Breadth First Searches and Breadth-First Search Trees

Discussion

A breadth-first search of a directed graph visits all the immediate neighbors first then the those vertices that are a distance two away from the starting vertex next, then those three away and so on. The algorithm to perform a breadth-first search is generally implemented using a FIFO queue. Depending upon the starting vertex, some of the vertices may not be reached. In that case if we wish to ensure that all vertices are a part of one of the breadth-first search trees, it is necessary to choose the next unvisited vertex as the starting vertex and repeat the search. We must then continue that process until all the vertices have been reached.

The pseudo-code for the breadth-first search algorithm is shown below, where `G` is the graph and `s` represents the starting vertex:

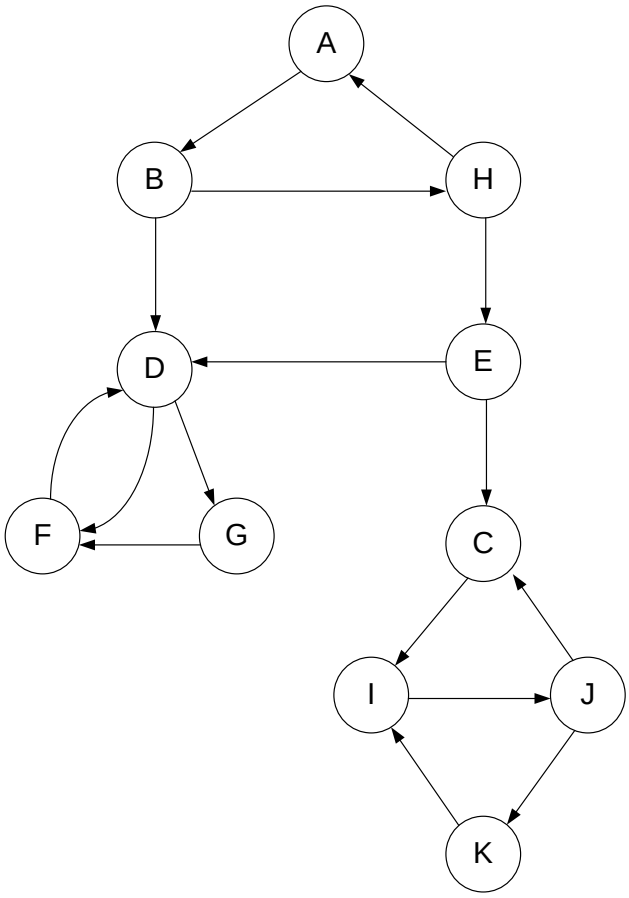
```
bfs (G, s)
  mark s as discovered
  enqueue s
  while the queue is not empty
    dequeue v
    for each vertex w adjacent to v
        if w is undiscovered
          mark w as discovered
        dequeue w
    mark w as finished
```

If we wish to produce the breadth-first search tree with the above algorithm, we need to output the edges as each new vertex is discovered. In the case that multiple starting vertices are needed to discover all vertices, the result will be a forest of breadth-first search trees.

For a given graph and a specified starting vertex, the breath-first search tree is unique.

Sample Problem

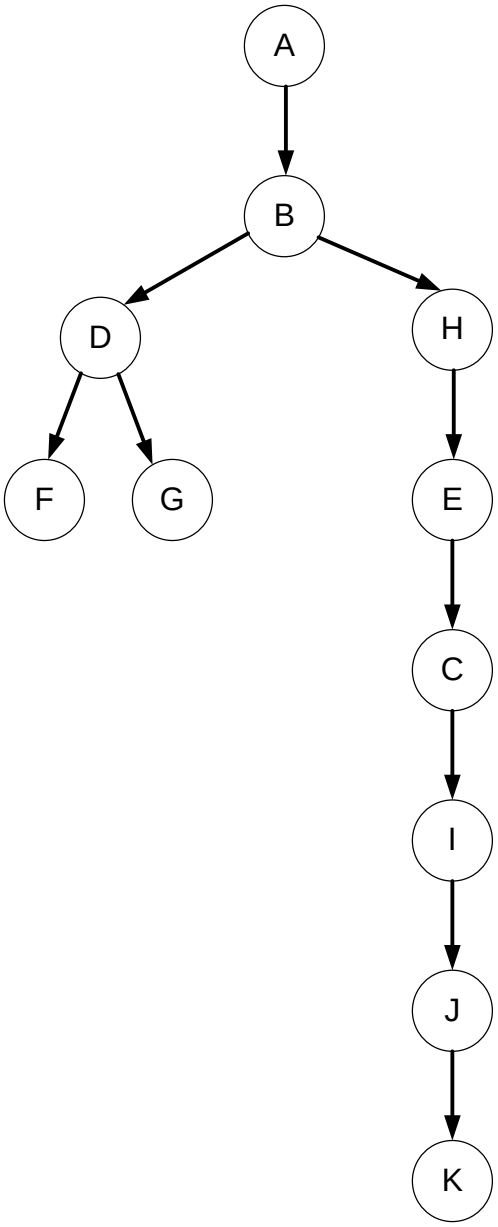
Given the following directed graph:



Perform a breadth-first search on the graph assuming that the vertices and adjacency lists are listed in alphabetical order. Show the breadth-first search tree that is generated.

Solution

The resulting breadth-first search tree is shown below:



For the graph supplied, using A as the starting vertex, all vertices belong to a single breadth-first search tree.

Depth-First Search Trees and Edge Classification

Discussion

A depth-first search of a directed graph visits all the vertices along a particular path before it moves to the next path. The algorithm to perform a depth-first search could be implemented using a stack, but it is more often done with recursion. Just as was the case with the breadth-first search, some of the vertices may not be reached. In that case if we wish to ensure that all vertices are a part of one of the depth-first search trees, it is necessary to choose the next unvisited vertex as the starting vertex and repeat the search. We must then continue that process until all the vertices have been reached.

The pseudo-code for the depth-first search algorithm is shown below, where `G` is the graph and `v` represents the starting vertex:

```
dfs (G, v)
  mark v as discovered
  for each vertex w adjacent to v
    if w is undiscovered
      dfs(G, w)
    else
      check vw without visiting w
    mark v as finished
```

If we wish to produce the depth-first search tree with the above algorithm, we need to output the edges as each new vertex is discovered. In the case that multiple starting vertices are needed to discover all vertices, the result will be a forest of depth-first search trees.

By maintaining a counter, we can label each vertex that is reached during a search with start and finish times. When a vertex is discovered, we label its start time as the current value of the counter, then increment the counter. When a vertex is marked as finished, we label its finish time with the current counter and then increment it.

By first defining *ancestor* and *descendant*, we can classify the edges of a depth-first search tree search tree into four categories. A vertex *v* is an ancestor of a vertex *w* if *v* is on the path from the root to *w*. In that case, we also say that *w* is a descendant of *v*.

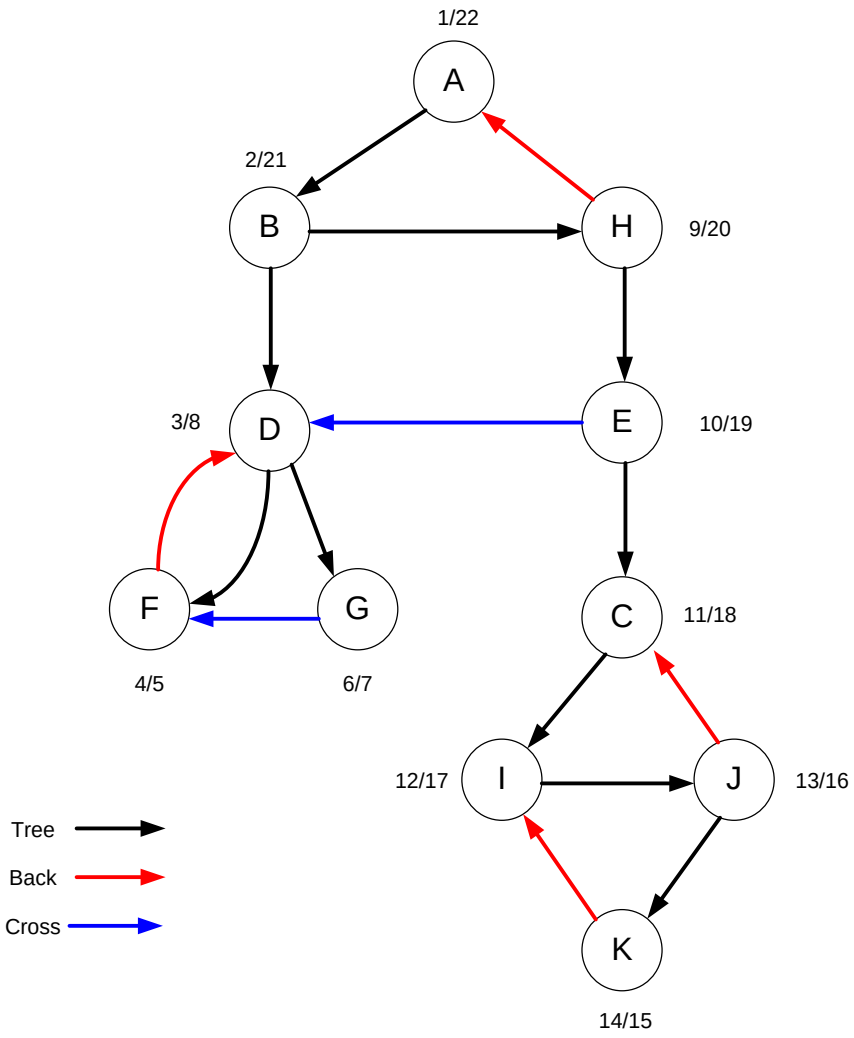
An edge *vw* is a *tree edge* if *w* is undiscovered when *w* is explored. It is a *back edge* if *w* is an ancestor of *v*. It is a *forward edge* if *w* is a descendant of *v*, and *w* is already discovered. It is *cross edge* if *w* is neither an ancestor nor descendant of *v*.

In the above algorithm, tree edges result when *w* is undiscovered. When *w* is already discovered, the edge is one of the three remaining types. If the *w* is not finished, it is a back edge. Otherwise it is either a forward or cross edge. To make that determination we must check whether there is another path from *v* to *w*. If there is, it is a forward edge, otherwise it is a cross edge.

Sample Problem

Perform a depth-first search on the graph shown in the first sample problem assuming that the vertices and adjacency lists are listed in alphabetical order. Classify each edge as tree, forward, back, or cross edge. Label each vertex with its start and finish time

Solution



For this particular graph, there don't happen to be any forward edges.

Topological Orders

Discussion

Every directed acyclic graph (DAG) defines a partial order on the set of vertices of the graph. We define that order by saying that $v \leq w$ if there is a path from v to w and adding the fact that $v \leq v$ for all vertices v. A topological order of a DAG is any total or linear order that preserves the partial order, which means that if we designate $\leq_1$ as the partial order and $\leq_2$ as the topological order, for all vertices v and w, if $v \leq_1 w$ then $v \leq_2 w$ must also be true

Topological orders are not unique. A DAG may have many topological orders. If we enumerate the vertices at their finish times we obtain a reverse depth-first topological order. We can obtain a breadth-first topological order with the following algorithm written in pseudo-code:

```
while there are vertices remaining
  enumerate all vertices with no predecessors
  remove from the graph those vertices and all edges that emanate from them
```

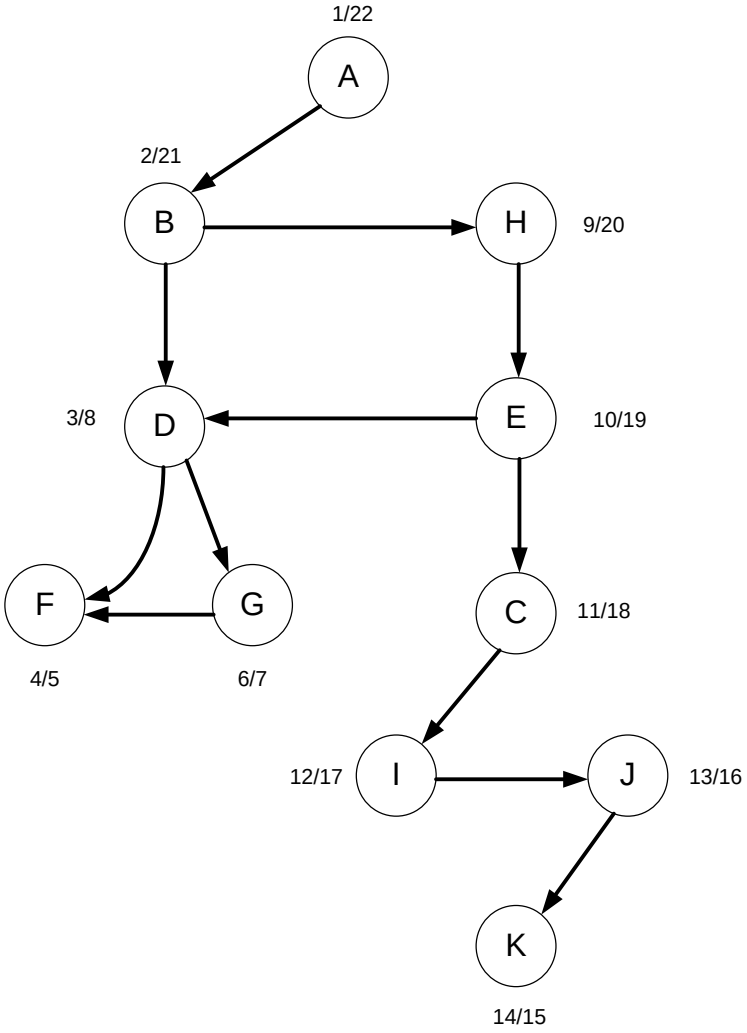
Because we can enumerate the vertices with no predecessors at a given point, in any order, breadth-first topological orders are not unique.

Sample Problem

Remove all the back edges from the graph provided in the first sample problem so it becomes a DAG. Perform a depth-first search recording the start and finish times. Using those finish times, provide the topological order that is produced. Provide one breadth-first topological order for that graph.

Solution

Shown below is a depth first search of the graph with the back edges removed. It is labeled with the start and finish times resulting from that search.



Enumerating the vertices as they finish we get the following enumeration:

F G D K J I C E H B A

If we reverse the above enumeration we get the following depth-first topological order:

A B H E C I J K D G F

To obtain the breadth-first topological order we begin with vertices with no predecessors. A is the only such vertex, so it must be first. We remove A and the edge AB, then B is the only vertex with no predecessors, so it is enumerated next. We then remove B and the edges BD and BH from the graph. H is the only vertex with no predecessors, so it is enumerated next. We remove H and the edge HE. E is the only vertex with no predecessors, so it is enumerated next. We remove E and the edges EC and ED. At this point both C and D have no predecessors so we can

enumerate them in either order. We remove them both along with the edges DF, DG and CI. At this point both G and I have no predecessors so we can enumerate them in either order. We continue this process until no vertices remain. One possible resulting breadth-first topological order is the following:

A B H E C D G I F J K

Strongly Connected Components

Discussion

A directed graph is *strongly connected* if there is a directed path between every pair of vertices. A *strongly connected component (SCC)* of a directed graph is a maximal strongly connected subgraph.

An important observation is that the vertices produced by any depth-first search of a directed graph will contain all the vertices of one or more strongly connected components. If we choose the starting vertices is the correct order, we will be guaranteed to enumerate the vertices of exactly one strongly connected component on each search. The way we obtain that correct order is first producing a depth-first topological order of the vertices by pushing them on a stack as the vertices finish. Next we observe that the strongly connected components of every graph are identical to the strongly connected components of its transpose graph. Those observation lead to the following algorithm for determining strongly connected components:

```
push the vertices onto a stack at their finish times during a depth-first
  search of the original graph to obtain a topological order
transpose the graph
perform a depth-first search on the transpose graph choosing the starting
  vertices in the topological order, skipping all discovered vertices
```

Sample Problem

Determine the strongly connected components of the graph provided in the first sample problem using the above algorithm. Show the final depth-first search of the transpose graph labeled with its start and finish times. Identify the strongly connected components based on that search.

Solution

We choose the starting vertices from the following order obtained in the previous problem:

A B H E C I J K D G F

The depth-first search starting a A produces the SCC consisting of A H B. The next undiscovered vertex in the topological order is E. A depth-first search starting at E gives the SCC consisting only of E. The next undiscovered vertex in the topological order is C. The depth-first search starting a C gives the SCC consisting of C J I K. The next undiscovered vertex in the topological order is D. The depth-first search starting at D gives the SCC consisting of D F G.

The transpose graph labeled with the start and finish times is shown below: