An Analysis of the Performance of a Modified ERX Genetic Algorithm Applied to the Traveling Salesman Problem In Comparison to Other Common Solutions

Niels Turley

Braydon Jones

Brigham Skarda

Abstract

The traveling salesman problem is a very popular np-hard problem that has attracted the attention of many professionals because of its trivial understanding and non-trivial solutions along with having a range of relevant applications in the world. After researching many different strategies and given the time needed to understand and implement some of the leading algorithms in this field, our team decided to implement an original version of the genetic algorithm taking influence from the Edge Recombination Crossover (ERX). The algorithm is capable of solving 45 cities in under two minutes. We left the algorithm open to a range of potential future modifications that could further optimize the approach.

1 Introduction

The Traveling Salesperson Problem (TSP) objective is to find the shortest possible path between a set of cities, touching each city exactly one time and ending where the route began. We use the genetic algorithm implementation to solve TSP, drawing from the ERX (Edge Recombination Crossover), originally proposed by Whitney et al. (1989). First, this paper will compare and contrast the effectiveness and overall approach to other popular Traveling Salesman Problems (TSP) such as the greedy and branch-and-bound algorithms. We then detail our genetic algorithm including an explanation of the process for selection, crossover, mutation, replacement, and termination. We will conclude this paper by exploring other modifications that could be implemented with our algorithm that may lead to better results.

2 The Greedy Approach

2.0 Introduction

The greedy algorithm creates a valid path between cities by starting with a random city and picking the nearest neighbor of that city. If there is a cycle created prematurely, then it starts over but with a different starting city.

2.1 Space Complexity

The space complexity for this algorithm is O(n). The reason for this is that the algorithm never has to store more than just a list of all of the cities. It does need to know all of the neighbors of each city, but since we are doing this on a city by city basis and not storing all of these neighbors at one time, our space never exceeds the list of cities, or n.

2.2 Time Complexity

The time complexity for this algorithm is $O(n^2)$. This is because for each city in our list, we have to find the nearest neighbor. The amount of neighbors that we have to look through is proportional to n. That means overall we have a time of $O(n^2)$.

Here are the parts of the code that specifically contribute to this complexity:

- Because the greedy algorithm starts from a random city, it is possible that there is not a valid path starting from that city. This means that we may have to loop through a few cities to find a valid path. In an absolute worst case scenario this may loop through all n cities. This would increase the time complexity to $O(n^3)$, but the reality is that the vast

majority of the time the greedy approach will find a valid solution within the first couple iterations.
- After selecting our start city we loop through all of the other cities to find the next closest unvisited city. This is O(n).
- Then we go to the closest city and find the next closest unvisited city to that one. In total we will have to visit n cities.

This makes our total time complexity $O(n^2)$ as we have to visit n cities, and for each city we have to check n cities to find which one is the closest.

3 The Genetic Approach

3.0 Introduction

In general, a genetic algorithm is an algorithm that seeks a stable state by modeling the natural process of evolution. Genetic algorithms can be described in four steps:

1. Creation of an initial population.
2. Selection of generally good individuals from that population that will pass their attributes on to the next generation.
3. Generation of children through crossover and mutation.
4. Repeat steps 2 and 3 until some level of convergence is achieved.

This type of algorithm expands on the local search approach in that it tries to, in a random but systematic way, search many population spaces and make corrections as opposed to just one population space.

While we were able to build most of the genetic algorithm ourselves, the most challenging part of this process for the traveling salesman problem is combining the paths in a valid way. This is known as crossover, and there is much literature supporting various approaches. Our algorithm uses a modified Edge Recombination Crossover technique, which crosses the edges between the two parents (as opposed to the cities crossing over).[1]

3.1 Our Implementation

3.1.1 Creation of Initial population

To apply this algorithm to the traveling salesman problem, we generated an initial population of 50 individuals using the greedy algorithm described in section 2 also with some randomness.

3.1.2 Selection of Parents

From that population, we took the five best individuals (the “elites”) to automatically continue on to the next generation without modification. This allows the best solutions to remain in the population at the cost of some diversity.

The remainder of the parents were determined using tournament selection. Within each tournament, five individuals were randomly selected from the population. The individual with the best fitness (lowest path cost) was used as a parent for the next generation.

3.1.3 Generation of Children

There are two steps in the creation of the children: crossover and mutation.

For crossing over, we used an approach similar to Edge Recombination Crossover (ERX). The ERX crossover operator works by taking two parents from the population and creating an "edge map" for each edge in both of the parents. The child is generated by randomly picking a city, then using the edge map to determine the closest neighbor that city has. Note that this is different than a simple greedy approach, as the chosen neighbor is not necessarily the closest neighbor to the city, but is instead the closest neighbor found in the parents' edges (which may not include the nearest city). This closest neighbor is added to the child's route, and the process is repeated.

The mutation operator works by randomly swapping a random amount of edges (we found $Pr_{\text{Edge Switch}}[0.1\%]$ to work well for our tests) from this new child. This introduces an element of genetic randomness to the children that provides the possibility of finding better paths.

For both the crossover and the mutation operators, repair mechanisms were used to fix the child if the route generated was invalid. As the edges were being added during crossover, if the proposed neighbors weren't valid to add to the current route, a 2-opt local search was performed to insert the neighbor into a more appropriate spot within the child's route. During the mutation, if the mutation created an invalid child, the random mutation was performed again on the originally generated child. Both of these repair mechanisms had escape conditions that prevented infinite loops from occurring, but could lead to the addition of an invalid child to the next generation.

3.1.4 Repeat

From here we continue the selection of parents and generation of children until the algorithm hit a stagnation limit, which was determined when the best solution found did not change for a set number of iterations (10 * the problem size).

3.2 Space Complexity

The space complexity of the genetic algorithm is largely dependent on the size of the population being used (p). Just like the greedy algorithm, we have to store the path between all of the cities, giving us O(n) space complexity. Unlike the greedy algorithm we have to store multiple paths at a time (one for each individual in the population). This means that our space complexity comes out to O(np). As long as the population is constant though, it can be factored out. In our case we set the population to always be 50 individuals. This means that our total space complexity can be simplified to O(n).

3.3 Time Complexity

The time complexity for this algorithm is $O(gn^2)$ where g is the number of generations created. g is a difficult variable to give a theoretical value to because every run of the algorithm will make a different number of generations. Thus the best that can be done is estimating the value of g through empirical analysis.

Besides g, the n^2 is the best we can do because we used the greedy algorithm to generate our initial population.

Here are all of the parts of the genetic algorithm that could contribute to the total time complexity:

- Creation of the initial population. We use the greedy algorithm p times to get the initial population. This gives a time complexity pn^2. Since our population is a constant 50 we can factor it out to get a time complexity of $O(n^2)$.
- Selection of parents for the next generation is O(n) because we go through all of the individuals in the current generation once and select the good ones to be parents.
- Edge Recombination Crossover involves looping through each city, and its neighbors as well. Since each city could possibly have n-1 neighbors the complexity for Edge Recombination Crossover is $O(n^2)$
- Mutation is worst-case O(n). This is because we only allow n number of swaps to occur. The reality is that only about 5% of the cities will be swapped around during a mutation.
- Children creation is $O(n^2)$. We must generate up to p new children. Every child requires an edge recombination, and a mutation. Thus we get $O(p * (n^2 + n))$ which simplifies to $O(n^2)$.

To summarize the whole genetic algorithm we have the following steps:

Initial Population Creation $O(n^2)$

Generation $O(g)$
(Following steps are done g times).

Parent Selection $O(n)$

Child Creation

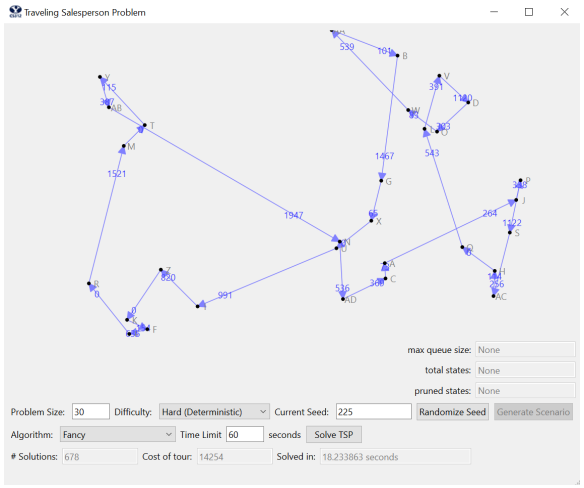
ERX $O(n^2)$

Mutation $O(n)$

Total: $O(n^2 + g*(n +(n^2 +n))$

Simplified Total: $O(gn^2)$

3.4 Typical Result



3.5 Pros and Cons

We decided upon this genetic algorithm because it would effectively allow us to look at many solutions to the problem at once and, in theory, take the best parts from each solution while making minor improvements and changes along the way.

The mutation mechanic allows new possibilities to constantly be introduced to solution, and hopefully find a better solution.

One of the major cons of such an algorithm is the unpredictability in the number of generations that will be made. This can be somewhat offset by setting a time limit though. Similar to the way in which branch and bound works best with a time limit.

Another major con of this approach is the large number of sample spaces that need to be analyzed makes it somewhat slower when compared to the greedy algorithm.

4 Results

4.0 Data

To compare our genetic algorithm, we tested it against other popular TSP solution algorithms including random permutation, greedy, and branch & bound. Our results are listed in the table below, respectively. TB marks problem attempts that were unable to find a solution in the allotted time constraint (t=10 minutes or 600 seconds).

	Random		Greedy			Branch & Bound			Genetic		
# Of Cities	Time	Path Length	Time	Path Length	% of Random	Time	Path Length	% of Greedy	Time	Path Length	% of Greedy
15	0.0014	19595	0.00044	12428	0.64	7.89	9436	0.76	2.38	9930	0.8
23	0.0024	31535	0.00063	16245	0.52	228.95	9592	0.74	16.96	13210	0.83
30	0.027	41311	0.0014	18815	0.46	TB	TB	TB	17.69	16062	0.86
45	0.82	61159	0.00256	23244	0.38	TB	TB	TB	108.9	21310	0.92
60	55.21	85106	0.0044	27685	0.33	TB	TB	TB	359.18	30520	1.1
100	TB	TB	0.012	36665	NA	TB	TB	TB	390.96	62295	1.7
150	TB	TB	0.031	51226	NA	TB	TB	TB	TB	TB	TB
200	TB	TB	0.022	58533	NA	TB	TB	TB	TB	TB	TB

Table 1: Time and path test data for selected algorithms.

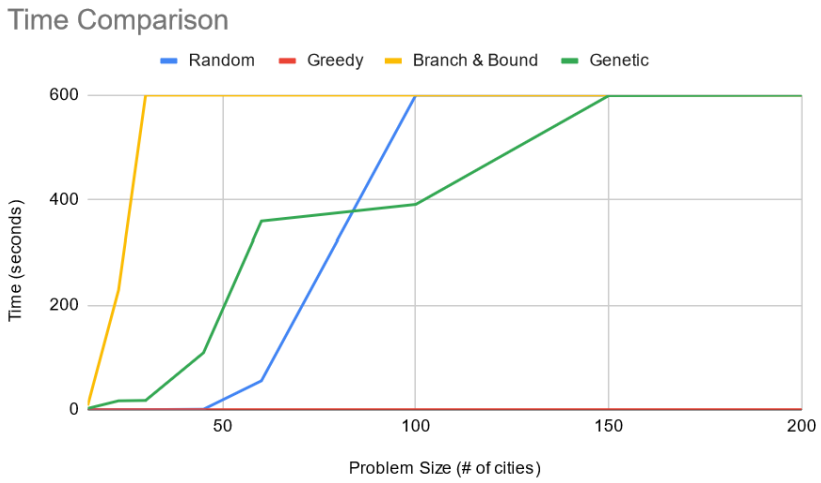


Figure 1: Time comparison of selected algorithms. Note that the upper bound (t=600 seconds) indicates the algorithm ran to the time bound and was manually stopped.

Figure 2: Solution comparison of selected algorithms. Units of path length are arbitrary distance values between cities. Line values are given for tests where the algorithm could find a solution.

4.1 Analysis

The exponential nature of the branch and bound algorithm becomes apparent when comparing it to the other algorithms as it quickly exceeds the time limits we set to find its optimal answer.

What is particularly interesting is the use case of our algorithm. The time comparison shows that our genetic algorithm does find a solution nearly as quickly as the greedy algorithm. But for problem sizes of up to about 50 cities, the genetic algorithm finds a solution that is about 10-15% shorter than the greedy algorithm. That is to say that if one is willing to wait slightly longer for a problem less than 50 cities, they could get a more optimal solution than the greedy algorithm provides.

One of the major uncertainties about the genetic algorithm was the number of generations it would make, as this would affect time complexity. Our data set wasn't quite large enough to give us a good conclusion on this. But it appears to be somewhere between a constant of 3 and n. We come to this conclusion because the time to solve is growing less than would be expected for a time complexity of n^3, but slightly faster than would be expected of a constant factor.

In either case it is also apparent that the constant factor accompanying the genetic algorithm is much larger than that of the greedy algorithm. While the greedy algorithm is $O(n^2)$, the constant factor is so small that the line on the time comparison table is flat.

5 Conclusion and Future Steps

Our modified ERX approach performs well compared to other common algorithms such as random permutation and simple greedy, though it may take some time to find good solutions. Further testing would find the ideal genetic constants: population size, parent size, mutation rate, constant of convergence, and elite size. We suspect that this could be done via a machine learning approach where, over time, the algorithm would learn on its own which values are ideal based on the problem at hand. Another modification could be implementing an edge-sensitive mutation operator, such as displacement (shifting a subset of edges) or inversion (reversing a subset of edges), as the major selling point of ERX is the preservation of important edges.

6 References

1. [Larranaga, Kuijpers,Murga, and Dizdarevic 1999] P. LARRANAGA, C.M.H. KUIJPERS, R.H. MURGA, I. INZA and ˜ S. DIZDAREVIC. "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators." Artificial Intelligence Review. 129-170
2. Whitley, D., Starkweather, T. & D'Ann Fuquay (1989). Scheduling Problems and Travelling Salesman: The Genetic Edge Recombination Operator. In Schaffer, J. (ed.) Proceedings on the Third International Conference on

Genetic Algorithms, 133–140. Los Altos, CA: Morgan Kaufmann Publishers.