MBA 614

Traveling Salesman Problem

A Genetic Algorithm Approach

Gavin Gee

3/30/2012

Executive summary

This project returns to the traveling salesman problem done in class during the semester. In class we solved the problem by calculating the next nearest city. This solution solves the same problem by using a different technic called a genetic algorithm. I first heard about genetic algorithms in a numeric optimization class I took while I was an undergraduate student. Genetic algorithms are excellent for solving problems that require too many computations to solve directly. As the Traveling Salesman Problem (TSP) has 36! potential solutions, a genetic algorithm is perfectly situated to solve it. Since the genetic algorithm by its nature is based on a random sample the end solution is not always the same. Although the solution has a range of outcomes between 2800 to 5000, over 86% of the outcomes are better than the next nearest city solution. To guarantee a solution that is better than the next nearest city I place the entire algorithm in a loop that is preformed ten times. Then the range of outcome drops to 2800 to 4200 between 500 and 1700 better than the solution we found in class.

Implementation Documentation

The genetic algorithm is a complicated process that essentially uses Darwin's theory of Natural Selection to identify an optimal solution to a given problem. The process considers each potential solution to be a genome in a population. By selecting the "best" genomes to become parents to the future generation of genomes we create a population of genomes that are constantly progressing toward an optimal end. In this case the genome is a route to visit all of the hotels and the solution is the optimal path one should take to minimize the distance traveled. The process is done in a series of steps, each of which will be highlighted below.

Initial Population Creation

The first step is to create a random population. The population is made up of 100 individual genomes. Each genome is randomly created. In this case, each genome contains a potential route of hotels to visit. To be specific the genome is a numeric array that contains 36 numbers between 0 and 35 with no duplicates. The first genome is created by generating a random number and placing it in an array, then another random number is generated and checked to see if it already exist in the array, if it does not then it is placed next in the array, if so another random number is generated. This process is repeated until the array is complete with all 36 hotels. In this initial population there are 100 different genomes.

Selecting Parents

To determine what genomes will go on to become parents for the next generation the genomes are grouped in groups of 5. To ensure that the grouping is random I first create an array of 100 random numbers between 0 and 99 (and making sure that each number is only used once). Then the first 5 numbers in the array are matched with the index of the initial population array and the genome correlated to that index number is selected. For each of the 5 genomes selected the total distance of the array is calculated. The 2 genomes with the two shortest distances are selected to be parents.

Creating the Next Generation

Once the parents are selected we create two separate children. The children are selected by combining elements of each parent and putting them into the children. Typically with a genome you take the first half of the genome from one parent and combing it with the second half of the genome of the second parent. This method clearly doesn't work with the TSP because you will get duplicate hotels and also missing other hotels in the children genomes.

To get around the problem of duplication a "greedy" selection method is chosen. For child1 I take the first hotel in the first parent (for this example we will call it 14) and make that the first hotel in the child. I then look to see what hotels connect to my first hotel in my second parent. I look at the distance to both of those hotels and then I choose the one that has the shortest distance. If the selected hotel is not already in child1 I place it next in the child1 array, if it is already in child1 then I use the other connecting hotel. If both connecting hotels are already in child1 then I choose a random number until I find one that has not been used in child1.

At this point an example may help clarify how the children are created. Below a simplified version of two parent genomes can be seen. For simplicity this example will only contain 10 hotels.

Parent1: [8 9 4 0 2 5 1 7 3 6]

Parent2: [3 9 5 0 7 6 2 1 8 4]

Child1 would be created by taking the first hotel in Parent1, 8, and adding it to its route. For the next hotel I take the connecting hotels to 8 in Parent2. The connecting hotels are 1 and 4. I check the distance to both hotels and then take the one that is the closet. For this example we will say that hotel 1 is closer than hotel 4 to hotel 8. In that case child1 would now look like this:

Child1: [8 1]

The third hotel is added by taking the connecting hotels to hotel 1 in parent 1, those are 5 and 7. I then check to relative distance from those hotels to hotel 1 and choose that hotel for child1. In this example we will say 7 was the closest. Now, child1 has the following hotels:

Child1: [8 1 7]

Next we determine what hotel is closer to hotel 7, either 0 or 6. If 0 has the shortest distance to 7 we are left with:

Child1: [8 1 7 0]

This pattern is repeated until we reach a situation where the closest hotel is already listed in Child1. If child1 has reached this point:

Child1: [8 1 7 0 4]

Then the only connecting hotel to hotel 4 in parent2 is hotel 8. Hotel 8 however, has already been used in Child1. In this case a random number between 0 and 10 will be selected. If the random number is not already contained in Child1 then it will be added to Child1. This pattern will continue until Child1 contains all 10 hotels.

In Child2 the only difference in the process is that the first hotel will be the first hotel from Parent2 or in this case hotel 3 so Child2 will start this way: Child2: [3 7 …]

The Mutation

Since our group initially had 5 genomes we will need to make sure that we have a fifth genome to pass along for future iterations. The fifth genome is created by mutating one of the parents (the first four genomes are Parent1, Parent2, Child1, and Child2). This mutation is done by selecting two random places in the genome and switching the genes located in each position. To illustrate if our parent1 looked like this:

Parent1: [8 9 4 0 2 5 1 7 3 6]

And our two random numbers were 3 and 7. We would take the hotels in place 3 and place 7 and swap them. The result would be this:

Mutation: [8 9 1 0 2 5 4 7 3 6]

As you can see above hotels 1 and 4 have switched places to create the new mutant genome. Now we have 5 genomes that are ready for the next iteration.

Iteration process

Once the each of the groups of 5 have come up with their new 5 genomes to pass on to the next generation we are ready to iterate the whole process of creating a new generation over again. I have experimented with doing between 100 and 10000 iterations. It seems that in most cases the process of improvement slows down after 150 iterations and it is fairly rare that the solution is improved with more than 400 iterations. As an example, here are the distances of the best routes on the first seven iterations of the process.

12698, 11490, 11380, 10759, 10264, 9219, 8937

As can be seen from this example it is not uncommon to see improvements of over 1000 miles in a single iteration during the first few dozen iterations. However, after 100 iterations it is rare to see improvements of more than 50 miles at a time.

Typically 300 iterations are run. The purpose of only doing 300 iterations is to give the ability of running the entire process multiple times. As mentioned earlier the random process of genetic algorithms does not guarantee the same result every time it is run. And although on one run you may not receive the best solution if the process is repeated 10 – 15 times the likely hood of finding a route less than 4000 miles is nearly 100%. Below is a histogram showing the results of 100 complete process with each one having 300 iterations.

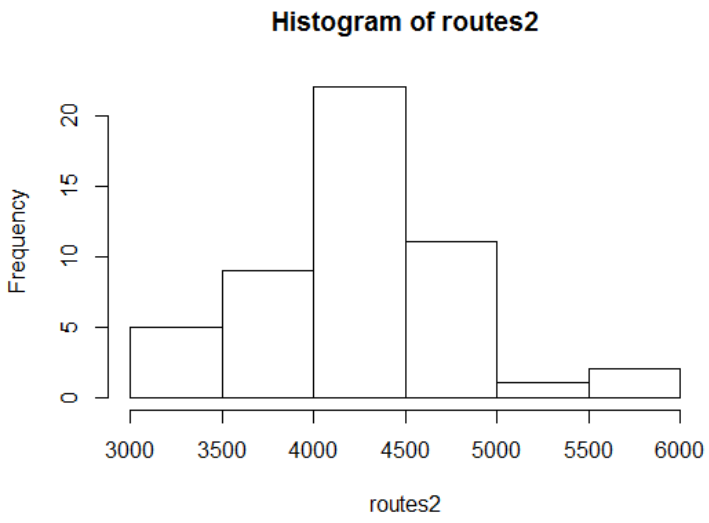

Figure 1: Histogram of Potential route distances

Figure 1 illustrates how the genetic algorithm will find a better solution than the next closest city in over 70% of its cycles. But by simply running the cycle multiple times a better solution is guaranteed. In the run illustrated above the best route ended up being 3179 miles, nearly 1500 miles better than the next closest city solution. If we run the program 35 more times using 400 iterations we can see that the distribution is not greatly improved.

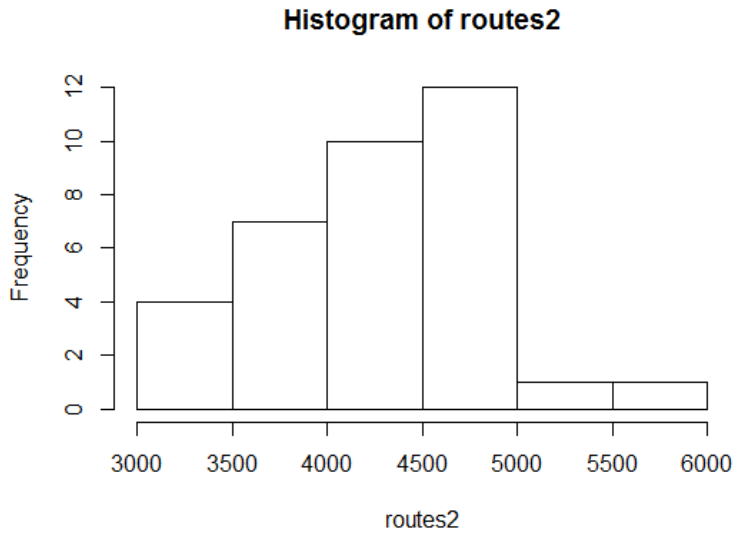

Figure 2: Distances after using 400 iterations

The iteration process can be time intensive. To run the process one time with 300 iterations will take approximately 3 minutes. If it is determined to run the entire process 10 times it will take approximately 10 minutes.

Discussion of learning and conceptual difficulties

There were three large hurdles for this project to work. First, determining how to use a genetic algorithm to solve the traveling salesman problem, second, was the actual coding of the project and finally, third, creating a solution that consistently beat the next closest city solution. The third element proved to be much more difficult than originally anticipated.

Understanding the genetic algorithm

There are dozens of explications on line of how to get the genetic algorithm to work on the traveling salesman problem, but I was surprised how difficult it still was to conceptually understand how to go about solving the problem. The first big hurdle was with finding a way to create children genomes without duplicating any hotels. Some people have created genomes that are binary strings with each value being associated with a potential solution. This is the most elegant solution to the problem, but it does not work once you have more than about 10 hotels. I found a professor who suggested the greedy selection, but even this was more complex than I had anticipated. It required extra if statements to make sure the hotel I was comparing it to was not the first or the last hotel in the parent and then handling the case when the closest city was actually already used proved to be one of the most complicated portions of genetic algorithm.

Complexity in coding

In the code there are over 500 lines and 9 sub procedures. As you might expect, when code becomes that long there can be a lot of confusion and potential for problems. Initially when I began coding the solution I felt that I would be able to use a lot of the code we created in class. For instance I felt the sub procedure named “beenThere” would be used multiple times throughout the code. But, I quickly found that passing the beenThere sub procedure different length arrays was creating some issues. So although I do feel that there are ways I could have manipulated the arrays so that I only needed one sub procedure, it was easier to create multiple sub procedures. As a result, there are 4 separate sub procedures that do essentially the exact same thing.

The code is lousy with if statements and loops. One of the difficulties of running code like this is tracking down where you are running into infinitely loops. There were some cases when the code was running in infinite loops, so I would try to debug it by running it step by step, the problem is that there are hundreds of thousands of steps in this code, so even if you have tracked down the problem to a single line it could still take you several hundreds more steps in different sub procedures and functions to track down the root of the problem. This took a lot of patients and required me to make sure I was only doing one step at a time to ensure everything was working before I moved on further.

Finally, one worry I had with the code that I ended up leaving in the end was the way I dealt with selecting random hotels once I found that the previously selected hotel was already found in the child. I did this the same way we did it in the next closest hotel solution, where a random hotel is selected and then checked to see if it is already contained in the current route. If it is contained the process just continues until an unvisited hotel is selected. This can be a problem when it is being used on 36 potential hotels, but it is even more of a problem when it is being used on 199 different genomes, as mine is when it is selecting which genomes to match together. My worry was that this would be a very

inefficient part of the algorithm, but although I am sure it does have times when it has to run over 300 times to find the final genome, it does seem to work okay.

Finding the best solution

The next closest hotel solution is a good solution, but clearly it is not the best. I initially thought that finding a better solution would be quite simple. In fact I played with the idea of not even having a mutation because I felt the solution would be simple to find, but as I quickly found getting a solution below 4000 miles is not so simple. The mutation ended up being a key aspect of finding sub 4000 mile solutions.

The purpose of the mutation stems from the difficulty of finding the global maximum in a multidimensional space. This can be visualized in a 3D space very easily. Below is a 3D plot of cos(y)* sin(x).

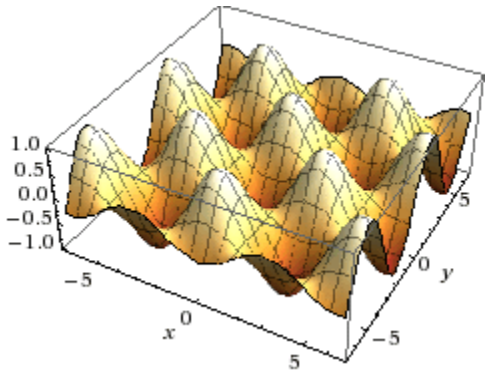

Figure 3: Example of multiple local maximum

One can see that there are many local maximum. In this case all of the maximum are equal, but what if each one of these little hills were a different height? What happens in the genetic algorithm is that in the initial population there are 200 different genomes scatter across the entire space with each genome potentially on a different hill. The genomes that are highest on their own particular hills become parents and over the first few iterations the new genomes jump from hill to hill. After the first 100 iterations or so though, all of the parents begin to converge on one hill. As both parents begin to mimic each other than all of the children become replicas of each parent and no progress is made. By the time this convergence happens typically you are on the peak of one of the hills. However, it is very possible that you are on top of one of the smallest hills in the entire solution space. If you reference figure 1 you can see that two solutions were near 6000 miles, this is an example of being on top of a very small hill.

The mutation makes it possible to “jump” from one hill to another after the genomes have converged on a single solution. This is often visible while watching the output of the best route on every iteration. It is not uncommon to see the following output: 4235, 4235, 4235, 4235, 4235, 4235, 4235, 4235, 4212, 4189, etc.

As you can see the solution had settled on a route that was 4235 miles, but eventually the mutation was able to swap two cities and “jump” to another hill that had a better maximum.

In my attempt to improve the solutions I tried to increase the number of hotels that were swapped in each mutation, but I found that switching multiple hotels decreased the likely hood that your mutation would be an improvement on your parent and as such it rarely became a parent in the next generation. After experimenting with several different types of mutation I found that the best solutions came from a simple swap of two hotels in the route.

Directions for Running Code

To run the code, first go to the "Create Route" tab.

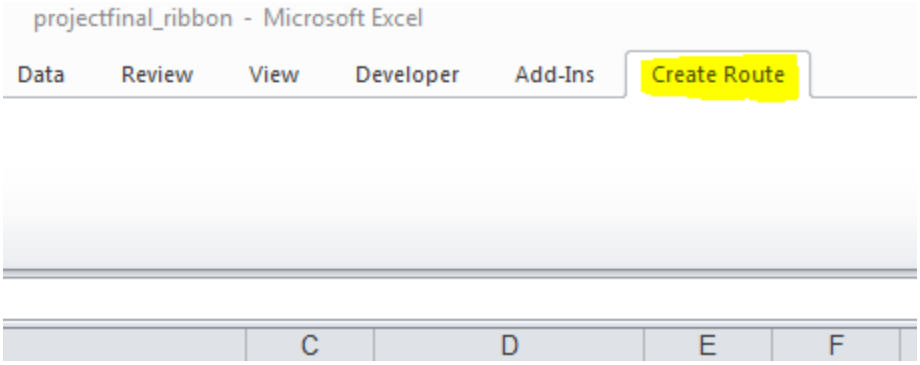

Figure 4: Screen Shot of Excel Sheet

On that tab there is one button. It is a picture of a map that says "Create Path". Clicking that button will cause the code to run.

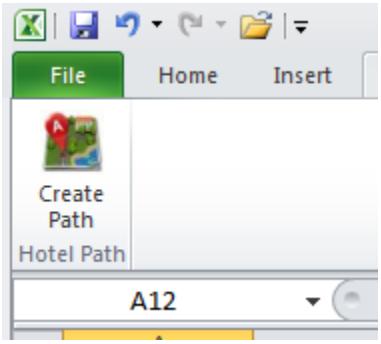

Figure 5: Create Route button that runs the code

Remember that the code will take approximately 8 minutes to run completely. Once the code is completely run you will have the distance of the path and a list of the order of hotels that the algorithm found to be the most efficient.

	A	B	C
1	**Total Milage**	**Suggested Route**	
2	2815	W Seattle	
3		Sheraton Sana'a Hotel	
4		Sheraton Seattle Hotel	
5		Four Points by Sheraton San Francisco Bay Bridge	
6		St. Regis Hotel, San Francisco	
7		W San Francisco	
8		Four Points by Sheraton Pleasanton	
9		Four Points by Sheraton Bakersfield	
10		The Westin Los Angeles Airport	
11		The Westin Long Beach	
12		St. Regis Resort, Monarch Beach	
13		Sheraton Suites San Diego at Symphony Hall	
14		The Westin Horton Plaza San Diego	
15		The Westin San Diego	
16		Sheraton La Jolla Hotel	
17		Sheraton Park Hotel at the Anaheim Resort	
18		Sheraton Anaheim Hotel	
19		Sheraton Cerritos Hotel	
20		Le Méridien at Beverly Hills	
21		Sheraton Los Angeles Downtown Hotel	
22		Sheraton Suites Fairplex	
23		Le Parker Méridien Palm Springs	

Figure 6: Sample Output displaying the suggested route

The image above is a screen shot of my best run. It is highly unlikely that you will get this same solution. As I mentioned in the paper this solution is based on survival of the fittest in a random population. In any random population it is unlikely that you will get the same result twice. What you will get is a solution that will be better than the next closest city solution we found in class.