Project Report
Methods and Models for Combinatorial Optimization

Elia Scandaletti - 2087934

December 23, 2024

1 Introduction

This project consists in the development and comparison of two methods to solve the Traveling Salesman Problem in a specific domain, namely the drilling of holes in electric panel boards. In particular, the problem consists in minimizing the time the drill needs to prepare each board. Since the time it takes to drill a single hole is assumed to be constant, the problem is equivalent to minimizing the time needed to move the drill through each hole position.

The first method is based on a mixed integer linear programming formulation of the problem and always yield an optimal solution. The second approach, instead, uses an heuristic method which may result in suboptimal solutions. On the other hand, it should require less computational resources, such as time, threads and memory.

The first method uses the CPLEX library, whereas the second one leverages tabu search. Both of them are implemented using C++.

Notation In the rest of the report we will be using the following symbol with the meaning described below:

N: the nodes in the graph representation of the problem, i.e. the set of holes on the board.

A: the arcs in the graph representation of the problem, i.e. the possible moves between two holes.

c_{ij}: the cost of traveling over an arc, i.e. the time the drill takes to move from hole i to hole j.

0: an arbitrary node from which the traveler starts, i.e. the first hole to be drilled.

Domain Characteristics and Assumptions The domain of the problem being restricted to the movement of a drill on electric panel boards allows to assume some realistic characteristics of the problem:

- holes are aligned in a grid, likely in rectangular shapes;
- the number of holes on each board is between 60 and 250.

A further assumption is that the drill can move at constant speed in any direction. We therefore assume that the time needed to move the drill from a hole to the next one is proportional to the linear distance between the two.

2 Exact Method

2.1 MILP Formulation

The exact method uses the following mathematical formulation of the problem.

$$
\begin{aligned}
\min \; & \sum_{i,j:(i,j)\in A} c_{ij} y_{ij} & & & (1)\\
s.t. \; & \sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j)\in A, j\neq 0} x_{kj} & = 1 & \quad \forall k \in N & (2)\\
& \sum_{j:(i,j)\in A} y_{ij} & = 1 & \quad \forall i \in N & (3)\\
& \sum_{i:(i,j)\in A} y_{ij} & = 1 & \quad \forall j \in N & (4)\\
& x_{ij} & \leq (|N|-1) y_{ij} & \quad \forall (i,j) \in A, j \neq 0 & (5)\\
& x_{ij} \in \mathbb{R} & & \quad \forall (i,j) \in A, j \neq 0 & (6)\\
& y_{ij} \in \{0,1\} & & \quad \forall (i,j) \in A & (7)
\end{aligned}
$$

The idea behind this formulation consider the TSP as a network flow problem.

There are two sets of variables. The real x_{ij} variables indicate the amount of flow from node i to node j, while the binary y_{ij} variables indicate whether there is any flow between the noe i and j.

The formulation assumes there is a flow of amount $(|N| - 1)$ coming out of the starting node 0. Then it constraints each node to:

- consume one unit of flow (**??**);
- forward the flow only to one node (**??**);
- receive the flow only to one node (**??**).

Finally, the constraint (**??**) ensures that if there is flow from node i to node j, then $y_{ij} = 1$. Th fact that if there is no flow from node i to node j, then $y_{ij} = 0$ is ensured by optimality.

This formulation corresponds to the one presented in [**gavish1978travelling**].

3 Heuristic Method

3.1 Algorithm Design

The chosen algorithm uses the tabu search meta-heuristic. Each component of the algorithm will be discussed in the following paragraphs.

Initial Solution To generate the initial solution the farthest node insertion heuristic has been used. This heuristic has been implemented in its deterministic flavour using the two farthest nodes as a starting loop.

2.5-opt Moves This type of move, described in [**johnson1997traveling**], is used to generate the neighborhood and consists in either a 2-opt move or a reposition move. A 2-opt move consists in inverting a sub-tour, while a reposition move consists in moving a single node to a different position in the tour. At each iteration there are $O(n^2)$ possible 2.5-opt moves.

Move Evaluation This type of move allows for incremental evaluation in constant time. In particular, for a 2-opt move that invert the sub-tour i-j, we have

$$\Delta C_{\text{2-opt}} = c_{p(i),j} + c_{i,s(j)} - c_{p(i),i} - c_{j,s(j)}$$

Instead, for a reposition move that shifts the node i just after the node j, assuming that $j \neq i$ and j is not immediately preceding i, we have

$$\Delta C_{\text{rep}} = c_{p(i),s(i)} - c_{p(i),i} - c_{i,s(i)} - c_{j,s(j)} + c_{j,i} + c_{i,s(j)}$$

Where $p(i)$ is the node preceding i in the current tour and $s(i)$ is the node following i in the current tour.

Solution Representation The solution is represented as a sequence of nodes in order to easily apply 2.5-opt moves. In the sequence each node appears exactly once.

In the solution, the traveler starts the tour from the first node and visits each node in the order they appear in the vector. After visiting the last node, the traveler goes back to the first one.

Tabu List The tabu list used in the algorithm is fixed-size and keeps in memory the last 2.5-opt moves performed. While choosing the next 2.5-opt move to perform, the algorithm excludes the ones in the list.

In some cases, though, the tabu list can be by-passed. This happens when a move which is tabu leads to a solution which is the best found so far.

Exploring Strategy The algorithms performs a tabu search using the steepest descent criteria on the 2.5-opt neighborhood. If the tabu search cannot find an improving solution for a given number of iterations, then the algorithm moves to the worst neighbor of the current solution and resume the tabu search from there.

Stopping Criteria The algorithms halts when one of the following conditions is met:

- the maximum number of non increasing iterations is reached;
- the maximum number of iterations is reached.

Configuration of the Algorithm The following parameters of the algorithm that can be configured:

- **Size of tabu list** This is the maximum number of moves stored in the tabu list. If this number is too high:
  - it takes up more memory;
  - it may exclude too many moves, including the ones that may lead to a (later) improvement.

  On the other hand, if it is too low, the risk of moving around a local minimum without escaping from it increases.

  Furthermore, the tabu list helps not to fall again in the previous local minimum after the diversification steps. Therefore, a short tabu list is less effective in diversification and may lead to an overall worse solution.
- **Maximum number of non-improving tabu iterations** This parameter determines how many non-improving 2.5-opt moves can be done before performing a diversification step. If it is too high, the risk of moving around the same local minimum without finding a better solution for a long time is high. On the other hand, if it is too low, the algorithm could miss the opportunity to escape from a local minimum without resorting to differentiation.
- **Maximum number of non-improving iterations** This parameter determines how many non-improving moves, including both 2.5-opt moves and diversification steps, can be performed before the algorithm halts. The higher this value is, the higher the risk of performing useless iterations is, where useless means that they do not contribute to the overall solution. On the other hand, if it is too low, the algorithm could miss the opportunity of finding better solutions.
- **Maximum number of iterations** This is the maximum number of moves that the algorithm can perform. If it is too low, the algorithm

may stop before reaching any good solution. If it is too high, the algorithm may run for a long time while yielding small or no improvement at all.

The risk of a high maximum number of iterations may be mitigated by reducing the maximum number of non-improving iterations.

4 Compilation and Execution

4.1 Compilation

To compile the programs, it is sufficient to execute the command `make` while in the `src/` directory. The compilation will result in two executable files in the same directory: `exact` and `tabu`.

4.2 Execution

The `exact` executable takes one argument which is the relative path of the input file. The program apply the exact method described in Section **??** to the input.

The `tabu` executable takes five arguments which are:

- relative path of the input file;
- the size of the tabu list;
- the maximum number of non-improving tabu iterations;
- the maximum number of non-improving iterations;
- the maximum number of iterations.

The program apply the heuristic method described in Section **??** to the input file, using the parameters specified in the arguments.

4.3 Input format

The input files are plain text files which contains an integer number and a matrix of floating point numbers.

The first number is interpreted as an integer and represent the size of the problem. Let's call it n. The following n^2 numbers are interpreted as floating point numbers and represent the cost matrix of the problem.

Usually, for ease of reading, the first number is on its own line, while the matrix is written on n lines, each containing n numbers. This is not enforced, though, since spaces, tabs and new lines are ignored.

5 Implementation

In this section, we first describe the classes used in both the methods, looking at their public interfaces. Then we focus on the classes representing the solvers used in each method, diving into the details of the algorithms.

5.1 Common Interfaces and Classes

The following classes are abstract and represent common interfaces used in both the methods.

Instance This class represents a TSP instance.
It provides two virtual methods:

- `n()`: returns the number of nodes in the instance;
- `cost(i, j)`: returns the cost of the arc from vertex i to vertex j.

Solution This class represents a solution for a TSP problem.
It provides two virtual methods:

- `length()`: returns the number of nodes in the solution;
- `evaluate(tsp)`: returns the cost of the solution when applied to the instance represented by `tsp`.

The following two classes implement the previous interfaces:

Matrix This class implements the `Instance` interface. It stores the problem it represents as a matrix of edge costs.
This class has one constructor:

- `Matrix(filename)`: reads the cost matrix from the file `filename`.

It adds to the interface the following method:

- `read(filename)`: reads the cost matrix from the file `filename`.

Path This class implements the `Solution` interface. It stores the problem it represents as a list of nodes, in the same order as they are visited by the traveler.
This class has three constructors:

- `Path(tsp)`: constructs a standard solution which visits the node of the `tsp` instance using the order of their label;
- `Path(sol)`: default copy constructor;
- `Path(seq)`: construct the solution from the vector `seq`.

Within `Path`, three nested classes are declared:

- `opt2`: represents a 2-opt move;
- `reposition`: represents a reposition move;
- `opt2_5`: represents a 2.5-opt move.

Each of these three classes implement the equivalence operator. Additionally, the `opt2_5` class implements the overloaded static method `from` that converts `opt2` and `reposition` moves in `opt2_5` moves.

`Path` also implements the following methods:

- `get_nth(n)`: returns the node in the n-th position of the tour;
- `evaluate_move(tsp, m)`: incrementally evaluate the effect of applying move `m` on itself, when considering the instance `tsp`;
- `apply_move(m)`: apply move `m` on itself;
- `randomize()`: static method that returns a random `Path`.

The `evaluate_move` and `apply_move` are overloaded for all the three types of moves.

5.2 MILP Formulation

The exact method required one additional class `Flow`.

The class has the following fields:

- `status`: is the CPLEX solver status;
- `env`: the CPLEX environment;
- `prob`: the CPLEX problem;
- `x_idx`: a map from each arc to the column of the corresponding x variable, such that x_{ij} is stored in the column `x_idx[i][j]` of `prob`;
- `y_idx`: the same as `x_idx` except that it refers to the y variables;
- `cost`: matrix that stores the cost of each edge, such that `cost[i][j]` corresponds to c_{ij}.

It has only one constructor, `Flow(tsp)`, which takes in input an instance and setup the fields described above. This is done in three steps: firstly `cost`, `env` and `prob` are initialized, then the columns are generated one by one, lastly the rows are generated one by one. At the end of the second step also `x_idx` and `y_idx` are setup.

After the problem is setup, it can be solved. This is done by invoking the method `solve()` which firstly solve the problem using the CPLEX solver, then builds the corresponding `Path` by looking at the values of the y variables.

The last method of the `Flow` class is `evaluate()`. This methods return the value of the objective function, therefore it must be called after `solve()`.

5.3 Tabu Search

This method required two classes to be implemented. The first one is the `FarthestInsertion` which implements a single method that solve the problem using the farthest insertion heuristic. The second one is `Tabu`.

The class `Tabu` implements only the method `solve` which takes several parameters that configure the algorithm. The parameters are:

- `tsp`: the problem to be solved;
- `best_sol`: the best solution known so far;
- `tabu_size`: the size of tabu list;
- `max_non_imp_local`: the maximum number of non-improving tabu iterations;
- `max_non_imp_global`: the maximum number of non-improving iterations;
- `max_iter`: the maximum number of iterations;
- `debug`: an optional output stream useful for debug, at every step the current state is output to this stream.

6 Experimental Results

6.1 Benchmark

The benchmark for the experiments is composed of 200 problems. These problems have a size ranging from 60 to 250, every 10. For each dimension, there are 10 instances.

Instances Generation The instances have been generated using a method based on the domain assumptions exposed in Section **??**. To generate a problem of size n, the generation method follows these steps:

1. Create a $n/2 \times n/2$ grid;
2. while the nodes are less than n:
   (a) choose four elements of the grid in a rectangular shape;
   (b) add the corresponding nodes to the instance;
3. if there are more nodes than needed, remove random nodes until there is the right amount.

The code used to generate the instances can be found in the file `generate.py`.

All the instances can be found in the `data/` folder. Each one is saved in a file named `tsp`n_X, where n is the size of the problem and X is a letter between A and J (included).

6.2 Execution

Tabu Configurations The tabu algorithm has been executed on each instance using 560 different configurations. Let n be the size of the instance, t the size of the tabu list, i the maximum number of non increasing tabu iterations, g the maximum number of non increasing iterations and m the maximum number of iterations. Then

$$\begin{aligned} t &= \frac{n}{10} + \frac{n}{10}k, && k = 0 \dots 9 \\ i &= 20 + 30k, && k = 0 \dots 7 \\ g &= 2i + \frac{i}{2}k, && k = 0 \dots 6 \\ m &= 10g \end{aligned}$$

Exact Method Configuration The exact method has been executed once on each instance using a 30 minutes timeout.

Results For each instance X, the results have been stored in two files:

- `results/exact/`X`.res`: contains the output of the execution of the exact method;
- `results/tabu/`X`.res`: contains the outputs of the heuristic method, each line is the output obtained using a different configuration, as described in Section **??**.

6.3 Results analysis

The results for each instance have been stored in the file `results/tabu/`X`.res`, where X is the name of the instance. Within each file, each line is the output of a run of the tabu algorithm.

The analysis of the results is focused on two aspects:

- finding the best configuration, given the size of the instance;
- comparing the exact and the heuristic methods in terms of effectiveness and efficiency.

The code used to perform the analysis in this section can be found in the file `analysis.ipynb`.

Size of Tabu List In order to find out the best size of the tabu list for each size of the problem, the following steps were taken:

1. for each instance, for each tabu size, list the values of all the solutions found;

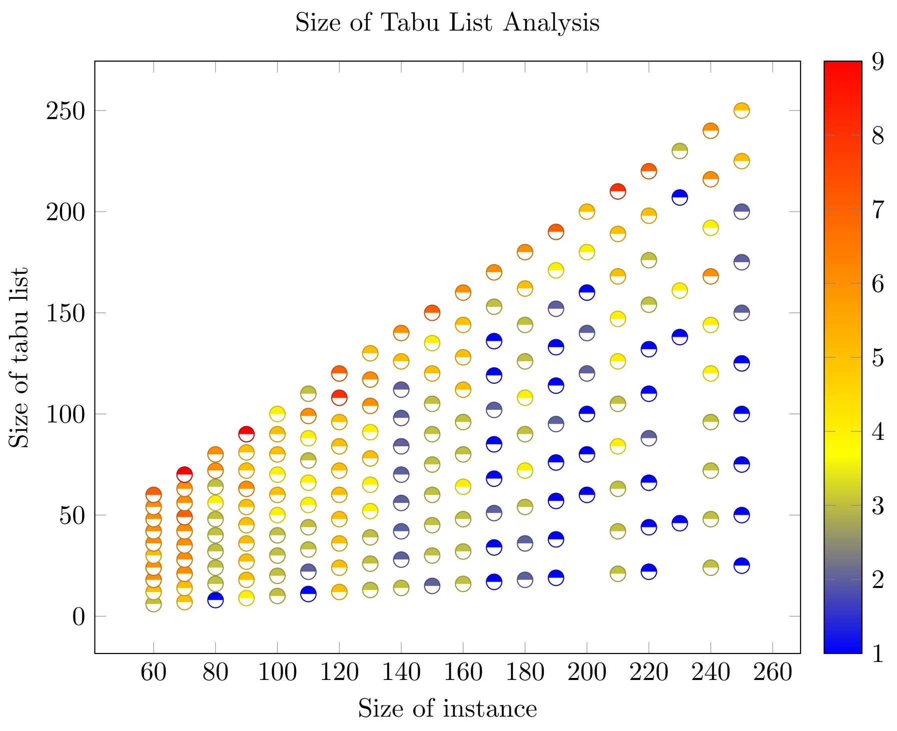

Figure 1: This graph compares the size of instances with the size of the tabu list used. The color of each point determines in how many instances that tabu size lead in average to the best solution for the problem.

2. for each instance, for each tabu size, calculate the average of the values found;
3. for each instance, find which tabu size(s) lead to the best average;
4. for each instance size, determine how many times the best average value has been found using a given tabu size.

In Figure **??**, it's clear that most of the times the best solutions were found with higher sizes of tabu list. This could mean two things: either the bigger the tabu list, the better the solution or the experiments should be run with greater values for the tabu list. Since the first hypothesis is clearly absurd, then using a tabu list with the same size of the problem is the best choice among the proposed configurations.

Effectiveness of the Diversification Strategy In this paragraph, we will use "nit" as an abbreviation of "maximum non increasing tabu iterations" and "nig" as an abbreviation of "maximum non increasing iterations" in order to shortly refer to the two parameters.

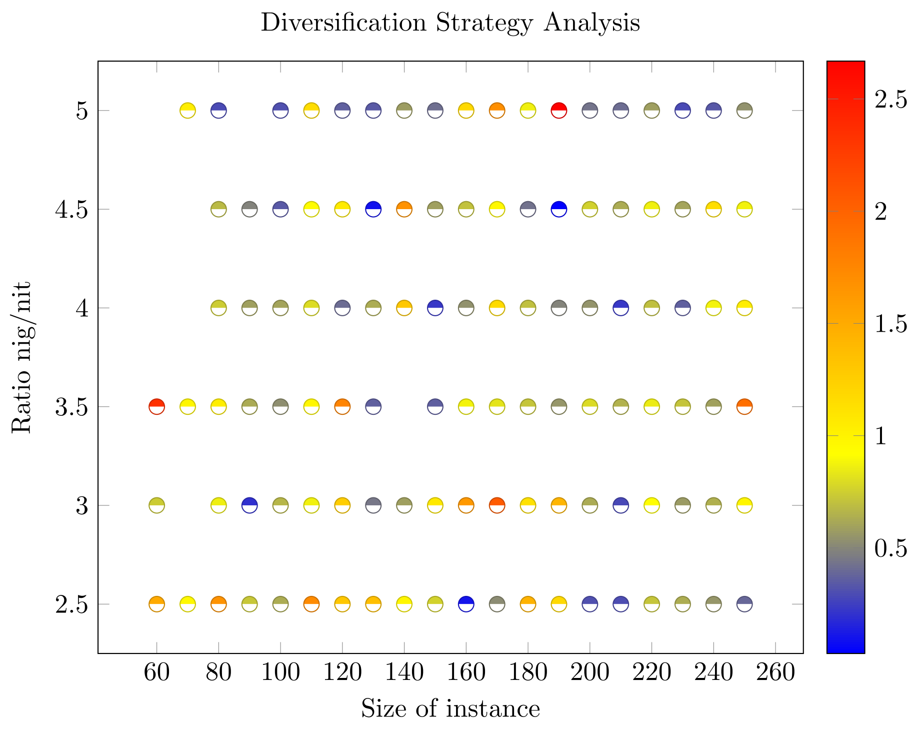

Figure 2: This graph compares the size of instances with the ratio between non improving tabu iterations and non improving iterations. The color of each point determines the average improvement obtained using that ratio w.r.t. the immediately smaller one.

In order to find out whether and when the diversification strategy is useful, the following steps were taken:

1. for each instance, for each nit, for each nig, find the best value of the result;
2. for each instance, for each nit, list the nigs that lead to an improvement when compared to smaller nigs and the respective percentile improvement;
3. for each instance, consider the ratio of the improving nigs with the used nit and for each ratio compute the average improvement.

The ratio between nit and nig is relevant, since it determines how many diversification steps can be performed at most.

In Figure **??**, we can see that in average increasing the nig/nit ratio almost always leads to an improvement in the solution. On the other hand, the higher the nig, the longer the execution time, for obvious reasons.

Note that for most problem sizes, the best improvement is when the ratio is 3.5, while for higher or lower values -except for a few lucky or unlucky exceptions- the improvement isn't higher than 1%.

Further experiments should be performed in order to find an upper bound to the nig/nit ratio after which no improvement are achieved.

6.4 Method Comparison

In order to compare the two methods, all the results of the exact methods have been considered, while, for the tabu method, only the configuration with maximal tabu size and nig/nit ratio have been considered. See Section **??** for more details on the configurations.

In Figure **??**, the points on the bottom and on the right represent the exact method results. While the lines in the bottom left of the graph represent the results of the heuristic methods, one color and style per instance size. Each point on each line represent a different configuration.

We can see that for the sizes between 60 and 90 the exact method has been able to find the optimal solution before the timeout. In these cases, of course, the exact solution is better than the heuristic one. On the other hand the exact method is much faster, around 100 times faster. Furthermore, the approximation gap is at most 6% and most of the time way lower.

Clearly, the bigger the size, the longer the execution of both methods is and the bigger the approximation gap is.

Instead, for sizes equal to or greater than 100, the exact method hasn't been able to terminate. Therefore, all these results are on the 1800 vertical line, with different approximations.

Predictably, the bigger the instance, the higher the approximation is, for both methods. It's easy to note, though, that the approximation of the exact method is much worse than the one of the heuristic method.

Note that for the instances bigger than 120, the results of the exact method are worse than the ones found by the Christofides algorithm.

In the second plot, we can the approximation gap of the heuristic method doesn't seem to decrease as time increases, as could be expected. This is due to the fact that time is strictly bound to the maximum number of non increasing iterations (nig), which in the results that we considered is strictly related to the maximum number of non increasing tabu iterations (nit). Increasing the nit doesn't necessarily lead to better solutions, therefore in our set of results an higher nig and a longer time don't necessarily lead to better results.

In conclusion we can say that, unless, we are highly confident that the exact algorithm concludes before the timeout, the heuristic method is approximately two orders of magnitude faster and yields better results.

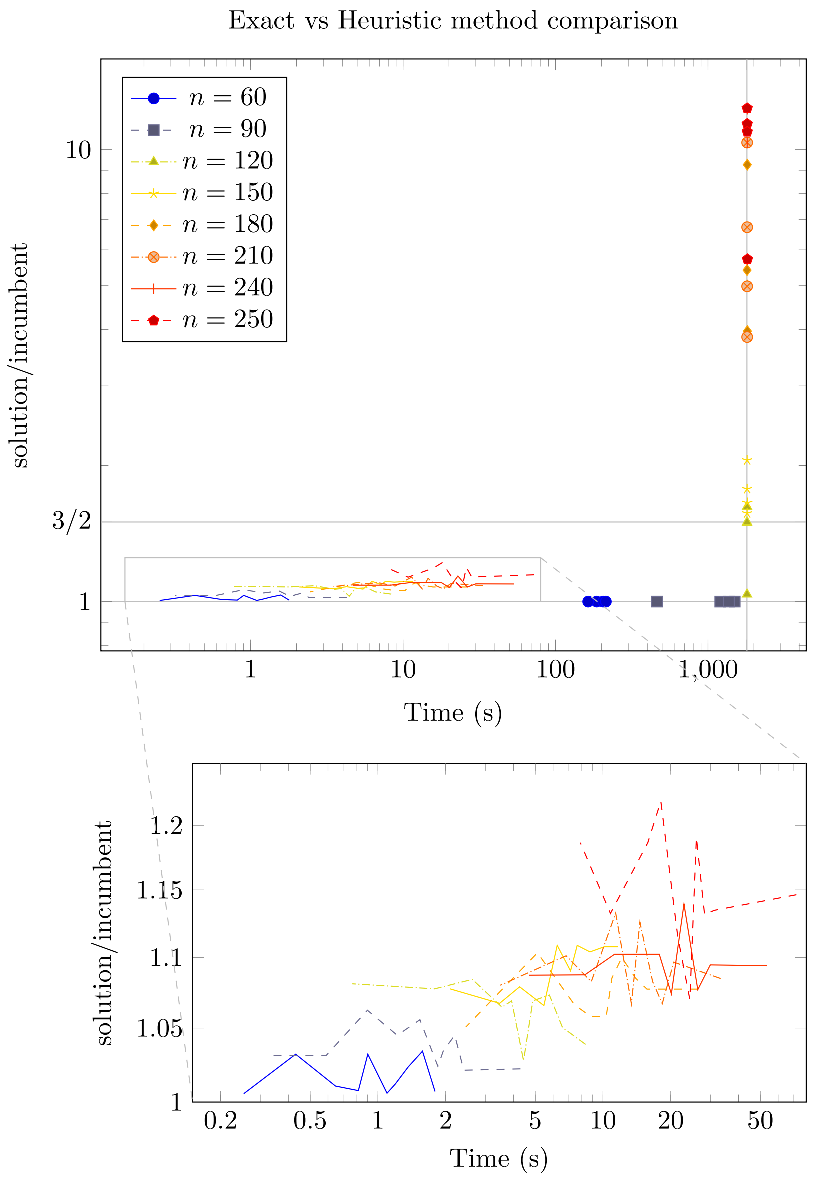

Figure 3: In the first graph, the single points represent the results of the exact method. Their color and shape depends on the size of the instance.
The lines represent the results of the heuristic method. There is one line per size of the instances, as described in the legend.
The bottom graph, is a zoom in on the top graph to better visualize the heuristic results.
Only some instances are shown in order to not overcrowd the plot.