Problem Set 4

Hongrui(Ray) Liu

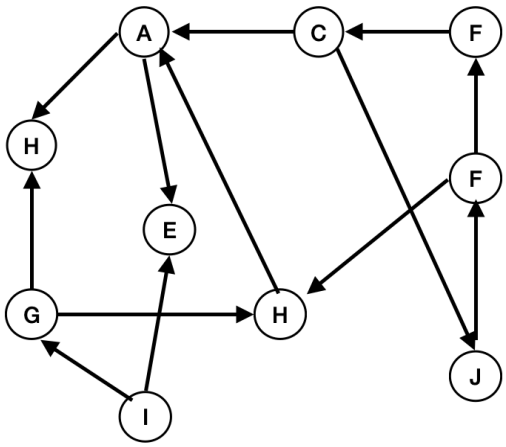
3.4

(a)

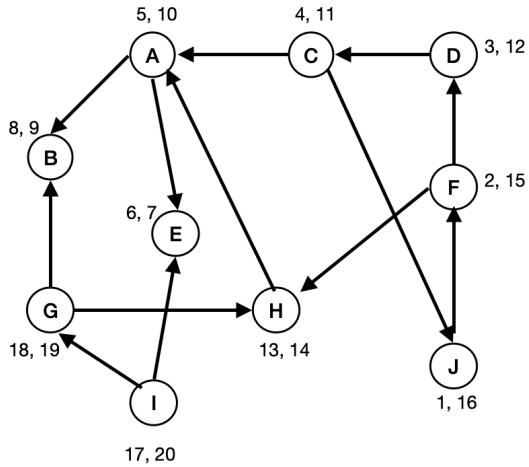
Solution:

Generally, we need to run DFS on the graph twice, and the only difference is that we need to run the first DFS on the reversed graph(all edges in the original graph are reversed).

1. Reverse the graph



2. Run the first DFS on the given reversed graph



3. Run the second DFS on the original graph

After execution of DFS, it requires the other DFS on the original graph in descending order of post-visit number. In this case, the order of implementing DFS on the graph should be : I, G, J, F, H, D, C, A, B, E

- We start off with I and end at G,

  In this case, nodes I, G, H is a strongly connected component. Therefore, SSC set = {{I, H, G}}

- We start off with J and end at F,

  In this case, nodes J, C, D, F is a strongly connected component. Therefore, SSC set = {{I, H, G}, {J, C, D, F}}

- We start off with A and end at A,

  In this case, node A is a strongly connected component. Therefore, SSC set = {{I, H, G}, {J, C, D, F}, {A}}

- Similarly, each of node B and node E is independently a SSC.

  And SSC set ={{I, H, G}, {J, C, D, F}, {A}, {B}, {E}}

(b)

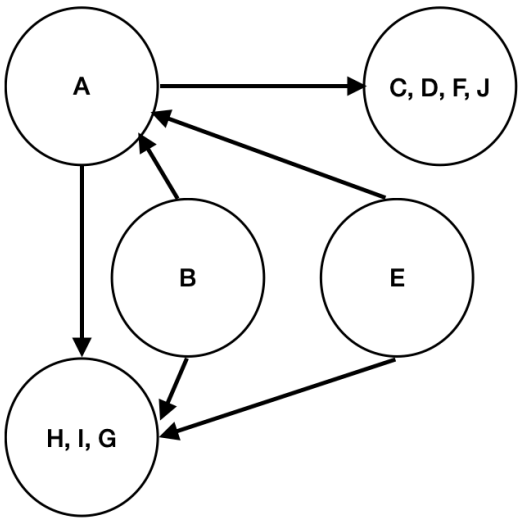
By definition vertex with indegree 0 is called source and vertex with outdegree 0 is called sink.

Therefore, the sink SSCs are : {C,D,F,J},{H,I,G}

And the source SSCs are : {B}, {E}

(c)

The meta graph is drawn below:



3.5

Solution:

Input: G = (V, E) Output: G^R

Set up a empty edge set S

for each vertex v in V

for each (u,v) in E add(v,u) to S

return G^R

Explanation: In order to reverse the graph, all we have to do is find all edges in the graph and reverse the nodes at both ends and return the reversed graph in the end.

Time complexity: O(V+E). Since we have to traverse all the nodes and edges in the graph.

3.7

Solution:

(a)

Generally, we use the 2-coloring strategy to tell whether a graph is bipartite, and what occurred to me first is that every time when we want to paint the connected nodes, we can use a loop to traverse all the nodes connected but the time complexity is not linear. Therefore, we have to figure out another approach for this strategy.

In this case, DFS comes out first.

IsBipartite(G):

for all the v in V: //initialize all the nodes in the graph

set v as “unpainted”

//After initialization, we can start the DFS coloring search

for all the v in V:

if v is “unpainted”

{ if Paint(v, “blue”) == false

return false

}

//after the whole loop， return true

Paint(v, color):

v = color // paint the current node v with the color

for each edge connected to v(v, u) in E:

if(u is “painted” && u.color == v.color)

//this means the neighbor is colored the same

//and we return false.

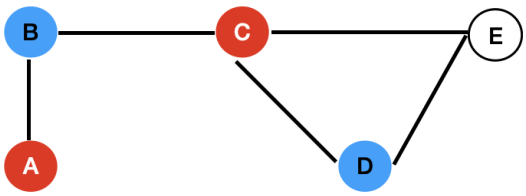
return false

else
Paint(u, -color) //Paint the other color on this node

(b)

In order to prove that an undirected graph is bipartite if and only if it contains no cycles of odd length, we could try to prove that an undirected graph is not bipartite iff it contains cycles of odd length.

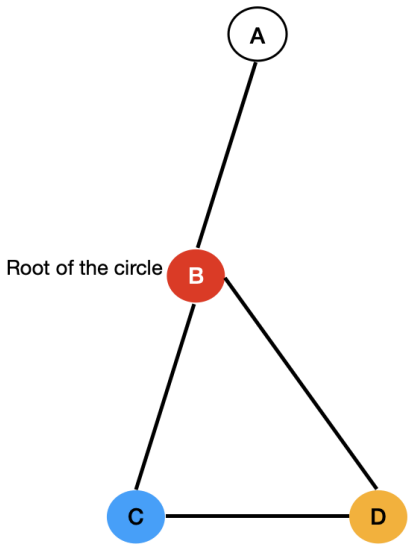
In this case, we can run the coloring algorithm on the graph given below.



From the given graph, we can know that it is impossible to color this graph with just 2 colors, which indicates that it's not bipartite. And hence prove that an undirected graph is bipartite if and only if it contains no cycles of odd length.

(c) If we want to make a graph with exact one odd-length cycle a bipartite, we need at least 3 colors. Here's the explanation for this:

When we make use of the coloring algorithm, we are actually calling DFS on the graph. And when the search cursor jumps to the node in the circle, the coloring approach will always return false since there will always be one vertex left and it is connceted to the root of DFS search where the cursor initially jumps into the circle. If there are only two colors, the current vertex can only be painted to the color identical to the root vertex. However, if there are three colors, the current vertex can be painted to the third color and it still meets the requirement for a bipartite graph. Here's an example for it:

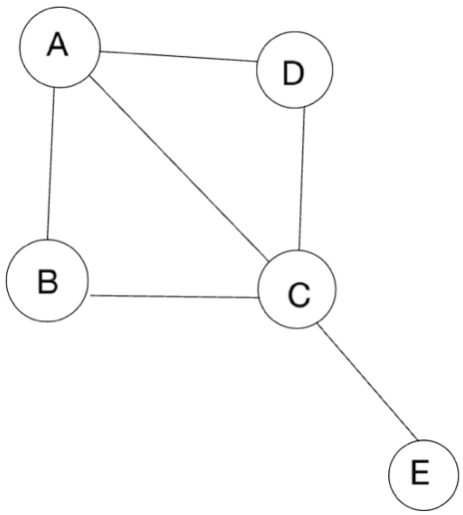


3.13

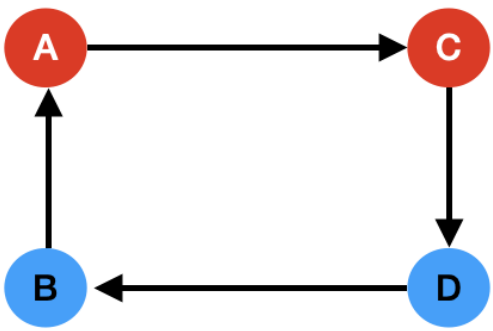
Solution:

(a) First, we need to run DFS on the graph, when we first traverse to the bottom of the graph, which means that the current node doesn't have un-visited nodes. In this case, we can just remove this node from the graph, and it's still connected.

For example:



So we run DFS on this graph with a source point on A, and we can have node E at the bottom of the tree. So we delete E from the graph and obviously it's still connected.

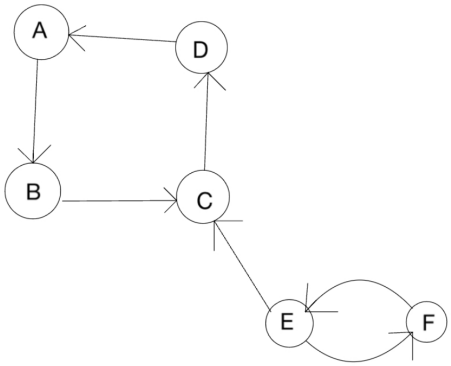


But, if we remove one random single edge from the graph, it would not be strongly connected.

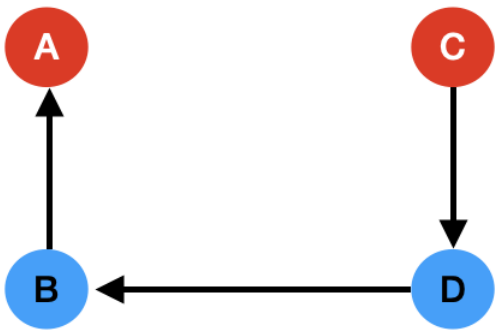
(b) Consider the given graph below: Since it's a cycle, it's always strongly conncected.

(c) If we want to make sure that adding one edge to the un-directed graph wouldn't make the graph a strongly connected graph, we have to ensure that the two strongly connected components in the graph are disjointed, which means there are no edges between them. Next, if we add one edge between the two connected components, obviously it can not make the graph

strongly connected since only nodes in one component can traverse all nodes in the graph. Here's an example for it:



For this specific example, I added an edge between component{A,B,C,D} and {E,F}. In this case, the nodes in the component can reach all the nodes in the graph whole nodes in the component{A,B,C,D} can not reach the nodes in the other component. Therefore, at least 2 edges are needed to get the graph connected.



3.16

Solution:

A certain path in this graph can be represented as a list of semesters for the CS curriculum. Each node in the path is the pre-requisite for the next node which corresponds to another course in the path.

Since a student can take as many courses as he wants, the minimal number of semesters would be at most the length of the longest path in the graph.(Each one single path, you can't proceed to the next course until you finish the current course)

Here's the algorithm for getting the minimal number of semesters:

Input: a graph(V, E)

Output: the minimal number of semesters required to finish the curriculum

Initialize a set S that holds all the nodes of the graph

integer i = 0// used for tracking the number of the minimal semesters

while the set is not empty find all the nodes in the graph that do not have pre-requisites. And delete all related nodes in the set.

i++

return i

Explanation: This is actually a greedy approach, since we are taking as many courses as possible in each semester. There are several different paths in a graph, and the maximum of the minimal number of semesters is the length of the longstest path in the graph. When we first start off with all the vertices that have no previous vertex, we are actually traversing multiple paths at the same time, they may intersect, but it does not matter. Only if we delete all vertices that have no previous nodes(if the previous nodes have been deleted from the set, it can be also called "node previous nodes"), it can always help us find a minimal answer.

Time complexity: O(V + E)