CS325

Project30 Report

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 thisfor 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 did other research throughout completing this project. Specifically, we 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. Basically, 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**

main()

* **Input**:
  + command line argument = text file
  + Text file will contain
    - City ID, X coordinate, Y coordinate on each line
* Main has a loop to collect an vector of the structs for the city ID, and the x/y coordinates
* Main calls nearestNeighbor()on the vector
* nearestNeighbor calls distance() to calculate the shortest distance to the next city
* nearestNeighbor() calls checkTime() for testing of the 180 seconds time limit
* Creates outfile with the expected **output** below
  + Name the output file as the input file’s name with .tour appended (for example input **tsp\_example\_1.txt** will output **tsp\_example\_1.txt.tour**)
* **Output**:
  + text file w/ n+1 lines
  + where n is the number of cities.
  + The first line is length of the tour our program computes
  + Next n lines should contain the city id’s in the order we visited them
    - Each city listed once
    - This is the certificate for our solution, if not valid we don’t get credit

nearestNeighbor(vector, start)

* **Input**: vector of structs & start time for timekeeping
* **Output**: outputfile which contains the order of the nearest neighbors visited in order of minimum distance, with the first line in the file containing the length of the tour computed

**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 such as implementing a k-d Tree. We are walking the tree based on x and being greater or less than current. The initial is aligned to x but that didn’t net us any better results than the previous implementation. So then we added the comparison to x and y and re-ran the testing.

**Pseudo code**

Input: a set of n points in d  distance

    P={p1, p1, ....pn}

Desired output:

   For each point p that is an element of P the nearest point to p

Runtime should be O(dn2)

NN (P)

for all i

  compute d(i,j) = pi - pj

   for i = 1 to n

     dist[i] = infinity

       for j = 1 to n

         if i != j and d[i j]< dist [i]

         then

             dist[i] <--d[i j]; NN[i]<--j

     return NN, dist

**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 |