Old Dominion University

Travelling Salesman Problem

Nicholas Mark Barnes

Introduction to Discrete Structures CS381

Dr. Ralph Grove

# Introduction

The Traveling Salesman Problem (TSP) is a problem that has intrigued scientists for decades. For my honors contract course, CS 381, Introduction to Discrete Structures, I'll be pursuing a variant of this problem listed as: “A salesman wants to take the shortest possible route through a number of cities that includes each city exactly once and return to his original location. Assuming no roadblocks, find the shortest route.” (http://www.math.uwaterloo.ca/tsp/problem/index.html)

As the problem has been around for quite some time, there are already numerous ways to solve the TSP. These solutions consist of exact solutions and heuristic solutions, which both have their advantages and disadvantages. Exact algorithms produce the optimal solution to the TSP. A possible exact solution consists of brute force, comparing all possible routes through all cities and finding the shortest route possible. (<http://www.mathematics.pitt.edu/sites/default/files/TSP.pdf>) However, the time required to calculate all of these routes would be proportional to the factorial of the number of cities in question minus one { (n-1)! } . For example, it would be easy to calculate the shortest distance through 3 cities, because there would only be two possible routes to compare, and the computation involved is minimal. But as the number of cities increased to 9 cities, the number of combinations of cities to travel between is increased to to (9-1)!, or 40320. This is even further accentuated when there are hundreds of cities, and the time it would take to calculate all possible routes would be impractical.

Another exact algorithm includes the branch and bound method, which divides the route into a tree. The tree can be explored by its branches, where each branch has a cost. A group of branches can be formed either breadth first, which takes the greatest costs and puts them together, depth first, which keeps track of the lowest cost route calculated so far (http://intelligence.worldofcomputing.net/ai-search/depth-first-branch-and-bound.html#.XNF97LvPxPY), or cost-first, which “expand[s] the node that adds the least overall cost to the partial objective function.” This, however, is still impractical as it is slow. (https://www2.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html)

So, many people have taken a heuristic approach, using artificial intelligence to narrow down the combinations of cities that might produce the shortest possible route. One such approach to this is called Nearest Neighbor, in which the program selects the closest city to the one it is currently on, then repeat this for all cities. Another heuristic solution is the Clarke-Wright heuristic. A hub vertex is identified, a starting cost is calculated from visiting all of the edges between points, the shortest routes along the edges are identified, and altogether the shortcuts are added to form a solution. (<https://www2.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html>)

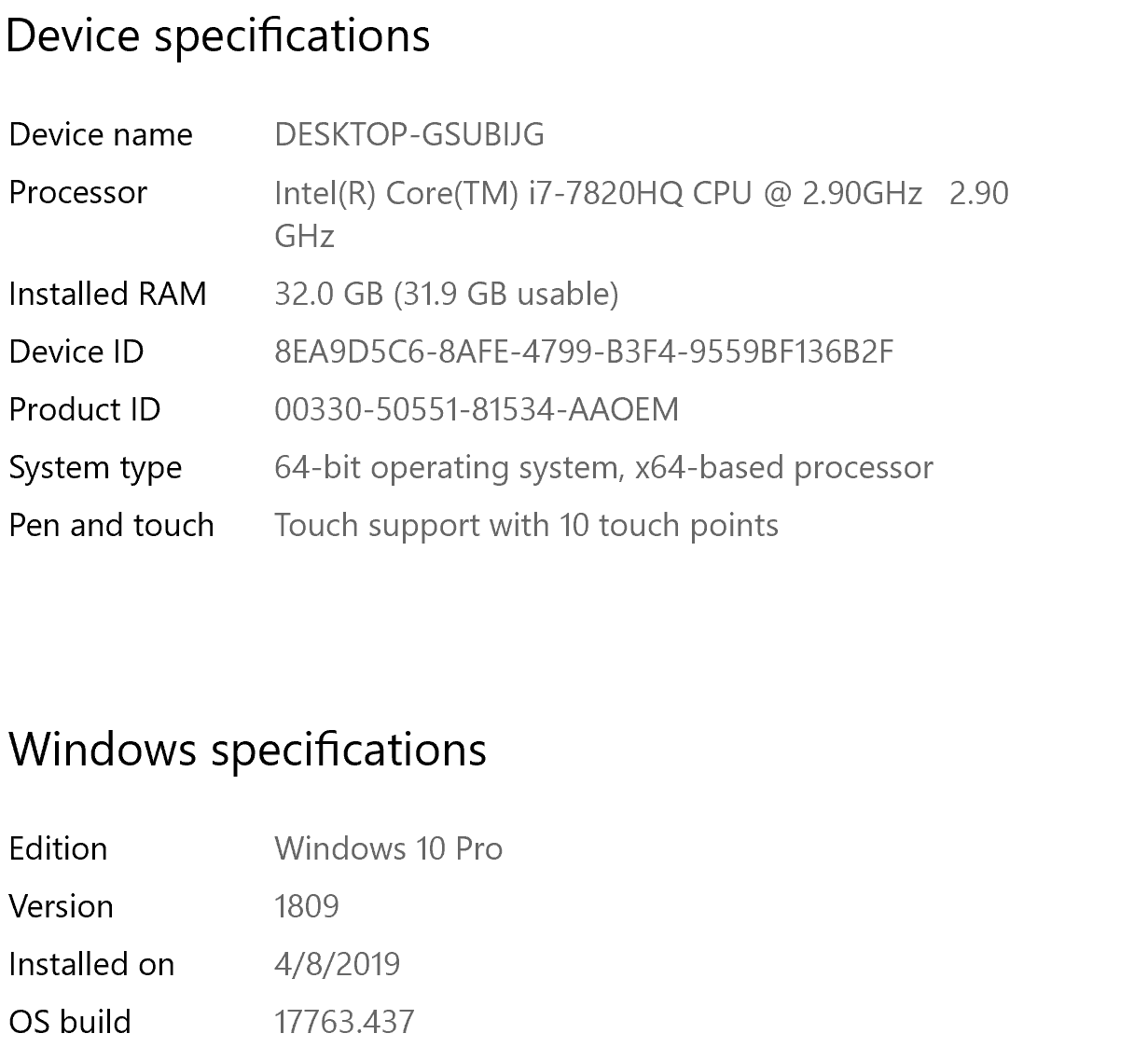
These approaches can be called approximation algorithms, which “may produce a Hamilton circuit with total weight W’ such that W ≤ W’ ≤ cW, where W is the total length of an exact solution and c is a constant.”(Discrete Mathematics and its Applications) In other words, the algorithms would produce satisfactory results that may not be exactly correct.

For the purposes of this assignment, I will be implementing a brute force algorithm that will take a small number of cities and find the shortest possible route. What follows is my proposition of a solution to this variation of the TSP solution.

What follows is the specifications of the device I ran my rests on, my description of the program and how it runs, and the issues I faced in developing the program.

I performed my experiments on a Dell Precision Laptop, and its specifications are described below.

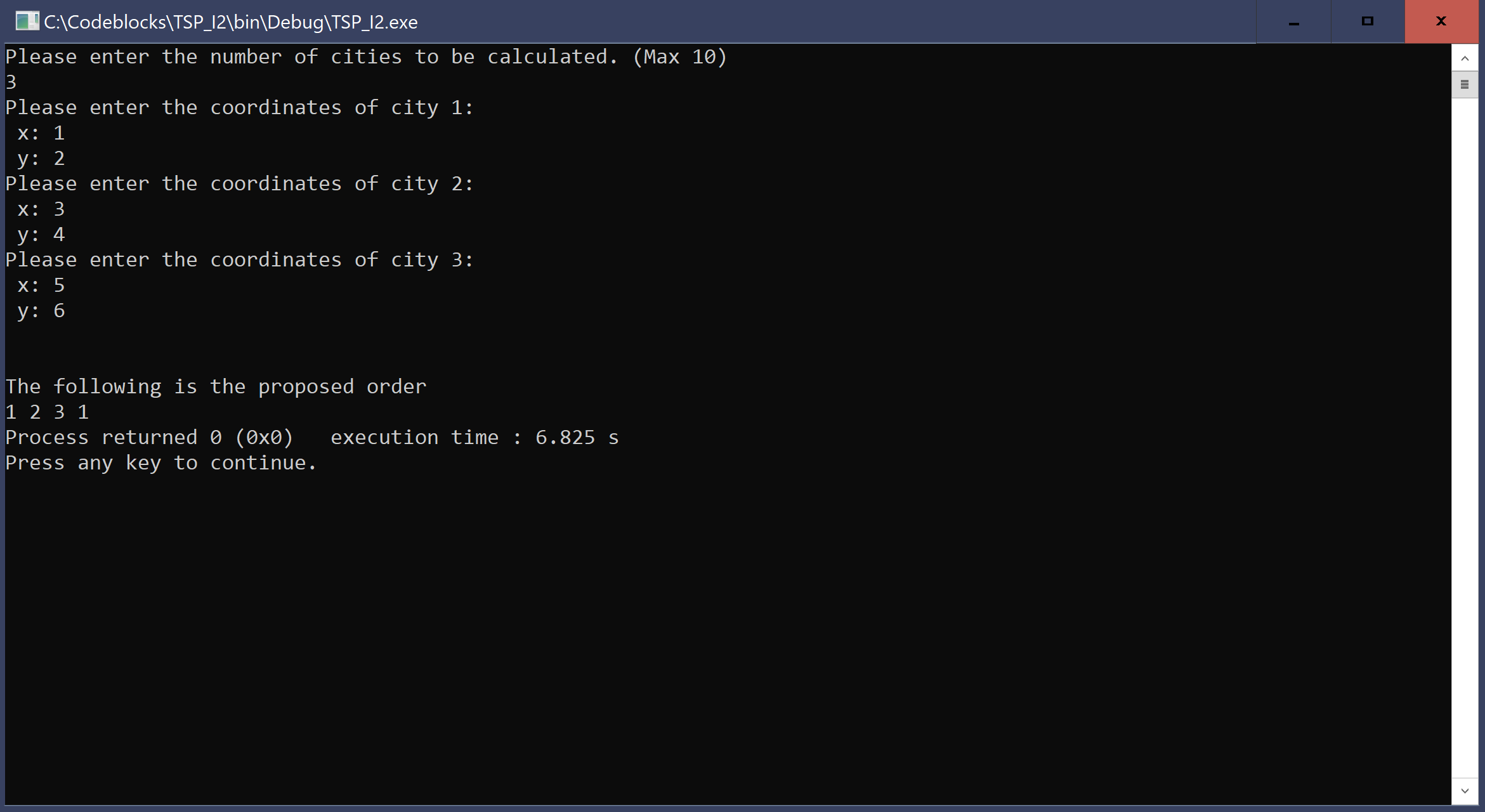
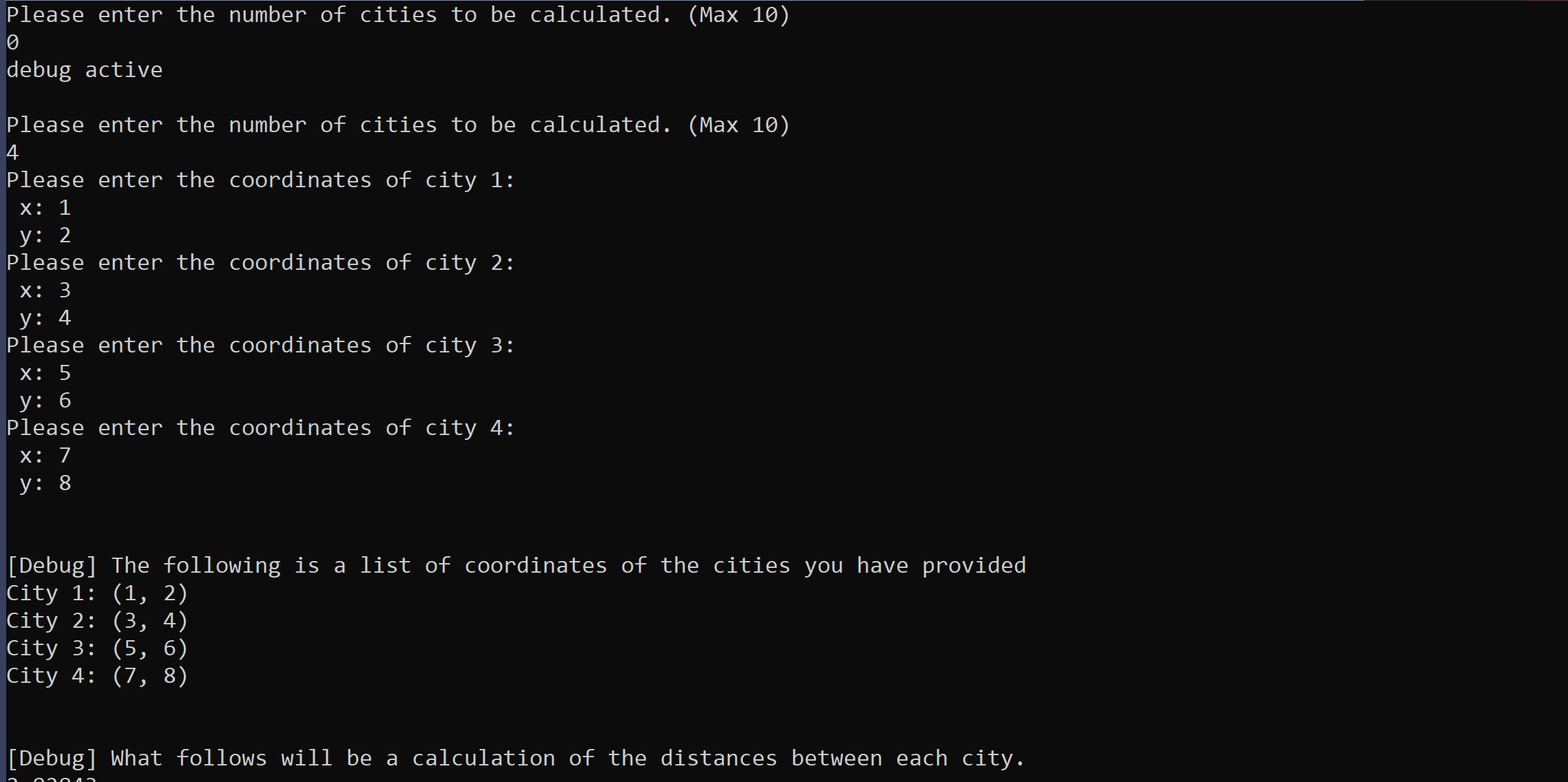
# Device Specifications



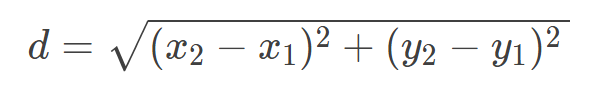
# Program Description

Before I created my solution, I needed to decide how I wanted to execute my program. My first choice was how I accepted data for the locations for each city. I chose to input it directly into the program for simplicity’s sake, so the user wouldn’t have to deal with an input file. My decision was that X and Y-coordinates for each city would be inputted, and that the program would calculate each possible route and their cost and output the optimal route.

The maximum number of cities that can be entered into this program is 20. However, after 13 cites, which takes 4 minutes to calculate each route, the time it takes to complete the operation is too large and is impractical. In addition, by inputting 0 into the number of cities, debug mode activates, which can display all the steps the program takes to calculate the optimal route.



The distance formula, given below, is used to calculate the distances between cities.

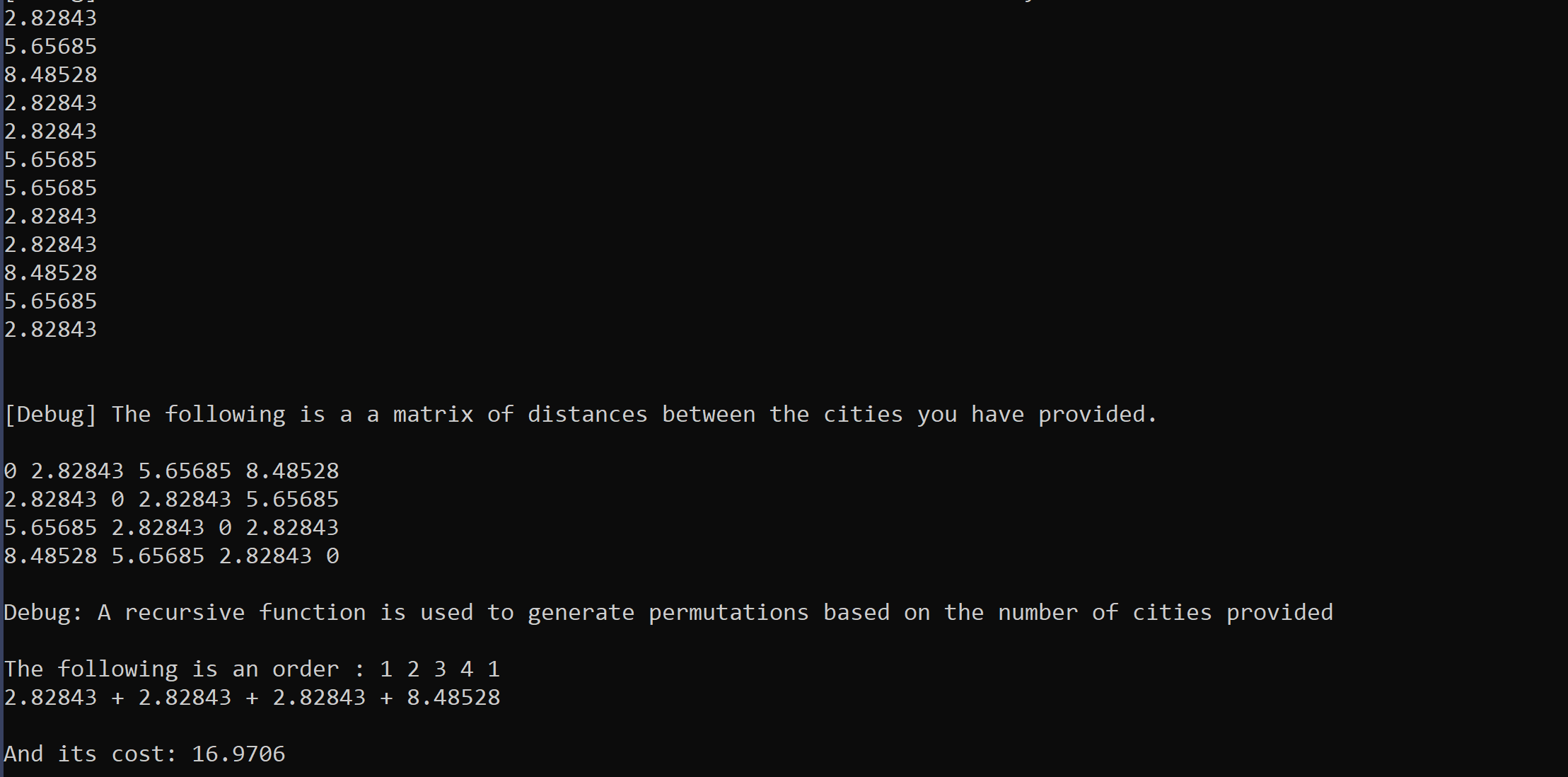


Where x1 and y1 are the corresponding coordinates for the first city and x2 and y2 are the corresponding coordinates for the second. The distance d is stored in a two-dimensional array

## cities[a][b]

where a and b are the cities which the distance is between.

Finally, to choose the optimal route, a recursive function that generates permutations based on the number of cities entered is called.



The permute() function generates all possible permutations and another function filterarray()filters routes that did not start with city one, since with how I implemented the program, permutations that did not start and end at one would not meet the requirement to “return to its original location” which would be city 1.

To clarify, a route {1,2,3,4,1} would pass through the filter, while another route {2,1,3,4,2} would not.

Calculatecost()is a function that takes the route given by permute()and uses the distances stored in the cities[a][b] array to calculate the sum of the distances between each city in the route.

All possible routes starting at city 1 are calculated and the route with the smallest cost, or with the least total distance, is chosen to be outputted at the end of the program.

All code is included in **Appendix 1**.

# Challenges and Difficulties

The challenges I faced in creating my solution lie in more of the advanced logic, such as implementing a recursive function and adding values from an array in a for loop. This line in particular:

## cost+=cities[temporder[k]-1][temporder[k+1]-1];

caused me many problems because of how positions in an array are called. Not only did I have to call the cities from the order array, but I had to subtract them by 1 in order to calculate the correct subscript.

In addition, there was also the issue of processing time for the program. I was hesitant to go over 10 cities to calculate the cost of each route, due to possible memory limitations. However, I eventually realized that after running over 10 cities, that only two of the routes are stored at a time, that being the route to be calculated and the smallest possible route calculated thus far. The permute() function takes the route to be calculated in the global array[20] which contains a list of names of the cities: 1-20. In regards to the coordinates of the cities I tested, I chose coordinates from a range of values 0-9, sometimes choosing them in order, sometimes in random, and sometimes in various amounts of digits (ex 1,123,12314).

As stated earlier, I ran tests for cities 1-13. The time elapsed for each test run was instantaneous for amounts 1-10. Since the number of routes calculated was (n-1)!, the max number of routes for this range was 362880. Afterwards, however, the max number of routes increases by a factor of 10. Calculation for 11 was still instantaneous with about three million routes, but 12 was a bit longer at two minutes, and 13 at 5 minutes total runtime. 14 is where I found some difficulty in running the program. The runtime for about six billion routes exceeded 30 minutes, so running the program any further just proves to be inefficient.

# Conclusion

In this paper, I gave a description of the Travelling Salesman Problem I conquered, possible solutions to the problem, a description of my solution, and the challenges I faced in developing my solution. In developing my solution, I learned how to apply my knowledge of programming and the logic I learned from CS381 in a real world problem. The TSP can be applied to numerous situations, ranging from the most efficient route to deliver pizza to residents of New York City, or soldering parts on a board in the most efficient space.

# Bibliography

1. “The Problem.” The Traveling Salesman Problem, Aug. 2015, www.math.uwaterloo.ca/tsp/problem/index.html.
2. Brucato, Corinne. “THE TRAVELING SALESMAN PROBLEM.” University of Pittsburgh, 2013.
3. “The Traveling Salesman Problem (TSP).” Extreme Algorithms, www2.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html.
4. Rosen, Kenneth H. Discrete Mathematics and Its Applications. 7th ed., McGraw-Hill, 2019.
5. Robin. “Artificial Intelligence.” Artificial Intelligence RSS, 18 Dec. 2009, intelligence.worldofcomputing.net/ai-search/depth-first-branch-and-bound.html#.XNF97LvPxPY.

# Appendix 1: Code

#include <iostream>

#include <cmath>

using namespace std;

int array[20] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}; //used to generate permutations

double cities[20][20]={0}; //initial declaration of cities and allocate memory. Any higher than this would be too resource intensive

double smallest=999999; //used to compare each of the paths

bool debug\_flag=0;

void permute(int k,int& size, int\* order);

void filterArray(int& size,int\* order);

void swap(int x, int y);

void calculatecost(int\* order, int& size);

/\*

Main:

Accepts input from the user for the number of cities to be calculated, as well as their locations in x and y coordinates. An input of 0 activates a debug mode which displays all possible routes and their cost. An input of 1 only has one possible path, so it calls the only possible path: {City 1}. An input greater than 6 in debug mode will display a warning tells the user that processing time will be drastically increased due to the output of all possible routes. The distance: sqrt((x2-x1)^2+(y2-y1)^2) is calculated between each city, stored in a two dimensional array, cities[x][y]. Finally, using a recursive function permute(), all routes are calculated, with the one with the lowest cost between cities outputted at the end of the program.

\*/

int main()

{

double location\_x[20];

double location\_y[20];

int order[21];

int num\_cities;

int a; //number of cities

int i; int j;//counters

Prompt:

cout<<"Please enter the number of cities to be calculated. (Max 10)"<<endl; //Prompt user to enter number of cities to calculate.

cin>>num\_cities;

a = num\_cities;//makes it easier to call

if (num\_cities == 0) //Activates a debug prompt, which shows all possible routes and the calculations used to calculate their costs.

{

if(debug\_flag==1)

{

cout<<"debug inactive"<<endl<<endl;

debug\_flag=0;

goto Prompt;

}

debug\_flag=1;

cout<<"debug active"<<endl<<endl;

goto Prompt;

}

if (num\_cities == 1) //Case: 1

{

cout<<"Only one possible path exists\n\n1\n";

goto Prompt;

}

if (num\_cities>6)

{

if (num\_cities>10){

cout<<"num cities greater than 10"<<endl<<endl;

}

if (debug\_flag==1){ //Disclaimer for the amount of time it can take to output large quantities of data.

cout<<"For general purposes, this program can handle up to ten cities.\n\nHowever, during debugging, as the number of cities increases,"<<

"the amount of time it will take will be much\nlonger in order to display each possible combination.\n\n\n"<<

"Please take care not to use too many cities when using the debugging function.\n\n";

}

}

for (i=0; i<a; i++) //accept coordinates from user

{

cout<<"Please enter the coordinates of city "<<i+1<<": \n x: ";

cin>>location\_x[i];

cout<<" y: ";

cin>> location\_y[i];

}

if (debug\_flag==1)

{

cout<<endl<<endl<<"[Debug] The following is a list of coordinates of the cities you have provided"<<endl;

for (i=0; i<a; i++)

{

cout<<"City "<<i+1<<": ("<<location\_x[i]<<", "<<location\_y[i]<<") "<<endl;

}

cout<<endl<<endl<<"[Debug] What follows will be a calculation of the distances between each city."<<endl;

}

///Use the distance formula to calculate the distances between each city

for (i=0; i<a; i++)

{

for (j=0; j<a; j++)

{

if (i!=j)

{

cities[i][j]= sqrt(pow(location\_y[j]-location\_y[i], 2) + pow(location\_x[j]-location\_x[i],2) );

if (debug\_flag==1)

cout<<cities[i][j]<<endl;

}

else

cities[i][j]=0;

}

}

if (debug\_flag==1)

{

cout<<endl<<endl<<"[Debug] The following is a a matrix of distances between the cities you have provided."<<endl;

for (i=0; i<a; i++)

{

cout<<endl;

for (j=0; j<a; j++)

{

cout<<cities[i][j]<<" ";

}

}

}

if (debug\_flag==1)

cout<<endl<<endl<<"Debug: A recursive function is used to generate permutations based on the number of cities provided"<<endl;

permute(a,a,order);

cout<<endl<<endl<<"The following is the proposed order"<<endl;

for (i=0; i<a; i++)

{

cout<< order[i]<<" ";

}

cout<<1;

//end of program

}

/\*

swap:

A swap function is implemented to generate permutations in the permute() function.

The array value in an x position is swapped with a value in the y position.

\*/

void swap(int x, int y){ //Swap function

int temp = array[x];

array[x]=array[y];

array[y]=temp;

return;

}

/\*

filterArray:

This function takes the possible route generated by permute() and

determines whether to calculate the cost based on the value in the [0]

position. If 1, filterArray calls a function that calculates the cost of the route.

\*/

void filterArray(int& size, int\* order){ //Only calculates cities with 1 as its starting position.

if (array[0]==1)

{

calculatecost(order,size);

if(debug\_flag==1)

cout<<"-------------------------------------------------";

}

return;

}

/\*

permute:

permute() is a recursive function generates permutations from an array order,

calling a swap function to swap positions of values in the array.

\*/

void permute(int k,int& size, int\* order){ //Recursive function that calls itself enough to calculate all possible routes (n-1)!

int i;

if (k==0)

filterArray(size,order);

else{

for (i=k-1;i>=0;i--){

swap(i,k-1);

permute(k-1,size,order);

swap(i,k-1);

}

}

return;

}

/\*

calculatecost:

This function calculates the cost for an inputted route array[]. The cost between cities are called for each position in the array, and the total cost is the sum of each position called, in addition to the distance between the final position and position [0]. The cost is compared to the smallest cost calculated - its stored if the cost is smaller, discarded if not. The order of the current optimal array is stored in order[n];

\*/

void calculatecost(int\* order, int& size){ //Calculates the cost of each possible route.

int k;

double cost=0;

int temporder[20];

int j; int i;

for (k=0;k<size; k++)

{

temporder[k]=array[k];

}

if (debug\_flag==1){

cout<<endl<<"The following is an order "<<": ";

for (k=0;k<size; k++)

{

cout<<temporder[k]<<" ";

}

cout<<1;

cout<<endl;

}

for(k=0;k<size-1;k++)

{

cost+=cities[temporder[k]-1][temporder[k+1]-1];

if (debug\_flag==1)

cout<<cities[temporder[k]-1][temporder[k+1]-1]<<" + ";

}

if (debug\_flag==1){

cout<<cities[array[size-1]-1][0]<<endl;

}

cost+=cities[array[size-1]-1][0];

if (cost<smallest)

{

smallest=cost;

for (k=0;k<size; k++)

{

order[k]=temporder[k];

}

}

if (debug\_flag==1){

cout<<endl<<"And its cost: "<<cost;

cout<<endl;

}}