

Project 2: Genetic Algorithm

In this report I will discuss my implementation of a genetic algorithm to solve the traveling salesman problem. I will talk about the problem in general, how my code works, and my code's results in that order. I was also the sole member of my team, so I filled every role.

TSP Using a Genetic Algorithm

This first section is mostly questions I have about this assignment and is unprofessional, so feel free to skim over it. First, I'd like to talk about this problem in general and more so the genetic algorithm we use. Unless I misunderstood something, our GA should've had only one chromosome which represented a route traveling through every city. I was wondering why this was how we'd implement a genetic algorithm because it seems like a poor solution. Every time we want to look at a different set of cities, our algorithm starts blind. The algorithm also, never learns anything which I find weird to not implement. Of course, I assume that we kept the scale on the small side since it's our first time coding any GA, but the simplistic nature of the algorithm seems impractical. Maybe I'm getting machine learning confused with genetic algorithms, but I thought chromosomes would be used to make decisions as opposed to being the decision, so that over generations the algorithm's population would learn. Regardless, I still find the genetic algorithm we did implement very interesting, it's just that there were some things I found in my testing that made me question using a genetic algorithm this simple.

My Code

I will list out and answer the questions in the guidelines PDF to streamline this part of the report.

- How were the cities and distances represented (as a data structure): I simply used a list called cityList that held City class variables where each variable had the city's x and y coordinates. City distances are all calculated on the fly when fitness values are calculated. I could've saved every city-to-city distance, but it's unnecessary for a project of this scope and would possibly be impractical for a project of a larger scope. Hopefully, it doesn't run too poorly on whatever computer you use to test it.

My Code

- How did you encode the solution space: The solution space was a 200x200-unit grid, the same one you used as an example at the bottom of the PDF, which has cities randomly placed

on it. For the chromosomes, they are an ordered list with each allele representing an index in cityList mentioned above. The indexes are then used to select 2 cities and calculate their distance apart for every pair of alleles (including the last and first alleles).

- How did you handle the creation of the initial population: I just randomly switched the order of a list containing numbers 0 to populationSize – 1, filling each new member of the population with one of those random lists.

- How did you compute the fitness score: I used the total route length and then added the average of the three longest city-to-city distances to get the fitness score. In this situation, a lower fitness score is a better fitness score. The second part of the equation is intended to punish members of the population that go to further cities. Sometimes it backfires, but never by much and it seems to help the population get on track sooner.

- Which parent selection strategy did you use? Why: I used a ranked roulette wheel because I like it the most. I tried a proportional wheel, but the implementation was messy, so I switched. It basically works the same way as described in the book.

- Which crossover strategy(ies) did you try? Which one worked out best: I just pick two random points of one parent and put the section in between those points into the other. I initially implemented a more set in stone take a random point from the first half and second half, but I switched to two completely random points because it created more diverse children. Each child's parents are chosen randomly from the full set of selected parents. The overlapping alleles are addressed in mutation.

- Which mutation strategy(ies) did you try? Which one worked out best: I do two things. First, I replace doubled up alleles with a random allele that was absent from the generated child. Second, 5/10 children have 2 random alleles swapped, adding needed variation. I only changed 5/10 because it worked better. I just tested out changing that part of the code while writing it and 5/10 seems to work better than 2/10 and 8/10, which I tested previously. Edit: I have switched it back to 8/10 since 5/10 seems to not create enough variation. Also, mutating every element makes the whole population really inconsistent, which is why I switched it back.

- Which strategy did you use for populating the next generation? Why: 1/3 of each generation are the parents of the last generation and 2/3 are new elements. This worked better than 100% new, though I haven't tested any other values. I'm almost certain that somewhere closer to 10% or lower is ideal though.

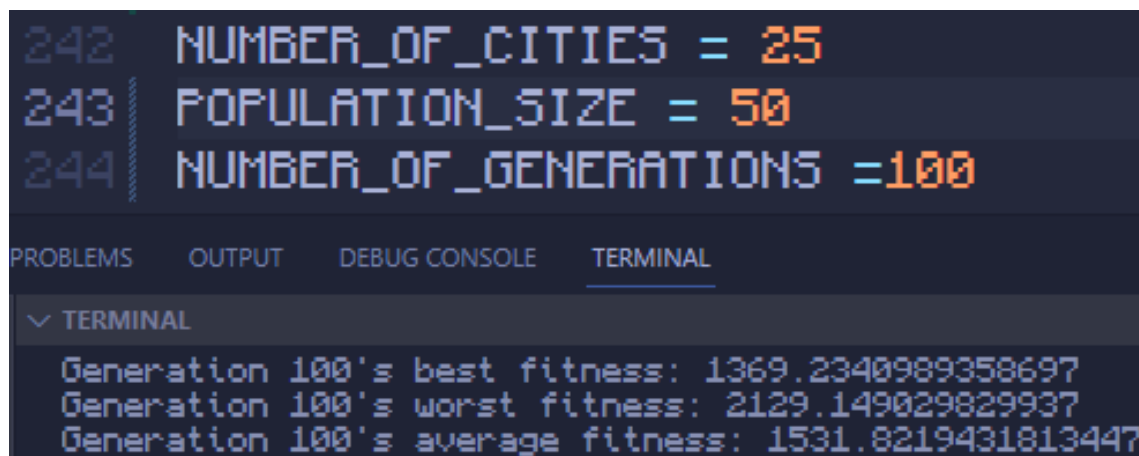
My Code

- Which stopping condition did you use? Why: A set number of generations. It's the easiest to implement and seems to be the most practical.

- What other parameters, design choices, initialization and configuration steps are relevant to your design and implementation: There's nothing special. I definitely designed it strangely. I used classes for everything because I'm still not super used to python, and I'm sure my code shows that. I now know that libraries could've sufficed, though I still don't know when it's better to use one versus the other. The only important thing is that there are no inputs, you have to manually change variables. There are some constants you can change throughout the code for different results, but for changing population size, number of generations, and number of cities, there are three ALL_CAPS variables to change.

Results

Note: Every one of these runs are on the same seed, and therefore the same set of cities. Also, unless specified, they mutate half of all children's chromosomes



The image shows a code editor with a dark theme. The top part displays three lines of code with line numbers 242, 243, and 244. The code sets variables: `NUMBER_OF_CITIES = 25`, `POPULATION_SIZE = 50`, and `NUMBER_OF_GENERATIONS = 100`. Below the code, there is a terminal window with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is active, showing the output of the program at generation 100: `Generation 100's best fitness: 1369.2340989358697`, `Generation 100's worst fitness: 2129.149029829937`, and `Generation 100's average fitness: 1531.8219431813447`.

```
242 NUMBER_OF_CITIES = 25
243 POPULATION_SIZE = 50
244 NUMBER_OF_GENERATIONS = 100
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

✓ TERMINAL

```
Generation 100's best fitness: 1369.2340989358697
Generation 100's worst fitness: 2129.149029829937
Generation 100's average fitness: 1531.8219431813447
```

The best this run got was a fitness of 1369, by chance happening in the last generation, and a total route length of 1278.

```
242 NUMBER_OF_CITIES = 25
243 POPULATION_SIZE = 100
244 NUMBER_OF_GENERATIONS = 1000
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

✓ TERMINAL

```
The Best Route found is: {'chromosome': [20, 10, 16, 8, 1, 15, 14, 12, 13, 11, 9, 7, 6, 5, 4, 3, 2]}
Generation 1000's best fitness: 1321.7773497038702
Generation 1000's worst fitness: 2088.187110282898
Generation 1000's average fitness: 1480.2380254727013
```

This run increased the population size and number of generations significantly. The best this run got was a fitness of 1321 and a total route length of 1209. Not a large improvement, but a steady one in every category.

```
242 NUMBER_OF_CITIES = 25
243 POPULATION_SIZE = 1000
244 NUMBER_OF_GENERATIONS = 100
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

✓ TERMINAL

```
The Best Route found is: {'chromosome': [11, 23, 13, 12, 14, 15, 1, 9, 7, 6, 5, 4, 3, 2, 8, 10]}
Generation 100's best fitness: 1007.6722357885776
Generation 100's worst fitness: 2171.492556369139
Generation 100's average fitness: 1198.2326618345496
```

This run switched the number of generations with the population size. The best this run got was a fitness of 991 and a total route length of 926. This is a vast improvement, showing that having a larger population leads to more successful reproduction and mutations. It is also possible that the population size has an even larger affect because of how I allow every parent into the next generation.

```
242  NUMBER_OF_CITIES = 25
243  POPULATION_SIZE = 1000
244  NUMBER_OF_GENERATIONS = 1000

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

▼ TERMINAL

The Best Route found is: {'chromosome': [11, 23, 13, 4, 19, 2, 10, 24, 12, 18, 22, 14, 16, 20, 21, 17, 15, 3, 25, 6, 8, 7, 5, 9, 1]
Generation 1000's best fitness: 991.8585791187206
Generation 1000's worst fitness: 2029.896668810824
Generation 1000's average fitness: 1139.4360945387662
```

This run is very intriguing for one reason, the best element of the last run stayed for at least most of the generations if not all of them. Also, there were no improvements to the best run over 900 additional generations. This hasn't happened before, so I'm a little confused. It's possible that a lot of the population ended up just imitating that one element, but I'm not sure. Though, the lack of improvements made me switch my mutation strategy back to 8/10 children getting mutated to increase variation.

```
242  NUMBER_OF_CITIES = 25
243  POPULATION_SIZE = 1000
244  NUMBER_OF_GENERATIONS = 1000

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

▼ TERMINAL

Generation 900's average fitness: 1355.1858285684518
The Best Route found is: {'chromosome': [15, 22, 14, 16, 20, 21, 17, 15, 3, 25, 6, 8, 7, 5, 9, 1]
Generation 1000's best fitness: 925.7981334679969
Generation 1000's worst fitness: 2649.0156613317736
Generation 1000's average fitness: 1414.2558886397846
```

The best this run got was a fitness of 925 and a total route length of 869. This mutated 8/10 children and seems to have led to the same problem the last test had. So, and improvement? I didn't notice that part till writing this out, but I guess that is a problem with my logic.

So, what can we see from these test runs. First, having a fairly large population of at least 100 is desirable to get a good selection of parents. Second, since I don't have a GUI, I can't

really tell if these routes actually look sensible. It actually seems like a large setback in terms of testing. Third and most importantly, the ways each part of this algorithm is implemented can greatly affect how the GA performs. For instance, how I make the next generation/do mutations works well, but only to a point. I honestly wonder how many members of generation 1000-10000 are just the same route.

One thing before I end this report, after some extra testing, I also believe I might be picking too much, if not way too much, of the population to be parents. I initially started with a population of 18, but with a far larger population, I think I'm diluting the parent pool. I'm not certain by any means, but I think that's what is causing the whole population to converge onto one route. Also, I allow for duplicate parents, which absolutely propagates that behavior.

In conclusion, this project was very cool and taught me a lot about GA's. I'm very curious to learn about more complicated genetic algorithms in the future. The only downside, outside of wanting to add more ways for the algorithm to "learn", is that I didn't get to compare my algorithm to someone else's. I don't really have a good grasp of how good my algorithm really is (especially without a GUI).