Exact and Genetic Algorithm approaches for the Travelling Salesman Problem

Elena Ferro
ID 2166466
Computer Science, 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 (GA). 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

The input instances are stored in `.dat` files, which contain as first line the number of holes, followed by the coordinates of each hole in the format `id x y`, where `id` is the incremental identifier of the hole, `x` is the x-coordinate and `y` is the y-coordinate. For example, a file with two holes would look like this:

```
2
0 1 2
1 3 4
```

This file format is employed also for the solution files produced by the GA, for simplicity reasons. In that case, the produced file is named as `<instance>_sol.dat`, where `<instance>` is the name of the input instance file.

For what concerns the solution files produced by the exact algorithm instead, they are stored CPLEX's standard `.sol` format. To avoid cluttering the solution file with unnecessary variables, only the non-zero ones are exported. This is achieved by setting the `CPX_PARAM_WRITELEVEL` parameter to `CPX_WRITELEVEL_NOZEROVARS` in the CPLEX environment.

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 (GG)‘ formulation or `mtz` for Miller, Tucker, and Zemlin (MTZ)'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 GG' 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 GA 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

The reason why another script is used for the second part is that the GA produces a solution in a different format than the exact algorithm. Its usage however, is identical:

```
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.

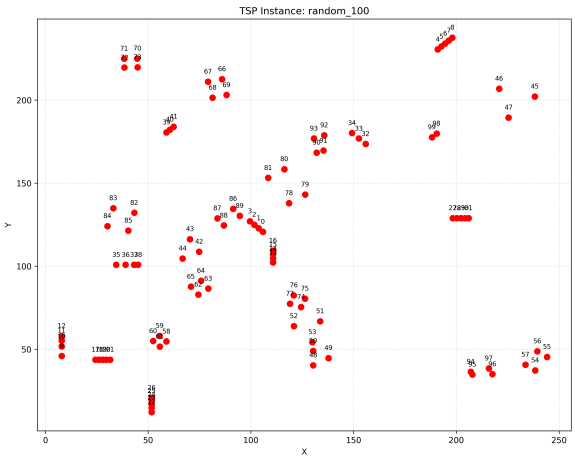


Fig. 1. Example of a random instance with 100 holes.

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.

The problems built according to these two formulations are then fed to the IBM CPLEX 22.11 solver through its C++ API.

II.1 Implementation details

This section provides an overview of some details regarding the exact approach, including certain aspects of the practical implementation.

II.1.1 Graph representation

A `Graph` class is used to represent the problem instances, containing a vector of `Node` objects, each representing a hole in the board. This class provides a convenient way to read input instances from files and store both the coordinates of points and their Euclidean distance.

[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.

II.1.2 Formulations

The `Formulation` class is an abstract base class that defines the interface for the two specific formulations: `GavishGraves` and `MillerTuckerZemlin`. Each formulation implements the methods necessary to create the variables and constraints required to solve the Travelling Salesman Problem (TSP). Furthermore, it contains common logic to setup CPLEX, solve the problem and export the solution.

II.1.3 Time logger

To ease the process of measuring the time taken by the solver to create the model and solve it, a `TimeLogger` class is provided. Once it has been instantiated and started, calling the `tick` method with a message will log the elapsed time since the last tick or the instantiation of the logger. At the end of the process, the `log_total_time` can be called to display the total time taken by the process.

II.1.4 Variables and constraints creation

Particular attention was given to the creation of variables and constraints. This section provides an overview of how the variables and constraints are created, including considerations for performance and memory usage.

II.1.4.1 Maps

Internally, CPLEX considers variables to be all stored in a single vectorial structure; it is therefore not straightforward to associate the variable's position in the vector with the corresponding indexes in the problem, after the variable has been created. To address this, both formulations use matrices to store, for each pair of coordinates (i, j), the variable's index in the vector. This allows for easy access to the variables when creating constraints or retrieving solutions. The two formulations differ in how they store these indices:

- for the GG formulation, two integer matrices are used, holding indices for variables x_{ij} and y_{ij};
- in the MTZ implementation instead, a single integer matrix is used to hold the indices for the variables x_{ij}, and a vector memorizes indices for the order variables u_i.

II.1.4.2 Optimizations

For optimal performance and memory efficiency, both formulations create all variables at once, rather than one by one. This approach minimizes the overhead associated with multiple calls to the CPLEX API, which can be costly in terms of performance. To do so, the formulations leverage the custom implemented `Constraints` and `Variables` support classes, that are iteratively filled with the necessary data and then fed to CPLEX in a single call.

Furthermore, only the minimum number of variables and constraints necessary to solve the problem are created. Assuming N to be the number of nodes in the problem, below is a summary of the number of variables and constraints created by each formulation.

II.1.4.2.1 Gavish and Graves formulation

The GG formulation creates the following variables:

1) y_{ij}: $n(n-1)$ variables, given that they are created for each pair of nodes (i, j) with $i \neq j$;
2) x_{ij}: $(n-1)^2$ variables, given that they are created for each pair of nodes (i, j) with $i \neq j$ starting from $j = 1$, since $j = 0$ corresponds to the depot node.

Therefore, a total of $n(n-1) + (n-1)^2 = (n-1)(2n-1) = 2n^2 - 3n + 1$ variables are created.

For constraints instead, it creates:

1) flow conservation: $(n-1)$ constraints, applied for all nodes except node 0 (depot);
2) outgoing degree: n constraints, one per node ensuring exactly one outgoing edge;
3) incoming degree: n constraints, one per node ensuring exactly one incoming edge;
4) flow-edge coupling: $n(n-1)-(n-1)=(n-1)^2$ constraints, created for all pairs (i,j) where both x_{ij} and y_{ij} variables exist.

So in total, $(n-1)+n+n+(n-1)^2=n(n+1)$ constraints are created.

II.1.4.2.2 Miller, Tucker, and Zemlin formulation

The MTZ formulation creates the following variables:

1) x_{ij}: $n(n-1)$ variables, given that they are created for each pair of nodes (i,j) with $i \neq j$;
2) u_i: $n-1$ variables, given that they are created for each node i except the depot node 0.

In total, the MTZ formulation creates $n(n-1)+(n-1)=n^2-1$ variables.

For what concerns the constraints, it creates:

1) outgoing degree: n constraints, one per node ensuring exactly one outgoing edge;
2) incoming degree: n constraints, one per node ensuring exactly one incoming edge;
3) subtour elimination: $(n-1)^2-(n-1)=(n-1)(n-2)$ constraints, created for all pairs (i,j) where $i \neq j$ and both $i,j \neq 0$;
4) lower bound for order variables: $(n-1)$ constraints, setting $u_i \geq 1$ for each node $i \neq 0$;
5) upper bound for order variables: $(n-1)$ constraints, setting $u_i \leq n-1$ for each node $i \neq 0$.

In total, the MTZ formulation creates $n+n+(n-1)(n-2)+(n-1)+(n-1)=n(n+1)$ constraints.

II.1.4.2.3 Comparison

To understand which implementation is more efficient in terms of memory, the number of variables and constraints created by each formulation can be compared.

- with regard to variables, GG produces $2n^2-3n+1$, while MTZ: n^2-1. Therefore, given that
$$2n^2-3n+1 < n^2-1 \Rightarrow 1 < n < 2 \tag{1}$$
the MTZ formulation is, in practice, producing less variables than GG with equal size of the problem.
- for what concerns constraints instead, they both create $n(n+1)$ and therefore they are equivalent in this regard.

To summarize, both formulation create a polynomial number of variables and constraints, more specifically $O(n^2)$.

II.2 Benchmark results

To evaluate the performance of the exact approach, a set of randomly generated instances of different sizes were solved with increasing time limits. The goal was to determine the largest instance that could be solved within each time limit. Note that to ensure a reliable comparison, more than one instance per size should be taken under consideration. However, due to the time constraints of this project, only one problem per size was used.

In order to simplify the process of reproducing this benchmark, a `benchmark.sh` script has been implemented, and is provided inside the `part_one` folder. It's sufficient to run it without arguments as follows:

```
sh benchmark.sh
```

The results will be stored in the `results` folder as text files.

Below is a summary of the instances used in the benchmark, which are all randomly generated problems with a number of nodes ranging from 5 to 200. The instances are named according to the number of nodes they contain, for example `random5` for a problem with 5 nodes.

`random5`	`random10`	`random20`
`random50`	`random100`	`random200`

The following time limits in seconds were taken under consideration: 1, 2, 5, 10, 20, 30, 60, 120 and 240.

Table I shows the time limit within which each instance was solved; the largest problem, `random200`, was not solved within the time limit of 240 seconds.

Table II instead, shows the largest instance which was solved within each time limit. For example, the largest instance that could be solved within 60 seconds was `random50`, while the largest instance that could be solved within 120 seconds was `random100`. From this table it is quite evident that, as expected, the performance of the exact approach degrades exponentially as the size of the problem increases: solving a 20 nodes problem indeed only took 2 seconds, however for a 50 cities one the elapsed time is 60 seconds, that is, 30 times larger.

TABLE I
INSTANCES BY SOLVING LIMIT

Instance	Time Limit (s)
random_5	1
random_10	1
random_20	1
random_50	5
random_100	120
random_200	N/A

TABLE II
LARGEST SOLVABLE INSTANCE BY TIMELIMIT

Time Limit (s)	Largest solvable instance
1	random_20
2	random_20
5	random_50
10	random_50
20	random_50
30	random_50
60	random_50
120	random_100
240	random_100

III. Genetic Algorithm approach

A C++ implementation of a Genetic Algorithm is proposed as an alternative approach to solve the TSP. The goal is to provide a faster way to find good solutions for large instances, which are too expensive to

solve using exact methods. The GA is a metaheuristic that mimics the process of natural selection, where the fittest individuals from a population of solutions are selected to create new generations.

III.1 Implementation details

III.1.1 Chromosomes

The implemented GA represents solutions as a vector of nodes, where each of them corresponds to a city in the TSP. A `Graph` class based on the one described in Section II.1.1 holds the coordinates of the cities and their Euclidean distances. To ease the process of evaluating the fitness, overriding a part of the path and saving the solution to file, a `Chromosome` class has been implemented as a wrapper around the `Graph` class.

III.1.2 Algorithm overview

To render the GA implementation more flexible and customizable, it has been designed to allow different operators to be plugged in for selection, crossover, mutation and replacement. The `GeneticAlgorithm` class orchestrates the execution of the algorithm, initializing the population and applying the operators in each generation.

The `start` method of the `GeneticAlgorithm` class constitutes the algorithm's entry point. It takes an hyperparameter configuration `HyperParam` as input, and will take care of initializing the population, applying the selection, crossover, mutation and replacement operators. Finally, it returns the best solution found once the stopping criteria are met.

III.1.2.1 A note on population initialization and stopping criteria

Given the necessity to run the algorithm using different techniques for population initialization and stopping criteria at the same time, the `GeneticAlgorithm` class has been designed to allow these to be provided as vectors, instead of single class instances. More specifically:

- the population initialization can be performed using multiple techniques, better described in Section III.1.3, and the final population is formed by combining the results of these techniques;
- the stopping criteria can be defined as a set of different conditions, such as a time limit or a maximum number of generations without improvement. The algorithm will stop when any of these conditions is met.

III.1.2.2 Parameters

Table III contains the list of the parameters that can be tuned in the GA implementation, along with their default values and descriptions.

TABLE III
PARAMETERS OF THE GENETIC ALGORITHM IMPLEMENTATION

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.

Parameter	Default	Description
`selection_tournament_size`	5	The size of the tournament for the n-Tournament Selection method. This is only relevant if the Selection method is set to N-Tournament Selection.
`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 the convex hull method, while the second value is the proportion initialized randomly.

Note that `population_size`, `parents_replacement_rate`, `selection_n_parents`, `time_limit_seconds` and `max_non_improving_generations` are not actually changed during the parameter tuning process, but are set to the default values. They have been included in the list as they are relevant to the algorithm's performance and can be manually adjusted if needed.

III.1.3 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, calculated by finding the smallest increase in distance when placing the city between two existing cities in the partial tour [3].

It has been observed by empirical tests that employing the Convex Hull Initialization method alone can lead to poor diversity in the initial population. This was primarily due to the fact that it starts to create tours based on the convex hull, which is the same for all chromosomes. The randomization introduced by inserting the internal cities in different positions is not sufficient to ensure a diverse population. To mitigate this issue, the Random Initialization technique has been integrated. 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.1.4 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.1.5 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 [4].

III.1.6 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 [4];
- 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.1.7 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.1.8 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.

By running the `GridSearch` class with different configurations for the randomly generated instances, it is possible to identify the best set of parameters for the GA implementation. The chosen configuration consistently performed the best across all instances, with the exception of `random_10`, where `Edge Recombination Crossover` outperformed `Order Crossover`.

The results of this tuning process are summarized below:

- Linear Ranking Selection;
- Order Crossover;
- Displacement Mutation with rate 0.1;
- Elitism Replacement with rate 0.3;
- Convex Hull to Random Initialization Ratio: $(0.2, 0.8)$.

IV. Computational results

In this section, a performance comparison between the exact and GA approaches is carried out.

Since the implemented GA has stochastic components, its results can vary between runs. Therefore, the algorithm was run five times for each instance and parameter combination, and the average results were considered for the comparison, so to obtain a more reliable estimation of the performance. In order to run this comparison, the `benchmark.sh` script can be used, found in the `part_two` folder of the project. It takes care of running the GA with the best parameters found in the previous section, and saving the results in the `results` folder. The script can be executed with the following command:

```
sh benchmark.sh
```

From the results summarized in Table IV, it emerges that the GA is able to provide a sufficiently accurate solution for problems of reduced size - below 10 nodes - although with a much longer computational time than the exact method. This might be accounted for by the fact that the exponential growth of the search space for the TSP is not yet significant for such small instances, allowing the exact method to find the optimal solution in a reasonable time.

For larger instances on the other hand, the GA is able to produce a solution much quicker, but with an excessively large optimality gap, which can be as high as 351% for the largest considered instance (100 nodes).

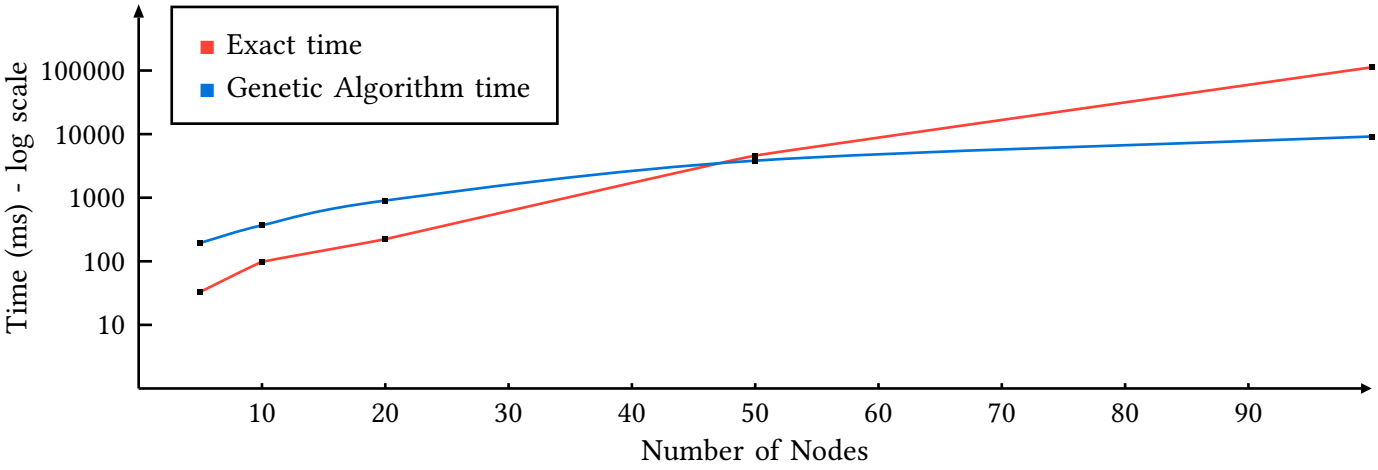
TABLE IV
Results summary for the exact and genetic algorithm approaches to the TSP

Problem name	Exact solution	Genetic solution	Exact time (ms)	Genetic time (ms)	Gap (%)
`random_5`	14.0736	14.0736	32ms	194ms	0.00%

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

Problem name	Exact solution	Genetic solution	Exact time (ms)	Genetic time (ms)	Gap (%)
`random_10`	40.2173	40.2455	97ms	367ms	0.07%
`random_20`	140.9538	212.9320	222ms	901ms	51.07%
`random_50`	599.8829	1678.9200	4597ms	3822ms	179.87%
`random_100`	1288.5805	5817.0549	112441ms	9172ms	351.43%

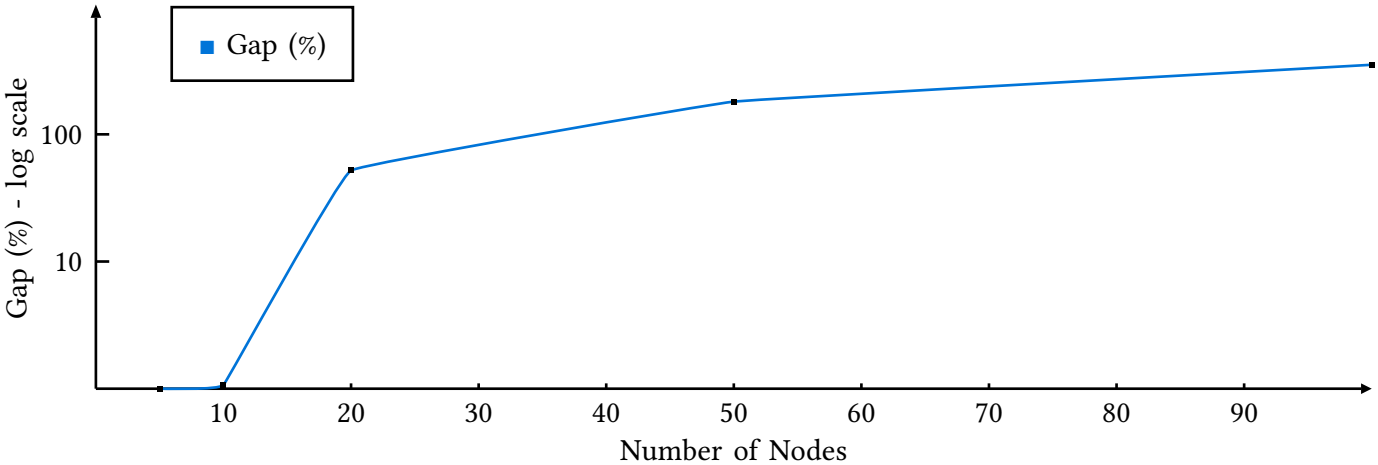
Graph 1 highlights that up to 50 nodes, the exact method produces a solution in a shorter time than the GA, while for larger instances. From the limited extent of this chart, it seems that the time taken by the exact method grows much faster than that of the GA, with the increase in the number of nodes. This is quite sensible given the nature of the TSP.



Graph 1. Exact vs Genetic Algorithm solution time

Graph 2 instead shows the growth of the optimality gap for the GA with respect to the number of nodes. The increase in steeper in the first 20 nodes, then it tends to stabilize and become more linear. This is likely due to the fact that the GA is able to find a good solution for small instances, but as the number of nodes increases, the search space becomes too large and the algorithm struggles to maintain a low gap.

Both the discussed plots are in logarithmic scale, to allow a better visualization of the results, given the large differences in time and gap values.



Graph 2. Optimality gap for Genetic Algorithm

Fig. 2 and Fig. 3 show the tours produced by the exact method and the GA for a random instance of 10 nodes. It can be observed that the GA solution not only isn't optimal, but also contains a subtour starting and ending at node 3.

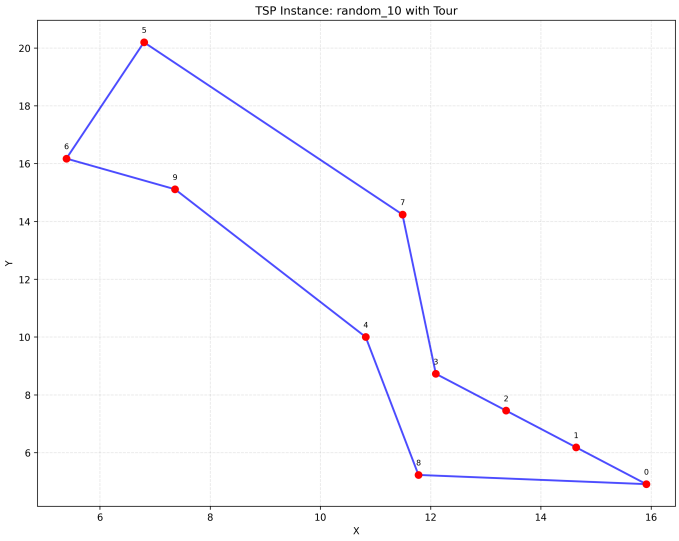


Fig. 2. Exact solution for the random 10 nodes TSP instance

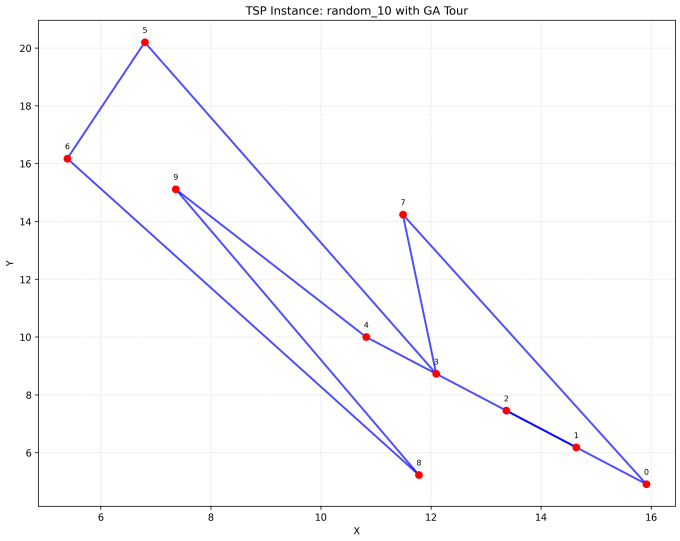


Fig. 3. Genetic Algorithm solution for the random 10 nodes TSP instance

V. Conclusion

The results of the computational experiments show that the GA is able to find good solutions for the TSP exclusively for rather small instances. This might be due to poor diversification in the population, which leads to premature convergence. Possible improvements could include the introduction of Local Search or heuristic based methods to enhance the initially generated population, or to the inclusion of other genetic operators.

Finally, both the benchmarking and the parameter tuning phases could be refined by using a larger set of instances, possibly with different characteristics, to better assess the performance of the algorithm.

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] B. Zhu and W.-F. Xie, "An Improved Genetic Algorithm With Euclidean Geometry and Hybrid Heuristic Crossover for Traveling Salesman Problem," in *2023 IEEE Congress on Evolutionary Computation (CEC)*, 2023, pp. 1–8. doi: 10.1109/CEC53210.2023.10254179.

[4] 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*.