Connectivity

Suppose we are given a graph $G = (V, E)$ represented as an adjacency list and two particular nodes $s, t \in V$. A natural question to ask is: Is there a path from $s$ to $t$ in $G$? This is the problem of $s$-$t$ connectivity. As a side note, $s$ stands for “source” and $t$ stands for “target.” One can also think of the connectivity problem as a traversal problem: from $s$, which nodes are reachable? A traversal of the graph beginning at $s$ would answer such a question.

We will introduce two natural algorithms for solving this problem: breadth-first search (BFS) and depth-first search (DFS). Later in the course, you will see several other uses for these algorithms beyond graph connectivity.

Before introducing specific algorithms, we can consider a more general, high-level description of a process to find the nodes reachable from a source $s$:

- $R$ will consist of nodes to which $s$ has a path
- $R = \{s\}$
- while there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  - Add $v$ to $R$

In fact in an undirected graph, this “algorithm” finds the vertices in the connected component containing $s$. The reason “algorithm” is in quotes is because the above process is under-specified: in the while loop, how do we decide which edge to consider next? The BFS and DFS algorithms below give two different ways to do this.

1 Breadth-First Search (BFS)

Perhaps the simplest algorithm for graph traversal is breadth-first search (BFS) in which we explore outward from $s$ in all possible directions, visiting nodes one “layer” at a time. Thus, we start at $s$ and visit all nodes that are joined by an edge to $s$—this is the first layer of the search. We then visit all additional nodes that are joined by an edge to any node in the first layer—this is the second layer. We continue this way until no new nodes are encountered.

We can define “layers” more formally:

- $L_0 = \{s\}$
- $L_{k+1}$ is the set of all nodes that do not belong in $\bigcup_{i=0}^{k} L_i$ and that have an edge to some node in $L_k$. 
One can also think about BFS not in terms of layers, but in terms of a tree $T$ rooted at $s$. More specifically, for each node $v \neq s$, consider the moment when $v$ is first discovered by the BFS algorithm. This happens when some node $u$ in layer $L_k$ is begin examined, and we find that it has an edge to the previously unseen node $v$. At this moment, we add the edge $(u,v)$ to the tree $T$—$u$ becomes the parent of $v$. We call the tree $T$ produced this way a breadth-first search tree.

The algorithm is described below. We will use a queue to determine which nodes to visit next, and we will use an array $\text{discovered}$ to keep track of which nodes have been visited. For simplicity, we assume the vertices are integers numbered from 1 to $|V|$. The first implementation keeps track of the BFS tree via the $\text{parent}$ array, while the second implementation keeps track of both the BFS tree and the levels $L_i$. Depending on the scenario, one may be more useful than the other.

---

**Breadth-First Search**

*Input:* A graph $G = (V, E)$ implemented as an adjacency list and a source vertex $s$.

*Output:* A BFS traversal of $G$

```
BFS(G,s)
for each v ∈ V do
    discovered[v] = FALSE
    parent[v] = NIL

Let Q be an empty Queue
Q.enqueue(s)
discovered[s] = TRUE

while Q is not empty do
    v = Q.dequeue()
    for each u ∈ Adj[v] do
        if discovered[u] = FALSE then
            discovered[u] = TRUE
            Q.enqueue(u)
            parent[u] = v
```

---

```
BFS(G,s)
for each v ∈ V do
    discovered[v] = FALSE

discovered[s] = TRUE
L[0] = {s}
i = 0
while L[i] is not empty do
    Let L[i+1] be a new list
    for each v ∈ L[i] do
```
for each $u \in \text{Adj}[v]$ do
  if $\text{discovered}[u] = \text{FALSE}$ then
    $\text{discovered}[u] = \text{TRUE}$
    $\text{parent}[u] = v$
    $L[i+1].\text{append}(u)$
  $i = i + 1$

Note, the order in which we visit each node in a particular level doesn’t matter. There could be multiple BFS trees.

To see how the algorithm works, suppose we start with $s$ as node 1 in the above graph. With $L_0 = \{1\}$, the first layer of the search would be $L_1 = \{2, 3\}$, the second layer would be $L_2 = \{4, 5, 7, 8\}$, and the third layer would be just $L_3 = \{6\}$. At this point, there are no further nodes that could be added. Note in particular that nodes 9 through 13 are never reached by the search.

For the BFS started at $s = 1$ on the tree in Figure 1, the BFS tree is shown below. The solid edges are the edges in $T$, the dashed edges are edges in $G$ that aren’t in $T$. The first few steps of the execution that produces this tree can be described in words:

- Starting from 1, $L_1 = \{2, 3\}$.
- Layer $L_2$ is then grown by considering the nodes in $L_1$ in any order. If we examine node 2 first, then we discover nodes 4 and 5, so 2 becomes their parent. When we examine node 3, we discover 7 and 8 (5 has already been discovered), so 3 becomes their parent.
- We then consider the nodes in $L_2$ (WLOG we consider them in ascending order). The only new node discovered is node 6, which is discovered through node 5 and so becomes the child of node 5.
- No new nodes are discovered when node 6 is examined. The BFS traversal thus terminates.
BFS Properties

We now prove some properties related to BFS and BFS trees.

If we define the distance between two nodes as the minimum number of edges on a path joining them, we have the following property:

**Proposition 1.** For each \( k \geq 1 \), the layer \( L_k \) produced by BFS consists of all nodes at distance exactly \( k \) from \( s \). Moreover, there is path from \( s \) to \( t \) in \( G \) if and only if \( t \) appears in some layer (i.e. \( t \) is visited by the BFS traversal).

**Proof.** Seeking contradiction, suppose WLOG that \( i < j - 1 \), or equivalently \( i + 1 < j \). Consider the point in the BFS algorithm when the edges incident to \( x \) were begin examined. Since \( x \) belongs to layer \( L_i \), the only nodes discovered from \( x \) belong to layers \( L_{i+1} \) and earlier. Hence if \( y \) is a neighbor of \( x \), then it should have been discovered by this point at the latest and hence should belong to \( L_{i+1} \) or earlier. Since \( i + 1 < j \), we have a contradiction.

Runtime of BFS

**Proposition 3.** Running BFS on a graph \( G = (V, E) \) given as an adjacency list takes \( O(m+n) \) time, where \( n = |V| \) and \( m = |E| \). That is, BFS is a linear time algorithm.

**Proof.** It is easy to see that the algorithm is \( O(n^2) \): it visits each node once, and examines all its neighbors, performing \( O(1) \) work for each. Since there are \( n \) nodes, each having \( O(n) \) neighbors, the runtime is \( O(n^2) \).
To get a tighter bound, note that for each node, we only examine its neighbors, so for each node \( v \in V \) we do \( O(\deg(v)) \) work. Since
\[
\sum_{v \in V} \deg(v) = 2m
\]
by the handshake lemma, the runtime is \( O(n + 2m) = O(n + m) \).

\[\square\]

## 2 Depth-First Search (DFS)

While BFS explores nodes level-by-level, depth-first search searches “deeper” in the graph whenever possible. In particular, DFS explores edges out of the most recently discovered vertex \( v \) that still has unexplored edges leaving it. Once all of \( v \)’s edges have been explored, the search “back tracks” to explore edges leaving the vertex from which \( v \) was discovered. This process continues until we have discovered all vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then DFS selects one of them as a new source and it repeats the search from that source. The algorithm repeats until it has discovered every vertex.

Note that unlike BFS—which creates a tree—DFS creates a forest, called the depth-first search forest. Moreover, the DFS algorithm will keep track of several things:

- At every point in the algorithm, a vertex will have one of three colors: white (undiscovered), grey (discovered and currently examining), or black (finished).

- Each vertex \( v \) will be timestamped with its discovery time \( v.d \) and finish time \( v.f \). These are related to the color of \( v \): \( v.d \) is the time when \( v \) is first colored grey, and \( v.f \) is the time when \( v \) is colored black.

- At the end of the algorithm, each edge will be classified either as a tree edge, back edge, forward edge, or cross edge.

We will discuss each of these in depth. First, let us present the algorithm. DFS can be implemented either recursively or using a stack. We give a recursive version here (the implementation using a stack is very similar to the BFS implementation using a queue, except with “queue” replaced with stack, “enqueue” replaced with “push,” and “dequeue” replaced with pop throughout). As before, the DFS forest is represented by a parent array. Start/finish times and colors are also maintained using arrays:

### Depth-First Search

*Input:* A graph \( G = (V, E) \) implemented as an adjacency list.

*Output:* A DFS traversal of \( G \)

\[
\text{DFS}(G) \\
\quad \text{for each } v \in V \text{ do} \\
\quad \quad \text{color}[v] = \text{WHITE} \\
\quad \quad \text{time} = 0 \\
\quad \text{for each } v \in V \text{ do} \\
\quad \quad \text{if color}[v] = \text{WHITE then}
\]
DFS works for both directed and undirected graphs. Consider the graph below:

A DFS traversal of this graph is given in Figure 2. In that traversal, we visit nodes in numerical order (this is arbitrary: DFS can visit the nodes in any order). Pay careful attention to the start/finish times of each node and how these relate to each node’s color. Also, be sure to note which edges are included in the DFS forest (the red edges) and which aren’t. Note that there are two trees in the resulting forest: one consists of the vertices \{1, 2, 4, 5\} and the other consists of the lone vertex \{3\}.

**Runtime of DFS**

The procedure \texttt{DFS-VISIT} is called once per vertex. At each vertex \(v\), we iterate through \(v\)'s neighbors (potentially calling \texttt{DFS-VISIT} on the neighbor), and thus this iteration takes \(\sum_{v \in V} \deg(v) = O(m)\) time, not including the calls to \texttt{DFS-VISIT}. The runtime of DFS is therefore \(O(n + m)\).
Figure 2: A DFS traversal. The start/finish times are given for each node, and the red edges are those that are included in the DFS forest.

**DFS Properties and Extensions**

Here we explore some of the reasons for keeping track of node colors and start/finish times. We will see that they reveal valuable information about the structure of a graph.

First note that each DFS traversal naturally gives rise to a DFS forest: \( u \) is a parent of \( v \) if and only if \( \text{DFS-VISIT}(G,v) \) was called during the search of \( u \)’s adjacency list. Additionally, \( v \) is a descendant of \( u \) if and only if \( v \) is discovered during the time in which \( u \) is gray. We record that in the following lemma, as
it will be important later on:

**Lemma 1.** In the DFS forest, \( v \) is a descendant of \( u \) if and only if \( v \) is discovered during the time in which \( u \) is gray.

Another important property of DFS is that discovery and finish times have *parenthesis structure*. If we represent the discovery of vertex \( u \) with a left parenthesis “(u” and represent its finishing by a right parenthesis “u)” then the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested. For example, in the DFS traversal from figure 2, the parenthesis structure looks like:

\[
(1 \ (2 \ (4 \ 4) \ (5 \ 5) \ 2) \ 1) \ (3 \ 3)
\]

where the bottom line represents time.

The following theorem characterizes this parenthesis structure and gives its relationship to the structure of the DFS forest:

**Theorem 1. (Parenthesis Theorem)** In any DFS of a (directed or undirected) graph \( G = (V, E) \), for any two vertices \( u \) and \( v \), exactly one of the following three conditions holds:

- The intervals \([u.d, u.f]\) and \([v.d, v.f]\) are entirely disjoint, and neither \( u \) nor \( v \) is a descendant of the other in the DFS forest
- The interval \([u.d, u.f]\) is contained entirely within the interval \([v.d, v.f]\), and \( u \) is a descendant of \( v \) in a DFS tree
- The interval \([v.d, v.f]\) is contained entirely within the interval \([u.d, u.f]\), and \( v \) is a descendant of \( u \) in a DFS tree

**Proof.** First, consider the case in which \( u.d < v.d \) (i.e. \( u \) was discovered first). There are two subcases, according to whether \( v.d < u.f \) or not. If \( v.d < u.f \), then \( v \) was discovered while \( u \) was still gray, which implies that \( v \) is a descendant of \( u \). Moreover, since \( v \) was discovered more recently than \( u \), all of its outgoing edges are explored, and \( v \) is finished, before the search returns to and finishes \( u \). In this case, therefore, the interval \([v.d, v.f]\) is entirely contained within the interval \([u.d, u.f]\). In the other subcase, \( u.f < v.d \), and since \( u.d < u.f \) always, we have

\[
u.d < u.f < v.d < v.f
\]

Thus the intervals \([u.d, u.f]\) and \([v.d, v.f]\) are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

The case in which \( v.d < u.d \) is symmetric.
Corollary 1. (Nesting of Descendants’ Intervals) Vertex \( v \) is a proper descendant of vertex \( u \) in the DFS forest for a (directed or undirected) graph \( G \) if and only if \( u.d < v.d < v.f < u.f \).

The next theorem is extremely important, and vies another characterization of when one vertex is a descendant of another in the DFS forest.

Theorem 2. (White Path Theorem (WPT)) In a DFS forest of a graph \( G = (V, E) \), vertex \( v \) is a descendant of vertex \( u \) if and only if at the time \( u.d \) when the search discovers \( u \), there is a path from \( u \) to \( v \) in \( G \) consisting entirely of white vertices.

Proof.

⇒ Direction: If \( v = u \), then the path from \( u \) to \( v \) contains just vertex \( u \), which is still white when we set the value of \( u.d \). Now suppose that \( v \) is a proper descendant of \( u \) in the DFS forest. By the previous corollary, \( u.d < v.d \) and so \( v \) is white at time \( u.d \). Since \( v \) was an arbitrary descendant of \( u \), this implies all vertices on the unique simple path from \( u \) to \( v \) in the DFS forest are white at time \( u.d \).

⇐ Direction: Seeking contradiction, suppose there is a path of white vertices from \( u \) to \( v \) at time \( u.d \), but \( v \) does not become a descendant of \( u \) in the DFS tree containing \( u \). WLOG assume that every vertex other than \( v \) along the path becomes a descendant of \( u \) (otherwise, let \( v \) be the closest vertex to \( u \) along the path that doesn’t become a descendant of \( u \)). Let \( w \) be the predecessor of \( v \) in the path, so that \( w \) is a descendant of \( u \) (it may be the case that \( w = u \)). Again by the previous corollary, \( w.f < u.f \). Because \( v \) must be discovered after \( u \) is discovered, but before \( w \) is finished, we have

\[
u.d < v.d < w.f \leq u.f\]

The parenthesis theorem implies that the interval \([v.d, v.f]\) is entirely contained within the interval \([u.d, u.f]\). By the previous corollary, \( v \) must after all be a descendant of \( u \), a contradiction. \( \square \)

Classifying Edges

Another interesting property of DFS is that it can be used to classify the edges of the input graph \( G = (V, E) \). The type of each edge can provide important information about a graph. For example, we will prove later that a directed graph is acyclic if and only if a DFS search on the graph yields no back edges.

We define four types of edges below:

Definition Let \( G = (V, E) \) be any graph and let \( G_{DFS} \) be the DFS forest produced by a DFS on \( G \).

A. **Tree Edges** are edges in the DFS forest \( G_{DFS} \). \((u, v)\) is a tree edge if \( v \) was first discovered by exploring edge \((u, v)\).

B. **Back Edges** are those edges \((u, v)\) connected a vertex \( u \) to an ancestor \( v \) in a DFS tree in \( G_{DFS} \).

C. **Forward Edges** are those non-tree edges \((u, v)\) connected a vertex \( u \) to a descendant \( v \) in a DFS tree in \( G_{DFS} \).

D. **Cross Edges** are all other edges. They can go between vertices in the same DFS tree, as long as
one vertex is not an ancestor of the other, or they can go between vertices in different DFS tree in $G_{DFS}$

For example, in the DFS traversal from Figure 2, edge (1, 2) is a tree edge, edge (5, 1) is a back-edge, edge (3, 1) is a cross edge, and there are no forward edges (if we added the edge (1, 4) to the graph and ran DFS in ascending order of vertices as before, then the edge (1, 4) would be a forward edge in the DFS forest).

We can classify edge $(u, v)$ based on the color of $v$ when the edge is first explored:

- If $v$ is white, $(u, v)$ is a tree edge
- If $v$ is gray, $(u, v)$ is a back edge
- If $v$ is black, $(u, v)$ is either a forward or a cross edge

An undirected graph may entail some ambiguity about how to classify edges, since $(u, v)$ and $(v, u)$ are really the same edge. In such a case, we classify the edge as the first type in the classification list that applies. Equivalently, we classify the edge according to whichever of $(u, v)$ or $(v, u)$ the DFS encounters first.

**Theorem 3.** In a DFS of an undirected graph $G$, every edge of $G$ is either a tree edge or a back edge.

**Proof.** Let $(u, v)$ be an arbitrary edge of $G$, and suppose WLOG that $u.d < v.d$. Then the search must discover and finish $v$ before it finishes $u$ (while $u$ is gray), since $v$ is on $u$’s adjacency list. If the first time that the search explores edge $(u, v)$, it is in the direction from $u$ to $v$, then $v$ is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction from $v$ to $u$. Thus $(u, v)$ becomes a tree edge. If the search explores $(u, v)$ first in the direction from $v$ to $u$, then $(u, v)$ is a back edge, since $u$ is still gray at the time the edge is first explored. □