Computer Systems Final Research Paper
Using Genetic Algorithms to Optimize Traveling Salesman Problems
2007-2008

Ryan Honig

June 9, 2008

1 Abstract

In this paper I will discuss a genetic approach to finding near-optimal soltuions to Traveling Salesman Problems. I will also discuss how using a heuristic to generate an initial pool affects the run time and solutions found by my program. Lastly, I will discuss the difficulties in creating a program to find near-optimal solutions to asymmetric traveling salesman problems.

2 Purpose and Background

The main purpose of my project is to develop my own genetic algorithm that can find near-optimal solutions for symmetric Traveling Salesman Problems. Once this was done I made it my goal to create a program to find near-optimal solutions to asymmetric Traveling Salesman Problems, but this goal turned out to be very lofty and hard to achieve.

This is a good problem to tackle because it is fairly complex and deals both with some complex algorithms and with some higher level math. By finding an efficient and optimal solution to the traveling salesman problem, it can be applied to the larger NP-complete field of optimization problems which can contribute to many fields of study. The TSP has been around for a long time, but more efficient programs for solving the TSPs are still being created. Many different algorithms have been used to attempt to solve TSPs, including heuristics, genetic algorithms, colony based simulations, and pure brute force programs. Heuristics are the best for finding 'good', but not optimal, paths fairly quickly, while genetic algorithms take longer but find more optimal paths. Brute force programs will of course always find the most optimal solution, but it might take a nearly endless amount of time to do so. The last general method, colony based simulations, are the most different of the four main solving types, and while I don't know as much about them as I do the other types, I know that they can be used to find very good solutions in a relatively short amount of time.

The paper: "New Genetic Local Search Operators for the Traveling Salesman Problem" by Bernd Freisleben and Peter Merz details how a good way to create an algorithm for the Traveling Salesman Problem is to use a basic heuristic to find the initial pool of paths and then use the genetic algorithm on this pool of paths to find a near-optimal solution. This is the structure that one of the versions of my program took. Another approach that is detailed by Marco Dorigo and Luca Maria Gambardella in "Ant Colonies for the Traveling Salesman Problem" is to use a simulated ant colony to solve a TSP data set. While this is not the most efficient way of solving a TSP, it can find very near-optimal solutions. In a colony based simulation such as this, the program will run through a pool of paths, and if part of a path is found to be effective, more parts of the colony will travel it and label it as an effective path. One of the most interesting articles that I found on the Traveling Salesman Problem is "Genetic Algorithms for the Traveling Salesman Problem: A Review of Representations and Operators". This article does a comparison of the different types of algorithms used to solve TSPs and their different way of representing the data. The biggest question that I would like to answer through my project is what combination of algorithms can create the most efficient and optimal traveling salesman program.

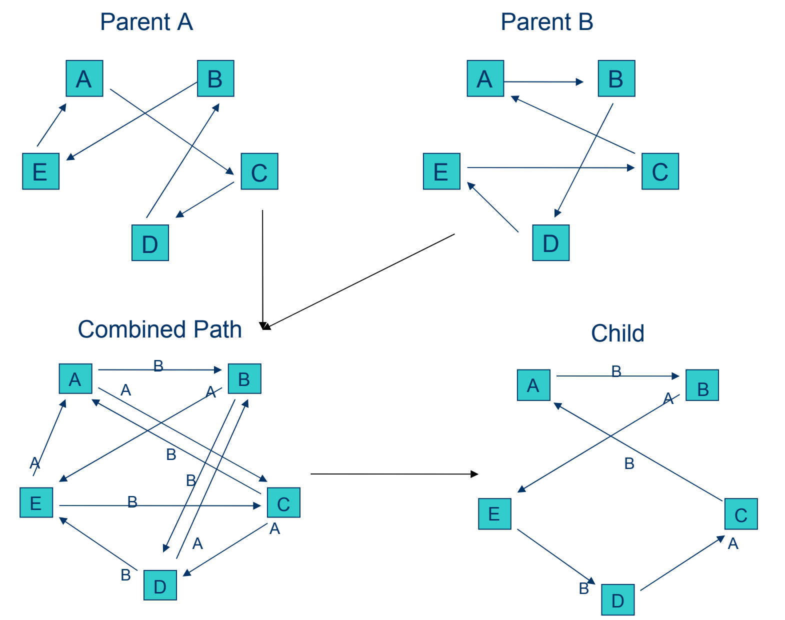
3 Development

3.1 Initial Algorithm

With my project, my goals was to develop an efficient algorithm that can find near-optimal solutions for both symmetric and asymmetric traveling salesman problems. My algorithm will be a mix of basic heuristics and the more complex genetic algorithms.

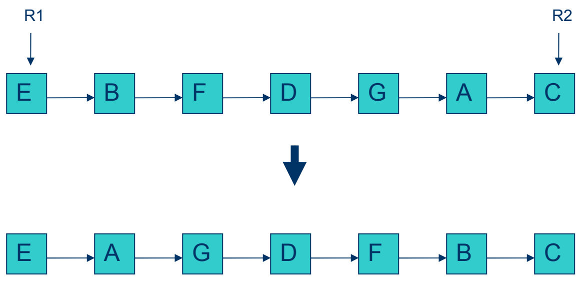
I began by creating a program that used a simple genetic algorithm that would reverse a section of a parent path and then replace it in the pool if it had a shorter path than the parent. I began testing this with data sets from the TSPLib website, which can be found here: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/. After finding that my solutions were off by multiple powers of ten, I discarded that algorithm and began a new one.

3.2 Genetic Algorithm



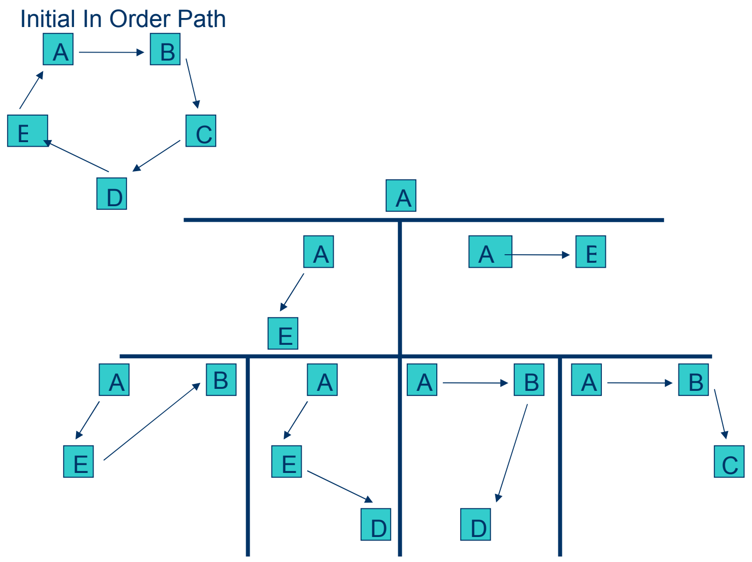
With my new program, I decided to take an approach that combined a few different genetic algorithm techniques that I read about during my research. This new algorithm starts by creating an initial pool of fifty random, legal paths. For each iteration of the genetic algorithm it will select two parent paths at random from the pool and use them to create a child path. In order to do this, all of the links between each point on both of the parent paths are then compiled into one set of links. The program then alternates choosing a link from each of the two parents to create the crossover. If the program gets to a point where none of the next links that it can choose from are legal, then a greedy algorithm takes over and completes the broken path by linking to the next closest, unused points.

3.3 Mutation Algorithm



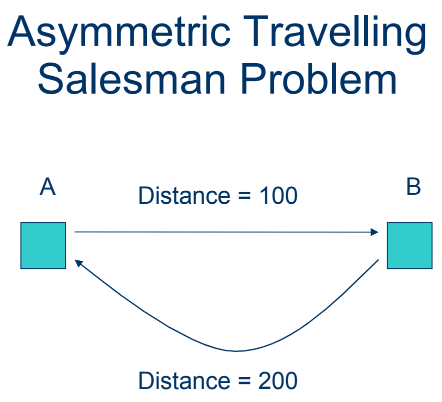
During second quarter I created a mutation method. This mutation method keeps the pool from being populated by the same path, since it has a chance of changing one of the paths in the pool. When the genetic algorithm runs it will continue filling the pool with the paths that it finds are better than are already in it, so if one path is found to be slightly better than all of the other paths, than the pool might begin to be filled with nothing but that one path, having a mutation method helps to fix this. My mutation method has a one in fifty chance of occuring. When a mutation does occur, two points are selected at random on the path, and then the path in between these two points is reversed. Once my mutation method was implemented, it significantly helped my program because it allowed the pool to continue running even if it got stuck on a single path that wasn't anywhere close to the optimal solution.

3.4 Pool Generating Heuristic



During second quarter, I also created a heuristic to generate the initial pool of paths. I created the heuristic, hoping that it would produce better results more efficiently by starting with a pool that isn't random. The heuristic I devised first picks a random point out of all of the points from the data set that the salesman must travel to. It then finds which two points are the closest to that point. Two paths are then created starting at the first point, and going to each of the other two new points. Then, for each of those two points, it finds the next two closest points, and creates two more new paths, thus doubling the number of paths being made. It continues doing this, doubling the number of paths until it only creates enough to fill a pool of fifty paths, thus filling the pool, at which point it will just continue using a greedy algorithm by picking the next closest point, until a full traverse of the points is acheived. I will discuss how this heuristic did in my results section.

3.5 Asymmetric Travelling Salesman Program



During third and fourth quarter, I spent much of my time working on converting my original random pool program so that it could read in and find near-optimal solutions to asymmetric travelling salesman problems. In an asymmetric traveling salesman problem, the distance between any pair of points is different whether it is going from A to B or B to A. This is used to simulate the real world, in which going between two different places would require you to take different routes. I am currently working with the data set BR17 which has 17 points. Although there are only 17 points in this data set, since an asymmetric data set contains a distance for the path there and back between every two pairs of points, the amount of data in this data set is actually 272 pairs of distances, which is closer to the order of n-squared.

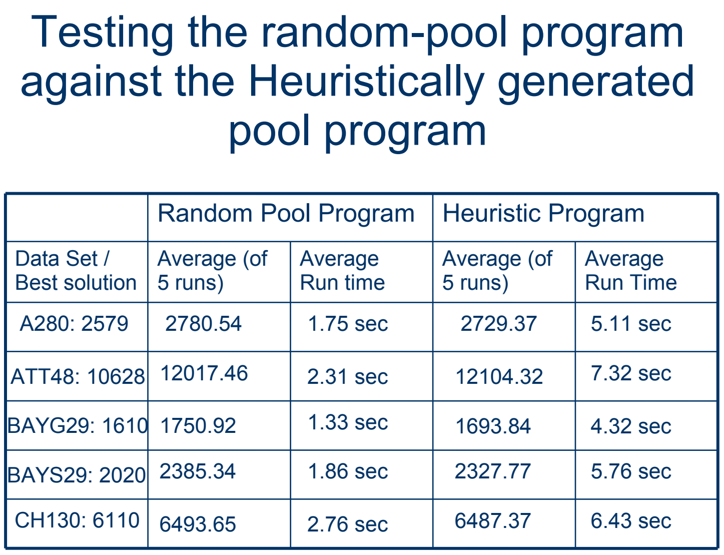
While the data files for symmetric traveling salesman problems are in the form of a list of coordinates, the data files for asymmetric problems consist of a grid of the distances between the pairs of points. This was the first thing to present major difficulties to my program. While I was initially able to read this grid into a matrix in the program, I wasn't totally sure how to go about performing the genetic algorithm on this grid. I eventually came up with the idea of creating two matrices of absolute coordinates for the points. One would have the coordinates of the points if you were going in a clockwise direction and one would have the coordinates of the points if you were going in a counterclockwise direction. While this seemed like it might work in theory, I still had a hard time getting my genetic algorithm to run on it. After weeks of attempting to change my algorithm to run off of these two sets of data, I found that I was nearing the end of fourth quarter, so I decided to halt work on this part of the project. If next year, someone decides that they want to continue work in this area, creating a genetic algorithm to find optimal solutions to asymmetric traveling salesman problems would be a great area to explore.

4 Results and Discussion

After testing my initial algorithm that reversed sections of the paths, I was not surprised to find that my solutions to data sets were multiple powers of ten off from the best known solutions. I knew that since my initial algorithm was based off of single parent genetics, it would not work very well.

I then created the genetic algorithm to find better solutions. When I first began testing this algorithm, my program would often fill its pool with copies of the same path, which would prevent it from finding a solution any better than that one. In order to correct this I implemented a mutation method to free up the pool. This worked and my program ran a lot better.

I then created my heuristic, hoping that it would produce better results by starting with a pool that isn't random, and possibly even be faster. When testing the heuristic program with the same data sets that I used to test the program with the randomly generated pool, I found that the solutions were only slightly better, but the program took mush longer to run.



Testing the random-pool program against the Heuristically generated pool program

	Random Pool Program		Heuristic Program	
Data Set / Best solution	Average (of 5 runs)	Average Run time	Average (of 5 runs)	Average Run Time
A280: 2579	2780.54	1.75 sec	2729.37	5.11 sec
ATT48: 10628	12017.46	2.31 sec	12104.32	7.32 sec
BAYG29: 1610	1750.92	1.33 sec	1693.84	4.32 sec
BAYS29: 2020	2385.34	1.86 sec	2327.77	5.76 sec
CH130: 6110	6493.65	2.76 sec	6487.37	6.43 sec

As you can see from my data, while the heuristically-generated pool program found slightly better solutions on most of the data sets, with the exception of data set ATT48, on every case it took more than twice as long to run than the randomly generated pool program did. In the end, I decided that the amount of improvement offered by my heuristic was not enough to justify the much longer run times that the program took.

5 Bibliography

—Dorigo, Marco and Gambardella, Luca Maria. ”Ant colonies for the Traveling Salesman Problem”. http://code.ulb.ac.be/dbfiles/DorGam1997bio.pdf

—Freisleben, Bernd and Merz, Peter. ”New Genetic Local Search Operators for the Traveling Salesman Problem”. http://www.rfai.li.univ-tours.fr/pagesperso/rousselle/do

—Larranaga, P., Kuijpers, C.M.H., Murga, R.H., Inza, I., and Dizdarevic, S. ”Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators”. http://wedhusprucul.tripod.com/skripsi/tsp.pdf

—University of Heidelberg Department of Computer Science. ”TSPLIB”. http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

—Voudouris, Christos. ”Guided Local Search and Its Application to the Traveling Salesman Problem”. http://www.cs.essex.ac.uk/CSP/papers/VouTsa-GlsTsP-Ejor98.pdf

—Yang, Cheng-Hong and Nygard, Kendall E. The effects of initial population in genetic search for time constrained traveling salesman problems. http://portal.acm.org/citation.cfm?id=170791.170875-coll=Portaldl=ACM-CFID=15521145-CFTOKEN=37709823

6 Appendices

6.1 An Overview of the Traveling Salesman Problem

The Traveling Salesman Problem is a problem in which a set of points is given and you want to find the shortest path that travels between each point once and then returns to the starting point. A symmetric problem is one in which the distance between towns A and B is the same as the distance between towns B and A. An Asymmetric problem is one in which the distance between towns A and B is different from the distance between towns B and A.

6.2 What is a Genetic Algorithm?

A Genetic Algorithm is a process for an algorithm that simulated genetics. First a pool of solutions is generated. Then for each generation of the program that is run, 2 of the solutions in the pool are chosen at random. These two solutions are then somehow combined to create a child solution. A fitness function is then used to determine whether the child solution is better than other solutions in the pool. If it is, then it will replace a solution in the pool. This process continues for many generations, until an optimal solution is found.