Bryce Bristow, Evin Colignon, & Romano Garza
Group 45
CS325 Analysis of Algorithms
December 1, 2017

Group Project Report: The Travelling Salesman Problem (TSP)

Algorithms Researched

The Nearest Neighbour Algorithm:

Description: The nearest neighbour (NN) algorithm is a greedy algorithm which starts the salesman at a random index, marks it as traversed, and chooses the closest unvisited city as his next move. After all cities have been visited, the salesman returns home. On average, the algorithm returns a path of 1.25 times the optimal path. However, there are many arrangements of cities that lead NN to perform poorly. For example:

"Consider a ladder

```
a----b----c
|    |    |
d----e----f
```

Say length of a-b is 2 and length of a-d is 1. The optimal route is a-b-c-f-e-d-a, 10 units long. Starting at a, NN would produce a-d-e-b-c-f-a which is $7+\sqrt{17}>11$ units long."

Nearest Neighbor Pseudocode:

```
NearestNeighbor():
    READ coordinate data from input file into tsp_vector  // vector of coordinate structs

    INIT solution structure containing an integer of the full distance and a vector,
        path, containing the full path through the cities. Initialize full distance
        attribute to zero.

    DETERMINE random index of tsp_vector  from which to begin graph traversal

    Add the starting node to the solution path
```

```
SET starting node to visited by removing it from tsp_vector

WHILE (tsp_vector is not empty)
        smallest_distance ← infinity        // shortest edge to another city
        sd_index ← 0                            // index of closest city

        FOR each coordinate in tsp_vector
                IF distance to coordinate > smallest_distance
                        Update smallest_distance and sd_index

        Add coordinate with the smallest distance to the solution path
        Add smallest distance result to the solutions full distance
        SET the coordinate with the smallest distance to the current starting node
        Remove that coordinate from tsp_vector to mark it as visited

PRINT results to output file
```

Nearest Neighbor Sources:

https://cs.stackexchange.com/questions/13742/when-does-the-nearest-neighbor-heuristic-fail-for-the-traveling-salesperson

https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

Ant Colony Optimization:

Description: Ant colony optimization (ACO) algorithms use the swarm behavior of ants as a metaphor to search for optimal paths through graphs, like in the Travelling Salesman Problem. In nature, ants wander randomly while leaving pheromone trails. When other ants encounter a pheromone trail, they are more likely to follow it than to continue randomly wandering, and while following the pheromone trails they reinforce the trail by adding their own pheromones. Meanwhile, pheromones evaporate over time and lose their strength. Pheromones accumulates more slowly on longer trails since it takes an ant longer to travel and return (reinforce) along its path, and more pheromone evaporates. Shorter paths get marched over more frequently so pheromone accumulates more rapidly. Ultimately, when a short path is found, pheromone accumulates quickly along the shortest paths, leading other ants to prefer those paths and to reinforce them, eventually resulting in all ants following the shortest paths.

To apply an ACO algorithm to the Travelling Salesman Problem, one creates a set of “ants” and sends each ant on a round trip along the given cities, visiting each city once and depositing “pheromone” along its path. At each city, the ant chooses its next city based on a combination of its distance (shorter distances are preferable) and its pheromone concentration (higher pheromones are preferable). At the end of each iteration, the ant with the shortest traversal reinforces its path with more pheromone. Finally, some fraction of pheromone “evaporates” from each edge and then the process is repeated. After a small number of iterations, pheromone should accumulate distinctly on a good approximation for the optimal path.

Ant Colony Optimization Pseudocode:

```
ACO():
        READ coordinate data from input file into tsp_vector  // vector of coordinate structs

        INIT adjacency matrix for the pheromone concentration of each edge, starting at 0
        INIT adjacency matrix for the weight of each edge, compute using pythagorean theorem

        // ant constructor takes node as argument in constructor, sets the ants current_node
        // attribute to that node
        INIT vector ants to hold one ant object for each node of the graph
        FOR each index in vector
                INIT one ant object

        Iterations ← 0
        WHILE iterations < 5:
                FOR each ant in ants
                        // Amongst unvisited neighbor cities
```

Compute next edge to travel using weight pheremone combination

// Travel to the next city
set current city to the neighbor from the chosen edge

// Apply pheromone as inverse function of weight
Apply pheromone to chosen edge on pheremone adjacency matrix

Add the weight of the edge to the ants tour_length attribute

// tour vector attribute stores order of cities visited
Add the edge to the ants tour attribute vector

INIT results vector with enough indexes to hold each city

Traverse pheremone adjacency matrix

WHILE result vector not full
// should be left with one distinct pheremone route through graph
Push next city onto results vector if significant pheremone on edge

Return results vector

Ant Colony Optimization Sources:

http://www.ef.uns.ac.rs/mis/archive-pdf/2011%20-%20No4/MIS2011_4_2.pdf

https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms

staff.washington.edu/paymana/swarm/stutzle99-eaecs.pdf

Christofides Algorithm:

Description: The Christofides algorithm approximates the Travelling Salesman Problem solution by first finding a minimum spanning tree T of the graph G. We then find a minimum spanning tree M of the subgraph formed with all odd-degree vertices of T. T and M are combined to make a graph E with all edges of even degree. Then we find an Euler tour of E and remove the repeated visits to vertices in the Euler tour to form a Hamiltonian circuit. The resulting Hamiltonian circuit should approximate the optimal solution for the TSP within 1.5OPT.

Christofides Algorithm Pseudocode:

```
READ coordinate data from input file into tsp_vector  // vector of coordinate structs

// Create graph with input vertices, drawing edges between every vertex
INIT graph G                                          // adjacency matrix

// Create minimum spanning tree T of G
INIT graph T ← Kruskal's Algorithm(G)

// Create subgraph O of T containing only T's odd degree vertices
INIT graph O ← G
FOR each vertex in O
        IF vertex's degree is even
                DELETE vertex from O

// Create minimum-weight perfect matching subgraph M of O
INIT graph M ← Edmond's matching algorithm (O)

// Combine M with T to form graph E, which has only vertices of even degree
INIT graph E ← graph T ∪ graph M

// Compute Euler tour of graph E
INIT vector C ← Fleury's algorithm (E)
INIT vector RESULT

// Remove repeated visits to vertices in Euler tour to form HAM CIRCUIT
FOR each vertex in C
        IF current vertex not in RESULT
        PUSH vertex to RESULT
```

Return vector RESULT

Christofides Algorithm Sources:

http://ranger.uta.edu/~gdas/Courses/Fall2004/advAlgos/finalPresentations/Jae%20Sung%20Choi_Christofides%20Algorithm%20Implementation.ppt

https://en.wikipedia.org/wiki/Christofides_algorithm

Verbal Description of Implemented Algorithm: Repetitive Nearest Neighbor Algorithm

The algorithm takes a vector of tsp-coordinate structures as input. Each tsp-coordinate has an x and y coordinate as well as an identifier associated with it. Because the time to run increases significantly as n increases, we chose to cap larger values at a certain number of repetitions: Example3 was capped at 10 repetitions and Test-7 was capped at 500 repetitions. To start, the tsp-coordinate of the last index is chosen as the current tsp-coordinate and the index we will try to find an optimal nearest neighbor path with first. It is then popped from a copy of the passed in vector so that it will not be tried again. Also, it is added to a separate vector of tsp-coordinates that indicates a potential optimum path. Next, while a copy of the passed in vector is not empty, the distance from each tsp-coordinate to the current is checked. The distances are checked by a method that involves finding the hypotenuse between the two coordinates. This distance is compared to a minimum value. If it is less than the minimum, it is set as the closest tsp-coordinate to the current one in the path. When all have been checked, the closest tsp-coordinate is added to the path vector and it's distance from the past tsp-coordinate is added to the running total for path distance. Next, it is marked as traversed by it's removal from the copy of the passed in vector. Lastly, the loop starts over with the last added tsp-coordinate as the current coordinate. As previously stated, the loop will continue adding coordinates to the path till all tsp-coordinates have been removed from the original vector. After all elements have been added, the total distance of this nearest neighbour path is compared to the smallest solution distance found so far (initially infinity). If the distance is lower, the solution is saved as the new solution with the smallest distance. The algorithm starts over, this time trying the next lower index values as the starting place.

Discussion: Why we selected the Repetitive Nearest Neighbor Algorithm

We selected the Repetitive Nearest Neighbor (NN) algorithm because of our familiarity with greedy approaches to solving problems, because the algorithm on average returns results within the bounds required in the assignment, and because the other algorithms we researched had a high potential for lengthy debugging problems which could cause us to miss the deadline.

We've spent a great deal of time in this course working with greedy algorithms and seeing the benefit to applying them to problems with the greedy property, specifically the reduced time complexity of solving the problem. While the Traveling Salesman Problem does not have the greedy subproperty, our research informed us that on average, outside of special arrangements, the Nearest Neighbor algorithm returns a 1.25OPT approximation for the TSP. Therefore the algorithm was thought to likely pass the specified approximation requirements (**see note). Finally, both the Ant Colony Optimization method and Christofides's algorithm are based on concepts that are more abstract and less familiar to us. This meant there was a greater likelihood we would run into problems during the implementation of the algorithm, potentially causing us to miss the deadline while debugging algorithms which we had only just begun to grasp.

As we tested our results using just one iteration of the Nearest Neighbor algorithm, we found that it didn't consistently return results within the approximation we wanted. So, we transformed our Nearest Neighbor algorithm into the Repetitive Nearest Neighbor algorithm. The Repetitive Nearest Neighbor algorithm executes a Nearest Neighbor tour of the graph starting from each vertex in the set, then returns the shortest tour from the set. This implementation update returned better approximations for the Traveling Salesman Problem.

Results

- [Your "best" tours for the three example instances and the time it took to obtain these tours. No time limit.]

Three example instances

Tour Name	Distance	Ratio	Time (sec)
Example 1	130921	1.21045	0.014
Example 2	2975	1.15355	0.401
Example 3 (only 10 repetitions)	1930785	1.22739	33.384

- [Your best solutions for the competition test instances. Time limit 3 minutes and unlimited time.]

Competition test instances

Tour Name	Distance	Time 1 (sec)
Input 1	5911	0.004
Input 2	8011	0.024
Input 3	14826	0.237
Input 4	19711	1.812
Input 5	27128	14.207
Input 6	39469	110.252
Input 7 (only 500 repetitions)	61656	176.413