Travelling Salesman Problem Project

Group 9 Team Members:
Zheng Du duz@oregonstate.edu
Patrick Levy levyp@oregonstate.edu
Trevor D Worthey wortheyt@oregonstate.edu

Research

The three algorithms researched were: the Held-Karp algorithm, an exact dynamic programming approach; the Nearest-Neighbor algorithm, an approximate algorithm that uses a straightforward and efficient method; and the Simulated-Annealing algorithm, which uses a decreasing ‘temperature’ variable to gradually “narrow in” or “anneal” the scope of the search from global to local in order to find an approximate solution.

Additionally, because Trevor came in to the implementation side of things late during the week, he chose to separately implement the Christofides algorithm -- especially since the Held-Karp algorithm, while historically interesting, would be impractical to implement for the input sizes provided. The Christofides algorithm has some interesting things to note, and we have included a section discussing its implementation below the discussions of the other three algorithms.

Our team’s research for each algorithm, including pseudocode, is presented below.

Held-Karp Algorithm

The Held-Karp algorithm for solving the TSP utilizes Dynamic Programming to improve on the worst-case running time of a brute force solution. It was actually developed separately by Held-Karp and Bellman, in the same year in 1962. This is an exact algorithm, and by utilizing the optimal substructure of the problem, reduces running time from O(n!) in the case of brute force, to $O(n^2 2^n)$.

The optimal substructure of the TSP can be phrased as such:

"Any subpath of a tour of minimum distance is itself of minimum distance"

Unfortunately, because there is no way to know whether a subproblem will be necessary to compute, we need to look at every possible subproblem.

The algorithm uses a bottom-up approach: begin with the simplest paths, i.e. those containing only one edge, and store 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 you did the first part correctly); and if so, use the stored value instead of re-calculating the value.

The Held-Karp method uses sets to represent sub-paths. 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 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 our 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]

            # That is, store the length of (1,k) as a tuple: the set {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 element of s

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

References

http://www.cs.man.ac.uk/~david/algorithms/graphs.pdf
https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm
http://www.mafy.lut.fi/study/DiscreteOpt/tspdp.pdf
https://www.youtube.com/watch?v=-JjA4BLQyqE

Nearest Neighbor

The nearest neighbor method of solving the Traveling Salesman Problem is a simple and straightforward approximation algorithm. Because it is in an approximation it does not necessarily produce an optimal result. Depending on the vertices that are being analyzed it may actually produce a fairly poor result, but for many data sets it is regarded as a very quick way to obtain a pretty good solution. The main idea behind the nearest neighbor algorithm is to construct a tour by always choosing a next city based simply on which unvisited neighboring city is closest.

The upper bound running time for calculating a tour order and a total tour distance starting at an arbitrary city using this method would be $O(n^2)$. We can understand this running time bound because this algorithm contains two nested loops in which we examine every city in the list. The outer loop is required so that we eventually traverse every city in the list while generating our tour. The inner loop is utilized each time we add a city to our tour and need to find the nearest neighbor to visit next. In the worst case this inner loop could require us to examine every city in the list in order to find the nearest neighbor.

If the starting city is not specified we must choose an arbitrary starting city for our tour. In this scenario we can most likely improve our result by re-running our algorithm for each city in our list as the starting city. This implementation is referred to as the Repetitive Nearest Neighbor Algorithm. While this should improve our result, it also increases our running time dramatically for large data sets. Our upper bound running time would now be $O(n^3)$ since it would require an additional loop through our 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 our "optimal" solution
Initialize an array, O, to indicate the best order for our "optimal" tour
Initialize a variable, i = 0, to indicate which city will be our 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, in which the vertices are visited in this iteration
      Mark all vertices in V as not visited
      Initialize the current vertex, C = V[i], our 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, N
            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 we have found
      If (d < D)
         D = d
         O = o
```

References

https://www.youtube.com/watch?v=E85l6euMsd0
https://en.wikipedia.org/wiki/Travelling_salesman_problem
https://www.math.ku.edu/~jmartin/courses/math105-F11/Lectures/chapter6-part4.pdf
https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

Simulated-Annealing

The Simulated Annealing (SA) Algorithm is an approximation method, so it runs fast, yet it can provide a result which is close to the optimal. As the name suggests, this algorithm is inspired by annealing in metallurgy. Annealing is a process by which a metal is heated to a certain temperature, and then gradually cooled down, reducing defects in the metal.

The basic idea of the algorithm is to start with a random path, and then compare with another random solution which is close to the existing one, replacing the existing solution with the new solution if it is better.

The special part of Simulated Annealing algorithm is that there is a variable called 'temperature', it's a concept from heating the metal, it starts at a high temperature, let's say 500, when the temperature is high, this algorithm is more likely to accept a new solution even it is worse than the existing one, the rationale behind which is to overcome some local optimal solution. If it's a greedy method, only accepting solutions better than the current, it may be trapped in a local optimal situation, and not able to find the global optimal solution. The temperature will be lowered down gradually to 0 throughout the course, which means this algorithm will slowly transform into a greedy algorithm, only accepting solutions better than the current.

By controlling the 'temperature cool down' process, this algorithm will first find where the global optimal roughly locates, and then drill down to a more exact local optimal solution.

Simulated-Annealing Pseudocode
The step starts from 0 until kmax, during this process, the temperature() function returns lower and lower temperature, P() is the acceptance probability function, it determines if the new solution to be accepted, E() is the energy function determines the energy/cost/quality of one solution, and the random function returns 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 neighbour, $s_{new} \leftarrow$ neighbour(s)
    - If $P(E(s), E(s_{new}), T) \geq$ random(0, 1):
        - $s \leftarrow s_{new}$
- Output: the final state s

References

https://en.wikipedia.org/wiki/Simulated_annealing
http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/
https://www.youtube.com/watch?v=0rPZSyTgo-w
https://www.youtube.com/watch?v=W-aAjd8_bUc

Christofides

The Christofides algorithm for TSP was developed by Nicos Christofides and published in 1976. To this day, it remains the algorithm with the best guaranteed approximation ratio of 1.5 (3/2) times the optimal solution. The algorithm is constrained to certain inputs, however -- namely, the graph must be in a metric space (e.g., drawable on a Euclidean / Cartesian plane).

Christofides uses several sub-algorithms to arrive at a final answer. Given a graph G = (V, w), the algorithm proceeds as follows:

```
* Create a Minimum Spanning Tree T of G
* Find the set of vertices with odd degree in T
* Find a minimum weight perfect matching M in the induced sugraph from vertices O
* Form a multigraph H by combining the edges in M and T (a multigraph can have multiple edges for each vertex pair)
* Form a Eulerian circuit in H (one that traverses every edge exactly once)
* Create a Hamiltonian circuit from the Eulerian one by bypassing repeated vertices
```

By far the most difficult aspect of implementing this algorithm was finding a solution for Minimum Weight Perfect Matching for a general graph. The most common algorithms, the Hopcroft-Karp, Hungarian, and Ford-Fulkerson algorithms are designed only for weighted or unweighted bipartite graphs (ones in which the vertices can be divided into two disjoint sets). The Edmonds Blossom algorithm is the de-facto algorithm for computing MWPM, and is easy to understand in theory, but turned out to be quite complex to implement, given it in itself requires several sub-algorithms based on finding augmenting paths, plus expanding and contracting 'blossoms'.

There is a very good in-depth explanation of the Blossom algorithm here:
http://demonstrations.wolfram.com/TheBlossomAlgorithmForWeightedGraphs/

Ultimately I ended up using a working Python implementation of the Blossom algorithm from github user koniiiik to complete the Christofides implementation. There were two other helpful implementations found online, of the Eulerian Circuit and Minimum Spanning Tree algorithms, which proved useful sources for understanding those subproblems, and adaptations of these implementations made their way into my final code. Citations are included where appropriate in christofides.py and below.

Unfortunately, the results of the Christofides algorithm did not prove to be fruitful compared to the other algorithms we implemented. The average performance suffered greatly as data sets grew large -- of the 7 test cases, only the first 5 ran in a reasonable time on flip. This may be in part due to the unoptimized code in blossom.py, but unfortunately there was not time to refactor it. As far as answers goes, of the two results that were able to run in the examples, both came

out well above the 1.25x optimal threshold -- though the asnwers were in keeping with the promised hard max of 1.5.

While the Christofides algorithm did not turn out to be very efficient or provide the best answers, over the course of implementing it I learned more on the subject of graph theory than I knew existed, and got the chance to investigate the implementations of a number of interesting sub-algorithms. I am extremely glad I chose to undertake this challenge, even if I was not able to write every piece of code myself

References and Sources

https://en.wikipedia.org/wiki/Christofides_algorithm
https://en.wikipedia.org/wiki/Matching_(graph_theory)
http://demonstrations.wolfram.com/TheBlossomAlgorithmForWeightedGraphs/
https://en.wikipedia.org/wiki/Eulerian_path

[MSTs] https://gist.github.com/siddMahen/8261350
[Eulerian Circuit] https://github.com/feynmanliang/Euler-Tour/blob/master/FindEulerTour.py
[Blossom Algorithm] https://github.com/koniiiik/edmonds-blossom

Selected Algorithm

When testing our implementations, we found that the Simulated Annealing algorithm performed well on small datasets, but became inefficient after about n=1000. Our best results by using the Simulated Annealing algorithm to further optimize a tour that was initially obtained using the Nearest Neighbor Algorithm. This method worked quite well, but becomes too slow to be within the bounds of the 3 minute time limit for large data sets.

We decided to use the Simulated Annealing algorithm with Nearest Neighbor Start when input size is smaller than 1000, since it leads to better results, and use Nearest Neighbor by itself when the data set is greater than 1000, which can give us better running speed.

The running result can be found in Figure 1.

Final Results - All runs are under 3 minutes

	Final Algorithm: Simulated Annealing with Nearest Neighbor Start on sets < 1000, Nearest Neighbor on sets > 1000	
	distance	time
example 1 (optimal: 108159)	110213 (1.019 x optimal)	0:00:08.175879
example 2 (optimal: 2579)	2856 (1.107 x optimal)	0:00:28.642953
example 3 (optimal: 1573084)	1964948 (1.249 x optimal)	0:03:14.140106
test case 1	5373	0:00:05.551701
test case 2	7547	0:00:10.714713
test case 3	13093	0:00:23.989541
test case 4	19014	0:00:50.612805
test case 5	27749	0:01:43.743609
test case 6	40933	0:00:03.156718
test case 7	63780	0:00:20.020647

Figure 1, The running result using final solution

Best Tour for Example Instances - No Time Limit

	Best Tour for Example Instances - No Time Limit		
	distance	time	
example 1 (optimal: 108159)	110213 (1.019 x optimal)	0:00:08.175879	Simulated Annealing with Nearest Neighbor Start at fast cooling rate
example 2 (optimal: 2579)	2856 (1.107 x optimal)	0:00:28.642953	
example 3 (optimal: 1573084)	1964948 (1.249 x optimal)	0:03:14.140106	Nearest Neighbor

Best Tour for Competition Instances - 3 Minutes

	Best Tour for Competition Instances - 3 Minutes		
	distance	time	
test case 1	5373	0:00:05.063916	Simulated Annealing with Nearest Neighbor Start at fast cooling rate
test case 2	7547	0:00:10.714713	
test case 3	13093	0:00:23.989541	
test case 4	19014	0:00:50.612805	
test case 5	27749	0:01:43.743609	
test case 6	40933	0:00:03.156718	Nearest Neighbor
test case 7	63780	0:00:20.020647	

Best Tour for Competition Instances - No Time Limit

	Best Tour for Competition Instances - No Time Limit		
	distance	time	
test case 1	5373	0:00:05.063916	Simulated Annealing with Nearest Neighbor Start at different cooling rates
test case 2	7524	0:00:10.714713	
test case 3	12658	0:51:21.620264	
test case 4	17908	1:23:28.536752	
test case 5	25479	2:53:40.008337	
test case 6	38128	5:28:31.147020	
test case 7	63780	0:00:20.020647	Nearest Neighbor

Tour Comparisons

	Nearest Neighbor		Repetitive Nearest Neighbor	
	distance	Time (h:m:s)	distance	Time (h:m:s)
example 1 (optimal: 108159)	150393 (1.39 x optimal)	0:00:00.005715	130921 (1.21 x optimal)	0:00:00.390988
example 2 (optimal: 2579)	3210 (1.245 x optimal)	0:00:00.068573	2975 (1.15 x optimal)	0:00:17.480679
example 3 (optimal: 1573084)	1964948 (1.249 x optimal)	0:03:12.593408		> several hours

test case 1	5926	0:00:00.002381	5911	0:00:00.121639
test case 2	9503	0:00:00.008397	8011	0:00:00.810149
test case 3	15829	0:00:00.054093	14826	0:00:12.441197
test case 4	20215	0:00:00.204929	19711	0:01:40.864690
test case 5	28685	0:00:00.775977	27128	0:13:10.079939
test case 6	40933	0:00:03.056673	39469	1:43:38.161108
test case 7	63780	0:00:19.048232		> several hours

	Lin Kernighan		Simulated Annealing with Nearest Neighbor Start			
			Cooling Rate: Fast		Cooling Rate: Normal	
	distance	time	distance	time	distance	time
example 1	117216	00:01.84	115216	00:15.34	114961	01:15.96
example 2	2765	01:10.45	2815	00:37.73	2771	04:25.78
example 3			1962539	28:55.55	1960076	4:56:25.22
test case 1	5786	00:00.85	5497	00:05.17	5373	00:53.54
test case 2	8064	00:07.44	7659	00:09.61	7699	01:39.12
test case 3	13130	03:12.36	13484	00:23.87	12719	03:53.37
test case 4			19354	00:47.60	17963	08:06.56
test case 5			27744	01:33.28	26597	15:37.69
test case 6			40831	03:08.56	39901	31:40.41
test case 7			64377	08:15.97	69750	1:32:28.82

	Simulated Annealing

	Cooling Rate: Fast		Cooling Rate: Normal		Cooling Rate: Slow	
	distance	time	distance	time	distance	time
example 1	114442	00:17.15	111801	01:33.82	111213	51:18.78
example 2	3363	00:30.59	2848	06:39.49	2792	2:53:36.31
example 3	58727736	29:49.98				
test case 1	5472	00:11.58	5395	01:56.56	5333	35:51.66
test case 2	7992	00:21.67	7689	03:38.73	7384	1:06:22.11
test case 3	15502	00:36.60	12978	05:17.07	12628	2:42:37.87
test case 4	30885	01:11.97	19343	11:40.72	18028	5:21:56.88
test case 5	76427	02:04.02	35044	19:36.09	26141	7:51:12.87
test case 6	207058	03:49.32	76435	53:33.31		
test case 7	758518	09:56.53	260891	3:23:01.80		

	Christofides	
	distance	Time (h:m:s)
example 1 (optimal: 108159)	149368 (1.38 x optimal)	0:00:00.192494
example 2 (optimal: 2579)	3736 (1.45 x optimal)	0:00:17:169692
example 3 (optimal: 1573084)		
test case 1	7470	0:00:00.056303
test case 2	10302	0:00:00.321526
test case 3	17479	0:00:08.705200

test case 4	23698	0:01:15.744157
test case 5	33087	0:16:41.897662
test case 6		
test case 7		