CS325

Project Report Group 30

Amy Weller, Matthew Toro, Joseph Valencia

**A description of at least three different methods/algorithms for solving the Traveling Salesman**

Our team researched three different algorithms for tackling the traveling salesman problem. The Traveling Salesman Problem page on Wikipedia was used as the source. There are algorithms that actually arrive at the optimal solution and there are algorithms that arrive at approximate solutions that are proven to be within a certain range of the optimal solution. The three we researched were: a brute force algorithm; a dynamic programming algorithm called the Held-Karp algorithm; and a greedy algorithm known as the Nearest Neighbor algorithm.

Description of Brute Force: The brute force implementation arrives at an optimal solution by trying every possible permutation of the city list. Since every permutation is calculated, the run time of such a brute force algorithm is O(n!). The implementation uses Heap's algorithm to calculate the tour for each permutation of the city list. We used the Wikipedia page for Heap's algorithm as a source for its description and implementation. Heap's algorithm is an algorithm that swaps indices in an array to create every possible permutation of that array. Normally, Heap's algorithm will output the array permutation once it arrives at a new one, but we modified it to calculate the tour for that city list. We keep a tally of the minimum tour and once the algorithm finishes, return that minimum tour value.

Pseudo-Code:

tsp (length, cities)

int c [length]

set every value in c to 0

int minTour = calculateTour(length, cities)

int j = 0

while (j < length)

   if c [j] < j

      if j is even

         swap(cities[0], cities[j])

      else

         swap(cities [c [j] ], cities[j] )

      int temp = calculateTour(length, cities)

      if temp < min

         minTour = temp

      c [ j ] += 1

      j = 0

   else

      c [j] = 0

      j++

return minTour

calculateTour(length, cities)

tour = 0

for (k = 0; k < length - 1; k++)

   tour+= distance(cities[k], cities[k+1])

tour+=distance(cities[k], cities[0])

// distance is a sub-function to calculate the distance formula between two cities

return tour

Description of Held-Karp Algorithm: The dynamic programming approach for solving the Traveling Salesman problem starts with identifying the sub-problem inherent in the problem. That is: "Every subpath of a path of minimum distance is itself of minimum distance". Based on this, Held-Karp breaks the graph itself up into smaller sub-graphs/sub-problems until it is able to minimize the smallest possible problem. These answers are then saved as solutions to minimized graphs that are used to solve larger sub-problems until the entire graph is solved. It is impossible to know which sub-problems will produce the minimized graph, so the algorithm provides a solution for each of them.

To summarize, basically we need to create sets of 1 elements to ... n-2 elements. From there, we solve for the minimized paths of the smallest elements, and we use those solutions to solve for the minimized paths of the larger elements. After we obtain the minimized path for the n-2 element, we continue down the chain in order to select the minimized paths for each of the n-3...1 elements until we have the final solution.

The worst case scenario for this algorithm is O(2^n \* n^2), but a problem is that it uses a lot of memory, to the point of O(2^n \* n) in order to store all of the previous solutions. Thus, we can only use this for problems of a certain size, and for larger problems we must use different algorithms that may not arrive at an optimal solution.

Pseudo-Code:

TSP (graph, size)

initialize an adjacency matrix C

for k = 2 to size, k++

C({k}, k) = distance at 1, k

for s = 2 to size-1

for all paths, S, sized 2 to size

for all elements in S

C(S, k) = min of current path versus the prior min

tour = min from C({2, 3, . . . , n}, k) + dk,1

return (tour)

Description of Nearest Neighbor Greedy Algorithm: This algorithm takes the greedy strategy of starting at an arbitrary vertex and always choosing the next vertex in the tour by whichever has the minimum distance from the current vertex. The advantage of this algorithm is that it can run quickly compared to other algorithms. However, it is not guaranteed to arrive at optimal solutions.

Pseudo-Code:

tsp (graph, size)

set first vertex in graph as the starting vertex

then for each vertex in the graph

then for each edge from the vertex

calculate distance for every other path

save current distance in a temporary variable

if smaller than current minimum, set to minimum

mark current vertex as visited, it will not be repeated

return minimum

Other Research: Our group also researched a data structure known as the k-d Tree, and a couple of different approximation algorithms that we could potentially use. We researched k-d trees because our greedy algorithm was not meeting the margins required for full points in the assignment. We used the Wikipedia page for k-d trees and http://www.geeksforgeeks.org/k-dimensional-tree/ for research on this data structure. The k-d tree is a variation on the binary search tree that stores points in dimensional space. When a new point is inserted into the tree, different comparisons can be made to determine whether the point should go to the left or right of its parent. We found the k-d tree helpful for sorting the cities in the route before using the nearest neighbor algorithm on them.

In addition, the textbook's example of using a minimum spanning tree as an approximation algorithm was used as research to try and implement that algorithm. Unfortunately, this algorithm performed worse than our current algorithm so we chose not to use it. However, upon further research, it was discovered that the minimum spanning tree algorithm can be optimized using the Christofide's algorithm. The Wikipedia page for the Christofide's algorithm was used as the source for learning more about it. The Christofide's algorithm, though, requires the use of another complicated algorithm, the Blossom algorithm, in order to find minimum weight perfect matching pairs in a graph. This was deemed beyond the means of this course so we also abandoned the research on this algorithm.

**Verbal Description of our Algorithm: High Level Design for NN Implementation to solve TSP**

Our program first accepts the input file from the command line. From there, it saves each line in the input file as a struct that holds the city id, the city's x-coordinate, and the city's y-coordinate and pushes it into a vector. During this process, we select every tenth city, calculate it's distance from the starting city, and collect a sum of the distances. We collect, at most, five different averages by looping five times and dividing the sum of the distances by different amounts. These threshold amounts are used to create k-d trees to sort the route in different orders based on distance from that threshold. We run the algorithm on the tree routes created and save the best tour for that threshold/tree route. Lastly, we run the nearest neighbor algorithm with the best average threshold (the one that had the minimum tour) and save those results to the output file.

The nearest neighbor algorithm accepts a vector of cities as input. It starts at the very first city in the vector, sets it to visited, and begins the nested for loop. The outer loop will run for every vertex in the graph. Then, the inner loop will run for every edge from the current vertex. The inner loop will calculate the distance from the current vertex to every other vertex that has not been visisted yet. It will save the minimum distance and update the current vertex's nearest neighbor via pointer and set the next vertex to be calculated as the current vertex's nearest neighbor. Once the outer loop has run through every vertex in the graph, the very last vertex to be ran in the outer loop will have its distance to the start calculated and added to the tour value. The algorithm then runs through the pointers of nearest neighbors to print out the route to the output file.

**A discussion on why you selected the algorithm(s).**

Our team selected the Nearest Neighbor Algorithm. While the brute force and the Held-Karp Algorithm actually arrive at optimal solutions, they also run pretty slow (one is factorial and one is exponential). Without knowing any details about the input we can expect for the project, we were more comfortable with the greedy algorithm giving us an approximate solution than dealing with the slower algorithms. We decided we could always make additions to the algorithm to help improve the solution in case it didn’t meet the requirements of being within 1.25 bound of the optimal solution.

After some testing, we realized we’re not meeting the 1.25 bound of the optimal solution.

|  |  |  |  |
| --- | --- | --- | --- |
| **Inputfile name** | **Best tour distance** | **Optimum provided** | **Ratio (<=1.25** |
| tsp\_example\_1.txt | 150393 | 108159 | 1.39 |
| tsp\_example\_2.txt | 3210 | 2579 | 1.24 |
| tsp\_example\_3.txt | 1964948 | 1573084 | 1.25 |

We then researched options to optimize the algorithm. One such way was implementing a k-d Tree in order to sort the route before plugging it into the algorithm. Initially, we were inserting into the tree based on x and being greater or less than the root. However, this didn’t net us any better results than the previous implementation. So then we added the comparison of distance from the root but our results were still unsatisfactory. Finally, we came up with a solution of a threshold value by which to judge whether an inserted point should go to the left or right of the root. This threshold is calculated by taking averages of the distance from the root in the input file. If a point's distance is less than the threshold, it goes to the left of the parent; otherwise, it goes to the right. This started giving us the results we were looking for and we decided to optimize further by calculating five different threshold values and taking the best tour from those values.

**Pseudo code**

main()

accept command line input of the input file name

loop through the input file

save city id, x coordinate, and y-coordinate on each line

calculate distance from the starting city every tenth city

for k = 0 to 5

calculate a different average from the sum distance so far

build a k-d tree based on that average as the threshold value

create a vector route for the nearest neighbor algorithm by removing from the tree

run nearest neighbor on this route

save tour value if less than previous

build a k-d tree based on best average, I.e, the one that gave the minimum tour

create a vector route for the nearest neighbor algorithm by removing from the tree

run the nearest neighbor on this route

write results to the output file

nearestNeighbor(vector, outputFile)

set current to first city in vector

set current to visited

for j = 0 to size of vector

minimum = LARGE\_VALUE

for k = 0 to size of vector

if vector[k] has not been visited

calculate distance from vector[j] to vector[k]

if distance calculated is less than minimum

minimum = distance calculated

vector[j].nearestNeighbor = vector[k]

vector[j].distanceToNeighbor = minimum

tour+=vector[j].distanceToNeighbor

current city = vector[j].nearestNeighbor

set current city to visited

place tour value in output file

loop through cities, placing nearest neighbors in output file

**Your “best” tours for the three example instances and the time it took to obtain these tours. No time limit**

|  |  |  |  |
| --- | --- | --- | --- |
| **Inputfile name** | **Best tour distance** | **Optimum provided** | **Ratio (<=1.25)** |
| tsp\_example\_1.txt | 135,240 | 108,159 | 1.25 |
| tsp\_example\_2.txt | 3,114 | 2,579 | 1.20 |
| tsp\_example\_3.txt | 1,927,168 | 1,573,084 | 1.22 |

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Inputfile name** | **Best tour distance 3 minutes** | **Duration (seconds)** | **Best tour distance unlimited** |
| test-input-1.txt | 5,926 | 0 | 5,926 |
| test-input-2.txt | 8,625 | 0 | 8,625 |
| test-input-3.txt | 15,149 | 0 | 15,149 |
| test-input-4.txt | 20,215 | 0 | 20,215 |
| test-input-5.txt | 27,948 | 0 | 27,948 |
| test-input-6.txt | 40,792 | 2 | 40,792 |
| test-input-7.txt | 62,451 | 11 | 62,451 |