Mary May, Quinn Campfield, Daniel Jones
Winter 2020, CS 325

The Traveling Salesman Problem

Traveling Salesman Problem: Dynamic Programming

Dynamic programming is an optimization technique used to store the solutions of subproblems in order to use later instead of having to recursively call a function. This helps by significantly cutting down on run time. Programs can go from being exponential to polynomial. Dynamic Programming is made up of an optimal substructure and overlapping subproblems. The optimal substructure must consist of an optimal solution to a problem containing within it an optimal solution to subproblems and optimal solution to the entire problem is built in a bottom up manner from the optimal solutions to subproblems stored in a table.

Using dynamic programming to solve the traveling salesman problem. There are at most ($2^n * n$) subproblems, and each one takes linear time to solve. Therefore, time complexity for this algorithm would be $O(n^2*2^n)$. This runtime is exponential however still better than O(n!). With a large number of vertices though this could still take a very long time.

Subproblem:

If size of S is 2, then S must be {1, i},

C(S, i) = dist(1, i)

Else if size of S is greater than 2.

C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

One example of an algorithm that uses dynamic programming to solve the traveling salesman problem is the Held-Karp algorithm. This algorithm was proposed by Richard E. Bellman, Michael Held, and Richard M. Karp in 1962. Their algorithm gained popularity since its running time for solving an n-city instance is the best asymptotic bound that has been achieved to date. The runtime being O(n2*2n). However, this algorithm is still exponential time. The pseudocode for this algorithm is as follows:

```
HK-TSP( ){
    C({1}, 1) = 0
        If size of S is 2, then S must be {1, i},
            C(S, i) = dist(1, i)
        Else if size of S is greater than 2.
            for s = 2 to n:
                for all subsets S ⊆ {1, 2, . . . , n} of size s and containing 1:
                    C(S, 1) = ∞
                for all j ∈ S, j 6= 1:
                    C(S, j) = min{C(S − {j},i) + dis(j,i)} where j belongs to S, j != i
                    and j != 1.
    return min_j C({1, . . . , n}, j) + d_j1
```

```
}
```

Sources:
https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/
https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf
https://www.geeksforgeeks.org/dynamic-programming/
https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm#Pseudocode[5]
https://www.math.uwaterloo.ca/~bico/papers/comp_chapterDP.pdf
CS 325 Lecture notes

Traveling Salesman Problem: Nearest Neighbor

Nearest Neighbor is an algorithm that can be used to solve the Traveling Salesman Problem (TSP). The basis of this algorithm is that you will start at a vertex v, in any graph of plotted points g. From there, the path that will be chosen will be the path that is the shortest, or has the lowest cost. This step will be executed again once we are at the next vertex in the graph. Once we traverse from the first vertex to the second vertex, the first vertex will become marked as "visited", and once we make a decision from the second vertex, it will also be added to the visited group. This will run until all vertices have been visited a single time. However choosing a random vertex v and starting the algorithm from there may not yield the optimal solution the first time, as there could be other paths that actually yield a more optimal solution due to the costs of the paths connected to each vertex.

After analyzing the Nearest Neighbor algorithm to solve the TSP, we have come to find out the runtime for it is $O(n^2 \lg_2(n))$. As always when there are more inputs it will take longer to run. There are pros and cons to using Nearest Neighbor over other algorithms used to solve the TSP. In a research PDF regarding this problem, when looking at Nearest Neighbor, Genetic, and Greedy Heuristic at n=100, NN resulted in cost of 26,664 and time of 2.5 seconds, Genetic had a cost of 25,479 and time of 25 seconds, and Greedy Heuristic had a cost of 23,311 and time of .07 seconds. Once we increase n to n=1000, NN results in a length of 83,938 with a time of 95.5 seconds, Genetic had a length of 282,866 and a time of 468 seconds, and Greedy Heuristic had a length of 72,801 and time of 127 seconds. Based off of this data it shows that while NN does not ALWAYS produce the optimal solution (oftentimes it still can), it does keep the time elapsed low when the size of n is larger.

Pseudocode:

```
nnTSP() {
        Int cost = 0;
        //J is starting point for the ham cycle
        Select random node j;
        Add vertex j to the visited array
        Visited[0] = j;
        //Get the minimum path from any vertex connected to the vertex j
```

J = the vertex chosen
Return Cost += min(path to v1, v2,... vn);
//Decrement the size of the unvisited array
sizeUnvisited--;
//Select the just selected vertex and add that to the visited array
Visited[1] = j;
Once all the cities are visited we will have the path of the visited cities along with the added up total cost
}

Sources:
http://digitalfirst.bfwpub.com/math_applet/asset/3/TSP_NN.html
https://blog.usejournal.com/a-quick-introduction-to-k-nearest-neighbors-algorithm-62214cea29c7
https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761
https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
https://www.ijeat.org/wp-content/uploads/papers/v4i6/F4173084615.pdf
CS325 Lecture Notes

Traveling Salesman Problem: Christofides' Algorithm

Christofides' Algorithm tuses minimum spanning trees and Euler cycle to find a solution to the TSP problem. It is a heuristic algorithm so it usually works for all inputs and gives nearly the right answer every time. For an input of vertices V you have to then create a minimum spanning tree T. A minimum spanning tree is a subset of edges between vertices that connects all vertices together by using the smallest distance. After creating the mst T there will be some points that are of odd degree where possibly only one other vertex is connected to it. We can use these points to form a perfect matching graph G. We can then combine the graph, G, with the mst, T, to create a connected graph. Then form a Euler's cycle from this connected graph and then go back and skip repeated vertices so you don't waste distance.

When this algorithm is used it is found that it has a theoretical running time of $O(n^{1.5})$ where n is the number of cities in the TSP problem. The solution it generates is a maximum of 1.35xoptimal. Given the approximate time for the algorithm it runs relatively quickly. Also the solution is a maximum of 135% of the optimal solution which can be a lot but that is the maximum so it likely won't ever be that high. We need to make every vertex have an even degree because if they all are even a Euler tour can be generated from all of the vertices and then shortcut after making to become more optimal.

Pseudocode:
ChristofidesTSP(array of Cities) {
//calculate MST for Cities
Tree MST = Prim(Cities)
cities*Odds

```
        For cities in MST
                Find if they are odd degrees of connections
                Odds+= all odd degree vertices
        Turn into perfect matching graph with these points
        cities*Both
        For Odds length
                both+=Odds graph
        For MST length
                both+=MST graph
        Euler= Create a euler circuit

        For Euler circuit
                If vertex used multiple times
                        shorten by skipping one instance
}
Prim(array of Cities){
        For cities:
                Find closest unused Vertex
                Include into graph update to used
        Construct tree
}
```

Sources:
https://personal.vu.nl/r.a.sitters/AdvancedAlgorithms/2016/SlidesChapter2-2016.pdf
https://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/
https://en.wikipedia.org/wiki/Minimum_spanning_tree
https://xlinux.nist.gov/dads/HTML/christofides.html
https://arxiv.org/pdf/1802.01242.pdf
CS325 Lecture Notes

Algorithm Implemented: Nearest Neighbor

Our group decided to implement the Nearest Neighbor algorithm because we felt most confident on the pseudocode and according to research, nearest neighbor showed to be the most promising in terms of run time. As stated earlier, nearest neighbor doesn't always produce the optimal solution, but it does keep the time elapsed low when the size of n is larger. We felt that this was important in getting our runtimes to be under five minutes. We implemented our algorithm in C++. The pseudocode we used is the same as described above in the "Traveling Salesman Problem: Nearest Neighbor" section. In order to keep track of the points, we created a city class. We stored "cities" in a deque in order to keep track of the tour. Our group collaborated via github and communicated via text messaging as well as meeting up once to discuss implementation.

Mary May, Quinn Campfield, Daniel Jones
Winter 2020, CS 325

To find our best tours and to find the time it took to obtain these tours we used the clock() function. We selecting our “best” tours for each example instances (tsp_example_1.txt, tsp_example_2.txt, tsp_example_3.txt) and test inputs (test-input-1, test-input-2, test-input-3, test-input-4, test-input-5, test-input-6, test-input-7). The times and distances are collected are recorded below.

Case	Distance	Time (Seconds)	Algorithm
test-input-1	5911	0	Nearest Neighbor
test-input-2	8011	.05	Nearest Neighbor
test-input-3	14826	1.22	Nearest Neighbor
test-input-4	19711	11.26	Nearest Neighbor
test-input-5	27128	96.65	Nearest Neighbor
test-input-6	39469	865.65	Nearest Neighbor
test-input-7	N/A	Did not finish in reasonable time	Nearest Neighbor
tsp_example_1	130921	.02	Nearest Neighbor
tsp_example_2	2975	2.03	Nearest Neighbor
tsp_example_3	N/A	Did not finish in a reasonable time	Nearest Neighbor