The Traveling Salesman Problem (TSP)

CS 325 - Analysis of Algorithms
Group 3: Jon-Eric Cook, Shannon Jeffers, Peter Moldenhauer
12/01/17

The traveling salesman problem (TSP) is classified as an NP-Hard problem in theoretical computer science. Given a set of cities and the distance between each pair of cities, the TSP aims to find the shortest possible route that visits every city exactly once and then returns to the starting point. Wikipedia states that this problem was “first formulated in 1930 and is one of the most intensively studied problems in optimization”.

For this project, the group members researched three different algorithms for “solving” the traveling salesman problem. Once this initial research was complete, we selected one of the three algorithms to further design and implement to find the best possible tour from the provided input data. This report describes each of the three algorithms that were researched as well as which algorithm the group decided to implement and why this algorithm was chosen. This report also contains the pseudo-code for each algorithm, the “best” tours for the three provided example instances, and the best solutions for the competition test instances.

Algorithm #1 - The Nearest Neighbor Algorithm:

According to Wikipedia, “the Nearest Neighbor Algorithm was one of the first algorithms used to determine a solution to the Traveling Salesman Problem”. The main goal of the Nearest Neighbor Algorithm is that it finds a Hamiltonian Circuit. This means that it finds a path through the graph of cities and distances but only uses each vertex (city) once. Once all cities are visited, the path then returns to the starting city. Therefore, as soon as one city is visited, it is taken out of play and only the remaining unvisited cities can be chosen from to travel to next. There are three basic steps of the Nearest Neighbor Algorithm to “solve” the Traveling Salesman Problem:

1. Start with a given vertex as the “starting city” in the graph of the cities (vertices) and distances between each city (edges). Choose the edge with the least weight (least distance, i.e. the next closest city) and then use that as the first edge in your total path.
2. Continue in this manner by selecting the next closest city that you have not already visited each time as you traverse the graph. It should be noted that for each city in the graph that you visit, it cannot be visited again. Therefore, the total options of selecting which city to travel to next will get less and less as you traverse the graph.
3. Lastly, when you have visited every city (vertex) in the graph, return to the starting vertex. Note: this return trip back to the starting city is also added to the total distance of the path taken throughout the graph.

In other words, a simpler description of the Nearest Neighbor Algorithm could be as follows: Start at a given vertex. Travel the shortest route out of that vertex. Then from this vertex travel to the next shortest vertex that you have not visited before. Continue this process until you visit each vertex and then travel back to the starting vertex.

The pseudocode for the Nearest Neighbor Algorithm can be written out in the following manner…

```
NN(Graph G){ //G is represented as 2D matrix of vertices and distances between them
        s = starting/current vertex // initially leave it unvisited until the end
        tour = [] // blank list to hold all of the cities that will make up the final tour to take
        while(all vertices have not been visited){
                n = infinity
                for each neighbor vertex v of s:
                        if (distance of neighbor is < n):
                                n = distance of neighbor
                                nextVertex = v
                s = nextVertex
                s.visited = true // mark s as visited
                tour.append(s) // add s to the final tour
        }
        return tour
}
```

The Nearest Neighbor Algorithm can be drawn out by using either a graph or a 2D matrix. It is not required to draw out a graph to complete this algorithm but it may be useful to visualize it as the algorithm progresses.

The Nearest Neighbor Algorithm is a greedy algorithm because with each iteration, the next closest available city is always chosen to travel to next. Once a city is chosen, it is removed from the list of available cities and never considered again. Compared to the Brute Force Algorithm, the Nearest Neighbor Algorithm is much more efficient at “solving” the Traveling Salesman Problem. However, despite being quicker and more efficient than Brute Force, the Nearest Neighbor Algorithm might not always result in the optimal solution for the problem. This algorithm does not guarantee the best solution but it will find a solution that is close to the optimal solution. For this reason, the Nearest Neighbor Algorithm is described as an “approximate algorithm”.

Algorithm #2 - Cheapest Link Algorithm:

Similar to the Nearest Neighbor Algorithm, the Cheapest Link Algorithm is also a greedy algorithm that can be used to “solve” the Traveling Salesman Problem. The Cheapest Link Algorithm is another “approximate algorithm” which is fairly efficient but may not always result in

the optimal solution for the problem. The main goal of the Cheapest Link Algorithm is that it constructs a Hamilton Circuit which can then be used to “solve” the Traveling Salesman Problem.

Given a graph of cities and distances between them, the Cheapest Link Algorithm begins at a given vertex (city) and chooses the edge with the smallest weight (the “cheapest” edge), randomly breaking ties. The algorithm then keeps choosing the “cheapest” edge unless it either closes a smaller circuit or results in three selected edges coming out of a single vertex. This process continues until the Hamilton Circuit is complete.

In other words, the Cheapest Link Algorithm works by repeatedly choosing the cheapest link in the graph (of cities and distances) that 1) doesn’t close the circuit and 2) doesn’t create a vertex with three edges coming out of it. These cheapest links are added to the tour until it needs one more edge to complete it, at which point condition (1) is removed so the cheapest link that does not create a vertex with three edges will then be added and the tour is complete.

The pseudocode for the Cheapest Link Algorithm can be written out in the following manner…

```
CheapestLink(Graph G, startingVertex v)
        edgeDistance = infinity
        hamCircuit = []

        for each edge e connected to v:
                If (e.distance < edgeDistance)
                        edgeDistance = e.distance
        e.colored = true // color the chosen edge
        hamCircuit.append(e) // add e to the final hamilton circuit

        while(hamilton circuit is not complete) {
                choose next cheapest uncolored edge unless:
                        edge closes a smaller circuit &&
                        new edge results in 3 colored edges coming out of a single vertex
                e.colored = true // color the chosen edge
                hamCircuit.append(e)
        }
        return hamCircuit
}
```

Similar to the Nearest Neighbor Algorithm (described above), the Cheapest Link Algorithm is efficient but not optimal. The Cheapest Link Algorithm might not always result in the optimal solution for the problem. This algorithm does not guarantee the best solution but it will find a solution that is close to the optimal solution. For this reason, the Cheapest Link Algorithm, like the Nearest Neighbor Algorithm, is described as an “approximate algorithm”.

Algorithm #3 - 2-OPT (or "2-Optimal"):

According to Wikipedia, the 2-OPT Algorithm is a "simple local search algorithm first proposed by Croes in 1958 for solving the Traveling Salesman Problem". The main idea behind 2-OPT is that it finds a tour throughout the given graph of cities/distances that crosses over itself and then it reorders this tourso that it does not cross over itself.

In order to use 2-OPT, a tour must already exist. A tour can be created by using a quick greedy algorithm (as this starting tour does not need to be optimal). The 2-OPT algorithm then works to optimize the existing tour into a tour that is optimal. It selects an optimal solution by performing a "swap" where it exchanges 2 edges to create a new tour. This edges used in the swap are chosen at random. If they result in an improvement, this version of the tour is kept otherwise the algorithm proceeds with the last version. A complete 2-OPT local search will compare every possible valid combination of the swapping mechanism.

The pseudocode for the 2-OPT Algorithm can be written out in the following manner…

```
2opt(Graph G) {
        Get size of tour
        Create copy of tour
        Run loop until no further improvements are made
                        Get current tour distance
                        Loop through size of the tour -1 for variable i
                                Loop through size of the tour for variable k
                                        Call swap on I and k
                                        Get new tour distance
        Compare old distance and new distance, if new distance is shorter set a bool to say we
        made an improvement
                Set best distance to new distance
                Update tour to include the change
}

swap(I, k) { // pesudocode for swap function
        Get size of the tour
        Add  all edges up until I to a tour copy
        Add edges i-k in reverse order to tour copy
        Add remaining edges to tour.
}
```

The 2-OPT Algorithm is slightly more complex than the Nearest Neighbor and Cheapest Link Algorithms but it is still far more efficient than the brute force approach to solve the Traveling Salesman Problem. The naive implementation of 2-OPT runs in O(n^2) time. This time

complexity is due to selecting an edge, searching for another edge and then completing an appropriate move. While the 2-OPT Algorithm is a good contender to “solve” the Traveling Salesman Problem, 2-OPT is often used to solve similar problems to TSP such as the Vehicle Routing Problem.

Algorithm chosen to implement:

Our group decided to implement the Nearest Neighbor Algorithm to “solve” the Traveling Salesman Problem. We decided to choose this specific algorithm because it is a simple algorithm, it is easy to understand and there are numerous online resources to aid us in the implementation. The other two algorithms that were researched (Cheapest Link, 2-OPT) appeared to result in more optimal solutions than Nearest Neighbor, but the simplicity of the Nearest Neighbor Algorithm played a larger factor in the final decision of which algorithm to choose. All of the group members felt that the simplicity of the Nearest Neighbor Algorithm would result in fewer coding errors and less time spent on debugging versus if we went with one of the other two algorithms that were researched. Even though Nearest Neighbor may not be the most optimal algorithm out there to “solve” TSP, we felt that it would still be enough for the cases and input data that we are to work with for this project.

Results of the best tours for the three example instances:

Below is a table of all of the resulting data from our algorithm for the three example instances and the time it took to obtain the tours. There was no time limit for these example instances.

File	Experimental Result	Optimal Result	Ratio	Time
tsp_example_1.txt	131607	108159	1.216	0.01 sec
tsp_example_2.txt	3118	2579	1.208	0.02 sec
tsp_example_3.txt	1952748	1573084	1.241	53.5 sec

Results of the best solutions for the competition test instances:

Below is a table of all of the resulting data from our algorithm for the competition test instances. The competition requires that our algorithm finds the best possible solution within 3 minutes.

File	Experimental Result	Time
test-input-1.txt	6277	0.01 sec

test-input-2.txt	9146	0.01 sec
test-input-3.txt	15493	0.01 sec
test-input-4.txt	20861	0.03 sec
test-input-5.txt	28030	0.18 sec
test-input-6.txt	40078	0.77 sec
test-input-7.txt	63546	6.05 sec

References:

https://www.youtube.com/watch?v=G8a8bJuQxnw

https://www.youtube.com/watch?v=RK6eu3jvKzw

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

http://www.technical-recipes.com/2017/applying-the-2-opt-algorithm-to-traveling-salesman-problems-in-java/

https://www.math.ku.edu/~jmartin/courses/math105-F11/Lectures/chapter6-part4.pdf

https://ocw.mit.edu/courses/sloan-school-of-management/15-053-optimization-methods-in-management-science-spring-2013/lecture-notes/MIT15_053S13_lec17.pdf