Lecture 8 Graphs
Scribes: Romil Verma, Juliana Cook (2015), Virginia Williams, **Date:** Oct 23, 2018
and Seth Hildick-Smith (2016), G. Valiant (2017), M. Wootters (2017)
Adapted From Virginia Williams' lecture notes

1 Graphs

A **graph** is a set of **vertices** and **edges** connecting those vertices. Formally, we define a graph G as $G = (V, E)$ where $E \subseteq V \times V$. For ease of analysis, the variables n and m typically stand for the number of vertices and edges, respectively. Graphs can come in two flavors, **directed** or **undirected**. If a graph is undirected, it must satisfy the property that $(i, j) \in E$ iff $(j, i) \in E$ (i.e., all edges are bidirectional). In undirected graphs, $m \leq \frac{n(n-1)}{2}$. In directed graphs, $m \leq n(n-1)$. Thus, $m = O(n^2)$ and $\log m = O(\log n)$. A connected graph is a graph in which for any two nodes u and v there exists a path from u to v. For an undirected connected graph $m \geq n - 1$. A *sparse graph* is a graph with few edges (for example, $\Theta(n)$ edges) while a *dense* graph is a graph with many edges (for example, $m = \Theta(n^2)$).

1.1 Representation

A common issue is the topic of how to represent a graph's edges in memory. There are two standard methods for this task.

An **adjacency matrix** uses an arbitrary ordering of the vertices from 1 to $|V|$. The matrix consists of an $n \times n$ binary matrix such that the $(i, j)^{th}$ element is 1 if (i, j) is an edge in the graph, 0 otherwise.

An **adjacency list** consists of an array A of $|V|$ lists, such that $A[u]$ contains a linked list of vertices v such that $(u, v) \in E$ (the neighbors of u). In the case of a directed graph, it's also helpful to distinguish between outgoing and ingoing edges by storing two different lists at $A[u]$: a list of v such that $(u, v) \in E$ (the out-neighbors of u) as well as a list of v such that $(v, u) \in E$ (the in-neighbors of u).

What are the tradeoffs between these two methods? To help our analysis, let $deg(v)$ denote the **degree** of v, or the number of vertices connected to v. In a directed graph, we can distinguish between out-degree and in-degree, which respectively count the number of outgoing and incoming edges.

- The adjacency matrix can check if (i, j) is an edge in G in constant time, whereas the adjacency list representation must iterate through up to $deg(i)$ list entries.
- The adjacency matrix takes $\Theta(n^2)$ space, whereas the adjacency list takes $\Theta(m + n)$ space.
- The adjacency matrix takes $\Theta(n)$ operations to enumerate the neighbors of a vertex v since it must iterate across an entire row of the matrix. The adjacency list takes $\deg(v)$ time.

What's a good rule of thumb for picking the implementation? One useful property is the sparsity of the graph's edges. If the graph is **sparse**, and the number of edges is considerably less than the max ($m \ll n^2$), then the adjacency list is a good idea. If the graph is **dense** and the number of edges is nearly n^2, then the matrix representation makes sense because it speeds up lookups without too much space overhead. Of course, some applications will have lots of space to spare, making the matrix feasible no matter the structure of the graphs. Other applications may prefer adjacency lists even for dense graphs. Choosing the appropriate structure is a balancing act of requirements and priorities.

2 Depth First Search (DFS)

Given a starting vertex, it's desirable to find all vertices reachable from the start. There are many algorithms to do this, the simplest of which is depth-first search. As the name implies, DFS enumerates the deepest

paths, only backtracking when it hits a dead end or an already-explored section of the graph. DFS by itself is fairly simple, so we introduce some augmentations to the basic algorithm.

- To prevent loops, DFS keeps track of a "color" attribute for each vertex. Unvisited vertices are white by default. Vertices that have been visited but still may be backtracked to are colored gray. Vertices which are completely processed are colored black. The algorithm can then prevent loops by skipping non-white vertices.[1]
- Instead of just marking visited vertices, the algorithm also keeps track of the tree generated by the depth-first traversal. It does so by marking the "parent" of each visited vertex, aka the vertex that DFS visited immediately prior to visiting the child.
- The augmented DFS also marks two auto-incrementing timestamps d and f to indicate when a node was first discovered and finished.

The algorithm takes as input a start vertex s and a starting timestamp t, and returns the timestamp at which the algorithm finishes. Let $N(s)$ denote the neighbors of s; for a directed graph, let $N_{out}(s)$ denote the out-neighbors of s.

Algorithm 1: init(G)

```
foreach v ∈ G do
    color(v) ← white
    d(v), f(v) ← ∞
    p(v) ← nil
```

Algorithm 2: DFS(s, t) $s \in V$. t = time, s is white

```
color(s) ← gray
\\ d(s) is the discovery time of s
d(s) ← t
t++;
\\ In the loop below, replace N(s) with N_out(s) for directed G
foreach v ∈ N(s) do
    if color(v) = white then
        p(v) ← s
        \\ update t to be the finish time of DFS starting at v:
        t ← DFS(v, t)
        t++
\\ finish time
f(s) ← t
\\ s is finished
color(s) ← black
return f(s)
```

There are multiple ways we can search using DFS. One way is to search from some source node s, which will give us a set of black nodes reachable from s and white nodes unreachable from s.

Algorithm 3: DFS(s) \\ DFS from a source node s

```
init(G)
DFS(S, 1)
```

Another way to use DFS is to search over the entire graph, choosing some white node and finding everything we can reach from that node, and repeating until we have no white nodes remaining. In an undirected graph this will give us all of the connected components.

[1] In the slides, white, gray, and black are replaced with light green, orange, and dark green, respectively.

Algorithm 4: DFS(G) \\ DFS on an entire graph G

```
init(G);
t ← 1
foreach v ∈ G do
    if color(v) = white then
        t ← DFS(v, t)
        t++
```

2.1 Runtime of DFS

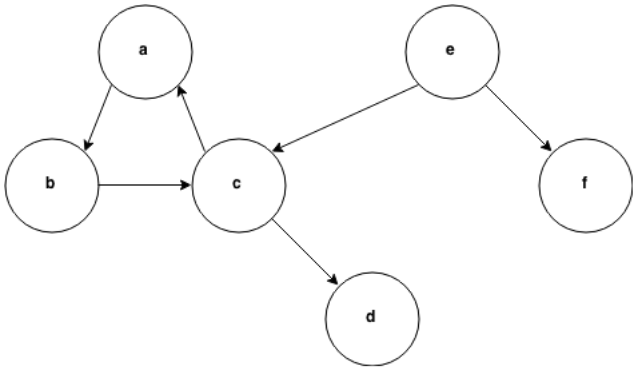
We will now look at the runtime for the standard DFS algorithm (Algorithm 2).

Everything above the loop runs in $O(1)$ time per node visit. Excluding the recursive call, everything inside of the for loop takes $O(1)$ time every time an edge is scanned. Everything after the for loop also runs in $O(1)$ time per node visit.

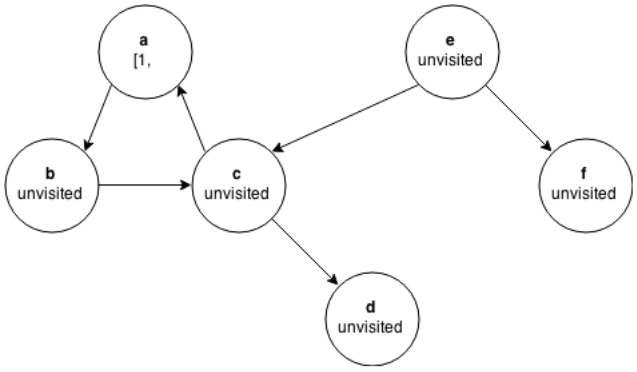
We can express the runtime of DFS as $O(\#$ of node visits + # of edge scans). Assume we have a graph with n nodes and m edges. We know that the # of node visits is $\leq$ n, since we only visit white nodes and whenever we visit a node we change its color from white to gray and never change it back to white again. We also know that an edge (u, v) is scanned only when u or v is visited. Since every node is visited at most once, we know that an edge (u, v) is scanned at most twice (or only once for directed graphs). Thus, # of edges scanned is $O(m)$, and the overall runtime of DFS is $O(m + n)$.

2.2 DFS Example

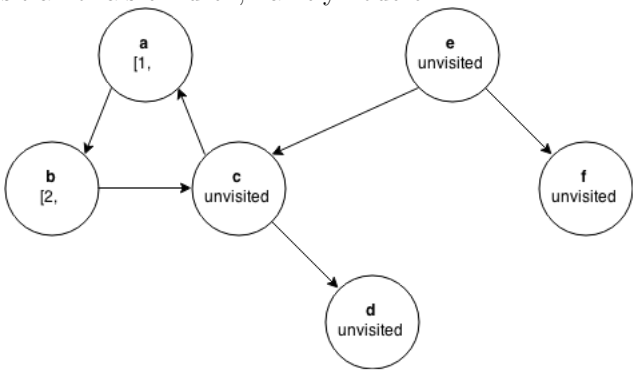
We will now try running DFS on the example graph below.



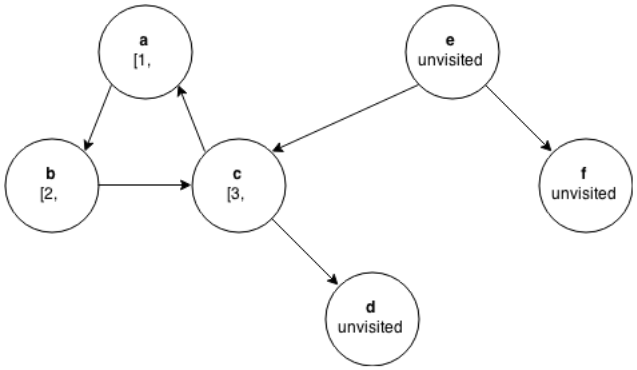
We mark all of the nodes as unvisited and start at a white node, in our case node a.



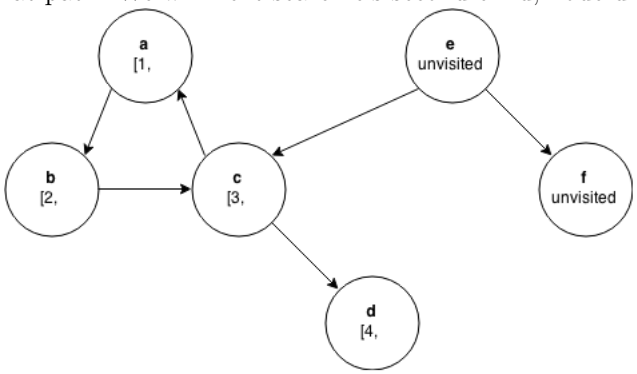
From node a we will visit all of a's children, namely node b.



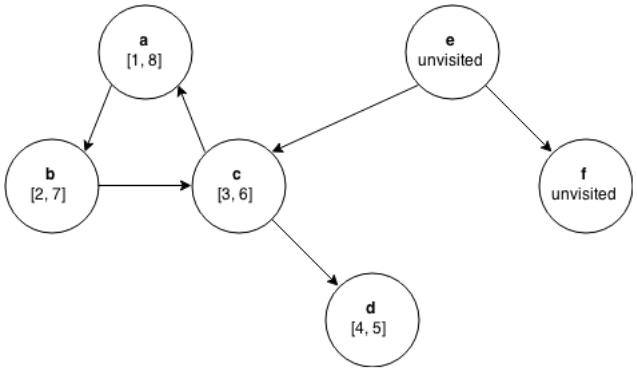
We now visit b's child, node c.



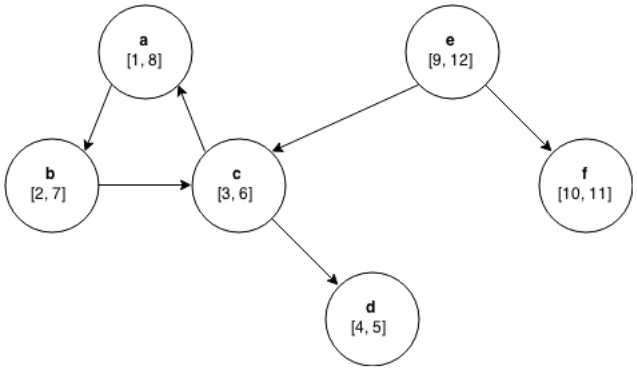
Node c has two children that we must visit. When we try to visit node a we find that node a has already been visited (and would be colored gray, as we are in the process of searching a's children), so we do not continue searching down that path. We will next search c's second child, node d.



Since node d has no children, we return back to its parent node, c, and continue to go back up the path we took, marking nodes with a finish time when we have searched all of their children.



Once we reach our first source node a we find that we have searched all of its children, so we look in the graph to see if there are any unvisited nodes remaining. For our example, we start with a new source node e and run DFS to completion.

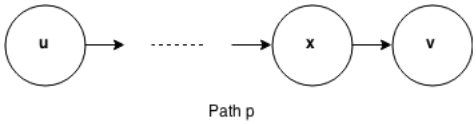


3 Breadth First Search (BFS)

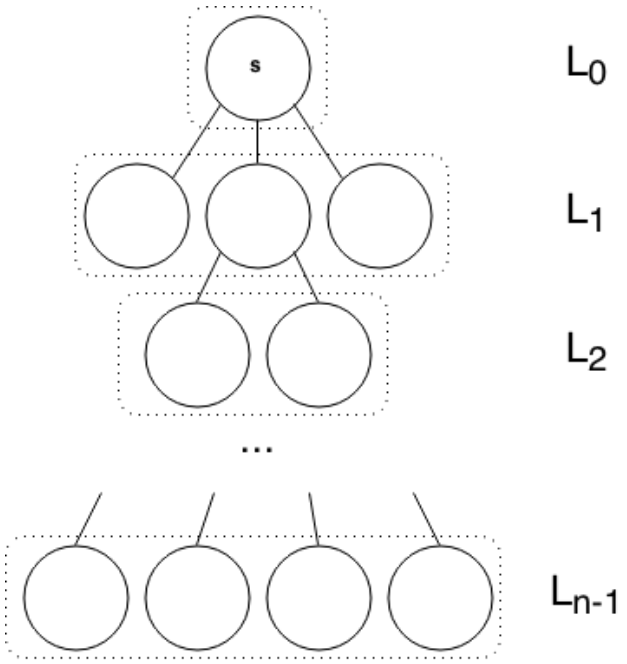
In depth first search, we search "deeper" in the graph whenever possible, exploring edges out of the most recently discovered node that still has unexplored edges leaving it. Breadth first search (BFS) instead expands the frontier between discovered and undiscovered nodes uniformly across the breadth of the frontier, discovering all nodes at a distance k from the source node before nodes at distance $k + 1$.

BFS(s) computes for every node $v \in G$ the distance from s to v in G. $d(u, v)$ is the length of the shortest path from u to v.

A simple property of unweighted graphs is as follows: let P be a shortest $u \to v$ path and let x be the node before v on P. Then $d(u, v) = d(u, x) + 1$.



BFS(s) computes sets L_i, the set of nodes at distance i from s, as seen in the diagram below.



Algorithm 5: BFS(s)

```
Set vis[v] ← false for all v;
Set all L_i for i = 1 to n − 1 to ∅;
L_0 = s;
vis[s] ← true;
for i = 0 to n − 1 do
    if L_i = ∅ then
        exit
    while L_i ≠ ∅ do
        u ← L_i.pop();
        \\ In the loop below, replace N with N_out for a directed graph.;
        foreach x ∈ N(u) do
            if vis[u] is false then
                vis[u] ← true;
                L_{i+1}.insert(x);
                p(x)← u;
```

3.1 Runtime Analysis

We will now look at the runtime for our BFS algorithm (Algorithm 5) for a graph with n nodes and m edges. All of the initialization above the first for loop runs in $O(n)$ time. Visiting each node within the while loop takes $O(1)$ time per node visited. Everything inside the inner foreach loop takes $O(1)$ time per edge scanned, which we can simplify to a runtime of $O(m)$ time overall for the entire inner for loop.

Overall, we see that our runtime is $O(\#\text{ nodes visited} + \#\text{ edges scanned}) = O(m + n)$.

3.2 Correctness

We will now show that BFS correctly computes the shortest path between the source node and all other nodes in the graph. Recall that L_i is the set of nodes that BFS calculates to be distance i from the source node.

Claim 1. *For all i, $L_i = \{x | d(s,x) = i\}$.*

Proof of Claim 1. We will prove this by (strong) induction on i. Base case ($i = 0$): $L_0 = s$.

Suppose that $L_j = \{x | d(s,x) = j\}$ for every $j \leq i$ (induction hypothesis for i).

We will show two things: (1) if y was added to L_{i+1}, then $d(s,y) = i+1$, and (2) if $d(s,y) = i+1$, then y is added to L_{i+1}. After proving (1) and (2) we can conclude that $L_{i+1} = \{y | d(s,y) = i+1\}$ and complete the induction.

Let's prove (1). First, if y is added to L_{i+1}, it was added by traversing an edge (x,y) where $x \in L_i$, so that there is a path from s to y taking the shortest path from s to x followed by the edge (x,y), and so $d(s,y) \leq d(s,x) + 1$. Since $x \in L_i$, by the induction hypothesis, $d(s,x) = i$, so that $d(s,y) \leq i+1$. However, since $y \notin L_j$ for any $j \leq i$, by the induction hypothesis, $d(s,y) > i$, and so $d(s,y) = i+1$.

Let's prove (2). If $d(s,y) = i+1$, then by the inductive hypothesis $y \notin L_j$ for $j \leq i$. Let x be the node before y on the $s \to y$ shortest path P. As $d(s,y) = i+1$ and the portion of P from s to x is a shortest path and has length exactly i. Thus, by the induction hypothesis, $x \in L_i$. Thus, when x was scanned, edge (x,y) was scanned as well. If y had not been visited when (x,y) was scanned, then y will be added to L_{i+1}. Hence assume that y was visited before (x,y) was scanned. However, since $y \notin L_j$ for any $j \leq i$, y must have been visited by scanning another edge out of a node from L_i, and hence again y is added to L_{i+1}. □

3.3 BFS versus DFS

If you simplify BFS and DFS to the basics, ignoring all timestamps and levels that we would usually create, BFS and DFS have a very similar structure. Breadth first search explores the nodes closest and then moves outwards, so we can use a queue (first in first out data structure) to put new nodes at the end of the list and pull the oldest/nearest nodes from the top of the list. Depth first search goes as far down a path as it can before coming back to explore other options, so we can use a stack (last in first out data structure) which pushes new nodes on the top and also pulls the newest nodes from the top. See the pseudocode below for more detail.

Algorithm 6: DFS(s) \\ s is the source node

```
T ← stack
push s onto T
while T is not empty do
    u ← pop from top of T
    push all unvisited neighbors of u on top of stack T
```

Algorithm 7: BFS(s) \\ s is the source node

```
T ← queue
push s onto T
while T is not empty do
    u ← pop from front of T
    push all unvisited neighbors of u on back of queue T
```