COMPSCI 230: Discrete Mathematics for Computer Science March 25, 2019

Lecture 18

Lecturer: Debmalya Panigrahi *Scribe: Kevin Sun*

1 Overview

In the last lecture, we began our study of directed graphs with the notion of strongly connected components. In this lecture, we will study two fundamental algorithms—depth-first search and breadth-first search—that allow us to systematically search in any graph.

2 Searching in a Graph

The language of graphs allows us to formally model a rich set of real-world problems. For example, as we saw in Lecture 15, the problem of graph coloring allows us to color regions on a map, assign time slots to final exams, and solve many other questions that naturally arise.

Another important concept in graphs is the notion of *searching*. Given a directed graph, there are many natural search-related questions that one might ask: Is the graph acyclic? Does there exist a path from vertex s to vertex t? If so, what is the length of the shortest path from s to t? All of these questions can be addressed by the topics we will discuss in this lecture.

These search questions also arise naturally in the real world. For example, if the vertices of the graph represent courses and the edges represent prerequisite requirements (edge (u, v) exists iff course u is a prerequisite for course v), then the graph must be acyclic if every student is expected to take every course. If the vertices represent locations on a map and the edges represent roads (edge (u, v) represents a road between location u and location v), then the shortest path from one vertex to another gives the fastest way to travel from one location to another.

In this lecture, we will learn about Depth-First Search and Breadth-First Search. These two algorithms solve the problems mentioned above (and many others), and they form the foundations of many algorithms in computer science. Throughout this lecture, we will assume that $G = (V, E)$ is a directed graph with n vertices and m edges. Note that these algorithms can be used for undirected graphs as well: an undirected edge $\{u, v\}$ is equivalent to two directed edges (u, v) and (v, u).

3 Depth-First Search

Depth-first search (DFS) is an algorithm that allows us to systematically explore every vertex of a directed graph. The strategy used by DFS is the following: start at any vertex s and explore one path leading away from s until nothing new can be discovered. Along this path, the search may encounter many "forks" in the road; DFS arbitrarily picks a path that leads somewhere new until it gets stuck. At this point, DFS backtracks until the last choice of path was made, and chooses another path to completely explore. This procedure is repeated until all vertices have been visited.

Note that it is sometimes necessary to restart the procedure starting at a completely new vertex. For example, this is necessary if the first vertex chosen has no outgoing edges, or the graph contains multiple components that are not connected by any edge.

While DFS explores the graph, it also maintains a global time variable τ initialized to $\tau = 1$. The algorithm uses this τ variable to mark each vertex u with the first time it is seen, as well as the time at which DFS backtracks away from u. These times are known as the *pre-* and *post*-values of u, which we denote by $pre(u)$ and $post(u)$, respectively. Every time a vertex is marked, the value of τ increases by one. Thus, we have $pre(u) < post(u)$ for every $u \in V$, and the *pre-* and *post*-values are exactly between 1 and $2n$ (inclusive).

Algorithm 2 provides the formal pseudocode for a complete run of DFS, while Algorithm 1 is a recursive helper function. Note that Algorithm 1 alone is not enough because it is possible that some vertices cannot be reached from s. In this case, as mentioned above, DFS must restart the procedure starting at a new vertex.

Algorithm 1 explore(G, s)

Input: A directed graph $G = (V, E)$; a vertex $s \in V$.
Output: Two positive integers $pre(u), post(u)$ for every vertex u reachable from s.
Note: The variable τ is global, initialized to 1. The default value of visited() is False.

```
1: visited(s) = True
2: pre(s) = τ
3: τ = τ + 1
4: for all (s, v) ∈ E do
5:     if visited(v) == False then
6:         explore(G, v)
7: post(s) = τ
8: τ = τ + 1
```

Algorithm 2 Depth-First-Search(G)

Input: A directed graph $G = (V, E)$.
Output: Two positive integers $pre(u), post(u)$ for every vertex $u \in V$.

```
1: for all v ∈ V do
2:     visited(v) = False
3: for all u ∈ V do
4:     if visited(u) == False then
5:         explore(u)
```

Note that in explore(G, s), the algorithm can visit the neighbors of s in any arbitrary order. Furthermore, Depth-First-Search(G) can visit the vertices $u \in V$ in arbitrary order. Thus, the *pre-* and *post*-values are not necessarily unique.

We now illustrate the first few steps of DFS on the graph shown in Fig. 1, with ties broken by alphabetical order. We begin at vertex a, so $pre(a) = 1$ and there are three options: b, e, d. Since b appears first alphabetically, we explore b which sets $pre(b) = 2$. From b we explore c, so $pre(c) = 3$. The only edge leading away from c is (c, a), but a has already been visited, so our exploration of c terminates and we set $post(c) = 4$. We then backtrack to b, and since (b, c) is the only outgoing edge, we set $post(b) = 5$. We then backtrack to a, where the exploration continues with the edge (a, d). The values in Table 2 give the full results of DFS.

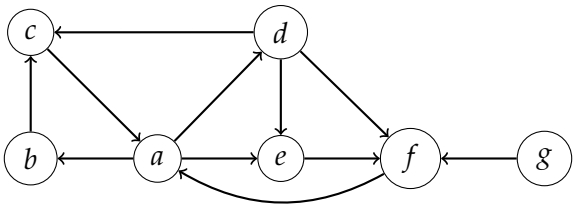

Figure 1: A directed graph with 7 vertices and 9 edges. In this lecture, we will be illustrate DFS and BFS on this graph.

	a	b	c	d	e	f	g
$pre()$	1	2	3	6	7	8	13
$post()$	12	5	4	11	10	9	14

Table 1: The full results of running DFS on the graph in Fig. 1, with all ties in the algorithm broken by alphabetical order.

3.1 DFS Tree

Whenever we run DFS on a graph $G = (V, E)$, we create a structure known as a *DFS tree*. In general, a "DFS tree" may actually be a set of multiple rooted trees, but we still refer to the entire structure as a "tree." The vertex set of a DFS tree is V, and the edges are the ones that pass the condition in line 5 of explore(G, s). In other words, the edges of a DFS tree are the ones that the DFS used to continue down an exploration path. Note that if s is the first vertex DFS explores, and s can reach every vertex in V, then in this case the DFS tree is actually a tree.

Now observe that the edges of a DFS tree indeed form a forest on the vertex set V. This is for the following reason: (u, v) is a tree edge if explore(G, v) is called while explore(G, u) is being executed. Furthermore, each vertex is explored exactly once because as soon as u is explored, we set visited(u) to be True. Thus, a cycle cannot appear in a DFS tree because its existence would imply that a vertex was explored more than once.

Based off a DFS tree, we can partition the set of edges in the graph. Every edge $(u, v) \in E$ can be categorized as exactly one of the following:

1. *Tree edge*: DFS explores v while in the process of exploring u.
2. *Forward edge*: u is a non-parent ancestor of v in the DFS tree.
3. *Back edge*: v is an ancestor of u in the DFS tree.
4. *Cross edge*: u and v have no ancestor-descendent relationship in the DFS tree.

Fig. 2 gives the partition of the graph in Fig. 1. Notice that each back (red) edge forms a cycle when added to the set of tree (green) edges. Furthermore, the forward (blue) edges point from ancestor to descendent, and the cross (dashed) edges connect nodes with no ancestor/descendent relationship.

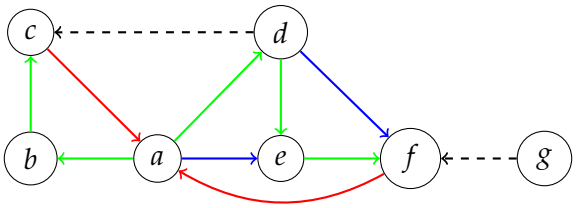

Figure 2: The DFS tree corresponding to a DFS run on Fig. 1, with ties broken in alphabetical order. Tree edges are green, forward edges are blue, back edges are red, and cross edges are dashed.

4 Breadth-First Search

Although DFS can help us determine whether or not a vertex t is reachable from a vertex s, the result may not be the shortest path. For example, in the graph in Fig. 1, DFS discovered a path from a to d that uses two edges: (a, e) and (e, d). However, the shortest path from a to d only uses one edge: (a, d). To find the shortest path between two vertices s and t, we can use an algorithm known as breath-first search (BFS). Before we describe the algorithm, we first formalize the notion of distance in graphs.

Definition 1. *Let $G = (V, E)$ be a directed graph, and let s and t be two vertices of G. The* distance *between s and t, denoted by $dist(s, t)$, is the length of the shortest path from s to t. If t is not reachable from s, then we say the distance from s to t is infinite, i.e., $dist(s, t) = \infty$.*

The strategy used by BFS is the following: start at some vertex s and explore the remaining vertices one "layer" at a time, with each layer being farther from s than the previous. The algorithm fully explores each "layer" before moving onto the next, and hence it is called *breadth*-first search. Algorithm 3 gives the formal pseudocode for BFS.

Algorithm 3 Breadth-First-Search(G, s)

Input: A directed graph $G = (V, E)$; a vertex $s \in V$.
Output: An integer value $d(s, u)$ (possibly ∞) for every vertex $u \in V$.
Note: The default value of $d(\cdot, \cdot)$ is ∞, and Q is initially an empty queue.

```
1: d(s,s) = 0
2: Push(Q,s)
3: while Q is not empty do
4:     u = Pop(Q)
5:     for all (u,v) ∈ E do
6:         if d(s,v) = ∞ then
7:             Push(Q,v)
8:             d(s,v) = d(s,u) + 1
```

Notice that BFS uses a queue to maintain the "layer-by-layer" approach. The vertices closer to s are added to Q before vertices farther from s, and the queue allows us to process all of the closer vertices first before processing the farther ones.

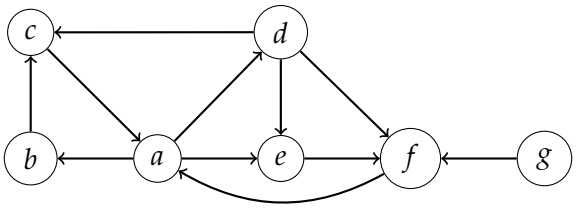

Figure 3: The graph from Fig. 1, drawn here for convenience.

	a	b	c	d	e	f	g
$d(a,\cdot)$	0	1	2	1	1	2	∞

Table 2: The full results of running BFS on the graph in Fig. 3, with all ties in the algorithm broken by alphabetical order.

We now step through the BFS algorithm on the graph in Fig. 3 a few times to better understand the process. We start with $s = a$, so the only element of Q is a. Thus, the first element popped is a, and none of the three neighbors b, d, e have been visited, so they are pushed onto Q in that order and their distances are set to $0 + 1 = 1$. The next element popped is b, and (b, c) is the only outgoing edge, so c is pushed onto Q. Later, after we've processed d and e, we'll pop c and set $d(s, c) = d(s, b) + 1 = 1 + 1 = 2$. This process continues until f is popped since f has no outgoing edges. Note that g is never pushed onto the queue, so $d(a, g)$ remains at ∞. We can verify that g is not reachable from a by visually inspecting the graph.

4.1 BFS Tree

Recall that a run of DFS creates a structure known as a DFS tree, which is technically a forest since it may contain multiple rooted trees. Similarly, a run of BFS creates a tree structure known as a *BFS tree*. If we strictly follow Algorithm 3, then the resulting BFS tree contains exactly one tree, which is rooted at s and contains the vertices reachable from s. However, we can extend the BFS procedure to create a forest by restarting the algorithm at an unvisited vertex as long as such a vertex exists. This idea is captured in Lines 3-5 of Algorithm 2—Algorithm 3 can be modified similarly.

An edge (u, v) is in the BFS tree if, when u was popped from Q, vertex v satisfied $d(v) = \infty$. In other words, if vertex v was unvisited after u was popped from the queue, then u is the parent of v in the BFS tree. Thus, in both a DFS and a BFS "tree," a tree rooted at r contains the vertices reachable from r, and the edges of the tree are the ones used by the algorithm to reach those vertices.

Furthermore, BFS categorizes every edge of the original graph as tree, cross, or back. (The definitions are identical to the ones given for a DFS tree.) Fig. 4 gives the partition of edges from Fig. 3 according to the BFS that starts at a and breaks ties alphabetically. This run of BFS produced 5 tree edges, 4 cross edges, and 2 back edges. Notice that there are no forward edges—in general, BFS never categorizes any edge as a forward edge, and we will now prove this.

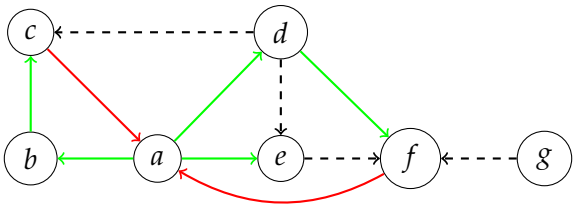

Figure 4: The BFS tree corresponding to a run of BFS on Fig. 3, with ties broken in alphabetical order. Tree edges are green, back edges are red, and cross edges are dashed.

Theorem 1. *A run of BFS will never categorize any edge as a forward edge.*

Proof. For contradiction, assume (u, v) is a forward edge. This means u is a non-parent ancestor of v, so there exists a path from u to v in the BFS tree with at least 2 edges. Let x be the parent of v in the tree. The existence of the path tells us that u entered Q before v, so when u was popped, v must have either been in the queue or would be immediately added. However, (x, v) being a tree edge implies v was added when x was popped, contradicting our conclusion that v was added when u was popped (or earlier). □

Finally, consider a tree with root r of the BFS tree. The edges of this tree form the shortest path from r to every vertex reachable from r. In other words, the final value of $d(s, u)$ in Algorithm 3 is indeed equal to $dist(s, u)$. For example, in Fig. 4, we can see that the green path from a to each vertex (except g, which a cannot reach) is a shortest path. This property follows from the "layer-by-layer" approach taken by BFS, though we will not rigorously prove it here. For this reason, a BFS tree is sometimes known as a *shortest-path tree*. In general, the shortest paths obtained via BFS are the ones that begin at the roots of the trees of the BFS tree.

4.2 Application: Shortest Paths

We've seen that the BFS tree rooted at s gives a shortest path from s to every vertex rooted at s. Now we will generalize this notion to edges with lengths: suppose each edge e has a positive integer *length* value. (We can think of the previous cases as all lengths being 1.) For example, consider the graph in Fig. 5. Running BFS on this graph might lead us to conclude that the shortest path from a to c is (a, c) (which has length 3), when in reality it is (a, b, c) (length 2).

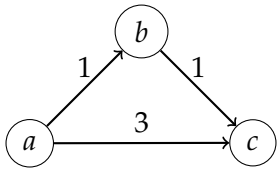

Figure 5: A directed graph with edge lengths. The shortest path from a to c is (a, b, c), and this path has length $1 + 1 = 2$.

However, we can fix this problem by subdividing each edge according to its length, so that the new edges each have length 1. In Fig. 6, we see that by adding two new vertices to the graph in Fig. 5, we can create a new graph whose edge lengths are all 1. Of course, this technique

easily generalizes to any graph with positive integer edge lengths. An edge with length ℓ can be subdivided by placing $\ell - 1$ vertices on the edge, and all new edges are assigned length 1. By running BFS on this new graph starting at vertex s, we can obtain the distances from s to all other vertices in the original graph.

But, this technique has two major shortcomings: it does not work if the edge lengths are anything other than positive integers, and even in that case, the process is very expensive in terms of running time because it introduces a large number of non-existent vertices to model a long edge. These shortcomings can be overcome by more sophisticated algorithms for finding shortest paths that are beyond the scope of our current discussion.

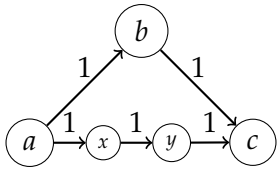

Figure 6: The graph from Fig. 5 with the edge (a, c) subdivided, so that all edges of this new graph have length 1.

Optimal substructure: One property we would like to note about shortest paths is known as the optimal substructure property. This property is quite intuitive and easy to understand, but given its importance in algorithm design, we shall state the property here. Suppose $p = (s, u_1, u2, \ldots, u_k, t)$ is a shortest path from vertex s to vertex t. Now consider two vertices x, y on this path, so p is of the form $(s, \ldots, x, \ldots, y, \ldots, t)$. The optimal substructure property states that the path from x to y that follows p is the shortest path from x to y. The reasoning is quite intuitive: if there were a shorter path from x to y, then we could use that path to create a shorter path from s to t.

5 Summary

In this lecture, we studied depth-first search and breadth-first search, two fundamental algorithms in discrete mathematics. We looked at the corresponding trees they create, and observed some basic properties regarding these two procedures.