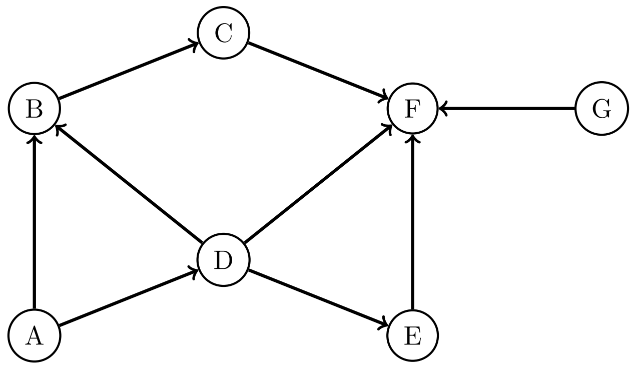
CS 61B Spring 2018 — Graphs

Discussion 11: April 3, 2018

Graphs



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?

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}
```

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

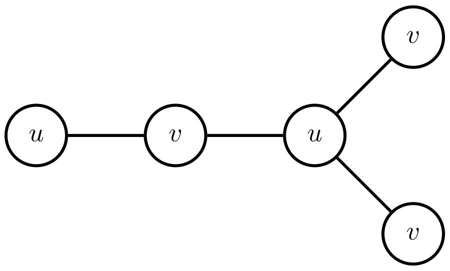
```
DFS preorder: ABCFDE
DFS postorder: FCBEDA
BFS: ABDCEF
```

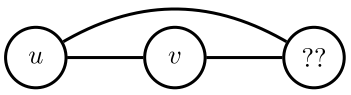
1.3 Give a valid topological sort of the graph. (*Hint*: Consider the reverse postorder of the whole graph.)

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).

Graph Algorithm Design

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?





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: $O(E + V)$.

$\boxed{2.2}$ 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 $E > V$.

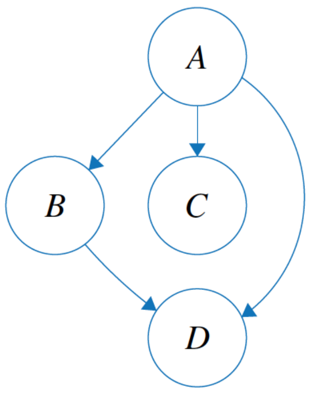
The key realization here is that the shortest directed cycle involving a particular source vertex s is just the shortest path to a vertex v that has an edge to s, along with that edge. Using this knowledge, we create a shortestCycleFromSource(s) subroutine. This subroutine runs BFS on s to find the shortest 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 $O(EV + V^2) \in O(EV + EV) \in O(EV)$.

$\boxed{2.3}$ 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.



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.