**CS 344: Design and Analysis of Computer Algorithms** **Rutgers: Fall 2019**

# Lecture 15

October 28th, 2019

*Instructor: Sepehr Assadi*

## 1 Graph Search

Recall that the goal (or at least one of the goals) is to find all vertices reachable from a given vertex $s$ in a graph $G(V, E)$ (directed or undirected).

### 1.1 Depth-First-Search Algorithm

In the previous lecture, we went over the DFS algorithm defined as follows.

**Algorithm:** Initialize an array $mark[1:n]$ with 'FALSE'. Run the following recursive algorithm on $s$, i.e., return `DFS`$(s)$:

```
DFS($v$) :

1. If $mark[v]$ = 'TRUE' terminate.
2. Otherwise, set $mark[v]$ ='TRUE' and for every neighbor $u \in N(v)$ (this is the list of neighbors of $v$
   that we have direct access to in the adjacency list representation):
     • Run DFS($u$).
```

At the end, we return all vertices $v$ with $mark[v]$ = 'TRUE' as the answer, i.e., as vertices that are in the same connected component of $s$. Moreover if $G$ is *directed*, the marked vertices will be vertices that are *reachable* from $s$ (but may not necessarily reach $s$ also).

### 1.2 Breadth-First-Search Algorithm

The second algorithm for graph search that we visit is *breadth-first-search* or *BFS* for short. Unlike DFS that was going deeper and deeper inside the graph, BFS instead explores the graph "layer by layer". Let us formalize this in the following.

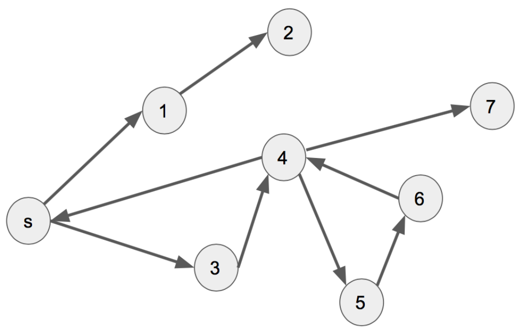
We start from the vertex $s$ and place all neighbors of $s$ inside a *queue* data structure $Q$: A queue is simply a "First-In First-Out (FIFO)" data structure that we can *append* to one end of it and *extract* from the *other* end (the opposite of this data structure is a *stack* which is "Last-In First-Out (LIFO)": we can append to one end of it and extract from the *same* end). Then we pick the next vertex from $Q$ and append all neighbors of this vertex to the queue (so they appear after the previous vertices). Similar to DFS, here we also need to mark the vertices so as to not revisit the same vertex many times. We now give the formal algorithm.

```
Algorithm BFS($s$) :

1. Initialize an array $mark[1:n]$ with 'FALSE'.
2. Create a queue data structure $Q$ and insert $s$ to $Q$.
```

3. While $Q$ is not empty:
    (a) Let $v$ be the first vertex of $Q$ and dequeue (remove) this vertex from $Q$.
    (b) If $mark[v]$ = 'TRUE'; continue from the beginning of the while-loop.
    (c) Otherwise, let $mark[v]$ = 'TRUE' and for $u \in N(v)$: insert $u$ to the end of $Q$.
4. At the end, we return all vertices $v$ with $mark[v]$ = 'TRUE' as the answer.

**Example:** Let us start with the following example.



- Iteration (1) (of while-loop): $Q = \{s\}$ only – we dequeue $s$, mark it, and insert 1 and 3 in $Q$;
- Iteration (2): $Q = \{1, 3\}$ – we dequeue 1, mark it, and insert 2 in $Q$;
- Iteration (3): $Q = \{3, 2\}$ – we dequeue 3, mark it, and insert 4 in $Q$;
- Iteration (4): $Q = \{2, 4\}$ – we dequeue 2 and mark it;
- Iteration (5): $Q = \{4\}$ – we dequeue 4, mark it, and insert $s$, 5 and 7 in $Q$;
- Iteration (6): $Q = \{s, 5, 7\}$ – we dequeue $s$, observe it is marked, and ignore it;
- Iteration (7): $Q = \{5, 7\}$ – we dequeue 5, mark it, and insert 6 in $Q$;
- Iteration (8): $Q = \{7, 6\}$ – we dequeue 7 and mark it;
- Iteration (9): $Q = \{6\}$ – we dequeue 6, mark it, and insert 4 in $Q$;
- Iteration (10): $Q = \{4\}$ – we dequeue 4, observe it is marked, and ignore it;
- Iteration (11): $Q = \emptyset$ – we exit the while-loop.

*Note:* Similar to the case of DFS also, here also we could first check for every $u \in N(v)$, if $mark[u]$ = 'TRUE' already or not and if it is the case, skip inserting it to $Q$ right there; while this is certainly a useful optimization in practice, it does not change the asymptotic worst case runtime of the algorithm and hence we can safely ignore it for the purpose of this course.

**Proof of Correctness:** The proof is almost identical to that of DFS and for a good reason: by changing the data structure $Q$ from a queue to a *stack*, the resulting algorithm would be identical to DFS! Because of this, we do not repeat the proof here (and anyway we are going to prove a stronger statement for BFS).

**Runtime Analysis:** For each vertex $v$, we run the entire body of the while-loop at most once (which takes $O(|N(v)|)$ time) and after that it is marked and we only dequeue the vertex in $O(1)$ time (this time can be *charged* to the time spent by vertex $u$ that inserted $v$ for the second (or third, etc.) time to the queue $Q$ instead of accounting for it again by vertex $v$). As such, the total runtime of the algorithm is at most $O(n) + c \cdot \sum_{v \in V} |N(v)| = O(n + m)$.

We shall note that all the extensions of DFS algorithm such as finding cycles or spanning trees etc. can also be done exactly the same for BFS. In the next section, we see that BFS can actually do even more!

## 2 BFS for Shortest Path in Unweighted Graphs

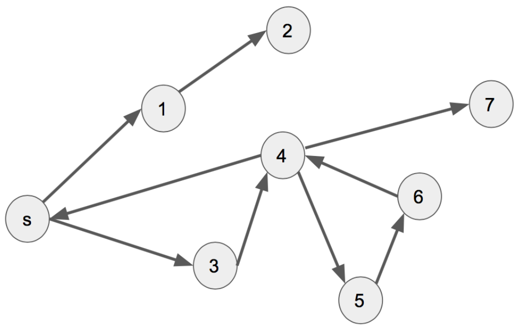
An important property of BFS is that in *unweighted* graphs[1], it can find *shortest paths* from vertex $s$ (starting point of BFS) to all other vertices in the graph. Roughly speaking, this is true because of the "layer by layer" exploration of this algorithm. In order to do this, we only need to store some intermediate values in the algorithm as well.

**Algorithm** `BFS-SHORTEST-PATH`($s$):

```
1. Initialize an array $d[1:n]$ with $+\infty$ (we use this array to do the work of the $mark$ also).
2. Create a queue data structure $Q$, insert $s$ to $Q$, and let $d[s] = 0$.
3. While $Q$ is not empty:
   (a) Let $v$ be the first vertex of $Q$ and dequeue (remove) this vertex from $Q$.
   (b) For $u \in N(v)$:
       i. If $d[u] \neq +\infty$ go to the next neighbor.
       ii. Let $d[u] = d[v] + 1$ and insert $u$ to the end of $Q$.
4. At the end, for every vertex $v \in V$, outputs $d[v]$ as the distance of $s$ to that vertex (where $+\infty$ means
   that there is no path from $s$ to $v$).
```

We run this algorithm on the same example graph we used for BFS (repeated here again).

**Example:** Let us start with the following example.



In the following, we show the content of $Q$ and $d$ at the *beginning* of each iteration.

- Iteration (1) (of while-loop): $Q = \{s\}$ and $d = [0, +\infty, +\infty, +\infty, +\infty, +\infty, +\infty, +\infty]$;

[1]So far we only considered unweighted graphs but a weighted graph is defined exactly the same way by adding weights to the edges.

- Iteration (2): $Q = \{1, 3\}$ and $d = [0, 1, +\infty, 1, +\infty, +\infty, +\infty, +\infty]$;
- Iteration (3): $Q = \{3, 2\}$ and $d = [0, 1, 2, 1, +\infty, +\infty, +\infty, +\infty]$;
- Iteration (4): $Q = \{2, 4\}$ and $d = [0, 1, 2, 1, 2, +\infty, +\infty, +\infty]$;
- Iteration (5): $Q = \{4\}$ and $d = [0, 1, 2, 1, 2, +\infty, +\infty, +\infty]$;
- Iteration (6): $Q = \{5, 7\}$ and $d = [0, 1, 2, 1, 2, 3, +\infty, 3]$;
- Iteration (7): $Q = \{7, 6\}$ and $d = [0, 1, 2, 1, 2, 3, 4, 3]$;
- Iteration (8): $Q = \{6\}$ and $d = [0, 1, 2, 1, 2, 3, 4, 3]$;
- Iteration (9): $Q = \emptyset$ and $d = [0, 1, 2, 1, 2, 3, 4, 3]$ – we exit the while-loop here.

**Proof of Correctness:** We now prove that $d[v]$ at the end of the algorithm is equal to the actual distance from $s$ to $v$ for every $v \in V$. The proof is by induction on the length of the shortest-paths from $s$ to all other vertices. Let us formalize this as follows. For every $0 \leq \ell \leq n-1$, define:

$$D(\ell) := \{v \in V \mid d[v] = \ell\};$$
$$T(\ell) := \{v \in V \mid \text{distance of } s \text{ to } v \text{ in } G \text{ is } \ell\}.$$

Our induction hypothesis is that for every $0 \leq \ell \leq n-1$, $D(\ell) = T(\ell)$. Note that by proving this we will be done since for any vertex $v$, $v \in D(d[v]) = T(d[v])$ where the former is the set of vertices with distance $d[v]$ from $s$; hence $v$ indeed has distance $d[v]$ from $s$.

We prove this by induction of $\ell$. The base case of $\ell = 0$ clearly holds because the only $d[s] = 0$ in the algorithm and so $D(0) = \{s\}$. At the same time the only vertex at distance 0 from $s$ is $s$ and so $T(0) = \{s\}$, hence $D(0) = T(0)$.

For the induction step, suppose this is true for *some* integer $\ell$ and we prove it for $\ell + 1$. By induction hypothesis, $D(\ell) = T(\ell)$ is exactly the set of vertices which are at distance $\ell$ from $s$. To prove that $D(\ell+1) = T(\ell+1)$, we need to prove that if $u \in D(\ell+1)$ then $u$ also belongs to $T(\ell+1)$, and if $w \in T(\ell+1)$ then $w \in D(\ell+1)$ also. Each part is proven below:

(a) $u \in D(\ell+1) \implies u \in T(\ell+1)$: Since $u \in D(\ell+1)$, there should exists a vertex $v \in D(\ell)$ such that $u \in N(v)$ (so that $d[u] = d[v] + 1$ in the algorithm). By induction hypothesis, since $D(\ell) = T(\ell)$, we know that $v$ is at distance $\ell$ from $s$. So distance of $u$ to $s$ is *at most* $\ell + 1$. At the same time, we know that $u \notin T(0) \cup \ldots \cup T(\ell)$ because $u \notin D(0) \cup \ldots \cup D(\ell)$ and by induction hypothesis, the union of former sets is equal to the union of the latter sets and $u$ clearly cannot be in both $D(\ell+1)$ and some $D(\ell')$ for $\ell' \leq \ell$. This means that distance of $u$ from $s$ is *exactly* $\ell + 1$. So $u \in T(\ell+1)$.

(b) $w \in T(\ell+1) \implies w \in D(\ell+1)$: Since $w \in T(\ell+1)$, there should exists a vertex $v \in T(\ell)$ such that $w \in N(v)$ ($v$ is the vertex 'before' $w$ in the shortest path from $s$ to $w$). By induction hypothesis, since $T(\ell) = D(\ell)$, we know that when visiting $v$ if $d[w] = +\infty$, we would have set $d[w] = d[v] + 1 = \ell + 1$; but $d[w]$ is equal to $+\infty$ at this point because in the algorithm, all vertices in $D(\ell)$ are visited before any vertex in $D(\ell+1)$, and so by the time we visit $v$, we have not visited $w$ yet and so $d[w] = +\infty$ at this point. This means that $d[w] = \ell + 1$ after visiting $v$ and so $w \in D(\ell+1)$.

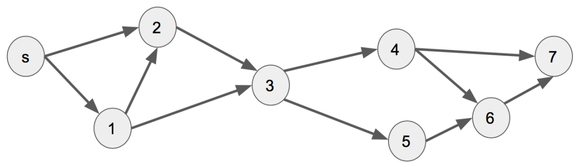
This concludes the proof of the induction step, and hence the correctness of the algorithm follows.

**Runtime Analysis:** Runtime of this algorithm is asymptotically exactly the same as the original BFS and is hence $O(n + m)$.

## 3 Topological Sorting

We now consider another fundamental problem on *directed* graphs: *topological sorting* of a directed acyclic graph (DAG). Roughly speaking, a topological ordering of a DAG is an ordering of vertices $u_1, \ldots, u_n$ such that any edge can only be directed from a vertex with *lower* index to a vertex with *higher* index. More formally, a topological ordering is any ordering $\prec$ such that for any edge $(u, v)$ we have $u \prec v$. We emphasize that topological ordering of a DAG may *not* be unique.

**Example:** In the following DAG:



- The ordering $s \prec 1 \prec 2 \prec 3 \prec 4 \prec 5 \prec 6 \prec 7$ is a topological ordering of this DAG.
- The ordering $s \prec 1 \prec 2 \prec 3 \prec 5 \prec 4 \prec 6 \prec 7$ is a topological ordering of this DAG (we replaced 4 and 5 which is okay because there is no edge from one to the other).
- The ordering $s \prec 2 \prec 1 \prec 3 \prec 4 \prec 5 \prec 6 \prec 7$ however is *not* a topological ordering of this DAG since there is an edge $(1, 2)$ and so we cannot have $2 \prec 1$.

One can see that a directed graph has a topological ordering if and only if it is a DAG (it is a good exercise to prove this on your own). Based on this, we have the topological sorting problem.

**Problem 1** (**Topological Sorting**)**.** Given a directed graph $G(V, E)$ either output a topological ordering of $G$ if $G$ is a DAG or output that $G$ is *not* a DAG.

There are different algorithms for topological sorting. We propose one of the simple ones here. The algorithm is simply as follows: we pick a vertex with in-degree equal to 0 and place it in the beginning of the ordering; remove all its outgoing edges from the graph, and repeat. Also if at any point we can no longer find a vertex of in-degree 0, we return that the graph is not a DAG.

In the next lecture, we go over how to implement this algorithm efficiently and prove its correctness.