Graphs and Breadth-First Search

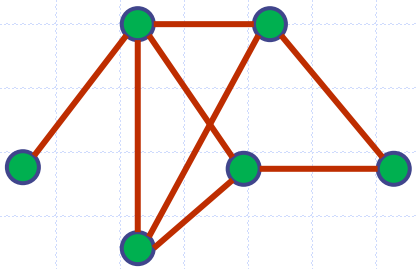
Chapter 22

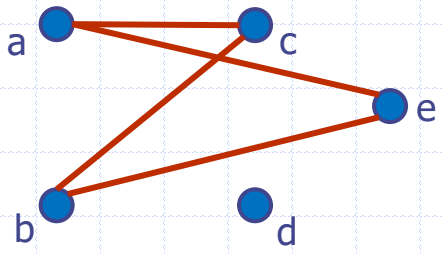
Graphs

A graph G is a pair (V, E) where V is an arbitrary set and E is a set of 2-element subsets of V.

The elements of V are called the vertices or nodes of the graph. The elements of E are called the edges or arcs of the graph.

A graph can be visualized by drawing elements of V as dots, and elements of E as line segments between dots.





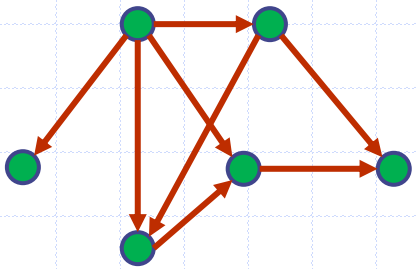
V = {a, b, c, d, e}
E = {ac, cb, be, ea}

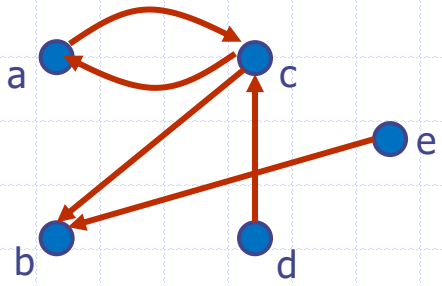
Digraphs

A directed graph or digraph G is a pair (V, E) where V is an arbitrary and E is a subset of V × V.

The difference from a graph is that digraph edges are directed from one vertex to another.

A digraph can be visualized by drawing elements of V as dots, and elements of E as arrows between dots.



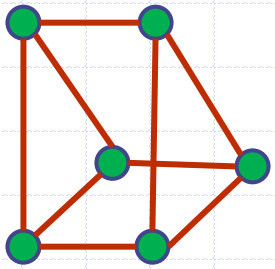


V = {a, b, c, d, e}
E = {ac, ca, cb, dc, eb}

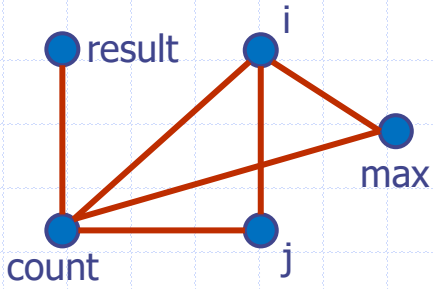
Applications

Graphs and digraphs have a multitude of applications in computing.

Consider a graph (V, E) where V = some processors and E = physical connections between the processors. This is a network interconnection graph.

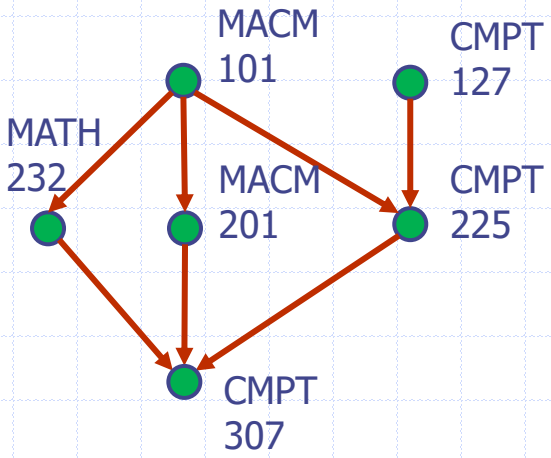


Or a graph (V, E) where V = the variables in a program, and E contains an edge v_1v_2 if the two variables v_1 and v_2 must be in memory at the same time. This is called a conflict graph.

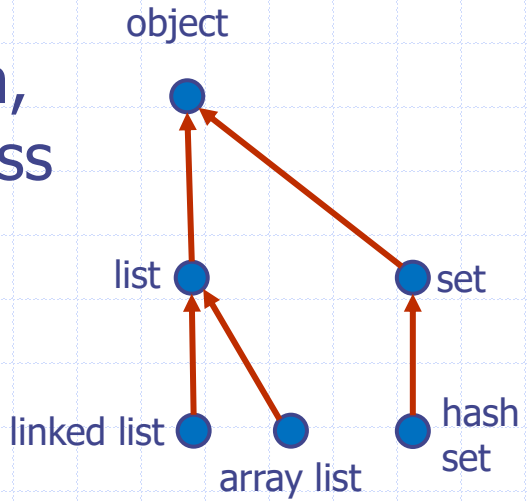


Applications

Consider a digraph (V, E) where V = courses in university and edge **ab** is in E if **a** is a prerequisite of **b**. This is the prerequisite digraph.



Or a digraph (V, E) where V = the classes in an object-oriented program, and E contains an edge c_1c_2 if the class c_1 is a subclass of class c_2. This is called a class hierarchy.



Representations of Graphs

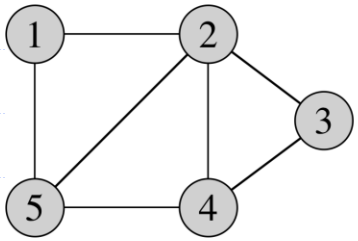
There are three major representations of graphs and digraphs in widespread use.

The first of these is the adjacency-list representation. Here each vertex stores a list of adjacent vertices.

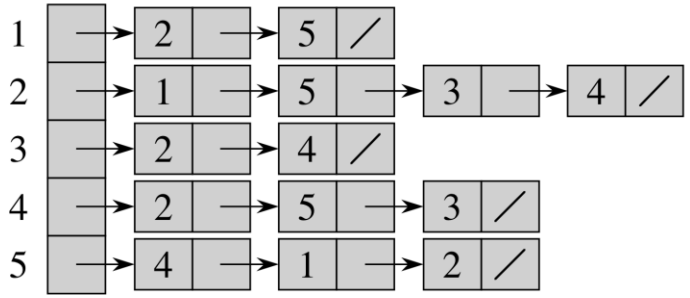
The second of these is the adjacency-matrix representation. Here a $|V| \times |V|$ matrix M, where an entry M(i, j) stores 0 if there is no edge from i to j, or 1 if there is an edge from i to j.

Adjacency-list is excellent for sparse graphs and okay for dense ones. Adjacency-matrix is good for dense graphs.

Representations of Graphs



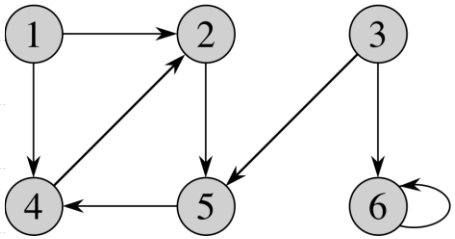
(a)



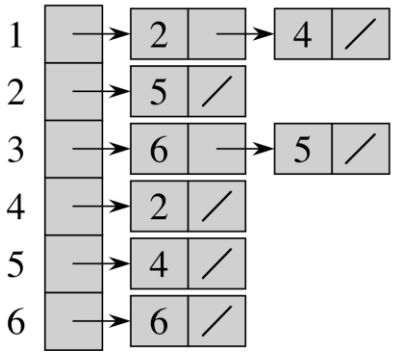
(b)

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

(c)



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Representations of Graphs

The third representation is the edge-list representation. Here each vertex stores a list of adjacent edges.

The edge-list representation is very similar to the adjacency-list representation. It is excellent for sparse graphs, and for when edges have data associated with them. It is the most frequently found object-oriented representation.

An edge-list representation typically has an object for the graph that contains a sequence (list or array) of **vertex object** references. Each vertex object contains a list of edge object references for the edges it is a part of. Each edge object contains references to the two vertex objects for the vertices that are its endpoints.

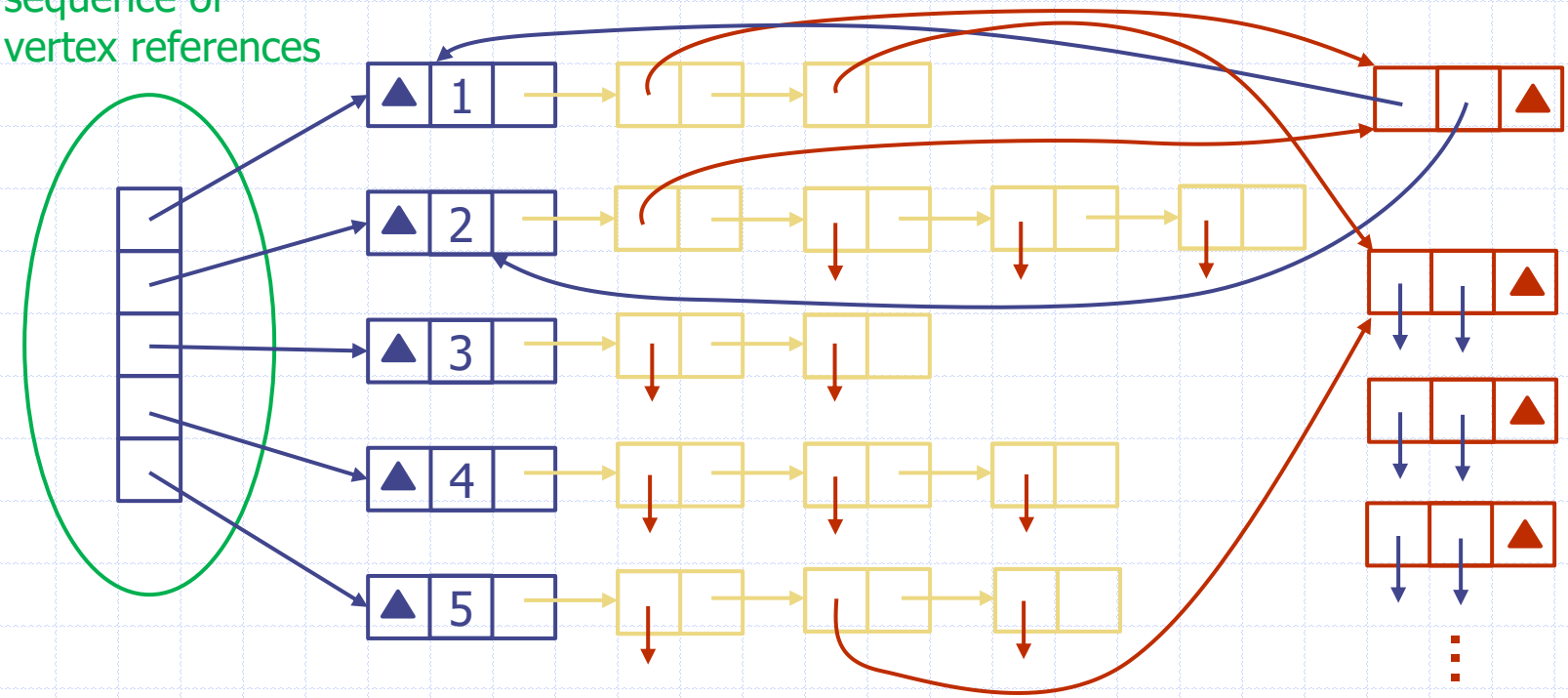
Theoretically, adjacency-list and edge-list representations are nearly equivalent, so the text does not go into edge-list representations.

Edge-List Representation

Graph object: sequence of vertex references

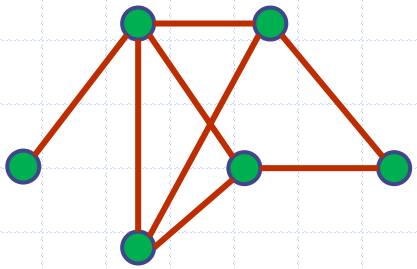
Vertex objects: data ▲ and a list of edges

Edge objects: data ▲ and two vertex references

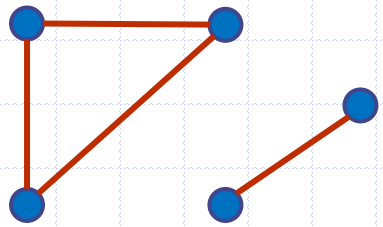


Breadth-First Search

Breadth-first search (BFS) is a way of examining every node in a connected graph. A graph is connected if every pair of vertices is joined by a path of one or more edges.



connected



not connected

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

Breadth-First Search

Breadth-first search can optionally label each vertex with its distance from the start vertex. It can also optionally construct a tree, known as a Breadth-First Search Tree.

In a BFS tree, each node v except the start node has a predecessor or parent node, which is the node whose exploration first discovered v. A BFS tree is a tree of minimum height contained in the graph that is rooted at the start node and contains all of the nodes of the graph.

BFS is a classic worklist algorithm, where the work to be done is kept in a list (or other data structure), with items added as they are discovered and removed as they are processed.

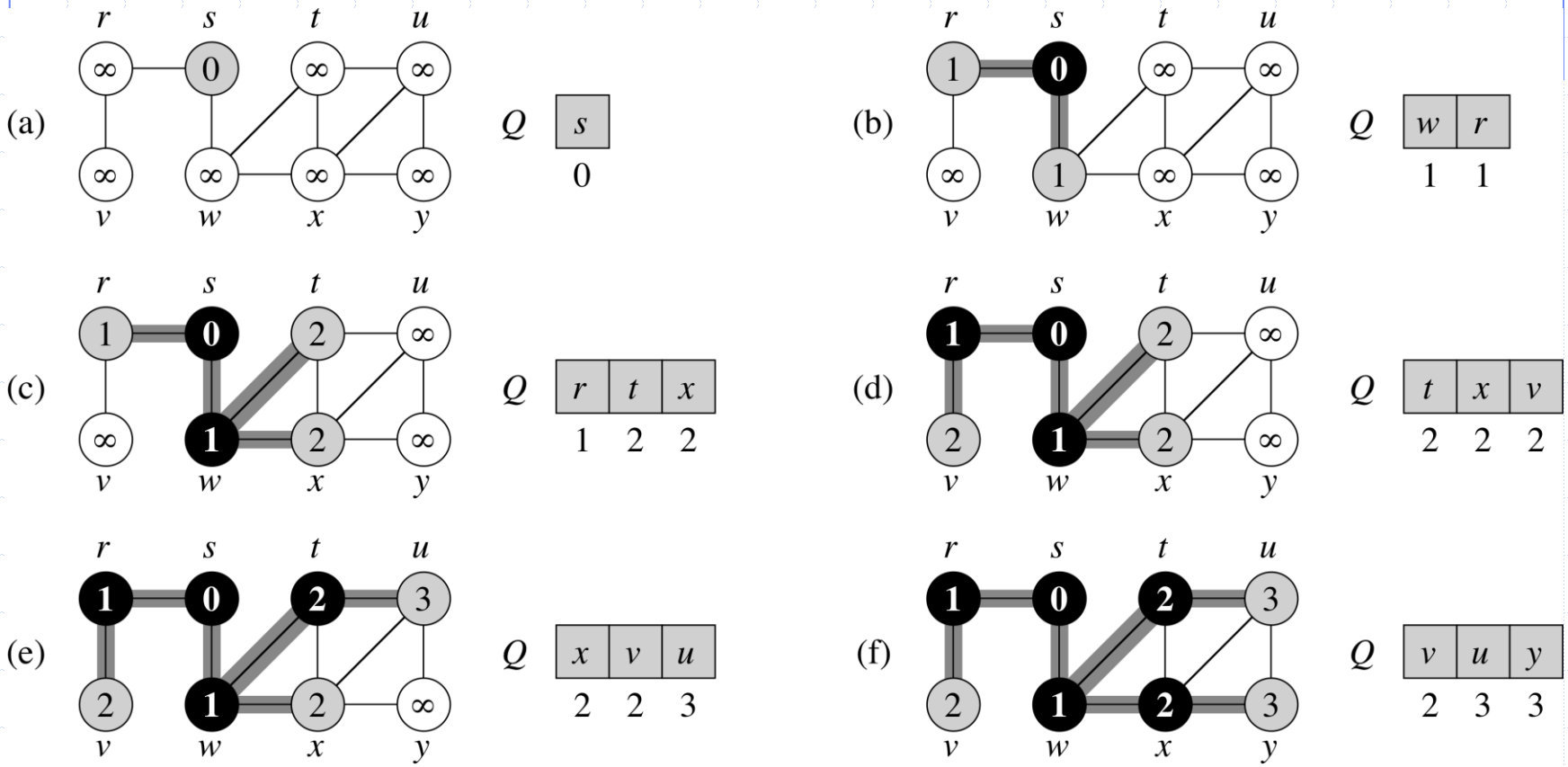
Breadth-First Search

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

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

    s.status = DISCOVERED
    s.distance = 0
    s.predecessor = nil

    Q = new Queue                               // Q is "worklist"
    Q.enqueue(s)
    ...
```

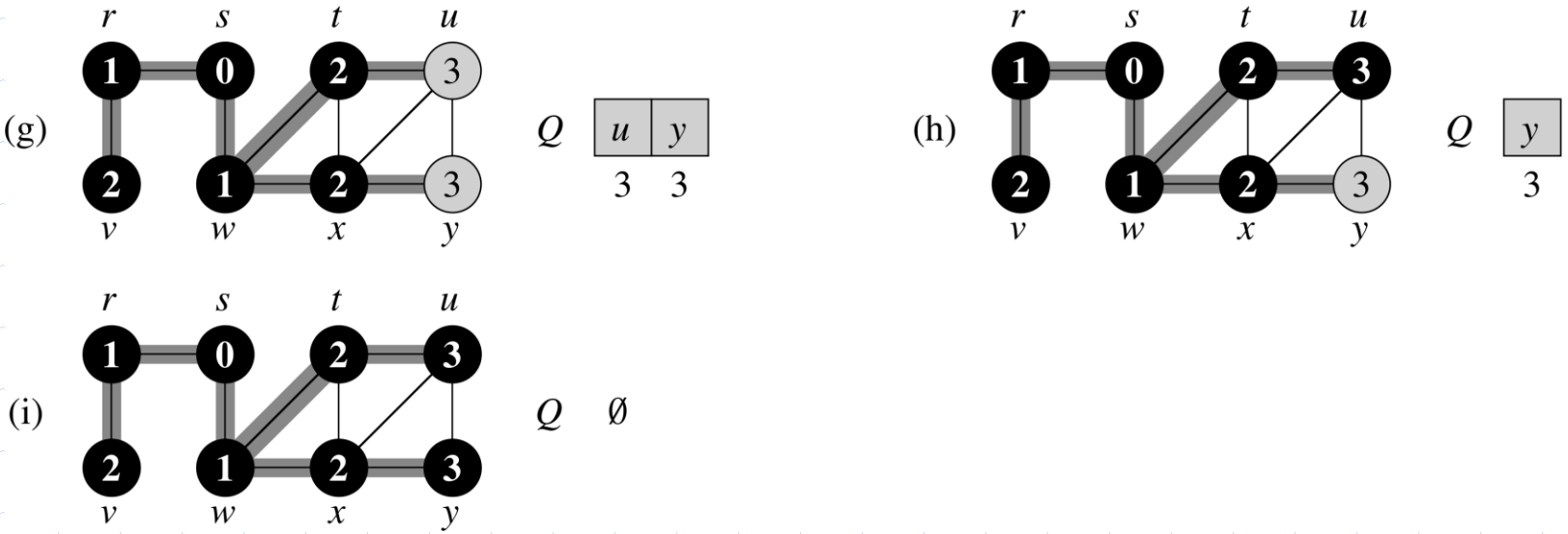
Breadth-First Search

```
while !Q.isEmpty()
        u = Q.dequeue()
        for each v adjacent to u                    // process u
                if v.status = UNDISCOVERED
                        v.status = DISCOVERED
                        v.distance = u.distance + 1
                        v.predecessor = u
                        Q.enqueue(v)

        u.status = PROCESSED
```

Breadth-First Search Example



Breadth-First Search Example



Invariant:

At "while !Q.isempty" the queue contains all vertices with status = DISCOVERED.

Breadth-First Search Analysis

Vertices are marked UNDISCOVERED only at initialization.

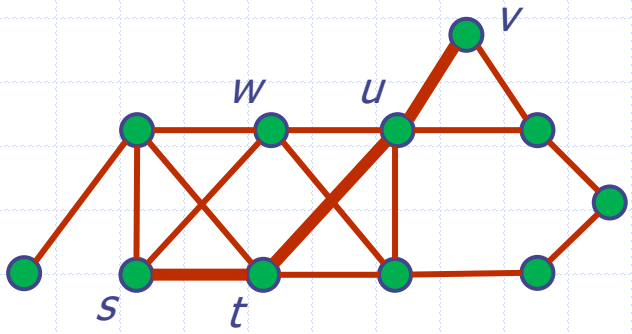
The "if v.status is UNDISCOVERED" test ensures that each vertex is enqueued at most once, and therefore dequeued at most once. Each of these operations takes constant time, so it's O(V) for the queue operations.

The adjacency list of each vertex is scanned exactly once, when the vertex is dequeued. Thus the time spent scanning adjacency lists is at most O(E).

Initialization costs O(V), so the time for BFS is O(V+E). This is linear in the size of the adjacency list representation.

Distance and Shortest Paths

Let the shortest-path distance (or simply distance) $\delta(s, v)$ from vertex s to vertex v be the minimum number of edges in any path from s to v. If there is no path from s to v, then $\delta(s, v) = \infty$.



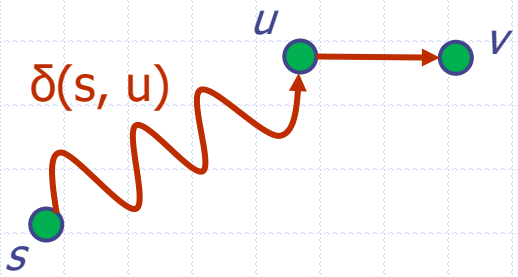
$\delta(s, v) = 3$

stuv is a shortest path
swuv is also a shortest path

A path of length $\delta(s, v)$ from s to v is called a shortest path from s to v.

Lemma on Shortest Paths

Lemma. Let G = (V, E) be a directed or undirected graph. Let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.



Proof. If u is reachable from s, then a shortest path from s to u followed by the edge from u to v is a path from s to v and has length $\delta(s, u) + 1$. If u is not reachable from s, then $\delta(s, u) = \infty$ and the result holds. ■

BFS distance lemma

Lemma. Let $G = (V, E)$ be a directed or undirected graph. Suppose BFS is run on G from a given source vertex s. Upon termination, for each $v \in V$, the value $v.distance$ computed by BFS satisfies $v.distance \geq \delta(s, v)$.

Proof. By induction on the number of ENQUEUE operations. The inductive hypothesis (IH) is that $v.distance \geq \delta(s, v)$ for all $v \in V$.

The basis is when s is first ENQUEUEd. The IH holds here because $s.distance = 0 = \delta(s, s)$ and for all other vertices v, $v.distance = \infty \geq \delta(s, v)$.

BFS distance lemma

For the inductive step, let v be an UNDISCOVERED vertex that is discovered during the loop for vertex u's neighbors.

$v.distance = u.distance + 1$ by the assigment in the **if**

$\geq \delta(s, u) + 1$ by the IH

$\geq \delta(s, v)$ by the previous lemma

Vertex v is then enqueued, and it is never enqueued again because its status has changed to DISCOVERED and never changed back to UNDISCOVERED. So v.distance never changes again, and the IH is maintained. ■

Queue distances lemma

Lemma. Suppose that during the execution of BFS on graph G = (V, E), the queue Q contains the vertices $\langle v_1, v_2, \ldots v_r \rangle$, where v_1 is the head of the queue and v_r the tail. Then

$v_r.distance \leq v_1.distance + 1$, and
$v_i.distance \leq v_{i+1}.distance$ for $i = 1, 2, \ldots, r-1$.

Proof. By induction on the number of queue operations. The basis is when the queue contains only s, when the lemma trivially holds (r = 1).

If v_1 is dequeued, then v_2 becomes the new head. By induction, $v_r.distance \leq v_1.distance + 1$, which is $\leq v_2.distance + 1$.

Queue distances lemma

All other inequalities are unaffected, so the lemma holds after a dequeue.

If some vertex v is enqueued while scanning vertex u's neighbors, v becomes v_{r+1}. u was dequeued before scanning commenced, and it satisfied

$$u.distance \le v_1.distance.$$

So
$$v_{r+1}.distance = v.distance = u.distance + 1 \le v_1.distance + 1$$

From the IH we have $v_r.distance \le u.distance + 1$, and so $v_r.distance \le v_{r+1}.distance$. Again, the remaining inequalities are unaffected and so the lemma holds after v is enqueued. ■

Queue distances corollary

Corollary. Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then

$$v_i.\text{distance} \leq v_j.\text{distance}$$

at the time when v_j is enqueued.

Proof. Apply previous lemma and note that each vertex receives a finite distance value at most once during the course of the BFS. ■

Correctness of BFS

Theorem. Let BFS be executed on G = (V, E) from source vertex s. Then BFS discovers every vertex that is reachable from s, and upon termination, v.distance = δ(s, v) for all vertices $v \in V$. Furthermore, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to v.predecessor followed by the edge (v.predecessor, v).

Proof. To prove v.distance = δ(s, v) we use proof by contradiction. So assume that some vertex gets a distance value that is not equal to its shortest path distance. Let v be a vertex with minimum δ(s, v) that receives such an incorrect value. Trivially, $v \neq s$.

Correctness of BFS

By the BFS distance lemma, v.distance ≥ δ(s, v), so it must be that v.distance > δ(s, v). Let u be the vertex immediately preceding v on a shortest path from s to v. Then δ(s, v) = δ(s, u) + 1. Now u.distance = δ(s, u) and v.distance > δ(s, v) = δ(s, u) + 1 = u.distance +1. (*)

Consider v.status when BFS dequeues u. If v.status is UNDISCOVERED, then in processing u we set v to DISCOVERED and v.distance to u.distance +1, contradicting (*). If v.status is PROCESSED, then it was already removed from the queue and has v.distance ≤ u.distance, by the Queue Distances Corollary. This again contradicts (*). If v.status is DISCOVERED, then it was in

Correctness of BFS

the queue when u was dequeued, and the Queue Distances Lemma tells us v.distance ≤ u.distance +1, again contradicting (*).

As every case has led to a contradiction, we have that v.distance = δ(s, v) for all vertices v ∈ V. All vertices reachable from s must be discovered, as otherwise their distance value would be infinite. Next, observe that if v.predecessor = u, then v.distance = u.distance + 1. Therefore we obtain a shortest path from s to v by taking a shortest path from s to u followed by the edge uv. ■

Be sure to read the end of section 22.2 on BFS Trees.