Group Members: Ryan Wholey, Lauren Mackey, Andrew Sturtevant
Group Number: Group 28
Assignment: Group Project
Course: CS 325

Index:

Algorithms Considered:

Branch and Bound:

General summary:

The idea of the algorithm is to calculate the best possible “branch” or route and “bound” or cut
off the routes that must be longer than our current “best”. Each evaluated node has a calculation associated with it based on a matrix reduction cost and its preceding path. A general outline of reduction is described in the algorithm overview section, but for our purposes here the reduction is process of finding the cost associated with travelling to that node. Once we reach the end of a path (no more nodes to go to) we update our “best” and look at all other possibilities. If there are any that are less than our best, we explore those as well. If there are any that are worse than our best, we can ignore them because we’ve found a path that is complete that costs less than that incomplete path which can only increase, never decrease.

Initial Thoughts:

I’m a bit worried about the time and space aspect because were implementing in python. The algorithm requires creation of nXn matrices for each investigated node which will absolutely begin to take up space as we grow our n node size. More required space can sometimes help reduce the time an algorithm takes but in this case, we’ll have to construct new matrices for each node so I think it might make our approach very slow. Probably not slower than finding the actual tour though, as that is problem is NP Hard. My hope is that snipping possible branches off of our state tree will outweigh the cost of creating all of these matrices.

algorithm overview:
Step 1: Get initial cost for node 1

1) Write the initial cost matrix
    a) fill in the matrix with the corresponding edge weight between the two vertices
    b) for spaces where nodes connect to themselves, use infinity because this is a minimization problem
    c) if a node cannot be reached, write it in as infinity

2) Reduce the initial cost matrix
    a) start with row reduction
        i) scan each row taking the minimum value
        ii) subtract the minimum for that row from every item in that row so that one item is 0
    b) next do column reduction
        i) scan each column taking the minimum value
        ii) subtract the minimum from each item like with row reduction
2) To find the cost of the first node, we sum all of the reduction minimums, ie the minimum from each row + the minimum from each reduced column

Step 2: Get costs for children
1) For each potential unvisited child, create a new cost matrix from the parent cost matrix
    a) When looking at our child, we want to make sure we don't go back the way we came, so we change the value of the row and columns corresponding to the backtracking paths to be infinity, that way we won't choose them because we'd like to minimize
    b) reduce after adding infinities
    c) child cost will be cost of edge from parent to current child + cost of parent reduction + cost of any excess reduction done in the child reduction step
2) After reducing all children in the current level, scan all unexplored children (on all levels) and determine the minimum, this will be the next node to explore

Step 3: Upon finding a leaf node:
1) Update the minimum possible cost (the initial reduction cost)
2) Compare the new minimum to all other possible child routes
    a) if they are all greater, we have found our solution (because we have found a path and it is less than the other incomplete branches)
    b) If there are any smaller than our newly found minimum, explore that child until another leaf node is found and repeat

Psudocode:

```
// mark edges in a matrix noting ay edges connecting any vertices
get_inital_cost_matrix(current_node, vertex_list, edge_list)
   create emtpy cost matrix
   for each vertex:
      for each vertex:
         if there's an edge:
            store the weight in the cost matrix
         if not or the vertex is itself:
            store infinity
   return cost matrix
```

```
// subtract minimum number for each row, each column sum reductions for cost
reduce_matrix_and_get_cost(cost matrix)
    row reduction cost = 0
    column reduction cost = 0

    for each row in cost matrix:
        get minimum value
        row reduction cost = row reduction cost + minimum
        if not zero:
            for each item in row:
                item = item - minimum
   for each column in cost matrix:
        get minimum value
        column reduction cost = column reduction cost + minimum
        if not zero:
            for each item in column
                item = item - minimum
   reduction cost  = row reduction cost + column reduction cost
   return reduction cost

// save best
best = None
// save possible children
child_costs = []

// main branch and bound
branch_and_bound(vertex_list, edge_list, current_vertex = None, parent_cost_matrix=None)
    current_vertex = current_vertex or vertex_list[0]
    remove current from child costs if its there

    parent_cost_matrix = parent_cost_matrix or get cost matrix (current_vertex, vertex_list,
edge_list)

    if best is none:
        best = parent_cost
    if vertex is a leaf node:
        update best

    for each connected child:
        child_cost_matrix = get_cost_matrix(child, parent_matrix)
        set parent pointer on child_cost_matrix
        add child_cost_matrix to child_costs
```

```
if vertex is leaf:
    check all other child costs against current best
    if all worse than leaf's best:
        return calculate and return path from parent pointers
    else run branch_and_bound(vertex_list, edge_list, best_child, parent_cost_matrix)
```

Research Documents:
I found [1] to be incredibly helpful as I was having a hard time reading through many of the academic papers that I've come across. Professor Bari gives a really detailed walkthrough about how the branch and bound implementation works, taking care to discuss each step.

This paper [2] gives a brief overview of other heuristics for solving the TSP. It was really cool to see different approaches separated into different groupings, it helped me get a better mental model of the different approaches.

This document [3] was a bit much for me initially, but after spending some time with the algorithm I can follow along somewhat. I think if we decide to use this algorithm as the one we'd like to implement, some of the ideas in this paper will be very useful for how to turn the concept of the algorithm into code, which is always the hardest part for me.

I have a decent idea now of how the branch and bound dfs approach would work to solve the TSP, but I wanted to know if it was a "good" solution when compared to other approaches. The research in this paper [4] on a depth first search branch and bound suggests that it outperforms local search algorithms (greedy?) which is another algorithm our group has is investigating.

References:
[1] Abdul Bari, "Traveling Salesman Problem - Branch and Bound" April 13, 2018
"https://www.youtube.com/watch?v=1FEP_sNb62k

[2] Christian Nilsson, "Heuristics for the Travelling Salesman Problem"
https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf

[3] Richard Weiner, "Branch and Bound Implementations for the Traveling Salesman Problem" March 2003 http://www.jot.fm/issues/issue_2003_03/column7.pdf

[4] Weixiong Zhang, "Depth-First Branch-and-Bound versus Local Search: A Case Study" 2000
https://pdfs.semanticscholar.org/b0cb/70cc6b3670909bed45a16fa99657082e122c.pdf

---

Lin-Kernighan

The LK algorithm has been shown to be one of the more successful approximations to the TSP problem. It starts with an initial tour of cities and works to optimize this tour. Generally, it works by completing the following steps:

- With each iteration of the algorithm, a number, k, is decided upon. This is the number of edges that should be replaced to obtain a shorter overall tour.
- When these edges are deleted, the algorithm decides how to put the links (cities) back together with new edges to achieve a shorter tour.
- To do this, the algorithm considers the set of possible exchanges it can make. If it's able to find a shorter tour with any of these exchanges, it makes them and thus creates a new tour.
- The algorithm repeats this process until no exchange yields an improvement.

More detailed:

- Let's say T is the current tour. On each iteration step, we look for two sets of links (cities), X = {x1… xr} and Y = {y1… yr}, such that if X links are deleted and replaced with Y links, we'll get a better tour. This exchange is an r-opt move, where r is a variable number of links that will be swapped.
- In order to determine what links to include in X and Y, we add them one by one, based on the following criteria:
    - *Sequential exchange criterion:* There must be a shared endpoint between Xi and Yi (where Xi is the current link being considered to be added to X and Yi is the current link being considered to be added to Y). There must also be a shared endpoint between Xi+1 and Yi. Further, the chain of links added to X and Y must be closed, so if we are going to add Xi and Yi, we must make sure it satisfies this.
    - *Feasibility criterion:* If Xi is chosen, it's endpoint must join to another endpoint in X to form a tour.
    - *Positive gain criterion:* Yi should only be chosen if the Gain from the proposed exchanges is positive. We can calculate the Gain by summing up all of the individual gains for each link exchange:
        - gain(i) = cost(Xi) - cost(Yi) or the gain from swapping the individual links, Xi and Yi.
        - Gain(i) = gain(1) + … + gain(i)
    - *Disjunctivity criterion:* X and Y must be disjoint sets.
- Once we've gone through all links and determined which ones should be added to X and Y, we make the swap between X and Y. We then repeat the entire process until no positive Gain exchanges can be found.

Pseudocode:

1. Generate random tour T
2. Choose an endpoint t1, set i equal to 1
3. Choose a link X1 that has endpoints t1, t2 in T
    a. Set X1 to tried
4. Choose a link Y1 that has endpoints t2, t3 not in T where the Gain(1) is positive
    a. If this is not possible, go to step 12
5. Increment i by 1
6. Choose a link Xi that has endpoints t2i-1, t2i in T that satisfies:
    a. If t2i is joined to t1 we will have a closed tour T' in the graph
    b. AND Xi does not equal Ys for all s < i
    c. If T' is a better tour solution than T, reset T to equal T':
        i. Set all other previously tried links to untried.
        ii. Go to step 2.
7. Choose a link Yi that has endpoints t2i, t2i+1 not in T that satisfies:
    a. Gain(i) is positive
    b. Yi does not equal Xs for all s <= i
    c. AND Xi+1 exists
    d. If this Yi exists, go to step 5.
8. If we can find an untried alternative for Y2, set i equal to 2, go to step 7.
9. If we can find an untried alternative for X2, set i equal to 2, go to step 6.
10. If we can find an untried alternative for Y1, set i equal to 1, go to step 4.
11. If we can find an untried alternative for X1, set i equal to 1, go to step 3.
12. If we can find an untried alternative for t1, go to step 2.
13. Stop

As a note, the Lin-Kernighan algorithm is the general approach to this problem, but there have been other implementations of it that have been shown to do better. One such implementation is the Lin-Kernighan-Helsgaun (LKH) algorithm, which produces one of the best approximations to the TSP problem currently available. The main difference with this algorithm is that it doesn't restrict the searching of links to swap to the five nearest neighbors, and rather opens up the search to make swaps.

Sources:

- http://akira.ruc.dk/~keld/research/LKH/LKH-1.3/DOC/LKH_REPORT.pdf
- https://www.youtube.com/watch?v=2swNkneUmg0
- https://www.youtube.com/watch?v=GsMZYDBFJv4

Greedy

This algorithm is pretty straightforward in its approach. First we would take all of our input and create an adjacency list (Edge list in the Pseudocode) with the distance between each city. We'd then sort that list such that the minimum distance is the first element in the adjacency list. Once the sort is complete, we take the shortest distance edge, add it to our Tour, while also keeping track of the degree of each vertex. We continue adding the next shortest distance to the tour list as long as the edge meets the two following conditions:

- It does not cause a vertex to have a degree greater than 2
- It does not create a cycle while the Tour list has a length less than the number of cities (vertices)

Once we have N number of edges added to the Tour list, we can stop the loop

This algorithm is mostly bound by creating the adjacency list and also sorting that list. To create the adjacency list is O(n^2) and the sorting (assuming mergesort or some other efficient sorting) is O(n^2log n)

For our assignment, I believe this will be a fairly easy implementation, the only thing I am having trouble visualising and getting down on paper is reconstructing the graph such that we can output it properly. This particular method does not provide super optimal answers, according to Nillson, only getting within 15-20% of the optimal answer.

Pseudocode

```
Vertex
	#
	X-coord
	Y-coord
	degree
List of Vertices

Input:
	For each line in the file
		Create a vertex
		Append to list of vertices

Edge:
	Vertex v1
	Vertex v2
	Distance = sqroot((v1.x-coord - v2.x-coord)^2 + (v1.y-coord - v2.y-coord)^2)
	visited

Create Edge List:
	For every vertex i in list of Vertices
```

```
                For every j in list of Vertices:
                        If i.# = j.#:
                                Create edge, override distance to be infinity
                        Else create edge between vertex i and j, append Edge to the list of
                edges

Mergesort the edge list by distance

For each edge, starting with shortest distance until the Tour list has N edges:
        See if Edge is in Tour list:
                If not, check if either of its vertices have a degree greater than 2
                        If not, add to Tour List
                        Check that it does not create a cycle if Tour List length is less than N
                                If it creates a cycle, remove edge from Tour list, move on to
                                next edge
                        Increase degree of both vertices by 1
                        Add distance to the total distance in the Tour

Output:
        Take tour list, list v1 of the first edge as starting vertex
        Then v2 of that edge
        Then, find another edge that has v2 of first edge as a shared vertex, print the third
        vertex. Repeat this step until all edges in Tour list have been visited.
```

References:
https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf
http://lcm.csa.iisc.ernet.in/dsa/node186.html

Description of Implemented Algorithm and Why We Selected It:

We ended up implementing the Greedy algorithm described above, and then we added in a 2-opt method to optimize the original tour created by Greedy (if time allowed for it). We first selected only Greedy as our algorithm because we thought this would be a fairly straightforward way to get a near-optimal solution. After implementing this, though, we realized that some of the test files did not return a solution within the 1.25 bound. Because of this, we added a 2-opt method to further optimize the tour returned from Greedy. This allowed us to reach within-bounds solutions for all examples. We also ended up rewriting our code from Python to C++ to make sure we met the time requirements.

The Greedy method we implemented closely matches what is described above: we create Vertex objects for each city, and then we create Edge objects to store all possible Vertex

connections. We then sort a list of Edges, with the shortest distance between Vertices coming first. We take from the top of this list and add it to a tour list if it meets the following criteria:

- It does not cause a vertex to have a degree greater than 2
- It does not create a cycle while the Tour list has a length less than the number of cities (vertices)

We continue adding from the top of the sorted list, until we no longer have a Vertex to add. Once we have a starting tour list, we create an array of Vertices visited in order. We then check the time - if we have enough time left to optimize this initial tour, we do so by calling a 2-opt method. This method checks every possible pair of vertices to see if swapping them will make the overall tour distance shorter. We continue this check until there are no longer improvements to be made to the tour and then return the optimized tour. While performing these optimizations, we are also checking the time to see if we need to return before we are able to fully optimize.

At that point, we have an optimized tour. We print the result to an output file.

Pseudocode:

```
Vertex
	#
	X-coord
	Y-coord
	degree

Edge:
	Vertex v1
	Vertex v2
	Distance = sqroot((v1.x-coord - v2.x-coord)^2 + (v1.y-coord - v2.y-coord)^2)

Input:
	For each line in the file
		Create a vertex
		Append to list of vertices
	Create an Edge List

Create Edge List:
	Create Connection Matrix of all Vertex connections
	For every vertex i in list of Vertices
		For every j in list of Vertices:
			If i.# = j.#:
				Continue
			Else:
				Create edge between vertex i and j
				Append Edge to the list of edges
				Set values in Connection Matrix to true

Mergesort the edge list by distance
```

```
Create a Tour List:
	For each edge in Edge List:
		If both vertices have degrees less than 2:
			Check that it won't create a cycle if we add these vertices
			Check if the edge has the last vertex to be added

			If no cycle is created or a cycle is created but it's the last vertex:
				Increase degree of both vertices by 1
				Add the edge to the tour

Create an Ordered Tour:
	Add the first vertex in the first edge of the original Tour List to the Ordered Tour
	Set the search city to the second vertex of that first edge
	Remove this edge from the Tour List

	While there are still Edges in the Tour List:
		For each Edge in the Tour List:
			If the id of either Vertex of that Edge matches the search city id:
				Add this Vertex to the Ordered Tour
				Reset the search city to the opposite Vertex of that Edge
				Remove this Edge from the Tour List
			Restart the search index

Write Output:
	Get total distance and write
	Iterate through Ordered Tour and print each id
```

2-Opt Source (not listed above): https://en.wikipedia.org/wiki/2-opt

---

Best Example Problem Solutions:

NON COMPETITION FILES				
	Route Weight	Given Optimal	Ratio	Time(sec)
tsp_example_1	112426	108159	1.039451178	0.15
tsp_example_2	2748	2579	1.065529275	16.14
tsp_example_3	1821699	1573084	1.158043054	447.73

Best Competition Problem Solutions

	Route Weight	Time(sec)
test-input-1	5601	0.05
test-input-2	7678	0.51
test-input-3	12540	6.29
test-input-4	17977	68.93
test-input-5	24958	170
test-input-6	37056	170
test-input-7	59636	170