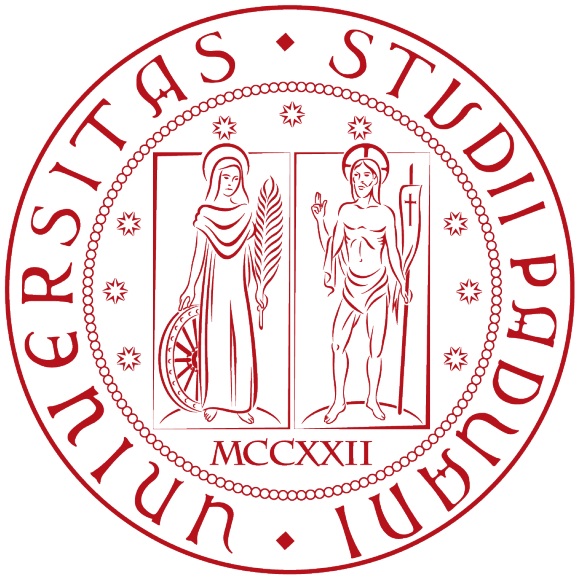


Department of Mathematics

Computer Science Master's degree

Course of Methods and Models for Combinatorial Optimization a.y. 2019/20

TSP project report

Author:
Alessandro Zangari

Student id:
1207247

December 24, 2024

Contents

List of Figures

List of Tables

Abstract

The following report documents the course project, which demanded the development of two algorithms to solve the Travelling Salesman Problem. The first algorithm uses the C IBM CPLEX APIs to obtain the exact solution. The second algorithm is an heuristic method, whose functioning is inspired by the Lin-Kernighan heuristic.

1 Introduction

1.1 Description of the problem

The task of the exercise was to develop two algorithms capable of solving the Travelling Salesman Problem (TSP), specialized to the domain of drilling holes in electric panels. Given a sequence of holes to be drilled in an electric panel, the algorithms should be used to find a sequence of holes that minimizes the cost of drilling the whole panel, where the costs are given by the euclidean distances between holes. For simplicity it is assumed that the cost of drilling every hole can be ignored.
The first algorithm implements an exact methods using the IBM CPLEX optimization suite. The second algorithm is an approximated heuristic, inspired by the Lin-Kernighan algorithm [LK73]. Some implementation details has been taken from [Mah17].
Both algorithms have been tested on synthetic instances produced taking into account the applicative domain. The two algorithms are described in the following pages, and tests and results are reported in the last sections.

1.2 Description of provided material

The delivered material includes all the produced code and everything necessary to run the program. The provided archive comes with the following structure:

```
/
├── bin/ ..... Binary files folder (after compiling)
├── build/ ..... Build files folder (after compiling)
├── files/ ..... Logs folder (after running)
├── instances/ ..... Contains csv files to load
├── plots/
├── src/
│   ├── utils/
│   │   ├── cpxmacro.hpp
│   │   ├── params.hpp
│   │   ├── python_adapter.hpp ..... Wrapper class for Python embedding
│   │   ├── variadic_table.hpp ..... Library to print tables
│   │   └── yaml_parser.hpp ..... Parser for the configuration file
│   ├── calibrate.cpp ..... Calibration script
│   ├── config.yml
│   ├── CPLEX.cpp
│   ├── CPLEX.hpp
│   ├── LK.cpp
│   ├── LK.hpp
│   ├── main.cpp ..... Program main
│   ├── Pair.cpp
│   └── Pair.hpp
```

```
├── plot_script.py ..... Script to plot with Python
├── test.cpp ..... Test script
├── Tour.cpp
├── Tour.hpp
├── TSPinstance.cpp
├── TSPinstance.hpp
├── TSPsolution.cpp
├── TSPsolution.hpp
├── utilities.cpp
└── utilities.hpp
```

1.2.1 How to run

The program has the following dependencies:

- `g++ 7` or higher;
- `IBM ILOG CPLEX Optimization Studio 12.8` or higher;
- `Python 2.7`.

To test the program, some instances should be generated or imported by placing them in the `instances/` folder. Some files are already provided in this folder to launch some tests. To generate new instances set `generate_instances: true` in the `config.yml` file, and the other parameters as desired. If `N_min = 20, N_incr = 5, N_max = 30`, three `csv` files containing the coordinates of every point will be generated, with size 20, 25 and 30 respectively, and will be placed in the instances folder. After that it is possible to run the exact method or the heuristic on these instances. To do that, add to the list in `instances_to_read` the filename (without extension) of the files that should be read and set `solve_heur` and `solve_cplex` as desired. Some reasonable parameters for the heuristic are already set, but it is possible to change them if necessary. A description of these specific parameters is provided in section 4.6. The script can then be run from a Linux environment with `make && ./bin/main`. The algorithm writes its output in file `solLK.txt` and `solCPLEX.txt` under the `files/` folder. When solving more than one problem, these files can also be checked while the program is running to monitor its execution. Additionally running the heuristic also produces an image of the final solution and places it in folder `plots/`. Furthermore, running both CPLEX and the heuristic together will produce plots with a comparison of the execution times and error values in the same folder.

1.3 Development environment

The specifics of the machine used for development, calibration and test of the program are listed in table 1.

Table 1: Hardware and software specifics

Specific	Value
OS	Ubuntu 18.04.4 LTS 64 bit
Processor	Intel Core i7-7500U 2.70GHz × 4
Main memory	16 GB

2 Dataset generation

In order to test the two algorithms, a procedure to build a synthetic dataset of different sizes has been provided.
The procedure to generate the TSP instances receives as an input a number N of points (which symbolises holes) and generates N pairs representing the coordinates in space of the points on a $N \times N$ square canvas. These points are distributed in a way to resemble the regularity usually found in the disposition of holes in electric panels.
To do this the procedure draws some regular polygons with up to 10 sides. Each polygon is generated independently and may overlap with the others, creating not perfectly regular shapes, but neither a completely random point distribution. An example of such an instance is shown in fig. 1. All this operations, as well as the function to load already generated instances from `csv` files, are implemented in `TSPinstance.cpp`.

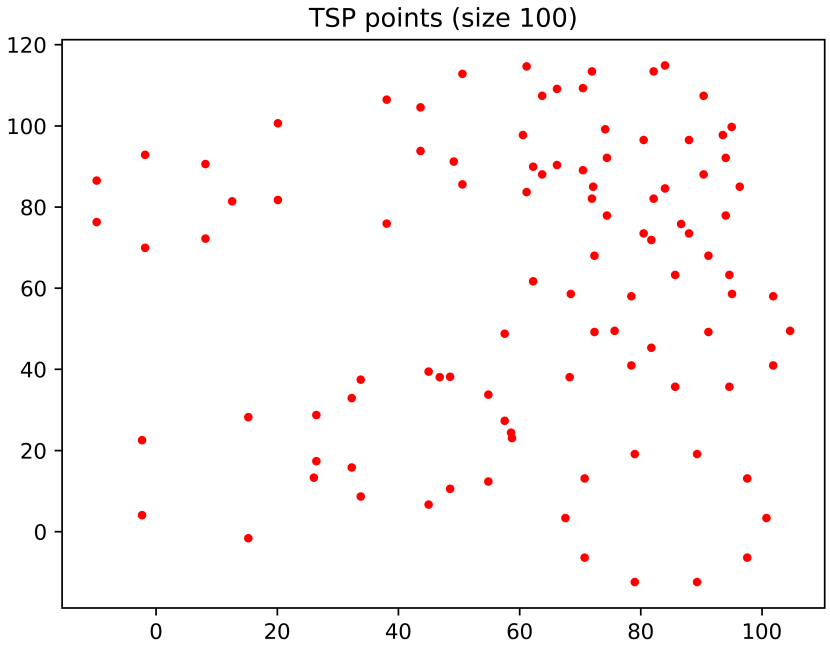


Figure 1: A generated instance of with size N = 100

3 Exact method

The exact algorithm makes use of the IBM CPLEX C APIs to solve a problem to optimality. In order to model the TSP problem a network flow model is used, as described in the assignment text.
The specific linear programming model used and its decision variables are presented below. The set A is the set of edges of the graph, which in this case is a complete graph.

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

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

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

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

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

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

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

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

Actually, when adding the variables x and y to the model, variables $x_{ij}\ \forall\ (i,j) \in A, j \neq 0$, instead of being unbounded like in constraint 6, they are forced to be in range $[0, N-1]$. The model is still valid, since constraint 5 also bounds the variables in this range.

3.1 Creation of the model

The linear programming model is set up in the function `setupLP` in file `CPLEX.cpp`.
During the creation of the model, its decision variables x_{ij} and y_{ij} are first added one by one (for each possible value of i and j) through the appropriate CPLEX functions to a C array. Variables with $i = j$ are not added, since no loops on the same point is allowed. Considering that this variables have to be referenced later during the definition of the constraints, their position in the CPLEX array is recorded in two bidimensional vectors, one for the x, called `xMap`, and one for y variables, named `yMap`. In this way variable x_{ij} can be later referenced easily with `xMap[i][j]` for all i and j. After defining the variables of the problem, the constraints 2 - 5 are inserted. Constraint 5 was reformulated as $x_{ij} - (|N|-1)y_{ij} \leq 0$ because the CPLEX APIs require all the variables in the left-hand side and only constants in the right side of the equation.

The model can be solved calling the function `solve`, which calls the IBM CPLEX solving routines for a minimization problem. The solution can then be retrieved using the method `getSolution` which returns a `TSPsolution` object, which wraps many useful information about the final solution, and is used for both the exact method and the heuristic one.

4 Heuristic method

The implemented heuristic is inspired by a popular local search method called Lin-Kernighan heuristic, originally proposed in 1973 in [LK73]. The algorithm can be considered a generalization of the k-opt algorithm: one of the drawbacks of this algorithm is that the parameter k must be fixed in advance. Instead, the Lin-Kernighan algorithm decides at each iteration, for ascending values of k, whether an interchange of k edges provides a better solution. Thus, the algorithm is specified in terms of exchanges that can convert one tour into another: given a feasible interchange of k edges (a *k-opt move*), the algorithm tries to determine if there exists a *(k+1)-opt* move that improves the tour further. During each iteration, given a feasible tour, the algorithm repeatedly performs exchanges that reduce the length of the current tour, until a tour is reached for which no exchange yields an improvement. This process may be repeated many times from initial tours generated in some possibly randomized way [Hel00].
So at each iteration the algorithm tries to determine the largest set $X = \{x_1, x_2, ..., x_j\}$ and $Y = \{y_1, y_2, ..., y_j\}$ such that if edges in X (also called the *broken* edges) are removed and replaced by Y (the *joined* edges) the produced tour is a feasible improved (i.e. less costly) solution. Of course, a naive brute force algorithm searching for this sets, would quickly become impractical to use, due to the exponential running time. In order to produce a reasonably efficient local search procedure, the algorithm reduces the search space using the following criteria:

1. *Sequential exchange criterion*: each pair of edges (x_i, y_i) and (y_i, x_{i+1}) must share one vertex. If t_1 and t_2 are the vertices of edge x_1, then in general for all $i \geq 1$ exchanges are performed this way: $x_i = (t_{2i-1}, t_{2i})$ and $y_i = (t_{2i}, t_{2i+1})$ and $x_{i+1} = (t_{2i+1}, t_{2i+2})$.
   All *sequential* k-opt moves can be found by concatenating smaller sequential moves, but *non sequential* moves are never considered by this algorithm. An example of such a move is given in fig. 2.

2. *Feasibility criterion*: for every $i \geq 3$, $x_i = (t_{2i-1}, t_{2i})$ is chosen so that if t_{2i} is connected to t_1 the resulting configuration is a tour (i.e. a feasible solution). It can be seen that at most one choice for x_i satisfies this constraint, as showed in fig. 3. Exceptionally, when $i = 2$, the originally proposed algorithm allowed for the choice of an x_i which violates this rule. According to the original paper this was done to strengthen the procedure by giving the algorithm a way to recover from previous wrong choices, but allowing this kind of backtracking at every levels would significantly increase the running time;

3. *Positive gain criterion*: let $g_i = cost(x_i) - cost(y_i)$ be the gain by exchanging two edges, and let $G_i = g_1 + g_2 + ... + g_i$ be the partial sum of the gains up to the i^{th} exchange. This

criterion requires that each y_j is chosen in a way such that G_j is positive. This choice is justified by the fact that if a sequence of numbers has a positive sum, there is a cyclic permutation of these numbers such that every partial sum is positive [Hel00];

4. *Disjunctivity criterion*: sets X and Y must be disjoint.

In the implemented algorithm, the move size threshold which allows for the exceptional violation of the *feasibility criterion* (set to 2 in the paper), is not fixed but it can be modified in the configuration file `config.yml`. In this way the influence of this parameter on the solution quality could be tested and optimized.

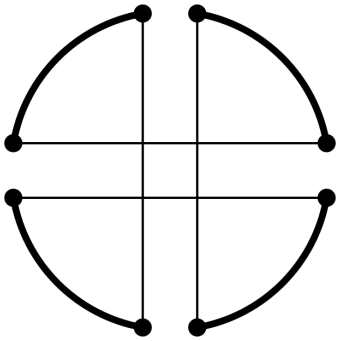
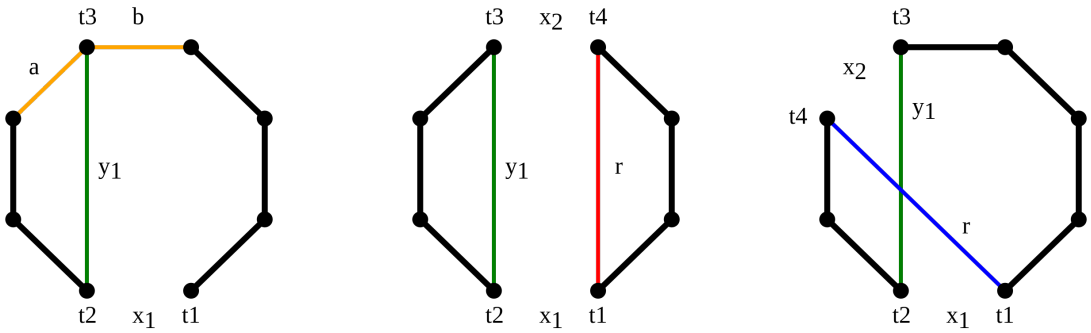


Figure 2: A non sequential *double bridge* move with $k = 4$



(a) The two possibles edges to break after choosing y_1: a, b

(b) Breaking edge b violates the feasibility criterion

(c) Breaking edge a allows to close the tour correctly

Figure 3: Only one candidate for x_i satisfies the feasibility criterion, which is enforced for i greater than a fixed threshold value. In this case breaking edge a is the right choice, since removing edge b and relinking with edge $r = (t_4, t_1)$ would not produce a feasible tour.

4.1 Stopping criterion

The algorithm uses the same stopping criterion suggested in the original paper. In particular, when the algorithm finds a feasible tour with a move of size k, it checks whether the total gain is the best seen so far. If it is, then it is the best improvement seen and it saves the current solution, and tries to improve further with a $(k+1)$-opt move. If this does not improve the solution the algorithm returns the solution it saved. On the other hand, if the size k move yields a feasible tour which is not the best seen so far, the algorithm stops, since it knows there is a better move, with size smaller than k, that produces a better tour.
By accepting only improving solutions, the algorithm is guaranteed to eventually stop, since it cannot loop over the same local optima.

4.2 Tour structure

A tour is maintained as a vector v of pairs, where v_i gives a pair with the two vertices which precedes and follows i in the current tour. To this purpose every vertices (point) of the instance is numbered from 0 to $N-1$ and its number is used as index to access the tour structure. Creation of such structure takes time $O(N)$ from a list of vertices where N is the problem size. This allows constant time checks and retrieval of the two candidates for removal. Moreover, the distance between each vertex (the cost of each edge) is kept in a $N \times N$ sized matrix.

4.3 Neighbourhood function

In a local search algorithm the neighbourhood function describes which solutions are explored at every step. In this case every neighbour is generated with a k-opt move from the initial solution and the algorithm tries to increase k at each step.
In the described algorithm, for a given i, a step consists in breaking an edge of the tour x_i and replacing it with and edge y_i chosen with the criteria described in section 4. Every step generates a new neighbour. If we fix the vertex t_{2i-1} belonging to joined edge y_{i-1}, there are exactly two edges that can be broken. That's because the *sequential exchange criterion* requires that x_i must share an endpoint with y_{i-1} and this endpoint is exactly t_{2i-1}. Additionally, by the *feasibility criterion*, there is only one choice of x_i that will produce a feasible tour, while breaking the other will inevitably split the graph into two connected components (as seen in fig. 3).
Starting from vertex t_{2i-1} the algorithm analyses the two possibilities giving the precedence to the edge with higher cost, since is the best one to remove. Only if the feasibility criterion is violated the procedure will try to remove the other edge.
In order to choose an edge y_i, the algorithm finds all possible edges starting in t_{2i} which are

- not part of the current solution;
- not already broken (i.e. $\notin X$);
- not already joined (i.e. $\notin Y$).

A simple approach would be to choose as y_i the candidate edge with the lowest cost, since we want to maximize the gain. The paper suggests using a less greedy approach, by looking also at the successive edge x_{i+1}. This has the additional benefit of avoiding useless search, as would happen by joining a promising candidate only to find out, when selecting the next edge to break, that no such operation is possible (for example because the only alternative has already been broken).
Thus for each candidate edge to join, the possible x_{i+1} are analysed, and the potential gain of adding y_i and removing x_{i+1} is computed. Since the algorithm does not know which choice of x_{i+1} gives the feasible tour, if neither choice can be discarded because of other reasons (e.g. already broken, intensification constraints), both choices are considered and the potential gain is the average of the two gains. So the potential gain is the gain expectation that can be achieved by joining y_i and breaking x_{i+1}. Ideally, the neighbourhood function should predict which candidate x_{i+1} is the actual feasible choice that the algorithm will eventually make. Unfortunately, checking this for every possible candidate is quite costly for instances with considerable size, since checking if a sequence of edges represent a feasible tour takes time $O(N)$ on average, and the number of candidates grows linearly with N too.
Instead of using the average potential gain to rank the candidates for joining, two alternative approaches using the worst gain and the best possible gain has been tested. Calibration tests on an instance of 90 points determined that this approach produced slightly better results than the others.

4.4 Enhancements

This basic algorithm scheme can be enhanced in many ways. The next sections describe some strategies implemented in this project.

4.4.1 Search intensification

Between the iterations the algorithm may find some 'good' edges which belongs to the optimal solution or to some reasonably good one. Subsequently some time may be lost by repeatedly breaking and adding them, while not exploring some more promising neighbours.
In addition these 'good' edges are likely to be present in many of the best solutions (local optima) found between iterations. In order to direct the search towards more promising solutions, the algorithm keeps track of the edges shared by best S solutions by computing their intersection.
On the other hand, if the top-S solutions do not include the global optimum, this approach may prevent the algorithm from breaking some edges which do not belong to the optimum, thus preventing the algorithm from finding it.
To mitigate this problem, the intensification procedure is applied only after a specified number of improving moves (i.e. after a fixed i). This means that the algorithm has a way to escape this additional constraint for a small number of moves, allowing it to recover from previous sub-optimal choices.

Additionally, after a local optima is found, the objective value is tested to see if it is better than one of the top-S solutions and in this case the worst solution is replaced with the new one and the intersection is updated.
A max-heap data structure is used in order to efficiently replace the worst (i.e. with higher objective value) local optima, and the collection of edges of each tour is maintained in a `std::set` ordered container, which allows the intersection procedure to take time $O(N \cdot S)$ where N is the number of edges in a feasible tour.
Instead of considering just the top-S solutions to do the intersection, the original paper proposed to keep the intersection of all tours from the beginning. During some early tests this seemed to worsen the results with respect to the runs without intensification, so this change was made, and a small reduction of running time has been registered, since the algorithm excludes some choices from its neighbourhood. In choosing the parameter S it is important to keep in mind that intensification is done only after S local optima are found, so one should ensure that this number is suitable for the problem dimension, since a very large number with a small problem would probably be ineffective.

4.4.2 Avoiding duplicate solutions

The authors of the algorithm stated that 30% to 50% of running time could be saved by keeping track of previous local optima and stopping the algorithm if the current local optima is the same seen before. This is justified by the fact that if the algorithm couldn't improve this solution before, it's unlikely it can do it now. While this sounds reasonable, in the proposed implementation no particular speed-up was registered. In part, this may be due to different computing capabilities, considering that the algorithm was devised in 1973.
Since trying to improve a previously found local optima will likely fail, the proposed algorithm adopts a slightly different strategy in this eventuality: instead of stopping, it restarts from a different starting edge and proceeds normally until it finds a different solution, or it determines no better tour exists. While this does not guarantee an improvement in running time, it makes the algorithm redirect the search in a different direction, which is probably more useful that exploring the neighbourhood of a previous solution.
To do this, an `std::set` data structure is used, which maintains a sorted collection (ordered by tour cost) of all the feasible tours found at each iteration. This container guarantees logarithmic insertion and search operations.
As a possible improvement to this strategy, one could keep track of the edges modified while converging to the previously encountered solution, and make sure that at least some of these are not added again, which can potentially lead to the same solution again, or to a very similar one.

4.5 Random restart

The described heuristic is run from a randomized initial solution, like the one in fig. 4. This simple strategy allows the algorithm to escape local optima and differentiate the provided solutions. Using an initial solution generated with a fast constructive heuristic has been tried, and a faster convergence has been registered, even though the produced solution was often not better than the solution generated from a random initial solution. Despite this, the starting tour has a significant impact on the heuristic performance, since it determines the edges that will be tried for the first move. Indeed the algorithm often does not explore more than the first two-three candidates to be joined. This explains why the initial tour greatly contributes to the final result.
By restarting the heuristic many times on the same instances of the synthetic dataset of size > 90, on average, the heuristic will produce tours 0.5% from the optimal one, but the best solution found during a reasonable number of restarts is usually much closer to the optimal one. This strategy has been implemented in the file `utilities.cpp`, where the procedure `runILK` is defined for this purpose. Setting the desired number of restarts in the `config.yml` configuration file (parameter `LK_iterations`) will restart the algorithm the specified number of times, and the final result will be the best solution found.

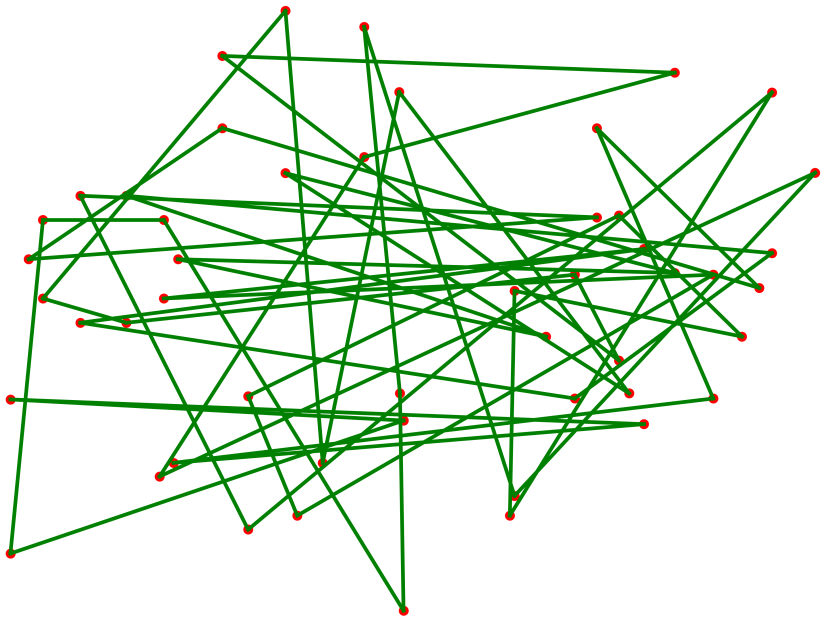


Figure 4: Random initial path on a problem with 50 points

4.6 Parameters

Table 2 includes a description of each parameter to be set in file `config.yml` before starting the heuristic. Calibration of this parameters are discussed in the next section.

Table 2: Parameters of the heuristic algorithm

Parameter	Description
K	The maximum size of an improving k-opt move. If the algorithm doesn't find a move of size $\leq K$ it will stop searching
I	Max number of improving iterations
max_neighbours	The maximum number of edges y_i to consider at every iteration
backtracking_threshold	During the search for an exchange of size $2 \leq j \leq backtracking\_threshold$, if the feasible choice of edge x_j fails to improve the solution, the algorithm will consider the other possibility, even if it is not feasible. In this case the only way to improve is by satisfying the feasibility criterion with exchanges of size $> j$
intens_min_depth	The minimum move size before applying the intensification constraints (as discussed in section 4.4.1)
intens_n_tours	The number of best solutions to keep and to use for intensification. The algorithm will compute this number of local optima before applying intensification (it is the S discussed in section 4.4.1)
LK_iterations	The number of times to restart the heuristic from a random tour

This heuristic may require some tuning to be usable with bigger problems, since some parameters can significantly increase the running time. To determine the next move, the heuristic considers at most `max_neighbours` candidates for joining, and it may do up to `K` moves per iteration, so the worst case complexity of a single search for the best possible move is $O(max\_neighbours^K)$. This process may be repeated for up to `I` iterations. Using random restarts this entire procedure is repeated exactly `LK_iterations` times. High `backtracking_threshold` values allow the algorithm to violate the feasibility criterion and try more moves, but will also require more time. Low values, like 3 or 4, worked well during testing, even though 2 is more likely the best parameter for instances of size greater than 300.
If the algorithm spend too much searching for a move (i.e. searching for an exchange), the `K` parameter should be lowered. On the other hand, if the algorithm tends to do too many iterations, this number may be limited by lowering `I`. Reducing `K` and increasing `I` allows the

algorithm to do a bigger number of smaller improvements. On the other hand increasing `K` and reducing `I` will allow for a lower number of better improvements.

4.6.1 Calibration

Some parameters in table 2 may affect the running time and the solution quality. In order to determine how much these parameters influenced this metrics and to select the best ones for the final tests, some trials were done on two different instances, one of size 90 and one of size 198 taken from TSPLIB[1]. Given some sets of parameters to try, every possible combination has been tested by restarting the heuristic 10 times on each combination. The results are reported in tables and discussed in the next paragraphs. Listed information includes the average percentage error, whether the optimal solution was found, the average number of iterations and the total time required for the 10 iterations. The final score is obtained by iterating the heuristic 10 times on the same instance and takes into account the average percentage relative error (w.r.t. the optimal solution computed using the exact algorithm), called ARE and the total execution time for the 10 iterations, called TT. The score is given by eq. (8).

$$\text{Score} = \frac{1}{1 + (ARE * TT)} \tag{8}$$

These results were obtained by using all the features described before, and by compiling with `-O3` optimization flag. Both tests were done with `K = 100`.
The calibration procedure is implemented in file `calibrate.cpp`, and more tests can be done by editing the file and running the script with commands `make calibrate && ./bin/calibrate`.

Synthetic instance of size 90

Table 3 reports the results of the calibration procedure on an instance with 90 points. Other tests, which are not reported here, were run using the other two ranking strategies to sort the candidates to be joined, discussed in section 4.3. Strategies using the best and average gain performed similarly, while the strategy sorting by worst gain performed poorly. Three different calibrations for all three strategies has been run and the average gain strategy was selected because it reached the optimal solution more frequently then the one using the best gain.
As can be seen in table 3, considering more neighbours during the selection of the edge to join (5 instead of 2) results in a lower average error and higher running time. Additionally with value set to 5 the optimal value was found in all cases but one.
Additional tests were done with `backtracking_depth` set to 2 and 3. Lowering this parameter reduces running time and also the frequency of finding the optimum, so in the table only the runs with depth 4 or 5 are listed.

[1] http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html

Table 3: Results of calibration on 90 points, with best row in yellow

Max neighbours	Backtracking depth	Intensification depth	Intensification tours	Avg error (%)	Optimal	Avg iterations	Total time (s)	Score
2	4	5	20	1.483	Yes	41	0.892	0.431
2	4	5	40	1.753	**No**	45	1.010	0.361
2	**4**	**5**	**60**	**0.656**	**Yes**	**42**	**0.848**	**0.643**
2	4	8	20	1.640	Yes	46	1.037	0.370
2	4	8	40	1.312	Yes	45	0.868	0.468
2	4	8	60	2.409	Yes	48	0.944	0.305
2	4	10	20	2.652	**No**	44	0.854	0.306
2	4	10	40	2.509	**No**	47	0.967	0.292
2	4	10	60	2.495	Yes	46	0.902	0.308
2	5	5	20	1.212	Yes	44	1.212	0.405
2	5	5	40	1.426	**No**	45	1.303	0.350
2	5	5	60	0.927	Yes	45	1.298	0.454
2	5	8	20	1.568	Yes	43	1.456	0.305
2	5	8	40	0.570	Yes	43	1.114	0.611
2	5	8	60	1.754	Yes	44	1.241	0.315
2	5	10	20	1.668	Yes	44	1.325	0.312
2	5	10	40	1.369	Yes	45	1.332	0.354
2	5	10	60	1.697	Yes	45	1.376	0.300
5	4	5	20	1.682	Yes	42	4.568	0.115
5	4	5	40	0.656	Yes	38	4.225	0.265
5	4	5	60	1.953	Yes	42	5.667	0.083
5	4	8	20	1.383	Yes	40	4.415	0.141
5	4	8	40	0.927	Yes	43	4.225	0.203
5	4	8	60	0.898	Yes	44	5.029	0.181

Table 3: Results of calibration on 90 points (continuation)

Max neighbours	Backtracking depth	Intensification depth	Intensification tours	Avg error (%)	Optimal	Avg iterations	Total time (s)	Score
5	4	10	20	2.409	Yes	44	5.918	0.066
5	4	10	40	2.267	**No**	42	4.295	0.093
5	4	10	60	0.984	Yes	42	4.123	0.198
5	5	5	20	1.796	Yes	41	10.190	0.052
5	5	5	40	0.656	Yes	41	9.736	0.135
5	5	5	60	1.682	Yes	41	10.696	0.053
5	5	8	20	1.112	Yes	41	11.016	0.075
5	5	8	40	1.868	Yes	44	13.273	0.039
5	5	8	60	1.440	Yes	40	11.377	0.058
5	5	10	20	0.342	Yes	42	9.264	0.240
5	5	10	40	0.870	Yes	39	9.811	0.105
5	5	10	60	0.143	Yes	39	6.631	0.514

TSPLIB d198

In order to select some good parameters for a non trivial instance, and additional calibration was done on a drilling problem with 198 points from TSPLIB, called `d198`.

Since the instance is bigger, tests were done with higher values for `max_neighbours`, `backtracking_depth` up to 4 (5 is too much for big instances), and the number of tours used for intensification has been limited to 20 or 50. Table 4 shows the results. This time the score is much lower because of the higher running time, so it has been multiplied by 100, to make it more readable.

Table 4: Results of calibration on problem d198, with best row in yellow

Max neighbours	Backtracking depth	Intensification depth	Intensification tours	Avg error (%)	Optimal	Avg iterations	Total time (s)	Score (×100)
3	2	5	20	7.414	No	173	16.206	0.825
3	2	5	50	7.644	No	172	13.098	0.990
3	2	8	20	6.534	No	169	15.293	0.991
3	**2**	**8**	**50**	**6.715**	**No**	**160**	**13.949**	**1.056**
3	2	10	20	7.536	No	170	15.132	0.869
3	2	10	50	7.561	No	172	14.319	0.915
3	3	5	20	6.835	No	168	38.250	0.381
3	3	5	50	6.646	No	160	35.921	0.417
3	3	8	20	6.686	No	173	40.246	0.370
3	3	8	50	6.375	No	166	35.907	0.435
3	3	10	20	7.032	No	164	39.012	0.363
3	3	10	50	6.178	No	167	36.073	0.447
3	4	5	20	6.369	No	172	92.831	0.169
3	4	5	50	6.440	No	180	96.476	0.161
3	4	8	20	6.453	No	182	101.603	0.152
3	4	8	50	6.226	No	179	98.509	0.163
3	4	10	20	6.429	No	174	96.207	0.161
3	4	10	50	6.487	No	173	94.989	0.162
5	2	5	20	7.122	No	157	23.273	0.600
5	2	5	50	5.933	No	160	22.145	0.755
5	2	8	20	6.446	No	156	23.502	0.656
5	2	8	50	6.582	No	158	21.338	0.707
5	2	10	20	6.055	No	162	22.772	0.720
5	2	10	50	6.065	No	163	22.899	0.715

Table 4: Results of calibration on problem d198 (continuation)

Max neighbours	Backtracking depth	Intensification depth	Intensification tours	Avg error (%)	Optimal	Avg iterations	Total time (s)	Score (×100)
5	3	5	20	6.189	No	177	86.192	0.187
5	3	5	50	5.750	No	172	81.449	0.213
5	3	8	20	6.144	No	164	83.906	0.194
5	3	8	50	5.798	No	160	76.600	0.225
5	3	10	20	**4.911**	No	172	88.131	0.231
5	3	10	50	6.416	No	171	81.883	0.190
5	4	5	20	6.350	No	185	338.084	0.466e−1
5	4	5	50	6.419	No	177	337.700	0.461e−1
5	4	8	20	6.291	No	182	338.429	0.469e−1
5	4	8	50	6.160	No	177	312.465	0.519e−1
5	4	10	20	6.269	No	176	336.557	0.474e−1
5	4	10	50	6.213	No	181	336.470	0.478e−1

No execution arrived at the optimal solution, but the original paper suggests that with more restarts better solutions can be found. In three runs with `max_neighbours = 5`, `backtracking_depth = 3` the best error of the 10 iterations was below 1.0, in particular 0.374, 0.773 and 0.425 with `intensification_depth` to 5, 8 and 10 respectively. In the first case the average error was 4.911, which is the best one reported in the table (in bold). In this case the final score strongly penalises the executions with higher backtracking depth because of the longer execution time, but the average error tends to decrease with `max_neighbours = 5` and `backtracking_depth = 3`.

5 Tests and results

The described heuristic has been tested on instances generated as described in section 2, and its solution has been compared with the optimal solution obtained with the exact method. This has been done for instances of size up to 110, since the exact method employed too much time to solve bigger problems. In addition the same tests have been performed on some symmetric drilling problems from TSPLIB, namely `d198`, `d493` and `d657`, with respect to the optimal solutions listed at [Tsp95]. The parameters were set with the following values:

- K = 400 (so unbounded for size 10-110)
- I = 1000
- `max_neighbours = 5`
- `backtracking_threshold = 3`
- `intens_min_depth = 5`
- `intens_n_tours = 50`

Parameter `backtracking_threshold` and `intens_n_tours` were set to a compromise between the best values found during calibration, and `intens_min_depth` was set to 5, since this value seemed to work best with size 90. Finally `max_neighbours` was set to 5 because it resulted in a lower average error in both calibrations.

5.1 Tests on synthetic dataset

For each problem size to test, one instance was generated, and the heuristic was run on it first with 5 random restarts, then with 15 and 30. It was run 10 times for each different parameter, and the mean, standard deviation, minimum and maximum value were computed for both execution times and relative percentage error with respect to the optimal value. Tables 5 - 12 contains the results for some problems. Figure 5 shows the heuristic solution for the bigger problem. Plots in figures 6 and 7 summarise the results for tests with 15 restarts. In addition figure 8 compares the execution times of the two methods. Here the heuristic time is the arithmetic mean of the 10 executions with 15 restarts, while the exact method was run once for every problem. Finally, table 13 lists the average execution time of the exact method over 5 executions on different instances. Figure 9 shows a comparison between the average execution times of the two methods (not scaled).

Table 5: Test results on size 10

	Percentage relative error				Execution time			
Restarts	**Mean**	**Std. dev**	**Min**	**Max**	**Mean**	**Std. dev**	**Min**	**Max**
5	0	0	0	0	0.125e−2	0.127e−2	0.433e−3	0.438e−2
15	0	0	0	0	0.409e−2	0.311e−2	0.140e−2	0.974e−2
30	0	0	0	0	0.114e−1	0.445e−2	0.614e−2	0.219e−1

Table 6: Test results on size 30

	Percentage relative error				Execution time			
Restarts	**Mean**	**Std. dev**	**Min**	**Max**	**Mean**	**Std. dev**	**Min**	**Max**
5	0	0	0	0	0.127	0.121e−1	0.109	0.149
15	0	0	0	0	0.407	0.616e−1	0.326	0.503
30	0	0	0	0	0.766	0.829e−1	0.669	0.971

Table 7: Test results on size 50

	Percentage relative error				Execution time			
Restarts	**Mean**	**Std. dev**	**Min**	**Max**	**Mean**	**Std. dev**	**Min**	**Max**
5	0	0	0	0	0.139	0.205e−1	0.119	0.179
15	0	0	0	0	0.448	0.791e−1	0.347	0.605
30	0	0	0	0	0.897	0.914e−1	0.784	0.105e1

Table 8: Test results on size 70

	Percentage relative error				Execution time			
Restarts	**Mean**	**Std. dev**	**Min**	**Max**	**Mean**	**Std. dev**	**Min**	**Max**
5	0	0	0	0	0.418	0.469e−1	0.331	0.479
15	0	0	0	0	0.121e1	0.607e−1	0.109e1	0.130e1
30	0	0	0	0	0.241e1	0.109	0.226e1	0.264e1

Table 9: Test results on size 80

	Percentage relative error				Execution time			
Restarts	**Mean**	**Std. dev**	**Min**	**Max**	**Mean**	**Std. dev**	**Min**	**Max**
5	0.192	0.236	0	0.481	0.617	0.167	0.427	0.108e1
15	0	0	0	0	0.183e1	0.178	0.148e1	0.213e1
30	0	0	0	0	0.374e1	0.291	0.332e1	0.420e1

Table 10: Test results on size 90

	Percentage relative error				Execution time			
Restarts	**Mean**	**Std. dev**	**Min**	**Max**	**Mean**	**Std. dev**	**Min**	**Max**
5	0.308	0.344	0	0.880	0.794	0.965e−1	0.661	0.959
15	0.880e−1	0.176	0	0.440	0.231e1	0.127	0.214e1	0.259e1
30	0.880e−1	0.176	0	0.440	0.453e1	0.172	0.419e1	0.485e1

Table 11: Test results on size 100

	Percentage relative error				Execution time			
Restarts	**Mean**	**Std. dev**	**Min**	**Max**	**Mean**	**Std. dev**	**Min**	**Max**
5	0.320	0.209	0.128	0.769	0.950	0.160	0.689	0.125e1
15	0.128	0.152	0	0.513	0.277e1	0.237	0.246e1	0.337e1
30	0.769e−1	0.628e−1	0	0.128	0.577e1	0.526	0.518e1	0.722e1

Table 12: Test results on size 110

	Percentage relative error				Execution time			
Restarts	**Mean**	**Std. dev**	**Min**	**Max**	**Mean**	**Std. dev**	**Min**	**Max**
5	0.484	0.254	0	0.806	0.129e1	0.145	0.106e1	0.155e1
15	0.202	0.197	0	0.504	0.418e1	0.390	0.360e1	0.514e1
30	0.806e−1	0.134	0	0.403	0.831e1	0.604	0.740e1	0.929e1

Table 13: Test execution times of the exact method over 5 runs

Problem size	Mean time (s)	Std. dev	Min	Max
10	0.220e−1	0.150e−1	0.556e−2	0.466e−1
20	0.248	0.404e−1	0.188	0.306
30	0.118e1	0.145	0.969	0.138e1
40	0.225e1	0.126e1	0.114e1	0.466e1
50	0.973e1	0.677e1	0.301e1	0.208e2
60	0.103e2	0.412e1	0.432e1	0.149e2
70	0.252e2	0.164e2	0.507e1	0.542e2
80	0.594e2	0.472e2	0.202e2	0.133e3
90	0.148e3	0.129e3	0.319e2	0.400e3
100	0.165e3	0.587e2	0.669e2	0.249e3
110	0.222e3	0.517e2	0.130e3	0.280e3

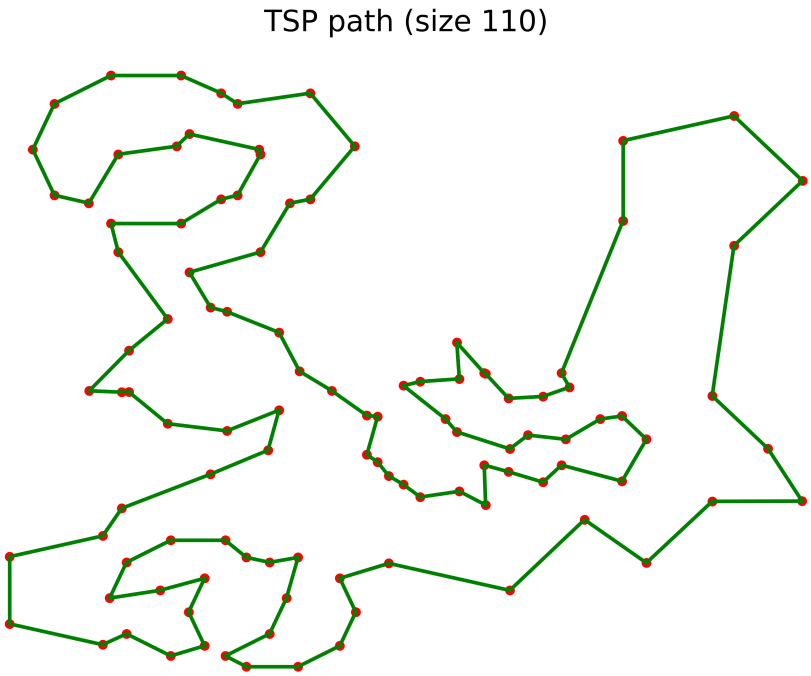


Figure 5: Heuristic solution of problem with size 110 (optimal)

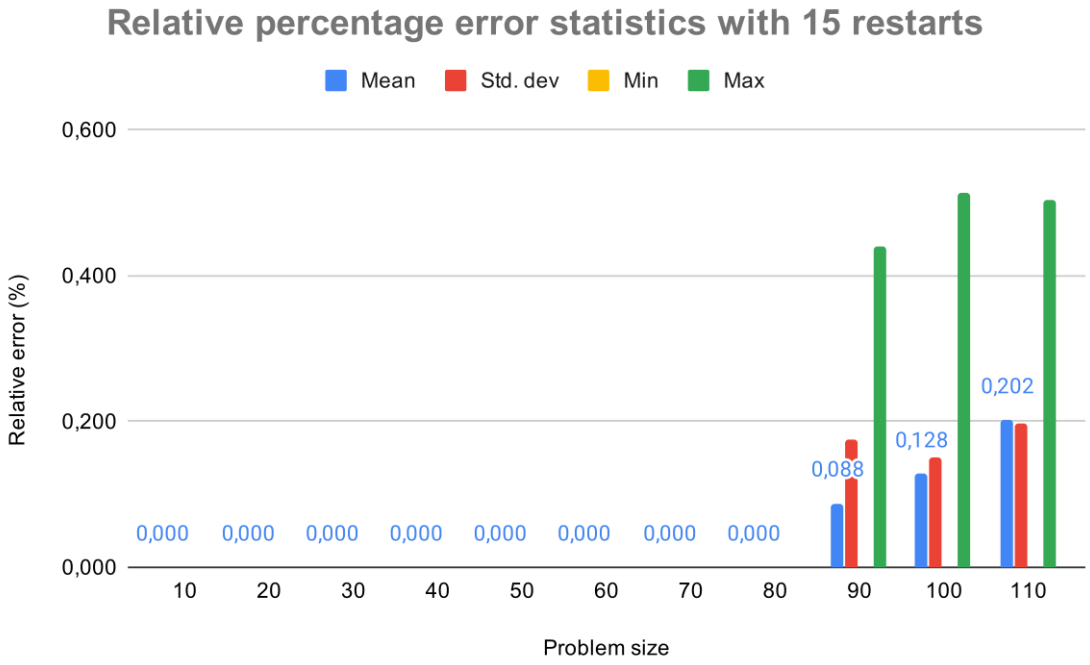


Figure 6: Error statistics over 10 runs with 15 restarts

Figure 7: Execution time statistics over 10 runs with 15 restarts

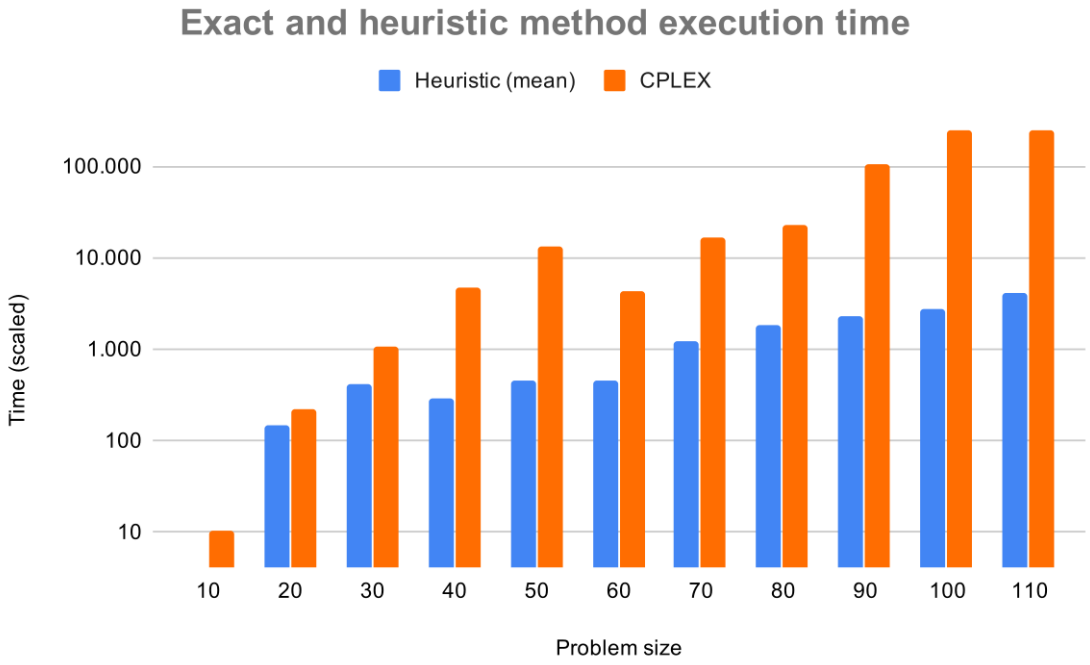


Figure 8: Comparison of execution time of both methods ($\times$ 1000, logarithmic scale)

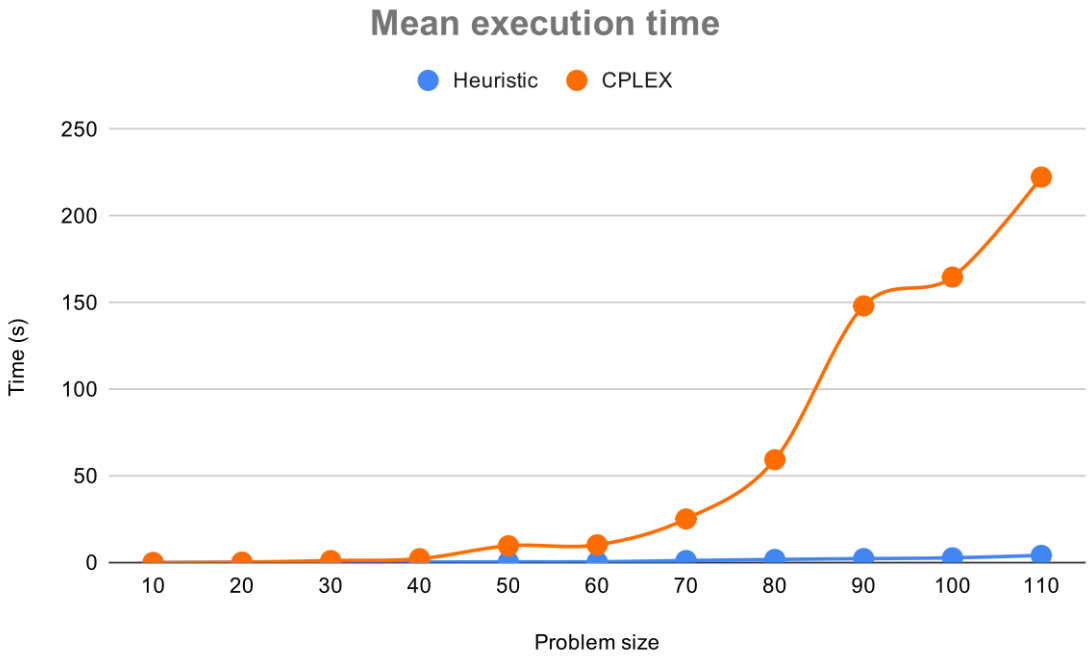


Figure 9: Mean execution time of both algorithms. The heuristic average is computed with 15 restarts and 10 iterations on the same instance, while the CPLEX time is averaged over 5 iterations on different instances

5.2 Tests on bigger instances

Table 14 lists the results of testing the heuristic method with some bigger instances taken from TSPLIB. Extensive tests would require much time, so these results come from a single execution of the heuristic with the same parameters used for previous tests, but with 30 restarts instead of 15. For problem `d657` parameter `backtracking_threshold` was lowered to 2 to reduce execution time. The produced solution paths are shown in figures 10, 11 and 12.

Table 14: Results on TSPLIB problems

Problem	Size	Optimal value	Heuristic value	Relative error (%)	Time (s)
d198	198	15780	15852.9	0.462	266
d493	493	35002	35428.1	1.217	3638
d657 *	657	48912	49892.1	2.004	2032

*`backtracking_threshold = 2`

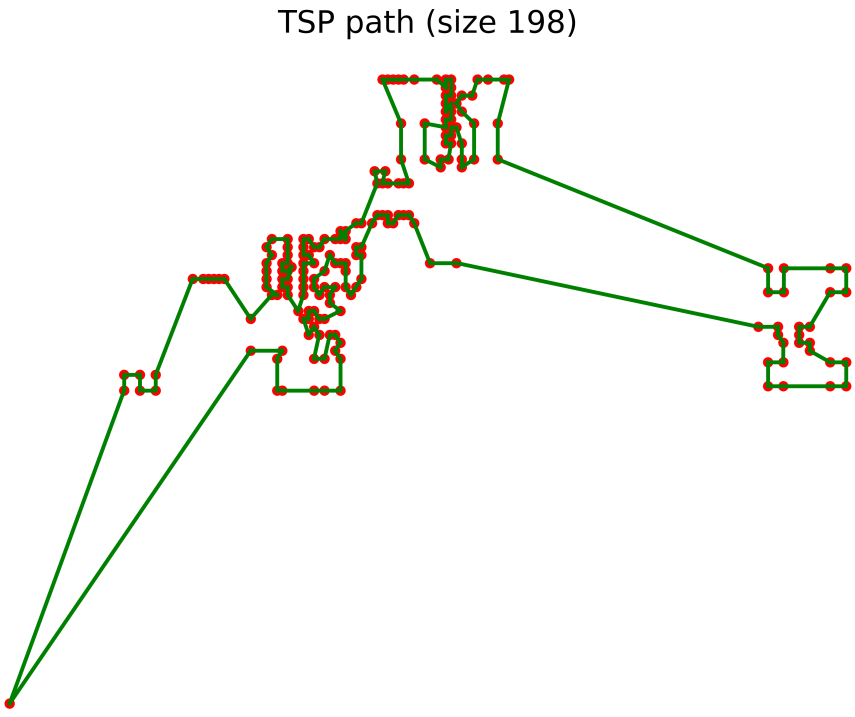


Figure 10: Heuristic solution for d198

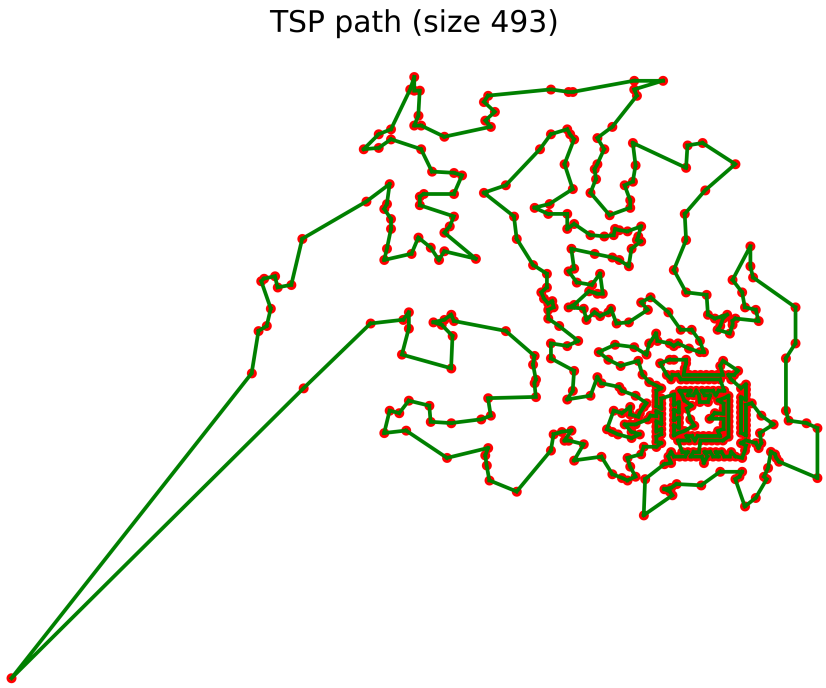


Figure 11: Heuristic solution for d493

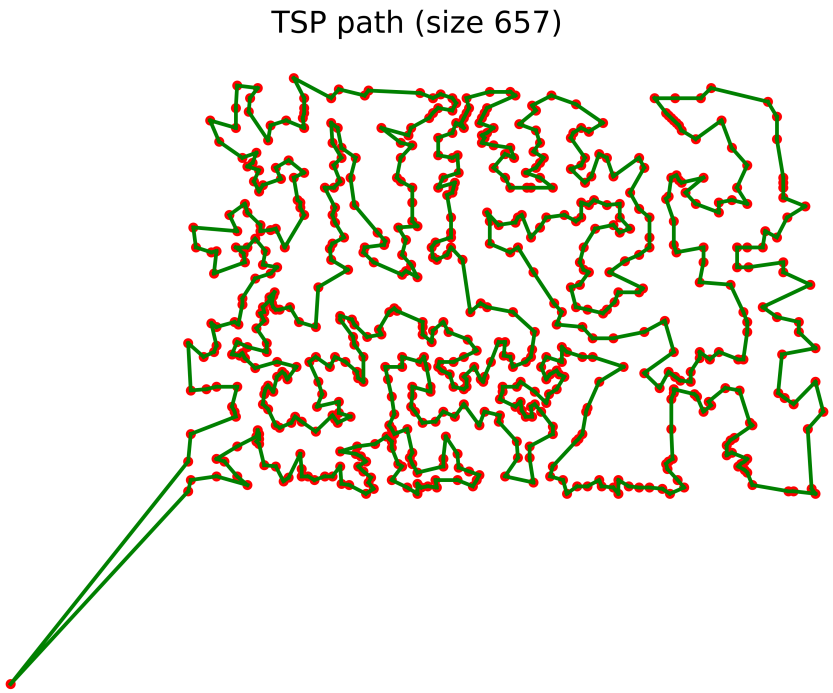


Figure 12: Heuristic solution for d657

6 Conclusions

The implemented heuristic can be expected to find very good (if not optimal) solutions for problem of small size and outperforms the exact algorithm in every conducted test. For problem of size greater than 80 the gap in execution time between the two algorithms becomes very large, 1 minute on average for the exact method and a few seconds for the heuristic that will likely find the optimal solution anyway.
Tests on TSPLIB drilling problems suggests encouraging performances on non trivial problems, even though the runtime grows, and some parameters may be adjusted to favour execution time at the expense of solution quality. For the optimization of the drilling paths over electric boards, the heuristic method seems the preferable choice even for small boards, since it often finds the optimal solution with a reasonable number of restarts.
Performance on instances with thousands of holes remains to be tested. Literature suggests that by refining this algorithm further, good results could be achieved.

6.1 Possible improvements

The Lin-Keringhan local search algorithm received many enhancements and adaptations since its original publication that can be found in literature, so many of these ideas could be exploited to improve the hereby described algorithm.
A first improvement could relate to the tour representation, since there are better data structures to represent an Hamiltonian cycle. For example [Glo96] proposed *ejection chains*, structures that would allow for a fast (constant time) lookup when deciding if breaking a new edge allows a feasible tour or not. This would improve the efficiency and efficacy of the neighbourhood function and thus of the entire algorithm.
Another possible improvement was suggested in [LK73]. In the original implementation of the Lin-Kernighan local search, once a local optima was found, some effort was done to improve the solution further, by applying some non sequential exchanges, like the one shown in fig. 2. It was reported that this methods considerably improved the solutions in some cases, while in other situations it provided little benefit.
Moreover, some refinements could be applied in case a duplicate solution is encountered, as already suggested in section 4.4.2, and the intensification strategy could be refined by using a probabilistic approach to give some more flexibility.
Finally a speed-up could be achieved by exploiting multithreading to launch more executions concurrently. At the moment, every restart shares with the previous executions the set of solutions found, in order to avoid converging to the same solution again, so parallelization is not feasible, but this strategy could be reviewed.

References

[Glo96] Fred Glover. Ejection chains, reference structures and alternating path methods for traveling salesman problems. *Discrete Applied Mathematics*, 65(1):223 – 253, 1996. First International Colloquium on Graphs and Optimization. (Cited on page 28.)

[Hel00] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106 – 130, 2000. (Cited on pages 8 and 9.)

[LK73] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973. (Cited on pages 4, 8, and 28.)

[Mah17] Arthur Mahéo. Implementing lin-kernighan in python. https://arthur.maheo.net/implementing-lin-kernighan-in-python/, 2017. [Online; accessed 18-February-2020]. (Cited on page 4.)

[Tsp95] Best known solutions for symmetric tsps. http://elib.zib.de/pub/mp-testdata/tsp/tsplib/stsp-sol.html, 1995. [Online; accessed 18-February-2020]. (Cited on page 20.)