Research

I did research on: the Held-Karp algorithm, a dynamic programming approach; the nearest-neighbor algorithm; an approximation algorithm that used a straightforward method; and a simulated-annealing algorithm, which uses a decreasing temperature variable to gradually narrow the scope of the search from global to local in order to find an approximate solution.

Held-Karp Algorithm

The Held-Karp algorithm for solving the TSP uses dynamic programming to improve on the worst cast running time of a brute force algorithm. The algorithm was developed independently by Held-Karp and Bellman in 1962. The algorithm works be utilizing the optimal substructure of the problem, and then reduces the running time from O(n!) of a brute force method to $O(n^2 2^n)$.

The optimal substructure is: “Any subpath of a tour of minimum distance is itself of minimum distance”

However, there is no way of knowing whether a subproblem will be necessary to compute, so every subproblem must be looked at.

The algorithm begins looking at the simplest paths (the ones that contain only one edge), and stores the values of each path. Then when calculating more complex paths, determine if the path contains one of the subpaths already computed (which it should if the first part is done correctly) if so, use the stored value instead-of recalculating the value.

The algorithm uses sets to represent subpaths. The method can be phrased as such:

Pick an arbitrary starting node and label it 1. For each subset S of nodes excluding 1, calculate the minimum cost of traversing a path that visits every member of the set (ending at some city in c != 1), and add the cost of traversing from 1 to c. Start with sets of size 1, and repeat the process, incrementing the number of elements, until the subset being used is equal to the set of all nodes excluding 1. The result, after adding the distance of (1, c) will be the minimum distance for a tour of all cities.

Pseudocode for Held-Karp

```
heldkarp(G, n):
#G is the input graph, n is the number of nodes
#pick some node to use as node 1, this might as well be the first in-order element of graph G

    for k from 2 to n:        #loop through each node in the graph
        C ( { k }, k ) = d [ 1, k ]

        #store the length of ( 1, k ) as a tuple: the set of { k } and the single element k
        #C( {k}, k) will hold the value of the shortest path to node k through the set of
        nodes {k}

    for s from 2 to n-1:
        for all subsets of { 2 … n } such that S contains s:
            for all elements k of S:
            C( S, k ) = min( C( { k } m, m ) + d_m, k )
                #for m != k being some elements of s

    optimum = min ( C { 2 … n }, k0 + d_k, 1 )
    #optimum will hold the optimal route
```

Nearest Neighbor Algorithm

The nearest neighbor method of solving the TSP is an approximation algorithm. Since it is an approximation algorithm it does not necessarily find the optimal route. Depending on the vertices that are being analyzed, it may produce a bad result. But for many data sets, it is used to find a pretty good solution. The main concept of the algorithm is to choose the next city based on which unvisited city is nearest to the current city.

The upper bound running time for determining a tour order and a tour distance starting at an arbitrary city using this method would be $O(n^2)$. This is because the algorithm contains two nested loops where every city in the list is examined. The outer loop is used to traverse every city in the list while generating the tour. The inner loop is used each time a city is added to the tour and needs to find the nearest neighboring city to visit. In the worst case, the inner loop would need to examine every city in the list to find the nearest neighbor.

If the starting city is not specified, an arbitrary starting city must be chosen. In the scenario, the results can be improved upon by re-running the algorithm for each city in the list as the starting city. This implementation is called the Repetitive Nearest Neighbor algorithm. While this improves the result, it increases the running time from $O(n^2)$ to $O(n^3)$ since it would require an additional loop to go through the entire list of cities.

Nearest Neighbor Pseudocode

```
read the list of vertices in as an array, V
initialize a variable, D, to indicate the shortest distance found for the optimal solution
initialize array, O, to indicate the best order for the optimal solution
initialize a variable, i = 0, to indicate which city will be the first starting city

while i < length of V, i++
        initialize a variable, d = 0, to store the total distance of the tour
        initialize an array to hold the order, o, of visited vertices in the iteration
        mark all vertices in V as not visited
        initialize the current vertex, C = V [ i ], as the starting city
        Push C into the array of visited vertices, o

        while (length of o) < (length of V):
                find the closest vertex in V that has not been visited
                add the distance from C to N onto d
                push the N into o to keep track of the order
                mark C as visited

        add the distance from C back to the starting vertex to d
        keep track of the best solution that has been found
        if ( d < D )
                D =  d
                O = o
```

Simulated-Annealing Algorithm

The simulated-annealing (SA) algorithm is an approximation algorithm whose general idea is to start with a random path, and then compare it with another random solution which is close to the existing one. And then the algorithm will replace the existing solution with the solution if the new solution is better.

The interesting part about the SA algorithm is that there is a “temperature” variable. This variable starts off high, for example 1000. When the temperature variable is high, the algorithm is more likely to accept a new solution even if it is worse than the existing one. The rationale behind this is to avoid a local optimal solution. Greedy algorithms can fall short by accepting only the immediate optimal solution because it may be trapped in a local optimal solution instead of finding the global optimal solution. As the program runs, the temperature variable is lowered to 0. Towards the end of the run, the algorithm will become a greedy algorithm. By lowering the temperature variable, the algorithm will find an approximate global solution and then iterate to find a local optimal solution.

SA Pseudocode

The step starts from 0 until kmax, during this process, the temperature() function returns decreasing lower temperatures. P() is the acceptance probability function which determines if the new solution will be accepted. E() is the energy function which determines the cost of one solution, and the random function returns a random number between 0 and 1.

Let $s = s_0$
for $k = 0$ through k_{max} (exclusive):
    $T \leftarrow$ temperature (k/k_{max})
    pick a random neighbor, $s_{new} \leftarrow$ neighbor (s)
    if P (E (s), E (s_{new}), T) >= random (0, 1):
        $s \leftarrow s_{new}$
output: the final state s

Selected Algorithm

I found that the simulated annealing algorithm works very well on small datasets but becomes inefficient after n=1000. I found the best results were by using a greedy algorithm such as the nearest neighbor algorithm first and then optimizing the solution using the simulated annealing algorithm. I found that this finds an optimal solution when the input is smaller than n=1000 and then using the nearest neighbor algorithm when the data set is larger than n=1000.