to compile the solver's `C++17` sources again.

Running `random-baseline`

```
./scripts/random-baseline.sh
```

E.3 Running the `Python` Scripts

Please keep in mind that running either the benchmark or calibration scripts will take many hours. First of all, please install the required third-party dependencies using `pip`:

```
python3 -m pip install -r ./python/requirements.txt
```

Running the Benchmark Script

```
./scripts/benchmark.sh
```

Running the Calibration Script

```
./scripts/calibrate.sh
```