CS 325 Project Group 21

12/01/2017

Christopher Merrill, Debra Wilkie, Kyle Wollman

Descriptions of three different methods/algorithms for solving the Traveling Salesman Problem along with pseudocode:

Method 1: Nearest Neighbor Approximation

The Nearest Neighbor algorithm provides an approximate solution to the travelling salesman problem. It is a greedy algorithm that always selects the city nearest to the current city. Although it is relatively quick (Complexity = $O(n^2)$), it averages a solution that is 25% longer than the actual shortest path.

The concept of the Nearest Neighbor algorithm is as followed:

1. Set an arbitrary vertex as current vertex.
2. Find the shortest edge connecting current vertex and an unvisited vertex V.
3. Set the current vertex to V.
4. Mark V as visited.
5. If all the vertices in domain are visited, end the loop.
6. Go to step 2.

Pseudocode:

Given an array G which represents a graph containing n vertices, the NEAREST-NEIGHBOR algorithm can be implemented as followed:

```
NEAREST-NEIGHBOR (G[0...n]) {
    R[n-1]
    currentVertex = G[0]
    R.pushback(currentVertex)
    allVisited = false

    WHILE (!allVisited) {
        shortestEdgeLength = infinity
        nearestNeighbor = null

        FOR each edge E of currentVertex {
                IF $E_i$.length < shortestEdgeLength {
                    shortestEdge = $E_i$.length
                    nearestNeighbor = Vertex connected to $E_i$
                }
        }
        currentVertex = closestVertex
        R.pushback(currentVertex)
        VISITED(currentVertex)

        IF (CHECK-ALL-VISITED(G)) {
                allVisited = true
        }
    }
    RETURN R
}
```

In this case, R is an array that represents the path taken by the salesmen. The first vertex in G is marked as the current vertex and added to R. We initialize a variable *allVisited* to false, which will be used to indicate when all vertices have been visited by the salesman.

At this point we begin looping through each of the vertices in G until all have been visited. We first check each edge E that is connected to the current vertex, to determine which route is the shortest (in other words, the "Nearest Neighbor" to the current vertex). When the shortest edge is determined, we set the vertex it is connected to as the Nearest Neighbor. We then add it to R as the next vertex in the travelling salesman's route, and mark it as visited. Last we check if all the vertices in G have been visited, if not we continue with the loop.

References:
[1] http://www.people.vcu.edu/~gasmerom/MAT131/nearest.html

[2] https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

Method 2: Christofides Algorithm

Christofides algorithm is a type of heuristic algorithm which is designed to sacrifice optimization for speed and are often used to solve NP-complete problems. The traveling salesman problem is a perfect example of a problem that there is no known efficient way to quickly find a solution, although the solution can be verified when given. Nicos Christofides in 1976 created an algorithm that produces a cost that is a 3/2 approximation (within 3/2 of the optimum) for the traveling salesman problem.

The definition of the Christofides algorithm according to Dictionary of Algorithms and Data Structures is a heuristic algorithm to find a near optimal solution to the traveling salesman problem.
Step 1: Find a minimum spanning tree T
Step 2: Find a perfect matching M among vertices with odd degree
Step 3: Combine the edges of M and T to make a multigraph
Step 4: Find a Euler Cycle in G
Step 5: Find Hamiltonian Cycle visiting every vertex only once

Traveling Salesman Problem:
You are given a set of n cities; your goal is to calculate distances between cities and find an ordering (tour) so that the total distance is minimized.

Step 1: Minimum Spanning Tree
Using Prim's Algorithm from the Introduction to Algorithms
MST-Prim (G, w, r) //G graph; w=weight; r= vertex
All vertices not in the tree are in a priority queue based on the minimum weight connected to a vertex

```
    Q= V[G]; //Q is the priority queue
        for each u ∈ Q
            key[u] = ∞; //setting up a table
            key[r] = 0; //the first vertex starts at zero.
        while(Q != null) //while it is not empty
            u= ExtractMin(Q); //find the minimum in the queue
            for each v ∈ Adj[u] //for each vertex adjacent to u(the min in the queue)
                if (v ∈ Q and w(u,v) < key[v])
                p[v] =u;
                key[v]=w(u, v);
```

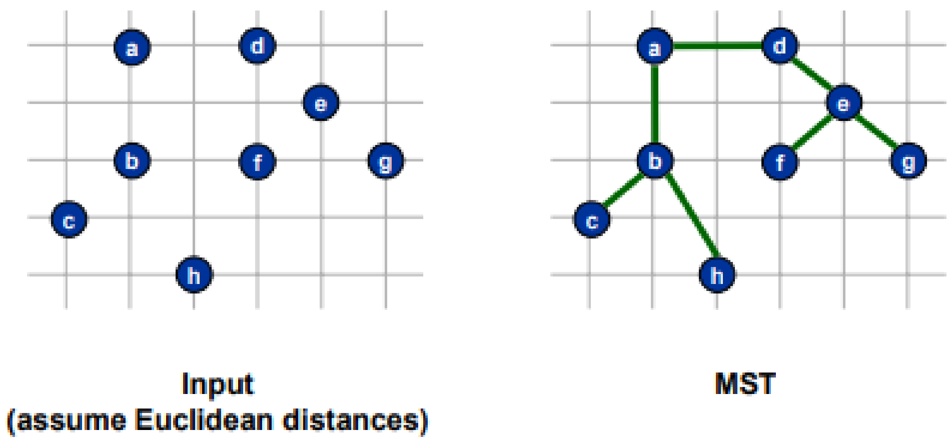


Image from http://www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf

Prim's minimum spanning tree algorithm creates an array Q where all vertices not in the tree are in a priority queue based on the minimum weight connected to a vertex. It iterates through each vertex in the queue, choosing the minimum weight and adding the vertex to the tree until all vertices have been added to the tree.

Step 2: Find a perfect matching among vertices with odd degree

```
PerfectMatching (oddList)  //OddList is the list of odd vertices
    while (oddList is not empty)
        v = oddList.begin()
        length = ∞
        for (u ∈ oddList)
            if (distance(u, v) < length)
                length = distance
                closest = u
        G.addEdge(closest, u)
        oddList.remove(closest)
```

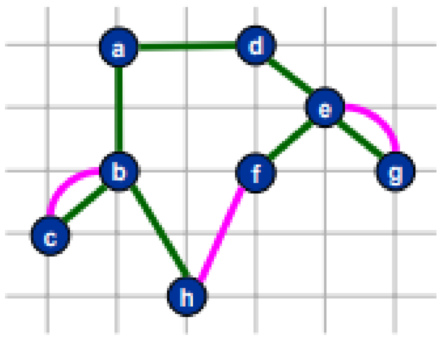
This algorithm finds the perfect match among the odd vertices.

Step 3: Combine the edges of M and T to make a multigraph

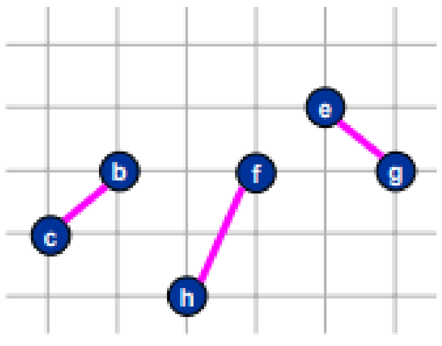
Combine the Minimum Spanning Tree and the Perfect Matching to create a multigraph.

```
CombineGraph (G)
    V[]= G.getallVertices()
    for (u ∈ V)
       E[] = G.getEdges(V[u])
    for (e ∈ E)
       this->addEdge(V[u], E[e])
```

CS 325 Project Group 21
12/01/2017
Christopher Merrill, Debra Wilkie, Kyle Wollman



G' = MST + Matching



Matching M

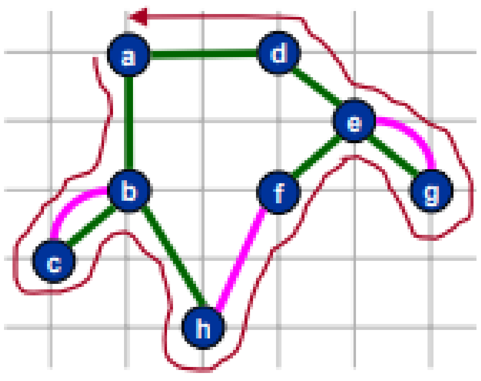
Image from http://www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf

Step 4: Find a Euler Cycle in G
Euler algorithm picks a node if the node has a neighbor delete the edges, if no neighbor add it to the cycle until all the nodes have been processed.

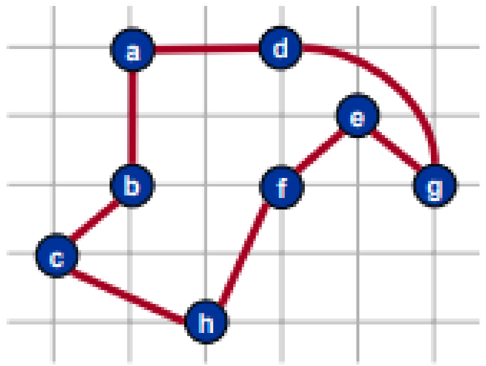
```
 tour(u)
      for each vertex V
           if (edge=(u, v) in E)
              remove e from E
              tour(v)
           else (no edge)
      add u to tour  //add u with no edge and after removing edge
```

Step 5: Find Hamiltonian Cycle
Skipping visited nodes in Euler Circuit

```
HamCycle (G, adj[][])  //graph with adjacency arrays
      path[v] //current permutation
      while(there are still vertices)
           check= true
        for ( v to n-1)
           if adj[path[i], path[i+1]]== false
              check = false
        path = next permutation check
```



E = Eulerian tour in G'



Hamiltonian Cycle H

Image from http://www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf

CS 325 Project Group 21
12/01/2017
Christopher Merrill, Debra Wilkie, Kyle Wollman

References:

[1] Kenny, Vincent, Nathal, Matthew and Saldana, Spencer, *Heuristic Algorithms*. Northwestern University. https://optimization.mccormick.northwestern.edu/index.php/Heuristic_algorithms

[2] Black, Paul, *Christofides Algorithm*, Dictionary of Algorithms and Data Structures. https://www.nist.gov/dads/HTML/christofides.html

[3] Cormen, Thomas H, Leiserson, Charles E, Rivest, Ronald L, Stein, Clifford, *Introduction to Algorithms Third Edition.* The MIT Press, Cambridge, Massachusetts.

[4] Algorithmist, *Euler tour*, http://www.algorithmist.com/index.php/Euler_tour

[5] Wayne, Kevin, *Approximation Algorithms,* Theory of Algorithms. Princeton University. http://www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf

[6] Jaimini, Vaibhav, *Hamiltonian Path.* Hackerearth. https://www.hackerearth.com/practice/algorithms/graphs/hamiltonian-path/tutorial/

Method 3: Held-Karp Algorithm

The Held-Karp algorithm provides an optimal solution to the traveling salesman problem using a dynamic programming approach. In order to use dynamic programming, there must be optimal substructure and overlapping subproblems. In the case of the TSP we consider a subset of cities S that ranges in size from 1 to n, n = number of cities. The optimal substructure is the shortest path D_{1j} from the starting city, call it 1, to some finishing city *j* visiting every city in S exactly once. Since we are starting at 1, ending at *j,* and visiting each city once, we only need to consider the base cases where *j* = 1. In this instance, if S contains just city 1, then the distance will be zero. If S contains other cities, then the value will be infinity because there is no way to go from 1 to 1 through other cities visiting only 1 once. To calculate the distances, start by iterating the size of the subproblem *m* = 2 to *m = n*. For each subproblem m iterate through the subset S and for each j in S that is not equal to 1 calculate the minimum distance from the starting city to some vertex k contained in S such that $k \neq j$, finally adding the distance from *k* to *j*. Vertex *k* represents the city just before the chosen destination *j*. For the largest S of size n we consider the shortest path from 1 to all cities relying on the previously calculated values. The final calculation involves making the finding the minimum distance from all previously considered *j's* back to 1, the starting city. This is the distance of the minimum distance for the TSP.

Pseudocode:

```
Let A = 2 dimensional array indexed by subset S of size {1, 2, ... , n} and destinations j {1, 2, … , n}

A[S, 1] = {0 if S{1} , ∞ if S > 1} //base case

For m = 2 to n //size of subproblem
        for each S size 1 to n
         for each j in S, j ≠ 1
              A[S, j] = min {A[S – {j} , k] + $D_{kj}$} //k contained in S, k ≠ j
return $\min_j$ [A [{1, 2, … , n}, j] + $D_{j1}$
```

There are 2^n choices for S and n choices for the final destination *j* giving $O(n\ 2^n)$ number of subproblems. For each subproblem you have to consider n – 1 *k* vertices, so for each subproblem you are doing O(n) work. The total running time complexity is therefore $O(n^2\ 2^2)$, which is very bad, but significantly better than the O(n!) that it would take to simply solve the problem by brute force.

References:
[1] https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf

[2] https://www.coursera.org/learn/algorithms-npcomplete/lecture/uVABz/a-dynamic-programming-algorithm-for-tsp

Why we selected the Nearest Neighbor algorithm.

We chose to implement the Nearest Neighbor algorithm because it gives a relatively close approximation of the optimal tour within polynomial time. Compared to the Christofides Algorithm it averages a better approximation at 1.25 times optimal compared to 1.5 times optimal. Other algorithms that deliver the optimal solution would not be able to compute the tour in the 3-minute time limit, so we decided not to implement any algorithms, such as Held-Karp, with non-polynomial running time. We were also limited by our time to implement and experience. For these reasons we couldn't implement any algorithms requiring advanced methods such as the Ant Colony System which simulates a large colony of virtual ants that explore many possible paths at once and mark the best paths discovered.

Description and Pseudocode of our implementation.

The first task we needed to accomplish in our implementation was reading in the city data that would be processed. Our code takes the file name storing the city data as a command line argument. In the file the cities are stored one per line. Each line represents the city id, the x coordinate, and the y coordinate. Our code reads each line in the file into a vector representing each city on the map and stores its x and y coordinates along with a third value that represent whether or not that city has been visited on the tour yet. These city vectors are added to a graph vector, which is then passed to the nearest neighbor algorithm by reference, along with an empty vector that will represent the route of the best tour. Before calling the algorithm, we start a clock which we end once the algorithm returns to the calling function. This keeps track of how long it takes to run the algorithm. This information is outputted to the terminal and the path length along with the order of the cities in the tour is written to an output file.

Given an array G which represents a graph containing n vertices and R an empty array to contain the route taken, the NEAREST-NEIGHBOR algorithm can be implemented as followed:

```
NEAREST-NEIGHBOR (G[0...n], R[]) {

    currentVertex = G[random city]
    R.pushback(currentVertex)
    allVisited = false
        VISITED(G[start city])
    WHILE (!allVisited) {
        shortestEdgeLength = infinity
        nearestNeighbor = null
        FOR each edge E of currentVertex {
            If E.visited = false;
            {
                $E_i$.length = sqrt((currentVertex.x - G[i].x)$^2$ +
                        (crurrentVertex.y - G[i].y)$^2$)

            IF $E_i$.length < shortestEdgeLength {
                shortestEdge = $E_i$.length
                nearestNeighbor = Vertex connected to $E_i$
            }
        }
            Length = Length + shortestEdge
        currentVertex = closestVertex
        R.pushback(currentVertex)
        VISITED(currentVertex)

        IF (CHECK-ALL-VISITED(G)) {
            allVisited = true
        }
    }
        finalLength = sqrt((lastCityVistied.x - firstCityVisited.x)$^2$ +
                (lastCityVistied.y - firstCityVisited.y)$^2$)

        Length = Length + finalLength
    RETURN Length
}
```

In this case, R is an array that represents the path taken by the salesmen. We calculate a random starting city and start the tour at the city. The starting city is marked as the current vertex and added to R. We initialize a variable *allVisited* to false, which will be used to indicate when all vertices have been visited by the salesman.

At this point we begin looping through each of the vertices in G until all have been visited. We first check each edge E that is connected to the current vertex, to determine which route is the shortest (in other words, the "Nearest Neighbor" to the current vertex). We calculate the Euclidean distance from the current city to each other city that hasn't been visited yet each time updating the shortest distance. When the shortest edge is determined, we set the vertex it is connected to as the Nearest Neighbor. We then add it to R as the next vertex in the travelling salesman's route, and mark it as visited. Last we check if all the vertices in G have been visited, if not we continue with the loop. In order to check if all vertices have been visited we use a helper function that loops through each city in the graph G, checking its visited value. The first time it finds a city that has not been visited in returns false back to the calling function and the nearest neighbor algorithm continues to execute. Once all cities have been visited we add the final distance from the final city in the tour back to the starting city to the total distance and return that distance back to the calling function. As the graph and route vectors were passed by reference there is no need to return those.

CS 325 Project Group 21

12/01/2017

Christopher Merrill, Debra Wilkie, Kyle Wollman

Our best tours for the three example instances and the time it took to obtain these tours:

Example 1: Best tour utilizing Nearest Neighbor algorithm
```
flip1 ~/algorithms/Group 220% nn tsp_example_1.txt
Time to execute Nearest Neighbor: 0 seconds.
Length of Tour: 130921
```
Ratio: 1.210   Best/Optimal = 130921 / 108159

Example 2: Best tour utilizing Nearest Neighbor algorithm
```
flip1 ~/algorithms/Group 222% nn tsp_example_2.txt
Time to execute Nearest Neighbor: 0.02 seconds.
Length of Tour: 3186
```
Ratio: 1.235   Best/Optimal = 3186 / 2579

Example 3: Best tour utilizing Nearest Neighbor algorithm
```
flip1 ~/algorithms/Group 224% nn tsp_example_3.txt
Time to execute Nearest Neighbor: 73.21 seconds.
Length of Tour: 1959085
```
Ratio: 1.245   Best/Optimal = 1959085 / 1573084

Our best solutions for the competition test instances:

Competition testing results utilizing Nearest Neighbor algorithm:

Group 21			
Test Case	**Size**	**Min Distance**	**Time**
1	50	6421	0.00sec
2	100	8594	0.00sec
3	250	15821	0.01sec
4	500	21597	0.07sec
5	1000	27912	0.29sec
6	2000	41732	1.15sec
7	5000	63054	7.50sec