# Traveling Salesman Problem

Nathan Bennet ○ Jessie Little ○ Ian Stanfield

## Researched Algorithms

*Nearest Neighbor:*

The Nearest Path solution to the Traveling Salesman Problem is an efficient algorithm, but does not always produce an optimal solution. Nearest Path was an early solution to TSP. It is very easy to implement and runs in O(n^2 log(n)) time.

1. Choose an unused starting city.
2. Choose the shortest path to the nearest city that has not already been visited.
3. Repeat step 2 until all vertices have been visited.
4. Return to starting city.
5. Repeat from step 1 until the algorithm has been run starting from every city.

This approach to solving TSP ended up being a good solution for extremely large inputs, but did not provide an optimal enough solution for it to be useful alone on some smaller inputs.

**Pseudocode**

```
// unvisited = array of city id's
// distance = array of x and y locations
nearestNeighbor(unvisited, distance, startingCity)
   path = []
   distance = 0
   path[0] = unvisited.remove(startingCity)

    while unvisited.length > 0:
       nearestCity = getNearest(unvisited, distance, path[len – 1])
       path.append(unvisited.remove(nearestCity[1]))
       distance += nearestCity[0]

    return [path, distance]

// Run nearest neighbor using every city as a starting point
greedyNeighbor(unvisited, distance)
   bestPath = null
   distance = infinity

    for i in unvisited.length:
       path = nearestNeighbor(unvisited, distance, unvisited[i])

       if path[1] < distance:
          bestPath = path[0]
```

```
        distance = path[1]

    return [distance, bestPath]
```

*Minimum Spanning Tree 2-approximation:*

The 2-approximate solution using a minimum spanning tree is a relatively simple. For this algorithm to work, our problem instances must satisfy the Triangle-Inequality. Since we are only considering the special case where the cities are locations in a 2D grid, with the distance between points being given by their Euclidean distance, we know that the Triangle-Inequality is satisfied. At a high level there are 3 steps to this algorithm ("Travelling Salesman Problem | Set 2 (Approximate using MST)", 2017).

1. Select a starting point for this salesman. This will also be the ending point. We will title this point 0.
2. Construct a minimum spanning tree using 0 as the root.
3. List the vertices of the minimum spanning tree in preorder walk and add 1 to the end.

Step 1 is straightforward, so we'll start by taking a closer look at steps 2 and 3.

**Step 2: Minimum Spanning Tree**

We have a couple of options for a minimum spanning tree. We could use Prim's algorithm or Kruskal's algorithm. Kruskal's algorithm has a running time of O(E lg V), whereas Prim's has a running time of O(E + V log V) when a priority binary heap is used, where E is the number of edges and V is the number of vertices. Since our graph for the traveling salesman has a lot of edges, we will use Prim's as we may gain a little efficiency by avoiding sorting. In order to get the full efficiency of Prim's algorithm, an adjacency list with a minimum priority queue or a minimum binary heap is usually used ("Greedy Algorithms | Set 6", 2017). Because we will be using a complete graph, I am going to avoid using a heap and use an array. We will need to traverse the entire array (or tree) in order to update the distances and find the next minimum. As we update distances we will shuffle the next smallest distance to the end, where it can then be popped off the back. The algorithm would take O(V + E) since swaping the minimum forward cost O(1) each time and taking the next minimum off the back of the array is also O(1). Something to note, is that this algorithm is only efficient because we are dealing with a complete graph. If the graph was not complete than a heap would be more efficient. Our algorithm is as follows:

Let graph be an adjacency list for the traveling salesman instance. This assumes a starting point at city 0.

```
PrimMST(cities):
v = number of vertices
cities = list of city coordinates
        let key[] be a new list of size v containing lists of size 3. For each element in key, the list of size 3
            contains [index of list +1, -1, infinity]
        set the last element in key to [0, -1, 0]
    let parent[] be a new list of size v and set all values to null
    While key is not empty:
        p = get minimum off end of key
        parent[p[0]] = p[1]
        for each element in key:
            if n is in Q and key[n] > distance(p,n)
                key[n] = distance(p,n)
```

```
                    parent[n] = p
                if not the first element of the array:
                if key[previous] <= key[current]:
                    swap key[previous] and key[current]
    return parent
```

**Step 3: Preorder walk**

To build our preorder walk we will want to start by taking the results of prim's algorithm and building an adjacency list.

```
primAdj(parent):
    let tree[] be a new list of length(parent)
    for i=1; i < len(parent); i++  (since we know the parent of the first one is null, we start at 1)
        append i to tree[parent[i]]
    return tree
```

Now that we have our new adjacency list tree, we will need to traverse it via a depth first search. Due to the fact that our graph is a minimum spanning tree, we don't have to worry about keeping track of which nodes have been visited, since each node can only be visited once ("Depth-First Search").

```
getCycle(tree):
    let cycle[] be a new empty list
    let stack[] also be a new empty list
    push 0 on stack
    while stack is not empty
        temp = pop stack
        append temp to cycle
        for each item i in temp's adjacency list
            push tree[temp][i] onto stack
    return cycle
```

## *Christofides' Algorithm:*

Christofides' algorithm is an algorithm developed by a computer scientist named Nicolas Christofides in 1976. It currently holds the best approximation ratio for the traveling salesman problem across general metric space, providing an approximation ratio of 3/2 where the computed solution is guaranteed to be within a factor of 3/2 of the optimal solution.

In order, the first step is **generating a minimum spanning tree** from the provided graph. For our theoretical implementation, we can use Kruskal's algorithm to devise a minimum spanning tree of our graph in O(E log V), where E is the number of edges and V is the number of vertices. Alternatively, we can use Prim's algorithm to get a minimum spanning tree in $O(V^2)$ time, where V is the number of vertices. In our real world implementation, Kruskal's algorithm represented a significant computational bottleneck as it requires sorting the entire set of edges (by their weight) provided by the graph. In detail, we iterate through all vertices in the graph, and create a

set for each vertex. We then iterate through our sorted edges and add them to our minimum spanning tree if the destination vertex and origin vertex are not in the same set. If that edge can be added, we then combine the sets that contain our destination and origin vertices. We then return the edges of our minimum spanning tree.

```
EDGE_LIST = edges sorted by weight
For EACH_VERTEX:
        Create an set containing that vertex, managed by a MASTER_SET.

MST_EDGES = []
EDGE_INDEX = 0
While length of mst_edges < length of vertices - 1:
        With EDGE at EDGE_LIST[EDGE_INDEX]
        EDGE_INDEX += 1
        If ORIGIN_VERTEX and DEST_VERTEX in EDGE are not in the same set:
                Add EDGE to MST_EDGES
                Union sets containing ORIGIN and DEST in MASTER_SET.

Return MST_EDGES
```

The next step is **finding the vertices with an odd number of edges** incidental to them. To find these vertices, we iterate through each edge in our minimum spanning tree, and keep a count for edge origin and destination vertex on that edge. We then iterate through those counts, and if that count is odd, we add that vertex to our set of odd vertices.

```
ODD_VERTICES_COUNT = {}
For each EDGE in EDGES:
        Count that EDGE as incidental to ORIGIN AND DESTINATION.
        ODD_VERTICES_COUNT[DESTINATION] += 1
        ODD_VERTICES_COUNT[ORIGIN] += 1

Return ODD_VERTICES_COUNT
```

Next, we **generate a minimum perfect matching from our odd vertices**. The perfect matching of a set of a vertices is the set of edges which connect all vertices exactly once, such that all the edges returned have no common vertices among them. We minimize this graph such that the sum of the edges is as low as possible. In our implementation we create a stack of odd numbered vertices from our previously derived set, and pop each vertex off the stack. For that vertex, we find it's nearest neighbor by finding the minimum weight edge in our edges. We then remove that nearest neighbor from our stack, and add that edge to our minimum perfect matching set. This method of producing a minimum perfect matching is efficient, and easy to implement, but it is not guaranteed to produce the absolutely minimized perfect matching. First, the list of vertices with an odd number of edges is shuffled to produce a non-deterministic result. Each time we find the minimum edge leaving this vertex, we remove the destination vertex from circulation. This is the locally optimal choice, but in some cases removing that destination vertex might cause the result to not be globally optimal. In theory, we could use a more optimal, but difficult to implement, algorithm like the Hungarian, or Kuhn-Munkres algorithm to produce

a more optimal solution in polynomial time. Next, we combine our edges from our minimum perfecting matching and our minimum spanning tree to form a multigraph where each vertex has an even number of edges.

```
RESULT = []
While length of ODD_VERTICES > 0:
        ORIGIN_VERTEX = ODD_VERTICES.pop()
        Find the least costly edge from ORIGIN_VERTEX to some DEST_VERTEX.
        Remove DEST_VERTEX from ODD_VERTICES
        Append that edge to RESULT

Return RESULT
```

Now we wish a **compute a euler cycle** over our multigraph, where each edge is used exactly once. Using a dictionary of neighbors per each vertex in the multigraph, we iterate through the list of edges in our minimum spanning tree multigraph. We pick a vertex, and add it to our result list. While our multigraph edge list has edges, we enumerate through our result list, and while our neighbor dictionary for that result vertex has vertices we enumerate through edges in our minimum spanning multigraph, and if that edge is composed of our origin and destination vertex, we should delete that edge from our minimum spanning multigraph. From our neighbor lists, we should delete each vertex from the others' list, and add the destination vertex to our result, and set our new origin vertex as the destination vertex.

```
With a dictionary of NEIGHBOR vertices per each vertex
START_V = an origin vertex from our minimum_spanning_muligraph
RESULT = [some neighbor of ORIGIN_VERTEX]
While our minimum spanning multigraph still has edges in it:
        for each result_vertex in RESULT:
                While that result_vertex has neighbors:
                        destination = neighbor of result_vertex
                        Delete the edge result → destination from min-span-multigraph.

        Delete destination from origin's neighbors
        Delete origin from destinations neighbors.

        Append destination to result.

        RESULT_VERTEX = destination

Return RESULT
```

Finally we iterate through the previously computer euler cycle and removes vertices that have already been visited, and then iterate through that shortened path to find the sum of the weights from one vertex to the next. This is our solution to the traveling salesman problem.

In our real-world implementation, Christofides' algorithm has variable performance. The algorithm as implemented encounter significant performance bottlenecks while building the graph from the provided data, running through each provided vertex per each vertex (since all vertices have edges to all other vertices), resulting in a runtime of $O(V^2)$. As it stands, the runtime characteristic of the algorithm is dominated by the amount of work required to construct the graph (as it is implemented), and so it has a runtime of $O(V^2)$.

Against our test graphs, Christofides' only notable achievements are optimal results for the largest datasets, at the expense of longer runtimes. When taken in context, these achievements only represent marginally more optimal results, and significantly longer runtimes. As an example, Christofides' as implemented achieves a 1.224 approximation ratio in 791 seconds over 15,112 vertices, whereas Nearest Neighbor nets a 1.249 approximation reaction in 225 seconds over the same graph. Over 5000 vertices, Nearest Neighbor nets 63981 optimal distance in 30 seconds against Christofides' computing a 62221 optimal distance in 75 seconds. Assuming the lower number is more accurate, we can observe a pattern of Christofides' computing marginally more accurate distances in more than twice the time of Nearest Neighbor, resulting in a dubious tradeoff of complexity and accuracy.

## *2-Opt Tour Optimization:*

2-opt is a way of improving an already found tour. The premise is fairly simple. You just traverse a tour, swapping two edges at a time. If the edge swap returns a shorter distance then you keep the new tour. Otherwise you revert back to the last best version. In order to test all possible edge swaps, the algorithm starts at the first edge and swaps it with each subsequent edge. It then does the same with the second, third, fourth, and so on until it reaches the last edges. These loops are also encased in a loop, which stays active until improvements are no longer being made (Nilsson, C.).

For our algorithm we will need to pass in an already found tour, and a list of city coordinates.

```
opt_2(tour, cities) :
        size = tour length
        best_dist = distance of starting tour
        i = 2
        while i < 2:
                j=0
                while j < size-1:
                        k=j+1
                        while k < size:
                                new_tour = []
                                swap(j, k, tour, new_tour)
                                new_dist = distance of new_tour
                                if new_dist < best_dist:
                                        i = 0
                                        tour = new_tour
                                        best_dist = new_dist
                        k++
                j++
```

```
        k++
        return [best_dist, tour]

swap(one, two, tour, new_tour):
        size = tour length
        i = 0
        for i = 0 up to but not including one
                append tour[i] to new_tour
        goBack = 0
        for i = one to two
                append tour[two-goBack] to new_tour
                goBack++
        for i = two + 1 up to but not including size
                append tour[i] to new_tour
```

*Other Researched Algorithms:*

**Arora's PTAS:** We looked into one algorithm discovered by Sanjeev Arora, called Arora's Polynomial Time Approximation Scheme (PTAS). This algorithm is supposed to have exceptionally efficient running times. The general idea was to create a grid, move the input points to the nearest grid point, partition the grid into small subproblems, and then use dynamic programming. In practice this algorithm seemed to be very arduous, and is very rarely implemented (Zachariasen, M., 2006).

**Dynamic Programming:** Given that the 1st test instance for the competition was only 50 cities, we thought maybe we could get an exceptionally accurate tour using dynamic programming. However running time for a dynamic programming algorithm is generally O(n^2 * 2^n). This means that a dynamic programming algorithm would be 2^n times slower than the Nearest Neighbor algorithm, which we decided would most likely be way too slow to meet the 3 minute time requirement ("Travelling Salesman Problem | Set 1", 2017).

## Algorithm Implementation

Our team ended up implementing all of our algorithms. However, for our final version, we decided to stick with using the Nearest Neighbor with the 2-opt optimization for smaller datasets. This provided the best solution in the majority of instances that we tested, and for those that it did not, it was fairly close to the best.

**Nearest Neighbor Pseudocode:**

```
        // unvisited = array of city id's
        // distance = array of x and y locations
        nearestNeighbor(unvisited, distance, startingCity)
          path = []
          distance = 0
          path[0] = unvisited.remove(startingCity)
```

```
    while unvisited.length > 0:
        nearestCity = getNearest(unvisited, distance, path[len – 1])
        path.append(unvisited.remove(nearestCity[1]))
        distance += nearestCity[0]

    return [path, distance]

// Run nearest neighbor using every city as a starting point
greedyNeighbor(unvisited, distance)
    bestPath = null
    distance = infinity

    for i in unvisited.length:
        path = nearestNeighbor(unvisited, distance, unvisited[i])

        if path[1] < distance:
            bestPath = path[0]
            distance = path[1]

    return [distance, bestPath]
```

**2-opt Pseudocode:**

```
//tour = and already found tour
//cities = list of city coordinates.

opt_2(tour, cities) :
        size = tour length
        best_dist = distance of starting tour
        i = 2
        while i < 2:
                j=0
                while j < size-1:
                        k=j+1
                        while k < size:
                                new_tour = []
                                swap(j, k, tour, new_tour)
                                new_dist = distance of new_tour
                                if new_dist < best_dist:
                                        i = 0
                                        tour = new_tour
                                        best_dist = new_dist
                        k++
                j++
        k++
```

```
        return [best_dist, tour]

swap(one, two, tour, new_tour):
        size = tour length
        i = 0
        for i = 0 up to but not including one
                append tour[i] to new_tour
        goBack = 0
        for i = one to two
                append tour[two-goBack] to new_tour
                goBack++
        for i = two + 1 up to but not including size
                append tour[i] to new_tour
```

**Results**

| Example Cases | Distance | Time (seconds) |
|---|---|---|
| **tsp_example_1.txt** | **116317** | **2** |
| **tsp_example_1.txt** | **2744** | **173** |
| **tsp_example_1.tx** | **1964948** | **296** |

| Competition Cases | Distance | Time (seconds) |
|---|---|---|
| **test-input-1.txt** | **5639** | **1** |
| **test-input-2.txt** | **7523** | **6** |
| **test-input-3.txt** | **12846** | **166** |
| **test-input-4.txt** | **19711** | **62** |
| **test-input-5.txt** | **27128** | **102** |
| **test-input-6.txt** | **39764** | **126** |
| **test-input-7.txt** | **62624** | **121** |

## References


(n.d.). Retrieved March 09, 2018, from https://www.coursera.org/learn/algorithms-on-graphs/lecture/x2FM2/prims-algorithm

Barnwal, A. (n.d.). Greedy Algorithms | Set 2 (Kruskal's Minimum Spanning Tree Algorithm). Retrieved March 11, 2018, from https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/

Christofides algorithm. (2017, July 11). Retrieved March 11, 2018, from https://en.wikipedia.org/wiki/Christofides_algorithm

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. Cambridge (Inglaterra): Mit Press.

Depth-First Search in Python. (n.d.). Retrieved March 09, 2018, from http://www.koderdojo.com/blog/depth-first-search-in-python-recursive-and-non-recursive-programming

Greedy Algorithms | Set 6 (Prim's MST for Adjacency List Representation). (2017, June 20). Retrieved March 09, 2018, from https://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-mst-for-adjacency-list-representation/

Kruskal's algorithm. (2018, March 14). Retrieved March 14, 2018, from https://en.wikipedia.org/wiki/Kruskal's_algorithm

Nearest Neighbour Algorithm. (2018, Jan. & feb.). Retrieved March 07, 2018, from https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

Nilsson, C. (n.d.). *Heuristics for the Traveling Salesman Problem*(Tech.). doi:https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf

Notes on Matching. (2012, September 8). Retrieved March 11, 2018, from http://www.cs.cornell.edu/courses/cs6820/2014fa/matchingNotes.pdf

Ribeiro, P. (n.d.). Eulerian Tour. Retrieved March 11, 2018, from http://www.dcc.fc.up.pt/~pribeiro/estagio2008/usaco/3_3_EulerianTour.htm

Roughgarden, T. (2016, February 25). CS261: A Second Course in Algorithms Lecture #16: The Traveling Salesman Problem. Retrieved March 11, 2018, from http://theory.stanford.edu/~tim/w16/l/l16.pdf

Saiyed, A. R. (2012). The Traveling Salesman problem. Indiana State University, 5-9. Retrieved March 7, 2018, from http://cs.indstate.edu/~zeeshan/aman.pdf

Travelling Salesman Problem | Set 2 (Approximate using MST). (2017, March 22). Retrieved March 09, 2018, from https://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/

Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming). (2017, July 10). Retrieved March 16, 2018, from https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

Zachariasen, M. (2006). *Metric and Euclidean TSP*(Tech.). doi:http://www.diku.dk/OLD/undervisning/2006-2007/2006-2007_b2_415/tspappr.pdf