Recall from last day:
Exploring Graphs – visit all reachable from some “source”

BFS

Adjacency lists:
1 → 2, 3, 6, 8
2 → 1, 4, 5
3 → 4, 5, 6

Cautious search:
check everything one edge away, then two, etc.

Order in which vertices are discovered

BFS tree

Pseudocode — see lecture 11: Runtime $O(n + m)$

Applications:
- shortest path (# edges) from root vertex $v_0$ to any node $v = \text{level of } v$
- testing if a graph has a cycle
- testing if a graph is bipartite
BFS to test bipartiteness

\( G \) is bipartite if \( V \) can be partitioned into \( V_1 \cup V_2 \) (\( V_1 \cap V_2 = \emptyset \)) such that every edge has one end in \( V_1 \) and one end in \( V_2 \).

Note that a bipartite graph cannot have an odd cycle.

Recall exercises from Lecture 11:

- If \((u, v) \in E\) then \(\text{level}(u), \text{level}(v)\) differ by 0 or 1.
- \(G\) has a cycle if and only if we discover a dashed edge, that is, we encounter a vertex that has already been discovered.
Run BFS. \( V_1 = \text{odd levels} \quad V_2 = \text{even levels} \).

Test if this works: for each edge \((u, v) \in E\) check that \(u, v \in V_1\) or \(u, v \in V_2\).

1. If YES then \(G\) is bipartite.

2. If NO then there is an edge \((u, v)\) with \(u, v\) both in \(V_i\) \((i = 1\) or \(2\)).
   
   We know that level\((u)\) and level\((v)\) differ by 1 or 0.
   
   If 1, then one in \(V_1\), one in \(V_2\). So \(u, v\) are in the same level, say \(k\).
   
   Let \(z = \text{least common ancestor of } u, v\).

   Cycle formed by path \((u, z)\), path \((z, v)\), and edge \((v, u)\) has length \(2k + 1\) — odd.
   
   Then \(G\) is not bipartite.

---

This proves:

**Lemma** \(G\) is bipartite if and only if it does not have an odd cycle.

Note: The proof is via an algorithm that finds a bipartition OR an odd cycle.
Depth First Search

Adjacency lists:

- **a**: b, c, d
- **b**: e, g, d, c, a
- **c**: a, b, d
- **d**: a, b, c
- **e**: b, f, g
- **f**: e
- **g**: b, e

**DFS tree**

- Order in which vertices are discovered: a, b, e, f, g, d, c
- Order of finishing: f, g, e, c, d, b, a

- Use a stack to store vertices that have been discovered but must still be explored.
- Note: orders depend on order in adjacency lists.
As a recursive program (stack is implicit):

```plaintext
DFS(v)
mark(v) := discovered
for u ∈ AdjacencyList(v) do
  if u is undiscovered then
    DFS(u); parent(u):=v; (u,v) is a tree edge
  else
    (v,u) is a non-tree edge (unless u = parent(v))
  fi
od
mark(v) := finished
```

DFS
mark all vertices undiscovered
for v ∈ V do  # this handles multiple components
  if v is undiscovered  # start a new tree rooted at v
    DFS(v)
```

As with BFS, we should store more information as we do this: store parent pointers, distinguish tree edges and non-tree edges. (see additions above)

Runtime: $O(n + m)$ (same argument as for DFS)
DFS gives rich structure:

- partition into separate trees (connected components)
- edge classification
- vertex order: order of discovery, order of “finishing” (more on this for directed graphs)

**Lemma** DFS($v_0$) reaches all vertices connected to $v_0$.

**Proof** Suppose there is a path $v_0v_1\cdots v_f$.
- Look at the last vertex discovered: $v_i$.
- Then we explore all neighbours of $v_i$ including $v_{i+1}$ (more formal by induction).

**Lemma** All non-tree edges join ancestor and descendant.

**Proof** $v$ is an ancestor of $u$ and $u$ is a descendant of $v$.
- Cannot have edge $(x, y)$: Suppose $x$ discovered first. Then in DFS($x$) we examine neighbour $y$. So $y$ is discovered before $x$ finishes and $y$ appears in subtree of $x$.

Ex. Enhance code to number the connected components and record the component of each vertex.
Enhancing DFS to compute discover and finish times.

\[
\begin{align*}
\text{time} & := 1 \quad \# \text{ initialization} \\
\text{DFS}(v) & \\
\text{mark}(v) & := \text{discovered} \\
\text{discover}(v) & := \text{time}; \quad \text{time} := \text{time} + 1 \\
\text{for } u \in \text{AdjacencyList}(v) \text{ do} & \\
\quad \text{if } u \text{ is undiscovered then} & \\
\quad \quad \text{DFS}(u) & \\
\quad \text{od} & \\
\text{finish}(v) & := \text{time}; \quad \text{time} := \text{time} + 1
\end{align*}
\]

Abbreviate \( d(v) = \text{discover}(v) \) and \( f(v) = \text{finish}(v) \).
Discover and finish times form a parenthesis system.
If \( d(v) < d(u) \) then

\[
\begin{bmatrix}
\text{d}(v) & \text{d}(u) & \text{f}(u) & \text{f}(v)
\end{bmatrix}
\begin{bmatrix}
\text{d}(v) & \text{f}(v) & \text{d}(u) & \text{f}(u)
\end{bmatrix}
\quad \text{or} \quad \\
\]

because interval \( d(v), f(v) \) is time on stack.
DFS to find 2-connected components

Graph $G$

$G$ is connected but removing one vertex $b$ or $e$ disconnects it.

Definition: $v$ is a cut vertex if removing $v$ makes $G$ disconnected.

Cut vertices are bad in networks.

DFS from $e$

Next day: finding cut vertices using DFS