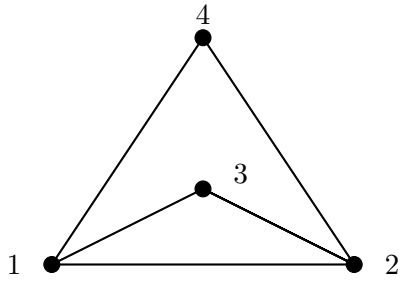
CS 341: ALGORITHMS (S18) — LECTURE 11
GRAPH EXPLORATION: BREADTH-FIRST SEARCH

ERIC BLAIS

1. Graphs

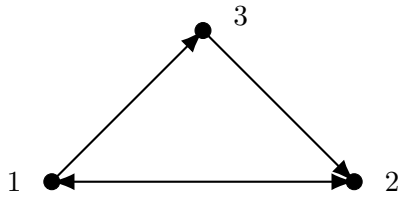
A *graph* $G = (V, E)$ is a set V of *vertices* (sometimes called *nodes*) and a set $E \subseteq V \times V$ of *edges* connecting pairs of vertices. Whenever we work on graphs, we will let $n = |V|$ denote the number of vertices in the graph and $m = |E|$ denote the number of edges of the graph.

An *unordered graph* is a graph whose edges are undirected.



The vertex set of this graph is $V = \{1, 2, 3, 4\}$ and the edge set of this graph is $E = \big\{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}\big\}$.

An *ordered graph* has edges that are directed from one end vertex to the other.



The vertex set of this graph is $V = \{1, 2, 3\}$ and its edge set is $E = \big\{(1, 2), (1, 3), (2, 1), (3, 2)\big\}$.

1.1. Definitions.

There are a number of basic definitions that we will use.

Adjacent: Two vertices u and v are *adjacent* (or *neighbours*) if they are connected by an edge.

Incidence: A vertex v is *incident* to an edge e if it is one of its endpoints.

Degree: The *degree* of a vertex v is the number of edges to which it is incident.

In/Out-degree: In directed graphs, the *in-degree* of a vertex v is the number of edges that end at v and the *out-degree* of v is the number of edges that start at v.

Path: A *path* in a graph is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that for each $1 \leq i < k$, $(v_i, v_{i+1}) \in E$.

Simple path: A *simple path* is a path in which no vertices appear more than once.

Cycle: A *cycle* is a path of length at least 3 that starts and ends at the same vertex and contains no other vertex repetitions.

Connectedness: An undirected graph is *connected* if every two vertices in V is connected by a path.

Tree: A *tree* is a connected graph that does not contain any cycles.

Connected component: A *connected component* of a graph is a maximal connected subgraph.

1.2. **Computing with graphs.** There are two natural representations of graphs that are useful when we solve computational problems on them: the adjacency matrix representation, and the adjacency list representation.

The *adjacency matrix* representation of a graph $G = (V, E)$ is an $n \times n$ matrix $A \in \{0,1\}^{n \times n}$ where

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

for every pair i, j of vertices. When the graph G is unordered, $A[i,j] = A[j,i]$ is always satisfied. The adjacency matrix representation of a graph always has space complexity $O(n^2)$ (even when the graph is *sparse*, i.e. when $m \ll n^2$). But with this representation we can answer the question "is i adjacent to j" in constant time for any pair of vertices.

The *adjacency list* representation of a graph $G = (V, E)$ is a set of n linked lists, one for each vertex in the graph, in which the list associated with v stores all the neighbours of v in G. The space complexity of this representation is $O(n + m)$ (which is much better than $O(n^2)$ when $m \ll n^2$)) in the Word RAM model where we set the length of the words to be $O(\log n)$ so that vertex labels can be stored in a single word.

With the adjacency list representation, the query "is i adjacent to j" requires traversing a linked list and therefore has time complexity $O(n)$. But identifying all the neighbours of a vertex v can be done in time $O(\deg(v))$—we will use this fact in our graph exploration algorithms.

2. Breadth-first search

There is a basic building block used in many different graph algorithms: the exploration problem. In this problem, from a starting vertex $s \in V$, we wish to visit all the vertices that are reachable from s in the graph G.

Definition 11.1. An instance of the Graph Exploration problem is a graph $G = (V, E)$ and a vertex $s \in V$. A valid solution to this instance is a list of all the vertices in the connected component of G that contains s.

There are two fundamental algorithms for solving the Graph Exploration problem: Breadth-first search (BFS), and Depth-first search (DFS).

The *breadth-first search* algorithm can be thought of as the cautious explorer: first, it finds all the vertices that can be reached from s by following only a single edge, then it finds all the vertices that can be reached by following 2 edges from s, then the ones reachable by following 3 edges from s, etc.

In the implementation of breadth-first search, each vertex in the graph is assigned one of three statuses: `undiscovered`, `discovered`, and `explored`. Initially, all vertices except the start vertex s are marked `undiscovered`, and s itself is marked `discovered`. Then at every step of the search, we choose one `discovered` vertex v and traverse the list of list of

its neighbours, marking all of the ones that are `undiscovered` as `discovered` and ignoring the other ones. Once we have finished visiting the list of neighbours of v, we change its status as `explored`. We then continue the process with another `discovered` vertex until no vertices with that status are left.

Breadth-first search, as described above, can be implemented efficiently with the adjacency list representation of vertices by using a queue to store the `discovered` vertices.

Algorithm 1: BFS$(G = (V, E), s)$

```
for each v ∈ V \ {s} do
    status[v] ← undiscovered;
status[s] ← discovered;
Q ← {s};
L ← ∅;
while Q ≠ ∅ do
    v ← Dequeue(Q);
    for each w ∈ Adj(G, v) do
        if status[w] = undiscovered then
            status[w] ← discovered;
            Enqueue(Q, w);
    Add(L, v);
    status[v] ← explored;
return L;
```

It's worth seeing the BFS algorithm in action by going over an example.

[See CLRS §22 for a complete example.]

Let's wait a little bit before proving the correctness of the BFS algorithm: it is easiest to do so when we consider the distance variant of the algorithm in the next section. For now, let's jump ahead to examine the time complexity of the algorithm.

Theorem 11.2. *The time complexity of the BFS algorithm is $O(n + m)$.*

Proof. By only adding vertices in the queue when they are `undiscovered` and by changing their status to `discovered` when we do so (and by never resetting the state of any vertex to `undiscovered` after the initialization), we guarantee that each vertex is added to the queue at most once, so that the total number of queue operations is $O(n)$ and since those operations take constant time, the time complexity of those operations is $O(n)$.

Furthermore, since the adjacency list of each vertex is visited at most once (when that vertex is dequeued) and a constant amount of time is spent for each element of those lists, the total amount of time spent in the inner for loop is $O(m)$.

The initialization also has time complexity $O(n)$, so the total time complexity of the BFS algorithm is $O(n) + O(n) + O(m) = O(n + m)$. □

3. Applications of BFS

Breadth-first search is a very simple algorithm, but it is amazingly useful: there are many fundamental problems on graphs that can be solved with only very minor modifications to the BFS algorithm. Let's examine a few.

3.1. **Single-source shortest path.** A common task when dealing with graphs is finding the shortest path between vertices. One of the common variants of this problem is the *single-source shortest path*, where we want to find the shortest path between a special source vertex s and all the other vertices in the graph.

Definition 11.3 (Single-source shortest path (SSSP))**.** Given a graph $G = (V, E)$ and a vertex $s \in V$, find the shortest path between s and every vertex in $V \setminus \{s\}$.

(When s and v are in different conected components in G, by convention we say that the shortest path between s and v has length ∞.)

The BFS algorithm already essentially solves the SSSP problem, since the first time that we visit a vertex v, it is always by visiting the last edge of a shortest path to v. All that we need to do to complete the algorithm is add a few extra lines to the BFS algorithm.

Algorithm 2: BFS-SSSP$(G = (V, E), s)$

```
for each v ∈ V \ {s} do
    status[v] ← undiscovered;
    dist[v] ← ∞;
status[s] ← discovered;
dist[s] ← 0;
Q ← {s};
while Q ≠ ∅ do
    v ← DEQUEUE(Q);
    for each w ∈ Adj(G, v) do
        if status[w] = undiscovered then
            status[w] ← discovered;
            dist[w] ← dist[v] + 1;
            ENQUEUE(Q, w);
    status[v] ← explored;
return dist;
```

We are now ready to prove the correctness of both the original BFS algorithm and the BFS-SSSP algorithm. The key statement in both proofs of correctness is the following assertion.

Lemma 11.4. *Fix any graph $G = (V, E)$ with diameter D. For each $d = 0, 1, 2, \ldots, D + 1$, there is a moment in the BFS-SSSP algorithm's execution on G at which point*

1. *All nodes at distance at most d from s have their distance correctly set;*
2. *All other nodes have their distance set to ∞; and*
3. *The set of nodes that are in the queue (and, equivalently, the set of nodes at state* `discovered`*) is the vertices at distance exactly d from s.*

Proof. We prove the statement by induction on d. In the base case, when $d = 0$ the three conditions hold right before we enter the while loop.

For the induction step, assume that the statement is true for distance $d - 1$. Let's say that there are exactly k nodes at distance $d-1$ from s that are in the queue at that moment. Consider what happens after we run the while loop k more times. Then all those nodes

have been processed, and all the neighbours of those k nodes that had not been previously discovered are added to the queue and have their distance set to d. But these are exactly the nodes that are at distance exactly d from s—they all have distance at most d from s since they are reachable by following one more edge from a vertex at distance $d-1$ from s, and they are at distance at least d since otherwise, by the induction hypothesis, they would have been discovered and had their distance set earlier. The remaining nodes at distance at least $d+1$ from s still have their distance set to ∞, so the three conditions hold for d as well. □

The proof of correctness of the BFS-SSSP and BFS algorithms follow almost immediately from the Lemma.

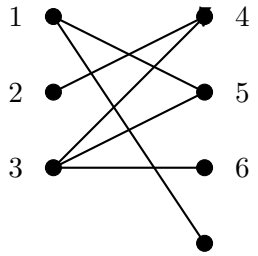
Theorem 11.5. *The BFS-SSSP algorithm solves the Single-Source Shortest Path problem.*

Proof. The lemma guarantees that when the algorithm exits the while loop, all nodes that are reachable from s have the correct distance set and all other ones still have distance ∞. □

Theorem 11.6. *The BFS algorithm solves the Graph Exploration problem.*

Proof. In the original BFS algorithm, instead of keeping track of the distance of each vertex to s, we added it to the list of vertices reachable from s that is output at the end of the algorithm. Therefore, the list of vertices output by BFS corresponds exactly to the set of vertices with finite distance reported by BFS-SSSP, which is just the set of vertices in the same connected component as s. □

3.2. **Testing bipartiteness.** A graph $G = (V, E)$ is *bipartite* if there is a partition of V into two parts V_L and V_R such that every edge in E connects a vertex in V_L to one in V_R. For example, the following graph is bipartite,



but if we add the edge $(2, 3)$ to the above graph, it is no longer bipartite.

Definition 11.7 (Testing bipartiteness problem)**.** Given a connected graph G, determine if it is bipartite or not.

The BFS algorithm can be modified easily to solve this problem. We start choosing any arbitrary vertex s and put it in the Left part V_L. (This is again an arbitrary choice that does not matter: we can always flip the labels of the parts V_L and V_R of a bipartite graph without changing the graph itself.) Then every time we visit a new vertex, it must be in the other part of the graph than the neighbour that led to this vertex. So whenever we discover a vertex, we also know which of the two parts V_L or V_R to assign it to, if the graph is indeed bipartite.

It remains to verify that the graph is actually bipartite: this is done simply by checking that whenever we encounter an edge between two vertices that have already been discovered,

those two vertices are in different parts. This will be true for all edges if and only if the graph is bipartite. The resulting algorithm is as follows.

Algorithm 3: BFS-Bipartite($G = (V, E)$)

```
s ← any vertex in V;
for each v ∈ V \ {s} do
    status[v] ← undiscovered;
    dist[v] ← ∞;
status[s] ← discovered;
InLeftPart[s] ← True;
Q ← {s};
while Q ≠ ∅ do
    v ← DEQUEUE(Q);
    for each w ∈ Adj(G, v) do
        if status[w] = undiscovered then
            status[w] ← discovered;
            InLeftPart[w] ←!InLeftPart[v];
            ENQUEUE(Q, w);
        else if InLeftPart[v] = InLeftPart[w] then
            return False;
    status[v] ← explored;
return True;
```

The proof of correctness of the algorithm is left as an exercise. Another good exercise is to see how to generalize the algorithm so that it can test whether *any* graph (connected or not) is bipartite. (Hint: a graph is bipartite if and only if all its connected components are bipartite.)

3.3. **Spanning tree.** A *spanning tree* of a connected graph $G = (V, E)$ is a tree $T = (V, E')$ where $E' \subseteq E$ and T is also a connected graph. Spanning trees are useful for a number of reasons; for instance, they can be used to store a sparse representation of G (with only $n - 1$ edges) that still enables us to find a path in G between every two vertices $v, w \in V$.

Definition 11.8 (Spanning tree problem)**.** Given a connected graph $G = (V, E)$, output a spanning tree of G.

We can represent a rooted tree with an array of n nodes, where the entry for node v is its parent in the tree and the entry for the root simply has $\emptyset$ as its parent. The following simple modification of the BFS algorithm computes a spanning tree of connected graphs using this representation.

Algorithm 4: BFS-SpanningTree($G = (V, E), s$)

```
for each v ∈ V \ {s} do
    status[v] ← undiscovered;
    dist[v] ← ∞;
status[s] ← discovered;
parent[s] ← ∅;
Q ← {s};
while Q ≠ ∅ do
    v ← DEQUEUE(Q);
    for each w ∈ Adj(G, v) do
        if status[w] = undiscovered then
            status[w] ← discovered;
            parent[w] ← v;
            ENQUEUE(Q, w);
        else if InLeftPart[v] = InLeftPart[w] then
            return False;
    status[v] ← explored;
return parent;
```

We again leave the proof of correctness of this algorithm as an exercise. We will also revisit spanning trees in later lectures.