Depth-First Search

Chapter 22

Depth-First Search

Depth-first search (DFS) is, like breadth-first search (BFS), a way of examining every node in a connected graph. Both BFS and DFS have variants that will examine every node in an arbitrary (not necessarily connected) graph.

The textbook presents the connected variant of BFS and the disconnected version of DFS. A footnote is provided at the start of section 22.3 to try to justify this. I soundly disagree with that footnote and will present the connected version of DFS before presenting the disconnected DFS.

Like BFS, DFS starts at a single node and looks for neighbors to explore, recursively exploring their neighbors, and those neighbors' neighbors, etc.

Depth-First Search

BFS can optionally label each vertex with its distance from the start vertex; DFS cannot. Both algorithms can also optionally construct a tree.

In a (connected-variant) DFS tree, each node v except the start node has a predecessor or parent node, which is the node whose exploration first discovered v. DFS trees have different properties than BFS trees.

DFS and BFS are classic worklist algorithms, where the work to be done is kept in a list with items added as they are discovered and removed as they are processed. The worklist for DFS is implicitly maintained on the recursion stack.

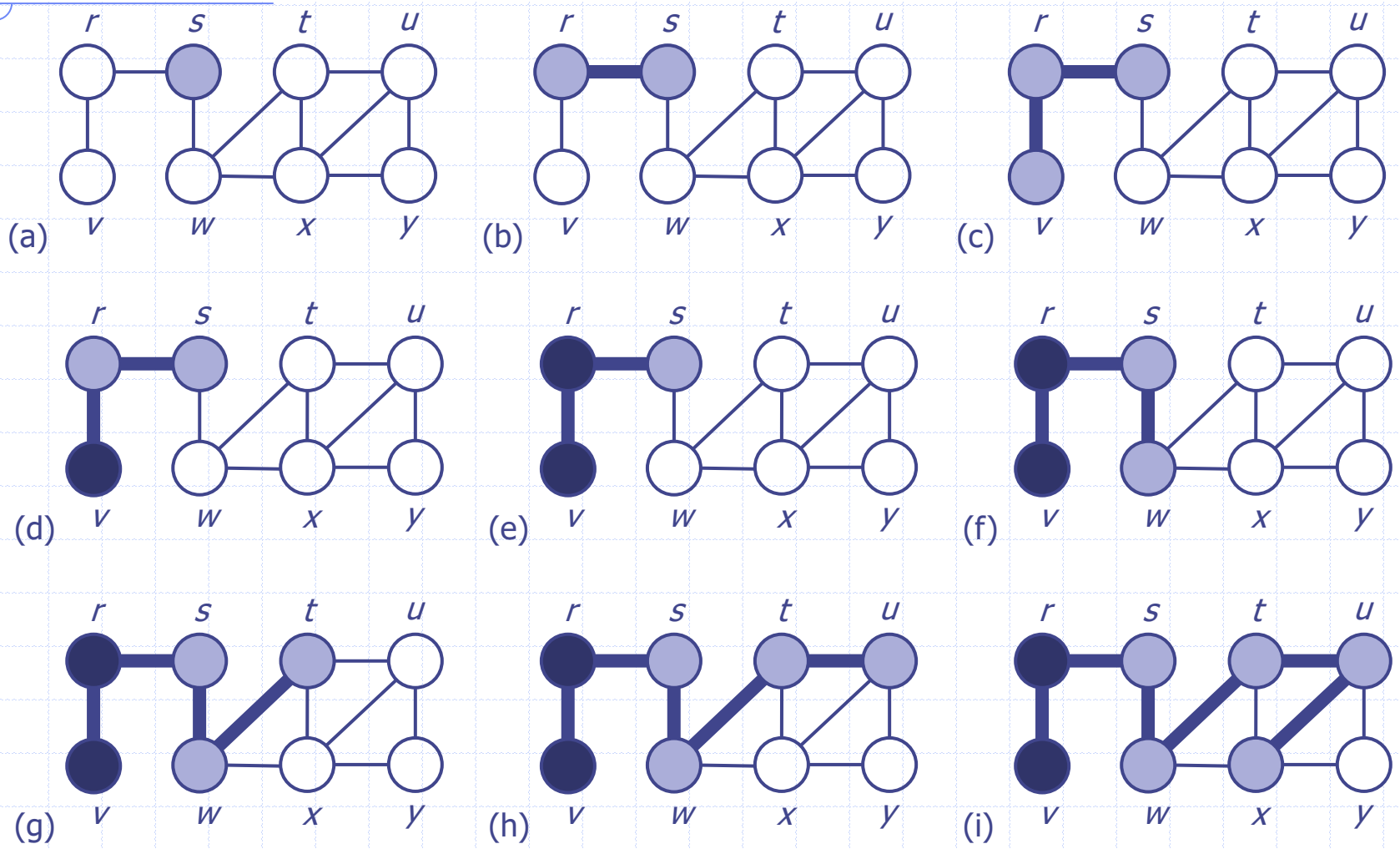
Depth-First Search

```
Graph.DFS1(s)                          // s is start vertex

    for each vertex u in G:            // G is "this"
        u.status = UNDISCOVERED
        u.predecessor = nil

    DFS-VISIT(s)
```

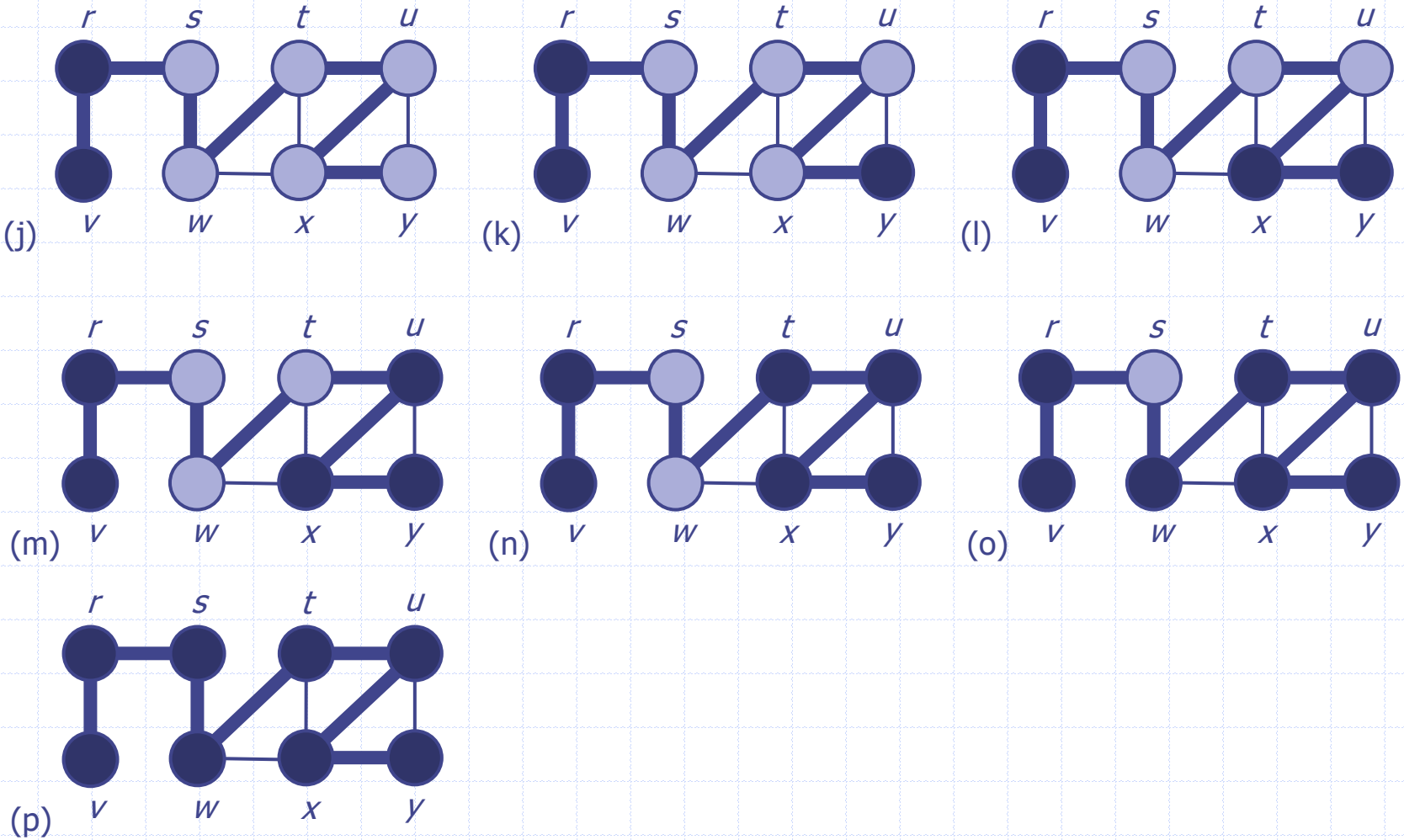
Depth-First Search

```
DFS-VISIT(u)
    u.status = DISCOVERED

    for each v adjacent to u                    // process u
        if v.status = UNDISCOVERED
            v.predecessor = u
            DFS-VISIT(v)

    u.status = PROCESSED
```

Depth-First Search Example



Depth-First Search Example



Depth-First Search

The depth-first search from the text will handle a graph that is not connected, but it loses the ability to have a specific start vertex.

DFS (connected or arbitrary) can optionally compute the discovery and finish times of each vertex. A global counter time is maintained, and it is incremented and recorded whenever a vertex is discovered or has finished processing.

Depth-First Search

```
Graph.DFS2()                                    // no start vertex
    for each vertex u in G:                     // G is "this"
        u.status = UNDISCOVERED
        u.predecessor = nil                     } O(V)
    time = 0                                    O(1)
    for each vertex u in G:
        if u.status = UNDISCOVERED              } O(V) + total time in
            DFS-VISIT(u)                                 DFS-VISIT
```

Depth-First Search

```
DFS-VISIT(u)
    u.status = DISCOVERED
    time = time + 1
    u.discovered = time

    for each v adjacent to u
        if v.status = UNDISCOVERED
            v.predecessor = u
            DFS-VISIT(v)

    u.status = PROCESSED
    time = time + 1
    u.finished = time
```

called at most once per vertex

O(1)

O(V)

O(degree(u))

O(E)

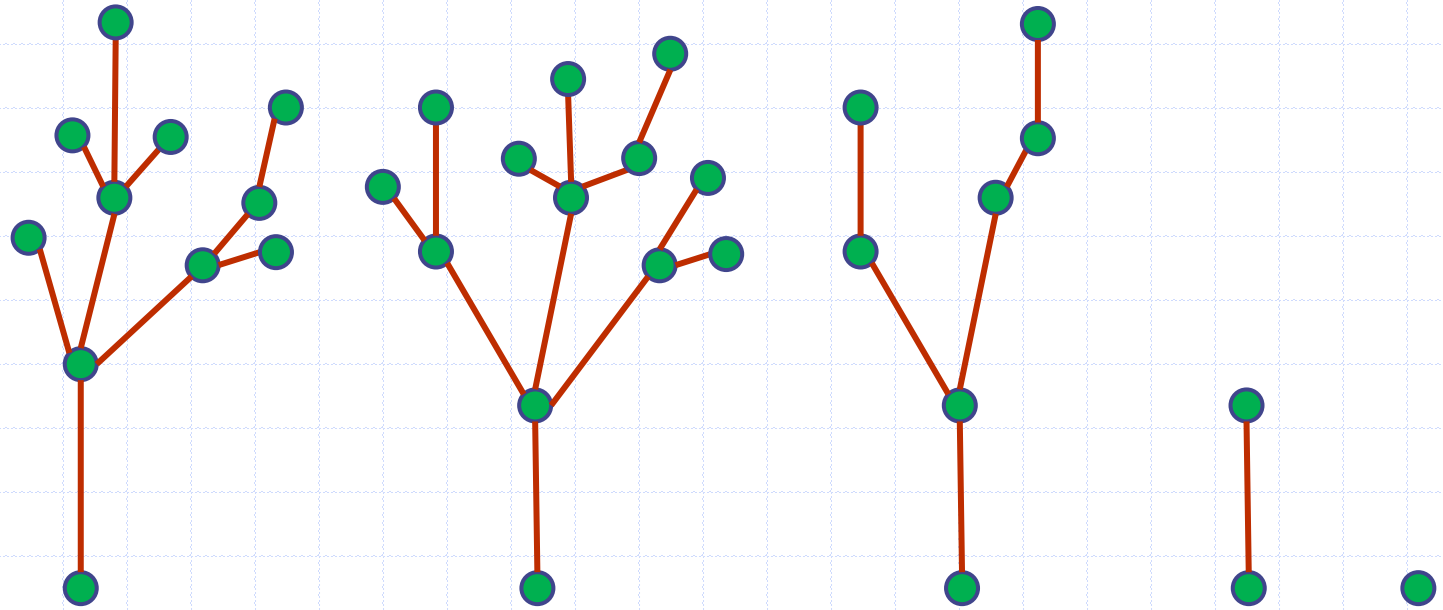
O(1)

O(V)

O(V + E)

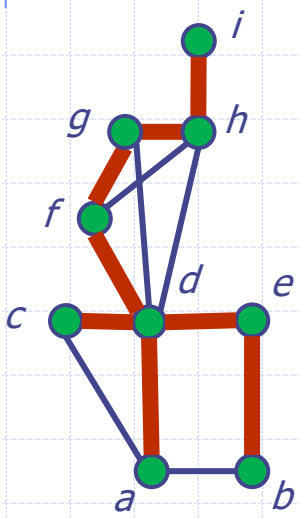
The Predecessor Graph

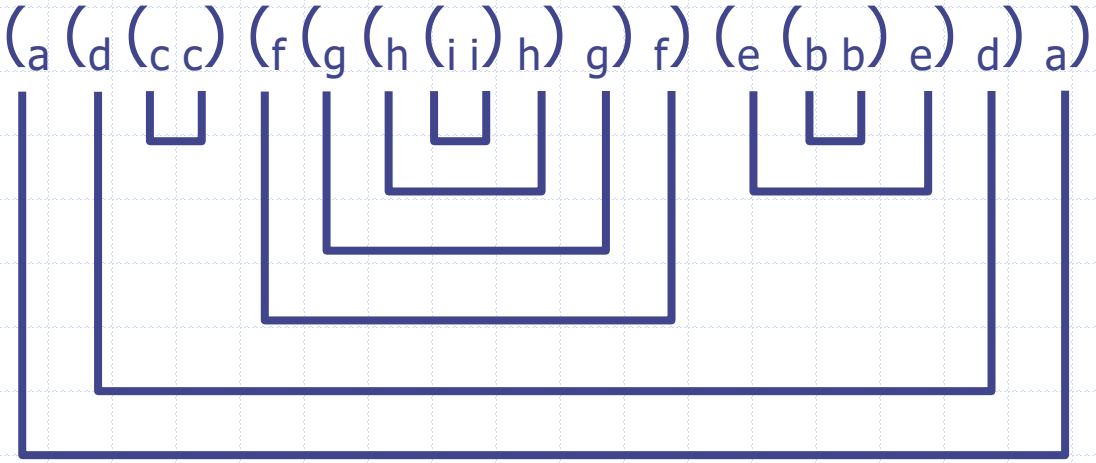
The predecessor graph computed by DFS on a graph G is the subgraph with E = {uv | u = v.predecessor}. This is a forest (collection of trees). If G is connected, the forest consists of a single tree.



Parenthesis Structure

Let us represent the discovery of vertex u with a special open-parenthesis $($u and the finishing of vertex u with a special matching close-parenthesis u$)$. Then the history of all discoveries and finishings in a DFS of a graph makes a well-formed nested-parenthesis expression.





Parenthesis Theorem

Theorem. In any DFS of a (directed or undirected) graph G = (V, E), for any two vertices u and v, exactly one of the following holds:

- the intervals [u.discovered, u.finished] and [v.discovered, v.finished] are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval [u.discovered, u.finished] is contained entirely within the interval [v.discovered, v.finished] and u is a descendant of v in the depth-first forest, or
- the interval [v.discovered, v.finished] is contained entirely within the interval [u.discovered, u.finished] and v is a descendant of u in the depth-first forest.

Parenthesis Theorem Proof

Proof. Consider the case in which u.discovered < v.discovered. We examine subcases based on whether v.discovered < u.finished.

If this condition is true, then v was discovered while u was still in the DISCOVERED state. This implies that v is a descendant of u. Since v was discovered later than u, all of v's outgoing edges are explored, and v is finished, before the search returns to and finishes u. Thus the interval [v.discovered, v.finished] is contained within the interval [u.discovered, u.finished].

If the condition is false, u.finished < v.discovered, and since v.discovered < v.finished, the intervals are disjoint. Because they are disjoint, neither u nor v was discovered when the other was still in the DISCOVERED state, so neither is a descendant of the other.

Parenthesis Theorem Proof

The case in which v.discovered < u.discovered is symmetric. ■

Corollary. Vertex v is a proper descendent of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if

u.discovered < v.discovered < v.finished < u.finished.

Undiscovered Path Theorem

Theorem. In a depth-first forest of a (directed or undirected) graph G = (V, E), vertex v is a descendant of vertex u if and only if at the time the search discovers u, vertex v can be reached from u along a path consisting entirely of vertices with UNDISCOVERED status.

Proof. If v is a descendant of u, let w be any vertex on the path between u and v in the DFS tree; w is thus a descendant of u. By the previous corollary, u.discovered < w.discovered, so w is UNDISCOVERED at the time u.discovered.

Suppose v can be reached from u along a path of UNDISCOVERED vertices at time u.discovered, but v does

Undiscovered Path Theorem

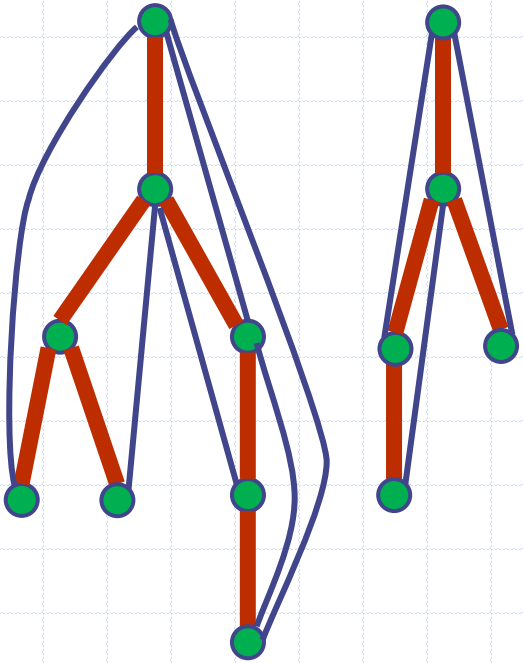
not become a descendant of u in the DFS tree. (Let v be the closest such vertex to u.) All other vertices from u to v along the UNDISCOVERED path become descendants of u. Let w be the predecessor of v in this path. By the parenthesis corollary, w.finished ≤ u.finished. Now, u.discovered < v.discovered < w.finished ≤ u.finished. So by the parenthesis theorem, [v.discovered, v.finished] is contained entirely within [u.discovered, u.finished] and v must be a descendant of u. ■

Classification of edges

DFS can be used to classify the edges of a graph or directed graph. We divide the edges into four classes:

1. Tree Edges are edges in the DFS forest (predecessor graph).
2. Back Edges are those edges (u, v) connecting a vertex u to an ancestor v in a DFS tree. Self-loops (edges (u, u)) are considered to be back edges.
3. Forward Edges are those nontree edges connecting a vertex u to a descendent v in a DFS tree.
4. Cross edges are all other edges.

Classification of edges

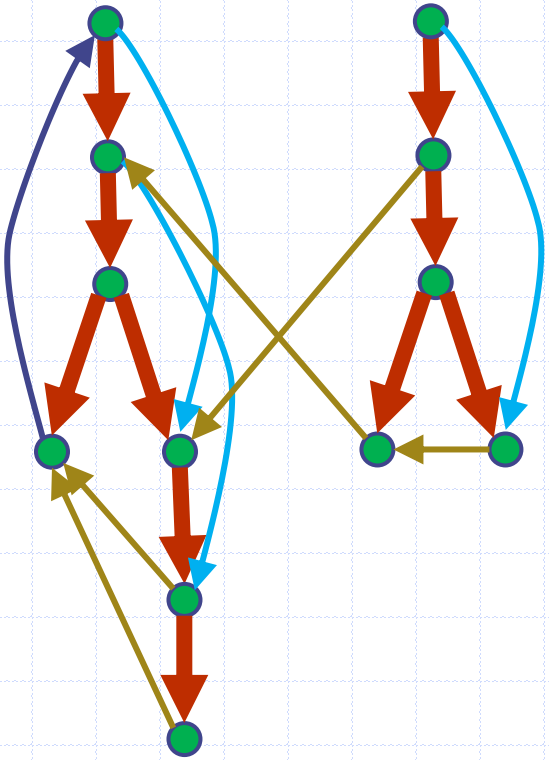


tree

back

forward

cross



Classification of edges

The DFS algorithm itself can classify the edges as it operates. The key is to classify an edge (u, v) when it is first explored.

1. If v is UNDISCOVERED, (u, v) is a tree edge.
2. If v is DISCOVERED, (u, v) is a back edge.
3. If v is PROCESSED, (u, v) is a forward or cross edge.
    - 3a. If u.discovered < v.discovered, (u, v) is a forward edge.
    - 3b. If u.discovered > v.discovered, (u, v) is a cross edge.

Classification of edges

Theorem. In a DFS of a undirected graph G, every edge of G is either a tree edge or a back edge.

Proof. Let (u, v) be any edge of G with u discovered before v. Then, v must be discovered and finished before u finishes. If (u, v) is explored first from u, then (u, v) becomes a tree edge. If (u, v) is explored first from v, then (since u is already DISCOVERED), (u, v) becomes a back edge. ■

Breadth-first Search for Arbitrary Graphs

```
Graph.BFS2()                                  // no start vertex

    for each vertex u in G:                   // G is "this"
        u.status = UNDISCOVERED
        u.distance = ∞
        u.predecessor = nil

    for each vertex s in G:
        if s.status = UNDISCOVERED
            s.status = DISCOVERED
            s.distance = 0
            s.predecessor = nil
            BFS-VISIT(s)
```

Breadth-first Search for Arbitrary Graphs

```
BFS-VISIT(s)
    Q = new Queue
    Q.enqueue(s)
    while !Q.isEmpty()
        u = Q.dequeue()
        for each v adjacent to u
            if v.status = UNDISCOVERED
                v.status = DISCOVERED
                v.distance = u.distance + 1
                v.predecessor = u
                Q.enqueue(v)

        u.status = PROCESSED
```