Charles Balconi
cbalconi@cs.rochester.edu
CSC242
Term Project
Due: 5/7/03

A Study of Methods Used to Solve NP Hard Optimization Problems

Overview:

Many optimization algorithms have been implemented to provide solutions to computationally hard problems; however, there are often tradeoffs between an algorithm's efficiency, effectiveness, and complexity. This paper explores some of the optimization techniques used to solve the Euclidean Traveling Salesman Problem (TSP), and compares the aforementioned qualities of classical algorithms and genetic algorithms with one another.

Background and Motivation

The traveling salesman problem is simple to describe: Given 'n' distinct points on a two-dimensional plane, find the shortest closed tour that visits each point exactly once. The only known exact solution to this problem, however, has exponential time complexity, for one must compute every permutation of potential orderings, which makes finding the exact solution O(n!). So, we have to settle for an approximate solution that is as close to the optimal as possible. This is where the myriad of approximation algorithms come into play. My experimental findings are from my implementation of two classical algorithms and three genetic algorithms.

Methods

I wrote my program exclusively in Java so that I could display graphical output easily. It is broken up into several classes some of which handle bookkeeping, and others that contain code for the approximation algorithms. The four classes: Graph, Matrix, Tree, Tour are the classes that handle the operation of the graph representation while classes: TwoOpt, Draw, Ga, Solve are classes that contain the algorithms. The class: 'makegraph' is just a utility class that creates a graph file of user specified size. To obtain a solution from one of the algorithms, the user must use either 'Draw' or 'Solve' with the appropriate command and then enter in the graph size and name of the graph file. The program will then either output both a graphical display of the running algorithm and numerical length of the shortest found tour (Draw.java) or will output just the numerical tour lengths along with a text file 'tour.txt' containing the points and edges the algorithm found. ('tour.txt' can be input to 'DisplayGraph.java').

Basically, in order to determine the efficacy of one of my approximation algorithms I ran each one several times on several different graph sizes and measured the runtime of each and the final tour length. Some of the algorithms are deterministic and arrive at the same solution every time, but most of the algorithms produce a different output each time, so I took the best of several runs on each graph size.

The graph simulation space is a 1000x1000 square grid, where graph points are (x,y) coordinates from 0-1000. Thus distance calculations are done using the distance formula for points on a 2-D plane.

Results

This section offers a brief description of how each algorithm works along with some data from runs on graphs of different sizes.

Results for classical algorithm 1: (accessed by '-dfstsp' in class 'Solve')

How it works:

This algorithm first computes the minimum spanning tree of the graph, which is an $O(n^2)$ operation. Then it takes the pre-order depth-first search traversal of the minimum spanning tree and returns it as the tour. This is a deterministic algorithm, and always arrives at the same solution due to the fact that the MST is always the same, regardless of where the algorithm begins its construction. It is the simplest, fastest approximation technique I implemented, but as a consequence it has the longest tour lengths. This algorithm does guarantee that the solutions it finds are within a factor of two of the optimal solution however, and since most of my other algorithms are at least as good as this one, I know that most of my solutions are reasonably close to the optimal.

Data:
Graph size 20:
Runtime: 50ms.
Tour Cost: 5061.4351

Graph size 50:
Runtime: 50ms.
Cost: 7816.0942

Graph size 75:
Runtime: 60ms.
Cost: 8941.6292

Graph size 100:
Runtime: 60ms.
Cost: 10528.6205

Graph size 200:
Runtime: 110ms.
Cost: 14280.1256

Graph size 500:
Runtime: 220ms.
Cost: 22069.0993

Classical Algorithm 2: (accessed by –2opt in ‘Draw’ or ‘Solve’)

How it works:

This algorithm is known as the 2-optimal algorithm. Basically it works by starting with a random initial tour and improves on that tour by inverting sub-tours that shorten the tour. I basically translated the description given by *Mertens* [3] into code. The algorithm searches every sub-tour of the graph in its current state and takes the sub-tour whose inversion decreases the tour length the most. It does this until no such sub-tour can be found that decreases the tour length. When this point is reached, the tour is said to be 2-optimal. It is a relatively simple, yet highly effective method for approximating a good TSP solution.

Data:
Best of 5 runs.

Graph size 20:
Runtime: 50ms.

Tour Cost: 3776.3682

Graph size 50:
Runtime: 160ms.
Cost: 5888.6747

Graph size 75:
Runtime: 220ms.
Cost: 6916.8267

Graph size 100:
Runtime: 440ms.
Cost: 8189.4132

Graph size 200:
Runtime: 3630ms.
Cost: 11248.5274

Graph size 500:
Runtime: 69370ms.
Cost: 18262.6542

Genetic Algorithm 1: (accessed by –rtsp in 'Draw' and 'Solve')

This algorithm is a very simple kind of genetic algorithm, and might be better referred to as an evolutionary algorithm. Like in the 2opt approach, a random tour is created initially, but instead of searching for the best sub-tour to invert, the algorithm simply mutates several different sub-tours at random and takes the one that reduces the tour length the most. It is a purely elitist approach since the next generation's parent is merely the best of the mutant offspring of the previous parent. Eventually the algorithm gets itself stuck in a local minima it can't get out of, but my testing has shown that this doesn't happen until it's found a pretty good solution. It is worth noting that the long runtimes are due primarily to the termination condition. Since the algorithm doesn't really know when to stop I introduced a termination condition where after 200000 iterations of no change, the process halts. For practical purposes this can stand to be a bit lower, but it is necessary for large graphs. The best way to run this algorithm is to visually confirm its progress using the graphical output in 'Draw' since the algorithm will run until the user closes the display window.

Data:
Best of 5 runs.

Graph size 20:
Runtime: 770ms.
Tour Cost: 3776.3682

Graph size 50:
Runtime: 1810ms.
Cost: 5858.9583

Graph size 75:
Runtime: 2690ms.
Cost: 7268.8169

Graph size 100:

Runtime: 4290ms.
Cost: 8437.8524

Graph size 200:
Runtime: 13620ms.
Cost: 11743.5401

Graph size 500:
Runtime: 102930ms.
Cost: 18223.8199

Genetic Algorithm 2: (accessed by –swap in 'Draw' only)

This algorithm implements a different mutation strategy. Instead of inverting sub-tours, which has the effect of swapping edge pairs that reduce the tour length, this algorithm simply swaps two nodes' ordering in the tour sequence. This is done a number of times, and the swap that reduces the tour the most becomes the new parent. Again, it is purely elitist, and this swapping strategy is much worse than the inversion mutation scheme. The quality of the tours created with this approach are worse than even the MST-DFS algorithm even for small graphs. I consider it an example of how NOT to mutate offspring for the next generation TSP solutions.

Data:
Runtimes are approximated, since they must be attained manually using the graphical display.

Graph size 20:
Runtime: 70ms.
Tour Cost: 4149.5467

Graph size 50:
Runtime: 150ms.
Cost: 7877.1795

Graph size 75:
Runtime: 220ms.
Cost: 10579.8520

Graph size 100:
Runtime: 350ms.
Cost: 13461.3144

Graph size 200:
Runtime: 22000ms.
Cost: 23047.2067

Graph size 500:
Runtime: approx. 5min.
Cost: 47119.2030

Genetic Algorithm 3: (accessed by –shuffle in 'Draw' and 'Solve')

This algorithm is difficult to categorize, however it's closer to a genetic algorithm than a classical algorithm. It incorporates two strategies in its search. It begins with a random tour and then computes the initial 2-optimal tour of it. Then, a random sub-tour is selected and mutated into a random, unordered sub-tour. Then the 2-opt algorithm is run again until a new 2-optimal tour is created. This process continues until the user halts the process or the optimal tour is found. The mutation process is similar to

simulated annealing, in that it allows 'bad' solutions to be introduced in order to help break the algorithm out of a local minimum. Once the initial 2opt algorithm completes, the 'working' tour is constantly mutating, and only updates the graphical display when the new tour is better than the current minimum. So, in this sense, the graphical display does not represent a 'live' search, but rather the best results of several (possibly hundreds) of mutations. For large graphs (size > 100) it is best to use the graphical display in 'Draw' to determine when to terminate, because the termination condition in 'Solve' is rather arbitrary since the algorithm tends to find increasingly better solutions as it runs. It does, however seem to find the optimal tour for graphs < 100 fairly quickly however.

Data:
For graph sizes up to 100, the tour lengths returned are likely optimal, for the 200 and 500 node graphs, I terminated the search when it stopped finding improved tours at a regular rate. Practically speaking, this algorithm isn't particularly useful for graphs larger than 200 nodes. However it is outstanding for graphs less than 100 nodes.

Graph size 20:
Runtime: 320ms.
Tour Cost: 3776.3682

Graph size 50:
Runtime: 4290ms.
Cost: 5738.9187

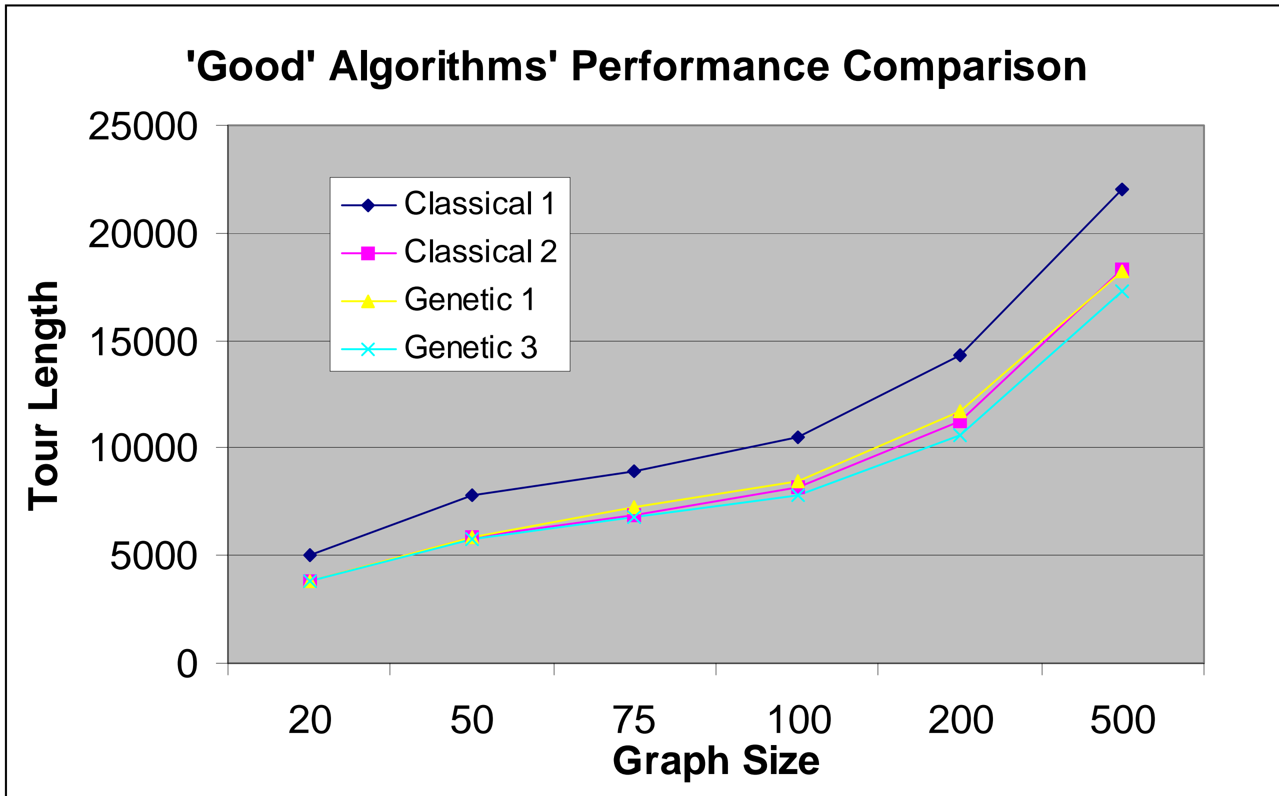
Graph size 75:
Runtime: 10270ms.
Cost: 6817.0592

Graph size 100:
Runtime: 14780ms.
Cost: 7777.9700

Graph size 200:
Runtime: 212120ms.
Cost: 10573.2596

Graph size 500:
Runtime: 1700770ms.
Cost: 17278.3858

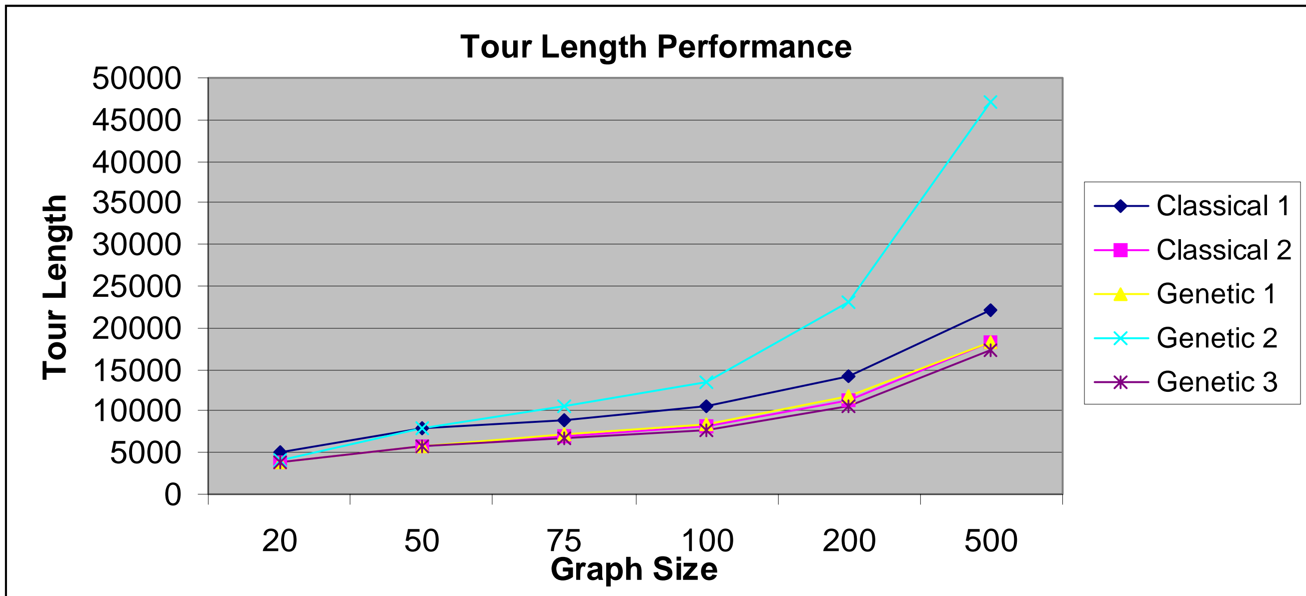
Here's a graph of the tour lengths for the 4 best algorithms. Their solutions are fairly competitive.

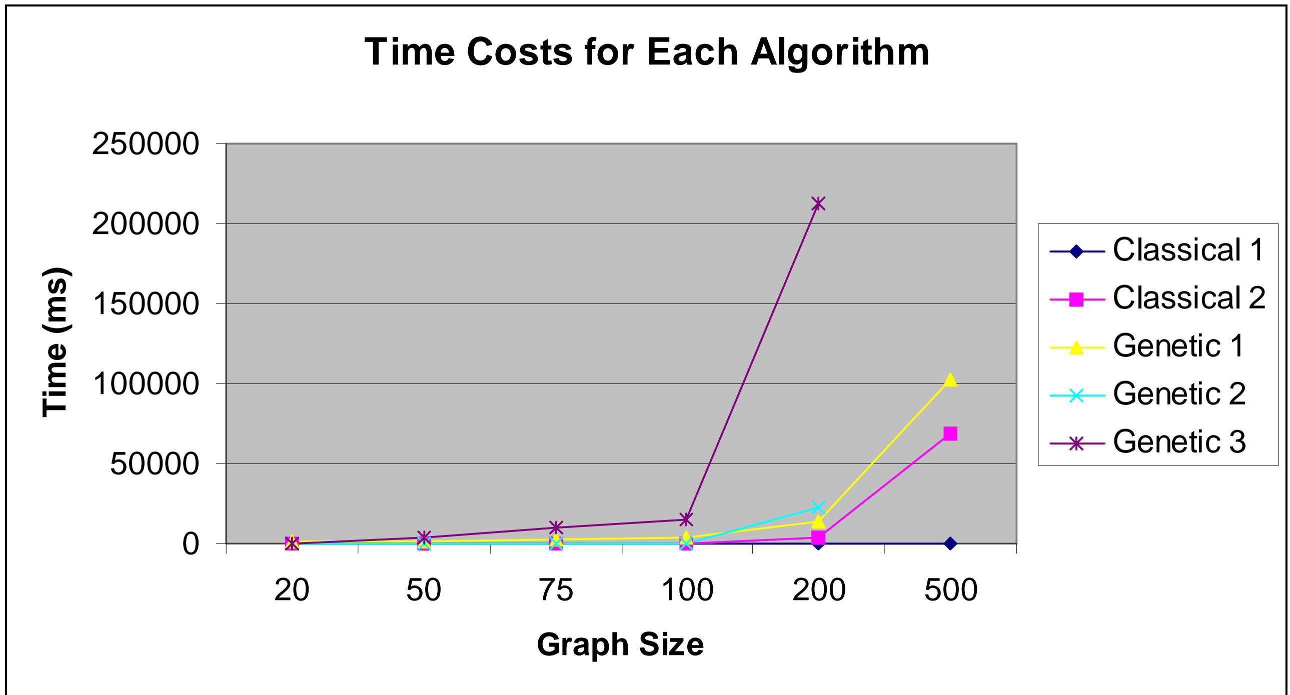


If we assume that 'Genetic algorithm 3' has found the optimal tours for the graphs of size 100 or less, then, the other three algorithms have the following proximity to the optimal solution:

Graph Size	20	50	75	100
Classical 1	65.97%	63.81%	68.84%	64.64%
Classical 2	100%	97.78%	98.54%	94.71%
Genetic 1	100%	97.91%	93.37%	91.52%

Here is the plot of all five algorithms. For graphs much larger than 100, the swap mutation algorithm really balloons up in its tour length, however the other algorithms remain pretty much the same.





Discussion:

About the class 'Ga.java'

Within this class I attempted to implement a more generic Genetic Algorithm that didn't rely solely on elitism as its selection method and mutation as it's principle evolutionary mechanism. The class does work from class 'Solve' using the '-ga' command, however the results are not very impressive. Basically the algorithm doesn't seem to be able to make forward progress, and only marginally improves on the initial tour. It does, however, incorporate crossover and tournament selection. The algorithm works as follows: A population of random tours is created and then sorted according to their tour lengths. The top half of the tours then mate with each other using a greedy crossover approach. ([1] *Jackson on how he did selection*) Crossover, as it applies to the TSP, is particularly delicate because parents cannot simply swap subsequences in their solutions as is usually the case. So, the crossover method I implemented uses a nearest neighbor scheme. I tried to incorporate the method described by *Louis* [2] where he describes the greedy crossover approach. It works by taking the first node of one of the parents and finding which node is closest to that node in one of the parents. It then takes that node as the next node and repeats the process. If a node has no edges that have not yet been visited in the child tour, then an unvisited node is chosen at random. The process repeats until a new child tour is complete. Once a child tour has been fully created with crossover a chance of mutation is applied, where a random sub-tour is inverted if the child is to be mutated. Once all of the children are created, they are appended to the population and it is sorted again, and the worst third of the population is discarded. The process repeats with the new generation to hopefully find a better solution in the next generation.

Some conclusions to draw from the data:

If we take into account the time costs for each algorithm, as is shown by the above data, we can see that we pay a huge price for the good solutions generated by 'Genetic 3' for graph sizes much larger than 100. This happens because the part of the algorithm that searches for better solutions operates much slower with larger graphs. The above data suggests that the best algorithm that balances time and quality

of tours for graphs of arbitrary size is Classical 2, the 2-optimal approach. Genetic 1 is a close second, and in fact would probably run a little faster if a scaling termination condition were built in. (i.e. terminating after fewer failed mutations for smaller graphs, then scaling up for larger ones). However, as long as the graph sizes are small, it is to our advantage to pay the extra cost of a few seconds to get the optimal solution from 'Genetic 3.' For extremely large graphs (greater than 1000 nodes) 'Classical 1' is the only reasonable choice.

As I discovered, evolutionary and genetic algorithms can prove to be formidable approximation strategies, as is most clearly evident in my 'Genetic 3' algorithm. While my GAs evolved through mutation only and were purely elitist in the selection of members of the next generation the application of randomized sub-sequences helped to break tours out of local minima, and I'm convinced through exhaustive testing that the solutions found by 'Genetic 3' (the '–shuffle' algorithm) are optimal for graphs of 100 nodes or less.

References Cited:

1. Jackson, William. *A Genetic Algorithm for the TSP*. http://ouray.cudenver.edu/~wcjackso/TSP.html
2. Louis, Shushil J. *Modified GAs for TSPs*. http://gaslab.cs.unr.edu/docs/techreports/gong/node3.html
3. Mertens Stephan. *TSP Algorithms in Action: Improving Solutions*. http://itp.nat.uni-magdeburg.de/~mertens/TSP/node3.html