TSP Project

Everett Williams
CS Student, Oregon State University
1500 SW Jefferson St.
Corvallis, OR 97331
williaev@oregonstate.edu

Julian Weisbord
CS Student, Oregon State University
1500 SW Jefferson St.
Corvallis, OR 97331
weisborj@oregonstate.edu

Timothy Fye
CS Student, Oregon State University
1500 SW Jefferson St.
Corvallis, OR 97331
fyet@oregonstate.edu

ABSTRACT

This paper summarizes group sixteen's research on three algorithms purposed for solving the TSP problem. A brief description is provided with each algorithm, along with pseudocode detailing feasible implementations. The paper closes by disclosing the group's analysis of researched algorithms, the implementation we selected, and the determinants which lead us to our decisions. All references are cited in section 6 below.

Keywords

Traveling Salesman Problem (TSP); 2-OPT Algorithm, Christofides Algorithm, Ant Colony Optimization Algorithm

INTRODUCTION

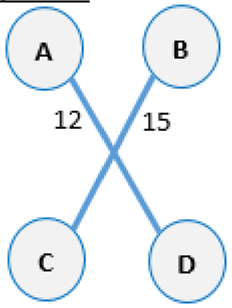
The Traveling Salesman Problem calls for a solution to find the shortest path given a set of vertices in a graph. The problem stipulates that from a starting point of a given city, a salesman should travel city to city until he visits every single city, then returns to the point of origin in such a way that he traveled the shortest distance. Our group focused on three specific algorithms as possible solutions to the TSP. The first algorithm we researched was the 2-OPT algorithm. This algorithm approaches a solution by optimizing a single route via restructuring areas where the route crosses over itself. As the program progresses, each route is evaluated and run through the swapping routine. The second algorithm our team investigated was the Christofide's algorithm. This method depends on determining a minimum spanning tree in a graph, then tracking previously visited vertices to construct a Hamiltonian tour (as to visit each "city" once). The final algorithm our group researched was the Ant Colony Optimization algorithm. This method relies on leaving "trails" while traversing a graph, then leveraging the residual-data to determine optimal future decisions. A more detailed summary of each respective algorithm is provided in the following sections.

1. 2-OPT ALGORITHM

1.1 Description

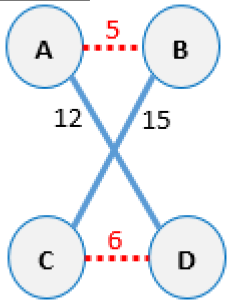
The 2-OPT Algorithm is predicated on the analysis and optimization of improving a whole-part solution by iterative, local steps. The preliminary stages consist of locating a tour (or multiple tours) in this case we used a nearest neighbor algorithm that starts at a given vertex, then, adds the closest vertex to the tour. It continues in this fashion until all vertices are visited. Once a tour has been found, the algorithm is purposed to compare the lengths between two crossing edges with a counter solution. The counter solution is the theoretical length of removing the current edges and establishing new edges in such a manner the connection to local vertices are swapped. This, in essence, uncrosses the edges so there is a direct path to potentially adjacent vertices. To provide a visualization, let us consider a graph G which contains a tour. Let us also consider a subset of four vertices that are a part of graph G's tour. We will call these vertices A, B, C, and D. Figure 1.1 depicts the graph, shows the current edges, and also defines lengths of the edges.

Figure 1.1



Here we can see that we have an edge connecting AD with a length of 12 units and an edge connecting BC with a length of 15 units. At this point, we now check our counter solution. This is done by swapping the edges, such that A connects to B and C connects to D. This is depicted in Figure 1.2 below.

Figure 1.2



If the counter solution provides a more optimal tour, then the original edges are deleted and replaced with that of the alternative paths. In our example, we see the original total length was 27. However, we have found a more optimal solution with a total length of 11 via the 2-OPT technique. This same process would be repeated iteratively until the tour in question reaches an optimal solution.

1.2 Pseudocode

```
TwoOPT(tour)
    do {
        gain = 0
        swap = False
        for all i to tour length n
            for j to tour length n
                alternativeGain = distance(i,j) + distance(i+1,j+1)
                -distance(I,i+1) – distance(j,j+1)
                if gain > alternativeGain
                    minI=I, minJ=j
                    gain = alternativeGain
                    swap = True
        if swap == True:
            swap vertices
    }while gain < 0;
    return tour
```

2. CHRISTOFIDE’S ALGORITHM

2.1 Description

In order to find an optimal solution to the traveling salesman problem, Christofides algorithm takes in a matrix of distances between nodes in the map and produces a route that on average will be less than 1.5 times the optimal path. From the coordinate matrix, a minimum spanning tree (MST) is created using an algorithm such as Kruskal’s or Prim’s. Then we find the Minimum-Weight Perfect Matching(no two edges share a common vertex) graph from the odd vertices of the MST. This is a very important step leading to locating the shortest path. Next up is to make the Minimum-Weight Perfect Matching graph and MST into a multigraph and convert it into an Euler Cycle path which finds a path of the multigraph that starts and ends at the same vertex. This is necessary because TSP’s goal is to produce the shortest path that goes through each coordinate in the input matrix. Finally, the end result is a Hamiltonian Path, created from the Euler Cycle by removing repeated vertices.

2.2 Pseudocode

```
coordinates = file.read()
def christofides(coordinates):

        graph = distance_matrix(coordinates)
        mst = kruskals_alg(graph) # get min spanning tree
        odd_verts = every other vertex in mst
        min_weight = []
        for vertex in odd_verts:
                find distance to each other vertex:
                        min_weight.append(shortest_dist)
        euler_cycle = euler_tour(multigraph(min_weight, mst))
        return solution = shortcut_euler_tour(euler_tour)
```

3. ANT OPTIMIZATION ALGORITHM

3.1 Description

The third algorithm our group studied to solve the TSP problem was the “Ant Colony Optimization” algorithm. The design of this algorithm is predicated on the behavior of ants in the wild, and works to mimic their instinctual navigation processes. An ant leaves a residual pheromone trail in its footpath while it searches for food. As a group of ants disperse in search for an optimal route to a food source, the ant who is able to locate the closest food source will arrive back at the nest before the others. The traversal of its path both two and from the nest will result in a stronger pheromone trackway than the trails left by the un-returned ants. The next wave of ants are generally drawn to follow the most pungent passageway. This effectively creates a positive feedback loop. As more ants travel the path, the stronger the pheromone trail becomes, and the more likely ants will be to choose that direction. It is by this method that ants eliminate unnecessarily long or inefficient paths and tread an optimal course.

The “Ant Colony Optimization” (or ACO) algorithm attempts to bring the essence of the aforementioned methodology to computer science applications. More specifically, this algorithm can be designed to help solve shortest path problems (in our case, the TSP). Provided a Graph G, “ant objects” can be placed on a node. Those ants would select which edge to go to next relative how “desirable” their options are. Desirability can be a metric derived from a combination of a few things: if an ant has visited the node already, the magnitude of the “pheromone”, and the distance of nodes. This would be an iterative process until the ants complete a tour. Afterward, the “trails” could be updated with the optimal details. This can be used as a “desirability” metric for future tour constructions.

3.2 Pseudocode

```
For time = 1 to number of cities
    For all ants
        While ant has not finished a tour
            Select next city c to move to with optimal “desirability”
            (pass to a “desirability helper function” which will
            calculate the best moves relative node, graph, and
            distance level factors)
        Calculate the length of ant’s tour
    Update all “trails” based on tours
End
```

4. GROUP SELECTION JUSTIFICATIONS

An important part of the selection process was to consider the strength and weaknesses of all the aforementioned algorithm. Though there are certainly methods to combat weakness and foster strengths for each respective algorithm, our team conferred to decide which factors we wanted to endeavor as a unit. The Ant Colony Optimization Algorithm offered an excellent iterative heuristic to leverage parallel feedback from multiple sources (ants). The more data ants generate, the better the solution can be optimized. That said, we felt the primary disadvantage of this implementation was that ACO builds one optimal solution through heavy use of probability. For the first iteration, an ant (or ants) may construct a tour that is not *most* optimal. For each subsequent iteration, the probability those same paths will be utilized to generate future tour decisions is higher than discovery of alternative optimal paths. We next evaluated the Christofide's algorithm. One of the biggest advantages of this implementation is the ability to capitalize on reduction. The algorithm simplifies the problem by splitting the graph into a minimum spanning tree. It can then determine the Hamiltonian tour. A disadvantage, like all solutions to TSP, is that it is an exponential algorithm. The more cities/nodes which are added to the data set, the worse the program will perform. The last algorithm we analyzed was the 2-OPT algorithm. This implementation, like Christofide's and the others, is subjected to an exponential worst case running time as well. That said, our group appreciated the advantage of route optimization via edge replacement. The notion of optimizing crossing routes by constructing more direct edges offers the ability to build a more optimal path by *improving* a specified tour. Additionally, since this is an iterative method we don't have to rely on probability or chance. We can be assured that a tour will be locally optimized at each step. For these reasons, our group choose the 2-OPT method as our implementation of choice.

5. DESCRIPTION OF OUR IMPLEMENTED 2-OPT SOLUTION

5.1 Description

We have already introduced the 2-OPT algorithm in a previous section, however we will clarify our particular implementation of the program here. Our group chose to develop our 2-OPT solution in python. We start by reading city data in from a file, creating a new instance of an object for each city, and then adding the instances into an array titled "cityArray". We then declare a new variable for our final distance and initialize it to infinity. [For testing purposes we set conditional statements to define how many tours will create later in the program based off input size, but for implementation had the ability to create a tour for all possible starting points in the provided graph]. Next, we start a timer so we could determine our system's run time. A loop subsequently leverages the nearest neighbor approach to build a tour for each starting point. Our nearest neighbor function will return an array of cities that comprise a tour. The distances for the tour are calculated via a tourDistance function (which accepts a tour array as an argument), and then we compare the length of the tour with the best tour we have found so far. If the most recent tour is more optimal than the previous, we store it as our new best tour. Otherwise we retain the old tour. As the loop exits, we are now ready to call our implementation of the Two_OPT function. We pass two parameters in to the function, the cityArray and our best tour (aka finalTour) that we found during our aforementioned loop. We have provided the pseudo-code of this function in section 5.2 for a thorough description of its design. The basic premise is quite simplistic. It performs "swaps" if a more optimal path can be made by uncrossing edges (see section 1.1 above for additional description if desired). The Two_OPT function will return an optimized tour, which we pass to our tourDistance function to calculate the new distance. Finally, we write our data to a new file for review.

5.2 Pseudocode

A copy of our instituted pseudocode that outlines out Two_OPT function is provided below.

```
Def Two_OPT:
    while True:
        minchange = 0
        swap = False
        for i in range(len(tour)-2):
            for j in range(i+2, len(tour)-1):
                change = dist(i,j) + dist(i+1,j+1)
                -dist(I,i+1) – dist(j,j+1)
                if minchange > change
                    minI=I, minJ=j
                    minchange = change
                    swap = True
                    if minchange >=0:
                        break
        if swap == True:
            swap vertices
    return tour
```

6. CONCLUSION: BEST TOUR RESULTS

As the conclusion of our group project, we will provide our implementation's best tours for the three example instances in addition to our solutions for the competition instances.

Our "best" tours for the three example instances were as follows:

- Instance-1: Time = 1.7s, Length = 115157, Ratio=1.06
- Instance-2: Time = 174.2s, Length = 2808, Ratio=1.09
- Instance-3: Time = 10hr, Length = 1955903, Ratio=1.24

Our results for the competition instances were as follows:

- Input-1: Time = 1s, Length = 5373
- Input-2: Time = 3s, Length = 7639
- Input-3: Time = 130s, Length = 12672
- Input-4: Time = 18s, Length = 20557
- Input-5: Time = 100s, Length = 28445
- Input-6: Time = 140s, Length = 39828
- Input-7: Time = 17min, Length = 63444

7. REFERENCES

[1] Anon. The ACM Computing Classification System (CCS). Retrieved August 13, 2017 from http://dl.acm.org/ccs/ccs.cfm

[2] Brownlee, Jason. Clever Algorithms: Nature-Inspired Programming Recipes. Retrieved August 13, 2017 from http://www.cleveralgorithms.com/nature-inspired/swarm/ant_colony_system.html

[3] Lettman, Jeff. Ant Colony Optimization. Retrieved August 13, 2017 from https://www.youtube.com/watch?v=xpyKmjJuqhk

[4] Yang, Jinhui. Shi, Xiaohu. Marchese, Maurizio. Liange, Yanchun. An ant colony optimization method for generalized TSP problem. Retrieved August 13, 2017 from http://www.sciencedirect.com/science/article/pii/S1002007108002736

[5] Burtscher, Martin. A High-Speed 2-OPT TSP Solver. Retrieved August 16, 2017 from http://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf

[6] Kuang, Eric. A 2-opt-based Heuristic for the Hierarchial Traveling Salesman Problem. Retrieved August 16, 2017 from http://honors.cs.umd.edu/reports/kuang.pdf

[7] MIT OpenCourseWare. The Traveling Salesman Problem. Retrieved August 16, 2017 from https://ocw.mit.edu/courses/sloan-school-of-management/15-053-optimization-methods-in-management-science-spring-2013/lecture-notes/MIT15_053S13_lec17.pdf

[8] Cornell. Design and Analysis of Algorithms. Retrieved August 16, 2017 from http://www.cs.cornell.edu/courses/cs681/2007fa/Handouts/christofides.pdf

[9] Cowen, Lenore. Park, Jisoo. Lecture 3: The Traveling Salesman Problem. Retrieved August 16, 2017 from http://www.cs.tufts.edu/comp/260/Old/lecture4.pdf

[10] Shekhawat, Anirudh. Poddar, Pratik. Boswal, Dinesh. Ant colony Optimization Algorithms: Introduction and Beyond. Retrieved on August 16, 2017 on http://mat.uab.cat/~alseda/MasterOpt/ACO_Intro.pdf

[11] Polo, Guilherme. Reducing the time of heuristic algorithms for the Symmetric TSP. Retrieved on August 17, 2017 on https://www.slideshare.net/gpolo/reducing-the-time-of-heuristic-algorithms-for-the-symmetric-tsp

[12] Asmerom, Ghidewon. Minimum Spanning Tree Algorithms. Retrieved on August 18th, 2017 on http://www.people.vcu.edu/~gasmerom/MAT131/mst.html

[13] Rahul, D.S. pyton-christofides 0.1.2 Retrieved on August 18th, 2017 on https://pypi.python.org/pypi/python-christofides/0.1.2

[14] Communal Authors Eulerian path. Retrieved on August 18th, 2017 on https://en.wikipedia.org/wiki/Eulerian_path

[15] Communal Authors. Christofides algorithm. Retrieved on August 18th, 2017 on https://en.wikipedia.org/wiki/Christofides_algorithm