Group 15 Assignment Report

An Evolutionary Algorithm for the Travelling Sales Man problem

Gonzalez Juan Carlos

October 01, 2024

1 Introduction

The Travelling Salesman Problem (TSP) is one of the classic optimization problems in which the salesperson has to determine the shortest path by travelling all the cities in such a way that each city needs to be visited once and the salesman should return to the city that he had started initially. Reducing the overall travel distance or cost of traveling among visiting all the cities is the main aim of the travelling salesman problem. TSP is especially difficult with big datasets because it belongs to the NP-hard class of problems, in which the number of cities has an exponential effect on how long it takes to discover the best solution.[1].

In real-world situations, the TSP solution has great practical significance. The optimization of operations in several sectors, such as manufacturing, logistics, and transportation, depends on finding answers to TSP-like challenges. For example:

- Logistics and Delivery Services: In order to reduce fuel costs and delivery times, businesses such as UPS, FedEx, and Amazon need to figure out the best delivery routes. The TSP assists in determining the most economical route between many locations.
- Manufacturing: The TSP represents the issue of minimizing machine movement between different places, cutting manufacturing time and operating costs in circuit board design or sheet metal drilling.
- Transit and Urban Planning: Effective TSP solutions can minimize time and resource consumption in public transit routes, airline scheduling, and even road repair planning.

Despite its straightforward formulation, the TSP is unable to find the optimal solution for the computation of large graphs. In order to overcome this, we use genetic algorithms. Genetic Algotihms (GA) is one of the approximation techniques that quickly identifies sub-optimal solutions. Due to its ability to replicate the processes of evolution and natural selection, GA serves as a tool for exploring the vast solution space of TSP in situations requiring quick decisions [1].

we will show that although there are alternatives that give exact solutions, one of the advantages of using GA is that it obtains nearly optimal solutions without the high time complexity associated with exact methods, which motivated our use of GA.

2 Problem: The traveling salesman problem (TSP)

Despite how easy it seems, solving the issue of a traveling salesperson is really rather challenging. Mathematicians have searched for a solution to this riddle for over a century. The problem's simplicity and the possibility of seeing potential solutions are its most attractive features [2].

The ability of genetic algorithms to solve the travelling sales person because of their large search spaces. Given that the Traveling Salesperson Problem (TSP) exhibits NP-hardness[2], it is not advisable to employ traditional algorithms for its resolution. Genetic algorithms are well-known for their ability to evade local optima in a population-based search to investigate multiple possibilities sequentially. To maintain a healthy equilibrium between exploring and exploiting the search space, GAs use evolutionary operators such as mutation and crossover to continuously improve solutions. Several types of TSPs, such as dynamic or constraint-based versions, can utilize GAs due to their adaptability and diversity. A potent tool for confronting the computational complexity of the TSP, its heuristic nature allows them to identify high-quality approximation solutions in a fair period of time.

Definition 1 *Let $G(V, E)$ be a complete, simple graph. The set V contains the vertices $1, 2, \ldots, n$ representing cities, and the set of edges E contains all the unordered pairs (i, j) with $i, j \in V$. Each edge (i, j) has weight $w_{ij} \in R$, Representing the cost of travel T from city i to city j.This cost is defined as:*

$$cost(T) = \sum_{(j,k) \in edges(T)} w_{j,k}.$$

If there is a route from s to t when for every $s, t \in V$ there exists a path from s to t using only edges from edges(T), and every $i = 1, \ldots, n$ is present in exactly two edges from T. This definition is based on the details provided in [2, p. 3].

A genetic algorithm develops populations of candidate tours via *mutation* and *crossover* to optimize the overall tour cost, which is the sum of the edge weights. We use this structure to solve the Traveling Salesman Problem (TSP). The halting condition, which ensures convergence, discovers an optimum or nearly optimal solution when the variation in the tour costs of the previous few iterations becomes zero [3].

3 Algorithm: Genetic Algorithm

We followed the traditional approach to implement the Genetic Algorithm Proposed (GAP): initial population, crossover and mutation, fitness calculation, and pruning. Our algorithm is capable of handling negative weights (see 3.2). However, it does not handle no-complete graphs (the user can add missing edges with big weights) and directed graph. We will describe the most important elements in our approach:

- $P = \{T_1, T_2, \ldots, T_{sp}\}$: This is the population in our GA-P, which changes every step and has a fixed size sp, where $T_i \in S$.
- sp: The size of the population.
- $O = \{T_1, T_2, \ldots, T_{\binom{\lfloor ps \cdot cp \rfloor}{2}}\}$: The offspring of the population, where $T_i \in S$. The size of the offspring is $\binom{\lfloor ps \cdot cp \rfloor}{2}$, as explained in Section 3.3.
- $cp \in [0, 1]$: The proportion of the population P that will be reproduced using the crossover-mutation algorithm.
- $mt \in [0, 1]$: The probability of a mutation occurring in a child.
- k: The size of the window used to calculate if a solution has converged.

In this chapter we will describe the code behind each step, modifications that we have done to the code we have developed, and in general our interpretation of the algorithms we have found implementing a version no so different to the one proposed in the literature, but in the next chapter discussing the best hyperparameters model and suggesting a crossover randomization threshold, which we have hypothesized could improve the performance of the algorithm.

3.1 Solution representation

We use the chromosome (T) for the solution representation. We use the term "tour" for T in some parts or the work to make easier understand our proposal, and because the T follows the definition 1.

Definition 2 *Let a tour* $T = (t_1, t_2, ..., t_n)$ *an ordered list with size* $n = |V|$ *where* $t_i \in V$ *and* $t_i \neq t_j \forall t_i, t_j \in T$. *We define a possible solution* T *which represents the* $edges(T) = \{(t_1, t_2), (t_2, t_3), ..., (t_i, t_{i+1}), .., (t_{n-1}, t_n)\} \cup (t_n, t_1)$, *the sets of all the edges which contains the tour in the graph* G.

We can define the set of possible solutions $S = T_0, T_1, T_2, ..., T_m$ *as the set of all possible chromosomes for the graph* G.

3.1.1 Optimal T^* and Suboptimal T^s

Definition 3 *Let* T^* *be a tour in* S. *We say that* T^* *is optimal if*

$$cost(T^*) = \min\{cost(T) \mid T \in S\}.$$

Definition 4 *Let* T^s *be a tour in* S *found in step* n *by applying a Genetic Algorithm to* G. *We say that* T^s *is sub-optimal if*

$$\sigma(T_{n-k}, T_{n-k+1}, \ldots, T_{n-1}, T_n) = 0, \sigma\text{: standard deviation}$$

for a moving analysis window of size k, *where* $T_n = T^s$ *and* T_i *is the tour found at step* i.

3.2 Fitness function $F(T)$

In our approach, something we wanted to consider is negative cost. A negative cost in a TLP could mean that in some paths there is an incentive. For example, a sales problem could mean there are routes that mean earning instead of costing transportation. This can be handled by translating cost before the algorithm application. However, formulating an algorithm that can handle negative costs could simplify the implementation phase.

Min-Max Normalization

Definition 5 *Let $cost(T)$ be the cost of a tour T in the set of all tours S. To normalize the costs to a range $[0, 1]$, let $cost_{\min}$ and $cost_{\max}$ be the minimum and maximum costs in the population, respectively. The normalized cost of a tour T, denoted as $cost^{norm}(T)$, is calculated as follows:*

$$cost^{norm}(T) = \frac{cost(T) - cost_{\min}}{cost_{\max} - cost_{\min}}$$

If $cost_{\min} = cost_{\max}$, then all tours have the same cost, and we set $cost^{norm}(T) = 0$ for all tours $T \in P \cup O$.

Fitness Calculation

Definition 6 *The fitness of a tour T, denoted as $F(T)$, is calculated by transforming the normalized cost to give higher values to better tours. Then, the chances of selecting a tour T for reproduction are obtained by normalizing the fitness values across all tours:*

$$F(T) = \frac{1 - cost^{norm}(T)}{\sum_{T_i \in P \cup O} (1 - cost^{norm}(T_i))}$$

where $P \cup O$ is the set of all tours in the combined population and offspring. This normalization ensures that the sum of all fitness values is 1, which can be used as probabilities for different forms of population preservation.

3.3 crossover and mutation

The crossover phase in a Genetic Algorithm (GA) should be implemented by mixing the genetic codes from two chromosomes. Traditionally, these chromosomes are represented as sequences of 0s and 1s. However, following various approaches and based on the work of [1], traditional mixing is unsuitable for this context, at least not for our current solution's implementation. This is because we need to ensure that nodes are not repeated in the solution and that all nodes are covered.

Following animal logic, we apply a mutation to all offspring with probability mt. To address this, we have implemented the algorithm 1.

Algorithm 1 GA-P: Crossover and Mutation for Population

Require: Population P, crossover proportion cp, mutation threshold mt
Ensure: New offspring population O

$O \leftarrow \emptyset$ ▷ Initialize offspring set
Randomly select pairs of chromosomes (T_a, T_b) from P using combinations, where the sample size is $\lfloor sp \cdot cp \rfloor$
for all pairs (T_a, T_b) **do**
    Randomly choose a crossover point $cv \in [0, |V|]$
    Split T_a and T_b at the crossover point, forming dna_a and dna_b
    Form offspring:
        child_a = first part of dna_a + nodes from dna_b not in the first part
        child_b = first part of dna_b + nodes from dna_a not in the first part
    for each offspring child in {child_a, child_b} **do**
        if random probability $< mt$ **then**
            Swap two randomly chosen nodes in the offspring's DNA
        end if
    end for
    Add child_a and child_b to O
end for
return O ▷ Return new offspring population

Analyzing the code, since we are using all combinations of size 2 in $\lfloor ps \cdot cp \rfloor$:

$$|O| = \binom{\lfloor ps \cdot cp \rfloor}{2}$$

3.4 GA-P

We refer to the algorithm developed in this work as GA-P, building upon previous references. The algorithm incorporates all the code and functions described below (see Algorithm 2). Typically, it is sufficient to use the function `run` to initiate the entire process.

Algorithm 2 TSP Genetic Algorithm (GA-P)

Require: List of nodes V, list of weighted edges E, population proportion pp, crossover proportion cp, crossover threshold th, mutation threshold mt, exploration probability ep, size of the window to check convergence k

Ensure: Optimized tour and its cost

Initialize population size $sp \leftarrow \lfloor |V| \cdot pp \rfloor$
Set step_number $\leftarrow 0$
Initialize random population P of size sp
Generate node weights using edge weights, $W \leftarrow$ create weight dictionary from E
function STEP
    Generate new members $O \leftarrow$ CROSSOVERANDMUTATION(P, cp, th, mt)
    $P \leftarrow P \cup O$
    Remove duplicate tours from P
    cost_list $\leftarrow [cost(T_i) \mid T_i \in P]$
    fitness_list $\leftarrow [F(T_i) \mid T_i \in P]$
    Randomly assign 1 to $\lfloor ep * ps \rfloor$ elements in fitness_list
    fitness_list $\leftarrow sort$(fitness_list, descending=True)
    $P \leftarrow$ first sp elements of fitness_list
    Increment step count step_number $\leftarrow$ step_number $+ 1$
    return New population P, and their costs
end function
function RUN
    $T^s \leftarrow []$, $costs \leftarrow []$

    while (`length of` $costs < k$) or $(\sigma(costs[-k:]) \neq 0)$ **do**
        P, cost_list $\leftarrow$ STEP()
        $T^s \leftarrow \arg\min_{T \in P} \text{cost}(T)$
        $costs \leftarrow costs \cup [cost(T^s)]$
    end while
    return $T^s, cost(T^s)$
end function

4 Experimental part

In this section, we will describe the hardware used for testing and the methodology for the hyperparameter tuning.

4.1 System Information for Python Experiments

Category	**Details**
CPU Information	
Architecture	x86_64
CPUs	24 (12 cores, 2 threads per core)
Model	AMD Ryzen 9 5900X 12-Core Processor
Memory (RAM) Information	
Total	31 GiB
Storage Information	
Type	NVMe SSD
Operating System Information	
Distribution	Ubuntu 22.04.4 LTS (jammy)
Python Version	
Python	3.8.11
Version Control System	
System	GitHub

Table 1: Summary of system information relevant for Python experiments

4.2 Data for experimentation

Since we wanted to test different weights and sizes, we developed a method for generating complete graphs $G(V, E, W)$, where the weights W are sampled from a uniform distribution, $W \sim \text{Uniform}(A, B)$, with $A, B \in R$ (see 3).

Algorithm 3 Generate Complete Graph

Require: Number of nodes n, random weight range $[A, B]$
Ensure: A complete graph with nodes V, edges E, and weights W

1: $V \leftarrow \{0, 1, \ldots, n-1\}$ ▷ List of nodes
2: $E \leftarrow \text{permutations}(V, 2)$ ▷ Generate all possible edges
3: Assign random weights to edges:

$$EW \leftarrow \{(u, v, w) \mid (u, v) \in E, w \sim \text{Uniform}(A, B), w \in W\}$$

4: **return** V, EW ▷ List of nodes and weighted edges

4.3 Hyper-parameter tunning

To optimize various hyperparameters, we developed a process to test multiple graph sizes: [20, 40, 60, 80]. An improvement to this process was made by introducing a comparison based on the 50th observation from each scenario. By determining the percentile rank of this 50th observation among the costs collected over 100 steps, we can assess the quality of each hyperparameter configuration. Since we do not have the optimal route for all randomly generated graphs, our approach measures how effective each hyperparameter setting is by evaluating its ability to reach a lower percentile; a lower percentile indicates better performance in finding lower-cost routes.

Algorithm 4 Hyperparameter Optimization for Genetic Algorithm

Require: A hyperparameter to optimize, ft; a list of values to test, $vs = [v_1, v_2, \ldots, v_m]$; a set of graph sizes $|V| \in [20, 40, 60, 80]$; and a comparison step s, where $s = 50$

Ensure: Plot of percentile ranks for different values of ft over multiple graph sizes

1: Define the hyper-parameter to optimize: ft
2: $vs \leftarrow [v_1, v_2, \ldots, v_m]$ ▷ Set of values to test for ft
3: **for each** $|V|$ in $[20, 40, 60, 80]$ **do**
4: Generate a random complete graph $G(V, E, W)$ with $|V|$ nodes
5: Fix all GA parameters except for ft
6: Initialize list of GAs with different ft values:

$$\text{gas} = [\text{GA}(params + v_1), \text{GA}(params + v_2), \ldots, \text{GA}(params + v_m)]$$

7: Run each GA in parallel using multithreading
8: costs $\leftarrow [\text{cost}(ga_1), \text{cost}(ga_2), \ldots, \text{cost}(ga_m)]$
9: flat_costs $\leftarrow \bigcup_{i=1}^{m} \text{cost}(ga_i)[100]$
10: percentiles $\leftarrow$ [calculate_percentile(costs$(ga_i)[50]$, flat_costs) for $ga_i \in$ gas]
11: **end for**
12: **Plot Results:** Create a plot comparing vs and the corresponding percentile ranks for each graph size

4.4 Algorithm comparison

For comparison, we use three algorithms: our GA-P, GA-N (a version of the genetic algorithm with a non-randomized crossover threshold fixed at $|V|/2$), and the Branch and Bound algorithm, a traditional method for solving the problem. The implementation of the Branch and Bound algorithm is based on the code found in [4].

The algorithm we have developed to compare three alternatives is Algorithm 5.

Algorithm 5 Simulating TSP Solutions with Various Algorithms

Require: Parameter set for GA-P and GA-N: $params = \{population\_prop : 1.1, crossover\_proportion : 0.8, mutation\_thd : 0.3, exploration\_prob : 0.2\}$

Ensure: Results for three TSP algorithms on randomly generated graphs of varying sizes

$results \leftarrow []$
$graphs\_generated \leftarrow []$ ▷ Graphs generation for reproducibility
$ids \leftarrow 0$ ▷ Simulation id
$range\_a \leftarrow 3, range\_b \leftarrow 21$
$SN = 10$ ▷ Set repetitions for each scenario
for each $|V|$ in range $[range\_a, range\_b]$ **do**
    for each repetition in SN **do**
        $G(V, E, W) \leftarrow$ generate_network$(|V|, A = 0, B = 300)$ ▷ Using algorithm 3
        $graphs\_generated$.append$(ids, G\})$
        Run Branch & Bound TSP:
        Get solution: (T^* and $cost(T^*)$)
        $results$.append$(\{"branch_bound", ids, G, T^*, cost(T^*)\})$
        Run GA-P:
        Get solution: (T^s and $cost(T^s)$)
        $results$.append$(\{"GA-P", ids, G, T^s, cost(T^s)\})$
        Run GA-N:
        Get solution: (T^s and $cost(T^s)$)
        $results$.append$(\{"GA-N", ids, G, T^s, cost(T^s)\})$
        $ids \leftarrow ids + 1$
    end for
end for

4.4.1 Branch and Bound algorithm

For the Branch and Bound algorithm, we used the Python implementation found in [4]. The code was originally written in Python 2, but we made the needed changes to make it work in Python 3. The changes were focused on changing some functions deprecated in Python 3 in the original code. And adding a wrapper to make it work similar to our GA-P solution.

4.4.2 Time Complexity Analysis

To estimate the time complexity of the Branch and Bound algorithm, we utilized the code provided in [5]. Specifically, we applied an exponential curve fitting using the equation:

$$a, b = \text{scipy.optimize.curve\_fit}(\lambda\ t, a, b : a \cdot \text{numpy.exp}(b \cdot t), x, y)$$

While a full explanation of this method is beyond the scope of this work, this approach aims to approximate the time complexity of Branch and Bound. The objective is to gain an empirical understanding of how the algorithm's execution time scales, facilitating a comparison with the GA-P approach.

A good explanation of the Branch and Bound algorithm can be found in [6]. Following this process, the algorithm may take $|V|!$ comparisons in the worst case when the weights are constant. Therefore, during the phase when the algorithm decides whether to explore more edges, it may need to check all tours. In a complete graph, this amounts to $|V|!$ tours, or $(|V| - 1)!$ if a starting node is fixed.

5 Results and Analysis

5.1 Result for Hyper-parameter tuning

Following the process described in the section 4.3 we choose the values:

Hyper-parameter	**Value**
Population Proportion (*pp*)	1.1
Crossover Proportion (*cp*)	0.8
Mutation Threshold (*mt*)	0.3
Exploration Probability (*ep*)	0.2

Table 2: Final hyper-parameter values after tuning

5.1.1 Population proportion (*pp*)

These are the results for the process in *pp*:

	20 Nodes	**40 Nodes**	**60 Nodes**	**80 Nodes**	**AVG**
0.1	90.0	91.0	88.0	88.0	89.25
0.3	79.0	74.0	61.0	68.0	70.50
0.5	64.0	64.0	64.0	61.0	63.25
0.7	34.0	52.0	53.0	49.0	47.00
0.9	42.0	44.0	41.0	38.0	41.25
1.1	28.0	39.0	37.0	31.0	33.75
1.3	54.0	27.0	42.0	36.0	39.75
1.5	15.0	17.0	22.0	26.0	20.00
1.7	10.0	4.0	7.0	9.0	7.50
1.9	12.0	31.0	14.0	16.0	18.25

Table 3: Performance of different population proportions on complete graphs of varying sizes. Each row represents a different value of the hyperparameter `population_prop`, and the columns show the performance measured on graphs with 20, 40, 60, and 80 nodes, as well as the average performance across all sizes. Lower values indicate better performance.

We chose a population proportion (*pp*) value of 1.1 over other options with better average costs because higher values significantly increase computational cost. While

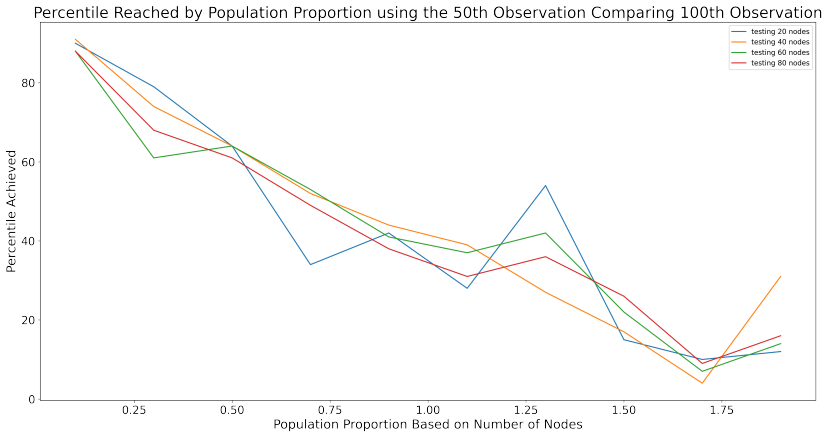

Figure 1: Relationship between population proportion and the percentile achieved for different graph sizes. The plot compares how varying the population proportion affects the performance of the genetic algorithm using the 50th observation as a benchmark against the 100th observation.

moving from $pp = 0.3$ to $pp = 1.1$ yields a substantial improvement of 36.75 in average cost, the benefits of further increasing pp beyond 1.1 do not match this level of improvement. Therefore, $pp = 1.1$ provides a balanced trade-off between computational efficiency and performance gains.

5.1.2 Crossover proportion (*cp*)

These are the results for the process in *cp*:

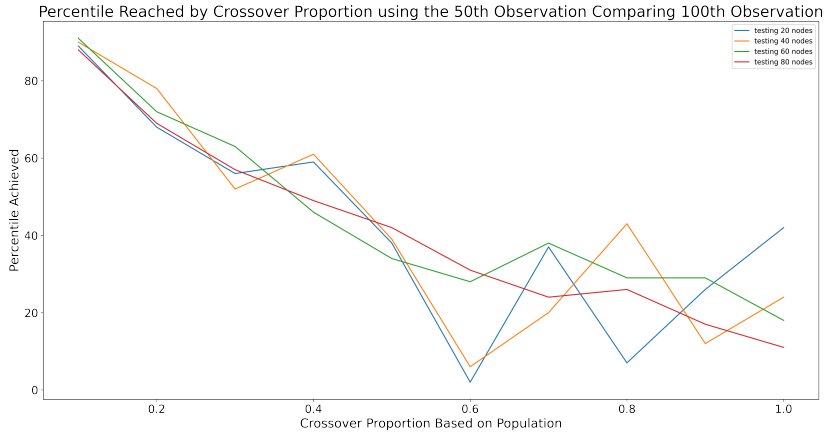

Figure 2: Relationship between crossover proportion and the percentile achieved for different graph sizes. The plot compares how varying the crossover proportion affects the performance of the genetic algorithm, using the 50th observation as a benchmark against the 100th observation.

	20 Nodes	40 Nodes	60 Nodes	80 Nodes	AVG
0.1	89.0	90.0	91.0	88.0	89.50
0.2	68.0	78.0	72.0	69.0	71.75
0.3	56.0	52.0	63.0	57.0	57.00
0.4	59.0	61.0	46.0	49.0	53.75
0.5	38.0	39.0	34.0	42.0	38.25
0.6	2.0	6.0	28.0	31.0	16.75
0.7	37.0	20.0	38.0	24.0	29.75
0.8	7.0	43.0	29.0	26.0	26.25
0.9	26.0	12.0	29.0	17.0	21.00
1.0	42.0	24.0	18.0	11.0	23.75

Table 4: Performance of different crossover proportions on complete graphs of varying sizes. Each row represents a different value of the hyperparameter `crossover_proportion`, and the columns show the performance measured on graphs with 20, 40, 60, and 80 nodes, as well as the average performance across all sizes. Lower values indicate better performance.

We chose a crossover proportion (cp) value of 0.8 because it provides a balanced trade-off between performance improvement and computational efficiency. While increasing cp from 0.3 to 0.8 results in a notable improvement in average performance, reducing the average cost significantly, further increasing cp beyond 0.8 yields diminishing returns and can increase computational complexity. Therefore, $cp = 0.8$ was selected as it achieves strong performance improvements without excessive computational costs.

5.1.3 Mutation threshold (mt)

These are the results for the process in mt:

	20 Nodes	40 Nodes	60 Nodes	80 Nodes	AVG
0.1	67.0	54.0	44.0	61.0	56.50
0.2	41.0	28.0	53.0	34.0	39.00
0.3	21.0	48.0	35.0	32.0	34.00
0.4	61.0	57.0	24.0	44.0	46.50
0.5	24.0	44.0	25.0	47.0	35.00
0.6	63.0	44.0	35.0	8.0	37.50
0.7	51.0	13.0	9.0	44.0	29.25
0.8	28.0	16.0	47.0	35.0	31.50
0.9	51.0	43.0	62.0	51.0	51.75
1.0	46.0	30.0	69.0	62.0	51.75

Table 5: Performance of different mutation thresholds on complete graphs of varying sizes. Each row represents a different value of the hyperparameter `mutation_thd`, and the columns show the performance measured on graphs with 20, 40, 60, and 80 nodes, as well as the average performance across all sizes. Lower values indicate better performance.

	20 Nodes	40 Nodes	60 Nodes	80 Nodes	AVG
0.0	33.0	31.0	43.0	42.0	37.25
0.02	67.0	54.0	46.0	44.0	52.75
0.04	0.0	61.0	45.0	40.0	36.50
0.06	70.0	42.0	7.0	53.0	43.00
0.08	46.0	11.0	19.0	51.0	31.75
0.1	82.0	64.0	63.0	57.0	66.50
0.12	12.0	68.0	36.0	40.0	39.00
0.14	16.0	42.0	42.0	57.0	39.25
0.16	57.0	40.0	64.0	33.0	48.50
0.18	49.0	4.0	37.0	59.0	37.25

Table 6: Performance of different exploration probabilities on complete graphs of varying sizes. Each row represents a different value of the hyperparameter `exploration_prob`, and the columns show the performance measured on graphs with 20, 40, 60, and 80 nodes, as well as the average performance across all sizes. Lower values indicate better performance.

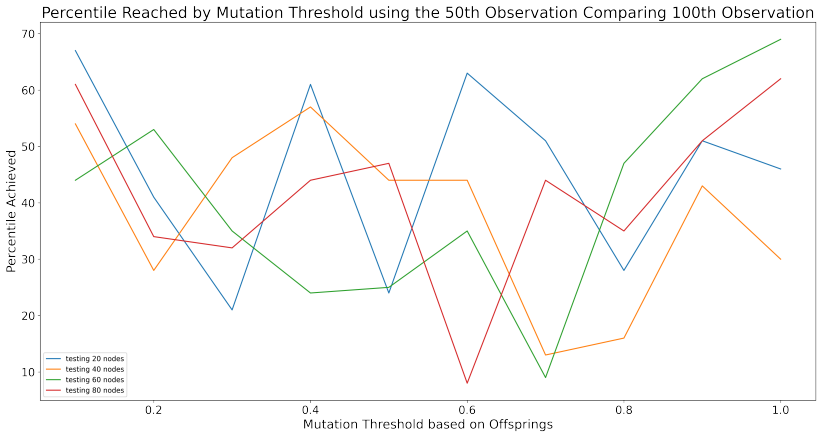

Figure 3: Relationship between mutation threshold and the percentile achieved for different graph sizes. The plot compares how varying the mutation threshold affects the performance of the genetic algorithm, using the 50th observation as a benchmark against the 100th observation.

We selected a mutation threshold (mt) value of 0.3 because it provides a strong balance between performance and mutation effect. The results show that while varying mt does not significantly affect the computational cost, decreasing mt further does not consistently lead to better performance. Thus, $mt = 0.3$ is chosen as it offers good performance without any unnecessary changes to the mutation process that would not yield additional benefits.

5.1.4 Exploration Probability (*ep*)

These are the results for the process in *ep*:

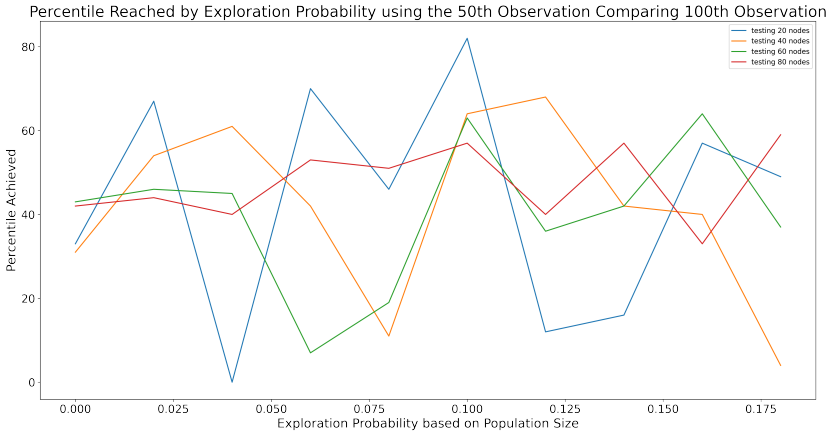

Figure 4: Relationship between exploration probability and the percentile achieved for different graph sizes. The plot compares how varying the exploration probability affects the performance of the genetic algorithm, using the 50th observation as a benchmark against the 100th observation.

We selected an exploration probability (*ep*) value of 0.2 because it maintains a balance between performance and diversity of solutions. While we did not observe significant improvements or large performance gains with higher *ep* values, they often led to worse solutions. By using $ep = 0.2$, we allow some less optimal solutions to survive into the next generation, promoting diversity without sacrificing overall solution quality. This balance helps maintain a healthy exploration-exploitation trade-off in the algorithm.

5.2 Results of algorithm comparison

The process for compare the different nodes using the algorithm 5 took 4h 13min and the results were the following.

	Simulations Count	**Average Cost**			**Cost Std. Dev.**		
Nodes	All	B&B	GA-N	GA-P	B&B	GA-N	GA-P
3	10	531.7	531.7	531.7	172.36	172.36	172.36
4	10	524.2	524.2	524.2	213.45	213.45	213.45
5	10	524.8	524.8	524.8	150.08	150.08	150.08
6	10	544.9	544.9	544.9	108.05	108.05	108.05
7	10	506.1	506.1	506.1	148.88	148.88	148.88
8	10	545.5	545.5	545.5	165.02	165.02	165.02
9	10	621.5	621.5	621.5	144.54	144.54	144.54
10	10	582.0	592.3	582.0	189.70	187.91	189.70
11	10	596.9	641.5	620.0	126.39	113.37	101.50
12	10	618.6	646.4	647.3	109.26	111.15	96.52
13	10	618.6	672.6	684.3	113.26	114.70	116.09
14	10	640.7	690.4	684.9	101.67	100.97	126.04
15	10	579.9	650.9	624.3	181.45	183.11	188.85
16	10	560.6	628.6	617.4	137.33	166.26	167.56
17	10	539.9	701.8	657.6	91.23	117.94	157.44
18	7	630.14	791.85	692.57	71.158	59.97	80.59

Table 7: Summary of simulations showing the count, average costs, and standard deviation of costs for Branch and Bound (B&B), Genetic Algorithm with Normal crossover (GA-N), and Genetic Algorithm with Proposed crossover (GA-P) across different graph sizes.

	Average Time (s)			**Time Std. Dev. (s)**		
Nodes	B&B	GA-N	GA-P	B&B	GA-N	GA-P
3	0.000035	0.001563	0.001915	0.000009	0.000329	0.000301
4	0.000057	0.003097	0.003336	0.000002	0.000046	0.000104
5	0.000159	0.005097	0.005432	0.000016	0.000087	0.000112
6	0.000564	0.005877	0.006295	0.000094	0.000365	0.000345
7	0.001833	0.009249	0.010786	0.000918	0.001600	0.001378
8	0.007581	0.016343	0.015181	0.004340	0.004158	0.002940
9	0.036392	0.020157	0.022949	0.018138	0.005146	0.007158
10	0.132656	0.030390	0.033429	0.120006	0.009906	0.006290
11	0.396178	0.035748	0.041281	0.275419	0.010021	0.006956
12	1.442373	0.063747	0.057556	1.129157	0.020795	0.026573
13	4.273159	0.077925	0.078178	3.647951	0.042751	0.035898
14	24.252709	0.080164	0.087980	20.927455	0.017405	0.021631
15	45.307447	0.116063	0.103721	48.913851	0.042122	0.032998
16	169.365427	0.158363	0.134725	298.464014	0.045923	0.057062
17	162.926439	0.167231	0.177309	167.780035	0.031795	0.056514
18	1586.576320	0.173900	0.187774	879.050806	0.033951	0.043561

Table 8: Average computation time (in seconds) and standard deviation for Branch and Bound (B&B), Genetic Algorithm with Normal crossover (GA-N), and Genetic Algorithm with Proposed crossover (GA-P).

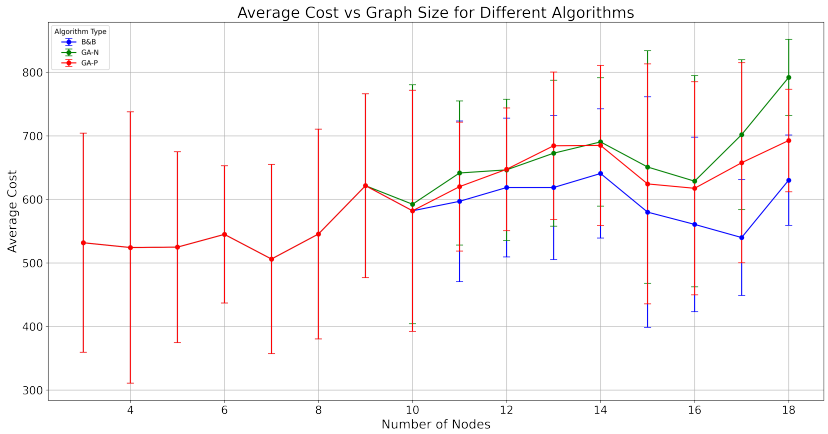

Figure 5: Average cost vs graph size for different algorithms. The plot compares the average cost for each algorithm type as the graph size increases. The vertical lines represent the standard deviation of the costs over multiple runs for each graph size and algorithm type. Note that GA-P proposed is overall better than GA-N. The standard deviation is never 0 because is the average along different G).

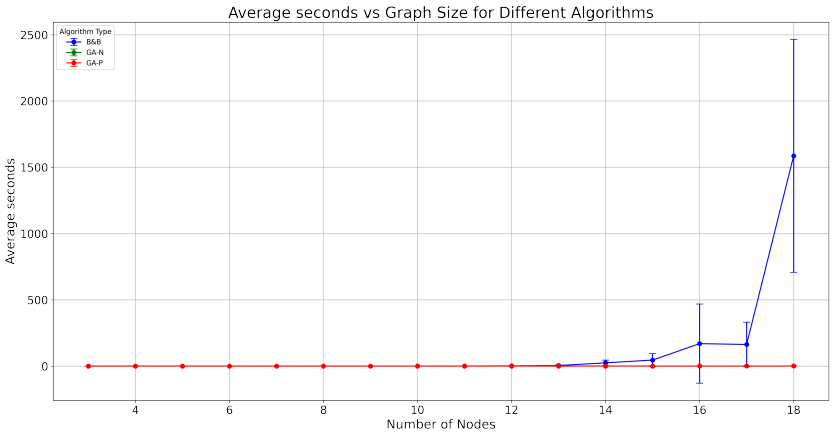

Figure 6: Average computation time vs graph size for different algorithms. The plot demonstrates how the average computation time increases with the number of nodes for each algorithm type. The Branch and Bound (B&B) algorithm shows an exponential increase in time as the number of nodes grows, while the Genetic Algorithms (GA) approaches maintain a more consistent time complexity. However, the GAs have a larger deviation from the optimal solution (see Figure 5.2), which can be minimized by increasing the number of iterations.

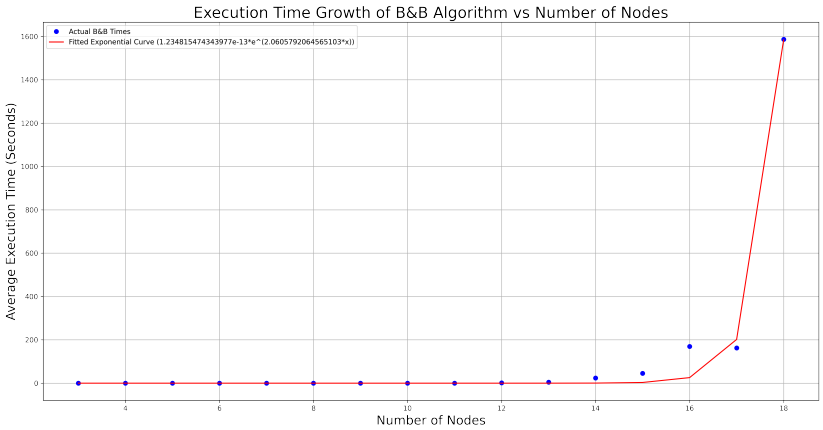

Figure 7: Execution time growth of the Branch and Bound (B&B) algorithm as the number of nodes increases. The blue dots represent the actual average execution times for B&B across different graph sizes. The red line represents the fitted exponential curve, with the form $1.23 \times 10^{-13} e^{2.06x}$, where x is the number of nodes. The fitted curve shows the exponential growth of execution time as graph size increases.

6 Conclusions

In summary, this work has demonstrated that using a Genetic Algorithm (GA) is less time-consuming and provides near-optimal solutions compared to methods like Branch and Bound (see Section 5.2).

Our results indicate that the GA finds optimal solutions for small graph sizes, suggesting that exploring larger windows (K) may enable finding optimal solutions for larger instances as well.

We also experimented with a randomized crossover threshold (GA-P) and an exploration parameter. The randomized crossover improved solution quality, while the exploration parameter had a negative impact. This could be due to its influence during the fitness step, allowing suboptimal solutions to persist across generations. An interpretation is that a higher crossover proportion (0.8) combined with random mutation in offspring maintains a controlled inclusion of non-optimal solutions, which is beneficial in small amounts. The full potential of the exploration parameter might only be apparent in graphs with a larger number of nodes and edges (beyond the 80-node limit tested in this study).

In essence, GA-P offers a viable alternative when exact methods like Branch and Bound or brute-force search are infeasible, and it performs well when execution limits are imposed by the user.

Future work could focus on: (1) extending the algorithm to handle non-complete graphs; (2) refining the exploration process by using fitness values as selection probabilities (since these values already lie between 0 and 1, summing to 1); and (3) running parallel GAs with different initial populations and then combining their solutions. Additionally, addressing the issue of duplicate solutions represented in different ways (e.g., solutions starting from different nodes but representing the same tour) could enhance population diversity and solution quality.

References

[1] A. Bashir and M. K. Srivastava, "Implementation of genetic algorithm using the traveling salesman problem in cloud," 2021.

[2] I. Droste, "Algorithms for the travelling salesman problem [bachelor thesis]," 2017. https://studenttheses.uu.nl/bitstream/handle/20.500.12932/29854/BachelorthesisIsabel.

[3] K. S. J. S. R. M. S. C. Sahib Singh Juneja, Pavi Saraswat, "Travelling salesman problem optimization using genetic algorithm," pp. 264–268, IEEE, 2019.

[4] M. Bahri, "Tsp solver: Branch and bound implementation for the traveling salesman problem." https://github.com/mostafabahri/tsp-solver/, 2020. Accessed: 2024-09-30.

[5] kennytm, "How to do exponential and logarithmic curve fitting in python? i found only polynomial fitting." https://stackoverflow.com/a/3433503, 2010. Accessed: 2024-09-29.

[6] A. Rastogi, A. K. Shrivastava, N. Payal, R. Singh, and A. Sharma, "A proposed solution to travelling salesman problem using branch and bound," *International Journal of Computer Applications*, vol. 65, no. 5, 2013.