Final Project: Traveling Salesman (TSP)

James Hippler, Kyle Martinez, Caitlin Dudley
Project Group 2
Spring 2018

Introduction

The traveling salesman problem (TSP) is classified as an NP-Hard problem in theoretical computer science. TSP asks the question: ”Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” 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”.

We wanted find a reasonable path such that the distance of such path is no greater than 1.25× the shortest possible route. To accomplish this, we have researched three algorithms: Nearest Neighbor, Christofides‘, and a 2-opt heuristic algorithm (also known as Lin-Kernighan heuristics). Once our initial research was completed, we selected two of the algorithms to further design and implement to find the best possible tour from the provided input data.

This report first describes and provides pseudo code for each of the researched algorithms. Next, this report discusses which algorithms were chosen for implementation and why the algorithms were selected. Finally, this report includes the results of our “best” tours for the three provided example instances and the best solutions for the competition test instances.

Researched Algorithms

Algorithm 1: The Nearest Neighbor Algorithm

Description

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". The algorithm itself runs in $O(n^2)$ time.

Pseudocode

```
Nearest_neighbor (//input graph/matrix):
        select a starting/current vertex, call it s
        create an empty list to hold all cities, with all cities unvisited
        while all vertices have not been visited:
                n = infinity
                for each neighbor vertex v of s:
                        if distance to neighbor is < n:
                                n = distance of neighbor
                                nextVertex = v
                s = nextVertex
                mark s as visited
                add s to the final tour
        return list of cities
```

Algorithm 2: Christofides' Algorithm

Description

Christofides algorithm attempts to solve the Travelling Salesman Problem (TSP) through approximation. It was developed and published by Nicos Christofides in 1976. The algorithm assumes that the provided distances in the problem set will form a metric space, are symmetric, and obey the triangle inequality. Considering that the algorithm provides only and approximation, it relatively quick determining a solution but can only guarantee the solution is accurate within a factor of 3/2 (1.5) of the optimal solution length. The algorithm finds a double minimum spanning tree with the additional minimum-weight matching calculation. This is aspect of the algorithm consumes the most time and greatly impacts the overall complexity. Christofides as one of the original approximation algorithms and helped prove that approximation could be implemented as a practical approach to intractable problems. The Christofides algorithm as initially referred to as the Christofides heuristic because the term algorithm as not commonly applied toward approximation algorithms.

Pseudocode

Initially a graph G will need to be generated containing a complete set of vertices V (cities) along with their nonnegative weights w (travel costs) associated with every edge

(connection) in the graph *G*. This will make *G=(V,w)* as our instance of the Travelling Salesman Problem. According to triangle inequality, it should be the case that *w(ab)+(bc) ≥ w(ac)* for every three vertices *a, b, c*.

Cristofides:

Create minimum spanning tree (MST) of graph G.
Establish a set of vertices with odd degrees in T as O.
O should have an even number of vertices T based on the rules of the handshaking lemma.
Find minimum-weight perfect matching M in the induced subgraph.
Given by the vertices in O.
Combine the edges of M and T to form connected multigraph H.
Each vertex of H must have an even degree.
Form Eulerian circuit in H.
H is Eulerian because it is connected with only even-degree vertices.
Convert Eulerian circuit (discovered in previous step) in Hamiltonian Circuit.
Ignoring repeating vertices (shortcutting).

Algorithm 3: 2-opt Heuristic

Description

2-opt is an improvement heuristic. This means that it does not construct a solution from scratch - it takes an already constructed solution and iterates over it, hopefully producing a better solution. This means that a TSP valid tour must be generated before 2-opt can work its magic.

2-opt

Given a valid TSP tour, if a better TSP tour can be obtained by deleting 2 edges and adding 2 edges, than the tours are 2-adjacent. A TSP tour is 2-optimal when it cannot be improved further. The 2-opt heuristic is searching for a 2-adjacent tour which is lower cost than the current one. This is repeated until the tour is 2-optimal, and this would be a locally optimal TSP tour (though may not be the optimal solution). It is important to note that when the edges are deleted and added, they must still form a tour (it must be connected).

An algorithm using 2-opt can use a swap function to handle the adding and deleting of edges. The way this works is summarized on wikipedia:

```
2optSwap(route, i, k) {
    1. take route[0] to route[i-1] and add them in order to new_route
    2. take route[i] to route[k] and add them in reverse order to new_route
    3. take route[k+1] to end and add them in order to new_route
    return new_route;
```

The way this pseudo code works is to construct a new route, swapping the places of route[i] and route[k], leaving the rest in tact. This means that 2 edges will be deleted from and 2 will be added to the original tour to make a new tour (route[i - 1] to route[i], replaced by route[i-1] to route[k]. And route[k] to route[k+1], replaced with route[i] to route[k+1]).

Essentially the 2 swapping is trying to uncross any paths that are crossed, creating a shorter tour.

Pseudocode

```
Generate initial tour T
currentRoute = T

While T can be improved:
        bestOutcome = T.cost

        # for pairs of edges, swap 2 of them and check for improvement
        For i = 1 ... number of nodes - 1:
                For j = i + 1 ... number of nodes:
                        newRoute = 2optSwap(currentRoute, i, j)
                        Calculate newOutcome
                        If newOutcome < bestOutcome:
                                bestOutcome = newOutcome
                                currentRoute = newRoute
```

Implementation

We chose to implement the nearest neighbor and 2-opt heuristic algorithms. We chose nearest neighbor because it is essentially greedy which makes a good choice for being efficient and quick. Additionally, because it runs ”fairly” fast for most input sizes, we were able to set it up to try each possible starting point. While fast, nearest neighbor does not always return the optimal result. Thus, we implemented the 2-opt algorithm to improve optimality of our tour.

Nearest neighbor was chosen as a simple, fast way to create an initial tour. Our implementation works by passing the list of cities and their coordinates to the function which generates the tour. We chose the first city in the list as our starting point. From there, we create an unvisited list, and put all the others cities in it. We then loop while there are still unvisited cities and greedily choose the shortest path to that city. As the paths are chosen, the total distance is summed up.

Our 2-opt implementation works by iterating over the cities in a tour, swapping two adjacent cities, and evaluating the change in the distance of the route after the swap. If it is an improvement (a decrease in distance), then it uses that new route as the current best route. It then repeats. It terminates based on how many improvements it makes. If it does not make an improvement after 20 swaps in a row, then it ends and uses the current best route as the best route. It then returns the route and the distance.

Initially, we weren't going to use the termination method we chose. We were going to simply have the algorithm terminate if there was no swap with the current city that was an improvement. This led to mediocre results, our improvement over the nearest neighbor generated tour was not enough. After some research, we found the methodology to have it traverse further into the tour, even if it couldn't find an improvement on a city. If this happened 20 times in a row, then we said our tour was good enough. This led to much shorter tours.

We chose these two algorithms over Christofides' because while Christofides' is fast, it doesn't guarantee tours within the threshold given by the assignment. This was the chief reason for choosing the nearest neighbor and 2-opt combination.We believed it would yield a more optimal tour despite being slower.

Results

See below for results from the 3 example instances and 7 test instances.

For both the example and test instances, the nearest neighbor algorithm was run, followed by the 2-opt algorithm. Additionally, tsp_example_1.txt was run using the "-t" mode, whereas tsp_example_2.txt and tsp_example_3.txt were run using "-c" as the mode. This was done in order to get timely results. All competition files were run using the "-c" mode.

"Best" tours for the 3 example instances (no time limit):

File	Tour result	Optimal tour	Ratio <= 1.25	Time (seconds)
tsp_example_1.txt	115057	108159	1.06	11.095895052
tsp_example_2.txt	2835	2579	1.099	157.039761782
tsp_example_3.txt	1963589	1573084	1.25	384.487470865

Best tours for the competition test instances (3 minute time limit):

File	Tour result	Time (seconds)
test-input-1.txt	5639	3.00820398331
test-input-2.txt	7720	27.5437572002
test-input-3.txt	12751	157.080205202

test-input-4.txt	18828	157.062119007
test-input-5.txt	28179	157.38245821
test-input-6.txt	40751	159.366771936
test-input-7.txt	63739	172.008966923

References

1. https://en.wikipedia.org/wiki/Travelling_salesman_problem
2. https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
3. https://pdfs.semanticscholar.org/9fc2/8fac2603cfda11da21033a58bbb5c7b75e09.pdf
4. http://cs.indstate.edu/~zeeshan/aman.pdf
5. https://en.wikipedia.org/wiki/Christofides_algorithm
6. https://100algorithms.wordpress.com/2014/11/20/100-2-christofides-algorithm
7. http://personal.vu.nl/r.a.sitters/AdvancedAlgorithms/2016/SlidesChapter2-2016.pdf
8. https://www.quora.com/What-is-the-optimal-and-best-algorithm-for-solving-the-traveling-salesman-problem
9. https://heuristicswiki.wikispaces.com/Christofides+%28Heuristic%29
10. http://www.technical-recipes.com/2012/applying-c-implementations-of-2-opt-to-travelling-salesman-problems/
11. Networks 3: Traveling salesman problem - MIT OpenCourseWare, https://ocw.mit.edu/courses/sloan...optimization.../MIT15_053S13_lec17.pdf
12. Wikipedia, https://en.wikipedia.org/wiki/2-opt