Group 13
Avant Mathur
Tatyana Vlaskin
Nicholas Madani
CS325 Project 4

The Traveling Salesman Problem, deals with creating the ideal path that a salesman would take while traveling between cities. The solution to any given TSP would be the shortest way to visit a cpecific number of cities, vising each city only once and then returning to the starting point.

There are multiple optimal algorithms that exist for solving the TSP, but using most of them it is computationally infeasible to obtain the optimal solution to the TSP. Heuristics, instead of optimal algorithms, are extensively used to solve TSP solutions and provide solution that balances speed and accuracy. Most of the heuristics approximations algorithms create tour quickly, but not necessary optimally. Heuristics algorithms can be classified as:

1. Construction algorithm - tour is constructed by including points in the tour, usually one at a time, until a complete tour is developed

2. Improvements algorithms- a given initial solution is improved, if possible, by transposing 2 or more points in the initial tour

3. Hybrid algorithms – is a construction algorithms to obtain an initial solution and then improve it using an improvement algorithms.

Initially, we tried to solve this problem using Construction Algorithm. According to the Wikepedia, the Nearest Neighbor algorithm is the simplest and most straight forward TSP heuristics. The key to this algorithm is to always visit the nearest city, then return to the starting city when all the other cities are visited:

The pseudocode is:

****

Step 1: Select a random starting location - in our case it will the first city in the file

Step 2: Find the nearest unvisited city and go there

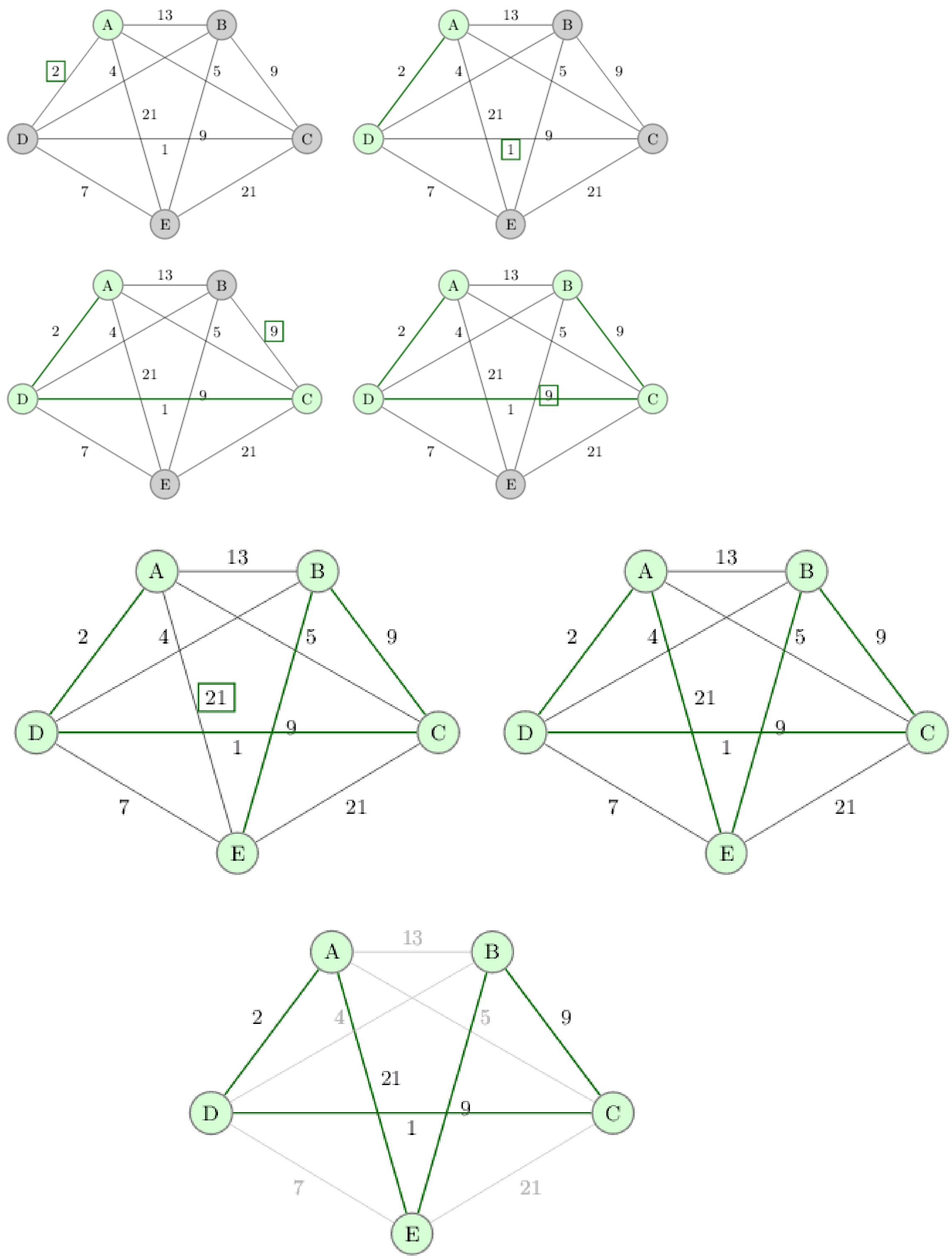
Step 3: Check if there are any unvisited cities left. If yes, repeat step 2

Step 4: Return the first city

This algorithm will quickly yield a short tour, but it rarely finds the most optimal solution.

http://cs.indstate.edu/~zeeshan/aman.pdf provides a very thorough step by step approximation solution by the Nearest Neighbor method. The solution presented in the paper is for 5 nodes. The tour is started with node A in that particular case and the nearest neighbor algorithm is applied.

The solution presented in the paper is provided below:



The total distance of the path A->D->C->B->E->A obtained is 2+1+8+8+21 =42. After we are done with node A, we iterate though each starting node until all nodes are analyzed. Thus, the time

complexity if O(n^2).

Using this example and the pseudocode presented above, we implemented, the Nearest Neighbor algorithm and tested it using the example cases and competition files.

The results that we got are presented below:

Test Case	Minimum tour	seconds to get a solution
tsp_example_1.txt	130921	0.01 seconds
tsp_example_2.txt	2975	0.75
tsp_example_3.txt	see comments below	see comments below

We tried to run tsp_example_3.txt tour, which has 15112 cities and after 3 hours, we gave up and terminated the run. So for this tour, we decided not to test all starting positions. This will be accomplished by setting a time restriction. The time restriction will be 299 seconds, which is almost 5 minutes. When the algorithm is started, the city at position 0, which is the first city listed in the file will be set as a starting position of the tour and the best order of the cities will be calculated using the nearest neighbor algorithm. If time allows [if the best sequence of cities for the first city will be found in less than 299 seconds, we will move on to city # 1 as the starting city and determine if it will give a better result.

Initially, we were planning to start at a random city and only optimize tour for that randomly selected starting city, but because for the competition part of the assignment, we have 5 minutes to get "optimal" result, so we decide to let tsp_example_3.txt tour to run for a longer time.

To impose a restriction, we created a variable duration1. At the beginning of the run, we start the clock and at the end of each iteration of the Nearest Neighbor algorithm, the duration of the run time is computed: duration1 = (clock() - begin) / (double) CLOCKS_PER_SEC; //end timer. If the duration of the run is less than < 300 seconds, we move on to the next starting city. However, if we start running the algorithm when the time elapsed is less than 300 seconds, it does not necessarily mean that it will be finished before 300 seconds, which was confirmed when we ran the competition tours. See the results below:

Test Case	Minimum tour	seconds to get a solution
tsp_example_1.txt	130921	0.01 seconds
tsp_example_2.txt	2975	0.75

tsp_example_3.txt	1924638	290.22

Test Case	Competition Min. Tour < 5 min.	Minimum Tour any time seconds
test-input-1.txt	5911	0.0001
test-input-2.txt	8011	0.03
test-input-3.txt	14826	0.56
test-input-4.txt	19711	4.47
test-input-5.txt	27128	35.54
test-input-6.txt	39469	285.82
test-input-7.txt	61947	300.81

The last tour completes in 300.81 seconds, which is over 5 minutes, so we need to impose a harsher restrictions on the time. Different times were tested and it was determined that optimal number is 288 seconds. If the time during the start of the iteration is 288 seconds, the algorithm will be able to complete iteration before reaching 300 seconds. The test files and the competition files were rerun with this restriction and we got the following results:

Test Case	Minimum tour	seconds to get a solution
tsp_example_1.txt	130921	0.01 seconds
tsp_example_2.txt	2975	0.75
tsp_example_3.txt	1924638	308.26

Test Case	Competition Min. Tour < 5 min.	Minimum Tour any time
test-input-1.txt	5911	0.0001 seconds
test-input-2.txt	8011	0.03
test-input-3.txt	14826	0.56
test-input-4.txt	19711	4.47
test-input-5.txt	27128	35.54
test-input-6.txt	39469	287.04
test-input-7.txt	61947	288.79

It seems that everything looks good and we could stop now but according to the assignment description, the optimal tour length for test cases 1, 2, and 3 are 108159, 2579 and 1573084,

respectively. Thus, solutions that we got using the Nearest Neighbor algorithms are not that great.

According to the references that we read [all the references provided below], for N cities randomly distributed on a plane, the algorithm on average yields a path 25% longer than the shortest possible path. However, there exist many specially arranged city distributions which make the Nearest Neighbor algorithm give the worst route. For this reason, we decided to take a look at the improvement algorithms and the one that we've chosen to implement is 2-opt algorithm. The 2-opt algorithm was first proposed by Croes, although the basic move has already been suggested by Flood.

The 1st step in the 2-opt algorithm is to generate an initial tour. We will be using the initial tour generated by the Nearest Neighbor algorithm and then run the optimization part of the 2-opt algorithm. The 2-opc algorithm work by taking 2 edges from the tour and doing a pairwise exchange. If the resulting solution is smaller than the initial solution, it is stored as the candidate for the future consideration. If not the new solution is discarded. Another way to say is when the new "optimal" solution is found, the old solution is discarded. This continues until all the pairwise exchanges are considered and the local min is reached. two edges from the tour, and reconnecting the two paths created. When the cities are swapped, the new distance is checked to see if it is lower than the old tour. If the new distance is lower, the new distance becomes the new tour. The algorithm continues until it iterates though all pairs and the local minimum is reached.

The pseudocode was taken from the following reference:

http://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf

****

```
do {
  minchange = 0;
  for (i = 0; i < cities-2; i++) {
    for (j = i+2; j < cities; j++) {
      change = dist(i,j) + dist(i+1,j+1)
        - dist(i,i+1) - dist(j,j+1);
      if (minchange > change) {
        minchange = change;
        mini = i;  minj = j;
  } } }
  // apply mini/minj move
} while (minchange < 0);
```

Distance matrix: $O(n^2)$ time and space

Doubly-nested for loop to visit $O(n^2)$ edge pairs

It suffices to compute change in length: O(1) time

O(n) iterations needed to reach local minimum: overall algorithm is $O(n^3)$

To get the solution to the TSP, we will first run the nearest city algorithm to generate the initial tour and then run the 2-opt algorithm to optimize the tour. The time restriction will be the same as above. While running the completion cases, it was determined that for tour #7, our algorithm does not find the "optimal" tour within 5 minutes, so we had to impose harsher restrictions. During the testing stage, it was determined that if the iteration of the algorithm starts when the time is 250 sec, the iteration will be complete before 5 minutes. We will try to run this for all possible starting cities if the run will be within our time limit, which we set to be 5 minutes. Results that we got:

Test Case	**Minimum tour**	**seconds to get a solution [seconds]**
tsp_example_1.txt	109975	0.24
tsp_example_2.txt	2669	13.1
tsp_example_3.txt	1696625	471.47

Test Case	**Competition Min. Tour < 5 min.**	**Minimum Tour any time[seconds]**
test-input-1.txt	5373	0.05
test-input-2.txt	7437	0.47
test-input-3.txt	12553	10.47
test-input-4.txt	17603	77.54
test-input-5.txt	24456	250.67
test-input-6.txt	34878	252.54
test-input-7.txt	54679	253.09

After we satisfied the requirements of the assignment, we decide to try to find the optimal solution of the TSP person. To solve the Travelling Salesman Problem (TSP) as accurately as possible, the solution used is the Dynamic Programming approach. For this project, there were several functions created to complete this TSP problem:

- main()
- printMatrix(adjacencyMatrix, numberOfVertices)
- getFileInfo(inputFile)
- getAdjMatrix(vectorOfVertices)
- getTSPTour(adjacencyMatrix, numberOfVertices, outputFile)
- nCr(n, r)
- createInfArray(size)

All of these functions and a structure (Vertex) was created. The struct Vertex contains three elements x, y, and name to represent a common Vertex placed on the graph that connects to all other nodes. A vertex can contain the integer point (x, y) of where it is located on a 2-dimensional plane and the string name (like A, B, C, 1, 2, 3, etc.).

For the main() function it must accept at least 2 arguments (the call to this tsp program, and an input file of which to read vertex information). If there are fewer than 2 arguments, the program will not run and state input file name is missing. When there are at least 2 arguments then the 2nd argument will be read in as a filename and if the file is valid then it continues on with the TSP problem otherwise it will error and say the file cannot be open. In a good filename, it will read all of the vertices from the file, create the adjacency matrix, get the TSP Tour and length information and then delete all resources used.

For the printMatrix() function, it accepts 2 parameter, an adjacency matrix and the number of vertices in this graph. This function is primarily optional and used to debug and make sure all of the computations of the distances are correct for the graph. It opens up an optional file outputMatrix.txt to write all values of the adjacency matrix into it.

For the getFileInfo() function it accepts 1 parameter, an already opened input file. It needs to create a set of vertices from the data it reads in. Using a while loop that loops through all information in the file, it creates a new vertex, reads in a grouping of data (name, x, y) or one line of the file and then if it is not a blank line of information, then it will push this vertex data onto the vector of Vertices. Once the while loop is completed then it returns that vector of Vertices.

For the getAdjMatrix() function is accepts 1 parameter, the vector of Vertices. It then creates an adjacency matrix of the size of the vector of Vertices and then goes through every row and column of this adjacency matrix. The adjacency matrix (i, j) pair represents the distance needed to travel from vertex i to vertex j. If i = j then the adjacency matrix value will be 0 because no distance travelled is needed to go from a vertex to itself. If i < j (upper triangular part of the matrix), then we use the distance formula between the two points using the formula: D=(x2-x1)2+(y2-y1)2 where (x_1, y_1) is vertex i and (x_2, y_2) is vertex j. And the lower triangular part where i > j (below the main diagonal) can do the same thing as the upper triangular part but a shortcut is that these values are already filled in before by the upper triangular part. So all we have to do is say that the (i, j) pair is filled in the location of the (j, i).

The helper function nCr() is an iterative version of the mathematical equation ${}_nC_r$ which is the following formula: nCr=n!(n-r)!r! . If it is rewritten so that it is in iterative form then it can shorten the amount of time needed to solve this combinations problem. The numerator would be the multiplication of values of: n, n-1, n-2, ..., n-r+1 and the denominator would be the multiplication of values of r, r-1, r-2, r-3, ..., 1. Divide the two together and that is the solution to this combinations problem which is a lot faster than trying to compute n!, (n - r)!, and r! together.

The helper function createInfArray() creates a matrix full of infinite (very large values) it creates an array of size, size, and then allocates to each location a large value like 99999. Once it finishes that it returns the array created.

The biggest part of this project is the getTSPTour() which has three parameters, the adjacency matrix, the size or the number of vertices in the graph, and an output file to place the solution of the length of the tour and the tour direction. There are a couple of data structures used here, a map (dictionary) which maps a set to an integer array and a set which contains all of the vertices that are included in the tour. The set is all of the vertices that are included already in this combination for the tour and it maps to an array of size (number of vertices) which contains a list of lengths given such that the i-th position in the array is the situation where the i-th vertex is used to get this tour. The starting condition (in the map) starts with vertex 0 with an array of values of 0 for position 0 and 99999 for all other positions using the createInfArray() function. Now the tour must go through including all vertices (s going from 1 to size of the graph) one time so that it creates a Hamiltonian Cycle which is one of the conditions needed for the TSP Problem to have a solution. Then it needs to create permutations of subsets of size 2 and larger to get the solution of this tour. One way would be create each set altogether which is the more inefficient way, or the other way would be to create bit permutations and convert them. So when s = 1 it will create all permutations of bits of size, size. If size = 7, s = 1 then it must create 1000000, 0100000, 0010000, 0001000, 0000100, 0000010, and 0000001. To do that it uses a bit wise operation using operators <<, piping, AND operations, and concatenating. Then once it creates the permutation then it attaches a 1 to the end because each tour must start with the first vertex (0). Then it will change this number from its binary form to set form. For example, a combination of 1001001 must output that vertices {0, 3, 6} which creates our subset combination for this situation. Then it goes through two for loops of subsets of adding new paths to this tour. The J iterator goes through all vertices of the subset that are not the first starting vertex (0) and the I iterator goes through all vertices that are not the J iterator. It creates a new set removing the J iterator's vertex from the set as the vertex chosen to create this new tour and minimizes the currently best solution path of mapSet(vSet → array[J]) such that each mapSet(vSet - {J} + d_{IJ} → array[I]). Once all of these subsets are considered then it will start with the last set S = {0, 1, 2, 3 ... size} and then look at all of the choices. Whichever has the smallest J such that mapSet(vSet → array[J]) + d_{J0} is minimized which is to complete the cycle going to vertex 0, this is the solution to the TSP problem. To calculate the tour itself, the problem works backwards from S = {0, 1, 2, 3, ... size} down to an empty vertex. The pivot is the current vertex the program is on right now. For the current S that it is on, it finds the smallest J such that mapSet(vSet → array[J]) + $d_{J,pivot}$ is minimized. Once the J is found then J becomes part of the tour in that order and then the pivot is changed to J and continues on.

<u>Algorithm Analysis</u>

For the printMatrix function, there are two loops here that run for the size of vertices times the size of vertices in the graph, N. Therefore this function runs in $O(N^2)$ time.

The getFileInfo function has one loop that runs through all vertices in the graph N. Therefore this function takes O(N) time.

The getAdjMatrix function has two loops that run for the size of vertices times the size of vertices in the graph, N. Thus this function runs in $O(N^2)$ time.

The function nCr has two side-by-side loops, one for computing the multiplication of n to n-r+1 and one for computing the multiplication of r to 1. Therefore at max n and r = 0 it will have to do N operations which takes O(N) time.

The function createInfArray has only one single loop that goes through the size of the number of vertices in the graph N. Thus it runs in O(N) time.

The largest function getTSPTour has two main for loops. One that goes through N vertices to reach the end for the Hamiltonian Cycle. Inside of that there is a loop to go through all possible subsets of S to go through all possible combinations for this problem. The operations with each subset requires going through calculating nCr which takes O(N) time, converting the binary permutation to the subset which takes O(log N) time because it is a problem of recurrence relation T(n) = T(n/2) + c which under the Master Theorem of O(log N) and then going through removing one vertex for each linear combination which is N choices. Therefore this inner loop runs at $O(2^{N+\log N+N}) = O(2^{2N})$ which is really $O(2^N)$ for the inner-most loop, O(N) for the outer loop so the total run time for calculating the minimum TSP tour is $O(N2^N)$. After that calculation then it looks at all possible combinations for S = all vertices. This takes N calculations to solve because each current path to the starting vertex must be found for the tour. Finally to calculate the tour it takes N nodes to go through each situation parsing the Hamiltonian Cycle one node at a time. So this operation will take O(N) total time. Which means the total time used is $O(N) + O(N2^N) + O(N) = O(N2^N + 2N) = O(N2^N)$.

One of the main advantages with using the Dynamic Programming version of the Travelling Salesman Problem is that it can accurately solve the problem correctly. The downside to this Dynamic Programming version is that of its run time O(N * 2^N). We couldn't get these larger cities to run fully for N >= 50 because there are so many combinations to process that it would take days, months, even years to solve this problem. If a faster approach (like Nearest Neighbour is used yes it could finish the problem in a relatively short amount of time. But there are many instances where the Nearest Neibour Approach could give an answer that is way off the mark even if on paper it is easy to tell what the true answer is supposed to be. If we have a short tour and accuracy is important, then we can go ahread and use the dynamic algorithm. However, when we are dealing with large number of cities, then we need to use approximation algorithms that we used to get solutions to the test files for this assignment.

Dynamic Algorithm Code:

main.cpp

```
// The #include files for this project.
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
```

```
#include <vector>
#include <set>
#include <map>
#include <cmath>

using namespace std;

// Structure for a Vertex.
// Contains a name and a position (x, y).
struct Vertex
{
	int x, y;
	string name;
};

// DEBUG FLAGS
// These are used if you want to print out specific information to the screen/file
// Set all of these to FALSE if you want them to not display.
const bool DEBUG_FLAG_A = true;
const bool DEBUG_FLAG_ADJ_MAT = true;

// Function Headings
void printMatrix(int **adjMatrix, int numV);
vector<Vertex *>* getFileInfo(ifstream &input);
int** getAdjMatrix(vector<Vertex *> *v);
void getTSPTour(int **adjMatrix, int size, string outputFile);
long long nCr(int n, int r);
int* createInfArray(int size);

int main(int argc, char *argv[])
{
	// If there is only one argument, the file input is missing.
	if (argc < 2)
		cout << "Input File Name Missing..." << endl;
	else
	{
		// Open up the 2nd argument file as an input file.
		ifstream input;
		input.open(argv[1], ios::in);

		// If the file can be opened.
		if (!input.fail())
		{
			// Get a list of vertices from this file in argv[1].
			vector<Vertex *>* vertices = getFileInfo(input);

			// Get an adjacency matrix of vertices from this file.
			int **adjMatrix = getAdjMatrix(vertices);

			// Print this matrix to outputMatrix.txt if needed.
			if (DEBUG_FLAG_ADJ_MAT)
				printMatrix(adjMatrix, vertices->size());

			// Compute the TSP Solution to this set of
vertices/adjacency matrix.
			getTSPTour(adjMatrix, vertices->size(), argv[1]);
```

```
                        // Delete all information for vertices and adjacency
matrix.
                        for (unsigned int i = 0; i < vertices->size(); i++)
                        {
                                delete[] adjMatrix[i];
                                delete vertices->at(i);
                        }
                        delete vertices;
                        delete[] adjMatrix;
                }
                // If the file cannot be open, report to the screen.
                else
                        cout << "ERROR: Cannot Open File." << endl;
        }

        return 0;
}

// Function: PrintMatrix()
// Takes an adjacency matrix and number of vertices (numV) and prints it to
outputMatrix.txt
// OPTIONAL: Used only when the debug flag DEBUG_FLAG_ADJ_MAT is set to true.
void printMatrix(int **adjMatrix, int numV)
{
        // Open outputMatrix.txt for writing.
        ofstream output;
        output.open("outputMatrix.txt");

        // Go through each vertex row.
        for (int i = 0; i < numV; i++)
        {
                // Go through each vertex column.
                for (int j = 0; j < numV; j++)
                        // Print out each matrix element.
                        output << setw(4) << adjMatrix[i][j] << " ";
                output << endl;
        }

        // Close the file.
        output.close();
}

// Function: GetFileInfo()
// Takes a file, that is openable and gets all vertices from the file.
// Returns a vector of all vertices from the file.
vector<Vertex *>* getFileInfo(ifstream &input)
{
        // Create a new vector of vertices.
        vector<Vertex *> *vertices = new vector<Vertex *>();

        // While there are still vertices to read from the file.
        while (!input.eof())
        {
                // Create a new vertex.
                Vertex *newV = new Vertex();

                // Read information for this vertex, name, x, and y.
```

```
                input >> newV->name >> newV->x >> newV->y;

                // If this is not a blank line, then push this vertex onto the
vector.
                if (newV->name != "")
                        vertices->push_back(newV);
        }

        // Return the vector of vertices.
        return vertices;
}

// Function: GetAdjMatrix()
// Takes a set of vertices and computes the Adjacency Matrix for these vertices.
// Uses the Distance formula to find the distance between any two vertices
(rounded).
int** getAdjMatrix(vector<Vertex *> *v)
{
        // Get the number of vertices in this graph.
        int size = v->size();

        // Create a new adjacency matrix of size 2x2.
        int **adjMatrix = new int*[size];

        // For each vertex row.
        for (int i = 0; i < size; i++)
        {
                // Create a column for that vertex row i.
                adjMatrix[i] = new int[size];

                // Go through all columns for that vertex row i.
                for (int j = 0; j < size; j++)
                {
                        // If i and j match, then it takes no distance to reach
itself.
                        if (i == j)
                                adjMatrix[i][j] = 0;
                        // If i > j (lower part of the matrix) then we computed this
already use the adj[j][i]
                        // position to retrieve that value quickly.
                        else if (i > j)
                                adjMatrix[i][j] = adjMatrix[j][i];
                        else
                        {
                                // Otherwise, calculate the distance using the
distance formula.
                                // D = SQRT((y2 - y1)^2 + (x2 - x1)^2)
                                int xDiff = v->at(i)->x - v->at(j)->x;
                                int yDiff = v->at(i)->y - v->at(j)->y;
                                adjMatrix[i][j] = (int)round(sqrt(xDiff * xDiff +
yDiff * yDiff));
                        }
                }
        }

        // Return this adjacency matrix to the caller.
        return adjMatrix;
```

```
}

// Function: GetTSPTour()
// Creates a TSP Tour through the entire set of vertices for this set of vertices
// with this adjacency matrix.  Solves this by finding the accurate minimum
distance
// and the actual tour solving it backwards direction.
void getTSPTour(int **adjMatrix, int size, string outputFile)
{
	// Create a dictionary (map) of sets -> array
	// A set represents the combination of vertices incorporated so far.
	// The array represents the minimal distances for that particular set with
that index used.
	map<set<int>, int *> mapSet;

	// Start with only {0} as a set.
	set<int> one;
	one.insert(0);

	// Create an infinite array with values, 0 in the first location when it
is used.
	int *oneClear = createInfArray(size);
	oneClear[0] = 0;

	// Insert this {0} -> [0, 99999, 99999, ...., 9999] item into the
dictionary.
	mapSet.insert(std::pair<set<int>, int *>(one, oneClear));

	// Go through size number of steps for this problem since it can only go
through each
	// vertex one time.
	for (int s = 1; s < size; s++)
	{
		// Make permuations of vertex groupings (subsets) so that we can
go through
		// Each possible combination in an efficient way.
		long long w = (long long)pow(2, s) - 1;

		// Calculate the number of combinations needed (nCr) for this size
s.
		long long it = nCr(size - 1, s);

		// Go through all subsets of size s
		for (int i = 0; i < it; i++)
		{
			// Create a temporary set vSet which will be all vertices
used with this permutation.
			// A permutation of:   10011 uses vertices 0, 1, and 4.
			// A permutation of: 1001001 uses vertices 0, 3, and 6.
			set<int> vSet;

			// Change the permutation into vertex form.  Uses a binary
conversion to solve this
			// part.  This is to avoid using up more time than needed
for this solution.
			// Val is the binary combination including the first node
for every combination.
```

```
                        long long max = (long long)pow(2, size), val = (w << 1) +
1;
                        int top = size;

                        // While the binary number to process (val) still is larger
than 0.
                        // We continue to process.
                        while (val > 0)
                        {
                                // If the number is divisible by max then this
vertex is used (a 1).
                                // If the number is not divisible then it is a 0
for this location.
                                if (val / max > 0)
                                {
                                        // Insert into vSet and decrease the value
of val by max.
                                        vSet.insert(top);
                                        val = val % max;
                                }
                                // Cut the divisor by 2 and decrease the vertex num
by 1.
                                max = max / 2;
                                top = top - 1;
                        }

                        // Create a new set of information for this vSet created
by the binary converter.
                        mapSet.insert(std::pair<set<int>, int *>(vSet,
createInfArray(size)));

                        // Go through each vertex of that new set created.
                        for (set<int>::iterator iterJ = vSet.begin(); iterJ !=
vSet.end(); ++iterJ)
                        {
                                // If the vertex of the set is not the first, then
we continue on.
                                if (*iterJ != 0)
                                {
                                        // Set the minimum value to be 99999 (max).
                                        long long minVal = 99999;

                                        // Go through all other vertices of this
set.
                                        for (set<int>::iterator iterI =
vSet.begin(); iterI != vSet.end(); ++iterI)
                                        {
                                                // If the J iterator is not the same
as the I iterator, then do a min function.
                                                if (*iterJ != *iterI)
                                                {
                                                        // Create a new set of
vertices from vSet.
                                                        set<int> newSet = vSet;

                                                        // Remove the vertex J from
vSet (this is when J is used).
```

```
                                                        newSet.erase(*iterJ);

                                                        // See if theis new path
using J from this grouping is better.
                                                        long long val =
mapSet[newSet][*iterI] + adjMatrix[*iterI][*iterJ];

                                                        // If it is then update.
                                                        if (val < minVal)
                                                                minVal = val;
                                                }
                                        }

                                        // Make this minVal the best value for this
set of vertices situation.
                                        mapSet[vSet][*iterJ] = minVal;
                                }
                        }

                        // If w is not 0 (this never reaches 0 so we don't have to
worry, this is just for
                        // the first case when s = 1.
                        if (w != 0)
                        {
                                // This section gets the next appropriate
combination of s bits in a permutation.
                                // Uses a lot of bit wise actions to do that.
                                long long v = w;
                                long long t = (v | (v - 1)) + 1;
                                w = t | ((((t & -t) / (v & -v)) >> 1) - 1);
                        }
                }

                // Print out this to note when s bits are done.
                if (DEBUG_FLAG_A)
                        cout << "DONE, s = " << s << endl;
        }

        // Start minimum value at 99999.
        // This is to find the actual minDistance for this TSP Tour.
        long long minVal = 99999;
        set<int> vSet;

        // Start with a set containing all vertices.
        for (int i = 0; i < size; i++)
                vSet.insert(i);

        // Go through each possible choice of all vertices.
        // See whcih direction to go to minimize the distance used.
        for (set<int>::iterator iterJ = vSet.begin(); iterJ != vSet.end();
++iterJ)
        {
                if (mapSet[vSet][*iterJ] + adjMatrix[*iterJ][0] < minVal)
                        minVal = mapSet[vSet][*iterJ] + adjMatrix[*iterJ][0];
        }

        // Print out to the file output.txt.
```

```
        ofstream output;
        output.open((outputFile+".tour").c_str(), ios::out);

        // Output the minVal (distance for this TSP).
        output << minVal << endl;

        // This is to calculate the tour of the TSP starting from the end and working
backwards.
        // Pivot starts at 0.
        int pivot = 0;

        // While there are vertices to go through (vSet)
        while (vSet.size() > 0)
        {
                // Start with minVal = 99999.
                int pos;
                minVal = 99999;

                // Go through all vertices in vSet currently.
                for (set<int>::iterator iterJ = vSet.begin(); iterJ != vSet.end();
++iterJ)
                {
                        // See if when we remove iterJ does using the edge from pivot
to iterJ create a min
                        // distance.  If so this is the best path.  Otherwise
nothing.
                        if (mapSet[vSet][*iterJ] + adjMatrix[*iterJ][pivot] <
minVal)
                        {
                                // Remember the minimum distance used and the
vertex used to get there.
                                minVal = mapSet[vSet][*iterJ] +
adjMatrix[*iterJ][pivot];
                                pos = *iterJ;
                        }
                }

                // Remove this edge's vertex used (pos).
                vSet.erase(pos);

                // Create this as a new pivot (pos).
                pivot = pos;

                // Output that we used this vertex now.
                output << pos << endl;
        }

        // Close the output file.
        output.close();

        // Free all memory from the map.
        for (map<set<int>, int *>::iterator iter = mapSet.begin(); iter !=
mapSet.end(); ++iter)
                delete[] iter->second;
}

// Function: nCr()
```

```
// Computes nCr using linear time.
// nCr = n! / (n - r)!r! = (n * n-1 * n-2 * ... * n-r+1) / (r * r-1 * r-2 * ...
* 1)
long long nCr(int n, int r)
{
	// Calculate the numerator and denominators.
	long long numer = 1, denom = 1;

	// Calculate the numerator: n * n-1 * n-2 * ... * n-r+1
	for (int i = n; i >= n - r + 1; i--)
		numer *= i;

	// Calculate the denominator: r * r-1 * r-2 * ... * 1
	for (int i = 1; i <= r; i++)
		denom *= i;

	// Return the quotient of the numerator / denominator.
	return (numer / denom);
}

// Function: CreateInfArray()
// Create a new array of values such that every value is initialized to a very
large number
// Used to find the correct minDistance.
int* createInfArray(int size)
{
	// Create a new array of size, size.
	int *newArr = new int[size];

	// Go through each element in the array and set to 99999 (large number).
	for (int i = 0; i < size; i++)
		newArr[i] = 99999;

	// Return this new infinite array.
	return newArr;
}
```

****

Adopted from the following references:

http://konferenca.unishk.edu.al/icrae2013/icraecd2013/doc/2070.pdf

https://github.com/Lagrewj/CS325_Project4/blob/master/tsp.cpp

https://github.com/samlbest/traveling-salesman/blob/master/algorithms.cpp

https://github.com/CaiqueReinhold/STD_GA_TSP/blob/17004e42c85db3701546708936a0af66c399c471/GA/Cities.cpp

http://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

http://en.wikipedia.org/wiki/2-opt

https://github.com/caseybalza/cs325-project4/blob/master/algorithms.cpp

https://github.com/ethanspiro/traveling-salesman/blob/master/cs325_project4_tsp.cpp

http://cs.indstate.edu/~zeeshan/aman.pdf

http://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf

http://www.technical-recipes.com/2012/applying-c-implementations-of-2-opt-to-travelling-salesman-problems/

http://en.wikipedia.org/wiki/Travelling_salesman_problem

http://bardzo.be/0sem/NAI/rozne/Comparison%20of%20TSP%20Algorithms/Comparison%20of%20TSP%20Algorithms.PDF

http://konferenca.unishk.edu.al/icrae2013/icraecd2013/doc/2070.pdf