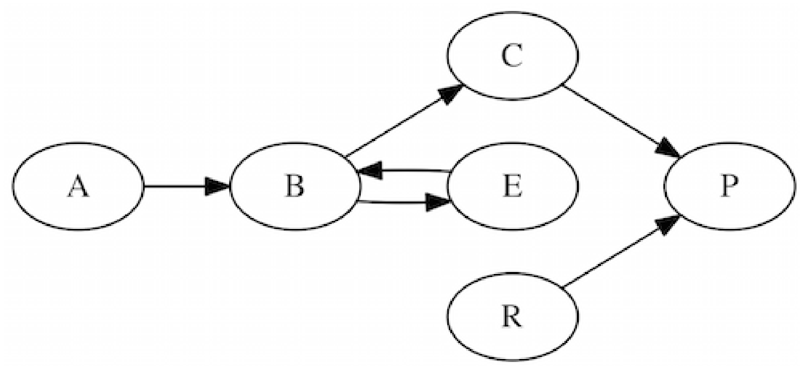


1 Graph Representations

For the graph above, draw the adjacency list and adjacency matrix representation.

2 DFS and BFS

Give the DFS Preorder, DFS Postorder, and BFS order of the graph starting from vertex A. Whenever there is a choice of which node to visit next, visit nodes in alphabetical order.

```
DFS Preorder: ABCPE
DFS Postorder: PCEBA
BFS Order: ABCEP
```

3 Topological Sorting

Which edge would we need to remove so that there exists a topological sort for the graph above? Give a valid topological sort (Hint: Use DFS Postorder).

```
We'd need to remove either the edge from B to E or E to B.
```

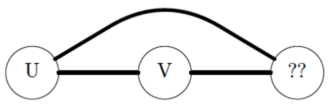
Supposing we remove the edge from E to B, we can find the DFS Postorder of the remaining graph from A and then R (or R then A, either way works).

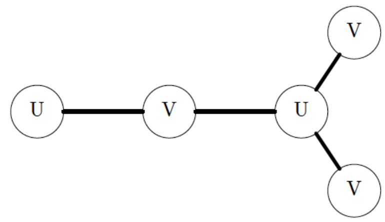
If we remove the edge from E to B, then the DFS Postorder from A is the same as above: PCEBA. We then find the posvisit order of R. This gives us an overall postorder of PCEBAR.

A valid topological ordering is then RABECP.

4 Graph Algorithm Design: Bipartite Graphs

An undirected graph is said to be bipartite if all of its vertices can be divided into two disjoint sets U and V such that every edge connects an item in U to an item in V. For example, the graph on the left is bipartite, whereas on the graph on the left is not. Provide an algorithm which determines whether or not a graph is bipartite. What is the runtime of your algorithm?






```
To solve this problem, we simply run a special version of DFS or BFS from any
    vertex. This special version marks the start vertex with a U, then each
    of its children with a V, and each of their children with a U, and so
    forth. If any vertex already has a U and the visited vertex has a V (or
    vice-versa), then the graph is not bipartite.

If the graph is not connected, we repeat this process for each connected
    component.

If the algorithm completes, marking every vertex in the graph, then it is
    bipartite.
```

5 Extra Algorithm Design: Shortest Directed Cycles

Provide an algorithm that finds the shortest directed cycle in a graph in O(EV) time and O(E) space.

The key realization here is that the shortest directed cycle involving a particular source vertex is just some shortest path plus one edge back to s. Using this knowledge, we can create a shortestCycleFromSource(s) subroutine. This subroutine first runs BFS on s, then checks every edge in the graph to see if it points at s. For each such edge originating at vertex v, it computes the cycle length by adding one to distTo(x) (which was computed by BFS).

This subroutine takes $O(E+V)$ time because it is BFS. To find the shortest cycle in the entire graph, we simply call shortestCycleFromSource() for each vertex, resulting in an $V*O(E+V) = O(EV+V^2)$ runtime. Since $E > V$, this is just $O(EV)$.

6 Extra: Daniel's Dare for the Daring

Master brogrammer, Edwin Edgehands decides to try his hand at implementing the Depth First traversal algorithm. Here is Edgehands' pseudocode:

```
Create a new Stack of Vertices
        Push the start vertex and mark it
        While the fringe is not empty:
                pop a vertex off the fringe and visit it
```

```
        for each neighbor of the vertex:
                if neighbor not marked:
                        push neighbor onto the fringe
                        mark neighbor
```

Your TA, Joshua Shrug claims that the above traversal isn't quite DFS. Give an example graph where it may not traverse in DFS order.