

Genetic Algorithms to find Near Optimal Solutions to the Traveling Salesman Problem(TSP)

Karl Leswing Computer Systems Laboratory 2007-2008

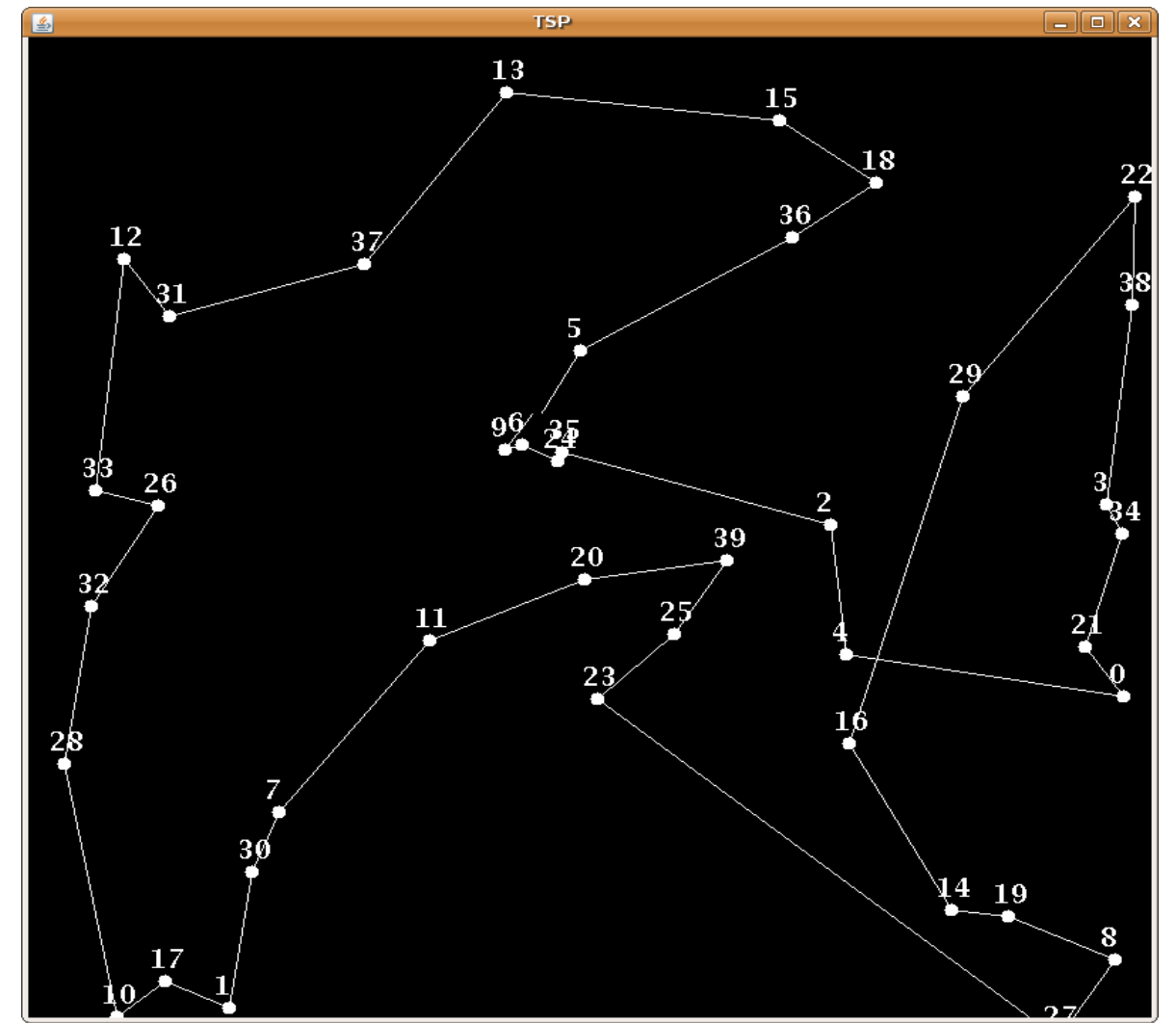
The Traveling Salesman Problem (TSP) is the classic nondeterministic polynomial-time hard(NP-hard) problem. The problem goes as such Given a number of cities and the costs of traveling from any city to any other city, what is the cheapest round-trip route that visits each city exactly once and then returns to the starting city? Although the problem is stated so simply and discreetly it is in fact a very difficult problem to solve. To simply use brute force to check all possible solutions it would require a $O(N!)$. This quickly becomes very difficult to do even for relatively small n. Therefore it becomes pertinent to find near optimal solutions through other methods using more realistic times.

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.

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. 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.



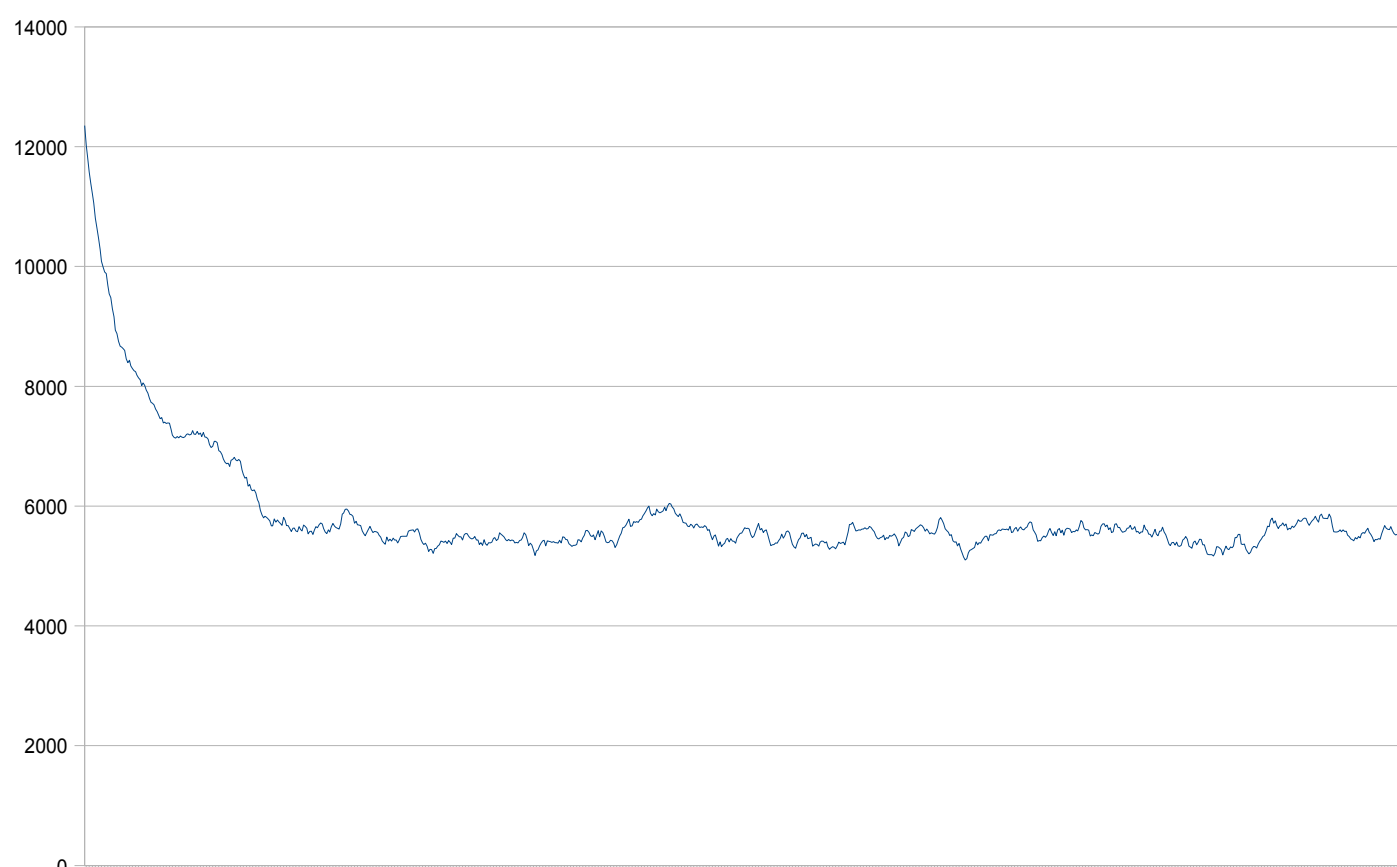
The main part of selection is the fitness algorithm. The fitness algorithm assesses a fitness level for each solution based on how good the solution is. A better solution is supposed to have a higher fitness level. Because the main thing I am measuring is the distance of a path, higher distances are 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.

Fitness Level Verses Time



A Real World Problem

My college basketball prediction uses a genetic algorithm to give different weightings to 27 different statistics including shooting percentage, offensive rebound percentage, and tempo. This Genetic Algorithm currently takes seven hours to converge. For the future I am hoping on using a double point mutation and a greedy crossover to make it converge more quickly. I also hope that adding linear combinations of statistics will make my GA converge on a solution that has higher than 70% accuracy.

