Week 5 Applied Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the applied classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to these problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, pseudocode may be provided where it illustrates a particular useful concept.

Implementation checklist

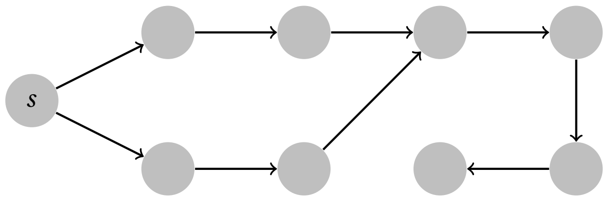
It will be most beneficial for your learning if you have completed this checklist **before** the tutorial.

By the end of week 5, write Python code for:

- Breadth-first search
- Depth-first search

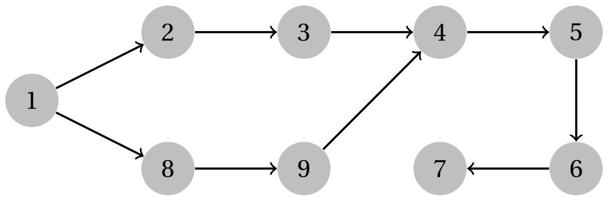
Problems

Problem 1. **(Preparation)** Label the vertices of the following graph in the order that they might be visited by a depth-first search, and by a breadth-first search, from the source s.

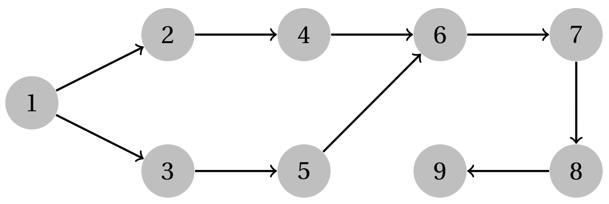


Solution

There are two possible solutions for depth-first search depending on which order you traverse the edges. One valid order is the following.



The other valid order is to visit 8 & 9 before 2 & 3. A valid order for breadth-first search is the following.



The other valid order swaps nodes 2 and 3 with each other, and nodes 4 and 5 with each other.

Problem 2. Write pseudocode for an algorithm that given a directed graph and a source vertex, returns a list of all of the vertices reachable in the graph from that source vertex. Your algorithm should run in $O(V+E)$ time.

Solution

This problem can be solved with a depth-first search or breadth-first search. The vertices reachable from a given vertex are simply those that are visited by a search when that vertex is the starting node. So we simply perform a DFS from s and then return a list of the nodes that were visited.

```
function REACHABLE(G = (V, E), s)
    Set visited[1..$n$] = False
    DFS(S)
    Set reachable = empty array
    for each vertex u = 1 to $n$ do
        if visited[u] then
            reachable.append(u)
        end if
    end for
    return reachable
end function

function DFS(u)
    visited[u] = True
    for each vertex v adjacent to u do
        if not visited[v] then
            DFS(V)
        end if
    end for
end function
```

Problem 3. Devise an algorithm for determining whether a given undirected graph is two-colourable. A graph is two-colourable if each vertex can be assigned a colour, black or white, such that no two adjacent vertices are the same colour. Your algorithm should run in $O(V+E)$ time. Write pseudocode for your algorithm.

Solution

To two colour a graph, the key observation to make is that once we pick a colour for a particular vertex, the colours of all of its neighbours must be the opposite. That is, once we decide on a colour for one vertex, all other reachable vertices must be set accordingly, we do not get to make any more decisions. Secondly, it does not matter whether we decide to set the first vertex to white or to black, since changing from one to the other will just swap the colour of every other vertex. With these observations in mind, we can proceed greedily.

Let's perform depth-first search on the graph, and for each vertex, if it has not been coloured yet, select an arbitrary colour and then recursively colour all of its neighbours the opposite colour. If at any point a vertex sees that one of its neighbours has the same colour as it, we know that the graph is not two-colourable. Here is some pseudocode that implements this idea.

```
function TWO_COLOUR(G = (V, E))
    Set colour[1..$n$] = null
    for each vertex u = 1 to $n$ do
        if colour[u] = null then
            if DFS(u, BLACK) = False then
                return False
            end if
        end if
    end for
    return True, colour[1..$n$]
end function

// Returns true if the component was successfully coloured
function DFS(u, c)
    colour[u] = c
    for each vertex v adjacent to u do
        if colour[v] = c then    // A neighbour has the same colour as us!
            return False
        else if colour[v] = null and DFS(v, opposite(c)) = False then
            return False
        end if
    end for
    return True
end function
```

Here, opposite(c) is a function that returns `WHITE` or `BLACK` if c is `BLACK` or `WHITE` respectively.

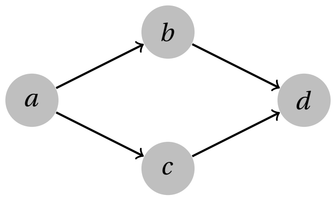
Problem 4. This problem is about cycle finding as discussed in the course notes.

(a) Explain using an example why the algorithm given for finding cycles in an undirected graph does not work when applied to a directed graph.

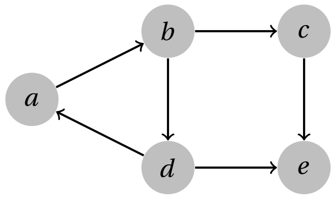
(b) Describe an algorithm based on depth-first search that determines whether a given directed graph contains any cycles. Your algorithm should run in $O(V + E)$ time. Write pseudocode for your algorithm.

Solution

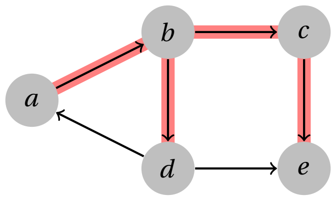
The undirected cycle finding algorithm works by checking whether the depth-first search encounters an edge to a vertex that has already been visited. This works fine for undirected graphs, but when the graph is directed, this might yield false positives since there could be multiple paths to the same vertex with the edges in the same direction, which does not constitute a cycle. For example, the algorithm would falsely identify this as a cycle:



To correct this, we need to think a bit more carefully about the edges that depth-first search examines during traversal. In a different example like the following, the cycle a, b, d if identified would in fact be correct, while b, c, e, d would not.



How do we distinguish between the two? Suppose that the edges traversed by the search are those denoted in red below.



The critical observation is that when the search looks along the edge (d, e) and sees that e has already been visited, we see that the branch that visited e is a different branch than the one that visited d. Because of this, the edges will be in the same direction, and hence not a cycle. However, when the search looks along the edge (d, a) and notices that a has already been visited, we observe that a is a parent of the current branch of the search tree. This means that there is a path from $a \rightsquigarrow d$, and since we just discovered an edge $d \to a$, we have found a directed cycle!

So, to write an algorithm for directed cycle finding we need to perform a depth-first search and keep track of which branches of the search tree are still active, and which ones are finished. Whenever we encounter an edge to an already visited vertex, we identify it as a cycle only if the target vertex is still active, otherwise it is part of a different branch and hence not part of a cycle. Therefore instead of marking each vertex as visited or not, we will have three possible states, unvisited, inactive, and active. A vertex is active if its descendants are still being explored. Once we finish exploring a node's descendants, we mark it as inactive. The algorithm might then look like the following

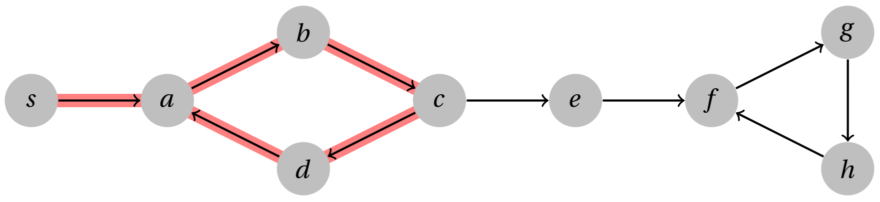
```
function CYCLE_DETECTION(G = (V, E))
    Set status[1..$n$] = Unvisited
    for each vertex u = 1 to $n$ do
        if status[u] == Unvisited and DFS(u) then
            return True
        end if
    end for
    return False
end function

function DFS(u)
    status[u] = Active
    for each vertex v adjacent to u do
        if status[v] = Active then    // We found a cycle
            return True
        else if status[v] = Unvisited and DFS(v) = True then
            return True
        end if
    end for
    status[u] = Inactive    // Finished this branch - mark as inactive
    return False
end function
```

Problem 5. Describe an algorithm for finding the **shortest** cycle in an unweighted, directed graph. A shortest cycle is a cycle with the minimum possible number of edges. You may need more than linear time to solve this one. Write pseudocode for your algorithm.

Solution

Since we are asking for the shortest cycle, the first thing that we should try is breadth-first search since it visits vertices in order of distance. Suppose that we run the ordinary cycle detection algorithm but using a breadth-first search in place of depth-first search. Will this find the shortest cycle for us? Unfortunately not. See the following example in which we start the search from the source s.



The cycle a, b, c, d is of length four and a distance of one from the source, and it will be found by the time the search reaches a distance five. However, the shortest cycle f, g, h is distance five away, and will not be found until the search reaches distance eight.

To fix this, let's simply perform multiple breadth-first searches, one from every possible source vertex. If the source vertex is contained within a cycle of length k, then the cycle will be detected when the search reaches distance k, before any longer cycle has had a chance to be detected. By trying every possible source, we are therefore guaranteed to find the shortest cycle. This solution will take $O(V(V+E))$ time since we perform $|V|$ breadth-first searches and each one takes $O(V+E)$ time.

Notice finally that if we are searching from a vertex contained inside the shortest cycle, then the first already-visited vertex encountered will in fact be the source itself, so we can just check for that in our implementation.

```
function SHORTEST_CYCLE(G = (V, E))
    Set best_found = $\infty$
    for each vertex s = 1 to $n$ do    // Try every vertex as the source
        best_found = min(best_found, BFS(s))
    end for
    return best_found
end function

function BFS(s)
    Set dist[1..$n$] = $\infty$
    dist[s] = 0
    Set queue = Queue()
    queue.push(s)
    while queue is not empty do
        u = queue.pop()
        for each vertex v adjacent to u do
            if v = s then    // We made it back to the source – we found a cycle!
                return dist[u] + 1
            else if dist[v] == ∞ then
                dist[v] = dist[u] + 1
                queue.push(v)
            end if
        end for
    end while
    return ∞
end function
```

Have a think about how you would solve this problem for undirected graphs. The same general ideas work, but some extra care is needed to account for some situations that do not appear in the directed case.

Problem 6. In this question we consider a variant of the single-source shortest path problem, the *multi-source shortest path* problem. In this problem, we are given an unweighted graph and a set of many source vertices. We wish to find for every vertex v in the graph, the minimum distance to any one of the source vertices. Formally, given the sources $s_1, s_2, ...s_k$, we wish to find for every vertex v

$$\mathrm{d}[v] = \min_{1 \le i \le k} \mathrm{dist}(v, s_i).$$

Describe how to solve this problem using a modification to breadth-first search. Your algorithm should run in $O(V + E)$ time.

Solution

The solution to this problem is rather simple. We perform a breadth-first search on the graph, but instead of beginning with a single vertex in the queue at distance zero, we place all of the given source vertices in the queue with distance zero. The algorithm then proceeds exactly the same. This works because the breadth-first search will discover all of the vertices that are a distance one from every source vertex before discovering vertices of distance two and so on. If a vertex has already been discovered via another source, then the breadth-first search can safely ignore it since the first source to find it must be the closest one.

An alternate solution is to add a new *super source* vertex to the graph and connect it to all of the given sources. After performing breadth-first search from the super source, the distances to all of the vertices will be one greater than the answer, since we had to travel along one additional edge from the super source to the real sources. Therefore we can subtract one from each of the distances and return those.

Problem 7. Recall the definition of two-colourability from Problem 3. Describe an algorithm for counting the number of valid two colourings of a given undirected graph.

Solution

First, run the algorithm from Problem 3 to check whether the graph is two colourable at all. If it is not, return zero. Otherwise, observe that after selecting a colour for a vertex, every other vertex in the same component is fixed to a particular colour. We can therefore colour each component two ways, hence the total number of possible colourings is

$$2^{\text{number of connected components}}.$$

We can compute the number of connected components using depth-first search, so this algorithm takes $O(V + E)$ time.

Problem 8. Write pseudocode for an algorithm that counts the number of connected components in an undirected graph that are cycles. A cycle is a non-empty sequence of edges $(u_1, u_2), (u_2, u_3), ...(u_k, u_1)$ such that $u_1, u_2, \ldots, u_k$ are all distinct vertices.

Solution

To solve this problem, we notice that for a component to be a cycle, it must be the case that the degree (number of adjacent edges) of every vertex is exactly two. Armed with this fact, the solution to this problem is to perform a depth-first search much like the ordinary connected components algorithm and add in a check to see whether the degree of a vertex is not two.

```
function COUNT_CYCLE_COMPONENTS(G = (V, E))
    Set visited[1..$n$] = False
    Set num_components = 0
    for each vertex u = 1 to $n$ do
        if visited[u] == False then
            if DFS(u) then
                num_components = num_components + 1
            end if
        end if
    end for
    return num_components
end function

// Return true if the component containing u is a simple cycle
function DFS(u)
    visited[u] = True
    Set is_cycle = True if degree(u) = 2 else False
    for each vertex v adjacent to u do
        if visited[v] == False then
            if DFS(v) = False then
                is_cycle = False
            end if
        end if
    end for
    return is_cycle
end function
```

Supplementary Problems

Problem 9. Write pseudocode for a non-recursive implementation of depth-first search that uses a stack instead of recursion.

Solution

The function looks almost identical to the recursive version. We simply replace the recursive calls by pushing the corresponding vertex onto the stack, and loop until the stack is empty.

```
function DFS(u)
    Create empty stack
    stack.push(u)
    while stack is not empty do
        u = stack.pop()
        visited[u] = True
        for each vertex v adjacent to u do
            if not visited[v] then
                stack.push(v)
            end if
        end for
    end while
end function
```

Problem 10. Argue that the algorithm given in the course notes for detecting whether an undirected graph contains a cycle actually runs in $O(V)$ time, not $O(V+E)$ time, i.e. its complexity is independent of $|E|$.

Solution

If the given graph is acyclic, then $E \leq V-1$, so the ordinary depth-first search complexity of $O(V+E)$ is just $O(V)$. If instead the graph contains a cycle, then the depth-first search will find it as soon as it examines an edge to an already-visited vertex. Up until this point, none of the edges examined led to already-visited vertices, so they formed a forest of depth-first search trees, which means that there were at most $V-1$ of them. After we examine the cycle-producing edge, the algorithm immediately terminates, and hence it did at most $O(V)$ work in total.

Problem 11. Consider a directed acyclic graph representing the hierarchical structure of n employees at a company. Each employee may have one or many employees as their superior. The company has decided to give raises to m of the top employees. Unfortunately, you are not sure exactly how the company decides who is considered the top employees, but you do know for sure that a person will not receive a raise unless all of their superiors do.

Describe an algorithm that given the company DAG and the value of m, determines which employees are guaranteed to receive a raise, and which are guaranteed to not receive a raise. Your algorithm should run in $O(V^2+VE)$ time.

Solution

We can rephrase this problem in terms of topological orderings. An employee will receive a raise if they are among the top m employees, which means that they must be in the first m elements of a topological order. However, we can not simply compute a topological order and check the first m elements since the order is not guaranteed to be unique. More concretely, an employee is guaranteed a raise if they are in the first m elements of **every** possible topological order. Similarly, an employee is guaranteed to not get a raise if they are never in the first m elements in **any** topological order.

In order to determine this, consider a particular vertex v in a directed acyclic graph. The vertices that must come after it in a topological ordering are all of the vertices that it can reach, since they are all of its dependants. Conversely, all of the vertices that must come before v are the ones that can reach v. All other vertices could go before or after v.

Therefore we want to count for each employee, the number of employees that are reachable in the company DAG, and conversely, the number of employees that can reach them in the company DAG. We can count the number of vertices reachable from a particular employee in $O(V+E)$ time with a simple depth-first search. Similarly, to count the number of employees that reach them, we can perform a depth-first search in the company DAG with all of the edges reversed. Let the number of reachable employees be r, and the number of employees that reach them be s. An employee is guaranteed to be in the first m elements of any topological order if and only if

$$r \geq n-m.$$

Similarly, an employee is guaranteed to not be in the first m elements if and only if

$$s \geq m.$$

We can therefore run this procedure for every employee and output the results. We perform two depth-first searches per employee, taking $O(V+E)$ time each, and hence the total time complexity is $O(V^2+VE)$ as required.

Problem 12. A Hamiltonian path in a graph $G = (V, E)$ is a path in G that visits every vertex $v \in V$ exactly once. On general graphs, computing Hamiltonian paths is NP-Hard. Describe an algorithm that finds a Hamiltonian path in a directed acyclic graph in $O(V + E)$ time or reports that one does not exist.

Solution

A Hamiltonian path must begin at some vertex and then travel through every other vertex without ever revisiting a previous one. The first thing to note is that the starting vertex must therefore be a vertex with no incoming edges, since such a vertex can never be visited in any other way by the path. This means that a Hamiltonian path will not exist if there are multiple such vertices. This should remind us of something similar, topological orderings!

Let's argue that a Hamiltonian path must be a topological ordering of a graph. Consider the sequence of vertices in a Hamiltonian path. If a vertex has an edge that travels to a previous vertex (meaning that the two are out of topological order), then this would produce a cycle in the graph, which is impossible since the graph is acyclic by assumption. Therefore a Hamiltonian path, if one exists, is a topological order of the graph.

Suppose that a Hamiltonian path exists. Therefore, for every pair of vertices in the graph, one must be an ancestor of the other. This means that for any pair of vertices, we can determine which one must appear earlier in any possible topological order. Therefore, there is a unique topological order.

Finally, suppose that there is a unique topological order, then this implies that there are no pairs of vertices in the order that are not adjacent, since otherwise they could be swapped without breaking the dependency constraints. Since every pair of vertices in the order must be adjacent, this constitutes a Hamiltonian path.

We can conclude that a Hamiltonian path exists if and only if the graph has a unique topological order, and if so, the Hamiltonian path is the topological order. So we can just find any topological order (in case there is a unique topological order, it will be the one found) and test whether there is a path visiting all vertices in that order (in which case a Hamiltonian path exists).

Problem 13. (Advanced) Given a directed graph G in adjacency matrix form, determine whether it contains a "*universal sink*" vertex. A universal sink is a vertex v with in-degree $|V|-1$ and out-degree 0, ie. a vertex such that every other vertex in the graph has an edge to v, but v has no edge to any other vertex. **Your algorithm should run in $O(V)$ time.** Note that this means that you cannot read the entire adjacency matrix and meet the required complexity.

Solution

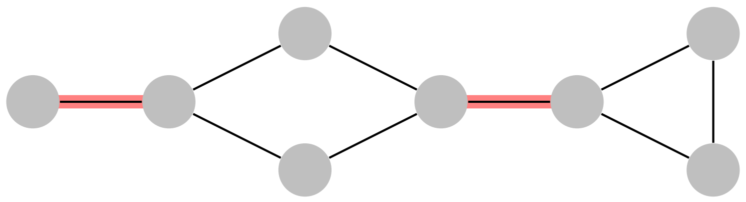
This problem is quite tricky, as we are not allowed to even read the entire input if we are to solve it in the required complexity. This means that we need a way to eliminate nodes from consideration very quickly. Let's make some observations that will help. First of all, there can only be one universal sink if any, since if every vertex has an edge to v, then no other vertex has no outgoing edges. For a node to be a universal sink, it must have no outgoing edges, which means that its row in the adjacency matrix must contain all zeros. It must also have every possible incoming edge, which means that its column should contain all ones except on the main diagonal. This means that if we look at an entry $A[i][j]$ in the adjacency matrix, we can deduce the following:

1. If $A[i][j] = 1$, then vertex i is definitely not a universal sink, since it has an outgoing edge.
2. If $A[i][j] = 0$ and $i \neq j$, then vertex j is definitely not a universal sink, sink it is missing an incoming edge.

This means that every time we look at an entry in the matrix (except the diagonal), we get to discard someone from consideration. This is good. We just need a way to never consider the same vertex twice, and we are in the clear. To do so, let's start by making a linked list containing all of the vertices. We will also make an array `bad`$[1..n]$ such that `bad`$[v]$ is **True** if we have eliminated v from consideration.

With all of this set up, we do the following. For each column of the adjacency matrix, check whether this vertex has already been discarded, and if so, move to the next column. If it has not, examine the entries in this column. If we find a 1, then mark that row as bad. If we find a 0 not on the diagonal, mark the current column as bad and move on. This is almost fast enough, except that we might scan top-to-bottom too many columns, leading to $O(V^2)$ complexity. This is where the linked list comes in. Whenever we wish to scan a column, scan only the elements presently in the linked list (except the diagonal). Therefore, for every element that is scanned, one bad vertex is encountered. The bad vertex is removed from the linked list so that we never have to consider it ever again (thus the total complexity of this part will be $O(V)$). When there is only one remaining candidate vertex, using the adjacency matrix, we can trivially check in time $O(V)$ if it is a universal sink or not.

Problem 14. (Advanced) This problem is about determining another interesting property of a graph G, namely its *bridges*. A bridge is an edge that if removed from the graph would increase the number of connected components in the graph. For example, in the following graph, the bridges are highlighted in red.



(a) Prove that an edge is a bridge if and only if it does not lie on any simple cycle in G.

(b) Suppose that we number the vertices of G in the order that they are visited by depth-first search. Denote this quantity by `dfs_ord`$[v]$ for each vertex v. Define the following quantity `low_link`$[v]$ for each vertex v such that

$$\texttt{low\_link}[v] = \min \begin{cases} \texttt{dfs\_ord}[v], \\ \texttt{dfs\_ord}[u] : \text{for any vertex } u \text{ reachable from } v \text{ via unused edges after visiting } v \end{cases}$$

Explain how the quantity `low_link` can be computed in $O(V+E)$ time.

(c) Explain how the quantities `dfs_num` and `low_link` can be used to determine which edges are bridges.

(d) Write pseudocode that uses the above facts to implement an algorithm based on depth-first search for determining the bridges of a given undirected graph in $O(V+E)$ time.

Solution

(a) Given a graph G, suppose that an edge $e \in E$ is a bridge such that its removal would disconnect two vertices u and v. We will show that e cannot lie on a cycle. Suppose for contradiction that e lies on a cycle. There was originally a path $u \rightsquigarrow v$ that went through the edge e, ie. $u \rightsquigarrow e_{\text{in}} \to e_{\text{out}} \rightsquigarrow v$. Since e is on a cycle, there is a path around the cycle $e_{\text{in}} \rightsquigarrow e_{\text{out}}$. Therefore there is a path $u \rightsquigarrow e_{\text{in}} \rightsquigarrow$

$e_{\text{out}} \rightsquigarrow v$, hence u and v are not disconnected which is a contradiction.

Now suppose that e is not a bridge, ie. there is no pair of connected vertices u, v who can only reach each other by crossing e. We will show that e lies on a cycle. Consider the pair of vertices $u = e_{\text{in}}$ and $v = e_{\text{out}}$. Since e is not a bridge and u and v are in the same connected component, there is a path from u to v that does not use the edge e. Since e connects u and v, we have that $u \underset{\text{Not via } e}{\rightsquigarrow} v \underset{e}{\rightarrow} u$ is therefore a cycle.

(b) We can compute `low_link` by noticing that the low link of a particular vertex v is the minimum low link of all vertices adjacent to v via unused edges. This is because if we can reach vertex u via some path of length longer than one, then one of the vertices that we are adjacent to is on that path and can therefore also reach it. In other words, we can write

$$\texttt{low\_link}[v] = \min\left(\texttt{dfs\_ord}[v], \min_{\substack{u \in V \\ u \text{ adjacent to } v \\ (u,v) \text{ unused}}} \texttt{low\_link[u]}\right).$$

This means that at each vertex v during depth-first search, we should traverse all of our children and then update our `low_link` to the minimum out of all of our adjacent vertices that are connected to us by an unused edge. This requires only a constant amount of extra work for each edge in the graph during depth-first search, hence it can be computed in $O(V + E)$ time.

(c) Observe that $\texttt{low\_link}[v]$ describes the earliest vertex that we can travel back to after reaching v for the first time without using any already-used edges. If it is possible to travel to v's parent u, then we could travel from v to u and then back down to v and hence there is a cycle containing the edge (u, v). If we can travel to any vertex that we visited before u, then we can travel from there to u by following the edges that the depth-first search took and hence there is a cycle containing (u, v). Otherwise if it is not possible to reach any vertex that we visited before v, then there is no way to cycle back to v and hence there is no cycle containing (u, v). Let (u, v) be an edge oriented in the direction that the depth-first search traverses it. Using the observation above, (u, v) is a bridge if and only if

$$\texttt{low\_link}[v] > \texttt{dfs\_num}[u].$$

(d) Since we can compute `dfs_ord` and `low_link` in $O(V + E)$ time and use them to determine the bridges, we can determine the bridges in $O(V + E)$ time as well.

```
function FIND_BRIDGES(G = (V, E))
    Set dfs_counter = 1
    Set dfs_ord[$u$] = low_link[$u$] = null for all $u \in V$
    for each vertex u = 1 to $n$ do
        if dfs_ord[$u$] = null then
            DFS($u$)
        end if
    end for
end function

function DFS(u)
    dfs_ord[$u$] = low_link[$u$] = dfs_counter
    dfs_counter = dfs_counter + 1
    for all edges $e = (u, v)$ adjacent to $u$ do
        if edge $e$ has not been traversed yet then
            if dfs_ord[$v$] = null then
                Mark $e$ as traversed
                DFS(v)
                if low_link[$v$] > dfs_ord[$u$] then
                The edge $e$ is a bridge
            **end if**
        **end if**
        low_link[$u$] = min(low_link[$u$], low_link[$v$])
    **end if**
**end for**
**end function**
```