Graphs

- **Graph** $G = (V, E)$
  - $V =$ set of vertices
  - $E =$ set of edges $\subseteq (V \times V)$

- Types of graphs
  - **Undirected:** edge $(u, v) = (v, u)$; for all $v, (v, v) \not\in E$ (No self loops.)
  - **Directed:** $(u, v)$ is edge from $u$ to $v$, denoted as $u \rightarrow v$. Self loops are allowed.
  - **Weighted:** each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbb{R}$.
  - Dense: $|E| \approx |V|^2$.
  - Sparse: $|E| << |V|^2$.
  - $|E| = O(|V|^2)$

Graphs

- If $(u, v) \in E$, then vertex $v$ is adjacent to vertex $u$.
- Adjacency relationship is:
  - Symmetric if $G$ is undirected.
  - Not necessarily so if $G$ is directed.
- If $G$ is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then $G$ is a tree.
- Other definitions in Appendix B (B.4 and B.5) as needed.
Adjacency Lists

- Consists of an array $Adj$ of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to $u$.

If weighted, store weights also in adjacency lists.

Pros and Cons: adj list

- Pros
  - Space-efficient, when a graph is sparse.
  - Can be modified to support many graph variants.

- Cons
  - Determining if an edge $(u, v) \in G$ is not efficient.
    - Have to search in $u$’s adjacency list. $\Theta(\text{degree}(u))$ time.
    - $\Theta(V)$ in the worst case.

Adjacency Matrix

- $|V| \times |V|$ matrix $A$.
- Number vertices from 1 to $|V|$ in some arbitrary manner.
- $A$ is then given by:
  $$A[i, j] = a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

$A = A^T$ for undirected graphs.
Space and Time

- **Space:** $\Theta(V^2)$.
  - Not memory efficient for large graphs.
- **Time:** to list all vertices adjacent to $u$: $\Theta(V)$.
- **Time:** to determine if $(u, v) \in E$: $\Theta(1)$.
- Can store weights instead of bits for weighted graph.

### Some graph operations

<table>
<thead>
<tr>
<th>adjacency matrix</th>
<th>adjacency lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertEdge</td>
<td>O(1)</td>
</tr>
<tr>
<td>isEdge</td>
<td>O(1)</td>
</tr>
<tr>
<td>#successors?</td>
<td>O(V)</td>
</tr>
<tr>
<td>#predecessors?</td>
<td>O(V)</td>
</tr>
</tbody>
</table>

### traversing a graph

**Where to start?**
Will all vertices be visited?
How to prevent multiple visits?

### Graph Definitions

- **Path**
  - Sequence of nodes $n_1, n_2, \ldots n_k$
  - Edge exists between each pair of nodes $n_i, n_{i+1}$
  - Example
    - A, B, C is a path
Graph Definitions

- **Path**
  - Sequence of nodes $n_1, n_2, \ldots, n_k$
  - Edge exists between each pair of nodes $n_i, n_{i+1}$
  - Example
    - A, B, C is a path
    - A, E, D is not a path

- **Cycle**
  - Path that ends back at starting node
  - Example
    - A, E, A

- **Simple path**
  - No cycles in path

- **Acyclic graph**
  - No cycles in graph

Connected graph

Every node is reachable from some node in graph

Reachable

Path exists between nodes

Unconnected graphs
Graph-searching Algorithms

- Searching a graph:
  » Systematically follow the edges of a graph to visit the vertices of the graph.
- Used to discover the structure of a graph.
- Standard graph-searching algorithms.
  » Breadth-first Search (BFS).
  » Depth-first Search (DFS).

Breadth-first Search

- Input: Graph $G = (V, E)$, either directed or undirected, and source vertex $s \in V$.
- Output:
  » $d[v] =$ distance (smallest # of edges, or shortest path) from $s$ to $v$, for all $v \in V$. $d[v] = \infty$ if $v$ is not reachable from $s$.
  » $\pi[v] = u$ such that $(u, v)$ is last edge on shortest path $s \rightleftharpoons v$.
    • $u$ is $v$’s predecessor.
  » Builds breadth-first tree with root $s$ that contains all reachable vertices.

Breadth-first Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  » A vertex is “discovered” the first time it is encountered during the search.
  » A vertex is “finished” if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  » White – Undiscovered.
  » Gray – Discovered but not finished.
  » Black – Finished.

BFS for Shortest Paths

- Finished
- Discovered
- Undiscovered
BFS(G, s)
1. for each vertex u in V[G] – {s}
   do color[u] ← white
2. d[u] ← ∞
3. π[u] ← nil
5. color[s] ← gray
6. d[s] ← 0
7. π[s] ← nil
8. Q ← Φ
9. enqueue(Q, s)
10. while Q ≠ Φ
11. do u ← dequeue(Q)
12. for each v in Adj[u]
13. do if color[v] = white
14. then color[v] ← gray
15. d[v] ← d[u] + 1
16. π[v] ← u
17. enqueue(Q, v)
18. color[u] ← black
Example (BFS)

Q: y

Example (BFS)

Q: ∅

Example (BFS)

Analysis of BFS

- Initialization takes $O(|V|)$.
- Traversal Loop
  - After initialization, each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(|V|)$.
  - The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is $\Theta(|E|)$.
- Summing up over all vertices $\Rightarrow$ total running time of BFS is $O(|V| + |E|)$, linear in the size of the adjacency list representation of graph.
**Breadth-first Tree**

- For a graph \( G = (V, E) \) with source \( s \), the **predecessor subgraph** of \( G \) is \( G_{\pi} = (V_{\pi}, E_{\pi}) \) where
  - \( V_{\pi} = \{ v \in V : \pi[v] \neq nil \} \cup \{ s \} \)
  - \( E_{\pi} = \{ (\pi[v], v) : v \in V_{\pi} - \{ s \} \} \)
- The predecessor subgraph \( G_{\pi} \) is a **breadth-first tree** if:
  - \( V_{\pi} \) consists of the vertices reachable from \( s \) and
  - for all \( v \in V_{\pi} \), there is a unique simple path from \( s \) to \( v \) in \( G_{\pi} \) that is also a shortest path from \( s \) to \( v \) in \( G \).
- The edges in \( E_{\pi} \) are called **tree edges**.
  \[ |E_{\pi}| = |V_{\pi}| - 1. \]

**Depth-first Search (DFS)**

- Explore edges out of the most recently discovered vertex \( v \).
- When all edges of \( v \) have been explored, backtrack to explore other edges leaving the vertex from which \( v \) was discovered (its **predecessor**).
- “Search as deep as possible first.”
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

**Depth-first Search**

- **Input:** \( G = (V, E) \), directed or undirected. No source vertex given!
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and 2|V|.
    - \( d[v] = \text{discovery time} \) (\( v \) turns from white to gray)
    - \( f[v] = \text{finishing time} \) (\( v \) turns from gray to black)
  - \( \pi[v] : \text{predecessor of } v = u \), such that \( v \) was discovered during the scan of \( u \)'s adjacency list.
- Coloring scheme for vertices as BFS. A vertex is
  - “discovered” the first time it is encountered during the search.
  - A vertex is “finished” if it is a leaf node or all vertices adjacent to it have been finished.

**Pseudo-code**

**DFS(G)**
1. for each vertex \( u \in V[G] \)
2.   do color[\( u \)] \( \leftarrow \) white
3.   \( \pi[u] \) \( \leftarrow \) NIL
4.   \( d[u] \) \( \leftarrow \) 0
5.   \( f[u] \) \( \leftarrow \) 0
6. for each vertex \( v \in V[G] \)
7.   do if color[\( u \)] = white
8.     then DFS-Visit(u)

**DFS-Visit(u)**
1. color[\( u \)] \( \leftarrow \) GRAY // White vertex \( u \) has been discovered
2. \( time \) \( \leftarrow \) time + 1
3. \( d[u] \) \( \leftarrow \) time
4. for each \( v \in Adj[u] \)
5.   do if color[\( v \)] = WHITE
6.     then \( \pi[v] \) \( \leftarrow \) u
7. DFS-Visit(v)
8. color[\( u \)] \( \leftarrow \) BLACK // Blacken \( u \); it is finished.
9. \( f[u] \) \( \leftarrow \) time \( \leftarrow \) time + 1

Uses a global timestamp \( \text{time} \).
Example (DFS)

1. u → v → w
2. x → y → z
3. Example (DFS)
4. u → v → w
5. x → y → z
6. Example (DFS)
7. u → v → w
8. x → y → z
9. Example (DFS)
10. u → v → w
11. x → y → z
12. Example (DFS)
Example (DFS)
Analysis of DFS

- Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each white vertex $v \in V$ when it’s painted gray the first time. Lines 3-6 of DFS-Visit is executed $|\text{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$
- Total running time of DFS is $\Theta(|V| + |E|)$.

Recursive DFS Algorithm

Traverse( )
for all nodes $X$
set $X$.tag = False
Visit (1st node)
Visit ($X$)
for each successor $Y$ of $X$
if ($Y$.tag = False)
Visit ($Y$)

Parenthesis Theorem

Theorem 22.7
For all $u, v$, exactly one of the following holds:
2. $d[u] < d[v] < f[v] < f[u]$ and $v$ is a descendant of $u$.

- Like parentheses:
  - OK: $() [ ] ( [ ] ) ( )$
  - Not OK: $() [] [ ]$
  
Corollary
$v$ is a proper descendant of $u$ if and only if $d[u] < d[v] < f[v] < f[u]$.

Example (Parenthesis Theorem)
**Depth-First Trees**

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G_\pi = (V, E_\pi)$ where $E_\pi = \{(\pi[v], v) : v \in V$ and $\pi[v] \neq \text{nil}\}$.

> How does it differ from that of BFS?

> The predecessor subgraph $G_\pi$ forms a *depth-first forest* composed of several *depth-first trees*. The edges in $E_\pi$ are called *tree edges*.

**Definition:**
Forest: An acyclic graph $G$ that may be disconnected.

---

**Classification of Edges**

- **Tree edge**: in the depth-first forest. Found by exploring $(u, v)$.
- **Back edge**: $(u, v)$, where $u$ is a descendant of $v$ (in the depth-first tree).
- **Forward edge**: $(u, v)$, where $v$ is a descendant of $u$, but not a tree edge.
- **Cross edge**: any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

**Theorem:**
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

---

**White-path Theorem**

**Theorem 22.9**
$v$ is a descendant of $u$ in *DF-tree* if and only if at time $d[u]$, there is a path $u \sim v$ consisting of only white vertices. (Except for $u$, which was *just* colored gray.)

---

**Classifying edges of a digraph**

- $(u, v)$ is:
  - Tree edge – if $v$ is white
  - Back edge – if $v$ is gray
  - Forward or cross - if $v$ is black

- $(u, v)$ is:
  - Forward edge – if $v$ is black and $d[u] < d[v]$ ($v$ was discovered after $u$)
  - Cross edge – if $v$ is black and $d[u] > d[v]$ ($u$ discovered after $v$)
More applications

- **Does directed G contain a directed cycle?** Do DFS if back edges yes. Time O(V+E).
- **Does undirected G contain a cycle?** Same as directed but be careful not to consider (u,v) and (v,u) a cycle. Time O(V) since encounter at most |V| edges (if (u, v) and (v, u) are counted as one edge), before cycle is found.
- **Is undirected G a tree?** Do dfsVisit(v). If all vertices are reached and no back edges G is a tree. O(V)

---

**C# Interfaces**

```csharp
using System;
using System.Collections.Generic;

namespace OhioState.Collections.Graph {
    /// <summary>
    /// Edge provides a standard interface to specify an edge and any data associated with an edge within a graph.
    /// </summary>
    /// <typeparam name="N">The type of the nodes in the graph.</typeparam>
    /// <typeparam name="E">The type of the data on an edge.</typeparam>
    public interface IEdge<N, E> {
        /// <summary>
        /// Gets the Node label that this edge emanates from.
        /// </summary>
        /// <returns>The edge.</returns>
        E GetEdgeLabel(N fromNode, N toNode);
        /// <summary>
        /// Exception safe routine to get the label on an edge.
        /// </summary>
        /// <param name="fromNode">The node that the edge emanates from.</param>
        /// <param name="toNode">The node that the edge terminates at.</param>
        /// <returns>The edge.</returns>
        E TryGetEdgeLabel(N fromNode, N toNode, out E edge);
        /// <summary>
        /// Gets the number of edges in the graph.
        /// </summary>
        /// <returns>The number of edges in the graph.</returns>
        int NumberOfEdges { get; }
    }
}
```

---

**C# Interfaces**

```csharp
using System;
using System.Collections.Generic;

namespace OhioState.Collections.Graph {
    /// <summary>
    /// IGraph provides a standard interface to specify a graph.
    /// </summary>
    public interface IGraph<N, E> {
        /// <summary>
        /// Iterator for the nodes in the graph.
        /// </summary>
        /// <returns>An enumerable of nodes.</returns>
        IEnumerable<N> Nodes { get; }
        /// <summary>
        /// Iterator over the parents or immediate ancestors of a node.
        /// </summary>
        /// <returns>An enumerable of nodes.</returns>
        IEnumerable<N> ParentNodes(N node);
        /// <summary>
        /// Iterator for the edges in the graph, yielding IEdge's.
        /// </summary>
        /// <returns>An enumerable of edges.</returns>
        IEnumerable<IEdge<N, E>> Edges { get; }
        /// <summary>
        /// Tests whether an edge exists between two nodes.
        /// </summary>
        /// <param name="fromNode">The node that the edge emanates from.</param>
        /// <param name="toNode">The node that the edge terminates at.</param>
        /// <returns>True if the edge exists, False otherwise.</returns>
        bool ContainsEdge(N fromNode, N toNode);
        /// <summary>
        /// Gets the number of nodes in the graph.
        /// </summary>
        /// <returns>The number of nodes in the graph.</returns>
        int NumberOfNodes { get; }
    }
}
```

---

**C# Interfaces**

```csharp
using System;
using System.Collections.Generic;

namespace OhioState.Collections.Graph {
    /// <summary>
    /// IFiniteGraph provides a standard interface to specify a finite graph.
    /// </summary>
    public interface IFiniteGraph<N, E> : IGraph<N, E> {
        /// <summary>
        /// Gets the label on an edge.
        /// </summary>
        /// <param name="fromNode">The node that the edge emanates from.</param>
        /// <param name="toNode">The node that the edge terminates at.</param>
        /// <returns>The label on the edge.</returns>
        E GetEdgeLabel(N fromNode, N toNode);
        /// <summary>
        /// Exception safe routine to get the label on an edge.
        /// </summary>
        /// <param name="fromNode">The node that the edge emanates from.</param>
        /// <param name="toNode">The node that the edge terminates at.</param>
        /// <returns>The label on the edge.</returns>
        E TryGetEdgeLabel(N fromNode, N toNode, out E edge);
        /// <summary>
        /// Gets the number of edges in the graph.
        /// </summary>
        /// <returns>The number of edges in the graph.</returns>
        int NumberOfEdges { get; }
        /// <summary>
        /// Gets the number of nodes in the graph.
        /// </summary>
        /// <returns>The number of nodes in the graph.</returns>
        int NumberOfNodes { get; }
    }
}
```