**Jacob Rook**

**Data Structure and Algorithms II**

**Project 3**

**Functional Decomposition**

**User-defined data structures used as parameters in the functions**

**#define** MAXCITIES 20

**double** CITYGRAPH[MAXCITIES][MAXCITIES]; //Adjacency Matrix to simulate a weighted graph

/\*

\* Main variables the TravelingSalesman program runs off of

\*/

**int** CITIES; //Number of cities to visit

**int** TOURSNGEN; //Number of tours in a generation

**int** GENERATIONS; //Number of generations to run

**int** MUTATIONS; //Number of mutations in a generation

/\*

\* @brief Tour Tour structure to hold the tour in linear order and the

\* weight of the tour.

\*

\* @var cityTour Array to store the tour. Cities are defined by the index

\* in the adjacency matrix CITYGRAPH. Starting city is assumed to be the 0th

\* index and is placed at the end of the tour. The first element in the cityTour

\* array (element 0) is the first city visited and element 1 the second, etc.

\*

\* @var tourWeight Variable stores the total weight of the tour in cityTour

\* starting at city 0 and ending at city 0.

\*/

**typedef** **struct** T {

**int** cityTour[MAXCITIES];

**double** tourWeight;

}Tour;

Tour BESTTOUR; //Best tour found by the brute force algorithm

Tour BESTGENTOUR; //Best tour founf by the genetic algorithm

/\*

\* Two Tour pointers to simulate a heap for a given generation. Memory will be

\* allocated to create an array based heap to sort the tours in the generation.

\* There are two heaps to translate some elements from one generation to the other.

\*/

Tour \*GenHeap1;

**int** GenHeap1Size;

Tour \*GenHeap2;

**int** GenHeap2Size;

**Files and Functions in the Program**

/\*

\* @file rook-j-p3.c File contains the main function for Project 3, the traveling

\* salesman problem. The main function asked the user for the desired input and

\* sends that information to the Traveling Salesman algorithms. The main function

\* asks the user if they want to generate a new cityWeights.txt if so the genNewWight

\* function is called. There is not input validation, so any invalid input will crash

\* the program.

\*

\* @author Jacob Rook

\* @date 11/02/2017

\* @info Course COP 4534

\* Project 3

\*/

**Rook-j-p3.c**

/\*

\* @file TravelingSalesman.h File contains all of the function prototypes used

\* for the Traveling Salesman problem. This file contains two algorithms that are

\* used to try to determine the best path to visit all cities once, return to the

\* first, with the lowest cost possible. There is a brute fore method that just

\* test every single permutation of the city arrangement which is (n - 1)! complex

\* and a genetic algorithm that create generations of tours, determines the best,

\* mutate the best, and repeat for a number of times.

\*

\* @author Jacob Rook

\* @date 11/02/2017

\* @info Course COP 4534

\* Project 3

\*/

**TravelingSalesman.c TravelingSalesman.h**

//BEGIN OF HEAP FUNCTIONS

/\*

\* @brief HeapEnqueue Adds a tour to the given Heap to create a generation.

\* Since there are multiple Heaps, the specific Heap with it's size has to be

\* passed.

\*

\* @param TourEnque Tour to be added to the Heap

\*

\* @param Heap Array of Tours to simulate Heap

\*

\* @param HeapSize Variable to keep tract of the Heap size

\*/

**void** **HeapEnqueue**(Tour \*TourEnque, Tour Heap[], **int** \*HeapSize);

/\*

\* @brief HeapDequeue Takes a tour out of the heap, replaces it with the

\* lowest member of the heap, percolates the member down, then returns the tour

\* taken out of the heap

\*

\* @param Heap Heap to be Dequeued

\*

\* @param HeapSize Size of the given Heap

\*

\* @return Tour Returns the least costly tour in the Heap

\*/

Tour **HeapDequeue**(Tour Heap[], **int** \*HeapSize);

/\*

\* @brief isHeapEmpty Used to determine if a given heap is empty by the

\* Heap size

\*

\* param HeapSize Size of the Heap

\*

\* return int returns 1 if the heap is empty, else 0

\*/

**int** **isHeapEmpty**(**int** \*HeapSize);

/\*

\* @brief percolateDown Percolates a node in the Heap down into the correct

\* place at a given index

\*

\* @param Heap Heap to implement the percolateDown

\*

\* @param HeapSize Size of given Heap

\*

\* @param index Index to start the perolateDown algorithm

\*/

**void** **percolateDown**(Tour Heap[], **int** \*HeapSize, **int** index);

//END OF HEAP FUNCTIONS

//START OF TRAVELINGSALESMAN FUNCTIONS

/\*

\* @brief strartTravel Function that initiates the traveling salesman

\* algorithm. The function calls initTourVar which initializes the important

\* variables, calls exeBruteForce, then calls execGentetic which are the two

\* solutions to the traveling salesman problem. The function also determines

\* the time each algorithm takes to execute.

\*/

**void** **startTravel**(**int** numCities, **int** numTours, **int** numGen, **double** percentMut);

/\*

\* @brief initTourVar Function initializes the important variables used

\* in the simulation. Namely: CITIES, TOURSNGEN, GENERATIONS, MUTATION, CITYGRAPH

\* and the first generation. Function calls populateGraph to initialize CITYGRAPH

\* and calls initGeneration to initialize the first generation.

\*

\* @param numCities Value for CITIES

\*

\* @param numTour Value for TOURSNGEN

\*

\* @param numGen Value for GENERATIONS

\*

\* @param perentMut Double used to calculate how many mutations in a generation

\*/

**void** **initTourVar**(**int** numCities, **int** numTours, **int** numGen, **double** percentMut);

/\*

\* @brief genNewWeight Function is used to generate a new set of weights

\* for the cityWeights.txt file used in the simulation. The function uses the

\* rand() function to chose doubles between 0 and 100. The range can be changed

\* in the function. The user must call the function themselves to generate a

\* new cityWeights.txt file.

\*/

**void** **genNewWeight**(**void**);

/\*

\* @brief populateGraph Function reads weights from cityWeights.txt file and

\* stores the weights in CITYGRAPH. The cityWeights.txt file must be filed with

\* 380 doubles to fill the whole graph.

\*/

**void** **populateGraph**(**void**);

/\*

\* @brief initGeneration Function dynamically allocates the memory for the

\* two heaps for the generations. Creates an array of size GENERATIONS + 1. The

\* first generation is also initialized with the permutation algorithm used by the

\* brute fore algorithm. The first generation is filled with the first permutations

\* of the algorithm.

\*/

**void** **initGeneration**(**void**);

/\*

\* @brief exeBruteForce Function uses an iterative approach to produce every

\* permutation possible with the given number of cities. After each permutation

\* is made the weight of the tour is calculated and if it is lower than the current

\* lowest tour, the tour is saved and the algorithm continues until all permutations

\* are tested and the lowest cost tour is found. The lowest cost tour is stored

\* in BESTTOUR

\*/

**void** **execBruteForce**(**void**);

/\*

\* @brief initTour Function initializes the passed tour with a tour of

\* 1->2->3->...->CITIES - 1 and stores the calculated tour weight in it's variable

\*/

**void** **initTour**(Tour \*init);

/\*

\* @breif fact Function calculates the factorial of a given number

\* and returns it

\*

\* @param n Number to find the factorial for

\*

\* @return factorial Returns the factorial of n

\*/

**int** **fact**(**int** n);

/\*

\* @brief swap Function swaps two cities positions in a given tour

\*

\* @param p Index to switch p with

\*

\* @parm q Index to switch q with

\*

\* @parm TourSwap Tour to do the swap to

\*/

**void** **swap**(**int** p, **int** q, Tour \*TourSwap);

/\*

\* @brief compareTour Function compares two Tours to determine which has the

\* lower weight. returns 1 if Tour1 is less than Tour2

\*

\* @param Tour1 First tour to compare

\*

\* @param Tour2 Second tour to compare

\*

\* @return int Returns 1 if Tour1 is less than Tour2, 0 if Tour1 is

\* greater than Tour2

\*/

**int** **compareTour**(Tour \*Tour1, Tour \*Tour2);

/\*

\* @brief setTourEqual Function copies the tour information from T2 to T1.

\*

\* @param T1 Tour that is being set

\*

\* @param T2 Tour that is being copied

\*/

**void** **setTourEqual**(Tour \*T1, Tour \*T2);

/\*

\* @brief execGenetic Function executes the second method of determining the

\* least cost tour. The function creates new generations based from the last

\* generation. The best two tours are copied from the last generation and saved in

\* the new generation. Then the top two generations are used to create mutations

\* to fill the generation. The rest of the generation is filled with random

\* permutations created by tourRandPerm. execGenetic uses populateGeneration

\* to switch between the two heaps, populateGenerate allows for the heaps to

\* get the tours of interest.

\*/

**void** **execGenetic**(**void**);

/\*

\* @brief populateGeneration Function creates a new generation through the

\* Tour array Heap2. The top two tours in Heap1 are transfer to Heap 2. Then,

\* those tours are used to create mutations. After the mutations are added,

\* random permutations are added to the generation until the generation is full.

\*

\* @param Heap2 New generation to be created

\*

\* @param Heap2Size Variable to store the size of the heap for Heap2

\*

\* @param Heap1 Old generation to be used to help create Heap2

\*

\* @param Heap1Size Variable to store the size of the heap for Heap1

\*/

**void** **populateGeneration**(Tour Heap2[], **int** \*Heap2Size, Tour Heap1[], **int** \*Heap1Size);

/\*

\* @brief tourMutate Function Mutates a given tour by swapping the cities

\*

\* @param Mut Tour to be mutated

\*

\* @param MUTRANGEMAX Variable to help determine how mutated a mutation will

\* be. The higher the max, the more the tour is mutated.

\*

\* @param MUTRANGEMIN Variable to help determine how mutated a mutation will

\* be. The higher the min, the more the tour is mutated.

\*/

**void** **tourMutate**(Tour \*Mut, **int** MUTRANGEMAX, **int** MUTRANGEMIN);

/\*

\* @breif tourRandPerm Function generates random tour permutations

\* by use of a mixture of the permutation algorithm the brute force algorithm

\* uses and use of the rand() function.

\*

\* @param Perm Tour to store the random permutation.

\*/

**void** **tourRandPerm**(Tour \*Perm);

/\*

\* @brief calTourWeight Function calculates the weight of a given tour by

\* adding up the weights between each city. Function store the weight into the

\* structures variable and also returns the weight;

\*

\* @param T Tour to be calculated

\*

\* @return double Returns the weight of the given tour.

\*/

**double** **calTourWeight**(Tour \*T);

/\*

\* @brief printTour Prints the passed tour with the cities linearly

\* aligned separated by arrows. Also print the tour weight.

\*

\* @param T Tour to be printed

\*/

**void** **printTour**(Tour \*T);

/\*

\* @brief printResults Prints the results of the simulation to the output.

\* That is the number of cities run, optimal cost from brute force, time the

\* brute force algorithm took, cost from the genetic algorithm, time it took to

\* run, and the percent of optimal that the genetic algorithm produced.

\*/

**void** **printResults**(**double** BFTime, **double** GTime);

/\*

\* @brief caltimeDiff Calculates the time difference between two timespec

\* variables and returns the difference as a double

\*

\* @param End End time used to calculate the difference

\*

\* @param Start Start time used to calculate the difference

\*

\* @return double Returns the double representation of the difference

\*/

**double** **calTimeDiff**(**struct** timespec \*End, **struct** timespec \*Start);

//END OF TRAVELINGSALESMAN FUNCTIONS