Keisha Arnold, Jacob Karcz, David Haskell
Group 11
CS 325: Project 3 Report

A description of at least three different methods/algorithms for solving the Traveling Salesman Problem along with pseudocode. Summarize any other research your group did.

In researching the various algorithmic solutions to the Traveling Salesman Problem (TSP), we found that the algorithms were categorized as either exact or heuristic (approximation) algorithms.

Because the TSP is considered NP-complete, exact algorithms are not very efficient, especially for problems with large numbers of cites. Indeed, a Brute Force algorithm is given the asymptotic complexity of O(n!), which for large numbers of cites, could take days for an exact solution.

As for approximation algorithms, the emphasis and research is not so much asymptotic complexity and speed, but the guarantee of how close the solution is to the optimal solution. One of the first approximation algorithms was Christofides' algorithm, which guarantees a solution which is at most 1.5 times the optimal. Recent research has reached a guaranteed solution of 1.4 times the optimal.

There are many approximation algorithms that do not have an upper bound, or guaranteed solution, but still remain useful due to their simplicity to implement and their short run times. One such algorithm is the Nearest Neighbor algorithm, which has many different forms and implementations. It is discussed below. It is also an interesting solution algorithm in that it describes most accurately how a human may approach the problem, as if there actually were a 'traveling salesman" visiting a list of cities, and returning home. Indeed, much research has been done in the area of cognitive psychology that studies how the human mind approaches such a problem.

Our group also discussed the benefits of a potential unexpected breakthrough in which it is shown that P = NP. This would negate this entire project, and we would each get an A+!!!!

Exact algorithms:

Brute Force - O(n!)

For our TSP problem, if we are given a set of n cities and the distance between each pair of cities, we need to find the minimum path length that visits each city exactly once and terminates at the starting city. The brute force algorithm will examine all possible permutations of cities and keep the path that is the shortest. There are n! permutations of n cities, but since we are using one city as our starting and ending point then we actually need to look at n-1! permutations. Therefore, our brute-force algorithm will take O(n!) time.

Solving TSP by brute force has some obvious benefits and drawbacks. It is straightforward and quite easy to implement since it just looks at all possible paths and finds the shortest one. Unlike the approximation or greedy algorithms it is guaranteed to find the optimal solution. However, given that the brute force algorithm examines all possible permutations of cities, it's obvious that as our set of cities grows larger, we'll have more cities to look at and thus it will take a very long time to get an optimal answer. So for a small number of cities, the brute force algorithm is a good choice since we'll be guaranteed an optimal answer and will take O(n!) time given n cities. But as our set of cities grows larger the running time goes up exponentially. As an example, if we had a set of 11 cities, our brute force algorithm would look at 10! = 3,628,800 permutations. If we added one more city to our set we would be looking at 11! = 39,916,800 permutations. This means we're looking at 11 times the amount of permutations just by adding a single city, meaning we can expect our running times to be about 11 times slower with 12 cities compared to 11 cities.

The following is pseudocode for a TSP brute force algorithm, it takes as input a map/list of cities and outputs the best tour and best score found. We could have a TSP_map class which would provide us with some helper functions to add cities to our map, calculate the distance between cities, calculate the score of a given tour/path, and get a starting tour.

```
TSP_brute_force(TSP_map cities) {
        T <= get an initial tour
        best_tour <= T
        best_score <= score(T)

        while there are still more permutations of T
                generate a new permutation of T
                if score(T) < best_score
                        best_tour <= T
                        best_score <= score(T)
        print best_tour and best_score
```

Bellman-Held-Karp (DP)- O(2^n•n^2):

```
BHK algorithm TSP (Graph G, n cities)
        #case for the subset of cities where each set is a single city
        for k = 2 to n do
                C({k}, k) = d(1, k)
        end for

        #case for subsets of containing 2 to n cities
        for s = 2 to n-1 do
                for all S ⊆ {2, . . . , n}, |S| = s do
                        for all k ∈ S do
                                #min distance for a complete tour:
                                {C (S, k) = min(m ≠ k, m ∈ S) [C (S − {k}, m) + d(m, k) ]}
                        end for
                end for
        end for

        opt = min(k ≠ 1) [C ({ 2, 3, . . . , n}, k) + d(k, 1)]
        return (opt)
end
```

The Bellman/Held-Karp algorithm has a an excruciatingly slow worst case execution time of O(2^n•n^2) (and a space consumption of O(2^n•n). Though this dynamic programming implementation is technically better than O(n!) brute force method, it is still too slow for calculating large sets of data.

This complicated algorithm computes the solutions to all subproblems starting with the smallest and chooses the optimal path for all nodes so far. Like other DP solutions, it keeps a copy of all smaller solutions that have already been computed for the larger problems to reference. Since it doesn't know which subproblem will yield the best solution, it computes every path and chooses the best one. To compute the minimum distance TSP tour, it uses the final equation to generate the lst node, and repeats for the other nodes. At the end it simply selects the optimal solution from its stockpile of solutions.

Approximation algorithms:

Nearest Neighbor -1.25 optimal (on average) - Greedy - O(lgV) to O(n^2)

The Nearest Neighbor (NN) algorithm is a greedy algorithm that starts in a randomly chosen “home” city and simply chooses the nearest unvisited city. From there, it simply repeats this choice, until all cities are visited, at which point, it returns to the home city.

The NN algorithm can be implemented in several ways. If the cities are viewed as nodes and the distances between cities are weighted edges, then the TSP can be solved in a similar fashion as a Minimum Spanning Tree (MST) problem. Solutions can be found using Prim’s or Kruskal’s algorithms. Each of these can give a solution for an MST in anywhere from O(VlgV+E) to O(V^2) running time. The running times can depend upon what data structure is used. Each of these algorithms do require an initial sorting of the weighted edges, which can be done in O(V) or O(lgV) run time. The difference in the TSP algorithm and the MST algorithms is that the TSP algorithm does not allow for nodes to have a degree of more than two. This creates a cycle, rather than a tree.

The NN algorithm is an approximation algorithm, and can find solutions that are, at average, <= 1.25 the optimal solution. It can, however, give solutions that are > 2 the optimal. But because it is a greedy algorithm, it is fairly fast (O(n^2)), and therefore can be run multiple times, once for each of the cites as the home city, thereby overlooking the worst case paths.

The following pseudocode is based upon Prim’s algorithm with a 2D array data structure. It’s running time is O(N^2).

```
TSP_NN_Prims {
        Find all distances between nodes, store in sorted 2D array d[N][N]
        Add all vertices into list Q[N]
        Initialize routeArray[N] to NULL
        pathLength = 0

        Begin at homeNode = u
        Add u to routeArray
        While Q not empty{
                Choose minimum w(u, v) for all v Adj[u] in Q
                Add w(u, v) to pathLength
                Add u to routeArray
                u = v
        }
        Add w(u, firstNode) to pathLength  // return to home node
}
```

Another version of a greedy algorithm is based upon Kruskal’s algorithm. It chooses the shortest untraveled path on the entire map first, connecting nodes as necessary, making sure

not to form a cycle until the last edge is added to the path. It has a similar running time of O(N^2).

```
TSP_ShortestEdgeFirst_Kruskals {
        Find all distances (edges) between nodes, store in sorted 2D array d[N][N]
        Add all vertices into list Q[N]
        pathLength = 0

        While (edges exist in d[N][N] that != infinity) {
                (Choose minimum edge in d[N][N]
                If (edge is not incident with a vertex with degree of two OR will not form a cycle{
                        Add edge distance to path length
                        Remove edge from d[N][N] //  make it = infinity
                }
        }
        Add last remaining edge distance to pathLength  // complete the cycle
```

The above algorithm will return the pathLength, but not the path itself (the ordered list of nodes). This can be achieved by using an appropriate data structure (such as a linked list) to store information such as adjacent nodes.

Christofides' TSP Algorithm- 1.5 optimal with a O(n^4) running time

```
christofidesTSP(G, n)
        Find minimum spanning tree T for (G, n)
        Let O be the set of vertices with odd degree (O will have an even number of vertices)
        Let M be the minimum cost perfect matching given by the vertices in subgraph O
        Let G' be the union of MST T and matching edges M (each vertex has even degree)
        Let E be a Euler tour in G'
        Then make E into a Hamiltonian circuit, H, by short-cutting (skipping repeated vertices)
        Return E, the optimal TSP path.
```

On average, Christophides' algorithm is 1.5•optimal with a O(n^4) running time. This is an easily explained, difficult to implement algorithm that yields acceptable results. Unfortunately, its many facets make this a fairly complex solution that only amounts to 1.5•optimal. Nearest Neighbor algorithm is clearly the more efficient solution.

Other Research:

Since we knew we had a 3 minute time constraint, we wanted to streamline our program as much as possible. When calculating the distance between cities, the result from the distance

formula is a floating-point number which then must be rounded to the nearest integer. Since this rounding is performed on each pairs of cities in our set we researched faster implementations of the rounding function. Simply truncating our floating-point number would have been fast, but not accurate since we want to round to the nearest integer. It turns out that many rounding functions in the C library involve expensive overhead function calls to handle overflow/do conversion, etc and that overhead adds up when performing it on every pair of cities in our set.
This source: (https://stackoverflow.com/questions/17035464/a-fast-method-to-round-a-double-to-a-32-bit-int-explained)
provided a faster method to round a double to an int by manipulating the bits of the double number to convert it to an int. Also, by using a #define macro we are using the preprocessor and inlining it directly into the code, so there isn't any function call overhead. Neat!

A discussion on why you selected the algorithm(s).

We ended up implementing the Nearest Neighbor approximation algorithm. The brute force algorithm was only practical for a small amount of cities and tsp_example_3 contained 15,111 cities, so we chose not to implement this algorithm. The DP Bellman-Held-Karp algorithm is a slight improvement on brute force, taking O(2^n•n^2) time but still would have been too slow given our city sets. Out of our three approximation algorithms our group researched, we decided upon the Nearest Neighbor algorithm because it was a nice compromise between time and optimal results. Although it doesn't give an exact solution like brute force or Bellman-Held-Karp, it will find an approximate 1.25 solution in a decent O(n^2) running time. In fact, the running time for the NN algorithm is fast enough to give a decent (1.25 times optimal) is under a few seconds, even for very large amount of cities. Because of this, it is possible to run the algorithm multiple times for each group of cities, and to take the best overall result as the final result.

The approximate solution to the TSP problem differs according to which city is chosen as the start, or "home" city. To get the best approximation, the algorithm is run one time for each city in the group. Although research has indicated that this NN algorithm is capable of a "worst possible tour" for certain "worst-case" city arrangements[1], we found that the results for each separate run only varied by as little as 3%. We took advantage of this last fact to allow for the 3 minute time limit when working with large groups of cities; we simply stopped the algorithm before completing a solution for each and every starting city. The amount that we lost in optimization was… zero. This can be seen by comparing the 3-minute and unlimited time results for test-input-7, below. This happened because the best solution for this particular group of cities occurred when the starting city was city 872, which was calculated at 135 seconds into the overall algorithm. Indeed, it is even possible that the best solution could be one of the first few that were run, in which case the solution for a very short run would match that of a full length run. This is chance however; the opposite could happen as well.

In this same vein of thought, this NN algorithm is capable of giving a decent approximation with one single run from a randomly chosen starting city. In this case, a decent

[1] G. Gutin, A. Yeo and A. Zverovich, Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. Discrete Applied Mathematics 117 (2002), 81-86.

approximate solution can be found in just a few seconds, even for a large group of cities, such as that in tsp_example_3.txt and test-input-7.txt. This is what makes this algorithm useful for practical purposes.

If we knew that our input city sets would always be small (less than 100 or so) we would implement one of the exact algorithms. Likewise, if we knew our input city set would always be large we would implement an approximation algorithm which doesn't find an exact solution but will find some solution quicker than an exact algorithm. Given that we did not know what our input city sets were for the competition, we chose a middle road with the Nearest Neighbor algorithm, sacrificing a bit of optimization but gaining a lot more efficiency. The asymptotic run time of the Nearest Neighbor algorithm is O(n^2), whereas the asymptotic run times for the Brute Force and the Bellman-Held-Karp algorithms are O(n!) and O(2^n•n^2), respectively. This makes the latter two exact algorithms completely impractical for even moderately sized groups of cities. And this is precisely why approximation algorithms are studied and used.

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

tsp_example_1

Time: 2.19981 ms

Tour Distance: 130,921 (1.21 approx)

tsp_example_2
Time: 90.6498 ms
Tour Distance: 3,008 (1.16 approx)

Tsp_example_3
Time: 20,043,800 ms = 20,043.8 s = 334.0633 min = 5.5677 hours
Tour Distance: 1,910,419 (1.21 approx)

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

test-input-1
Time (3 min/unlimited): 2.2 ms
Tour Distance: 5,911

test-input-2
Time (3 min/unlimited): 7.5 ms
Tour Distance: 8,011

test-input-3
Time (3 min/unlimited): 99 ms
Tour Distance: 14,826

test-input-4
Time (3 min/unlimited): 770 ms
Tour Distance: 19,711

test-input-5
Time (3 min/unlimited): 6009 ms = 6.009 s
Tour Distance: 27,128

test-input-6
Time (3 min/unlimited): 49,140 ms = 49.14 s
Tour Distance: 39,609

test-input-7
Time (3 min): 179,885 ms = 179.885 s = 2.99 min
Tour Distance: 61,376

test-input-7
Time (unlimited): 756,718 ms = 756.718 s = 12.6119 min
Tour Distance: 61,376