Spring ‘17

# Project 3: TSP

Group #14

Team: Aleita, Philip, Elizabeth
6-9-2017

## TSP Algorithm Research

We have been asked to research three different algorithms for solving the Traveling Salesman Problem. We have decided to explore Nearest neighbor, Christofides algorithm, and the 2-OPT algorithm in hopes of finding an algorithm that meets both performance and optimization requirements for solving this problem in adequate time and with adequately optimal solutions for best case scenarios.

## Nearest neighbor algorithm (Greedy)

A heuristic which is one of the simplest solutions to our Traveling Salesman Problem. This algorithm operates by finding and visiting the nearest city which has not already been visited and traveling to. This method will always choose the shortest route available which has not been traveled before. For this reason, it is a greedy solution. This method of solving the traveling salesman problem should stay within 25% of accepted optimal answer [1]. As mentioned above this is a greedy algorithm which quickly produces an answer in (O(n*2)) time

The nearest neighbor algorithm functions by first sorting the available edges by shortest distance. Then adds the shortest edge to the tour, and of course removing the edge from the available edge list so as to not repeat paths. The algorithm will repeat this process until we are given n edges in the tour. As n edges are reached and added we return to the start location.

**Nearest Neighbor Pseudo-code:**

```
function nearestNeighbor(destinations)
        If start = emtpy
                start = destinations[0];
        travelTo = destinations
        path = [start]
        travelTo.remove(start)
        while travelTo
                nearest = min([distance(path[-1], item) for item in travelTo])
        path.add(nearest)
        travelTo.remove(closest)
  return path
```

**Nearest Neighbor Algorithm References**

[1] https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf
[2] http://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/

## Christofides Algorithm

Published in 1976, the Christofides heuristic algorithm improved upon existing approximation algorithms by guaranteeing that its solutions would be within a factor of 3/2 optimal. For traveling salesman type problems, with cost matrix that satisfy the triangularity condition, the Christofides approach is a significant balance of efficiency and optimality. While not as quick as other approximation algorithms (Nearest Neighbor), Christofides offers a better solution to many problems where accuracy matters more than speed, such as DNA sequencing. Simply stated, the Christofides algorithm has 6 steps:

1. Generate a Minimum Spanning Tree from input graph
2. Find odd degree vertices from MST
3. From odd degrees find the minimum weight matching subgraph.
4. Combine matching subgraph & MST.
5. Find a Euler tour (E) of minimum weight graph.
6. Find the Hamiltonian cycle of E

The above pseudo-code outlined by Wikipedia is good for a quick overview, but once you start working through the individual steps it becomes clear that there are many parts to keep track of and details matter. Rather than write a bulky explanation we have written out a more explanatory pseudo-code of the Christofides algorithm below.

**Christofides Pseudo-code:**

```
Function christofides_alg():
    Input = graph of v cities with coordinates

    Calculate distance between each city (edges) to create edge weight matrix (adjacency
    matrix.) Technically all vertices are adjacent to each other until route is determined.

    Generate Prim's Minimum Spanning Tree:
        <Prim's MST algorithm>
    Find odd vertices in MST:
        for each vertex in MST
            if(sum of edges adjacent to v is odd)
                vertex = odd degree
        return oddVertices

    Determine Minimum Weight Matching:
        Create MW vector
        while(OddVertices != empty set)
            previous vertex = INF
            for each vertex in O
                if next vertex < previous vertex
                previous vertex = next vertex
            push edge into MW
            pop vertex from OddVertices
        return MW
```

*Combine MST & Minimum Graph:*

```
        for each edge in MST
            push edge in Cgraph
        for each edge in MW
            push edge in Cgraph
        return combined graph
```

*Find Euler Tour:*

```
        input = adjacency matrix
        create stack
        set counter i to 0
        while stack is != empty OR size of adjMatrix[i] > 0
            if size of adjMatrix == 0
                path = i
                next = top of stack
                pop/remove top of stack
                i = next
            else
                path = i
                neighbor = last element of adjMatrix
                remove edge of (u,v) and (v,u) in adjmatrix
                push i onto stack
                i = neighbor
        return tour
```

*Turn Euler circuit into Hamiltonian path:*

```
        input = graph & orig-path
        create vector to hold ham-path
        push back path[0] to ham-path
        for i = 0 to size of orig-path
            for k = 0 to size of ham-path
                if orig-path[i] == ham-path[k]
                    delete orig-path[i]
                else
                    push back to ham-path
        return ham-path
```

**Christofides References:**

[1] https://en.wikipedia.org/wiki/Christofides_algorithm

[2] http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/
[3] J. A. Hoogeveen. 1991. Analysis of Christofides' heuristic: Some paths are more difficult than cycles. *Oper. Res. Lett.* 10, 5 (July 1991), 291-295. DOI=http://dx.doi.org/10.1016/0167-6377(91)90016-I
[4] N. Christofides, Worst-case analysis of a new heuristic for the travelling salesman problem, Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.
[5] GeeksForGeeks.org for : Prim's MST, Euler & Hamiltonian
[6] Eulerian Path and Circuit. www.graph-magics.com/articles/euler.php

## 2 -OPT (Croes) Algorithm

The 2-OPT algorithm is an optimization algorithm that has the potential to improve an already established traveling salesperson (TSP) tour. This algorithm was developed by G.A. Croes in 1958 [1]. The premise behind this algorithm is it performs a local search in an established TSP tour rearranging the order of the vertices in the tour checking if there is any overall improvement from each rearranged, newly created path. The rearrangements are made by systemically traversing all vertices in the tour taking segments of the original tour and reversing the order of those vertices to check for overall improvement. The idea is this systematic reversal has the potential to eliminate a crossover of two paths that wastes distance from doubling back across the path. The 2-OPT algorithm consists of a for-loop nested in a for-loop checking if the reversed segment starting range optimizes the overall distance traveled. Because of this iterative execution the improvement ultimately depends on the input tour of the algorithm. Further optimization can occur by looping through the 2-OPT algorithm until no further optimization can occur [2].

**2-OPT Pseudo-code:**

Referenced: [2], [3]

T: The input tour consisting of a valid set of vertices

```
OPT2 ( T ) :
    returnT;                              //Initialize a return optimized tour
    currentDistance = T.total tour distance
    currentPath = T.path                  //Path stored in T
    count = T.size                        //count holds the number of vertices in T


    //Note: These preceding for-loops may be looped for further optimization
    for index = 0 to count-1
            for index2 = index + 1  to count
                newPath = OPT2Swap( index, index2, currentPath )
                tempDistance = newPath.distance
                if  tempDistance < currentDistance
                    currentDistance = tempDistance
```

```
                                    currentPath = newPath
 returnT.path = currentPath                      //Set optimized path to return
 return returnT

//Helper function to OPT2 -  rearranges the paths to check for unwinding routes
OPT2Swap ( index, index2, currentPath )
    newPath;                                     //Initialize a new path to return
    for index3 = 0 to index-1
        newPath.push( currentPath[index] )
    for index4 = index2 to index (dec)           //This reverses the order from the original path
        newPath.push( currentPath[index] )
    for index4 = index2 +1 to currentPath.size
        newPath.push( currentPath[index] )

        return newPath;
```

**2-OPT References:**

[1] https://www.cs.ubc.ca/~hutter/previous-earg/EmpAlgReadingGroup/TSP-JohMcg97.pdf
[2] https://en.wikipedia.org/wiki/2-opt
[3] http://www.technical-recipes.com/2012/applying-c-implementations-of-2-opt-to-travelling-salesman-problems/

## Implementation

We initially choose to do a common heuristic method, nearest neighbor, and Christofides due to research proven efficiencies for timely and effective execution of problem. Unfortunately, after much effort to implement the Christofides algorithm we chose to abandon it because it would not ultimately produce enough of an improvement. We decided our time would best be served improving the Nearest Neighbor algorithm with an option to use 2-OPT (when the test cases aren't too big). After testing we made an educated decision that the results of our greedy algorithm were good, if not better than what we expected to get with the Christofides, and therefore we could fore-go further toil and still be competitive.

From this we learned that more important than always getting the perfect or right answer, is getting the best answer that meets & hopefully exceeds requirements while not wasting resources – time and labor. We all have busy finals schedules with multiple large projects, which we believe to be a valid representation of the real world demands, where we would not have unlimited time to develop one part of a project (be it code or not), but balance all objectives.

### Implementation Pseudocode:

*We are only including a sample of our implementation code as it is long and we have already outlined the principle parts in the research sections. This sample is of our 2nd algorithm which leverages both the Nearest Neighbor and 2-OPT, to provide both a "speedy" and improved optimal solution.*

```
struct Tour algorithm_2(struct Graph inputGraph, int cycles)
{

    struct Tour returnOPT2; //Create a tour to return the final results for output

    //Run the input graph first through the nearest neighbor algorithm
    struct Tour greedyReturn = algorithm_1(inputGraph);

    //Calculate the distance of the initial returned tour
    int greedyDistance = greedyDistance = pathDistnace(greedyReturn.path);

    //count is the number of vertices in the path
    int count = greedyReturn.path.size();

    //Holds the best path calculated thus far.
    vector<struct Vertex> currentPath = greedyReturn.path;

    int optCycles = 0; //Number of optimization cycles

    //Cycle through optimization until number of cycles is equal to input
    while(optCycles < cycles)
    {
        //Best logged distance thus far
        int distnace2Beat = pathDistnace(currentPath);
        //Cycle through all vertices in path
        for(int index = 0; index < count-1; index++)
        {
            //Cycle through all vertices in path in conjuntion with outer loop
            for(int index2 = index+1; index2 < count; index2++)
            {
                //Swap vertices to form newpath without potential path crossings
                vector<struct Vertex> newPath = OPT2Swap(index, index2, currentPath);

                //Calculate new distance along path
                int tempDistance = pathDistnace(newPath);

                //If this new potential path beats the original path in length then make new currentPath
                if(tempDistance < distnace2Beat)
                {
                    currentPath = newPath;
                    distnace2Beat = tempDistance;
                    returnOPT2.tourLength = tempDistance;
                }
            }
        }
        optCycles++; //Increment cycle
    }

    //Build the return tour for output
    //Set best path in return Tour
    returnOPT2.path = currentPath;

    //Cycle through best path generating cities vector
    for(int index6 = 0; index6 < returnOPT2.path.size(); index6++)
    {
        returnOPT2.cities.push_back(returnOPT2.path[index6].name);
    }

    //Calculate and assign the number of cities
    returnOPT2.numCities = returnOPT2.path.size();

    return returnOPT2;
```

*For more detail see separately submitted source code TSP.cpp*

## Results:

**Example Instance Results - No Time Limit**

Tour 1 Results (including time):
Using nearest neighbor along with 2-OTP with 10 cycles.
Algorithm ran in (sec): 0.25
Best total distance: 110282
Ratio: 110282/108159 = 1.0196

Tour 2 Results (including time):
Using nearest neighbor along with 2-OTP with 10 cycles.
Algorithm ran in (sec): 9.71
Best total distance: 2775
Ratio: 2775/2579 = 1.076

Tour 3 Results (including time):
Using only nearest neighbor.
Algorithm ran in (sec): 5.12
Best total distance: 1964948
Ratio: 1964948/1573084 = 1.2491

**Competition Results - 3 Minutes**

Tour 1 Results (including time):
Using nearest neighbor along with 2-OTP with 20 cycles.
Algorithm ran in (sec): 0.16
Best total distance: 5639

Tour 2 Results (including time):
Using nearest neighbor along with 2-OTP with 20 cycles.
Algorithm ran in (sec): 1.05
Best total distance: 7740

Tour 3 Results (including time):
Using nearest neighbor along with 2-OTP with 20 cycles.
Algorithm ran in (sec): 13.37
Best total distance: 12718

Tour 4 Results (including time):
Using nearest neighbor along with 2-OTP with 10 cycles.
Algorithm ran in (sec): 49.05
Best total distance: 18009

Tour 5 Results (including time):
Using nearest neighbor along with 2-OTP with 4 cycles.
Algorithm ran in (sec): 151.92
Best total distance: 25007

Tour 6 Results (including time):
Using nearest neighbor only.
Algorithm ran in (sec): 0.08
Best total distance: 40933

Tour 7 Results (including time):
Using nearest neighbor only.
Algorithm ran in (sec): 0.52
Best total distance: 63780

**Competition Results - Unlimited Time**

Tour 1 Results (including time):
Using nearest neighbor along with 2-OTP with 20 cycles.
Algorithm ran in (sec): 0.16
Best total distance: 5639

Tour 2 Results (including time):
Using nearest neighbor along with 2-OTP with 20 cycles.
Algorithm ran in (sec): 1.01
Best total distance: 7740

Tour 3 Results (including time):
Using nearest neighbor along with 2-OTP with 20 cycles.
Algorithm ran in (sec): 13.79
Best total distance: 12718

Tour 4 Results (including time):
Using nearest neighbor along with 2-OTP with 10 cycles.
Algorithm ran in (sec): 48.23
Best total distance: 18009

Tour 5 Results (including time):
Using nearest neighbor along with 2-OTP with 10 cycles.
Algorithm ran in (sec): 379.29
Best total distance: 25003

Tour 6 Results (including time):
Using nearest neighbor along with 2-OTP with 1 cycle.

Algorithm ran in (sec): 298.15
Best total distance: 37496

Tour 7 Results (including time):
Using only nearest neighbor.
Algorithm ran in (sec): 0.56
Best total distance: 63780