Overview

- **Graph Definitions Recap & Graph Connectivity Problems**
  - Definitions
  - Connectivity Problems

- **Search Techniques I: Breadth-First Search (BFS)**
  - Shortest Paths
  - Bipartite Graphs

- Acknowledgements
A graph $G(V, E)$ is the following data:
1. a set of vertices $V$ (usually $V = [n]$)
A graph $G(V, E)$ is the following data:

1. a set of vertices $V$ (usually $V = [n]$)
2. a set of edges (directed or undirected) $E$ (usually $|E| = m$)

- if undirected, edges will be sets $\{u, v\}$, where $u, v \in V$, thus $E \subset \binom{[n]}{2}$

- if directed, edges will be tuples $(u, v)$, thus $E \subset V \times V$
A graph $G(V, E)$ is the following data:

1. A set of vertices $V$ (usually $V = [n]$)
2. A set of edges (directed or undirected) $E$ (usually $|E| = m$)
   - If undirected, edges will be sets $\{u, v\}$, where $u, v \in V$, thus $E \subset [n]_2$
   - If directed, edges will be tuples $(u, v)$, thus $E \subset V^2$

Note that in directed case order matters!
A graph $G(V, E)$ is the following data:

1. a set of vertices $V$ (usually $V = [n]$)
2. a set of edges (directed or undirected) $E$ (usually $|E| = m$)

   - if **undirected**, edges will be sets $\{u, v\}$, where $u, v \in V$, thus $E \subset \binom{[n]}{2}$
   - if **directed**, edges will be tuples $(u, v)$, thus $E \subset V^2$

Note that in directed case order matters!

**Graph representations:** let $G([n], E)$ be a graph

1. Adjacency matrix: $n \times n$ matrix $A$ where

   \[
   A_{ij} = 1 \text{ iff } \{i, j\} \in E \quad \text{(undirected)}
   \]
   \[
   A_{ij} = 1 \text{ iff } (i, j) \in E \quad \text{(directed)}
   \]

2. Adjacency list:
Graph Definitions Recap & Graph Connectivity Problems
- Definitions
- Connectivity Problems

Search Techniques I: Breadth-First Search (BFS)
- Shortest Paths
- Bipartite Graphs

Acknowledgements
Graph Connectivity

- Given a graph $G(V, E)$, two vertices $u, v \in V$ are **connected** in $G$ if there is a path from $u$ to $v$.
Graph Connectivity

Given a graph $G(V, E)$, two vertices $u, v \in V$ are connected in $G$ if there is a path from $u$ to $v$

A subset $S \subseteq V$ is connected if, for any $u, v \in S$, we have that $u$ and $v$ are connected
Graph Connectivity

- Given a graph $G(V, E)$, two vertices $u, v \in V$ are connected in $G$ if there is a path from $u$ to $v$
  - A subset $S \subseteq V$ is connected if, for any $u, v \in S$, we have that $u$ and $v$ are connected
  - A graph is connected if $V$ is connected
Graph Connectivity

- Given a graph $G(V, E)$, two vertices $u, v \in V$ are *connected* in $G$ if there is a path from $u$ to $v$.
- A subset $S \subseteq V$ is connected if, for any $u, v \in S$, we have that $u$ and $v$ are connected.
- A graph is connected if $V$ is connected.
- A *connected component* is a maximally connected subset of vertices.

Important basic questions:

1. Is $G$ connected?
2. Can we find all the connected components of $G$?
3. Given $u, v \in V$, are they connected?
4. Given $u, v \in V$, can we output a shortest path between $u, v$?
Given a graph $G(V, E)$, two vertices $u, v \in V$ are connected in $G$ if there is a path from $u$ to $v$

- A subset $S \subseteq V$ is connected if, for any $u, v \in S$, we have that $u$ and $v$ are connected
- A graph is connected if $V$ is connected
- A connected component is a maximally connected subset of vertices

Important basic questions: given a graph $G$

1. is $G$ connected?
2. can we find all the connected components of $G$?
3. given $u, v \in V$, are they connected?
4. given $u, v \in V$, can we output a shortest path between $u, v$?
Graph Definitions Recap & Graph Connectivity Problems
- Definitions
- Connectivity Problems

Search Techniques I: Breadth-First Search (BFS)
- Shortest Paths
- Bipartite Graphs

Acknowledgements
Breadth-First Search

- **Input:** graph $G(V,E)$, vertex $s \in V$ (adjacency list)
- **Output:** all vertices in $G$ reachable from $s$
Breadth-First Search

- **Input:** graph $G(V, E)$, vertex $s \in V$ (adjacency list)
- **Output:** all vertices in $G$ reachable from $s$
- **BFS Algorithm:**
  1. **Initialization:**
     - array $\text{visited}[v] = 0$ for all $v \in V$.
     - queue $Q = \emptyset$
  2. **Start:**
     - $\text{ENQUEUE}(Q, s)$
     - $\text{visited}[s] = 1$
  3. **While** $Q \neq \emptyset$:
     - $u = \text{DEQUEUE}(Q)$
     - for each neighbor $v$ of $u$:
       - if $\text{visited}[v] = 0$ then $\text{ENQUEUE}(Q, v)$ and $\text{visited}[v] = 1$
Initialization costs $O(n)$

Each vertex $v$ is enqueued at most once

if we traverse it and $\text{visited}[v] = 0$

when we dequeue a vertex $v$, run loop for $\deg(v)$ iterations

Thus, running time is:

$$O \left( n + \sum_{v \in V} \deg(v) \right) = O(m + n)$$
Correctness & Structural Lemma

**Lemma (Connectivity)**

\( G \) has an \( s \) – \( t \) path iff \( \text{visited}[t] = 1 \) at the end of BFS algorithm.
Correctness & Structural Lemma

Lemma (Connectivity)

$G$ has an $s - t$ path iff $\text{visited}[t] = 1$ at the end of BFS algorithm.

- $\exists s - t$ path $\Rightarrow$ $\text{visited}[t] = 1$
  1. Take path $s = u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_{k-1} \rightarrow u_k = t$
  2. By induction, each $u_i$ is added to $Q$ and thus we have $\text{visited}[u_i] = 1$
     - If $u_i$ not added until we visit $u_{i-1}$, then we enqueue it when visit $u_{i-1}$

Bonus:
- If $\text{graph}$ is connected: $\text{visited}[v] = 1$ for all $v \in V$ connected component containing $s$: return all vertices $v \in V$ with $\text{visited}[v] = 1$ if there is $s - t$ path for vertex $t \in V$: just check if $\text{visited}[t] = 1$

Can find all connected components:
- once BFS finishes, scan visited array to find a vertex $u$ that hasn't been visited yet, run BFS starting from this vertex $u$ iterate until all vertices are visited
Correctness & Structural Lemma

Lemma (Connectivity)

\( G \) has an \( s - t \) path iff \( \text{visited}[t] = 1 \) at the end of BFS algorithm.

- \( \exists \ s - t \) path \( \Rightarrow \) \( \text{visited}[t] = 1 \)
  1. Take path \( s = u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_{k-1} \rightarrow u_k = t \)
  2. By induction, each \( u_i \) is added to \( Q \) and thus we have \( \text{visited}[u_i] = 1 \)
     If \( u_i \) not added until we visit \( u_{i-1} \), then we enqueue it when visit \( u_{i-1} \)

- \( \text{visited}[t] = 1 \) \( \Rightarrow \exists \ s - t \) path
  1. **Idea:** trace back an \( s - t \) path from algorithm
  2. Let \( u_0 \) be vertex where \( \text{visited}[t] \) was set to 1, and inductively, let \( u_i \) be
     vertex where \( \text{visited}[u_{i-1}] \) was set to 1.
  3. Process has to stop, as we enqueue each vertex at most once, and can
     only stop at \( s \) (as process stops when queue is empty).

Bonus:

Can also answer if graph is connected: \( \text{visited}[v] = 1 \) for all \( v \in V \) connected component containing \( s \): return all vertices \( v \in V \) with \( \text{visited}[v] = 1 \) if there is an \( s - t \) path for vertex \( t \in V \): just check if \( \text{visited}[t] = 1 \)

Can find all connected components:

Once BFS finishes, scan visited array to find a vertex \( u \) that hasn't been
visited yet, run BFS starting from this vertex \( u \) iterate until all vertices are visited.
Lemma (Connectivity)

\[ G \text{ has an } s - t \text{ path iff } \text{visited}[t] = 1 \text{ at the end of BFS algorithm.} \]

- Correctness of algorithm follows from lemma
Lemma (Connectivity)

\( G \) has an \( s \) – \( t \) path iff \( \text{visited}[t] = 1 \) at the end of BFS algorithm.

- Correctness of algorithm follows from lemma
- **Bonus**: can also answer
  - if graph is connected: \( \text{visited}[v] = 1 \) for all \( v \in V \)
  - connected component containing \( s \): return all vertices \( v \in V \) with \( \text{visited}[v] = 1 \)
  - if there is \( s \) – \( t \) path for vertex \( t \in V \): just check if \( \text{visited}[t] = 1 \)
Correctness & Structural Lemma

**Lemma (Connectivity)**

$G$ has an $s - t$ path iff $\text{visited}[t] = 1$ at the end of BFS algorithm.

- Correctness of algorithm follows from lemma
- **Bonus:** can also answer
  - if graph is connected: $\text{visited}[v] = 1$ for all $v \in V$
  - connected component containing $s$: return all vertices $v \in V$ with $\text{visited}[v] = 1$
  - if there is $s - t$ path for vertex $t \in V$: just check if $\text{visited}[t] = 1$
- Can find all connected components:
  - once BFS finishes, scan visited array to find a vertex $u$ that hasn’t been visited yet,
  - run BFS starting from this vertex $u$
  - iterate until all vertices are visited
BFS Tree

- From our proof of lemma, can trace path from $s$ to $t$ for every visited vertex
  1. Let the “parent of $v$,” denoted $p[v]$, be the vertex $u \in V$ such that the BFS algorithm sets $\text{visited}[v] = 1$ while looping through $u$.
  2. Let $T \subset E$ be the set of edges $\{v, p[v]\}$
  3. Let $U \subset V$ be the connected component of $s$
BFS Tree

- From our proof of lemma, can trace path from $s$ to $t$ for every visited vertex
  1. Let the “parent of $v$,” denoted $p[v]$, be the vertex $u \in V$ such that the BFS algorithm sets $\text{visited}[v] = 1$ while looping through $u$.
  2. Let $T \subseteq E$ be the set of edges $\{v, p[v]\}$
  3. Let $U \subseteq V$ be the connected component of $s$

- The graph $(U, T)$ is a tree, called the **BFS tree**
BFS Tree

- From our proof of lemma, can trace path from $s$ to $t$ for every visited vertex
  1. Let the “parent of $v$,” denoted $p[v]$, be the vertex $u \in V$ such that the BFS algorithm sets $\text{visited}[v] = 1$ while looping through $u$.
  2. Let $T \subset E$ be the set of edges $\{v, p[v]\}$
  3. Let $U \subset V$ be the connected component of $s$

- The graph $(U, T)$ is a tree, called the **BFS tree**

- Why is it a tree?
  - $(U, T)$ is connected and and $|T| = |U| - 1$ by our proof of the lemma
  - edges cannot form a cycle, since each parent must appear before its children in the algorithm
Augmented Breadth-First Search

(Augmented) BFS Algorithm:

1. **Initialization:**
   - array visited[v] = 0 for all \( v \in V \).
   - queue \( Q = \emptyset \)
   - array \( p[v] = \text{NULL} \) for all \( v \in V \)

2. **Start:**
   - ENQUEUE(\( Q, s \))
   - visited[s] = 1

3. **While** \( Q \neq \emptyset \):
   - \( u = \text{DEQUEUE}(Q) \)
   - for each neighbor \( v \) of \( u \):
     - if visited[\( v \)] = 0 then:
       - ENQUEUE(\( Q, v \))
       - visited[\( v \)] = 1
       - \( p[v] = u \)
Another useful property of the BFS algorithm is that we obtain *shortest paths* between \( s \) and any other vertex \( u \in V \)!\(^1\)

\(^1\)For unweighted graphs.
Another useful property of the BFS algorithm is that we obtain shortest paths between \( s \) and any other vertex \( u \in V \! \).

Idea: can simply add “levels” to the BFS algorithm.

- Each vertex \( v \) gets a level \( \ell(v) \). (initially set to \( \infty \))
- Set \( \ell(s) = 0 \), and whenever add \( v \) to queue, set \( \ell(v) = \ell(p[v]) + 1 \)
- Induction: level of a vertex equals its distance to \( s \), since each vertex
Augmented Breadth-First Search

(Augmented) BFS Algorithm:

1. Initialization:
   - array visited[v] = 0 for all v ∈ V.
   - queue Q = 0
   - array p[v] = NULL for all v ∈ V
   - array ℓ[v] = ∞ for all v ∈ V

2. Start:
   - ENQUEUE(Q, s)
   - visited[s] = 1
   - ℓ[s] = 0

3. While Q ≠ ∅:
   - u = DEQUEUE(Q)
   - for each neighbor v of u:
     - if visited[v] = 0 then:
       - ENQUEUE(Q, v)
       - visited[v] = 1
       - p[v] = u
       - ℓ[v] = ℓ[u] + 1
Bipartite Graphs

**Bipartite Graph:** we say that $G(V, E)$ is a bipartite graph if we can partition $V = L \sqcup R$ such that:

1. $L \cap R = \emptyset$
2. $E$ only has edges of the form $\{u, v\}$ where $u \in L$ and $v \in R$

Can use BFS algorithm to check whether graph is bipartite. Simply run BFS and partition $V = L \sqcup R$ with:

- $L := \{u \in V | \ell(u) \equiv 0 \text{ mod } 2\}$
- $R := \{u \in V | \ell(u) \equiv 1 \text{ mod } 2\}$

Run BFS again and check if there is an edge between two vertices of $L$ or two vertices of $R$. If there is, return non-bipartite. Else, return bipartite.
Bipartite Graphs

- **Bipartite Graph**: we say that $G(V, E)$ is a bipartite graph if we can partition $V = L \sqcup R$ such that:
  1. $L \cap R = \emptyset$
  2. $E$ only has edges of the form \{u, v\} where $u \in L$ and $v \in R$

- Can use BFS algorithm to check whether graph is bipartite
Bipartite Graphs

- **Bipartite Graph**: we say that $G(V, E)$ is a bipartite graph if we can partition $V = L \sqcup R$ such that:
  1. $L \cap R = \emptyset$
  2. $E$ only has edges of the form $\{u, v\}$ where $u \in L$ and $v \in R$

- Can use BFS algorithm to check whether graph is bipartite

- Simply run BFS and partition $V = L \sqcup R$ with:
  
  $L := \{u \in V \mid \ell(u) \equiv 0 \mod 2\}$ and $R := \{u \in V \mid \ell(u) \equiv 1 \mod 2\}$

- Run BFS again and check if there is an edge between two vertices of $L$ or two vertices of $R$.
  - If there is, return non-bipartite
  - Else, return bipartite
Bipartite Graphs

- **Bipartite Graph:** we say that $G(V, E)$ is a bipartite graph if we can partition $V = L \sqcup R$ such that:
  1. $L \cap R = \emptyset$
  2. $E$ only has edges of the form $\{u, v\}$ where $u \in L$ and $v \in R$

- Can use BFS algorithm to check whether graph is bipartite
- Simply run BFS and partition $V = L \sqcup R$ with:
  \[ L := \{ u \in V \mid \ell(u) \equiv 0 \mod 2 \} \quad \text{and} \quad R := \{ u \in V \mid \ell(u) \equiv 1 \mod 2 \} \]
- Run BFS again and check if there is an edge between two vertices of $L$ or two vertices of $R$.
  - If there is, return non-bipartite
  - Else, return bipartite
Correctness of Algorithm

- Easy to see that algorithm always correct when we return bipartite, as we checked there are no edges within $L$ or $R$
Correctness of Algorithm

- Easy to see that algorithm always correct when we return bipartite, as we checked there are no edges within $L$ or $R$
- Hard case: is the algorithm correct when we return NO?
  
  Graph bipartite $\Leftrightarrow$ NO odd cycles\textsuperscript{1}

\textsuperscript{1}MATH 239/249
Correctness of Algorithm

- Easy to see that algorithm always correct when we return bipartite, as we checked there are no edges within $L$ or $R$
- Hard case: is the algorithm correct when we return NO?

  Graph bipartite $\iff$ NO odd cycles

- Let $T$ be BFS tree of $G$ with root $s$.
  - Suppose we find an edge between vertices $u, v \in L$ (w.l.o.g.)
  - Let $w$ be lowest common ancestor of $u, v$ in $T$, and let $P_{uw}, P_{wv}$ be the paths $u \rightarrow w$ and $w \rightarrow v$ in $T$.
  - Consider cycle $C := \{u, v\} \cup P_{uw} \cup P_{wv}$.
    - Since $\ell(u), \ell(v) \equiv 0 \mod 2$ and $|P_{uw}| = \ell(u) - \ell(w)$, $|P_{wv}| = \ell(v) - \ell(w)$, we have
      $$|P_{uw}| \equiv |P_{wv}| \equiv -\ell(w) \mod 2$$
    - Thus $|P_{uw}| + |P_{wv}| + 1 \equiv 1 \mod 2 \Rightarrow C$ is odd cycle.
Remarks

- Above can be modified to give algorithmic proof that graph is bipartite iff no odd cycles
- Linear time algorithm to find odd cycle of undirected graph
- Having odd cycle is a “short proof” of non-bipartiteness (and easy!)
Acknowledgement

- Based on Prof. Lau’s lecture 05
  
  https://cs.uwaterloo.ca/~lapchi/cs341/notes/L05.pdf
Cormen, Thomas and Leiserson, Charles and Rivest, Ronald and Stein, Clifford. (2009)
*MIT Press*

Kleinberg, Jon and Tardos, Eva (2006)
Algorithm Design.
*Addison Wesley*