CS2040S: Data Structures and Algorithms

Problem Set 8

Due: Friday, April 14, 11:59pm

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). You can do so by leaving a comment at the start of your .java file. Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

When the travelling salesman meets the MST

The travelling salesman problem (often abbreviated as TSP) is one of the most famous problems in combinatorial optimization. Like many good problems, it is a simple question that is easy to state. You are given a set of cities $v_0, v_1, \ldots, v_n$. You know the distance[1] between any pair of cities. What is the fastest route that visits every city exactly once, and then returns to where you started?

See Figure 1 for an example of the travelling salesman problem, along with three possible solutions. A valid tour visits every city exactly once, and ends back at the first node—forming a cycle[2]. The goal is to find the shortest valid tour.

The travelling salesman problem has a reputation for being very hard: it is well-known to be NP-hard, meaning that there is no polynomial time algorithm that can find an optimal tour *unless* $P = NP$. In reality, though, the travelling salesman problem is not that difficult: we can efficiently calculate solutions that are *pretty good*. That is, for a given set of cities, we can find a tour that is approximately as good as the best tour[3].

In this problem set, you will be developing a solution for the travelling salesman problem based on minimum spanning trees. The algorithm you develop will be a 2-approximation scheme, i.e., it will guarantee that the tour it discovers is at most twice as long as the optimal tour.

[1] For today, we are focusing on the *Euclidean* travelling salesman problem where the distance between two cities is the Euclidean distance in the 2d-plane.

[2] Technically, a valid tour is known as a Hamiltonian Cycle.

[3] In fact, where the distances are Euclidean, you can get an approximation that is as close as you like! This is known as a PTAS: polynomial time approximation scheme.

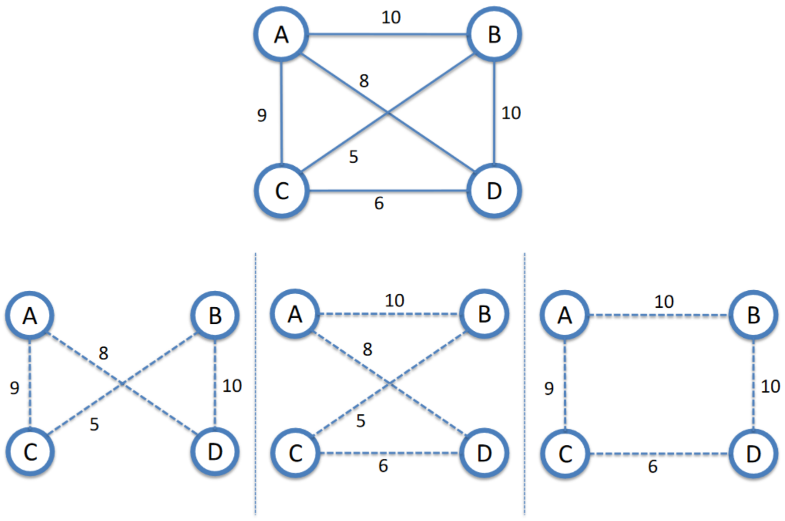

Figure 1: The top graph is an example of a TSP problem containing 4 cities $\{A, B, C, D\}$. On the bottom, with dashed lines, are three sample tours. The first (A, D, B, C, A) has cost 32, the second (A, D, C, B, A) has cost 29, and the third (A, B, D, C, A) has cost 35. Notice that each tour visits all four cities exactly once.

TSP-MST Algorithm. In describing the algorithm[4], we will think of the map as a graph $G = (V, E)$ where V is a set of n cities and E is a set of $n(n-1)/2$ undirected edges connecting every pair of cities. The algorithm consists of three steps:

1. Find a minimum spanning tree T of the graph G. (You can use any efficient algorithm for finding a minimum spanning tree.) Notice that the cost of the tree T is always less than the cost of the optimal tour (since if you remove any one edge from the optimal tour, it forms a spanning tree).

2. Perform a depth-first-search walk of the tree T. When you perform the depth-first-search, remember every time you visit a node. Every node in the graph appears at least twice in the DFS walk. (Every edge in the tree T is crossed exactly twice in the DFS walk.) Let $D = d_0, d_1, d_2, d_3, \ldots, d_{2n-1}$ be the $2n$ cities visited on the DFS treewalk. Notice that D has cost at most twice the optimal tour (since each edge is crossed twice), and visits every city at least once. It is not a valid tour, however, since cities are visited more than once.

3. Take short-cuts to avoid revisiting cities. For example, if you are in city d_i and have already visited city d_{i+1}, then skip city d_{i+1} and go directly to d_{i+2}. (If you have already visited d_{i+2}, then skip it and go on to the next city.) Since the distances satisfy the triangle inequality, these short-cuts can only decrease the length of the tour. Now you have a valid tour that is at most twice as long as the optimal tour.

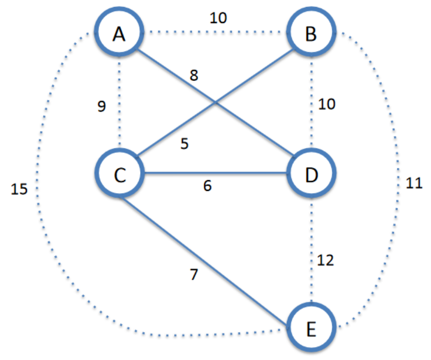

Figure 2: Here, we demonstrate how the algorithm works. In the figure, we have already calculated a minimum spanning tree, which consists of four edges: (A, D), (B, C), (C, D), and (C, E). (The dashed edges are not part of the spanning tree.) Next, starting at node A, we perform a depth-first-search treewalk, remembering every time we visit a node:

$$\mathbf{A} \to \mathbf{D} \to \mathbf{C} \to \mathbf{B} \to C \to \mathbf{E} \to C \to D \to A$$

We have put in bold the first time the DFS traversal visits a node, and we have used red to indicate later (unnecessary) visits. Finally, to devise the final tour, we skip the cities we have already visited, yielding the following tour:

$$\mathbf{A} \to \mathbf{D} \to \mathbf{C} \to \mathbf{B} \to \mathbf{E} \to A$$

Notice that the first city is visited again at the end of the tour, completing the cycle.

[4]Note that this 2-approximation algorithm only works if the problem instance satisfies the traingle inequality. In the context of this question, it means that the cost of going from city A to city C is less than the cost of going from city A to city B, then city B to city C.

TSPMap class. You are provided, as part of this problem, with the java class `TSPMap`. This class encapsulates the basic functionality for storing the locations of a set of cities, calculating the distance between the cities, and drawing a map of the cities. Note that you should not be modifying this class in your solution. The constructor `TSPMap(String fileName)` takes a map filename as an input and reads in all the points in the file. You can then access these points with the following methods:

- `int getCount()`: returns the number of points.
- `Point getPoint(int i)`: returns point i.
- `double pointDistance(int i, int j)`: calculates the distance between points i and j.

Each point can also store a single *link* to some other point. This link may be the parent edge in a spanning tree, or it might be the next point to visit on an MST tour. You can access the link using the following methods:

- `void setLink(int i, int j)`: sets a link from i to j, and redraws the screen immediately.
- `void setLink(int i, int j, boolean redraw)`: sets a link from i to j and redraws only if `redraw==true`.
- `void eraseLink(int i)`: erases the outgoing link from i and redraws the screen immediately.
- `void eraseLink(int i, boolean redraw)`: erases the outgoing link from i and redraws only if `redraw==true`.
- `int getLink(int i)`: returns the current outgoing link from i.

Whenever you modify the map, you may want to call the `redraw()` method to redraw the map.

The map class contains a static nested class `Point` which contains the information for a point. Notably, this class stores the x and y coordinates for a point, as well as the link. It includes functionality to calculate the distance between two points.

The main method has an example of how to use the map class. It reads in a file `hundredpoints.txt`, and then sets up the links: each point i is linked to point $i+1$, and the last point is linked back to point 0—creating a valid (if suboptimal) tour. Finally, it calls `redraw` to draw the tour.

Assignment. Your task is to implement a class `TSPGraph` that implements the `IApproximateTSP` interface. This interface supports four methods:

- `MST(TSPMap map)`: Calculate a minimum spanning tree for the points on the map. When the method returns, each point in the map should have its link set to its parent in the spanning tree.

  State the algorithm you have implemented and briefly explain the asymptotic running time in terms of n, where n is the number of cities.

- `TSP(TSPMap map)`: Calculate a good TSP tour using the algorithm described above: perform a depth-first search on a minimum spanning tree, using short-cuts to ensure that each point is visited only once. When the method returns, each point in the map should have its link set to the next point on the tour. Note that TSP is a standalone method, MST should be called within it.

  Briefly explain the asymptotic running time of this function in terms of n, where n is the number of cities.

- `isValidTour(TSPMap map)`: Returns true if the links on the map form a valid tour. (Note: a tour can be valid even if it is far from optimal.) This may be useful for debugging.

- `tourDistance(TSPMap map)`: If the links on the map form a valid tour, then this method returns the total length of that tour (i.e., the sum of all the edges on the tour). If the links do not form a valid tour, return -1. Note that the length of the tour includes the cost of returning back to the start when the tour is finished.

Note that a TSP implementation using a different algorithm than the one provided will only receive partial credit, even if it is a working solution.

You are also free to use the provided implementation of a priority queue `TreeMapPriorityQueue` as part of your solution. As its name suggests, this class implements the priority queue ADT with a `TreeMap` and includes the method `decreasePriority`, which Java's `PriorityQueue` does not provide.

You are **not** allowed to use any existing Java MST's libraries , e.g `PrimMinimumSpanningTree`.

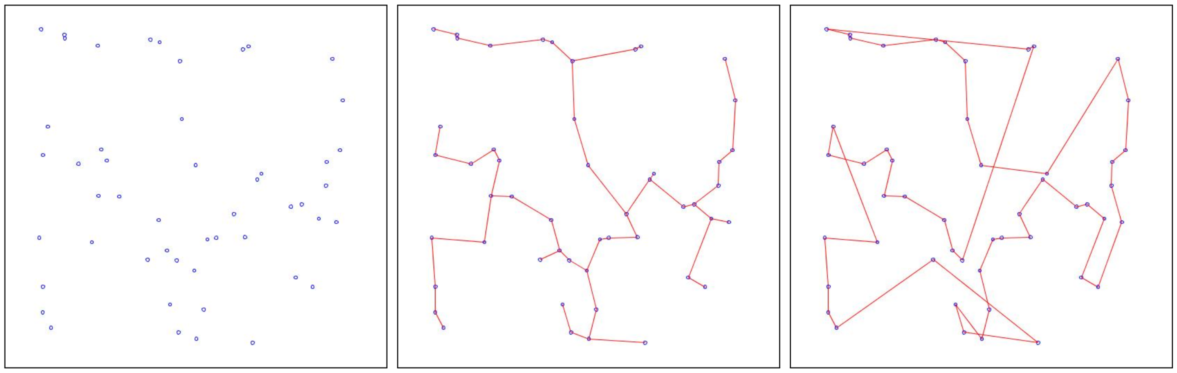

Figure 3: This figure contains an example using the file "fiftypoints.txt". The first image contains the fifty points. The second image depicts a minimum spanning tree. The third image depicts a valid tour for the travelling salesman problem.