Today

1. Graphs
2. Depth First Search
3. Reachability

Scheduling: coloring.

Directed acyclic graphs.

Exam Slot 1.
Exam Slot 2.
Exam Slot 3.

Graph \( G = (V, E) \).

Matrix Representation.

\[
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 0
\end{pmatrix}
\]

\( V = \{0, 1, 2, 3, 4, 5\} \)
\( E = \{(0,1), (0,2), (0,5), (1,3)\} \)

Adjacency List

<table>
<thead>
<tr>
<th>( u )</th>
<th>( V(u) )</th>
<th>( E(u) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 2, 5</td>
<td>1, 4</td>
</tr>
<tr>
<td>1</td>
<td>0, 2, 3, 4, 5</td>
<td>0, 1</td>
</tr>
<tr>
<td>2</td>
<td>1, 3, 4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1, 3, 5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0, 1, 2, 4</td>
<td>5</td>
</tr>
</tbody>
</table>

Test your understanding..

Adiacency list of node 0?
(A) 0 : 1
(B) 0 : 1, 2
(C) 0 : 2

How many edges?
(A) 2
(B) 3
(C) 4

2 entries for each edge!

Exploring a maze.

Theseus: Wants to find the minatour in the maze.
Theseus has access to a Ball of Thread and a Chalk!
Explore a room: Mark room with chalk.
For each exit,
- Look through exit. If marked, next exit.
- Otherwise go in room unwind thread.
- Explore that room.
- Wind thread to go back to “previous” room.

Where is the minatour?
Reachability problem in a Graph.

Problem: Find out which nodes are reachable from A.
Need digital analogues of the chalk and ball of thread.
We will use array (visited) for chalk and stack for thread.

Explore.

Chalk.
Stack is Thread.

Explore builds tree.
Tree and back edges.

Explore(v):
1. Set visited[v] := true
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w).

Correctness.

Explore(v):
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w).

Property:
All and only nodes reachable from A are reached by explore.

Only: when u visited.
stack contains nodes in a path from a to u.
All: if a node u is reachable.
there is a path to it. Assume: u not found.

z is explored. w is not!
Explore (z) would explore(w), or it was already explored!
Contradiction.

Running Time.

Explore(v):
2. For each edge (v,w) in E
3. if not visited[w]: Explore(w).

How to analyse?
Let n = |V|, and m = |E|.
T(n, m) ≤ (d) T(n − 1, m) + O(d)
Exponential

Don't use recurrence!

Depth first search.

Process whole graph.
DFS(G)
1: For each node u,
2: visited[u] = false.
3: For each node u,
4: if not visited[u] explore(u)

Running time: O(|V| + |E|).
Intuitively: tree for each "connected component".
Several trees or Forest! Output connected components?

DFS and connected components.

Change explore a bit:
explore(v):
2. previsit(v)
3. For each edge (v,w) in E
4. if not visited[w]: explore(w).
5. postvisit(v)

previsit(v):

DFS(G):
0. Set ccnum := 0.
1. for each v in V:
2. if not visited[v]
3. explore(v)
4. ccnum = ccnum+1

Each node will be labelled with connected component number.
Runtime: O(|V| + |E|).

Connected Components.
Introspection: pre/post.

previsit(v):
2. clock := clock + 1

postvisit(v):
2. clock := clock + 1

DFS(G):
0. Set clock := 0.

Clock: goes up to 2 times number of vertices.
First pre: 0

Property:
For any two nodes, u and v, \([\text{pre}(u), \text{post}(u)]\) and \([\text{pre}(v), \text{post}(v)]\) are either disjoint or one is contained in other.
Interval is “clock interval on stack.”
Either both on stack at some point (contained) or not (disjoint.)
Let’s just watch it work!

Directed graphs.

\[ G = (V, E) \]

vertices \(V\).

edges \(E \subseteq V \times V\).

Edge: \((u, v)\)

From \(u\) to \(v\).

Tail – \(u\)

Head – \(v\)

Cycle in a directed graph?

Fast algorithm for finding out whether directed graph has cycle?

For each edge \((u, v)\) remove, check if \(v\) is connected to \(u\)

\(O(|E|(|V| + |E|))\)?

Linear Time (i.e. \(O(|V| + |E|)\))? 

Directed Acyclic Graph

Thm: A graph has a cycle if and only if there is a back edge in any DFS.

Proof:
We just saw: Back edge \(\Rightarrow\) cycle!

In the other direction: Assume there is a cycle \(v_0 \rightarrow v_1 \rightarrow v_2 \cdots \rightarrow v_k \rightarrow v_0\)

Assume that \(v_0\) is the first node explored in the cycle

(without loss of generality since can renumber vertices.)

When \(\text{explore}(v_0)\) returns all nodes on cycle explored.

All \(\text{int}(v)\) in \(\text{int}(v_0)\): \(\Rightarrow\) \((v_0, v_k)\) is a back edge.

Cycle \(\Rightarrow\) back edge!

Testing for cycle.

Thm: A graph has a cycle if and only if there is a back edge in any DFS.

Example: Pre/Post numbering.

```plaintext
Edge (u, v) is tree edge iff \([\text{pre}(v), \text{post}(v)] \subset [\text{pre}(u), \text{post}(u)]\).

u on stack before v.

Edge (u, v) is back edge iff \([\text{pre}(u), \text{post}(u)] \subset [\text{pre}(v), \text{post}(v)]\).

v on stack before u on stack. Path from v to u!

No edge between u and v if disjoint intervals.
```

Fast checking algorithm.

Thm: A graph has a cycle if and only if there is a back edge.

Algorithm ??

Run DFS.

\(O(|V| + |E|)\) time.

For each edge \((u, v)\): is \(\text{int}(u)\) in \(\text{int}(v)\)?

\(O(|E|)\) time.

\(O(|V| + |E|)\) time algorithm for checking if graph is acyclic!

“Hello” before “Goodbye”

No cycles! Can tell in linear time!

Really want to find ordering for build!
Linearize.

**Topological Sort**: For $G = (V, E)$, find ordering of all vertices where each edge goes from earlier vertex to later in acyclic graph.

**Topological Sort Example.**

```
A  E  F  B  G  D  C

A linear order: A, E, F, B, G, D, C
In DFS: When is A popped off stack?
Last! When is E popped off? second to last. ...
```

Topological Sort: DFS

Last post order should..
(A) be first in linearization!
(B) be last in linearization!

(A). First!

**Property**: Every edge in a DAG $(u, v)$ has $\text{post}(u) > \text{post}(v)$.

*Proof*: No back edges in DAG.

Tree and Forward edge $(u, v)$:

- int$(u)$ contains int$(v)$: $\text{pre}[u], \text{pre}[v], \text{post}[v], \text{post}[u]$
- Cross edge $(u, v)$: int$(u) > \text{int}(v) \Rightarrow \text{post}[u] > \text{post}[v]$

Topological Sort: linearize.

**Property**: Every edge in a DAG $(u, v)$ has $\text{post}(u) > \text{post}(v)$.

Top Sort: output in reverse post order number.
Runtime: $O(|V| + |E|)$.

..procedure postvisit outputs during DFS

```python
def postvisit(v): result.append(v).
```

..reverse result.