Final Project Report
CS325 - Project Group 14
Katherine Isabella, Kenny Lew, Yu Ju Chang

Introduction

The traveling salesman problem (TSP) states that a salesman must visit n cities, making a tour such that each city is visited exactly once and finishes at the starting city. The salesman must also make the tour's total cost, which is the cumulative distances between cities along the tour, be a minimum. The straightforward way to solve this problem would be a brute force method, where all possible solutions are generated and each of these solutions is checked until the minimum distance tour (the optimal solution) is found. This brute force method guarantees that the optimal solution will be found; however, one can easily see that as the number of cities gets larger, the number of possible solutions grows very quickly. With n cities, there are n! combinations of solutions. Therefore, the runtime would be O(n!).

Because of the difficulties with finding an optimal solution in polynomial time, many approximation algorithms have been developed to find approximate solutions to the TSP and similar problems. We have researched three different algorithms for solving the Traveling Salesman Problem: 2-Opt, Minimum Spanning Tree (MST) Heuristic, and Simulated Annealing.

Research: 2-Opt

The 2-Opt algorithm will find the optimal solution of the path with running time of O(n^3). It basically checks 2 paths and by changing the two paths if the distance is improved, it keeps that new path and continue to do so throughout the tour. The pseudocode for 2-Opt is:
(https://en.wikipedia.org/wiki/2-opt)

```
minchange <- 0
while do minchange < 20
  best_distance = tour.tourdistance() // current distances
    from i = 0 to number of cities-1
      from  j=i+1 to number of cities
         set new_route to  2optswap(i, k) //call 2-opt swap function to swap routes
         set new_distance to  findtotaldistance( new_route) // calculate new total distance
         if new_distance is less than best_distance    //if improved perform resets
            set existing_rout to new_route
            set minchange to 0
            set best_distance to new_distance
     minchange++
```

(http://www.technical-recipes.com/2012/applying-c-implementations-of-2-opt-to-travelling-salesman-problems/)

```
2optswap(i, k) // 2opt swap function
//1. take route[0] to route[i-1] and add them in order to new_route
for(a=0...i-1)
    new_tour.setcity(a, getcity(a))
//2. take route[i] to route[k] and add them in reverse order to new_route
temp = 0
for(a=i...k)
   new_tour.setcity(a, getcity(k-temp))
temp++
```

```
//3. take route[k+1] to end and add them in order to new_route
for(a=k+1...tour.size)
   new_tour.setcity(a, getcity(a)
```

Research: MST Heuristic

The MST Heuristic algorithm creates a minimum spanning tree and then doubles every edge of the minimum spanning tree. Then the algorithm takes this doubled graph and computes a Euler tour. Afterwards, any repeated vertices in the tour are removed so every vertex is only visited once. The running time of MST Heuristic algorithm would give O(E lg E) and this algorithm is a 2-approximation algorithm. The pseudocode is:

"compute the MST T of the input G
construct the graph H by doubling every edge of T
compute an Euler tour C of H
// every v ∈ V is visited at least once in C
shortcut repeated occurrences of vertices in C to obtain a TSP tour" (Roughgarden 4)

(I researched Christofides algorithm first so the majority of the information below will be the same or similar to Christofides' explanation under "Other Research")

Expanded Pseudocode:

```
// algorithm is assuming the graphTSP will be a hashmap with the key's being the cityID's and the values
// being a vector of pairs (the adjacency list) where the key is the vector connect to the original vector
//and the value is the distance between the two vectors
mstHeuristic(graphTSP [0…n], cityIDs [0…n] ){

//using Kruskal's algorithm get the minimum spanning tree graph
mstGraph = getMST(graphTSP);

//double the graph using the doubleGraph function so we can then compute a Euler tour -(O(E))
doubleMST;

for(int i = 0; i < mstGraph.size(); i++){
        add to doubleMST, mstGraph[i];
        add to doubleMST, mstGraph[i];
}

//use the getEulerTour function to compute the Euler tour
eulerTour = getEulerTour(doubleMST);


//shorten the graph to remove any duplicate vertices
shortendedGraph;
```

```
eulerShortenMap[];

//create a hashmap to easily check if we used the vertex already or not
for(x = 0, x to combinedGraph.length, x incremenets){

        add to eulerShortenMap all the vertices in the combinedGraph as the key with the value being false;

}


//shorten the Euler tour by removing duplicate vertices takes O(n) where n is the eulerTour's length
for(x=0, x to eulerTour.length, x increments){




//if we use the eulerShortenMap /hashmap data structure this check can be done in constant time
if(vertex at eulerTour[x] is not in shortendedGraph){

        add vertex at eulerTour[x] to shortendedGraph;

        update eulerShortenMap so the key of the vertex at eulerTour[x] has a value of true;

}


//return the final tour result
return shortendedGraph;


-------FUNCTIONS USED ABOVE--------

//function to calculate the MST of a graph using Kruskal's algorithm
getMST(Graph g[0 … n]){

mstGraph [];
setsArray[];

//add all the vertices to setsArray
for (x = 0, x to n, x increments){
        add vertex of g[x] to setsArray;
}

//depending on data structure this may take more than O(E lg E) but with correct structure should only take the time
//it takes for mergeSort to run
sortedEdges = return from mergeSort() all the edges of G sorted into increasing order;

for( x = 0, x to sortedEdges.length, x increments){

        //using setsArray, check if the sets of the vertices which are connected by sortedEdges[x] are the same;
        if (sortedEdges[x]v1 and sortedEdges[x]v2 are not in the same set){
                add sortedEdges[x] and the vertices it connects to mstGraph;
                in setsArray, create a union of the two vertices connected by sortedEdges[x];
```

```
        }

}
return mstGraph;
}


//using Hierholzer's Algorithm http://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/
getEulerTour(doubledGraph){

eulerTour;
currentPath;

//doesn't matter where we start so just pick any vertex in doubledGraph
//to place the first vertex from the doubledGraph into currentPath;
currentPath{doubledGraph[V0]};
currentVertex = doubledGraph[V0];


while (there are unused edges in doubledGraph){

        if (currentVertex has an unused edge){
                add the connected vertex to currentPath;
                currentVertex = the connected vertex;
        }
        else{
                //work backwards through currentPath to find a vertex with unused edge
                add currentVertex to the eulerTour;
                currentVertex = previous currentVertex;
        }
}

combine eulerTour and currentPath by putting the contents of currentPath at the end of eulerTour;

reverseEuler;

//need to reverse the eulerTour found to get the correct tour path to follow
for (x  =  eulerTour.length, x > 0, x decrements){
        reverseEuler = eulerTour[x-1];
}

return reverseEuler;
}
```

Running Time:

From the above pseudocode, one can see there are several running times for each of the other algorithms within MST Heuristic algorithm.

For Kruskal's algorithm we have an overall running time of O(E lgE). We first created the setsArray by going through all the vertices in the graph O(V). Then we used merge sort to sort all the edges (O (E lg E)). Next, we go through all the edges to see if the two vertices' sets are the same and, if we need to, create a union. These set operations can be nearly constant if done properly so we have O (E) for this section. Therefore, the running time for this algorithm is O(E lg E), the time it takes to sort the edges.

For doubling the edges within the MST, we will need to go through each of the edges once which will be O(E).

The Euler tour takes O(E) time as we go through each of the edges once.

Finally, we shorten the Euler Tour and remove the duplicate vertices. By using a hashmap to keep track of which vertices have been shortened already, this can be done in O(E) time.

Therefore, the running time which dominates this algorithm is Kruskal's algorithm O(E lg E). However, it should be noted that in order to use the mstHeuristic algorithm, the function expects an adjacency list with all the edges' weights already calculated. As all the vertices connect to one another in this TSP problem, creating this adjacency list with weights will be O(V^2).

Research: Simulated Annealing

Simulated annealing was inspired by the metallurgical practice of annealing metalwork. Annealing is the process of repeatedly heating and cooling a material to form a desired internal structure. The heating increases the energy of the material's atoms and therefore the randomness of atom movement, which gives them the opportunity to align themselves, and the cooling locks these atoms into place. Simulated annealing tries to mimic this process.

Before discussing simulated annealing, we must have an overview of a similar algorithm – hill climbing. Hill climbing is an optimization technique that iteratively searches for a better solution by randomly changing a single element of the solution. If the change produces a better solution, this solution is taken to be the new solution and the process is repeated until the solution no longer improves. As one can see, it can be easy to reach a point where no more better solutions are found, while still not finding the global optima. With simulated annealing, we can avoid becoming stuck within a local optimum by allowing the algorithm to sometimes accept worse solutions.

The Simulated Annealing algorithm is a randomized optimization algorithm, which uses the process of annealing to find a better solution. A temperature variable is used to simulate the heating process, and this variable is "cooled" as the algorithm runs. The temperature starts high initially to facilitate higher probabilities of accepting new solutions, for better or worse. As the temperature is cooled, only better solutions are selected. This allows the algorithm to escape being trapped within a local minimum, unlike a similar algorithm hill climbing. This acceptance is determined by an acceptance probability, which is tied closely to the temperature such that a high temperature leads to a higher acceptance probability and a lower temperature to a lower acceptance probability. A very small temperature would essentially be basic hill climbing; whereas, a very large temperature is like a random walk.

The input to the algorithm is an initial solution to the problem. This solution can be generated with any other method (e.g., Nearest Neighbor) or could be a random tour. Two things that are not explicitly defined in this pseudocode is the cooling schedule function, cooling_schedule(), and the acceptance probability. These are choices to be made by the programmer. The cooling schedule dictates how fast the temperature will decrease (or "cool") as the algorithm runs. The acceptance probability calculates a probability that the algorithm would still accept the candidate solution if it was worse than the current solution. ΔE is the difference between the cost of the current solution and the cost of the candidate solution; that is, how much better or worse the candidate solution is. $\Delta E > 0$ would mean that the candidate solution is better. The generalized pseudocode is as follows:

```
TSP-SA (solution):
solution = a random solution                    // select a random solution for optimization
current = solution                              // set initial solution the previously generated solution
Temperature = inf                               // some very high number to start

for t = 1 to inf:
  Temperature = cooling_schedule(t)             // how fast the temperature decreases (cools)
  if T = 0
     return current
   new_solution = a solution derived from randomly changing an element of current
   ΔE = cost(current) - cost(new_solution)
   if ΔE > 0
     current = new_solution                     // if it is better, then take it
   else
     current = next depending on acceptance probability   // may still take the solution
 return current
```

The steps in more detail are as follows:

1. Start with some prior solution. This can be a random tour or an arbitrarily selected tour from a set of solutions.
2. Pick a new candidate tour at random from all neighbors of the existing tour. This candidate tour might be better or worse compared to the existing tour.
3. If the candidate tour is better than the existing tour, accept it as the new tour.
4. If the candidate tour is worse than the existing tour, it may still be accepted, depending on the acceptance probability. This probability is a function of how much worse the candidate is compared to the current tour, and the temperature of the annealing process. A higher temperature is more likely to accept an worse tour.
5. Go back to Step 2 and repeat. Upon each iteration, the temperature is lowered depending on the cooling schedule until a desired low temperature is reached.

<u>Research: Christofides Algorithm</u>

Pseudocode:

Final Project Report
CS325 - Project Group 14
Katherine Isabella, Kenny Lew, Yu Ju Chang

Below is the basic pseudo-code I found in my research:

"compute the MST T of the input G
compute the set W of vertices with odd degree in T
compute a minimum-cost perfect matching M of W
construct the graph H by adding M to T
compute an Euler tour C of H

// every v ∈ V is visited at least once in C
shortcut repeated occurrences of vertices in C to obtain a TSP tour" (Roughgarden 6)

I expanded upon this pseudocode and gave more detailed pseudocode specific to how I envision our TSP project using this algorithm:

```
// algorithm is assuming the graphTSP will be a hashmap with the key's being the cityID's and the values
// being a vector of pairs (the adjacency list) where the key is the vector connect to the original vector
//and the value is the distance between the two vectors
christofidesAlgorithm(graphTSP [0…n], cityIDs [0…n] ){

//using Kruskal's algorithm get the minimum spanning tree graph
mstGraph = getMST(graphTSP);

//get which vertices have an odd number of edges in the mstGraph. Mstgraph returns a hashmap of
//cityID as key and vectors as values ( for the adjacency list)
oddEdged[];

for(x = 0, x to mstGraph.length, x increments){
if(mstGraph[cityIDs[x]].length % 2 != 0){
add the vertex to oddEdged;
}
}

//for this algorithm I was researching Edmond's Blossom algorithm. I found this algorithm pretty hard to
//wrap my head around and was unable to expand the pseudocode for this part. To see some
//pseudocode and read more about it see: http://www.imsc.res.in/~meena/matching/edmonds.pdf and
// https://en.wikipedia.org/wiki/Blossom_algorithm  and https://brilliant.org/wiki/blossom-algorithm/ .
// There is an improved O(n^3) version of this algorithm created by Lawlwer as mentioned in
//Christofides paper (Christofides 4)
minimumMatchingGraph = getMinMatch(graphTSP, oddEdged);


combinedGraph = combination of the mstGraph and the minimumMatchingGraph;
//complete the Euler tour
eulerTour = getEulerTour(combinedGraph);

shortendedGraph;
eulerShortenMap[];
```

```
//create a hashmap to easily check if we used the vertex already or not
for(x = 0, x to combinedGraph.length, x incremenets){
        add all the vertices in the combinedGraph to eulerShortenMap as the key with the value being false;
}

//shorten the Euler tour by removing duplicate vertices
for(x=0, x to eulerTour.length, x increments){

//if we use the eulerShortenMap /hashmap data structure this check can be done in constant time
if(vertex at eulerTour[x] is not in shortendedGraph){
        add vertex at eulerTour[x] to shortendedGraph;
        update eulerShortenMap so the key of the vertex at eulerTour[x] has a value of true;
}
return shortendedGraph;
}

-------

//function to calculate the MST of a graph using Kruskal's algorithm
getMST(Graph g[0 … n]){

mstGraph [];
setsArray[];

for (x = 0, x to n, x increments){
        add vertex of g[x] to setsArray;
}

sortedEdges = return from mergeSort() all the edges of G sorted into increasing order;

for( x = 0, x to sortedEdges.length, x increments){

//using setsArray, check if the sets of the vertices which are connected by sortedEdges[x] are the same;
if (sortedEdges[x]v1 and sortedEdges[x]v2 are not in the same set){
        add sortedEdges[x] and the vertices it connects to mstGraph;
        in setsArray, create a union of the two vertices connected by sortedEdges[x];
}

}
return mstGraph;
}

//using Hierholzer's Algorithm http://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/
getEulerTour(combinedGraph){

eulerTour;
currentPath;

//doesn't matter where we start so just pick any vertex in combinedGraph
//to place the first vertex from the combinedGraph into currentPath;
```

```
currentPath{combinedGraph[V0]};
currentVertex = combinedGraph[V0];


while (there are unused edges in combinedGraph){

        if (currentVertex has an unused edge){
                add the connected vertex to currentPath;
                currentVertex = the connected vertex;
        }
        else{
                //work backwards through currentPath to find a vertex with unused edge
                add currentVertex to the eulerTour;
                currentVertex = previous currentVertex;
        }
}
combine eulerTour and currentPath by putting the contents of currentPath at the end of eulerTour;
reverseEuler;

//need to reverse the eulerTour found to get the correct tour path to follow
for (x  =  eulerTour.length, x > 0, x decrements){
        reverseEuler = eulerTour[x-1];
}
eulerTour = reverseEuler;

return eulerTour;
}
```

Running Time:

From the above pseudocode, one can see there are several running times for each of the other algorithms within Christofides' algorithm.

For Kruskal's algorithm we have an overall running time of O(E lgE). We first created the setsArray by going through all the vertices in the graph O(V). Then we used merge sort to sort all the edges (O (E lg E)). Next, we go through all the edges to see if the two vertices' sets are the same and, if we need to, create a union. These set operations can be nearly constant if done properly so we have O (E) for this section. Therefore, the running time for this algorithm is O(E lg E), the time it takes to sort the edges.

For finding the set of vertices with an odd number of edges in the minimum spanning tree, we go through the list of vertices, meaning O(V) running time.

In order to compute the minimum-cost perfect matching M of W, there are several algorithms out there. The one mentioned in Christofides' paper is an algorithm by Lawler O(n^3).

In order to combine the two graphs we need to go through each of the vertices in the minMatchingGraph and add their adjacency lists to the mstGraph, accounting for the fact we might have duplicate edges. (O(V)).

The Euler tour takes O(E) time as we go through each of the edges once.

Finally, we shorten the Euler Tour and remove the duplicate vertices. By using a hashmap to keep track of which vertices have been shortened already, this can be done in O(E) time.

Therefore, the running time which dominates this algorithm is O(n^3), the time it takes to computer the minimum-cost perfect matching M of W.

Accuracy in finding Optimal Solution: This algorithm is a 3/2 approximation algorithm (Roughgarden 6-7)

Limitations: Can only be used in metric Traveling Salesman Problems, where there is a triangle inequality for the distances between the cities.

Description of project algorithm

For our project algorithm, we decided to store the data on nodes which contain the city id, x and y coordinate, previous node, next node and an adjacency list of distances to the other cities. First we read in the data from the command line and store the data into the node. This data is then used to create an adjacency matrix.

Next, we use the nearest neighbor algorithm to connect the cities and create an initial solution. The nearest neighbor algorithm takes the tour's most recently added city, finds the next closest city (minimum distance) to it, and adds that city to the tour. The algorithm will repeat until it all the cities have been added to the tour. It then connects the last city to the starting city to complete the tour.

After we have this initial solution, we then optimize the route by using Simulated Annealing. The Simulated Annealing algorithm uses a 2-Opt swap technique to swap two randomly selected cities on the tour and recalculates the total distance of the new tour. If this change improves the solution, then it is kept. If the change is worse, an acceptance probability is calculated to determine the likelihood of the algorithm still keeping the change. This acceptance probability is influenced by the temperature, which is decreasing as the algorithm runs. High temperature increases the acceptance probability and low temperature results in low probabilities. This means that when temperature is higher early in the algorithm, the Simulated Annealing process would accept be able to escape from local minima if necessary by accepting a worse solution. Likewise, in the late stages of the algorithm runtime, the temperature is low so it becomes more unlikely to accept worse solutions. The algorithm continues in this Simulated Annealing process until the minimum temperature is reached.

After we have the approximated solution, we write the total distance and the route taken into the tour file.

Discussion

The nearest neighbor algorithm was chosen based on the fact it is easy to implement and it would provide

a good starting solution. After we implemented the nearest neighbor, we ran some of the given tests and realized we needed to optimize the algorithm more. We then decided to use 2-Opt algorithm to further optimize our initial solution. As discussed above in the research portion, this algorithm optimizes by going through all possible city pairs and swapping them, keeping the shortest tour. Both nearest neighbor and 2-Opt are easily implemented, and popular algorithms to be used together for the TSP problem.

However, the 2-Opt algorithm runs for a very long time for large datasets. This is due to searching each possible pair of cities. Therefore, the 2-Opt was altered so that instead of sequentially searching each pair of cities and swapping them to find a better solution, we randomly selected a pair of cities to perform the 2-Opt swap. This was essentially Hill Climbing. This idea was taken further by implementing Simulated Annealing which was similar to Hill Climbing but allowed for worse solutions to be accepted for the greater good of possibly finding a better solution later on.

Pseudo Code

```
read in data file
store data into node
create unordered map

// nearest neighbor algorithm
do until all cities have been visited
   set current location to a random city from the map
   add city to current route
   add current location to visited vector
   connect the current city to its next closest city
   set current city to next
   add city to current route
connect the last city visited to the first city visited

//simulated annealing
set bestDistance to nearest neighbor solution distance
set temperature to 1  // low temperature so that not too many worse solutions are accepted
set coolingRate to 0.99  // slow cooling schedule
set minTemperature to 0.0001
define acceptanceProbaility, randomProbability
set number of iterations to 1000

while temperature is larger than min temparature
   for j = 0 to number of iterations
   set i and k to random cities

      //2-Opt swap
      set new route to swap cities i and k in the current route
      calculate new route distance
```

```
    if no change to distances
        continue to next iteration
    if new distance is better than best distance
        set current route to new route
    else
        set acceptanceProbability to exp^((bestdistance - new distance) / temperature))
      set randomProbability to random number between 0 and 1
      if acceptanceProbability is larger than random probability
          set current route to new route   // accept the worse route
  if 3 minute time limit has been reached
      stop loop
  update the temperature according to the cooling schedule
write solution to output file
```

Best tours:

Test Case	Group Solution	Optimal Solution	Ratio	Time (sec)
tsp_example_1.txt	114150	108159	1.055	30.41
tsp_example_2.txt	2854	2579	1.107	141.21
tsp_example_3.txt	1937424	1573084	1.232	335.08

Competition tours:

Timed - 3 minutes:

Test Case	Size	Min Distance	Time (sec)
1	50	5496	20.71
2	100	7655	44.08
3	250	12923	127.25
4	500	17993	172.77
5	1000	26051	174.42
6	2000	39703	174.41
7	5000	63421	174.62

Final Project Report
CS325 - Project Group 14
Katherine Isabella, Kenny Lew, Yu Ju Chang

Unlimited time:

Test Case	**Size**	**Min Distance**	**Time (sec)**
1	50	5496	20.71
2	100	7655	44.08
3	250	12923	127.25
4	500	17941	312.06
5	1000	25162	707.18
6	2000	37594	1641.79
7	5000	61934	4744.76

Final Project Report
CS325 - Project Group 14
Katherine Isabella, Kenny Lew, Yu Ju Chang

References

2-Opt

https://en.wikipedia.org/wiki/2-opt

https://link.springer.com/article/10.1007/s00453-013-9801-4

http://www.technical-recipes.com/2012/applying-c-implementations-of-2-opt-to-travelling-salesman-problems/

http://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf

MST Heuristic / Christofides

Christofides, Nicos. *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem*. Management Science Research Group, Graduate School of Industrial Administration, Carnegie-Mellon University, 1976. (Link with article: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA025602)

Roughgarden, Tim. (2016). Lecture #16: The Traveling Salesman Problem. Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu. Online. http://theory.stanford.edu/~tim/w16/l/l16.pdf

http://www.cs.princeton.edu/~wayne/cs423/lectures/approx-alg-4up.pdf

http://www.cs.tufts.edu/comp/260/Old/lecture4.pdf

http://www.cs.cornell.edu/courses/cs681/2007fa/Handouts/christofides.pdf

http://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/

http://www.imsc.res.in/~meena/matching/edmonds.pdf
https://en.wikipedia.org/wiki/Blossom_algorithm

https://brilliant.org/wiki/blossom-algorithm/

http://personal.vu.nl/r.a.sitters/AdvancedAlgorithms/2016/SlidesChapter2-2016.pdf

Simulated Annealing

https://www.youtube.com/watch?v=K7vc60jn1KU

https://www.youtube.com/watch?v=enNgiWuIHAo

https://en.wikipedia.org/wiki/Hill_climbing

https://en.wikipedia.org/wiki/Simulated_annealing

Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.