Homework 2: Search and Optimization

1 Introduction

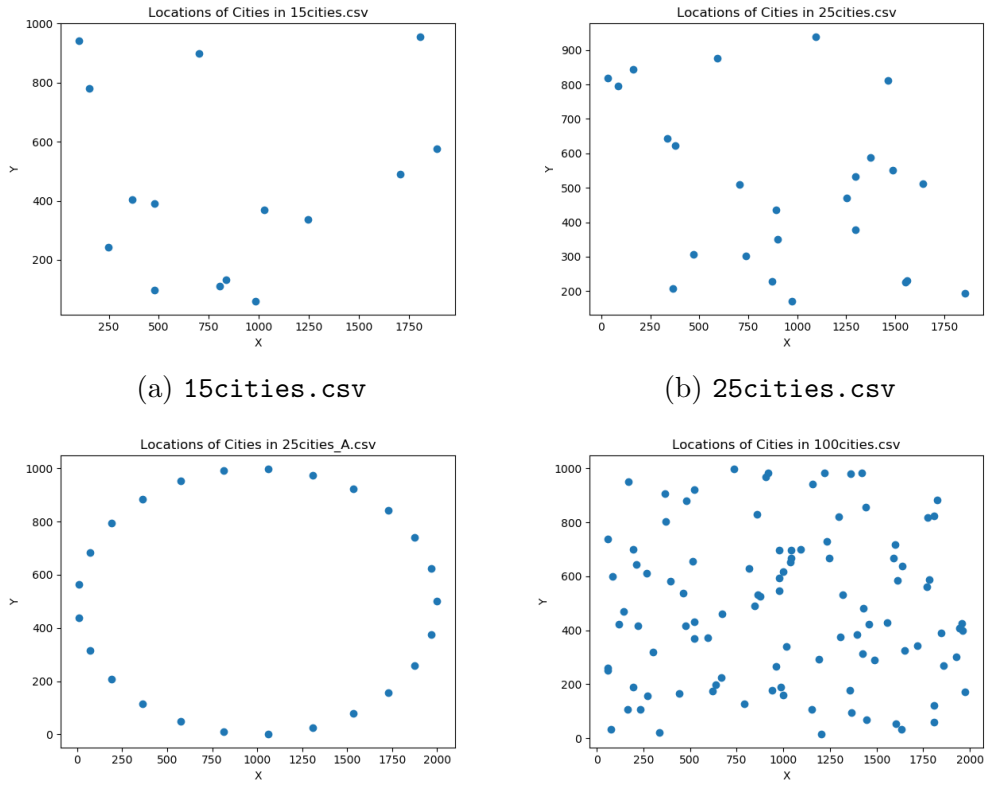
The Traveling Salesman Problem is a well-explored problem that has been shown to be NP-Complete. Thus, the solution space dramatically increases as problem size grows, making it infeasible to perform a brute force search for solutions. In this report, we implement three different search algorithms (Epsilon-Greedy, Simulated Annealing, and Evolutionary Algorithms) and evaluate their performance on various sizes of the traveling salesman problem.

2 Traveling Salesman Problem

The traveling salesman problem (TSP) asks, given a set of cities, find the shortest path that will visit every city. This problem has been proved to be NP-hard, thus solution space scales polynomially. This makes a brute force search infeasible. Instead of finding the solution to TSP, we use search algorithms to find approximately "good" solutions.

3 Our Datasets

In this project, we are presented with four data sets consisting of x, y coordinates of towns to visit. These data sets consist of 15, 25, and 100 cities, as well as a second 25 city dataset with a different city distribution. To get a better sense of how the cities are distributed in each dataset, Figure 1 visualizes where the cities are located for each data set.



(a) `15cities.csv` (b) `25cities.csv`

(c) `25cities_A.csv` (d) `100cities.csv`

Figure 1: Plots of city locations for the four data sets.

We observe relatively well distributed points for all but the `25cities_A` dataset. The `25cities_A` dataset have its cities arranged in a circle and it would be interesting to compare the results between `25cities` dataset and `25cities_A` dataset.

4 Experimental Setup

To evaluate the performance of each of the three algorithms, we encode the datasets into distance matrices, where the x,y-th entry corresponds to the distance between city x and city y. Then we ran each of our three algorithms on the four datasets. In order to account for randomness in initialization and selection, we performed 10 runs with different seeds for each trial. For each run, we record the shortest path found at the end of the algorithm, the number of unique solutions the algorithm explored, and the CPU time necessary to run the search algorithm. In the next sections, we describe our three algorithms and their implementation details.

4.1 Our Three Algorithms

4.1.1 Epsilon Greedy Search

For epsilon greedy search, we begin by choosing a random state, that is a random order of cities to visit. Then for every epoch, we perform a single swap operation, randomly choosing two cities and swap the order in which we go to each city. Then we evaluate the performance of our new state. Out of our current and new state, we choose the better state (state with shorter path length) with probability $1 - \epsilon$ and choose the worse state with probability ϵ.

For the experiments in Section 4, we use $\epsilon = 0.001$ and run for 100,000 epochs.

4.1.2 Simulated Annealing

In simulated annealing, we also begin by choosing a random state. Then for every epoch, we generate a new state by performing a swap operation. If the new state has a shorter path length than the old state, then we immediately select the new state. Otherwise, we calculate the temperature and select the new state with probability

$$P(select\ newstate) = \exp^{\frac{current\_score - new\_score}{temperature}}$$

.

For the experiments in Section 4, we use the following linear temperature function:

$$temperature = 100000 - 0.01 * t$$

where t is the current epoch number. We have also tried other temperature functions such as exponential ($temperature = 100000 * 0.995^t$) and logarithmic ($temperature = \frac{100000}{\log(1+t)}$), but we found that the linear function gave the best results. In our experiment, we run simulated annealing for 100,000 epochs.

4.1.3 Evolutionary Algorithm

Finally, for the evolutionary algorithm, we initialize a population of states. For every epoch, we mutate every state by performing a random number of swaps. The number of swaps is randomly decided between 1 and $\lfloor 0.3 * number\ of\ cities \rfloor$. We then create a new pool by combining our current states and the new states and then use a binary tournament to trim our population back to its original size. The binary tournament strategy randomly pairs members of the population and selects the path with the lowest length.

For the experiments in section 4, we start with a population of 50 states and run 2000 epochs of training.

5 Results

In this section, we examine the average run time, number unique solutions explored and shortest path found for each of the three algorithms. We also provide a performance comparison between `25cities` and `25cities_A` dataset. Each of the three algorithms were run ten times on each of the four datasets. In general, the experiments seem to be relatively repeatable, with small standard deviations observed across all runs.

5.1 Run Time

First, let us examine the performance of each algorithm in terms of time taken to run each algorithm. The results of our experiments can be found in Figure 2 and Table 1.

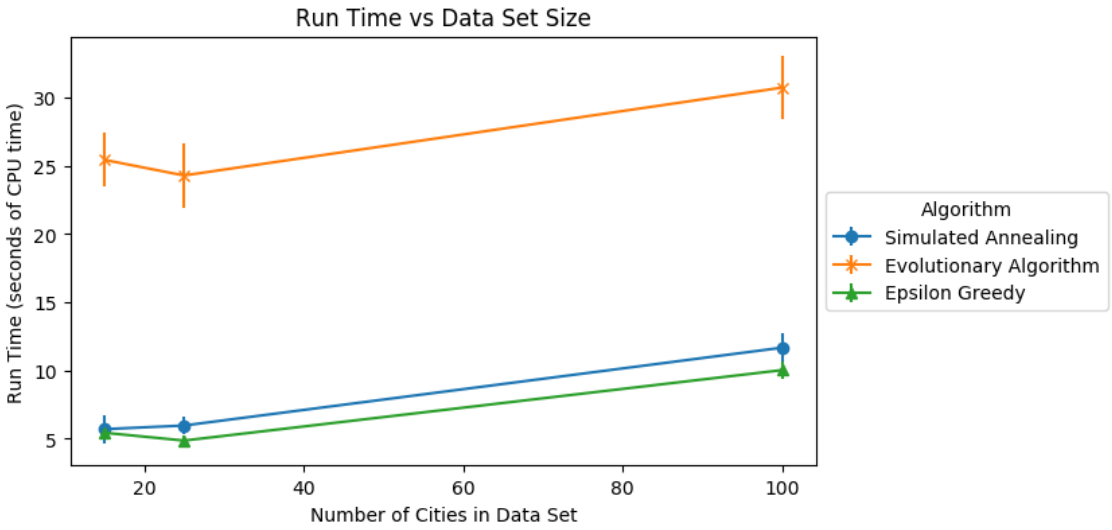


Figure 2: A plot of the average run time in CPU seconds over 10 trials given number of cities. See Table 1 for values.

	Simulated Annealing		Evolutionary Algorithm		Epsilon Greedy	
Number of Cities	**Mean**	**St. Dev**	**Mean**	**St. Dev**	**Mean**	**St. Dev**
15	5.6779	1.0061	25.4266	1.954	5.4161	0.5437
25	5.9441	0.6625	24.2803	2.3365	4.84	0.4116
100	11.6555	1.0368	30.724	2.3127	10.0091	0.6635

Table 1: Average run time in seconds of CPU time found across 10 trials

These results were pretty close to what was expected of each algorithm. Epsilon-Greedy was the fastest running algorithm due to its simplicity. It only needed to mutate a single time per epoch and deciding whether to keep the new successor was decided via a single call to the random function.

Simulated annealing took a bit longer than Epsilon-Greedy. This is expected since it has the added complexity of having to calculate a temperature and compute a new probability threshold over each epoch to decide whether to take the successor state.

Finally, evolutionary algorithm took the longest. This is also expected since the evolutionary algorithm must keep a population of agents and update the whole population every epoch. This additional overhead leads to a much slower run time compared to the two other algorithms.

Noe that even though the evolutionary algorithm and epsilon greedy algorithm appear to perform faster in the 25 city case than the 15 city case, the mean computation times are within one standard deviation of each other and thus the difference is not statistically significant.

5.2 Amount of Solution Space Explored

Let us look at the amount of unique solutions explored. The results of our experiments are summarized in Figure 3, Table 2, and Table 3.

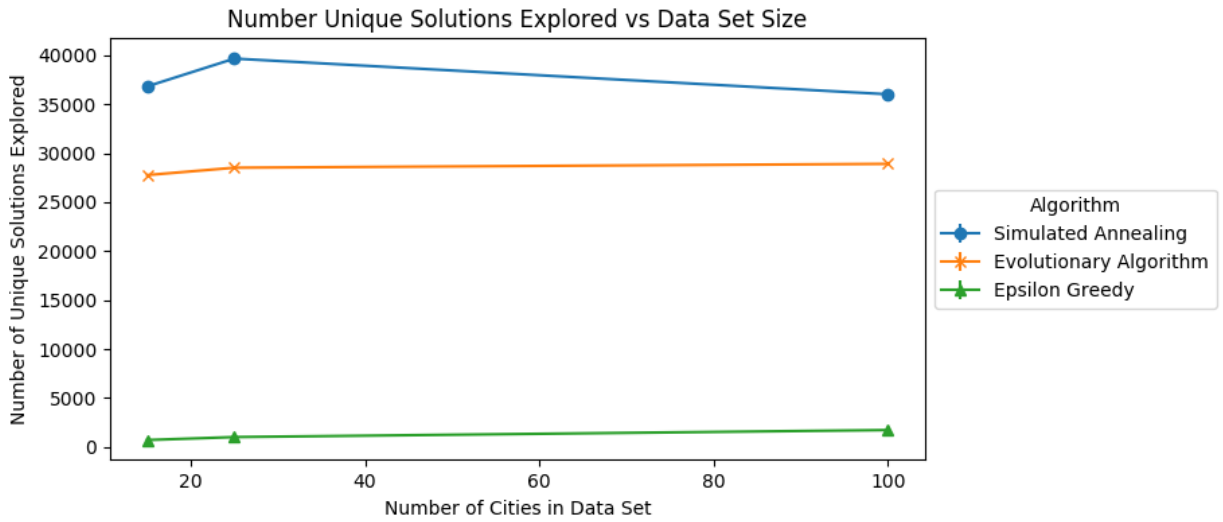


Figure 3: A plot of the average number of unique solutions found over 10 trials given number of cities. Note that the standard deviations are 2 orders of magnitude below the data, thus cannot be seen on the plot. See Table 2 for values.

	Simulated Annealing		**Evolutionary Algorithm**		**Epsilon Greedy**	
Number of Cities	**Mean**	**St. Dev**	**Mean**	**St. Dev**	**Mean**	**St. Dev**
15	36813.8	273.6892	27762.8	176.0374	724.7	53.6899
25	39650.7	168.0637	28513.4	164.2201	1025.1	119.8444
100	36011	225.8438	28907.8	101.6817	1741.9	170.5652

Table 2: Average number of unique solutions found across 10 trials

	Simulated Annealing		Evolutionary Algorithm		Epsilon Greedy		Total Solution Space Size
DataSet	Num. Solutions Generated	% Explored	Num. Solutions Generated	% Explored	Num. Solutions Generated	% Explored	
15	36814	10^{-6}	27763	10^{-6}	725	10^{-8}	15!
25	39651	10^{-19}	28513	10^{-19}	1025	10^{-21}	25!
25A	27774	10^{-19}	28501	10^{-19}	876	10^{-21}	25!
100	36011	10^{-152}	28908	10^{-152}	1742	10^{-153}	100!

Table 3: A table summarizing the number of unique solutions generated by each algorithm on each data set as well as the percentage of the total solution explored by each algorithm. Note that only the order of magnitude of the percentage is listed considering how small the percentage is.

First, we observe that the Epsilon-Greedy algorithm does not explore much of the solution space at all. This is because once Epsilon-Greedy has found a relatively good solution, it has low (ϵ) probability of leaving the good solution to explore alternative paths. Additionally, since the mutation results in a state that is similar to the parent state with only one pair of towns swapped, it does not explore very far before reentering a local optimum.

It is interesting to note how the number of unique solutions explored seem to remain relatively constant or even decrease with number of cities in the data set. This indicates that the algorithms are finding local optima and not exploring as much as they can. Some potential solutions to addressing this would be to have a smaller epsilon (for epsilon-greedy), using a slower decreasing temperature function (for simulated annealing).

In the case of the evolutionary algorithm, this indicates that the population is becoming too homogeneous and not maintaining diversity. This could be a result of choosing binary tournament as my selection process. One way to address this problem would be to include more randomness in the selection process.

I did try adding an additional randomness component by modifying my algorithm to randomly select 10% of the population to automatically win in the tournament and be in the next generation. This did increase my number of unique solutions explored to be on the order of 70,000 versus 30,000 from before. Unfortunately this also had an added side effect of decreasing my

solution quality because it had the chance of removing the best solution from the population.

Given more time, I would have like to try considering a different way to add randomness while still maintaining the guarantee that the best solution remains in the pool.

In terms of the number of solutions found by each algorithm, we see that all the algorithms only explore a very small section of the search space. As the problem grows larger, the algorithms do search more of the solution space; however the solution space grows much faster, thus leading to the very small percentages seen in Table 3. This is because for a traveling salesman problem consisting of n towns, there are $n!$ unique solutions. This is already a very large amount for the 16 city case and completely infeasible to brute force as n grows larger.

5.3 Solution Quality: Shortest Path Length Found

Let us examine the shortest paths found by each algorithm. The results from our trials can be found in Figure 4 and Table 4.

	Simulated Annealing		Evolutionary		Epsilon Greedy	
Number of Cities	**Mean**	**St. Dev**	**Mean**	**St. Dev**	**Mean**	**St. Dev**
15	4110.1	208.6	4460.0	262.4	4857.4	435.2
25	5263.5	210.8	6126.6	381.8	6033.8	587.3
100	18087.9	815.8	31649.2	1434.1	22429.5	1917.9

Table 4: Average shortest path length found across 10 trials

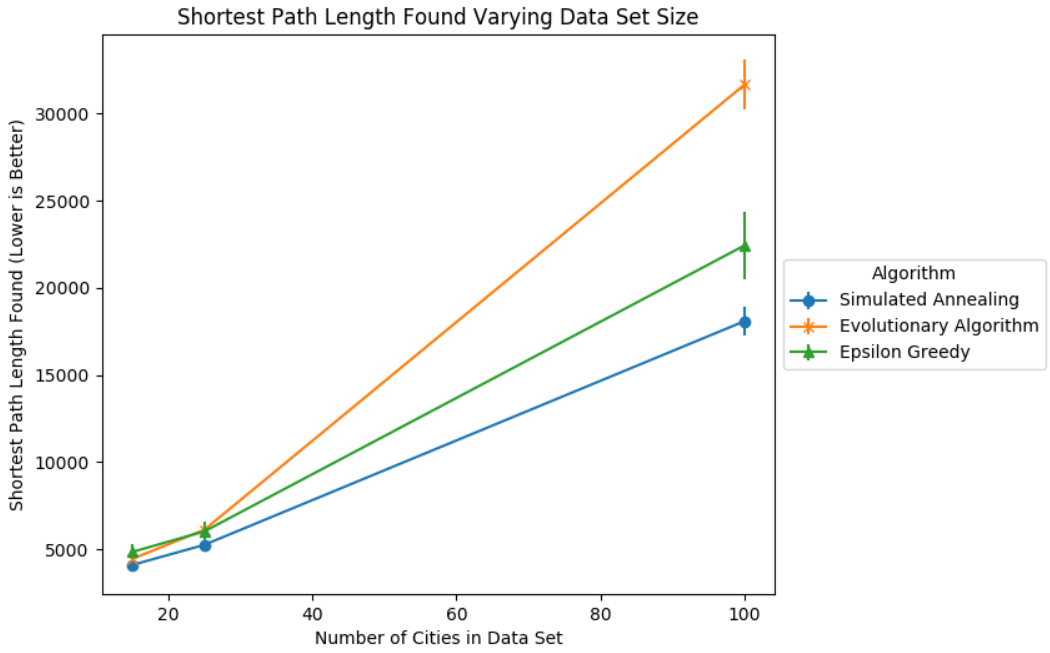


Figure 4: A plot of the average shortest path found over 10 trials given number of cities. See Table 4 for values.

From our experiments, we observe a couple of interesting trends. In all cases, simulated annealing outperforms the other two algorithms. While this is expected for epsilon greedy, the performance difference between simulated annealing and evolutionary algorithm was surprising. This discrepancy seems increases as we increase the size of the problem. For the 100 city case, we see that simulated annealing outperforms our evolutionary algorithm, finding paths that are half the length of solutions found by our evolutionary algorithm for the 100 city case.

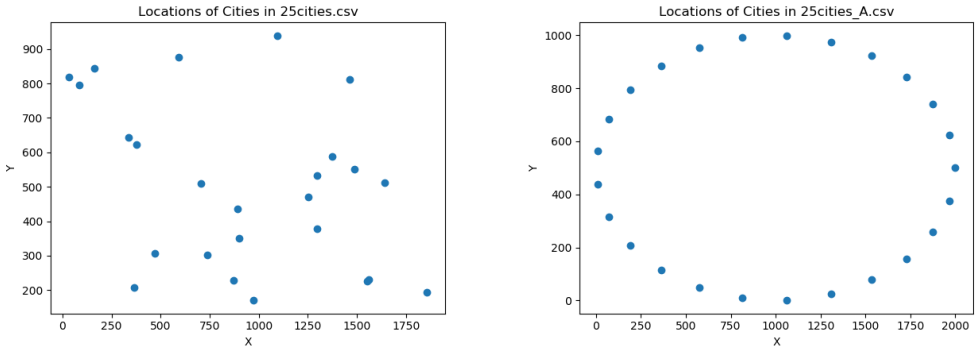
The reason for evolutionary algorithm's poor performance was touched upon previously in Section 5.2. The paths with poor performance were selected out from the population too quickly even with the added randomness of pairing in binary tournament. As proposed before, I hypothesize a selection strategy that would maintain higher diversity via randomness would greatly help the evolution algorithm's performance.

Another aspect of the problem is the number of epochs and agents used. For my experiments, I chose to run the evolutionary algorithm with 50 agents and 2000 epochs due to constraints in processing power and time. The algorithm could potentially yield better solutions if allowed to run longer and with the diversity-maintaining selection strategy described above.

One interesting comparison to look at is the performance difference between simulated annealing and epsilon greedy. Recall the two algorithms are very similar, except simulated annealing has the added complexity of maintaining a temperature function to allow for more exploratory behavior in the first epochs. This added complexity is reflected in simulated annealing's longer run time compared to epsilon greedy. However, one can see the payoff with simulated annealing finding paths around 5000 units shorter than epsilon-greedy for the 100 city case.

5.4 Effect of City Distribution

Finally, let us look at the impact of different city distributions on the performance of each of the algorithms. In Figure 5, we once again have the plots of city locations of each data set. The results of our experiments are summarized in Tables 5 , 6, and 7.



(a) `25cities.csv` (b) `25cities_A.csv`

Figure 5: A plot of the city locations for the two 25 city data sets. Notice the difference in city distribution.

	Simulated Annealing		Evolutionary Algorithm		Epsilon Greedy	
Data Set	**Mean**	**St. Dev**	**Mean**	**St. Dev**	**Mean**	**St. Dev**
25 cities	5.9441	0.6625	24.2803	2.3365	4.84	0.4116
25 cities_A	5.0452	0.3535	22.9648	1.7511	5.2448	0.1667

Table 5: Average run time in seconds of CPU time across 10 trials.

	Simulated Annealing		Evolutionary Algorithm		Epsilon Greedy	
Data Set	**Mean**	**St. Dev**	**Mean**	**St. Dev**	**Mean**	**St. Dev**
25 cities	39650.7	168.0637	28513.4	164.2201	1025.1	119.8444
25 cities_A	27774.1	364.2143	28500.7	144.9	876.3	138.7934

Table 6: Average number of solutions explored across 10 trials.

	Simulated Annealing		Evolutionary Algorithm		Epsilon Greedy	
Data Set	**Mean**	**St. Dev**	**Mean**	**St. Dev**	**Mean**	**St. Dev**
25 cities	5263.491	210.8897	6126.581	381.8469	6033.831	587.3326
25 cities_A	4839.333	441.0361	7689.575	912.9915	7049.779	1184.039

Table 7: Average shortest path length found across 10 trials.

First, comparing the running time between the two data sets, both took around the same amount of time to run for each algorithm. This makes sense as the basic mechanisms of the algorithms are the same, even with the differing distribution of data. Since both data sets have the same number of cities, the necessary computation remains.

Next, looking at the average number of unique solutions explored, we see that running our algorithms on the circle-distributed `25cities_A` dataset yields fewer unique solutions explored for simulated annealing and epsilon greedy. This indicates that with the circle distribution, the algorithms converge on some local minima and do not explore as much as with the more randomly distributed dataset (`25cities`). We hypothesize that this is because with the randomly distributed dataset, the algorithms have more incentive to explore other states because the wide distribution cities allows more states that have higher reward value.

It is interesting to note that the evolutionary algorithm does not show significant difference in states explored. This could be caused by the fact that the evolutionary algorithm starts with a randomly initialized population of states, thus it has more of a chance than the other algorithms to continue to explore those unfit states until they are selected out of the population.

Finally, looking at the solution quality, there is actually little statistical difference between the solution qualities, as they are within a standard deviation of each other for all but the evolutionary algorithm. The algorithms' performance on the circularly distributed dataset follows the same trend as the widely distributed, with simulated annealing performing the best, then epsilon greedy, followed by the evolutionary algorithm. The performance difference between the evolutionary algorithm and epsilon greedy are not significant with the rather large standard deviations for both.

6 Conclusion

In this report, we used simulated annealing, evolutionary, and ϵ-greedy algorithms to find approximate solutions for the traveling salesman problem. Between simulated annealing and ϵ-greedy, we showed that the additional complexity and therefore running time of simulated annealing allows it to yield better solutions than ϵ-greedy. On the evolution algorithm side, we learned the importance of maintaining diversity in the population by including more sources of randomness in the selection process. We also showed that despite searching very little of the massive solution space, our search algorithms were able to find relatively good solutions.