GRAPHS
These are not Graphs

...not the kind we mean, anyway
These are Graphs

K_5

K_{3,3}
Applications of Graphs

- Communication networks
- The internet is a huge graph
- Routing and shortest path problems
- Commodity distribution (flow)
- Traffic control
- Resource allocation
- Geometric modeling
- ...

Graph Definitions

- A directed graph (or digraph) is a pair \((V, E)\) where
  - \(V\) is a set
  - \(E\) is a set of ordered pairs \((u, v)\) where \(u, v \in V\)
    - Sometimes require \(u \neq v\) (i.e. no self-loops)

- An element of \(V\) is called a vertex (pl. vertices) or node
- An element of \(E\) is called an edge or arc

- \(|V|\) is the size of \(V\), often denoted by \(n\)
- \(|E|\) is size of \(E\), often denoted by \(m\)
Example Directed Graph (Digraph)

\[ V = \{a, b, c, d, e, f\} \]
\[ E = \{(a, b), (a, c), (a, e), (b, c), (b, d), (b, e), (c, d), (c, f), (d, e), (d, f), (e, f)\} \]

\[ |V| = 6, \ |E| = 11 \]
An undirected graph is just like a directed graph, except the edges are unordered pairs (sets) \{u,v\}

Example:

\[
V = \{a,b,c,d,e,f\} \\
E = \{\{a,b\}, \{a,c\}, \{a,e\}, \{b,c\}, \{b,d\}, \{b,e\}, \{c,d\}, \{c,f\}, \{d,e\}, \{d,f\}, \{e,f\}\}
\]
Some Graph Terminology

- **u** is the source, **v** is the sink of (u,v)
- **u, v, b, c** are the endpoints of (u,v) and (b, c)
- **u, v** are adjacent nodes. **b, c** are adjacent nodes

- **outdegree** of **u** in directed graph: number of edges for which **u** is source
- **indegree** of **v** in directed graph: number of edges for which **v** is sink
- **degree** of vertex **w** in undirected graph: number of edges of which **w** is an endpoint

outdegree of **u**: 4  
indegree of **v**: 3  
degree of **w**: 2
More Graph Terminology

- **path**: sequence of adjacent vertexes
- **length of path**: number of edges
- **simple path**: no vertex is repeated

Simple path of length 2: (b, c, d)

Simple path of length 0: (b)

Not a simple path: (b, c, e, b, c, d)
More Graph Terminology

- **cycle**: path that ends at its beginning
- **simple cycle**: only repeated vertex is its beginning/end
- **acyclic graph**: graph with no cycles
- **dag**: directed acyclic graph

Cycles: (b, c, e, b)  (b, c, e, b, c, e, b)

Simple cycle: (c, e, b, c)

Graph shown is not a dag

Question: is (d) a cycle?

No. A cycle must have at least one edge
Intuition: A dag has a vertex with indegree 0. Why?

This idea leads to an algorithm:

A digraph is a dag if and only if one can iteratively delete indegree-0 vertices until the graph disappears.

Is this a dag?
Intuition: A dag has a vertex with indegree 0. Why?

This idea leads to an algorithm:

A digraph is a dag if and only if one can iteratively delete indegree-0 vertices until the graph disappears.
Intuition: A dag has a vertex with indegree 0. Why?

This idea leads to an algorithm:

A digraph is a dag if and only if one can iteratively delete indegree-0 vertices until the graph disappears.
Is this a dag?

- Intuition: A dag has a vertex with indegree 0. Why?

- This idea leads to an algorithm:
  A digraph is a dag if and only if one can iteratively delete indegree-0 vertices until the graph disappears.
Is this a dag?

- Intuition: A dag has a vertex with indegree 0. Why?
- This idea leads to an algorithm:
  A digraph is a dag if and only if one can iteratively delete indegree-0 vertices until the graph disappears.
Is this a dag?

- Intuition: A dag has a vertex with indegree 0. Why?

- This idea leads to an algorithm:
  A digraph is a dag if and only if one can iteratively delete indegree-0 vertices until the graph disappears.
Is this a dag?

- **Intuition:** A dag has a vertex with indegree 0. Why?

- This idea leads to an algorithm:
  
  A digraph is a dag if and only if one can iteratively delete indegree-0 vertices until the graph disappears.
We just computed a topological sort of the dag. This is a numbering of the vertices such that all edges go from lower- to higher-numbered vertices.

Useful in job scheduling with precedence constraints.
Graph Coloring

Coloring of an undirected graph: an assignment of a color to each node such that no two adjacent vertices get the same color.

How many colors are needed to color this graph?
Graph Coloring

A **coloring** of an undirected graph: an assignment of a color to each node such that no two adjacent vertices get the same color.

How many colors are needed to color this graph?

3
An Application of Coloring

- Vertices are jobs
- Edge \((u,v)\) is present if jobs \(u\) and \(v\) each require access to the same shared resource, so they cannot execute simultaneously
- Colors are time slots to schedule the jobs
- Minimum number of colors needed to color the graph = minimum number of time slots required
A graph is **planar** if it can be embedded in the plane with no edges crossing.

Is this graph planar?
Planarity

A graph is **planar** if it can be embedded in the plane with no edges crossing.

Is this graph planar? **YES**
A graph is planar if and only if it does not contain a copy of $K_5$ or $K_{3,3}$ (possibly with other nodes along the edges shown)
Detecting Planarity

Early 1970’s John Hopcroft spent time at Stanford, talked to grad student Bob Tarjan (now at Princeton). Together, they developed a linear-time algorithm to test a graph for planarity. Significant achievement.

Won Turing Award
The Four-Color Theorem

Every planar graph is 4-colorable
(Appel & Haken, 1976)

Interesting history. “Proved” in about 1876 and published, but ten years later, a mistake was found. It took 90 more years for a proof to be found.

Countries are nodes; edge between them if they have a common boundary. You need 5 colors to color a map — water has to be blue!
The Four-Color Theorem

Every planar graph is 4-colorable
(Appel & Haken, 1976)

Proof rests on a lot of computation! A program checks thousands of “configurations”, and if none are colorable, theorem holds.

Program written in assembly language. Recursive, contorted, to make it efficient. Gries found an error in it but a “safe kind”: it might say a configuration was colorable when it wasn’t.
Bipartite Graphs

A directed or undirected graph is bipartite if the vertices can be partitioned into two sets such that all edges go between the two sets.
The following are equivalent

- $G$ is bipartite
- $G$ is 2-colorable
- $G$ has no cycles of odd length
Traveling Salesperson

Find a path of minimum distance that visits every city
Representations of Graphs

Adjacency List

Adjacency Matrix

1   2   3   4
1   0   1   0   1
2   0   0   1   0
3   0   0   0   0
4   0   1   1   0
Adjacency Matrix or Adjacency List?

- n: number of vertices
- m: number of edges
- d(u): outdegree of u

Adjacency Matrix
- Uses space $O(n^2)$
- Can iterate over all edges in time $O(n^2)$
- Can answer “Is there an edge from u to v?” in $O(1)$ time
- Better for dense graphs (lots of edges)

Adjacency List
- Uses space $O(m+n)$
- Can iterate over all edges in time $O(m+n)$
- Can answer “Is there an edge from u to v?” in $O(d(u))$ time
- Better for sparse graphs (fewer edges)
Graph Algorithms

• Search
  – depth-first search
  – breadth-first search

• Shortest paths
  – Dijkstra's algorithm

• Minimum spanning trees
  – Prim's algorithm
  – Kruskal's algorithm
Depth-First Search

• Follow edges depth-first starting from an arbitrary vertex $r$, using a stack to remember where you came from
• When you encounter a vertex previously visited, or there are no outgoing edges, retreat and try another path
• Eventually visit all vertices reachable from $r$
• If there are still unvisited vertices, repeat
• $O(m)$ time

Difficult to understand!
Let’s write a recursive procedure
Depth-First Search

boolean[] visited;

node u is visited means: visited[u] is true
To visit u means to: set visited[u] to true

Node u is REACHABLE from node v if there is a path (u, ..., v) in which all nodes of the path are unvisited.

Suppose all nodes are unvisited.
The nodes that are REACHABLE from node 1 are 1, 0, 2, 3, 5
The nodes that are REACHABLE from 4 are 4, 5, 6.
boolean[] visited;

To “visit” a node \( u \): set \( \text{visited}[u] \) to true.

Node \( u \) is REACHABLE from node \( v \) if there is a path \((u, \ldots, v)\) in which all nodes of the path are unvisited.

Suppose 2 is already visited, others unvisited.

The nodes that are REACHABLE from node 1 are 1, 0, 5

The nodes that are REACHABLE from 4 are 4, 5, 6.
/** Node u is unvisited. Visit all nodes that are REACHABLE from u. */

```java
public static void dfs(int u) {
    visited[u] = true;
}
```

Let u be 1
The nodes that are REACHABLE from node 1 are 1, 0, 2, 3, 5
/** Node u is unvisited. Visit all nodes that are REACHABLE from u. */

```java
public static void dfs(int u) {
    visited[u] = true;
    for each edge (u, v)
        if v is unvisited then dfs(v);
}
```

Let u be 1
The nodes to be visited are 0, 2, 3, 5

Have to do dfs on all unvisited neighbors of u
/** Node u is unvisited. Visit all nodes that are REACHABLE from u. */
public static void dfs(int u) {
    visited[u] = true;
    for each edge (u, v)
        if v is unvisited then dfs(v);
}

Let u be 1
The nodes to be visited are 0, 2, 3, 5

Suppose the for each loop visits neighbors in numerical order. Then dfs(1) visits the nodes in this order:
1, 0, 2, 3, 5
/* Node u is unvisited. Visit all nodes that are REACHABLE from u. */
public static void dfs(int u) {
    visited[u] = true;
    for each edge (u, v)
        if v is unvisited then dfs(v);
}

Example: There may be a different way (other than array visited) to know whether a node has been visited

Example: Instead of using recursion, use a loop and maintain the stack yourself.

That’s all there is to the basic dfs. You may have to change it to fit a particular situation.
Breadth-First Search (BFS)

BFS visits all neighbors first before visiting their neighbors. It goes level by level.

Use a queue instead of a stack
- stack: last-in, first-out (LIFO)
- queue: first-in, first-out (FIFO)

dfs(0) visits in this order: 0, 1, 4, 5, 2, 3, 6
bfs(0) visits in this order: 0, 1, 2, 3, 4, 5, 6

Breadth-first not good for the Bfly: too much flying back and forth
Summary

- We have seen an introduction to graphs and will return to this topic on Thursday
  - Definitions
  - Testing for a dag
  - Depth-first and breadth-first search