Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
TODAY

- Directed Graphs
- Digraph API
- Digraph search
Directed graphs

**Digraph.** Set of vertices connected pairwise by *directed* edges.
Vertex = intersection; edge = one-way street.
## Digraph applications

<table>
<thead>
<tr>
<th>digraph</th>
<th>vertex</th>
<th>directed edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection</td>
<td>one-way street</td>
</tr>
<tr>
<td>web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey relationship</td>
</tr>
<tr>
<td>WordNet</td>
<td>synset</td>
<td>hypernym</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence constraint</td>
</tr>
<tr>
<td>financial</td>
<td>bank</td>
<td>transaction</td>
</tr>
<tr>
<td>cell phone</td>
<td>person</td>
<td>placed call</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>citation</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>inheritance hierarchy</td>
<td>class</td>
<td>inherits from</td>
</tr>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
</tbody>
</table>
Some digraph problems

Path.  Is there a directed path from \( s \) to \( t \)?

Shortest path.  What is the shortest directed path from \( s \) to \( t \)?

Topological sort.  Can you draw the digraph so that all edges point upwards?

Strong connectivity.  Is there a directed path between all pairs of vertices?

Transitive closure.  For which vertices \( v \) and \( w \) is there a path from \( v \) to \( w \)?

PageRank.  What is the importance of a web page?
DIRECTED GRAPHS

- Digraph API
- Digraph search
% Digraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
3->2
4->3
4->2
5->4
::
11->4
11->12
12->9
Set-of-edges digraph representation

Store a list of the edges (linked list or array).
Maintain a two-dimensional $v$-by-$v$ boolean array; for each edge $v \rightarrow w$ in the digraph: $\text{adj}[v][w] = \text{true}$.
Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.
Adjacency-list graph representation: implementation

```java
//for each edge call addEdge twice
addEdge(v, w);

public void addEdge(int v, int w){
    node *q;
    //acquire memory for the new node
    q=(node*)malloc(sizeof(node));
    q->vertex=w;
    q->next=NULL;
    //insert the node to beginning of the linked list
    q->next=adj[v];
    adj[v]= q;
}

typedef struct node
{
    struct node *next;
    int vertex;
}node;

node * adj [13];
```
### Digraph representations

**In practice.** Use adjacency-lists representation.
- Algorithms based on iterating over vertices pointing from \( v \).
- Real-world digraphs tend to be sparse.

#### Digraph representations

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge from ( v ) to ( w )</th>
<th>edge from ( v ) to ( w )?</th>
<th>iterate over vertices pointing from ( v )?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>( E )</td>
<td>( 1 )</td>
<td>( E )</td>
<td>( E )</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>( V^2 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( V )</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>( E + V )</td>
<td>( 1 )</td>
<td>outdegree(( v ))</td>
<td>outdegree(( v ))</td>
</tr>
</tbody>
</table>

- huge number of vertices, small average vertex degree
DIRECTED GRAPHS

- Digraph API
- Digraph search
Reachability

Problem. Find all vertices reachable from $s$ along a directed path.
Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

**DFS (to visit a vertex v)**

Mark v as visited.
Recursively visit all unmarked vertices w pointing from v.
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$. 

A directed graph
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$.

![Directed Graph]

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>–</td>
</tr>
</tbody>
</table>
To visit a vertex \( v \):

- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices pointing from \( v \).

**Depth-first search**

Visit 0: check 5 and check 1
Depth-first search

To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$.

\[
\begin{array}{c|c|c}
\text{v} & \text{marked[]} & \text{edgeTo[]} \\
\hline
0 & T & - \\
1 & F & - \\
2 & F & - \\
3 & F & - \\
4 & F & - \\
5 & T & 0 \\
6 & F & - \\
7 & F & - \\
8 & F & - \\
9 & F & - \\
10 & F & - \\
11 & F & - \\
12 & F & - \\
\end{array}
\]

visit 5: check 4
Depth-first search

To visit a vertex \( v \):
• Mark vertex \( v \) as visited.
• Recursively visit all unmarked vertices pointing from \( v \).

\[
\begin{array}{cccc}
 v & \text{marked[]} & \text{edgeTo[]} \\
0 & T & - \\
1 & F & - \\
2 & F & - \\
3 & F & - \\
4 & T & 5 \\
5 & T & 0 \\
6 & F & - \\
7 & F & - \\
8 & F & - \\
9 & F & - \\
10 & F & - \\
11 & F & - \\
12 & F & - \\
\end{array}
\]

visit 4: check 3 and check 2
To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$.

### Depth-first search

#### Visit 3: check 5 and check 2
Depth-first search

To visit a vertex \( v \):
- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices pointing from \( v \).

Visit 3: check 5 and check 2
Depth-first search

To visit a vertex $v$:
• Mark vertex $v$ as visited.
• Recursively visit all unmarked vertices pointing from $v$.

visit 2: check 0 and check 3
To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$.

**Depth-first search**

**Visit 2:** check 0 and check 3

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>
Depth-first search

To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$.
Depth-first search

To visit a vertex \( v \):

- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices pointing from \( v \).
To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$.

Visit 4: check 3 and check 2
To visit a vertex \( v \):

- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices pointing from \( v \).
To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$. 

\[ \begin{array}{c|c|c}
   v & \text{marked}[] & \text{edgeTo}[] \\
   \hline
   0 & T & - \\
   1 & F & - \\
   2 & T & 3 \\
   3 & T & 4 \\
   4 & T & 5 \\
   5 & T & 0 \\
   6 & F & - \\
   7 & F & - \\
   8 & F & - \\
   9 & F & - \\
   10 & F & - \\
   11 & F & - \\
   12 & F & - \\
\end{array} \]
Depth-first search

To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$.

**visit 0**: check 5 and check 1

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>
Depth-first search

To visit a vertex \( v \):

- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices pointing from \( v \).
To visit a vertex $v$

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$. 

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>
Depth-first search

To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$.
Depth-first search

To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$. 

![Diagram of a graph with vertices and edges marked.]

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>
To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$.

深度优先搜索（Depth-first search）
void DFS(int i)
{
    node *p = adj[i];
    visited[i]=1;
    while(p != NULL)
    {
        i = p->vertex;
        if(!visited[i])
            DFS(i);
        p = p->next;
    }
}

typedef struct node
{
    struct node *next;
    int vertex;
}node;

//GLOBAL PARAMETERS
node * adj [13];
int visited[13];

Same as undirected graph
Reachability application: program control-flow analysis

Every program is a digraph.
• Vertex = basic block of instructions (straight-line program).
• Edge = jump.

Dead-code elimination.
Find (and remove) unreachable code.

Infinite-loop detection.
Determine whether exit is unreachable.
Every data structure is a digraph.
- Vertex = object.
- Edge = reference.

**Roots.** Objects known to be directly accessible by program (e.g., stack).

**Reachable objects.** Objects indirectly accessible by program (starting at a root and following a chain of pointers).
Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

• Mark: mark all reachable objects.
• Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).
Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a **digraph** algorithm.

---

**BFS (from source vertex s)**

---

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- remove the least recently added vertex v
- for each unmarked vertex pointing from v:
  - add to queue and mark as visited.