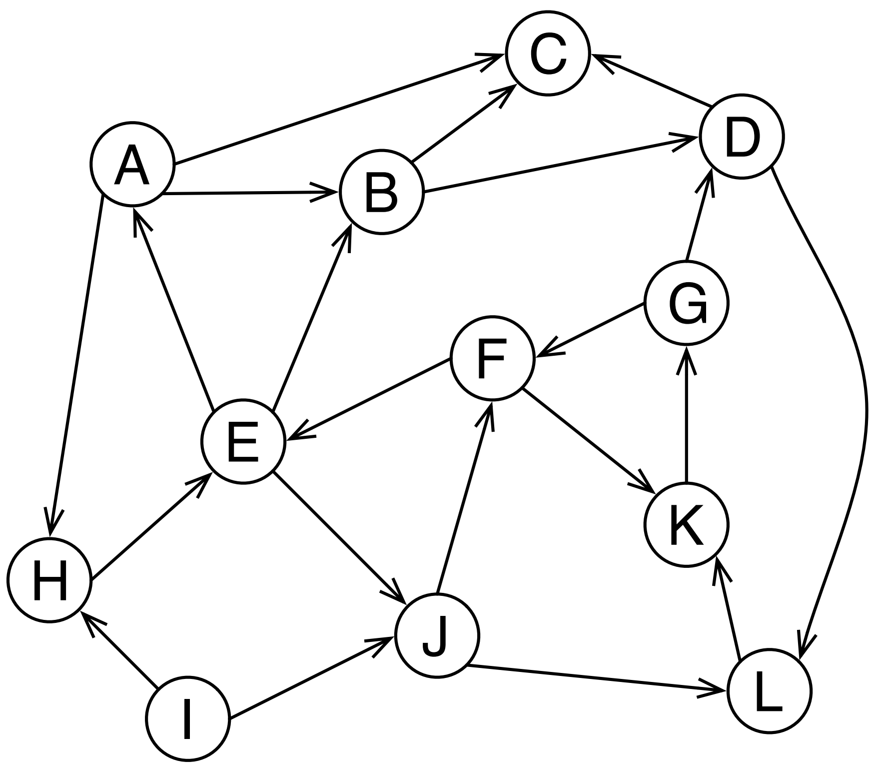
Executing Searches

- Trace an execution of `breadth_first_search()` on the following directed graph starting from vertex A.
- Highlight the search tree.
- Find a shortest path from vertex A to vertex G.



Concepts

Justify the true statements and give a counterexample for the false statements.

- If u can reach w and v can reach w in a directed graph, then either u can reach v or v can reach u.

  False: Consider the graph with only two edges (u, w) and (v, w).

- If there is a *walk* from a vertex u to a vertex v in a directed graph, then there is a *path* from u to v.

  True: Consider a $u - v$ walk. If it is not yet a path then it visits some vertex, say w, twice. Get a shorter walk by skipping all vertices between the two occurrences of w. Repeat until you are left with a path.

- A directed graph with n vertices and n edges must contain a cycle.

  False: Consider the graph with edges $(a, b), (b, c), (a, c)$.

- An undirected and connected graph with n vertices must contain at least $n - 1$ edges.

  True: Build a search tree (i.e. the `reached` dictionary from our Python code) starting from an arbitrary vertex v. All vertices except v (so $n - 1$ of them) were reached along some edge. All vertices are in the dictionary (as the graph is connected) so we see $n - 1$ different edges accounted for in the search tree alone.

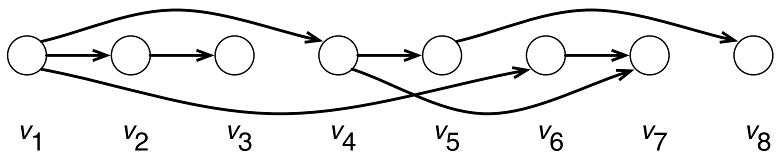
- An undirected graph with n vertices and n edges must contain a cycle.

  True: Build a search tree from an arbitrary vertex from each connected component. There are at most $n - 1$ edges in total appearing in these search trees. So some edge (u, w) is not in these search trees.

  We get a cycle by going from u to w using the path in the search tree, followed by the edge (w, u) to get back to u. Recall the edges are undirected, so we can travel along them in either direction.

Topological Ordering via Depth-First Search

Let G be a directed graph. A topological ordering of G is an ordering $v_1, \dots, v_n$ of its vertices such that for every edge (v_i, v_j) we must have $i < j$. Example:



1. If G contains a cycle, then it does not have a topological ordering. Why?

   Solution
   Let $v_1, \dots, v_k$ be a cycle. Then v_i would have to appear after v_{i-1} in any topological ordering for each $2 \leq i \leq k$. But then v_k would be ordered after v_1 and we would have the edge (v_k, v_1) pointing back to v_1. So, such an ordering is not possible if G contains a cycle.

2. For any two vertices u, v, if u can reach v by some path then every topological ordering of G must have u appearing before v. Why?

   Solution
   Say such a path is $u = v_1, \dots, v_k = v$. Then v_i must be ordered before v_{i+1} for every $1 \leq i \leq k-1$. Thus, $u = v_1$ must be ordered before $v = v_k$.

3. Create an ordered list of vertices using a depth-first search where a vertex v is added to the list the moment after all neighbours of v are recursively explored. In Python:

```
def do_dfs(curr, prev):
   if curr in reached:
      return
   reached[curr] = prev
   for succ in g.neighbours(curr):
      do_dfs(succ, curr)
   order.append(curr) # new part for this worksheet
```

Here, `order` is in the same scope as `reached` and is initially `[]`.

Consider the contents of `order` after one depth-first search. Show that if G does not have a cycle, then there is no directed edge (u, v) such that u appears before v in `order`. Thus, reversing `order` would produce a topological ordering of all vertices that were reached in the search.

Solution

Suppose that such a search produced an ordering that ordered u before w where (u, w) is an edge. We show G contains a cycle. We first observe that when `do_dfs(curr, prev)` was called with $curr = u$ that the call with $curr = w$ was already on the call stack.

To see this, if w was not even reached when $curr = u$, then the search would have recursively continued with $curr = w$ which would have placed w in $order$ before u. If w was reached when $curr = u$ and the recursive call for $curr = w$ had already completed, then w would already have been in $order$ when u was added to $order$.

So, the recursive call with $curr = w$ is still on the call stack when the call with $curr = u$ starts. Since u was recursively seen (possibly after a sequence of recursive calls) when $curr = w$ then there is a path from w to u. We now see a cycle, follow this $u - w$ path and follow up with the edge (u, w).

4. Use these ideas to design an algorithm that will find a topological ordering for any directed graph that does not contain a cycle, even if no single vertex can reach all others.

   $O(|V| + |E|)$ running time is possible. **Hint**: try the above depth-first search modification on an example where the start vertex cannot reach all other vertices. What can you do once this search is done to order the remaining vertices?

   Solution
   If such a search did not visit all vertices, then simply start a new search from an unreached vertex but start it with the old reached dictionary.

   Pseudocode:

```
reached = {}
order = []
for each vertex v
   if v no in reached
      do_dfs(v, v) (the modified search above, using this reached)
return order
```

   When done, there is no edge (u, w) where u appears before w in $order$ for essentially the same reason as above (you should double check this because the setting is slightly different, but the idea is the same).

   The running time is $O(|V| + |E|)$: each vertex will have its neighbours explored once throughout the entire search.