The Generalized Travelling Salesman Problem

Finding the shortest possible USA 49 States Tour

CO353 - Final Project Brandon Yeh

1 Overview

In the Generalized Travelling Salesman Problem (GTSP), the nodes V are partitioned into k clusters $V_1, V_2, ..., V_k$ and the goal is to find a minimum cost circuit that includes exactly one node in each cluster.

In this report, a well known genetic algorithm published by John Silberholz and Bruce L. Golden is used to attempt to find a good solution to the 49 tour problem. http://josilber.scripts.mit.edu/GTSP.pdf.

However, this genetic algorithm only tests instances with up to 1000 nodes and a large number of clusters with approximately 5 nodes each. There is no computational data provided to show the performance of such an algorithm with siginificantly greater nodes and few smaller clusters. More specifically, this report will outline the performance of such an algorithm that contains 49 clusters of, on average, 2300 nodes each and 115475 nodes in total.

1.1 Technologies Used

A script was created and used to convert the lines in `usa115475_cities.txt` to ensure consistent tokenization of each x coordinate, y coordinate, city, county and state per line.

Implementation of this algorithm is done in Java 8 Version 74, using only the native libraries provided in the SDK. The computer used to run the computational tests have the following properties:

1. Acer Aspire 5755G-9417 with an Intel Core i7-2670QM 2.2GHz Processor
2. Operating System: Ubuntu 14.04.4 LTS running Linux Kernel 3.13.0-83-generic (i686)
3. 2x Crucial 4GB Single DDR3 1600 MT/s PC3-12800 CL11 SODIMM 204-Pin 1.35V/1.5V Notebook Memory CT51264BF160BJ
4. Samsung Electronics 840 EVO-Series 250GB 2.5-Inch SATA III Single Unit Version Internal Solid State Drive MZ-7TE250BW

Finally, Google's Static Maps API is used to visualize the tour across the continental United States.

2 Problem Formulation

This section of the report will formally define the problem that the modified genetic algorithm will solve.

The following data is to be used as input to the genetic algorithm:

1. A list of 49 States in the continental USA
2. A list of 115475 cities in the above 49 states provided from the following url: http://www.math.uwaterloo.ca/tsp/data/usa/usa115475_cities.txt

Furthermore, the following assumptions are made to simplify the problem:

1. The travel distance between 2 cities provided in the list of cities above is the Euclidean distance between the pair of cities. This identical to the EUC_2D method in TSPLIB.

Finally, the problem is defined formally:

> Given a list of 49 states in the continental USA and a list of cities within each state, find the shortest tour through each 49 states, visiting exactly one city in each state.

3 The Genetic Algorithm

As noted in the overview, the algorithm used to sove this 49 State USA tour is inspired by John Silberholz and Bruce L. Golden albiet, with a few modifications.

3.1 Input Parameters

The following input parameters are defined in the algorithm itself:

```
private int INITIAL_POPULATION_SIZE;
private int NUMBER_OF_ISOLATED_POPULATIONS;
private int TOTAL_GENERATIONS;
private int REPLICATON_SIZE;
private int REPRODUCTION_SIZE;
private int TERMINATION_CONDITION;
```

We first define the size of each isolated population as well as the number of isolated populations. Furthermore, for each replication and reproduction phase, we define how many tours are replicated and how many tours are generated by mating. Finally, we define how many generations the isolated population will evolve as well as a cutoff point for the replication and reproduction of the merged population.

3.2 Data Structures

In this algorithm, the following data structures are used to minimize the computational impact on processing the algorithm.

We define a *City* class as follows with the following instance variables

```
private Location coord;
private String city;
private String county;
private String state;
```

Where *Location* contains the x, y coordinates of the city and the other fields contain the city, county and state. Furthermore, an API is provided for this class to access and modify fields.

To easily access cities within a given state, a *HashMap* is used. Specifically, the code used is: `HashMap <String, ArrayList<City>> states;`. As part of the mrOX requires an iteration through all cities within a state, this setup provides constant time lookup to get information on all the cities in a state and then linear time to iterate through all the cities.

We finally define a *Tour* class as follows with the following instance variables:

```
private ArrayList <City> tour;
private int tourSize;
private double tourLength;
private int maxSize;
```

The main reasoning to use an *ArrayList* for a tour is that each pairwise adjacent city can be defined has having an edge between them. Then the order of cities in the *ArrayList* is the order of cities that the tour goes through. This allows the algorithm to compute the tour length in linear time, report the tour city by city in linear time and also easily replace any city in a tour with another city.

3.3 Isolated Population Generation

The algorithm begins with an initial set of isolated populations, generated randomly then optimized with the 2-opt algorithm. The algorithm will generate multiple sets of populations based on the `NUMBER_OF_ISOLATED_POPULATIONS` parameter, each with a population size of `INITIAL_POPULATION_SIZE`.

3.4 Natural Selection

The natural selection subroutine for this algorithm is inspired by the simple concept of natural selection. Given each isolated population's population size, we consistently segment this population into two groups. One group of tours that are destined to move on to the next generation and the other group destined to die off. We order the tours within the population in ascending order and select the best n tours, defined by `REPLICATION_SIZE` for replication and the remaining tours for disposal.

3.5 Reproduction

Reproduction of new tours is inspired by the Ordered Crossover technique implemented by the paper. The algorithm will select two random tours that were assigned for replication, and run the Rotational Ordered Crossover algorithm (rOX). The resulting child is then added to the population.

3.6 Population Merging

Once each isolated population has been processed through a set number of generations as defined by `TOTAL_GENERATIONS`. The final step is to merge all the isolated populations together into one large population and then repeat the natural selection and reproduction process again. However some differences in the subroutines are present in each cycle compared to a cycle in an isolated population.

In the natural selection subroutine, as we are selecting the top n tours defined as `REPLICATION_SIZE` * `NUMEBR_OF_ISOLATED_TOURS`. However, there may be cases where two or more tours have the same tour length. In this case, if there exists more than one tour with the same tour length in the replication group, we discard all but the first tour found.

In the reproduction subroutine, child tours are also generated using the Ordered Crossover technique, however the more computationally intensive modified rotational ordered crossover technique (mROX). Finally, the best tour returned by the mrOX heuristic is then run through a 2-opt local improvement heuristic, further improving the tour length.

3.7 Termination

Once the merged population begins to interate through generations, there may be a point where the tour set will converge on a single solution, or it takes a painstakingly long time to minimally improve the best solution found so far. To take care of these issues, a `TERMINATION_CONDITION` parameter is added to force the algorithm to terminate after a set number of iterations.

However, there may be cases where there does exist better solutions in the solution space and to ensure that the termination isn't done forcefully, an additional constraint on termination was added to allow a new generation of tours that may only occur after a few generation. The algorithm will then terminate when the `TERMINATION_CONDITION` is reach and if the best tour is stagnated for over 10 generations. This allows the algorithm to terminate once there is very mimimal possibility for significant progress.
This also creates a metric which may be interesting to observe and that is defined as *Generation Stagnancy*. That is, how many generations have passed without an improvement to the best solution. The largest number of stagnant generations may be interesting to observe as if a large percentage of the generations have not produced a better solution, it may be better to terminate the merging and take the best one found to date.

4 Heuristics and Subroutines

4.1 Importing Data

After the provided data has been parsed and modified for ease of tokenization, a linear time algorithm is used to import the data for use.

4.2 Random Tour Generation

Random tour generation, as its namesake, is a strictly psuedo random way to generate a tour. For each state, we randomly select a city and append it to the end of the tour. Furthermore, we generate a random permutation of the states so that the order in which the states are traversed is random.

4.3 Two Opt Heuristic

The 2-opt heuristic implemented in this algorithm is a traditional implementation. Simply put, the algorithm will find two edges that cross over, and uncross them to improve the tour. This heuristic will continue searching for all pairs of edges to uncross until there are no more edges that cross each other.

4.4 Ordered Crossover

The Ordered Crossover(OX) reproduction heuristic is implemented as outlined in the paper. Furthermore, rotational Ordered Crossover was implement albiet without the reversal of strings operation and modified rotational Ordered Crossover was implemented as well. Although mrOX is computationally expensive, linear time traversal through the cluster was done to maximize efficiency.

4.5 Output Data, Reporting and Metrics

To facilitate data recording and analytics, several data points within each iteration of the algorithm are printed to *stdout* as well as stored in log files. Types of data reported include:

1. Current iteration of algorithm
2. Best Tour Length found so far
3. Elapsed time at end of iteration of algorithm
4. Current tour generation stagnation values

Some additional processing of these log files can be done to further analyze the performance of this algorithm. Although these types of analytics are outside the scope of this report and are not tested, some metrics that may prove to be interesting include:

1. Average improvement rate of tours per generation
2. Number of stagnant generations within a testing instance
3. Ratio of stagnent generations vs improved generations

Metrics like these can be used to analyze the performance of the algorithm and help determine whether there are better input parameters that can be used to further improve the performance of the algorithm. Additionally, these metrics may also provide insight on whether there may be opportunities to use a different algorithm to further improve tour length when it seems that this algorithm is convergining on some value and minimal progess is being made.

5 Testing

Computational tests were done by varying the values of each individual input parameter. The goal is to determine whether a specific input paramter would significantly improve tour length generation. Additionally, three types of tests were done to compute the best tour available. The two type of tours are as follows:

1. Control Computation Instances

   Theses computational tests begin strictly with the input parameters and are run until completion. Then appropriate values in the tour are recorded.

2. Importing Best Control Tours Instances

   These computational tests allow the option to import any tour that was outputted as a result of running a Control Computation Instance. In order to prevent any errors propagating, these tours are calculated and recorded separately, noting which tours were imported as well as the results of the best tour generated.

The main reasons to segment these computational tests is to determine how effective the algorithm will be in finding a better tour, given the best tours already computed. Furthermore, to ensure that results are not polluted, the recording of the tours that allow imports are separated so as to not influence the results of a completely random instance of the algorithm.

5.1 Summary

Testing of various isolated population sizes have shown that very large isolated population sizes may not necessarily produce significantly better results. As each generation takes significantly longer, and as the solution coverges, there must exist a lot of wasteful computation.

The best solution found without importing any predefined tours had the following parameters:

```
private int INITIAL_POPULATION_SIZE = 100;
private int NUMBER_OF_ISOLATED_POPULATIONS = 10;
private int TOTAL_GENERATIONS = 25;
private int REPLICATON_SIZE = 0.4 * INITIAL_POPULATION_SIZE;
private int REPRODUCTION_SIZE = 0.6 * INITIAL_POPULATION_SIZE;
private int TERMINATION_CONDITION = 150;
```

With a tour length = 109427.

The top 5 results contain four tests that have the following above parameters. Furthermore, the generation stagnancy parameter for termination condition seems to be proving useful as several of these tours either run for over 250 generations to terminating at 150. So it is clear that depending on the random generation/modification components of the tour, the solution space may either quickly converge or extrememly slowly converge on the best tour. It

seems that the generation stagnancy is effective in drawing out the potential in a population.

The best solution while importing the best control tours has the following parameters:

```
private int INITIAL_POPULATION_SIZE = 50;
private int NUMBER_OF_ISOLATED_POPULATIONS = 5;
private int TOTAL_GENERATIONS = 25;
private int REPLICATON_SIZE = 0.4 * INITIAL_POPULATION_SIZE;
private int REPRODUCTION_SIZE = 0.6 * INITIAL_POPULATION_SIZE;
private int TERMINATION_CONDITION = 150;
```

The best tour length found so far using this method has length 109380.

The computational tests for these types of instances are very interesting as the best imported solutions tend to dominate all other solutions. Furthermore, the max stagnant generations values go up significantly and very early in the algorithm. It seems that using existing tours will significantly skew the solution set, and having 71 stagnant generations in a row in 150 iterations shows that the algorithm is not generating any better solutions at a good probability. Therefore, these tests can validate that the control tours generated are of near optimal length.

The full results can be found in the spreadsheet, *results.xlsx*, provided with this report.