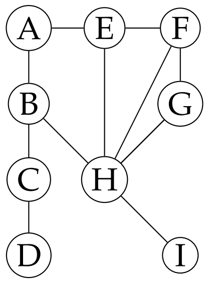
## Warm-up

**Problem 1.** Consider the following undirected graph.



a) Starting from $A$, give the layers the breadth-first search algorithm finds.

b) Starting from $A$, give the order in which the depth-first search algorithm visits the vertices.

**Solution 1.**

a) $L_0 = \{A\}$
$L_1 = \{B, E\}$
$L_2 = \{C, F, H\}$
$L_3 = \{D, G, I\}$

b) There are a number of different solution possible, when the algorithm can choose multiple nodes to recurse on. This example picks the lexicographically first node, but other choices are just as valid.
$A, B, C, D, H, E, F, G, I$

## Problem solving

**Problem 2.** An undirected graph $G = (V, E)$ is said to be bipartite if its vertex set $V$ can be partitioned into two sets $A$ and $B$ such that $E \subseteq A \times B$. Design an $O(n + m)$ algorithm to test if a given input graph is bipartite using the following guide:

a) Suppose we run BFS from some vertex $s \in V$ and obtain layers $L_1, \ldots, L_k$. Let $(u, v)$ be some edge in $E$. Show that if $u \in L_i$ and $v \in L_j$ then $|i - j| \leq 1$.

b) Suppose we run BFS on $G$. Show that if there is an edge $(u, v)$ such that $u$ and $v$ belong to the same layer then the graph is not bipartite.

c) Suppose $G$ is connected and we run BFS. Show that if there are no intra-layer edges then the graph is bipartite.

d) Put together all the above to design an $O(n + m)$ time algorithm for testing bipartiness.

**Solution 2.**

a) If $i = j$ then we are done. Otherwise, assume without loss of generality that $u$ is discovered by BFS before $v$; in other words, assume $i < j$. When processing $u$, BFS scans the neighborhood of $u$. Since $v$ ends up in layer $L_j$ and $j > i$, this means that $v$ had not been discovered yet at the time we started processing $u$. But then that means that $v$ will be placed in the next layer since $(u, v)$ is an edge; in other words, $j = i + 1$, and the property follows.

b) Suppose $u, v \in L_i$ and $(u, v) \in E$. Then we can form an odd cycle by taking the shortest path from $s$ to $u$, the edge $(u, v)$ and the shortest path from $v$ to $s$; this cycle has length $2i + 1$. Now if the graph was bipartite the vertices in the cycle should alternate between the $A$ set and the $B$ set, but that is not possible if the cycle has odd length.

c) Let $A$ be the even layers $L_0, L_2, \ldots$, and let $B$ be the odd layers $L_1, L_3, \ldots$. Because the graph is connected, every vertex belongs to some layer, so $(A, B)$ is a partition of $V$. Because there are no intra-layer edges, we have $E \subseteq A \times B$.

d) Given an input graph $G$, find its connected components. For each component, we run BFS and check if there are intra-layer edges; if not, partition the vertices into odd and even layer vertices.
The correctness of the algorithm follows readily from the previous tasks.
We use DFS to compute the connected components of $G$, which takes $O(n + m)$ time. Since these connected components together form the entire graph, running BFS on each of them has total running time $O(n + m)$. (Note that we can't say much about the running time per connected component, since they may vary from $O(1)$ to $\Omega(n)$ in size.) Checking for each edge if its endpoints are in different layers takes $O(1)$ time per edge, so $O(m)$ time in total. Hence, the running time is dominated by the DFS and BFS computations, which take $O(n + m)$ time.

**Problem 3.** Give an $O(n)$ time algorithm to detect whether a given undirected graph contains a cycle. If the answer is yes, the algorithm should produce a cycle. (Assume adjacency list representation.)

**Solution 3.** We run DFS with a minor modification. Every time we scan the neighborhood of a vertex $u$, we check if the neighbor $v$ has been discovered before and whether it is different than $u$'s parent. If we can find such a vertex then we have our cycle: $v, u, \text{parent}[u], \text{parent}[\text{parent}[u]], \ldots, v$.

We only need to argue that this algorithm runs in $O(n)$ time. Consider the execution of the algorithm up the point when we discovered the cycle. After the $O(n)$ time spent initializing the arrays needed to run DFS, each call to DFS-VISIT takes time that is proportional to the edges discovered. However, up until the time we find the cycle we have only discovered tree edges. So the total number of edges is upper bounded by $n - 1$. Thus, the overall running time in $O(n)$.

**Problem 4.** Let $G = (V, E)$ be an $n$ vertex graph. Let $s$ and $t$ be two vertices. Argue that if $\text{dist}(s, t) > n/2$ then there there exists a vertex $u \neq s, t$ such that every path from $s$ to $t$ goes through $u$.

**Solution 4.** Run BFS starting from $s$. Let $L_0, L_1, \ldots$ be the layers discovered by the algorithm. Recall that $L_0 = \{s\}$ and let $L_k$ be the layer $t$ belongs to. By our assumption that $\text{dist}(s,t) > n/2$, it follows that $k > n/2$. We claim that there is a layer $L_i$ for some $1 < i < k$, that has a single node; we call this a small layer. Otherwise, i.e., if all layers between $L_0$ and $L_k$ had two or more vertices, then

$$\sum_{i=1}^{k-1} |L_i| \geq 2(k-1) > 2\,(n/2 - 1) = n - 2.$$

But this cannot be, since there are at most $n-2$ vertices between $s$ and $t$.

Now that we know that there is a small layer, let's see how we can use this. Let $u$ be the vertex in small layer $L_i$. We claim that $u$ is in every $s$-$t$ path. Suppose, for the sake of contradiction, that there is a path from $s$ to $t$ that does not go through $u$. Such a path must have an edge $(x,y)$ where $x \in L_a$ and $y \in L_b$ and $a < i < b$. But this cannot be since when BFS visited $x$, it should have added $y$ to layer $L_{a+1}$, whereas $a < i < b$ implies that $b - a > 1$. A contradiction. Hence, $u$ must appear in every $s$-$t$ path.

**Problem 5.** In a directed graph, a *get-stuck* vertex has in-degree $n-1$ and out-degree $0$. Assume the adjacency matrix representation is used. Design an $O(n)$ time algorithm to test if a given graph has a get-stuck vertex. Yes, this problem can be solved without looking at the entire input matrix.

**Solution 5.** Let $A$ be the adjacency matrix of the graph; assume vertices are labeled $1, \ldots, n$. For ease of explanation, we say that an entry $(u,v)$ of the matrix is 1 if there is an edge from $u$ to $v$, and 0 otherwise. If $i$ is a get-stuck vertex then the $i$th row of $A$ is the all zeros vector, and the $i$th column of $A$ is the all ones vector with the exception of its $i$th entry, which is zero.

Now starting on the top left corner of the matrix consider the following walk:

1. If we are in some entry $(j,j)$, then we move right.
2. Otherwise, if we are at $(i,j)$ for $i \neq j$, then we move down if we see a 1 and we move right if we see a 0.

The walk ends when we would exit the matrix (i.e., try to access a non-existing cell), either through the right side of $(n,n)$ or the right side of another cell.

We claim that if there is a get-stuck vertex $i$ we will leave the matrix through the right side and on $i$'s row. To see this, we first observe that our walk always stays above the diagonal of the matrix. This implies that we will eventually reach column $i$. Since column $i$ consists of only ones above the diagonal, when our walk reaches it, we will move down until we reach $(i,i)$. Next, because the whole row $i$ is made up of zeros, we'd move right at all of these entries, thus exiting through the right side on the $i$th row.

If there is no get-stuck vertex we will still fall through the right side, so the row we exit at should still be checked.

An $O(n)$ time algorithm for finding a get-stuck vertex would then perform the walk. If the walk exists through the right side on row $i$, then we check if $i$ is indeed a get-stuck vertex (we only need to check $2(n-1)$ entries of $A$) and output accordingly. Hence, the algorithm inspects only $O(n)$ entries and spends $O(1)$ time per entry, thus the algorithm runs in $O(n)$ time in total.

**Problem 6.** Let $G$ be an undirected graph with vertices numbered $1 \dots n$. For a vertex $i$ define $\text{small}(i) = \min\{j : j \text{ is reachable from } i\}$, that is, the smallest vertex reachable from $i$. Design an $O(n+m)$ time algorithm that computes $\text{small}(i)$ for *every* vertex in the graph.

**Solution 6.** *(Sketch)* Run DFS. Each tree in the DFS forest is a connected component in the graph. For each component $C$ find the node $a \in C$ with the smallest label; for each vertex $b \in C$, set $\text{small}(b) = a$.

The algorithm is correct as every vertex in a connected component can reach the same set of vertices, namely, the whole connected component.

Running DFS takes $O(n+m)$ time. After that doing the scan of each component and setting the small values takes $O(n)$ time. Thus the total running time is $O(n+m)$.

**Problem 7.** In a connected undirected graph $G = (V, E)$, a vertex $u \in V$ is said to be a cut vertex if its removal disconnects $G$; namely, $G[V - u]$ is not connected.

The aim of this problem is to adapt the algorithm for cut edges from the lecture, to handle cut vertices.

a) Derive a criterion for identifying cut vertices that is based on the `down-and-up`$[\cdot]$ values defined in the lecture.

b) Use this criterion to develop an $O(n+m)$ time algorithm for identifying all cut vertices.

**Solution 7.** Recall that the main idea for the cut edges algorithm, was to run DFS on $G$ and for every $v \in V$ set $\text{level}(v)$ of the vertex $v$ in the DFS tree. Then we defined $\text{down-and-up}(u)$ be the highest level (closer to the root) we can reach by taking any number of DFS tree edges down and one single back edge up.

a) First note that the leaves of the DFS tree cannot be cut vertices, since the rest of the DFS tree keeps the graph together.
Secondly, note that if the root has two or more children, then it must be a cut vertex since there are no edges connecting its children subtrees (we only have DFS tree edges or back edges connecting a node to one of its ancestors).
Finally, consider an internal node $u$ other than the root. Note that after we remove $u$, it may be the case that the subtree rooted at one of its children gets disconnected from the rest of the graph. Let $v$ be one of $u$'s children. If $\text{DOWN-AND-UP}(v) < \text{LEVEL}(u)$ then after we remove $u$ the subtree rooted at $v$ remains connected by the edge that connects some descendant of $v$ to some ancestor of $u$. Otherwise, if $\text{DOWN-AND-UP}(v) \geq \text{LEVEL}(u)$ then $u$ is a cut vertex.

b) The algorithm first runs DFS and computes $\text{level}(u)$ and $\text{DOWN-AND-UP}(u)$ for all $u \in V$. If the root has two or more children, it is declared a cut vertex. For any other internal node $u$, we check if it has a child $v$ such that $\text{DOWN-AND-UP}(v) \geq \text{LEVEL}(u)$. If this is the case, we declare $u$ to be a cut vertex.
The correctness of the algorithm rests on the observations made in the previous point. The running time is dominated by running DFS and computing LEVEL and DOWN-AND-UP, which can be done in $O(n+m)$ time.

**Problem 8.** Let $T$ be a rooted tree. For each vertex $u \in T$ we use $T_u$ to denote the subtree of $T$ made up by $u$ and all its descendants. Assume each vertex $u \in T$ has a value $A[u]$ associated with it. Let $B[u] = \min\{A[v] : v \in T_u\}$. Design an $O(n)$ time algorithm that given $A$, computes $B$.

**Solution 8.** Perform a post-order traversal of the tree (you can do this in a graph just as well as in the tree data structure, as long as you keep track from which edge you used when you first arrived at a vertex). Recall that this means that for each vertex $u$, you first visit the subtree of each child before visiting $u$. When visiting $u$, compute $B[u]$ as the minimum of $A[u]$ and $B[v]$ for each child $v$ of $u$.

The correctness of the algorithm rests on the following observation

$$B[u] = \min_{x \in T_u} A[u] = \min(A[u], \min_{x \in T_{v_1}} A[x], \dots \min_{x \in T_{v_k}} A[x]) = \min(A[u], B[v_1], \dots, B[v_k]),$$

where $v_1, \dots, v_k$ are the children of $u$.

The time complexity is $O(n)$ because of the amount of work done at each vertex $u$ is $O(|N(u)|)$. Adding up over all vertices we get that the overall running time is $O(n)$ because $\sum_{u \in T} |N(u)| = 2(n-1)$.

**Problem 9.** Let $G$ be a connected undirected graph. Design a linear time algorithm for finding all cut edges by using the following guide:

a) Derive a criterion for identifying cut edges that is based on the `down-and-up`$[\cdot]$ values defined in the lecture.

b) Use this criterion to develop an $O(n+m)$ time algorithm for identifying all cut edges.

**Solution 9.**

a) First note that only DFS-tree edges can be cut edges because other edges can never disconnect the graph (the DFS tree keeps the graph in one piece).
Let $(u, v)$ be a DFS-tree edge. Assume that $u$ is $v$'s parent in the DFS tree. If $(u, v)$ is not a cut edge then some vertex in $T_v$ (the subtree rooted at $v$) must have a back edge to $u$ or a higher vertex; i.e., if $\textsc{down-and-up}(v) \leq \textsc{level}(u)$.

b) The algorithm first runs DFS and computes $\textsc{up}[u]$ at the highest level up the tree that we can reach from u by taking a back edge. Then we compute $\textsc{down-and-up}(u) = \min_{v \in T_u} \textsc{up}(v)$ for all $u \in V$ (see previous problem). Then for each DFS-tree edge $(u, v)$, where $v$ is a child of $u$, the algorithm declares $(u, v)$ to be a cut edge if $\textsc{down-and-up}(v) > \textsc{level}\ (u)$.
The correctness of the algorithm rests on observations made in the previous point.
The running time is dominated by running DFS and computing LEVEL, UP and DOWN-AND-UP, which can be done in $O(n+m)$ time.