Solving Traveling Salesman Problems Using Genetic Algorithms

Abstract

Evolution is organic life's tool for surviving and adapting to their environment in an optimal way. Over the course of many generations changes can sweep across an organism and drastically alter it in a way that allows it to be more successful in its environment by solving problems it encounters regularly however, this takes a very long time to occur. Genetic algorithms are algorithms designed to emulate natural evolution and solve problems through many generations of development on a computer. Genetic algorithms can be applied to solve many different scenarios but for this project I will be focusing on their application to traveling salesman problems in specific. In a traveling salesman problem there is a list of cities that need to all be visited once and only once but the salesman wants to traverse all of the cities in the shortest overrall distance. With genetic algorithms the traveling salesman problem can be solved simply through many iterations of the algorithm and although it might not always end up with the absolute best solution it will always be a better solution than random. This project is focused on how a genetic algorithm can be tweaked and optimized to solve a traveling salesman problem very effectively.

1 Introduction

1.1 Background on Genetic Algorithms

In order to more completely understand genetic algorithms it is important to first have a little background information about how organic evolution, which they are based on, optimizes things in the real world. Evolution is extremely interesting to me since it revolves around a fairly simple idea but manages to solve so many things and is the primary reason humans are the way they are today. Evolution revolves around the idea of "fitness," although fitness is typically associated with physical prowess in this case it can be mental as well. Fitness as it pertains to evolution is simply how well something performs in terms of a certain scenario for example, in a situation where the all of the food in a certain environment is in tall trees far from the ground a turtle would be considered to have low fitness and a giraffe would be considered to have a high fitness level. The level of fitness corresponds to what gets carried over to future generations of animals in the area as well since an animal with too low of fitness simply will not be able to eat and survive while animals that have no issue getting to the food will thrive. This idea of only the fit survive can be applied to just about any problem solving scenario.

1.2 Applying Survival of the fittest to Traveling Salesman Problems

With how old the traveling salesman problem is there are many different approaches to it but the most interesting part of the traveling salesman problem to me is its complexity. As humans we can take a quick glance at a map with all of the cities mapped on it and with fairly little effort we can sketch out a path that while not necessarily the best path possible is still pretty close to optimal. That being said it takes software a bit more work to solve the same problem and while there are heuristics such as nearest neighbor and greedy

algorithms that can be applied to solve it I think genetic algorithms are more interesting. With a genetic algorithm its not necessary to use a heuristic in constructing possible solutions and as long as fitness is the driving factor for which candidate solutions are chosen for improvement the optimal solution can be brute forced in a way. Genetic algorithms can solve traveling salesman problems simply through massive amounts of iteration. Starting from completely random initial tours a genetic algorithm can pick the fittest candidate tour from each generation and hold on to it while adjusting the other possible solutions and checking to see if any new solutions are better than the one carried over from the previous generation. By generating and mutating through thousands of generations of populations in the hundreds genetic algorithms can reliably provide optimal solutions to traveling salesman problems.

2 Related Works

Abstract

The traveling salesman problem is well-known problem that has been around for quite awhile. The problem entails a number of nodes representing cities spread about an area where the objective is to travel to all of the cities exactly once in the shortest distance possible. There are a number of ways to solve this problem however I will be focusing on using a genetic algorithm. Genetic algorithms emulate biological evolution and solve problems primarily through iteration and mutation. Over the course of the years many people have applied genetic algorithms to traveling salesman problems and this paper is primarily a review of such efforts.

2.1 Introduction

A genetic algorithm is a procedure in which a computer solves problems in an organic way. Natural selection and evolution are biology's way to solve the problems different species encounter in their environment and allow species to optimize themselves through generations of struggle. Through simulation computers are able to iterate through many generations of solutions for a certain problem thereby evolving them to obtain an optimal solution. There is a bit more to it than that but the process can be condensed into four primary components which are crossover, mutation, population management, and initial population. The first step in applying a genetic algorithm is to create an initial population according to the population size component of the algorithm. Then these possible solutions are combined with other solutions in whats called crossover. After the solutions are mixed together to create more possible solutions they are then mutated to ensure a diverse population. Finally these solutions are judged upon their optimality in regards to the solution which is called their fitness. [2] With these steps completed the cycle then starts again, excluding the initial population generation, and continues until a solution with high enough fitness is achieved.

2.2 Genetic Algorithms as Applied to Traveling Salesman Problems

Applying a genetic algorithm to a traveling salesman problem is fairly straightforward, however certain precautions must be taken to ensure that the algorithm runs smoothly and effectively. The four main components crossover, mutation, initial population, and population management are expanded upon in the following sections and the precautions necessary are explained in their respective sections below.

2.2.1 Crossover

When the algorithm is performing the crossover it is very important for the algorithm to ensure that the resulting solution includes all of the cities to establish a valid solution. Along with ensuring a valid solution the crossover portion of a traveling salesman problem is typically performed by breaking one of the candidates into a smaller set, comparing it directly to the other initial set. There are a number of different approaches that have been studied regarding the crossover portion of genetic algorithms applied to traveling salesman problems that range from purely random to very precise. The first method is to perform gene selection based

on a crossover factor that can favor either one of the two parents or can be set to the middle and completely randomized. [5] Another method is to perform the crossover based on maintaining some of the order from the parents either by grabbing groupings based from the parents or by simply keeping individual cities in the same position and then building around them. Other crossover methods involve using heuristics to favor the more optimal parent of the two during the crossover and emphasize keeping a large chunk of it intact versus the other less optimal one. [6] Crossover sometimes also involves a small amount of mutation but there is also a separate mutation portion involved in genetic algorithms that will be discussed in the next section. Once the crossover is completed the results begin to represent evolved offspring of the previous generation leading into the mutation stage.

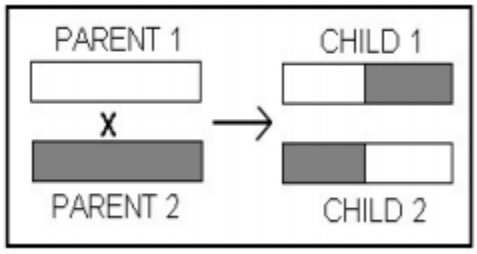

Figure 1: Diagram illustrating the basic concept of crossover. [4]

2.2.2 Mutation

The mutation stage is where the offspring are partially modified without influence from another gene. Modifying offspring after the crossover stage is an important step in a genetic algorithm in that it helps prevent the algorithm from getting stuck or hung up on local extrema. As the algorithm works to converge on an acceptable solution it is possible that the algorithm could end up tunneling and focusing on converging on a local minimum or maximum however, slightly modifying the offspring adds a small amount of divergence. In reintroducing some divergence to the solution the algorithm avoids getting hung up on local extrema thereby ensuring a higher accuracy. All of the studies I have looked into agree that it is important to keep mutations small though as too much mutation can prevent convergence entirely leading the algorithm to fail in finding an accurate solution. For traveling salesman problems most of the mutations performed are very straightforward. One example of mutation is exchange mutation which, as the name implies, simply picks two cities in the tour and swaps their placement. Another strategy for mutation is to pick a random city in the tour and just move it to another position in the tour while maintaining the order of the other cities. Mutation strategies can be more complex though and strategies such as inversion mutation, where a group of cities have their order inverted, and scramble mutation, where a group of cities have their order randomly scrambled, are also effective.

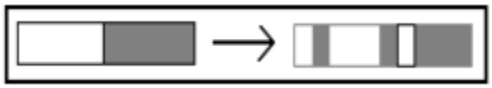

Figure 2: Diagram illustrating the basic concept of mutation. [4]

[4]

2.2.3 Initial Population

The population size for the sets of offspring is another crucial portion of a genetic algorithm as too large of a population could make the algorithm take longer than necessary to run whereas too small of a population

might not find an optimal enough solution. The optimum initial population size is between three and five times the number of cities. [4] The population itself can be generated randomly or it can be generated according to a simple heuristic such as a nearest neighbor algorithm. [9] Like population size the heuristic can also adversely affect the algorithm in that too specific of an algorithm may not generate enough genetic diversity leading to an initial population that is tunneled upon a local optimum. However it can also be greatly beneficial to have a heuristic in that it could produce more optimal starting point allowing the algorithm to converge faster on a final solution. [3]

2.2.4 Population Management

Population management isnt a separate stage of the genetic algorithm in the way that mutation and crossover are but it is still an important part nonetheless. There are two main problems worth keeping track of the first of which is duplicate checking. It doesnt make any sense to have duplicate tours in the population since they bring no diversity to the algorithm and having multiples of the same solution doesnt mean that solution is any better than any others. Therefore it can be quite beneficial to check new individual tours when they are created to ensure they are not a duplicate of a tour already in the population and if they are they can be discarded. [8] The other main population issue to look for is having multiple versions of the same solution simply with different starting or ending points. In the traveling salesman problem it doesnt matter where the tour begins if the cities are all still in the same order. The same can be said for reverse orders. Although these seem to add diversity to the population they do not and keeping them around is not beneficial and as such they can be discarded just like duplicates. [8]

2.3 Conclusion and Project Direction

Many people have done thorough experimentation on genetic algorithms when they are applied on traveling salesman problems and going off of their experiments I have an idea of what to expect from my experiments. It has been shown that despite genetic algorithms needing many iterations to find optimal solutions there are a number of ways they can be improved and provide optimal solutions to traveling salesman problems in reasonable time. [7]

3 Problem Approach

3.1 Necessity for an Effective Algorithm

There has been a lot of exploration into genetic algorithms as used to solve traveling salesman problems that being said my approach is to use a very simple genetic algorithm. The algorithm I'm using is a publicly available java implentation of a genetic algorithm for traveling salesman problems. [1] Although it is fairly easy for a person to glance at a map with cities and sketch out a fairly optimal route the only way to find the absolute best possible route is to run through all of the possible tour options. If there were only three total cities to the tour their would be 3 x 2 x 1 possible tours and it would be simple enough to check all 6 resulting routes and determine the optimal one. However in the case of a traveling salesman problem with twenty cities that is 20! possible tours or 2432902008176640000. There is no way a human could by hand run through all of the possible routes to determine the optimal route in a reasonable amount of time. Of course there are certain routes that could be entirely discounted and ignored but there are still many many routes to look through and it is essentially impossible to check all of the routes by hand. This is where the java based genetic algorithm comes into play.

3.2 Algorithm Details

In order for a genetic algorithm to properly generate solutions for a traveling salesman problem there are a couple of important distinctions to be made. Both of the distinctions revolve around the important tour criteria of visiting each city once and only once as in a genetic algorithm with things like mutation and

crossover it can be very easy to generate candidate tours that do not follow the constraints set forth. For starters there is a simple way to ensure that mutation does not break this constraint when it is performed. Mutation is important for the genetic algorithm as it helps increase genetic diversity which is a necessity for evolution to occur. Without genetic diversity the algorithm will not progress and will end up hung up on a solution near where the original population began. To guarantee that the mutation function of the algorithm does not break the requirement of visiting each city once and only once swap mutation can be employed. As swap mutation only uses pre-existing portions of the tour it cannot rid the tour of a city or add a duplicate city to the tour both of which would nullify the validity of any output solutions. In the java code the variable mutationRate controls the amount of mutation the tours undergo. The mutationRate is set to a number between 0 and 1 and compared against a randomly generated number between 0 and 1. If the random number generated is less than the mutation rate then the tour undergoes swap mutation, otherwise it is left alone. This essentially means that mutationRate corresponds to a percentage chance of mutation, for example if mutationRate = .5 then that would be a 50% chance of mutation. The other main

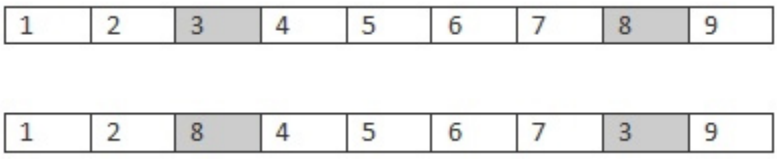

Figure 3: Diagram illustrating swap mutation. [1]

portion of a genetic algorithm that has the ability to break candidate solutions so that they're no longer valid is the crossover portion. In crossover a number of "parent" tours are mixed into "offspring" tours. Like mutation, crossover also provides genetic diversity to the candidate population and ensures that the genetic algorithm will continue to progress towards the best solution possible rather than stagnate at an early solution. However when mixing tours like crossover will need to do, it is necessary to preserve the requirement of all cities visited once and only once. In order for the candidate tours to retain their validity a specific crossover style referred to as ordered crossover can be employed. In ordered crossover a section of a candidate solution is chosen from one of the parents and then from there another parent solution is combined from start to finish with the chunk from the first parent. In this combining phase the order is preserved from the parent's respective sections and any city selections that would create duplicates are simply skipped. In the java code the number of the tours involved in the crossover portion is controlled by the tournamentSize variable. The last two portions of the genetic algorithm that could be tweaked were the variables popSize and

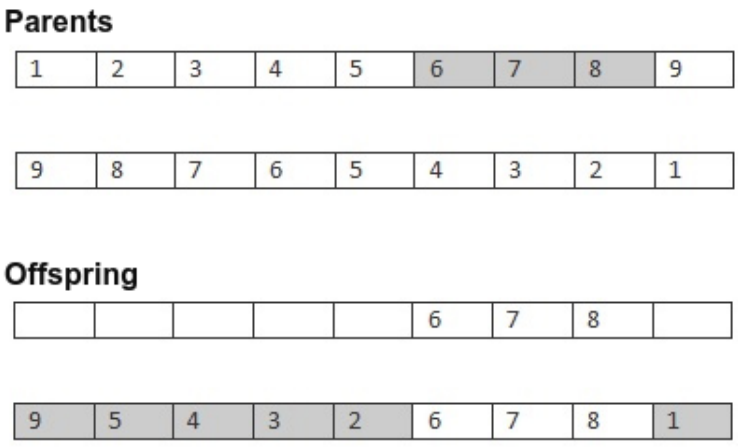

Figure 4: Diagram illustrating ordered crossover. [1]

generations. Both of these were simple integers and essentially controlled how many different solutions would be attempted. The popSize variable controlled the overall number of candidate tours so if this variable were set to 50 that would mean that there would be 50 candidate tours for each generation. Then the generations variable was there to control how many generations of candidate tours the algorithm would run through. So using the above example of popSize = 50 if the generations = 100 that would mean that the population of 50 candidate tours was run through the genetic algorithm 100 times. There was another variable in the code that was a boolean variable named elitism. This variable controlled whether the best solution would carry over from each generation or if each generation would essentially be a fresh start. Setting this to true kept the best solution from generation to generation whereas setting it to false would have each generation start anew.

4 Experiment Design and Results

4.1 Experimental Setup and Goal

The experiment had two primary goals, the first of which was to examine how altering the population size, number of generations, mutation rate, and tournament size affected the algorithm and the second objective was to adjust the above variables as needed to get the algorithm to run so that it would find the best possible solution at least 90% of the time. The first step of the experiment was to run a control group that I would be able to use to see how any alterations affected the algorithm. Anytime I gathered data I ran the algorithm 50 times straight recording the initial distance, final distance, average runtime, and the percentage of the time the best solution possible was found. For the control group the variables were set so that popSize = 500, generations = 1000, mutationRate = 0.015, tournamentSize = 5, and elitism = true. Once 50 trials were run with those values I ran the trials again with different values for popSize, generations, mutationRate, and tournamentSize making sure to only have one of the values at a time differ from the control group. By isolating individual variables and their effects on the algorithm when compared to the control group I was able to gain insight into how each variable affected runtime and optimality. After all of the variables were tested I then changed them as I saw fit so that I could get the genetic algorithm to find the best solution possible at least 90% of the time while keeping runtime as low as possible.

4.2 Experimental Results

For all 350+ trials of the genetic algorithm I kept the same 20 cities with the exact same coordinates. My primary focus for the experiment was to see both how individual variables would affect the outcome and also how much it take to get the algorithm to find the best solution 90% of the time. With this in mind it was best for me to keep the same 20 cities through all of the trials so that I could have as little variation as possible throughout. That being said there was still a bit of variation as by design genetic algorithms have an element of randomness to them. It was also in my best interest to keep elitism set to true for the code as with elitism set to false the algorithm doesn't converge as it doesn't preserve the best tour from the previous generation.

Table 1: This control group illustrates there is much room for improvement.

Control Group
Popsize = 500
Generations = 1000
MutationRate = 0.015
TournamentSize = 5
Best: 13/50 = 26%
Runtime: 1.8sec
Avg Final Distance: 898.06

Table 2: These trials show how each of the idividual variables affects the algorithms success.

Increased Population	**Increased Generations**	**Increased Mutation**	**Increased Tournament**
popSize = 1000	popSize = 500	popSize = 500	popSize = 500
generations = 1000	generations = 2000	generations = 1000	generations = 1000
mutationRate = 0.015	mutationRate = 0.015	mutationRate = 0.03	mutationRate = 0.015
tournamentSize = 5	tournamentSize = 5	tournamentSize = 5	tournamentSize = 10
Best: 27/50 = 54%	Best: 20/50 = 40%	Best: 24/50 = 48%	Best: 12/50 = 24%
Runtime: 2.7sec	Runtime: 2.6sec	Runtime: 1.9sec	Runtime: 2.1sec
Avg Final Distance: 879.28	Avg Final Distance: 891.38	Avg Final Distance: 880.06	Avg Final Distance: 906.58

Table 3: These trials showcase how much effort it takes the algorithm to reach the best tour more reliably.

1st Attempt at 90%	**2nd Attempt at 90%**	**3rd Attempt at 90%**	**4th Attempt at 90%**
popSize = 1000	popSize = 1000	popSize = 1000	popSize = 1000
generations = 2000	generations = 3000	generations = 5000	generations = 3000
mutationRate = 0.03	mutationRate = 0.03	mutationRate = 0.03	mutationRate = 0.035
tournamentSize = 5	tournamentSize = 5	tournamentSize = 5	tournamentSize = 5
Best: 34/50 = 68%	Best: 32/50 = 64%	Best: 32/50 = 64%	Best: 32/50 = 64%
Runtime: 4.5sec	Runtime: 6.3sec	Runtime: 10.1sec	Runtime: 6.4sec
Avg Final Distance: 873.98	Avg Final Distance: 873.82	Avg Final Distance: 875.12	Avg Final Distance: 874.54

Table 4: These last few trials showcase how dramatic the increase in runtime is just to get to the 90% mark.

5th Attempt at 90%	**6th Attempt at 90%**	**7th Attempt at 90%**
popSize = 1500	popSize = 1500	popSize = 1500
generations = 3000	generations = 4000	generations = 7000
mutationRate = 0.035	mutationRate = 0.035	mutationRate = 0.035
tournamentSize = 5	tournamentSize = 5	tournamentSize = 5
Best: 41/50 = 82%	Best: 42/50 = 84%	Best: 45/50 = 90%
Runtime: 9.1sec	Runtime: 12.1sec	Runtime: 20sec
Avg Final Distance: 867.92	Avg Final Distance: 866.24	Avg Final Distance: 864.98

4.3 Result Analysis

The results obtained from the experiment were very interesting to me. When it came to adjusting individual variables increasing population size, number of generations, and mutation rate all increased the effectiveness of the genetic algorithm allowing it to converge faster. However increasing tournament size had the opposite effect and actually harmed the algorithm's convergence ability leading to less overall reliability. It makes sense that the increased population size and number of generations would help the algorithm become more reliable as they both add more iteration to the algorithm which the runtime reflects.

4.3.1 Population Size vs. Total Number of Generations

The first variable I changed after completing the control group of trials was population size. I knew that increasing the population size would improve the algorithm's ability to find the best solution but I did not know it would be as effective as it was. When it came to sacrificing runtime for more reliable results I figured that increasing the number of generations would be the best course of action but that proved to be wrong. In comparing the first two trials despite having nearly identical runtimes, 2.7 seconds for increased population and 2.6 seconds for increased generations, the increased population size found the best solution 14% more of the time. Alongside this it also had an average final distance of 879.28 as opposed to the average final distance of 891.38 for the increased number of generations. It makes sense however as the increased

population size allows for more genetic diversity whereas the increased number of generations is mostly just more iterations in search of a better value.

4.3.2 Mutation Rate

I thought the most interesting factor was how effective mutation rate was for increasing the effectiveness of the algorithm. In my 5th attempt at 90% with an increased mutation rate I was able to increase the amount of time the algorithm found the best solution to 82% of the time with a runtime of 9.1 seconds. Comparing that to my 3rd attempt where I had a runtime of 10.1 seconds but only found the best solution 64% of the time shows just how effective tweaking the mutation rate can be. However when I attempted to increase the mutation rate further above 3.5% the algorithm became less effective as it was mutating to a point where it was harming the convergence of the algorithm.

4.3.3 Tournament Size

As the results in Table 2 illustrate increasing the tournament size actually harmed the algorithm overall. The tournament size corresponds to how many different tours are compared against each other and combined during the crossover portion of the genetic algorithm. During my testing it seemed that changing the tournament size to below 5 harmed the algorithm as well as changing it to anything above 5. That being said I left it at 5 for the rest of the trials to keep the algorithm in its best shape for aiming for the 90% goal.

4.3.4 Getting to 90%

To be honest getting to the algorithm to find the best solution 90% of the time took more work than I had anticipated. The early gains in consistency were quite easy, going from the algorithm acheiving the best solution 26% of the time to 54% of the time only increased the runtime by approximately 1 second. To increase the effectiveness by 28% and it only needed 1 more second of runtime was great and looked very promising, however making the jump from 54% to 82% required another 6.4 seconds. With everything being adjusted how the previous trials showed would be optimal it ended up taking a popSize of 1500, generations of 7000, mutation rate of 3.5%, and a total runtime of around 20 seconds to get the algorithm to where I wanted it to be. When I initially set out to test the individual variables and saw the control groups average runtime of a little under 2 seconds I was hopefully that I could keep the runtime down and still acheive the 90% I wanted to acheive. However as the data clearly illustrated to me getting those last few percent took a lot more iteration and runtime from the algorithm.

5 Conclusion and Future Experimentation

When it was all complete I was impressed by just how well a very simple genetic algorithm such as this could be applied to solve a traveling salesman problem. The algorithm simplicity was also a blessing in that it made it very straightforward to tweak and adjust as needed for the specific problem I was testing. Even when the algorithm was only finding the best solution less than half the time its average solution length was still very close to the optimal solution and in reality would be good enough for most scenarios. It did take a surprising amount of effort to tune the algorithm into finding the best solution more often than not and in doing so it also dramatically increased the runtime as well. In the future it would be very beneficial to flesh out the algorithm a bit more and add some heuristics to the crossover, mutation, and initial population portions of the algorithm. Adding some sort of system to favor genes from better fitness scoring tours during the crossover and mutation phases of the algorithm could greatly increase its effectiveness although one would have to be careful not to completely rid the algorithm of too much genetic diversity. As far as improving the initial population with a heuristic goes even something as simple as a nearest neighbor hueristic for the initial population would improve the algorithm as well and allow it to have a higher quality gene selection from the get go.

References

[1] Applying a genetic algorithm to the traveling salesman problem. http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5. Accessed: 2016-04-07.

[2] H. Braun. On solving travelling salesman problems by genetic algorithms. In *Parallel problem solving from nature*, pages 129–133. Springer, 1990.

[3] S. Chatterjee, C. Carrera, and L. A. Lynch. Genetic algorithms and traveling salesman problems. *European journal of operational research*, 93(3):490–510, 1996.

[4] N. Gambhava and G. Sanghani. Traveling salesman problem using genetic algorithm. 2003.

[5] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–168. Lawrence Erlbaum, New Jersey (160-168), 1985.

[6] P. Larrañaga, 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*, 13(2):129–170, 1999.

[7] B. Reisleben and P. Merz. A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 616–621. IEEE, 1996.

[8] L. V. Snyder and M. S. Daskin. A random-key genetic algorithm for the generalized traveling salesman problem. *European Journal of Operational Research*, 174(1):38 – 53, 2006.

[9] N. L. Ulder, E. H. Aarts, H.-J. Bandelt, P. J. van Laarhoven, and E. Pesch. Genetic local search algorithms for the traveling salesman problem. In *Parallel problem solving from nature*, pages 109–116. Springer, 1990.