CS 325 - 400
Project Group 8
Anne Harris, Lowell Tyner, Kirsten Wollam

Traveling Salesman Problem

Algorithm Research

Our group did some general research on the Traveling Salesman Problem and first came up with a list of algorithms of interest. This list included: brute force, dynamic programming (Held Karp), greedy choice (nearest neighbor or nearest fragment), Match Twice and Stitch (MTS), branch and bound, Christofides Algorithm, and k-opt or 2-opt heuristics. We narrowed it down to a dynamic programming option and a few approximation algorithm options, Lowell researched Held Karp, Kristen research MST and Christofides Algorithm, and Anne researched Nearest Neighbor and its related algorithms. Our individual summaries are below.

Held-Karp Summary

Overview

Held-Karp (HK) is a bottom up, dynamic programming algorithm that runs slowly, but finds an optimal solution rather than an approximation. It runs in $O(2^n n^2)$ which is considered to be very slow, but that is better than a brute force which runs in O(n!). While brute force compares every possible path permutation to find a minimum, HK stores solutions to sub problems. In this case, a sub problem is the subset of a complete path. A subset of the minimum path is itself a minimum. And because it is bottom up, the algorithm starts with the smallest subsets of solutions, so that larger subsets can use the previously calculated value.

Algorithm

Given N nodes (cities) and an adjacency matrix giving costs (distances) the idea is to pick a starting node, and for increasing sizes of subsets, determine the minimum cost for that size of subset to get back to that node. Starting at size of subset 0, then calculates all costs to travel to any neighboring city (direct from adjacency matrix). For subset of size 1, calculate the costs to get to every neighboring node plus the cost to get to the start node from that neighboring node. For subset of size S greater than 1 and less than N, for every combination of nodes reachable from the start node of size S, calculate the minimum cost path, adding an additional node to the already calculated paths.

Pseudocode

```
G = adjacency matrix representation of n cities(nodes)
HELD-KARP(G)
Distance[S{}][totalDistance]//keep track of distance of each subset
Node = start node
Initialize all values of Distance to 0 or infinity -for sets that only
contain Node, set to 0, otherwise set to infinity
Distance[S{}][Node] = 0 or infinity
```

```
for m = 2 through n
      for every subset S{} of size m that contains Node
            for each destination node j in S{} not equal to Node
                  Distance[S{}][j] = min of k nodes not equal to
            j{Cost[S{} -{j}][k]+G_kj}
      Return min for j not equal 1 {A[S{1...n}, j], + G_Node j}
```

Conclusion

Given that it returns an optimal solution, it may be worth considering for smaller data sets. But the running time is so poor that it may not be realistic for the 3 minute running time of the competition.

MST Heuristic Approach

Overview

This is an approximation algorithm that has an upper bound of 2 times the optimum, but often does much better. It was the first algorithm identified that guaranteed a result within a constant bound of the optimum. This algorithm runs in O(mlogn), as it is dominated by the time it takes to find the MST, and is actually quite simple.

Algorithm

1. Find the minimum spanning tree (using any MST algorithm).
2. Pick any vertex to be the root of the tree.
3. Traverse the tree in *pre-order*.
4. Return the order of vertices visited in pre-order.

Pseudocode

```
Int V                          // number of vertices in graph
int G[V][V]                    //adjacency matrix to represent graph
int T[V][V]                    //adjacency matrix to represent MST, all values
set to 0
int edgeCount = 0;      // number of edges added, starts at 0

// create an array to track selected vertex
int selected[V]; set selected to false;

// choose 0th vertex (or any vertex to start at) and make it true
selected[0] = true;

int x, y;          //  row number, col number

//run till edgeCount < V -1 to make complete MST via Prim's
while (edgeCount < V - 1) {
  //For every vertex in selected find the minimum distance to any vertex not
in selected
 int min = INF;
 x, y = 0
 for (int i = 0; i < V; i++) {
```

```
            if (selected[i]) {
          for (int j = 0; j < V; j++) {
                        if (!selected[j]) { // not in selected
                                    if (min > G[i][j]) {
                                                min = G[i][j];
                                                x = i;
                                                y = j;
T[x][y] , T[y][x] = G[x][y]
selected[y] = true;
edgeCount++;
// run depth first search to create pre-order walk
W                 // array to track walk order
Int length        // to hold cumulative walk length
Stack S{ }        // to hold vertices in work
Int u, prev       //
Set selected back to false for all values
Push 0 to S   //or any starting vertex that was used
//cycle through all vertices pushing unvisited ones to the stack
prev = u
 While(S)
            u = pop S // get top vertex off stack
            append u to W
            selected[u] = true
            for( int k=0, k < V, k++)
                        if(selected k == false && T[u][k])
                                    push k to S
            length += T[u][prev]
            prev = u
```

Conclusion

This is a simple and effective option that has a good run time. Could be combined with other options or as a starting point for options that improve on a given tree.

Christofides Algorithm

Overview

This algorithm is an improvement on the basic MST heuristic approach described above. It provides the best known accuracy of any of the known approximation algorithms and an upper bound of 3/2 of optimum but at a steep cost in terms of run time at O(n^3). This is because the matching step dominates and takes O(n^3) to run.

Algorithm

1. Find the minimum spanning tree (using any MST algorithm)
2. identify vertices with odd degree
3. find the minimum perfect matching of these vertices and add these edges to the MST
4. find a Euler tour
5. walk the tour skipping any nodes already visited

Conclusion

While this provides the best accuracy it does it at a computational cost, and may not work quickly enough on large sample sets to satisfy the 3 minute requirement. Could set it up to run this version instead of the MST heuristic for "small" data sets only. They share significant chunks of code.

Nearest Neighbor Summary

Overview

Nearest Neighbor is a greedy, approximation algorithm and was one of the first algorithms used to find a solution for the Traveling Salesman Problem. The salesman starts at a random city and continues to visit the next closest city until they all have been visited. Usually a sub-optimal solution but runs in very efficient time. On average, the Nearest Neighbor algorithm produces a path that is 25% longer than the shortest possible path.
The run time is $O(n^2)$ since it looks at each vertex and the distance to all other vertices from that vertex.

There exist many city distributions that make Nearest Neighbor worst case, this may be something to consider with our datasets as there are time limits and likely very big datasets to test.

Algorithm Steps

1. Start at random vertex and set as current vertex
2. Find the nearest unvisited edge, V
3. Set current vertex to V
4. Mark V as visited and add to path and total distance
5. Check to see if all vertices have been visited
    a. If so, terminate program
    b. Else, return to step two

Pseudocode

```
//pass a vector of all cities C
//pass in an int for the a starting point
nearestNeighbor(C, startCity)
      //create a vector of all visited cities, V
      V = {}
      //set the current city to the start
      currentCity = startCity
      V.push(startCity)
      //initialize the total distance traveled to 0
      totalDistance = 0
      //initialize closest city distance to infinity
      closestCityDist = INF
      //initialize count to 0
      Count = 0
      //declare closest city
      closestCity

      //while the city vector is not empty
      while count < C.size()-1
            // set the closest city distance to infinity
```

```
closestCityDist = INF
//loop through the remaining non visited cities
for i in C
      //if i is not visited and i is not the current city
      if i != V and i != currentCity
            //calculate the distance between the locations
            distance = i.location – currentCity.location
                  //if the calculated distances is less than the
                  //closest cities distance, set that as the
                  //closest city
                  if distance < closestCityDist
                        closestCity = i
                        closestCityDist = distance
//add the closest city to the visited vector
V.push(closestCity)
//change the current city to the closest one
currentCity = closestCity
// add the distance to the running total
totalDistance += closestCityDist
//remove the city from the cities list
C.pop(closestCity)
```

Conclusion

Nearest Neighbor executes fairly quickly and would likely be a great option to stay under the 3 minute restriction. However, it usually doesn't yield the optimal solution. I think this will be a viable option for large sets of data. I also saw a few solutions mention combining this with the 2-opt technique, this could be another option if we are not meeting the requirements when testing.

Repetitive Nearest-Neighbor Algorithm

Overview

This will run the nearest neighbor algorithm for each vertex as a starting point, and choose the shortest route. I think this is $O(n^3)$. Since NNA runs in $O(n^2)$ and it must be run for each vertex, O(n), thus $n * n^2 = n^3$.

Algorithm Steps

1. Find the nearest neighbor tour with x as starting point and calculate route
2. Store result of tour length as a variable L if it is less than current L
3. Check if every city has been used to start
    a. If so, go to step 4
    b. Else, go to step 1 with x+1
4. Choose starting city that corresponds with L as the start city to yield the shortest route

Pseudocode

```
C     //vector of cities
Graph G    //adjacency matrix of cities
Int currentDistance
Int bestDistance
```

```
Int bestCity
For i in Cities:
      currentDistance = NearestNeighbor(G,i)
      If currentDistance < bestDistance
            bestCity = i
            bestDistance = current distance

Return bestCity
```

Conclusion

This seems like it will be extremely useful for smaller datasets and is a great compliment to the original Nearest Neighbor algorithm. We will have to determine a size of the dataset to to run for all, a size to run half of the cities, and a size to not run at all.

Nearest Fragment/Multi-Fragment Heuristic Algorithm

Overview

The Nearest Fragment Heuristic Algorithm connects groups of nearest uninvited cities using a distance matrix. From my research, it continually links two sets of nearest cities to each other and then starts to connect the groups of pairs. Does not guarantee optimal solution but it will run quickly with a “reasonable good” solution.

Steps

1. Set a matrix up that contains the distance between all cities in the tour
2. Find the shortest distance and group the two cities together
3. Find second shortest distance and group those two cities together
4. Find smallest edge whose cities follow this criteria:
    a. Neither city can already be linked to one or more city
    b. Adding an edge between the cities must not result in a closed tour
5. Repeat until all cities end up being connected to two other cities

Conclusion

I haven’t been able to find a lot of information about this algorithm online, for the purposes of this assignment, I think we will be within the requirements if we use Nearest Neighbor or Repetitive Nearest Neighbor.

Algorithms Implemented

Our group chose to implement MST Heuristic Approach and Nearest Neighbor Algorithm. We chose these because they are approximation algorithms. Since we didn’t know what the input data would look like, but we could assume they would include large sets of cities, we wanted to go with something that would get us a near optimal solution in efficient time. We used an adjacency matrix as our data structure. This allowed us to calculate the distance before running any of the algorithms so that when the algorithm was executing it only had to look up a value, not calculate every time.

The implementation for MST heuristic was very close to that in the pseudocode. The function took an adjacency matrix of distances, the number of cities, a starting city, and a vector to store the path in. It returned the total distance for the path. The function then created a second adjacency matrix to store the Minimum Spanning Tree that was found using Prim's algorithm along with an array to indicate which vertices were already in the tree. Prim's algorithm over the matrix, each time adding the vertex that is closest to any of the already added vertices, until all of the vertices have been added. Once the MST was found, a depth first search was used to create a preorder list of the visited vertices. Any vertex that was already on the list was simply skipped over. As each vertex is added to the path, the length between it and the previous vertex was also added to the total length. At the end the distance from the last city to the start city was also added.

The implementation for Nearest Neighbor follows the pseudocode closely but varies slightly as we decided to use an adjacency matrix as our data structure. The implemented algorithm parameters include a dynamic adjacency matrix, an integer representing the number of cities, an integer representing the starting cities, and a vector passed by reference representing the path taken. It then creates an array for the visited cities using the total number of cities passed in and sets these all to have the value false. Then a variable to represent the current city is set to the start city passed in, its corresponding value in the visited array is set to true, and it's added to the path vector. Then variables for the closest city, total distance and count are created, as well as a variable to represent the minimum distance is initialized to infinity (a constant).

From here the Nearest Neighbor algorithm starts a while loop, executing while the count is less than the total number of cities minus one. The minimum distance is set to infinity again and then it enters a for loop to loop through the current cities adjacencies. Within this loop it tests to see if the city it's checking has already been visited, if not, it compares the city being checked distance to the current minimum distance. If it's less than the minimum distance it will set the minimum distance to equal that distance and set the closest city to equal the city being checked. Once this loop has completed it adds the closest city to the path vector, the minimum distance found to the total distance, sets the city as true in the visited array, updates the current city variable to be this closest city, and finally increases the count. This will go through each city until the while loop is done.

Once the while loop has finished, it adds the final path back to the starting city to the path vector and to the total distance. It then deletes temporary visited array and returns the total distance as an integer.

We decided against implementing the Held-Karp algorithm because we had were getting strong results from the approximation algorithms. After the test driver program was written that read in files from disk and created the adjacency matrix to pass into the TSP algorithms, we were getting some unexpected results for the MST algorithm, so we were considering relying only on the NN algorithm for the competition, however the MST algorithm issue got sorted out and we were able to take advantage of it for the competition. The adjacency matrix created by the test driver file was manually tested against a small data set to ensure it was feeding the algorithms accurate data.

For both the MST and Nearest Neighbor algorithms, we wrote a function to test some or all of the cities as the start city to see which yielded the shortest tour. In doing this we hoped to find a solution closer to the optimal one. It was unrealistic to test every city for the large data sets, so we implemented this process depending on how many total cities were in the data set. When the total number of cities was less than or equal to 250 which tested each city as a starting city using both MST and Nearest Neighbor, less than or equal to 4,000 which tested all cities using the Nearest Neighbor algorithm but not MST. For less than or equal to 5,000, which was specific to test data 7, we only tested one city using Nearest Neighbor that we had found to be optimal. This was because the data set was too large to complete a complete repetitive

Nearest Neighbor in under 3 minutes. Anything greater than 5,000 ran the nearest neighbor algorithm n/100 times. We found this helped find an even shorter tour than simply picking a random city or starting at the first city, and since we ran this differently depending on size it didn't impact our overall efficiency.

Using both our MST and Nearest Neighbor approaches, we were able to find solutions for all of the example files that were within the 1.25 bounds as well as solutions for all of the competition case files that were under 3 minutes.

Best Tours for Example Instances

Test Case	Tour Distance	Total Time (s)	Optimal	%	Number of cities
Example 1	123348	0.07	108159	114.04%	75
Example 2	2975	0.1	2579	115.35%	279
Example 3	1918511	229.6	1573084	121.96%	15111

Best Solutions for Competition Test Instances

Test Case	Tour Distance	Total Time (s)	Number of cities
Test 1	5911	0.01	49
Test 2	8011	0.28	99
Test 3	14826	11.31	249
Test 4	19711	0.81	499
Test 5	27128	6.36	999
Test 6	39469	51.43	1999
Test 7	61466	1.58	4999

Sources:

https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
https://en.wikipedia.org/wiki/Travelling_salesman_problem
https://cs.stackexchange.com/questions/65166/implement-multi-fragment-heuristics-for-the-traveling-salesman-problem
https://www.coursera.org/learn/algorithms-npcomplete/lecture/uVABz/a-dynamic-programming-algorithm-for-tsp
https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm