Digraphs and Graphs:

- Basic notions, representations, examples
- Graph Search: Depth-first, breadth-first, best-first
Graphs: Basic Definitions

Graphs are the most basic model of collections of information, generalizing trees to allow arbitrary links between nodes. There are two flavors, Directed and Undirected.

A Directed Graph (or Digraph) is:

- A set $V$ of Vertices (or: Nodes) containing (possibly):
  - A Label (1, 2, ... A, B, ... etc.)
  - Data fields (boolean flags, counters, etc.)
- A set $E$ of Edges (links) connecting vertices; edges may have labels or data (e.g., weight or cost) associated with them.

$E$ is usually expressed as a relation on $V$, i.e., $E$ consists of pairs of vertices:

$(source, target)$

$E$ is a subset of $V \times V$ (Cartesian Product of $V$)

$V = \{ A, B, C, D \}$

$E = \{ (A,B), (A, C), (B, C), (D,B), (D,C) \}$
Graphs: Basic Definitions

Graphs are the most basic model of collections of information, generalizing trees to allow arbitrary links between nodes.

A Directed Graph (or Digraph) is:

- A set $V$ of Vertices (or: Nodes) containing (possibly):
  - A Label ($1, 2, \ldots, A, B, \ldots$ etc.)
  - Data fields (boolean flags, counters, etc.)
- A set $E$ of Edges (links) connecting vertices; edges may have labels or data (e.g., weight or cost) associated with them.

$E$ is usually expressed as a relation on $V$, i.e., $E$ consists of pairs of vertices:

- (source, target)

$E$ is a subset of $V \times V$ (Cartesian Product of $V$)

$V = \{ A, B, C, D \}$

$E = \{ (A,B), (A, C), (B, C), (D, B), (D, C) \}$

Note: Like a tree, but

- Any vertex can be connected to any other vertex;
- There is no root.
Graphs: Basic Definitions

A Directed Graph (or Digraph) is:

- A set $V$ of Vertices (or: Nodes) containing (possibly):
  - A Label (1, 2, ... A, B, ... etc.)
  - Data fields (boolean flags, counters, etc.)
- A set $E$ of Edges (links) connecting vertices; edges may have labels or data (e.g., weight or cost) associated with them.

Edges have a direction (e.g., one way streets)

An Undirected Graph has the additional feature that all edges are two way, i.e., the relation $E$ is symmetric:

$$(A, B) \text{ is in } E \text{ iff } (B, A) \text{ is in } E$$

Edges do NOT have a direction (e.g., two-way streets).

“Graph” can mean either, but generally is assumed to be undirected.

Undirected Graph $G$

$V = \{ A, B, C, D \}$

$E = \{ (A, B), (A, C), (B, C), (D, B), (D, C), (B, A), (C, A), (C, B), (B, D), (C, D) \}$
Graphs: Basic Definitions

A Directed Graph (or Digraph) is:
- A set V of Vertices (or: Nodes) containing (possibly):
  - A Label (1, 2, ... A, B, ... etc.)
  - A set E of Edges (links) connecting vertices; edges may have labels or data (e.g., weight or cost) associated with them.
  - Data fields (boolean flags, counters, etc.)
- Edges have a direction (e.g., one way streets)

An Undirected Graph has the additional feature that all edges are two way, i.e., the relation E is symmetric:

(A, B) is in E iff (B, A) is in E

Edges do NOT have a direction (e.g., two-way streets).

“Graph” can mean either, but generally is assumed to be undirected.

Undirected Graph G

V = \{ A, B, C, D \}

E = \{ (A,B), (A, C), (B, C), (D, B), (D, C), (B,A), (C,A), (C,B), (B,D), (C,D) \}
A Directed Graph (or Digraph) is:

- A set $V$ of Vertices (or: Nodes) containing (possibly):
  - A Label ($1, 2, \ldots, A, B, \ldots$ etc.)
  - Data fields (boolean flags, counters, etc.)
- A set $E$ of Edges (links) connecting vertices; edges may have labels or data (e.g., weight or cost) associated with them.

Edges have a direction (e.g., one way streets)

An Undirected Graph has the additional feature that all edges are two way, i.e., the relation $E$ is symmetric:

$(A, B)$ is in $E$ iff $(B, A)$ is in $E$

Edges do NOT have a direction (e.g., two-way streets).

“Graph” can mean either, but generally is assumed to be undirected.

Undirected Graph $G$

$V = \{ A, B, C, D \}$

$E = \{ (A, B), (A, C), (B, C), (D, B), (D, C), (B, A), (C, A), (C, B), (B, D), (C, D) \}$

But generally are drawn without arrow-heads.
Graphs: Basic Definitions

We will focus on Undirected Graphs for this lecture.

Graphs are EVERYWHERE in computer science and mathematics, and you have been looking at them for years……
Graphs: Basic Definitions

And you've been using graphs for years......
Graphs: Basic Definitions

Basic Notions of Graphs:
- **Vertex** (Vertex set V)
- **Edge** (Edge set E)

The degree of a vertex is the number of edges it participates in.

Graph $G$

$V = \{A, B, C, D\}$

$E = \{(A,B), (A, C), (B, C), (D,B), (D,C), (B,A), (C,A), (C,B), (B,D), (C,D)\}$

- C has degree 3;
- D has degree 2
Graphs: Basic Definitions

Basic Notions of Graphs:
- **Vertex** (Vertex set $V$)
- **Edge** (Edge set $E$)

The degree of a vertex is the number of edges it participates in.

Two vertices are **adjacent** if there is an edge between them.

Graph $G$

$V = \{ A, B, C, D \}$

$E = \{ (A,B), (A, C), (B, C), (D, B), (D, C), (B, A), (C, A), (C, B), (B, D), (C, D) \}$

C has degree 3; D has degree 2
Graphs: Basic Definitions

Basic Notions of Graphs:
Vertex (Vertex set \( V \))
Edge (Edge set \( E \))

The degree of a vertex is the number of edges it participates in.

Two vertices are adjacent if there is an edge between them.

A path is a sequence of adjacent edges.

Graph \( G \)

\[ V = \{ A, B, C, D \} \]

\[ E = \{ (A,B), (A, C), (B,C), (D,B), (D,C), (B,A), (C,A), (C,B), (B,D), (C,D) \} \]

Path: A, C, D
Basic Notions of Graphs:

Vertex (Vertex set V)

Edge (Edge set E)

The degree of a vertex is the number of edges it participates in.

Two vertices are adjacent if there is an edge between them.

A path is a sequence of adjacent edges. The length of a path is the number of edges.

Graph G

V = { A, B, C, D }

E = { (A,B), (A, C), (B,C), (D,B), (D,C), (B,A), (C,A), (C,B), (B,D), (C,D) }
Graphs: Basic Definitions

Basic Notions of Graphs:

**Vertex** (Vertex set V)

**Edge** (Edge set E)

The degree of a vertex is the number of edges it participates in.

Two vertices are adjacent if there is an edge between them.

A path is a sequence of adjacent edges.

A **cycle** (or loop) is a path that begins and ends on the same vertex.

A, C, D, B, A

**Graph G**

\[ V = \{ A, B, C, D \} \]

\[ E = \{ (A,B), (A, C), (B,C), (D,B), (D,C), (B,A), (C,A), (C,B), (B,D), (C,D) \} \]
Graphs: Basic Definitions

Basic Notions of Graphs:

Vertex (Vertex set V)

Edge (Edge set E)

The degree of a vertex is the number of edges it participates in.

Two vertices are adjacent if there is an edge between them.

A path is a sequence of adjacent edges.

A cycle (or loop) is a path that begins and ends on the same vertex.

A, C, D, B, A

A simple cycle has no other repeated vertices (i.e., does not cross itself).

A, C, B, A, C, D, B, A is not simple.

V = { A, B, C, D }

E = { (A,B), (A, C), (B,C), (D,B), (D,C), (B,A), (C,A), (C,B), (B,D), (C,D) }
Basic Notions of Graphs:

Vertex (Vertex set V); Edge (Edge set E)

The degree of a vertex is the number of edges it participates in.

Two vertices are adjacent if there is an edge between them.

A path is a sequence of adjacent edges.

A cycle (or loop) is a path that begins and ends on the same vertex.

A, C, D, B, A

A simple cycle has no other repeated vertices (i.e., does not cross itself).

A, C, B, A, C, D, B, A is not simple.

Two vertices are connected if there is a path between them.

A set of vertices is connected if there is a path between each pair; a graph is connected if there is a path between any two nodes.

{ A, B, C, D } is connected; so is {E, F}
Graphs: Basic Definitions

Basic Notions of Graphs:

Vertex (Vertex set V); Edge (Edge set E)

Two vertices are adjacent if there is an edge between them.

A path is a sequence of adjacent edges.

A cycle (or loop) is a path that begins and ends on the same vertex.

A, C, D, B, A

A simple cycle has no other repeated vertices (i.e., does not cross itself).

A, C, B, A, C, D, B, A is not simple.

Two vertices are connected if there is a path between them.

A set of vertices is connected if there is a path between each pair; a graph is connected if there is a path between any two nodes.

The set of connected components of G is the collection of maximal connected subsets of vertices. The connected components of G are {A, B, C, D} and {E, F}
Basic Notions of Graphs:

- **Vertex** (Vertex set V); **Edge** (Edge set E)

Two vertices are adjacent if there is an edge between them.

A path is a sequence of adjacent edges.

A **cycle (or loop)** is a path that begins and ends on the same vertex.

- A, C, D, B, A

A simple cycle has no other repeated vertices (i.e., does not cross itself).

- A, C, B, A, C, D, B, A is not simple.

Two vertices are connected if there is a path between them.

A set of vertices is connected if there is a path between each pair; a graph is connected if there is a path between any two nodes.

A tree is a acyclic, connected graph.
Graphs: Basic Definitions

Implementing Graphs.

Vertex set $V$ is a list of vertices:
$V = (A, B, C, D, E, F)$

Edge set is a boolean array:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Graphs: Basic Definitions

Implementing Graphs.

Vertex set V is a list of vertices:

V = (A, B, C, D, E, F)

Edge set is a boolean array: Note the symmetry!

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementing Graphs.

Vertex set $V$ is a list of vertices:
$V = (A, B, C, D, E, F)$

Edge set is a boolean array; you could optimize by only storing half!

\[
\begin{array}{cccccc}
A & B & C & D & E & F \\
A & & T & T & & \\
B & T & & T & T & \\
C & T & T & & T & \\
D & T & T & T & & \\
E & & & & T & \\
F & & & & T & \\
\end{array}
\]
Graphs: Basic Definitions

Implementing Graphs.

Vertex set $V$ is a list of vertices:

$$V = (A, B, C, D, E, F)$$

OR, edge set is an integer array giving the weights of the edges:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- A
- B
- C
- D
- E
- F

- A to B: 4
- A to C: 2
- B to C: 1
- B to D: 3
- B to F: 4
- C to D: 1
- C to F: 2
You need a number of methods:

**Adjacent(v)** = list of vertices adjacent to v, in some order (if scan across the row for v would be in order of set V, but could be otherwise)

**Degree(v)** = number of T values in row for v

Etc. (more as we develop the algorithms)
Graph Search Algorithms are all generalizations of the following algorithm for Depth-First Search

Suppose that vertices have a flag:

```java
boolean visited;

searchGraph(V, E) {
    foreach (v in V) // initialize all vertices
        v.visited = false;
    foreach (v in V)
        if(!v.visited)
            DFS(v);
}

DFS(v) {
    if(!v.visited) {
        visit(v); // do something, e.g. print out
        v.visited = true;
        foreach (u in Adjacent(v)) {
            DFS(u);
        }
    }
}
```
To consider the general search algorithms, we have to rephrase this as a non-recursive algorithm.

Here is depth-first search again:

```java
searchGraph(V, E) {
    foreach (v in V) // initialize all vertices
        v.visited = false;
    foreach (v in V)
        if(!v.visited)
            DFS(v);
}

DFS(v) {
    Stack S = new Stack();
    S.push(v);
    while( !S.isempty() ) {
        Vertex u = S.pop();
        if(!u.visited) {
            visit(u);
            u.visited = true;
            foreach (w in Adjacent(u))
                if(!w.visited)
                    S.push(w);
        }
    }
}
```
Graphs: Basic Definitions

To consider the general search algorithms, we have to rephrase this as a non-recursive algorithm.

By replacing the Stack with a Queue, we have Breadth-First Search

boolean visited;
searchGraph(V, E) {
    foreach (v in V) // initialize all vertices
        v.visited = false;
    foreach (v in V)
        if(!v.visited)
            BFS(v);
}

BFS(v) {
    Queue S = new Queue();
    S.enqueue(v);
    while(!S.isempty()) {
        Vertex u = S.dequeue();
        if(!u.visited) {
            visit(u);
            u.visited = true;
            foreach (w in Adjacent(u))
                if(!w.visited)
                    S.enqueue(w);
        }
    }
}

Graph G
Graphs: Basic Definitions

To consider the general search algorithms, we have to rephrase this as a non-recursive algorithm. By replacing the Stack with a Priority Queue, we have Best-First Search.

```java
boolean visited;

searchGraph(V, E) {
    foreach (v in V)               // initialize all vertices
        v.visited = false;
    foreach (v in V)
        if(!v.visited)
            BFS(v);
}

BFS(v) {
PQueue S = new PQueue();
    S.enqueue(v);
    while (!S.isempty()) {
        Vertex u = S.dequeue();
        if(!u.visited) {
            visit(u);
            u.visited = true;
            foreach (w in Adjacent(u))
                if(!w.visited)
                    S.enqueue(w);
        }
    }
}
```

Optional Slide: You are not responsible for best-first search.