CS325 - Project 3: The Travelling Salesman Problem (TSP)
Group 17
Members:

· Michael Lewis
· Patrick Mullaney
· Matt Nutsch

Date: 6-9-2017

Summary of problem/assignment:
For Project 3, we are attempting to research potential algorithms that may provide a solution to the Traveling Salesperson Problem (TSP). For this assignment, we must research potential solutions to the Traveling Salesperson Problem. We must then implement our best algorithm that correctly solves the problem most efficiently. To quantify this, will be given 10 instances of the problem. Three example instances will be provided with corresponding input cases and optimal tour lengths. It is required that the ratio of our solution/optimal <= 1.25, in an unlimited amount of time. Solutions must be verified using the provided tsp-verifier.py testing procedure. An additional seven instances will be provided as part of a class competition. We will also use these cases for testing our algorithm's ability to find an optimal solution in a minimal time frame.

Algorithms researched:
1. Naïve Algorithm
2. Nearest Neighbor Algorithm

2B. Repetitive Nearest Neighbor Algorithm

3. Cheapest Link Algorithm
4. 2-Optimal Algorithm

Competition program:
The program we are submitting uses the 2-Opt Algorithm for inputs of sizes no greater than 500, and the Repetitive Nearest Neighbor Algorithm for any input size greater than 500. Results of all competition input, for the 3-minute time limit, are included in the Results section. Results of all example and competition input, for an unlimited time limit, are also included in the results section. Additionally, tables including results and data from each algorithm are provided toward the end of this report. For reference, the Concorde TSP solver (Cook) from the University of Waterloo was used to determine optimal results for the competition files, for comparison. This data is also included in the tables for each algorithm.

References:
Several references were used for the purposes of our research. These are cited in the Works Cited section at the end of this report.

Algorithm 1: Naïve Algorithm

Synopsis:
This algorithm is labeled naïve, as there is no real strategy, other than visiting all vertices. We simply visit each city in the graph, traveling in the order provided in the input file. Once the last city is visited, the we return to the initial city to complete the tour.

Discussion:
This approach may be one of the simplest solutions one could implement, as it only requires following the ordering from the input file. Even though it is likely that this naïve approach could compete with an algorithm that attempts to employ a strategy such as approximating a solution or using a heuristic, it is useful in that it demonstrates a workable, albeit non-optimal solution, which runs in polynomial time. The optimality of this solution will depend on how the cities had been ordered in the original input file. For example, if the cities in the original input file are already listed in an optimal ordering, then this algorithm will produce an optimal solution in polynomial time. To get a result, the naïve method simply calculates the sum of the distance between each city in the input file, as well as the distance from the last city to the initial city. But if any cities are not in the optimal order, then the algorithm will not have an optimal solution, and such sub-optimality would depend on how many cities are not placed in their optimal order, and how far they are from their optimal order.

Pseudocode:

```
naiveAlgorithm(cities[])
        path[]
        import input file into cities[]  // each city has an id and xy coordinates
        distanceTotal = 0
        prevCity = currCity = cities[0]
        path[0] = currCity
        addPath(prevCity)
        for i = 1 to size(cities[])
                currCity = cities[i]
                distance_X = currCity[‘x’] – prevCity[‘x’]
                distance_Y = currCity[‘y’] – prevCity[‘y’]
                distance = sqrt((distance_X)^2 + (distance_Y)^2)
                distanceTotal += distance
                path[i] = currCity
                prevCity = currCity

        */ Finds distance from last city to first city and adds to total/*
        currCity = cities[0]
        distance_X = prevCity[‘x’] – currCity[‘x’]
        distance_Y = prevCity[‘y’] – currCity[‘y’]
```

```
distance = sqrt((distance_X)^2 + (distance_Y)^2)
distanceTotal += distance

return distanceTotal
```

Algorithm 2: Nearest Neighbor Algorithm

Synopsis:
The premise of the Nearest Neighbor Algorithm is essentially a “greedy” approach. The basis is the salesman starts at a random city and at each city, the salesman visits its closest neighbor until all cities have been visiting. The algorithm essentially trades correctness for speed in the sense that it usually produces a short tour quickly, but it is known to usually not be the optimal tour.

Discussion:
Interestingly for an algorithm of this nature, it’s believed you can gauge the correctness of the result based on the last stages of the tour. If the length of the last stages are relatively close to the first stages of the tour, the resulting tour can be assumed to be reasonable, however if there is a large discrepancy, the assumption is that there are better tours.

Pseudocode:
*Researched/modeled after wikipedia with modifications.

```
nearestNeighbor(cities[])

    /* Start at a random city, initialize length of tour to 0 and declare path array. */
    Current = cities[random];
    Path[];
    Path distance = 0;
    Current.visited = true;
    Path[0] = current;

    /* Find the nearest city (shortest path between current and any unvisited city, set
    Current city to nearest neighbor, mark visited, and repeat until all cities are visited */
    while(cities are unvisited)
        /* Initialize nearest to arbitrarily large number.*/
        Int nearest = infinity;

        for(i = 0, i < cities.size)
            /* Calculate distance between cities */
            Int dist = nearestInt(current, cities[i])

            /* If dist is less than current nearest, update until we have shortest edge*/
            if(dist < nearest)
                Nearest = dist;
                Next city = cities[i]

        /* End for loop, update next, mark visited, add distance, add next city to path. */
```

```
        Current = next city;
        Current.visited = true;
        Path distance += nearest;
        addPath(current);
    /* End while loop */
```

***Algorithm 2B: Repetitive Nearest Neighbor (DP):** Researched this along with nearest neighbor and essentially this is a modification of Nearest Neighbor where the the Nearest Neighbor algorithm is run on each city and the algorithm returns the shortest of all tours. The implementation improved the variance between Nearest Neighbor's solution and the optimal solution, however runtime increased exponentially. To address this, I modified the algorithm, retaining the greedy nature of the nearest neighbor, but instead implementing a dynamic programming table to avoid recalculating distances. This resulted in a significant reduction in runtime, but retained the better ratio to optimal solutions of the original algorithm.

Additionally, to address runtime performance on larger inputs, I further modified the algorithm. Instead of running nearest neighbor on all cities for larger inputs, the algorithm is instead run on a percentage of the cities. This method combines the speed of nearest neighbor and the attempt to get closer to the optimal solution with repetitive nearest neighbor, however avoids the overhead of running nearest neighbor on all cities for larger inputs. We can still hope to achieve a better result by running nearest neighbor on a random sampling than running on just one city.

Pseudocode:

```
repetitveNearestNeighbor(cities[])

    Tours[];
    int** table; // Table to track distance calculations.

    /* Run nearest neighbor on all cities, track distances as they are calculated. */
    if(cities.size < large input)
        for(i = 0 to cities.size)
            Tour t = nearestNeighbor(table);
            addTour(t);
    else(input is large)
        Int r = random number 0-9;
        Int s = cities.size/100;

        for(int i = r to cities.size, i+s)
            Tour t = nearestNeighbor(table);
            addTour(t);

    /* Initialize to arbitrarily large number. */
    Int shortest = infinity;
```

```
/* Iterate through all tours to find shortest. */
for(i = 0 to tours.size)
        /* If tour length is shorter than current shortest, update. */
        if(tour[i].distance < shortest)
                Shortest = tour[i].distance;
                t = tour[i];

Return tour t;
```

Algorithm 3:Cheapest Link Algorithm

Synopsis:
Cheapest Link is an algorithm which attempts to find the shortest by sequentially adding the shortest edges in the graph. The rationale is that by taking the shortest graph edges possible, then we get a reasonably short distance every time.

Discussion:
We decided to try this algorithm after researching Google Maps. Google Maps actually uses numerous algorithms in parallel to find the best path to a destination. However "Cheapest Arc", a derivative of Cheapest Link was the first algorithm mentioned in Google's documentation.

Google's documentation turned out to not be that helpful in writing the code, other than providing inspiration and direction to the algorithm. Online lecture videos about the Cheapest Link algorithm provided clearer explanations on how to implement it. In particular the online video "Cheapest Link Algorithm" by MSLT Mathematics was very helpful in explaining the algorithm.

One complication that created longer run times for this algorithm is that in order to perform its function, the algorithm must first create a list of all possible edges in the graph. This was necessary, because the input data consisted only of vertices (cities) and the Cheapest Link algorithm looks at edges. The added run time of calculating the edges before beginning on identifying the best path put this algorithm at a disadvantage.

Running the algorithm multiple times did not have an effect on the average distance results. The algorithm follows a very deterministic approach to finding a recommended path. As a result, the algorithm always finds the same path (and distance) when presented with the same input.

Pseudocode:

```
Main code
{
//Create a list of edges from the list of cities
For every city loop
        For every city loop again
                Add an edge to a vector containing the list of edges
                Calculate the length for the edge (distance between origin city and destination
city)
                Skip items where the origin and the destination city are the same

Sort the list of edges by length in ascending order
```

//Select the edges to use
Loop
    Sequentially examine each available edge, starting with the shortest length
    Skip edges where the origin city has already been used.
    Skip edges where the destination city has already been used.
    Skip edges where we already used the inverse of the edge (where origin and destination are swapped).
    Skip edges which would make a circuit too early.
    Break the loop if we reach a Hamiltonian circuit, or if we run out of edges
    Otherwise add the current edge to a list of edges to use

Organize the selected selected edges into a cohesive path, based on the origin city of each each edge and destination city of the predecessor edge. (loop)

Calculate the total distance and copy the values to the output. (loop)
}

distanceFromCL //utiltity function for finding the distance between two cities
{ Given two sets of x and y coordinates, find the distance between them }

compareByLength() //utility function used in sorting
{ return a value indicating if the length of the first parameter is less than the length of the second parameter }

checkIfACircuitFormed() //function to check if adding an edge would create a premature circuit, a Hamilton circuit, or no circuit
{
    Check if the parameter edge's destination is already in the list

    If so, then loop for the number of edges already selected
        Iterate to the next edge based on the destination of the previous edge
        If the iterator's destination = parameter edge's destination AND the number of edges selected matches the number of cities, then return that we found a Hamiltonian circuit.
        Else If the iterator's destination = parameter edge's destination AND the number of edges selected does NOT match the number of cities, then return that we found a premature circuit.

    If we finish the loop without already returning a value, then return that adding the parameter edge would not create a circuit.

}

Algorithm 4: 2-Optimal Algorithm

Synopsis:
2-Optimal (2-Opt) is a heuristic approach that attempts to find an improved solution to TSP through removing edges that cross over a graph by reordering vertices. If the algorithm can find a shorter path by deleting two longer edges and reconnecting the corresponding vertices with shorter edges, it will improve the tour's distance. The algorithm uses a 2-Opt swap function to test for this possibility, and when applicable, swaps two vertices at a time so that they are connected, eliminating the edge that had previously crossed over the graph. In order to be able to optimize the path, the 2-Opt algorithm must first create an initial, arbitrary tour (Webb). For this, we reused code from our Nearest Neighbor Algorithm. The algorithm then tests combinations of two edges to determine if any combination can be removed and replaced with a better solution.

Discussion:
We decided to give this algorithm further consideration after reading about it in the *Travelling Salesperson Problem* article on Wikipedia, in which it is also referred to as the "pairwise exchange." Implementing this algorithm would obviously be more challenging than the Naïve Algorithm, but did appear to be as daunting as other options such as the Simulated Annealing or Neural Network options that were introduced in the same article. Furthermore, our research indicated that the 2-Opt heuristic could yield results that were a significant improvement over other options. The C++ Implementation of 2-opt to the 'Att48' Travelling Salesman Problem article proved highly valuable for our implementation, as cited in our source code. The "ATT48" is a commonly used input set of 48 cities, and while that input was not used in our competition, the strategy and examples provided in the article were still wholly applicable. After successfully implementing the algorithm, the promises of efficiency were realized, as it gave superior results to our other implementations. The downside is that the algorithm's speed visibly diminishes as input sizes grow large. For this reason, we were only able to use it for four of our competition instances.

Pseudocode:
```
twoOpt(cities[])
        for i = 0 to totalVertices
                tourLength = seededTour(adjMatrix, tour_order, totalVertices, i)
                twoOptSearch(adjMatrix, tour_order, tourLength, fnameOutput, totalVertices)
                if tourLength < best_tourLength
                        bestTourLength = tourLength
                        write to file
        return bestTourLength

/* Converts data from input file into adjacency matrix */
makeAdjMatrix(readFile, totalVertices)
```

```
        scan every three numbers of file (identifier, x vertex, y vertex)
        for i = 0 to totalVertices
                for j = 0 to totalVertices
                        adjMatrix[i][j] = distForm(x_Vertices[i], y_Vertices[i],
                        x_Vertices[j], y_Vertices[j])
                return adjMatrix

/* Calculates distance between each city in tour, returns rounded value */
distForm(x1, y1, x2, y2)
        xDist = x2 - x1
        yDist = y2 - y1
        distance = sqrt(xDist * xDist + yDist * yDist)
        round distance
        return distance

/* Searches through all vertices for pairs of edges to swap if better length */
twoOptSearch(adjMatrix, tour, tourLength, fnameOutput, totalVertices)
        better = true
        prevLength = tourLength
        while better
                better = false
                for i = 1 to totalVertices - 1 and searchNotComplete
                        for j = i + 1 to totalVertices and searchNotComplete
                                prevLength = tourLength
                                swapEfficient(adjMatrix, tour, tourLength, totalVertices, i, j)
                                if tourLength < prevLength
                                        /* New length is better than previous */
                                        better = true
                                        searchComplete = true
                                        write to file

/* Builds new tours to swap vertexA and vertex in the route order to find a more efficient route */
twoOptSwap(newTour, tour, totalVertices, vertexA, vertexB)
        nodeMin = MIN(vertexA, vertexB)
        nodeMax = MAX(vertexA, vertexB)
        indexNew = 0

        /* Builds newTour as result of tour[0] through tour[nodeMin - 1] */
        for i = 0 to i nodeMin
                newTour[indexNew] = tour[i]
                increment indexNew

        /* Builds newTour as result of tour[nodeMax], descending through tour[nodeMin] */
```

```
        for i = nodeMax to nodeMin
                newTour[indexNew] = tour[i]
                increment indexNew

        /* Builds newTour as result of tour[nodeMax + 1] through tour[totalVertices] */
        for i = nodeMax + 1 to totalVertices
                newTour[indexNew] = tour[i]
                increment indexNew

/* Performs swap of vertexA and vertexB in route order, if it produces a more optimal result */
swapEfficient(adjMatrix, tour, tourLength, totalVertices, vertexA, vertexB)
        nodeMin = MIN(vertexA, vertexB)
        nodeMax = MAX(vertexA, vertexB)
        indexNew = 0
        lengthRemoved = lengthAdded = 0
        newTour[totalVertices]

        /* Replaces removed distances with added distances */
        if nodeMax + 1 < totalVertices
                lengthRemoved = adjMatrix[tour[nodeMin-1]][tour[nodeMin]] +
                        adjMatrix[tour[nodeMax]][tour[nodeMax + 1]]
                lengthAdded = adjMatrix[tour[nodeMin-1]][tour[nodeMax]] +
                        adjMatrix[tour[nodeMin]][tour[nodeMax + 1]]
        /* Handles condition in which tour wraps around to beginning city */
        else
                lengthRemoved = adjMatrix[tour[nodeMin-1]][tour[nodeMin]] +
                        adjMatrix[tour[nodeMax]][tour[0]]
                lengthAdded = adjMatrix[tour[nodeMin-1]][tour[nodeMax]] +
                        adjMatrix[tour[nodeMin]][tour[0]]

        tourDiff = lengthRemoved - lengthAdded

        /* Recalculates tour if shorter length is found */
        if tourDiff > 0
                /* Builds newTour as result of tour[0] through tour[nodeMin - 1] */
                for i = 0 to nodeMin - 1
                        newTour[indexNew] = tour[i]
                        increment indexNew

                /* Builds newTour as result of tour[nodeMax], descending through tour[nodeMin]
        */
                for i = nodeMax to nodeMin
                        newTour[indexNew] = tour[i]
```

```
                increment indexNew

            /* Builds newTour as result of tour[nodeMax + 1] through tour[totalVertices] */
            for i = nodeMax + 1 to totalVertices - 1
                newTour[indexNew] = tour[i]
                increment indexNew

            replace tour with newTour
            tourLength = tourLength - tourDiff

/* Builds optimized tour by tracking cities visited and each nearestCity to update tourLength */
seededTour(adjMatrix, tour, totalVertices, cityZero)
    tourLength = 0
    cityVisited[totalVertices]
    cityNumber = 0
    prevCity = 0

    nearestDist = infinity
    nearestCity = totalVertices

    /* Initializes cities in cityVisited array to 0 */
    for i = 0 to totalVertices
        cityVisited[i] = 0

    /* Accounts for start city as visited in tour by assigning = 1*/
    cityVisited[cityZero] = 1
    tour[0] = cityZero
    cityNumber = 1

    /* Determines if starting at tour's previous city improves result */
    while cityNumber < totalVertices)
        prevCity = tour[cityNumber - 1]
        for k = 0 to totalVertices - 1
            if cityVisited[k] == false and adjMatrix[prevCity][k] < nearestDist
                nearestDist = adjMatrix[prevCity][k]
                nearestCity = k

        /* Adds nearest city to tour */
        tour[cityNumber] = nearestCity
        tourLength = tourLength + nearestDist
        cityVisited[nearestCity] = 1
        increment cityNumber
```

```
/* Updates nearestDist and nearestCity */
        nearestDist = infinity
        nearestCity = totalVertices

/* Adds return distance from last city back to starting city to tourLength */
tourLength = tourLength + adjMatrix[tour[0]][tour[totalVertices-1]]

return tourLength
```

Best Tours and Times

Best tours and times for the three example instances:

Best tours with time lengths for the three example instances:				
Input	**Algorithm used**	**Length**	**Ratio**	**Runtime**
tsp_example_1.txt	2-Opt	109067	1.008	0.211 seconds
tsp_example_2.txt	2-Opt	2654	1.029	9.213 seconds
tsp_example_3.txt	Repetitive Nearest Neighbor	1918173	1.219	1311.359 seconds

Best tours with time lengths for the competition instances - 3 minute limit:			
Input	**Algorithm used**	**Length**	**Runtime**
test-input-1.txt	2-Opt	5373	3.178 seconds
test-input-2.txt	2-Opt	7487	3.410 seconds
test-input-3.txt	2-Opt	12459	15.851 seconds
test-input-4.txt	2-Opt	17318	167.672 seconds
test-input-5.txt	Repetitive Nearest Neighbor	27128	23.657 seconds
test-input-6.txt	Repetitive Nearest Neighbor	39469	22.336 seconds
test-input-7.txt	Repetitive Nearest Neighbor	61906	136.996 seconds

Best tours with time lengths for the competition instances - Unlimited time:			
Input	**Algorithm used**	**Length**	**Runtime**

test-input-1.txt	2-Opt	5373	3.178 seconds
test-input-2.txt	2-Opt	7487	3.410 seconds
test-input-3.txt	2-Opt	12459	15.851 seconds
test-input-4.txt	2-Opt	17318	167.672 seconds
test-input-5.txt	2-Opt	24071	2927.883 seconds
test-input-6.txt	Repetitive Nearest Neighbor	39469	22.336 seconds
test-input-7.txt	Repetitive Nearest Neighbor	61906	136.996 seconds

Algorithm data:

Optimal results:
tsp_example_1.txt: 108159
tsp_example_2.txt: 2579
tsp_example_3.txt: 1573084
test-input-1.txt: 5333*
test-input-2.txt: 7384*
test-input-3.txt: 12067*
test-input-4.txt: 16720*
test-input-5.txt: 22976*
test-input-6.txt: 32448*
test-input-7.txt: unknown

***Optimal results found using Concorde TSP Solver

(Ratio = Algorithm result/Optimal result):

Naïve Algorithm			
Input	**Length**	**Ratio**	**Avg. Runtime**
tsp_example_1.txt	150781	1.391	0.138 seconds
tsp_example_2.txt	2808	1.089	0.146 seconds
tsp_example_3.txt	112310765	71.395	0.230 seconds
test-input-1.txt	27301	5.093	0.132 seconds
test-input-2.txt	48053	6.452	0.133 seconds
test-input-3.txt	123877	10.203	0.139 seconds
test-input-4.txt	247675	14.773	0.146 seconds
test-input-5.txt	522297	22.699	0.141 seconds
test-input-6.txt	1050395	32.341	0.152 seconds
test-input-7.txt	2580008	unknown	0.171 seconds

Nearest Neighbor Algorithm			
Input	**Length**	**Ratio**	**Avg. Runtime**
tsp_example_1.txt	123422	1.141	0.145 seconds
tsp_example_2.txt	3059	1.186	0.168 seconds
tsp_example_3.txt	1925103	1.224	35.556 seconds
test-input-1.txt	5672	1.064	3.168 seconds
test-input-2.txt	7710	1.044	3.195 seconds
test-input-3.txt	14217	1.178	3.187 seconds
test-input-4.txt	20130	1.204	3.231 seconds
test-input-5.txt	27558	1.199	3.364 seconds
test-input-6.txt	39883	1.229	4.024 seconds
test-input-7.txt	63224	unknown	8.305 seconds

Repetitive Nearest Neighbor Algorithm			
Input	**Length**	**Ratio**	**Avg. Runtime**
tsp_example_1.txt	130921	1.210	0.151 seconds
tsp_example_2.txt	2975	1.154	0.671 seconds
tsp_example_3.txt	1918173	1.219	1311.359 seconds
test-input-1.txt	5911	1.108	0.156 seconds
test-input-2.txt	8011	1.085	0.165 seconds
test-input-3.txt	14826	1.229	0.583 seconds
test-input-4.txt	19711	1.179	3.503 seconds
test-input-5.txt	27128	1.181	26.337 seconds

test-input-6.txt	39469	1.216	22.336 seconds
test-input-7.txt	61906	unknown	136.996 seconds

Cheapest Link			
Input	**Length**	**Ratio**	**Avg. Runtime**
tsp_example_1.txt	152260	1.408	0.330 seconds
tsp_example_2.txt	3570	1.384	60.97 seconds
tsp_example_3.txt	---	---	Run did not finish
test-input-1.txt	6478	1.215	0.04 seconds
test-input-2.txt	9623	1.303	0.75 seconds
test-input-3.txt	15990	1.325	44.89 seconds
test-input-4.txt	23104	1.382	714.0 seconds
test-input-5.txt	31109	1.354	10,464.29 seconds
test-input-6.txt	---	---	Run did not finish
test-input-7.txt	---	---	Run did not finish

2-Opt Algorithm			
Input	**Length**	**Ratio**	**Avg. Runtime**
tsp_example_1.txt	109067	1.008	0.211 seconds
tsp_example_2.txt	2654	1.029	9.213 seconds
tsp_example_3.txt	---	---	Run did not finish
test-input-1.txt	5373	1.008	3.178 seconds
test-input-2.txt	7487	1.014	3.410 seconds

test-input-3.txt	12459	1.032	15.851 seconds
test-input-4.txt	17318	1.036	167.672 seconds
test-input-5.txt	24071	1.048	2927.883 seconds
test-input-6.txt	---	---	Run did not finish
test-input-7.txt	---	---	Run did not finish

Works Cited

Cook, William. “Concorde TSP Solver.” http://www.math.uwaterloo.ca/tsp/concorde.html, July 2016. Web. 6 June 2017.

Erickson, Jeff. “Algorithms, Etc.” http://jeffe.cs.illinois.edu/teaching/algorithms/, 30 December 2014. Web. 6 June 2017.

“Nearest neighbour algorithm.” https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm, 6 November 2016. Web. 6 June 2017.

“Traveling salesman problem.” https://en.wikipedia.org/wiki/Travelling_salesman_problem, 2 June 2 2016. Web. 6 June 2017.

Webb, Andrew. “C++ Implementation of 2-opt to the ‘Att48’ Travelling Salesman Problem,” http://www.technical-recipes.com/2012/applying-c-implementations-of-2-opt-to-travelling-salesman-problems/, 20 April 2012. Web. 7 June 2017.

MSLC Mathematics. "Cheapest Link Algorithm," https://www.youtube.com/watch?v=nYKsLRxIBmA, 7 July, 2014. Web. 8 June 2017.