CS 325 Project 4: The Travelling Salesman Problem (TSP)

Group 12
Kyle Guthrie
Michael C. Stramel
Alex Miranda

December 2, 2016

Summary

Problem Statement

The purpose of the project was to research and implement algorithms for the Traveling Salesman Problem (TSP). We were provided 10 total problem instances, which consisted of numbered cities and their corresponding x,y coordinates. The goal was to devise agorithms that solve tours to visit each city exactly once and return to the originating city. Additionally, these tours should be found quickly (or efficiently). Our goal was to implement algorithms that, for all 10 problem instances, calculated tour lengths of no worse than 25% over the optimal tour lengths. Additionally, for a subset of the problem instances, we were to discover feasible solutions in 3 minutes or less. For three of the problem instances (example problem set), we were given the optimal tour lengths. For the other seven problem instances (competition problem set), we used [1] to approximate the optimal tour lengths.

Algorithms Researched/Implemented

Our research included Google searches regarding TSP algorithms, and readings of various texts such as [2] and [3] . See the bibliography at the conclusion of this report for a full list of references. Ultimately, we implemented four different algorithms in Python or C. We will call these algorithms: braindead tour, nearest neighbor, depth first search of the minimum spanning tree, and 2-opt. Details and pseudocode for each implementation can be found in the following pages.

Results

Complete results (runtimes and calculate tours) for each problem instance for each algorithm can be found in the last pages of this document. We found that certain algorithms were fast and that others provided answers under the 25% threshold, but it was ultimately our 2-opt algorithm that provided the best combination of the two. It is for this algorithm that we submitted competition results.

Algorithm: Braindead Tour

Algorithm Description

This algorithm is called braindead tour because it requires little to no thinking. It simply goes from city 1 to city 2 to ... to city n as ordered in the input file.

Algorithm Discussion

This algorithm was implemented early on just as a way to test file I/O and to understand the provided helper programs such as tsp-verifier. No real research was done for this algorithm because it wasn't considered a serious solution, but we are including it in this report for completeness. For this implementation we used Python 3.5. The algorithm calculates and sums the distances between each city in the order they exist in the input file (plus distance from the last city back to the first). It runs very quickly in O(n) time. The efficiency of the algorithm in terms of calculated tour vs optimal is entirely dependent on the ordering of the cities in the input file. For the three example cases provide we saw routes as good as 9% over optimal and as bad as 71 times optimal.

Algorithm Pseudo-code

```
braindeadTour()
    cities = []
    read date from file into cities[] // each city has id, x, and y attributes

    totalDist = 0
    prevCity = cities[disc[0]]
    for i = 1 to len(cities)
        eachCity = cities[i]
        addDist = dist(eachCity, prevCity)
        totalDist += addDist
        prevCity = eachCity
    addDist = dist(prevCity, cities[disc[0]])
    totalDist += addDist

    write totalDist to file
    write contents of disc[] to file // 1 item per line

dist(cityOne, cityTwo)
    dx = cityOne['x'] - cityTwo['x']
    dy = cityOne['y'] - cityTwo['y']
    dxSq = dx ^ 2
    dySq = dy ^ 2
    return ((dxSq + dySq) ^ 0.5)
```

Algorithm: Depth First Search of Minimum Spanning Tree

Algorithm Description

We learned about the minimum spanning tree heuristic from a discussion in [3] . The concept is to find the minimum spanning tree (MST). Then the tour order is set by the discovery order while performing a depth first search (DFS) on the minimum spanning tree.

Algorithm Discussion

We implemented this algorithm because it seemed relatively simple. For this implementation we used Python 3.5. At first we implemented Prim's algorithm to find the MST but we found it to be too slow. Therefore, we switched to using Kruskal's algorithm to find the MST which provided a better running time of O(n lg n). Finally, according to [3], this heuristic is typically 15% to 20 over optimal in practice. Unfortunately, while it was decently fast for smaller input sizes, it was inefficent for very large input sizes ($n \geq 5000$). Also, for the three example cases it didn't provide the expected results relative to optimal. Depending on the case the route found was 31% to 88% worse than optimal.

Algorithm Pseudo-code

```
DFSofMST()
    cities = []
    read data from file into cities[] // each city has id, x, and y attributes

    adjMatrix=[len(cities)][len(cities)]
    for i = 0 to len(cities) - 1
        for j = i + 1 to len(cities)
            ij = dist(cities[i][j])
            adjMatrix[i][j] = ij
            adjMatrix[i][j] = ji

    mst = kruskalsAlg(adjMatrix)
    adjList = mstToAdjList(adjMatrix, mst)
    disc = dfs(adjList)

    totalDist = 0
    prevCity = cities[disc[0]]
    for i = 1 to len(disc)
        eachCity = cities[disc[i]]
        addDist = dist(eachCity, prevCity)
        totalDist += addDist
        prevCity = eachCity
    addDist = dist(prevCity, cities[disc[0]])
```

```
    totalDist += addDist

    write totalDist to file
    write contents of disc[] to file // 1 item per line

dist(cityOne, cityTwo)
    dx = cityOne['x'] - cityTwo['x']
    dy = cityOne['y'] - cityTwo['y']
    dxSq = dx ^ 2
    dySq = dy ^ 2
    return ((dxSq + dySq) ^ 0.5)

dfs(adjList)
    vstd = []
    stack = []
    stack.append(0)

    while(len(stack) > 0)
        u = stack[len(stack) - 1]
        stack.remove(stack[len(stack) - 1])
        if not (u in vstd)
            vstd.append(u)
            auxStack = []
            for eachAdj in reverse_sorted(adjList[u].items()):
                v = eachAdj[0]
                if not (v in vstd):
                    auxStack.append(v)
            while len(auxStack) > 0
                v = auxStack[0]
                stack.append(v)
                auxStack.remove(v)

    return vstd

mstToAdjList(adjMatrix, mst)
    adjList = {}
    for i = 0 to len(adjMatrix)
        adjV = {}
        for j = 0 to len(mst)
            if i == mst[j]
                adjV[j] = adjMatrix[i][j]
        adjList[i] = adjV
    return adjList

popMin(prqu)
```

```
    minU = prqu[0]
    prqu.remove(prqu[0])
    return minU

decreaseKey(prqu, cost)
    for i = 0 to len(prqu)
        for j = 0 to len(prqu)
            if cost[prqu[i]] < cost[prqu[j]]
                prqu[i] = prqu[i] + prqu[j]
                prqu[j] = prqu[i] - prqu[j]
                prqu[i] = prqu[i] - prqu[j]

def kruskalsAlg(adjMatrix)
    edges = []
    for i = 0 to len(adjMatrix) - 1
        for j = i + 1 to len(adjMatrix)
            e = edge(i, j, adjMatrix[i][j])
            edges.append(e)
    edges.sort(key = lambda x: x.d)

    prev = [None for x in range(len(adjMatrix))]
    vstd = []
    for e in edges:
        if (not(e.v in vstd))
            vstd.append(e.v)
            prev[e.v] = e.u
    return prev
```

Algorithm: 2-Optimal Tour

Algorithm Description

The 2-opt algorithm is a simple local search algorithm that has application to solve the Traveling Salesman Problem. The main idea behind it is to take a route that crosses over itself and reorder that route so that it no longer overlaps. We first encountered K-optimal tours in [3] and learned more through it's Wikipedia article [4] . This is the only algorithm which we implemented in C. We chose C over Python for the speed improvement.

Algorithm Discussion

We decided on this algorithm because it was easy to implement conceptually, it's fairly fast, and it provides the most optimum tour lengths of any algorithm we implemented. Also the algorithm is relatively efficient to compute the path improvement for a given path swap which provides for a favorable runtime. The other algorithms we wrote up were constructive while 2-OPT is a local search heuristic meaning it improves on an exisiting path. Because 2-OPT is a local search heuristic it needs to be combined with an efficient constructive algorithm which we choose nearest neighbor. For the three example cases 2-opt found routes between 4% and 6% over optimal.

Algorithm Pseudo-code

```
TSP_2OPT_SEARCH(adj_matrix, tour, tour_length, file_name, num_pts):
    improved = true
    old_tour_length = tour_length

    while improved:
        improved = false
        exit_early = false

        for i = 1 to num_pts - 1 and not exit_early:
            for j = i + 1 to num_pts - 1 and not exit_early:
                old_tour_length = tour_length
                TSP_2OPT_SWAP_EFFICIENT(adj_matrix, tour, tour_length, num_pts, i, j)
                if tour_length < old_tour_length:
                    improved = true
                    exit_early = true

                    write the improved tour to file

TSP_2OPT_SWAP(new_tour, tour, num_pts, nodeA, nodeB):
    min_node = MIN(nodeA, nodeB)
    max_node = MAX(nodeA, nodeB)
    new_tour_idx = 0
```

```
    for i = 0 to min_node:
        new_tour[new_tour_idx] = tour[i]
        new_tour_idx++

    for i = max_node to min_node:
        new_tour[new_tour_idx] = tour[i]
        new_tour_idx++

    for i = max_node + 1 to num_pts:
        new_tour[new_tour_idx] = tour[i]
        new_tour_idx++

TSP_2OPT_SWAP_EFFICIENT(adj_matrix, tour, tour_length, num_pts, nodeA, nodeB):
    min_node = MIN(nodeA, nodeB)
    max_node = MAX(nodeA, nodeB)
    new_tour_idx = 0

    removed_path_length = 0
    added_path_length = 0

    new_tour[num_pts]

    if max_node + 1 < num_pts:
        removed_path_length = adj_matrix[tour[min_node - 1]][tour[min_node]] +
            adj_matrix[tour[max_node]][tour[max_node + 1]]
        added_path_length = adj_matrix[tour[min_node - 1]][tour[max_node]] +
            adj_matrix[tour[min_node]][tour[max_node + 1]]
    else:
        removed_path_length = adj_matrix[tour[min_node-1]][tour[min_node]] +
        adj_matrix[tour[max_node]][tour[0]];

        added_path_length = adj_matrix[tour[min_node-1]][tour[max_node]] +
        adj_matrix[tour[min_node]][tour[0]];

    path_delta = removed_path_length - added_path_length

    if path_delta > 0:
        for i = 0 to min_node - 1:
            new_tour[new_tour_idx] = tour[i]
            new_tour_idx++

        for i = max_node to min_node:
            new_tour[new_tour_idx] = tour[i]
            new_tour_idx++
```

```
for i = max_node + 1 to num_pts - 1:
    new_tour[new_tour_idx] = tour[i]
    new_tour_idx++

tour_length -= path_delta
```

Algorithm: Nearest Neighbor

Algorithm Description

This algorithm starts the salesman at a random city in the tour and repeatedly visits the nearest city until all have been visited. We implemented this algorithm in Python 3.5. We first encountered the nearest neighbor algorithm when researching general TSP algorithms at [5] . We learned more through it's specific Wikipedia article at [6] .

Algorithm Discussion

We chose nearest neighbor because it is essentially greedy which makes a good choice for being efficient and quick. We found that this algorithm satisfies the 1.25 requirement for most cases. Additionally, because it runs "fairly" fast for most input sizes, we were able to set it up to try each possible starting point. For larger input sizes we had the algorithm write results every time it found a newer/better route. This allowed us to find an acceptable route under 3 minutes (though there may have been better routes with unlimited time). For the three example cases 2-opt found routes between 15% and 22% over optimal. The algorithm runs in $O(n^2)$ time.

Algorithm Pseudo-code

```
Nearest_Neighbor():
    cities = []
    // each city object has id, x and y attributes
    read data from input file into the cities array

    adjMatrix=[len(cities)][len(cities)]
    for i = 0 to len(cities) - 1:
        for j = i + 1 to len(cities):
            ij = dist(cities[i][j])
            adjMatrix[i][j] = ij
            adjMatrix[i][j] = ji

    min_tour_dist = MAX_INT
    min_tour_order = []

    for i = 0 to len(cities) - 1:
        tour_cities = copy of cities array values w/o references to orig array
        tour_order = []

        tour_order.append(tour_cities[i].id)
        tour_cities.remove(tour_cities[i])
        tour_distance = 0
```

```
while tour_cities length > 0:
    cur_city = cities[tour_order[len(tour_order) - 1]]
    min_dist = MAX_INT
    min_city = -1

    for j = 0 to len(tour_cities) - 1:
        this_dist = adjMatrix[cur_city.id][tour_cities[j].id]

        if this_dist == -1:
            this_dist = cartesian dist between cur_city and tour_cities[j]
            adjMatrix[cur_city.id][tour_cities[j].id] = this_dist
            adjMatrix[tour_cities[j].id][cur_city.id] = this_dist

        if this_dist < min_dist:
            min_dist = this_dist
            min_city = tour_cities[j]

    tour_order.append(min_city.id)
    tour_cities.remove(min_city)
    tour_distance = tour_distance + min_dist
u = cities[tour_order[0].id]
v = cities[tour_order[len(tour_order) - 1]].id
this_dist = adjMatrix[u][v]
if this_dist == -1:
    this_dist = distance between cities[tour_order[0]]
                and cities[tour_order[len(tour_order) - 1]]
    adjMatrix[u][v] = this_dist
    adjMatrix[u][v] = this_dist

tour_distance = tour_distance + this_dist

if tour_distance < min_tour_dist:
    min_tour_dist = tour_distance
    min_tour_order = copy of tour_order's array by value

    write the output file to the path specified
```

Summary Stats: Example 1

```
input size:                76
optimal tour length:       108159

algorithm:                 braindead tour
tour length:               150781
len/opt len:               1.39
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0667

algorithm:                 depth first search of minimum spanning true (prim's)
tour length:               139725
len/opt len:               1.29
runtime(d:hh:mm:ss.ssss): 0:00:00:01.4073

algorithm:                 depth first search of minimum spanning true (kruskal's)
tour length:               141813
len/opt len:               1.31
runtime(d:hh:mm:ss.ssss): 0:00:00:00.1616

algorithm:                 nearest neighbor
tour length:               130921
len/opt len:               1.21
runtime(d:hh:mm:ss.ssss): 0:00:00:00.2948

algorithm:                 2-OPT with Nearest Neighbor initial tour
tour length:               109067
len/opt len:               1.008
runtime(d:hh:mm:ss.ssss): 0:00:00:0.0352
```

Summary Stats: Example 2

```
input size:                280
optimal tour length:       2579

algorithm:                 braindead tour
tour length:               2808
len/opt len:               1.09
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0671

algorithm:                 depth first search of minimum spanning true (prim's)
tour length:               3819
len/opt len:               1.48
runtime(d:hh:mm:ss.ssss): 0:00:01:27.5504

algorithm:                 depth first search of minimum spanning true (kruskal's)
tour length:               3890
len/opt len:               1.51
runtime(d:hh:mm:ss.ssss): 0:00:00:00.7930

algorithm:                 nearest neighbor
tour length:               2975
len/opt len:               1.15
runtime(d:hh:mm:ss.ssss): 0:00:00:04.4363

algorithm:                 2-OPT w/ Nearest Neighbor initial tour
tour length:               2654
len/opt len:               1.029
runtime(d:hh:mm:ss.ssss): 0:00:00:2.2387
```

Summary Stats: Example 3

```
input size:                15112
optimal tour length:       1573084

algorithm:                 braindead tour
tour length:               112310765
len/opt len:               71.40
runtime(d:hh:mm:ss.ssss): 0:00:00:00.2477

algorithm:                 depth first search of minimum spanning true (prim's)
tour length:               DNF
len/opt len:               DNF
runtime(d:hh:mm:ss.ssss): DNF

algorithm:                 depth first search of minimum spanning true (kruskal's)
tour length:               2955932
len/opt len:               1.88
runtime(d:hh:mm:ss.ssss): 0:11:46:52.8194

algorithm:                 nearest neighbor
tour length:               1917269
len/opt len:               1.22
runtime(d:hh:mm:ss.ssss): DNF

algorithm:                 2-OPT w/ Nearest Neighbor initial tour
tour length:               1675756
len/opt len:               1.065
runtime(d:hh:mm:ss.ssss): 0:06:39:16.6653
```

Summary Stats: Test/Competition 1

```
input size:                50
optimal tour length:       5333 (from Concorde)

algorithm:                 braindead tour
tour length:               27301
len/opt len:               5.12
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0636

algorithm:                 depth first search of minimum spanning true (prim's)
tour length:               DNF
len/opt len:               DNF
runtime(d:hh:mm:ss.ssss): DNF

algorithm:                 depth first search of minimum spanning true (kruskal's)
tour length:               8397
len/opt len:               1.57
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0546

algorithm:                 nearest neighbor
tour length:               5911
len/opt len:               1.11
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0808

algorithm:                 2-OPT w/ Nearest Neighbor initial tour
tour length:               5373
len/opt len:               1.0075
runtime(d:hh:mm:ss.ssss): 0:00:00:0.0265
```

Summary Stats: Test/Competition 2

```
input size:                100
optimal tour length:       7384 (from Concorde)

algorithm:                 braindead tour
tour length:               48053
len/opt len:               6.51
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0631

algorithm:                 depth first search of minimum spanning true (prim's)
tour length:               DNF
len/opt len:               DNF
runtime(d:hh:mm:ss.ssss): DNF

algorithm:                 depth first search of minimum spanning true (kruskal's)
tour length:               12681
len/opt len:               1.72
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0768

algorithm:                 nearest neighbor
tour length:               8011
len/opt len:               1.08
runtime(d:hh:mm:ss.ssss): 0:00:00:00.2705

algorithm:                 2-OPT w/ Nearest Neighbor initial tour
tour length:               7487
len/opt len:               1.01
runtime(d:hh:mm:ss.ssss): 0:00:00:0.1035
```

Summary Stats: Test/Competition 3

```
input size:                250
optimal tour length:       12067 (from Concorde)

algorithm:                 braindead tour
tour length:               123877
len/opt len:               10.27
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0639

algorithm:                 depth first search of minimum spanning true (prim's)
tour length:               DNF
len/opt len:               DNF
runtime(d:hh:mm:ss.ssss): DNF

algorithm:                 depth first search of minimum spanning true (kruskal's)
tour length:               21611
len/opt len:               1.79
runtime(d:hh:mm:ss.ssss): 0:00:00:00.3322

algorithm:                 nearest neighbor
tour length:               14826
len/opt len:               1.23
runtime(d:hh:mm:ss.ssss): 0:00:00:03.6776

algorithm:                 2-OPT w/ Nearest Neighbor initial tour
tour length:               12459
len/opt len:               1.03
runtime(d:hh:mm:ss.ssss): 0:00:00:4.0371
```

Summary Stats: Test/Competition 4

```
input size:                500
optimal tour length:       16720 (from Concorde)

algorithm:                 braindead tour
tour length:               247675
len/opt len:               14.81
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0667

algorithm:                 depth first search of minimum spanning true (prim's)
tour length:               DNF
len/opt len:               DNF
runtime(d:hh:mm:ss.ssss): DNF

algorithm:                 depth first search of minimum spanning true (kruskal's)
tour length:               31483
len/opt len:               1.88
runtime(d:hh:mm:ss.ssss): 0:00:00:01.8154

algorithm:                 nearest neighbor
tour length:               19711
len/opt len:               1.18
runtime(d:hh:mm:ss.ssss): 0:00:00:24.9158

algorithm:                 2-OPT w/ Nearest Neighbor initial tour
tour length:               17318
len/opt len:               1.04
runtime(d:hh:mm:ss.ssss): 0:00:00:49.9898
```

Summary Stats: Test/Competition 5

```
input size:                1000
optimal tour length:       22976 (from Concorde)

algorithm:                 braindead tour
tour length:               48053
len/opt len:               2.09
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0687

algorithm:                 depth first search of minimum spanning true (prim's)
tour length:               DNF
len/opt len:               DNF
runtime(d:hh:mm:ss.ssss): DNF

algorithm:                 depth first search of minimum spanning true (kruskal's)
tour length:               44272
len/opt len:               1.93
runtime(d:hh:mm:ss.ssss): 0:00:00:10.8341

algorithm:                 nearest neighbor
tour length:               27128
len/opt len:               1.18
runtime(d:hh:mm:ss.ssss): 3 minute limit hit

algorithm:                 2-OPT w/ Nearest Neighbor initial tour
tour length:               24085
len/opt len:               1.05
runtime(d:hh:mm:ss.ssss): 3 minute limit hit
```

Summary Stats: Test/Competition 6

```
input size:                2000
optimal tour length:       32448 (from Concorde)

algorithm:                 braindead tour
tour length:               1050395
len/opt len:               32.37
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0762

algorithm:                 depth first search of minimum spanning true (prim's)
tour length:               DNF
len/opt len:               DNF
runtime(d:hh:mm:ss.ssss): DNF

algorithm:                 depth first search of minimum spanning true (kruskal's)
tour length:               61594
len/opt len:               1.90
runtime(d:hh:mm:ss.ssss): 0:00:01:24.4163

algorithm:                 nearest neighbor
tour length:               39834
len/opt len:               1.23
runtime(d:hh:mm:ss.ssss): 3 minute limit hit

algorithm:                 2-OPT w/ Nearest Neighbor initial tour
tour length:               34405
len/opt len:               1.06
runtime(d:hh:mm:ss.ssss): 3 minute limit hit
```

Summary Stats: Test/Competition 7

```
input size:                5000
optimal tour length:       Unknown

algorithm:                 braindead tour
tour length:               2580008
len/opt len:               Unknown
runtime(d:hh:mm:ss.ssss): 0:00:00:00.0938

algorithm:                 depth first search of minimum spanning true (prim's)
tour length:               DNF
len/opt len:               DNF
runtime(d:hh:mm:ss.ssss): DNF

algorithm:                 depth first search of minimum spanning true (kruskal's)
tour length:               DNF
len/opt len:               DNF
runtime(d:hh:mm:ss.ssss): 3 minute limit hit (no results generated)

algorithm:                 nearest neighbor
tour length:               62110
len/opt len:               Unknown
runtime(d:hh:mm:ss.ssss): 3 minute limit hit

algorithm:                 2-OPT w/ Nearest Neighbor initial tour
tour length:               55149
len/opt len:               Unknown
runtime(d:hh:mm:ss.ssss): 3 minute limit hit
```

References

[1] W. Cook. (Nov. 2016). Concorde, [Online]. Available: http://www.math.uwaterloo.ca/tsp/concorde.html.

[2] T. H. Cormen, *Introduction to algorithms*. MIT Press, 2009.

[3] S. S. Skiena, *The Algorithm Design Manual*. Springer, 2008.

[4] *2-opt*. [Online]. Available: https://en.wikipedia.org/wiki/2-opt.

[5] *Travelling salesman problem*. [Online]. Available: https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[6] *Nearest neighbour algorithm*. [Online]. Available: https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm.