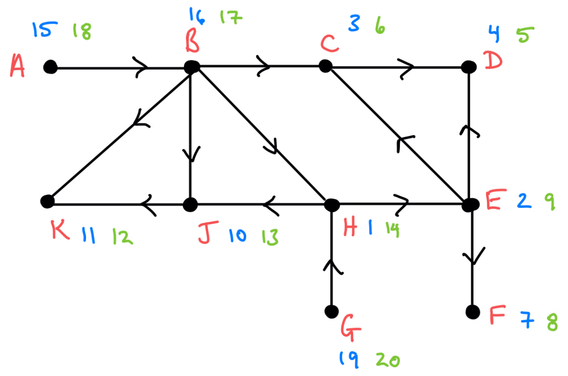
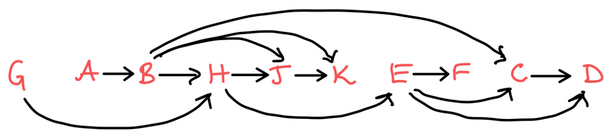
Practice Set 12: solutions

.

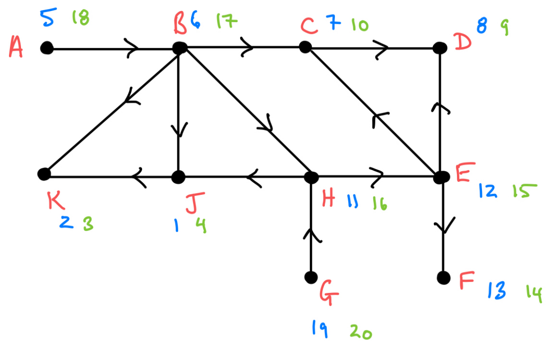
Problem 1

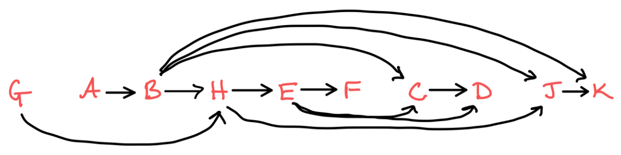
If DFS starts at vertex H, the time stamps are shown below. The resulting topological sort is shown on the right.



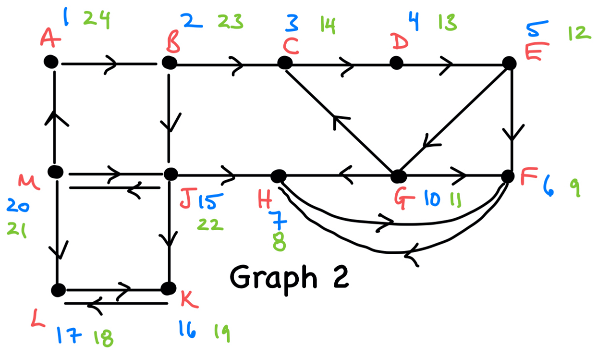


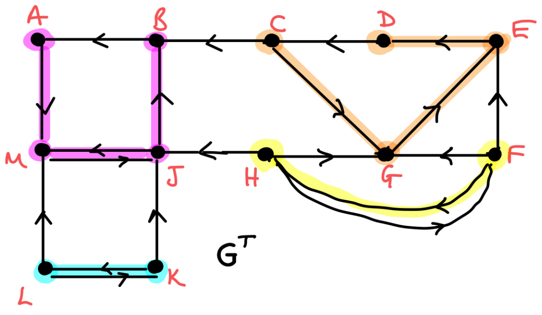
If we start at vertex J, a different topological sort would result.





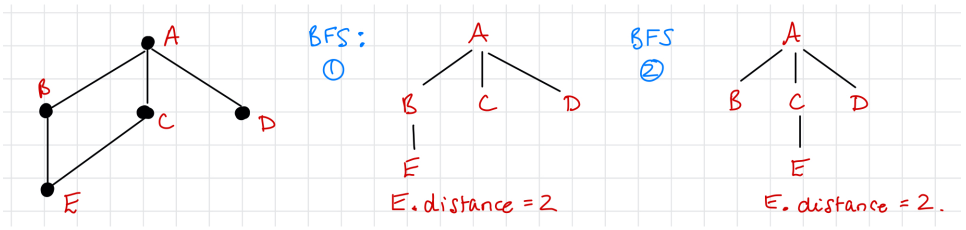
For graph 2, we first perform DFS on the original graph, then perform DFS(G^T) processing the vertices in decreasing order of finish time. The resulting DFS trees represent the strongly connected components. There are exactly 4 strongly connected components: $\{A, B, J, M\}, \{C, D, E, G\}, \{L, K\}, \{H, F\}$.





Problem 2

- Yes, it is possible to have two different BFS trees for the same graph. The BFS tree shape depends on the order in which we process the neighbours of a vertex. In the first example below, the neighbors of A are processed as B, C, D and in the second example, they are processed as C, B, D. In both cases, the distance attribute of the vertices remains the same.



- An undirected connected graph has only back edges in the DFS tree. As DFS executes, it visits all vertices that are connected by a path to the source vertex. Therefore any edge that is discovered is either to a vertex that is not yet visited (in which case it would be a tree edge), or to a vertex that has already been visited. If it has already been visited, then it must be an ancestor, and therefore the edge must be a back edge. In fact, when we discover a back edge in an undirected graph, the back edge indicates that there is a cycle in the graph.

Problem 3

We can run DFS on the graph and immediately stop as soon as we find a back edge. Recall that a back edge would be an edge that leads to a node that was already visited (but not the node's parent!), and therefore it must represent a cycle. The algorithm below takes as input any vertex u in the graph G, and determines if there is a cycle that is reachable from vertex u.

```
Find-cycle(u)
    u.visited = true
    for each v in Adj[u]
        if v.visited = false
            v.parent = u
            if Find-cycle(v) = true
                return true
        else if v ≠ u.parent   #Found a back-edge
            return true
    return false   #No cycle was found, return false
```

The runtime of this is $O(V)$ instead of $O(V + E)$ since we don't loop through all the neighbors of a vertex unless they go to new nodes.

Problem 4

Each vertex v will have an additional attribute $v.children$ which is a list of all the children of the DFS tree of vertex v. In the initialization of DFS, we assume all these lists are initialized to empty. The updated code for DFS is shown below:

```
Dfs-visit-with-children(u)
   u.visited = true
   for each v in Adj [u]
      if v.visited = false
         v.parent = u
         u.children.add(v)
         DFS-visit-with-children(v)
```

Now that we have a list of children for each vertex in the DFS tree, we can use this list to print out the vertices of the DFS tree using a pre-order traversal:

```
PrintDFS(u)
   Print u.key
   for each v in u.children:
      PrintDFS(v)
```

Problem 5

If an adjacency matrix were used instead of an adjacency list, the main while loop would be re-written with the updated code show in red. Notice that instead of looping through all vertices in the adjacency list, instead we loop through all vertices in the adjacency matrix and verify which vertices are actually neighbors of u:

```
while $Q \neq empty$
    u = DEQUEUE(Q)
    for all $v$ in $V$
        if $Adj[u][v] = 1$
            if $v.visited = false$
                $v.visited = true$
                $v.distance = u.distance + 1$
                $v.parent = u$
                Add $v$ to $u.children$
                $ENQUEUE(Q, v)$
```

When a node u is dequeued during BFS, the *entire row* of the adjacency matrix would need to be searched in order to identify the neighbors of u. The runtime of the entire for-loop is now $O(V)$ instead of $O(E)$. Since the for loop is executed once per vertex, the runtime of BFS is now $O(V^2 + V) = O(V^2)$. For sparse (few edges) graphs, the adjacency list approach from class has a better runtime: $O(V + E)$. Therefore using the adjacency list is more efficient when the number of edges in the graph is much less than $O(V^2)$.

Problem 6

Use DFS and color the vertices as you traverse the graph. Start by coloring the initial node black ($u.color = black$), and at every step, color a new visited node the *opposite* color of the parent. If you come across visited vertex, check if it is the same color as the current node, and if so, return that it is not possible.

```
DFS-visit-BW(u)
    u.visited = true
    for each v in Adj[u]
        if v.visited = false
            if u.color = black
                v.color = white
            else
                v.color=black
            v.parent = u
            if DFS-visit-BW(v) = false
                return false
        else if v.color = u.color
            return false
    return true
```

Problem 7

As in the above problem, we start with any vertex as the source vertex, and color it black. Then each other vertex we encounter is colored the *opposite color* from its parent. If ever we encounter two adjacent nodes of the same color, we return false.

Initialize BFS as usual:

For all vertices in G, initialize them as unvisited: for all $v \in V$ set $v.visited = false$. Initialize the parent pointers to NIL for each vertex. Initialize the list $v.children$ to NIL for each vertex. Pick any vertex s as the starting node. Set $s.visited = true$ and $s.distance = 0$ and set s.color = BLACK. Add s to Q.

Carry out main while loop of BFS:

```
while $Q \neq empty$
```

```
        u = DEQUEUE(Q)
        for each $v$ in $Adj[u]$
            if $v.visited = false$
                $v.visited = true$
                if $u.color = black$
                      $v.color = white$
                else
                      $v.color = black$
                $v.distance = u.distance + 1$
                $v.parent = u$
                Add $v$ to $u.children$
                $ENQUEUE(Q, v)$
            else
                if $v.color = u.color$
                      return FALSE
    return TRUE
```

Problem 8

- A *sink* vertex is a vertex that has out-degree 0 . It is like a dead-end, in that there are no edges out of the sink. Now we explain why every DAG must have a sink vertex. Suppose you walk around the graph from vertex to vertex. Assuming there were no sink, you must never hit a dead end, which means you are able to keep walking forever. But this is impossible because there are no cycles in a DAG. Therefore, we arrive at a contradiction, and there must be a sink.

  A *source* vertex is a vertex that has in-degree 0. Similarly, there must be a source vertex. If we flip the directions of each vertex, then we can use the same argument as above.

- The idea is to start by find a *source* vertex, to remove it and place it in the topological sort. The steps below use a linked list L to keep track of the vertices with indegree 0. The topological sort is returned in the list T:

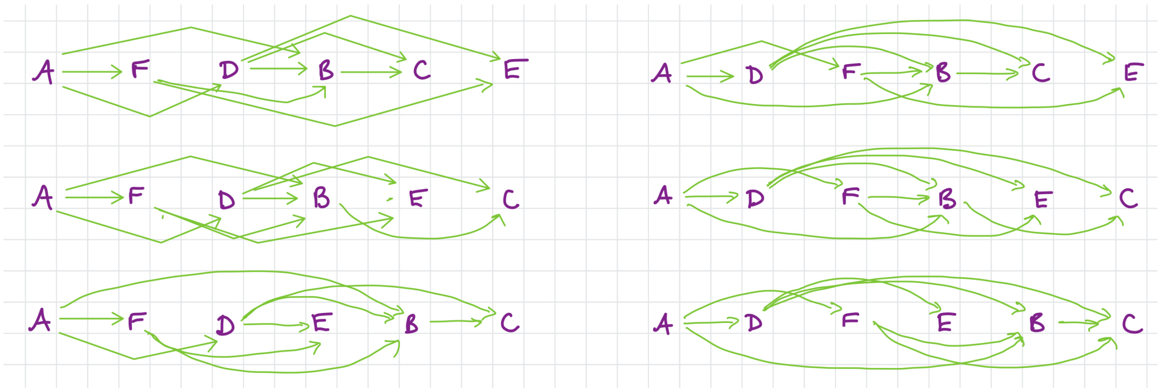
  1. Compute in-degree of each vertex and store the in-degree in $v.indegree$.
  2. For all $v \in V$, if $v.indegree = 0$ then add v to the list L.
  3. Repeat until L is empty:

  ```
      Remove a vertex $u$ from $L$ and place it at the end of $T$.
      For all $v$ in $Adj\ [u]$
          $v.indegree = v.indegree - 1$.
          if $v.indegree = 0$
              add $v$ to $L$
  ```

  Steps 1 and 2 above take time $O(V + E)$. In step 3, each vertex is removed from L once, and the for loop is executed once for all neighbors of the removed vertex. Therefore the for loop is executed a total of $O(E)$ times. The overall runtime of the algorithm is $O(V + E)$.

Problem 9

Vertex A must be the first vertex, since it is the only *source* in the graph. After vertex A, we could follow with either vertex F or vertex D. There are three possible topological sorts in each of these cases, giving a total of 6 possible topological sorts. The diagrams are shown below:



Problem 10

We saw in class how to use DFS to find the strongly connected components of a graph. In this problem, it remains to determine how to store the appropriate information so that we can print out the vertices in each SCC. In problem 4 we updated DFS-visit to store the children of a vertex. We will use this the usual DFS-visit algorithm with time-stamps on the first call to DFS, and then use the updated DFS-visit algorithm from problem 4 when the input is G^T, so that we keep track of each strongly connected component.

```
PrintSCC(G)
Step 1: Carry out usual DFS using DFS-visit with time stamps
    for each $v$ in $V$
        if $v.visited = false$
            DFS-visit(v)
Step 2: Create graph $G^T$
    Copy vertices from V to the set of vertices in $G^T$
    For each edge $e = (u, v)$ in $E$
        Add edge $(v, u)$ to $E^T$
        Add vertex $u$ to $Adj[v]$.
    For each $v$ in V   # reset all vertices as unvisited.
        $v.visited = false$
Step 3: Sort vertices in order of decreasing finish time
    $L$ = Sort vertices in $V$ by decreasing $v.finish$.
Step 4: Run DFS on the transpose graph. Each restart corresponds to another SCC
    count = 1
    For each $v$ in $L$
        if $v.visited = false$
            print("Component", count)
            DFS-visit-with-children(v)
            PrintDFS(v)
            count = count +1
```

Problem 11

BFS: return true if there is a path from s to t.

Use vertex s as the source vertex. The idea is to simply return true as soon as we find vertex t. We update the while loop of BFS below:

```
while $Q \neq NIL$
    u = DEQUEUE(Q)
    for each v in Adj[u]
        if v = t return true
        else if v.visited = false
```

```
            v.visited = true
            v.parent = u
            Add v to u.children
            ENQUEUE(Q,v)
    return false
```

DFS: return true if there is a path from s to t.

We can update $DFS(s)$ to $DFS(s,t)$ by simply passing the target vertex t to DFS-visit(s,t). The update the DFS-visit is given below:

```
DFS-visit(u,t)
  u.visited = true
  if u= t return true
  for each v in Adj[u]
    if v.visited = false
      v.parent = u
      if DFS-visit(v,t) = true
        return true
  return false
```

BFS: return true if there is a path of alternating colors from s to t

Initialize BFS as usual using vertex s. Next, we update the main while loop of BFS so that it only visit neighbors that are the opposite color of the parent.

Carry out main while loop of BFS:

```
    while $Q \neq empty$
      u = DEQUEUE(Q)
      for each $v$ in $Adj[u]$
        if $v.visited = false$ and $v.color$ is the opposite of $u.color$
          if $v = t$ return true
          $v.visited = true$
          $v.distance = u.distance + 1$
          $v.parent = u$
          Add $v$ to $u.children$
          $ENQUEUE(Q, v)$
  return False
```

DFS: return true if there is a path of alternating colors from s to t.

We can use $DFS(s)$ and only visit neighbors of the opposite color of the current node. We stop when we reach target t. As usual, the first call to DFS-visit is with the source vertex s: DFS-visit(s,t). The updated algorithm is given below:

```
DFS-visit(u,t)
  u.visited = true
  if u= t return true
  for each v in Adj[u]
    if v.visited = false and $v.color$ is the opposite of $u.color$
      v.parent = u
      if DFS-visit(v,t) = true
        return true
  return false
```

Problem 12

Almost the same idea as in the solution to problem 3, in that we use DFS and look for back edges. Notice that for directed graphs, a back edge is an edge that leads to a vertex that is already visited, but not yet finished. Therefore we run DFS and keep track of the finish times during DFS-visit. When we encounter an edge that leads to a vertex that is visited, but does *not yet* have a finish time, then we know this is a back edge, and therefore we have found a cycle.

```
Find-directed-cycle(u)
    u.visited = true
    time = time+1
    for each v in Adj[$u$]
        if v.visited = false
            v.parent = u
            if Find-directed-cycle(v) = true
                return true
        else if v.finish = NIL
            return true
    u.finish = time
    return false
```

Problem 13

We use a distance attribute as in BFS. The attribute $x.distance$ is the length of the path from the original source vertex u to vertex x in the DFS tree. All distance attributes are initialized to 0. We can update the DFS-visit algorithm so that it keeps track of the maximum distance among all of the children of a node, and return the largest.

```
DFS-visit-distance(u)
        $u.visited = true$
        maxdist = 0
        For each $w \in Adj[u]$
            If $w.visited = false$
                $w.parent = u$
                $w.distance = u.distance + 1$
                dist = DFS-visit-distance(w)
                maxdist = max(dist, maxdist)
        return max(maxdist, u.distance)
```

Problem 14

Model this problem as a directed graph G, where the tourist attractions and hotel (H) are vertices, and the directed edges represent the shuttles between the sites. Notice that when we leave our hotel and visit a sequence of tourist attractions, and come back to our hotel, we are following along a directed cycle from vertex H in the directed graph. Since we are lazy, we want to come back to the hotel as fast as possible, which means we want the **shortest** cycle from H. We saw in class how to use DFS to find a cycle. We can use BFS in this case, since it will search out from vertex H one unit of distance at a time, and find the first available cycle edge that leads back to H. The algorithm is almost the same as the original BFS algorithm, adding only a constant number of operations in the while loop and therefore runs in time $O(V + E)$. We use H as the source, and we don't need to store children or parent vertices.

```
LazyTourist(G)
       set $H.distance = 0$
       set $H.visited = true$
       add $H$ to $Q$
       while $Q \neq empty$
           u = DEQUEUE(Q)
           for each $v$ in $Adj[u]$
               if $v.visited = false$
                   $v.visited = true$
                   $v.distance = u.distance + 1$
```

```
    $ENQUEUE(Q, v)$
else if $v = H$        # found a directed cycle back to the hotel
    print “Found a route back to the hotel”.
    Number of sites visited is : $u.distance$
```