CS 325
Project Group 19: Thomas Mathew, John Fitzpatrick, Ryan Mackey, Connor Pacala

Methods for Solving the Traveling Salesman Problem

Method 1: Brute Force- check all permutations, pick the best. Takes a LONG time.
http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/CG-Applets/TSP/notspcli.htm
https://en.wikipedia.org/wiki/Brute-force_search
http://www.cs.sfu.ca/CourseCentral/125/tjd/tsp_example.html
Brute-forcing the TSP will always yield the optimal result, however it has an extremely long running time, O(n!). Even a small case of 20 locations would take years to finish. The idea behind brute-force and TSP is simple: calculate every possible tour and pick the shortest. For example, assume we have cities A, B and C. A brute force algorithm would examine the following tours: ABC, ACB, BCA, BAC, CBA, and CAB. Each distance would be calculated and then compared with the shortest tour being selected.

Method 2: Greedy/Nearest Neighbor
https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
http://www.austincc.edu/powens/+Topics/HTML/06-5/06-5.htm
http://www1.cs.columbia.edu/CAVE/publications/pdfs/Kumar_ECCV08_2.pdf
This is the simplest algorithm to solve TSP in a somewhat efficient manner. It is a greedy algorithm where a city is selected as the starting city *A*, and added to set **S**. Examine all nodes that connect to *A* and select the node with the shortest distance, node *B*. Add node *B* to **S**. From this point the algorithm will examine all nodes that connect to the most recently added node in set **S**. If it has neighbors that are not in set **S**, then the neighbor with the shortest distance is added to **S**. If all neighbors are in **S** then it forms a connection to the first node in **S**, thus completing the tour.

Method 3: Minimum Spanning Tree
https://www.ics.uci.edu/~eppstein/161/960206.html
https://en.wikipedia.org/wiki/Minimum_spanning_tree
This solutions starts by finding the minimum spanning tree of all cities. Then starting at any city, a depth-first search is performed and the sequence **S** of cities visited is stored. The algorithm then follows the sequence of cities in **S**, keeping a list **L** of those that have been visited. When a city in list **L** is encountered it is skipped and the next city in **S** (which is not already in **L**) is added to **L**. After all cities in **S** have been visited, the list **L** contains the sequence of cities that is the solution.

Method 4: Christofides Algorithm
http://www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf
https://en.wikipedia.org/wiki/Christofides_algorithm
This algorithm is based in the minimum spanning tree solution, but also contains a method called *minimum-weight perfect matching* to ensure all vertices connect to a positive number of edges. This is accomplished by starting with a minimum spanning tree and adding all the vertices with odd edges to a set **O**, then going through each vertex in **O** and match it with the

closest vertex which is also in **O** until all are matched. Once this is accomplished, one can perform the same depth-first search sequence and city traversal (bypassing those already visited as above). Although this seems very similar to the minimum spanning tree solution, this algorithm gets much closer to the optimal value ($\leq 1.5 \cdot OPT$) compared to the minimum spanning tree ($\leq 2.0 \cdot OPT$).

Method 5: 2-Opt

https://en.wikipedia.org/wiki/2-opt
https://webcache.googleusercontent.com/search?q=cache:CCXpeYNU_9IJ:https://www.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html+&cd=4&hl=en&ct=clnk&gl=us

2-Opt is part of a family called k-opt, where you start with a tour (probably from the Nearest Neighbor method because it would be fastest), then you improve it iteratively. In 2-opt, you take two non-adjoining edges and remove them, leaving you with two "open" cycles (each are open because they are missing an edge, imagine a U, with two vertices at the top on each side). You then check the distance between the vertices to see if the sum of the resulting two new edges would be smaller than the sum of the previous two edges, if they are, you change the edges and then move on to the next two edges. You would do this iteratively, so for a given edge, you would check with the edge two away from it (because you don't check with adjoining edges), then three edges away, etc. until you reach the edge two away on the opposite side. The others in the k-opt family choose *k* number of edges to remove and check, and some don't use the adjacent-edge rule for all the edges.

Method 6: Held-Karp algorithm

https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm
http://stackoverflow.com/questions/22985590/calculating-the-held-karp-lower-bound-for-the-traveling-salesmantsp
http://www.csd.uoc.gr/~hy583/papers/ch11.pdf

The idea behind the Held-Karp algorithm is simple: divide the list of cities up, solve each subproblem, then link each subproblem in an optimal way……

Group 19 TSP Solution

A verbal description of your algorithm(s) as completely as possible. You may select more than one algorithm to implement.

The 2-opt algorithm for the TSP problem focuses on eliminating parts of routes that cross over themselves. The first step is to generate a route that we can improve upon using 2-opt. We will use the simple Nearest Neighbor approach to generate our route. Nearest Neighbor

works by simply taking the shortest-distance, non-touched node option available at each step of the algorithm, until all nodes have been touched. This result will often be suboptimal as the algorithm is “short-sighted”.

This is where 2-opt comes in. 2-opt takes an existing route and improves upon it by eliminating inefficient steps. The main work is done via swapping and thus reversing the order of the nodes. Then, we compare the distance of our new route vss the distance of the route that we just had. If the distance has decreased, we keep going and keep improving. Otherwise we stop. IT is possible to set up the algorithm to target specific parts of the route by changing the for loops. Or, we can just check the full route until improvements are no longer found.

A discussion on why you selected the algorithm(s).

We looked at a number of algorithms to solve the TSP. The first one to consider is a brute-force algorithm that checks all solutions and chooses the best one. While this algorithm will always find the best solution, it’s time cost is gigantic. We then looked at a nearest neighbor/greedy algorithm, which picks the shortest-distance unvisited node at each step. This one was tempting as it was simple to understand, but the flaws are very obvious; as it’s trivial to draw a problem up where this algorithm fails miserably at minimizing total distance. Other considerations included the Held-Karp, MST, and Christofides solutions.

We eventually decided on the 2-opt algorithm however. The ease of understanding for the nearest-neighbor was appealing, but the results were not. 2-opt does a good job addressing its shortcomings. As a more iterative algorithm, we also felt more comfortable putting this algorithm together and implementing it.

Pseudo code for the algorithm

Citations:https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
https://en.wikipedia.org/wiki/2-opt

First, for the Nearest neighbor algorithm to build the route:

1. Pick an arbitrary vertex to start with as the current vertex.

2. Check all distances from the current vertex to all unvisited vertices. Let the shortest edge lead to a vertex “V”..
3. Now we set the current vertex to “V”.
4. V gets marked as visited.
5. If all vertices have been marked as visited, exit, we are finished and have our route.
6. Otherwise loop back to step #2.

Then, we pass that route into the 2opt algorithm, to improve it and shorten total distance. First we define the 2 opt procedure itself:

```
  2opt(route, i, k)
{
      1. take route[1] 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;
}
```

Then we apply the procedure using 2 “for” loops, until the “new_distnace” is no longer < “best_distance”, with best_distance signifying the current distance of the route.

```
 repeat until no improvement is made {
      start_again:
      best_distance = calculateTotalDistance(existing_route)
      for (i = 0; i < number of nodes eligible to be swapped - 1; i++) {
         for (k = i + 1; k < number of nodes eligible to be swapped; k++) {
            new_route = 2optSwap(existing_route, i, k)
            new_distance = calculateTotalDistance(new_route)
            if (new_distance < best_distance) {
               existing_route = new_route
               goto start_again
            }
         }
      }
  }
```

Best results for example files

File Name	Distance	Time (ms)
tsp_example_1	150393	14
tsp_example_2	3210	35

tsp_example_3	1964948	6544

Best results for competition files

File Name	Distance	Time (ms)
test-input-1.txt	5926	10
test-input-2.txt	9503	17
test-input-3.txt	15829	35
test-input-4.txt	20215	53
test-input-5.txt	28685	104
test-input-6.txt	40933	201
test-input-7.txt	63780	821