Genetic Algorithm to Solve the Traveling Salesman Problem

Table of Contents

Background

Genetic algorithms (GAs) are a type of evolutionary algorithm as they are useful “for solving optimization problems in which the solution consists of a large number of permutations or choices” (Hurbans, 2020). The life cycle of a GA has eight stages: (1) encode solution space (2) set algorithm parameters (3) create initial population (4) measure fitness of individuals (5) select parents (6) reproduce offspring (7) populate next generation (8) measure fitness of individuals. In other words, from a population, the fitness of individuals is measured and parents are selected to reproduce based on their fitness score. The next generation is then created by choosing which offspring and individuals from the previous population survive. The cycle continues until a stopping condition is reached.

The traveling salesman problem (TSP) is an algorithmic problem that attempts to find the shortest path between several cities. In the TSP, a salesman needs to visit all cities in a list. The salesman must avoid cities that have already been visited and must then return to the starting city.

Answers to Questions

1. **How were the cities and distances represented (as a data structure)?**

   The cities and distances were represented with a class. Within the class, cities were represented using a matrix with (x, y) coordinates. The distances were calculated using the Pythagorean theorem.

2. **How did you encode the solution space?**

   We set the population to be all the routes. We rank these routes based on their fitness score, which is the inverse of the route distance. Use tournament selection for the parents of the next generation. We set the crossover to be a random selection of the parent set, and appending the missing data from the second parent.

3. **How did you handle the creation of the initial population?**

   We use a function createRoute() to create a random route between the cities. A separate function, createPopulation() loops through createRoute(). This also lets us choose the population size.

4. **How did you compute the fitness score?**

   Fitness score is treated as route distance. Therefore, a shorter distance leads to a higher fitness score.

   distance = sqrt((xDis ** 2) + (yDis ** 2))

   routeDistance() calculates the total route distance. So:

   fitness = 1 / routeDistance()

5. **Which parent selection strategy did you use? Why?**

   The code uses Tournament Selection strategy. In this strategy, a fixed number of individuals (specified by the `tournament_size` parameter) are randomly selected from the population, and the one with the best fitness is chosen as a parent for crossover. The tournament selection method is used here because it is a good compromise between exploitation (selecting the best individuals) and exploration (maintaining diversity in the population). It allows some less-fit individuals to be selected, which helps avoid premature convergence on sub-optimal solutions.

6. **Which crossover strategy(ies) did you try? Which one worked out best?**

   The code uses a variant of the Ordered Crossover (OX) strategy. This strategy first selects a subset of the first parent (randomly selects start and end indices), and then fills up the remaining part of the route with the genes from the second parent, in the order they appear, while not duplicating any cities. This strategy is particularly well-suited to the Traveling Salesman Problem (TSP) because it ensures that all cities are visited exactly once, maintaining the integrity of the solution.

7. **Which mutation strategy(ies) did you try? Which one worked out best?**

   The mutation strategy we used was swapping two cities in a tour. The mutation occurs with a certain probability (mutation_rate: which is inputted when the function genetic_algorithm() is initialized) for each child generated during the crossover process. If the random number is less than the mutation rate then two cities in the child tour are randomly selected and swapped. This mutation helps us to introduce variation in our search space.

8. **Which strategy did you use for populating the next generation? Why?**

   The population of the next generation is completely replaced by the offspring generated by crossover and mutation. Each new individual is created by selecting two parents using tournament selection, performing crossover to create a child, and then possibly mutating that child. This is also known as the "generational" model of genetic algorithms. The reason for using this model here might be that it allows the population to make large leaps across the fitness landscape, potentially accelerating the search for a good solution. However, it also has the risk of losing good solutions rapidly if they don't get selected for reproduction.

9. **Which stopping condition did you use? Why?**

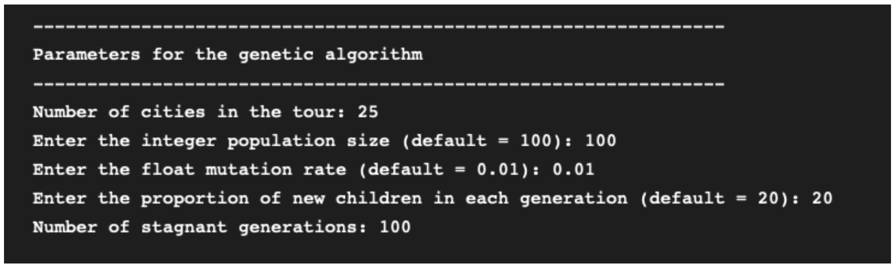
   We set the stopping point at 100 stagnant generations. We check the change between each generation. Once we are deep enough into iterations that we reach a local optimum, we stop once we do not observe significant changes for 100 generations.

10. **What other parameters, design choices, initialization and configuration steps are relevant to your design and implementation?**

    We kept the city selection to 20 out of the 25 total. This introduces slight variety in each execution of the algorithm. We also used tournament selection for the parents of the next generation.

11. **Which (simple) experiments have you run to observe the impact of different design decisions and parameter values? Post their results and your comments.**

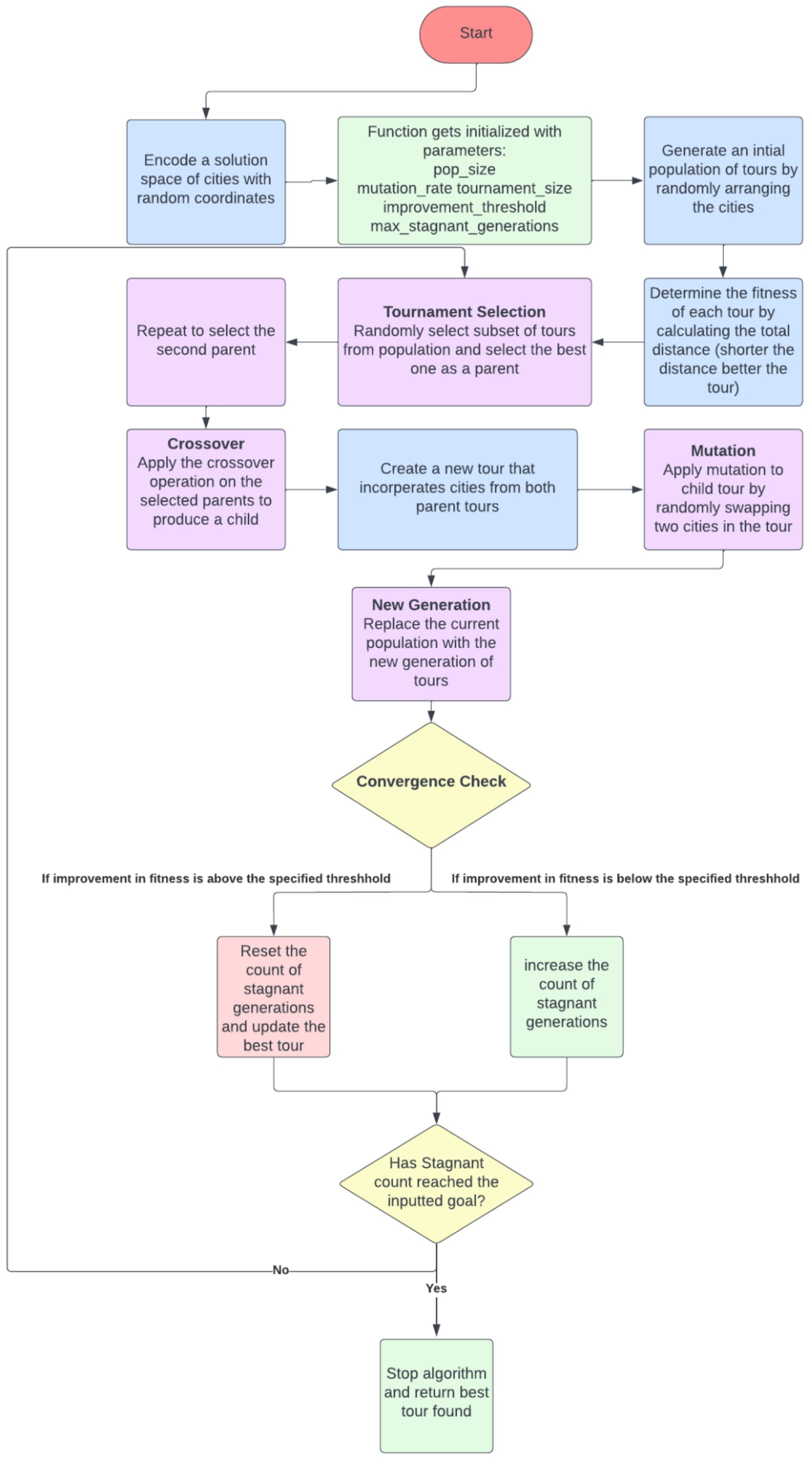
    We ran simple experiments and observed how having different design decisions and parameters values impacted the results calculated by the genetic algorithm. These experiments included altering parameters values such as population size, mutation rate, proportion of new children in each generation as seen in the figure below.



```
--------------------------------------------------------------
Parameters for the genetic algorithm
--------------------------------------------------------------
Number of cities in the tour: 25
Enter the integer population size (default = 100): 100
Enter the float mutation rate (default = 0.01): 0.01
Enter the proportion of new children in each generation (default = 20): 20
Number of stagnant generations: 100
```

Solution Design

Figure 1. Solution design created using lucidchart.com.


Start
Encode a solution space of cities with random coordinates
Function gets initialized with parameters: pop_size mutation_rate tournament_size improvement_threshold max_stagnant_generations
Generate an intial population of tours by randomly arranging the cities
Determine the fitness of each tour by calculating the total distance (shorter the distance better the tour)
Tournament Selection Randomly select subset of tours from population and select the best one as a parent
Repeat to select the second parent
Crossover Apply the crossover operation on the selected parents to produce a child
Create a new tour that incorperates cities from both parent tours
Mutation Apply mutation to child tour by randomly swapping two cities in the tour
New Generation Replace the current population with the new generation of tours
Convergence Check
If improvement in fitness is above the specified threshhold
If improvement in fitness is below the specified threshhold
Reset the count of stagnant generations and update the best tour
increase the count of stagnant generations
Has Stagnant count reached the inputted goal?
No
Yes
Stop algorithm and return best tour found

Implementation: Pseudocode

```
class City:
    #city  represented by a point in 2D space with basic coordinates bc im
not smart enough to do trig
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Method: calculate distance between two cities
    def distance(self, city):
        return ((self.x - city.x)**2 + (self.y - city.y)**2)**0.5

class Tour:
    # tour is a list of cities
    def __init__(self, cities):
        self.cities = cities

    # Method: calculate total distance of the selected tour
    def total_distance(self):
        return sum(self.cities[i].distance(self.cities[i + 1]) for i in
range(len(self.cities) - 1)) + self.cities[-1].distance(self.cities[0])

def genetic_algorithm(cities, pop_size, mutation_rate, tournament_size,
improvement_threshold, max_stagnant_generations):

  population = initialize_population(cities, pop_size)
    stagnant_generations = 0
    best_tour = min(population, key=lambda tour: tour.total_distance())
    last_best_fitness = best_tour.total_distance()

    while stagnant_generations < max_stagnant_generations:
        new_population = []

        for _ in range(pop_size):
            parent1 = tournament_selection(population, tournament_size)
            parent2 = tournament_selection(population, tournament_size)
            child = crossover(parent1, parent2)
            mutate(child, mutation_rate)
            new_population.append(child)

        population = new_population
        current_best_tour = min(population, key=lambda tour:
tour.total_distance())
        current_best_fitness = current_best_tour.total_distance()

        if last_best_fitness - current_best_fitness >
improvement_threshold:
            best_tour = current_best_tour
            last_best_fitness = current_best_fitness
            stagnant_generations = 0
        else:
            stagnant_generations += 1

    return best_tour

def initialize_population(cities, pop_size):
    #creating pop_size tours with randomly shuffled cities
    return [Tour(random.shuffle(cities)) for _ in range(pop_size)]

def tournament_selection(population, tournament_size):
    # Select tournament_size randomly chosen tours from the population and
return the best one
    return min(random.sample(population, tournament_size), key=lambda
tour: tour.total_distance())

def crossover(parent1, parent2):
    # implementation of ordered crossover
    # random subset of parent1, keep the order of cities in this subset,
    # then fills the remainder of the route with the cities from parent2
in the order they appear in parent2
    subset_start, subset_end =
sorted(random.sample(range(len(parent1.cities)), 2))
    child_cities = parent1.cities[subset_start:subset_end]
    child_cities += [city for city in parent2.cities if city not in
child_cities]
    return Tour(child_cities)

def mutate(tour, mutation_rate):
    # Swap two cities in the tour with a probability of mutation_rate
    if random.random() < mutation_rate:
        i, j = random.sample(range(len(tour.cities)), 2)
        tour.cities[i], tour.cities[j] = tour.cities[j], tour.cities[i]

# Function to perform crossover (or mating)
def crossover(parent1, parent2):
    # Select a random subset of genes from the first parent
    subset_start, subset_end = sorted(random.sample(range(1,
len(parent1.cities) - 1), 2))
    child_cities = [parent1.cities[0]] +
parent1.cities[subset_start:subset_end]
    # Add any missing genes from the second parent in order
    for city in parent2.cities:
        if city not in child_cities and city != parent1.cities[0]:
            child_cities.append(city)
    # Make sure the child route starts and ends at the same city
    child_cities += [parent1.cities[0]]
    return Tour(child_cities)

# Mutates tour by swapping two cities
def mutate(tour, mutation_rate):
    if random.random() < mutation_rate:
        tour_cities = tour.cities[1:-1]  # exclude first and last city
        i, j = random.sample(range(len(tour_cities)), 2)
        tour_cities[i], tour_cities[j] = tour_cities[j], tour_cities[i]
        # rebuild the tour with the mutated cities
        tour.cities = [tour.cities[0]] + tour_cities + [tour.cities[0]]
```

Implementation: Program Input and Output

The full program can be accessed on Google Colab with the following link:

https://colab.research.google.com/drive/1yzKkKLjPIuQtf5F7QStqbU86HzKAtB6Z?usp=sharing.

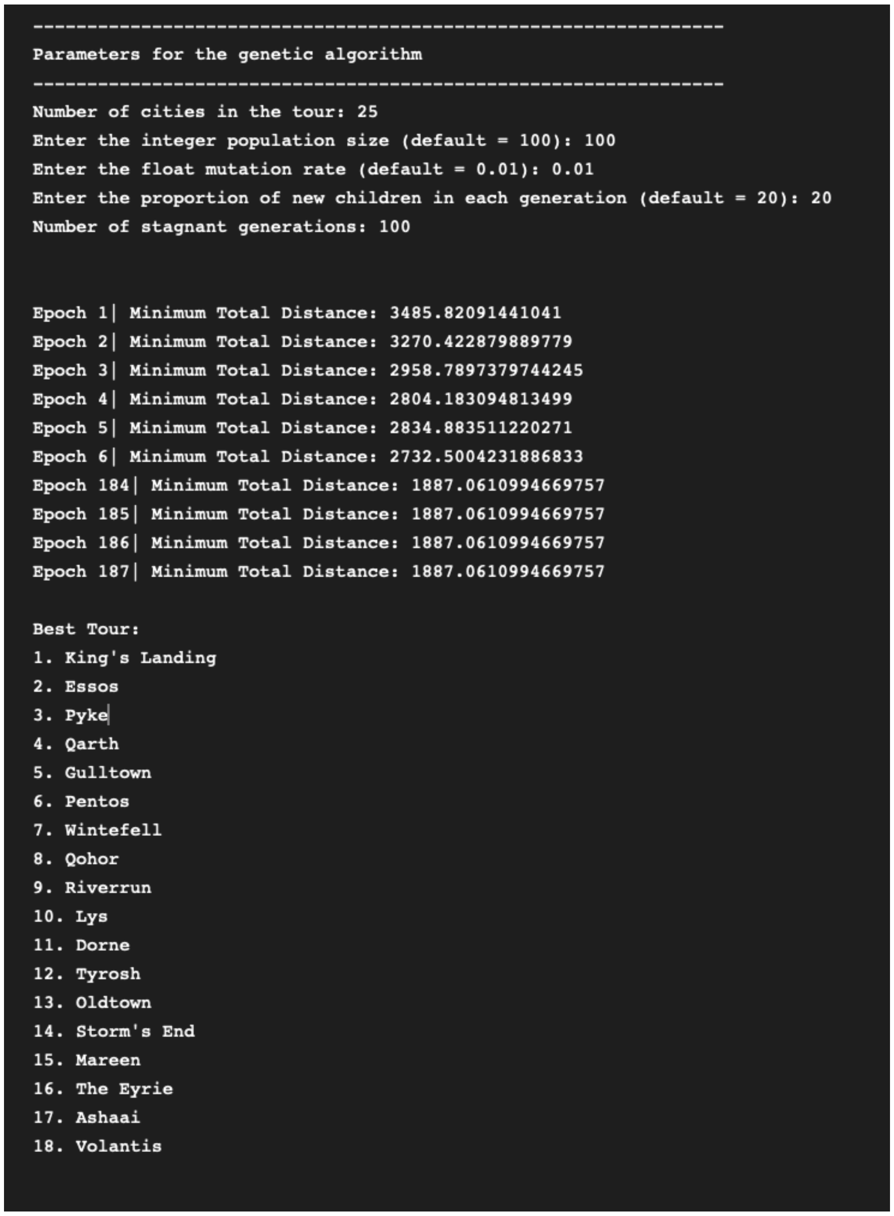
Figure 2. Input for the TSP. This includes a list of the names of 25 cities. These cities will be assigned random coordinates.

```
#list of city names

city_names = [

    "Wintefell", "King's Landing", "Braavos", "Pentos", "Riverrun",
"Dorne", "Highgarden",

    "Lannisport", "The Vale", "The Eyrie", "Mareen", "Volantis", "Qarth",
"Ashaai", "Pyke",

    "Oldtown", "Storm's End", "Gulltown", "White Harbor", "Qohor", "Lys",
"Lorath",

    "Stormlands", "Essos", "Tyrosh"

]

#list of cities with random coordinates

city_list = [City(name, random.uniform(-200.0, 200.0),
random.uniform(-200.0, 200.0)) for name in city_names]
```

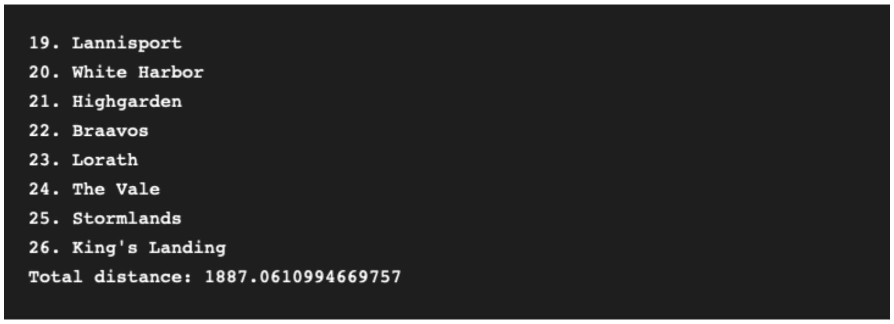
Figure 3. Output format of the TSP.

```
# Print the names of the cities in the best tour and its total distance

print("Best tour:", best_tour)

print("City names in the best tour:")

for city in best_tour.cities:

    print(city.name)

print("Total distance:", best_tour.total_distance())
```

Figure 4. Run number 1: Input and Output. Run progress was truncated due to space constraints on this paper.

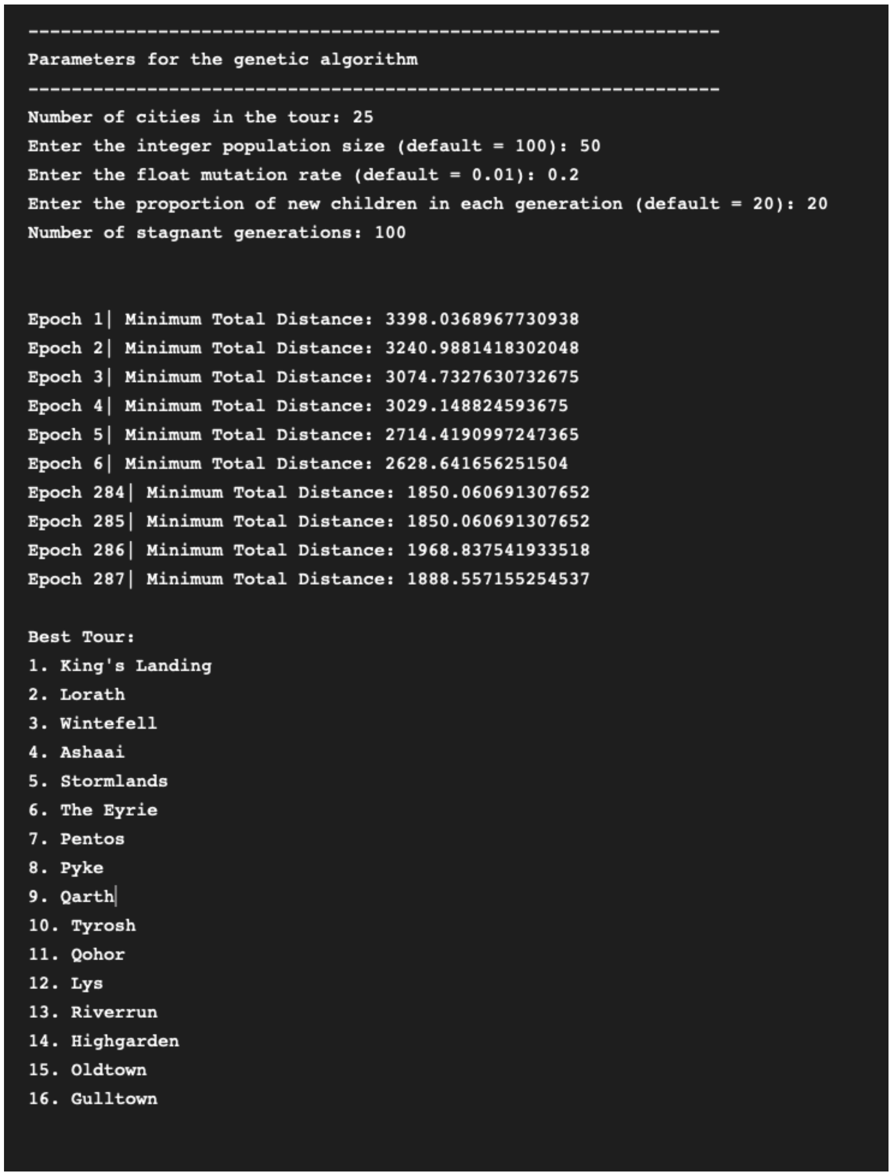


```
----------------------------------------------------------------
Parameters for the genetic algorithm
----------------------------------------------------------------
Number of cities in the tour: 25
Enter the integer population size (default = 100): 100
Enter the float mutation rate (default = 0.01): 0.01
Enter the proportion of new children in each generation (default = 20): 20
Number of stagnant generations: 100


Epoch 1| Minimum Total Distance: 3485.82091441041
Epoch 2| Minimum Total Distance: 3270.422879889779
Epoch 3| Minimum Total Distance: 2958.7897379744245
Epoch 4| Minimum Total Distance: 2804.183094813499
Epoch 5| Minimum Total Distance: 2834.883511220271
Epoch 6| Minimum Total Distance: 2732.5004231886833
Epoch 184| Minimum Total Distance: 1887.0610994669757
Epoch 185| Minimum Total Distance: 1887.0610994669757
Epoch 186| Minimum Total Distance: 1887.0610994669757
Epoch 187| Minimum Total Distance: 1887.0610994669757

Best Tour:
1. King's Landing
2. Essos
3. Pyke
4. Qarth
5. Gulltown
6. Pentos
7. Wintefell
8. Qohor
9. Riverrun
10. Lys
11. Dorne
12. Tyrosh
13. Oldtown
14. Storm's End
15. Mareen
16. The Eyrie
17. Ashaai
18. Volantis
```

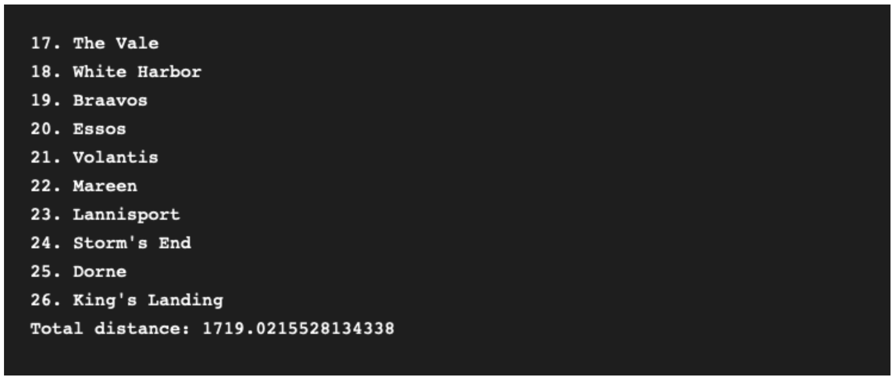


```
19. Lannisport
20. White Harbor
21. Highgarden
22. Braavos
23. Lorath
24. The Vale
25. Stormlands
26. King's Landing
Total distance: 1887.0610994669757
```

Figure 5. Run number 2: Input and Output. Run progress was truncated due to space constraints on this paper.

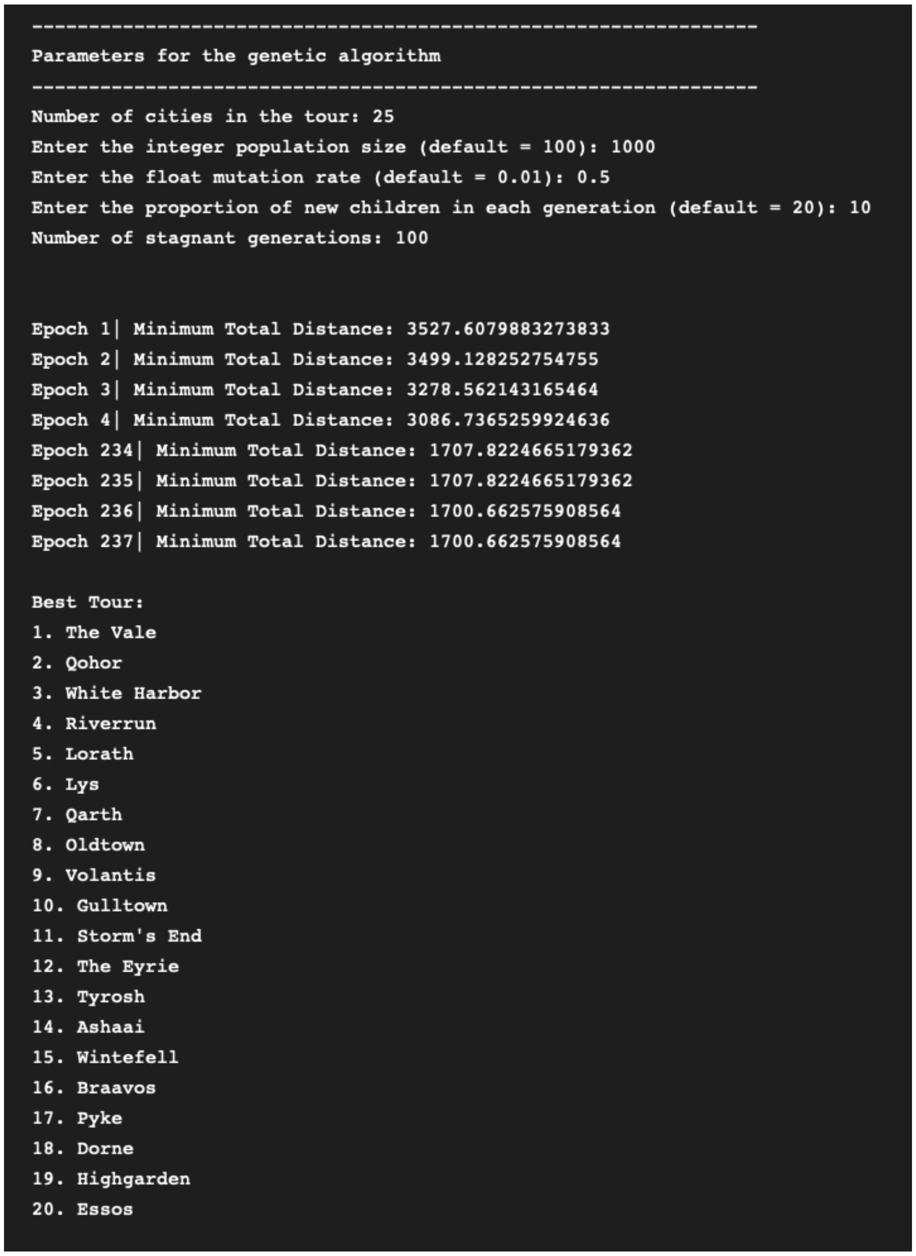


```
----------------------------------------------------------------
Parameters for the genetic algorithm
----------------------------------------------------------------
Number of cities in the tour: 25
Enter the integer population size (default = 100): 50
Enter the float mutation rate (default = 0.01): 0.2
Enter the proportion of new children in each generation (default = 20): 20
Number of stagnant generations: 100


Epoch 1| Minimum Total Distance: 3398.0368967730938
Epoch 2| Minimum Total Distance: 3240.9881418302048
Epoch 3| Minimum Total Distance: 3074.7327630732675
Epoch 4| Minimum Total Distance: 3029.148824593675
Epoch 5| Minimum Total Distance: 2714.4190997247365
Epoch 6| Minimum Total Distance: 2628.641656251504
Epoch 284| Minimum Total Distance: 1850.060691307652
Epoch 285| Minimum Total Distance: 1850.060691307652
Epoch 286| Minimum Total Distance: 1968.837541933518
Epoch 287| Minimum Total Distance: 1888.557155254537

Best Tour:
1. King's Landing
2. Lorath
3. Wintefell
4. Ashaai
5. Stormlands
6. The Eyrie
7. Pentos
8. Pyke
9. Qarth
10. Tyrosh
11. Qohor
12. Lys
13. Riverrun
14. Highgarden
15. Oldtown
16. Gulltown
```

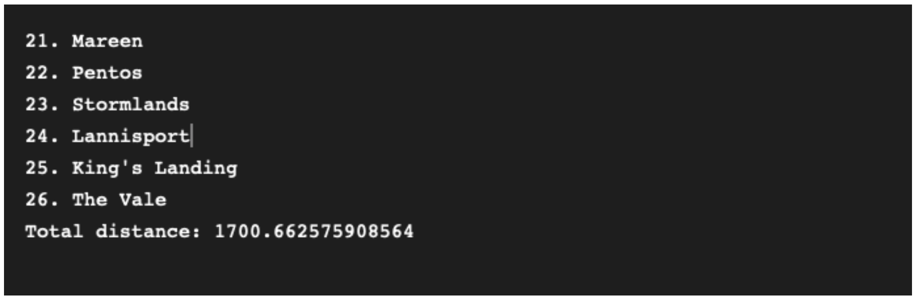


```
17. The Vale
18. White Harbor
19. Braavos
20. Essos
21. Volantis
22. Mareen
23. Lannisport
24. Storm's End
25. Dorne
26. King's Landing
Total distance: 1719.0215528134338
```

Figure 6. Run number 3: Input and Output. Run progress was truncated due to space constraints on this paper.

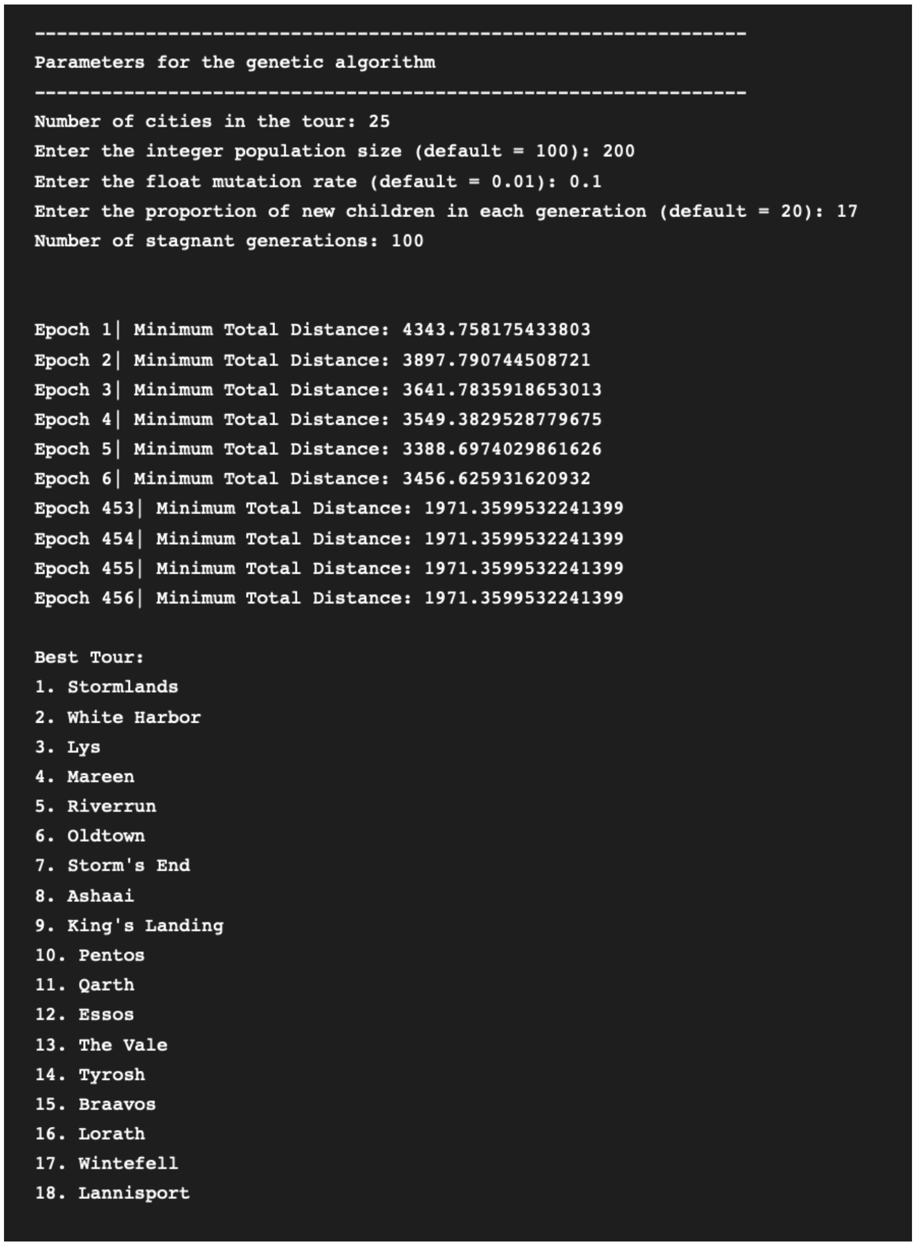


```
----------------------------------------------------------------
Parameters for the genetic algorithm
----------------------------------------------------------------
Number of cities in the tour: 25
Enter the integer population size (default = 100): 1000
Enter the float mutation rate (default = 0.01): 0.5
Enter the proportion of new children in each generation (default = 20): 10
Number of stagnant generations: 100


Epoch 1| Minimum Total Distance: 3527.6079883273833
Epoch 2| Minimum Total Distance: 3499.128252754755
Epoch 3| Minimum Total Distance: 3278.562143165464
Epoch 4| Minimum Total Distance: 3086.7365259924636
Epoch 234| Minimum Total Distance: 1707.8224665179362
Epoch 235| Minimum Total Distance: 1707.8224665179362
Epoch 236| Minimum Total Distance: 1700.662575908564
Epoch 237| Minimum Total Distance: 1700.662575908564

Best Tour:
1. The Vale
2. Qohor
3. White Harbor
4. Riverrun
5. Lorath
6. Lys
7. Qarth
8. Oldtown
9. Volantis
10. Gulltown
11. Storm's End
12. The Eyrie
13. Tyrosh
14. Ashaai
15. Wintefell
16. Braavos
17. Pyke
18. Dorne
19. Highgarden
20. Essos
```

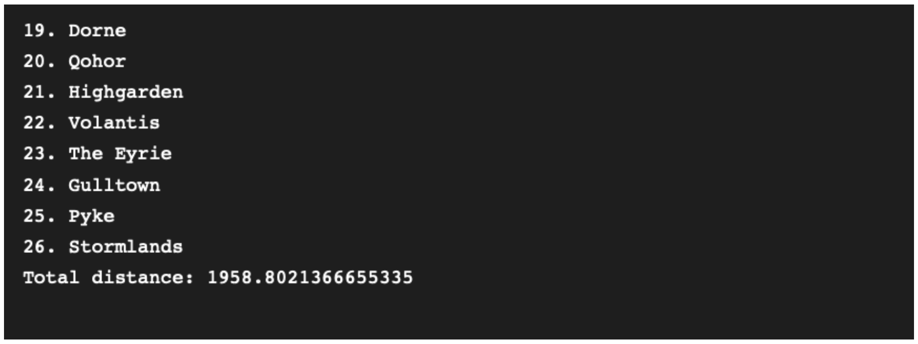


```
21. Mareen
22. Pentos
23. Stormlands
24. Lannisport
25. King's Landing
26. The Vale
Total distance: 1700.662575908564
```

Figure 7. Run number 4: Input and Output. Run progress was truncated due to space constraints on this paper.

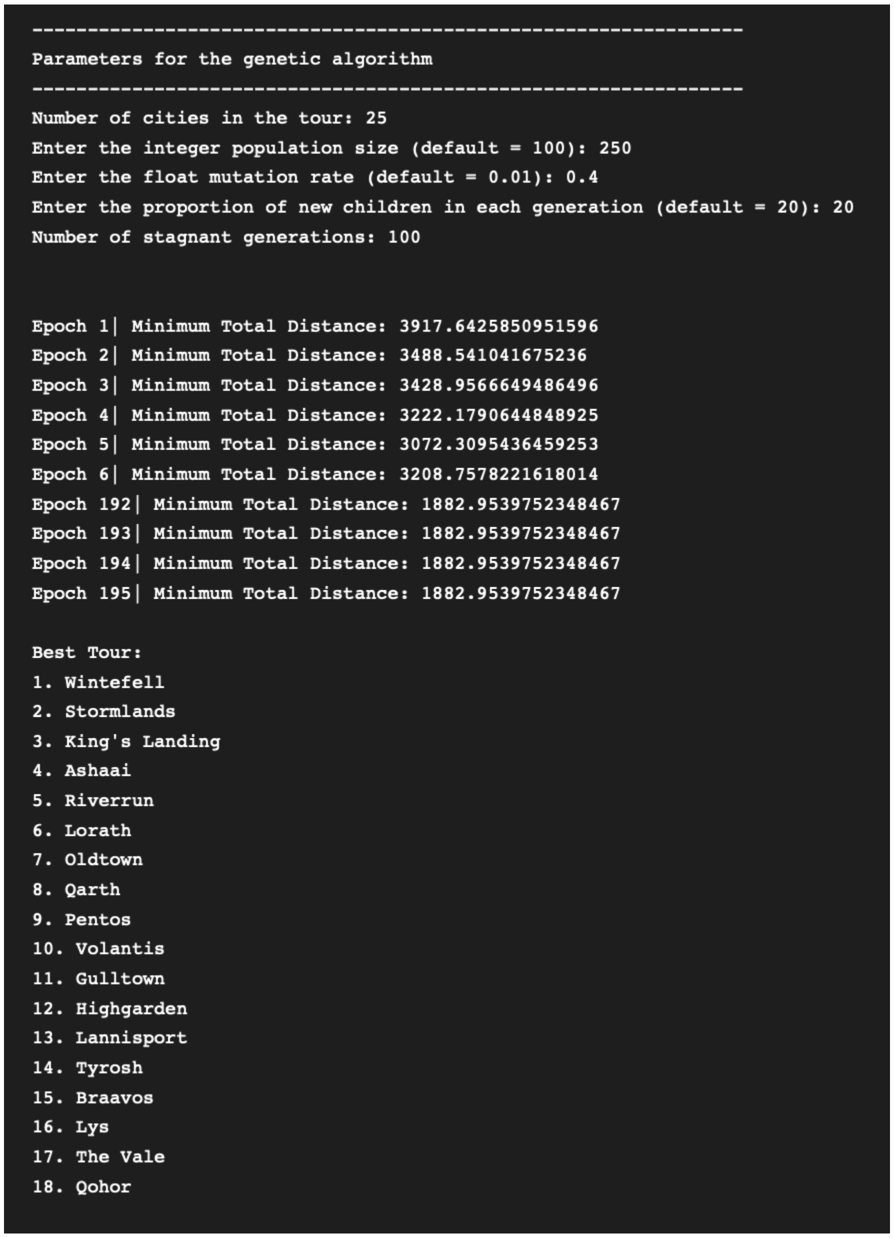


```
----------------------------------------------------------------
Parameters for the genetic algorithm
----------------------------------------------------------------
Number of cities in the tour: 25
Enter the integer population size (default = 100): 200
Enter the float mutation rate (default = 0.01): 0.1
Enter the proportion of new children in each generation (default = 20): 17
Number of stagnant generations: 100


Epoch 1| Minimum Total Distance: 4343.758175433803
Epoch 2| Minimum Total Distance: 3897.790744508721
Epoch 3| Minimum Total Distance: 3641.7835918653013
Epoch 4| Minimum Total Distance: 3549.3829528779675
Epoch 5| Minimum Total Distance: 3388.6974029861626
Epoch 6| Minimum Total Distance: 3456.625931620932
Epoch 453| Minimum Total Distance: 1971.3599532241399
Epoch 454| Minimum Total Distance: 1971.3599532241399
Epoch 455| Minimum Total Distance: 1971.3599532241399
Epoch 456| Minimum Total Distance: 1971.3599532241399

Best Tour:
1. Stormlands
2. White Harbor
3. Lys
4. Mareen
5. Riverrun
6. Oldtown
7. Storm's End
8. Ashaai
9. King's Landing
10. Pentos
11. Qarth
12. Essos
13. The Vale
14. Tyrosh
15. Braavos
16. Lorath
17. Wintefell
18. Lannisport
```

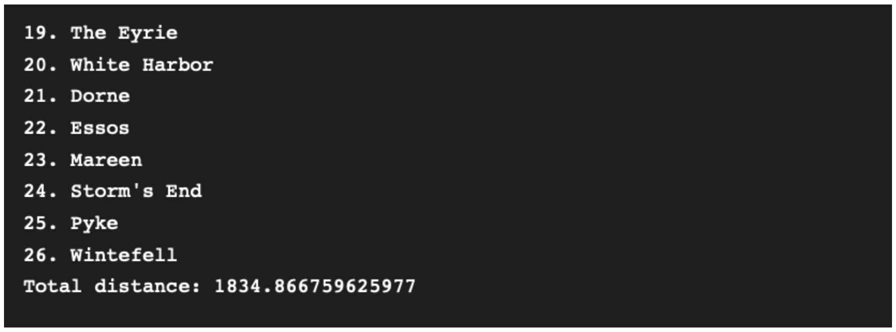


```
19. Dorne
20. Qohor
21. Highgarden
22. Volantis
23. The Eyrie
24. Gulltown
25. Pyke
26. Stormlands
Total distance: 1958.8021366655335
```

Figure 8. Run number 5: Input and Output. Run progress was truncated due to space constraints on this paper.



```
----------------------------------------------------------------
Parameters for the genetic algorithm
----------------------------------------------------------------
Number of cities in the tour: 25
Enter the integer population size (default = 100): 250
Enter the float mutation rate (default = 0.01): 0.4
Enter the proportion of new children in each generation (default = 20): 20
Number of stagnant generations: 100


Epoch 1| Minimum Total Distance: 3917.6425850951596
Epoch 2| Minimum Total Distance: 3488.541041675236
Epoch 3| Minimum Total Distance: 3428.9566649486496
Epoch 4| Minimum Total Distance: 3222.1790644848925
Epoch 5| Minimum Total Distance: 3072.3095436459253
Epoch 6| Minimum Total Distance: 3208.7578221618014
Epoch 192| Minimum Total Distance: 1882.9539752348467
Epoch 193| Minimum Total Distance: 1882.9539752348467
Epoch 194| Minimum Total Distance: 1882.9539752348467
Epoch 195| Minimum Total Distance: 1882.9539752348467

Best Tour:
1. Wintefell
2. Stormlands
3. King's Landing
4. Ashaai
5. Riverrun
6. Lorath
7. Oldtown
8. Qarth
9. Pentos
10. Volantis
11. Gulltown
12. Highgarden
13. Lannisport
14. Tyrosh
15. Braavos
16. Lys
17. The Vale
18. Qohor
```



```
19. The Eyrie
20. White Harbor
21. Dorne
22. Essos
23. Mareen
24. Storm's End
25. Pyke
26. Wintefell
Total distance: 1834.866759625977
```

Limitations

Solving the traveling salesman problem (TSP) using a genetic algorithm (GA) has some limitations. One of these limitations is due to the fact that GAs are “not guaranteed to find the absolute best solution” (Hurbans, 2020). A further limitation of GA’s is due to the parameters of the algorithm, which can be difficult to configure. If the parameters are not configured correctly, the algorithm will not produce diversity at the beginning. The lack of diversity at the beginning can result in the algorithm being unable to produce a global best solution as it might “get stuck in a local best solution” (Hurbans, 2020). This could be remedied by increasing the mutation rate. However, if the mutation rate is too high, too much diversity is introduced to the point where the algorithm would not produce good solutions.

Future Improvements

A future improvement to the solution to the traveling salesman problem (TSP) using genetic algorithms (GAs) implemented would be to create a web-based solution for the problem. This web-based solution would need to be implemented without breaking the baseline solution. Furthermore, this web-based solution could be created with Flask for example.

Conclusion

A solution for the traveling salesman problem (TSP) using genetic algorithms (GAs) was implemented in Python using Jupyter notebooks and Google Colab. The algorithm was run five times, with each run having different parameters. The input and output for each run can be seen above. In each run, the shortest path between the cities or the best tour was calculated along with the total distance traveled.

References

Hurbans, Rishal. *Grokking Artificial Intelligence Algorithms: Understand and Apply the Core Algorithms of Deep Learning and Artificial Intelligence in this Friendly Illustrated Guide Including Exercises and Examples*. Manning, 2020.

Russell, Stuart Jonathan, and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

Stoltz, Eric. “Evolution of a salesman: A complete genetic algorithm tutorial for Python.” *Towards Data Science*, 17 July 2018, https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35. Accessed 16 June 2023.

“Traveling Salesman Problem using Genetic Algorithm.” *GeeksforGeeks*, 21 February 2023, https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/. Accessed 16 June 2023.