Problem Set 4

All parts are due Thursday, April 7 at 11:59PM.

Name: Arturo Chavez-Gehrig

Collaborators: Sabrina Ibarra

Part A

Problem 4-1.

(a)
```
class chaintable()
    def init:
        self.hashfunc = $h()$
        self.data=[] array of size M
        for spot in self.data:
            spot = empty linked list
    def insert(k,v)
        location = self.hashfunc(k)
        if self.data[location]== empty linked list:
            insert k into front of linked list in the form (k,v)
        for element in self.data[location]:
            if element == k:
                return 'Already there'
            if element == None
                element = (k,v)
                return
    def search(k):
        location = self.hashfunc(k)
        spots = self.data[location]
        for elt in spots: (iterate from beginning)
            if elt == (k,v):
                return v
            if elt == empty:
                return 'not there'
    def Delete(k):
        location = self.hashfunc(k)
        spots = self.data[location]
        for elt in spots
            if elt == (k,v)
                reassign pointers to remove it
                return
            if elt == empty
                return 'not there'
```

(b) Chaintable is correct because it works for all three operations, thus any combination of them. Insert is correct because it would never overwrite an existing nonidentical key. It will go to the hashed location and write to the next available location in the linked list. If the key does match, it is overwritten, so no duplicates ever exist. Search works because if k is in the table, it must hash to one location and be in the linked list in that slot. Our search alg goes to that spot and examines all elements of the list until a match is found. If it isn't found, it correctly returns that the key is not in the table.

Delete works because search works so if k is in the table, it will be found. The delete method rearranges pointers so the parent of k links to the child of k (thus removing k for the linked path and by extension, the hash table).

(c) class OpenAddressTable()

```
    def init:
        self.hashfunc = k
        self.data = array of size k initialized with None elements
    def insert(k,v)
        i = 0
        repeat j = h(k,i) until i == m
            if self.data[j] == k:
                return 'Already there'
            if self.data[j] == Null or 'deleted'
                self.data[j] = (k,v)
                return success
            i = $i + 1$
        return Fail
    def search(k)
        i = 0
        repeat j = h(k,i) until self.data[j] == Null or i == m
            if self.data[j] == k:
                return v
            i = $i + 1$
        return 'Not There'
    def delete(k):
        i = 0
        repeat location = h(k,i) until i m:
            if self.data[location] == (k,v):
                self.data[location] = 'deleted'
                return
            elif self.data[location] == null
                return 'not here'
```

(d) Open address table is correct since it works for all three operations, maintaining only one copy of each key in the table. Insert is correct because it will probe each slot in the order specified by the hash function h(k,i). It will then input the entry into the first open spot (a null or 'deleted slot). If the key already exists, the probe will find it and update the value with the new input.

Search is correct because if k is in the table it will be found before a 'null' is reached in the sequence of probed locations. This is because insertion would have placed it there if the spot was vacant and delete maintains the spot isn't null.

Delete is correct because it removes the key from the table but doesn't ruin the search or insert methods. Since when a key is deleted, it is replaced with a 'deleted' marker, the search and insert methods know to either continue search or place the new key in that slot, respectively.

Problem 4-2.

(a) In order to find the disconnected nodes, we will take advantage of the property that BFS explores all reachable nodes of the graph. We will select a start node arbitrarily and run BFS (following, the pseudocode of the recitation notes in Figure 1 attached at the end). In this version of BFS nodes are labeled by a color signifying if they have been reached. White = not reached, grey = reached and being currently explored, and black = reached and fully explored. We augment this version of BFS to return the color dictionary. Once we run BFS, taking O($V + E$) time, we compare each node in the given vertex list and check its 'color' if any vertices are still white, the graph is not connected. This takes O(V), so total run time is O($V + E$)

```
def CheckConnected$(V, E)$:
    pick start vertex V
    explored = BFS$(G, V)$
    for node in V:
        if color[node] == 'white'
            return 'unconnected'
    return 'connected'
```

(b) Given DFS tree if root node has more than 1 child, then we can conclude that root is a critical vertex. The proof references Figure 2 attached at the end.

Proof by contradiction:

We know that DFS explores the whole connected graph. We know that DFS will explore all paths derived from a child of a node before turning and picking an alternate child and exploring its path. Suppose s is not a critical vertex, but if it has 2 children. That implies that node D is reachable from node A if s was removed. Contradiction ! Since A and D are on separate branches of the DFS tree, this cannot be true. Property of DFS is that if D was reachable from A without traveling through s, it would appear as a descendant of A.

(c) T is a critical vertex if its descendants have no edges to T's ancestors.

In Figure 3 attached at the end, assume B is critical. This is a contradiction since it has a back edge from B's child C to B's ancestor A. Therefore B cannot be critical.

However, if edge E didn't exist, then B would be critical.

(d) We augment build DFS tree to track the two aforementioned properties of critical vertices. We will create a dictionary with each node as a key and a flag as the value to signify if the node is a root and contains multiple children or is any node and contains no back-edge between its descendants and its ancestors. We initialize the dictionary with 'False' flags for both. As we traverse the graph in DFS we will keep track when these properties are violated and accordingly update the flags in those dictionaries. For the case we find a back-edge to an ancestor, we will work backwards to correct the flag until we reach the destination of the back-edge or reach another 'True' flag.

Once we have built this structure, we iterate through the vertex list, check the values in the dictionary at each vertex and append the node to the critical vertices list if there is a 'True' flag.

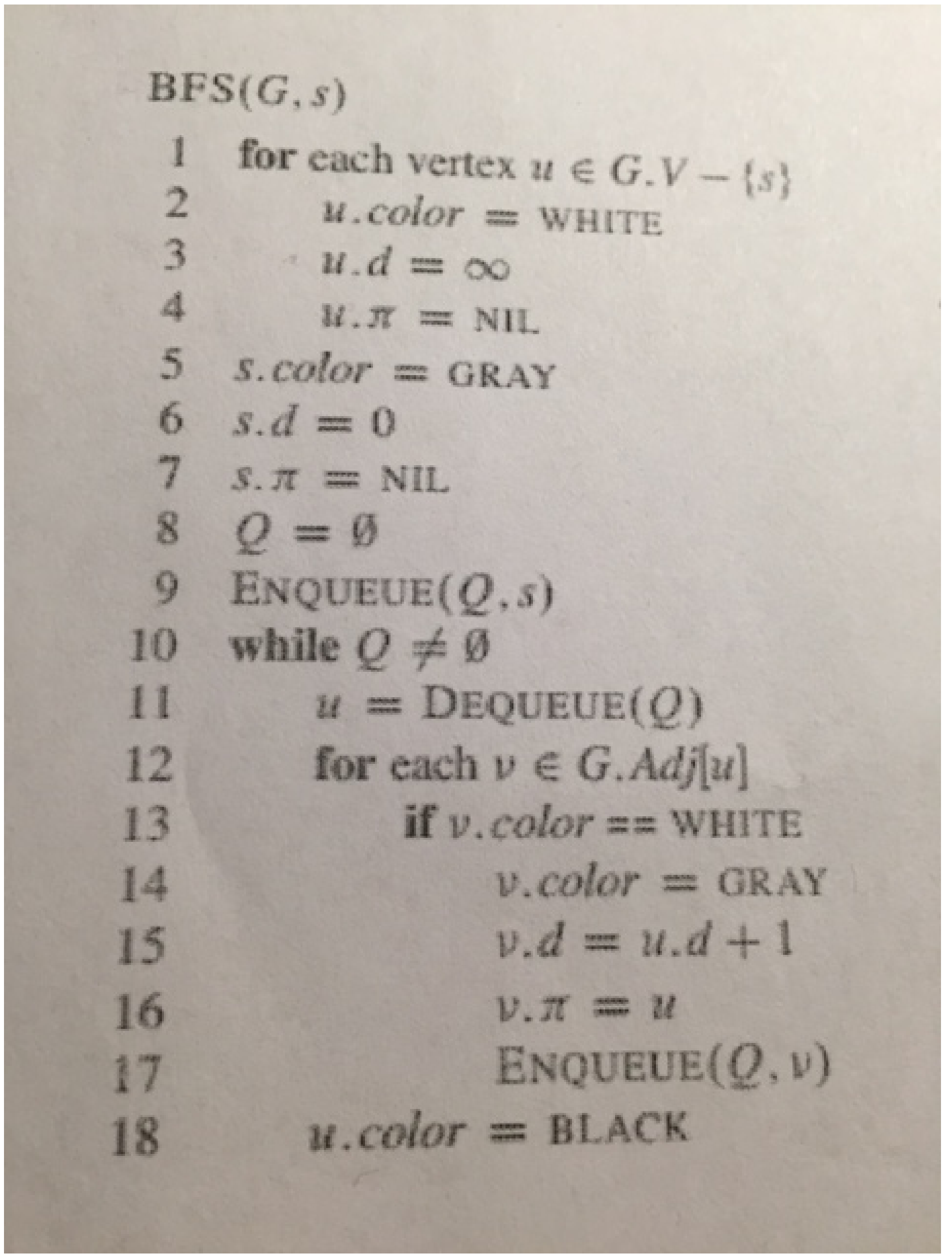
The run time of this alg is O($V + E$) to build the augmented DFS tree and requires an iteration on the vertices to see if any are critical vertices. This takes O(V) time and total run time is O($V + E$).

```
def Get-Critical-Nodes(graph):
    final-list = empty list
    DFS-Tree = Build-DFS-Tree(graph, the first element in vertex list)
    for node in DFS-Tree:
        if Flag == True:
            final-list.append(node)
    return final-list
def BuildDFSTree(graph, start):
    visit (start)
    for each connected-vertex of nodes adjacency list:
        if the connected-vertex is not visited:
            if connected-vertex is second child of node: // if node.child isnt none
                Flag = True
            if connected-vertex has back-edge:
                Flag = False
                Change previous nodes in Tree to True till reach destination of back-edge
or another flag == False
            Build-DFS-Tree(graph, connected-vertex)
            add edge (node to connected-vertex) to Tree
    return Tree
```

(e) Given that there are no critical vertices on the original graph, we can use our alg developed in 2D, but run V times removing 1 edge from the graph each time. If we find a critical vertex in the graph after removing vertex V, we can add (V,E) to the list of critical pairs. 2D Alg runs in O($V + E$) and we run V times, so O(V($V + E$)). After building the list, since we will have redundant copies of critical vertex pairs ((A,B) and (B,A) for example). We will iterate through the list and remove the extra copies. This runs in O(V) time. Total run time for this alg is O(V($V + E$)) as specified by the problem statement.

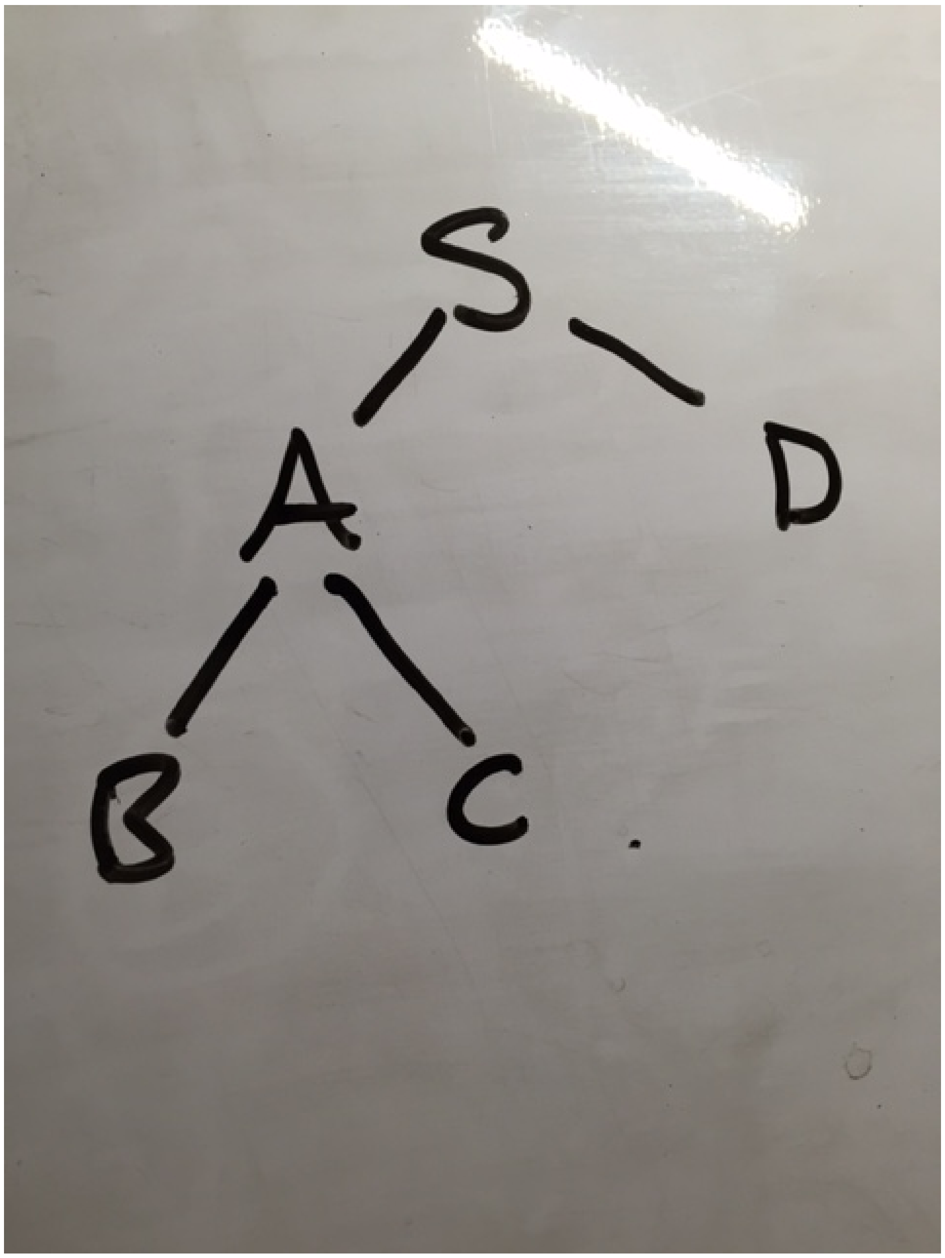
```
def Double-Critical-Nodes(graph):
    final-set-pairs = { }
    for vertex in graph:
        new-graph = copy of graph except with the vertex deleted
        connected-critical = Get-Critical-Nodes(new-graph)
        for connected-node in connected-critical:
            final-set.add((node, connected-node))
    return final-set-pairs in a list type
```

(1)

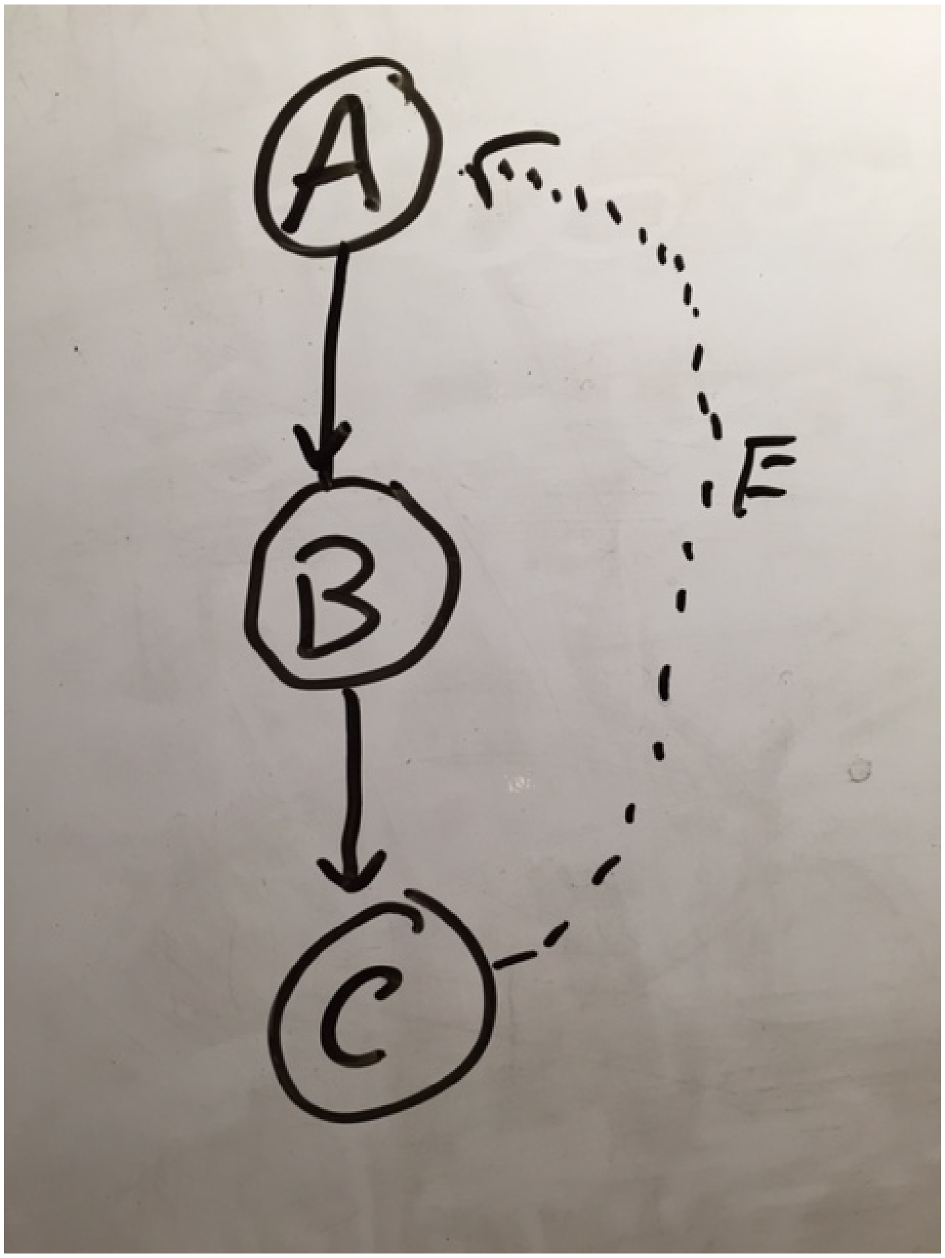


```
BFS(G, s)
 1  for each vertex $u \in G.V - \{s\}$
 2      $u.color$ = WHITE
 3      $u.d = \infty$
 4      $u.\pi$ = NIL
 5  $s.color$ = GRAY
 6  $s.d = 0$
 7  $s.\pi$ = NIL
 8  $Q = \emptyset$
 9  ENQUEUE($Q, s$)
10  while $Q \neq \emptyset$
11      $u$ = DEQUEUE($Q$)
12      for each $v \in G.Adj[u]$
13          if $v.color$ == WHITE
14              $v.color$ = GRAY
15              $v.d = u.d + 1$
16              $v.\pi = u$
17              ENQUEUE($Q, v$)
18      $u.color$ = BLACK
```

(2)

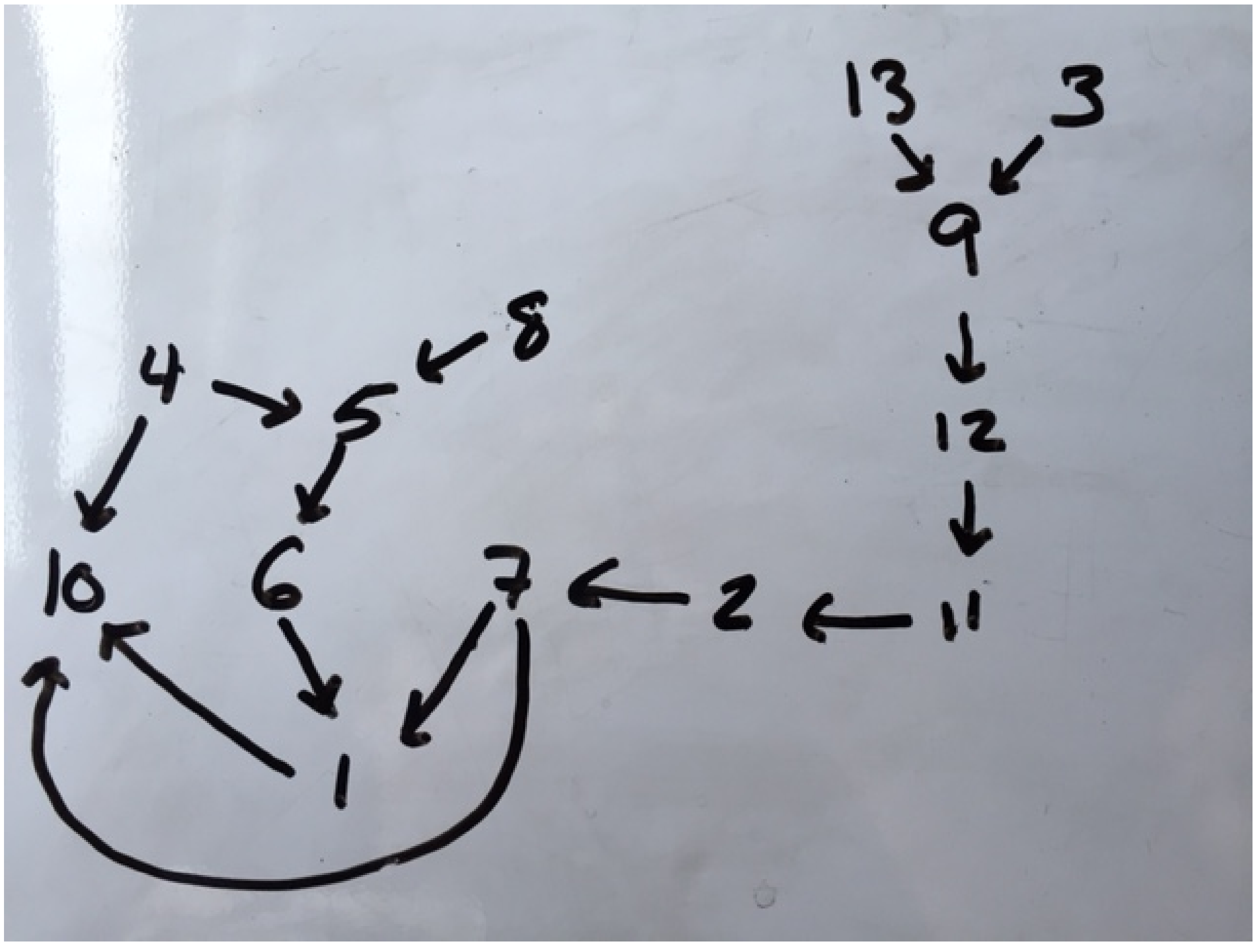


(3)



Problem 4-3.

(a)

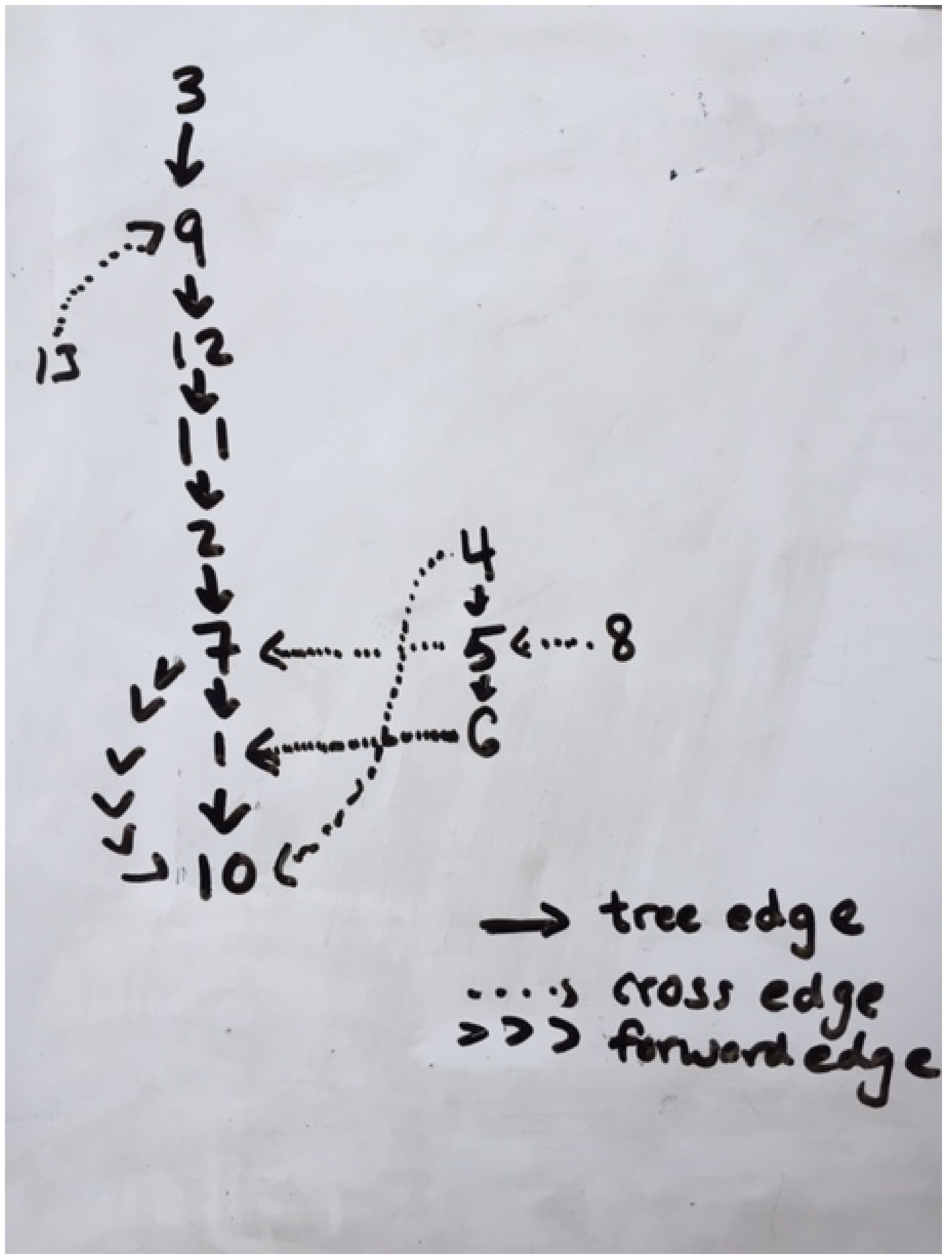


$$\begin{tikzcd}
 & & & 13 \arrow[dr] & & 3 \arrow[dl] \\
 & & & & 9 \arrow[dd] & \\
4 \arrow[r] \arrow[dd] & 5 \arrow[dd] & 8 \arrow[l] & & & \\
 & & & & 12 \arrow[d] & \\
10 & 6 \arrow[d] & 7 \arrow[dl] \arrow[ll, bend left=60] & 2 \arrow[l] & 11 \arrow[l] & \\
 & 1 \arrow[ul] & & & &
\end{tikzcd}$$

(4)

(b)

(5)



(c) The sequence of discovered nodes is: $3, 9, 12, 11, 2, 7, 1, 10, 4, 5, 6, 8, 13$. This sequence does not produce a 'valid' lemon meringue pie because many of the constraints are not met. The violated constraints are: 4 must precede 10, 6 must precede 1, 13 must precede 9, 8 must precede 5, 5 must precede 7, and these are not satisfied in the discovered list.

(d) The sequence of finish events in reverse order: $13, 8, 4, 5, 6, 3, 9, 12, 11, 2, 7, 1, 10$. All constraints are satisfied in this ordering, so this should produce a delicious lemon meringue pie. But I don't like lemon meringue pie so I don't know if it'll be good or not.

(e) This situation cannot occur because if u finishes before v, it violates one of the core elements of the DFS algorithm. The mechanism in which DFS explores a graph from a start vertex is that it selects an edge and explores nodes using a 'last in, first out' order. This is the 'depth' of depth first search. The algorithm takes a path as 'deeply' as possible until it reaches a node with no unexplored children. If there is an edge from u to v, then DFS will explore it and continue to search on v. DFS will not resume search on u until v finishes. If u finishes before v, there must be no directed edge between u and v. Obviously, a contradiction. Therefore, no situation exists where u finishes before v when they share an edge.

Part B

Problem 4-4. *Submit your implemented python script.*