Solving the Travelling Salesman Problem Using Genetic Algorithms

Mehmet Görkem Basar

Berliner Hochschule für Technik
Mathematical Modelling, SoSe23

Juli 2023

Abstract

The Traveling Salesman Problem (TSP) is a classic algorithmic problem in the fields of computer science and operations research, focusing on optimization. It concerns a salesman who needs to find the shortest possible route that allows them to visit a set of cities and return to their original location. This paper presents a study on the application of Genetic Algorithms (GAs) for solving the TSP.

Genetic Algorithms are heuristic search techniques used in computing to find exact or approximate solutions to optimization and search problems. They are categorized as global search heuristics, offering a robust search in complex spaces, which make them particularly suitable for the TSP.

In my project, I implemented a Genetic Algorithm in Python, conducted a series of experiments, and analyzed the solutions obtained. My GA variant includes the integration of elitist selection, order-based crossover, and swap mutation. I performed multiple runs of the algorithm, maintaining the same city locations while allowing for the stochastic nature of the GA to generate varying solutions.

The analysis provides insights into the performance of the Genetic Algorithm for solving the TSP, considering the quality of solutions and consistency of results across multiple runs. This study aims to contribute to the broader understanding of Genetic Algorithms' efficiency and reliability when applied to combinatorial optimization problems, such as the TSP.

Contents

1 Introduction

1.1 Background and Problem Statement

The Traveling Salesman Problem (TSP) is a classic algorithmic problem in the field of computer science and operations research which focuses on optimization. In this problem, a salesman is given a list of cities and must determine the shortest route that allows him to visit each city once and return to his original location. Despite its simple formulation, the TSP is an NP-hard problem and an active area of research due to its relevance in logistics, vehicle routing, and other real-world optimization problems.

Genetic algorithms (GAs) are a type of search heuristic that mimic the process of natural evolution, known for their ability to find good solutions for optimization problems like the TSP. They leverage bio-inspired operators such as mutation, crossover (reproduction), and selection. The potential of GAs to explore a large solution space effectively makes them a suitable choice for solving complex problems like TSP.

In this paper, we delve into the details of applying a genetic algorithm to the TSP. This work was motivated by the desire to study the performance and behaviour of a genetic algorithm in solving TSP and to identify the implications and benefits of parameter tuning in Genetic Algorithms.

We begin with an overview of genetic algorithms and their application in the TSP, providing a theoretical framework for the uninitiated. We subsequently discuss our Python implementation of the genetic algorithm for solving the TSP and elaborate on the specific design decisions made during the coding process. Finally, we present the results of our experiments, discuss the insights gained from them, and propose potential avenues for future work.

1.2 Aim and Scope

The primary aim of this study is to explore and demonstrate the application of genetic algorithms for solving the Traveling Salesman Problem (TSP). This includes detailing the theoretical basis of genetic algorithms, explaining the process of their implementation in Python, and evaluating their effectiveness in finding optimal or near-optimal solutions to the TSP.

The scope of the study is confined to the application of a specific type of genetic algorithm, which incorporates particular selection, crossover, and mutation methods, to instances of the TSP. Our research includes the development of a Python-based solution, a series of controlled experiments using various parameters, and a comprehensive analysis of the results obtained.

While our study provides an in-depth analysis of the performance of genetic algorithms on TSP, it should be noted that the results might not generalize to all instances of TSP or other combinatorial optimization problems. Additionally, there are other metaheuristic techniques, such as simulated annealing, ant colony optimization, and particle swarm optimization, which can also be effective at solving TSP and might outperform genetic algorithms under certain conditions. These methods, however, fall outside the scope of this study and present potential avenues for future research.

2 Theoretical Framework

2.1 Genetic Algorithms: An Overview

The genetic algorithm (GA) is the most widely known type of evolutionary algorithm. It was initially conceived by Holland as a means of studying adaptive behaviour, as suggested by the title of the book describing his early research: Adaptation in Natural and Artificial Systems[4].

Genetic algorithms are evolutionary algorithms inspired by natural selection, where potential solutions to a problem evolve towards optimality over successive generations. They begin with an initial population of potential solutions. Each solution's fitness is evaluated, representing how well the problem is solved.

During each generation, solutions (individuals) are selected for reproduction based on fitness, with fitter individuals more likely to be chosen. These individuals undergo crossover, emulating biological reproduction, where pairs of solutions (parents) combine to produce new solutions (children). Beneficial traits from each parent can thus potentially be combined in the children.

Mutation, introducing small random changes, follows crossover. This operator diversifies the solutions and allows for exploration of the solution space, preventing early convergence to sub-optimal solutions.

In the Traveling Salesman Problem context, an individual can represent a city-visiting route, and the individual's fitness can be its total route distance. Using selection, crossover, and mutation processes, the genetic algorithm evolves the population over generations to find the shortest possible route.

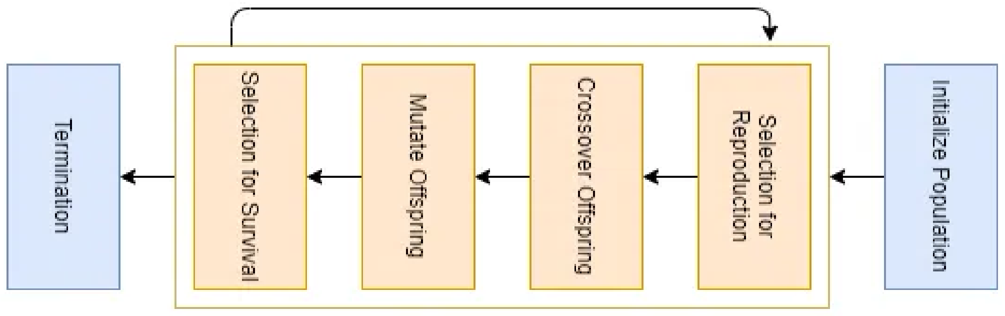

Figure 1: Basic Structure of Genetic Algorithm

2.2 Travelling Salesman Problem(TSP)

The Traveling Salesman Problem is a classic optimization problem in which a salesman is given a list of cities, and the objective is to find the shortest possible route that visits each city exactly once and returns to the starting city. The problem is to determine the order in which the cities should be visited to minimize the total distance traveled by the salesman.

Mathematical Background:

Let n be the number of cities, which are denoted as $1, 2, \ldots, n$.

Let d_{ij} represent the distance between city i and city j, where d_{ij} is a non-negative value, and $d_{ij} = d_{ji}$ since traveling from city i to city j has the same distance as traveling from city j to city i.

The objective is to find a permutation of cities $P = (p_1, p_2, \ldots, p_n)$ that minimizes the total distance:

$$\text{Minimize} \quad \sum_{i=1}^{n-1} d_{p_i,p_{i+1}} + d_{p_n,p_1}$$

Here's a visualization of the TSP

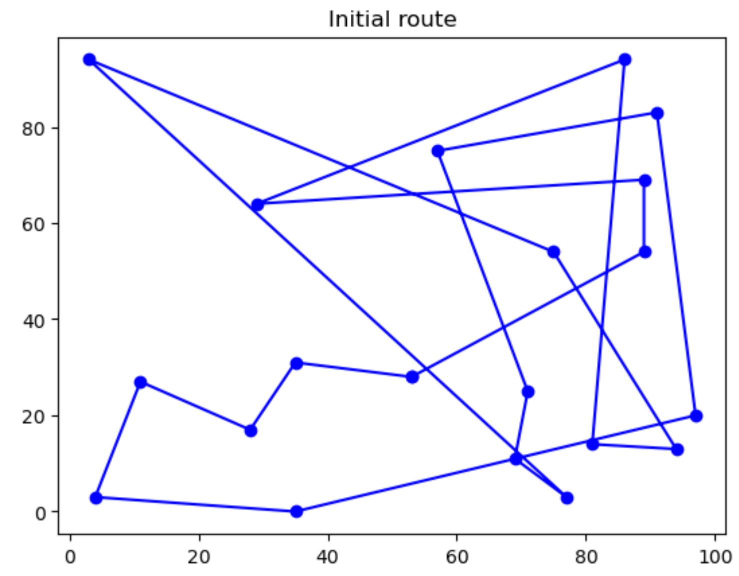

Figure 2: Initial Route

Initial shortest distance: 872.5732782826942

There are 20 dots , each dot represents a city.Our goal is to minimize total distance between routes, to do that we are going to use genetic algorithms.

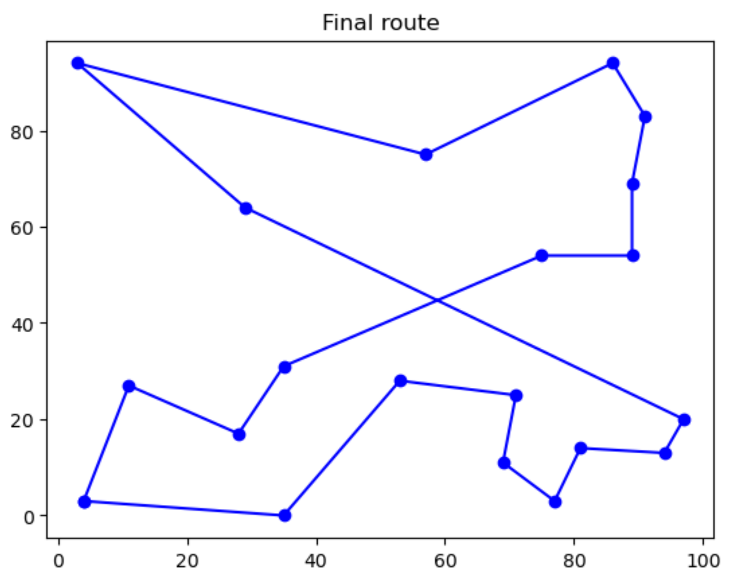

Figure 3: Final Route

Final shortest distance: 514.843947022949

After applying our GA for 300 generations we minimized our total distance from 872 to 514.

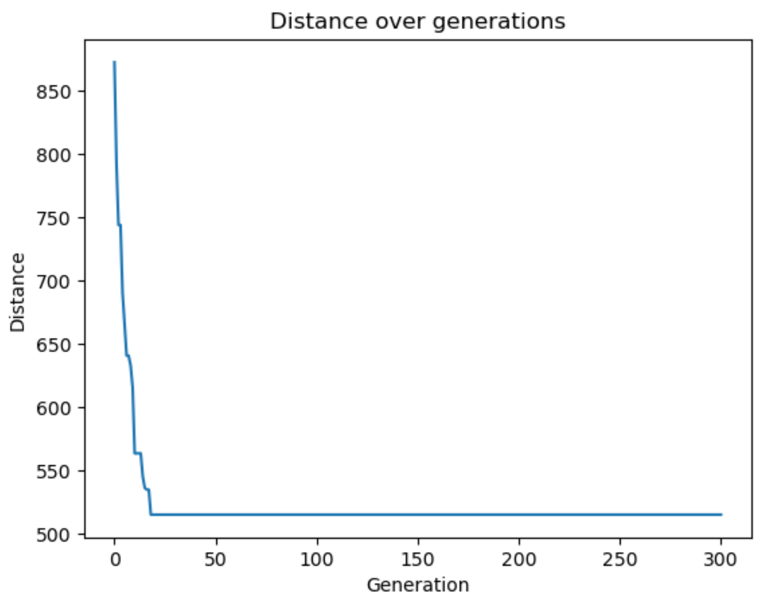

Figure 4: Distance over genarations

Improvement: 357.7293312597452
Over the next chapters we're going to explain how did we do this.

2.3 Genetic Algorithms in TSP

Implementing a genetic algorithm for the Traveling Salesman Problem involves tailoring the general genetic algorithm to the specific requirements of the problem. Key aspects include defining a suitable representation of potential solutions (individuals), implementing appropriate selection, crossover, and mutation operations, and setting a fitting evaluation function.

These are; Individual Representation, Fitness Function, Selection, Crossover and Mutation.

By integrating these components into a genetic algorithm, it is possible to create an effective and powerful optimization tool for solving the Traveling Salesman Problem. The algorithm's performance can be influenced by various factors, including the size of the population, mutation rate, elite size and the number of generations allowed for the evolutionary process.

2.4 Fitness Function

The evaluation function is essential for guiding the population's adaptation to meet specific requirements. It defines improvement within the genetic algorithm and represents the task to be solved. Technically, it assesses the quality of genotypes by converting them to phenotypes and measuring their fitness in the phenotype space. This function plays a crucial role in determining which solutions are more likely to survive and contribute to the next generation. Overall, it is the cornerstone of the genetic algorithm, enabling the search for high-quality solutions to the problem at hand.[1]

In our Genetic Algorithm fitness function is "calculate distance(route)", computes the sum of the Euclidean distances between successive cities in the route, including the distance between the last city and the first city. This is in line with the problem definition of the TSP, which requires the route to return to the starting city after visiting all the other cities.

The Euclidean distance between two cities is calculated as the square root of the sum of the squares of the differences in their x and y coordinates, using the distance(city) method of the City class. This represents the direct (straight-line) distance between the two cities.

Let's denote two cities as City1(x1, y1) and City2(x2, y2). The Euclidean distance D between City1 and City2 is given by:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

To calculate the total distance of a route R with n cities, denoted as City1, City2, ..., Cityn, the formula is:

$$\text{Total Distance}(R) = \\ D(\text{City}_1, \text{City}_2) + D(\text{City}_2, \text{City}_3) + \ldots + D(\text{City}_{n-1}, \text{City}_n) + D(\text{City}_n, \text{City}_1)$$

By defining the fitness function as the total distance of the route, our genetic algorithm is guided to search for routes with smaller total distances, which are considered better or more 'fit'. This aligns with the objective of the TSP, which is to find the shortest possible route that visits each city once and returns to the starting city.

By evolving the population over generations through selection, crossover, and mutation guided by this fitness function, our genetic algorithm is able to progressively find shorter and shorter routes, working towards solving the TSP.

2.5 Selection Mechanism

The selection mechanism in a genetic algorithm is a key process that selects candidate solutions (referred to as individuals) for reproduction in the next generation. The objective of the selection process is to bias the choice towards better solutions. This makes it an integral part of a genetic algorithm because it influences the algorithm's ability to converge towards the optimal solution.

The main characteristics of tournament selection could be summarized as follows.

. Tournament selection only uses local information.

. Tournament selection is very easy to implement and its time complexity is small.

. Tournament selection can be easily implemented in a parallel environment[2]

In our implementation for the Traveling Salesman Problem (TSP), the selection process uses a technique known as tournament selection.

```
Function TOURNAMENT_SELECTION(population, tournament_size):
    1. Set tournament as a random subset of population of size tournament_size.
    2. Return the individual with the shortest route from the tournament.
End Function
```

In the tournament selection method, a subset of the population is selected randomly and the individual with the highest fitness in that subset is chosen for the next generation. This process is repeated until the population of the next generation is filled.

2.6 Crossover Operation

The crossover operation, also known as recombination, is a process where two parent solutions are combined to create new offspring for the next generation. The aim of the crossover operation is to generate new solutions that contain parts of both parent solutions, in the hope that the new solutions will be better than either of the parents.

The specific type of crossover used in our implementation is order crossover. In order crossover, a subset of the route is taken from the first parent, and the remaining cities are filled in according to their order in the second parent. This maintains the relative order of the cities in the routes, which is important in the context of the Traveling Salesman Problem (TSP) as it is a sequence-based problem.[3]

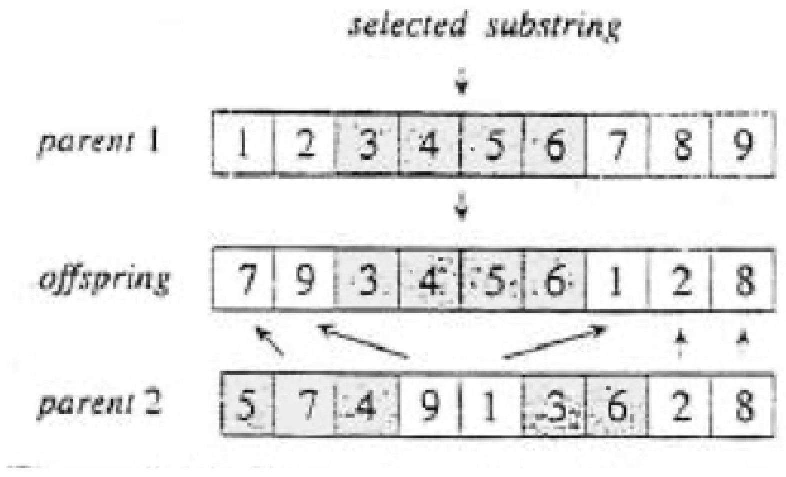

Figure 5: Order Crossover [3]

The function order "crossover(parent1, parent2)" in our code performs this operation. It begins by selecting a subset of the route from the first parent. Then, it fills in the remaining cities according to their order in the second parent. The resultant child inherits the structure from both parents, potentially combining their strengths to form a better solution. This operation is key to creating diversity and novelty in the population, driving the search towards optimal or near-optimal solutions.

2.7 Mutation Operation

Mutation is another crucial operation in the Genetic Algorithm (GA) process. It introduces randomness in the population and is used to maintain and introduce diversity in the genetic population. This helps in preventing the population from converging towards a local minima and helps to explore the larger search space for potential better solutions.

The specific type of mutation used in our implementation is the Swap Mutation. It's a simple and effective mutation technique used in problems dealing with order or sequence such as the Traveling Salesman Problem (TSP).

In Swap Mutation, two positions are selected randomly in a route and their positions are swapped. This means that two cities in the route swap their positions, thereby creating a new route that is a slight variation of the original route

In Swap Mutation,two positions are selected randomly in a route and their positions are swapped.This means that two cities in the route swap their positions, thereby creating a new route that is a slight variation of the original route.For example, we pick location 3 and 6 in path code $(1-2-3-4-5-6-7-8-9)$ for a nine-city TSP and get $(1-2-6-4-5-3-7-8-9)$ as in Figure 3. [2]

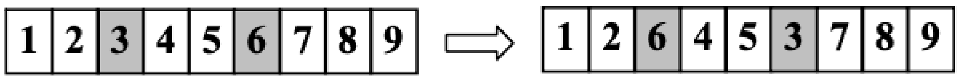
Figure 6: Swap Mutation [2]

The function ”swap mutation(route, mutation rate)” in our code performs this operation. For each city in the route, it checks a randomly generated number against the mutation rate. If the random number is less than the mutation rate, it selects another random city in the route and swaps the two cities. This operation is key to ensuring that the GA does not get stuck in local optimum solutions, by continuously introducing minor variations in the population.

3 Python Implementation

This section of the paper discusses the Python implementation of the Genetic Algorithm for solving the Traveling Salesman Problem (TSP). Each aspect of the Genetic Algorithm process - initialization, selection, crossover, mutation, and the main Genetic Algorithm procedure - is implemented as a separate function, providing a modular and easily understandable structure. Below, we discuss the implementation of each function.

3.1 City Class

The City class is a simple object-oriented representation of a city, characterized by its x and y coordinates. It also includes a method for calculating the Euclidean distance to another city, which is used in determining the total distance of a route.

Listing 1: City class in Python

```python
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance

    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"
```

Compute Euclidean distance(Pythagorean theorem) between two cities, The expression self.x - city.x calculates the difference in x-coordinates between the current city and the other city (city). Taking the absolute value (abs) ensures that the difference is positive.

In the City class, the "repr" method is overridden to provide a string representation of a City object. It returns the coordinates of the city in the format (x,y), where x is the x-coordinate and y is the y-coordinate. This allows for a more readable display of City objects when printed or used in string contexts.

3.2 Initialization

The initialization function, "create initial population(population size, cities)", creates an initial population of random routes. Each route is a permutation of the input list of cities, representing a possible solution to the TSP.

Listing 2: Initialization step in Python

```python
def create_initial_population(population_size, cities):
    return [random.sample(cities, len(cities))

            for _ in range(population_size)]
```

3.3 Fitness Function

The fitness function, "calculate distance(route)", calculates the total distance of a given route. It is used to rank the routes in the population by their total distance, with shorter distances being better.

Listing 3: Fitness Function in Python

```python
def calculate_distance(route):
    return sum([route[i].distance(route[i + 1])

                for i in range(len(route) - 1)]) +
                   route[-1].distance(route[0])
```

3.4 Selection Mechanism

The selection mechanism, implemented by "tournament selection(population, tournament size)", uses tournament selection to select the parent routes for crossover. It randomly selects a subset of the population and chooses the best individual from this subset to become a parent.

Listing 4: Tournament Selection in Python

```python
def tournament_selection(population, tournament_size):
    tournament = random.sample(population, tournament_size)
    return min(tournament, key=calculate_distance)
```

3.5 Crossover Step

The crossover operation, implemented by "order crossover(parent1, parent2)", uses order crossover to combine two parent routes and create a new route for the next generation.

Listing 5: Crossover Step in Python

```python
def order_crossover(parent1, parent2):
    child = [None] * len(parent1)
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child[start:end] = parent1[start:end]

    filled = end - start
    pointer1, pointer2 = end, end
    while filled < len(parent1):
        if pointer1 >= len(parent1):
            pointer1 = 0
        if pointer2 >= len(parent1):
            pointer2 = 0
        if parent2[pointer2] not in child:
            if pointer1 < len(child):
```

```
            child[pointer1] = parent2[pointer2]
            pointer1 += 1
            filled += 1
        pointer2 += 1
    return child
```

3.6 Mutation Operation

The mutation operation, implemented by "swap mutation(route, mutation rate)", uses swap mutation to introduce small random changes into the routes, helping to maintain diversity in the population.

Listing 6: Crossover Step in Python

```
def swap_mutation(route, mutation_rate):
    new_route = route.copy()
    for i in range(len(route)):
        if random.random() < mutation_rate:
            j = random.randint(0, len(route) - 1)
            new_route[i], new_route[j] = new_route[j], new_route[i]
    return new_route
```

3.7 Main Genetic Algorithm Procedure

The main Genetic Algorithm procedure, implemented by "geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate, generations)", brings together all the previous functions to implement the full Genetic Algorithm process. It also includes functionality to plot the cities in a route and the distance over generations, providing a visual representation of the algorithm's progress.

Listing 7: Crossover Step in Python

```
# Main GA function: Implement the GA process with the specified parameters
def geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate,
    generations):
    # Create the initial population
    initial_population = create_initial_population(popSize, population)

    # Save initial shortest distance
    initial_route = min(initial_population, key=calculate_distance)
    initial_distance = calculate_distance(initial_route)
    print("Initial shortest distance: ", initial_distance)
    plot_cities(initial_route, 'Initial route')

    # List to store distances
    distances = [initial_distance]

    # Main loop to perform GA operations
    for generation in range(generations):
        new_population = []
        # Perform selection and store the selected individuals in the new
            population
        for _ in range(eliteSize):
            new_population.append(tournament_selection(initial_population,
                eliteSize))
        # Perform crossover and mutation to generate the rest of the new
            population
        for _ in range(popSize - eliteSize):
            parent1 = tournament_selection(initial_population, eliteSize)
            parent2 = tournament_selection(initial_population, eliteSize)
            child = order_crossover(parent1, parent2)
            child = swap_mutation(child, mutationRate)
            new_population.append(child)
        initial_population = new_population
        # Save shortest distance for this generation
        distances.append(calculate_distance(min(initial_population,
            key=calculate_distance)))

    # After all generations, output the best route and its distance
    final_route = min(initial_population, key=calculate_distance)
    final_distance = calculate_distance(final_route)
    print("Final shortest distance: ", final_distance)
    plot_cities(final_route, 'Final route')

    # Plot the distances over generations
    plt.figure()
    plt.plot(distances)
    plt.title('Distance over generations')
    plt.xlabel('Generation')
    plt.ylabel('Distance')
    plt.show()
    print("Improvement: ", initial_distance - final_distance)

# Create a list of cities
cityList = [City(random.randint(0, 100), random.randint(0, 100)) for _ in
    range(20)]

# Run the genetic algorithm with specified parameters
geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20,
    mutationRate=0.1, generations=300)
```

4 Experimental Setup and Results

In this section, we outline the experimental setup, including the parameters used for the Genetic Algorithm, as well as the results obtained from the experiments.

4.1 Mutation Rate vs Convergence

In this experiment, the impact of varying mutation rates on the Genetic Algorithm's convergence was investigated. Four mutation rates (0.001, 0.01, 0.1, and 1) were tested over 300 generations with a population of 100 and elite size of 20.

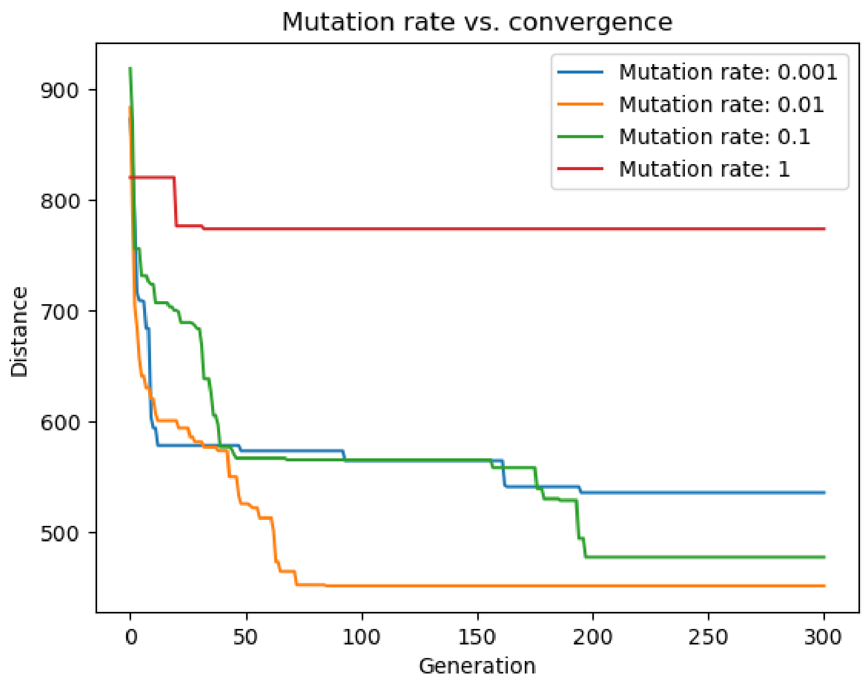

Figure 7: Mutation rate vs convergence

The resulting 'Mutation rate vs. Convergence' graph shows the effect of these mutation rates on the shortest route distance across generations. Some key observations:

1- Highest Mutation rate (1) didn't converged as much as other mutation rates , it shows that , high mutation rate for this problem isn't suitable.

2- In this case mutation rate(0.01) gave the best resluts. It converged final distance in the shortest amount of generations. It shows that mutation rate 0.01 is the optimum value for this problem.

4.2 Population Size vs Convergence

In this experiment, we explore the influence of varying population sizes on the convergence of our Genetic Algorithm. Four population sizes (50, 100, 200, and 300) were tested, each over 300 generations, with a fixed mutation rate of 0.01 and the elite size being 20% of each population size.

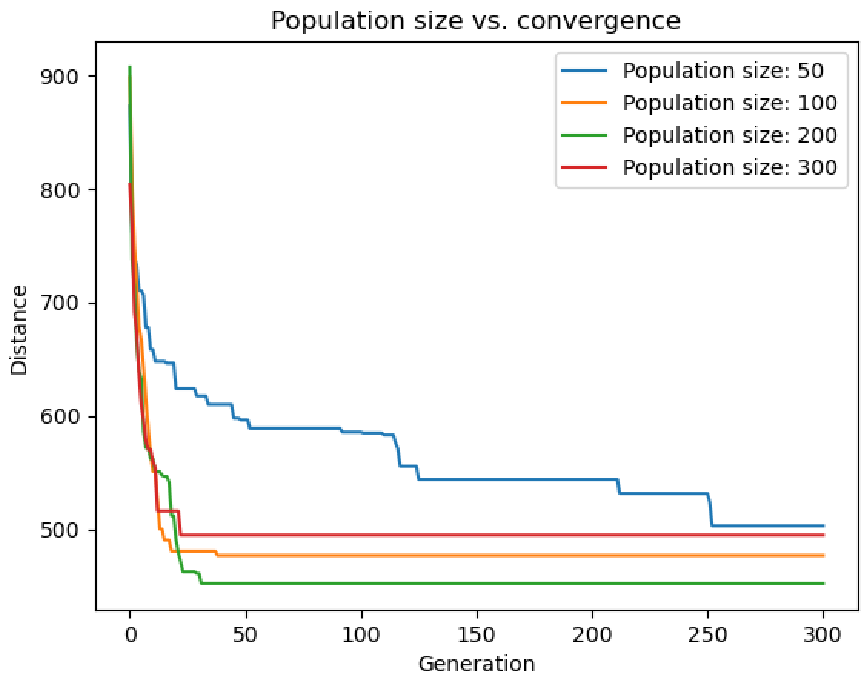

Figure 8: Population Size vs convergence

The resulting 'Population size vs. Convergence' graph illustrates how the different population sizes affected the shortest route distance over generations. Some observations:

1- Smallest(50) population size performed worse compared the other three. To converge to similar final distance it needed much higher generation time , which shows for this problem popsize 50 is not suitable.

2- Other three popsize performed similarly but popsize(200) gave the best results. Reaching final distance in shortest amount of genarations.

These results underscore the importance of population size in a Genetic Algorithm. Larger populations provide a broader search space for the algorithm, which can lead to better solution quality, but at the cost of increased computational resources.

4.3 Elite Size vs Convergence

The "elite size" in a genetic algorithm is a parameter that refers to the number of the best solutions (individuals) from the current population that are automatically passed to the next generation, without undergoing the typical genetic algorithm operations of selection, crossover, and mutation. In our genetic algorithm, the elites are determined by the tournament selection method that we've implemented. This method of elitism ensures that the eliteSize best individuals (routes, in this context) from the current population are always passed to the next generation without being altered by the genetic operations (crossover and mutation). By doing this, the genetic algorithm is guaranteed not to lose its best found solutions from one generation to the next.

In this experiment, we assess the influence of various elite sizes on the convergence of the Genetic Algorithm. Six elite sizes (5, 10, 20, 30, 40, and 50) were tested over 300 generations, with a fixed mutation rate of 0.01 and population size of 100.

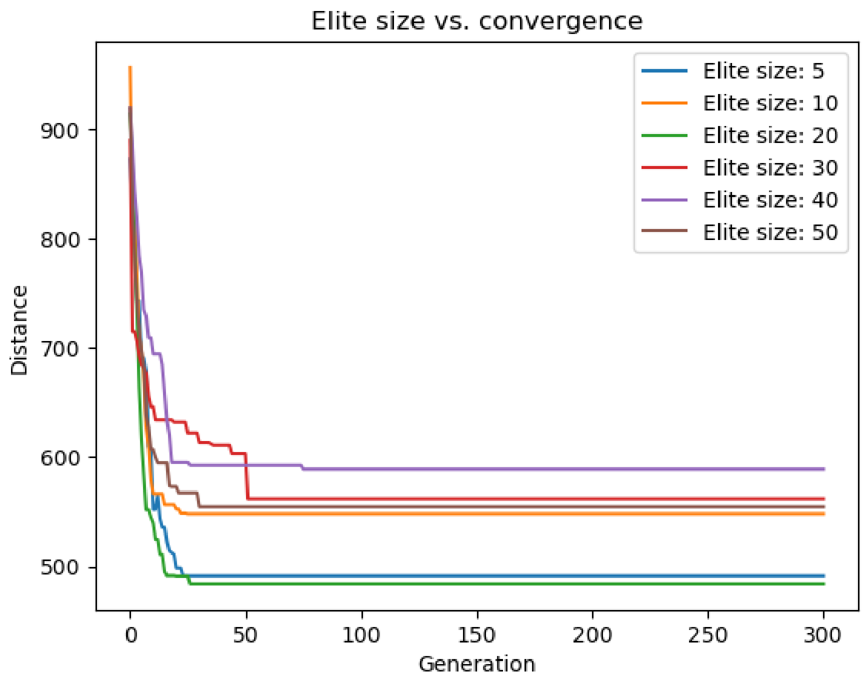

Figure 9: Elite Size vs convergence

The resulting 'Elite size vs. Convergence' graph visualizes the effects of different elite sizes on the shortest route distance across generations. Some key findings include:

1- Smaller elite sizes (5, 10, 20) exhibited the most rapid initial improvement and maintained a stable, gradual reduction in shortest route distance over shortest amount of generations.

2- Larger elite sizes (30, 40, 50) didn't converged as much as other three also they required more generations to converge to reasonable distance.

These results demonstrate the importance of balancing the elite size in a Genetic Algorithm. Too small an elite size may limit the algorithm's ability to retain the best solutions, while too large an elite size may cause premature convergence, limiting the overall potential for finding better solutions. The optimal elite size likely falls within a moderate range, allowing a balance between preserving the best solutions and maintaining diversity in the population.

4.4 Stochastic Behavior of Genetic Algorithm in Solving TSP

This experiment seeks to evaluate the stochastic behavior of the Genetic Algorithm (GA) when applied to solve the Traveling Salesman Problem (TSP).

Given that GAs are stochastic by nature (due to their inherent reliance on random selection, crossover, and mutation operations), multiple runs of the GA on the same problem can yield different results. The degree of variation in these results is influenced by the algorithm's specific design and parameter settings.

The experiment consists of running the GA with specific settings (population size of 100, elite size of 20, mutation rate of 0.01, and 300 generations) on the same problem (TSP with 20 cities) multiple times (500 trials) with setting a random seed(42). This allows the algorithm's inherent stochasticity to fully manifest itself. In another words this allows us to showcase, even though we're using the same locations(coordinates) after each trial we get different results, this means our GA behaving stochasticly.

For each run of the algorithm, the improvement in the shortest route's distance from the initial to the final generation is calculated and stored. These improvement values are then used to assess the GA's variability:

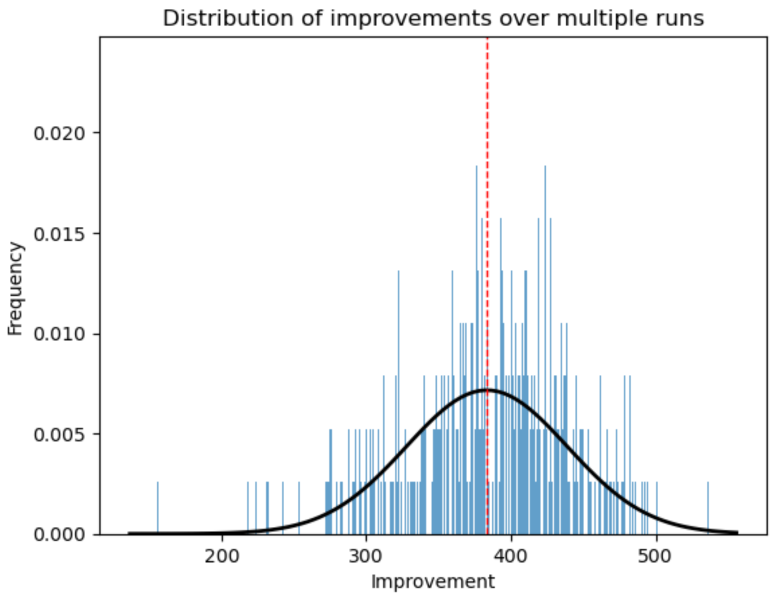

Figure 10: Distribution of improvements over multiple runs

Mean of improvements: 383.32431348784326
Standard deviation of improvements: 55.714777766077226
Minimum of improvements: 155.53867400774755
Maximum of improvements: 536.8164636387916

The mean improvement provides a measure of the average effectiveness of the GA in improving the initial solutions. The standard deviation of the improvements provides a measure of the variability in the GA's performance across different runs. The minimum and maximum improvements provide the range of the GA's performance.

The histogram plot displays the distribution of improvements across all runs, providing a visual representation of how frequently certain improvement values are observed. The fitted normal distribution curve provides an approximation of the overall distribution, and the red dashed line represents the mean improvement. If the histogram closely resembles the normal curve, it indicates that the improvements follow a normal distribution pattern, which can be helpful for further analysis and understanding the behavior of the GA. However, if the histogram exhibits a skewed or non-normal distribution, it suggests that the GA's performance is more complex and influenced by other factors.

Overall, this experiment provides a comprehensive assessment of the GA's stochastic behavior and its implications on the GA's performance in solving the TSP.

5 Discussion and Conclusion

The genetic algorithm (GA) employed in this study has shown its strength in solving the Traveling Salesman Problem (TSP). Through the use of GA operations such as selection, crossover, and mutation, the algorithm managed to generate significantly improved solutions over successive generations.

The role of the mutation rate, population size, and elite size in the performance of the GA was explored. Our findings indicate that these parameters are crucial in the performance of the GA.

The mutation rate was observed to have a substantial influence on the algorithm's performance. Lower mutation rates were found to allow the algorithm to refine the existing solutions more effectively, leading to faster convergence. However, too low a mutation rate risks getting stuck in local optima. The optimal mutation rate may vary depending on the specifics of the problem and initial population.

Similarly, the population size and elite size had a considerable impact on the algorithm's performance. Larger populations and elite sizes allow for more diversity in the population, increasing the likelihood of finding optimal or near-optimal solutions. However, they also increase the computational cost of the algorithm.

The stochastic nature of the GA was also evaluated. Despite using the same parameters and same location coordinates, the algorithm's runs yielded different results. The improvements varied around a mean, with a standard deviation representing the variability in the GA's performance. This stochastic behavior of the GA is an inherent property, resulting from the random operations (like selection, crossover, and mutation) that it utilizes.

In conclusion, the genetic algorithm is a powerful and flexible search and optimization tool, especially useful in solving complex problems like the TSP. The choice of GA parameters significantly influences its performance and computational cost. Therefore, careful tuning of these parameters is essential for obtaining optimal or near-optimal solutions efficiently. Moreover, understanding and managing the stochastic nature of the GA is critical for its effective application. Future work may include the application of the GA to more complex or larger-scale TSPs, as well as the incorporation of additional operations or heuristics to further enhance its performance.

6 References

[1]. A.E.Eiben, J.E. Smith 2015 Introduction to Evolutionary Computing (Natural Computing Series).

[2]. Xinjie Yu · Mitsuo 2010 Gen Introduction to Evolutionary Algorithms

[3]. Davis, Lawrence (1991). Handbook of genetic algorithms.

[4]. J.H. Holland. Adaption in Natural and Artificial Systems. MIT Press, Cambridge, MA, 1992. 1st edition: 1975, The University of Michigan Press, Ann Arbor.

7 Figures