Breadth-first search

Breadth-first search is a method for searching through all the nodes in a graph that also takes $\Theta(m+n)$ time. It may also be used to find the shortest path (in an unweighted graph) from a given node to every other node. The idea is to start at a node, and then visit the neighbors of the node, and then visit the neighbors of its neighbors, etc until the entire graph is explored. All vertices close to the node are visited before vertices far from the node.

When we first discover a node, we keep track of its **parent** vertex, i.e. which vertex triggered the initial visit to that node. These parent-child relations form a **breadth-first search tree**, which contains a subset of the edges in the graph, and all the vertices reachable from the start node. The root is the start node, and the neighbors of the root become the root's children in the breadth-first-search-tree, and the neighbors of the neighbors become the children of the neighbors, etc. Some of the edges in the graph are not included in the breadth first search tree, because they point to vertices that were already seen on higher levels of the tree.

Using the breadth-first-search tree, we find paths from the start node to every other node in the graph. (We can prove that those paths are the shortest paths possible, and we will do this in class today.) We also keep track of the distance from the start node to every other node, which is basically how many levels away you are in the breadth-first-search tree.

3.0.1 Brief algorithm

In breadth-first search, we give each vertex a color: white ("unvisited"), grey ("discovered"), or black ("finished"). We start by placing the start node onto a queue (and marking it grey). On each iteration of the whole loop, we remove an element from the queue (marking it black), and add its neighbors to the queue (marking them grey). We do this until the queue has been exhausted. Note that we only add white ("unvisited") neighbors to the queue, so we only dequeue each vertex once.

Briefly, we

```
Mark all vertices white, except the start node s, which is grey.
Add s to an empty queue Q.
while Q is nonempty: node =
    Dequeue(Q) for each neighbor in
    Adj[node]:
        if neighbor.color is white:
            neighbor.color = gray
            Enqueue(Q, neighbor)
    node.color = black
```

You'll notice this pseudocode doesn't really do anything, aside from coloring the vertices. To use breadth-first search in real life, you would probably want to modify this code to do something every time you processed a vertex.

For example, if you wanted to find the average number of followers of users on Twitter, you could start at a user, and use breadth-first search to traverse the graph. Every time you mark a user black, you can count how many followers they have, and add this number to a database. (This actually is technically possible using the Twitter API, but it would take you forever because Twitter restricts the rate at which you can get data from their servers.)

You might also wonder what is the point of marking users black, because black and grey users are pretty much equivalent in the pseudocode. The main point is to make the correctness proof easier. Having three colors more closely mirrors the three states a vertex can be in, which makes things easier to reason about.

3.0.2 Full algorithm

But I haven't gotten to the part where we compute the shortest paths. To find the shortest paths, we use the full algorithm, which is shown here.

The full algorithm keeps two additional attributes for each node: d (which is the distance from the start node to that node), and π (which is the "parent" node, i.e., the node right before it on the shortest path). Whenever BFS discovers a new node v by way of a node u, we can prove that this was the "best way" of discovering v, so v's parent is u, and v's distance from the source node is one plus u's distance from the source node.

Note: in depth first search there is also a d attribute, but it means something different (in DFS it refers to the discovery time of a vertex, whereas in BFS it refers to the distance).

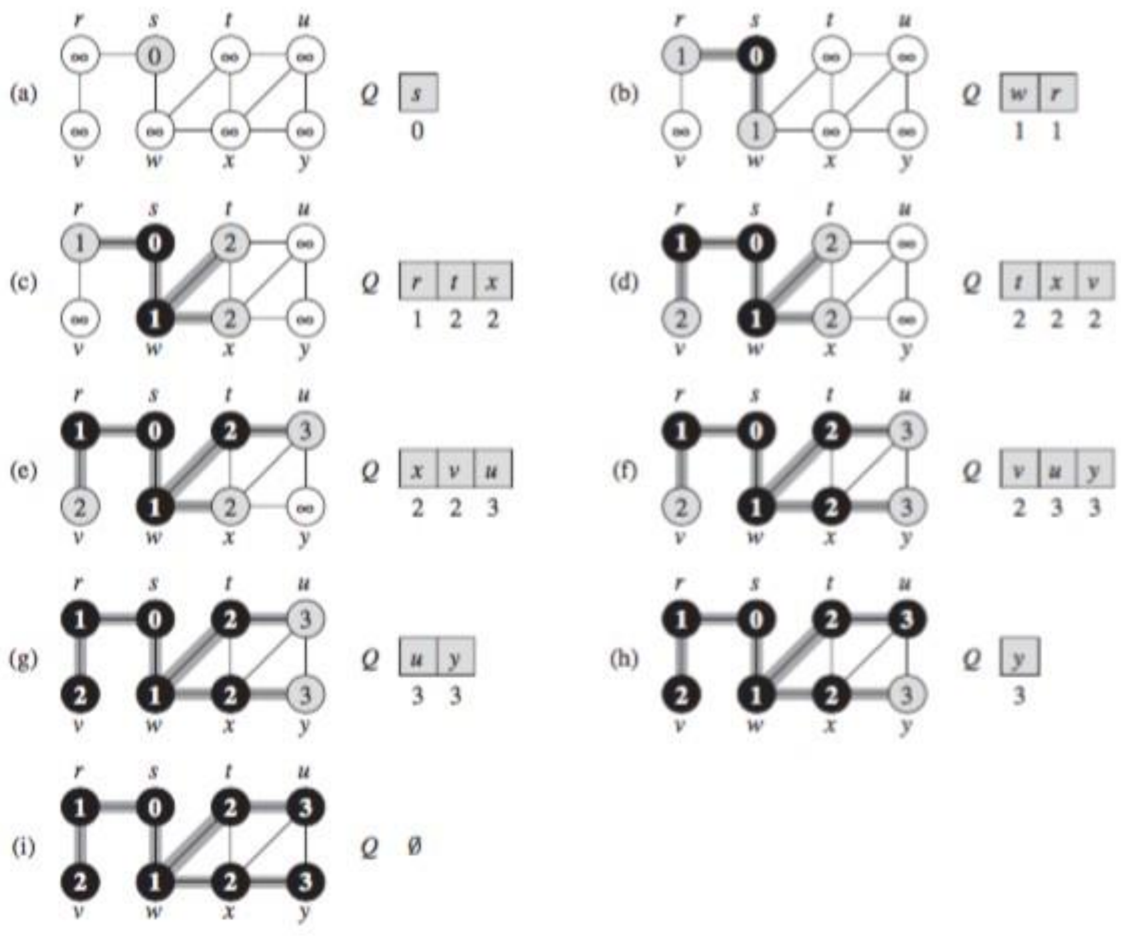
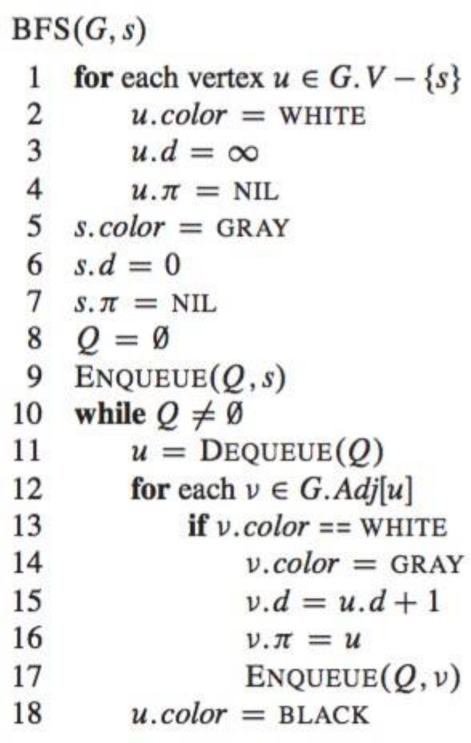


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of $u.d$ appears within each vertex u. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear below vertices in the queue.



```
BFS($G, s$)
for each vertex $u \in G.V - \{s\}$
    $u.color$ = WHITE
    $u.d = \infty$
    $u.\pi$ = NIL
$s.color$ = GRAY
$s.d = 0$
$s.\pi$ = NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in G.Adj[u]$
        if $v.color$ == WHITE
            $v.color$ = GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            ENQUEUE($Q, v$)
    $u.color$ = BLACK
```

3.0.3 Runtime

Breadth-first-search runs in $O(m + n)$. Because we only add white vertices to the queue (and mark them grey when they are added), and because a grey/black vertex never becomes white again, it follows that we only add a vertex to the queue at most once during the entire algorithm. So the total time taken by the queue operations is $O(n)$. The total number of iterations of the for loop is $O(m)$, because each adjacency list is iterated through at most once (i.e. when the corresponding vertex is removed from the queue), and there are m entries total if you aggregate over all of the adjacency lists.