Data Structures and Algorithms

Project 3

Travelling Salesman Problem

**Functional Decomposition**

This document serves as the technical documentation for all the methods/procedures/functions used in the implementation of the project titled above. The following are the method definitions/signatures for the methods used in the project.

**FILE1:** brutus.c

Signature: long f(int n);

Calculates and returns the factorial of (n) recursively, using the base case as f(n<=1) = 1. This function is used to generate the number of permutations there are for n cities. It is called by the *perm* function.

Signature: void perm(float adj[rs][cs], int\* s, int n);

The method generates the n! different permutations of n distinct cities in a Travelling Salesman Problem. This method is used to find an optimal Travelling-Salesman path for (n) cities using a Brute Force approach. The implementation for the method was provided by the project description.

*Signature*: void print\_perm(int\* s, int n);

This is a helper function used to print out a permutation of the cities while they are generated by the *perm* method. It prints out all the integers in the array called ‘s’ in order.

*Signature*: void init\_perm(int\* s, int n);

This helper routine initializes the array ‘s’ with an initial path of {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0}. (for n = 10) This initial path is then used by the *perm* method to generate all the other (n! - 1) permutations of the n cities.

*Signature*: float travel(float adj[rs][cs], int\* path, int n);

This method takes two inputs: the adjacency matrix for the travelling-salesman graph and a possible path and calculates the cost associated with that path by traversing it in the order. The adjacency matrix is used to find the weights of individual edges in the path.

**FILE2:** genetic.c

*Signature*: void init\_route(int\* path, int n);

The method is used to initialize a path in a generation. Given an *int\* path* it allocates memory for n integers and initializes all of them with zeroes.

*Signature*: void init\_gen(route\*\* routes, int ntours, int n);

This method, with the help of *init\_route* and *copy\_route* methods creates the first generation of routes using the exact same algorithm as that used by the Brute Force approach, but only calculates a fixed number of them, namely the number of tours per generation chosen by the user.

*Signature*: void create\_gen(route\*\* routes, int ntours, int n);

Given a generation of routes, this method creates the next generation of routes by choosing two elites out of the original generation, adding a few mutations of the elites and filling up the rest of the generation using freshly generated permutations of the last mutation.

*Signature*: void selection\_sort(route\*\* routes, int ntours, int n);

This helper function sorts the array of *route\** based on the *cost* value stored in the corresponding struct. This allows for sorting a generation of routes with respect to the associated cost and also simplifies the process of finding the elites in a generation.

*Signature*: void mutate(int\* path, int n);

This method is used to generate mutations of the elites in a brand-new generation of routes. The method swaps random cities from the elite routes to generate mutated routes. It is the most likely to choose the first element for the swap from the first half of the route and the second element for the swap from the second half of the elite route. The total number of mutations are divided equally with mutations of each of the two elites in the new generation.

*Signature*: void copy\_route(int\* orig, int\* copy, int n)

This method is used to copy the elements in one route into another. This method is called by the *init\_gen* method to initialize the first set of routes, as well as by the *create\_gen* method to create the fresh permutations.

**FILE3:** grapher.c

*Signature*: void readFile(float adj[ROWS][COLS]);

*Signature*: void print(float adj[ROWS][COLS]);

This method is called by the ‘cp’ command to recursively copy the individual files inside the directory specified by the ‘oldFile’ parameter. The new directory created gets the name specified in the parameter/argument ‘name’.

*Signature*: float travel(float adj[ROWS][COLS], int\* path, int n);

This method, beginning at the root of the ‘root’ node, using a depth-first search, traverses through the tree in form of a backtracking function to search for a file or directory with the name as specified in the ‘name’ argument/parameter. If found, the value at the ‘found’ integer variable is written to 1, if not, then the value is left to stay at 0, which it was set as default. At the first occurrence of the file or directory with the specified name, the method tracks its location all the way back to the root of the tree, and after storing the parent directory names into a stack, prints out the path to that file beginning from the root. The complete search performed by this method guarantees that if a file or directory with the name as in ‘name’ exists, it is found by the method.

**FILE4:** main.c

Signature: int main(void);