1 Graph Traversal Algorithms

Graph traversal algorithms, such as **Breadth First Search (BFS)** and **Depth First Search (DFS)**, allows us to visit all vertices and edges in a way that respects the structure of the graph.

1.1 BFS

Suppose I want to find the shortest path between two vertices. When we talk about distances in graphs, we generally have some cost, length, or weight associated with each edge.

For this class, we will consider the shortest path to mean the path with a minimal number of edges, which corresponds to assigning each edge a length of 1. To find the shortest path between two vertices, consider the following algorithm.

```
BFS(G, s)
for each $v \in V$
    v.dist = $\infty$
s.dist = 0
q.enqueue(s)
while q $\neq \emptyset$
    v = q.dequeue
    for each w s.t. (v, w) $\in E$
        if w.dist = $\infty$
            w.dist = v.dist + 1
            q.enqueue(w)
```

At first, it seems that the running time should be $O(|V||E|)$, since the while loop is iterating through vertices and the for loop is iterating through edges.

But if we examine the algorithm closely, we see the running time is much better. This is easier to see if we unroll the while loop iterations:

```
    for each w s.t. ($v_1$, w) $\in E$
        if w.dist = $\infty$
            w.dist = v.dist + 1
            q.enqueue(w)
    for each w s.t. ($v_2$, w) $\in E$
        if w.dist = $\infty$
            w.dist = v.dist + 1
            q.enqueue(w)
    for each w s.t. ($v_3$, w) $\in E$
        if w.dist = $\infty$
            w.dist = v.dist + 1
```

```
                q.enqueue(w)
    for each w s.t. ($v_4$, w) $\in E$
        if w.dist = $\infty$
            w.dist = v.dist + 1
            q.enqueue(w)
...
    for each w s.t. ($v_n$, w) $\in E$
        if w.dist = $\infty$
            w.dist = v.dist + 1
            q.enqueue(w)
```

Since each vertex is put into the queue at most once, the for loop only examines each edge at most once across all iterations. So the running time is actually $O(|V| + |E|)$.
The following are some examples of problems that can be solved using BFS.

1. Distance between two vertices.
2. Distance between a source and all vertices.
3. Distance between all pairs of vertices.
4. Determine if a graph is bipartite.
5. Determine the number of connected components.

Problems 1, 2, and 3 can all be solved using the algorithm written above, by either running it on a single source vertex or once for each vertex in the graph.

Problem 5 can be solved by running BFS until there are no vertices with dist = ∞ and keeping track of the number of times the algorithm restarts.
To solve problem 4, we can write an algorithm which leverages the following theorem:

Theorem 1.1. *Let G be an undirected graph. G is bipartite $\iff$ G contains no cycles of odd length.*

Proof. ($\Rightarrow$) We know this from the homework.

($\Leftarrow$) Suppose G has no odd cycles.

Let $u \in V$, partition V based on the parity of distance from u.

Then

$$X = \{v \in V : d(u, v) \text{ is even}\}$$

$$Y = \{v \in V : d(u, v) \text{ is odd}\}$$

If G is connected, then $X \cap Y = \emptyset$ and $X \cup Y = V$.

To keep the proof simple, from this point we will only talk about connected G. Once the connected case is understood, how to handle an unconnected graph should be easy to imagine.

For the sake of finding a contradiction later, assume X, Y is not a bipartition.

So there exists an edge connecting two vertices in X or two vertices in Y.

WLOG, lets say there exists $x_1, x_2 \in X$ such that $\{x_1, x_2\} \in E$.

Therefore, there exists a path P_1 of even length from u to x_1, and a path P_2 of even length from u to x_2.

Then P_1, P_2, and $\{x_1, x_2\}$ make a cycle of odd length!
$\rightarrow\leftarrow$

So X, Y must be a bipartition, demonstrating that G is a bipartite graph.

If G is not a connected graph, handle each connected component individually and use the same argument to show they are all bipartite. □

Not only does this prove the theorem, it is a **constructive proof**; it leads us directly to the following modification of our BFS algorithm:

```
Bipartite(G):
    bipartite = true
    for each v in V
        v.color = None
    for each v in V
        if v.color = None
            bipartite = bipartite and two_color(G, v)
    return bipartite

two_color(G, s):
    s.color = 0
    q.enqueue(s)
    while q.length != 0
        v = q.dequeue()
        for each w such that {v, w} in E
            if w.color = v.color
                return false
            else if w.color = None
                w.color = (v.color + 1) mod 2
                q.enqueue(w)
    return true
```

What is the running time here?

two_color runs in time $O(|V| + |E|)$.

Does this mean Bipartite runs in time $O(|V|(|V| + |E|)) = O(|V|^2 + |V||E|)$?

Each vertex is only colored once, thus the running time is $O(|V| + |E|)$.

What are examples of the best case and worst case running time?

1.2 DFS

While the core of BFS is to grow a frontier of explored vertices at equal distance from the starting vertex, DFS takes the approach of exploring as far into the graph as possible before widening the search.

```
**DFS**(G)
    for each u $\in V$
        u.color = white
        u.$\pi$ = nil
    time = 0
    for each u $\in V$
        if u.color = white
            Visit(G, u)

**Visit**(G, u)
    time = time + 1
    u.d = time
    u.color = gray
    for each v $\in G$.Adj[u]
        if v.color = white
            v.$\pi$ = u
            Visit(G, v)
    u.color = black
    time = time + 1
    u.f = time
```

The fields π, d, and f are all values we care about. For a vertex v, the vertex stored in v.π to be the "parent" of v, and helps us recover the order in which DFS visited the vertices. The d parameter is the "discovery" time of the vertex, and f is the "finishing" time of the vertex. Later on we will see problems and algorithms that use these parameters.

The running time of this algorithm is $O(|V| + |E|)$. Just like in BFS, if we carefully consider how the algorithm works, we see that each vertex is turned from white to gray and from gray to black at most once, and each edge is traversed just once.

With DFS, we also sometimes like to consider a classification of the edges based on how the algorithm runs. This classification depends on a specific instance of DFS; If DFS is ran again but makes different choices, the classification of edges may change.

First, we have the following classification of edges for a directed graph:

1. **Tree Edge:** (u, v) is a tree edge if v.π = u.
2. **Back Edge:** (u, v) is a back edge if v is an ancestor of u.
3. **Forward Edge:** (u, v) is a forward edge if u is an ancestor of v and (u, v) is not a tree edge.
4. **Cross Edge:** All other edges are cross edges.

For undirected graphs, we have the following classifications:

1. **Tree Edge:** {u, v} is a tree edge if v.π = u or u.π = v.
2. **Back Edge:** {u, v} is a back edge if v is an ancestor of u or if u is an ancestor or v.

Note that since the edges are undirected, there is no distinction between back and forward edges in an undirected graph. Also, since edges are undirected, it is not possible for a cross edge to exist.