**Technical Document**

*Zachary Petrusch, Nicklaus Benedict, Eric McAlpine, and Alex Lee*

**I. Class Structure**

The data is broken down into two easily-defined classes: Cities and Routes. These classes have been outfitted for ease-of-use. They both have copy-constructors and assignment operator overloads to make assigning values much easier, as well as un-argumented and argumented constructors for declaration. The City class acts as a container for the ID and coordinates for a city. It has a getter and setter for each data field. The Route class contains a versatile array of Cities. The array can be set using another array of Cities, the Route’s copy constructor, or using another Route with an assignment statement. The main use of the Route class is its distance function, which returns the overall distance between the cities, including the distance between the last city and the first.

The decision to use object oriented programming was made very early on to simplify the organization of the routes and the cities contained in them. The inclusion of the variety of methods to reassign a route makes using the classes very flexible, and speeds up debugging as well. Since the classes are relatively basic, no significant problems were encountered during the creation or testing process.

**II. The Genetic Algorithm**

We begin by reading in data from the file specified by the global variable FILE\_NAME. We then randomly created NUM\_ROUTE parent routes. This serves as the initial population. This initial population is sorted and the best solution is printed. Then, each successive generation is created by taking 50% of the best routes and combining them to get new routes that fill up the remaining 50% of the population. We use a merge sort to determine the best solutions.

For the sorting of our parents, we use a merge sort due to its efficiency and its ability to be parallelized. To start our project, we used a bubble sort. With 1000 parents the bubble sort version ran in approximately 13 hours. With the implementation of a merge sort in serial, our run time was reduced to approximately 57 minutes.

The crossover function that is used to generate the new solutions is based on the Edge Recombination Algorithm. This algorithm works by merging two routes into a new route that mostly consists of connections that existed in either of the parents. This algorithm also has an element of randomness in it, which can cause mutations through a few different methods. The algorithm works by taking a union of both parent’s adjacency matrices. This adjacency matrix represents any possibly connection that can be followed from one city to another in either of the parent routes. The route is then created by selecting the next city to travel to by valuing cities that have the least amount of connections over cities with more connections. If there is ever a point where there are no more possible connections within the parents that haven’t been used already for the current city then the next city is picked randomly from the remaining destinations that have not been travelled to – this is a mutation. Mutations are pretty rare, but do occur.

Each successive generation would once again have the top 50% best routes taken so that the best of the previous generation are not lost by creating the children. Whenever there is no improvement in the best solution from one generation to another generation we increment a counter. When the counter reaches NUM\_GENERATION\_STOPPER the program stops running and the current best solution is chosen.

**III. Parallel**

One of the areas of our code that is parallelized is the merge sort. If the global variable NUM\_THREADS is greater than one the merge sort will run in parallel. We set the OpenMP setting omp\_nested to true to allow for nested parallelism. Then, using parallel OpenMP sections, we divide the recursive merge sort calls up among NUM\_THREADS.

The area where most of the code spends its time is the cross-over function. Therefore we decided to add some aspect of parallelization to the cross-over. The way that this was done was by placing the whole Genetic Algorithm code inside of a parallel region. The handling of the control variables is done using an OpenMP Single command; so only one thread updates all the key variables. The call to the cross-over is placed inside of an OpenMP For directive, so that the work of creating the next generation is split across all spawned threads.

The merge sort will run in true serial if NUM\_THREADS is placed to 1, however the parallel region around the Genetic Algorithm code might have some overhead in this situation which would result in a stunted serial run time. That is why it is suggested to turn off the OpenMP master switch inside of Visual Studio if a serial run is desired. The instructions for turning off OpenMP in Visual Studio can be found in the User Manual.

**IV. Evaluation**

Test runs showed a small improvement in run-time in parallel versus serial. The speed up averaged around 1.2-1.5 times faster when ran in parallel. Because of the potential randomness in the creation of the first generation the best way to compare run times between solutions is to calculate iterations per second. An iteration is defined as the process of creating new children and finding the best solution out of the new generation mixed with the old generation. Since the number of iterations can vary from run to run, and this would influence total run time, it is better to compare how long the code took per iteration.

Another important issue is the suitability of solutions found. We found that our program could get solutions within 10% of the optimal solution for our data sets as long as suitable parameters were given. For bigger data set larger populations are needed to get answers that are very close to the optimal answer. There was no noticeable variance between the quality of solutions in parallel and serial. Below is data for two runs performed on a data set with 51 cities and an optimal solution of 426.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Type | Time (seconds) | Iterations | Iterations/Second | Speed Up | Solution | Suitability (optimal/solution) |
| Parallel | 2673.31 | 115 | 0.043017832 | 1.186 | 452.282 | 0.94189 |
| Serial | 3583.38 | 130 | 0.036278597 | - | 446.481 | 0.954128 |