CS 228: Introduction to Data Structures
Lecture 36

Graph Terminology

We say that node u is a ***neighbor*** of node v if (v,u) is an edge.

- In an ***undirected*** graph, the neighbor relation is *symmetric* — that is, if u is a neighbor of v, then v is a neighbor u —, and we say that two nodes that are neighbors of each other are ***adjacent***. The number of neighbors of a node v is called the ***degree*** of v.

- In a ***directed*** graph, the number of neighbors of a node v is called the ***out-degree*** of v, and the number of nodes that have v as a neighbor is called the ***in-degree*** of v. Equivalently, the out-degree of v is the number of edges directed out of v and the in-degree is the number of edges directed into v.

A ***path*** is a sequence of vertices such that each successive pair of vertices is connected by an edge. For example, one path in the graph of Figure 1 is

France-Luxembourg-Germany-Denmark

If the graph is directed, the edges that form the path must all be aligned with the direction of the path. A ***simple path*** is one with no repeated nodes. A ***cycle*** is a path with at least one edge whose first and last nodes are the same. A ***simple cycle*** is a cycle with no repeated edges or nodes (except the requisite repetition of the first and last nodes).

The ***length*** of a path or a cycle is its number of edges it traverses. E.g., in the street map graph above, `3−0−1−2−5` is a path of length `4`. It is perfectly OK to talk about a path of length zero, such as (the single node) `2`. The ***distance*** from one node to another is the length of the shortest path from one to the other.

An undirected graph is ***connected*** if there is a path from every node to every other node in the graph. A graph that is not connected consists of a set of ***connected components***, which are maximal connected subgraphs.

Intuitively, if the nodes were physical objects, such as knots or beads, and the edges were physical connections, such as strings or wires, a connected graph would stay in one piece if picked up by any node, and a graph that is not connected comprises two or more such pieces.

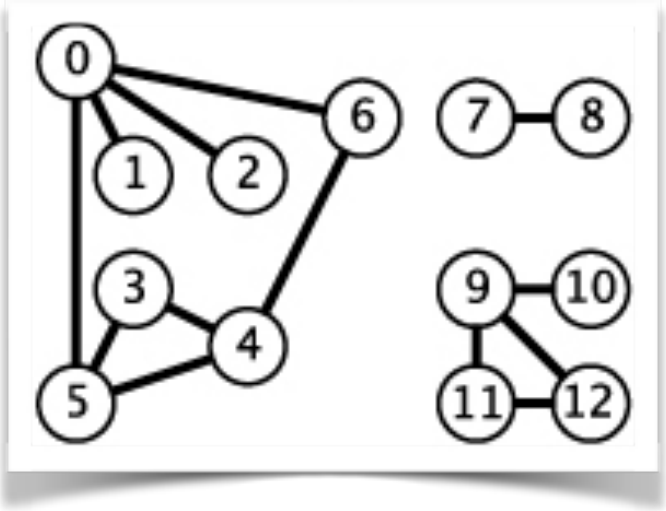


Figure 3. A disconnected graph. Its components are {0, 1, 2, 3, 4, 5, 6}, {7, 8}, and {9, 10, 11, 12}.

Generally, processing a graph necessitates processing the connected components one at a time.

In a directed graph, the existence of a path from u to v does not imply that there is a path from v to u. We say that a directed graph is ***strongly connected*** if for every pair of nodes u and v there is a path from u to v and a path from v to u. The graph in Figure 1 is strongly connected, but the one in the next figure is not. Notice, though that any directed graph can be decomposed into ***strongly connected components*** —subsets of nodes that are strongly connected, such that if you add any other node to the subset, it is no longer strongly connected.

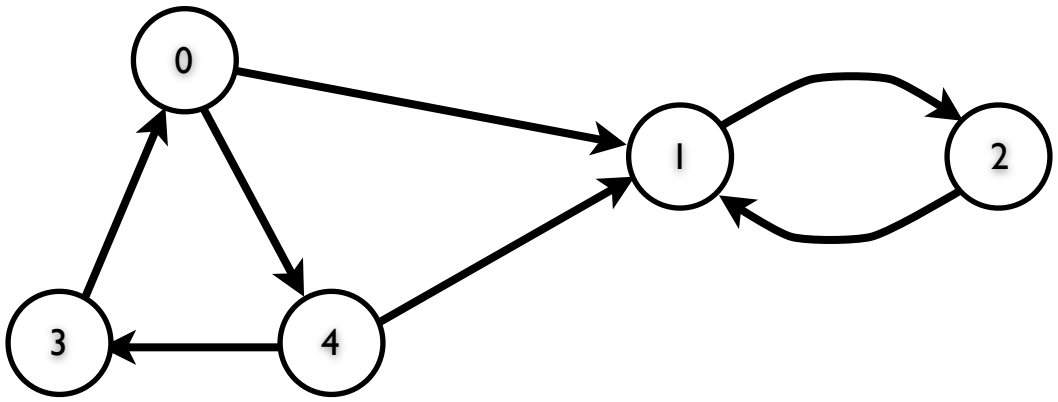


Figure 4. A graph that is not strongly connected. Its strongly connected components are {0, 3, 4} and {1, 2}.

We are now ready to define some common graph problems (there are many others):

- Given a starting node S, what other nodes are reachable by a path from S?

  Examples: web crawler, garbage collector

- Given nodes S and T, is there a path from S to T?

  Examples: path planning (robotics), model checking

- Given two nodes S and T, what is the shortest path from S to T? "Shortest" may mean fewest number of edges, or it may be based on an "edge cost" that measures something else.

Examples: network router (fewest number of hops), MapQuest (minimal driving time).

Representing Graphs

To manipulate graphs in a computer, we need data structures to represent them. We will study three approaches: adjacency matrices, adjacency lists, and a `HashMap`/`HashSet` structure.

In what follows, V and E denote the number of nodes and edges in a graph, respectively. For simplicity, we begin by assuming that the nodes are numbered 0 through V–1. We will drop this assumption later.

Adjacency Matrices

An ***adjacency matrix*** is a V-by-V array of boolean values. Each row and column represents a vertex of the graph. Set the value at row i, column j to `true` if (i, j) is an edge of the graph.

If the graph is undirected, the adjacency matrix is *symmetric*: row i, column j has the same value as row j, column i (see Figure 5).

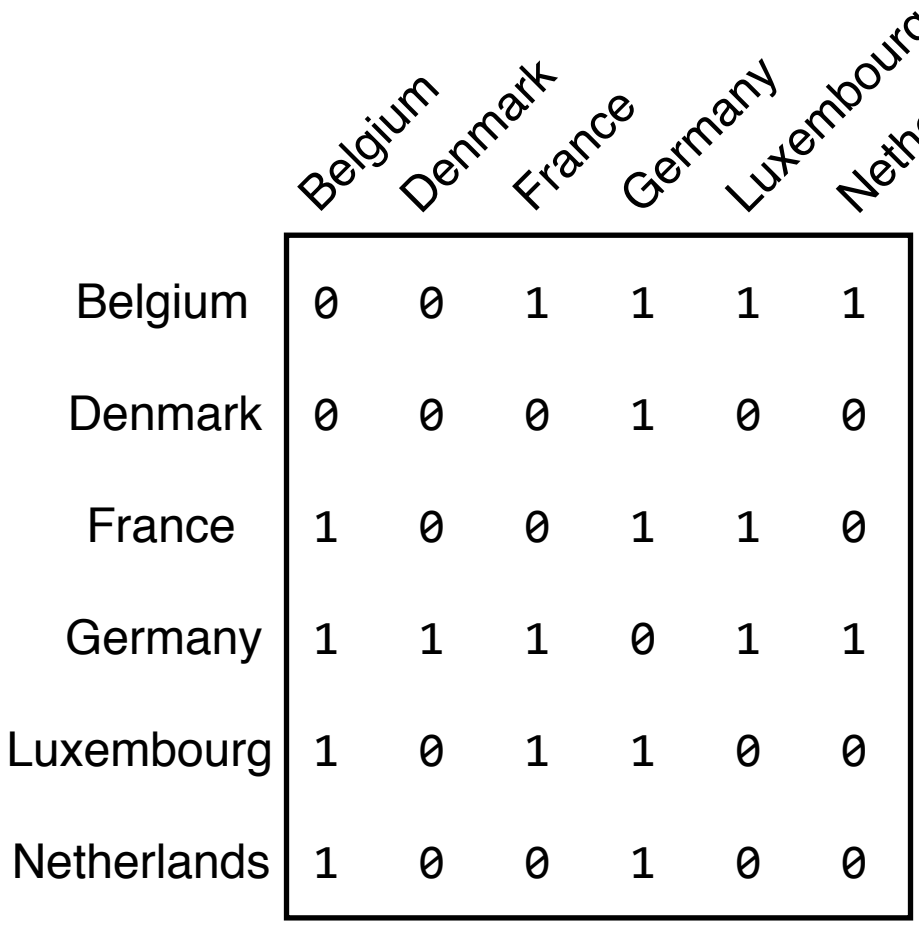


Figure 5. Adjacency matrix for the graph of Figure 2.

As the next figure shows, the adjacency matrix of a directed graph is not, in general, symmetric.

	0	1	2	3	4	5
0	0	1	0	1	0	0
1	0	0	1	0	1	0
2	0	0	0	0	0	1
3	1	0	0	0	0	0
4	0	0	0	1	0	0
5	0	0	1	0	1	0

Figure 5. Adjacency matrix for the graph of Figure 1.

Adjacency Lists

An ***adjacency list*** representation consists of an array with one entry per node. The entry for node v references the set of all neighbors of v (recall that u is a neighbor of v if (v,u) is an edge in the graph).

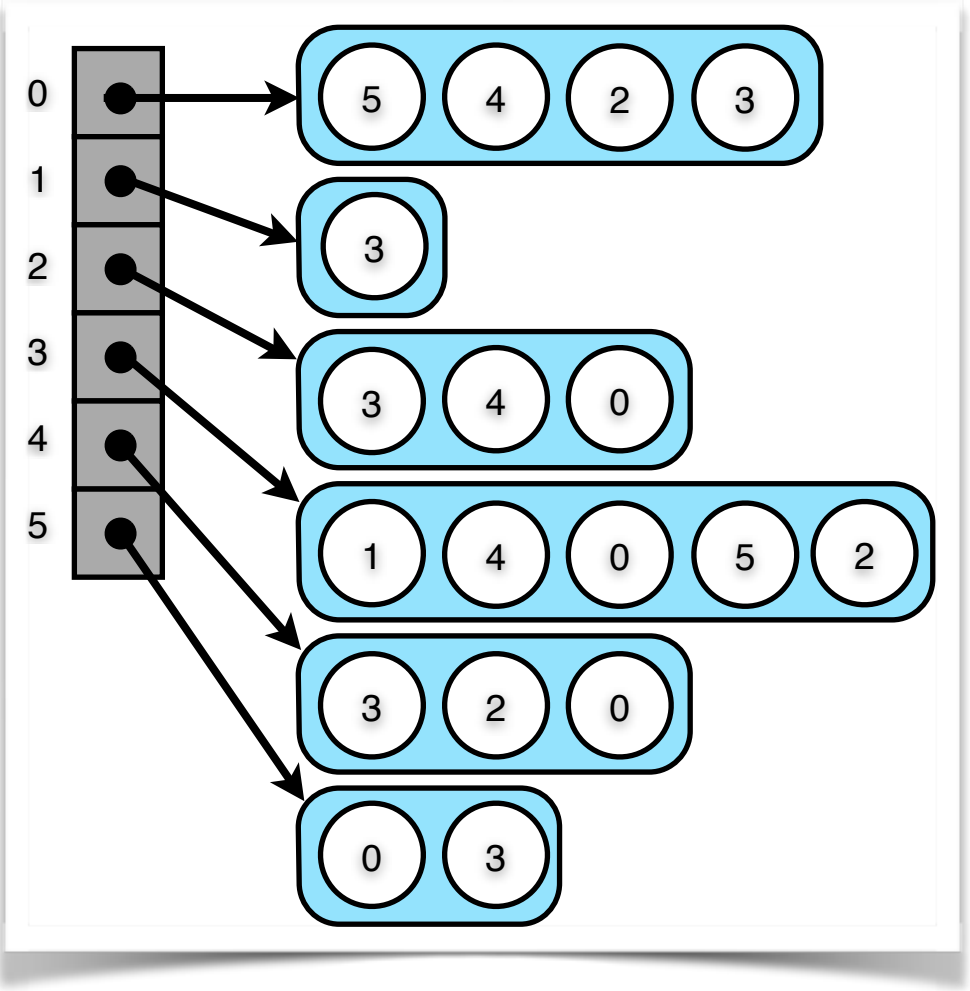


Figure 6. Adjacency list representation for the graph of Figure 2. Belgium = 0, Denmark = 1, France = 2, Germany = 3, Luxembourg = 4, Netherlands = 5.

For the street map example of Figure 1, the adjacency list representation would look like this.

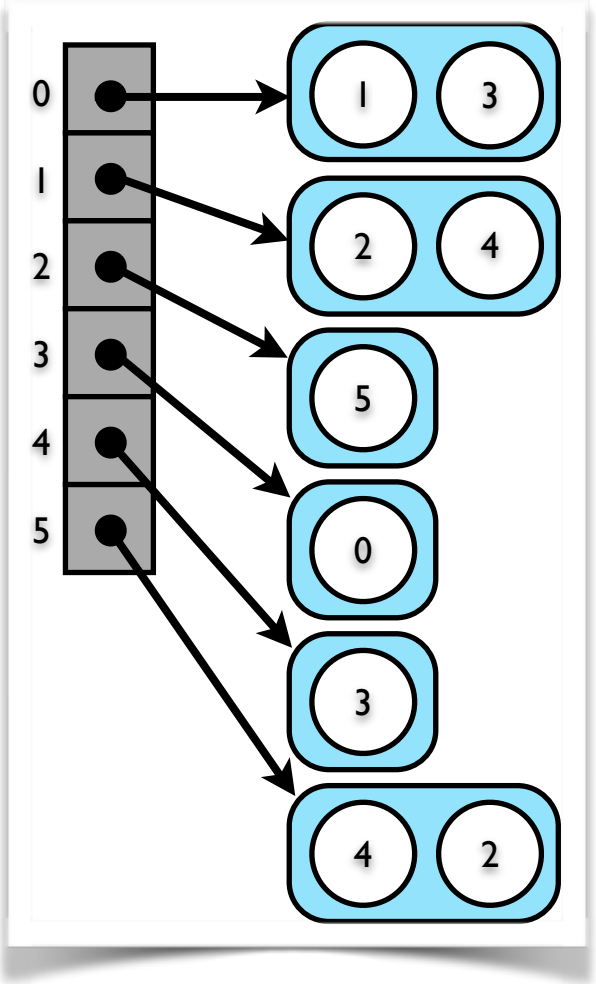


Figure 7. Adjacency list representation for the graph of Figure 1.

To represent the set of neighbors of a node, we can use any of the standard methods — arrays, linked lists, even binary search trees (because you can traverse a BST in linear time).

Adjacency Matrices versus Adjacency Lists

The next table compares adjacency matrices and adjacency lists for undirected graphs. V and E denote the number of nodes and edges, respectively.

	Adjacency Matrix	**Adjacency List**
Scanning the neighbors of v	O(V)	O(degree(v))
Test if u and v are neighbors	O(1)	O(min(degree(u), degree(v))
Space	$O(V^2)$	O(E + V)

Notes

- The time complexity to test whether two nodes are neighbors on an adjacency list assumes that we interleave scans of the lists of neighbors of u and v, stopping if u is found in v's list, v is found in u's list, or the end of either list is reached.

- The adjacency matrix bounds are identical for directed graphs.

- In a directed graph represented by an adjacency list, scanning the neighbors of u takes O(outdegree(u)) time.

- Scanning the neighbors of a node u in a directed graph represented by an adjacency list takes O(outdegree(u)) time; so does determining if v is a neighbor of u.

- The space requirement for an adjacency matrix is $O(V^2)$, because it is a V×V boolean matrix.

- The space requirement for an adjacency list is O(V+E):

  - There are V sets of neighbors (some might be empty).

  - For undirected graphs, each edge (u,v) contributes to *two* neighbor sets: v must be in the set of u's neighbors, and u must be in the set of v's neighbors.
    - Total contribution of all edges is 2E.

  - For directed graphs, each edge (u,v) contributes to only *one* neighbor set: v must be in the set of u's neighbors.
    - Total contribution of all edges is E.

The choice between adjacency matrices and adjacency lists depends on at least two factors: the number of nodes in the graph and the ***density*** of the graph. The density of a graph is the ratio of the number of edges to the maximum possible number of edges. For an undirected graph, the maximum number of edges is $V \cdot (V - 1)/2$, which is roughly $V^2/2$, and is achieved when there is an

edge between every pair of nodes[1]. For a directed graph, the maximum number of edges is $V \cdot (V - 1)$, which is roughly V^2, because for every two nodes u and v, we can have two edges: (u,v) and (v,u).

- If a graph has few nodes, an adjacency matrix is a good choice, regardless of the density, because it is simple and lightweight.

- If the graph is large and *sparse* (i.e., not dense), an adjacency list is more space-efficient than an adjacency matrix. It is also more time-efficient for scanning the neighbors of a node.

- For large dense graphs, adjacency matrices are a good choice: they are simple, lightweight, and space- and time-efficient. Note, though, that large dense graphs are rare. For instance, as of early 2014, the Facebook graph had about 1.23 billion users (nodes), but the average user had 338 friends[2] — that's *sparse*.

`HashMap` / `HashSet` Representation

The two graph representations we have seen have some limitations:

---

[1] A graph like this is said to be ***complete***.

[2] http://www.theguardian.com/news/datablog/2014/feb/04/facebook-in-numbers-statistics

1. Nodes are often complex objects with names, like “Bob” or “Australia”; they are not just integers.

2. In an adjacency list, checking if a node u is adjacent to another node v is time-consuming: you have to scan the lists for u and v sequentially, which can take time linear in the number of nodes. Checking adjacency in an adjacency matrix takes O(1) time, but, as we saw, such matrices are not space-efficient for large sparse graphs.

3. Graphs are often dynamic: nodes and edges keep getting added and deleted (think, e.g., of all the friending and un-friending that goes on in Facebook).

We can address issue 1 by using a `Map` object to map names of nodes to their respective lists of neighbors. If we use a `HashMap` from vertex names to `List` objects, we can find the list for any node in O(1) (under the usual assumption about `HashMaps`).

We can address issue 2 — testing for adjacency quickly — by representing the neighbors as a `HashSet`, rather than a List. Thus, the underlying map is of the form

```
HashMap<V, HashSet<V>>
```

This means that each of the following operations takes O(1) time (under the usual assumptions about `HashMaps`

and `HashSets`):

- accessing the set of neighbors of a node,
- testing if two nodes are adjacent,
- adding/deleting a node, and
- adding/deleting an edge.

The last two points address issue 3.

An implementation of the `HashMap`/`HashSet` representation of undirected graphs called `Graph<V>` is posted on Blackboard.

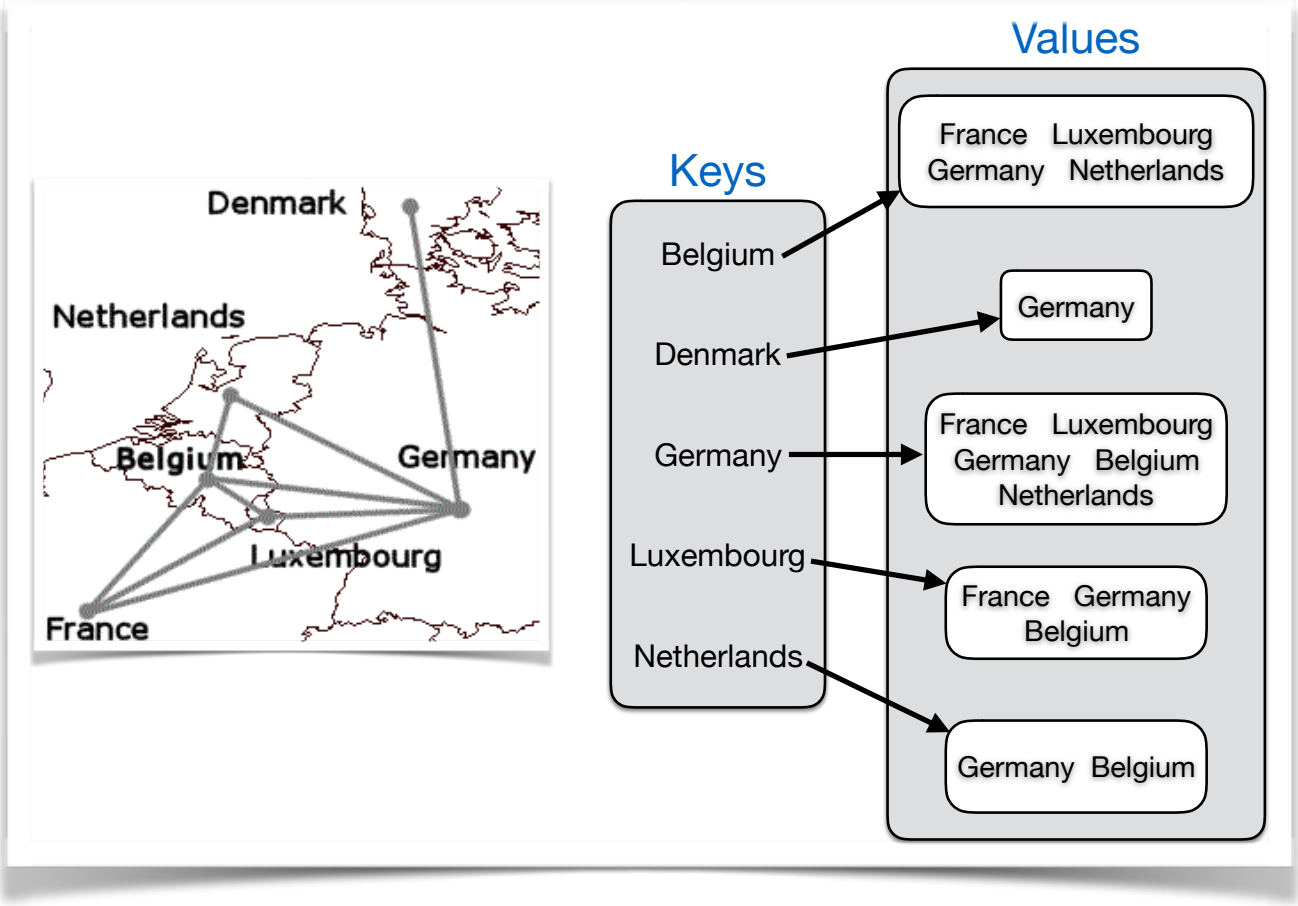


Figure 8. The graph of Figure 1 and its HashMap/HashSet representation.

Graph Traversals

A ***graph traversal*** is a way to explore the nodes and edges of the graph. The idea is similar to traversing a tree. This is not surprising: a tree is just a special kind of graph. We will study two types of graph traversals: depth-first search (DFS) and breadth-first search (BFS). DFS and BFS can be used on both directed and undirected

graphs. Java code and detailed examples for BFS and DFS are posted on Blackboard.

Note. Much of the material in the remaining lectures is based on the following book, which we shall refer to as "CLRS".

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms* (3rd ed.). MIT Press, 2009.

Breadth-First Search

Suppose we pick some node s as a starting node. The ***distance*** from s to a node v (possibly s itself) is the minimum number of edges in a (simple) path from s to v. The distance from s to itself is obviously 0.

BFS visits nodes by increasing distance from the starting node, s. BFS resembles level-order tree traversal. Like level-order tree traversal, BFS uses a queue Q. BFS also maintains a value dist(v) for each vertex v, which is an estimate of the distance from s to v. Initially,

- Q contains only s,

- dist(s) = 0 — the distance from s to itself is zero —, and

- dist(v) = ∞ for every node v other than s — since we haven't found any path from s to v yet[3] .

The algorithm then does the following while Q is not empty:

1. Let u be the node at the front of Q.

2. ***Process*** u: for each neighbor v of u, if we are discovering v for the first time, we make dist(v) = dist(u) + 1 and enqueue v.

3. Dequeue u.

Unlike level-order traversal of a tree, we may encounter the same node multiple times during a BFS (in a tree there is just one path from the root to any node, but this is not so for a general graph). To keep track of the status of the nodes, we maintain a color map that assigns each vertex v one of three colors:

- color(v) == white means v is undiscovered and unprocessed.

- color(v) == grey means v has been discovered — so it is in Q —, but has not been processed.

[3] In an actual program, instead of ∞, we'd use a very large number, such as `Integer.Max_Value`.

- color(v) == black means v has been discovered and processed, so it is no longer in Q.

Initially, color(s) = grey and, for every node other than s, color(v) = white. At all times, the queue contains all the grey nodes, and no other nodes.

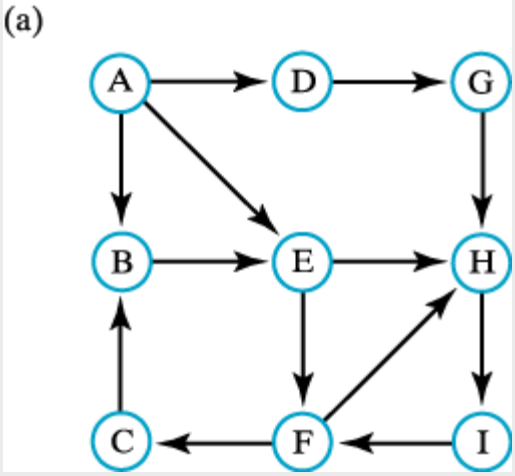
The pseudocode is on the next page.

```
BFS(G,s):
    let Q be an empty queue
    foreach node v in G except s
    |   color(v) = white
    |   dist(v) = ∞
    |   pred(v) = null

    color(s) = grey
    dist(s) = 0
    pred(s) = null
    Q.enqueue(s)

    while (!Q.isEmpty())
    |   let u = Q.front()
    |   foreach neighbor v of u
    |   |   if color(v) == white
    |   |   |   color(v) = grey
    |   |   |   dist(v) = dist(u) + 1
    |   |   |   pred(v) = u
    |   |   |   Q.enqueue(v)
    |   Q.dequeue()
    |   color(u) = black
    return dist and pred
```

BFS maintains two values for each node v. The first — which we mentioned last time — is dist(v), the current estimate of the distance from s to v. The second is pred(v), the ***predecessor*** of v on the shortest path from s to v. Initially, pred(v) is null. Suppose BFS first discovers

v when scanning the neighbors of some node u. Then, at this point, BFS sets pred(v) = u. Since each node other than s has at most one predecessor, the edges (pred(v),v), for v ≠ s, form a tree rooted at s, called a ***BFS tree***.

Example. Suppose we perform BFS on the graph below, starting at A. Assume that neighbors are scanned in alphabetical order.



Here is the queue (we alternate between red and black to mark successive levels —i.e., nodes at equal distance from the start):

```
---     -----     ----     ----     -----
 A  ->   BDE  ->   DE  ->   EG  ->   GFH
---     -----     ----     ----     -----

----     ----     ----     ---
 FH  ->   HC  ->   CI  ->   I  ->   empty
----     ----     ----     ---
```

v	A	B	C	D	E	F	G	H	I
dist(v)	0	1	3	1	1	2	2	2	3
pred(v)	null	A	F	A	A	E	D	E	H

Implementation note. We can represent dist as a map from nodes to integers, and pred as a map from nodes to nodes.