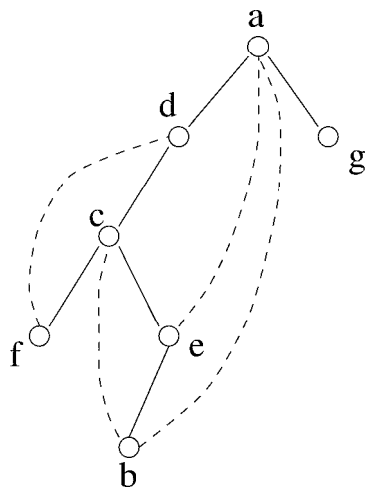
COMP 271 Design and Analysis of Algorithms
2003 Spring Semester
Solutions to Question Bank Number 2 (Selected Problems)

1 Basic Graph Problems

1. Suppose that G is not connected. Then it is possible to partition the vertex set of G as $(U, V-U)$ such that there is no edge from any vertex in U to any vertex in $V-U$. Then $\overline{G}$ contains a complete bipartite graph on U and $V-U$ as a spanning subgraph. It is easy to see that any complete bipartite subgraph is connected. If any spanning subgraph of a graph is connected, then the graph is connected.

2 Breadth First Search and Depth First Search

1. See figure.



7. We claim that G has a cycle if and only if *any* DFS of G produces a back edge. As in directed graphs, if there is a back edge, then clearly there is a cycle in the graph. And if there is no back edge, then by Theorem 22.10 on page 547 in CLR (Theorem 23.9, page 483, CLR, old edition), there are only tree edges, implying that the graph is a forest, and hence is acyclic.

   There is a subtlety in finding back edges in an undirected graph. In directed graphs, if the edge goes to a gray vertex, then this is back edge. However, in an undirected graph if u is the child of v in the DFS tree, then when the DFS visits u it will see an edge going from u to v, the other half of the tree edge. This should NOT be classified as a back edge. Therefore, before declaring an edge to be a back edge we

must test (1) that the edge goes to a gray vertex, and (2) that it does not go to the predecessor of this vertex. Actually, the algorithm only need check that the vertex is nonwhite. A black neighbor indicates the forward part of a back edge. Note that there are no cross edges in DFS for an undirected graph.

Here is the algorithm. Notice that as soon as a back edge is detected the algorithm stops. If we return from this procedure to the main procedure, then the graph has no cycle.

```
Cycle(G) {
    for each u in V do {                  // initialization
        color[u] = white;
        pred[u] = null;
    }
    for each u in V do {                  // visit all vertices
        if (color[u] == white) Visit(u)
    }
    output "No cycle";
}
Visit(u) {
    color[u] = gray;
    for each v in Adj[u] do {             // consider edge (u,v)
        if (color[v] == white) then {     // v unvisited?
            pred[v] = u;
            Visit(v);                     // visit v
        }
        else if (u != pred[v]) then {     // back edge detected
            output "Cycle found!";
            exit;                         // terminate
        }
    }
}
```

We claim that this procedure takes $O(|V|)$ time. Every edge in a DFS tree is a tree edge or a back edge. Whenever we find ANY back edge, we can say that we found a cycle, and terminate. So until we find the first back edge, we will only traverse tree edges. However, any tree in the graph of $|V|$ vertices can have at most $|V|-1$ edges. Hence the algorithm will terminate in $O(|V|)$ time (either exhausting the at most $|V|-1$ tree edges, or finding its first back edge and exiting).

3 Minimum Spanning Trees

1. (By contradiction.) Suppose to the contrary that the edge with the smallest weight, denoted $e = (u, v)$ is not in the MST. Let T be the MST. Suppose we add e to T. This creates a cycle in T' which must contain e. Let e' be any other edge on

this cycle. Since e is the smallest weight edge in the graph and edge weights are distinct, we have $w(e') > w(e)$. Let $T' = T \cup \{e\} - \{e'\}$. The removal of e' breaks the cycle, and hence T' is a spanning tree. The weight of T' is

$$w(T') = w(T) + w(e) - w(e') < w(T).$$

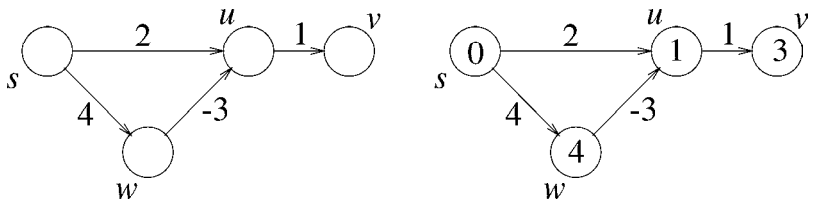
However this contradicts the hypothesis that T is the MST, and completes the proof.

3. This can be done in several ways. Here is a brief sketch of one solution. The idea is to modify Prim's algorithm by using a simple data structure for the priority queue so that queue operations can be performed in $O(1)$ time each, rather than $O(\log n)$ time. This can be done by storing all the priority queue items in two linked lists, L_1 containing items of weight 1 and L_2 containing items of weight 2 (the order of elements within each list is arbitrary). The details of `Extract_Min()` and `Decrease_Key()` are easy to work out and are left as an exercise.

4 Shortest Paths

1. (a) Order in which vertices are removed from priority queue: s, c, a, b.
   (b) On termination, $d[s] = 0; d[a] = 4; d[b] = 5; d[c] = 3$.
   (c) $d[b]$ is initially ∞. It decreases to 7 and then to 5.

2. The trick is to get Dijkstra's algorithm to finalize a vertex before its true minimum distance is known. Consider the following digraph. When s is processed, $d[u] = 2$ and $d[w] = 4$. We process u first, setting $d[v] = 3$. Next we process $d[v]$, which changes nothing. Finally we process w, setting $d[u] = 1$. The final value of $d[v] = 3$, but there is a path of length 2: $\langle s, w, u, v \rangle$.



The proof of Dijkstra's algorithm does not go through when negative edge weights are allowed, for the following reason. Recall that in case 2 of the proof, y lies on the shortest path from s to u and $y \neq u$. In the proof given in class, we claim that since y lies midway on the shortest path from s to u, $\delta(s, y) < \delta(s, u)$. This crucial

fact is true if the edge weights are all positive; however, it breaks down if negative edge weights are allowed!