TJHSST Senior Research Project
Genetic Algorithms to Find Near Optimal Solutions to the Traveling Salesman Problem
2007-2008

Karl Leswing

June 12, 2008

Abstract

My main areas of interest within Computer Science are machine learning, and optimization algorithms.

Keywords: Genetic Algorithms, Ant Colony Optimization, Traveling Salesman Problem

1 Introduction - Problem Statement

1.1 What is research

My research has primarily been in optimization algorithms. These algorithms are used to find optimum or near optimum solutions to various problems. This project is specializing in genetic algorithms. Genetic algorithms use a biological approach to solving problems. They simulate a real life environment with solutions reproducing and mutating.

1.2 Why is research done?

There is constant research into using optimization algorithms to solve NP hard problems such as the Traveling Salesman Problem(TSP). Non-deterministic

Polynomial-time hard time problems come up often, and slightly more efficient solutions are very valuable. The TSP is often used in Computer Science and discreet mathematics because it is so easy to explain yet so difficult to solve. In this project I use genetic algorithms to find near optimal solutions to the TSP, and another real world problem.

2 Background

2.1 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a good problem because of its simplicity to explain yet its incredible difficulty to solve. The problem is outlined as thus: given a number of cities and the costs of traveling from any city to any other city, what is the cheapest round-trip route that visits each city exactly once and then returns to the starting city? The solution space is 1/2(n-1)! for all n greater than 2 where n is the number of cities. Using brute force methodologies become difficult after the tour length is greater than 40 cities. Using dynamic programming, the run-time can be cut down to $O(n^2 * 2^n)$.

This runtime is still very large. Therefor it has become necessary to use search heuristics to find near optimum solutions in reasonable amounts of time.

2.2 Genetic Algorithms

Genetic algorithms have been proven to be effective at optimization and pattern finding. They have been used in the past to create more efficient boat designs and fiber optic cable. Genetic Algorithms are inspired by evolutionary biology and incorporate such concepts as inheritance, mutation, selection, crossover, and reproduction.

Pseudo-code for generic GA

1. Choose initial population
2. Evaluate fitness for each individual in the population
3. Repeat

- Select the best individuals to reproduce
- Breed new generations using crossover and mutation
- Evaluate fitness of the offspring
- Replace worst ranked part of population with offspring

This generic problem solving methodology can be applied to almost any type of NP-Hard problem.

i

3 Procedure and Methodology

For ease of coding I have broken my coding into two major segments, creating a prototype and expanding on my prototype. I initially created a genetic algorithm with inefficient sub algorithms, such as tournament selection and single point crossover. After I finished my initial prototype I expanded into more efficient algorithms to find solutions to harder problems more quickly.

3.1 Implementation

Each Genetic Algorithm consists of essentially four major parts.

- Initialization
- Selection
- Reproduction
- Mutation

I use a cycle representation of a TSP solution. This means that each solution is an array of integers which refer to the a city in a path which is actually just two integers, an x and y coordinate. This makes all the rest of the algorithms fairly easy to code. When I initialize my gene pool I fill the arrays with random paths. This means that I start from total random solutions.

The main part of selection is the fitness algorithm. The fitness algorithm assesses a fitness level for each solution based on how good the solution is. A

better solution is supposed to have a higher fitness level. Because the main thing I am measuring is the distance of a path, higher distances are worse. I am trying to minimize distance. To rectify this I made my fitness algorithm the distance of the worst path in the gene pool minus the distance of the current path in the gene pool. This gave fairly good results.

After you have the fitness level there are still many ways to select which solutions get to reproduce. I coded three different ways, elitism selection roulette selection, and combination selection. In elitism selection I randomly select 10 different solutions and the solution with the highest fitness level among those 10 gets to reproduce. For roulette selection I select the solutions stochastically. Each point of fitness essentially gives you one chance to be selected in a theoretical lottery. This way the better solutions get picked proportionally to how they are better more often.

I implemented two types of crossover for my cycle representation. The two types of crossovers represented were single point and double point crossover. These types of crossovers help to conserve edges in the Traveling Salesman problem an were shown to be almost equally effective.

For mutation I did single double point mutation. For this I randomly switched the positions of two cities next to each other in the TSP.

Another problem of genetic algorithms is you are never sure when to stop them. Theoretically the best solution should continue to improve until you reach the global optimum or get stuck in a local optimum. However when should you decide that your solution is good enough? It is a thin line between time and results. There are two main trains of thought for ending a genetic algorithm. One is to stop when the best solution has not improved for a certain number of generations, and the other is to stop when the average global fitness level has not improved for a certain number of generations. I decided to stop my algorithm after nine generations in which the average fitness level does not improve.

3.2 Real World Example

I have decided to add a real world problem to my project. I am attempting to predict winners in college basketball using a genetic algorithm to weight

different statistics. To do this I wrote a series of Ruby Scripts in which to take data from a website, put it in csv format using regular expressions, then wrote a program to normalize the data and actually run the Genetic Algorithm.

As I stated before a Genetic Algorithm consists of mainly four different parts.

- Initialization
- Selection
- Reproduction
- Mutation

My Genetic Algorithm for college basketball currently defines weightings for 27 different statistics, multiplies each weight by its statistic then multiplies everything by the RPI. The RPI is a good indicator of a teams strength of schedule, this way the weaker teams can not get boosted statistics simply by having an easier schedule. My Genetic Algorithm currently has an array based implementation of a solution, a stochastic selection, single point mutation, and blending for reproduction.

A solution in my genetic algorithm is an array with the weights of each of the variables. This is a pretty basic implementation of a solution and is conducive for assessing a fitness, selecting solutions to reproduce, and actually reproducing the solution.

To assess fitness of each solution I use the weightings given in each solution, multiply them by that a specific teams statistics, sum the total and scale by the RPI. This gives each team a number for each solution. If a team has a higher number than a team they are playing that is equivalent to my program guessing that they have won the game between them. I do this for every game that has been played in the 2008 season, 5233 in total. For every game that the solution guesses correctly its fitness a point is added to its fitness level, and a point is taken away each time it guesses a game incorrectly.

When creating new solutions I use a single point mutation and a blend crossover. For my mutation, 5 percent of the time I add a random amount

between -1 and 1 to a random weighting in a solution. For my crossover I select a random number between 0 and 1.5, and then multiple one of the parents weight for an individual weight by the random number and multiply the other parent by another random number between 0 and 1.5.

Currently my genetic algorithm has proved to be around 70 percent accurate, and converges to the global optimum in around 7 hours. In the future I am thinking about adding linear combinations of statistics and using different crossovers to make it converge more quickly.

3.3 Extensions

Although I have come out with a lot of new information from this project this project can still be taken farther.

These include but are not limited to:

- Matrix Representation of Path
- Better Fitness Algorithms
- Intersection Detection
- Multi-Threading to Reduce Finding Local Optimum

Each of these would reduce the time it would take to find a local or globally optimum solution. I am sad that I did not have enough time to at least try a different representation of the TSP rather than cycle representation.

4 Testing and Analysis

After analyzing convergence time of the genetic algorithm using different selection and reproduction algorithms I was able to see which algorithms are more effective for the cycle representation of the TSP. I saw that roulette selection was far more effective in finding a solution than the elitism selection. Elitism selection reaches it maximum number of cities at around 15 while the roulette selection can find near optimal solutions to up to 70 city TSPs. Also the roulette selection follows a much more logistic graph of average fitness levels.

I found very little difference between the single and double point crossover algorithms. This is partly due to my implementation of the TSP. Since my TSP can start at any city in my representation, the fact that you can crossover from any point or the first point matters very little.

To ensure the accuracy of my solutions to the TSPs currently I am checking them by eye. One of the beauties of the TSP is that the solution is obvious when one is looking at it. However, I will hopefully get pre-solved TSPs or brute force smaller TSPs to test my optimization genetic algorithms. Also since the main focus of my project is comparing global search heuristics the algorithms will check each other to see who finds the best solution in the smallest amount of time.

Another way of checking my solutions is against the greedy algorithm. I found that the greedy solution quickly deteriorates for larger solutions. It also has a runtime of $O(N^2)$.

This runtime quickly grows to larger amounts of time than is reasonable for large TSPs. The greedy solution was good to compare against because it is a simple fairly effective algorithm.

5 Results

My genetic algorithm has succeeded in finding near optimal solutions to TSP's of 50-60 cities in under two minutes. I declare this a success. This is comparable to TSP specific algorithms, and better than algorithms such as the greedy algorithm. From these problems I was able to better understand how genetic algorithms work, how to apply them to various problems and which Genetic Algorithm specific algorithms work best. This was a successful project.

6 Conclusion

My venture into the study of genetic algorithms has proven to be fruitful. The genetic algorithms found near optimal solutions is reasonable time to two different types of NP-Hard problems in this study. For the Traveling Salesman Problem the genetic algorithms were as effective as other cutting edge TSP specific algorithms in finding reasonably optimal solutions. For college basketball there is really nothing to compare my results to, however

70 percent accuracy is better than most professional college basketball analyzers.

The hardest part of the genetic algorithm is, and will continue to be, finding the correct implementation of the problem to model it after a real life gene pool. It is difficult to model solutions as a data type which can merge and combine with itself. After the initial implementation of the problem actually coding the GA is trivial. Genetic algorithms have a future in finding near optimal solutions to NP-Hard problems.