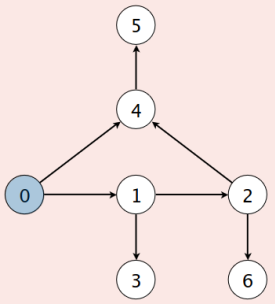
Precept Outline
- Review of Lectures 15 and 16:
  - Graphs and Digraphs
  - Graph search: DFS and BFS

Relevant Book Sections
- Book chapters: 4.1 and 4.2

A. Review: Graphs, Digraphs and Graph Search

Your preceptor will briefly review key points of this week's lectures. They may use the following graph to trace examples:



B. Assignment Overview: Wordnet

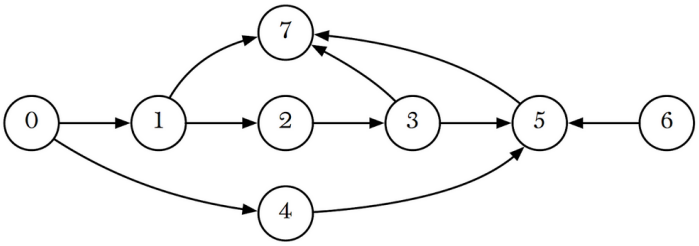
Your preceptor will introduce and give an overview of your fifth assignment. Please don't hesitate to ask questions!

Summary of the assignment.

- Implement a `ShortestCommonAncestor` class to calculate the shortest ancestral path between either two vertices or two sets of vertices in a rooted DAG.
  - A **rooted DAG** is a digraph that is acyclic and has one vertex - the root - that is an ancestor of every other vertex.
  - An **ancestral path** between two vertices v and w is a directed path from v to a common ancestor x, together with a directed path from w to the same ancestor x.
  - A **shortest ancestral path** is an ancestral path of minimum total length.
  - A **shortest ancestral path of two subsets of vertices** A and B is a shortest ancestral path among all pairs of vertices v and w, with v in A and w in B.
- Implement a `WordNet` *immutable* class for managing synsets and hypernyms, supporting operations to check if a word is a noun, finding the shortest common ancestor (using `ShortestCommonAncestor`), and computing the distance between nouns. The definition of all of these is in the assignment specification, but the summary is that you will be given a collection of terms and definitions and build a rooted DAG with them.
- Implement an `Outcast` class to determine the least related noun among a list of `WordNet` nouns, based on the sum of distances to other nouns.
- There is also a `readme.txt` where you will describe the best case and worst case order of growth of your solution, as well as answer a theoretical question.

C. Shortest Common Ancestor

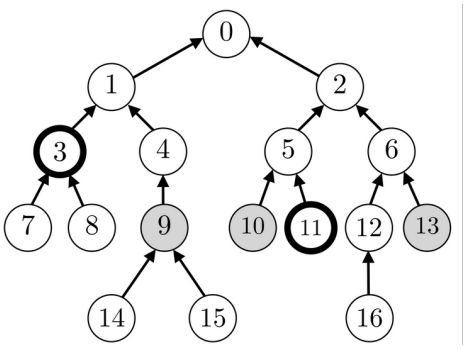
In the following digraph, compute the (smallest) sum of the path lengths from vertices 1 and 4 to all common ancestors. Find the shortest common ancestor and the shortest ancestral path (definitions are in the previous section).



Describe an algorithm for calculating the shortest common ancestor of two vertices v and w. Your algorithm should run in linear time (i.e., proportional to $V + E$).

How would your algorithm differ if we are interested in the shortest ancestral path between two sets of vertices A and B, rather than two vertices? (The shortest path between two sets is the shortest among any vertex u in A and any vertex w in B.

In the following example, $A = \{3, 11\}$ and $B = \{9, 10, 13\}$. The shortest common ancestor is 5 (between 10 and 11).



D. Rooted DAGs

In order to compute shortest common ancestors, we assumed the graph has a particular structure: it is directed and acyclic (i.e., is a DAG) and is also rooted (i.e., there is a unique common ancestor to all nodes). We will see now how to check for both of these things. Once again, our goal is to achieve an $O(V + E)$-time algorithm.

Describe an algorithm that uses graph search to detect if a directed graph has a cycle.

Describe an algorithm that detects if a DAG is rooted.

Assignment tips:

- `algs4` has `BreadthFirstDirectedPaths`, which you can use. That is, you don't have to re-implement BFS.
- `BreadthFirstDirectedPaths` has constructors that take either a single source vertex or a set of vertices.
- Input files (`synsets.txt` and `hypernyms.txt`) should not be read more than once!
- Test with multiple input files (see checklist and `wordnet.zip`). Create your own small test files to test certain graph structures and avoid working with large files (which can be slow).
- For extra credit, `BreadthFirstDirectedPaths` isn't enough and you have to re-implement BFS. However, you should first make sure that their code passes all test cases using `BreadthFirstDirectedPaths` before attempting to re-implement BFS for extra credit.