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 $E \subseteq V \times V$. Unless otherwise stated, we use m and n to denote $|E|$ and $|V|$, 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 $(j, i) \in E$ (aka all edges are bidirectional). Additionally, graphs can be unweighted or weighted. Weighted graphs are equipped with a weight function $w : E \to \mathbb{Z}$ (sometimes the weights are real numbers but in this class they will be integers). Occasionally, vertices can also have weights, but for the purposes of this class, we only consider edge-weighted graphs.

1.1 Representation

How do we represent graphs in memory? There are two standard methods for this task.

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

An **adjacency list** consists of an array Adj of $|V|$ lists, such that $Adj[u]$ 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 $Adj[u]$: 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). Often one only stores the out-neighbors.

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.

- Using an adjacency matrix one can check if (i, j) is an edge in G in constant time, whereas if one uses the adjacency list representation, one may have to iterate through up to $\deg(i)$ list entries to check whether j is in $Adj[i]$. As $\deg(i)$ can be $\Omega(n)$, this can be quite expensive.
- The adjacency matrix takes $\Theta(n^2)$ space, whereas the adjacency list takes $\Theta(m + n)$ space.
- Using an adjacency matrix, it takes $\Theta(n)$ operations to enumerate the neighbors of a vertex v since one must iterate across an entire row of the matrix. Using an adjacency list, the same thing takes only $O(\deg(v))$ 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 adjacency matrix representation 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 The problems.

For today's lecture we will assume that the input graph is unweighted, and that we are using the adjacency list representation. For directed graphs, the adjacency lists store the out-neighbors.

We will give algorithms for two problems today.

The first is, checking whether a given undirected graph is connected, and the second is computing the distances from a fixed source vertex to all other vertices in the (potentially directed) graph.

To define these problems, let us remind ourselves about paths and distances in graphs. A path from a vertex s to a vertex t in a graph $G = (V, E)$, is a sequence of vertices $s = v_0, v_1, \ldots, v_k = t$ so that for all $i \in \{0, \ldots, t-1\}$, $(v_i, v_{i+1}) \in E$. Two vertices s and t are said to be connected if there is a path between them. An undirected graph is connected if all pairs of its vertices are connected. (There are similar notions for directed graphs which we will discuss later on.)

The *length* of a path in an unweighted graph is the number of its edges. (In a weighted graph it is the sum of its edge weights.) A shortest path between s and t is a path from s to t with a minimum length among all s-t paths in G. The distance $d(s, t)$ from s to t is the length of a shortest s-t path in G.

The *Connectivity* problem is: given an undirected unweighted graph G, to determine whether G is connected.

The *Connected components* problem is: given an undirected unweighted graph G, return its connected components, i.e. a partitioning of its vertices where each partition induces a connected graph and there are no edges between vertices in different partitions.

The *Single Source Reachability (SSR)* problem is: given an unweighted graph G and a vertex s, determine all vertices t in G for which there is an s-t path (i.e. t is reachable from s).

The *Single Source Shortest Paths (SSSP)* problem is: given an unweighted graph G and a vertex s, determine for all vertices t in G, the distance $d(s, t)$. To obtain actual shortest paths, one also might want to return for every $t \in V$, with $d(s, t) < \infty$, a vertex $\pi(t)$, which is the "predecessor" of t on an s-t shortest path. That is, $d(s, \pi(t)) + 1 = d(s, t)$ and $(\pi(t), t) \in E$.

It is clearly more efficient to store and output $\pi(t)$ than the actual path. Every path from s to t can have length $\Omega(n)$, so listing a shortest path from s to every t could take $\Omega(n^2)$ output. This output results not only in large space usage but also in at least quadratic time to run the algorithm. Instead, we just output the predecessor nodes $\pi(t)$ using only $O(n)$ extra time and space. It's not hard to see that in order to reconstruct a shortest path from s to u, it is sufficient to know only the node $\pi(u)$ right before each node on the shortest path. (Start from u, look up $\pi(u)$, then look up $\pi(\pi(u))$, continuing until you've found s.) This compressed output only has size $O(n)$, which will allow our SSSP algorithms to run faster than $\emptyset(n^2)$.

One detail to keep in mind is that in order for our path reconstruction to work, our algorithms must be *consistent*; that is, if p is the chosen shortest path from s to t, then $\forall x$ on this shortest path, the chosen shortest path from s to x is completely on path p. Intuitively, this is not a very restrictive assumption, it just means we need to break ties between equivalent shortest paths consistently. This tie-breaking scheme should ensure that the shortest paths we pick are simple, so that there is a well-defined predecessor node for each node. This requirement is more crucial than the consistency described above.

3 Breadth First Search

Here we will present an algorithm called Breadth First Search (BFS) that is very efficient and can solve the Single Source Shortest Paths (SSSP) problem in unweighted graphs. Notice that if an algorithm solves SSSP from a source vertex s, it also solves Single Source Reachability (SSR) problem from s since the vertices reachable from s are exactly those that have finite distance from s. Moreover, any algorithm that solves SSR also solves the Connectivity problem in undirected graphs: an undirected graph is connected if and only if a source vertex can reach all other vertices in the graph. Thus, we can pick an arbitrary vertex s, solve SSR from it, and determine if all other vertices are reachable. Using SSR we can also solve the Connected Components problem: Repeat: pick an unvisited vertex s, run SSR from it, visiting all vertices reachable from s, and label these as one connected component; at the end return all connected components found. The running time for Connected Components and Connectivity is asymptotically the same as that for SSR. In summary, since the BFS algorithm will solve SSSP in linear time, as a byproduct, it will solve Connectivity, Connected Components and SSR in unweighted graphs in linear time as well.

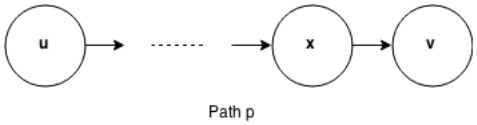
Let us assume now that we are given an unweighted directed graph $G = (V, E)$ in adjacency list format, a source $s \in V$ and we want to compute for every $t \in V$, the distance $d(s, t)$.

The basic idea of BFS is to discover vertices in layers according to their distance from s. Let L_j be the set of all vertices at distance j from s. Suppose we have computed every L_j for $j \leq k$. Then, the idea of BFS is to scan the edges incident to vertices in L_k (the "frontier"), check whether they are in some L_j for $j \leq k$ already, and if not, add them to L_{k+1}. This search finishes layer L_{k+1} and then starts scanning the edges out of vertices in L_{k+1} and so on.

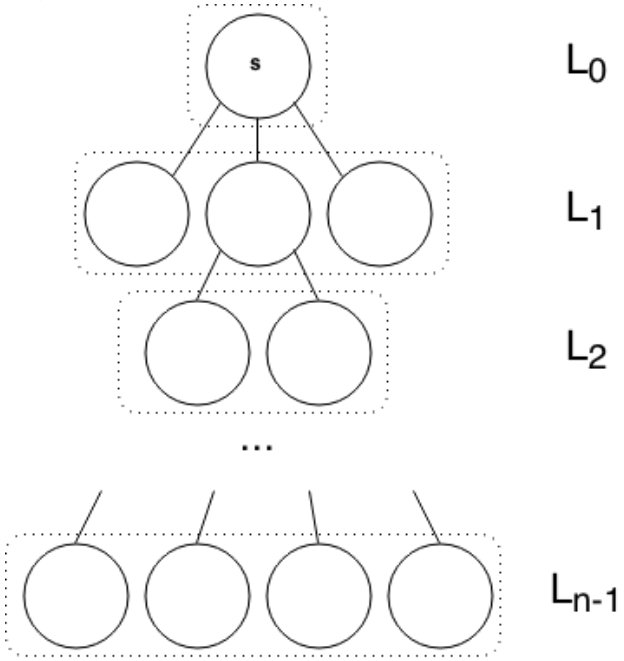
We now give some more details. (Let G be a fixed graph, BFS is run on G although we will not carry G around in the argument of BFS for brevity.)

BFS(s) computes for every node $v \in G$ the distance from s to v in G. As G is unweighted, $d(u, v)$ is the number of edges on 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.



BFS(s) keeps an array vis of bits, so that $vis[v]$ is true only if v has been visited by scanning an edge (or if $v = s$). Initially only s is visited and $L_0 = \{s\}$ since the only node at distance 0 from s is s. Then the scan proceeds as described before, however, when a vertex v is visited, its $vis[v]$ bit is set to true.

The sets L_i can be implemented as linked lists or stacks, or any data structure that allows constant time insert and pop operations. $N(v)$ is the set of out-neighbors of a vertex v; these are stored in an adjacency list.

The pseudocode is below.

3.1 Runtime Analysis

We will now look at the runtime for our BFS algorithm (Algorithm 1) 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

Algorithm 1: 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();
        foreach x ∈ N(u) do
            if vis[x] is false then
                vis[x] ← true;
                L_{i+1}.insert(x);
                p(x)← u;
```

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 $L_j = \{x | d(s,x) = j\} \forall j \leq i$ (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}.

□

4 A framework for graph search (optional)

BFS is particularly tailored to solve SSSP. If one merely wanted to solve SSR (i.e. to find the vertices reachable from a source s), BFS is only one of many ways to do this. We will give a general framework

here that defines a family of linear time algorithms that all solve SSR. Later on we will see another search algorithm called Depth First Search that falls into the framework.

4.1 Two ways to search

We will define two related families of algorithms, Search(s, G) and Search*(s, G) both parameterized by a data structure D. D is any data structure which can support insertions of an element ($D.insert(x)$) and popping of an element ($D.pop$). The data structure can decide which element to pop, and pop removes the element from D and returns it.

Let $t(n)$ be the running time of D when n is the maximum number of elements ever stored in D. Some examples of simple data structures D include stacks and queues, and these have $t(n) = O(1)$.

Search Here is pseudocode for Search; visited is a Boolean array of length $n = |V|$, and $G = (V, E)$ is represented via the adjacency lists Adj. In the beginning, only s is visited and all other vertices are not. At the end of the algorithm visited will only be true for vertices reachable from s and for all such vertices.

Search(s, G):
$visited[s] \leftarrow true$
For all $v \neq s$: $visited[v] \leftarrow false$
$D.insert(s)$
While $D.nonempty$:
    $v \leftarrow D.pop$
    For all $x \in Adj[v]$:
        If $visited[x] = false$:
            $visited[x] \leftarrow true$
            $D.insert(x)$.
Return $visited$

Search starts from the source and keeps visiting the unvisited neighbors of visited vertices. We will prove that regardless of how D decides which elements to pop, the algorithm runs efficiently and computes the vertices reachable from the source. For hilarity, we will call this the Super Mega Awesome Ultra Theorem of graph search.

Theorem 4.1 (Super Mega Awesome Ultra Theorem for graph search)**.** *If D implements insert and pop in $t(n)$ time each where no more than n elements are ever inserted, the Search(s, G) on an unweighted $G = (V, E)$ with $|V| = n, |E| = m$ runs in $O(m + n \cdot t(n))$ time, and at the end of the algorithm, for every $x \in V$, $visited[x] = true$ if and only if s can reach x.*

This theorem is particularly powerful for D for which $t(n) = O(1)$ such as stacks and queues, as it shows that SSR can be solved in linear time, and hence also Connectivity and Connected Components. We will later see that using different Ds can yield really cool other properties of the algorithm that allows us to solve even more problems in linear time!

Proof. First notice that no vertex can be inserted into D more than once since vertices are only inserted when visited is false, and when they are inserted, visited is set to true. Because a vertex is inserted at most once, it can only be popped at most once as well. Hence the number of times operations on D are called is at most n, and their contribution to the running time is $O(nt(n))$.

The rest of the running time of the algorithm is proportional to the number of times "For all $x \in Adj[v]$" is executed. This operation is just going through the edges incident to a vertex v that was just popped from D. Each edge (v, x) in G will be considered at most twice such a step - once when v is popped and once when x is popped (and we know that each vertex is popped at most once). Thus, the rest of the running time is proportional to the total number of edges, $O(m)$.

The final running time is $O(m + nt(n))$.

Now, let us show that at the end of the algorithm, for every $x \in V$, $visited[x] = true$ if and only if x is reachable from s.

First, let us use induction on the execution of the algorithm to show that if $visited[x] = true$, then there is a path from s to x. As a base case, when $visited[s]$ is set to true, s is reachable from s. Suppose now that up to step $i-1$ in the execution, if $visited[x] = true$ for some x, then x is reachable from s. Now consider step i. If this step is not of the form $visited[x] \leftarrow true$, then nothing changes. Otherwise, if it is of the form $visited[x] \leftarrow true$, then this step happened because x was a neighbor of v, and v was just popped from D. However, for v to be popped from D it must have $visited[v] = true$, and hence by induction there is a path from s to v. Adding the edge (v, x) to the end of this path shows that x is also reachable from s.

Now let us use induction on the distance from s to show that if x is reachable from s, then at the end of the algorithm, $visited[x] = true$. Suppose that for every vertex x at distance $\leq d$ from s, $visited[x] = true$ at the end of the algorithm. As a base case, for $d = 0$, there is only one vertex at distance 0 from s, namely s itself, and the algorithm does set $visited[s]$ to true. Now consider some vertex x at distance $d+1$ from s. Let y be the vertex right before x on a shortest s-x path. Then $d(s, y) = d$ and $(y, x) \in E$. By the induction hypothesis, at some point, the algorithm sets $visited[y] = true$. Thus, y is added to D, and hence at some point y is also popped from D (the number of insertions to D is at most n and hence every vertex that is in D is eventually popped). When y is popped, all its neighbors are considered, and in particular, the algorithm checks if $visited[x] = true$. If $visited[x] = true$, then we are done with the proof. Otherwise, the algorithm sets $visited[x]$ to true, and we are also done. □

BFS in the Search framework. BFS is the implementation of Search(s, G) in which D is a queue. We will interpret BFS slightly differently from the layer implementation from earlier, but we will see that it visits the vertices in the same order and computes their distances from the source.

We add a small amount of bookkeeping to the algorithm, which does not increase the running time. In particular, in addition to visited, we keep an array d of distances, so that at the end of the algorithm $d[v]$ will be $d(s, v)$. We also keep an array π of predecessors, so that if $d(s, v) < \infty$, then $\pi(v)$ is the vertex right before v on some s-v shortest path. (The only exception is that the predecessor of s is considered to be s itself.) The pseudocode is as follows (the new steps are in bold):

BFS(s, G):
$visited[s] \leftarrow true$
$\mathbf{d[s]} \leftarrow \mathbf{0}, \pi[\mathbf{s}] \leftarrow \mathbf{s}$
For all $v \neq s$: $visited[v] \leftarrow false, \mathbf{d[v]} \leftarrow \infty, \pi[\mathbf{v}] \leftarrow \mathbf{NIL}$
$D.insert(s)$
While $D.nonempty$:
    $v \leftarrow D.pop$
    For all $x \in Adj[v]$:
        If $visited[x] = false$:
            $visited[x] \leftarrow true, \mathbf{d[x]} \leftarrow \mathbf{d[v]} + \mathbf{1}, \pi[\mathbf{x}] \leftarrow \mathbf{v}$
            $D.insert(x)$.
Return $\mathbf{d}, \pi$

Search* We will slightly perturb Search, so that the visited value of a vertex is set not when it is first accessed but when it is popped from D. In other words, when we scan the edges out of a vertex, we will insert all of the found neighbors into D. Let's call this modified algorithm, Search*.

Search*(s, G):
For all v: $visited[v] \leftarrow false$
$D.insert(s)$
While $D.nonempty$:
    $v \leftarrow D.pop$

If $visited[v] = false$:
    $visited[v] \leftarrow true$
    For all $x \in Adj[v]$:
        $D.insert(x)$.
Return *visited*

Since we still scan the edges out of every visited vertex, this version of the algorithm will still correctly find exactly the vertices reachable from s. The correctness proof by induction is analogous to that for Search.

For the running time in m-edge, n-vertex graphs, we can provide a very similar argument as for Search in the last lecture: (1) Every edge (v, x) is scanned at most twice, once from v and once from x (and at most once, from v, in directed graphs) as it can only be scanned from v (or x) if v (or x) is not visited, and right before the scanning begins, visited[v] (or visited[x]) is set to true. (2) The number of insertions into D is exactly the number of edges scanned, so it can be at most $2m$ in an m-edge graph (for directed graphs it's at most m). Thus, the time due to operations on D is $O(mt(n))$ and the rest of the running time of the algorithm is $O(n)$

We get a theorem analogous theorem as before:

Theorem 4.2 (Super Mega Awesome Ultra Theorem* for graph search)**.** *If D implements insert and pop in $t(n)$ time each where no more than n elements are ever inserted, the Search*(s, G) on an unweighted directed or undirected graph $G = (V, E)$ with $|V| = n, |E| = m$ runs in $O(n + m \cdot t(n))$ time, and at the end of the algorithm, for every $x \in V$, visited$[x] = true$ if and only if s can reach x.*

In particular, if D is a queue or a stack, the running time is linear, $O(m + n)$.