Graphs

3.1 Basic Definitions and Applications

Course instructor:

Sikder Huq
Undirected Graphs

Undirected graph. $G = (V, E)$

- $V =$ nodes.
- $E =$ edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|, m = |E|.$

$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$

$E = \{1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6\}$

$n = 8$

$m = 11$
Some Graph Applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersections</td>
<td>highways</td>
</tr>
<tr>
<td>communication</td>
<td>computers</td>
<td>fiber optic cables</td>
</tr>
<tr>
<td>World Wide Web</td>
<td>web pages</td>
<td>hyperlinks</td>
</tr>
<tr>
<td>social</td>
<td>people</td>
<td>relationships</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey</td>
</tr>
<tr>
<td>software systems</td>
<td>functions</td>
<td>function calls</td>
</tr>
<tr>
<td>scheduling</td>
<td>tasks</td>
<td>precedence constraints</td>
</tr>
<tr>
<td>circuits</td>
<td>gates</td>
<td>wires</td>
</tr>
</tbody>
</table>
World Wide Web

Web graph.
- Node: web page.
- Edge: hyperlink from one page to another.
9-11 Terrorist Network

Social network graph.
- **Node:** people.
- **Edge:** relationship between two people.

Ecological Food Web

Food web graph.

- Node = species.
- Edge = from prey to predator.

Lecture 5
Graph Representation: Adjacency Matrix

Adjacency matrix. $n$-by-$n$ matrix with $A_{uv} = 1$ if $(u, v)$ is an edge.

- Two representations of each edge.
- Space proportional to $n^2$.
- Checking if $(u, v)$ is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.
**Graph Representation: Adjacency List**

**Adjacency list.** Node indexed array of lists.
- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if $(u, v)$ is an edge takes $O(\text{deg}(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.

degree = number of neighbors of $u$
Paths and Connectivity

**Def.** A path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, …, v_{k-1}, v_k$ with the property that each consecutive pair $v_i, v_{i+1}$ is joined by an edge in $E$.

**Def.** A path is **simple** if all nodes are distinct.

**Def.** An undirected graph is **connected** if for every pair of nodes $u$ and $v$, there is a path between $u$ and $v$. 
Cycles

Def. A cycle is a path $v_1, v_2, ..., v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct.

cycle $C = 1-2-4-5-3-1$
Def. An undirected graph is a tree if it is connected and does not contain a cycle.

Theorem. Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.
- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
Rooted Trees

**Rooted tree.** Given a tree T, choose a root node r and orient each edge away from r.

**Importance.** Models hierarchical structure.
Phylogeny Trees

Phylogeny trees. Describe evolutionary history of species.
GUI Containment Hierarchy

Describe organization of GUI widgets.

Reference: http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html
3.2 Graph Traversal
Connectivity

**s-t connectivity problem.** Given two nodes $s$ and $t$, is there a path between $s$ and $t$?

**s-t shortest path problem.** Given two nodes $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

**Applications.**
- Facebook.
- Maze traversal.
- Erdos number.
- Kevin Bacon number.
- Fewest number of hops in a communication network.
Breadth First Search

**BFS intuition.** Explore outward from \( s \) in all possible directions, adding nodes one "layer" at a time. Effect: find “shallow” paths to nodes.

**BFS algorithm.**

- \( L_0 = \{ s \} \).
- \( L_1 = \) all neighbors of \( L_0 \).
- \( L_2 = \) all nodes that do not belong to \( L_0 \) or \( L_1 \), and that have an edge to a node in \( L_1 \).
- \( L_{i+1} = \) all nodes that do not belong to an earlier layer, and that have an edge to a node in \( L_i \).

**Theorem.** For each \( i \), \( L_i \) consists of all nodes at distance exactly \( i \) from \( s \). There is a path from \( s \) to \( t \) iff \( t \) appears in some layer.
Implementing BFS

Q: What’s a good way to implement the above algorithm?

A: Use a queue for the “frontier”
Breadth First Search

Property. Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
Lecture 6
Breadth First Search: Analysis

Theorem. The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency list representation.

Pf.

- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L_i$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node $u$, there are $\leq n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\text{deg}(u)$ incident edges $(u, v)$
  - total time processing edges is $\sum_{u \in V} \text{deg}(u) = 2m$.

\[ \uparrow \]

each edge $(u, v)$ is counted exactly twice in sum: once in $\text{deg}(u)$ and once in $\text{deg}(v)$
Connected Component

Connected component. Find all nodes reachable from s.

Connected component containing node 1 = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}. 
Q1: Finding connected components

Give an algorithm to find the set of all connected components of an undirected graph.
**Connected Component**

**Connected component.** Find all nodes reachable from $s$.

---

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u,v)$ where $u \in R$ and $v \not\in R$
    Add $v$ to $R$
Endwhile

---

**Theorem.** Upon termination, $R$ is the connected component containing $s$.
- BFS = explore in order of distance from $s$. 

---
Q2: Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
Flood Fill

**Flood fill.** *Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.*
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- **Node**: pixel.
- **Edge**: two neighboring lime pixels.
- **Blob**: connected component of lime pixels.

recolor lime green blob to blue
Depth-first search

Use recursion

**DFS intuition.** Explore outward from \( s \) along one path as far as possible, and backtrack when you cannot progress. Effect: find faraway nodes.

**DFS(\( u \)):**
- Mark \( u \) as “Explored” and add \( u \) to \( R \)
- For each edge \((u,v)\) incident to \( u \)
  - If \( v \) is not marked “Explored” then
    - Recursively call DFS(\( v \))
Depth-first search

**Property.** For a given recursive call DFS(u), all nodes marked “Explored” between the beginning and end of this recursive call are descendants of u in T.

**Theorem.** Let T be a depth-first search tree, let x and y be nodes in T, and let (x,y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other.
Q3: BFS and DFS trees

We have a connected graph $G = (V, E)$ and a specific vertex $u$. Suppose we compute a DFS tree rooted at $u$, and obtain a tree $T$ that includes all nodes of $G$. Suppose we then compute a BFS tree rooted at $u$, and obtain the same tree $T$.

Prove that $G = T$. 

Answer

Suppose $G$ has an edge $e = \{a, b\}$ that does not belong to $T$.

As $T$ is a DFS tree, one of the two ends must be an ancestor of the other—say $a$ is an ancestor of $b$.

(*) Since $T$ is a BFS tree, the distance of the two nodes from $u$ in $T$ can differ at most by one.

But if $a$ is an ancestor of $b$, and (*) holds, then $a$ must be the direct parent of $b$. This means that $\{a, b\}$ is an edge in $T$. Contradiction.
Q4: Finding a cycle

Given a graph $G$, determine if it has a cycle. If so, the algorithm should output this cycle.

Answer: Assume that $G$ is connected; otherwise work on the connected components.

Run BFS from an arbitrary node $s$, and obtain a BFS tree $T$. If every edge of $G$ appears in the tree, then $G = T$ and there is no cycle.

Otherwise, there is an edge $e = (v, w)$ that is in $G$ but not in $T$. Consider the least common ancestor $u$ of $v$ and $w$ in $T$. We get a cycle from edge $e$ and paths $u-v$ and $u-w$ in $T$. 
3.4 Testing Bipartiteness

Lecture 7
Def. An undirected graph $G = (V, E)$ is **bipartite** if the nodes can be colored red or blue such that every edge has one red and one blue end.

**Applications.**
- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.
**Testing Bipartiteness**

**Testing bipartiteness.** Given a graph $G$, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

![a bipartite graph $G$](image1.png)

![another drawing of $G$](image2.png)
An Obstruction to Bipartiteness

Lemma. If a graph $G$ is bipartite, it cannot contain an odd length cycle.
An Obstruction to Bipartiteness

**Lemma.** If a graph $G$ is bipartite, it cannot contain an odd length cycle.

**Pf.** Not possible to 2-color the odd cycle, let alone $G$. 

![Graphs showing bipartite and non-bipartite properties](image)
Bipartite Graphs

Lemma. Let \( G \) be a connected graph, and let \( L_0, \ldots, L_k \) be the layers produced by BFS starting at node \( s \). Exactly one of the following holds.

(i) No edge of \( G \) joins two nodes of the same layer, and \( G \) is bipartite.
(ii) An edge of \( G \) joins two nodes of the same layer, and \( G \) contains an odd-length cycle (and hence is not bipartite).

Case (i)

Case (ii)
Bipartite Graphs

Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (i)

- Suppose no edge joins two nodes in the same layer.
- By previous lemma, this implies all edges join nodes on successive levels.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.
Bipartite Graphs

Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).
**Lemma.** Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.

(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

**Pf.** (ii)

- Suppose $(x, y)$ is an edge with $x, y$ in same level $L_j$.
- Let $z = \text{lca}(x, y) =$ lowest common ancestor.
- Let $L_i$ be level containing $z$.
- Consider cycle that takes edge from $x$ to $y$, then path from $y$ to $z$, then path from $z$ to $x$.
- Its length is $1 + (j-i) + (j-i)$, which is odd.

\[ z = \text{lca}(x, y) \]

\[ (x, y) \quad \text{path from y to z} \quad \text{path from z to x} \]
Obstruction to Bipartiteness

**Corollary.** A graph $G$ is bipartite iff it contains no odd length cycle.

![Diagram](image-url)

- **bipartite (2-colorable)**
- **not bipartite (not 2-colorable)**
Q1: Destroying paths

Suppose that an $n$-node undirected graph $G = (V, E)$ contains two nodes $s$ and $t$ such that the distance between $s$ and $t$ is strictly greater than $n/2$. Show that there must exist some node $v$, not equal to either $s$ or $t$, such that deleting $v$ from $G$ destroys all $s$-$t$ paths.

Give an algorithm with running $O(m+n)$ to find such a node.
**Answer**

Run BFS starting from $s$. Let $d$ be the layer where you encounter $t$. By assumption, $d > n/2$.

Now we claim that one of the layers $L_1, \ldots, L_{d-1}$ has a single node. Why? Because if not, then they account for at least $2(n/2) = n$ nodes. But $G$ has only $n$ nodes, and $s$ and $t$ are not in these layers.

Now let $L_i$ be the layer containing a single node $v$. Suppose we delete $v$. Consider the set $X$ of all nodes in layers $0, \ldots, i-1$. This set cannot contain $t$.

Any edge out of these nodes can only lead to a node in $L_i$ or stay in $X$, by the properties of BFS. But $v$ is the only node in $L_i$. 
Q2: Interference-free paths

Consider the following robotics question. You have an undirected graph $G = (V,E)$ that represents the floor plan of a building, and there are two robots located at nodes $a$ and $b$. The robot at node $a$ wants to move to node $c$; the robot at node $b$ wants to move to node $d$.

This is done using a schedule: a function that at each time step, specifies that a robot moves across a single edge. A schedule is interference-free if there is no point at which the two robots occupy nodes that are at a distance $\leq r$ from one another. (We assume that $a-b$ and $c-d$ are sufficiently far apart.)

Give an algorithm to tell if there is an interference-free schedule that the robots can use.
Don’t consider the graph $G$ but the “product” $H$ of $G$ with itself.

Nodes of $H$: pairs $(u,v)$ where $u$, $v$ are nodes of $G$.

Edges of $H$: $((u,v), (u', v'))$ where
1. Either $u = u'$ and there is an edge between $v$ and $v'$ in $G$
2. $v = v'$ and there is an edge between $u$ and $u'$ in $G$

Now delete from $H$ all nodes where there would be interference, getting a graph $H'$.

Check if there is a path from $(a,b)$ to $(c,d)$ in $H'$.

**Complexity**: $O(mn + n^2)$
3.5 Connectivity in Directed Graphs

Lecture 8
Directed Graphs

**Directed graph.** $G = (V, E)$
- Edge $(u, v)$ goes from node $u$ to node $v$.

**Ex.** Web graph - hyperlink points from one web page to another.
- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.
Graph Search

Directed reachability. Given a node $s$, find all nodes reachable from $s$.

Directed $s$-$t$ shortest path problem. Given two nodes $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page $s$. Find all web pages linked from $s$, either directly or indirectly.
**Strong Connectivity**

**Def.** Node $u$ and $v$ are **mutually reachable** if there is a path from $u$ to $v$ and also a path from $v$ to $u$.

**Def.** A graph is **strongly connected** if every pair of nodes is mutually reachable.
**Strong Connectivity**

**Def.** Node u and v are **mutually reachable** if there is a path from u to v and also a path from v to u.

**Def.** A graph is **strongly connected** if every pair of nodes is mutually reachable.

**Lemma.** Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

**Pf.** ⇒ Follows from definition.

**Pf.** ⇐ Path from u to v: concatenate u-s path with s-v path.

Path from v to u: concatenate v-s path with s-u path.

\[ \text{ok if paths overlap} \]
**Strong Connectivity: Algorithm**

**Theorem.** Can determine if $G$ is strongly connected in $O(m + n)$ time.
**Theorem.** Can determine if $G$ is strongly connected in $O(m + n)$ time.

**Pf.**
- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Run BFS from $s$ in $G^{rev}$.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. \[ \blacksquare \]
3.6 DAGs and Topological Ordering
Directed Acyclic Graphs

Def. An **DAG** is a directed graph that contains no directed cycles.

Ex. Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

Def. A **topological order** of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).
Precedence Constraints

Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).

Applications.
- Course prerequisite graph: course \(v_i\) must be taken before \(v_j\).
- Compilation: module \(v_i\) must be compiled before \(v_j\). Pipeline of computing jobs: output of job \(v_i\) needed to determine input of job \(v_j\).
Lemma. If $G$ has a topological order, then $G$ is a DAG.

Pf. (by contradiction)

Suppose that $G$ has a topological order $v_1, ..., v_n$ and that $G$ also has a directed cycle $C$. Let's see what happens.

![Diagram of a directed acyclic graph with vertices $v_1, v_i, v_j, v_n$ and a cycle $C$.]
Lemma. If $G$ has a topological order, then $G$ is a DAG.

Pf. (by contradiction)

- Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$. Let's see what happens.
- Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$; thus $(v_j, v_i)$ is an edge.
- By our choice of $i$, we have $i < j$.
- On the other hand, since $(v_j, v_i)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪
Directed Acyclic Graphs

Lemma. If $G$ has a topological order, then $G$ is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?
Directed Acyclic Graphs

Lemma. If $G$ is a DAG, then $G$ has a node with no incoming edges.

Pf. (by contradiction)
- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.
Lemma. If $G$ is a DAG, then $G$ has a node with no incoming edges.

Pf. (by contradiction)

Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.

Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$.

Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.

Repeat until we visit a node, say $w$, twice.

Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle. ·
Lecture 9
Directed Acyclic Graphs

Lemma. If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)
- Base case: true if $n = 1$. 
Directed Acyclic Graphs

Lemma. If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)
- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges. 

![Diagram of a DAG with a node $v$ and outgoing arrows]
Directed Acyclic Graphs

To compute a topological ordering of $G$:
Find a node $v$ with no incoming edges and order it first
Delete $v$ from $G$
Recursively compute a topological ordering of $G-\{v\}$
and append this order after $v$
Topological Sorting Algorithm: Running Time

**Theorem.** Algorithm finds a topological order in $O(m + n)$ time.

**Pf.**
- Maintain the following information:
  - $\text{count}[w] = \text{remaining number of incoming edges to node } w$
  - $S = \text{set of remaining nodes with no incoming edges}$
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete $v$
  - remove $v$ from $S$
  - decrement $\text{count}[w]$ for all edges from $v$ to $w$, and add $w$ to $S$ if $\text{count}[w]$ hits 0
  - this is $O(1)$ per edge

\[\]
Question

Can you have multiple topological orderings for a graph?
Q1: Reachability game

Suppose you have a bipartite directed graph with nodes in the two partitions colored red and blue, and two players: Red and Blue. Red and Blue play a game where a token gets moved along edges of the graph. At each point, the player whose name matches the color of the current node pushes the token. Initially the token is at $s$ (a red node).

The objective of the game is that Red wants the token to avoid a certain set of blue nodes $X$. Blue wants the token to get to $X$ at some point in the game; Red wants to avoid this. If the token gets to $X$ at any point, the game is over and Blue wins. Aside from this there is no time bound on the game.

Can you give an algorithm that, given the graph, $s$, and $X$, can tell if Red has a strategy to win this game?
Solution

Iteratively grow a set $S$ from which Blue can “force” Red to reach $X$

```
S := X
while (S ≠ S') {
    S' := S
    Add to $S$ every red node $u$ such that ALL neighbors of $u$ are in $S$
    Add to $S$ every blue node $v$ such that SOME neighbor of $v$ is in $S$
}
Check if the initial node lies in $S$
```

The set we compute is called the “attractor” of $S$.

We still need a proof!