Homework 7

Data Structures & Algorithms (Term III/2018–19)

built on 2019/06/25 at 22:26:09 ***due:*** *mon july 8 @ 11:59pm*

This assignment contains both a coding portion and a written portion. The aim of this assignment is to provide you with more experience solving algorithmic problems in Java and reasoning about code. This assignment requires a starter package, which can be downloaded from the course website.

READ THIS BEFORE YOU BEGIN:

- All your work will be handed in as a single zip file. Call this file `a7.zip`. You'll upload this to Canvas before the assignment is due.
- For the written part, you *must* typeset your answers and hand it in as a PDF file called `hw7.pdf`, which will go inside your zip file. No other format will be accepted. To typeset your homework, apart from Microsoft Word, there are LibreOffice and LaTeX, which we recommend. Note that a scan of your handwritten solution will not be accepted.
- Be sure to **disclose your collaborators in the PDF file.**
- For each task, save your work in a file as described in the task description.
- A script will process and grade your submission before any human being looks at it. *Do not use different function/file names.* The script is not as forgiving as we are.
- Use the Internet to help you learn: It's OK to look up syntax or how a function/class/method is used. It's **NOT** OK to look up how to solve a problem or answers to a problem. The goal here is to learn, not to just hand in an answer. If you wish to do that, just give us a link to the solution!
- You are encouraged to work with other students. However, **you must write up the solutions separately on your own and in your own words**. This also means you must not look at or copy someone else's code.
- Finally, the course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

Collaboration Policy: To facilitate cooperative learning, you are permitted to discuss homework questions with other students provided that the following whiteboard policy is respected. A discussion may take place at the whiteboard (or using scrap paper, etc.), but no one is allowed to take notes or record the discussion or what is written on the board. *The fact that you can recreate the solution from memory is taken as proof that you actually understood it, and you may actually be interviewed about your answers.*

Exercise 1: Union By Hand (2 points)

In class, we saw a number of union strategies that yield different tree heights and ultimately different running time. In this task, you will experiment with two union strategies.

The first strategy is union with depth control from class (union by size). The other strategy, called *path compression*, is an extension of the former strategy, where the only changes are to the `root` function. While `union` still joins the smaller tree into the larger tree, the new `root` function (only used in the path compression strategy) is as follows:

```
int root(int p) {
  if (id[p] != p)
      id[p] = root(id[p]);
  return id[p];
}
```

There are many interesting theoretical properties that path compression has. You'll see more later in Algorithms. For now, we'll study it empirically by looking at how they perform on a sequence of union operations.

Your Task: For each of these strategies, show the trees after performing the following sequence of union operations, in this exact order ($n = 10$ nodes):

`union`(1,0) `union`(2,3) `union`(0,2) `union`(5,4) `union`(6,7)
`union`(5,3) `union`(2,7) `union`(3,6)

Keep in mind, *only* the path compression strategy uses the new `root` function; the other strategy uses the original `root` function from class.

Exercise 2: Breadth-First Search Running Time (2 points)

Our breadth-first (BFS) implementation from class uses `HashSets` and `HashMaps` to keep track of progress. In this task, you'll reanalyze the running time of BFS if we use `TreeSets` and `TreeMaps` instead. To keep things simple, we'll work with the BFS code in Figure 1 that only reports vertices reachable from a source s. It has been rewritten from the version presented in class to make the operations more explicit.

You should know that the `TreeSet` implementation uses a balanced binary search tree (BST), so `add`, `contains`, and `remove`, unlike in a `HashSet`, take $O(\log S)$ per operation, where S is the size of the collection. Though `addAll` may perform other optimizations, for the purpose of this task, assume that `addAll(X)` simply repeatedly calls `add` on each element of X.

You should also assume that `UndirectedGraph G` is implemented as an adjacency table using `HashMaps`, so `G.adj` takes $O(1)$.

Exercise 3: A Tree, A Forest, or Just A Mess? (2 points)

For this task, save your code in `IsForest.java`

Remember that an undirected tree T is a simple, connected graph that has no cycle. Here, connectedness simply means any vertex in the graph can reach all the other vertices in the graph. A graph that can be partitioned into one or more trees that don't have common vertices are called a forest.

You will be working with an undirected, simple graph $G = (V, E)$. What is a little usual is that the edges are presented to you one by one as an `Iterable`, in an arbitrary order. You are to implement a function

```
public static int identifyTrees(int n, Iterable<Pair<Integer, Integer>> edges)
```

with the following specifications:

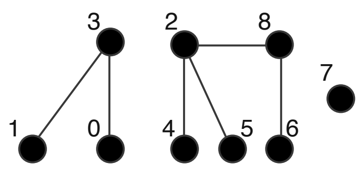
```
Set<Integer> nbrsExcluding(UndirectedGraph G, Set<Integer> vtxes, Set<Integer> excl) {
    Set<Integer> union = new TreeSet<>(); // not HashMap
    for (Integer src : vtxes) {
        for (Integer dst : G.adj(src))
          if (!excl.contains(dst)) { union.add(dst); }
    }
    return union;
}
Set<Integer> bfs(UndirectedGraph G, int s) {
    Set<Integer> frontier = new TreeSet<>(Arrays.asList(s));
    Set<Integer> visited = new TreeSet<>(Arrays.asList(s));

    while (!frontier.isEmpty()) {
        frontier = nbrsExcluding(G, frontier, visited);
        visited.addAll(frontier); // the i-th position is what's reached at i hops
    }

    return visited;
}
```

Figure 1: Basic Breadth-First Search Using `TreeSet`s.

- The input is the number of vertices in G (vertices are called $0, 1, \dots, n-1$) and an `Iterable` whose length is m, where each element is a `Pair` representing an undirected edge. Therefore, such a `Pair` e simply indicates that there is an edge between e.`first` and e.`second`.
- The function is to return the number of trees in the forest if G is a forest—or otherwise return 0 if G is not a forest.

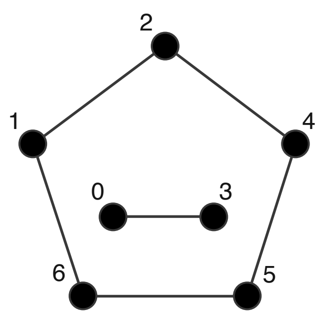
Expectations: Your algorithm must run in $O(m \log n + n)$ time or faster, and use at most $O(n)$ space. That is, it is *not* possible to store all the edges in your algorithm.

Example I:



The input is $n = 9$, where the iterable could give, for example, $(2,8), (1,3), (3,0), (5,2), (8,6), (2,4)$. The function will return 3 because G is a forest that is made up of *three* trees.

Example II:



The input is $n = 7$, where the iterable could give, for example, $(1,2), (0,3), (4,2), (4,5), (1,6), (6,5)$. The function will return 0 because G is not a forest; it is made up of a loop and a tree.

Exercise 4: HackerRank Problems (14 points)

For this task, save your code in `hackerrank.txt`

There are *seven* problems in this set. You **must** write your solutions in Java (1.8). You will hand them in electronically on the HackerRank website.

Important: You will write down your Hacker ID username in a file called `hackerrank.txt`, which you will submit as part of the assignment. This will be used to match you with your submission on HackerRank.

You can find your problems at

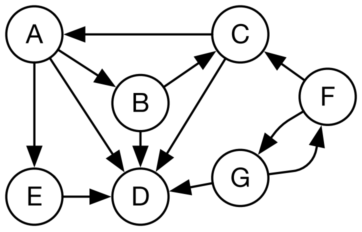
`https://www.hackerrank.com/muic-data-structures-t-318-assignment-7`

Exercise 5: Cycle Detection in Directed Graphs (2 points)

For this task, save your code in `Cycle.java`

Remember the cycle detection algorithm from class? In that algorithm, we used depth-first search (DFS) to determine whether an *undirected* graph has a cycle. Moreover, we found that the same algorithm doesn't work with directed graphs.

In this task, you'll come up with a new algorithm that works with directed graphs. Your input is a directed graph $G = (V, A)$ represented as a list of ordered pairs. For example, the following graph on left will be given to your program as the set on right:



```
[('A','B'),('A','E'),('A','D'),
('B','C'),('B', 'D'),
('C','A'),('C', 'D'),
('E','D'),
('F','C'),('F', 'G'),
('G','F')]
```

Your Task: To represent pairs in Java, we're providing a `Pair` class in `Pair.java`. You will implement a function

```
public static List<String> findCycle(ArrayList<Pair<String, String>> graph)
```

that takes as input a *directed* graph encoded as described and returns

- `null` if the graph contains no cycle;
- a list $[v_0, v_1, v_2, \dots, v_{k-1}]$ of distinct vertices, where $v_0 v_1 v_2 \dots v_{k-1} v_0$ is a cycle in the given graph.

Therefore, on the input graph above, one correct output (out of many possibilities) is `['A','B','C']`.

Performance Expectations: Your function must run in $O(m + n)$. Note that there may be vertices that aren't reachable from your favorite vertex.

Hints: Our discussion of DFS so far designates a vertex as visited once we visit it for the first time. You may wish to further separate visited vertices into

- Vertices that you have entered but have not exited; and
- Vertices that you have entered and have exited.

This can be done, for example, by keeping a `Map` instead of a `Set` for visited vertices, where the `Map` would store the status of the vertices.