Exact and Genetic Algorithm approaches for the Travelling Salesman Problem

Elena Ferro
ID 2166466
University of Padua
Methods and Models for Combinatorial Optimization
elena.ferro.7@studenti.unipd.it

I. Introduction

This report presents two approaches which were implemented to solve the Travelling Salesman Problem (TSP): an exact method and a genetic algorithm. More specifically, this project positions itself in the context of Printed Circuit Boards (PCBs) production, where the goal is to find the shortest path for a drill to take when boring holes in a board.

I.1 Project structure and usage

The project contains two folders corresponding to the two implemented approaches. Both parts contain an `include` directory for header files, a `samples` directory for input instances, and a `src` directory for the source code. The root directory also contains `random_instance_gen.py`, a script for generating random instances of the problem. Given that solutions are generated differently for the two approaches, each part has its own `plot.py` script to visualize the results.

For the sake of simplicity, these scripts have been written in Python, in order to leverage its plotting libraries and simplify the generation of instances.

Below is the structure of the project:

```
├── part_one
│   ├── include
│   ├── samples
│   ├── src
│   └── plot.py
├── part_two
│   ├── include
│   ├── samples
│   ├── src
│   └── plot.py
└── random_instance_gen.py
```

I.1.1 File formats

TODO: dat and solution files

I.1.2 Running part one

The following commands assume the user is initially positioned in the root directory of the project. The first part can be run with the following commands:

```
cd part_one
make
./main <instance> [formulation] [timeout]
```

Where `<instance>` is the path to the input instance, `formulation` is an optional parameter that can be either `gg` for Gavish and Graves' formulation or `mtz` for Miller, Tucker and Zemlin's formulation. `timeout` is an optional parameter that limits the maximum time in seconds to run the algorithm. If no timeout is specified, the algorithm will run until completion.

After running the algorithm, a `.sol` solution file will be generated in the same directory as the input instance, named as `<instance>_<formulation>.sol`. This file contains the solution to the problem, which can be visualized using the provided `plot.py` script.

Example of usage with a Gavish and Graves' formulation on an instance named `random_10.dat` with a timeout of 60 seconds:

```
./main samples/random_10.dat gg 60
```

The solution will be saved in `samples/random_10_gg.sol`.

I.1.2.1 Plot visualization

To visualize the solution, the `plot.py` script can be used. It takes the following arguments:

```
plot.py [-h] -d DAT [-s SOL] [-o OUTPUT]
```

Where `-d DAT` is the path to the input instance file, `-s SOL` is the path to the solution file (optional), and `-o OUTPUT` is the output file for the plot (optional). If no output file is specified, the plot will be displayed on screen. To visualize the solution generated in the previous step, one would run:

```
py plot.py -d ./samples/random_10.dat -s ./samples/random_10_gg.sol
```

This will produce an image showing the holes in the instance and the path taken by the drill to bore them, as well as the total distance traveled. The image will be saved in the same directory as the input instance, named as `<instance>_tour.png`.

The plot script can also be used to visualize the input instance without a solution file, in which case it will only show the holes in the instance. To achieve this, it's sufficient to run:

```
py plot.py -d ./samples/random_10.dat
```

I.1.3 Running part two

Analogous procedure can be followed for the second part of the project. The commands to run the second part are as follows:

```
cd part_two
make
./main <instance>
```

Note that the second part will already run the genetic algorithm with the tuned parameters, so no additional arguments are needed. The solution will be saved in the same directory as the input instance, named as `<instance>_sol.dat`.

I.1.3.1 Plot visualization

To visualize the solution generated by the genetic algorithm, the `plot.py` script can be used identically to the first part. An example of usage is as follows:

```
py plot.py -d ./samples/random_10.dat -s ./samples/random_10_sol.dat
```

This will produce an image showing the holes in the instance and the path taken by the drill to bore them, as well as the total distance traveled. The image will be saved in the same directory as the input instance, named as `<instance>_tour.png`.

I.2 Problem instances

In order to test the implemented algorithms, a set of random instances was generated. Each consists of a number of holes represented as points in a 2D plane. The distance is computed using the Euclidean distance formula.

The goal is that of generating instances which are representative of real-world scenarios for PCB production. To achieve this, points are generated according to one of the following patterns:

- line: three to five points are aligned in a straight line, with either a uniform or non-uniform random distance between them. The line can be orientated in several directions, including diagonally;
- triangle: three points are randomly placed in a triangular shape, with varying size and orientation. These can also be irregular triangles;
- rectangle: four points are placed in a rectangular shape, with varying size and orientation;
- parallelogram: four points are placed in a parallelogram shape, with varying size, orientation and skewness.

Other than these, a few points are randomly created. During this generation process, a minimum distance between points and patterns is enforced, to avoid overlapping holes. A density parameter controls the sparsity, with a higher density resulting in more closely packed holes. Note that these patterns are, in general, not very visible for smaller instances, but they become more apparent as the number of holes increases. Fig. 1 shows an example of a random instance with 100 holes, with quite evident patterns.

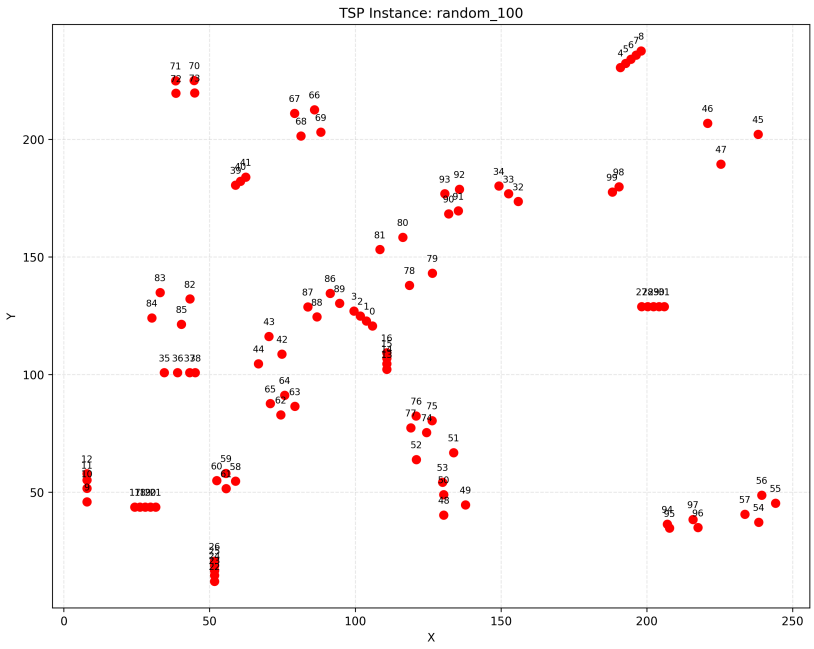


Fig. 1. Example of a random instance with 100 holes.

I.2.1 Script usage

The script used to generate random instances is available in the root directory of the project, under the name `random_instance_gen.py`. It can be run from the command line with the following syntax:

```
python3 random_instance_gen.py -s SIZE [-d DENSITY] -o OUTPUT_DIR
```

For instance, to generate a random instance with 100 holes and a density of 0.5, and save it in the `part_one/samples` directory, one would run:

```
python3 random_instance_gen.py -s 100 -d 0.5 -o part_one/samples
```

II. Exact approach

With regard to the exact approach, two compact formulations were implemented:

- the one proposed by Gavish and Graves (GG) [1], which uses a network flow-based approach to eliminate subtours[1];
- the one introduced by Miller, Tucker, and Zemlin (MTZ) [2], which uses additional variables representing the order or position of each city in the tour; the elimination is achieved by ensuring that these variables maintain a sequential order, thus preventing subtours.

II.1 Implementation details

TODO

II.2 Variables and constraints creation

II.2.1 Maps

TODO

II.2.2 Optimizations

TODO

III. Genetic algorithm approach

TODO introduction

III.1 Implementation details

TODO chromosome

III.1.1 Population initialization

The number of chromosomes in the initial population is determined by the `population_size` parameter and remains constant throughout the algorithm. Two techniques are used in conjunction to initialize the population:

- Random Initialization: generates a random permutation of cities for each chromosome, ensuring that all cities are included in the tour. This method provides a diverse starting point for the algorithm;
- Convex Hull Initialization: involves generating a convex hull, forming an initial partial route using the cities on the boundary of the hull. The remaining interior cities are then inserted into this partial tour one by one. Each of them is inserted at the position that results in the minimum incremental cost,

[1] In the context of the TSP, a subtour is a closed route that only visits a subset of the cities, not all of them, and is not connected to the rest of the route. These are invalid solutions and need to be eliminated to find the optimal tour.

calculated by finding the smallest increase in distance when placing the city between two existing cities in the partial tour.

Through empirical testing, it has been observed that employing the Convex Hull Initialization approach exclusively yielded an initial population suffering from poor diversity. This was primarily due to the fact that this technique tends to produce similar chromosomes, as it starts to create tours based on the convex hull, which is the same for all chromosomes. To mitigate this issue, the Random Initialization technique is used to introduce diversity into the population. The final population is then formed by combining both initialization methods, ensuring a diverse set of chromosomes that can effectively explore the solution space. The ratio of combination between the two is regulated through the `convex_hull_random_init_ratio` parameter.

III.1.2 Genetic Operators

This section describes the implemented operators used in the Genetic Algorithm.

III.1.2.1 Selection

The goal of the selection operator is to choose individuals from the current population to create a new generation. The selection process is based on the fitness of each individual, which is determined by the total distance of the tour represented by the chromosome. This phase should aim at identifying the fittest individuals, however, to avoid premature convergence, it is also important to maintain diversity between the solutions. The following selection schemes have been implemented:

- Linear Ranking Selection: sorts individuals by increasing fitness and assign a rank σ_i to each individual i in the population. The probability of selection is then given by $p_i = \frac{2\sigma_i}{N(N+1)}$, where N is the population size;
- n-Tournament Selection: randomly selects n individuals from the population and chooses the one with the best fitness. The process is repeated until the desired number of individuals is selected. The value of n can be tuned to balance exploration and exploitation.

III.1.2.2 Crossover

The aim of the crossover operator is to combine two parent chromosomes to create offspring that inherit characteristics from both parents. The following crossover methods have been implemented:

- Order Crossover (OX): selects a random subsection from one parent and copies it directly to the offspring. The remaining cities are then filled in the order they appear in the other parent, preserving the relative order of cities outside the copied section;
- Edge Recombination Crossover (ERX): focuses on preserving edges rather than simply city positions. It constructs an offspring by prioritizing edges present in either parent. It works by building a list of neighboring cities for each city from both parents and then iteratively selecting the next city based on which has the fewest available unvisited neighbors [3].

III.1.2.3 Mutation

Mutation is intended to introduce variability into the population by altering the chromosomes of individuals. This helps to maintain genetic diversity and prevent premature convergence. The following mutation methods have been implemented:

- Simple Inversion Mutation: selects a random subsection of the chromosome and reverses the order of the cities within that segment. This operation changes the order of a contiguous block of cities while keeping their relative positions outside the inverted segment intact [3];
- Displacement Mutation: involves selecting a random subtour from the chromosome and then reinserting it at a different, randomly chosen position. The cities outside the displaced subtour remain in their original relative order, and the displaced segment is inserted without altering its internal order.

III.1.2.4 Replacement

In the replacement phase, the new generation of individuals is created by replacing some or all of the individuals in the current population with the offspring generated by the crossover and mutation operators. The following replacement strategies have been implemented:

- Steady State Replacement: replaces the worst-performing individuals in the population with the newly created offspring. This approach ensures that the best individuals are retained while allowing for the introduction of new genetic material;
- Elitism Replacement: keeps few of the best individuals from the current population and replaces the rest with offspring.

III.2 Parameters

Below is a list of the parameters that can be tuned in the Genetic Algorithm implementation, along with their default values and descriptions.

Parameter	**Default**	**Description**
`population_size`	100	The number of chromosomes in the population.
`mutation_rate`	0.01	The probability of mutating a chromosome.
`parents_replacement_rate`	0.8	The proportion of parents that are replaced in the next generation.
`selection_n_parents`	300	The number of parents selected for crossover in each generation.
`selection_tournament_size`	5	The size of the tournament for the n-Tournament Selection method.
`time_limit_seconds`	60	The maximum time allowed for the algorithm to run, in seconds.
`max_non_improving_generations`	100	The maximum number of generations without improvement before stopping the algorithm.
`convex_hull_random_init_ratio`	$(0.1, 0.9)$	The ratio of convex hull initialization to random initialization in the population initialization phase. The first value is the proportion of chromosomes initialized using

Parameter	Default	Description
		the convex hull method, while the second value is the proportion initialized randomly.

III.2.1 Parameter tuning

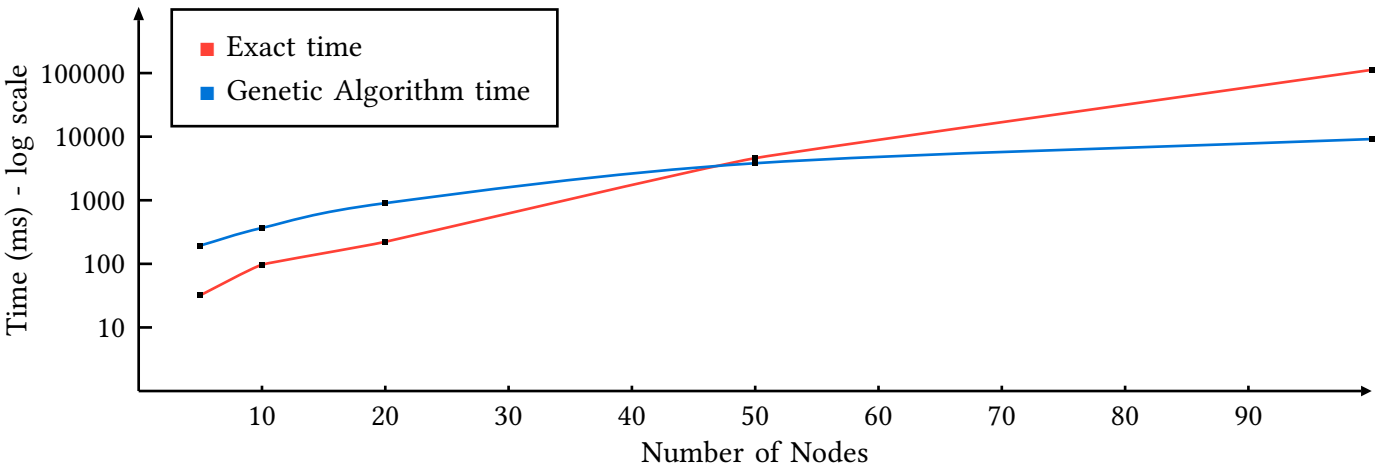
To ease the parameter tuning process, a `GridSearch` class heavily inspired by the one from Python's `scikit-learn` library has been implemented[2]. Given a problem instance, it allows a systematic search over a specified set of parameter, evaluating the performance of the algorithm for each combination. The performance is measured using a fitness function, which, in this case, is the total distance of the tour represented by the chromosome.

This process is repeated for instances of different sizes, so to obtain a more general understanding of the algorithm's performance. Clearly, to obtain meaningful results several runs should be performed on different instance sizes, since the algorithm has stochastic components. The results are then averaged over the different executions, and the best parameters are selected based on the average performance.

TODO: parameter tuning results

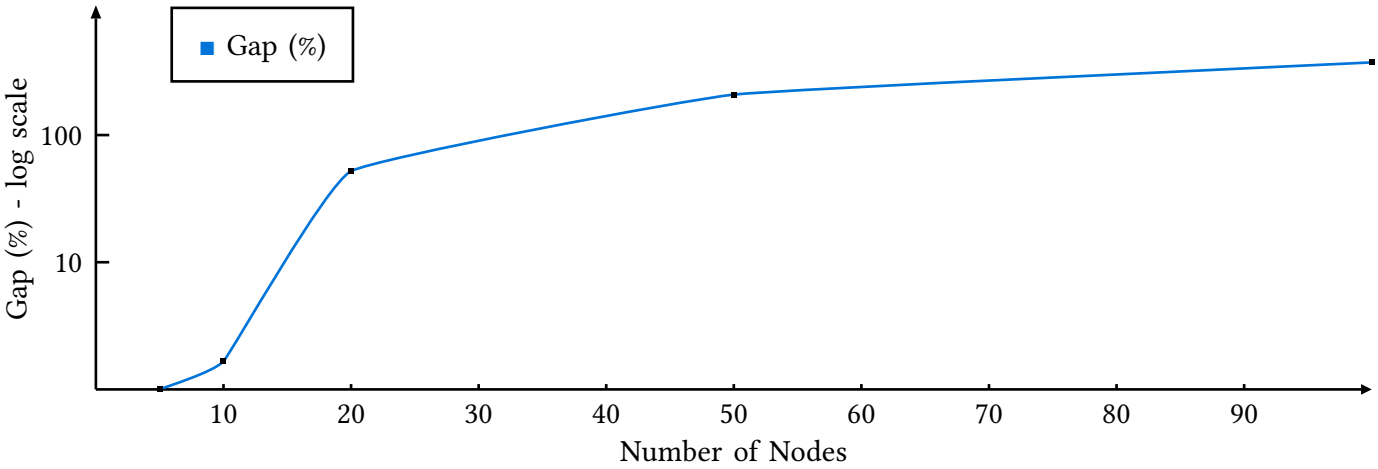
IV. Computational results

Problem name	Exact solution	Genetic solution	Exact time (ms)	Genetic time (ms)	Gap (%)
`random_5`	14.0736	14.0736	32ms	194ms	0.00%
`random_10`	40.2173	40.8873	97ms	367ms	1.67%
`random_20`	140.9538	214.2348	222ms	901ms	51.99%
`random_50`	599.8829	1847.6701	4597ms	3822ms	208.01%
`random_100`	1288.5805	6090.9116	112441ms	9172ms	372.68%



Graph 1. Exact vs Genetic Algorithm solution time

[2]https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html



Graph 2. Optimality gap for Genetic Algorithm

V. Conclusion

References

[1] B. Gavish and S. Graves, "The travelling salesman problem and related problems," Cambridge, MA, 1978.

[2] C. Miller, A. Tucker, and R. Zemlin, "Integer Programming Formulations and Traveling Salesman Problems," *Journal of the Association for Computing Machinery*, 1960.

[3] P. Larrañaga, C. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic, "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators," *Artificial Intelligence Review*.