Directed Graphs (Digraphs) and Graphs
 Definitions  Graph ADT  Traversal algorithms  DFS

Lecturer: Georgy Gimel’farb

COMPSCI 220 Algorithms and Data Structures
1. Basic definitions

2. Digraph Representation and Data Structures

3. Digraph ADT Operations

4. Graph Traversals and Applications

5. Depth-first Search in Digraphs
Graphs in Life: World Air Routes

http://milenomics.com/2014/05/partners-alliances-partner-awards/
Graphs in Life: Global Internet Connections

http://www.opte.org/maps/
Graphs in Life: Social Networks (Facebook)

A **digraph** $G = (V, E)$ is a finite nonempty set $V$ of **nodes** together with a (possibly empty) set $E$ of ordered pairs of nodes of $G$ called **arcs**.

$V = \{ 0, 1, 2, 3, 4, 5, 6 \}$

$E = \{ (0, 1), (0, 3), (1, 2), (2, 0), (2, 5), (2, 6), (3, 1), (4, 0), (4, 3), (4, 5), (5, 3), (5, 6), (6, 5) \}$

$^o$ Set $E$ is a **neighbourhood**, or adjacency **relation** on $V$. 
If \((u, v) \in E\),

- \(v\) is **adjacent** to \(u\);
- \(v\) is an **out-neighbour** of \(u\), and
- \(u\) is an **in-neighbour** of \(v\).

**Examples:**

- Nodes (points) 1 and 3 are adjacent to 0.
- 1 and 3 are out-neighbours of 0.
- 0 is an in-neighbour of 1 and 3.
- Node 1 is adjacent to 3.
- 1 is an out-neighbour of 3.
- 3 is an in-neighbour of 1. . .
- 5 is an out-neighbour of 2, 4, and 6.
A graph \( G = (V, E) \) is a finite nonempty set \( V \) of vertices together with a (possibly empty) set \( E \) of unordered pairs of vertices of \( G \) called edges.

\[
V = \{a, b, c, d, e, f, g, h\}
\]

\[
E = \{\{a, b\}, \{a, d\}, \{b, d\}, \{b, c\}, \{c, d\}, \{d, f\}, \{d, h\}, \{f, h\}, \{e, g\}\}
\]

\( ^\circ \) The symmetric digraph: each arc \((u, v)\) has the opposite arc \((v, u)\).

Such a pair is reduced into a single undirected edge that can be traversed in either direction.
Order, Size, and In-/Out-degree

The **order** of a digraph $G = (V, E)$ is the number of nodes, $n = |V|$.

The **size** of a digraph $G = (V, E)$ is the number of arcs, $m = |E|$.

For a given $n$, Sparse digraphs: $|E| \in O(n)$ Dense digraphs: $|E| \in \Theta(n^2)$

The **in-degree** or **out-degree** of a node $v$ is the number of arcs entering or leaving $v$, respectively.

- A node of in-degree 0 – a **source**.
- A node of out-degree 0 – a **sink**.
- This example: the order $|V| = 6$ and the size $|E| = 9$. 

![Diagram of a digraph with order and size annotations](image-url)
Order, Size, and In- / Out-degree

The **order** of a digraph $G = (V, E)$ is the number of nodes, $n = |V|$.  

The **size** of a digraph $G = (V, E)$ is the number of arcs, $m = |E|$.  

For a given $n$,  

- **Sparse** digraphs: $|E| \in O(n)$  
- **Dense** digraphs: $|E| \in \Theta(n^2)$  

The **in-degree** or **out-degree** of a node $v$ is the number of arcs entering or leaving $v$, respectively.  

- A node of in-degree 0 – a **source**.  
- A node of out-degree 0 – a **sink**.  
- This example: the order $|V| = 6$ and the size $|E| = 9$.  

Source

Sink
Order, Size, and In- / Out-degree

The **order** of a digraph $G = (V, E)$ is the number of nodes, $n = |V|$.

The **size** of a digraph $G = (V, E)$ is the number of arcs, $m = |E|$.

For a given $n$, \[ m = 0 \quad \text{Sparse digraphs: } |E| \in O(n) \quad \text{Dense digraphs: } |E| \in \Theta(n^2) \quad n(n - 1) \]

The **in-degree** or **out-degree** of a node $v$ is the number of arcs entering or leaving $v$, respectively.

- A node of in-degree 0 – a **source**.
- A node of out-degree 0 – a **sink**.
- This example: the order $|V| = 6$ and the size $|E| = 9$. 

\[ \text{Source} \quad \text{Sink} \]
A walk in a digraph $G = (V, E)$:

a sequence of nodes $v_0 v_1 \ldots v_n$, such that $(v_i, v_{i+1})$ is an arc in $G$, i.e., $(v_i, v_{i+1}) \in E$, for each $i$; $0 \leq i < n$.

- The length of the walk $v_0 v_1 \ldots v_n$ is the number $n$ of arcs involved.
- A path is a walk, in which no node is repeated.
- A cycle is a walk, in which $v_0 = v_n$ and no other nodes are repeated.

- By convention, a cycle in a graph is of length at least 3.
- It is easily shown that if there is a walk from $u$ to $v$, then there is at least one path from $u$ to $v$. 
Walk, Path, and Cycle

A **walk** in a digraph $G = (V, E)$:

a sequence of nodes $v_0 v_1 \ldots v_n$, such that $(v_i, v_{i+1})$ is an arc in $G$, i.e., $(v_i, v_{i+1}) \in E$, for each $i; 0 \leq i < n$.

- The **length** of the walk $v_0 v_1 \ldots v_n$ is the number $n$ of arcs involved.
- A **path** is a walk, in which no node is repeated.
- A **cycle** is a walk, in which $v_0 = v_n$ and no other nodes are repeated.

- By convention, a cycle in a graph is of length at least 3.
- It is easily shown that if there is a walk from $u$ to $v$, then there is at least one path from $u$ to $v$. 
A walk in a digraph $G = (V, E)$:

a sequence of nodes $v_0 v_1 \ldots v_n$, such that $(v_i, v_{i+1})$ is an arc in $G$, i.e., $(v_i, v_{i+1}) \in E$, for each $i; 0 \leq i < n$.

- The length of the walk $v_0 v_1 \ldots v_n$ is the number $n$ of arcs involved.
- A path is a walk, in which no node is repeated.
- A cycle is a walk, in which $v_0 = v_n$ and no other nodes are repeated.

- By convention, a cycle in a graph is of length at least 3.
- It is easily shown that if there is a walk from $u$ to $v$, then there is at least one path from $u$ to $v$. 
Walks, Paths, and Cycles in a Digraph: an Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Walk?</th>
<th>Path?</th>
<th>Cycle?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3 1 2</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>1 2 6 5 3 1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>4 5 6 5</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>4 3 5</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Walks, Paths, and Cycles in a Digraph: an Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Walk?</th>
<th>Path?</th>
<th>Cycle?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3 1 2</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>1 2 6 5 3 1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>4 5 6 5</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>4 3 5</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Walks, Paths, and Cycles in a Digraph: an Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Walk?</th>
<th>Path?</th>
<th>Cycle?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3 1 2</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>1 2 6 5 3 1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>4 5 6 5</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>4 3 5</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Walks, Paths, and Cycles in a Digraph: an Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Walk?</th>
<th>Path?</th>
<th>Cycle?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3 1 2</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>1 2 6 5 3 1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>4 5 6 5</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>4 3 5</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Walks, Paths, and Cycles in a Digraph: an Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Walk?</th>
<th>Path?</th>
<th>Cycle?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3 1 2</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>1 2 6 5 3 1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>4 5 6 5</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>4 3 5</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Walks, Paths, and Cycles in a Digraph: an Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Walk?</th>
<th>Path?</th>
<th>Cycle?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3 1 2</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>1 2 6 5 3 1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>4 5 6 5</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>4 3 5</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Walks, Paths, and Cycles in a Graph: an Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Walk?</th>
<th>Path?</th>
<th>Cycle?</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>ege</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>dbcd</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>dadf</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>abdfh</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
Walks, Paths, and Cycles in a Graph: an Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Walk?</th>
<th>Path?</th>
<th>Cycle?</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>ege</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>dbcd</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>dadf</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>abdfh</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
Walks, Paths, and Cycles in a Graph: an Example

Sequence | Walk? | Path? | Cycle? |
---|---|---|---|
$abc$ | yes | yes | no |
$egc$ | yes | no | no |
$dbcd$ | yes | no | yes |
$dadf$ | yes | no | no |
$abdfh$ | yes | yes | no |
Walks, Paths, and Cycles in a Graph: an Example

**Sequence** | **Walk?** | **Path?** | **Cycle?**
---|---|---|---
\[abc\] | yes | yes | no
\[ege\] | yes | no | no
\[dbcd\] | yes | no | yes
\[dadf\] | yes | no | no
\[abdfh\] | yes | yes | no
Walks, Paths, and Cycles in a Graph: an Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Walk?</th>
<th>Path?</th>
<th>Cycle?</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>ege</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>dbcd</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>dadf</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>abdfh</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
Walks, Paths, and Cycles in a Graph: an Example

**Sequence** | **Walk?** | **Path?** | **Cycle?**
---|---|---|---
$abc$ | yes | yes | no
$ege$ | yes | no | no
$dbcd$ | yes | no | yes
$dadf$ | yes | no | no
$abdfh$ | yes | yes | no
Digraph $G = (V, E)$: Distances and Diameter

The distance, $d(u, v)$, from a node $u$ to a node $v$ in $G$ is the minimum length of a path from $u$ to $v$.

- If no path exists, the distance is undefined or $+\infty$.
- For graphs, $d(u, v) = d(v, u)$ for all vertices $u$ and $v$.

The diameter of $G$ is the maximum distance $\max_{u,v \in V} [d(u, v)]$ between any two vertices.

The radius of $G$ is $\min_{u \in V} \max_{v \in V} [d(u, v)]$. 
Digraph $G = (V, E)$: Distances and Diameter

The **distance**, $d(u, v)$, from a node $u$ to a node $v$ in $G$ is the *minimum* length of a path from $u$ to $v$.

- If no path exists, the distance is undefined or $+\infty$.
- For graphs, $d(u, v) = d(v, u)$ for all vertices $u$ and $v$.

The **diameter** of $G$ is the *maximum* distance $\max_{u,v \in V} [d(u, v)]$ between any two vertices.

The **radius** of $G$ is $\min_{u \in V} \max_{v \in V} [d(u, v)]$. 

Path Distances in Digraphs: Examples

\[ d(0, 3) = \min\{\text{length of } 0, 3; \text{length of } 0, 1, 2, 6, 5, 3; \text{length of } 0, 1, 2, 5, 3\} = \min\{1; 5; 4\} = 1 \]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
- & 1 & 2 & 1 & \infty & 3 & 3 \\
u=0 & 2 & - & 1 & 3 & \infty & 2 & 2 \\
u=1 & 1 & 3 & - & 2 & \infty & 1 & 1 \\
u=2 & 3 & 1 & 2 & - & \infty & 3 & 3 \\
u=3 & 1 & 2 & 3 & 1 & - & 1 & 2 \\
u=4 & 4 & 2 & 3 & 1 & \infty & - & 1 \\
u=5 & 5 & 3 & 4 & 2 & \infty & 1 & - \\
u=6 & & & & & & & \\
\end{array}
\]

\[ d(0, 1) = 1, \ d(0, 2) = 2, \ d(0, 5) = 3, \ d(0, 4) = \infty, \ d(5, 5) = 0, \ d(5, 2) = 3, \]
\[ d(5, 0) = 4, \ d(4, 6) = 2, \ d(4, 1) = 2, \ d(4, 2) = 3 \]

**Diameter:** \( \max\{1, 2, 1, \infty, 3, \ldots, 4, \ldots, 5, \ldots, 1\} = \infty \)

**Radius:** \( \min\{\infty, \infty, \ldots, 3, \infty, \infty\} = 3 \)
Path Distances in Graphs: Examples

![Graph Diagram]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>u=a</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>u=b</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>u=c</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>u=d</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>∞</td>
<td>1</td>
<td>∞</td>
<td>1</td>
</tr>
<tr>
<td>u=e</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>u=f</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>1</td>
</tr>
<tr>
<td>u=g</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>u=h</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>∞</td>
<td>1</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>

- $d(a, b) = d(b, a) = 1$, $d(a, c) = d(c, a) = 2$, $d(a, f) = d(f, a) = 2$, $d(a, e) = d(e, a) = \infty$, $d(e, e) = 0$, $d(e, g) = d(g, e) = 1$, $d(h, f) = d(f, h) = 1$, $d(d, h) = d(h, d) = 1$

- **Diameter:** $\max\{0, 1, 2, 1, \infty, 2, \ldots, 2, \ldots, 2, \ldots, 0\} = \infty$
- **Radius:** $\min\{\infty, \ldots, \infty\} = \infty$
Diameter / Radius of an Unweighted Graph

$d(C, E) = d(E, C)$

$= \min\{1, 1 + 1, 1 + 1, 1 + 1 + 1, 1 + 1 + 1 + 1\} = 1$

$d(B, C) = d(C, B)$

$= \min\{1 + 1, 1 + 1 + 1, 1 + 1, 1 + 1 + 1 + 1, 1 + 1, 1 + 1 + 1 + 1\} = 2$

Radius = 1; diameter = 2.
Diameter / Radius of a Weighted Graph

\[ d(C, E) = d(E, C) = \min\{5, 2 + 1, 3 + 1, 2 + 3 + 1, 3 + 2 + 1\} = 3 \]

\[ d(B, C) = d(C, B) = \min\{3 + 2, 1 + 1 + 2, 1 + 5, 1 + 1 + 3, 2 + 3, 2 + 1 + 5\} = 4 \]

Radius = 2; diameter = 4.
The underlying graph of a digraph $G = (V, E)$ is the graph $G' = (V, E')$ where $E' = \{ \{u, v\} \mid (u, v) \in E \}$. 
Sub(di)graphs

A subdigraph of a digraph $G = (V, E)$ is a digraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$.

$G = \left( \begin{array}{l} V = \{0, 1, 2, 3, 4\}, \\ E = \{(0, 2), (1, 0), (1, 2), \\ (1, 3), (3, 1), (4, 2), \\ (3, 4)\} \end{array} \right)$

$G' = \left( \begin{array}{l} V' = \{1, 2, 3\}, \\ E' = \{(1, 2), (3, 1)\} \end{array} \right)$
A \textit{spanning} subdigraph contains all nodes, that is, \(V' = V\).

\[
G = \left( V = \{0, 1, 2, 3, 4\}, \left\{ (0, 2), (1, 0), (1, 2), (1, 3), (3, 1), (4, 2), (3, 4) \right\} \right)
\]

\[
G' = \left( V' = \{0, 1, 2, 3, 4\}, \left\{ (0, 2), (1, 2), (3, 4) \right\} \right)
\]
Induced Sub(digraph)s

The subdigraph **induced** by a subset $V'$ of $V$ is the digraph $G' = (V', E')$ where $E' = \{(u, v) \in E \mid u \in V' \text{ and } v \in V'\}$.
Digraphs: Computer Representation

For a digraph $G$ of order $n$ with the vertices, $V$, labelled $0, 1, \ldots, n - 1$:

**The adjacency matrix of $G$:**

The $n \times n$ boolean matrix (often encoded with 0’s and 1’s) such that its entry $(i, j)$ is true if and only if there is an arc $(i, j)$ from the node $i$ to node $j$.

**An adjacency list of $G$:**

A sequence of $n$ sequences, $L_0, \ldots, L_{n-1}$, such that the sequence $L_i$ contains all nodes of $G$ that are adjacent to the node $i$.

Each sequence $L_i$ may not be sorted! But we usually sort them.
Digraphs: Computer Representation

For a digraph $G$ of order $n$ with the vertices, $V$, labelled $0, 1, \ldots, n-1$:

**The adjacency matrix of $G$:**

The $n \times n$ boolean matrix (often encoded with 0’s and 1’s) such that its entry $(i, j)$ is true if and only if there is an arc $(i, j)$ from the node $i$ to node $j$.

**An adjacency list of $G$:**

A sequence of $n$ sequences, $L_0, \ldots, L_{n-1}$, such that the sequence $L_i$ contains all nodes of $G$ that are adjacent to the node $i$.

Each sequence $L_i$ may not be sorted! But we usually sort them.
**Adjacency Matrix of a Digraph**

![Digraph Diagram](image)

The number of 1’s in a row (column) is the out-(in-) degree of the related node.

**Adjacency matrix of $G$:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

0 – a non-adjacent pair of vertices: 
$(i, j) \notin E$

1 – an adjacent pair of vertices: 
$(i, j) \in E$
Adjacency Lists of a Graph

Graph $G = (V, E)$

<table>
<thead>
<tr>
<th>symbolic</th>
<th>numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 = a: b\ d$</td>
<td>1\ 3</td>
</tr>
<tr>
<td>$1 = b: a\ c\ d$</td>
<td>0\ 2\ 3</td>
</tr>
<tr>
<td>$2 = c: b\ d$</td>
<td>1\ 3</td>
</tr>
<tr>
<td>$3 = d: a\ b\ c\ f\ h$</td>
<td>0\ 1\ 2\ 5\ 7</td>
</tr>
<tr>
<td>$4 = e: g$</td>
<td>6</td>
</tr>
<tr>
<td>$5 = f: d\ h$</td>
<td>3\ 7</td>
</tr>
<tr>
<td>$6 = g: e$</td>
<td>4</td>
</tr>
<tr>
<td>$7 = h: d\ f$</td>
<td>3\ 5</td>
</tr>
</tbody>
</table>

Special cases can be stored more efficiently:

- A complete binary tree or a heap: in an array.
- A general rooted tree: in an array $pred$ of size $n$;
  - $pred[i]$ – a pointer to the parent of node $i$. 
### Digraph Operations w.r.t. Data Structures

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>arc ((i, j)) exists?</td>
<td>is entry ((i, j)) 0 or 1</td>
<td>find (j) in list (i)</td>
</tr>
<tr>
<td>out-degree of (i)</td>
<td>scan row and sum 1’s</td>
<td>size of list (i)</td>
</tr>
<tr>
<td>in-degree of (i)</td>
<td>scan column and sum 1’s</td>
<td>for (j \neq i), find (i) in list (j)</td>
</tr>
<tr>
<td>add arc ((i, j))</td>
<td>change entry ((i, j))</td>
<td>insert (j) in list (i)</td>
</tr>
<tr>
<td>delete arc ((i, j))</td>
<td>change entry ((i, j))</td>
<td>delete (j) from list (i)</td>
</tr>
<tr>
<td>add node</td>
<td>create new row/column</td>
<td>add new list at end</td>
</tr>
<tr>
<td>delete node (i)</td>
<td>delete row/column (i) and shuffle other entries</td>
<td>delete list (i) and for (j \neq i), delete (i) from list (j)</td>
</tr>
</tbody>
</table>
Adjacency Lists / Matrices: Comparative Performance

\[ G = (V, E) \quad \rightarrow \quad n = |V|; \quad m = |E| \]

<table>
<thead>
<tr>
<th>Operation</th>
<th>array/array</th>
<th>list/list</th>
</tr>
</thead>
<tbody>
<tr>
<td>arc ((i, j)) exists?</td>
<td>(\Theta(1))</td>
<td>(\Theta(\alpha))</td>
</tr>
<tr>
<td>out-degree of (i)</td>
<td>(\Theta(n))</td>
<td>(\Theta(1))</td>
</tr>
<tr>
<td>in-degree of (i)</td>
<td>(\Theta(n))</td>
<td>(\Theta(n + m))</td>
</tr>
<tr>
<td>add arc ((i, j))</td>
<td>(\Theta(1))</td>
<td>(\Theta(1))</td>
</tr>
<tr>
<td>delete arc ((i, j))</td>
<td>(\Theta(1))</td>
<td>(\Theta(\alpha))</td>
</tr>
<tr>
<td>add node</td>
<td>(\Theta(n))</td>
<td>(\Theta(1))</td>
</tr>
<tr>
<td>delete node (i)</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n + m))</td>
</tr>
</tbody>
</table>

\(^\circ\) Here, \(\alpha\) denotes size of the adjacency list for vertex \(i\).
General Graph Traversal Algorithm (Part 1)

algorithm traverse
Input: digraph $G = (V, E)$
begin
array $colour[n]$, $pred[n]$
for $u \in V(G)$ do
  $colour[u] \leftarrow$ WHITE
end for
for $s \in V(G)$ do
  if $colour[s] = \text{WHITE}$ then
    visit($s$)
  end if
end for
return $pred$
end

Three types of nodes each stage:
- WHITE – unvisited yet.
- GREY – visited, but some adjacent nodes are WHITE.
- BLACK – visited; only GREY adjacent nodes
algorithm visit
   Input: node \( s \) of digraph \( G \)
begin
   colour\[s\] \leftarrow \text{GREY}; \ pred[\!s] \leftarrow \text{NULL}
while there is a grey node do
   choose a grey node \( u \)
   if there is a white neighbour of \( u \)
      choose such a neighbour \( v \)
      colour\[v\] \leftarrow \text{GREY}; \ pred[\!v] \leftarrow u
   else colour\[u\] \leftarrow \text{BLACK}
end if
end while
end
Illustrating the General Traversal Algorithm

initialising all nodes WHITE
Illustrating the General Traversal Algorithm

visit(a); colour[a] ← GREY

e is WHITE neighbour of a:

colour[e] ← GREY; pred[e] ← a
Illustrating the General Traversal Algorithm

- **visit(a)**; \(\text{colour}[a] \leftarrow \text{GREY}\)
- \(\text{e is WHITE neighbour of a}\)
  - \(\text{colour}[e] \leftarrow \text{GREY}; \text{pred}[e] \leftarrow \text{a}\)
- **choose GREY a**: no WHITE neighbour:
  - \(\text{colour}[a] \leftarrow \text{BLACK}\)
Illustrating the General Traversal Algorithm

visit(a); colour[a] ← GREY

e is WHITE neighbour of a

colour[e] ← GREY; pred[e] ← a

choose GREY a: no WHITE neighbour:

colour[a] ← BLACK

choose GREY e: no WHITE neighbour:

colour[e] ← BLACK
Illustrating the General Traversal Algorithm

visit(b); colour[b] ← GREY

c is WHITE neighbour of b

colour[c] ← GREY; pred[c] ← b
Illustrating the General Traversal Algorithm

- **visit(b); colour[b] ← GREY**
- c is WHITE neighbour of b
  - colour[c] ← GREY; pred[c] ← b
- d is WHITE neighbour of c
  - colour[d] ← GREY; pred[d] ← c
Illustrating the General Traversal Algorithm

- visit(b); colour[b] ← GREY
- c is WHITE neighbour of b
  - colour[c] ← GREY; pred[c] ← b
- d is WHITE neighbour of c
  - colour[d] ← GREY; pred[d] ← c
- no more WHITE nodes:
  - colour[d] ← BLACK
  - colour[c] ← BLACK
  - colour[b] ← BLACK
Classes of Traversal Arcs

**Search forest** $F$: a set of disjoint trees spanning a digraph $G$ after its traversal.

An arc $(u, v) \in E(G)$ is called a **tree arc** if it belongs to one of the trees of $F$.

The arc $(u, v)$, which is not a tree arc, is called:

- a **forward arc** if $u$ is an ancestor of $v$ in $F$;
- a **back arc** if $u$ is a descendant of $v$ in $F$, and
- a **cross arc** if neither $u$ nor $v$ is an ancestor of the other in $F$. 
Basic Facts about Traversal Trees (for further analyses)

Theorem 5.2: Suppose we have run traverse on $G$, resulting in a search forest $F$.

1. If $T_1$ and $T_2$ are different trees in $F$ and $T_1$ was explored before $T_2$, then there are no arcs from $T_1$ to $T_2$.

2. If $G$ is a graph, then there can be no edges joining different trees of $F$.

3. If $v, w \in V(G)$; $v$ is visited before $w$, and $w$ is reachable from $v$ in $G$, then $v$ and $w$ belong to the same tree of $F$.

4. If $v, w \in V(G)$ and $v$ and $w$ belong to the same tree $T$ in $F$, then any path from $v$ to $w$ in $G$ must have all nodes in $T$. 
Basic Facts about Traversal Trees (for further analyses)

Theorem 5.2: Suppose we have run \textit{traverse} on \( G \), resulting in a search forest \( F \).

1. If \( T_1 \) and \( T_2 \) are different trees in \( F \) and \( T_1 \) was explored before \( T_2 \), then there are no arcs from \( T_1 \) to \( T_2 \).
2. If \( G \) is a graph, then there can be no edges joining different trees of \( F \).
3. If \( v, w \in V(G) \); \( v \) is visited before \( w \), and \( w \) is reachable from \( v \) in \( G \), then \( v \) and \( w \) belong to the same tree of \( F \).
4. If \( v, w \in V(G) \) and \( v \) and \( w \) belong to the same tree \( T \) in \( F \), then any path from \( v \) to \( w \) in \( G \) must have all nodes in \( T \).
Basic Facts about Traversal Trees (for further analyses)

Theorem 5.2: Suppose we have run traverse on $G$, resulting in a search forest $F$.

1. If $T_1$ and $T_2$ are different trees in $F$ and $T_1$ was explored before $T_2$, then there are no arcs from $T_1$ to $T_2$.

2. If $G$ is a graph, then there can be no edges joining different trees of $F$.

3. If $v, w \in V(G)$; $v$ is visited before $w$, and $w$ is reachable from $v$ in $G$, then $v$ and $w$ belong to the same tree of $F$.

4. If $v, w \in V(G)$ and $v$ and $w$ belong to the same tree $T$ in $F$, then any path from $v$ to $w$ in $G$ must have all nodes in $T$. 
Basic Facts about Traversal Trees (for further analyses)

Theorem 5.2: Suppose we have run traverse on $G$, resulting in a search forest $F$.

1. If $T_1$ and $T_2$ are different trees in $F$ and $T_1$ was explored before $T_2$, then there are no arcs from $T_1$ to $T_2$.

2. If $G$ is a graph, then there can be no edges joining different trees of $F$.

3. If $v, w \in V(G)$; $v$ is visited before $w$, and $w$ is reachable from $v$ in $G$, then $v$ and $w$ belong to the same tree of $F$.

4. If $v, w \in V(G)$ and $v$ and $w$ belong to the same tree $T$ in $F$, then any path from $v$ to $w$ in $G$ must have all nodes in $T$. 
Basic Facts about Traversal Trees (for further analyses)

Theorem 5.2: Suppose we have run traverse on $G$, resulting in a search forest $F$.

1. If $T_1$ and $T_2$ are different trees in $F$ and $T_1$ was explored before $T_2$, then there are no arcs from $T_1$ to $T_2$.

2. If $G$ is a graph, then there can be no edges joining different trees of $F$.

3. If $v, w \in V(G)$; $v$ is visited before $w$, and $w$ is reachable from $v$ in $G$, then $v$ and $w$ belong to the same tree of $F$.

4. If $v, w \in V(G)$ and $v$ and $w$ belong to the same tree $T$ in $F$, then any path from $v$ to $w$ in $G$ must have all nodes in $T$. 

Run-time Analysis of Algorithm \texttt{traverse}

In the \textbf{while-loop} of subroutine \texttt{visit} let:

- \( a \) \((A)\) be lower (upper) time bound to choose a GREY node.
- \( b \) \((B)\) be lower (upper) time bound to choose a WHITE neighbour.

Given a (di)graph \( G = (V, E) \) of order \( n = |V| \) and size \( m = |E| \), the running time of \texttt{traverse} is:

- \( O(An + Bm) \) and \( \Omega(an + bm) \) with adjacency lists, and
- \( O(An + Bn^2) \) and \( \Omega(an + bn^2) \) with an adjacency matrix.

Time to find a GREY node: \( O(An) \) and \( \Omega(an) \)

Time to find a WHITE neighbour: \( O(Bm) \) and \( \Omega(bm) \) (adjacency lists)
- \( O(Bn^2) \) and \( \Omega(bn^2) \) (an adjacency matrix)

- Generally, \( A, B, a, b \) may depend on \( n \).
- A more detailed analysis depends on the rules used.
Main Rules for Choosing Next Nodes

- **Depth-first search (DFS):**
  - Starting at a node \( v \).
  - Searching as far away from \( v \) as possible via neighbours.
  - Continue from the next neighbour until no more new nodes.

- **Breadth-first search (BFS):**
  - Starting at a node \( v \).
  - Searching through all its neighbours, then through all their neighbours, etc.
  - Continue until no more new nodes.

- More complicated priority-first search (PFS).
Depth-first Search (DFS) Algorithm (Part 1)

algorithm dfs

Input: digraph $G = (V(G), E(G))$

begin

stack $S$; array $colour[n], pred[n], seen[n], done[n]$

for $u \in V(G)$ do

$colour[u] \leftarrow$ WHITE; $pred[u] \leftarrow$ NULL

end for

time $\leftarrow$ 0

for $s \in V(G)$ do

if $colour[s] =$ WHITE then

dfsvisit($s$)

end if

end for

return $pred, seen, done$

end
Depth-first Search (DFS) Algorithm (Part 2)

algorithm dfsvisit
    Input: node s
    begin
        colour[s] ← GREY; seen[s] ← time + +;
        S.push_top(s)
        while not S.isempty() do
            u ← S.get_top()
            if there is a v adjacent to u and colour[v] = WHITE then
                colour[v] ← GREY; pred[v] ← u
                seen[v] ← time + +; S.push_top(v)
            else S.del_top();
                colour[u] ← BLACK; done[u] ← time + +;
            end if
        end while
    end
Recursive View of DFS Algorithm

algorithm rec_dfs_visit
  Input: node s
  begin
    colour[s] ← GREY
    seen[s] ← time + +
    for each v adjacent to s do
      if colour[v] = WHITE then
        pred[v] ← s
        rec_dfs_visit(v)
      end if
    end for
    colour[s] ← BLACK
    done[s] ← time + +
  end
DFS: An Example \((\text{seen}[v] \mid \text{done}[v])\): \(\text{time} = 0; 1\)
DFS: An Example ($seen[v] \mid done[v]$): $time = 1; 2$
DFS: An Example \((\text{seen}[v] \mid \text{done}[v])\): time = 2, 3
DFS: An Example \((seen[v] \mid done[v])\): \(time = 3; 4\)
DFS: An Example \((seen[v] \mid done[v])\): \(time = 4; 5\)
DFS: An Example \((\text{seen}[v] \mid \text{done}[v] \colon \text{time} = 5, 6)\)
DFS: An Example (seen[v] | done[v]): time = 6, 7
DFS: An Example ($seen[v] \mid done[v]$): $time = 7, 8$
DFS: An Example \((\text{seen}[v] \mid \text{done}[v])\): \(\text{time} = 8, 9\)
DFS: An Example \((seen[v] \mid done[v])\): \(time = 9, 10\)
Basic Properties of Depth-first Search

Next GREY node chosen ← the last one coloured GREY thus far.

- Data structure for this “last in, first out” order – a stack.

Each call to dfs_visit(v) terminates only when all nodes reachable from v via a path of WHITE nodes have been seen.

If (v, w) is an arc, then for a

- tree or forward arc: seen[v] < seen[w] < done[w] < done[v]
  - Example in Slide 52: (a, b) : 0 < 1 < 8 < 9; (b, c) : 1 < 2 < 5 < 8; (a, c) : 0 < 2 < 5 < 9;

- back arc: seen[w] < seen[v] < done[v] < done[w]:
  - Example in Slide 52: (d, a) : 0 < 6 < 7 < 9;

  - Example in Slide 52: (d, e) : 3 < 4 < 6 < 7;

Hence, there are no cross edges on a graph.
Tree, Forward, Back, and Cross Arcs

- **Tree arc**
- **Forward arc**
- **Back arc**
- **Cross arc**

(Example in Slide 52)
Using DFS to Determine Ancestors of a Tree

Theorem 5.5

Suppose that DFS on a digraph \( G \) results in a search forest \( F \). Let \( v, w \in V(G) \) and \( \text{seen}[v] < \text{seen}[w] \).

1. If \( v \) is an ancestor of \( w \) in \( F \), then
   \[
   \text{seen}[v] < \text{seen}[w] < \text{done}[w] < \text{done}[v].
   \]

2. If \( v \) is not an ancestor of \( w \) in \( F \), then
   \[
   \text{seen}[v] < \text{done}[v] < \text{seen}[w] < \text{done}[w].
   \]

Proof.

1. This part follows from the recursive nature of DFS.

2. If \( v \) is not an ancestor of \( w \) in \( F \), then \( w \) is also not an ancestor of \( v \).
   - Thus \( v \) is in a subtree, which was completely explored before the subtree of \( w \).
DFS: *seen/done*: step 1

Preorder (WHITE to GREY): *seen* A

1

Postorder (GREY to BLACK) *done*
DFS: seen/done: step 2

Preorder (WHITE to GREY): seen A B

Postorder (GREY to BLACK) done
DFS: *seen/done*: step 3

Preorder (WHITE to GREY): *seen* A B

Postorder (GREY to BLACK) *done* B
DFS: *seen/done*: step 4

Preorder (WHITE to GREY): *seen* A B S
1 2 4

Postorder (GREY to BLACK) *done* B
3
DFS: *seen/done*: step 5

**Preorder (WHITE to GREY):** *seen* A B S C
1 2 4 5

**Postorder (GREY to BLACK):** *done* B
3
DFS: seen/done: step 6

Preorder (WHITE to GREY): seen A B S C D 1 2 4 5 6

Postorder (GREY to BLACK) done B 3
DFS: **seen/done**: step 7

Preorder (WHITE to GREY): seen A B S C D

Postorder (GREY to BLACK) done B D
DFS: seen/done: step 8

Preorder (WHITE to GREY): seen A B S C D E
1 2 4 5 6 8

Postorder (GREY to BLACK) done B D
3 7
DFS: seen/done: step 9

Preorder (WHITE to GREY): seen A B S C D E H

Postorder (GREY to BLACK) done B D
DFS: seen/done: step 10

Preorder (WHITE to GREY): seen A B S C D E H G
1 2 4 5 6 8 9 10

Postorder (GREY to BLACK) done B D
3 7
DFS: \textit{seen/done}: step 11

Preorder (WHITE to GREY): \textit{seen} A B S C D E H G F

\begin{align*}
    &1/\sim \\
    &4/\sim
    &5/\sim
    &10/\sim
    &11/\sim
    &2/3
    &6/7
    &8/\sim
    &9/\sim

\end{align*}

Postorder (GREY to BLACK) \textit{done} B D

\begin{align*}
    &3/\sim \\
    &7
\end{align*}
DFS: *seen/done*: step 12

Preorder (WHITE to GREY): *seen* A B S C D E H G F 1 2 4 5 6 8 9 10 11

Postorder (GREY to BLACK) *done* B D F 3 7 12
DFS: **seen/done**: step 13

Preorder (WHITE to GREY): **seen** A B S C D E H G F

Postorder (GREY to BLACK) **done** B D F G
DFS: *seen/done*: step 14

Preorder (WHITE to GREY): *seen* A B S C D E H G F

1 2 4 5 6 8 9 10 11

Postorder (GREY to BLACK) *done* B D F G H

3 7 12 13 14
DFS: seen/done: step 15

Preorder (WHITE to GREY): seen A B S C D E H G F
1 2 4 5 6 8 9 10 11

Postorder (GREY to BLACK) done B D F G H E
3 7 12 13 14 15
DFS: seen/done: step 16

Preorder (WHITE to GREY): seen A B S C D E H G F 1 2 4 5 6 8 9 10 11

Postorder (GREY to BLACK) done B D F G H E C 3 7 12 13 14 15 16
DFS: \textit{seen}/\textit{done}: step 17

Preorder (WHITE to GREY): \textit{seen} A B S C D E H G F

Postorder (GREY to BLACK) \textit{done} B D F G H E C S
DFS: *seen/done*: step 18

Preorder (WHITE to GREY): *seen* A B S C D E H G F
1 2 4 5 6 8 9 10 11

Postorder (GREY to BLACK) *done* B D F G H E C S A
3 7 12 13 14 15 16 17 18
Determining Ancestors of a Tree: Examples

S → H: seen[S] = 4 < seen[H] = 9 < done[H] = 14 < done[S] = 17