Recall

- Graph $G = (V, E)$
- Set of nodes $V$ of size $n$
- Set of edges $E$ of size $m$

Adjacency List Representation

- Nodes numbered $1, \ldots, n$.
- $\text{Adj}[v]$ points to a list of all of $v$’s neighbors.

BFS Description

Define layer $L_i$ = all nodes at distance exactly $i$ from $s$.

Layers

- $L_0 = \{s\}$
- $L_1 =$ all neighbors of $L_0$
- $L_2 =$ all nodes with an edge to $L_1$ that don’t belong to $L_0$ or $L_1$
- $\ldots$
- $L_{i+1} =$ nodes with an edge to $L_i$ that don’t belong to any earlier layer.

$L_{i+1} = \{v : \exists (u, v) \in E, u \in L_i, v \notin (L_0 \cup \ldots \cup L_i)\}$

DFS Descriptions

Depth-first search: keep exploring from the most recently discovered node until you have to backtrack.

$\text{DFS}(u)$

Mark $u$ as "Explored"

for each edge $(u, v)$ incident to $u$ do

    if $v$ is not marked "Explored" then

        Recursively invoke $\text{DFS}(v)$

    end if

end for
Traversals Implementations

Maintain set of explored nodes and discovered

- Explored = have seen this node and explored its outgoing edges
- Discovered = the “frontier”. Have seen the node, but not explored its outgoing edges.

Generic Graph Traversal

Let $A$ = data structure of discovered nodes
Traverse($s$)
Put $s$ in $A$
while $A$ is not empty do
    Take a node $v$ from $A$
    if $v$ is not marked "explored" then
        Mark $v$ as "explored"
        for each edge $(v, w)$ incident to $v$ do
            if $w$ not marked "discovered" then
                mark $w$ as "discovered"
                Put $w$ in $A$  \( \triangleright w \) is discovered
            end if
        end for
        end if
    end while

Note: one part of this algorithm seems really dumb. Why?
Can put multiple copies of a single node in $A$.

Question 1: If $A$ is a queue (FIFO) is this BFS?
Question 2: If $A$ is a stack (LIFO) is this DFS?

Discovered?

- With discovered array, it’s not DFS! (So let’s not use it)

Let $A$ = data structure of discovered nodes
Traverse($s$)
Put $s$ in $A$
while $A$ is not empty do
    Take a node $v$ from $A$
    if $v$ is not marked "explored" then
        Mark $v$ as "explored"
        for each edge $(v, w)$ incident to $v$ do
            Put $w$ in $A$
                $w$ is discovered
        end for
        end if
    end while

BFS: $A$ is a queue (FIFO)
DFS: $A$ is a stack (LIFO)

Interlude (Data Structures)

Linked List:

- Always remove items from front (Head)
- Queue: Insert at Tail (FIFO)
- Stack: Insert at Head (LIFO)
- Insert/Removal are $O(1)$ operations.

BFS Implementation

Let $A$ = empty Queue structure of discovered nodes
Traverse($s$)
Put $s$ in $A$
while $A$ is not empty do
    Take a node $v$ from $A$
    if $v$ is not marked "explored" then
        Mark $v$ as "explored"
        for each edge $(v, w)$ incident to $v$ do
            Put $w$ in $A$  \( \triangleright w \) is discovered
        end for
        end if
    end while

Is this actually BFS? Yes
Running time? $\Theta(n + m)$
### DFS Implementation

Let $A$ = empty Stack structure of discovered nodes

```plaintext
Traverse(s)
Put s in A
while A is not empty do
  Take a node $v$ from A
  if $v$ is not marked "explored" then
    Mark $v$ as "explored"
    for each edge $(v, w)$ incident to $v$ do
      Put $w$ in A  // $w$ is discovered
    end for
  end if
end while
```

Is this actually DFS? Yes

What’s the running time?

### Back to Connected Components

FindCC($G$)

```plaintext
while There is some unexplored node $s$ do
  BFS($s$)
  Extract connected component $C(s)$.
end while
```

Running time for finding connected components?

**Naive:** $O(n + m)$ for each component $\Rightarrow O(c(n + m))$ if $c$ components.

**Better:**
- BFS on component $C$ only works on nodes/edges in $C$.
- Running time is $O(\sum_C |V(C)| + |E(C)|) = O(n + m)$.

### Bipartite Graphs

**Definition** Graph $G = (V, E)$ is bipartite if $V$ can be partitioned into sets $X, Y$ such that every edge has one end in $X$ and one in $Y$.

**Example** Student-College Graph in stable matching

**Counter example** Cycle of length $k$ for $k$ odd.

**Claim** If $G$ is bipartite then it cannot contain an odd cycle.

### Bipartite Testing

**Question** Given $G = (V, E)$, is $G$ bipartite?

How do we design an algorithm to test bipartiteness?

- BFS($s$) for any $s$, keep track of layers.
- Nodes in odd layers get color blue, even get color red.
- After, check all edges have different colored endpoints.

**Running time?** $O(n + m)$.

### Analysis of Bipartite Testing

**Claim** After running BFS on a connected graph $G$, either,

- There are no edges between two nodes of the same layer $\Rightarrow G$ is bipartite.
- There is an edge between two nodes of the same layer $\Rightarrow G$ has an odd cycle, is not bipartite.

$G$ bipartite if and only if no odd cycles.

### Directed Graphs

- **Directed Graph** $G = (V, E)$.
- $V$ is a set of vertices/nodes.
- $E$ is a set of ordered pairs $(u, v)$.
- Express asymmetrical relationship

**Examples** Twitter network, course schedule, web graph.
Adjacency Lists

Maintain two lists.
- Enter[v] contains all edges pointing to v.
- Leave[v] contains all edges pointing from v.

Strong Connectivity

Definition G is strongly connected if for every u, v ∈ V, there is a path from u to v and from v to u.

Problem Test if G is strongly connected?

Definition The strongly connected component containing vertex s is the set of all nodes with paths to and from s.

Think about Can you find all SCCs in linear time?

Directed Acyclic Graphs

Definition A directed acyclic graph (DAG) is a directed graph with no cycles.

Example Course prerequisites

- Math132
- CS187
- CS220
- CS240
- CS250
- CS311
- CS383

Can you find a way to take all of the courses?

Topological Sorting

Definition A topological ordering of G = (V, E) is an ordering v₁, v₂, ..., vₙ of the nodes, such that for all edges (vᵢ, vⱼ) ∈ E, we must have i < j.

Claim If G has a topological ordering, then G is a DAG.

Task

Problem Given DAG G, compute a topological ordering for G.

- Does one always exist?

\[
\text{topo-sort}(G)
\]

while there are nodes remaining do
  Find a node v with no incoming edges
  Place v next in the order
  Delete v and all of its outgoing edges from G
end while

Running time? O(n² + m) easy, O(m + n) more clever.
Topological Sorting Analysis

- In a DAG, there is always a node \( v \) with no incoming edges.
- Removing a node \( v \) from a DAG, produces a new DAG.
- Any node with no incoming edges can be first in topological ordering.

**Theorem** \( G \) is a DAG if and only if \( G \) has a topological ordering.

Graphs Summary

- Graph Traversal
  - BFS/DFS, Connected Components, Bipartite Testing
  - Traversal Implementation and Analysis
- Directed Graphs
  - Strong Connectivity
  - Directed Acyclic Graphs
  - Topological ordering