A graph consists of:

- A set of nodes or vertices (terms are interchangeable)
- A set of edges or arcs (terms are interchangeable)

Edges in graph can be either directed or undirected
15-1: Graphs & Edges

- Edges can be labeled or unlabeled
- Edge labels are typically the cost associated with an edge
- e.g., Nodes are cities, edges are roads between cities, edge label is the length of road
15-2: Graph Representations

- Adjacency Matrix
  - Represent a graph with a two-dimensional array $G$
    - $G[i][j] = 1$ if there is an edge from node $i$ to node $j$
    - $G[i][j] = 0$ if there is no edge from node $i$ to node $j$
  - If graph is undirected, matrix is symmetric
  - Can represent edges labeled with a cost as well:
    - $G[i][j] = \text{cost of link between } i \text{ and } j$
    - If there is no direct link, $G[i][j] = \infty$
15-3: Adjacency Matrix

- Examples:

```
0 1 2 3
0 0 1 0 1
1 1 0 1 1
2 0 1 0 0
3 1 1 0 0
```
15-4: Adjacency Matrix

Examples:

\[
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
2 & 0 & 0 & 0 & 0 \\
3 & 1 & 0 & 0 & 0 \\
\end{array}
\]
15-5: Adjacency Matrix

- Examples:

\[
\begin{pmatrix}
0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\]
Examples:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>-2</td>
<td>∞</td>
</tr>
</tbody>
</table>
Adjacency List

- Maintain a linked-list of the neighbors of every vertex.
- Array of $n$ lists, one per vertex
- Each list $i$ contains a list of all vertices adjacent to $i$. 
15-8: Adjacency List

• Examples:
15-9: Adjacency List

- Examples:

- Note – lists are not always sorted
Sparse vs. Dense

• Sparse graph – relatively few edges
• Dense graph – lots of edges
• Complete graph – contains all possible edges
  • These terms are fuzzy. “Sparse” in one context may or may not be “sparse” in a different context
Breadth-First Search

- Method for searching a graph
- Specify a source node in the graph
- Find all nodes reachable from that node
  - First find all nodes 1 unit away
  - Next find all nodes 2 units away
  - ... etc
Breadth-First Search

- Auxiliary Data Structures
  - “color” for each vertex – white, black, grey
    - Used to make sure we don’t visit vertices more than once
  - Parent of each vertex (Path to source node)
  - Distance of each vertex from source
BFS($G$, $s$)
for each vertex $u$ in $V[G]$ do
    color[$u$] $\leftarrow$ WHITE
    $d[u] \leftarrow \infty$
    $\pi[u] \leftarrow$ nil
color[$s$] $\leftarrow$ GRAY
$d[s] \leftarrow 0$
$Q \leftarrow \{s\}$
while $Q$ not empty do
    $u \leftarrow Q$.dequeue
    for each $v$ adj. to $u$
        if color[$v$] = WHITE
            color[$v$] $\leftarrow$ GRAY
            $d[v] \leftarrow d[u] + 1$
            $\pi[v] \leftarrow u$
            $Q$.enqueue($v$)
    color[$u$] $\leftarrow$ BLACK
15-14: Breadth-First Search

Q: b
Breadth-First Search

- BFS computes the shortest path from the start vertex to every other vertex
- We can run BFS on a directed or undirected tree
- Defines a “BFS Tree”
  - Parent pointers $p[v]
  - BFS Tree is directed
Breadth-First Search

Q:
Breadth-First Search

- BFS Running time:
  - \( V \) vertices
  - \( E \) edges
Breadth-First Search

• BFS Running time:
  • $V$ vertices
  • $E$ edges

• Running time $\Theta(V + E)$
  • In terms of just $V$, $O(V^2)$ (why?)
DFS(\(G\))

for each vertex \(v\) in \(G\) do
    color[\(v\)] \(\leftarrow\) WHITE
    \(\pi[\(v\)] = \text{nil}\)
    time \(\leftarrow 0\)

for each vertex \(v\) in \(G\) do
    if color[\(v\)] = WHITE
        DFS-VISIT(\(v\))
DFS-VISIT($v$, $G$)

\begin{align*}
&\text{color}[^v] \leftarrow \text{GRAY} \\
&\text{time} \leftarrow \text{time} + 1 \\
&d[^v] \leftarrow \text{time} \\
&\text{for each } u \text{ adjacent to } v \text{ in } G \text{ do} \\
&\quad \text{if color}[^u] = \text{WHITE} \text{ then} \\
&\quad \quad \pi[^u] \leftarrow v \\
&\quad \text{DFS-VISIT}[^u, G] \\
&\text{color}[^v] \leftarrow \text{BLACK} \\
&\text{time} \leftarrow \text{time} + 1 \\
&f[^v] \leftarrow \text{time}
\end{align*}
Depth-First Search

(Do DFS, show discover/finish times & Depth First Forest)
15-22: **Depth-First Search**

- DFS creates a Depth First Forest
- We can use DFS to classify edges:
  - Tree edges
    - edges in the Depth First Forest
  - Back Edges
    - edge \((u, v)\) that connects \(u\) to ancestor \(v\) in DFF
  - Forward edges
    - non-tree edge \((u, v)\) that connects \(u\) to descendent \(v\) in DFF
  - Cross Edges
    - Everything Else
Labeling edges

How could we label edges (tree/back/forward/cross) while we are doing DFS?
• Labeling edges

• How could we label edges (tree/back/forward/cross) while we are doing DFS?

• When examining edge \((u, v)\), if \(v\) is:
  • WHITE – tree edge
  • GRAY – back edge
  • BLACK – forward edge or cross edge
15-25: Depth-First Search

- Labeling edges
  - Can we have cross edges in a DFS of an undirected graph?
  - Can we have forward edges in a DFS of an undirected graph?
Depth-First Search

(Do DFS, show discover/finish times & Depth First Forest)
15-27: Depth-First Search

(Do DFS, show discover/finish times & Depth First Forest)
Topological Sort

- Directed Acyclic Graph, Vertices $v_1 \ldots v_n$
- Create an ordering of the vertices
  - If there a path from $v_i$ to $v_j$, then $v_i$ appears before $v_j$ in the ordering
- Example: Prerequisite chains
How could we use DFS to do a Topological Sort?

(Hint – Use discover and/or finish times)
How could we use DFS to do a Topological Sort?

(Hint – Use discover and/or finish times)

(What does it mean if node $x$ finished before node $y$?)
How could we use DFS to do a Topological Sort?

- Do DFS, computing finishing times for each vertex
- As each vertex is finished, add to front of a linked list
- This list is a valid topological sort
Second method for doing topological sort:

Which node(s) could be first in the topological ordering?

Node(s) with no incident (incoming) edges
Topological Sort

- Pick a node $v_k$ with no incident edges
- Add $v_k$ to the ordering
- Remove $v_k$ and all edges from $v_k$ from the graph
- Repeat until all nodes are picked.
• How can we find a node with no incident edges?
• Count the incident edges of all nodes
15-35: Topological Sort

for (i=0; i < NumberOfVertices; i++)
    NumIncident[i] = 0;

for(i=0; i < NumberOfVertices; i++)
    each node k adjacent to i
    NumIncident[k]++
for(i=0; i < NumberOfVertices; i++)
    NumIncident[i] = 0;

for(i=0; i < NumberOfVertices; i++)
    for(tmp=G[i]; tmp != null; tmp=tmp.next())
        NumIncident[tmp.neighbor()]++
Topological Sort

- Create NumIncident array
- Repeat
  - Search through NumIncident to find a vertex $v$ with $\text{NumIncident}[v] == 0$
  - Add $v$ to the ordering
  - Decrement NumIncident of all neighbors of $v$
- Set NumIncident[$v$] = -1
- Until all vertices have been picked
In a graph with $V$ vertices and $E$ edges, how long does this version of topological sort take?
In a graph with $V$ vertices and $E$ edges, how long does this version of topological sort take?

- $\Theta(V^2 + E) = \Theta(V^2)$
- Since $E \in O(V^2)$
Topological Sort

- Where are we spending “extra” time
Topological Sort

- Where are we spending “extra” time
  - Searching through NumIncident each time looking for a vertex with no incident edges
- Keep around a set of all nodes with no incident edges
- Remove an element $v$ from this set, and add it to the ordering
- Decrement NumIncident for all neighbors of $v$
  - If NumIncident[$k$] is decremented to 0, add $k$ to the set.
- How do we implement the set of nodes with no incident edges?
Topological Sort

- Where are we spending “extra” time
  - Searching through NumIncident each time looking for a vertex with no incident edges
- Keep around a set of all nodes with no incident edges
- Remove an element $v$ from this set, and add it to the ordering
  - If NumIncident[$k$] is decremented to 0, add $k$ to the set.
- How do we implement the set of nodes with no incident edges?
  - Use a stack
15-43: **Topological Sort**

- Examples!!
  - Graph
  - Adjacency List
  - NumIncident
  - Stack
More DFS Applications

- Depth First Search can be used to calculate the connected components of a directed graph
- First, some definitions and examples:
Strongly Connected Graph

- Directed Path from every node to every other node

- Strongly Connected

Diagram:

- Nodes: 1, 2, 3, 4, 5
- Edges: 1 to 4, 2 to 3, 4 to 5
**Strongly Connected Graph**

- Directed Path from every node to every other node

![Graph Diagram]

- Strongly Connected
Connected Components

- Subgraph (subset of the vertices) that is strongly connected.
**15-48: Connected Components**

- Subgraph (subset of the vertices) that is strongly connected.
Connected Components

- Subgraph (subset of the vertices) that is strongly connected.
Connected Components

- Subgraph (subset of the vertices) that is strongly connected.
Connected components of the graph are the largest possible strongly connected subgraphs.

If we put each vertex in its own component – each component would be (trivially) strongly connected. Those would not be the connected components of the graph – unless there were no larger connected subgraphs.
Connected Components

- Calculating Connected Components
  - Two vertices $v_1$ and $v_2$ are in the same connected component if and only if:
    - Directed path from $v_1$ to $v_2$
    - Directed path from $v_2$ to $v_1$
  - To find connected components – find directed paths
    - Use DFS: $d[v]$ and $f[v]$
Recall that we calculate the order in which we visit the elements in a Depth-First Search.

For any vertex $v$ in a DFS:

- $d[v] = \text{Discovery time} – \text{when the vertex is first visited}$
- $f[v] = \text{Finishing time} – \text{when we have finished with a vertex (and all of its children)}$
DFS Example

1 → 2
3
4 ← 6
5
7
8
15-55: DFS Example

d
f

1 → 2

d
f

3 → 4

d
f

5 → 6

d
f

7 → 8

d
f
15-56: DFS Example
15-57: DFS Example

- Node 1 with label d1f
- Node 2 with label df
- Node 3 with label df
- Node 4 with label df
- Node 5 with label df
- Node 6 with label df
- Node 7 with label df
- Node 8 with label df
DFS Example
15-59: DFS Example

Diagram showing a depth-first search example with nodes labeled 1 to 8 and directed edges.
15-60: DFS Example

- Node 1: d1f
- Node 2: d2f
- Node 3: d3f
- Node 4: d4f
- Node 5: d5f
- Node 6: d6f
- Node 7: d7f
- Node 8: d8f
15-61: DFS Example

- Node 1 to 2
- Node 3 to 4
- Node 5 to 6
- Node 7 to 8
- Node 4 to 5
- Node 6 to 4
15-62: DFS Example

1 → 2 → 4 → 6 → 8
2 → 4
3
4 → 5
5 → 6
6
7
8

Node relationships:
1 → 2
2 → 4
4 → 6
6 → 8
4 → 5
5 → 6

15-63: DFS Example

1 → 2 → 3 → 4 → 6 → 5 → 7 → 8
15-64: DFS Example

1 → 2 → 4 → 3 → 5 → 6 → 7 → 8
15-65: DFS Example

1 - 2 - 4 - 3 - 5 - 6 - 7 - 8
15-66: DFS Example

1 -> 2
2 -> 4
3 -> 4
4 -> 6
5 -> 6
6 -> 7
7 -> 8
8 -> 10
15-67: DFS Example

1 -> 2 -> 4
2 -> 1
3 -> 4
4 -> 3, 5
5 -> 6, 4
6 -> 5
7 -> 8
8 -> 7
9 -> 2
10 -> 1
11 -> 5
12 -> 7
15-68: DFS Example

1 → 2 → 4
2 → 3
3 → 4
4 → 6
5 → 6
6 → 8
7 → 8
8
15-69: DFS Example

1 → 2
2 → 4
4 → 3
4 ↔ 6
3 → 1
5 → 3
5 → 6
8 → 7
7 → 11
11 → 12
12 → 13

15-70: DFS Example

1 \rightarrow 2 \rightarrow 4 \leftarrow 3
3 \rightarrow 5
5 \rightarrow 6
6 \rightarrow 7
8 \rightarrow 14
14 \rightarrow 15
DFS Example

1  
2  
3  
4  
5  
6  
7  
8  

d 1 f 10
d 2 f 9
d 3 f 8
d 4 f 7
d 5 f 6
d 11 f 16
d 12 f 13
d 14 f 15
15-72: DFS Example
15-73: DFS Example

Diagram showing the order of visiting nodes using Depth-First Search (DFS) with nodes labeled as follows: 1, 2, 3, 4, 5, 6, 7, 8.
15-74: DFS Example
15-75: DFS Example
15-76: DFS Example

1 → 2
2 → 4
3 → 4
4 → 1
5 → 6
6 → 5
7 → 8
8 → 7
15-77: DFS Example

Diagram:
- Node 1 is connected to node 2.
- Node 2 is connected to node 3.
- Node 3 is connected to nodes 4 and 5.
- Node 4 is connected to nodes 1 and 3.
- Node 5 is connected to nodes 3 and 6.
- Node 6 is connected to node 5.
- Node 7 is connected to node 8.
- Node 8 is connected to node 7.

Nodes labeled with 'f' are visited before nodes labeled with 'd'.
15-78: DFS Example
15-79: DFS Example
15-80: DFS Example

Diagram showing a Depth-First Search (DFS) example with nodes labeled as follows:

- Node 1: d 1 f
- Node 2: d 2 f
- Node 3: d 4 f 7
- Node 4: d 3 f 8
- Node 5: d 5 f 6
- Node 6: d 5 f 6
- Node 7: d f
- Node 8: d f

The diagram illustrates the order in which nodes are visited during a DFS traversal.
DFS Example
15-82: DFS Example

Diagram showing a depth-first search example with nodes labeled from 1 to 8 and edges connecting them.
15-84: DFS Example

DFS Example Diagram:

1 -> 2
1 -> 3
3 -> 4
3 -> 5
5 -> 6
5 -> 7
6 -> 8
7 -> 8

Nodes and Edges:

- Node 1: (d 1, f 10)
- Node 2: (d 2, f 9)
- Node 3: (d 4, f 7)
- Node 4: (d 3, f 8)
- Node 5: (d 11, f)
- Node 6: (d 5, f 6)
- Node 7: (d f)
- Node 8: (d 12, f)
15-85: DFS Example

1 → 2 → 4
3 → 4 → 6
5 → 6
7 → 8
DFS Example

1

2

3

4

5

6

7

8

d 1 f 10
d 2 f 9
d 3 f 8
d 4 f 7
d 5 f 6
d 11 f
d 13 f 14

d 12 f
15-87: DFS Example

1 -> 2
1 -> 4
3 -> 2
3 -> 4
5 -> 3
6 -> 4
5 -> 6
7 -> 6
8 -> 7
10 -> 1
7 -> 4
11 -> 5
12 -> 8
13 -> 7
14 -> 8
9 -> 2
8 -> 4
6 -> 5
15 -> 6
13 -> 7
14 -> 8
15-88: DFS Example

1 -> 2

3 -> 1, 4

4 -> 2, 3

5 -> 3, 6

6 -> 4, 5

7 -> 5, 8

8 -> 6, 7
Given two vertices $v_1$ and $v_2$, what do we know if $f[v_2] < f[v_1]$?
Given two vertices $v_1$ and $v_2$, what do we know if $f[v_2] < f[v_1]$?

Either:
- Path from $v_1$ to $v_2$
- Start from $v_1$
- Eventually visit $v_2$
- Finish $v_2$
- Finish $v_1$
Given two vertices $v_1$ and $v_2$, what do we know if $f[v_2] < f[v_1]$?

Either:

- Path from $v_1$ to $v_2$
- No path from $v_2$ to $v_1$
  - Start from $v_2$
  - Eventually finish $v_2$
  - Start from $v_1$
  - Eventually finish $v_1$
If $f[v_2] < f[v_1]$:

- Either a path from $v_1$ to $v_2$, or no path from $v_2$ to $v_1$.
- If there is a path from $v_2$ to $v_1$, then there must be a path from $v_1$ to $v_2$.

$f[v_2] < f[v_1]$ and a path from $v_2$ to $v_1 \Rightarrow v_1$ and $v_2$ are in the same connected component.
Calculating paths

- Path from $v_2$ to $v_1$ in $G$ if and only if there is a path from $v_1$ to $v_2$ in $G^T$
  - $G^T$ is the transpose of $G - G$ with all edges reversed
- If after DFS, $f[v_2] < f[v_1]$
- Run second DFS on $G^T$, starting from $v_1$, and $v_1$ and $v_2$ are in the same DFS spanning tree
- $v_1$ and $v_2$ must be in the same connected component
• Run DFS on $G$, calculating $f[]$ times

• Compute $G^T$

• Run DFS on $G^T$ – examining nodes in *inverse order of finishing times* from first DFS

• Any nodes that are in the same DFS search tree in $G^T$ must be in the same connected component
Connected Components Eg.
Connected Components Eg.
Connected Components Eg.

1. Connected to 2 and 4
2. Connected to 1 and 9
3. Connected to 4
4. Connected to 2 and 9
5. Connected to 6 and 11
6. Connected to 4 and 5
7. Connected to 11
8. Connected to 12

Nodes 1, 3, 5, and 7 are part of the same connected component.
15-99: Connected Components Eg.

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8
15-100: Connected Components Eg.

- 1 connected to 2, 3
- 2 connected to 3, 4
- 3 connected to 4, 5
- 4 connected to 3, 5, 6
- 5 connected to 6, 7
- 6 connected to 5
- 7 connected to 6
- 8 connected to 7

1 connected to 10, 11
4 connected to 7, 5
5 connected to 11, 6
6 connected to 12, 5
7 connected to 13
8 connected to 12

D: Diana  
F: Frances
15-101: Connected Components Eg.
15-102: Connected Components Eg.