Deliverable D Process and Methods

Introduction:

This was a fun program to work on! Throughout the last couple weeks, my efforts have improved the processing time from a couple minutes to a couple seconds. I learned there are many options that can be adapted to find shorter paths. Overall, simulated annealing seems to be the best fit for my process and implementation.

Process overview in order of appearance:

1) Stored the value 100000 in a global int variable longEdge, so that I do not have to keep track of zeros every time it is needed.
2) Set Random variable rand to new Random method for a random int generator.
3) Set final boolean VERBOSE to false. ***If changed to true, all test prints and all attempted paths will print.*** (Suggested by Jessica Maistrovich – thought it was a great idea!)
4) Copied the method copyNodeList() from DelivC, which makes a copy of the g.nodeList.
5) Created addLongEdges to add edges with distances of 100,000 to copyNodeList().
6) Created method startToGoal() to place start city in first index, goal city in last index, and all other cities in between start and goal.
7) Created method calculateShortDistance() to sum all distances on path that are less than 100000.
8) Created Method calculateDistance() to sum all distances on the path.
9) Created method findMaxEdge to return the maximum edge of the current cities list.
10) Created method swapTwoCities() to swap a city returned from findMaxEdge() with a random city that is not a start or goal city.
11) Created method count() to determine how many times method betterPath() will iterate to determine possible paths.
12) Created method betterPath() to determine whether a better path is found based on a shorter distance calculated.
13) Created method printCities() to print the city abbreviations of the path it receives.
14) Post-mortem summary at end of report.

Methods breakdown in order of appearance:

public ArrayList<Node> copyNodeList()

This method was copied from DelivC. It accepts an array list of cities and returns a copy of it. The intent is to avoid affecting the original list of cities.

public ArrayList<Node> addLongEdges(ArrayList<Node> cities)

This method accepts an array list of cities, loops through the cities and, while the number of outgoing edges for each city is less than the number of cities, it loops through the cities again to determine whether there is an edge between the first loop of cities and the second loop of cities. If the city.findEdge(nextCity) != null, the edge count increases by one for each edge. If the edge is null, it creates a new temporary edge with distance of longEdge and adds the temporary edge to the outgoing edges.

This method evolved greatly over several attempts and reduced processing time from two minutes to less than a second. I believe the current time complexity is N^4. I am not sure what the original time complexity was.

public Node[] startToGoal(ArrayList<Node> path)

This method accepts an array list of cities and returns a new array with the start city at index 0 and the goal city in the last index. All other cities are added to the array in the order they appear within the original cities list.

public int calculateShortDistance(Node[] cities)

For files whose null edges dominate the path, this method accepts an array of cities and uses brute-force to return a sum all distances than 100000. It uses Local Search to search the current path for edges smaller than 100000, and then uses General Arc Constraint to add only the smaller edges.

public int calculateDistance(Node[] cities)

This method accepts an array of cities and returns the total distance in the current tour of cities.

public int findMaxEdge(Node[] cities)

This method accepts an array of cities and returns an index position of a city with a distance of 100000 as its previous edge.

public Node[] swapTwoCities(Node[] cities)

This method accepts an array of cities and swaps the index returned from findMaxEdge() with a random index between the start and goal cities. By using this method, the number of 100000 edges are reduced, and in most cases eliminated.

There have been many evolutions of the swapCities() method, starting with a simple statement in the for-loop to swap two random cities. I later created a the separate method. At one point I swapped up to ten cities at a time, but ran into the issue of random integers being identical and had an entire path of all the same city (oops!). Then I pared it back to four random integers and ensured they were all different. This worked fairly well, but did not address the edges of 100000 directly.

I wanted to continue to hone this method, or possibly reintroduce my previous swap method of four random cities, but ran out of time.

public int count(Node[] cities)

This method accepts an array of cities, squares its length and multiplies it by a number based on number of cities. The more cities, the higher the number to allow for more city swaps. It then returns the count that will be used in the for-loop of betterPath().

public void betterPath(Node[] cities)

This method accepts an array of cities, prints the original path and distance, and starts a for-loop to a maximum number of count(cities) to swap cities and calculate distances for each result. It prints the current distance and path each time a shorter path is found. This process works well for files whose edges are not dominated by 100000 distances.

Last-minute, I tried to create an if-statement to deal with paths whose edges of 100000 dominate. The goal was to use brute-force to calculate only the small edges. The if-statement said if c == count / 10 and findMaxEdge(cities) == longEdge, the distance = calculateShortDistance(cities). My for-loop, and possibly if-statement is not constructed accurately the if-statement it is never utilized. If I had more time, I would flesh the logic out and continue to work on output for files with dominating long edges.

It seems to me that although this method incorporates simulated annealing, it could better utilize heuristics to find a better path. An educated guess may help "guide" the process along and reduce iterations of small incremental changes for very large distances.

public String printCities(Node[] cities)

This method returns a string of abbreviations for cities on the current path.

<u>Post-mortum Summary</u>:

What I have learned about myself during this deliverable (and previous deliverables) is that my process for reasoning logic is solid, although the time it takes is slower than I would prefer. In general, I feel that the time it takes me to process information may be more than the time it takes other students in the class. I am hopeful my personal "time complexity" will improve with practice.

If time and energy were not limitations, I would:

- improve how count is calculated for the for-loop iterations to determine best path;
- introduce an additional method or two to swap cities in different ways; and
- fix the unused if-statement, or create another method that uses brute-force to ignore edges of 100000.

Overall, this has been a very fun deliverable to work on and I like that there are so many options to achieve the goal of finding a shorter path.