DFS, Topological Sort
• Mid-semester survey which can be accessed at https://illinois.edu/sb/sec/7058301.
• Midterm in two weeks!
• Review session the Thursday before.
How to traverse a graph?

**TRAVERESE(s):**
- put $s$ into the bag
- while the bag is not empty
  - take $v$ from the bag
  - if $v$ is unmarked
    - mark $v$
    - for each edge $vw$
      - put $w$ into the bag

stack = LIFO (DFS)
Queue = FIFO (BFS)
Priority Queue = lightest out
Random, etc
DFS

\[
\text{DFS}(v): \\
\text{if } v \text{ is unmarked} \\
\text{mark } v \\
\text{for each edge } vw \\
\text{DFS}(w)
\]

stack = LIFO (DFS)
DFS

DFS(v):
mark v
PreVisit(v)
for each edge vw
    if w is unmarked
        parent(w) ← v
        DFS(w)
PostVisit(v)

check if a node is marked before recursively exploring it. DFS(v) called once for each v.

Disconnected graphs?
DFS

**DFSAll(G):**

- **Preprocess(G)**
- for all vertices \( v \)
  - unmark \( v \)
- for all vertices \( v \)
  - if \( v \) is unmarked
  - DFS(\( v \))

check if a node is marked before recursively exploring it. DFS(\( v \)) called once for each \( v \).

Disconnected graphs?

How to label all the vertices in each component with same label?
DFS

I have found no connected components yet

CountAndLabel(G):

\[
\begin{align*}
\text{count} & \leftarrow 0 \\
\text{for all vertices } v & \\
\text{unmark } v & \\
\text{for all vertices } v & \\
\text{if } v \text{ is unmarked} & \\
\text{count} & \leftarrow \text{count} + 1 \\
\text{LabelComponent}(v, \text{count}) & \\
\text{return count} & 
\end{align*}
\]

LabelComponent(v, count):

\[
\begin{align*}
\text{mark } v & \\
\text{comp}(v) & \\
\text{for each } w & \\
\text{if } w \text{ is unmarked} & \\
\text{LabelComponent}(w, \text{count}) & 
\end{align*}
\]

Every time I find a new component increase counter
DFS

COUNTANDLABEL(G):

1. count ← 0
2. for all vertices v
   a. unmark v
3. for all vertices v
   a. if v is unmarked
      i. count ← count + 1
      ii. LABELCOMPONENT(v, count)
4. return count

LABELCOMPONENT(v, count):

1. mark v
2. comp(v) ← count
3. for each edge vw
   a. if w is unmarked
      i. LABELCOMPONENT(w, count)

I have found no connected components yet

Every time I find a new component increase counter

Label each vertex in the component with the index of the component
Sometimes I want to compute an order of the vertices in a graph which is consistent with pre or postorder traversal e.g. DP
Preorder/Postorder

\texttt{PrePostLabel}(G):
for all vertices \( v \)
\hspace{1em} \unmark v
\hspace{1em} clock \leftarrow 0
for all vertices \( v \)
\hspace{1em} \text{if } \( v \) \text{ is unmarked}
\hspace{3em} clock \leftarrow \text{LABELCOMPONENT}(v, \text{clock})

\texttt{LABELCOMPONENT}(v, \text{clock}):
\hspace{1em} \text{mark } v
\hspace{1em} pre(v) \leftarrow \text{clock}
\hspace{1em} clock \leftarrow \text{clock} + 1
for each edge \( vw \)
\hspace{3em} \text{if } \( w \) \text{ is unmarked}
\hspace{5em} clock \leftarrow \text{LABELCOMPONENT}(w, \text{clock})
\hspace{2em} \textcolor{red}{\text{post}(v) \leftarrow \text{clock}}
\hspace{2em} \text{clock} \leftarrow \text{clock} + 1
\hspace{2em} \text{return } \text{clock}
DFS

DFS(\(v\));
mark \(v\)
\(\text{PreVisit}(v)\)
for each edge \(vw\)
if \(w\) is unmarked
\(\text{parent}(w) \leftarrow v\)
DFS(\(w\))
\(\text{PostVisit}(v)\)

\textbf{Preprocess} (\(G\)):
\(\text{clock} \leftarrow 0\)

\textbf{PreVisit} (\(v\)):
\(\text{pre}(v) \leftarrow \text{clock}\)
\(\text{clock} \leftarrow \text{clock} + 1\)

\textbf{PostVisit} (\(v\)):
\(\text{post}(v) \leftarrow \text{clock}\)
\(\text{clock} \leftarrow \text{clock} + 1\)
DFS

pre(v) pre(w) post(w) post(v) pre(u) post(u)

(v,w) edge: intervals are nested

Different way of encoding the DFS tree for the recursive algorithm

**Preprocess(G):**

\[ \text{clock} \leftarrow 0 \]

**PreVisit(v):**

\[ \text{pre}(v) \leftarrow \text{clock} \]
\[ \text{clock} \leftarrow \text{clock} + 1 \]

**PostVisit(v):**

\[ \text{post}(v) \leftarrow \text{clock} \]
\[ \text{clock} \leftarrow \text{clock} + 1 \]
DFS

**COUNTANDLABEL(G):**

```plaintext
count ← 0
for all vertices v
  unmark v
for all vertices v
  if v is unmarked
    count ← count + 1
    LABELCOMPONENT(v, count)
return count
```

**LABELCOMPONENT(v, count):**

```plaintext
mark v
comp(v) ← count
for each edge vw
  if w is unmarked
    LABELCOMPONENT(w, count)
```

assumes the graph is undirected
What about directed graphs?
When is a graph connected?
DFS

In directed graph vertex u can reach vertex v iff there is a directed path from u to b.

\[ \text{u and v are strongly connected if u can reach v and v can reach u} \]

\[ \text{directed cycle!} \]
DFS for directed graphs

**DFSAll(G):**
for all vertices \( v \)
unmark \( v \)
for all vertices \( v \)
if \( v \) is unmarked
DFS(\( v \))

**DFS(\( v \)):**
mark \( v \)
\text{PREVISIT(} v \text{)}
\text{for each edge } v \rightarrow w
if \( w \) is unmarked
DFS(\( w \))
\text{POSTVISIT(} v \text{)}
Think of two extremes

1) There are no directed cycles (DAG)
2) Every two vertices have a directed cycle between them

How do I decide if a graph is a DAG or strongly connected?
Is it a DAG?

**IsAcyclic(G):**
- add vertex s
- for all vertices $v \neq s$
  - add edge $s \rightarrow v$
- $status(v) \leftarrow \text{NEW}$
- return $\text{IsAcyclicDFS}(s)$

**IsAcyclicDFS(v):**
- $status(v) \leftarrow \text{ACTIVE}$
- for each edge $v \rightarrow w$
  - if $status(w) = \text{ACTIVE}$
    - return False
  - else if $status(w) = \text{NEW}$
    - if $\text{IsAcyclicDFS}(w) = \text{FALSE}$
      - return False
  - $status(v) \leftarrow \text{DONE}$
- return True

A vertex can be NEW, ACTIVE, or DONE.

A directed cycle if and only if I reach an ACTIVE vertex from an ACTIVE vertex.
a vertex can be
NEW, ACTIVE or DONE

directed cycle if and only if
I reach an ACTIVE vertex
from an ACTIVE vertex
Is it a DAG?

\[
\text{IsAcyclicDFS}(v): \\
\text{status}(v) \leftarrow \text{ACTIVE} \\
\text{for each edge } v \rightarrow w \\
\quad \text{if } status(w) = \text{ACTIVE} \\
\quad \quad \text{return FALSE} \\
\quad \text{else if } status(w) = \text{NEW} \\
\quad \quad \quad \text{if IsAcyclicDFS}(w) = \text{FALSE} \\
\quad \quad \quad \quad \text{return FALSE} \\
status(v) \leftarrow \text{DONE} \\
\text{return TRUE}
\]

a vertex can be NEW, ACTIVE or DONE

directed cycle if and only if I reach an ACTIVE vertex from an ACTIVE vertex
Is it a DAG?

\[
\text{IsAcyclicDFS}(v):
\]

\[
\begin{align*}
\text{status}(v) & \leftarrow \text{Active} \\
\text{for each edge } v \rightarrow w & \quad \text{if } \text{status}(w) = \text{Active} \quad \text{return False} \\
\text{else if } \text{status}(w) = \text{New} & \quad \text{if IsAcyclicDFS}(w) = \text{False} \quad \text{return False} \\
\text{status}(v) & \leftarrow \text{Done} \\
\text{return True}
\end{align*}
\]

A vertex can be NEW, ACTIVE or DONE.

A directed cycle if and only if I reach an ACTIVE vertex from an ACTIVE vertex.
Is it a DAG?

**IsAcyclicDFS(v):**

- `status(v) ← Active`
- for each edge `v→w`
  - if `status(w) = Active`
    - return `FALSE`
  - else if `status(w) = NEW`
    - if `IsAcyclicDFS(w) = FALSE`
      - return `FALSE`
- `status(v) ← Done`
- return `TRUE`

A vertex can be
NEW, ACTIVE or DONE

directed cycle if and only if
I reach an ACTIVE vertex
from an ACTIVE vertex
Is it a DAG?

\[
\text{IsAcyclicDFS}(v):
\]

\[
\begin{align*}
\text{status}(v) & \leftarrow \text{ACTIVE} \\
\text{for each edge } v \rightarrow w & \\
\quad \text{if } \text{status}(w) = \text{ACTIVE} & \\
\quad \quad \text{return FALSE} & \\
\quad \text{else if } \text{status}(w) = \text{NEW} & \\
\quad \quad \quad \text{if IsAcyclicDFS}(w) = \text{FALSE} & \\
\quad \quad \quad \quad \text{return FALSE} & \\
\text{status}(v) & \leftarrow \text{DONE} \\
\text{return TRUE}
\end{align*}
\]

A vertex can be NEW, ACTIVE or DONE.

Directed cycle if and only if I reach an ACTIVE vertex from an ACTIVE vertex.
Is it a DAG?

\[
\text{IsAcyclic}(G): \\
\text{add vertex } s \\
\text{for all vertices } v \neq s \\
\text{add edge } s \rightarrow v \\
\text{status}(v) \leftarrow \text{NEW} \\
\text{return IsAcyclicDFS}(s)
\]

\[
\text{IsAcyclicDFS}(v): \\
\text{status}(v) \leftarrow \text{ACTIVE} \\
\text{for each edge } v \rightarrow w \\
\quad \text{if status}(w) = \text{ACTIVE} \\
\quad \text{return FALSE} \\
\quad \text{else if status}(w) = \text{NEW} \\
\quad \quad \text{if IsAcyclicDFS}(w) = \text{FALSE} \\
\quad \quad \text{return FALSE} \\
\quad \text{status}(v) \leftarrow \text{DONE} \\
\text{return TRUE}
\]

\[\text{time } O(|V|+|E|)\]

Why do I want to decide if graph is DAG?
Why do I want to find if DAG?

Make:
• huge database of files
• nodes are files,
• edges between files x and y: if I change file x, I need to recompile file y
• Sometimes people create file system that have cycles!
• Make has to find those cycles and prevent that.
• It also has to execute the compilation commands in the correct order in order to produce the final executable.
• Just DFS of dependency graph
• See DP memoization
Topological Sort

1 → 4 → 3 → 7 → 5 → 6 → 2
Topological Sort

\[ \text{TOPOLOGICALSORT}(G) : \]
\[ n \leftarrow |V| \]
\[ \text{for } i \leftarrow 1 \text{ to } n \]
\[ v \leftarrow \text{any source in } G \]
\[ S[i] \leftarrow v \]
\[ \text{delete } v \text{ and all edges leaving } v \]
\[ \text{return } S[1..n] \]
Topological Sort

\[
\text{\textbf{TopologicalSort}}(G) :
\]
\[
n \leftarrow |V|
\]
\[
\text{for } i \leftarrow 1 \text{ to } n
\]
\[
v \leftarrow \text{any source in } G
\]
\[
S[i] \leftarrow v
\]
\[
delete v \text{ and all edges leaving } v
\]
\[
\text{return } S[1..n]
\]
Topological Sort

**TOPOLOGICALSORT**(G):

\[
\begin{align*}
n & \leftarrow |V| \\
\text{for } i & \leftarrow 1 \text{ to } n \\
\quad & \quad \nu \leftarrow \text{any source in } G \\
\quad & \quad S[i] \leftarrow \nu \\
\quad & \quad \text{delete } \nu \text{ and all edges leaving } \nu \\
\text{return } S[1 \ldots n]
\end{align*}
\]
Topological Sort

**TopologicalSort**$(G)$:

$n \leftarrow |V|$
for $i \leftarrow 1$ to $n$
    $v \leftarrow$ any source in $G$
    $S[i] \leftarrow v$
    delete $v$ and all edges leaving $v$
return $S[1..n]$
Topological Sort

\[
\text{TOPOLOGICALSORT}(G) : \\
\text{begin} \\
\quad n \leftarrow |V| \\
\quad \text{for } i \leftarrow 1 \text{ to } n \\
\quad \quad v \leftarrow \text{any source in } G \\
\quad \quad S[i] \leftarrow v \\
\quad \quad \text{delete } v \text{ and all edges leaving } v \\
\quad \text{return } S[1..n] \\
\text{end}
\]
Topological Sort

\[
\text{TOPOLOGICALSORT}(G): \\
\quad n \leftarrow |V| \\
\quad \text{for } i \leftarrow 1 \text{ to } n \\
\quad \quad v \leftarrow \text{any source in } G \\
\quad \quad S[i] \leftarrow v \\
\quad \quad \text{delete } v \text{ and all edges leaving } v \\
\quad \text{return } S[1..n]
\]
Topological Sort

\[
\text{TOPOLOGICALSORT}(G): \\
\begin{align*}
n & \leftarrow |V| \\
\text{for } i & \leftarrow 1 \text{ to } n \\
& \quad v \leftarrow \text{any source in } G \\
& \quad S[i] \leftarrow v \\
& \quad \text{delete } v \text{ and all edges leaving } v \\
\text{return } S[1..n]
\end{align*}
\]
Topological Sort

\[ \text{TOPOLOGICAL SORT}(G) : \]
\[ n \leftarrow |V| \]
\[ \text{for } i \leftarrow 1 \text{ to } n \]
\[ v \leftarrow \text{any source in } G \]
\[ S[i] \leftarrow v \]
\[ \text{delete } v \text{ and all edges leaving } v \]
\[ \text{return } S[1..n] \]
Topological Sort

```
TOPOLOGICALSORT(G):
    n ← |V|
    for i ← n down to 1
        v ← any sink in G
        S[i] ← v
        delete v and all edges entering v
    return S[1..n]
```

- How to find sink?
- Naively $O(n)$ for each sink, total $O(n^2)$
- For source, even worse, cause the adjacency list representation doesn’t have pointers for incoming edges
- $O(n^2|E|)$ naively.

Dependency graph of software

Running time?
Topological Sort

```
TOPOLOGICALSORT(G):
    n ← |V|
    for i ← n down to 1
        v ← any sink in G
        S[i] ← v
        delete v and all edges entering v
    return S[1..n]
```

- Could do it with priority queue of out degrees in $O(|V|+|E|)$.
- Reverse the dag to help delete edges etc…
- Is there another way?

Dependency graph of software

Running time?
Topological Sort

• **Claim**: First vertex DONE in DFS below is sink.

\[
\text{IsAcyclic}(G):
\begin{align*}
\text{add vertex } s \\
\text{for all vertices } v \neq s \\
\text{add edge } s \rightarrow v \\
\text{status}(v) \leftarrow \text{NEW} \\
\text{return } \text{IsAcyclicDFS}(s)
\end{align*}
\]

\[
\text{IsAcyclicDFS}(v):
\begin{align*}
\text{status}(v) \leftarrow \text{ACTIVE} \\
\text{for each edge } v \rightarrow w \\
& \quad \text{if } \text{status}(w) = \text{ACTIVE} \\
& \quad \quad \text{return } \text{FALSE} \\
& \quad \text{else if } \text{status}(w) = \text{NEW} \\
& \quad \quad \text{if } \text{IsAcyclicDFS}(w) = \text{FALSE} \\
& \quad \quad \quad \text{return } \text{FALSE} \\
& \quad \text{status}(v) \leftarrow \text{DONE} \\
\text{return } \text{TRUE}
\end{align*}
\]
Topological Sort

• **Claim**: First vertex DONE in DFS below is sink.

• **Proof**: Assume, towards contradiction that \( v \) is DONE first but there is \( w: v \rightarrow w \).

There are 3 cases:

• \( w \) is NEW
• \( w \) ACTIVE
• \( w \) DONE
Topological Sort

• **Claim**: First vertex DONE in DFS below is sink.

• **Proof**:

Assume, towards contradiction that \( v \) is DONE first but there is \( w: v \rightarrow w \).

There are 3 cases:

• \( w \) is NEW

  \( w \) would be marked active and then be DONE first (contradiction)

• \( w \) ACTIVE

  \( v \) is active still, so there is a cycle

• \( w \) DONE

  (contradiction of DAG)

\( v \) is DONE first (contradiction)
Topological Sort

- **Claim:** The order by which vertices are DONE in DFS is a reverse topological order (proof?).

- Can just sort those vertices in stack and pop them for topological order, put them in sorted array.

```plaintext
TOPOLOGICALSORT(G):
  add vertex s
  for all vertices v ≠ s
    add edge s→v
    status(v) ← NEW
  TopoSortDFS(s)
  for i ← 1 to V
    S[i] ← Pop
  return S[1..V]

TopoSortDFS(v):
  status(v) ← ACTIVE
  for each edge v→w
    if status(w) = NEW
      PROCESSBACKWARDDFS(w)
    else if status(w) = ACTIVE
      fail gracefully
  status(v) ← DONE
  Push(v)
  return TRUE
```
Topological Sort

- **Claim**: The order by which vertices are DONE in DFS is a reverse topological order (proof?).
- Overkill, all I need is to be able to do some computation so that we respect dependencies

```plaintext
TOPOLOGICALSORT(G):
  add vertex s
  for all vertices \( v \neq s \)
    add edge \( s \rightarrow v \)
  status(v) ← NEW
  TopoSortDFS(s)
  for \( i \leftarrow 1 \) to \( V \)
    \( S[i] \leftarrow \text{POP} \)
  return \( S[1..V] \)
```

```
TOPOSortDFS(v):
  status(v) ← Active
  for each edge \( v \rightarrow w \)
    if status(w) = New
      PROCESSBACKWARDDFS(w)
    else if status(w) = Active
      fail gracefully
  status(v) ← Done
  \text{Push}(v)
  return True
```
Topological Sort

**ProcessBackward(G):**
- Add vertex \( s \)
- For all vertices \( v \neq s \)
  - Add edge \( s \rightarrow v \)
  - \( \text{status}(v) \leftarrow \text{New} \)

**ProcessBackwardDFS(v):**
- \( \text{status}(v) \leftarrow \text{Active} \)
- For each edge \( v \rightarrow w \)
  - If \( \text{status}(w) = \text{New} \)
    - \( \text{ProcessBackwardDFS}(w) \)
  - Else if \( \text{status}(w) = \text{Active} \)
    - Fail gracefully

- \( \text{status}(v) \leftarrow \text{Done} \)
- \( \text{Process}(v) \)

- The order that I want to do commutation is the order I mark things done.
- I process while I explore the node in DFS.
- Processes every node in the graph in reverse topological order.
- Check for DAG in there, unless I know it is DAG.
Topological Sort if DAG

\[
\text{processDagBackward}(G): \\
\quad \text{add vertex } s \\
\quad \text{for all vertices } v \neq s \\
\quad \quad \text{add edge } s \rightarrow v \\
\quad \text{unmark } v \\
\text{processDagBackwardDFS}(s)
\]

\[
\text{processDagBackwardDFS}(v): \\
\quad \text{mark } v \\
\quad \text{for each edge } v \rightarrow w \\
\quad \quad \text{if } w \text{ is unmarked} \\
\quad \quad \quad \text{processDagBackwardDFS}(w) \\
\text{process}(v)
\]

Where have we seen this before?
For $i < j$

$$LIS(i, j) = \begin{cases} 
0 & \text{if } j > n \\
LIS(i, j + 1) & \text{if } A[i] \geq A[j] \\
\max\{LIS(i, j + 1), 1 + LIS(j, j + 1)\} & \text{otherwise}
\end{cases}$$
DP=DFS

Memoized recursion is DFS
Dynamic Programming uses topological sort
Given DAG and I am interested in finding the longest path.
Given DAG and I am interested in finding the longest path.

\[
LLP(s,t) = \begin{cases} 
0 & \text{if } s=t \\
\max_{s \rightarrow v} \{1 + LLP(v,t)\} & \text{o.w} \\
-\infty & \text{s sink}
\end{cases}
\]

define \( \max \emptyset = -\infty \)

t is constant throughout
Longest Path in DAG

Given DAG and I am interested in finding the longest path.

\[ LLP(s,t) = \begin{cases} 
0 & \text{if } s = t \\
\max_{s \rightarrow v} \{1 + LLP(v,t)\} & \text{o.w}
\end{cases} \]

what data structure to use? thga graph! Memoize \( LLP(s,t) \) into node \( s \)!
Longest Path in DAG

Given DAG and I am interested in finding the longest path.

\[ LLP(s,t) = \begin{cases} 0 & \text{if } s = t \\ \max_{s \rightarrow v} \{1 + LLP(v,t)\} & \text{o.w} \\ -\infty & \text{s sink} \end{cases} \]

What order?
Reverse topological sort order!
Longest Path in DAG

```
LONGESTPATH(s, t):
    if s = t
        return 0
    if LLP(s) is undefined
        LLP(s) ← ∞
        for each edge s→v
            LLP(s) ← max{LLP(v), ℓ(s→v) + LONGESTPATH(v, t)}
    return LLP(s)
```

What is reverse topological order?
Just do DFS for reverse topological order!
it is also the naive recursive algorithm
Strong Connectivity

In directed graph vertex $u$ can reach vertex $v$ iff there is a directed path from $u$ to $v$

$\text{reach}(u) = \text{set of vertices } u \text{ can reach}$

$u$ and $v$ are strongly connected if $u$ can reach $v$ and $v$ can reach $u$
Strong Connectivity, SCC

- Strong connectivity is an equivalence relation
- Equivalence classes are called strongly connected components
- If G has a single strongly connected component: strongly connected
Strong Connectivity, SCC

- Strong connectivity is an equivalence relation
- Equivalence classes are called strongly connected components
- If G has a single strongly connected component: strongly connected
- When is G a DAG?
- Every SCC is a single vertex
Strong Connectivity, SCC

- How to compute SCC of vertex $u$ in $O(|V|+|E|)$ time?
  
  DFS($G,u$) gives us Reach($u$)
  
  DFS($G^{rev},u$) gives us all the stuff that can reach $u$
  
  Take intersection of both for SCC
Strong Connectivity, SCC

• How to compute SCC of vertex u in $O(|V|+|E|)$ time?
• Compute Reach(u) with DFS
• Compute $\text{Reach}^{-1}(u) = \{v: u \text{ is in Reach}(v)\}$ with DFS on reverse graph
• SCC is the intersection of the two sets.
• How to compute all SCC of a graph?
• Naive: $O(|V||E|)$ time.
• Can we do better?
For every directed graph $G$, $scc(G)$ is another (meta)graph: Contract each SCC of $G$ in one vertex and collapse parallel edges.
SCC Graph

For every directed graph G, scc(G) is another (meta)graph: Contract each SCC of G in one vertex and collapse parallel edges.
For every directed graph $G$, $\text{scc}(G)$ is another (meta)graph: Contract each SCC of $G$ in one vertex and collapse parallel edges.
For every directed graph $G$, $scc(G)$ is another (meta)graph: Contract each SCC of $G$ in one vertex and collapse parallel edges.
For every directed graph $G$, $scc(G)$ is another (meta)graph: Contract each SCC of $G$ in one vertex and collapse parallel edges.

$scc(G)$ ALWAYS A DAG!
For every directed graph $G$, $\text{scc}(G)$ is another (meta)graph: Contract each SCC of $G$ in one vertex and collapse parallel edges

- I can topologically sort the SCC.
- There is a sink SCC
- DFS starting from a vertex in $C$, reaches only vertices in $C$ and nothing else
Can compute all the SCC:

```plaintext
STRONG_COMPONENTS(G):
  count ← 0
  while G is non-empty
    count ← count + 1
    v ← any vertex in a sink component of G
    C ← ONE_COMPONENT(v, count)
    remove C and incoming edges from G
```

How to find a vertex in a sink component? (next time)