Graphs

CS61B Spring 2018 Discussion 11

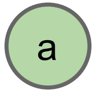
Administrivia

- Midterm II scores are out (@3262)
    - Regrade requests will open on **April 4th, at noon**
    - Regrade requests will close on **April 9th, at noon**
    - Special regrade instructions for FLIGHT
- Algorithm design problems (@3259)
- HW 4 due 4/4
- Post Midterm 2 one-on-ones (@3258)

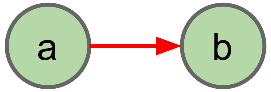
Graph Basics

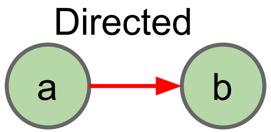
Graphs are data structures that have 2 components:

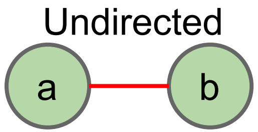
1. **Nodes (or vertices)**



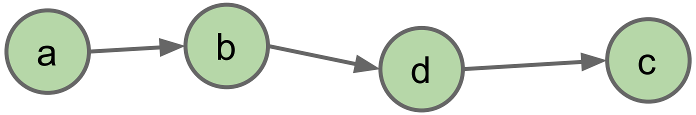
2. **Edges**



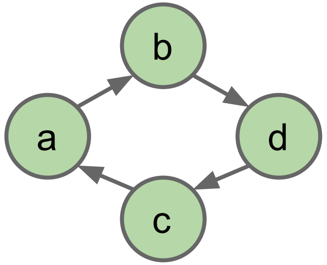


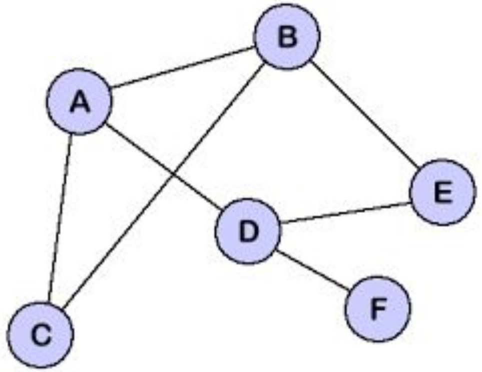


Path

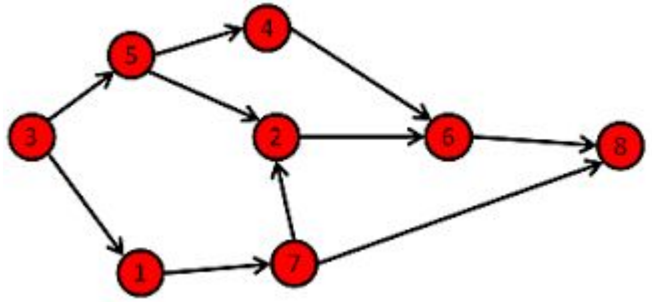


Cycle

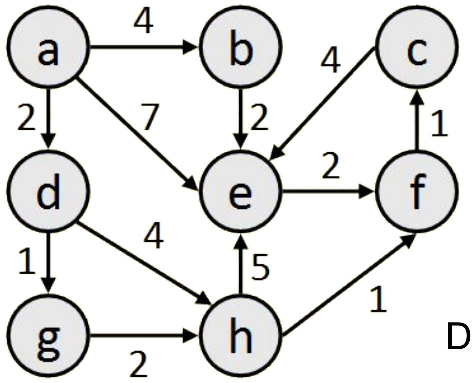




Undirected



Directed

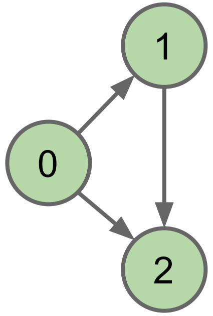


Directed with edge weights

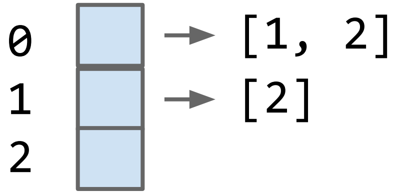
How to represent a graph

1. Adjacency Matrix

	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0



2. Adjacency list



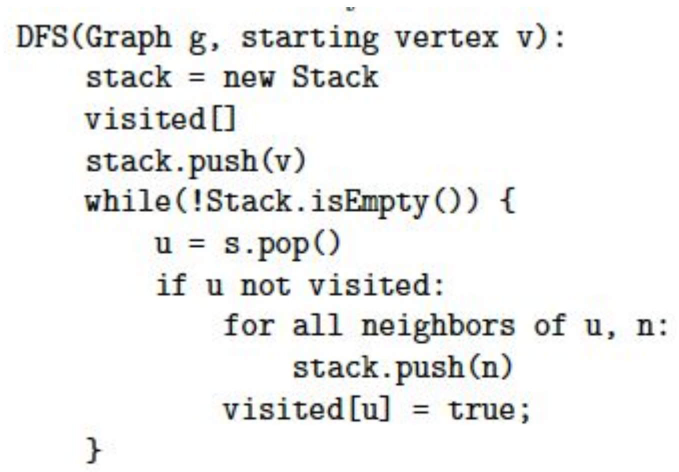
3. List of edges: (0, 1), (1, 2), (0, 2)

Runtimes

idea	addEdge(s, t)	for(w : adj(v))	printgraph()	hasEdge(s, t)	space used
adjacency matrix	Θ(1)	Θ(V)	Θ(V^2)	Θ(1)	Θ(V^2)
list of edges	Θ(1)	Θ(E)	Θ(E)	Θ(E)	Θ(E)
adjacency list	Θ(1)	Θ(1) to Θ(V)	Θ(V+E)	Θ(degree(v))	Θ(E+V)

Graph Traversals: DFS

DFS traversal of a graph is very similar to DFS traversal of a tree. The only difference is, since graphs can have cycles, you need to prevent your function from visiting nodes that have already been visited in an effort to avoid infinite loops.



```
DFS(Graph g, starting vertex v):
    stack = new Stack
    visited[]
    stack.push(v)
    while(!Stack.isEmpty()) {
        u = s.pop()
        if u not visited:
            for all neighbors of u, n:
                stack.push(n)
            visited[u] = true;
    }
```

PreOrder: Process the node when you first visit it

PostOrder: Process the node the last time you visit it.

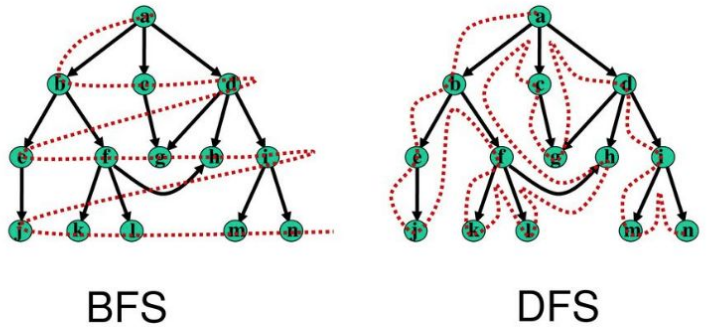
Graph Traversals: BFS

BFS differs from BFS in that instead of exploring a path all the way to the end, it takes a layer-by-layer approach

BFS pseudocode is exactly the same as DFS except with a Queue instead of a stack!

This is actually really cool, because it means we can recycle the entire algorithm but get a completely different traversal!

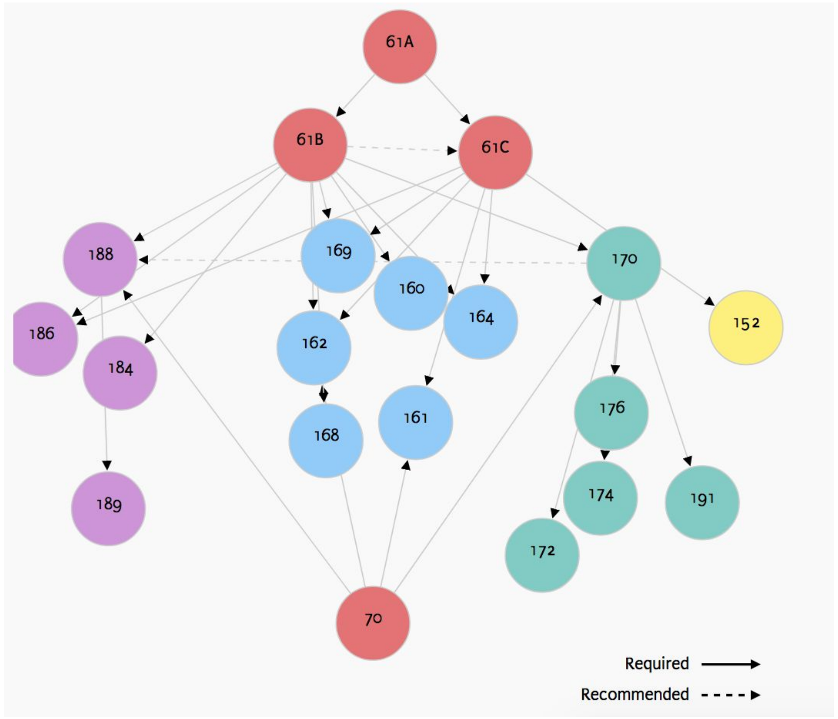
BFS also has the cool feature of finding the shortest path from one node to all other nodes in terms of number of edges.



Topological Sort

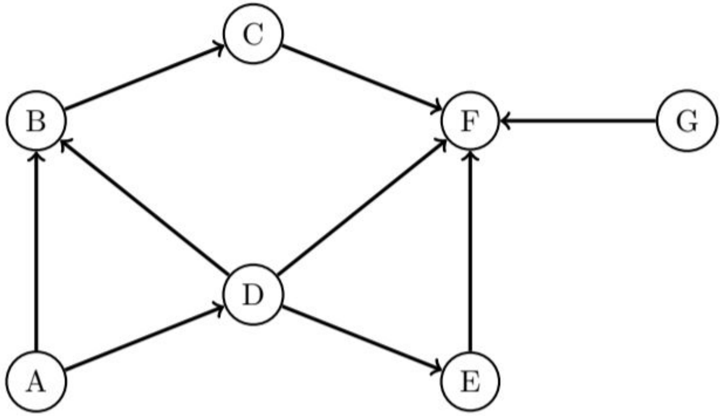
- Topological sort is a sort such that for each directed edge, u to v, u comes before v in the sort.
- Analogy: an edge from Class A to Class B means A is a prereq of B. A topological sort is an ordering of classes to take such that when you take a class, you've already taken the prereqs.
- How do we find the topological sort step by step ?
    - 1. Find a node with no incoming edges (in-degree of 0). These nodes are referred to as “sources". The first vertex in a topological sort MUST be a source!
    - 2. Process the vertex, then decrease the in-degree of its neighbors by 1. You can do this by either removing the node from the graph altogether (destructive) or manually decreasing the in-degrees of its neighbors (non-destructive).
    - 3. Mark the node as visited or processed
    - 4. Repeat starting from step 1 until all nodes have been visited.

Berkeley CS courses Topological Sort



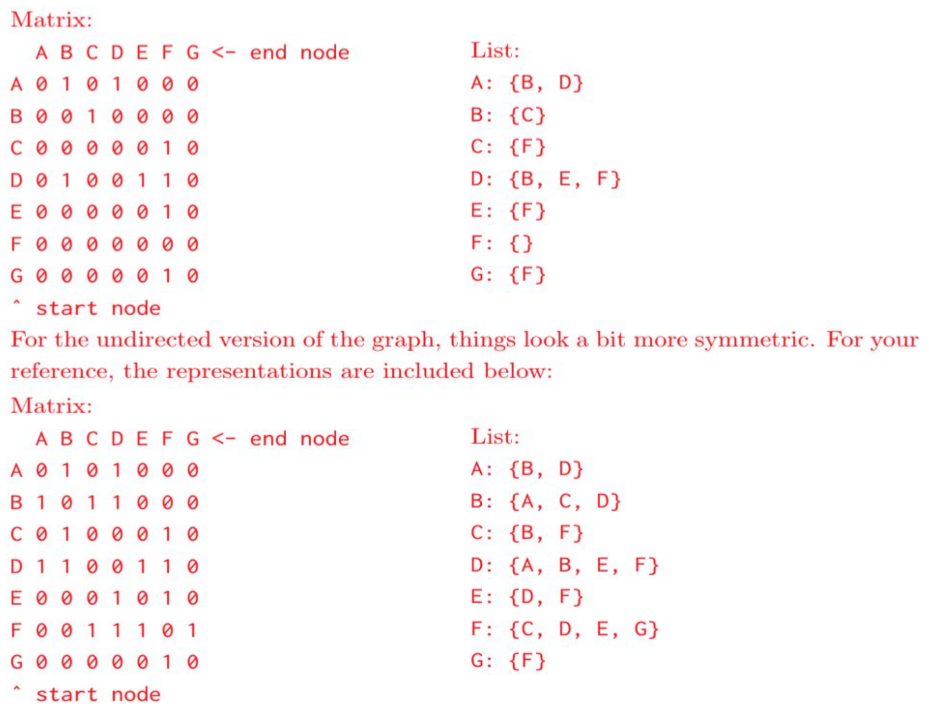
Give a valid topological sort for a student who is on the software track (light blue) given that they would need to take all of the red and blue classes.

Problem 1.1



Write the graph above as an adjacency matrix, then as an adjacency list. What would be different if the graph were undirected instead?

Problem 1.1 SOLUTION



Matrix:

```
  A B C D E F G <- end node
A 0 1 0 1 0 0 0
B 0 0 1 0 0 0 0
C 0 0 0 0 0 1 0
D 0 1 0 0 1 1 0
E 0 0 0 0 0 1 0
F 0 0 0 0 0 0 0
G 0 0 0 0 0 1 0
^ start node
```

List:

```
A: {B, D}
B: {C}
C: {F}
D: {B, E, F}
E: {F}
F: {}
G: {F}
```

For the undirected version of the graph, things look a bit more symmetric. For your reference, the representations are included below:

Matrix:

```
  A B C D E F G <- end node
A 0 1 0 1 0 0 0
B 1 0 1 1 0 0 0
C 0 1 0 0 0 1 0
D 1 1 0 0 1 1 0
E 0 0 0 1 0 1 0
F 0 0 1 1 1 0 1
G 0 0 0 0 0 1 0
^ start node
```

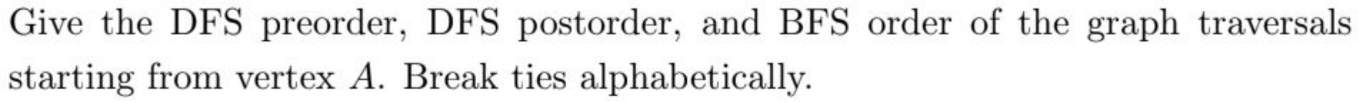
List:

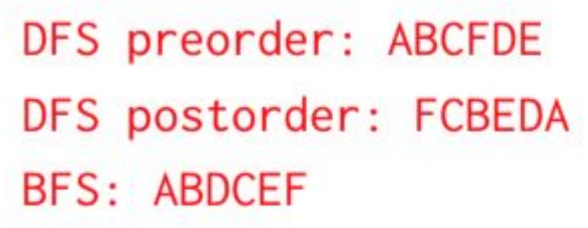
```
A: {B, D}
B: {A, C, D}
C: {B, F}
D: {A, B, E, F}
E: {D, F}
F: {C, D, E, G}
G: {F}
```

Problem 1.2

Give the DFS preorder, DFS postorder, and BFS order of the graph traversals starting from vertex A. Break ties alphabetically.

Problem 1.2 SOLUTION


Give the DFS preorder, DFS postorder, and BFS order of the graph traversals starting from vertex A. Break ties alphabetically.


DFS preorder: ABCFDE

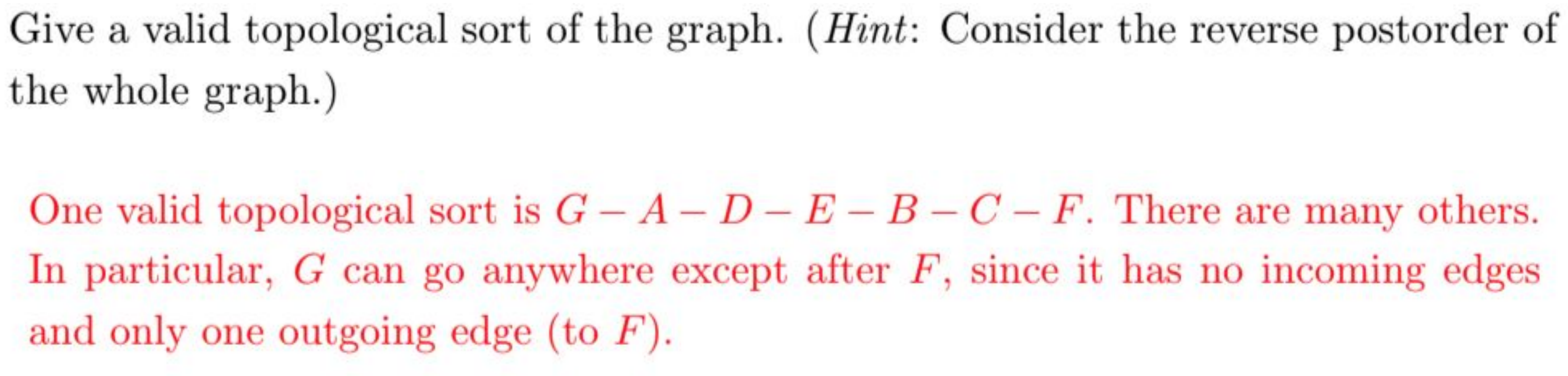
DFS postorder: FCBEDA

BFS: ABDCEF

Problem 1.3

Give a valid topological sort of the graph. (*Hint*: Consider the reverse postorder of the whole graph.)

Problem 1.3 SOLUTION

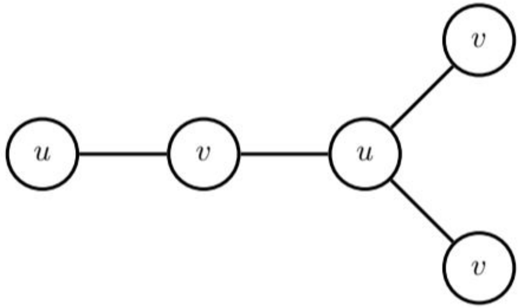


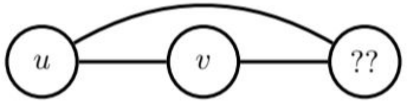
Give a valid topological sort of the graph. (*Hint*: Consider the reverse postorder of the whole graph.)

One valid topological sort is $G - A - D - E - B - C - F$. There are many others. In particular, G can go anywhere except after F, since it has no incoming edges and only one outgoing edge (to F).

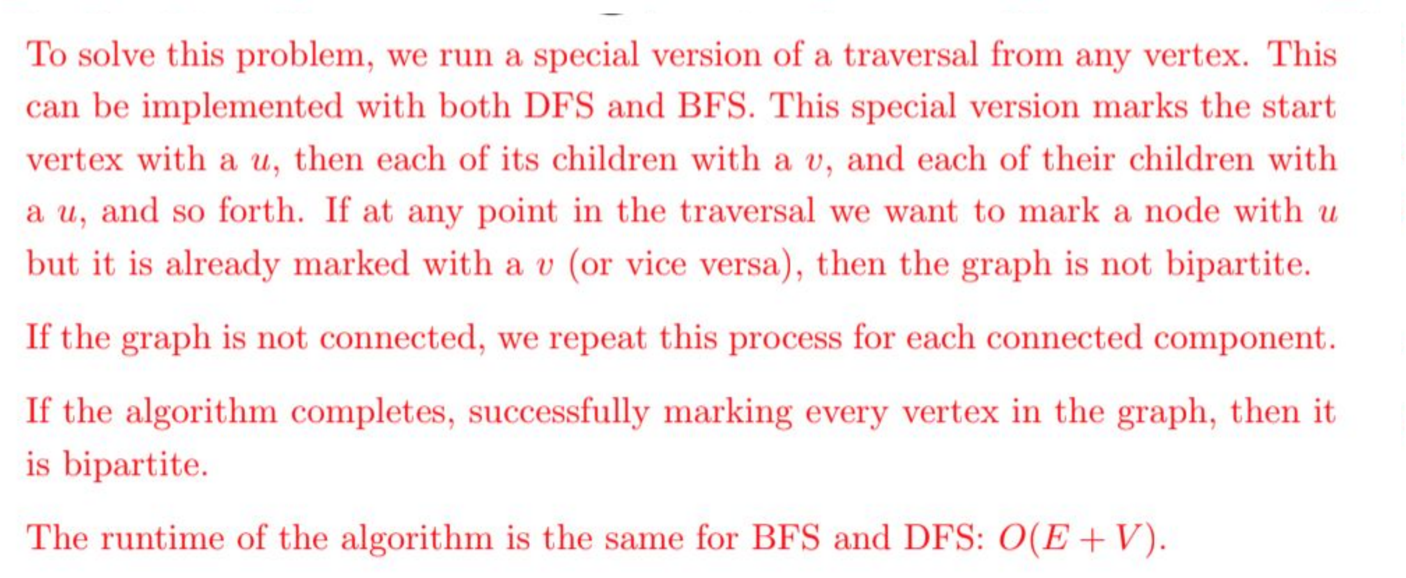
Problem 2.1

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 below, the graph on the left is bipartite, whereas on the graph on the right is not. Provide an algorithm which determines whether or not a graph is bipartite. What is the runtime of your algorithm?





Problem 2.1 SOLUTION



To solve this problem, we run a special version of a traversal from any vertex. This can be implemented with both DFS and BFS. 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 at any point in the traversal we want to mark a node with u but it is already marked with 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.

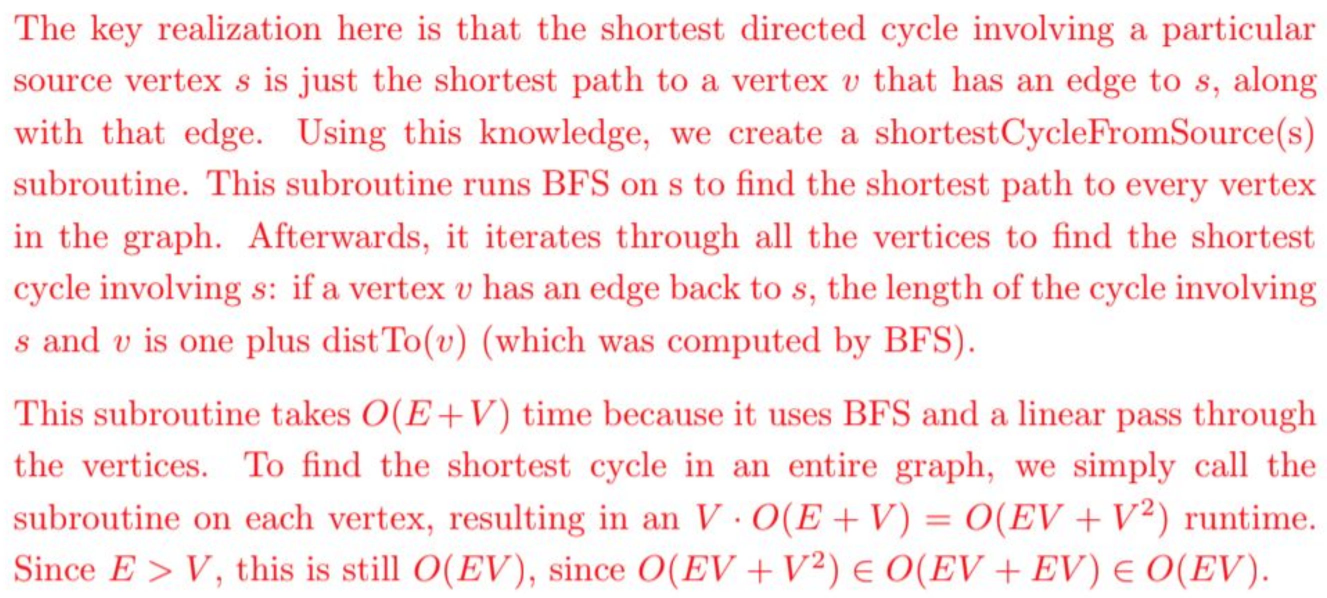
If the algorithm completes, successfully marking every vertex in the graph, then it is bipartite.

The runtime of the algorithm is the same for BFS and DFS: $O(E + V)$.

Problem 2.2

Provide an algorithm that finds the shortest cycle (in terms of the number of edges used) in a directed graph in $O(EV)$ time and $O(E)$ space, assuming $E > V$.

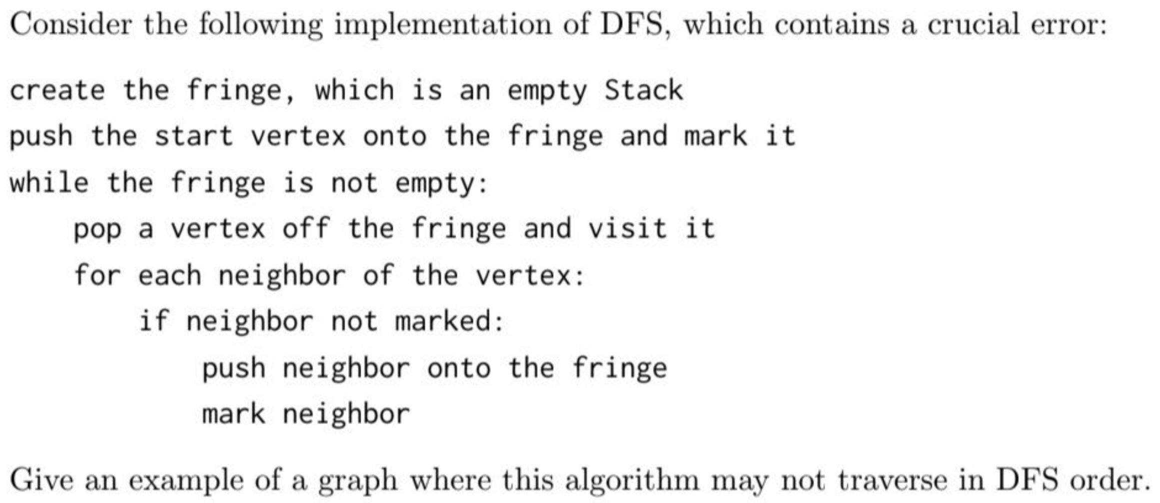
Problem 2.2 SOLUTION



The key realization here is that the shortest directed cycle involving a particular source vertex s is just the shortest path to a vertex v that has an edge to s, along with that edge. Using this knowledge, we create a shortestCycleFromSource(s) subroutine. This subroutine runs BFS on s to find the shortest path to every vertex in the graph. Afterwards, it iterates through all the vertices to find the shortest cycle involving s: if a vertex v has an edge back to s, the length of the cycle involving s and v is one plus distTo(v) (which was computed by BFS).

This subroutine takes $O(E+V)$ time because it uses BFS and a linear pass through the vertices. To find the shortest cycle in an entire graph, we simply call the subroutine on each vertex, resulting in an $V \cdot O(E+V) = O(EV+V^2)$ runtime. Since $E > V$, this is still $O(EV)$, since $O(EV+V^2) \in O(EV+EV) \in O(EV)$.

Problem 2.3

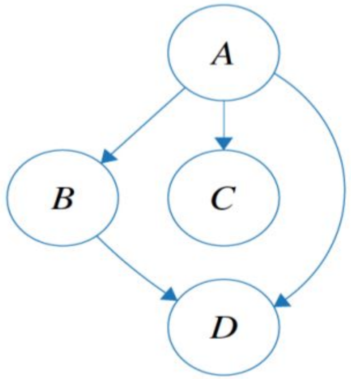


Consider the following implementation of DFS, which contains a crucial error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe 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
```

Give an example of a graph where this algorithm may not traverse in DFS order.

Problem 2.3 SOLUTION



For the graph above, it's possible to visit in the order $A - B - C - D$ (which is not depth-first) because D won't be put into the fringe after visiting B, since it's already been marked after visiting A. One should only mark nodes when they have actually been visited; in this example, we mark them before we visit them, as we put them into the fringe.