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. I used the Traveling Salesman Problem page on Wikipedia as a 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: The brute force algorithm. The dynamic programming algorithm called Held-Karp. Lastly the 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. So, 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, and return that minimum tour value.

Description of Held-Karp: the dynamic programing procedure makes use of the optimization property that "Every subpath of a path of minimum distance is itself of minimum distance". Thus, we break the graph itself up into smaller sub-graphs/sub-problems until we are able to minimize the smallest possible problem, and we make use of that and the saved solutions to other minimized graphs in order to solve the larger sub-problems until we have solved the entire graph. Since it is impossible to know which sub-problems will produce the minimized graph, we need to provide a solution for each of them. To sum this up, 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.

Description of Nearest Neighbor Greedy Algorithm: Researching on Wikipedia, the following is the high level description. These are the steps of the algorithm:

start on an arbitrary vertex as current vertex.

find out the shortest edge connecting current vertex and an unvisited vertex V.

set current vertex to V.

mark V as visited.

if all the vertices in domain are visited, then terminate.

Go to step 2.

The sequence of the visited vertices is the output of the algorithm. The nearest neighbour algorithm is easy to implement and executes quickly, but it can sometimes miss shorter routes which are easily noticed with human insight, due to its "greedy" nature. As a general guide, if the last few stages of the tour are comparable in length to the first stages, then the tour is reasonable; if they are much greater, then it is likely that there are much better tours. Another check is to use an algorithm such as the lower bound algorithm to estimate if this tour is good enough.

In the worst case, the algorithm results in a tour that is much longer than the optimal tour. To be precise, for every constant there is an instance of the traveling salesman problem such that the length of the tour computed by the nearest neighbor algorithm is greater than r times the length of the optimal tour. Moreover, for each number of cities there is an assignment of distances between the cities for which the nearest neighbor heuristic produces the unique worst possible tour.

**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 | 150393 | 108159 | 1.39 |
| tsp\_example\_2.txt | 3210 | 2579 | 1.24 |
| tsp\_example\_3.txt | 1964948 | 1573084 | 1.25 |

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

|  |  |  |
| --- | --- | --- |
| **Inputfile name** | **Best tour distance 3 minutes** | **Best tour distance unlimited** |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |