Recall from last day
Exploring graphs — visit all nodes, or all nodes reachable from some "source"

Breadth First

BFS

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

1 → 2, 3, 6, 8
2 → 1, 4, 5
3 → 4, 5, 6
...

BFS tree

order in which vertices are discovered

1, 2, 3, 6, 8, 4, 5, 7
1's neighbours 2's 6's

pseudocode — see lecture 11
run time O(n + m)

Applications:
- shortest path from root vertex $v_0$ to any node $v$
  $=$ level of $v$.
- testing if a graph has a cycle
- testing if graph is bipartite.
BFS to test bipartiteness

\[ G \text{ is bipartite if } V \text{ can be partitioned into } V_1 \cup V_2 \quad (V_1 \cap V_2 = \emptyset) \]

s.t. every edge has one end in \( V_1 \) and one end in \( V_2 \)

Note that a bipartite graph cannot have an odd cycle.
Run BFS. \( V_i = \text{odd levels} \quad V_2 = \text{even levels} \).  

Test if this works (check edges) \( \rightarrow \) this can be done during BFS  

- if YES \( \Rightarrow G \) is bipartite  
- if NO then there is an edge \((u, v)\) with  
  \( u, v \) both in \( V_i \) \((i = 1 \text{ or } 2)\)  
By Ex. 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 same level, say \(k\).  
  
Let \( z = \) least common ancestor of \( u, v \).  

\[
\text{Cycle formed by path } (u, z) \text{ path } (z, v) \text{ path } (u, v) \text{ has length } 2t + 1 \text{ - odd}
\]

Then \( G \) is not bipartite.  

This proves:  

**Lemma** \( G \) is bipartite iff it has no odd cycle.  

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

Bold search - go as far as you can; when there's nothing new to discover, retrace your steps to find something new.

Adjacency lists

- **a**: b, c, d
- **b**: e, g, d, c, a (note e is first)
- **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:
abefgdc

Order of finishing:
fgcedb

Note: orders depend on order in Adj. lists.

Use a stack to store vertices that have been discovered but must still be explored.
As a recursive program (stack is implicit):

\[ \text{DFS}(v) \quad \text{-- this will explore } v \]
mark \( (w) \leftarrow \text{discovered} \)
for each vertex \( u \) in Adjacency list \( (v) \) do
if \( u \) is undiscovered then
   \[ \text{DFS}(u); \text{parent}(w) \leftarrow v; (u,v) \text{ is a tree edge} \]
else \( (u,v) \) is a non-tree edge, unless \( u = \text{parent}(v) \)
end
mark \( (v) \leftarrow \text{finished} \)

\[ \text{DFS}(v) \]

mark all vertices undiscovered
for all vertices \( v \) -- this handles multiple components
if \( v \) is undiscovered -- start new tree rooted at \( v \)
   \[ \text{DFS}(v); \]
As with BFS, we should store more info as we do this:
- Store parent pointers, distinguish tree edges and non-tree edges (see changes above)

Run-time: \( O(n+m) \) (same argument as for BFS)

DFS gives rich structure:
- partition into separate trees
- 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 \( v_0, v_1, \ldots, v_f \).

Look at last vertex discovered \( v_i \).

Then we explore all neighbours of \( v_i \) including \( v_{i+1} \)
  (more formal by induction)

EX. Enhance code to number the connected components and record the component of each vertex.

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

\[ u \text{ is an ancestor of } v \]
\[ u \text{ is a descendant of } v \]

Cannot have edge \((x, y)\):

Suppose \(x\) discovered first

Then in \(\text{DFS}(x)\) we examine neighbor \(y\).

So \(y\) is discovered before \(x\) finishes and \(y\) appears in subtree of \(x\).

Enhancing DFS to compute discover & finish times

Initialize time \(< 1.

\[ \text{DFS}(u) \]

\[ \text{mark}(u) \leftarrow \text{discovered} \]
\[ \text{discover}(u) \leftarrow \text{time} ; \text{time} \leftarrow \text{time} + 1 \]

for each vertex \(u\) in Adj. list of \(u\) do

if \(u\) is undiscovered then

\[ \text{DFS}(u) \]

end

\[ \text{finish}(u) \leftarrow \text{time} ; \text{time} \leftarrow \text{time} + 1 \]

Abbreviate \(d(u) = \text{discover}(u)\), \(f(u) = \text{finish}(u)\).

Discover & finish times form a parenthesis system.

If \(d(v) < d(u)\) then

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

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

This graph is connected but removing one vertex b or e disconnects it.

A vertex is a cut vertex if removing it makes the graph disconnected. Cut vertices are bad in networks.

**Biconnected Components**

DFS from vertex e shown above

Next lecture: finding cut vertices using DFS