Project Group 48 Report

Aaron Berns
Eric Bridgeford
Ryan Murray

Introduction

For this project, all members decided to research as well as implement different solutions to the Travelling Salesperson Problem. We felt that by implementing three solutions rather that one, we would more thoroughly understand the challenges of the Traveling Salesperson Problem. An added benefit to implementing three different algorithms is that it greatly increased our chances of finding the most optimal solution we could. Given three algorithms to choose from, we could better analyze the efficiency of the algorithms by compare and contrast. The three algorithms that we chose to research and implement were Christophides Approximation algorithm, Genetic Algorithms and 2-opt algorithm. Descriptions of our research can be found below.

Research

Christofides Approximation Algorithm (Aaron Berns)

Published in 1976 by Nicos Christofides, this algorithm produces the best proven approximation ratio for general traveling salesman problems at 3/2 of the optimal length. The algorithm begins by creating a minimum spanning tree of a complete graph. It finds the vertices of the MST that have odd degree and matches them with each other based on the minimum weight of the connecting edges. The edges from the MST are combined with the edges found in the minimum weight perfect matching sequence to create a multigraph where all vertices have even degree. A Eulerian circuit is formed in this multigraph and this circuit is then turned into a Hamiltonian circuit by removing trips to repeated vertices, also known as shortcutting. The resulting path is the route the salesman should take and is at most 1.5 times as long as the optimal route. This algorithm was chosen by the group and its implementation is detailed towards the end of the document.

Pseudocode

```
// form a complete graph, G, from given coordinates
for each city i in input file
        for each city j = i+1
```

```
                edge = round(sqrt((i.x - j.x)^2 + (i.y - j.y)^2))
                add edge to i's edge list
                add edge to j's edge list

// create a MST of the graph using Prim's method
// based on lecture pseudocode
MSTPrim(G)
        Q = G

        // initialize Q
        for each vertex i in Q
                Q[i].key = maxInt
        Q[0].key = 0
        Q[0].p = 0

        while Q.size() != 0
                u = extractMin(Q)
                for each vertex v in u's edge list
                        if v is in Q and the edge weight of u to v < u.key
                                v.p = u
                                v.key = edge weight of u to v

// find the vertices with odd degree
FindOddDeg(G)

        // add first edge given eade
        for each vertex i in G
                i.degree = 1

        // add edges of children to each parent
        for each vertex i in G
                G[G[i].p].degree++

        // remove the edges from vertex 0 being a parent of itself
        G[0].degree -=2

// perform minimum weight perfect matching on vertices with odd degree
Match(G)

        // create array of vertices with odd degree, odds[]
        for each vertex i in G
                if i.degree is odd
                        add i to odds
```

```
        // match each vertex with another with least edge weight
                while odds.size() != 0
                        i = odds[0]
                        minMatch = maxInt
                        for each vertex j in i's edge list
                                if j is in odds and the weight of i to j < minMatch
                                        possMatch = j
                                        minMatch = weight of i to j

                        set i's match to possMatch
                        set i's matching edge to minMatch
                        set possMatch's match to i
                        set possMatch's matching edge to minMatch
                        remove i and possMatch from odds

// combine edges of MST with edges from matching to form multigraph
MultiGraph(G)

        // delete all edges in G
        for each vertex i in G
                delete all edges in i's edge list

        // add edges from MST stored in each vertex's p and key variables
                for each vertex i in G
                        add edge formed from i.p and i.key to i's edge list
                        add edge to p's edge list

        // add edges from matching
                for each vertex i in G
                        if i has a matching vertex
                                add edge with match and edge weight to i's edge list

// form a Eulerian circuit in multigraph from source s, return circuit
Euler(G, s)

        eStack[]
        eCircuit[]
        vCurrent = s    // current vertex being examined

        while vCurrent's edge list is not empty or eStack is not empty

                // vCurrent has neighbors
```

```
            if vCurrent's edge list is not empty
                add vCurrent to eStack
                vNext = last vertex in vCurrent's edge list
                remove last edge from vCurrent's edge list

                // remove edge from vNext's edge list
                find vCurrent in vNext's edge list
                remove edge

                vCurrent = vNext
            // vCurrent has no neighbors
            else
                add vCurrent to eCircuit
                set vCurrent to top of eStack
                remove top of eStack


// remove repeated vertices to form Hamiltonian circuit
Shortcut(eCircuit)

    tsp[]

    // add vertices in circuit to new circuit, ignoring vertices already in new circuit
    for each vertex i in eCircuit
        if i is not in tsp
            add i to tsp
```

Sources
https://en.wikipedia.org/wiki/Christofides_algorithm
http://www.graph-magics.com/articles/euler.php
https://stackoverflow.com/questions/8353554/how-to-implement-the-shortcutting-step-in-the-christofides-algorithm

Genetic Algorithms (Ryan Murray)

Terminology

Basic terminology for genetic algorithms is similar to the terminology used for genetics.

Gene: A variable in a solution.
Allele: The value of a gene.
Chromosome: A solution made up of genes.
Population: All solutions/chromosomes.

Algorithm

There are several steps in the GA algorithm.

1. Population initialization: The population of chromosomes (i.e. candidate solutions) is randomly generated. It is important to randomize because we want to avoid getting stuck in local optima. By creating a wide search space with randomized solutions, we minimize the risk of getting stuck in a local optimum. While this means the initial chromosomes are poor fits for an optimum solution, it is better than trying to generate good solutions from the start since our method of generating “good” solutions could end up creating many similar solutions which would all get stuck in the same local optimum.

After this step we begin looping through steps 2-4.

2. Fitness calculation: Each chromosome’s fitness is calculated using a function that generates a fitness score for each chromosome. This score is used in subsequent steps.

3. Parent selection: Two chromosomes are selected to survive and produce offspring.

Selection can be made in several different ways. Three popular methods are “roulette wheel” selection, tournament selection, and rank selection. In the roulette wheel method each chromosome is given a probability proportionate to its fitness of being selected as a parent. In tournament selection several chromosomes are chosen at random and the fittest chromosome of the sample is selected to be a parent. In rank selection each chromosome is ranked according to fitness in the range 1..N where N is the population size. The rankings are then used as the probability (ranking / population size) that a chromosome will be selected. While this means that the most fit chromosome has the highest probability of being selected, it is closer to random selection since the difference in probabilities between fit and less fit chromosomes are relatively small.

4. Crossover: This step may or may not occur, and is selected to occur based on a chosen probability, although the probability is usually very high. A random segment of two chromosomes is selected to swap between the chose. The same length and position of each chromosome will be exchanged with the other. For example, if we have the following chromosomes:

ATGCATGGC
GTCCCATTA

we might randomly select a length of 5 to crossover:

ATGCA | TGGC

GTCCC | ATTA

Which would give us:

GTCCCTGGC
ATGCAATTA

5. Mutation: This step actually has a very low chance of occurring. If it does happen, however, a random allele is altered. In typical genetic algorithm implementations this means flipping a single bit.

6. Replacement: A new population is constructed. This can be achieved in several ways. In “steady state” populations, only a couple of chromosomes are deleted and replaced at random. In “generational” populations all chromosomes except those selected to survive are deleted and replaced. In an elitist approach, the most fit chromosomes are always selected to reproduce, thus ensuring that the best solution found thus far is not lost between generations which can happen in steady state and generational approaches.

Replacement is achieved by continually selecting parents according to one of the above selection schemes and adding the resultant child to the new population until the size of the original population has been reached. Steps 2 through 6 are then repeated in a loop until the termination condition is reached which could be decided based on a sufficiently acceptable solution having been reached, a maximum number of iterations having been reached, or little change between successive generations.

6. Termination: The algorithm is terminated and the best solution found is returned.

Use in TSP

Normally problems in genetic algorithms are encoded using binary values and individual bits are manipulated. However, this strategy does not work in TSP because changing individual bits would cause the city labels to change to unpredictable values. Furthermore, we cannot have a solution where a city occurs more than once or does not occur at all, meaning values must be exchanged, not altered. As such, with the TSP problem we represent the cities using their labels and move the labels around in the chromosome. Mutations are achieved by swapping city positions. Crossovers are achieved in a similar manner, using the “Partially-Mapped Crossover” (PMX) method wherein cities are swapped with the city labelled in the same position on the other parental chromosome. For example, if we have two parents with a crossover point identified by “ | “

12 | 3456
63 | 4251

Then in the first chromosome 1 is swapped with 6:

62 | 3451

and then 2 is swapped with 3 for the result:

632451

The process is then repeated with the second chromosome using a copy of the original first parent.

The fitness test function in TSP is simple; we just compute the sum of the distances between cities for each chromosome. The lower the score for a chromosome, the more fit it is.

Pseudocode

Construct an initial population of random chromosomes (paths)
While a set amount of iterations is not reached:
Determine the fitness of every chromosome using the sum of the distance function between genes
While the new population's size is less than the old population:
Select two parents at random using binary tournament selection:
Select two chromosomes at random and return the one with higher fitness / shorter distance
Crossover a random segment of genes from each parent using PMX with probability $P_{crossover}$*:*
Randomly select a point from which to swap genes onward through the chromosome
for i := startOfCrossover to end of chromosome:
If parent1[i] != parent2[i]:
j = location of parent2[i] in parent1
swap parent1[i] and parent1[j]
Repeat for parent2
Mutate a gene at random in each child chromosome with probability $P_{mutation}$*:*
i := random(0, length of chromosome)
j := random(0, length of chromsome) and j != i
swap(chromosome, i, j)
Add the two resulting children to the new population
Return the best solution found

WORKS CITED

https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_fundamentals.htm
https://en.wikipedia.org/wiki/Genetic_algorithm#Optimization_problems
https://courses.cs.washington.edu/courses/cse473/06sp/GeneticAlgDemo/gaintro.html
http://www.sciencedirect.com/science/article/pii/S0377042705000774
McCall, John. Genetic algorithms for modeling and optimization. *Journal of Computational and Applied Mathematics*, 2005. 184, 205-222.

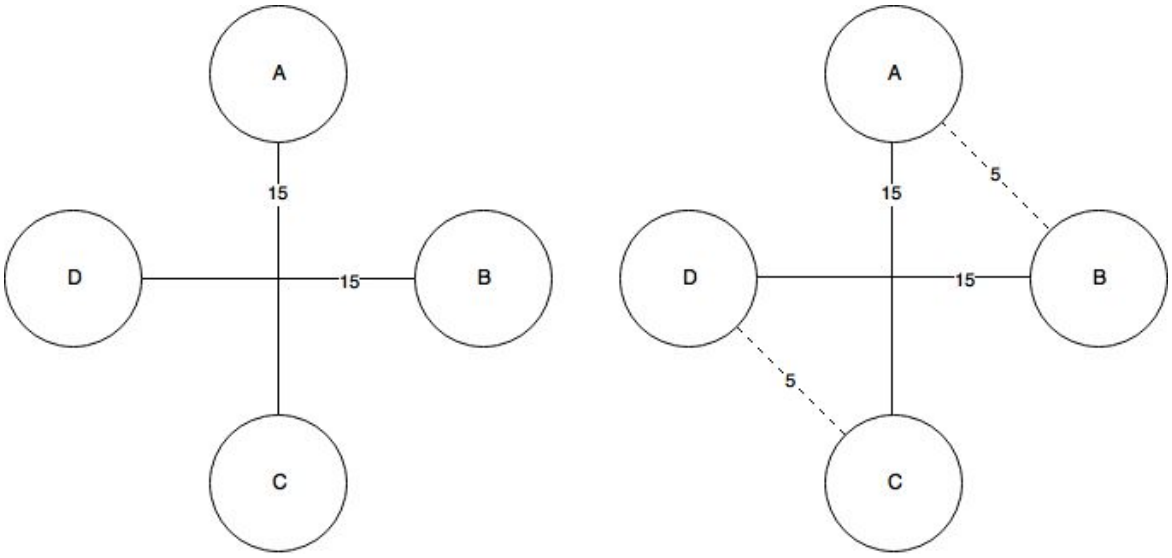
http://user.ceng.metu.edu.tr/~ucoluk/research/publications/tspnew.pdf
Ucoluk, Gokturk. Genetic algorithm solution of the TSP avoiding special crossover and mutation. *Intelligent Automation and Soft Computing*, 2002. 3, 265-272.

2-OPT Algorithm (Erik Bridgeford)

The 2-opt algorithm is a local search algorithm originally proposed in 1958 to solve the traveling salesperson problem. The 2-opt algorithm approaches the larger problem of the traveling salesperson by solving small problems iteratively first. This solution will utilize an algorithm to find the nearest neighbor to each node in order to find a tour for the traveling salesman that visits each node once. once we have a tour that visits all nodes, the 2-opt algorithm will compare the length of two crossing edges and swap them in order to “uncross” the paths so that we are left with a direct path between nodes. If these new uncrossed edges are more efficient than the previous ones, the old edges are removed and replaced by the new edges and added to the final optimized tour that will be used in the final solution. In essence, uncrossing any crossed paths.

For example, in Figure 1 below we have edges connecting A to C and B to D. These edges both have a length of 15. Next, the 2-opt algorithm will swap the edges so that A connects to B and C connects to D. These new proposed edges are represented by the dashed lines on the graph. We can see that they are both of length 5. Once the edges are swapped, the algorithm check if this new solution is more efficient that the previous edges. If so, it deletes the previous edges and replaces them with the new more efficient edges. In this example we see that the original solution was a total length of 30 and the new solution with swapped edges in 10. These steps occur iteratively until an optimal solution is found.

Fig 1.



Pseudocode:

twoOPT(array, tour)

```
    \\set minchange to -1 for while loop check
    minChange = -1

    While minChange < 0
    //swap is false until we find a more efficient set of edges
    Swap = false

    for i to length of tour
        for j to length of tour
            //subtracting the distances gives us the difference in distance from
```

```
                    //two nodes using original edges vs the counter solution edges
                    change = distance(from i to j) + distance( from i+1 to j+1) -
                    distance( from i to i+1) + distance (from j to j+1)
                If change < minChange
                //minChange gets value of change because change holds shortest route
                //between the nodes
                minChange = change
                minI = i
                minJ = j
                swap = true

        If swap == true
                Perform swap function

        return the now optimal tour
```

Best results for 2-opt test cases:

Tsp_example_1.txt = length 121671 in 4.18 seconds
Tsp_example_2.txt = length 2801 in 199.2 seconds
Tsp_example_3.txt = Inconclusive

sources :
https://ocw.mit.edu/courses/sloan-school-of-management/15-053-optimization-methods-in-management-science-spring-2013/lecture-notes/MIT15_053S13_lec17.pdf
https://en.wikipedia.org/wiki/2-opt
http://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf

Group Selection:

With all of our implementations complete, we began the process of determining our choice for the algorithm that we feel is best. To determine this, all group members ran the three test cases to gauge the speed and optimality of their algorithms. For the three test cases, there was no time limit for completion. However, the best algorithm of the three implemented would be used in the race, therefore time was a key factor in our decision of which algorithm was best.

Erik researched and implemented the 2-opt algorithm. The results of the test cases varied greatly. For the first two test cases, the 2-opt algorithm came within the bounds of success and were relatively optimal. Test case one came in with a length of 121671 in 4.18 seconds. Test case two came in with a length 2801 in 199.2 seconds. The third test case however has yet to be run completely, with multiple attempts lasting over twelve or more hours. These times compared to the times recorded for the Christofides algorithm, along with the potential for an unfinished run when dealing with very large data sets were key factors in deciding to go with Aaron's implementation of Christofides algorithm rather than the 2-opt implementation.

Ryan researched and implemented a genetic algorithm. The results of the test cases varied greatly depending on the settings used for the algorithm and the test. For example, results for tsp_example_1.txt ranged between 150,000 and 500,000 depending on the combination of population size, mutation rate, crossover rate, iterations, and selection strategy used. Unfortunately the algorithm could not be tweaked enough to deliver a solution close to optimal in a reasonable time in the cases tested, apparently due to premature convergence. As such, Aaron's implementation of the Christofides algorithm was chosen as the genetic algorithm as implemented could not deliver strong, fast results.

Written Explanation of Chosen Algorithm

The Christofides algorithm was implemented using C++. It used two structs to hold the data necessary for the implementation. The first struct, edge, was used to build an edge list and had variables for a vertex number and the distance to that vertex. The second struct, city, held each vertex's information and contained a vector of edge structs along with variables for the city number, coordinates, a key and parent used by the minimum spanning tree function, the degree of the vertex after the MST and finally the matching vertex and distance determined by the minimum weight perfect edge matching function.

The main function of the implementation prompts the user to enter the file name in which the city data is stored, opens that file for reading and another file with the same name with the addition of ".tour" for writing the results. Main reads each line of the input file, storing the city number and coordinates in a city struct before pushing each new city into a vector of city structs, called cGraph. Because C++ does not play nice with spaces at the end of a file, a possible duplicate of the final city is popped if it exists. Main then uses the calculates the distances between all of the vertices, storing each each between two vertices in each of the vertices' edge list, creating an undirected complete graph.

MSTPrim, a function that implements Prim's algorithm for finding a minimum spanning tree is then called and is passed cGraph by reference. The implementation is based on the pseudocode from the lectures. Inside MSTPrim, a vector of ints is formed to serve as the queue. To initialize the date, each city's key is set to the maximum integer value, each boolean variable that keeps track of what vertices are in the queue is set to true and each vertex number is pushed into the queue in the order in which they are stored in the graph. The key and parent of the first vertex is set to 0, making it the source. While the queue is not empty, a for loop compares all of the key values of the vertices in the queue to find the minimum. The min key value and the index of its vertex is saved. The edge list of the vertex at the saved index is scanned via a for loop, which checks to see if each edge is in the queue by checking the boolean value of the vertex and also checks to see if the edge value is less than the vertex's key. If both conditions are true, the edge is given the vertex at the saved minimum index as a parent and the edge key is updated with the edge distance. This relaxes the edges of every vertex when it is taken from the queue. After the edges are relaxed the minimum vertex is popped from the queue and its inQueue bool value is set to false so that it is not visited during future edge relaxing loops.

The next function to be called is findOddDegree, which totals all of the edges of each vertex. cGraph, which now has the MST data stored, is passed by reference. The first of two for loops initializes each vertices' degree variable to 1 as each has at least one edge connecting it to another vertex as it is a connected graph. The second for loop loops through each parent variable and uses the value stored there to access the parent vertex and increments the parent vertex's degree count by one as the child vertex adds an edge to its parent. The fact that the first vertex is set to the parent of itself is corrected by decrementing its degree variable by two.

Once we have an MST with some vertices of odd degree, the next step is to match those vertices with each other so that every vertex has even degree, a prerequisite for the upcoming Euler tour. This done by the match function, which is passed cGraph by reference. A vector of ints is created to store the number value of the vertices of odd degree. Each vertex in the graph is examined and if it has odd degree, it is assigned -1 to its match variable and pushed into the vector. Vertices of even degree are assigned -2. Initially the implementation returned approximations slightly higher than the accepted 1.25 value. To address this, the vertices with odd degree were sorted by the maximum matching edge weight in descending order. This assured that the largest edge values were not included as they might have been if all other edge options were taken when it came time for a vertex containing them in their edge list to be matched. This sorting was done using a nested for loop which scanned every edge in every edge list of the

vertices with odd degree that also had odd degree to find the maximum matching edge weight. The vertex and that distance were then the pushed pushed as an edge struct onto another vector of edges. Another nested looped examined the edge weights in the vector of edges and pushed vertex number of every odd vertex back into the cleared vector of ints with the vertices in order by descending max matching edge weights. Next, a nested for loop scanned each edge in the edge list of each vertex in the odds vector. Each edge in the list was checked for a -1 value in its match variable, if it contained a -1, which indicated that it was odd and unmatched, it's weight was compared to all other edge weights meeting the same condition until a minimum was found. An additional condition for the match was that the neither edge was already connected to the other in a parent-child relationship. The vertex and edge were then matched with the edge weight matching vertex number stored in each vertices' struct.

Next a multigraph is formed by the function multiGraph, which is passed cGraph by value and returns a vector of cities with only the edges contained in the MST and the matching process. To do this, each edge list in the graph was cleared. The MST edges were added to the edge lists of the parent and child via a for loop and the matching edges were added with another for loop which visited every vertex and if odd, ie didn't have a match value of -2, added the matching vertex and edge to its edge list. The new, leaner graph is returned and saved as eGraph.

Next an Eulerian circuit or Euler tour is performed by the eulerTour function, which is passed eGraph by reference as well as the source vertex, which always had a value of 0 for this application. It returns a vector of ints storing the vertex numbers that create the path of the circuit/tour. To do this, each vertex, vCurrent, starting with the source, has its edge list scanned. If it still has edges in its list, it is pushed onto a stack and the last edge in its edge list is removed from the edge list and becomes the next vertex of interest, vNext. vNext's edge list is examined and vCurrent is removed from it. vNext becomes vCurrent and the process is repeated as long as the current vertex has edges in its edge list (neighbors) or the stack is not empty. As vCurrent is examined, if it has an empty edge list, it is pushed on the circuit and the vertex at the top of the stack becomes vCurrent. After all edge lists are clear and all vertices have been examined, the resulting circuit/tour is returned.

Next, the Euler circuit is turned into a Hamiltonian circuit by removing all of the edges which connect vertices to vertices that have already been visited. This is done by the function shortcut, which is passed the Euler circuit and returns the Hamiltonian circuit, which is the traveling salesman route. A for loop traverses the elements of the Euler circuit, adding each to a hashmap with a value of 'true' and pushing each to the

Hamiltonian circuit vector. If a visited vertex comes up again, the map identifies it and it is ignored. Last, the source vertex is pushed on the end of the circuit, completing it before it is returned.

Now that the approximate route has been determined, the last step is to sum the edges in the circuit to get the total distance traveled. This is done by the function calcLength, which is passed the complete graph, cGraph by reference and the route vector. A nested for loop takes each vertex from the route in turn and scans its edge list for the edge weight of the next vertex in the route, adding the weight to the total distance. The last edge is added by finding the weight of the edge from the last vertex to the first. The distance is returned.

Main then writes the total distance and each vertex in the route to the output file and closes both it and the input file.

Example Tours

Example Number	Time	Distance
1	1.98 milliseconds	117603
2	12.3 milliseconds	3160
3	30.98 seconds	1900215

Competition Tours

Test Number	Time	Distance
1	1.3 milliseconds	6143
2	2.6 milliseconds	9847
3	10 milliseconds	15262
4	40 milliseconds	20498
5	170 milliseconds	28162
6	720 milliseconds	39833
7	4.6 seconds	62586