Homework 5

Student: Haoyuan Du, Shijun Yang

Purpose:

- Learn to formulate problems graph theoretically.
- Understand and apply graph algorithms presented in class.
- Design and analyze your own graph algorithms.

General Homework Policies:

- This homework assignment is due by the deadline given in Canvas. Late assignments will not be accepted and will receive a 0.
- Submit **two** files through Canvas:
  1. The `pdf` for the written portion of the assignment (it should generated by modifying the the LaTeX file for this assignment).
  2. The completed `DirectedGraph.java` and `TwoWayDirectedGraph.java` files. Before submitting please zip all the java files together and submit one `.zip` file. See the *Additional Programming Instructions* section below for more detailed instructions.
- Submit both the `pdf` (it should generated by modifying the LaTeX file for this pdf) and the completed `DirectedGraph.java` and `TwoWayDirectedGraph.java` files through Canvas. Before submitting please zip all the files together and submit one `.zip` file. See the *Additional Programming Instructions* section below for more detailed instructions.
- You may choose to work with a partner on this assignment. If you choose to work with a partner, only one assignment should be submitted and you and your partner will both receive the same grade. Make sure to include your partner's name on the homework.
- If you choose to work with a partner, you must add yourself and your partner to an empty group in Canvas one week before the assignment deadline (this is available from the `People` tab in Canvas).
- If you choose to work with a partner, your assignment should be a true joint effort, equally created, and understood by both partners.
- You are not allowed to consult outside sources, other than the textbook, your notes, the Java API and the references linked from Canvas (i.e., no looking for answers on the internet).[1]
- Getting *ANY* solutions from the web, previous students etc. is *NOT* allowed.[1]
- You are not allowed to discuss this assignment with anyone except for your partner (if you have one) or the instructor.[1]
- Copying code from anywhere or anyone is not allowed (even previous code you have written). Allowing someone to copy your code is also considered cheating.[1]
- Your work will be graded on correctness and clarity. Write complete and precise answers and show all of your work.
- Questions marked (PRACTICE) will not be graded and do not need to be submitted. However it is highly recommended that you complete them.

Homework 5

Student: Haoyuan Du, Shijun Yang

[1] *See the section of the syllabus on academic dishonesty for more details.*

Homework Problems:

1. (7 points) A *vertex cover* of a graph $G = (V, E)$ is a subset of the vertices $S \subset V$ that includes at least one endpoint of every edge in E. You will use *dynamic programming* to write and implement a linear-time algorithm for the following task.

   Input: A rooted tree T.
   Output: The size of the smallest vertex cover of T.

   You should model your dynamic programming solution off of the solution to the independent set problem given in class. You will implement your solution by adding code to the `vCover` method in the `DirectedGraph.java` file.

   (a) First, come up with the subproblem and recurrence (this should be very similar to those used for the independent set problem). Don't forget the base case(s)! Add these right here in the pdf. You should include a justification for why the recurrence is correct.

   > **Solution:** Subproblem: Let VertexCover(n) be the size of smallest vertex cover of the subtree rooted at node n. If n is a leaf, then VertexCover(n) = 0.
   > Recurrence: VertexCover(n) = $min(1 + \sum_{k \in n.children} VertexCover(k), \#n.children + \sum_{g \in n.grandchildren} VertexCover(g))$.
   > Justification: We have two cases in this problem. Including node n or not including node n.
   > If we include node n, then we have all edges between n and n's children, but we still need to know the children's vCover value. So it's $1 + \sum_{k \in n.children} VertexCover(k)$.
   > If we don't include node n, then we have to include all n's children (because they are the only endpoints of edges from n to n's children). Also, we need to plus all vCover value of n's grandchildren. Combine them together, we get $\#n.children + \sum_{g \in n.grandchildren} VertexCover(g)$.
   > Finally, we choose the smaller answer between two cases. So this recurrence is correct.

   (b) Next, complete the `vCover` method using your subproblem and recurrence from part (a). The `DirectedGraph.java` file implements a basic directed graph using an adjacency list. A directed graph is represented by an ArrayList of `DirectedGraphNode`s. Each `DirectedGraphNode` has a data field `neighbors` which is a linked list of the node's neighbors. *For this assignment your code will only be run on rooted forests. Each tree will have it's edges directed downward (away from it's root).* In other words, a node's `neighbors` field will be a linked list of its children. The methods `createSimpleGraph`, `createBinaryTree`, and `createForest` in the file `DirectedGraphDriver.java` all create such rooted forests but for testing purposes you will likely want to write methods that create additional types of forests that meet these constraints. Later in the semester we will talk about how you can determine if a graph is a tree (or not). Note that currently the `isAcyclic` method always returns `true` so you should not test your code on graphs with cycles.

   (c) For *EXTRA CREDIT* you can also implement a recovery method that prints the vertices in the minimum size vertex cover by adding code to the `printCover` method in the

`DirectedGraph.java` file. To receive any credit this method must run in linear time and follow the class style guidelines.

Please read the documentation provided in the two files `DirectedGraph.java` and `DirectedGraphDriver.java` carefully!

2. (5 points) To get in shape, you have decided to start running to work. You want a route that goes entirely uphill and then entirely downhill, so that you can work up a sweat going uphill and then get a nice breeze at the end of your run as you run faster downhill. Your run will start at home and end at work and you are given as input a map detailing the roads with n intersections and m one-way road segments, each road segment is marked uphill or downhill. You can assume home is intersection 1 and work is intersection n, and the input is given by two sets: E_U representing the uphill edges and E_D representing the downhill edges, where each set is given in adjacency list representation.

   Here is an example with 5 vertices, where vertex 1 is home and vertex 5 is work. The input are the following arrays of linked lists:

   $E_U = 1 \to 2; 3 \to 4, 3 \to 5.$
   $E_D = 1 \to 3; 2 \to 3, 2 \to 4; 4 \to 5.$
   There is one uphill-downhill way to go from 1 to 5: $1 \to 2 \to 4 \to 5$.

   You will devise an algorithm that determines if there is a route (regardless of the length) which starts at home, only goes uphill, and then only goes downhill, and finishes at work. Your algorithm should have linear $O(n + m)$ running time. You will implement your solution by adding code to the `isValidUphillDownhillPath` method within the `TwoWayDirectedGraph.java` file.

   Note that if your code does not implement a solution with the required running time $O(n+m)$ you will lose credit regardless of whether the test cases execute correctly.

3. (6 points) The police department in the city of St. Thomasville has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However, the city elections are coming up soon, and there is just enough time to run a *linear-time* algorithm.

   (a) Formulate this problem graph-theoretically (this should involve clearly stating what the vertices and edges of your graph are) and explain why it can indeed be solved in linear time.

   > **Solution:** So we can treat each intersection as a vertex in our graph and each street as a directed edge in graph. Each directed edge connects two vertices. To solve the question above, in fact, we are solving if this graph is a strongly connected component as a whole.
   > So we can use the SCC algorithm from the ppt start at page 22, which running time is linear (O(n+m), n is #vertices, m is #edges). Then, we check if the number of strongly connected component equals 1. If it's only one, then the answer is true. Otherwise, if it's more than one, false.

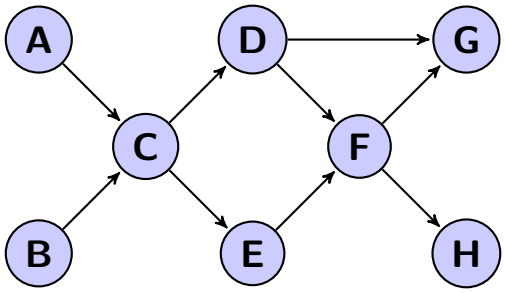
   (b) Suppose it now turns out that the mayor's original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-way streets, then no matter

where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how it too can be checked in linear time.

Solution: Firstly, we run the same SCC algorithm in Q3a which can give us some labeled SCCs. The running time is O(n+m), n is #vertices, m is #edges.
Then, if we start driving from town hall and can never go back, the only possibility is that we drive into a sink SCC at last. To prevent that, the SCC included town hall must be a sink SCC so that we never drive over this SCC's border and can return to the town hall finally.
Thus, we can run an algorithm that checking each vertices in town hall's SCC to see if anyone has an edge that is pointing to another vertex belonging to other SCCs. Since we have already computed the SCC numbers for each vertex, we just need to check if each pair of vertices (one is the vertex in town hall SCC, the other one is the related ending point) have same SCC number. This algorithm will take O(n+m).
Overall, this takes O(n+m) time which is linear.

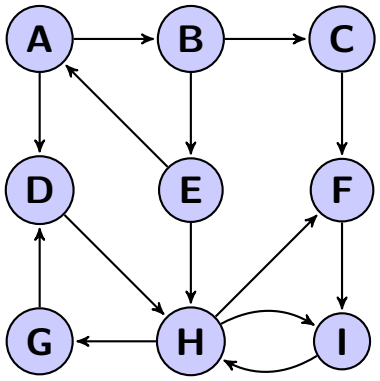
4. **(PRACTICE)** Given a directed graph $G = (V, E)$, we want to create another directed graph $G' = (V, E')$ such that: G' has the same strongly connected components as G. G' has the same component graph as G. ¥E is as small as possible (i.e. minimize $|E'|$). Describe a fast algorithm to compute G'. Prove that your algorithm is correct and justify its running time. Faster (in O notation) and correct is worth more credit.
Hint: Recall that a *connected* graph with $n-1$ edges is a tree. Note that two vertices in a tree cannot be in the same SCC - because there is a unique path between any two vertices in a tree and that path can only go in at most one direction (it might not be a directed path at all).

5. **(PRACTICE)** Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.



   (a) Indicate the pre and post numbers of the nodes.
   (b) What are the sources and sinks of the graph?
   (c) What topological ordering is found by the algorithm?
   (d) How many topological orderings does this graph have? List them all.

6. **(PRACTICE)** Run the strongly connected components algorithm on the following directed graph G. When doing DFS on G^R : whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.



(a) In what order are the strongly connected components (SCCs) found?

(b) Which are source SCCs and which are sink SCCs?

(c) Draw the "metagraph" (each meta-node is a SCC of G).

(d) What is the minimum number of edges you must add to this graph to make it strongly connected?

7. **(PRACTICE)** Give an efficient algorithm that takes as input a directed graph G and determines whether there is a vertex from which all other vertices are reachable. Prove your algorithm is correct and compute its running time. Faster (in O notation) and correct is worth more credit. (Note: The input graph G may not be acyclic.)

Additional Programming Instructions:

Note that your code will be automatically run on a standard set of test cases. In order to ensure that you do not lose points, follow the instructions below.

- Your code must compile without any errors using the version of Java on the lab computers. If your code does not compile you will not receive any points for the assignment.
- Do not modify any of the methods signatures (i.e. name, return type or input type). Note that you are always welcome (and encouraged) to add additional methods but these will not be run directly by the test code.
- You are not allowed to use packages (e.g. no statement `package ...` at the top of your file).
- No extra folders or files in your submission. Zip up only the files you need to submit not the folder they are in.
- Your solution should not print anything unless explicitly instructed to.

Grading Criteria:

Your work will be graded on both correctness **and clarity**. Write complete and precise answers and show all of your work. Your pseudo-code and proofs should follow the guidelines posted on Canvas and discussed in class.

Evaluation Rubric for Problem 1:

Component	Description
Subproblem and Recurrence (2 pt)	Make sure to state your subproblem definition and recurrence (including the base case(s)). You must also include a justification for why the recurrence is correct.
Style & Documentation (1 pts)	I'll be watching for style issues as well as correct output. See the syllabus for some general style guidelines (e.g. your code should be well-documented).
Correct Output on Test Cases* (4 pt)	Your code will be run on a standard set of test cases. Make sure to test your code thoroughly!
Extra Credit (1 pt)	Your `printCover` method will be run on a standard set of test cases. To receive any extra points this method must run in linear time and follow the class style guidelines.

*Note that if your code does not run in linear time $O(n)$ you will lose at least 2 points!

Evaluation Rubric for Problem 2.

Component	Requirement for Full Credit
Style & Documentation (1 pt)	I'll be watching for style issues as well as correct output. See the syllabus for some general style guidelines (e.g. your code should be well-documented).
Correct Output on Test Cases* (4 pts)	Your `isValidUphillDownhillPath` method will be run on a standard set of test cases. Make sure to test your code thoroughly!

*Note that if your code does not run in linear time $O(n + m)$ you will lose at least 2 points.

Evaluation Rubric for Problem 3(a) and (b).

Component	Requirement for Full Credit
Graph Formulation (1pt)	The answer clearly explains how this problems can be represented as a graph. It clearly states what the vertices and edges are. The explanation is in full sentences, and is easily understood by any student in the class.
Linear Time Algorithm (2pt)	Your solution includes a logical argument explaining why the solution can be found in linear time. This may involve giving pseudo-code and/or referring to an algorithm presented in class. The explanation is clear, correct and complete. It includes full sentences and is easily understood by any student in the class.