Approaching the Traveling Salesman Problem using a Genetic Algorithm

Joe Mansfield, Derek McKinlay, Bright Mazura

April 10, 2022
Professor Tony Martinez
Algorithm Design and Analysis
BYU Computer Science

Abstract
A genetic algorithm is an algorithm that attempts to create optimal solutions to a problem through simulated evolution. This paper focuses on the genetic algorithm that we created in order to solve the Traveling Salesman problem. We analyze the complexity of the algorithm, compare it to other algorithms for several sizes of the problem in an empirical analysis, and discuss ways in which it could be improved.

Introduction
The traveling salesman problem is a classic example of an NP-complete problem, a problem that cannot be solved in polynomial time. Since the processing power required to solve such problems becomes astronomically large quite quickly as the size of the problem increases, this has led to the creation of many creative methods to approximate a solution over the years. Our group has attempted to solve the problem using a genetic algorithm. Genetic algorithms work by taking existing solutions and simulating evolution of those solutions. This is done by crossing two solutions together, mutating solutions, introducing new solutions, and prioritizing the best solutions over multiple generations.

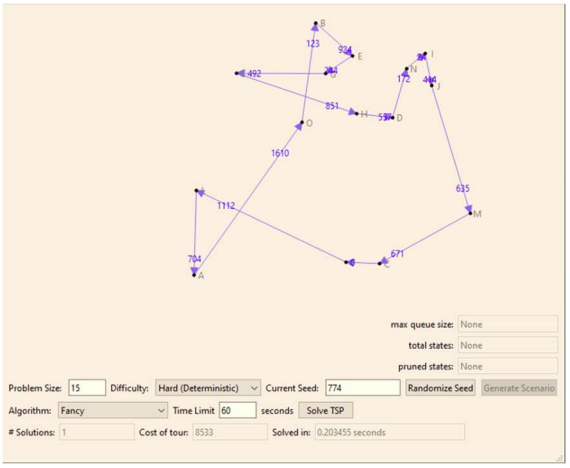
Genetic Algorithm Implementation and Analysis
Our algorithm starts by giving it the solution created with a greedy algorithm, as well as multiple randomly generated solutions until the population size reaches p, the value set by the user. The complexity of the greedy algorithm is $O(n^2)$. Ironically for a sparse graph, the randomly generated solutions are significantly less efficient than greedy.
Once the initial population has been generated, a loop runs for g times, with g being the set number of generations. For each time through the loop, it first copies the old generation, then generates new solutions and adds them to the generation using three methods.
The mutate function simply swaps two cities in the path. This is controlled by the mutation rate constant set by the user which is the percent chance that each member of the population will be mutated. The function is constant time, runs for every member of the population, and this loop runs once per generation, resulting in a total complexity of $O(pg)$
The cross over function takes a random slice from one of two paths fed into it, uses that as the base for a new path, then puts each city except for the cities included in the slice on to the end in the same order that they were in on the other path. The paths are decided by using a fitness score as a weighted probability that the path will be selected in order to mimic the way that more fit organisms are more likely to procreate in nature. We used the inverse of the path length as the fitness scores. Since this function adds each city one by one to the

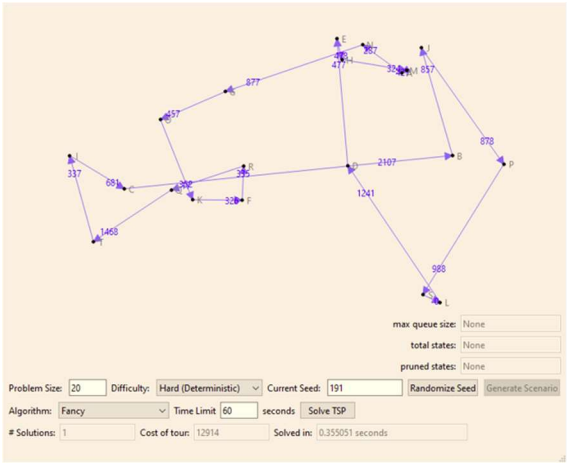
new path after checking to make sure they are not already in the path, the complexity is O(n). It runs each generation for a fraction of p times decided by the user resulting in a total complexity of O(npg).
In each generation, new randomly generated solutions are also added in order to introduce variation that can help escape local minima.
As the newly generated paths are added to the new population their fitness score is calculated and added to their solution object in order to avoid recalculating them. The cost of calculating a fitness score is O(n) because it is done by taking the inverse of the path length which must be generated by running through the path. This is done for every new solution generated, resulting in a total complexity of O(npg).

At the end of each generation, the new population is culled down to the original size after being sorted based on fitness values. The sort function in python has a complexity of O(nlogn). After sorting all members past the first p are removed, which is constant time. Since the size of the new population is a multiple of p and this runs each generation, the total complexity is O(gplogp).
The old population is then replaced by the new population and the loop runs g times. Adding the total complexity of each function we get a cost of $O(n^2 + p*\text{cost of random} + pg + npg + \text{cost of random} + npg + gplogp)$ which simplifies to $O(n^2 + p*\text{cost of random} + npg + gplogp)$

Empirical Performance

	Random		Greedy			Our Algorithm		
Cities	Time (sec)	Path Length	Time (sec)	Path Length	% of Random	Time (sec)	Path Length	% of greedy
15	0.000808	19716.4	0.0035882	11220.4	0.569089692	0.1771238	9549.4	0.85107483
20	0.001796	24719	0.0083706	12940	0.52348396	0.3678362	12416	0.95950541
25	0.0115682	34317.4	0.016177	15364	0.447702915	1.1495246	14384.4	0.93624056
30	0.0175674	40922	0.02814	16551.4	0.404462148	4.27478	16350.2	0.98784393
40	0.4789238	54261	0.0676256	19147.8	0.352883286	45.341117	19079.4	0.99642779
60	23.668255	79291.75	0.2455574	24768.6	0.312372977	TB	TB	TB
100	TB	TB	1.2512666	35832.2	N/A	TB	TB	TB
200	TB	TB	12.544147	56554.6	N/A	TB	TB	TB





Especially at small numbers of cities, our algorithm consistently generates solutions that are significantly better than greedy, but the time quickly suffers as the number of cities increases. This is largely due to the inefficiency of creating large numbers of random solutions. The path improvement compared to greedy also decreases as the number of cities increases.

Conclusion and Further Improvements

The obvious improvement that could be made is finding a better way to generate new solutions compared to creating random solutions. This could possibly be done by modifying the greedy algorithm to introduce a variable that when changed would create worse solutions that could then be used in the genetic algorithm. There are also many variables, such as population size, number of generations, mutation rate, and number of crossovers that could be modified to find the optimal values for this problem.

The main strength of a genetic algorithm is that it can create a better solution using preexisting solutions, so it is not the best way to solve the Traveling Salesman Problem without an efficient way to generate new solutions or an existing database of solutions. It is still an interesting way to look at the problem, performs quite well on simpler versions with fully connected maps, and could become a powerful method of solving the problem with improvements.

References

Mishra, C., 2022. *Traveling Salesman Problem using Genetic Algorithm - GeeksforGeeks*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/> [Accessed 11 April 2022].

Shiffman, D., 2022. *Traveling Salesperson with Genetic Algorithm*. [online] The Coding Train. Available at: <https://thecodingtrain.com/CodingChallenges/035.4-tsp.html> [Accessed 11 April 2022].

Yang, X., n.d. *Nature-inspired optimization algorithms*. 2nd ed.