Ann Kostecki
Charles Ledbetter
Matthew Northey

Group 40 Final Project

Introduction

In this document we discuss three possible methods for solving the traveling salesman problem. We discuss the greedy algorithm approach, the linear programming approach, and the dynamic programming approach. We explain why we have chosen the greedy algorithm approach for our implementation. We also share the times and minimum distances that our chosen algorithm produces.

Three Methods for Solving the TSP

Ann Kostecki: Greedy Algorithm

The greedy algorithm is an algorithm that makes the locally optimal choice in hopes that it will lead to a global optimum.[1] In the traveling salesman problem, the salesman must visit every city, while returning to the first city. If the salesman consistently chooses the next city as the one nearest to his current city, this should lead to an optimal or near-optimal route. However, as has been seen with greedy algorithms, it is not a guarantee that the end result will be optimal and it still is possible to have a solution that is far from optimal. As well, it is possible that a greedy algorithm may not find a route, even if one exists.

While looking at the greedy algorithm, there are two similar algorithms that the salesman can follow: nearest-neighbor and repetitive nearest-neighbor. The nearest-neighbor algorithm follows as described above, always choosing the nearest city to his current location, typically choosing a random city as its starting city.[2] The repetitive nearest-neighbor, however, repeats the nearest-neighbor algorithm with a different starting city for each run until all cities have been a starting point. After it is run for each city, the minimum cost route is chosen. While the repetitive nearest-neighbor algorithm will more closely retrieve an optimal solution, the time it takes to run becomes quite large as n grows larger. With both nearest-neighbor and repetitive nearest-neighbor, the salesman is constructing a Hamiltonian circuit.[3] In the average case, both the nearest-neighbor and repetitive nearest-neighbor algorithms are efficient but not optimal., as it is possible that it will miss shorter routes.[3]

Pseudocode:

```
//city attributes
Class city(first number, second number, third number)
City_id = first number in line
X_coord = second number
Y_coord = third number
Visited = false

 //distance

Int distance(x1, y1, x2, y2)
        d = nearestint(√(x1 – x2)2 + (y1 – y2)2)
        return d

 //read file and store cities
Vector<city> allCities
For each line in file
        city temp(id, x, y)
        allCities.push_back(temp)

 //start with first city, calculate distances for n-1 cities, choose minimum distance city
int i = 0
total_distance = 0
numCitiesVisited = 1
Vector<city> visitedCities
visitedCities.push_back(allCities[i])
Int minDistance = 99999999

 while numCitiesVisited < allCities.length
        j = 0

minDistance = 9999999
        if allCities[i].visited == false || i == closestCity

while j < allCities.length
        if allCities[j].visited == false
                r = distance(allCities[i], allCities[j])
```

```
            if r < minDistance && >0
                minDistance = r
                closestCity = j
        j++

allCities[closestCity].visited = true
visitedCities.push_back(allCities[closestCity])
total_distance += minDistance
allCities[i].visited = true
i = closestCity
numCitiesVisited++

 //complete trip back to origin city
tripBack = distance(allCities[closestCity], allCities[0])
totalDistance =+ tripBack
```

Charles Ledbetter: Linear Programming

According to wikipedia, "Linear programming is a method to achieve the best outcome (such as maximum profit or lowest cost) in a mathematical model whose requirements are represented by linear relationships."[4] In the Traveling Salesman Problem as implemented in the assignment the mathematical model is a graph with weighted edges in which every vertex must be visited exactly once, and every edge no more than once. In this way the Traveling Salesman is like a shortest path problem when solved by simplex optimization linear programming.

The standard form for an optimization linear programming problem is[5]:

Maximize $\sum_{j=1}^{n} c_j x_j$

This form indicates that the TSP problem is solved by summing each cx where c is a distance value, and x is an edge constant variable. It the case that the start/end vertex is known the time complexity of the solution formula is O(c(n(n+m))) where c is the number of iterations needed to solve the problem n is the number of vertices, and m the number of edges. In the case that the starting/ending point is not known the time complexity is O(n(ci(n(n+m)))) for (ci…cn)because the solution must be found from each vertex. Note that in this case c varies for each execution of the formula on a new starting/ending vector[6].

There are actually several variations of linear programming methods. These include: primal simplex, which is the form indicated so far, dual simplex, which uses two linked equations to solve the two problems[5], ellipsoid interior point, which is the first worst-case polynomial-time algorithm ever found for linear programming, and lastly barrier function interior point, which is the most popular interior point method in use today[1]. Most of these variations are not suited to solve the TPS problem as a shortest path problem in a reasonable time, but barrier function interior point could be used as a variant[7].

Lastly I would like to mention that in order to create the linear programming equation to solve the TSP problem the first and last vector will need to be indicated as two separate vectors with all the same connections and edge weights that are not connected to one another. Also the constraints must indicate that every vector is reached, but only so once.

Ann Kostecki
Charles Ledbetter
Matthew Northey

Pseudocode[12] [13]:

//objective

$\min c_1x_1 + c_2x_2 + \ldots + c_nx_n$

...where each x denotes a directed edge between two vertices and c indicates its weight. Replace each c with non-negative integer before computing.

//constraints

Set the sum of all routes to and from each vertex equal to one to indicate that each city visits another city only once and is visited only once. For example, in a scenario with only three cities, the following would comprise this set of constraints:

$x_{0,1} + x_{0,2} = 1$
$x_{1,0} + x_{1,2} = 1$
$x_{2,0} + x_{2,1} = 1$
$x_{1,0} + x_{2,0} = 1$
$x_{0,1} + x_{2,1} = 1$
$x_{0,2} + x_{1,2} = 1$

...where $x_{a,b}$ denotes an edge x, directed from vertex a to vertex b.

In order to prevent the possibility of multiple subtours, add an additional set of constraints using extra variables: $u_i + u_j + nx_{ij} \leq n - 1$ for all cases, such that $2 \leq i \neq j \leq n$. In the above scenario with three cities, the set of constraints would be:

$u_1 - u_2 + 2x_{1,2} \leq 1$
$u_2 - u_1 + 2x_{2,1} \leq 1$

Matthew Northey: Dynamic Programming

Dynamic programming is, "a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions."[8] In order to configure dynamic programming to solve the travelling salesperson problem, one must find the shortest path from one point, A, to each vertex while visiting every other vertex. The minimum distance among these paths will be the optimal solution.[9]

This can be achieved by defining the set C(S, i) as the cost of the minimum cost path visiting each vertex in set S once, starting and ending at vertex i, then start with all subsets of size two and calculate C(S,i), increase to size three and calculate, and so on. The overall runtime of this solution is $O(n^2 * 2n)$.

Ann Kostecki
Charles Ledbetter
Matthew Northey

Pseudocode[10] [11]:

```
//create matrix
vector M[1 … n][ subset of S - {s1} ]

//create index
vector X[ ][ ]

//initialize matrix with null set values to distance from vertex to start point
for i=2 to n
        M[ i ][ ∅ ] = distance(i, 1)

//calculate shortest distance
for j=1 to n - 1
        for all subsets A within (S - {s1}) that contain j vertices
                for k where k ≠ 1 and si is not in A
                        M[ k ][ A ] = min(distance(k,m), M[ m ][ A - {sm}])
                        m = sm contained within A
                        X[ k ][ A ] = value of m that gave minimum
        M[ 1 ][ S - {s1}] = min(distance(1,m), M[ m ][ S - {s1, sm}]
```

Our Chosen Method to Solve TSP

A Description of the Greedy Algorithm

The approach we chose to use to solve the Traveling Salesman Problem (TSP) is the greedy algorithm approach. Specifically the greedy heuristic we used was to find the nearest neighbor on each iteration and chose it as the next city. In this algorithm, the salesman starts at a city and continuously chooses the next nearest city (minimum distance) to travel to. When he has reached the last unvisited city, the salesman will return to the city where he started.

When we implemented the algorithm, we began with a CityGraph object that held the attributes for each city, including the x-coordinate, y-coordinate and if the city has been visited yet. It also held the sum of distances traveled between cities, which is initialized to 0. The coordinates and sum are later used to calculate distances between each city and keep the running sum. The visited attribute determines if the city has been traveled to yet or not, and, at the end of the tour, provides an ordered list of cities in order of visit. A city that has already been visited will not be visited again. This is insured by changing the name value of that city to -1 in the list of unvisited cities.

The actual implementation of the algorithm consists of three nested loops. An outer loop that depending on the size of n either loop through the algorithm n times, 100 times, 10, times, or 1 time. The point of this loop is to pick a new starting point for the calculation of the solution and compare to see which one

gives the best solution. With large enough values of n it becomes necessary to bound the iterations of the loop to keep the algorithm within reasonable time. The reasoning behind this is that for smaller values of n meeting the 1.25 ratio is a more precise procedure so more (or all) starting points should be checked. But with larger values of n it is more likely that any given starting point will provide the 1.25 ratio, so to reduce the needed computational time it is alright to simply try the first 100, 10, or 1 city. The outer loop also duplicates the CityGraph so that it can be modified without corrupting the original input data once it enter the middle and innermost loop. The unvisited list of cities within the copied CityGraph object are used to implement the nearest neighbor heuristic. The first city, and last distance (from last city to first city) are set and calculated by the outer loop. All other cities and distances are done within the two nested inner loops.

To calculate the distance, we used nearestint($\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$), where x and y referred to the coordinates of the two cities. On each iteration of the innermost loop of the algorithm every unvisited city's distance from the current city is calculated using this method. This information and the name of the city is stored within a City object (not to be confused with the CityGraph object). These City objects are each pushed to a vector and then sorted by the middle loop after the innermost loop has finished in order to find which of the cities is closest to the current city. The sort use is the built in C++ sort which is a merge sort algorithm. The outer loop then compares the summed distances from the current CityGraph to a solution CityGraph. If the current CityGraph has a better solution than the old solution CityGraph the old one is replaced. When all loops have ended the solution CityGraph is used to replace the original input CityGraph. The members of this object are than used to send the solution to file.

Justifications for using the Greedy Algorithm

Although there are a number of algorithms that can be used to solve TSP, as demonstrated by the earlier part of this document, we decided to implement the nearest neighbor greedy method. The primary reason for this choice was that the greedy method is the most straightforward approach to heuristicly solving TSP, which meant we could spend the time to develop code that was as clean as possible, with a reduced occurrence of bugs. This not only helped us understand the problem better, but also helped us avoid unintended inefficiencies that may have resulted from implementing more complicated code. Lastly, although the simplex linear programming solution seemed elegant, at larger values of n it approaches exponential time.

Pseudocode

```
//CityGraph object
CityGraph{
        Vector unvisited[]
        Vector visited[]
        Vector x[]
        Vector y[]
        sum
}
```

```
//City object
City{
        name
        distance  //holds distance from current city
}

//solveTSP algorithm : input a loaded CityGraph object, and the number of cities
solveTSP(cityG, n)
        intTemp = 0
        tempCity = empty City
        Vector cities[]
        tempG = empty CityGraph
        tempG.sum = 0
        solutionG = empty CityGraph
        solutionG.sum = INT_MAX
        k = 0
        bound = 0

        IF n > 4001
                Bound = 1
        ENDIF
        ELSEIF n > 2001
                Bound = 10
        ENDELSEIF
        ELSEIF n > 1001
                Bound = 100
        ENDELSEIF
        ELSE
                Bound = n
        ENDELSE

        FOR i = 0 to n
                tempG = cityG
                tempG.sum = 0
                tempG.visited.push_back(tempG.unvisited[i])
                tempG.unvisited[i] = -1
                k = i
```

Ann Kostecki
Charles Ledbetter
Matthew Northey

```
        WHILE tempG.visited not equal to n
            FOR j = 0 to n
                IF j not equal to i AND tempG.unvisited[i] not equal to -1
                    intTemp = (√(tempG.x[j] – tempG.x[k])^2 + (tempG.y[j] – ...
                                                                ...tempG.y[k])^2)
                    tempCity.name = j
                    tempCity.distance = intTemp
                    cities.push_back(tempCity)
                ENDIF
            ENDFOR

            SORT(cities)
            tempG.visited.push_back(tempG.unvisited[cities[0].name])
            tempG.unvisited[cities[0].name] = -1
            tempG.sum += cities[0].distance
            k = cities[0].name
            CLEAR(cities)
        ENDWHILE

        tempG.sum += (√(tempG.x[tempG.visited[0]] – tempG.x[tempG.visited[last]])^2 + ...
                            ...(tempG.y[tempG.visited[0]] – tempG.y[tempG.visited[last]])^2)
        IF tempG.sum < solutionS.sum
        solutionG = tempG
        ENDIF
    ENDFOR
    cityG = solutionG
END
```

Ann Kostecki
Charles Ledbetter
Matthew Northey

Three Sample Tour Results

	Solution Distance	Time (in seconds)
tsp_example_1.txt	130921	0.055 seconds
tsp_example_2.txt	3007	2.08 seconds
tsp_example_3.txt	1935667	43.691 seconds

Competition Tour Results

	Solution Distance	Time (in seconds)
test-input-1.txt	5793	0.021 seconds
test-input-2.txt	7965	0.128 seconds
test-input-3.txt	14848	2.145 seconds
test-input-4.txt	19564	17.736 seconds
test-input-5.txt	27388	156.136 seconds
test-input-6.txt	39154	63.47 seconds
test-input-7.txt	62676	42.478 seconds

Ann Kostecki
Charles Ledbetter
Matthew Northey

References

1 Greedy algorithm. (2018, March 4). Retrieved March 9, 2018, from https://en.wikipedia.org/wiki/Greedy_algorithm

2 Nearest neighbour algorithm. (2018, January 25). Retrieved March 9, 2018, from https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

3 Martin, J. (2011, December 18). The traveling salesman problem. Retrieved March 9, 2018, from http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter6-part4.pdf

4 Linear programming. (2018, February 11). Retrieved March 9, 2018, from https://en.wikipedia.org/wiki/Linear_programming#Interior_point

5 Linear programming methods. (2007, March 31). Retrieved March 9, 2018, from http://mars.wiwi.hu-berlin.de/mediawiki/teachwiki/index.php/Linear_Programming_Methods

6 Schutfort, Julianne. (2018). Introduction to Linear Programming. [PDF]. Retrieved from https://oregonstate.instructure.com/courses/1662148/pages/lp-lecture-notes-and-videos?module_item_id=17799586

7 Diaby, M., Karwan, M., & Sun, L. (2016, October 4). Small-Order-Polynomial-Sized Linear Program for Solving the Traveling Salesman Problem. Retrieved March 9, 2018, from https://arxiv.org/pdf/1610.00353.pdf

8 Dynamic programming. (2018, February 26). Retrieved March 9, 2018, from https://en.wikipedia.org/wiki/Dynamic_programming

9 Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming). (n.d.). Retrieved March 9, 2018, from https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

10 travelling salesman problem. (2015, May 21). Retrieved March 13, 2018 https://www.codeproject.com/Questions/993642/travelling-salesman-problem-using-dynamic-programm

11 Held-Karp Algorithm. Retrieved March 13, 2018, from https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm

12 Travelling Salesman Problem. Retrieved March 13, 2018, from https://en.wikipedia.org/wiki/Travelling_salesman_problem#Integer_linear_programming_formulation

13 How to Formulate Traveling Salesman Problem. Retrieved March 13, 2018, from https://cs.stackexchange.com/questions/50783/how-to-formulate-traveling-salesman-problem-tsp-as-integer-linear-program-ilp