CS 280 Problem Set 10 Solutions
Due May 3, 2002

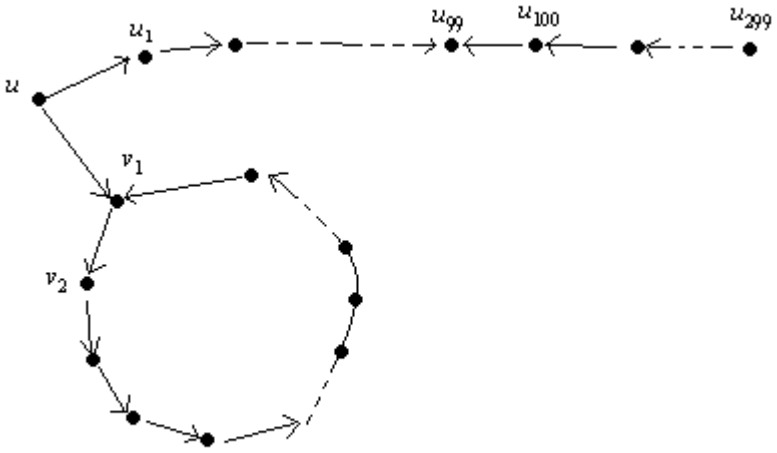
Part A

(1)
You are given a directed graph G with 1000 vertices. There is one weakly connected component of size 800 and several other weakly connected components of smaller size. There is one strongly connected component of size 500 and several other strongly connected components of smaller size.
(a) Suppose a Breadth First Search starting at vertex u visits 600 vertices. Is this vertex in the large weakly connected component? Why? Is this vertex in the large strongly connected component? Why?
(b) Suppose a Breadth First Search starting at vertex v visits 600 vertices when the search is run normally and visits 700 vertices when the search is run on the graph in which all directed edges are reversed. Is this vertex in the large weakly connected component? Why? Is this vertex in the large strongly connected component? Why?

Solution:
(a)
Yes, the vertex is in the large *weakly connected* component. We can prove this by contradiction. Suppose that the vertex is *not* in the large weakly connected component. Then, it must belong to a component that consists of *at most* 1000 – 800 = 200 vertices. However, the Breadth First Search visits 600 vertices, and clearly, the search algorithm cannot visit vertices that are not even weakly connected to it (no path). Thus, the vertex must be in the large weakly connected component.

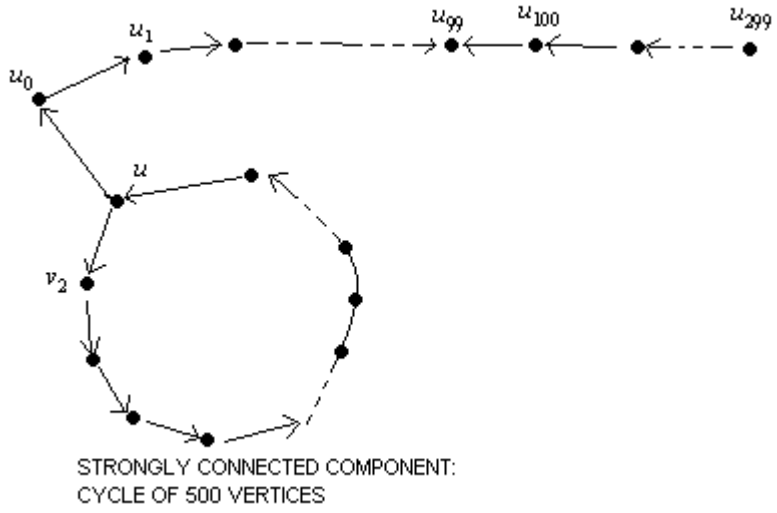
The vertex does not necessarily have to be in the large *strongly connected* component.
To elaborate, we shall give an instance of G in which the vertex does not lie in the strongly connected component, and another instance in which the vertex lies in the strongly connected component.

Looking at the weakly connected component, suppose that the strongly connected component is a cycle of 500 vertices $v_1, v_2, \ldots v_{500}$, such that there is a directed edge from vertex v_i to vertex v_{i+1}, and from v_{500} to v_1. These 500 vertices form a strongly connected component since it is possible to go from any of the 500 vertices to another vertex by going around the cycle. Suppose there is a directed edge from u to one of the 500 vertices in the cycle (say v_1). Suppose, also, that the remaining 299 vertices (call them $u_1, u_2, \ldots u_{299}$ in that order) in the weakly connected component forms a line of vertices, such that there is an edge from u to u_1, and from u_i to u_{i+1} if $i < 98$, and from u_{i+1} to u_i if $i > 98$.



Starting from u, the Breadth First Search (BFS) algorithm will visit vertices v_1 to v_{500}, as well as u_1 to u_{99}, resulting in a total search tree of 600 vertices. However, u does not belong to the connected component $\{v_1, v_2, \ldots, v_{500}\}$, since there is no way to get from any of the vertices in the component to u.

Next, we consider another instance of G in which u is in the connected component. Suppose that we make a slight change in the previous graph discussed above by replacing u with u_0, and defining $v_1 = u$. We also switch the direction of the directed edge (u, v_1) to (v_1, u). The updated graph is shown below:



Starting from u, the BFS algorithm will visit vertices v_1 to v_{500}, as well as u_1 to u_{99}, resulting in a total search tree of 600 vertices. Furthermore, since $u = v_1$ belongs to the "cycle," it belongs to the connected component $\{v_1, v_2, \ldots, v_{500}\}$.

(b)
Yes, the vertex v is still in the weakly connected component, for the same reason given in (1)(a).

Yes, the vertex v is in the large strongly connected component. We can prove this by contradiction. Suppose that the vertex is *not* in the large strongly connected component. Then, there must be a path leading from it to one of the vertex in the strongly connected component in G, otherwise, the search algorithm will not be able to visit any vertex in the strongly connected component, and thus, it will not visit more than 500 vertices. There must be no path leading from any vertex in the strongly connected component back to v, otherwise, v will belong to the strongly connected component, by definition.

Then, if we reverse all the edges in G, the strongly connected component still remains, since there will still be a path from any two nodes v_a and v_b. (To get from v_a to v_b, follow the reverse v_b to v_a path in original G; to get from v_b to v_a, follow the reverse v_a to v_b path in original G). However, there will no longer be a path from v to the strongly connected component, since that path is now *reversed*. Therefore, a BFS starting from v will visit at most 500 vertices, contradicting the given fact that it visits 700 vertices.

Therefore, v must be in the large strongly connected component.

(2)
Your relatives have decided that, as an expert in Computer Science, you should handle the difficult problem of planning seating assignments for the next meeting of your extended family. The difficulty is that several of your relatives are feuding and refuse to eat a meal in the same room with various other relatives. The event is being held in a banquet hall which has available many different dining rooms of many different sizes. The goal is to fit your relatives into as few rooms as possible without placing any feuding pair in the same room. Show how this problem can be solved using a graph coloring algorithm. In other words, assume you have an algorithm that colors a graph with as few colors as possible and show how to build a graph such that the graph-coloring solution or this graph easily leads to a solution for seating relatives. You should assume that the banquet hall has many rooms as you need and that, whatever size room you need, such a room is available.

Solution:
We can define the problem as follows:
Let each relative r correspond to a node $v \in V$.
For any pair of feuding relatives r_i and r_j, we include the undirected edge $(v_i, v_j) \in E$.
Then, for the graph $G = (V, E)$, we ask: what is the minimum number of colors required to color G so that no two adjacent nodes are assigned the same color. Presumably, the algorithm provided will solve the problem.

Then we claim:
(2.1) The graph can be colored with at most *k* colors if and only if there is a way of assigning all the relatives to at most *k* rooms such that no two feuding relative are assigned to the same room.

Proof of claim:
Suppose that the graph *G* can be colored with at most *k* colors. Then, we proceed to assign the relatives to the rooms as follows: if two nodes are assigned the same color, we allocate the corresponding relatives to the same room. Then, since there are at most *k* colors, the relatives are assigned to at most *k* rooms. Since any two adjacent nodes in the graph *G* are assigned different colors, we can also conclude that any two feuding relatives (corresponding to the adjacent nodes) are allocated different rooms as well.

Conversely, suppose that the relatives can be satisfactorily allocated to at most *k* rooms (i.e. no two feuding relatives in same room). Then, we proceed to color the graph G as follows: if two relatives are allocated to the same room, we assign the two corresponding nodes the same color (corresponding to the room). Then, since there are at most *k* rooms, we color the graph *G* using at most *k* colors. Furthermore, since no two feuding relatives are in the same room, and feuding relatives have an edge between their corresponding nodes, we can also conclude that no two adjacent nodes are assigned the same color as well.

Having proved (2.1), our construction is complete, and we can use this graph, and the graph-coloring algorithm to determine the minimum number of rooms required to seat all the relatives.

Part B

(3)
The Emperor has gone insane and has declared that from now on, every road will be a one-way road. Further, the Emperor declares that if a traveler leaves a town then it should not be possible to return to it. As the Emperor's Chief Advisor, you have the task of picking a direction for each road in the Empire (you are not allowed to close or destroy roads). If you fail to complete the task the Emperor will be very unhappy and will probably order that something rather unpleasant be done to you. You would prefer to avoid this - the Emperor is insane after all. Is there a way to assign directions so that the Emperor's rules are satisfied? Explain. To simplify the problem, you can assume that each road starts and ends at a town and that roads only cross at towns.

Solution:
Yes, there is a way to assign directions so that the Emperor's rules are satisfied.

We shall describe two ways of doing this.

Method 1:
First, we construct a graph G in which the each vertex represents a town, and each road between two towns is represented as an edge between the two corresponding vertices. Then we assign the integers 1 to n to all vertices v_1 to v_n so that each vertex gets a unique integer label. Finally, we require that any edge between two vertices v_i and v_j must go in the direction from the vertex with the smaller label to the vertex with the larger label. In other words, if v_i and v_j are connected, the edge will go from v_i to v_j if $j > i$, and from v_j to v_i if $i < j$. Since edges only go from vertices with smaller integer labels to vertices with larger integer labels, once we leave a vertex, we can only visit vertices with larger integer labels, and it is not possible to return. Therefore, the resulting graph satisfies the Emperor's rules.

Method 2:
Similarly, we construct a graph G in which the each vertex represents a town, and each road between two towns is represented as an edge between the two corresponding vertices. We choose a random vertex v in G, and declare that vertex to be the *starting point*. Similar to Dijkstra's algorithm, we require that the rest of the vertices be *unexplored* initially, except for the starting point v.

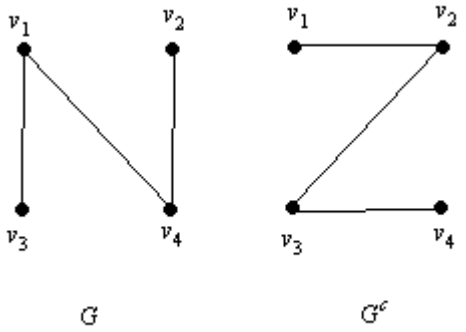
Next, we perform a Breadth-First Search (BFS) with v as the root. Starting from v, we assign directions to edges with one end in v so that all roads point away from v. We then visit the next node specified by the BFS, label it *explored*, and assign directions so that all unassigned roads point away from this node. We proceed until all roads have been assigned directions. We claim that the resultant directed graph G` will satisfy the Emperor's rules.

One important observation here is that at any instance in the execution of the algorithm, we cannot have roads leading from an unexplored node to an explored one, since we only assign road directions starting from explored nodes. Furthermore, we cannot have roads leading from an explored node to another explored one, since the nodes are explored *in order*, not simultaneously. Therefore, for any two adjacent nodes $u, v \in G$, one will be explored before the other; if u was explored in the BFS before v, the edge (u, v) will be directed from u to v, otherwise the edge will be directed from v to u. At any point in the execution of the algorithm, an edge can only be directed from an *explored* node to an *unexplored* one. Therefore, if u is visited before v, there will not be any path from v back to u, since u would have been explored, and edges can only lead to unexplored nodes. We have then proven that G` satisfy the rules set by the Emperor; if a traveler leaves a town, it is not possible to return to the town again.

---

(4)
A graph is *self-complementary* if it is isomorphic to its complement. G^c, the *complement* of graph G, has the same vertex set as G, but has an edge between vertices u and v if and only if in G there is *no* edge between u and v. Give an example of a graph that is self-complementary.

Solution:
An example of a self-complementary graph is shown below:



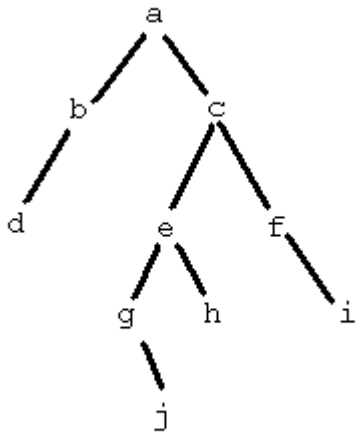
G consist of the vertices $V = \{v_1, v_2, v_3, v_4\}$, and edges $E = \{(v_1, v_3), (v_1, v_4), (v_2, v_4)\}$.
By definition, G^c consists of $V^c = V$, and $E^c = \{(v_1, v_2), (v_2, v_3), (v_3, v_4)\}$

There is a one-to-one and onto function f from V to V^c as follows: $f(v_1) = v_2$, $f(v_2) = v_4$, $f(v_3) = v_1$, and $f(v_4) = v_3$. Furthermore, the mapped vertices satisfy the property that v_a and v_b are adjacent in G if and only if $f(v_a)$ and $f(v_b)$ are adjacent in G^c. Therefore, the graphs G and G^c are isomorphic. Thus, G is a self-complementary graph.

Note: This is not the only solution, even though this is a rather simple one.

Part C

(5)
Consider the following tree:



(a) In what order are the nodes processed in a preorder traversal?
(b) In an inorder traversal?
(c) In a postorder traversal?

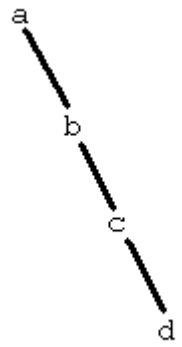
(d)Give an example of an ordered binary tree with 4 nodes for which nodes are processed in the same order for both the preorder traversal and the inorder traversal.

Solution:
(a) In a preorder traversal, the root is traversed first, followed by the left subtree, and finally, the right subtree.
Therefore, the order of traversal in this instance would be:
`a – b – d – c – e – g – j – h – f – i`

(b) In an inorder traversal, the left subtree is traversed first, followed by the root, and finally, the right subtree.
Therefore, the order of traversal in this instance would be:
`d – b – a – g – j – e – h – c – f – i`

(c) In a postorder traversal, the left subtree is traversed first, followed by the right subtree, and finally, the root.
Therefore, the order of traversal in this instance would be:
`d – b – j – g – h – e – i – f – c – a`

(d) The following tree would give the same order of traversal for both preorder and inorder traversals. The order is `a – b – c – d`.



---

(6)
(a) Give necessary and sufficient conditions on a graph *G* so that the depth-first tree and the breadth-first tree are the same.
(b) How many nonisomorphic spanning trees does K_4 have? Explain.

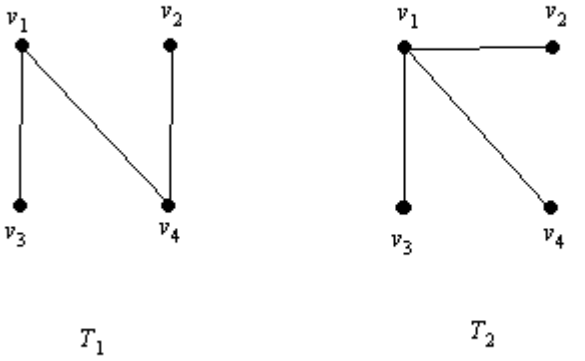
Solution:
(a) Suppose a depth-first tree T_D is formed from the graph *G*, starting at a particular node *v*. In a depth-first tree, there are no edges between nodes in different subtrees of T_D in the original graph *G*; any edge (*a*, *b*) must connect an *ancestor* node (not necessarily parent) to a *descendent* node in T_D.

Suppose a breadth-first tree T_B is formed from the graph G, starting at the same node v. In a breadth-first tree, the distance of the two nodes from v in T_B can differ by at most 1. In other words, all edges must connect nodes in either the same "layer" (or *height*) in T_B, or nodes which reside in neighboring "layers".

Combining the two requirements, we note that for $T_D = T_B$, any edge (a, b) in G must connect an ancestor node in T_D to a descendent node in the sane tree. Furthermore, the any edge (a, b) must connect two nodes which reside in neighboring "layers" of T_D.
The only possible edges, which could satisfy both requirements, are the original edges in the spanning trees T_D (or T_B).

Thus, $T_D = T_B = G$ in order for both the depth-first tree and the breadth-first tree to be the same.

(b) K_4 has 2 nonisomorphic spanning trees. They are shown in the diagram below.



T_1 and T_2 are the only two nonisomorphic spanning trees of K_4.
K_4 consists of 4 nodes. Therefore, a spanning tree of K_4 has three edges, and the sum of degrees of the vertices of the spanning tree must be $3 \times 2 = 6$.
By definition of a spanning tree, none of the vertices must have degree zero. Therefore, the only two ways to "allocate" the degrees of the four vertices such that they sum up to 6 are (1, 2, 2, 1) and (1, 3, 1, 1). They correspond to T_1 and T_2 respectively. Any other spanning tree is isomorphic to either of these two spanning trees.