CS2020: Data Structures and Algorithms (Accelerated)

Problem Set 7

Due: Friday, April 17, 11:59pm

Two problems. This week's problem set contains *two* problems, and you have a little less than two weeks to solve them. Please start early!

Graphs. Both problems this week are on graphs. The first is about shortest paths, and the second is about spanning trees.

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). Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 1. Mazes are Just Dynamite!

It's easy to get lost in life. This week we're going to develop a program to help us get unlost. Even better, we're going to give you some extra super-powers: the ability to break down the obstacles in your way, helping you to get to your destination even faster.

To start off, you are given a map of the maze you are lost in. Like most mazes, this maze you are in consists of rooms. Each of the rooms has doors to one or more other rooms. Your first job is to write a program that will explore the maze, as is, and discover the shortest path from your current location to a destination location. Simultaneously, your program will be expected to determine some other geographic information about the maze.

Your second job involves determining how best to use your superpower. You have been given the power to bypass some fixed number of walls. Please remember to specify your superpower:

- Able to walk through walls.
- Able to fly over walls.
- Strong enough to knock down walls.
- Possesses dynamite.
- Other.

For zero points of extra credit, explain how your superpowers work and how you acquired them (e.g., on Facebook). In any case, your job is to find the shortest path from the source to the destination, while bypassing no more than the allowed number of walls.

Preliminaries. We will provide you with some basic framework code for handling the maze. A maze consists of an $n \times n$ square grid of rooms. The rooms are numbered starting from the top left corner (which is $(0, 0)$). The first coordinate specifies the row number, and the second coordinate specifies the column number. For example, in a 5×5 maze, the bottom left corner is $(4, 0)$ and the top right corner is $(0, 4)$. Here is a pictorial example of a 5×5 maze:

```
   0 1 2 3 4
  ###########
0 #     # # #
  ### # # # #
1 # # #     #
  # ### ### #
2 #     #   #
  # ### # # #
3 # #   # # #
  # # # ### #
4 # # # #   #
  ###########
```

In this diagram, each wall is depicted using a hash symbol, i.e., `#`. There are five possible rooms in the first row, and six possible walls (including the left and right borders).

We provide you with two classes `Maze` and `Room` that represent the maze that your program will solve. The size of the maze is represented by the number of *rows* and *columns* in the maze (in the above example, both *rows* and *columns* are 5). The maze itself is represented by a matrix of rooms. You will be able to check if there exists a wall in the four directions of the room through the public methods `hasNorthWall()`, `hasSouthWall()`, `hasEastWall()` and `hasWestWall()`. On top of that, there is a public boolean attribute `onPath` that you can set (for printing of mazes).

The `Maze` class has a static method `readMaze(String fileName)` that reads in a maze from a text file and returns the maze object. We will provide several sample mazes for you to experiment with. We also provide a simplistic way of visualizing a maze through the static class `MazePrinter`. The static method `void printMaze(Maze maze)` of the `MazePrinter` class prints out a maze to the standard output[1].

In this problem set, you will implement two classes `MazeSolver` and `MazeSolverWithPower`, that will implement the provided interfaces `IMazeSolver` and `IMazeSolverWithPower` respectively.

Problem 1.a. The Average Coder

Joe the Average Coder is eager to solve this problem and implemented the class `MazeSolverNaive` (found in *MazeSolverNaive.java*). In particular, Joe implemented the `pathSearch` method that will return the **minimum** number of steps to get from a given starting coordinate to a target coordinate. He did so using the **Depth-First Search** traversal that he learned in his Algorithms and Data Structures class.

Take a look at Joe's code. Will his algorithm solve the shortest path in a maze problem correctly? Why or why not?

Problem 1.b. Exploring the Maze

Now implement the class `MazeSolve` that correcly implements the `IMazeSolver` interface. First, implement the method:

```
Integer pathSearch(int startRow, int startCol, int endRow, int endCol)
```

which searches for the shortest path from the specified start room to the specified end room. It should return an integer representing the minimum number of steps needed if a path is found, and `null` if no such path is available. When your search is complete, for every room R on the path, `R.onPath == true`; for every room R not on the path, `R.onPath==false`. If done correctly, if you execute `printMaze(Maze maze)` after your search is completed, it will draw the path correctly, as in Figure 1.

Next, implement the method: `Integer numReachable(int k)` that will return an integer indicating how many rooms there are such that the **minimum** number of steps required to reach it is k, based on your most recent `pathSearch` starting location. Your `numReachable` method should

[1]For an additional zero extra bonus points, implement a prettier graphical maze display and share it with the rest of the class.

```
###########
#PPPPP# # #
### #P# # #
# # #PPPPP#
# ### ###P#
#     #  P#
# ### # #P#
# #   # #P#
# # # ###P#
# # # #  P#
###########
```

Figure 1: Example maze that is solved (where you have no superpowers).

count the number of rooms reachable from the initial location for each possible number of steps. For example, how many rooms are such that the minimum path distance is 0? How many rooms are reachable with minimum 1 step? How many rooms are reachable with minimum 2 steps? etc. In the above example in Figure 1, the following holds:

```
0 Step: 1 Room
1 Steps: 1 Room
2 Steps: 2 Rooms
3 Steps: 1 Room
4 Steps: 2 Rooms
5 Steps: 4 Rooms
6 Steps: 5 Rooms
7 Steps: 5 Rooms
8 Steps: 3 Rooms
9 Steps: 1 Room
```

Notice that for each k, it outputs the number of rooms for which the minimum path distance is k exactly. If you calculate this for every initial starting point in the graph, then you can determine the *diameter* of the maze.

Problem 1.c. Maze Exploration for Real SuperPeople

In this part, as in the previous part, your job is to determine the shortest path from a specified source to a specified destination. However, you are also given the ability to bypass (demolish, jump over, or magically traverse) a fixed number of walls along the way. Of course, you will not be allowed to use your power to demolish the outer walls that surround your maze. You begin with a fixed amount of superpowers, and every time you demolish a wall, your power reduces by one. Your goal is to find the shortest path from the start to the end.

Create a new class `MazeSolverWithPower` that implements `IMazeSolverWithPower` which contains the overriden method:

`Integer pathSearch(int startRow, int startCol, int endRow, int endCol, int superpowers)`.

It should return an integer representing the minimum number of steps needed if a path is found, and `null` if no such path is available. As before, it should also update, for each room, the `onPath` attribute. (Notice that the `IMazeSolverWithPower` interface inherits from `IMazeSolver`, and hence must correctly implement all the methods from there as well.)

Hint: One strategy is to represent the state as a combination of the current room and the remaining amount of superpower. If you explore this graph of all possible states, you will visit all possible rooms, with all possible remaining amounts of superpower.

Problem 1.d. (Bonus) Sometimes you need to escape a maze, sometimes you need to build a maze[2]. Implement an algorithm for generating a maze. For an interesting review of different maze generation algorithms, see:

`http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap`

This blog entry summarizes 11 different methods for generating a maze. All of these techniques yield mazes that have only one route from start to finish (i.e., there are no cycles—they generate a tree). You might think about how to modify them to generate interesting graphs/mazes that have more than one possible solution.

Submit your maze generation algorithm, along with a discussion of the design decisions that you made, and your favorite maze that was generated by the algorithm (with no manual intervention).

Please share any fun mazes you come up with (whether by hand or by algorithm) on facebook!

[2]See, for example, *Inception* (2010).

Problem 2. 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[3] between any pair of cities. What is the fastest route that vists 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[4]. 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[5].

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.

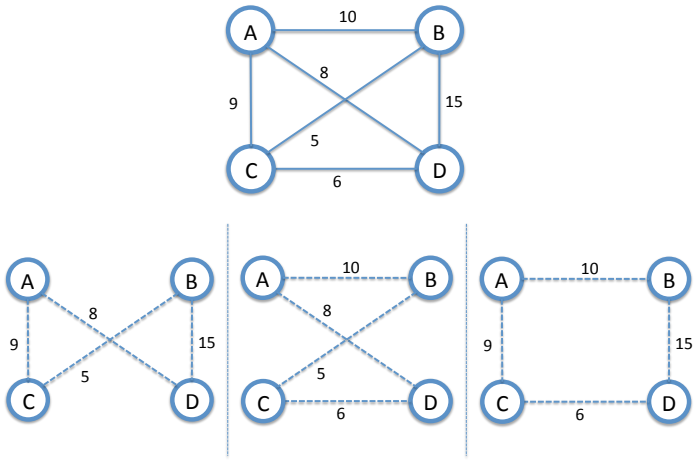

Figure 1: The top graph is an example of a TSP problem containing 4 cities $\{A, B, C, D\}$. On the bottom, with dashed lines, are three sample tours. The first (A, D, B, C, A) has cost 37, the second (A, D, C, B, A) has cost 29, and the third (A, B, D, C, A) has cost 40. Notice that each tour visits all four cities exactly once.

[3]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.

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

[5]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.

TSP-MST Algorithm. In describing the algorithm, 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.

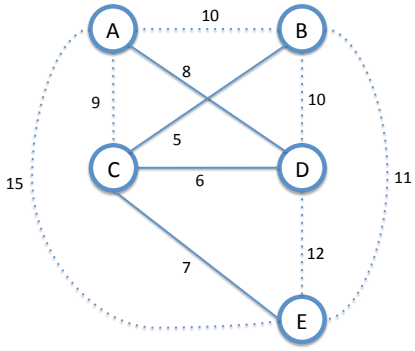

Figure 2: Here, we demonstrate how the algorithm works. In the figure, we have already calculated a minimum spanning tree, which consists of four edges: (A, D), (B, C), (C, D), and (C, E). (The dashed edges are not part of the spanning tree.) Next, starting at node A, we perform a depth-first-search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

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. 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)`: return point i.
- `double pointDistance(int i, int j)`: calculates the distance between points i and j.

Each point also has can 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 that `IApproximateTSP` interface. This interface supports five methods:

- `initialize(TSPMap map)`: Initializes your class with a new map containing points in a Euclidean plane.
- `MST()`: 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.
- `TSP()`: 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.
- `isValidTour()`: 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()`: 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). Note that the length of the tour includes the cost of returning back to the start when the tour is finished.

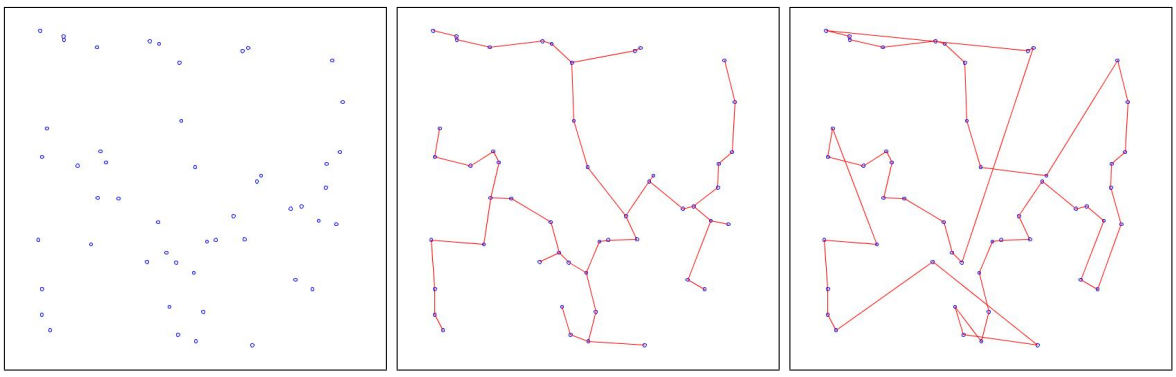

Figure 3: This figure contains an example using the file "fiftypoints.txt". The first image contains the fifty points. The second image depicts a minimum spanning tree. The third image depicts a valid tour for the travelling salesman problem.