

# CS325 Final Project Report

Nathan Perkins, Eric Newton, Ziwei Wu

June 8, 2018

## Introduction

The traveling salesman problem or TSP is a classic problem that's been studied in computer science since the 1930s. The problem is commonly phrased as the following: A traveling salesman needs to visit a set of cities and make sales. He can only visit each city once and by the end of the trip, he needs to make his way back to the start city. If he wants to minimize the total distance traveled, what is the optimal set of paths he should travel. The problem looks simple from the description. But it is an NP-complete problem - a set of problems without polynomial solutions. This means that even given the most powerful computer available, with sufficient large problem size, the computation for the optimal solution takes years.

This is where approximation algorithms come into play. If we are not seeking the absolute optimal solution, but a solution that's close enough to the optimal solution. We can employ a polynomial bounded algorithm to compute a suboptimal solution. In this report, we are investigating three approximation algorithms including minimum spanning tree, Ant Colony Algorithm, and Nearest-Neighbor with 2-OPT optimization. We first take a look at algorithms researched.

## Algorithms Research

### Minimum Spanning Tree

A minimum spanning tree or MST of Graph  $G = (V, E)$  is a spanning tree connecting all vertices of  $V$  such that the sum of edge's weights is as minimized. It is a polynomial bounded algorithm. Let's see how it can be applied to approximate a solution for TSP, we can break down the overall algorithm into three steps,

1. Find the minimum spanning tree of the cities,
2. Perform a depth-first search of the resulting tree,
3. Define the tour by order of vertices being discovered by DFS.

The resulting tour is at most twice the length of optimal TSP. In practice, the results are usually much better, anywhere 15% to 20% over the optimal solution. There are several ways

to find an MST including Prim's algorithm, Kruskal's algorithm, and Boruvka's algorithm. Out of three, Prim's algorithm was chosen due to its simple implementation. Another factor is that with a binary heap, it has a runtime of  $O(E \log V)$  which is sufficiently fast. Here is the Pseudocode: The array Q is initialized to contain all of the vertices. The algorithm

---

**Algorithm** Prim's (G(U,V))

---

```

for each vertex v in V do
     $key[v] = \infty$ 
     $parent[v] = NULL$ 
    insert v into Q
end for
 $key[0] = 0$ 
while Q is not empty do
     $v = Q.removeMin()$ 
    for u adjacent to v do
        if  $u \in Q$  and  $weight(u, v) < key[u]$  then
             $parent[u] = v$ 
        end if
         $key[v] = weight(u, v)$ 
    end for
end while

```

---

starts at vertex 0 with a key of 0 as the initial vertex and iterates through each vertex v not yet in the tree choosing the minimum edge weight already in the tree. Thus the adjacent vertex with the minimum edge weight is chosen and vertex v is added to the tree. This continues until Q is empty and all vertices have been added to the MST. We can use the MST graph we've produced with Prim's as input and assume the graph is already connected since we have found the MST. Therefore performing the DFS by visiting each vertex in the graph is simple, pseudocode as follows: Start with some vertex u. We can complete our

---

**Algorithm** DFS(G of MST, u)

---

```

 $u.visited = true$ 
ENQUEUE(u)
insert v into Q
for each  $v \in G.Adj[u]$  do
    if !v.visited then
        DFS(G,v)
    end if
end for

```

---

overall algorithm by adding discovered vertices to queue Q. Q now contains the vertices of the MST in the order they were discovered, which is our solution for TSP.

## Ants Colony Algorithm

Ants Colony Algorithm is an interesting algorithm that takes inspiration from nature. When there is a food source available for ants to forage. The ants are able to find the shortest path to the source over time. The mechanism is as the follows: at the beginning, each path leads to the food source has an equal probability of being traversed by the ants. As ants traverse the path, they leave trails of pheromone. As ants find their way back home, the shortest path would allow the ants to return quicker. The new patch of ants leaving from the starting point would have a higher probability to traverse the path with the higher level of pheromone. Over time, the shortest path would have the highest level of pheromone, and the less often traveled path would have pheromone slowly decreasing. Eventually, all ants would only traverse via the shortest path to the food source. This process can be considered as a positive feedback loop. The shortest path is reinforced over time.

Now let's apply this idea to TSP. Given a TSP with  $n$  cities and with distance  $d_{ij}$ . We distribute the ants randomly to each city. Let's assume that ants have memory and can remember what cities they have visited and would not visit them again. The ants would prefer to travel to the closer cities. The probability that a city  $j$  selected by ant  $k$  after city  $i$  is,

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{S \in U_k} [\tau_{iS}]^\alpha \cdot [\eta_{iS}]^\beta} & j \in U_k \\ 0 & otherwise \end{cases} \quad (1)$$

$\tau_{ij}$  is the intensity of pheromone between city  $i$  and the city  $j$ .  $\alpha$  is the parameter to regulate  $\tau_{ij}$ .  $\eta_{ij}$  is the closeness factor of the city  $i$  from city  $j$  and set to  $1/d_{ij}$ , where  $d_{ij}$  is the distance between city  $i$  and  $j$ .  $\beta$  is the parameter to regulate  $\eta_{ij}$ .  $U_k$  is the set of unvisited cities for each ant  $k$ .

By intuition, starting with  $l$  ants random distributed to each city. After  $n$  iterations ( $n$  is the number of cities), each ant has completed a tour. The shorter tour would have a higher chance of being traveled by more ants. We need a set of equations to model the pheromone level of each trail between cities as they are being traversed by each ant  $k$ .

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij} \quad (2)$$

$$\Delta\tau_{ij} = \sum_{k=1}^l \Delta\tau_{ij}^k \quad (3)$$

$$\Delta\tau_{ij}^k = \begin{cases} Q/L_k & \text{if ant } k \text{ travels on edge } (i, j) \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

Looking at equation 4, the function  $Q/L_k$  represents the increase in pheromone level, where  $Q$  is a constant and  $L_k$  is the length of the tour by ant  $k$  from city  $i$  and  $j$  in one iteration. If no ants traveled the path, the change is zero. Equation 3 represents the total increase

of pheromone for a path after taking account of all ants travel through the path after one iteration. Equation 2 is the update function,  $\rho \in [0, 1]$  is the regularizing parameter for  $\tau_{ij}$ . Given the sets of above equations, Let's take a look at the Pseudocode of Ant Colony Algorithm for solving TSP.

---

**Algorithm** Ant Colony Algorithm

---

Let  $U_k = \{x | x \in X \text{ and } x \notin G, \exists G \in tabu_k\}$

---

1. Initialize  
Set  $T=0$   
For every edge(i,j) set an intial  $\tau_{ij} = c$  for trail density and  $\delta_{ij} = 0$
  2. Set  $s=0$   
For  $k = 1$  to  $l$   
Place ant  $k$  on a city randomly. Placed city in  $visited_k$ .  
Place the group of city in  $tabu_k$
  3. Repeat until  $s \leq m$   
Set  $s = s + 1$   
For  $k = 1$  to  $l$   
Choose the next city to be visited according to the  $p_{ij}^k$  by equation 1  
Move the ant  $k$  to the selected city  
Insert the selected city in  $visited_k$   
Insert the group of selected city in  $tabu_k$
  4. For  $k = 1$  to  $l$   
Move the ant  $k$  from  $visited_k(n)$  to  $visited_k(l)$   
Compute the tour length  $L_k$  traveled by ant  $k$   
Update the shorest tour found  
For every edge (i,j)  
For  $k = 1$  to  $l$   
Update  $\tau_{ij}$  according to equation 2 to 4  
 $T++$
  5. If  $(T < TMax)$   
Empty all  $visited_k$  and  $tabu_k$   
Goto Step 2  
Else  
Print the shortest tour  
Stop
-

# Nearest Neighbor with 2-OPT Optimmmization

## Rationale for Choosing This Algorithm

This is our algorithm of choice for implementing an approximation algorithm. A big reason is that it is a very intuitive and simple algorithm. We were able to grasp the core concepts of algorithm very quickly. It is also fairly straightforward to implement, which allows us to get problems running quicker. The overall algorithm consists of two sub-algorithms, Nearest-Neighbor and 2-OPT optimization.

## Algorithm Description

The idea of Nearest-Neighbor is that given a Graph  $G(E,V)$ , we start at vertex  $v^0$  and keep going to the nearest, unvisited vertex until all vertice are visited. Finally, we can complete the tour by going back to the first vertex. This is clearly a greedy algorithm, and such that our solution may not be close enough to optimal solution. This is where 2-OPT optimization can improve the algorithm.

2-OPT is an optimization algorithm that can be run on an approximate solution to improve it iteratively. The main idea is that the algorithm will search for edges in the route that cross over each other and then swap the route amongst 4 vertexes so that they no longer cross. This will always improve the path because a route that contains a cross-over is always longer than the resulting route that removes the cross-over. An optimal solution for TSP will never contain a crossover.

There are situations in which swapping a route amongst 4 vertexes cannot fully remove a crossed path. Similar to double rotations in AVL trees, a single 2-OPT swap among 4 vertexes will leave the path crossed with some other vertexes. Then swapping those vertexes might cross it back again. 3-OPT is a further improvement that will analyze the path and make double swap as necessary. This can further optimize the route. Here are the pseudocodes. With 2-OPT algorithm in place, we are ready to write the implementation to test

---

**Algorithm** 2optSwap(path, i, k)

---

Let newRoute be an empty array  
Add  $route[0 : i - 1]$  to newRoute in order  
Add  $route[i : k]$  to newRoute in reverse order  
From  $route[k + 1]$  to the end of the route to newRoute in order

---

out the tour cases, we will refer the algorithm as 2-OPT from now on.

## Implementation

We implemented the 2-OPT algorithm using C++. The algorithm itself is fairly straightforward to the program. We did spend a big chunk of time creating functions to extract problem from the input files. In terms of data structure, we used vectors to store the adjust

---

**Algorithm** TSP(V,E)

---

```
let selected_V be an empty array
let path be an empty array
sort E in descending order of edge weight
while the length of selected_V is less than the length of V do
    find the next edge in E that doesn't have both connected vertice already selected
    add that edge to current path
    if either connected vertex is not in selected_V, add it
end while
while improvements are being made do
    improve_made = false
    best_distance = totalDistance(path)
    for i from 1 to number of nodes that can be swapped - 1 do
        for k from 1+1 to number of nodes that can be swapped - 1 do
            new_path = 2optSwap(path, i, k)
            new_distance = totalDistance(path)
            if new_distance < best_distance then
                path = new_path
                improvement_made=true
            end if
        end for
    end for
end while
```

---

edge list and created a struct to represent cities. With the implementation, we computed the solution for the example cases and competition cases.

## Results

### Example Cases

Table 1: 2-OPT for Solving Example Cases

Tour	Best Greedy Distance	Total Duration(seconds)	Approximation Ratio
1	130921	0.0251	1.2104
2	2975	0.0357	1.1535
3	1917050	66.3062	1.2180

2-OPT performed well for the three example cases. Example 1 and 2 only took trivial time to compute. Example 3 took 66 seconds which is a reasonable time. All the approximate ratios are in the 1.15 to 1.22 range, which is very close to the optimal solution. Let's take a look at the competition cases.

## Competition Cases

Table 2: 2-OPT for Solving Competition Cases

Tour	Best Greedy Distance	Total Duration(seconds)
1	5911	0.0264
2	8011	0.0266
3	14826	0.0500
4	19711	0.1251
5	27128	0.7759
6	39469	3.9298
7	61466	55.0527

2-OPT algorithm performed well in the competition cases in terms of runtime. All cases were computed in under one minute, and first five cases were under one second. We are confident that it would perform well in terms of approximation ratio based on the example cases.

## Conclusion

TSP is an NP-complete problem which poses a challenge for finding an optimal solution. TSP has applications in many domains including route planning, manufacturing, and in biology etc. Therefore an approximation algorithm that can provide a close enough approximation to the optimal solution is very valuable. In our research, we investigated three algorithms including Minimum Spanning Tree, Ant Colony, and Nearest Neighbor with 2-OPT optimization. We chose 2-OPT as the algorithm to implement due to its simple implementation. The algorithm performed very well in both the example and competition cases. All tour were computed within a reasonable timeframe and the approximation ratios were all less than 1.25 for the example cases. For our future studies, we would like to investigate more sophisticated algorithms that could further improve approximation ratio.

## References

- [1] Dfs algorithm for graph (with pseudocode, example and code in c, c++, java, python). <https://www.programiz.com/dsa/graph-dfs>. (Accessed on 06/08/2018).
- [2] Optimization - how does the 3-opt algorithm for tsp work? - computer science stack exchange. <https://cs.stackexchange.com/questions/19808/how-does-the-3-opt-algorithm-for-tsp-work>. (Accessed on 06/08/2018).
- [3] Andrius Blazinskas and Alfonsas Misevicius. Combining 2-opt, 3-opt and 4-opt with k-swap-kick perturbations for the traveling salesman problem. *Kaunas University of Technology, Department of Multimedia Engineering, Studentu St*, pages 50–401, 2011.

- [4] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
- [5] David Epstein. Ics 161: Design and analysis of algorithms lecture notes for feb. 8, 1996. *Apr*, 28:1–5, 1999.
- [6] Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.
- [7] Wikipedia contributors. 2-opt — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=2-opt&oldid=827105393>, 2018. [Online; accessed 9-June-2018].
- [8] Jinhui Yang, Xiaohu Shi, Maurizio Marchese, and Yanchun Liang. An ant colony optimization method for generalized TSP problem. *Progress in Natural Science*, 18(11):1417–1422, 2008.