CS 624
Lecture 12: Graph Walks

1 Breadth-first search

Let G be an undirected graph. Let the set of vertices in G be denoted by V, and the set of edges be denoted by E. If we want to make it clear which graph we are talking about, we can denote the set of vertices by $V[G]$ and the set of edges by $E[G]$.

> Another term for vertex is *node*. Another term for edge is *arc*.

One way to represent a graph is by a set *adjacency lists*, one for each vertex. That is, for each vertex $v \in V$, we have a list $Adj[v]$ consisting of those vertices u such that $(v, u) \in E$.

> Note, by the way, that $Adj[v]$ is really a set—the order doesn't matter. But since it is almost always implemented internally as a list, it is usually referred to as an "adjacency list".
>
> Also note that this representation works just as well for directed graphs. In this case, the edge (v, u) means the edge starting from v and ending at u. (Equivalently, its tail or "source" is v, and its head or "target" is u.)

For simplicity, we will assume that our graphs have no edges from a node to itself. (Such edges are generally called *self-loops*.)

1.1 Lemma

$$\sum_{v \in V} length\big(Adj[v]\big) = 2|E|$$

Proof. Each edge is a member of exactly 2 adjacency lists—one for each of its two vertices. So if we visit each element of each adjacency list, we will visit each edge exactly twice. □

1.2 Exercise

1. *What goes wrong with this proof if G is allowed to contain self-loops?*
2. *If you wanted to allow self-loops, how would you modify the statement of the Lemma?*

The algorithm in Figure 1 implements a breadth-first walk (also called a breadth-first search) on the undirected graph G, starting from some arbitrary node s.

The key mechanism in this algorithm is the use of a queue, denoted by Q.

```
BFS(G, s)

    // Initialization:
    for each vertex u ∈ V[G] \ {s} do
        color[u] ← WHITE
        d[u] ← ∞
        π[u] ← NIL
    color[s] ← GRAY
    d[s] ← 0
    π[s] ← NIL
    Q ← ∅
    ENQUEUE(Q, s)

    // Graph traversal:
    while Q ≠ ∅ do
        u ← DEQUEUE(Q)
        for each v ∈ Adj[u] do
            if color[v] = WHITE then
                color[v] ← GRAY
                d[v] ← d[u] + 1
                π[v] ← u
                Mark the edge from u to v as a "tree edge".
                ENQUEUE(Q, v)
        color[u] ← BLACK
```

Figure 1: This algorithm implements a breadth-first walk of the graph G starting at the vertex s.

Figure 2 shows the progress of this algorithm on an undirected graph. Note that all nodes are initially colored white. A node is colored gray when it is placed on the queue. And it is colored black when it is taken off the queue.

So the nodes colored white have not yet been visited. The nodes colored black are "finished". And the nodes colored gray are still being processed.

At the point when a node is placed on the queue, the edge from the first node in the queue (which is being taken off the queue) to that node is marked as a *tree edge* in the breadth-first tree. As we will see below, these edges actually do form a tree (called the breadth-first tree) whose root is the start node s.

What is the cost of this algorithm? There is one ENQUEUE and one DEQUEUE for each vertex. So the total cost of all the ENQUEUE and DEQUEUE operations is $2|V|$. The rest of the cost of the algorithm occurs when visiting the edges of each adjacency list. Each adjacency list is processed once (when its vertex is removed from the queue by a DEQUEUE operation). So the cost of processing the adjacency lists is the sum of their lengths. And we have seen that the sum of the lengths of all the adjacency lists is $2|E|$. Therefore, the total cost of the algorithm is $O(|V| + |E|)$.

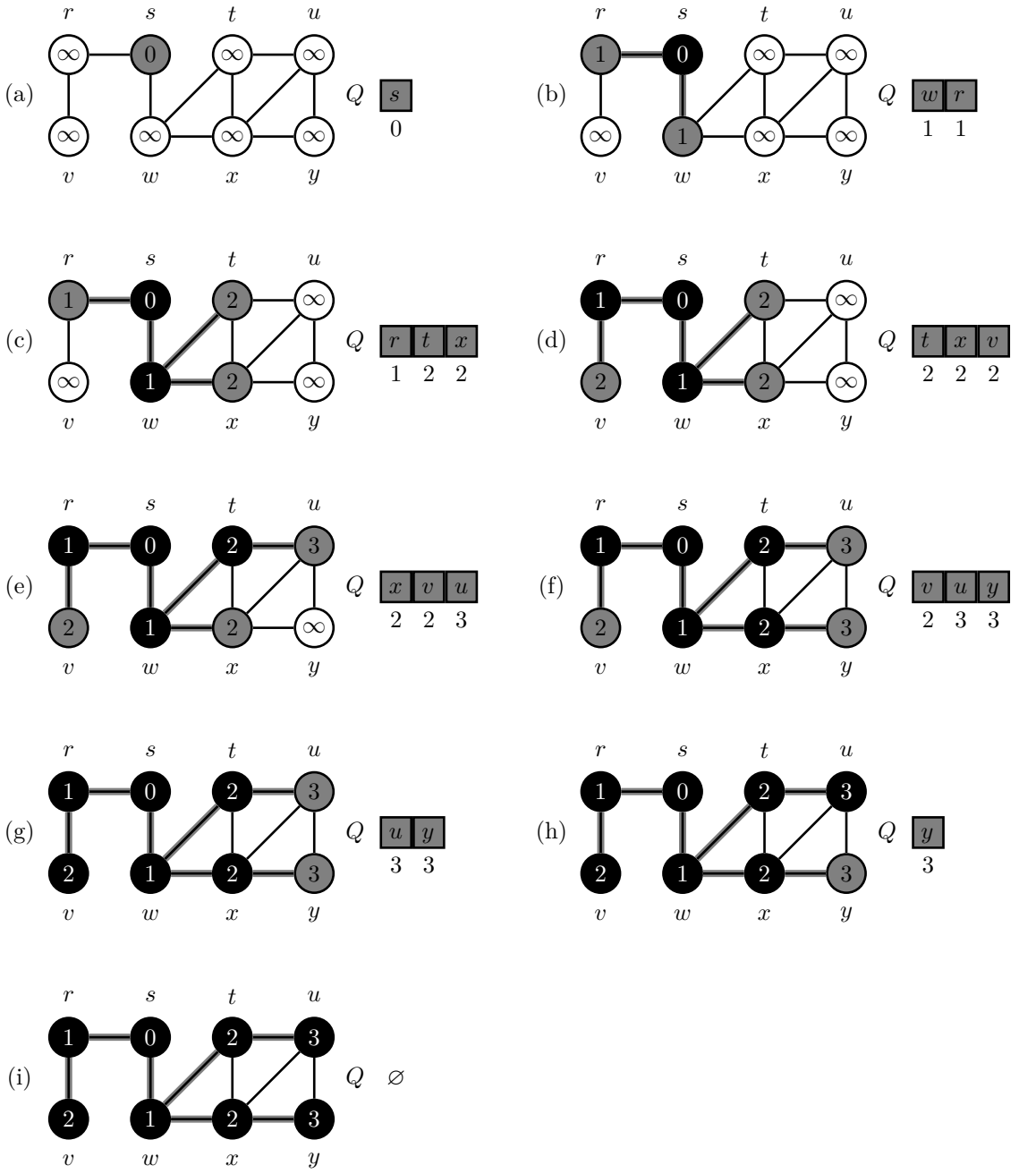

Figure 2: The progress of the breadth-first search algorithm on a connected undirected graph G. The start node is s. The tree edges produced by the algorithm are shown as thickened, and they in fact form a spanning tree of G.

Now let us prove some properties of this algorithm, and show that it does what we think it should:

1.3 Lemma *If G is connected, then the breadth-first tree constructed by this algorithm*

- *really is a tree, and*
- *contains all the nodes in the graph.*

PROOF. A node becomes the target of a tree edge when that node is placed on the queue. And since that only happens once, no node is the target of two tree edges.

Next, let us show that every node that is processed by the algorithm is reachable by a chain of tree edges from the root. It is enough to prove the following statement:

> At the time at which a node is placed on the queue, it is reachable by a chain of tree edges from the root.

Now this statement is clearly true at the beginning: There is only one node in the queue and it is the root.

We proceed by induction. Suppose it is true up to some point. Now when the next node v is placed on the queue, v is an endpoint of an edge whose other endpoint is the node at the head of the queue, and that edge is made a tree edge. And by the inductive assumption, the node at the head of the queue is reachable by a path of tree edges from the root. So taking that path and appending the new edge to it gives a path of tree edges from the root to v, which is what we needed to show.

Thus, every node that is processed by the algorithm is reachable by a chain of edges from the root.

That shows that the edges form a tree.

Suppose there was one node (v, say) that was not reached by this process. Then since G is connected, there would have to be a path from the root to v. On that path there would be a *first* node (w, say) which was not in the tree. (That node might be v, or it might come earlier in the path.) That means that the edge in the path leading to that node starts from a node in the tree. At some point, that node in the tree was at the head of the queue. Therefore, w would have been placed in the queue by the algorithm, and the edge to w would have been a tree edge. And that's a contradiction. □

1.4 Lemma *If at any point in the execution of the* BFS *algorithm the queue consists of the vertices $\{v_1, v_2, \ldots, v_n\}$, where v_1 is at the head of the queue, then $d[v_i] \leq d[v_{i+1}]$ for $1 \leq i \leq n-1$, and $d[v_n] \leq d[v_1| + 1$.*

Remark *That is, the assigned depth numbers increase as one walks down the queue, and there are at most two different depths in the queue at any one time (and if there are two, they are consecutive).*

PROOF. The result is true trivially at the start of the program, since there is only one element in the queue. So let us proceed by induction.

At any step, a vertex is added to the tail of the queue only when it is reachable from the vertex at the head (which is being taken off). And the depth assigned to the new vertex at the tail is 1 more than that of the vertex at the head. So by the inductive hypothesis it is greater than or equal to the depths of any other vertex on the queue, and no more than 1 greater than any of them. □

1.5 Lemma *If two nodes in G are joined by an edge in the graph (which might or might not be a tree edge), then their d values differ by at most 1.*

PROOF. Say the nodes are v and u. One of them is reached first in the breadth-first walk. Without loss of generality, say v is reached first. So v is put on the queue first, and reaches the head of the queue before u does. Now when v reaches the head of the queue, there are two possibilities:

- u has not yet been reached. In that case, when we take v off the queue, since there is an edge from v to u, u will be put on the queue and we will have $d[u] = d[v] + 1$.
- u has been reached and therefore is on the queue. In this case, we know from Lemma 1.4 that $d[v] \leq d[u] \leq d[v] + 1$.

And that concludes the proof. □

1.6 Theorem *If G is connected, then the breadth-first search tree gives the shortest path from the root to any node.*

PROOF. We know there is a path in the tree from the root to any node. Further, we know that the depth of any node in the tree is the length of the path in the tree from the root to that node.

So for each node v in the tree, we have

$$d[v] = \text{the length of the path in the tree from the root to } v$$

and let us set

$$s[v] = \text{the length of the shortest path in } G \text{ from the root to } v$$

We are trying to prove that $d[v] = s[v]$ for all $v \in G$. And certainly in any case we know just by the definition of $s[v]$ that $s[v] \leq d[v]$ for all v.

Suppose there is at least one node for which the theorem is not true. All the nodes w for which the statement of the theorem is not true satisfy $s[w] < d[w]$. Among all those nodes, pick one—call it v—for which $s[v]$ is smallest.

Now let u be the node preceding v on a shortest path from the root to v. We have

$$\begin{aligned} d[v] &> s[v] \\ s[v] &= s[u] + 1 \\ s[u] &= d[u] \end{aligned}$$

and hence

$$d[v] > s[v] = s[u] + 1 = d[u] + 1$$

But by Lemma 1.5, this is impossible. □

Based on this, we can easily write down an algorithm to print out the shortest path from a start node s to any other node v in the graph G. We assume that $\mathrm{BFS}(G, s)$ has already been run, so that each node x has been assigned its depth $d[x]$.

```
PRINT-PATH(G, s, v)
    if v = s then
        print s
    else if π[v] = NIL then
        print "no path from" s "to" v "exists"
    else
        PRINT-PATH(G, s, π[v])
        print v
```

The cost of this algorithm is just proportional to the number of vertices in the path, so it is $O\big(d[v]\big)$.