# CSE202 Homework1 Answer

Yue Wang, A53102167

October 6, 2015

## 1.

Given this question, the trivial solution comes first is to remove every vertex one by one and test if the graph is still connected. But this algorithm takes $O(n*(n+m))$ time which n denotes the number of nodes and m denotes the number of edges if we use BFS to test connectivity. So we need to analyze the properties of critical vertices.
If we pick a node in the graph and from this node, we build a tree. In the tree, we need to analyze three kinds of nodes.
1. Root node: If the root node has more than one child subtree, when removing the root, a node from one of its subtree cannot goes to another subtree.
So if the root node has more than one child subtree, it definitely is a critical vertex.
2. Leaf node: Leaf node cannot be critical vertex apparently because removing it doesn't influence other nodes and edges in the tree and the graph.
3. Non-leaf and Non-root node: if after we remove the node, all nodes who are the descendants of this node can go back to the node's ancestors, which means the graph is not cut off, then the node should not be critical vertex. On the contrary, if one of its descendants cannot go back to its ancestors, the node should be critical node.
Now, we get the algorithm: First, we build the spanning tree using the DFS. Then from the tree root, we test every node to see if it is the critical vertex according to the above three properties. And now we need to find out how to denote a node's descendants can go back to the node's ancestors. We define ancestor and descendant according to the sequence of nodes in the DFS phase, namely visited[i] means the order of i when it is first accessed. And in order to find out if a node can go back to its ancestors, we use ancestor[i] to denote the node's "oldest" ancestor it can access. Then the algorithm is as following:
(1)Initialize visited[i] to be -1 for i which means the node i is unvisited. And the set s is to store the critical vetices.
(2)Set a global counter "c" to denote the order of access
(3)DFS(v, parent):
(4) visited[v] = c++
(5) ancestor[v] = visited[parent]
(6) for each (v, u):
(7) if visited[u] == -1:
(8) if ancestor[u] < ancestor[v]:

```
(9)                     ancestor[v] = ancestor[u]
(10)            else:
(11)                DFS(u, v)
(12)                if ancestor[u] < ancestor[v]:
(13)                    ancestor[v] = ancestor[u]
(14)            if (ancestor[u] < ancestor[v]) or (v is the root and v has more than one child):
(15)                add v to s
```

Finally run the DFS(v, v) as randomly select a node from the graph.