Artificial Intelligence

Genetic Algorithms x Travelling Salesman Problem (TSP) - Gavin Skehan – 21440824

Contents

1. Introduction

The Travelling Salesman Problem (TSP) is a well-known NP-Hard combinatorial optimization problem. It states that given a set of cities, we must find the shortest path it takes to visit each city exactly once and return to the starting city, it represents a connected graph. Due to the complexity of this problem it cannot be solved in polynomial time for large datasets. [7]

Genetic Algorithms (GAs) are heuristic search algorithms that are based on the process of natural selection, they are used for search optimization problems such as TSP to find the near-optimal solutions using biologically inspired processes such as selection, crossover and mutation. [8] In this problem genomes represent a sequence of cities and the population consists of potential routes for the salesman. Genomes are altered using the process of selection, crossover and mutation to explore search space and provide diverse solutions in order to find the near-optimal solutions.

2. Implementation details and design choices

2.1 Overall Approach

The goal here is to implement a core genetic algorithm for the TSP which follows a clear structured process of phases. Population initialization creates a set number of possible solutions, fitness function to get the best scores, selection to select the parents to reproduce offspring. Random crossover and mutation between two candidates in the hope to generate new and improved offspring. Finally, calling the entire genetic algorithm with all the functions and analyse the performance and solution quality.

2.2 Population Initialization

The population is the set of possible solutions (tours), we want to generate the initial population of random tours. Using a random genome function that creates a list of city indices from 0 to length – 1, we generate random routes. Call the initialize function that takes the parameters of population size and the genome length which is the cities we have to visit, create the population.

2.3 Selection [1]

Selection is the process of picking the fittest candidate to be passed onto the next generation, this mimics the process of the evolution. Tournament Selection simulates a random tournament between candidates to compete with each other where the candidate with the best fitness score will pass onto the next generation, this happens multiple times across the initial population until we have a new population. Selection pressure measures the likelihood of participation in the tournament. A higher tournament size means weaker candidates are more likely to be competing with stronger candidates. The tournament size is set to a value of 3.

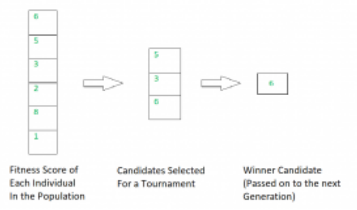

6
5
3
2
8
1
5
3
6
6
Fitness Score of Each Individual In the Population
Candidates Selected For a Tournament
Winner Candidate (Passed on to the next Generation)

Figure 1: Selection Process [1]

2.4 Reproduction

Genetic Algorithms are based on the processes of evolutions of life, reproduction is the key aspect that enables us to evolve and adapt better to the environment. In GAs, we produce our new populations generation after generation by using techniques such as crossover and mutation that are both directly effected by the selection mechanism which is the tournament selection in the case. Tournament selection randomizes the parents placed together to reproduce the offspring simulating an environment in which candidates are selected to reproduce based on their fitness.

Crossover is reliant on crossover rate, the rate determines how many parents exchange genetic information before creating the new population, controlling how parents genes are transferred with the aim for the offspring to inherit the best traits from both parents. Crossover tends to be most optimal in around 80% of the population.

Mutation is a randomized gene placement technique used in the hope to generate unique genomes with better fitness scores than previous populations. It introduces genetic diversity by altering part of the genome. Mutation is also governed by a mutation rate where the optimal value is less than 10% of the population.

Example of the process: Selection occurs, the chosen parents are placed in a mating pool. The number of offspring is determined by the population size (e.g. if the population is 100, produce 100 offspring). Parents are randomly paired for crossover based on a probability factor (crossover rate).

2.5 Crossover [2]

Crossover is the process of creating new individuals by combining genetic information (cities in this case) from two parents (populations). In this assignment we are combining solutions from two genomes and creating a new output solution. High crossover rates encourages exploration of the search space which can lead to quicker convergence but if it is too high the algorithm could get stuck at the local optima. Low rates will result is slow progress towards the optimal solution.

2.5.1 Order Crossover (OX1)

Simply chose a genome section from one parent, place it in the same position in the empty child genome, remove the numbers (cities) from the second parent and then fill in the rest of the child genome with the leftover numbers (cities) from left to right.

Given two parents P1[] and P2[] with empty genome as the offspring, we select a random substring within P1[] and insert it into offspring[]. Next step is to delete these cities from P2[] so we remove the possibility of duplication. Step 3 is to add the remaining cities from P2[] into offspring[] from left to right, this is the final output

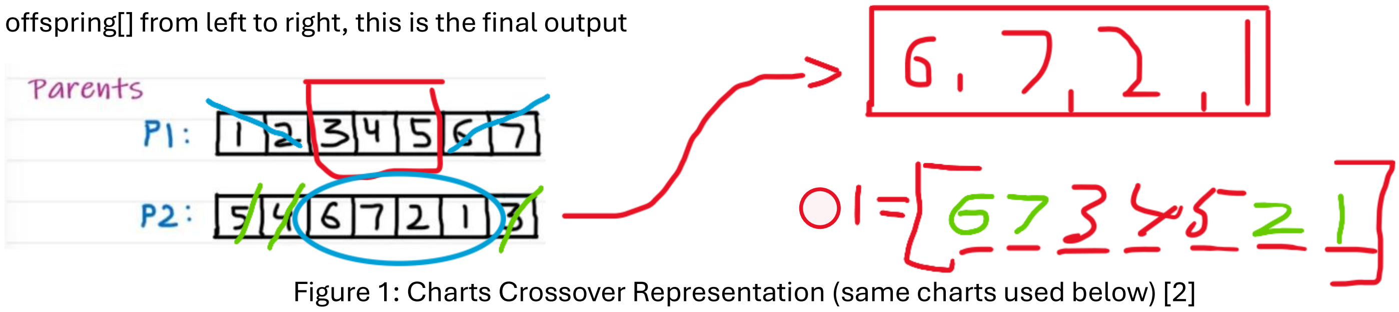
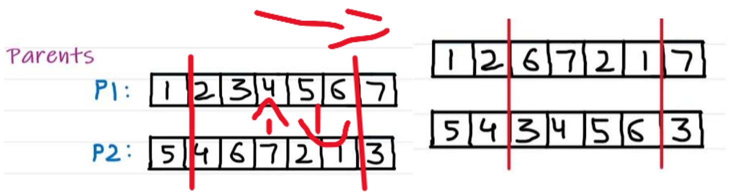


Figure 1: Charts Crossover Representation (same charts used below) [2]

2.5.2 Partially Mapped Crossover (PMX)

Select common crossover point and swap cities from parent 1 and parent 2. Now there will be duplication of cities in each genome. To remove duplication work out the mapping Spain(P1) swapped with France (P2) but France (P1) swapped with Ireland (P2). That means Spain maps to Ireland because Spain -> France -> Ireland. This also means that Ireland maps to Spain because Ireland -> France -> Spain.

Given two parents P1[] and P2[], select two random crossover points at the same index and perform crossover of the values from P1 to P2. Next, find the mapping relationship from substring and use the mapping to create the new offspring. This mapping only applies to cities outside the crossover points.



Determine the mapping relationship and map in unselected substring: Figures [2]

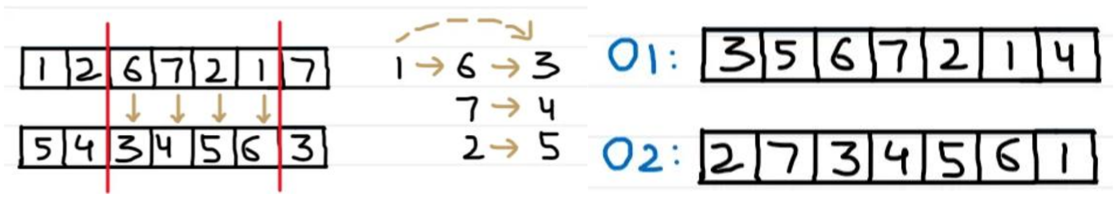


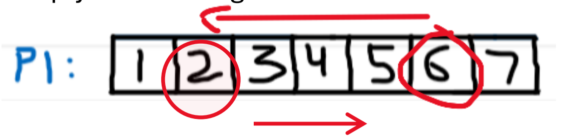
Figure 3: PMX crossover illustration [2]

2.6 Mutation [3]

Mutation is random change in a population, we simulate this through different mutation functions in the hope to generate better solutions. High mutation rate means genes are going to be more randomly ordered from generation to generation. High genetic diversity helps escape local optima and explore more of the search space but if it is too high it can lead to poor result due to random behaviour and slower convergence. Low rates means small search space and less diversity causing the algorithm to stay at the local optima.

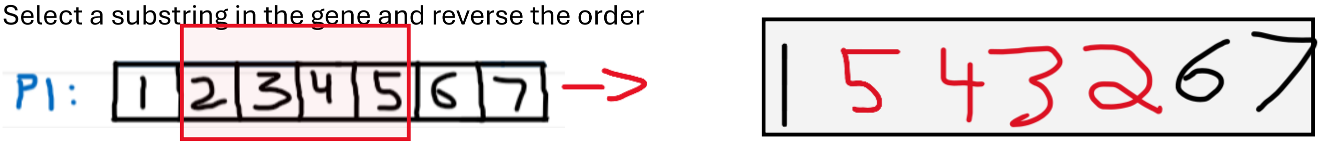
2.6.1 Swap Mutation

Simply select two genes from the chromosome and swap their values



2.6.2 Inversion Mutation

Select a substring in the gene and reverse the order



2.7 Fitness Evaluation

As specified in in the datafiles as ‘EDGE_WEIGHT_TYPE : EUC_2D’ is used to calculate the straight line distance between two cities using the x, y coordinates. This functionality is called in the calculate distance matrix function.

$$d = \sqrt{[(x2 - x1)2 + (y2 - y1)2]}$$

- D is Euclidean distance
- X1, y1 = city1
- X2, y2 = city2

Calculate_distance_matrix(cities): this function generates a 2D distance matrix representing the distance between every pair of cities in the problem.

Fitness Function is used to find the best fitness scores which is the minimum value of the tour distance, we have to ensure that we set the value to 'min' as the fitness function must return the minimum fitness value.

2.8 Genetic Algorithm()

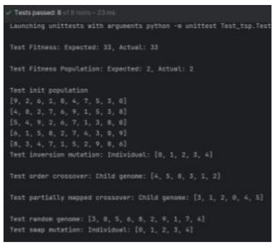
The Genetic Algorithm is takes in the TSPLIB format files as an input. It reads the file to extract city coordinates and construct a distance matrix using the Euclidean distances between the cities. Reading the TSPLIB files enables the algorithm to solve the TSP. The algorithm initializes the population of the given size with random routes, evaluates the fitness of the routes based on distance. It selects parents to be placed in a pool to be chosen for crossover and mutation, initializing the next population with the goal of evolving the fitness scores or in this case finding shorter routes. The algorithm continues until the generations is met or the stopping criteria is hit returning the best fitness score and the exact route of the best tour along with the average fitness and the computational time.

2.9 Termination Criteria

The termination criteria in this code is based on convergence. Convergence means that there is no more room for improvement meaning the bet results found for that particular scenario. The algorithm will stop if there is no improvement in the fitness score for a high number of consecutive generations i.e. 'No improvement for 50 consecutive generations'. This is important as if the algorithm is no longer making an improvements (convergence) we should return to the best fitness and move on to the next parameter check. This helps with computational costs by terminating a stale process.

2.10 Unit Tests Results

I created simple unit tests for each of the genetic algorithm functions to ensure that they run with the expected outputs.



```
✓ Tests passed: 8 of 8 tests – 23 ms
Launching unittests with arguments python -m unittest Test_tsp.Test

Test Fitness: Expected: 33, Actual: 33

Test Fitness Population: Expected: 2, Actual: 2

Test init population
[9, 2, 6, 1, 8, 4, 7, 5, 3, 0]
[4, 0, 2, 7, 6, 9, 1, 5, 3, 8]
[5, 4, 9, 2, 6, 7, 1, 3, 8, 0]
[6, 1, 5, 8, 2, 7, 4, 3, 0, 9]
[0, 3, 4, 7, 1, 5, 2, 9, 8, 6]
Test inversion mutation: Individual: [0, 1, 2, 3, 4]

Test order crossover: Child genome: [4, 5, 0, 3, 1, 2]

Test partially mapped crossover: Child genome: [3, 1, 2, 0, 4, 5]

Test random genome: [3, 0, 5, 6, 8, 2, 9, 1, 7, 4]
Test swap mutation: Individual: [0, 1, 2, 3, 4]
```

3. Experimental results and analysis

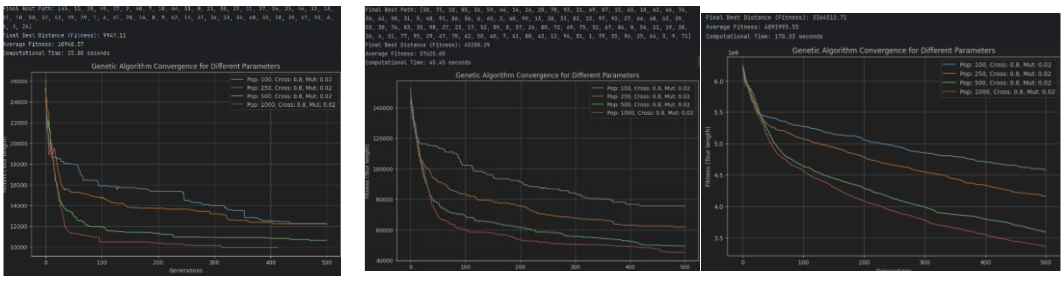
3.1 Performance Results

3.1.1 Different parameter settings

- Generations = 500
- Tournament size = 5
- Crossover = partially mapped crossover
- Mutation = swap mutation

1. Population

berlin52 *kroA100* *pr1002*



```
Final Best Path: [45, 15, 28, 49, 17, 2, 40, 7, 18, 44, 31, 0, 21, 30, 22, 11, 27, 26, 25, 46, 12, 13,
51, 10, 50, 32, 43, 19, 29, 1, 6, 41, 20, 16, 8, 9, 42, 14, 37, 36, 33, 34, 48, 35, 38, 39, 47, 23, 4,
5, 3, 24]
Final Best Distance (Fitness): 9947.11
Average Fitness: 10948.57
Computational Time: 25.88 seconds
```

```
Final Best Path: [50, 73, 18, 85, 26, 59, 44, 14, 16, 20, 78, 93, 21, 69, 87, 15, 65, 18, 62, 46, 76,
34, 61, 90, 31, 5, 48, 91, 86, 56, 0, 65, 2, 40, 99, 13, 28, 33, 82, 22, 97, 92, 27, 66, 68, 63, 39,
53, 30, 74, 83, 35, 98, 37, 23, 17, 52, 89, 0, 57, 24, 80, 72, 49, 75, 32, 67, 84, 8, 54, 11, 19, 38,
36, 4, 51, 77, 95, 29, 47, 70, 42, 58, 60, 7, 41, 88, 43, 12, 94, 81, 1, 79, 55, 96, 25, 64, 3, 9, 71]
Final Best Distance (Fitness): 45280.29
Average Fitness: 57625.08
Computational Time: 45.45 seconds
```

```
Final Best Distance (Fitness): 3364312.71
Average Fitness: 4091993.55
Computational Time: 170.33 seconds
```

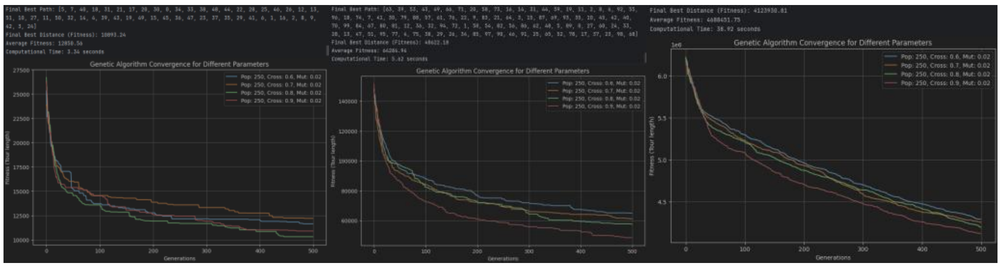
Genetic Algorithm Convergence for Different Parameters
Pop: 100, Cross: 0.8, Mut: 0.02
Pop: 250, Cross: 0.8, Mut: 0.02
Pop: 500, Cross: 0.8, Mut: 0.02
Pop: 1000, Cross: 0.8, Mut: 0.02
Fitness (Tour length)
Generations
1e6

Figure 4: Graphing different population values

We can see from the charts that the greater the population the better fitness scores achieved. I would assume that there is a certain threshold not to go over as it might have adverse effects or diminishing returns as it may increase computational time while not leading to better scores. Size of 1000 seems to be a very good value. With population size of 1000, the fitness score converged quickly causing the stop function to trigger too.

2. Crossover

berlin52 *kroA100* *pr1002*



```
Final Best Path: [5, 7, 40, 18, 31, 21, 17, 20, 30, 0, 34, 33, 38, 48, 44, 22, 28, 25, 46, 26, 12, 13,
51, 10, 27, 11, 50, 32, 14, 4, 39, 43, 19, 49, 15, 45, 36, 47, 23, 37, 35, 29, 41, 6, 1, 16, 2, 8, 9,
42, 3, 24]
Final Best Distance (Fitness): 10893.24
Average Fitness: 12850.56
Computational Time: 3.34 seconds
```

```
Final Best Path: [63, 39, 53, 43, 49, 66, 71, 20, 58, 73, 16, 14, 31, 44, 59, 19, 11, 2, 8, 6, 92, 55,
96, 18, 74, 7, 41, 30, 79, 88, 57, 61, 76, 22, 9, 83, 21, 64, 3, 15, 87, 69, 93, 35, 10, 43, 42, 40,
70, 99, 84, 67, 80, 81, 12, 36, 32, 94, 72, 1, 58, 54, 82, 56, 86, 62, 48, 5, 89, 0, 27, 60, 24, 33,
28, 13, 47, 51, 95, 77, 4, 75, 38, 29, 26, 34, 85, 97, 98, 46, 91, 25, 65, 52, 78, 17, 37, 23, 90, 68]
Final Best Distance (Fitness): 48622.18
Average Fitness: 64286.96
Computational Time: 5.62 seconds
```

```
Final Best Distance (Fitness): 4123930.81
Average Fitness: 4688431.75
Computational Time: 38.92 seconds
```

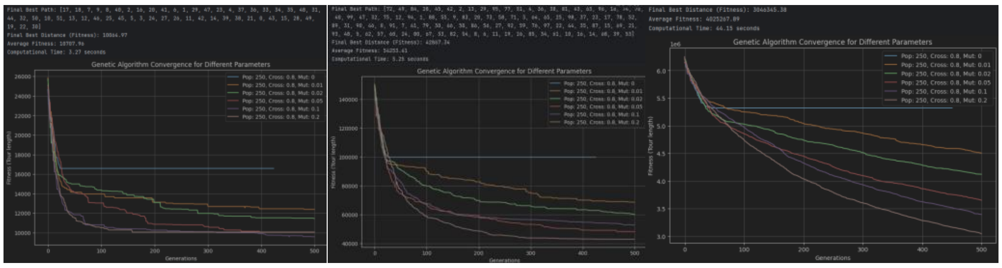
Genetic Algorithm Convergence for Different Parameters
Pop: 250, Cross: 0.6, Mut: 0.02
Pop: 250, Cross: 0.7, Mut: 0.02
Pop: 250, Cross: 0.8, Mut: 0.02
Pop: 250, Cross: 0.9, Mut: 0.02
Fitness (Tour length)
Generations
1e6

Figure 5: Graphing different crossover rate values

The crossover rate had varying results from each chart with berlin52 best rate being 0.8, followed closely by 0.9. However kroA1000 and pr1002 have the opposite which could be due to increased complexity, a higher population could yield better results but this is a good test. 0.06 and 0.7 were both poor compared to the other values. Running the test on more generations would be interesting as none of them triggered the stop function so there could be more improvements from each rate of crossover. Problems with larger datasets look to benefit more from greater crossover rates.

3. Mutation

berlin52 *kroA100* *pr1002*



```
Final Best Path: [17, 18, 7, 9, 8, 40, 2, 16, 20, 41, 6, 1, 29, 47, 23, 4, 37, 36, 33, 34, 35, 48, 31,
44, 32, 50, 10, 51, 13, 12, 46, 25, 45, 5, 3, 24, 27, 26, 11, 42, 14, 39, 38, 21, 0, 43, 15, 28, 49,
19, 22, 30]
Final Best Distance (Fitness): 10066.97
Average Fitness: 10707.96
Computational Time: 3.27 seconds
```

```
Final Best Path: [72, 49, 84, 26, 45, 42, 2, 13, 29, 95, 77, 81, 4, 36, 38, 81, 43, 63, 96, 16, 14, 70,
40, 99, 47, 32, 75, 12, 94, 1, 88, 55, 9, 83, 20, 73, 58, 71, 3, 64, 65, 25, 90, 37, 23, 17, 78, 52,
89, 31, 90, 66, 0, 91, 7, 61, 79, 30, 60, 38, 86, 56, 27, 92, 59, 76, 97, 22, 44, 35, 87, 15, 69, 21,
93, 48, 5, 62, 57, 68, 24, 00, 67, 53, 82, 54, 8, 6, 11, 19, 26, 85, 34, 41, 10, 16, 14, 08, 39, 53]
Final Best Distance (Fitness): 42867.34
Average Fitness: 54253.61
Computational Time: 5.25 seconds
```

```
Final Best Distance (Fitness): 3846345.38
Average Fitness: 4025267.89
Computational Time: 44.15 seconds
```

Genetic Algorithm Convergence for Different Parameters
Pop: 250, Cross: 0.8, Mut: 0
Pop: 250, Cross: 0.8, Mut: 0.01
Pop: 250, Cross: 0.8, Mut: 0.02
Pop: 250, Cross: 0.8, Mut: 0.05
Pop: 250, Cross: 0.8, Mut: 0.1
Pop: 250, Cross: 0.8, Mut: 0.2
Fitness (Tour length)
Generations
1e6

Figure 6: Graphing different mutation rate values

Mutation rate scores were clearly separated. 0, 0.01, 0.02 all performed poorly. 0.2 performed the best overall especially in charts two and three, but in berlin the rate of 0.1 slightly surpassed it. A mutation rate of 0 triggered the stop criteria early (premature convergence) with extremely poor results, this is expected as the populations will suffer from no genetic diversity. The greater the mutation in these ranges seems to work a lot better than a low mutation rate.

3.1.2 Computational Time

Comparing how different parameter setting effect the computational time (whole number) with their best solution.

- Population Size (1000, 1000, 1000)
  - Berlin: 25.88 seconds
  - KroA001: 44.45 seconds
  - Pr1002: 170.33 seconds
- Crossover Rate (0.8, 0.9, 0.9)
  - Berlin: 3.34 seconds
  - KroA001: 5.62 seconds
  - Pr1002: 38.92 seconds
- Mutation Rate (0.1, 0.2, 0.2)
  - Berlin: 3.27 seconds
  - KroA001: 5.25 seconds
  - Pr1002: 44.15 seconds

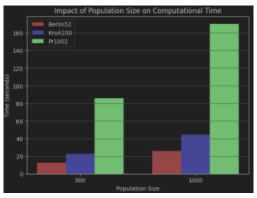


Figure 7: Effects of generations on computational time for each problem

We can see from the results that population size has a huge impact on the computational time especially for more complex problems which was expected as larger populations require more evaluations per generation. The computational time jumps significantly from kroA100 to pr1002 due to the increase in the number of possible solutions to evaluate. The exponential increase in computational time may not be worth the slightly better fitness score for KroA100 with increased population sizes as second best solutions generally same operators with 500 population instead of 1000. The jump in the large problem is understandable but in order to reach the near optimal solution the generations would have to be set to a huge number.

Crossover and mutation also see an increase from small to medium to large problems with a bigger jump from medium to large for computational time suggesting that as the problem grows, these parameters increase computational time.

3.1.3 Operators

I want to run a quick test on the two crossover and the two mutation functions to see the best performing pair on berlin52 with parameters = [1000, 0.8, 0.1] and 500 generations. I will run each twice and record best fitness score.

- Swap Mutation & Partially Mapped Crossover = 9102.99
- Swap Mutation and Order Crossover = 8646.67
- Inversion Mutation and Order Crossover = 7841.19
- Inversion Mutation and Partially Mapped Crossover = 7796.33 (very close to optimum)

From the test, Inversion Mutation and Partially Mapped combination is the best performing mutation and crossover operator. We only chose one selection mechanism, tournament selection. We could also try roulette selection but this is unnecessary for now. We will use this in the remaining plots.

3.2 Visualization

Fitness over Generations: for this section I will present the top result from the parameter tests on each dataset as it provided us with an insight on where best to look to find the best solution. I will run a parameter set of the best performing values. I will run the code 3 times for varying randomness to ensure we get best score for the testing.

Parameter_sets (population size, crossover rate, mutation rate) = [
(500, 0.1, 0.8),
(500, 0.1, 0.9),
(500, 0.2, 0.8),
(500, 0.2, 0.9),
(1000, 0.1, 0.8),
(1000, 0.1, 0.9),
(1000, 0.2, 0.8),
(1000, 0.2, 0.9)]

Best Performing plot for each TSP

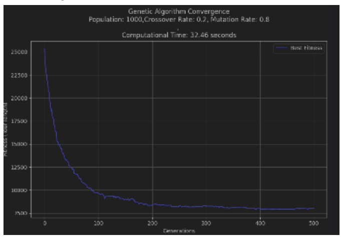
- **Berlin52: 7542**

Final Best Path: [30, 20, 22, 19, 49, 28, 29, 41, 1, 6, 16, 2, 44, 18, 40, 7, 8, 9, 42, 32, 50, 10, 51, 13, 12, 46, 25, 26, 27, 11, 24, 3, 5, 47, 23, 4, 14, 37, 39, 36, 33, 43, 15, 45, 38, 35, 34, 48, 31, 0, 21, 17]

Final Best Distance (Fitness): 7911.96

Average Fitness: 9401.25

Computational Time: 32.46 seconds



Converges early finding the local optima, triggering stopping criteria

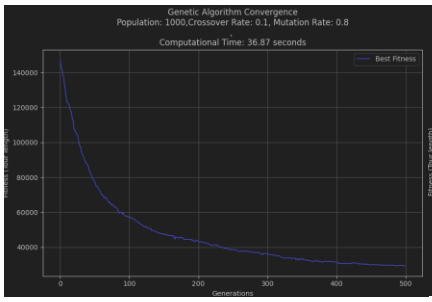
- **KroA100: 21282**

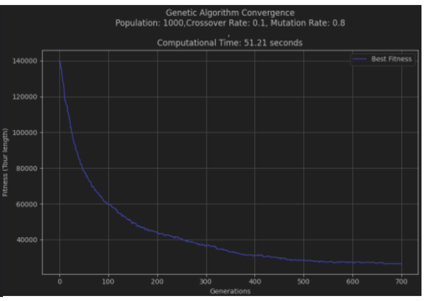
Final Best Path: [54, 82, 33, 45, 2, 28, 42, 99, 40, 70, 13, 47, 51, 29, 84, 38, 95, 77, 4, 36, 32, 75, 12, 94, 81, 43, 72, 67, 49, 1, 63, 53, 39, 68, 86, 56, 6, 8, 60, 50, 80, 24, 57, 27, 92, 20, 71, 98, 21, 93, 87, 15, 69, 65, 25, 64, 3, 52, 55, 41, 88, 30, 79, 18, 96, 74, 91, 7, 66, 0, 62, 48, 5, 89, 78, 17, 23, 37, 35, 9, 83, 46, 14, 58, 73, 16, 10, 31, 44, 97, 90, 22, 59, 61, 76, 34, 85, 19, 11, 26]

Final Best Distance (Fitness): 29361.87

Average Fitness: 47361.03

Computational Time: 36.87 seconds





500 Generations 700 Generations

Doesn't converge to local optima within the given generations

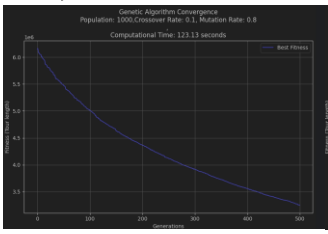
700 Generations:

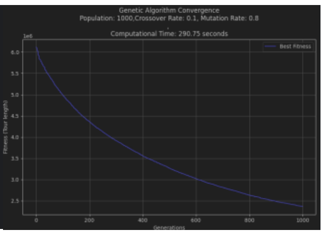
Final Best Distance (Fitness): 26505.14
Average Fitness: 42233.84
Computational Time: 51.21 seconds

- **Pr1002: 259045**

Final Best Path: [850, 551, 901, 818, 757, 825, 148, 32, 17, 305, 576, 328, 456, 589, 919, 882, 923, 674, 679, 961, 936, 953, 896, 859, 827, 534, 899, 705, 710, 617, 764, 875, 776, 902, 867, 684, 513, 197, 214, 548, 483, 408, 373, 224, 295, 549, 477, 476, 511, 600, 890, 1000, 804, 749, 957, 807, 744, 989, 716, 956, 702, 765, 734, 368, 352, 335, 733, 792, 449, 786, 929, 889, 516, 407, 276, 108, 736, 790, 654, 320, 342, 624, 228, 113, 300, 275, 307, 488, 376, 24, 90, 30, 93, 84, 401, 393, 683, 903, 564, 854, 504, 502, 240, 239, 367, 632, 611, 620, 360, 47, 97, 31, 366, 447, 458, 544, 826, 997, 601, 872, 778, 796, 638, 677, 644, 410, 364, 658, 378, 425, 59, 144, 996, 382, 391, 101, 481, 446, 216, 268, 55, 187, 203, 135, 54, 254, 259, 245, 75, 4, 735, 972, 745, 766, 598, 527, 579, 836, 762, 708, 390, 661, 395, 394, 49, 7, 13, 474, 583, 537, 486, 191, 445, 460, 557, 991, 384, 312, 282, 78, 2, 349, 389, 982, 769, 945, 725, 738, 937, 817, 794, 597, 442, 606, 887, 849, 533, 227, 179, 162, 233, 218, 215, 834, 536, 526, 524, 455, 609, 358, 668, 669, 363, 574, 560, 539, 559, 888, 949, 984, 505, 123, 278, 0, 161, 104, 117, 315, 28, 592, 594, 453, 438, 619, 641, 648, 468, 457, 211, 125, 299, 421, 472, 461, 463, 814, 602, 247, 464, 528, 508, 521, 876, 960, 603, 616, 768, 777, 262, 226, 225, 184, 163, 173, 172, 193, 265, 288, 383, 263, 98, 146, 145, 269, 353, 334, 343, 981, 732, 980, 985, 374, 426, 751, 771, 428, 285, 9, 43, 5, 252, 256, 475, 823, 880, 922, 841, 731, 992, 977, 469, 568, 492, 593, 543, 637, 650, 604, 714, 703, 798, 828, 802, 546, 626, 931, 909, 820, 894, 689, 782, 555, 531, 810, 773, 737, 664, 712, 670, 306, 302, 110, 966, 672, 727, 908, 970, 958, 707, 706, 691, 723, 907, 752, 545, 566, 116, 199, 208, 231, 990, 298, 249, 571, 809, 831, 893, 787, 873, 863, 552, 779, 761, 940, 698, 613, 430, 770, 916, 935, 926, 439, 830, 1001, 840, 270, 45, 204, 170, 82, 70, 11, 323, 19, 160, 152, 595, 567, 816, 878, 586, 287, 56, 63, 371, 377, 435, 682, 596, 420, 622, 645, 618, 659, 345, 946, 952, 950, 356, 347, 388, 623, 414, 651, 372, 190, 186, 243, 813, 805, 944, 424, 409, 885, 998, 580, 244, 137, 79, 77, 102, 124, 42, 577, 911, 862, 599, 783, 971, 667, 180, 200, 171, 175, 917, 774, 973, 729, 614, 709, 844, 781, 924, 915, 780, 791, 819, 847, 846, 910, 692, 979, 955, 988, 701, 671, 951, 900, 717, 581, 643, 584, 444, 968, 918, 969, 553, 454, 634, 365, 286, 318, 415, 466, 542, 835, 891, 800, 404, 385, 292, 332, 61, 58, 280, 149, 142, 167, 81, 316, 321, 26, 212, 529, 522, 44, 46, 403, 267, 229, 201, 153, 122, 130, 134, 625, 629, 370, 647, 642, 411, 387, 20, 22, 107, 91, 3, 88, 141, 139, 51, 64, 418, 459, 296, 309, 317, 29, 351, 423, 615, 660, 673, 355, 479, 478, 491, 490, 427, 182, 157, 301, 53, 12, 36, 189, 496, 499, 223, 192, 174, 198, 585, 515, 869, 864, 921, 928, 784, 852, 822, 470, 129, 241, 503, 202, 166, 158, 485, 126, 235, 236, 147, 230, 183, 518, 547, 569, 587, 554, 437, 359, 35, 324, 348, 346, 748, 337, 452, 575, 808, 904, 925, 663, 640, 605, 412, 441, 331, 397, 357, 728, 747, 964, 621, 713, 678, 760, 974, 978, 943, 983, 842, 578, 194, 136, 232, 281, 398, 399, 314, 221, 246, 686, 688, 962, 976, 845, 811, 608, 785, 897, 866, 930, 884, 871, 874, 905, 879, 775, 565, 509, 519, 517, 914, 711, 799, 806, 530, 858, 965, 934, 758, 392, 313, 94, 95, 39, 379, 86, 168, 188, 121, 257, 308, 994, 159, 154, 261, 143, 234, 495, 112, 330, 400, 339, 65, 693, 987, 636, 675, 933, 722, 754, 676, 700, 742, 696, 932, 719, 655, 662, 718, 440, 695, 870, 883, 877, 853, 857, 861, 272, 222, 74, 213, 237, 165, 369, 242, 264, 85, 10, 21, 34, 283, 350, 297, 993, 96, 57, 60, 73, 354, 375, 344, 341, 311, 37, 434, 294, 322, 132, 25, 16, 131, 115, 291, 293, 41, 451, 340, 610, 119, 106, 361, 639, 462, 443, 838, 892, 839, 895, 431, 83, 436, 832, 181, 156, 206, 582, 795, 697, 967, 665, 666, 906, 730, 721, 763, 886, 801, 868, 856, 812, 538, 413, 939, 913, 912, 572, 432, 558, 755, 756, 680, 759, 959, 947, 789, 724, 750, 631, 652, 448, 612, 656, 433, 467, 588, 532, 821, 920, 927, 690, 704, 405, 169, 251, 498, 523, 114, 303, 628, 429, 103, 111, 646, 48, 406, 289, 419, 396, 562, 995, 507, 266, 150, 250, 14, 6, 8, 89, 178, 570, 220, 494, 501, 489, 417, 422, 23, 18, 66, 87, 386, 380, 1, 76, 33, 329, 630, 687, 720, 739, 685, 362, 381, 635, 986, 681, 975, 999, 793, 465, 284, 563, 487, 155, 238, 176, 649, 741, 942, 753, 881, 865, 482, 185, 253, 120, 304, 217, 196, 591, 535, 500, 333, 68, 450, 338, 109, 255, 105, 128, 99, 473, 788, 963, 948, 743, 715, 520, 797, 694, 699, 767, 898, 772, 100, 92, 40, 80, 71, 138, 210, 207, 815, 550, 726, 740, 746, 803, 843, 848, 833, 860, 855, 590, 607, 627, 258, 271, 260, 195, 209, 177, 219, 248, 140, 290, 633, 38, 52, 151, 127, 277, 69, 67, 15, 319, 72, 327, 573, 541, 525, 540, 556, 512, 471, 402, 657, 336, 279, 133, 62, 27, 325, 326, 653, 273, 310, 274, 50, 118, 164, 205, 480, 514, 506, 497, 484, 824, 851, 837, 829, 510, 416, 493, 954, 941, 938, 561]

Final Best Distance (Fitness): 3252423.98
Average Fitness: 4286095.78
Computational Time: 123.13 seconds





500 generations 1000 generations

Huge dataset, expected not to converge within the given generations. The genetic algorithm would continue to refine the solution beyond my results with greater generations, indicating unbounded potential for greater results.

1000 generations:

Final Best Distance (Fitness): 2366381.96
Average Fitness: 3517490.24
Computational Time: 290.75 seconds

- The reason I set generations to higher value is because the algorithm does not converge to a known optima, you can see from the graph that the fitness score will continue to decrease over generations, but to get near the known optimum you will have to run the algorithm at an extremely high number of generations.

3.3 Convergence Analysis

Convergence is influenced by mutation and crossover rates as they directly impact how to populations evolve towards the best solution from generation to generation.

Mutation Rate Impact

- Low mutation rate: leads to stagnation as the population will struggle to create genetic diversity, the algorithm won’t escape the local optima
- Higher mutation rate: will slow convergence but provide us with better scores as it can generate a population with genetic diversity. If its too high there may be reverse effects as there will be too much randomness possibly creating worse populations

Crossover and Mutation Interaction

- A good balance between crossover and mutation rates will improve the fitness score providing diverse solutions while ensuring best traits are passed to the next generation
- Mutation rate of 0.1/0.2 with crossover set to 0.8/0.9 allows the algorithm to converge in good time with good solutions

Problem Size

- Small Problem: fast convergence due to small number of cities and search space
- Medium Problem: slightly longer convergence due to added number of cities, bigger spread between parameter settings in terms of fitness achieved so a lot more sensitive to parameter tuning
- Large Problems: much slower convergence due to significant number of cities in the search space. Would require more generations to see better solutions. Larger problems will work better with higher mutation rates to generate constant diversity which will speed up convergence

4. Comparison with Known Optimal Solutions

4.1 Results Chart

Dataset	Know Optimum	Best Found	% Error
Berlin52	7542	7911.96	4.90%
KroA100	21282	26505.14	24.56%
Pr1002	259045	2366381.96	813.53%

Figure 8: Results chart known optimum vs best found

4.2 Factors Affecting Suboptimal Performance

1. Not enough generations: some instances such as medium and larger problems will need an extensive number of generations, as seen above pr1002 has not fully converged with 1000 generations, that is why the error percentage is bad for pr1002.
2. Further parameter testing: getting the exact balance between exploration and exploitation for each problem would require a lot more testing. A grid search technique would be good but very exhausting. It can be used to find the scores of all possible combinations of parameters and return the best results.

5. Discussion of potential improvements

5.1 Elitism

Elitism will ensure that the top performing individuals across each generation will be passed onto the next generation, speeding up convergence. It mitigates the stochastic effects of selection methods such as tournament selection where even strong individuals might be eliminated due to random competition. By carrying over the best solutions elitism helps prevent the loss of optimal candidates and improves overall algorithm stability.

5.2 Time complexity

- Parallelization and concurrency: running multiple processes at the same time would significantly improve the computational time as one process isn't dependent on another to finish
- Memorization: can be used to 'bypass the need to re-evaluate redundant chromosome configurations to reduce time complexity' [5]. This involves storing previously computed values in a cache so it wont have to calculate the values again

5.3 Hyperparameter tuning

- Extensive grid searching the best parameters for each particular problem to fine-tune parameters

6. References

1. *GeeksforGeeks. (2018). Tournament Selection (GA). [online] Available at: https://www.geeksforgeeks.org/tournament-selection-ga/.*
2. *Garg, A. (2021). Crossover Operators in Genetic Algorithm - Geek Culture - Medium. [online] Medium. Available at: https://medium.com/geekculture/crossover-operators-in-ga-cffa77cdd0c8.*
3. *GeeksforGeeks. (2018). Mutation Algorithms for String Manipulation (GA). [online] Available at: https://www.geeksforgeeks.org/mutation-algorithms-for-string-manipulation-ga/.*
4. *Shendy, R. (2024). Traveling Salesman Problem (TSP) using Genetic Algorithm (Python). [online]* ***AI monks.io****. Available at: https://medium.com/aimonks/traveling-salesman-problem-tsp-using-genetic-algorithm-fea640713758.*
5. *Girsang, A. and Tanjung, D. (n.d.). Fast Genetic Algorithm for Long Short-Term Memory Optimization. [online] Available at: https://www.engineeringletters.com/issues_v30/issue_2/EL_30_2_17.pdf [Accessed 17 Feb. 2025].*
6. *NeuralNine (2024). Genetic Algorithms in Python - Evolution For Optimization. [online] YouTube. Available at: https://www.youtube.com/watch?v=CRtZ-APJEKI [Accessed 5 Nov. 2024].*
7. *Wikipedia Contributors (2019). Travelling salesman problem. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Travelling_salesman_problem.*
8. *Wikipedia Contributors (2019). Genetic algorithm. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Genetic_algorithm.*

Code Link

GitHub Repository: https://github.com/skehan0/AI
[6, 7]

User Guide

ReadMe Guide: https://github.com/skehan0/AI/blob/master/README.md