Travelling Salesman Problem - Genetic Algorithm

John Murray

Computer Science

Speed School of Engineering

jmmurr02@louisville.edu

1. Introduction

The Travelling Salesman Problem (TSP) is a problem in which the goal is the find the most efficient path between all of the nodes such that the path creates a Hamiltonian path. This report discusses using a genetic algorithm to try and find an optimal solution. In this approach, we will be creating a true Hamiltonian path.

2. Approach

The solution was written in Ruby with a graphical front-end written with Shoes. It is important to note that Ruby has no built-in graphic UI libraries or toolkits. Thus, third-party solutions must be used. The framework that I am using is called Shoes. It is a minimalistic framework for creating GUI applications in Ruby.

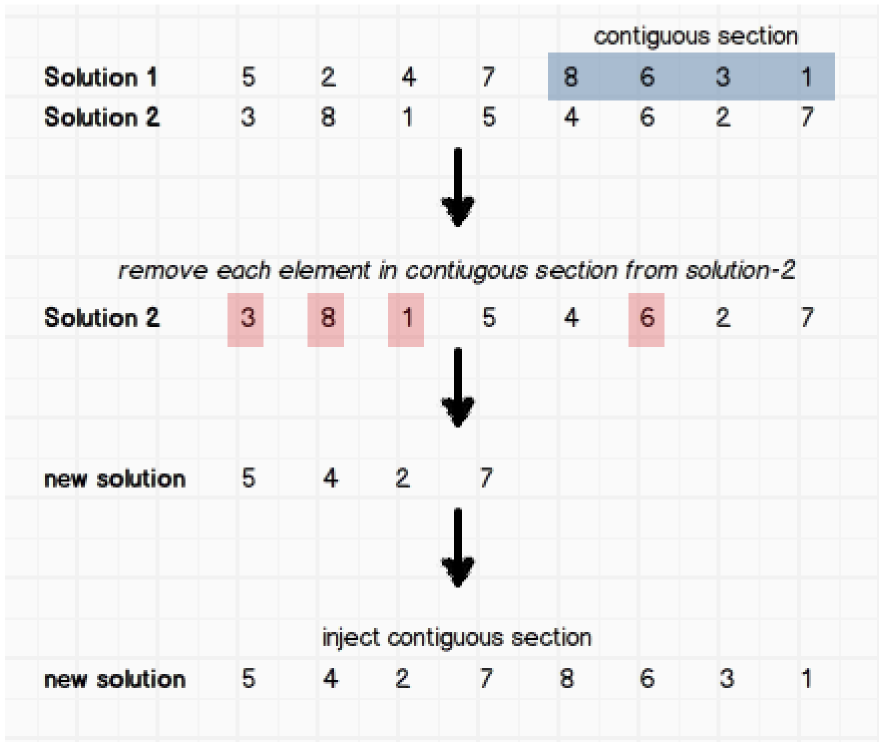
Beyond the GUI, the application follows the general format for genetic algorithms:

1. Generate a random set of solutions (100)
2. Rank them and pair the top 50 (25 pairs)
3. Cross-breed the pairs with each pair creating two new solutions
4. Introduce a mutation with probability α
5. Re-Rank the entire set and remove the bottom 50 (leaving 100 remaining)
6. Repeat steps 1-6 100 times in total (100 generations)

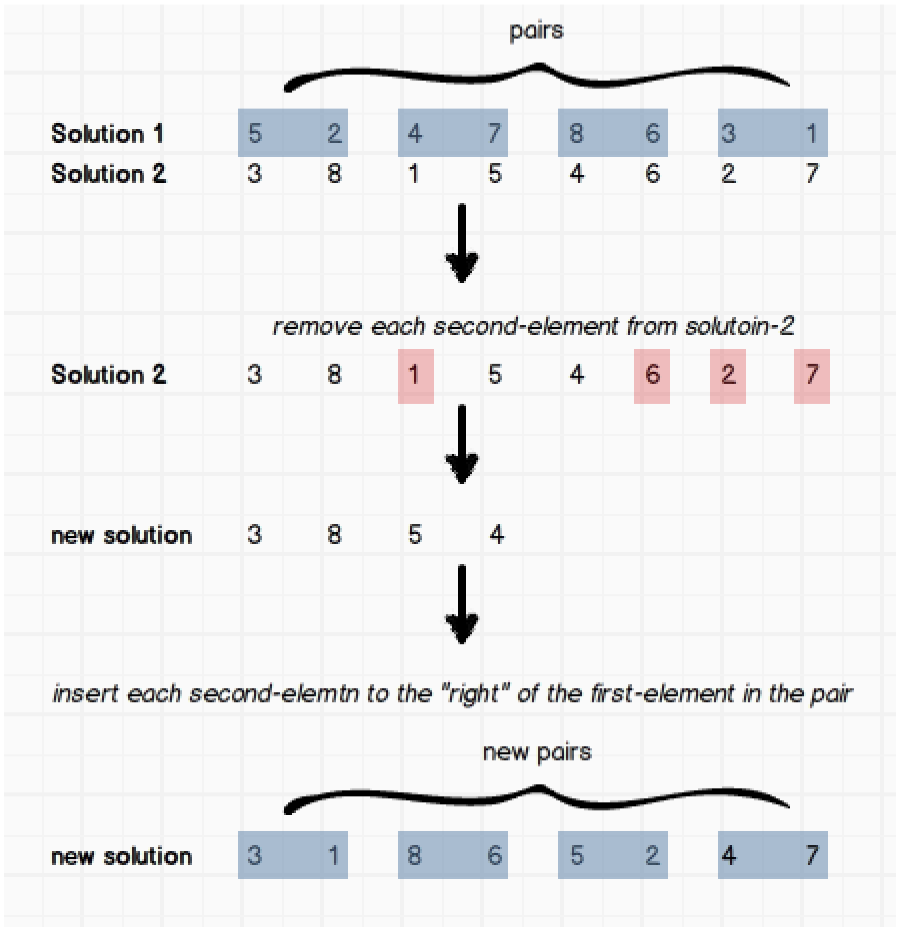
The algorithm was run with two different values of α(0.01 and 0.05) and two different cross-over algorithms. The cross-over algorithms were devised solely by me. However, given that the problem is a very common learning problem, I highly doubt that my algorithms are entirely unique.

2-a. Cross-Over Algorithm #1

For the first cross-over algorithms, I wanted to try and maintain a large, contiguous portion from one of the solutions. This means identifying one half of a solution and then removing all of those elements from the previous solutions path and then, lastly, inserting the solution somewhere in the path of the other solution. Graphically, it might make more sense:

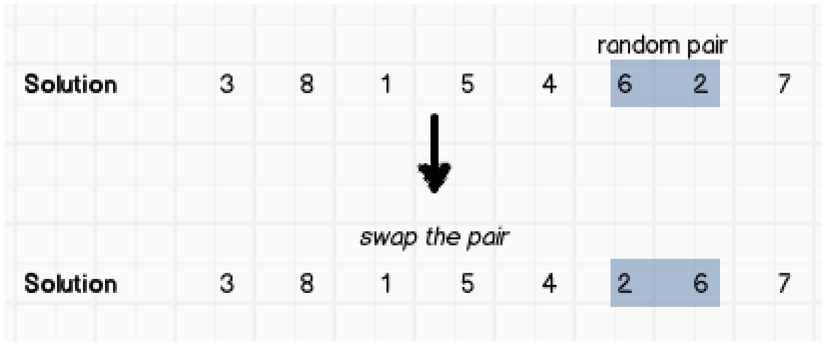

contiguous section
Solution 1 5 2 4 7 8 6 3 1
Solution 2 3 8 1 5 4 6 2 7
remove each element in contiugous section from solution-2
Solution 2 3 8 1 5 4 6 2 7
new solution 5 4 2 7
inject contiguous section
new solution 5 4 2 7 8 6 3 1

2-b. Cross-Over Algorithm #2
For the second cross-over algorithm, I wanted to try and maintain pairs with the assumption that a good solution has the most efficient path between two pairs. This involves identifying pairs in the first solution and then inserting them into the second solution. This is done by removing all second-items (from the pairs) from the second solution and then inserting them to the *right* of the first-item in the second solution. However, rather than trying to explain this further, I will once again illustrate it graphically as it will probably bring much more clarity:


pairs
Solution 1 5 2 4 7 8 6 3 1
Solution 2 3 8 1 5 4 6 2 7
remove each second-element from solutoin-2
Solution 2 3 8 1 5 4 6 2 7
new solution 3 8 5 4
insert each second-elemtn to the "right" of the first-element in the pair
new pairs
new solution 3 1 8 6 5 2 4 7

2-c. Mutation Strategy

The mutation strategy used was to find a random element pair within the solution and swap them. The pair would always be a contiguous pair and is illustrated graphically as such:



Fairly simple, but enough to (hopefully) break out of a local maximum.

3. The Results

Data

The problem was solved with a TSP file containing 100 data-points. However, the solution should work with any size data-file. However, do note that more data points will take both more CPU time and memory in the computer performing the computatoins.

Results
The program was run with two different cross-selection algorithms and two different mutation probabilities (α), yielding four sets of results. For each of these four sets, the program was run 100 times. Below is the average of each run.

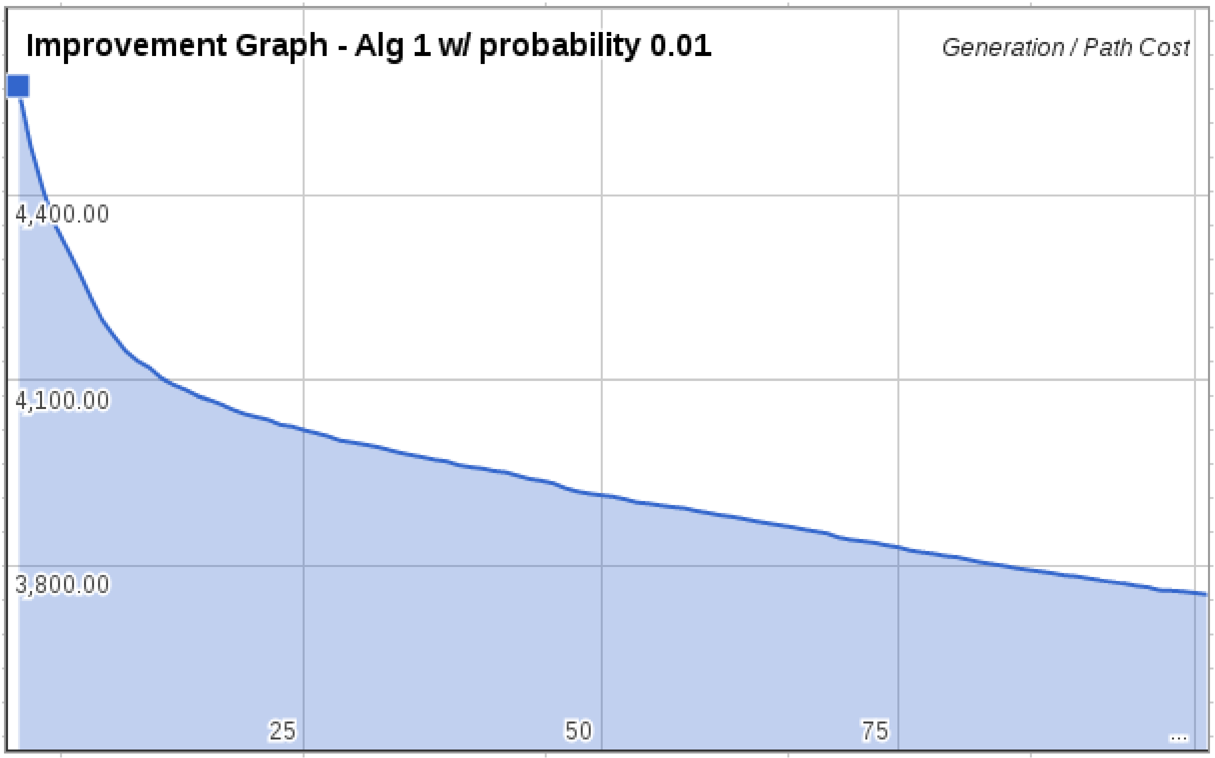
Run-Time (ms)

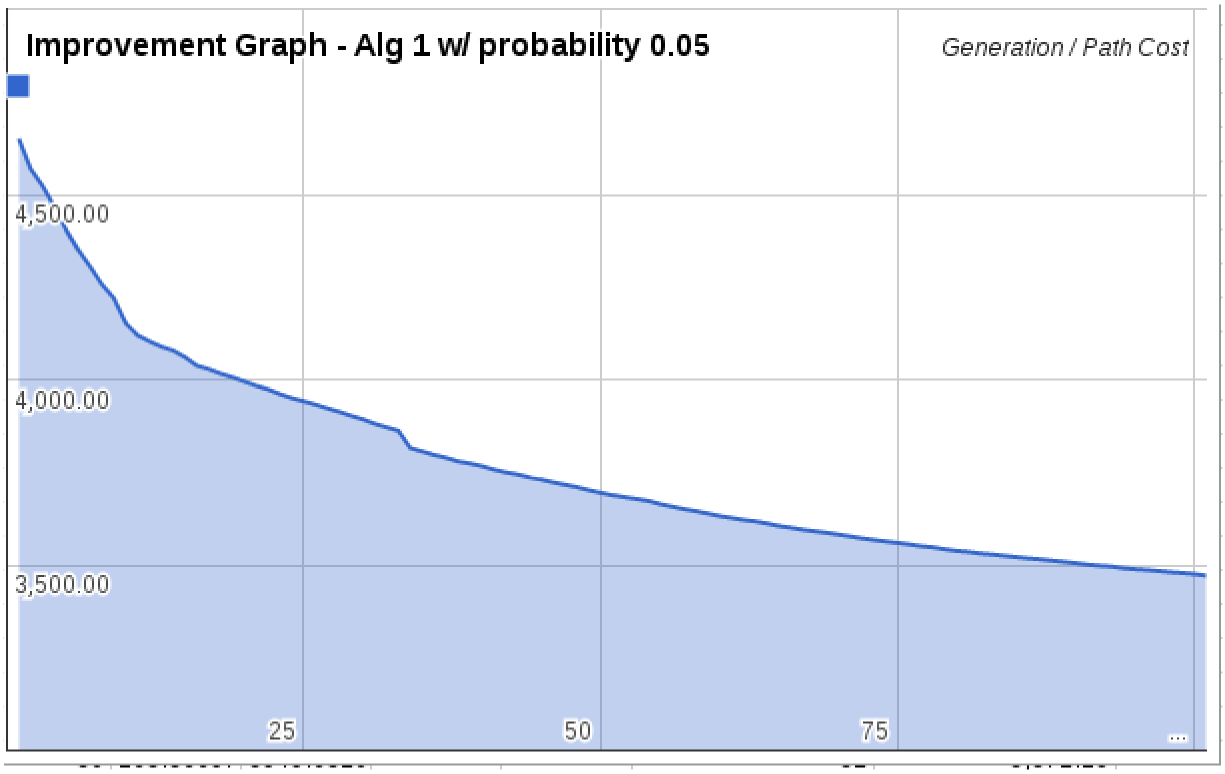
	Cross-Selection Alg. #1	**Cross-Selection Alg. #2**
$\alpha = 0.01$	16,912.02	17,158.26
$\alpha = 0.05$	21,712.60	21,623.09

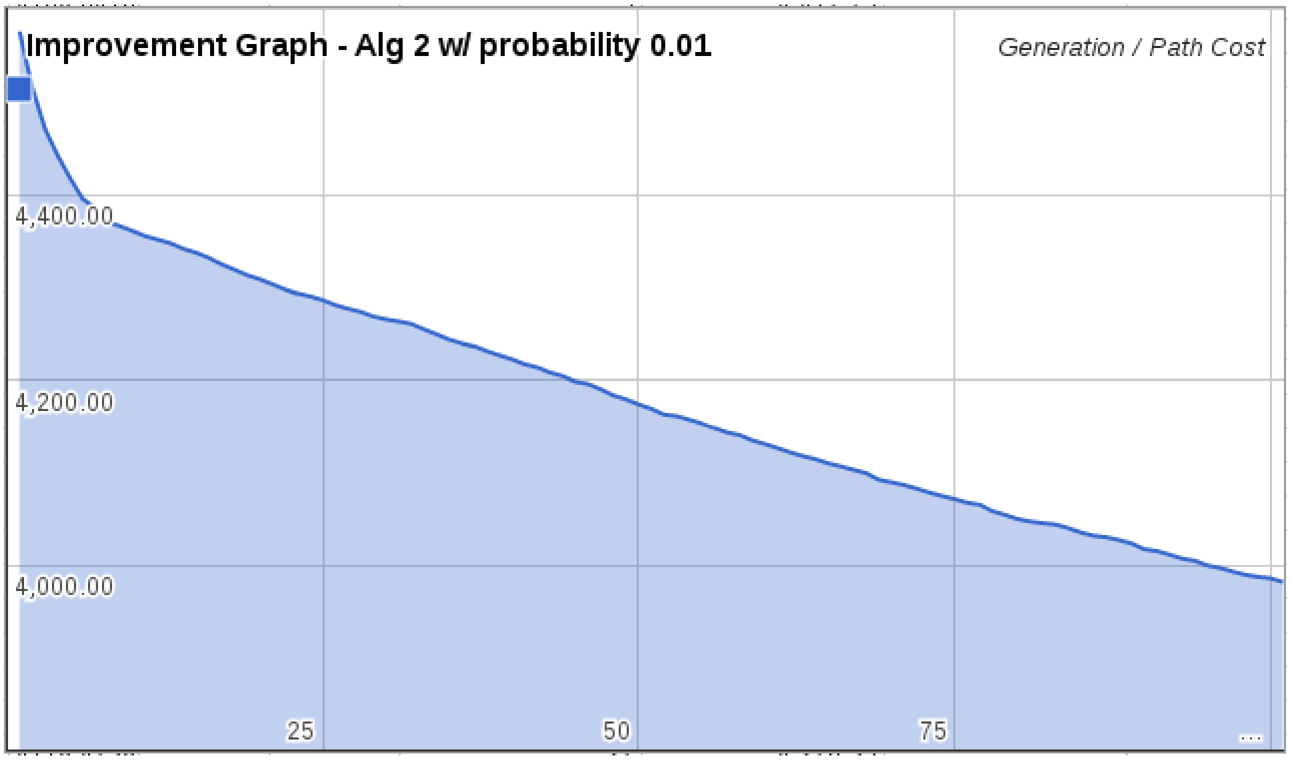
Final Solution Cost

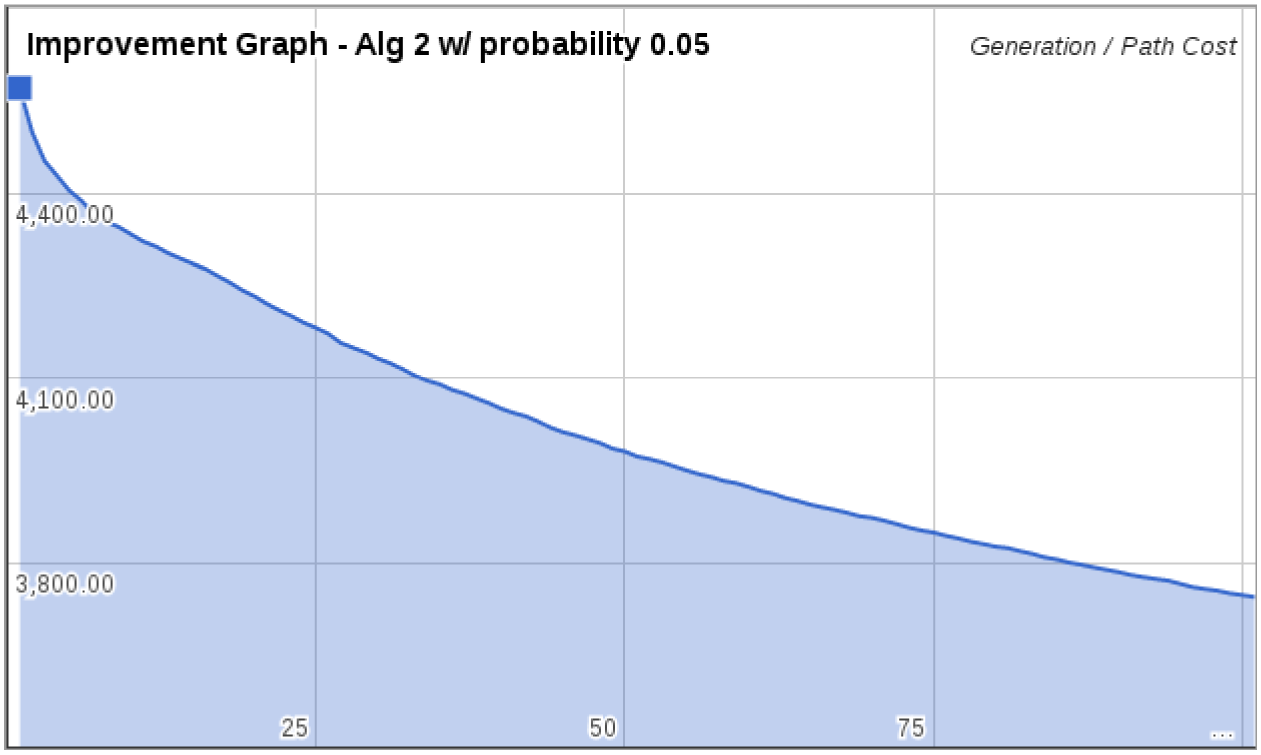
	Cross-Selection Alg. #1	**Cross-Selection Alg. #2**
$\alpha = 0.01$	3754.64	3985.68
$\alpha = 0.05$	3475.68	3747.21

Improvement Graphs
Below I have included an improvement graph for each data-set. Each graph indicates the data-set that is being represented.


Improvement Graph - Alg 1 w/ probability 0.01
Generation / Path Cost
4,400.00
4,100.00
3,800.00
25
50
75
...


Improvement Graph - Alg 1 w/ probability 0.05
Generation / Path Cost
4,500.00
4,000.00
3,500.00
25
50
75
...





Interpreting the Results
As you can see, both algorithms took about the same time to run. However, algorithm 1 with $\alpha = 0.05$ generated much better results. In general, algorithm 1 generates better

results that algorithm 2. We can attribute this to the particular cross-selection algorithm which seems to be a better algorithm than the other.

4. Critical Thinking

Problems Encountered

The most difficult part of this assignment to implement was the GUI. Although this should be a trivial part of the program, it was actually the most difficult given the environment. Ruby as very poor graphical capabilities and poor UI frameworks (outside of web development). The framework I used carries it’s own binaries of the Ruby runtime, meaning that not everything will work as expected. By breaking many standard language conventions and standard libraries, development can become sporadic at best. In particular, the *dynamic* view for the GUI was not achieved due to the fact that Fibers (Ruby co-routines) do not function within the UI framework (Shoes).

It could be advantageous to invest time in learning another, more standard, UI framework for the next project or Ruby GUI implementation. If this project were to be repeated, that is the only item that I would change. The other pieces of the assignment were straightforward and natural.

What Was Learned

Although I’ve read some about genetic algorithms and I understand their concept, I’ve never implemented one myself. This was a learning experience in that I now understand the real value is in the way that you breed various solutions together. If your breeding algorithm generates solutions that are too random, then you might not see too much improvement in your curve. Or, if your mutation algorithm does not perform enough change, then you still risk being stuck in a local maximum. These are details that I could not have learned without really getting *hands on* as in this project.

Impressions

Overall, I am impressed at how well the algorithm seemed to work given the size and complexity of the problem. The algorithm is one of the simplest solutions yet it worked well for a large data-set.

While it’s not guaranteed to generate the *most* optimal solution, I could see this type of approach being used in other areas (such as optimization problems perhaps?).