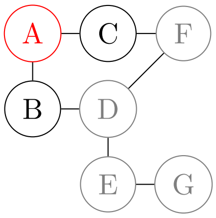
COMP20007 Design of Algorithms

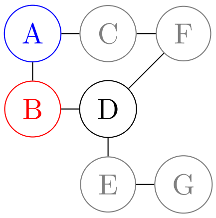
Week 5 Workshop Solutions

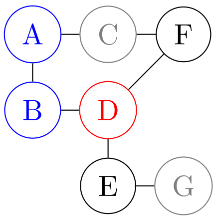
Tutorial

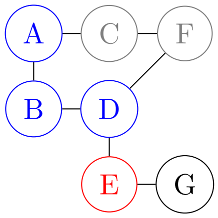
1. Depth First Search and Breadth First Search

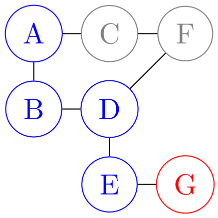
(i) Depth First: ABDEGFC The nodes visited are as follows. The node currently being visited is in red, the previously visited nodes are in blue and the nodes currently being considered are in solid black. Others are in gray.

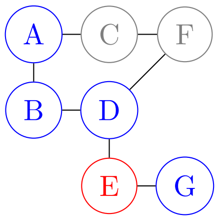
1. 

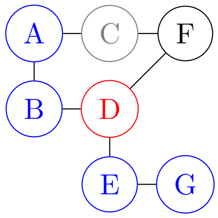
2. 

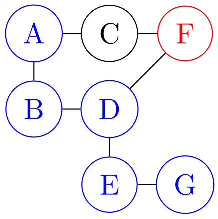
3. 

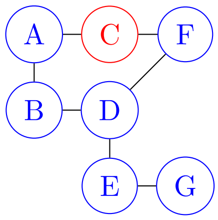
4. 

5. 

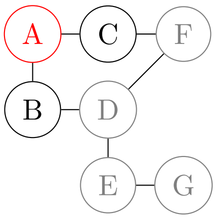
6. 

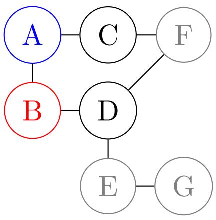
7. 

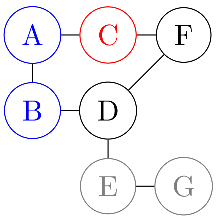
8. 

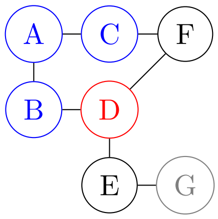
9. 

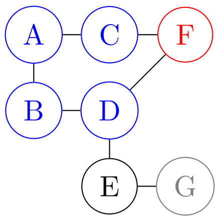
(ii) Breadth First: ABCDFEG

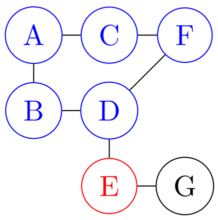
1. 

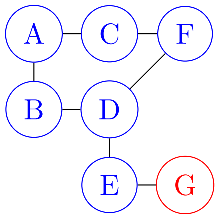
2. 

3. 

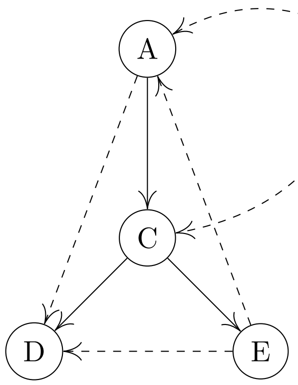
4. 

5. 

6. 

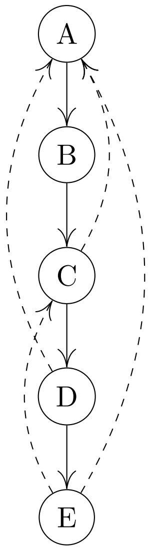
7. 

2. Tree, Back, Forward and Cross Edges For the directed version of this graph the DFS tree looks like this:



The solid edges are *tree* edges. The dashed edge (A, D) is a forward edge as it is from A to one of its non-children descendants. The edge (E, A) is a back edge as it connects E to a non-parent ancestor. The remaining edges (*i.e.*, (E, D), (B, A) and (B, C)) are all cross edges, as they connect vertices which are neither descendants nor ancestors of each other.

In the undirected version of this graph we get:[1]



We can see that only tree edges and back edges appear in the DFS forest for the undirected graph.

In fact undirected graphs only have tree and back edges:

- Suppose we had a forward edge (x, y), *i.e.*, y is a descendent of x. Since the graph is undirected, x is connected to y and visa versa. However we would have either visited y from x (making (x, y) a tree edge) or seen x while we're visiting y before y is popped from the stack (making (y, x) a back edge). So (x, y) can not be a forward edge in an undirected graph.
- Suppose we have a cross edge (x, y), *i.e.*, x is visited during some other part of the tree (*i.e.*, y has already been visited and popped). This cannot arise in an undirected graph though, since we would have visited x while we were visiting y since y connects to x and visa versa. Thus we can't have a cross edge.

3. Finding Cycles First, looking at DFS – it turns out that an undirected graph is cyclic if and only if it contains a back edge. We change the exploration strategy to find back edges:

function CYCLIC($\langle V, E \rangle$)
 mark each node in V with 0

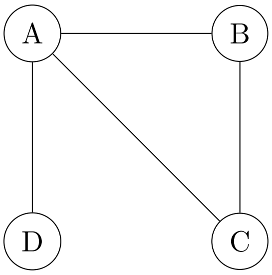
[1] *Update:* this has been updated to add a previously missing back edge from E to A

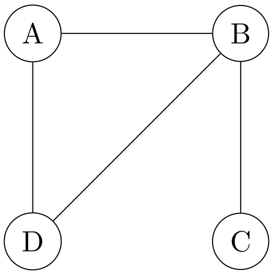
```
    for each v in V do
        if v is marked 0 then
            if DfsExplore(v) = True then                ▷ a back edge was found
                return True
    return False

function DfsExplore(v)
    mark v with 1
    for each edge (v, w) do                             ▷ w is v's neighbour
        if w is marked with 0 then
            if DfsExplore(w) then
                return True
        else
            if (v, w) is a back edge then
                return True
    return False
```

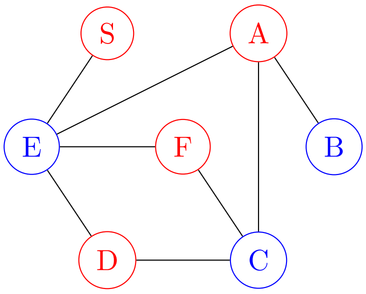
For breadth first search however we get cycles when there exist cross edges. Similar alterations to the breadth first search algorithm could be made to check for cross edges.

Sometimes depth-first search finds a cycle faster, sometimes not. Below, on the left, is a case where depth-first search finds the cycle faster, before all the nodes have been visited. On the right is an example where breadth-first search finds the cycle faster.





4. 2-Colourability First, the (only possible, up to swapping colours) 2-colouring of this graph is:



An undirected graph can be checked for two-colourability by performing a DFS traversal.

This begins by first assigning a colour of 0 (that is, no colour) to each vertex. Assume the two possible "colours" are 1 and 2.

Then traverse each vertex in the graph, colouring the vertex and then recursively colouring (via DFS) each neighbour with the opposite colour. If we encounter a vertex with the same colour as its sibling then a two-colouring is not possible.

In simpler terms, we're just doing a DFS and assigning layers alternating colours: *i.e.*, the first layer gets colour 1, then the second layer colour 2 *etc.* We know that we are not 2-colourable if we ever find a node which is adjacent to a node which has already been coloured the same colour.

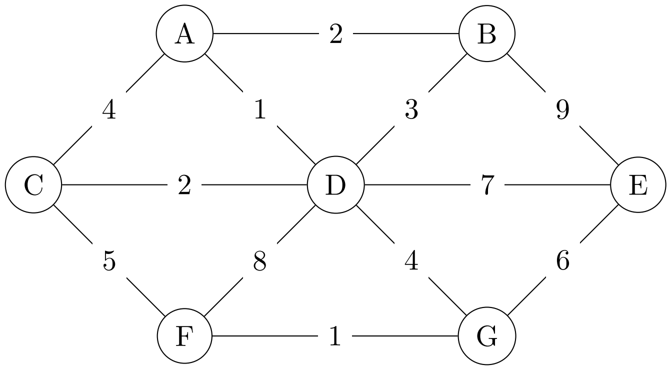
```
function IsTwoColourable(G)
    let ⟨V, E⟩ = G
    for each v in V do
        colour[v] ← 0
    for each v in V do
        if colour[v] = 0 then
            Dfs(v, 1)
    output True

function Dfs(v, currentColour)
    colour[v] ← currentColour
    for each node u in V adjacent to v do
        if u is marked with currentColour then
            output False and exit
        if u is marked with 0 then
            Dfs(u, 3 − currentColour)
```

As for 3-colourability and onwards, it turns out this is an NP-Complete problem, that is, it's the hardest class of problem we know. In practice this means that we only have exponential time algorithms to compile such a property of a graph.

To understand why we can't apply the same strategy for more than 2 colours we just need to think about the "choices" the algorithm needs to make at each step. For 2-colourability there is no choice, as a node has to be different to the node we're coming from. In 3-colourability and onwards the algorithm would have to start trying mutliple different combinations – this gives rise to the need for exponential time algorithms.

5. Single Source Shortest Path with Dijkstra's Algorithm



The following table provides the values of the priority queue at each time step (each column corresponds to each time step). The vertices which have already been removed from the queue do not have entries in that column. The subscript for each distance in the table indicates the vertex from which the vertex in question was added to the priority queue from. This is useful for tracing back shortest paths.

First with E as the source:

Node							
A	∞	∞	∞	8_D	8_D		
B	∞	9_E	9_E	9_E	9_E	9_E	
C	∞	∞	∞	9_D	9_D	9_D	9_D
D	∞	7_E	7_E				
E	0						
F	∞	∞	7_G	7_G			
G	∞	6_E					

Now with A as the source

Node							
A	0						
B	∞	2_A	2_A				
C	∞	4_A	3_D	3_D			
D	∞	1_A					
E	∞	∞	8_D	8_D	8_D	8_D	8_D
F	∞	∞	9_D	9_D	8_C	6_G	
G	∞	∞	5_D	5_D	5_D		

So the shortest path from E to A is cost 8 and goes $E \to D \to A$. The shortest path from A to F is cost 6 and goes $A \to D \to G \to F$.

6. Minimum Spanning Tree with Prim's Algorithm Running Prim's is almost the same as Dijkstra's, making a greedy (locally optimal) decision at each time step and taking the lowest cost vertex from the priority queue. The main difference is we take the cost of the single edge which connects each new vertex to the tree, rather than a cumulative cost. Again, keeping track of the vertex we come from gives us an easy way to read off the minimum spanning tree from the table.

Node							
A	0						
B	∞	2_A	2_A				
C	∞	4_A	2_D	2_D			
D	∞	1_A					
E	∞	∞	7_D	7_D	7_D	6_G	6_G
F	∞	∞	8_D	8_D	5_C	1_G	
G	∞	∞	4_D	4_D	4_D		

Summing up the final entry in each row gives us the total cost of the minimum spanning tree: 16. To find the edges in the minimum spanning tree we reach the vertex from the end of each row (except for A), e.g., (B, A), (C, D), (D, A), (E, G), (F, G), (G, D).