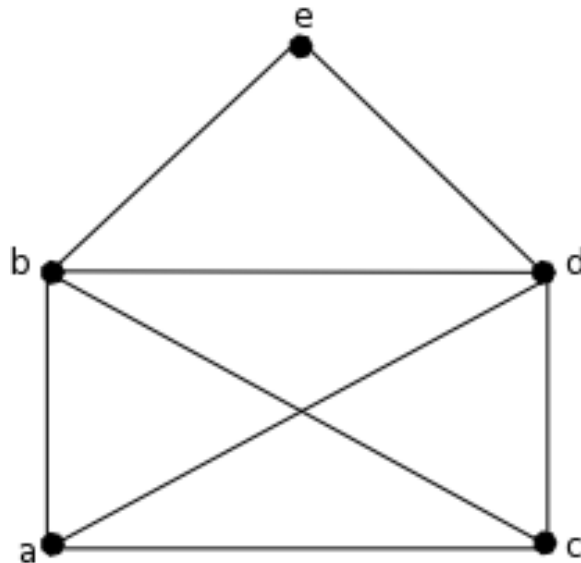


C241 Assignment 11: Graph Theory

Solutions

1) Use the following undirected graph $G = (V, E)$ to answer the questions below:



a) List the vertex set, V , and edge set, E , for this graph.

$$V = \{a, b, c, d, e\}$$

$$E = \{(e, b), (b, e), (e, d), (d, e), (b, d), (d, b), (b, a), (a, b), (a, c), (c, a), (a, d), (d, a), (c, b), (b, c), (c, d), (d, c)\}$$

b) The 'degree' of a vertex is the number of edges connected to it. Give an example of a vertex of degree 4.

vertex b has 4 edges connected to it, so it has degree 4

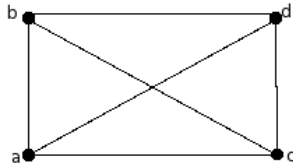
c) Give an example of a path of length greater than 3 from vertex b to d .

The length of a path is the number of edges in it. We'll need to repeat vertices (go through a cycle) to get a path long enough for this. One example would be: b, c, a, b, d .

d) Give an example of a cycle starting at a .

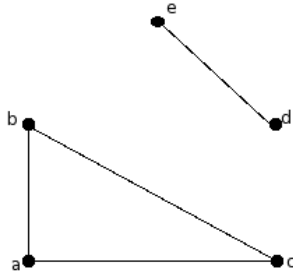
Remember that cycles need to start and end on the same vertex! a, d, c, a

e) Remember that a graph G' is a 'subgraph' of G if $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$. Give an example of a subgraph of graph G (draw it).



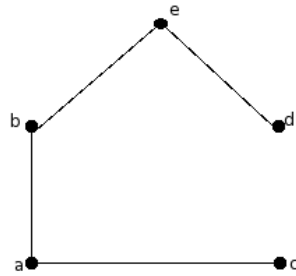
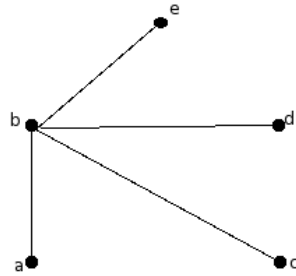
f) We say that a graph is 'connected' if for every pair of vertices in the graph there is some *path* that connects them (in other words, every vertex is reachable from every other vertex). This graph is connected. Which edges could you eliminate to make it so this graph was not connected, so that it had at least two separate components (so no paths connect the vertices in one component to the vertices in the other component)? (draw the resulting graph)

By eliminating (b, e) , (d, c) , and (b, d) we get the following graph:



g) A 'spanning tree' of G is a subgraph $T = (V_T, E_T)$ which does not include any cycles, and which does include all the vertices of G (so $V_T = V$). Give two examples of spanning trees for this graph (draw them).

Here are two spanning trees, one of height 1, and one of height 4:



h) For undirected graphs, a 'clique' of G is a subgraph of G in which every pair of vertices is adjacent. In other words, if $C = (V_C, E_C)$ is a clique in G , then C is a subgraph of G , and for all $u, v \in V_C$, if $u \neq v$ then the edge $(u, v) \in E_C$. What's the largest clique in G (draw it)? What's the smallest clique in G (draw it)?

The largest clique is the clique of size four made by the vertices a, b, c, d and all the edges that connect them. This is actually the subgraph we used in part e.

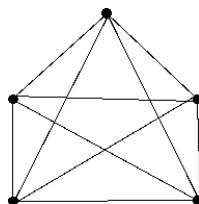
The smallest clique is technically a clique of size 0, with no vertices in it. The graph also has five cliques of size 1, since each single vertex by itself makes a clique (because a clique of size one doesn't have two different points in it, it doesn't need to have any edges connecting them).

i) What does it mean for a graph to be planar? Is this graph planar? (if yes, then prove it by re-drawing the graph appropriately, if no, explain why not).

A graph is planar if it's vertices and edges can be drawn in such a way that no two edges overlap. In the way this graph is drawn on the assignment, the edge between d and a crosses the edge between b and c . However, we can redraw the graph so that this is not the case (and thus this graph is planar). Just remove the current line connecting d and a , and instead draw a curve from d to a that goes around outside the box, to the right of the vertex c .

2) An undirected complete graph is an undirected graph in which every vertex is directly connected by an edge to every other vertex.

a) Draw a complete graph with 5 vertices.



b) If you could check the degree of a vertex in constant time, how could you check whether a graph with n vertices (and no reflexive edges) was complete in $O(|V|)$ time?

$|V|$ is the number of vertices in the graph. A complete graph is one in which every vertex is directly connected by an edge to every other vertex in the graph. So if a graph has n vertices, and each vertex has an edge to each of the other $n - 1$ vertices in the graph, the degree of any vertex in a complete graph with n vertices must be $n - 1$. Similarly, if every vertex in some graph has degree $n - 1$ (and there are no loop edges which connect a vertex to itself), then for each vertex there must be edges from it to the other $n - 1$ vertices in the graph, and the graph must be complete. All we need to do to figure out whether a graph is complete is check the degree of its vertices. If the time it takes for us to check whether a vertex has degree $n - 1$ is $O(1)$, then we can check this property for all of the vertices in the graph in $O(1 * |V|) = O(|V|)$ time.

c) What's the shortest spanning tree for a complete graph with n vertices?

The height of a tree is the number of edges on the longest path from the root to a leaf. Since each vertex in a complete graph is directly connected by an edge to every other vertex, we can choose one vertex as the root of the tree, and then span the graph by including in the tree all of the edges connected to this vertex, so that every other vertex in the graph is a leaf directly connected to this root. The height of this tree will be 1. The first spanning tree from part g of problem 1 would be a spanning tree of height 1 for a complete graph of size 5.

d) What's the tallest spanning tree for a complete graph with n vertices?

The tallest a tree of size n can possibly be is $n - 1$... a straight line of $n - 1$ edges leading from the root to the n th vertex, with no branching. This tree will be a subgraph of a complete graph (you can just trace the edges around the outside of the graph), so it's a valid spanning tree for a complete graph of size n . Thus the height of the tallest spanning tree for a complete graph of size n is $n - 1$. The second spanning tree from part g of problem 1 would be a spanning tree of height 4 for a complete graph of size 5.

e) *Bonus* How many different cliques, of any size, does a complete graph of size n have?

Remember that a clique is a subgraph in which every pair of vertices is directly connected by an edge. In other words, a clique is a subgraph which is complete.

If we start out with a complete graph $G = (V, E)$, and we make a subgraph by picking out any subset of its vertices and taking *all* the edges that connected those vertices in G , then our subgraph will also be complete (because no matter which vertices we picked, they all had to be directly connected by edges in the original, complete, graph... so they'll all be directly connected by edges in our subgraph, and that means our subgraph is a clique... for example, if we pick out any 4 vertices from the graph above, the subgraph consisting of them and the edges that connect them is a clique of size 4). So in other words, there will be a clique for every subset of V . How many cliques can there be? That's the same as: how many subsets of V can there be? $2^{|V|}$.

3) Bonus: An Euler Circuit is a cycle which includes every edge in the graph without repeating any edges. Does the graph from problem 1 have an Euler circuit? If it does have one, draw it. If it doesn't have one, explain why it doesn't. (Hint, what happens if the graph has a vertex with odd degree?)

Remember that a cycle must start and end on the same vertex! This graph does not have an Euler circuit, as no graph with a vertex of odd degree will (and vertex a here has degree 3). If you have a vertex with odd degree, say $2n + 1$ edges, you will be able to enter it on one edge and leave on a different edge n times (thus covering those $2n$ edges without repeating any). However, when you go to enter the vertex on the last edge, you will not be able to leave the vertex without traversing an edge you've already covered. Since we want a *cycle*, we can't have any dead ends. If this odd-degree vertex was not our starting vertex, we'd never make it back to the start to complete our cycle. And if it *was* our starting vertex, we'd still have a problem, since we're making a cycle: we can leave our starting vertex on one edge (covering 1 edge), and then pass back through it (in and out on different edges) another m times (covering $2m$ different edges), but we'll finally need to arrive back at it using an untraversed edge at the end of our cycle (covering 1 last edge): $1 + 2m + 1 = 2(m + 1)$ is an even number. So our starting vertex can't have an odd degree either.

4) A tree traversal is a recursive Depth-First Search algorithm for examining the values in every node of a tree. For binary trees there are three types of traversals: pre-order, in-order and post-order. Each algorithm starts with the root, at the top of the tree. A pre-order traversal first checks the value in the current node, and then recursively calls itself on the current node's left and then right subtrees (if they exist). A post-order traversal starts by recursively calling itself on the left and then the right subtrees (if they exist), and *then* it examines the value in the current node. An in-order traversal starts by calling itself on the left subtree, then checks the current node, and finally calls itself on the right-subtree (this traversal only makes sense when we're looking at binary trees, the others can be used in general).

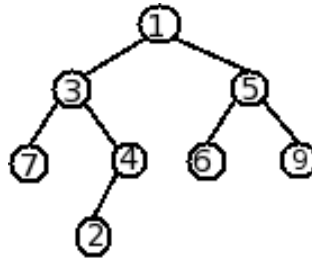
a) Write out all the values in this tree in the order they would be reached in a pre-order traversal: 1,2,7,4,2,5,6,9

b) Write out all the values in this tree in the order they would be reached in a in-order traversal: 7,3,2,4,1,6,5,9

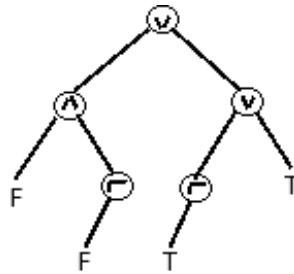
c) Write out all the values in this tree in the order they would be reached in a post-order traversal: 7,2,4,3,6,9,5,1

d) Which traversal should we use if we wanted to efficiently discover whether the number 13 was in the tree? (why?): We should use pre-order. We'd want to check the value of each node as soon as we reached it (to see whether it was 13 or not), so we could stop traversing the tree as soon as we found 13.

e) If this tree were a binary search tree, which traversal could we use to print out the contents of the tree in sorted order? (why?): In-order. Remember that binary search trees are organized so that the left sub-tree of every node contains values that are smaller than that node, and the right sub-tree contains values that are greater (or equal to). So if we do an in-order traversal, which will recursively look at the left subtree, then the node, then the right subtree, we'll end up examining the nodes in increasing order.



5) A parse tree is used to represent the underlying structure of statements. For instance, one might represent the boolean statement: $(F \wedge \neg F) \vee (\neg T \vee T)$ as the tree drawn below (where the truth values are stored in the leaves of the tree, and the operations are stored in the interior nodes). Write psuedo-code which, given a tree like this, will return the value of the boolean statement that the tree represents (either T or F). What type of tree-traversal does your solution use?



```

DFS(T: tree): boolean
{
    X = value(root(T))
    if X == " ∨ "
        return(DFS(left_subtree(T)) OR DFS(right_subtree(T)))
    else if X == " ∧ "
        return(DFS(left_subtree(T)) AND DFS(right_subtree(T)))
    else if X == " ¬ "

```

```

        return(NOT DFS(left_subtree(T))
else return X
}

```

Since the structure of the parse tree depends only on the content of the string it's parsing, we can figure out what subtrees the tree has (and what we should do with them) just by looking at the value of the current node. Thus, we're doing a pre-order traversal in this solution. Also, note that our final "else" case deals with the leaves of the tree. If the current node isn't a \neg , \vee , or \wedge , then it must be a leaf of the tree that holds a True or False value.

6) Below is pseudo code for the basic Depth First Search algorithm on general directed graphs (not just trees). Use it to answer the following questions.

```

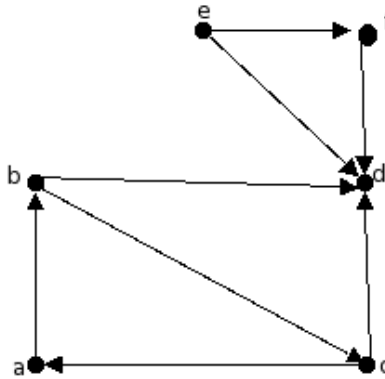
Outer_DFS(v)
{
    initialize_visited(); //the visited array keeps track of the
                          //vertices we've visited

    for each v in V; //V is the set of all vertices in the graph
    {
        if not( visited[v] == 1 )
        { Inner_DFS(v); }
    }
}

Inner_DFS(v)
{
    visited[v] = 1;

    for each n in neighbors(v); //the neighbors of v are the vertices that
    { //have an edge leading to them from v
        if not(visited[n] == 1)
        { Inner_DFS(n); }
    }
}

```



a) List the vertices of the graph above in the order they would be visited by the Depth-First-Search algorithm starting on vertex a (where a vertex is 'visited' when its spot in the Visited array is set to 1). Assume that the "for each" loops visit vertices in alphabetical order.

a, b, c, d, e, f

b) Why is the OuterDFS function necessary?

You want to traverse all of the vertices. If you only call `InnerDFS(a)` you'll only traverse the vertices that are reachable from a . The outer loop makes sure that if some vertices are left unvisited after `InnerDFS()` has been called, `InnerDFS()` is called again until eventually every vertex in V (the set of all vertices in the graph) has been visited.

c) Why is the Visited array necessary?

This ensures that vertices aren't visited more than once. Since we might have cycles in a general graph, a simple depth first search approach (like the post-order traversal we use for trees) could result in an infinite loop.

d) How could we use this algorithm to find what vertices can be reached by a path starting from a ? How could we find out vertices can be reached from e ? (Describe your algorithm in specific, detailed english; but don't write out the pseudo code).

Remember that InnerDFS(a) will visit every vertex that can be reached from some walk starting at vertex a (since it recursively calls itself on every vertex directly connected to a). So, if you want to know what vertices are reachable from some vertex v , you call InnerDFS(v) and then check which elements of the Visited[] array have a value of 1.

e) How could we adapt this algorithm to determine whether the graph G had any cycles? (Describe your algorithm in specific, detailed english) (Hint: look at your answer for c)

This is a bit tricky. You need to pay attention when InnerDFS() visits something that already has a 1 in the visited array, you need to clear the visited array periodically so that vertices that were hit on a previous depth first search (with a different starting vertex) don't interfere. The tricky part is figuring out exactly *when/how* you need to clear the Visited array. If you only clear it in the outer loop, and return that there's a cycle in the graph if you ever visit something you've visited before, then the call "InnerDFS(e)" on the graph above will falsely claim that there's a cycle (since d will be visited twice). In one possible implementation of DFS as a cycle-finding program, Outer DFS would remain the same and Inner DFS would look like this:

```
Inner_DFS(v)
{
    visited[v] = 1;
    for each n in neighbors(v); //the neighbors of v are the vertices that
    {                               //have an edge leading to them from v
        if not(visited[n] == 1)
            {Inner_DFS(n);}
        else
            {print: "Found a cycle!";}

        visited[n] == 0;
    }
}
```