# Genetic Programming: Travelling Sales Man

**Harsha Vardhan, Khurdula**

**CS59000 Expert Systems**

**Hajiarbabi, Mehrdad.**

**October, 12th 2023**

## Introduction

The current task at hand is to try and solve Travelling Sales Man/Person problem. It is an algorithmic problem that involves finding the shortest route between set of points. A sales person said to travel n cities, with the ability of visiting every city just once and reaching destination city at the end of his travel. The primary objective is to find the shortest path between the start and end cities. This has to be addressed using Genetic Programming.

## Approach

My approach to this problem is to develop a genetic algorithm that attempts to converge to the best solution possible for each generation. Here are the parameters that the algorithm accepts:

- Cities: Count of total number cities.
- Population size: Total routes that you might want to consider. (Count of combinations of routes possible.
- Iterations: Total number iterations/generations that the algorithm has to run for.

Expected output: The expected output for this algorithm is the best path possible for the travel. And the best distance at end of all possible generations/iterations. Along with overall performance.

## Algorithm

The algorithm I have in place is pretty simple:

---

*Algorithm TSP():*

*Params = get_user_input() //user input is obtained at constructor call.*

*Distances = create_random_distances() //create a distance matrix between n cities.*

*Population = create_random_population() //create n samples of possible routes.*

*While total_iterations<user_input_iterations: //run this until we complete iterations*

*Population = compute_distances(population) //compute distance of entire path*

```
Population_fitness = compute_fitness(Population) //calculate inverse fitness.
//filter this using random selection.
Selected_population = roulette_wheel(population)
//apply order crossover among two random parent population to create new population.
If selected_population = 0:
    Break //upon random selection it is possible that no children are selected.
Child_population = perform_crossover(population) //create new population.
If child_population = 0:
    Break
//now apply mutation over these routes over child population.
Child_population = apply_mutation(child_population) //internal mutation.
//now compute the fitness for each of the mutated child population
Child_fitness = compute_fitness(child_population)
Stats = compute_results(child_population, child_fitness) //find best paths.
Population = child_population //pass on the torch to newer generations.
Plot_results() //once we iterate through n generations, we must then plot the results.
```

---

**Results**

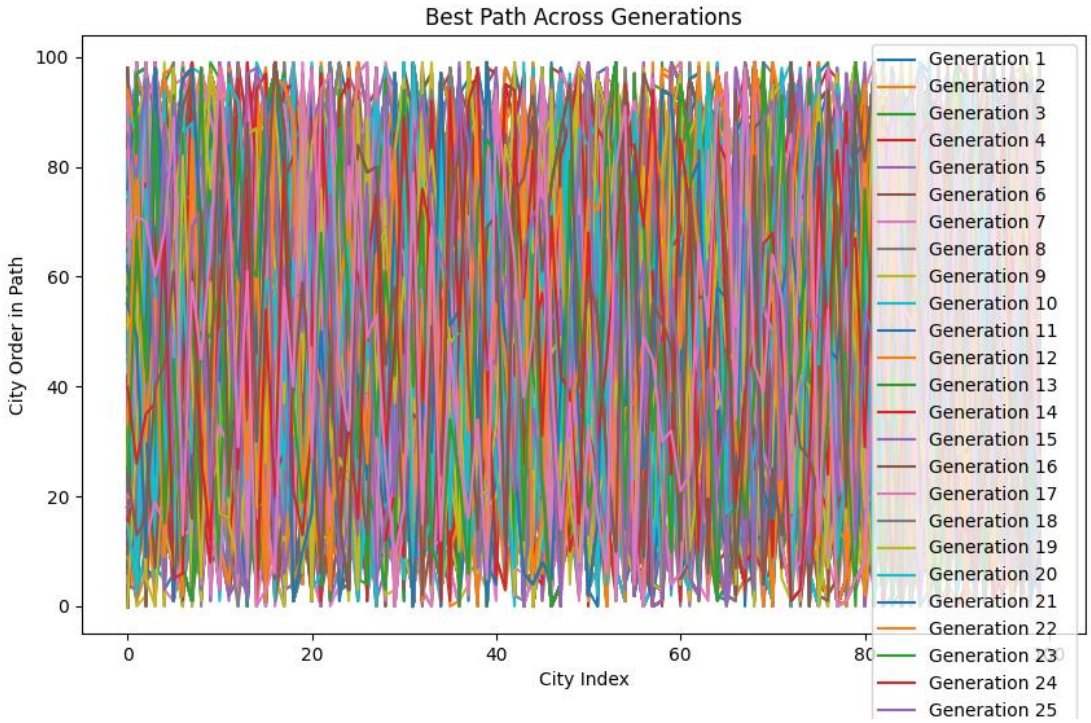
1) First Run:

For the first run we'll have the following parameters involved:

| | |
|---|---|
| **Cities** | 100 |
| **Population Size** | 1000 |
| **Iterations** | 1000 |

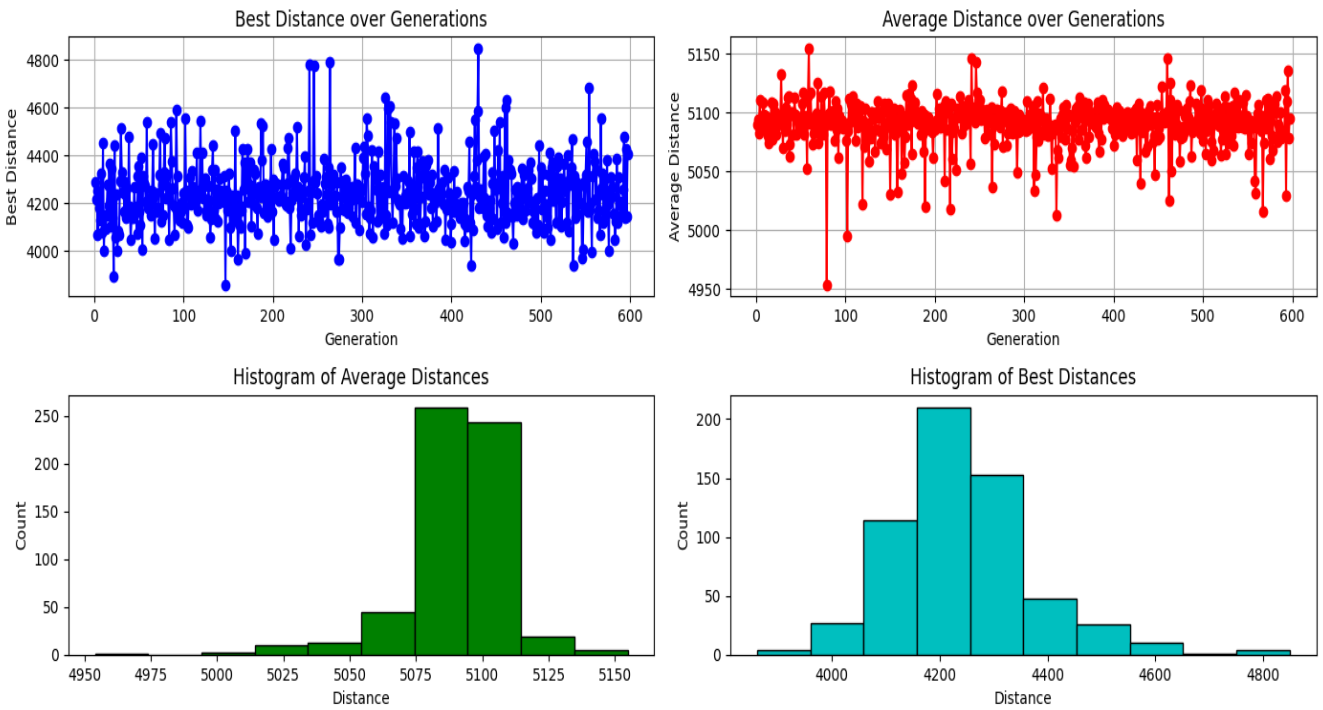
When I ran this problem with these parameters, algorithm ended abruptly at 598 as all children died out, and none of them made it to next generation of 599. However here is the generational evolution of the population, and the solution that we converged to:

To begin with, let's look at all paths traveled:



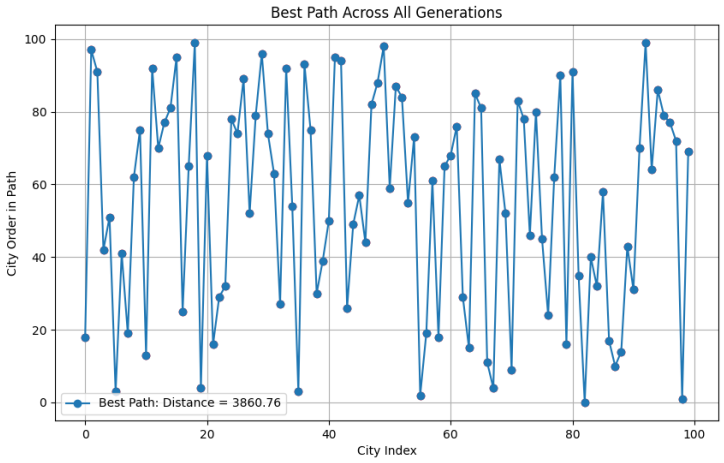

Plot: All paths visualized.

As we can see the plot is pretty messy, we'll look at the overall generational statistics:




For the random distances created for this instance of execution seem to have highest distance of 5150 units and average around 5000 – 5200 mark. There average of best distance is of count 200 for 4200. And only a very small percentage of best distances over generations seems to be less than 4000.

Having said that let's take a look at the best solution:




Plot: Solution for first run. Shortest Distance: 3860.76

Shortest path:

[51, 57, 22, 42, 85, 53, 36, 29, 59, 97, 91, 40, 13, 79, 18, 19, 34, 70, 31, 88, 1, 76, 27, 50, 32, 3, 9, 39, 74, 45, 90, 16, 14, 75, 9, 24, 12, 99, 81, 63, 73, 44, 49, 77, 0, 15, 94, 23, 61, 98, 95, 20, 26, 1, 11, 5, 80, 48, 67, 65, 66, 38, 10, 43, 83, 68, 4, 56, 78, 25, 21, 92, 28, 71, 69, 93, 33, 64, 8, 54, 17, 35, 6, 62, 87, 84, 82, 52, 86, 60, 47, 58, 46, 96, 55, 72, 2, 7, 96, 78]

---

2) Second Run

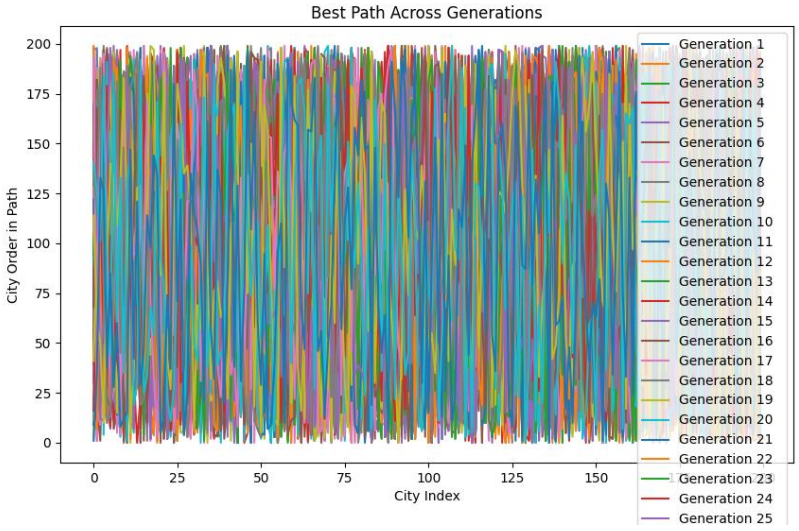
For the second run I want to increase the complexity, let's go with the following params:

| | |
|---|---|
| **Cities** | 200 |
| **Population Size** | 1100 |
| **Iterations** | 1000 |

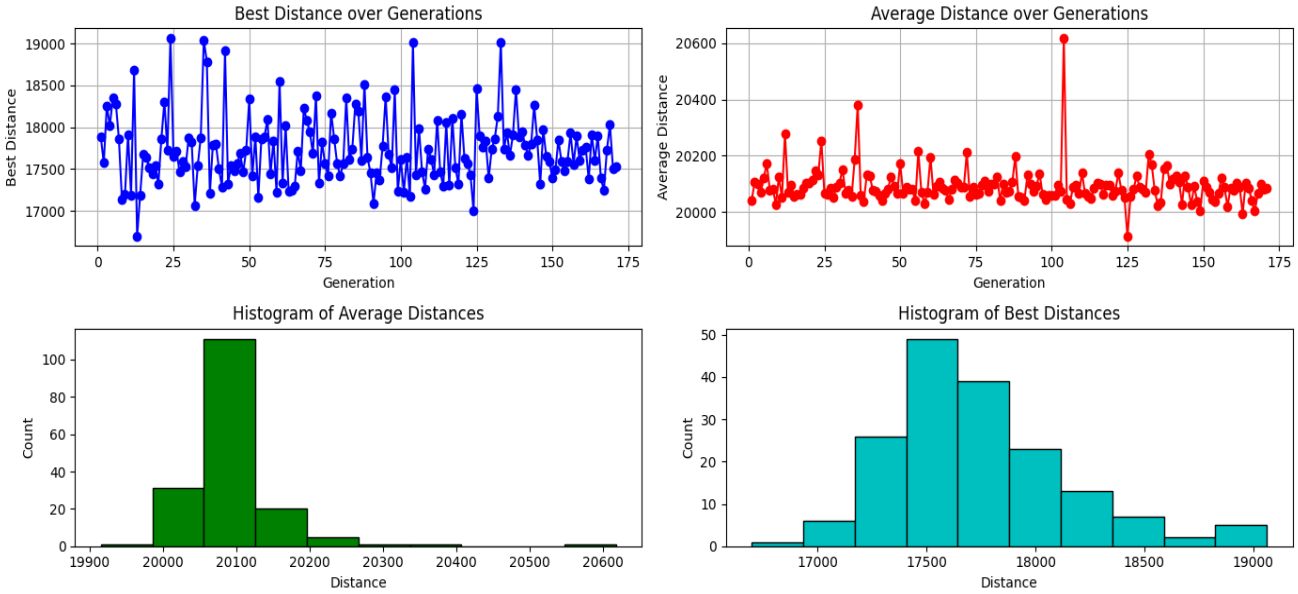
This time I've decided to increase the number of cities and population size to add more variety.

**Immediate observation:** The execution time has gone up, the algorithm ran for 2 minutes on my setup, might take longer depending on hardware resources available of the machine on which this algorithm is ran with aforementioned parameters.

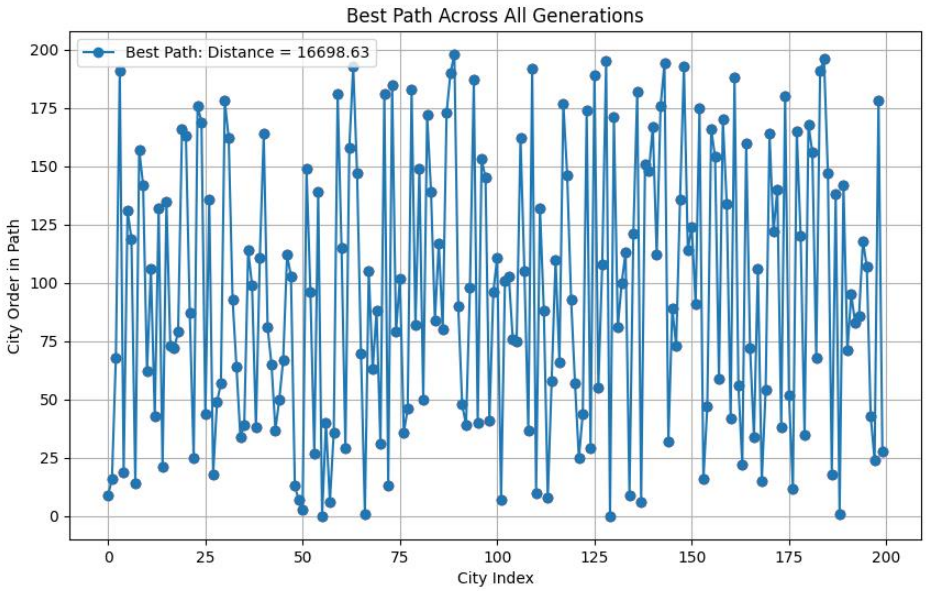
Let's take a look at all paths traversed:



Ah! Not much change in the chaos it is. However this time the algorithm ended at 172nd generation as population died out then. Let's take a look at the statistics for this run:




We can see a lot more fluctuation in the solution over generations. However the average solution of distance revolves around 20000 to 20100 units. Only a very small percentage of solutions have total distance less than 17000 as best solution, let's take a look at the visualization of the best path traversed:



The best path traversed across all generations has a total best distance of: 16698.63

And the path itself travelled between 200 cities is:

[9, 16, 68, 191, 19, 131, 119, 14, 157, 142, 62, 106, 43, 132, 21, 135, 73, 72, 79, 166, 163, 87, 25, 176, 169, 44, 136, 18, 49, 57, 178, 162, 93, 64, 34, 39, 114, 99, 38, 111, 164, 81, 65, 37, 50, 67, 112, 103, 13, 7, 3, 149, 96, 27, 139, 0, 40, 6, 36, 181, 115, 29, 158, 193, 147, 70, 1, 105, 63, 88, 31, 181, 13, 185, 79, 102, 36, 46, 183, 82, 149, 50, 172, 139, 84, 117, 80, 173, 190, 198, 90, 48, 39, 98, 187, 40, 153, 145, 41, 96, 111, 7, 101, 103, 76, 75, 162, 105, 37, 192, 10, 132, 88, 8, 58, 110, 66, 177, 146, 93, 57, 25, 44, 174, 29, 189, 55, 108, 195, 0, 171, 81, 100, 113, 9, 121, 182, 6, 151, 148, 167, 112, 176, 194, 32, 89, 73, 136, 193, 114, 124, 91, 175, 16, 47, 166, 154, 59, 170, 134, 42, 188, 56, 22, 160, 72, 34, 106, 15, 54, 164, 122, 140, 38, 180, 52, 12, 165, 120, 35, 168, 156, 68, 191, 196, 147, 18, 138, 1, 142, 71, 95, 83, 86, 118, 107, 43, 24, 178, 28]

---

One thing that we can observe as it supposed to be and can be redundant to point to but can prove that our algorithm works just fine is that the total best distance increases as we increase the number of cities, even though the distances among these cities are purely randomly generated.

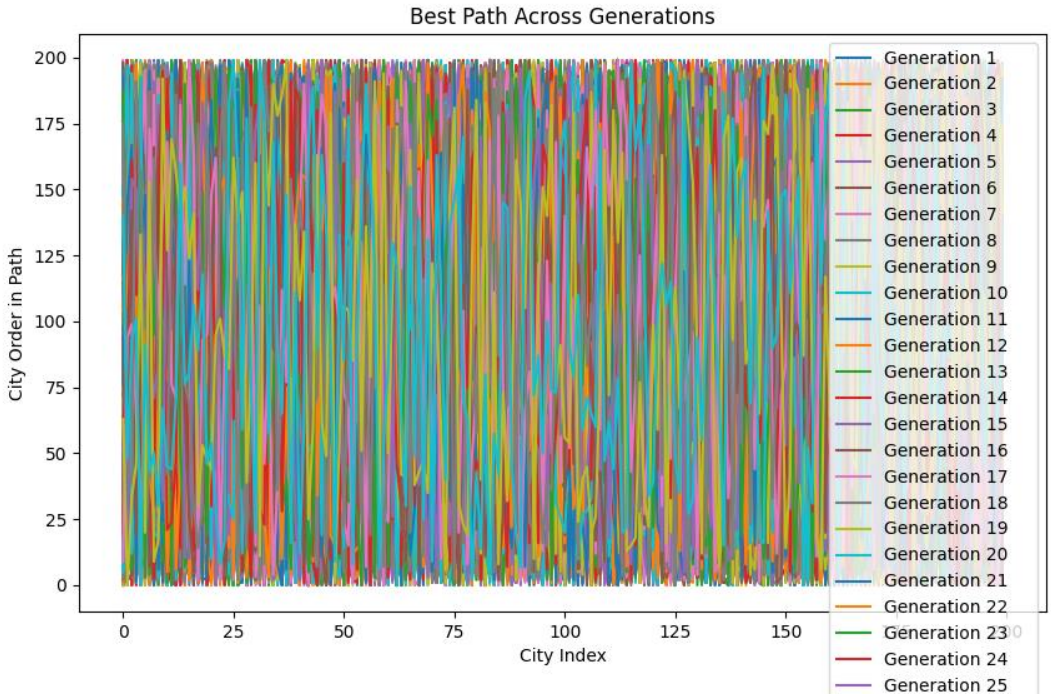
3) Third Run:

For the third run we'll use the following params:

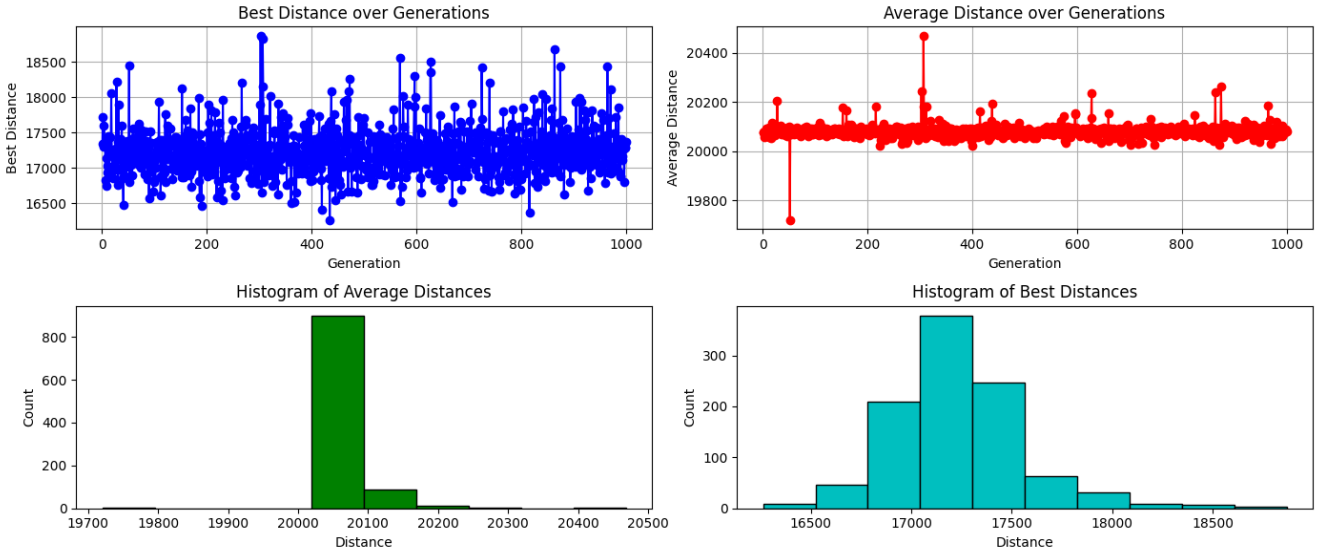
| | |
|---|---|
| **Cities** | 200 |
| **Population Size** | 10000 |
| **Iterations** | 1000 |

This took the highest time out of all iterations of this algorithm, it ran for an entire of 1000 iterations/generations without dying out. And took about 15 mins to do reach to 1000 iterations.

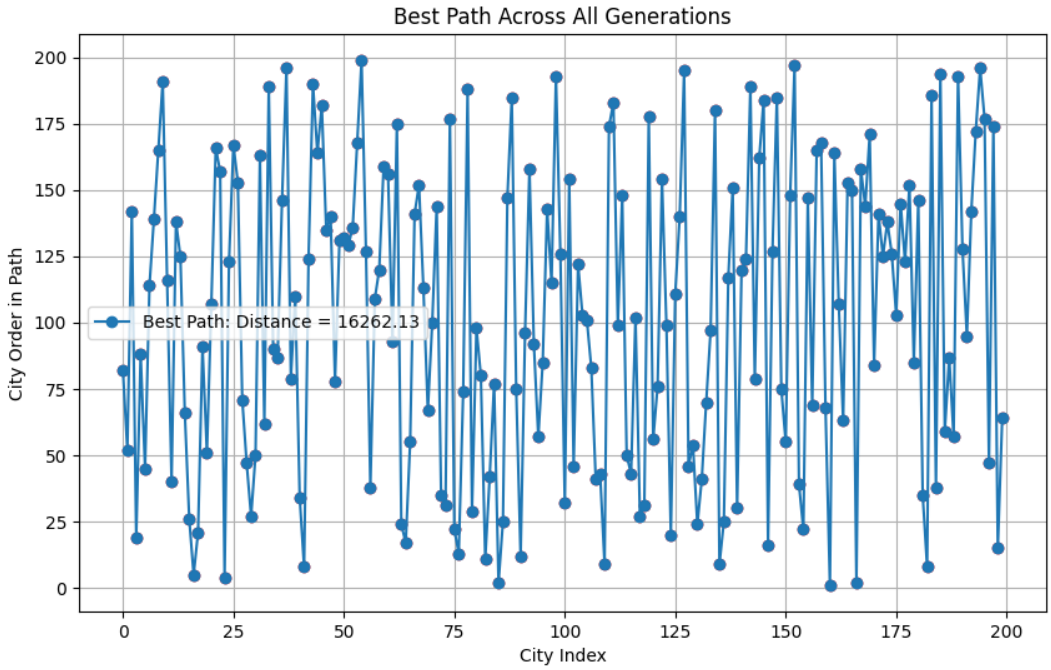
Let's look at the total paths traversed:




As we can see nothing and can no significant information from this, let's take a look at the stats over generations:




As we can see a much more stable average distance across 1000 generations and average best distance as well. All most 85% of the average distances are around 20000 to 20100. Let's take a look at the best path among the generations:




The best distance traveled across all these cities is: 16262.13 which is less than last run but again these distances between cities are totally randomly generated at start of the program.

Here's the best path:

[82, 52, 142, 19, 88, 45, 114, 139, 165, 191, 116, 40, 138, 125, 66, 26, 5, 21, 91, 51, 107, 166, 157, 4, 123, 167, 153, 71, 47, 27, 50, 163, 62, 189, 90, 87, 146, 196, 79, 110, 34, 8, 124, 190, 164, 182, 135, 140, 78, 131, 132, 129, 136, 168, 199, 127, 38, 109, 120, 159, 156, 93, 175, 24, 17, 55, 141, 152, 113, 67, 100, 144, 35, 31, 177, 22, 13, 74, 188, 29, 98, 80, 11, 42, 77, 2, 25, 147, 185, 75, 12, 96, 158, 92, 57, 85, 143, 115, 193, 126, 32, 154, 46, 122, 103, 101, 83, 41, 43, 9, 174, 183, 99, 148, 50, 43, 102, 27, 31, 178, 56, 76, 154, 99, 20, 111, 140, 195, 46, 54, 24, 41, 70, 97, 180, 9, 25, 117, 151, 30, 120, 124, 189, 79, 162, 184, 16, 127, 185, 75, 55, 148, 197, 39, 22, 147, 69, 165, 168, 68, 1, 164, 107, 63, 153, 150, 2, 158, 144, 171, 84, 141, 125, 138, 126, 103, 145, 123, 152, 85, 146, 35, 8, 186, 38, 194, 59, 87, 57, 193, 128, 95, 142, 172, 196, 177, 47, 174, 15, 64]

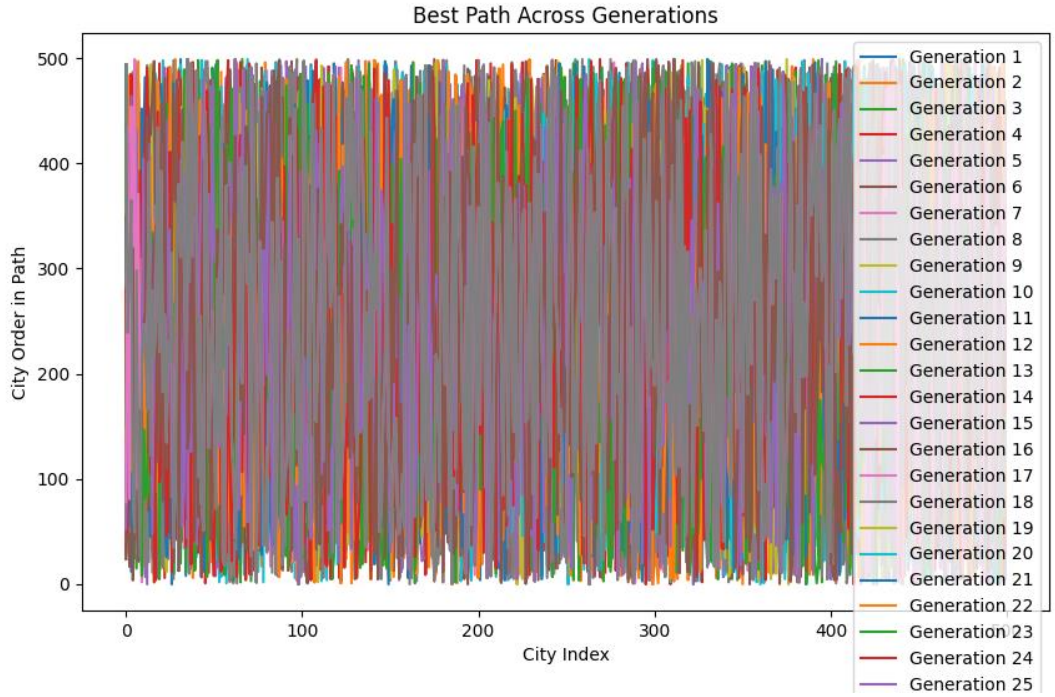
4) Fourth Run:

Let’s run the algorithm one last time with the following params:

| | |
|---|---|
| **Cities** | 500 |
| **Population Size** | 1000 |
| **Iterations** | 500 |

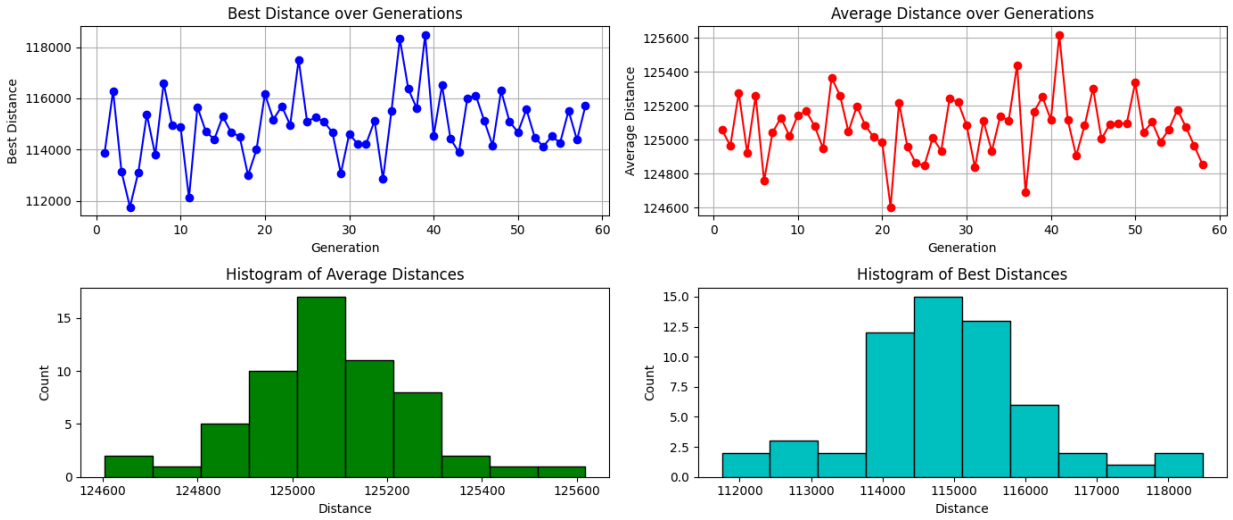
This did not take much time, however, there are a lot of cities involved now, making our visualizations unclear. However the algorithm came to an end rather quickly at only 39$^{th}$ Generation the entire population died out.

Let’s take a look at all paths traversed:




This would make a great painting for me looking at what it represents. Randomly generated solution space, however this is just too chaotic for mere 39 generations.

Let’s take a look at statistics across 39 generations:




As we can see there is no stable information as we only had 39 generations to work with.

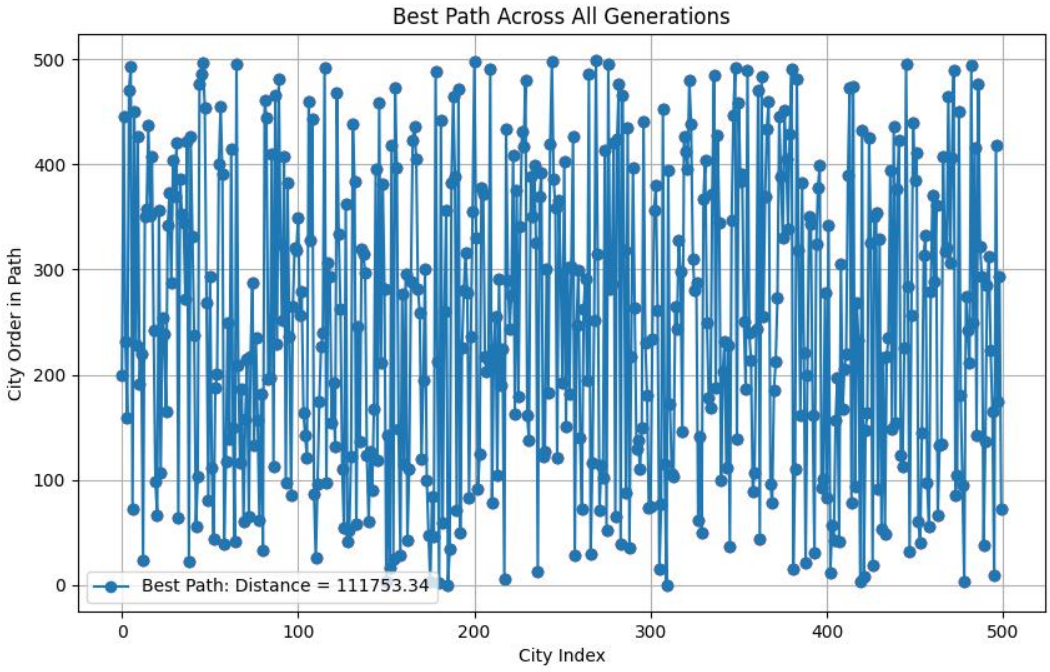
Let’s look at the final solution:

Path traversed:

[200, 445, 232, 159, 471, 493, 73, 450, 228, 427, 191, 220, 24, 350, 358, 437, 352, 407, 242, 99, 66, 356, 107, 254, 239, 165, 342, 373, 287, 404, 370, 420, 64, 386, 353, 345, 272, 422, 23, 426, 331, 238, 56, 103, 476, 486, 497, 454, 268, 81, 294, 112, 44, 188, 201, 400, 455, 391, 39, 118, 249, 139, 415, 150, 41, 496, 209, 117, 186, 60, 158, 215, 65, 218, 288, 133, 235, 157, 62, 182, 33, 461, 444, 196, 197, 410, 113, 466, 229, 481, 324, 252, 408, 98, 382, 236, 85, 265, 321, 318, 349, 257, 279, 164, 143, 121, 460, 328, 443, 87, 26, 96, 175, 227, 240, 492, 97, 307, 293, 154, 193, 132, 468, 334, 262, 110, 55, 362, 42, 52, 122, 439, 384, 58, 246, 137, 320, 315, 297, 124, 61, 127, 90, 168, 396, 119, 459, 212, 381, 282, 4, 142, 17, 418, 25, 473, 397, 148, 29, 277, 115, 296, 43, 111, 289, 423, 436, 405, 281, 259, 120, 195, 301, 100, 47, 3, 84, 46, 488, 213, 2, 442, 59, 260, 357, 0, 35, 383, 465, 388, 71, 472, 50, 226, 280, 316, 278, 83, 237, 355, 498, 330, 91, 125, 378, 372, 217, 203, 214, 491, 78, 225, 255, 104, 291, 190, 224, 6, 434, 290, 244, 276, 409, 163, 376, 179, 341, 431, 417, 480, 161, 138, 389, 351, 399, 325, 13, 369, 392, 122, 127, 300, 183, 419, 498, 386, 359, 121, 366, 295, 192, 403, 151, 302, 182, 303, 426, 28, 247, 299, 140, 73, 263, 291, 195, 486, 30, 117, 252, 499, 315, 71, 114, 102, 413, 52, 495, 281, 286, 421, 65, 424, 477, 39, 466, 319, 88, 435, 36, 218, 397, 264, 129, 138, 110, 150, 441, 231, 181, 74, 75, 234, 357, 380, 261, 15, 77, 453, 115, 0, 395, 172, 106, 103, 265, 243, 328, 298, 146, 427, 412, 396, 480, 438, 310, 280, 288, 62, 141, 50,

367, 404, 249, 178, 169, 372, 485, 188, 428, 345, 100, 203, 232, 112, 228, 37, 347, 447, 492, 139, 459, 384, 391, 251, 186, 490, 240, 214, 89, 107, 244, 470, 44, 484, 255, 370, 434, 460, 96, 78, 185, 213, 273, 446, 388, 330, 452, 405, 339, 429, 491, 16, 111, 481, 318, 161, 382, 221, 21, 200, 351, 343, 162, 31, 324, 378, 399, 93, 101, 278, 83, 342, 12, 57, 43, 157, 197, 41, 305, 167, 206, 220, 390, 473, 79, 474, 94, 269, 233, 4, 433, 8, 147, 164, 425, 325, 19, 350, 354, 91, 329, 53, 216, 49, 217, 235, 394, 149, 436, 155, 377, 423, 124, 113, 226, 496, 284, 32, 257, 440, 385, 411, 61, 40, 145, 314, 333, 97, 56, 279, 371, 289, 361, 66, 133, 134, 408, 317, 321, 465, 307, 406, 489, 86, 105, 450, 180, 95, 3, 274, 242, 211, 494, 250, 416, 143, 476, 322, 293, 38, 137, 285, 313, 223, 165, 9, 418, 175, 294, 72]

Best Path visualized:




The total distance for the best path is 1,11,753.34 units. That’s quite a lot. But then again these values might be higher due to the higher city count.