Introduction

- Graphs are abstract data types that are applicable to numerous problems
  - Can capture entities, relationships between them, the degree of the relationship, etc.
- This chapter covers basics in graph theory, including representation, and algorithms for basic graph-theoretic problems
- We’ll build on these later this semester

Types of Graphs

- A (simple, or undirected) graph \( G = (V, E) \) consists of \( V \), a nonempty set of vertices and \( E \) a set of unordered pairs of distinct vertices called edges
  
  \[ V = \{A, B, C, D, E\} \]
  
  \[ E = \{(A, D), (A, E), (B, D), (B, E), (C, D), (C, E)\} \]

Types of Graphs (2)

- A directed graph (digraph) \( G = (V, E) \) consists of \( V \), a nonempty set of vertices and \( E \) a set of ordered pairs of distinct vertices called edges

Types of Graphs (3)

- A weighted graph is an undirected or directed graph with the additional property that each edge \( e \) has associated with it a real number \( w(e) \) called its weight

Representations of Graphs

- Two common ways of representing a graph: Adjacency list and adjacency matrix
- Let \( G = (V, E) \) be a graph with \( n \) vertices and \( m \) edges
**Adjacency List**

- For each vertex \( v \in V \), store a list of vertices adjacent to \( v \)
- For weighted graphs, add information to each node
- How much is space required for storage?

```
\begin{array}{cccc}
\text{a} & \text{b} & \text{c} & \text{d} \\
\text{a} & \text{b} & \text{c} & \text{e} \\
\end{array}
```

**Adjacency Matrix**

- Use an \( n \times n \) matrix \( M \), where \( M(i,j) = 1 \) if \((i,j)\) is an edge, 0 otherwise
- If \( G \) weighted, store weights in the matrix, using \( \infty \) for non-edges
- How much is space required for storage?

```
\begin{array}{cccc}
\text{a} & \text{b} & \text{c} & \text{d} \\
\text{a} & 0 & 1 & 1 & 0 \\
\text{b} & 1 & 0 & 0 & 1 \\
\text{c} & 1 & 0 & 0 & 1 \\
\text{d} & 1 & 0 & 1 & 1 \\
\text{e} & 1 & 1 & 1 & 0 \\
\end{array}
```

**Breadth-First Search (BFS)**

- Given a graph \( G = (V, E) \) (directed or undirected) and a source node \( s \in V \), BFS systematically visits every vertex that is reachable from \( s \)
- Uses a queue data structure to search in a breadth-first manner
- Creates a structure called a **BFS tree** such that for each vertex \( v \in V \), the distance (number of edges) from \( s \) to \( v \) in tree is a shortest path in \( G \)
- Initialize each node’s **color** to **white**
- As a node is visited, color it to **gray** (\( \Rightarrow \) in queue), then **black** (\( \Rightarrow \) finished)

```
BFS(G, s)
```

```
1. for each vertex \( v \in V \setminus \{s\} \) do
2. \( \text{color}[v] = \text{white} \)
3. \( d[v] = \infty \)
4. \( \pi[v] = \text{nil} \)
5. end
6. \( \text{color}[s] = \text{gray} \)
7. \( d[s] = 0 \)
8. \( \pi[s] = \text{nil} \)
9. \( Q = \emptyset \)
10. while \( Q \neq \emptyset \) do
11. \( u = \text{Dequeue}(Q) \)
12. for each \( v \in \text{Adj}[u] \) do
13. if \( \text{color}[v] = \text{white} \) then
14. \( \text{color}[v] = \text{gray} \)
15. \( d[v] = d[u] + 1 \)
16. \( \pi[v] = u \)
17. \( \text{Enqueue}(Q, v) \)
18. end
19. end
20. \( \text{color}[u] = \text{black} \)
21. end
```

**BFS Example**

- Each vertex is colored **white** before visiting.
- **Gray** vertices are in the queue.
- **Black** vertices have been visited.

**BFS Example (2)**

- For each vertex \( v \in V \), store a list of vertices adjacent to \( v \)
- For weighted graphs, add information to each node
- How much is space required for storage?

```
\begin{array}{cccc}
\text{a} & \text{b} & \text{c} & \text{d} \\
\text{a} & \text{b} & \text{c} & \text{e} \\
\end{array}
```
BFS Properties

- What is the running time?
  - Hint: How many times will a node be enqueued?
- After the end of the algorithm, \( d[v] \) = shortest distance from \( s \) to \( v \)
  - \( d[v] \) = solves unweighted shortest paths
  - Can print the path from \( s \) to \( v \) by recursively following \( \pi[v], \pi[\pi[v]] \), etc.
- If \( d[v] = \infty \), then \( v \) not reachable from \( s \)
  - \( d[v] \) = solves reachability

Depth-First Search (DFS)

- Another graph traversal algorithm
  - Unlike BFS, this one follows a path as deep as possible before backtracking
  - Where BFS is “queue-like,” DFS is “stack-like”
- Tracks both “discovery time” and “finishing time” of each node, which will come in handy later

DFS(G)

\[
\begin{align*}
1 & \text{for each vertex } u \in V \text{ do} \\
2 & \quad \text{color}[u] = \text{WHITE} \\
3 & \quad \pi[u] = \text{NIL} \\
4 & \text{end} \\
5 & \text{time} = 0 \\
6 & \text{for each vertex } u \in V \text{ do} \\
7 & \quad \text{if color}[u] == \text{WHITE then} \\
8 & \quad \quad \text{DFS-Visit}(u) \\
9 & \text{end}
\end{align*}
\]

DFS-Visit(u)

\[
\begin{align*}
1 & \text{color}[u] = \text{GRAY} \\
2 & \text{time} = \text{time} + 1 \\
3 & \text{d}[u] = \text{time} \\
4 & \text{for each } v \in \text{Adj}[u] \text{ do} \\
5 & \quad \text{if color}[v] == \text{WHITE then} \\
6 & \quad \quad \pi[v] = u \\
7 & \quad \quad \text{DFS-Visit}(v) \\
8 & \text{end} \\
9 & \text{color}[u] = \text{BLACK} \\
10 & \text{f}[u] = \text{time} = \text{time} + 1
\end{align*}
\]
DFS Properties

- Time complexity same as BFS: $\Theta(|V| + |E|)$
- Vertex $u$ is a proper descendant of vertex $v$ in the DF tree iff $d[v] < d[u] < f[u] < f[v]$
  - Parenthesis structure: If one prints "(u" when discovering $u$ and "u)" when finishing $u$, then printed text will be a well-formed parenthesized sentence.

 DFS Properties (2)

- Classification of edges into groups
  - A tree edge is one in the depth-first forest
  - A back edge $(u, v)$ connects a vertex $u$ to its ancestor $v$ in the DF tree (includes self-loops)
  - A forward edge is a non-tree edge connecting a node to one of its DF tree descendants
  - A cross edge goes between non-ancestral edges within a DF tree or between DF trees
  - See labels in DFS example
- Example use of this property: A graph has a cycle iff DFS discovers a back edge (application: deadlock detection)
- When DFS first explores an edge $(u, v)$, look at $v$'s color:
  - color$[v]$ = WHITE implies tree edge
  - color$[v]$ = GRAY implies back edge
  - color$[v]$ = BLACK implies forward or cross edge

Application: Topological Sort

A directed acyclic graph (dag) can represent precedences: an edge $(x, y)$ implies that event/activity $x$ must occur before $y$.

Application: Topological Sort (2)

A topological sort of a dag $G$ is a linear ordering of its vertices such that if $G$ contains an edge $(u, v)$, then $u$ appears before $v$ in the ordering.

Topological Sort Algorithm

1. Call DFS algorithm on dag $G$
2. As each vertex is finished, insert it to the front of a linked list
3. Return the linked list of vertices

- Thus topological sort is a descending sort of vertices based on DFS finishing times
- Why does it work?
  - When a node is finished, it has no unexplored outgoing edges; i.e. all its descendant nodes are already finished and inserted at later spot in final sort

Application: Strongly Connected Components

Given a directed graph $G = (V, E)$, a strongly connected component (SCC) of $G$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, $u$ is reachable from $v$ and $v$ is reachable from $u$.

What are the SCCs of the above graph?
Transpose Graph

- Our algorithm for finding SCCs of $G$ depends on the transpose of $G$, denoted $G^T$.
- $G^T$ is simply $G$ with edges reversed.
- Fact: $G^T$ and $G$ have same SCCs. Why?

SCC Algorithm

1. Call DFS algorithm on $G$.
2. Compute $G^T$.
3. Call DFS algorithm on $G^T$, looping through vertices in order of decreasing finishing times from first DFS call.
4. Each DFS tree in second DFS run is an SCC in $G$.

SCC Algorithm Example

After first round of DFS:

Which node is first one to be visited in second DFS?

SCC Algorithm Example (2)

After second round of DFS:

SCC Algorithm Analysis

- What is its time complexity?
- How does it work?
  1. Let $x$ be node with highest finishing time in first DFS.
  2. In $G^T$, $x$'s component $C$ has no edges to any other component (Lemma 22.14), so the second DFS's tree edges define exactly $x$'s component.
  3. Now let $x'$ be the next node explored in a new component $C'$.
  4. The only edges from $C'$ to another component are to nodes in $C$, so the DFS tree edges define exactly the component for $x'$.
  5. And so on...