The Travelling Salesman Problem Group Project
CS352 – Winter 2018
Group 45

Brad Besserman, Yau Chan, Marc Tibbs
Due Date: March 16, 2018

1. Introduction

The travelling salesman problem is a famous NP-hard optimization problem. Given a list of cities and the distance between cities, the problem seeks an answer to the shortest possible route that visits each city and then returns to the original city. This problem has been intensively studied, and many unique algorithms have been developed to solve the problem.

This report evaluates four solutions to the travelling salesman problem in detail: the nearest neighbor algorithm, the Christofides algorithm, the simulated annealing algorithm, and the Held-Karp algorithm. The nearest neighbor algorithm, simulated annealing and Christofides algorithm are all approximation algorithms that run in polynomial time. The nearest neighbor algorithm is a basic greedy algorithm to solve the problem, while the Christofides algorithm is noteworthy because it was the first algorithm discovered to guarantee a solution that was at most 3/2 times the actual solution. The simulated annealing algorithm mimics the natural process of cooling metal to help guide it to a quick and simple solution. The Held-Karp algorithm is different from the other two solution because it is a precise solution to the problem. The algorithm uses dynamic programming and runs in $O(2^n n^2)$ time. The report also summarizes the results of our team's implementation of the nearest neighbor algorithm on a variety of test cases.

2. Nearest Neighbor Algorithm

The nearest neighbor algorithm is a greedy algorithm that solves the traveling salesman problem by continually choosing the nearest unvisited city to the current city until all cities have been visited. True to its nature as a greedy algorithm, the algorithm runs quickly and effectively. When given randomly generated city data the greedy algorithm will return a solution that is 20-25% longer than the optimal solution.

Test File	Algorithm Solution	Optimal Solution	Ratio
tsp_example_1.txt	130,921	108,159	1.21

tsp_example_2.txt	2,808	2,579	1.09
tsp_example_3.txt	1,934,200	1,573,084	1.23

While the greedy algorithm does run quickly and is easy to implement, there are some arrangements of cities which can make the nearest neighbor algorithm give the worst route. For example, it has been shown that ”for every n ≥ 2 there is an instance of ATSP (STSP) on n vertices for which [the greedy algorithm] finds the worst tour.” [1]

We chose the nearest neighbor (greedy) algorithm because it is relatively easy to implement and still works quickly and efficiently. With an average solution of 20-25% worse than the optimal solution, the algorithm also meets the requirements for this project.

Pseudocode for this algorithm follows:

```
def distance_squared(c1, c2):
    return (c1['x'] - c2['x'])**2 + (c1['y'] - c2['y'])**2

# cities is an array of city objects which have an id, x-coordinate, and y-coordinate
properties.
def get_nearest_neighbor(cities, city):
    # Dictionary for selecting nearest neighbor
    neighbors = {}

    # Add all distances_squared to neighboring cities to dictionary
    for neighbor in cities:
        neighbors[distance_squared(city, neighbor)] = neighbor

    # Return neighbor with least distance
    nearest_neighbor = neighbors[min(neighbors)]
    distance = int(round(sqrt(min(neighbors))))

    return nearest_neighbor, distance

def TSP_nearest_neighbor(cities):
    tour = []
    min_distance = infinity

    for city in cities:
        total_distance = 0

        # Start on arbitrary vertex.
        visited = [city]
        unvisited = []

        # Add all cities to unvisited list except the starting city.
        for city in cities:
```

```
            if city is not city:
                unvisited.append(city)

        # find an unvisited nearest neighbor, marked it visited, and add it's distance.
        while len(unvisited) > 0:
            nearest_neighbor, neighbor_distance = get_nearest_neighbor(unvisited,
visited[-1])
            visited.append(nearest_neighbor)
            unvisited.remove(nearest_neighbor)
            total_distance += neighbor_distance

        # add the distance between the first and last city to complete the tour.
        total_distance += round(sqrt(distance_squared(visited[0], visited[-1])))

        if total_distance < min_distance:
            tour = visited
            min_distance = total_distance

    return tour, min_distance
```

3. Christofides Algorithm

The Christofides algorithm is an approximation algorithm for the travelling salesman problem originally developed by Nicos Christofides in 1978 at the Imperial College of London. This algorithm is a 3/2 approximation algorithm, which means that it is guaranteed to give a solution within 3/2 of the optimal solution length. The algorithm runs in O(n^2 log(n)) time. [2] The Christofides algorithm performs multiple transformations on the original graph, and then calculates the Euler tour of the transformed graph. The Christofides algorithm is described in more detail below.

First the algorithm must be create a minimum spanning tree (T) of the graph (G). This can be done with Prim's algorithm, Kruskal's algorithm, or more efficient greedy algorithms for specific graph conditions.

The algorithm must then compile in a set the vertices in T that have an odd degree (O).

The algorithm then must find a minimum weight perfect independent edge set (M) from the induced subgraph of the vertices in O. An induced subgraph is a subgraph that contains a subset of the vertices, as well as all the edges that have both endpoints in the subset of vertices. An independent edge set is the set of edges in a graph without common vertices. A perfect independent edge set is an independent edge set that matches all vertices of the graph. The algorithm is guaranteed to find a perfect independent edge set because of the handshaking lemma, which states that every finite undirected graph has an even number of vertices of odd degree. This means that O has an even number of edges. Because O is an induced subgraph with even number of edges, there will always be a perfect independent edge set.

The algorithm must then combine the edges of M and T to form a graph J.

The algorithm must then form a Euler circuit (E) from J. A Euler circuit can be calculated using Hierholzer's algorithm.

Finally, the algorithm must convert E into a Hamiltonian circuit H by removing repeated vertices through a process known as shortcutting. When a repeated vertex is encountered, it is removed, and the edge before and after the removed vertex are connected by an edge. H is the approximate solution to the travelling salesman problem.

The traveling salesman algorithm is a 3/2 approximation, which is the best currently known approximation for the travelling salesman problem. The major drawback to the algorithm is that it is significantly more complex to implement than other approximations such as the nearest neighbor algorithm.

We selected the Christofides algorithm for research because it was the first 3/2 approximation algorithm for the travelling salesman problem. It is guaranteed to produce a solution within 3/2 of the optimal solution, and it has historical significance in the research of this problem. Ultimately, the team decided to implement the Nearest Neighbor algorithm instead of the Christofides algorithm because the Nearest Neighbor algorithm was less complex to implement.

Pseudocode for the Christofides algorithm follows:

```
# calculate approximate solution to TSP for a graph of cities
def TSP_christofides(cities):
   # final approximate TSP tour
   tour = []

   # calculate minimum spanning tree from cities graph
   # can be done with Prim's algorithm or Kruskal's algorithm
   T = minimum_spanning_tree(cities)
   # find the set of odd vertices in the MST, simple to find if graph is in adjacency list
format
   O = odd_vertices(T)
   # find the subgraph of cities that only contains the vertices in O and edges with both
endpoints in O
   I = induced_subgraph(cities, O)
   # find a perfect independent edge set (matching) of the induced subgraph
   # can be calculated using the Blossom algorithm
   M = perfect_independent_edge_set(I)
   # combine the MST and the perfect matching into a multigraph
   J = create_multigraph(T, M)
   # find a Euler circuit
   # can be calculated using Hierholzer's algorithm
```

```
    E = euler_circuit(J)
    # reduce the Euler circuit to a Hamiltonian path by shortcutting all duplicated
vertices
    H = shortcut_euler(E)

    return H
```

4. Held-Karp Algorithm

The Held-Karp Algorithm (or Bellman-Held-Karp) is a dynamic programming approach to solving TSP. It was proposed in 1962 by Richard Bellman and separately by Michael Held and Richard Karp [3]. This algorithm is not an approximation, but a precise solution that has an exponential runtime complexity of $O(2^n n^2)$ as there are 2^n sub-problems with n^2 work each.

What makes this a DP algorithm is the fact that every sub-path (sub-problem) of the minimum distanced-tour is itself, a minimum distance. It is considered one of the fastest precise solution to TSP found so far. However, it will not be our chosen algorithm as it is still very slow. Despite this, it was still necessary as part of our TSP algorithm research due to it being one of the earliest solutions to TSP (in addition, it is what panel 2 is referring to in the XKCD comic for this project).

The algorithm takes two inputs, the starting node/city and a list of nodes to visit. From here, we do the following:

1. Base case – no more nodes to visit, return the distance to the starting point
2. More nodes to visit, reduce the sub-problems:
    a. Consider each new neighbor node as a starting point
    b. Remove the new neighbor node we are considering
    c. Calculate the cost of visiting this new node and the cost from the previous node visits (*recursive part)*
    d. Return the minimum result from part c

```
def HeldKarpAlg(int startVertex, HashSet<int> CitiesSet, Node firstCity):
       if !CitiesSet.Any()    //Base case
              firstCity.NeighborCities = new Node[1] { new Node { Value = _vertices.First(),
Selected = true } };
              return _adjacencyMatrix[startVertex, 0];    //Return distance between starting
city and vertex

       double totalCost = double.MaxValue;
       int i = 0;
       int selectedIdx = i;
```

```
        firstCity.NeighborCities = new Node[CitiesSet.Count()];

        for destination in CitiesSet    //touring occurs here
                firstCity.NeighborCities[i] = new Node { Value = destination };
                double costOfVistingCurrentCity = _adjacencyMatrix[startVertex, destination];

                var newCitiesSet = new HashSet<int>(CitiesSet);
                newCitiesSet.Remove(destination);
                double costOfVisitingOtherCities = HeldKarpAlg(destination, newCitiesSet,
firstCity.NeighborCities[i]);//New sub-problem w/ next set of neighbors

                //Calculate cost of visiting new city and others so far
                double currentCost = costOfVistingCurrentCity + costOfVisitingOtherCities;

                if totalCost > currentCost
                        totalCost = currentCost;
                        selectedIdx = i;

                i++;

        firstCity.NeighborCities[selectedIdx].Selected = true;

        return totalCost;    //return the minimum total cost
```

5. Simulated Annealing

The simulated annealing algorithm is named after the metallurgic process of annealing and mimics the process undergone by misplaced atoms in a metal when its heated and then slowly cooled. At a high temperature, the molecules of a metal move freely with respect to one another. If the liquid is cooled slowly, this mobility is slowly lost. In this way the atoms are often able to line themselves up and form the metal that is completely ordered over a distance up to billions the times the size of an individual atom in all directions. This final form is the site of minimum energy for this system. The essence of the process is the slow cooling, which allows ample time for redistributing the atoms as they lose mobility. This, in turn, allows a low energy state to be achieved.

The simulated annealing algorithm uses two strategies in order to achieve a close to optimal result. The first strategy involves using the Metropolis–Hastings algorithm which is used to obtain a sequence of random samples from a probability distribution for which direct sampling is difficult. In our case, the Metropolis–Hastings algorithm accepts some increases to the tour distance in order to explore other, potentially better, routes. When the temperature setting of the algorithm is high, the algorithm will accept more increases to the tour length and a larger part of the solution space is explored by the algorithm. This allows the algorithm to get out of a local minimum in favor of finding a shorter route for the salesman.

The second strategy employed by the simulated annealing algorithm is its temperature variable that is used by the Metropolis–Hastings algorithm mentioned above. The algorithm starts off with a high temperature variable and after so many exchanges will lower the temperature. This in turns limits the number of increases to that are accepted by the algorithm as the process continues.

The Simulated Annealing algorithm handles the Traveling Salesman Problem using the following steps:

1. Generate an initial solution S for the a given number of cities.
2. Select a value for the initial temperature.
3. Repeat the following n number of times:
    a. Generate a neighborhood S' of S.
    b. Let the difference D = C(S') - C(S).
    c. If D < 0, let S be S'. (Decrease in tour length)
    d. If D >= 0, let S be S' with probability, exp(-D/T). (Increase in tour length)
        i. This is done by consulting the Metropolis-Hastings algorithm.
4. If one of stopping condition is satisfied, stop. Otherwise, decrease temperature and go back to step 3.

The pseudocode for the Simulated Annealing algorithm is shown below. The code was adapted from the Simulated Annealing algorithm from *Numerical Recipes* by Saul Teukolsky and William H. Press. [5]

```
void anneal(float x[], float y[], int iorder[], int ncity){
    // Max number of paths tried at any temperature.
    nover = 180 * ncity;

    // Max number of successful path changes before continuing.
    nlimit = 10 * ncity;

    path = 0.0;

    // Initial temperature.
    t =   30.0;

    // Calculate initial tour length.
    for (i = 1; i < ncity; i++){
        i1 = iorder[i];
        i2 = iorder[i+1];
        path += distance(x[i1], x[i2], y[i1], y[i2]);
    }

    // Close the tour by tying path ends together.
    i1 = iorder[ncity];
    i2 = iorder[1];
    path += distance(x[i1], x[i2], y[i1], y[i2]);
    idum = -1;
    iseed = 111;
```

```
best_distance = path;

// Try up to j temperatures
for (j = 1; j <= 605; j++){
    for (k = 1; k <= nover; k++) {
        do {
            // Choose beginning and end of segment.
            n[1] = 1 + (int)(ncity * ran3(&idum));
            n[2] = 1 + (int)((ncity-1) * ran3(&idum));

            // nn is the number of cities not on the segment.
            nn = 1 + ((n[1] - n[2] + ncity - 1) % ncity);
        } while (nn < 3);
       //Generate a random number to decide whether to do a transport or reversal.
        idec = irbit1(&iseed);

        // Do a transport.
        if (idec == 0) {
           // Randomly pick where the segment will be transported to.
            n[3] = n[2]+(int) (abs(nn-2)*ran3(&idum))+1;
            n[3] = 1 + ((n[3]-1) % ncity);

            // Calculate the difference the move will make to distance.
            de = trncst(x, y, iorder, ncity, n);

           // Consult the Metropolis-Hastings algorithm to confirm transport.
            ans = metrop(de, t);

           // Carry out the transport.
            if(ans) {
                ++nsucc;
                path += de;
                trnspt(iorder, ncity, n);
            }
         // Do a path reversal.
         } else {
           // Calculate the difference the move will make to distance.
            de = revcst(x, y, iorder, ncity, n);
            // Consult the Metropolis-Hastings algorithm to confirm reversal.
            ans = metrop(de, t);
            // Carry out the reversal.
            if (ans){
                ++nsucc;
                path += de;
                reverse(iorder, ncity, n);
            }
        }
        // Finish early if we have enough successful changes.
        if (nsucc >= nlimit) break;
    }
// Annealing Schedule (Reduce the temperature by the temperature factor.
    t *= TFACTOR;
    // After 5 unsuccessful attempts, stop.
    if (nsucc == 0){
       if(nnsucc == 5){
        return;
       }
```

```
            nnsucc++;
        }
    }
}
```

6. Example Tours

The team elected to implement a version of the self annealing algorithm as it was able to reach close to optimal solutions with relative speed. The charts below summarize it's performance on the provided test cases.

Example Cases				
File	Algorithm Solution	Optimal Solution	Ratio	Time
tsp_example_1.txt	108,159	108,159	1.000	120.11s
tsp_example_2.txt	2,584	2,579	1.002	124.02s
tsp_example_3.txt	1,742,691	1,573,084	1.107	503.39s

Competition Cases (3 Minutes)		
File	Algorithm Solution	Time
test-input-1.txt	5,333	120.58s
test-input-2.txt	7,384	120.71s
test-input-3.txt	12,102	123.52s
test-input-4.txt	16,925	121.58s
test-input-5.txt	23,650	123.81s
test-input-6.txt	33,815	133.56s
test-input-7.txt	54,410	175.95s

Competition Cases (Unlimited)	
File	Algorithm Solution
test-input-1.txt	5,333

test-input-2.txt	7,384
test-input-3.txt	12,093
test-input-4.txt	16,852
test-input-5.txt	23,438
test-input-6.txt	33,565
test-input-7.txt	53,286

7. Conclusion

This report outlined four solutions to the travelling salesman problem, the Nearest Neighbor algorithm, the Christofides algorithm, the self annealing algorithm and the Held-Karp algorithm. While the travelling salesman problem is NP-hard, good approximations exist in the nearest neighbor, the self annealing and Christofides algorithms. The Held-Karp algorithm is a precise solution that uses dynamic programming to provide a better performing alternative than a brute-force algorithm. As one of the first solutions to be better than brute-force in finding the optimal solution, it eventually lead to additional approximations and heuristics [4]. These algorithms are just a handful of the many that address this complex problem, but they provide a good overview of the approaches used to solve it.

References

[1] A. Z. G. Gutin A. Yeo, “Traveling salesman should not be greedy: Domination analysis of greedy-type heuristics for the tsp,” Discrete Applied Mathematics, vol. 117, no. 1-3, pp. 81–86, 2002. doi: https://www.sciencedirect.com/science/article/pii/S0166218X01001950.

[2] “Christofides | Optimization | Google Developers.” Google, Google, developers.google.com/optimization/reference/graph/christofides/.

[3] 'A dynamic programming approach to sequencing problems’, Michael Held and Richard M. Karp, *Journal for the Society for Industrial and Applied Mathematics* 1:10. 1962

[4] Williamson, D.: Analysis of the held-karp heuristic for the traveling salesman problem. Master’s thesis, MIT Computer Science (1990)

[5] Press, W. & Teukolsky, S., Cambridge University Press, Numerical Recipes (1986)