Aaron Boutin, Jessica Spokoyny, Edward Francis
CS 325: Analysis of Algorithms
Group Project 4: Methods/Algorithms for solving the Traveling Salesman Problem
March 17, 2017

The Traveling Salesperson Problem and approaches to solving it:

The Traveling Salesman Problem supposes a complete graph G with nonnegative edge weights, and seeks a minimum-weight cycle that visits each vertex in G exactly once. In other words, the solution is an optimal, minimum-weight, simple path starting and ending at the same vertex. The name of the problem comes from a typical scenario in which it arises: give a traveling salesman who must visit some number of cities before returning to the home office, what is the best, shortest route that the salesperson should take to visit every city on his or her schedule, without visiting a single city more than once? For this project, we added the further constraint that the cost of edges in the graph represent actual real world distances between points (cities) on a map. This allowed us to make use of approximation algorithms that require the graph data to respect the triangle inequality property, namely that any side of a triangle is shorter than the combined length of the triangle's other two sides.

1. Description of at least three different methods/algorithms for solving the Traveling Salesman Problem. Summarize any research.

There have been a great many algorithms proposed as methods for solving the traveling salesman problem efficiently, but to date the problem remains in the class of difficult problems for which no known polynomial-time solving algorithm is known to guarantee an optimal solution (although it is possible to verify a solution to the TSP in polynomial time). A brute force method can be used to the best tour within a given graph, but it is prohibitively time-consuming to apply as the number of cities to be visited grows large. Faster algorithms, such as a nearest-neighbor algorithm, can be used to generate a "quasi-optimal" solution to the TSP, but they do not guarantee that the solution provided will be perfectly optimal, or even within a specified distance from an optimal solution. Slightly more complex, yet still polynomial time, approximation algorithms have been developed that assure a solution that is within some approximation bound of the optimal tour length. 2-Approximation and Christofides' Algorithms are two such examples. In addition, there has been so much research into TSP algorithms that there are algorithms that can be applied to the solutions of other TSP algorithm as a sort of "second pass," to improve on a known tour and bring it closer to optimal. 2-OPT is an example.

Brute Force: O(n!)

The Brute Force method would make a list of all possible Hamiltonian Circuits, exhaustive. It calculates the weights for each of the circuits by adding up the weights of its edges. It then compares each of these weights and chooses the one with the minimum total weight. This takes up much time and space, but while so, it is guaranteed to find the optimal solution but not in polynomial time.

Source:
https://www.math.ku.edu/~jmartin/courses/math105-F11/Lectures/chapter6-part3.pdf

Nearest Neighbor

This algorithm first sets the starting city as the first city in the tour, which can be arbitrarily chosen. Then, it determines the cost of each edge of every adjacent vertex and selects the minimum weight edge. It then sets that vertex as the current, adds it to a set of visited vertices, and repeats, choosing the minimum weight edge of each adjacent vertex. Once every vertex has been visited, it is connected back to the original starting vertex. This is a greedy approach, but is relatively quick compared to other approximation algorithms.

Sources:
https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

2-Approximation Algorithm (Metric TSP)

The idea behind this algorithm follows from first finding a minimum spanning tree T of an undirected, weighted graph G. Then it duplicates each edge in the tree to make a Eulerian graph. We then take a Eulerian tour of this graph and convert it to a tour by going through each vertex in the same order in the tour, but skipping vertices already visited.

Sources:
https://people.orie.cornell.edu/dpw/orie6300/Recitations/Rec12.pdf
http://www.sfu.ca/~arashr/lecture25.pdf
http://www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf

Christofides' Algorithm

Much like the 2-Approximation Algorithm (Metric TSP), Christofides' algorithm takes an undirected, weighted Graph, G, and returns a 1.5-Approximation of the TSP. To do this, it first calculates a minimum spanning tree of graph G, G'. It determines a subset of vertices of G' that are of odd degree, which will be an even number. This subset is then processed into a minimum cost perfect match graph M. This will connect two vertices together, increasing their degree by one. Since it is a perfect match, all vertices will be used with each overall getting an increase in their degree by one. We then Union M and G'. This will make the odd vertices of G' even, and

add more edges. The result will be a graph in which we can make an Euler tour about, where each vertex is of even degree. This tour is performed as a DFS. The resulting tour will contain duplicate vertices. When then go through and where it comes upon a duplicate, it eliminates it from the path and connects the previous vertex to the next vertex from the eliminated vertex's position. The result is our approximated solution.

Sources:
http://www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf
https://people.orie.cornell.edu/dpw/orie6300/Recitations/Rec12.pdf
https://www.youtube.com/watch?v=zM5MW5NKZJg

2-OPT (improvement)
The main idea of this algorithm is to improve on an existing tour. We can take a tour generated by the Nearest Neighbor or Christofides' algorithms and make it closer to the optimal tour. The 2-opt algorithm reverses successive sections of a tour, adjusting endpoints each time, and checks if the resulting tour has a shorter length. If it does, then the tour is updated to the new route, otherwise it is left in place and the next "swap" is consider. Essentially, it takes a tour that has overlapping edges, and rearranges it so that the edges no longer cross over each other.

Sources:
https://en.wikipedia.org/wiki/2-opt
http://cs.indstate.edu/~zeeshan/aman.pdf

2. Verbal description of our algorithm(s) as completely as possible. We can select more than one algorithm to implement.

For this project, we elected to implement three basic types of algorithms: greedy, approximation, and second-pass optimization algorithms.

For our greedy algorithm we chose to implement a nearest neighbor algorithm. As described above, the algorithm takes a single city within the graph as a starting point and adds it to the route. It then begins by calculating the distance from that vertex to all other vertexes in the graph that have not been visited yet (that initially being all of the other vertexes). The closest next vertex is then made the new "current" vertex and is added to the route. As before, distances to all remaining neighbors are calculated, the closest vertex is added to the route and the process shifts to take that new vertex as the current vertex. The algorithm runs until all vertexes have been visited, so that the final vertex has no neighbors left to visit, and the tour is completed by traveling any remaining distances from that final vertex back to the starting vertex. The tour length is calculated as the sum of the distances between vertexes in the tour.

For our approximation algorithms, we tried implementing two variants: 2-approximation and Christofides' algorithm. Both algorithms begin in a similar fashion by finding a minimum spanning tree (MST) within the input graph. Prim's algorithm is used for both of our algorithms, because it was simple to implement, even with a very dense graph with numerous edges.

After finding the MST, the next step in the 2-approximation algorithm is to identify an Eulerian Circuit within the graph containing only the MST edges and then removing redundant vertices. Both steps are accomplished in our implementation by using a DFS from some starting vertex and listing the vertexes in the order in which they are first visited. The tour length is then calculated as the sum of the distances between the vertexes in the order listed, plus the distance from the final vertex back to the starting vertex.

For Christofides' algorithm, there is an additional step in between finding the MST and identifying an Eulerian Circuit, as is done in 2-approximation. The additional step is to find all vertexes in the MST graph with an odd number of connections, finding a "minimum cost perfect matching" between them, and adding the perfect matching edges to the MST graph. It turns out that finding a minimum cost perfect matching is a challenging problem in its own right, and we adopted a simplified solution that simply matches the odd vertices with the closest unmatched odd vertex, one by one (or two-by-two, depending on how you consider them to be removed from consideration). This greedy algorithm is not perfect but it has the benefit of being fast. Following the perfect matching step, our algorithm again uses a DFS to place the vertex in order, and then finds the route cost as the sum of the distances between them plus travel back from last vertex to first.

To each of our approximation algorithms 2-approximation and Christofides', we also added a step that re-runs the DFS from each possible starting vertex in the MST, and updates the tour length and tour order if a better solution is found.

Finally, for our second-pass optimization algorithm, we implemented a 2-opt algorithm that operates on a tour generated by our nearest neighbor, 2-approximation, or Christofides' algorithms to squeeze out a little bit more performance. The 2-opt algorithm iteratively tests variants of the starting tour that reverse some section of the vertexes between two points in the tour, to see if this produces a shorter tour. Considered graphically, it effectively attempts to find a shorter tour by un-crossing edges that are overlaid, so that they swap their connections to the endpoints of some segment of the tour that lies between them.

It is worth noting that 2-opt algorithm is by far the most time-consuming of our algorithms, though this time was frequently well-spent and resulted in much shorter tour lengths in several cases when allowed to run to completion. In order to keep with time limits imposed for certain

sections of the project, we incorporated a timing mechanism into our 2-opt loops (as well as into the DFS comparison portion of the approximation algorithms, with instructions to break and wrap up the results with the best known tour when a particular time limit was reached.

3. Discussion of why we selected the algorithm(s).

Trade-offs between computing time and quality of results were behind most of our decisions about which algorithms to implement. Our main reason for using nearest neighbor, for example, was speed - it produced decent or better results for the sample data for the largest data sets, when our slower approximation and optimization second-pass algorithms were unable to produce a good quality result within the time limits. For our approximation algorithms, we started by using 2-approximation as a way of ensuring a higher quality result, guaranteed to be within a factor of 2 of optimal, when time was not a factor, or when the size of the input data was small. We then moved to implement Christofides' algorithm because it had a reputation for providing even better results, guaranteed to be within a factor of 1.5 over optimal, while requiring similar time and memory resources to complete. Our Christofides' implementation would probably have completely replaced our 2-approximation implementation, except for the fact that our implementation took several shortcuts that kept its advantages over 2-approximation from being fully realized. First, our greedy approach to "minimum cost perfect matching" is probably better than random matching, but it is does not truly generate a "minimum cost" matching. Second, we were not completely confident that our DFS approach to ordering the vertexes was still appropriate after adding in the minimum cost perfect matching edges, or whether a more advanced approach such as Fleury's Algorithm was necessary to make the best use of the new edge information. Due to this uncertainty, we decided to use both 2-approximation and Christofides' algorithms to find the best possible result for each input data set.

4. Pseudocode for the algorithm(s).

Here is pseudocode for our Nearest Neighbor, 2-Approximation, Christofides', and 2-Opt algorithms.

```
NEAREST-NEIGHBOR(G, startingVertex V)
        Let citiesList be an array of size equal to the number of vertexes in G
        Set current = V
        Set bestPoint = V
        For(i = 0; i < number of Cities; i++)
                Let current = best
                Let current = bestPoint, citiesList[i] = current
                For each remaining city: (j = 0; j < number of Cities; j++)
```

Skip cities that have already been visited

Calculate distance to city and track shortest distance and city with that distance as “best”

Add closest city found to citiesList, and make that city the new current

Return cost of tour, sequence of cities giving that cost (citiesList)

2-APPROXIMATION(G, startingVertex V)

From V, use Prim’s algorithm to create MST for G, graph G1

Use DFS on G1 from V to place vertexes in order to visit

Get cost

For each vertex U in G1 not V, use DFS to place vertexes in order, and get tour cost

If tour cost is less, record U as best starting point, UCost as cost of tour from that point

Use DFS starting with U to update tour from best starting point

Return UCost and tour generated with U as starting point

CHRISTOFIDES(G, startingVertex V)

From V, use Prim’s algorithm to create MST for G, graph G1

Identify odd vertexes by getting the number of neighbors for each vertex in G1

Use greedy algorithm to match each odd vertex with a relatively close neighbor, and add edge between each matched pair to G1

Use DFS on updated MST graph G1 from V to place vertexes in order to visit

Get cost

For each vertex U in G1 not V, use DFS to place vertexes in order, and get tour cost

If tour cost is less, record U as best starting point, UCost as cost of tour from that point

Use DFS starting with U to update tour from best starting point

Return UCost and tour generated with U as starting point

2-OPT(G, &bestTour, &bestTourLength)

Let bestTour be an array of ints indicating the cities in known “best order”

Let bestTourLength be the length of the tour represented by BestTour

Let newTour be an array of ints of size equal to bestTour

Let newTourLength be an int that will hold the length of tempBestTour tour

Int improved = 1

while (improved == 1)

Let improve = 0

```
Let  bestTourLength be the length of the tour represented by BestTour
for (int i = 0; i < numCities - 1; i++)
        for (int j = i+1; j < numCities; j++)
                Create newTour by performing two-opt swap( In a temporary array of length total vertexes, swap i and j and place the vertexes between them in reverse order)
                Calculate newTourLength of temporary array
                If newTourLength < bestTourLength
                        Let bestTourLength = newTourLength
                        Let bestTour = newTour
                        Improved = 1
```

---

5. Our "best" tours for the three example instances and the time it took to obtain these tours. No time limit.

Our best tour length and time requirements are summarized in the following table. Our experimental data attached at the end of this report provides more detail on the algorithms used to generate each tour.

Source file	**Best tour length**	**Time (seconds)**
tsp_example_1.txt	111267	0.03
tsp_example_2.txt	2723	8.5
tsp_example_3.txt	1964948	6609

6. Our best solutions for the competition test instances. Time limit 3 minutes, and unlimited time.

Our best tour length and time requirements are summarized in the following table. Our experimental data attached at the end of this report provides more detail on the algorithms used to generate each tour.

	Unlimited Time		**Under 3 Minutes**	
Source file	**Best tour length**	**Time (seconds)**	**Best tour length**	**Time (seconds)**
test-input-1.txt	5382	0.012	5382	0.012

test-input-2.txt	7856	0.23	7856	0.23
test-input-3.txt	12624	11.16	12624	11.16
test-input-4.txt	17164	240.001	18407	175.01
test-input-5.txt	23567	5235.23	28171	150.56
test-input-6.txt	33164	151391	40933	22.56
test-input-7.txt	62587	3537.37	72141	175.43

Appendix: Experimental results

Example 1:

Number of cities = 76

Optimal = 108159

	Best tour length (unlim. time)	**Time (seconds)**	**Ratio to optimal**
NN	150393	0.04	1.39
NN + 2-OPT	125838	0.13	1.16
2-APPROX	123348	0.001574	1.14
2-APPROX + 2-OPT	113098	0.031246	1.03
CHRISTOF	134072	0.001611	1.39
CHRISTOF + 2-OPT	111267	0.03	1.01

Example 2:

Number of cities = 280

Optimal = 2579

	Best tour length (unlim. time)	**Time (seconds)**	**Ratio to optimal**
NN	3210	0.06	1.24
NN + 2-OPT	3069	1.98	1.19

2-APPROX	3584	0.052063	1.35
2-APPROX + 2-OPT	2789	10.41	1.06
CHRISTOF	3469	0.043519	1.44
CHRISTOF + 2-OPT	2754	13.1	1.06

Example 3:

Number of cities = 15112

Optimal = 1573084

	Best tour length (unlim. time)	**Time (seconds)**	**Ratio to optimal**
NN	1964948	6609	1.25
NN + 2-OPT	--	--	--
2-APPROX	2216826	179.116	1.41
2-APPROX + 2-OPT	2213027	176967	1.41
CHRISTOF	2451100	10002.5	1.56
CHRISTOF + 2-OPT	2452085	180.317	1.56

Test 1:

Number of cities = 50

Concorde solution: 5333[1]

	Unlimited Time			**Under 3 Minutes**		
	Best tour length	**Time (seconds)**	**Ratio to Concorde**	**Best tour length**	**Time (seconds)**	**Ratio to Concorde**
NN	5926	0.01	1.11	5926	0.01	1.11

[1] To get a sense for how our algorithms were performing with the test case data, we used an online TSP solver called Concorde: https://neos-server.org/neos/solvers/co:concorde/TSP.html . While the solver does not guarantee a perfect optimal tour solution, it provided us with a baseline number against which to measure our progress.

NN + 2-OPT	5637	0.16	1.06	5637	0.16	1.06
2-APPROX	6758	0.000625	1.27	6758	0.000625	1.27
2-APPROX + 2-OPT	5382	0.011617	1.01	5382	0.011617	1.01
CHRISTOF	6809	0.000627	1.28	6809	0.000627	1.28
CHRISTOF + 2-OPT	5759	0.009063	1.08	5759	0.009063	1.08

Test 2:

Number of cities = 100

Concorde solution: 7384

	Unlimited Time			**Under 3 Minutes**		
	Best tour length	**Time (seconds)**	**Ratio to Concorde**	**Best tour length**	**Time (seconds)**	**Ratio to Concorde**
NN	9503	0.58	1.29	9503	0.58	1.29
NN + 2-OPT	9375	0.16	1.27	9375	0.16	1.27
2-APPROX	9759	0.003458	1.32	9759	0.003458	1.32
2-APPROX + 2-OPT	7971	0.153948	1.08	7971	0.153948	1.08
CHRISTOF	10886	0.002951	1.47	10886	0.002951	1.47
CHRISTOF + 2-OPT	7856	0.229137	1.04	7856	0.229137	1.04

Test 3:

Number of cities = 250

Concorde solution: 12067

	Unlimited Time			Under 3 Minutes		
	Best tour length	**Time (seconds)**	**Ratio to Concorde**	**Best tour length**	**Time (seconds)**	**Ratio to Concorde**
NN	15829	0.14	1.31	15829	0.14	1.31
NN + 2-OPT	15405	2.20	1.28	15405	2.20	1.28
2-APPROX	16543	0.038989	1.37	16543	0.038989	1.37
2-APPROX + 2-OPT	12952	8.74845	1.07	12952	8.74845	1.07
CHRISTOF	18315	0.032311	1.52	18315	0.032311	1.52
CHRISTOF + 2-OPT	12459	11.1627	1.05	12459	11.1627	1.05

Test 4:

Number of cities = 500

Concorde solution: 16720

	Unlimited Time			Under 3 Minutes		
	Best tour length	**Time (seconds)**	**Ratio to Concorde**	**Best tour length**	**Time (seconds)**	**Ratio to Concorde**
NN	20215	0.41	1.21	20215	0.41	1.21
NN + 2-OPT	19576	26.28	1.17	19576	26.28	1.17
2-APPROX	22859	0.322231	1.37	22859	0.322231	1.37
2-APPROX + 2-OPT	17164	150.311	1.03	17164	150.311	1.03

CHRISTOF	25122	0.240784	1.50	25122	0.240784	1.50
CHRISTOF + 2-OPT	17374	291.294	1.04	19819	175.002	1.19

Test 5:

Number of cities = 1000

Concorde solution: 22976

	Unlimited Time			**Under 3 Minutes**		
	Best tour length	**Time (seconds)**	**Ratio to Concorde**	**Best tour length**	**Time (seconds)**	**Ratio to Concorde**
NN	28685	2.80	1.25	28685	2.80	1.25
NN + 2-OPT	28171	150.56	1.23	28171	150.56	1.23
2-APPROX	31749	3.73797	1.38	31749	3.73797	1.38
2-APPROX + 2-OPT	23567	5235.23	1.03	31373	175	1.37
CHRISTOF	34999	2.77501	1.52	34999	2.77501	1.52
CHRISTOF + 2-OPT	23615	5893.01	1.03	34417	175.004	1.50

Test 6:

Number of cities = 2000

Concorde solution: 32448

	Unlimited Time			**Under 3 Minutes**		
	Best tour length	**Time (seconds)**	**Ratio to Concorde**	**Best tour length**	**Time (seconds)**	**Ratio to Concorde**
NN	40933	22.56	1.26	40933	22.56	1.26

NN + 2-OPT	39665	844.02	1.22	40933	22.56	1.26
2-APPROX	45181	26.8314	1.39	45181	26.8314	1.39
2-APPROX + 2-OPT	33164	151391	1.02	45171	175	1.39
CHRISTOF	49286	22.3039	1.52	49286	22.3039	1.52
CHRISTOF + 2-OPT				49261	175.001	1.52

Test 7:

Number of cities = 5000

Concorde solution: 50577

	Unlimited Time			**Under 3 Minutes**		
	Best tour length	**Time (seconds)**	**Ratio to Concorde**	**Best tour length**	**Time (seconds)**	**Ratio to Concorde**
NN	63780	273.39	1.26			
NN + 2-OPT	62587	3537.37	1.24			
2-APPROX	72112	543.52	1.44	72141	175.384	1.43
2-APPROX + 2-OPT				71849	175	1.42
CHRISTOF	80553	356.034	1.60	80553	175.161	1.60
CHRISTOF + 2-OPT				80553	175.485	1.60