Problem Set 4

All parts are due Thursday, April 10 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Your Name: Eric Klinkhammer

Collaborators: Name1, Name2

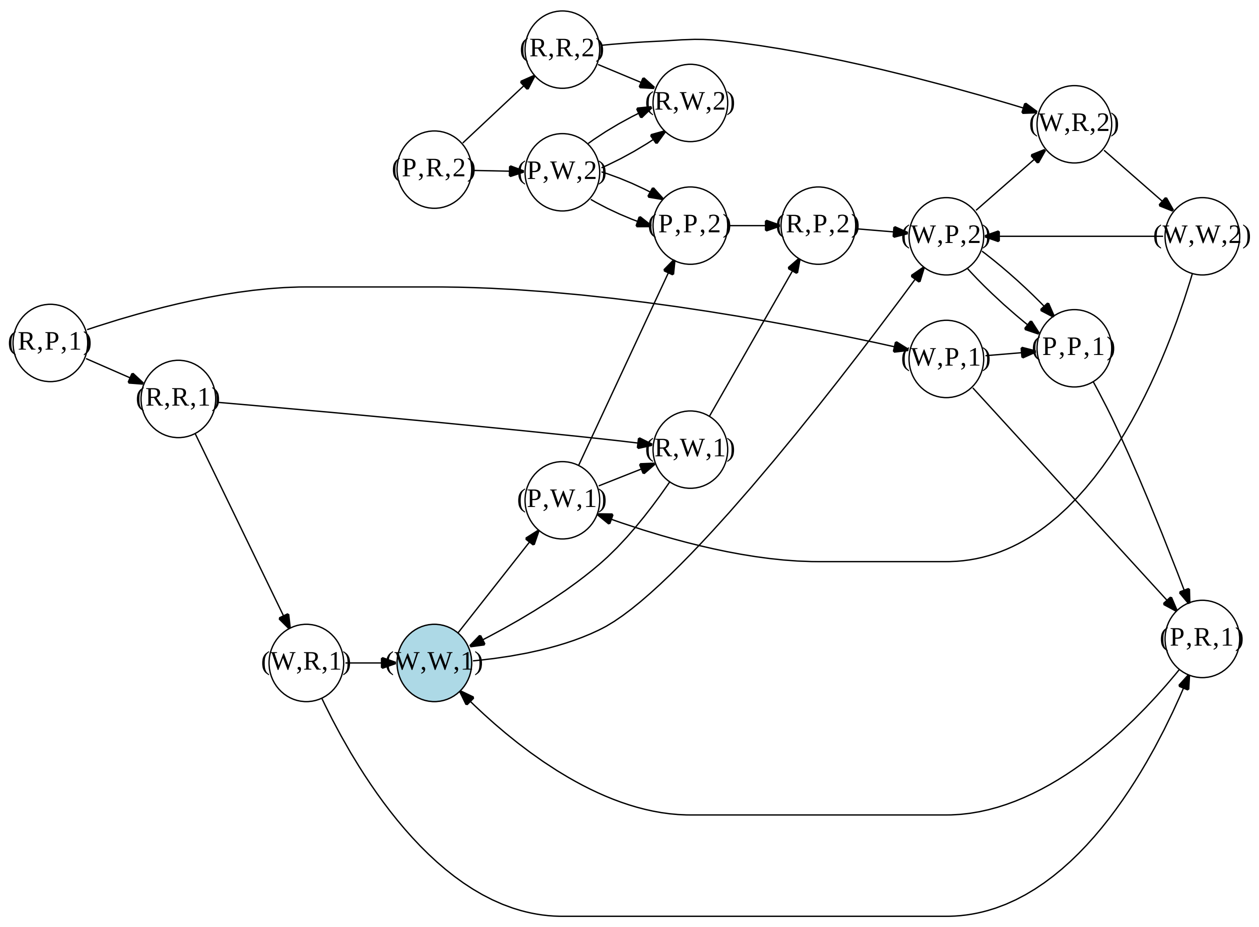
Part A

Problem 4-1. Using Breadth-First Search for Model-Checking an Algorithm

(a) Total Number of States

There are a total of 18 possible states; however, starting from (W,W,1), there are only 10 reachable states. There are three possible values for Color(1), three for Color(2), and two for Last, giving 3*3*2 = 18 possiblilities. The two with both colors being red are not allowed by the algorithm, as are states only reachable from those invalid states or capable of reacing them.

(b) Directed Graph Representation



(c) Adjacency List Representation
Node - Adjacency List
”(W,W,1)” - ”(P,W,1)”, ”(W,P,2)”;

”(P,W,1)” - ”(R,W,1)”, ”(P,P,2)”;

”(R,W,1)” - ”(W,W,1)”, ”(R,P,2)”;

”(P,P,2)” - ”(R,P,2)”;

”(R,P,2)” - ”(W,P,2)”;

”(W,P,2)” - ”(W,R,2)”, ”(P,P,1)”;

”(W,R,2)” - ”(W,W,2)”;

”(W,W,2)” - ”(P,W,1)”, ”(W,P,2)”;

”(W,P,2)” - ”(P,P,1)”, ”(P,R,1)”;

”(P,R,1)” - ”(W,W,1)”;

”(W,P,1)” - ”(P,P,1),”(P,R,1)”;

”(W,R,1)” - ”(P,R,1)”, ”(W,W,1)”;

”(R,P,1)” - ”(R,R,1)”, ”(W,P,1)”;

”(R,R,1)” - ”(W,R,1)”, ”(R,W,1)”;

”(R,R,2)” - ”(W,R,2)”, ”(R,W,2)”;

”(P,W,2)” -”(P,P,2)”, ”(R,W,2)”;

”(P,R,2)” - ”(R,R,2)”, ”(P,W,2)”;

”(P,W,2)” - ”(P,P,2)”, ”(R,W,2)”;

(d) Algorithm to Determine Reachable States
Determining the shortest path to all reachable nodes can be done with a modified breath first search.

```
initialize distance[] to infinity
init Queue q
enqueue initial node s
distance[s] = 0
while ( q is not empty ):
  u = dequeue()
  print u + d[u] # Outputs Node and length of shortest path to node
  for v in u.adjList():
    if ( distance[v] > d[u] + 1 ):
      enqueue(v)
      d[v] = d[u] + 1
```

(e) Output of Algorithm applied to Peterson’s Graph

$WW1$	0
$PW1$	1
$WP2$	1
$RW1$	2
$PP2$	2
$WR2$	2
$PP1$	2
$RP2$	3
$WW2$	3
$PR1$	3

(f) Safety
This proves the safety of the algorithm because none of the reachable states are the forbidden states. Because DFS exhaustively listed all of the reachable states, we know that it is impossible for the algorithm to be unsafe.

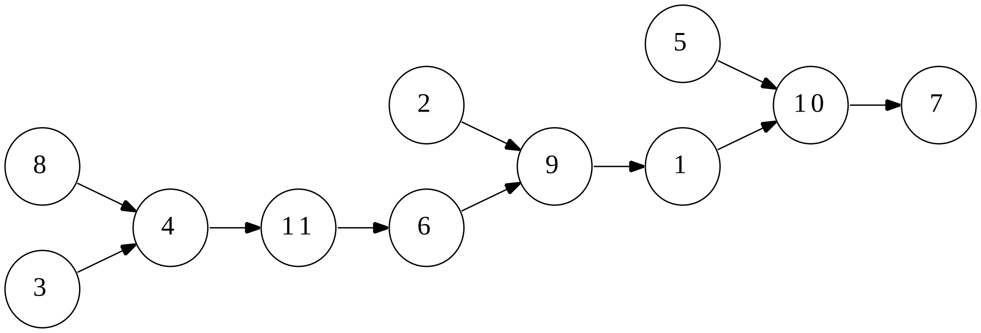
(g) Safety Algorithm without Adjacency List
The algorithm above can be modified slightly to determine the safety of the graph only without storing adjacency lists.
The idea is that instead of enqueueing the predetermined adjacency list, use the state transition rules to determine where to go.

```
initialize distance[] to infinity
init Queue q
enqueue initial node s
distance[s] = 0
while ( q is not empty ):
  u = dequeue()
  print u + d[u] # Outputs Node and length of shortest path to node
  for v in getListAdjacent(u):
    if ( distance[v] > d[u] + 1 ):
      enqueue(v)
      d[v] = d[u] + 1
      if v = RR1 or RR2:
        return false
return true

def getListAdjacent(Node u):
        # Returns the list of adjacent nodes by applying the rules
```

Problem 4-2. Depth First Lasagna

(a) Directed Graph



(b) Running DFS The following is a list of the discover and finish events running on the lasgna graph.
discover 1
discover 10
discover 7
finish 7
finish 10
finish 1
discover 2
discover 9
finish 9
finish 2
discover 3
discover 4
discover 11
discover 6
finish 6
finish 11
finish 4
finish 3
discover 5
finish 5
discover 8
finish 8

$$\begin{tikzcd}
5 \arrow[r] & 8 & & \\
3 \arrow[r] & 4 \arrow[r] & 11 \arrow[r] & 6 \\
2 \arrow[r] & 9 & & \\
1 \arrow[r] & 10 \arrow[r] & 7 &
\end{tikzcd}$$

I'm not entirely sure on what a DFS forest is supposed to look like. I made separate trees for each time I exhausted all possible descendants of my source node.

(c) Order of Discover Events
The order of discovered events is: 1, 10, 7, 2, 9, 3, 4, 11, 6, 5, 8. This is not a valid order for multiple reasons, including that it violates the requirement that 4 precede 10.

(d) Using DFS and Topological Sort
The reverse order of the finish steps is: 8, 5, 3, 4, 11, 6, 2, 9, 1, 10, 7. This order satisfies all of the constraints. It also seems like a reasonable order to cook a lasagna.

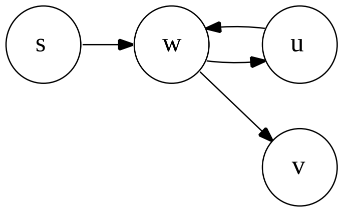
(e) Why DFS can be used to Topological Sort
There is no situation, when performing DFS on a DAG, that there is an edge from u to v, but v finishes after u.

It is a property of DFS that all descendants of a node will be processed before the parent node. So, since there is an edge from u to v, v is a descendent of u. However, if v did finish after u, then u must be a descendent of v. The only situation this is possible is if there exists a cycle between the two points, but that is impossible in an acyclic graph, so the described situation never occurs.

(f) DFS for Cyclic Graphs
This graph contains a path from u to v, but v appears before u in the DPS tree.

$$\begin{tikzcd}
s \arrow[r] & v \arrow[r, bend right] & u \arrow[l, bend right]
\end{tikzcd}$$

This graph contains a path from u to v, but v appears to the right of u.



Problem 4-3. Social Network Structure

(a) Find Communities
To find all communities in a network, I propose to loop through all the vertices, and when encountering one that is not in a community, perform BFS on that node, adding all nodes that are connected to that community.

```
init community_hash to all Nones
current_com = 0;
q = empty_queue()
list # This is a list of lists
for all v in V:
  if ( community_hash(v) == None ):
    community_hash(v) = current_com;
    list(current_com) = new list()
    while ( v != null ):
      for u in v.adjacent():
        if ( community_hash(u) == None ):
          community_hash(u) = current_com
          list[current_com].append(v)
          q.euqueue(u)
      v = q.dequeue()

return list
```

This algorithm runs in $O(|V| + |E|)$ time. The outer loop will cycle through all the vertices and do constant work, except when performing BFS, for a time of $O(|V|)$. That inner loop, which performs the BFS, does a constant amount of work per edge. While it does revisit nodes, it only vists each edge once, performing $O(|E|)$ work.

It is important here to note that, while it might appear to be doing $O(|VE|)$ work, this is not the case. Consider the two exteme example - the graph is completely connected. In the outer loop, only the first if check is true, and it traverses all the edges once in the inner loops. All the other if checks do a constant work unrelated to the number of edges.

(b) Conditions Root of DFS Tree is Hub
The root of a DFS tree is a hub if it has multiple children. A property of DFS is that all the descedants are visited before processing the parent. In the DFS tree, this means that all lower levels are processed before the root. In order for a node to have multiple children, it's right child must not be reachable from any node in the left's subtree.

Let's call this potential hub u. If u has only one child, then, because it is a root, it's removal will only disconnect itself from the graph, and it is not a hub. However, say it has two children, v and w. Since they are both descedents, and it is an undirected graph, there exists a path (though u) from v to w. However, v's subtree contains all nodes reachablefrom v without going through u. Therefore, since w is not in v, the only path from v to w is through u, and u serves as a hub for the community.

(c) Conditions that a Node in a DFS Tree is a Hub
If a node, not a root, is a hub, then it holds that there is no descedant of that node that has a back edge to an ancestor of the hub node. This is true because if such a back edge existed, there would be a path (through that back edge) effectively around the hub node, meaning it wouldn't be a hub. In order for it to be a hub, its removal has to completely separate all nodes below it from the rest of the tree, which only happens in the absense of back edges ”over” the hub node.

(d) Identify Hubs
To identify all of the hubs in a graph in $O(|V| + |E|)$ time, first construct the DFS tree, together with the corresponding arrays denoting time visited and time finished, in $O(|V| + |E|)$ time (as in book). Additionally, when constructing this tree, make it doubly linked.

Next, traverse the tree backwards (the order DFS finishes nodes) and store, for each node, the most ancestral node it connects to. There are two possiblities. Either that node has a back edge that links it to a higher (later time finished, earlier visit) node that is higher than all of its children, or, through its children, it is linked to such a node. This node can be the parent node.

When determining this number, you would have to loop through all of a nodes adjacency list to determine the potential backedge node, and then compare that node to its children's highest back edge node, choosing the greater. In this mannor, after traversal once through the tree (in $O(|V| + |E|)$ time) every node has stored with it the highest node in the tree it can form a path to.

Finally, traverse the list of nodes. If a node has any children whose highest ancestral node is that node, or in other words if there is no path with back edges past the node in question, then that node is a hub. Another way of saying that is if the most ancestral node that the children of a hub can form a path to is not an ancestor of the hub, then the hub is a real hub.

A crucial component of this algorithm is that with the arrays recording time visited and time finished, it is possible to determine, in constant time, whether a node is an ancestor or not.