Directed Graph, DAGs and Topological Sort

Lecture 16

Part I

Connectivity on Undirectd Graphs

Connectivity Problems on Undirected Graphs

Algorithmic Problems

1. Given graph G and nodes u and v, is u *connected* to v?
2. Given G and node u, find all nodes that are connected to u.
3. Find all connected components of G.

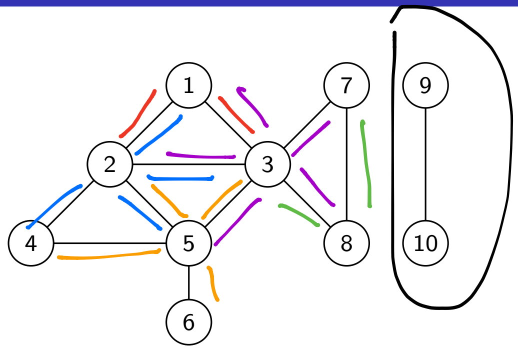
Can be accomplished in $O(m+n)$ time using **BFS** or **DFS**.
BFS and **DFS** are refinements of a basic search procedure which is good to understand on its own.

Basic Graph Search in Undirected Graphs

Given $G = (V, E)$ and vertex $u \in V$. Let $n = |V|$.

```
Explore(G, u):
    array Visited[1..n]
    Initialize: Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    while (ToExplore is non-empty) do
        Remove node x from ToExplore              ← O(1)
        for each edge (x, y) in Adj(x) do         ← O(1)      2n
            if (Visited[y] == FALSE)                          2m
                Visited[y] = TRUE
                Add y to ToExplore                ← O(1)
                Add y to S
    Output S
```

Example



bag
To Explore

Visited

1 2 3 4 5 6 7 8 9 10

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Proof Sketch.

- Once $Visited[i]$ is set to $TRUE$ it never changes. Hence a node is added only once to $ToExplore$. Thus algorithm terminates in at most n iterations of while loop.

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = con(u)$.

Proof Sketch.

- Once $Visited[i]$ is set to $TRUE$ it never changes. Hence a node is added only once to $ToExplore$. Thus algorithm terminates in at most n iterations of while loop.
- If $v \in \mathrm{con}(u)$, then $v \in S$.
- If $v \notin \mathrm{con}(u)$, then $v \notin S$.
- Thus $S = \mathrm{con}(u)$ at termination.

$\square$

Properties of Basic Search

Depth First Search (**DFS**): use stack data structure to implement the list ***ToExplore***

```
RecursiveDFS(v):
  if v is unmarked
      mark v
      for each edge vw
          RecursiveDFS(w)
```

```
IterativeDFS(s):
  Push(s)
  while the stack is not empty
      v ← Pop
      if v is unmarked
          mark v
          for each edge vw
              Push(w)
```

Properties of Basic Search

DFS and **BFS** are special case of BasicSearch.

1. Depth First Search (**DFS**): use stack data structure to implement the list ***ToExplore***
2. Breadth First Search (**BFS**): use queue data structure to implementing the list ***ToExplore***

Search Tree

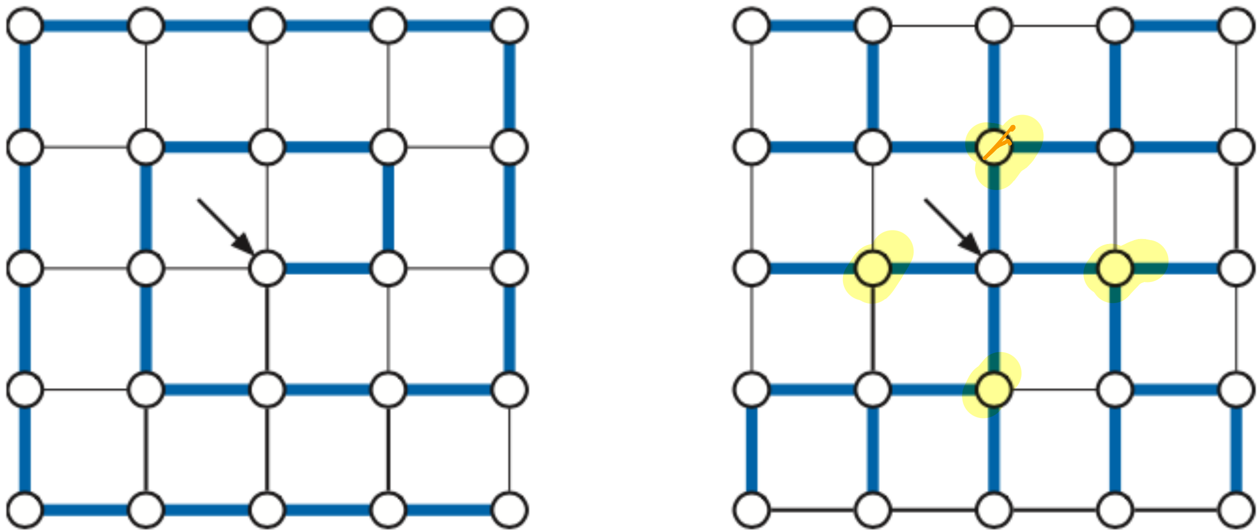
One can create a natural search tree T rooted at u during search.

```
Explore(G, u):
    array Visited[1..n]
    Initialize: Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    Make tree T with root as u
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x, y) in Adj(x) do
            if (Visited[y] == FALSE)
                Visited[y] = TRUE
                Add y to ToExplore
                Add y to S
                Add y to T with x as its parent
    Output S
```

T is a spanning tree of con(u) rooted at u

Spanning tree

A depth-first and breadth-first spanning tree.



Finding all connected components

Exercise: Modify Basic Search to find all connected components of a given graph G in $O(m+n)$ time.

1 connected component $O(m+n)$

Part II

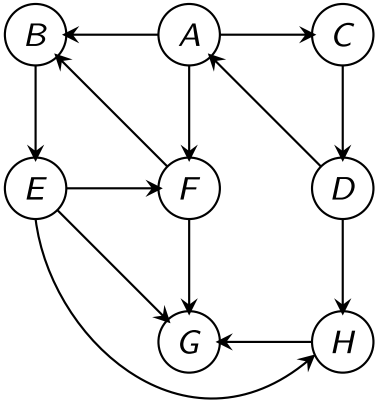
Directed Graphs

Directed Graphs

Definition

A directed graph $G = (V, E)$ consists of

1. set of vertices/nodes V and
2. a set of edges/arcs $E \subseteq V \times V$.



An edge is an *ordered* pair of vertices. (u, v) different from (v, u).

Examples of Directed Graphs

In many situations relationship between vertices is asymmetric:

1. Road networks with one-way streets.
2. Web-link graph: vertices are web-pages and there is an edge from page p to page p' if p has a link to p'. Web graphs used by Google with PageRank algorithm to rank pages.
3. Dependency graphs in variety of applications: link from x to y if y depends on x. Make files for compiling programs.
4. Program Analysis: functions/procedures are vertices and there is an edge from x to y if x calls y.

Directed Graph Representation

Graph $G = (V, E)$ with n vertices and m edges:

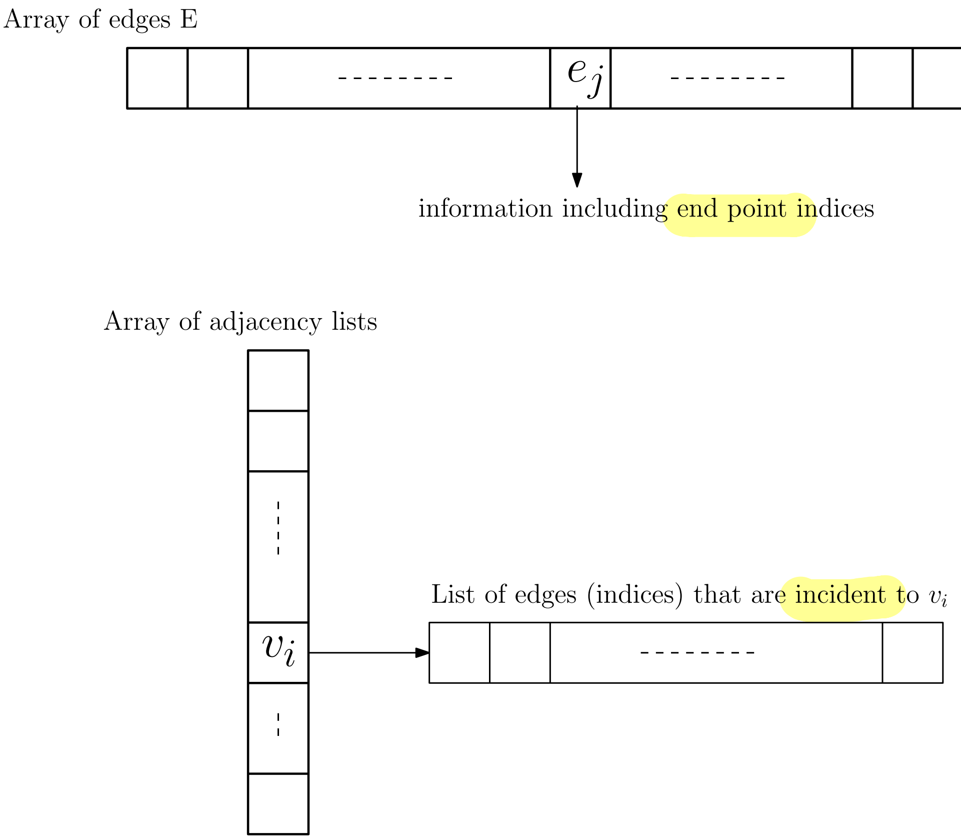
1. **Adjacency Matrix**: $n \times n$ asymmetric matrix A. $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ if $(u, v) \notin E$. $A[u, v]$ is not same as $A[v, u]$.
2. **Adjacency Lists**: for each node u, $Out(u)$ (also referred to as $Adj(u)$) and $In(u)$ store out-going edges and in-coming edges from u.

Default representation is adjacency lists.

undirected $\begin{bmatrix} & 1 \\ 1 & \end{bmatrix}$ directed $\begin{bmatrix} & 1 \\ 0 & \end{bmatrix}$

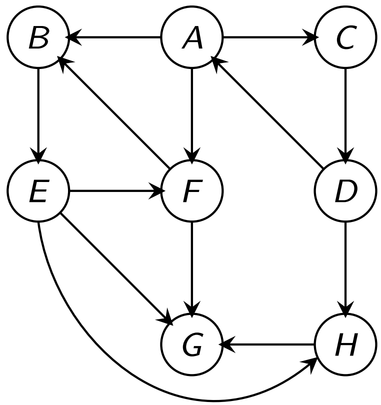
A Concrete Representation for Directed Graphs

Concrete representation discussed previously for undirected graphs easily extends to directed graphs.



Directed Connectivity

Given a graph $G = (V, E)$:



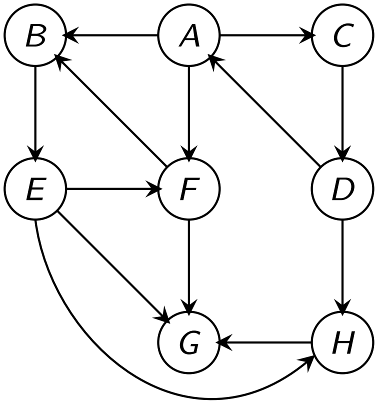
A C D ✓

A D C ✗

A **(directed) path** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from v_1 to v_k.
By convention, a single node u is a path of length 0.

Directed Connectivity

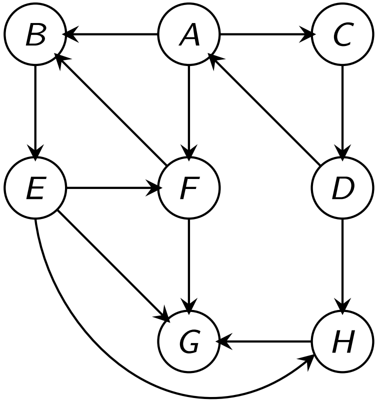
Given a graph $G = (V, E)$:



A vertex u can reach v if there is a path from u to v.

Directed Connectivity

Given a graph $G = (V, E)$:

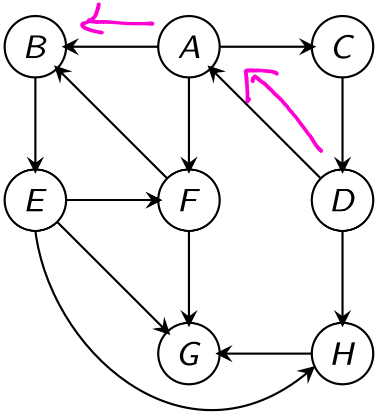


A vertex u can reach v if there is a path from u to v.

Let $\textbf{rch}(u)$ be the set of all vertices reachable from u.

Directed Connectivity

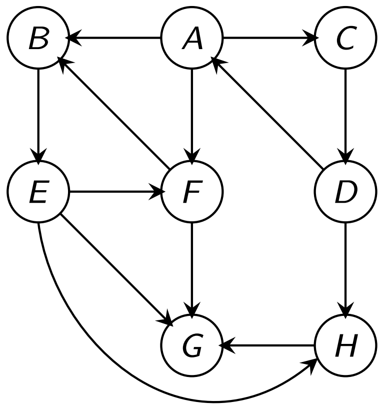
Asymmetricity: D can reach B but B cannot reach D



DAB

Directed Connectivity

Asymmetricity: D can reach B but B cannot reach D



Questions:

1. Is there a notion of connected components?
2. How do we understand connectivity in directed graphs?

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

Define relation C where uCv if u is (strongly) connected to v.

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

Define relation C where uCv if u is (strongly) connected to v.

Proposition

C is an equivalence relation, that is reflexive, symmetric and transitive.

Connectivity and Strong Connected Components

Definition

Given a directed graph G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

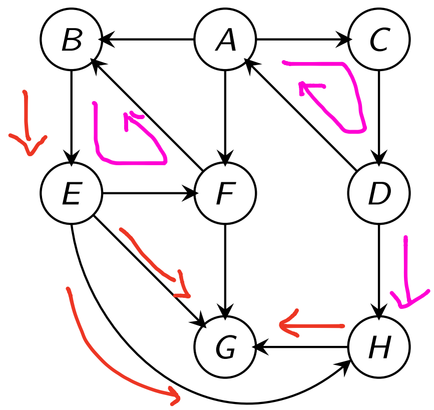
Define relation C where uCv if u is (strongly) connected to v.

Proposition

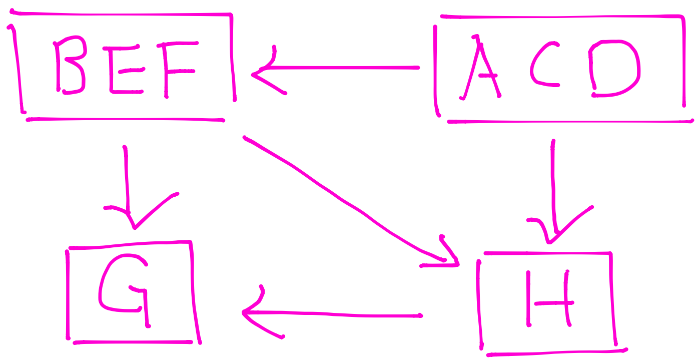
C is an equivalence relation, that is reflexive, symmetric and transitive.

Equivalence classes of C: *strong connected components* of G.
They *partition* the vertices of G.
$\mathrm{SCC}(u)$: strongly connected component containing u.

Strongly Connected Components: Example



meta-graph



Directed Graph Connectivity Problems

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute $\mathrm{rch}(u)$.
3. Given G and u, compute all v that can reach u, that is all v such that $u \in \mathrm{rch}(v)$.
4. Find the strongly connected component containing node u, that is $\mathrm{SCC}(u)$.
5. Is G strongly connected (a single strong component)?
6. Compute *all* strongly connected components of G.

Basic Graph Search in Directed Graphs

Given $G = (V, E)$ a directed graph and vertex $u \in V$. Let $n = |V|$.

```
Explore(G,u):
    array Visited[1..n]
    Initialize: Set Visited[i] = FALSE for 1 ≤ i ≤ n
    List: ToExplore, S
    Add u to ToExplore and to S, Visited[u] = TRUE
    Make tree T with root as u
    while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each edge (x,y) in Adj(x) do
            if (Visited[y] == FALSE)
                Visited[y] = TRUE
                Add y to ToExplore
                Add y to S
                Add y to T with edge (x,y)
    Output S
```

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = rch(u)$.

Properties of Basic Search

Proposition

Explore(G, u) *terminates with* $S = rch(u)$.

Proposition

T *is a search tree rooted at* u *containing* S *with edges directed away from root to leaves.*

Algorithms via Basic Search - I

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute $\text{rch}(u)$.

Use $\textit{Explore}(G, u)$ to compute $\text{rch}(u)$ in $O(n + m)$ time.

Algorithms via Basic Search - II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \mathrm{rch}(v)$.

Algorithms via Basic Search - II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \mathrm{rch}(v)$.

   Naive: $O(n(n+m))$

Algorithms via Basic Search - II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \mathrm{rch}(v)$.

   Naive: $O(n(n+m))$

Definition (Reverse graph.)

Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Algorithms via Basic Search - II

1. Given G and u, compute all v that can reach u, that is all v such that $u \in \mathrm{rch}(v)$.

   Naive: $O(n(n+m))$

Definition (Reverse graph.)

Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Compute $\mathrm{rch}(u)$ in G^{rev}!

1. **Running time:** $O(n+m)$ to obtain G^{rev} from G and $O(n+m)$ time to compute $\mathrm{rch}(u)$ via Basic Search.

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

1. Find the strongly connected component containing node u. That is, compute $\mathrm{SCC}(G, u)$.

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

1. Find the strongly connected component containing node u. That is, compute $\mathrm{SCC}(G, u)$.

$\mathrm{SCC}(G, u) = \mathrm{rch}(G, u) \cap \mathrm{rch}(G^{rev}, u)$

Algorithms via Basic Search - III

$\mathrm{SCC}(G, u) = \{v \mid u \text{ is strongly connected to } v\}$

1. Find the strongly connected component containing node u. That is, compute $\mathrm{SCC}(G, u)$.

$\mathrm{SCC}(G, u) = \mathrm{rch}(G, u) \cap \mathrm{rch}(G^{rev}, u)$

Hence, $\mathrm{SCC}(G, u)$ can be computed with $Explore(G, u)$ and $Explore(G^{rev}, u)$. Total $O(n + m)$ time.

Algorithms via Basic Search - IV

1. Is G strongly connected?

Algorithms via Basic Search - IV

1. Is G strongly connected?

Pick arbitrary vertex u. Check if $\mathrm{SCC}(G, u) = V$.

Algorithms via Basic Search - V

1. Find *all* strongly connected components of G.

Algorithms via Basic Search - V

1. Find *all* strongly connected components of G.

```
While G is not empty do
    Pick arbitrary node u
    find S = SCC(G, u)
    Remove S from G
```

Algorithms via Basic Search - V

1. Find *all* strongly connected components of G.

```
While G is not empty do
    Pick arbitrary node u
    find S = SCC(G, u)
    Remove S from G
```

Algorithms via Basic Search - V

1. Find *all* strongly connected components of G.

```
While G is not empty do
    Pick arbitrary node u
    find S = SCC(G, u)
    Remove S from G
```

Running time: $O(n(n+m))$.

Algorithms via Basic Search - V

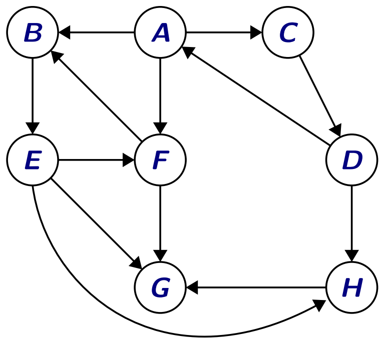
1. Find *all* strongly connected components of G.

```
While G is not empty do
    Pick arbitrary node u
    find S = SCC(G, u)
    Remove S from G
```

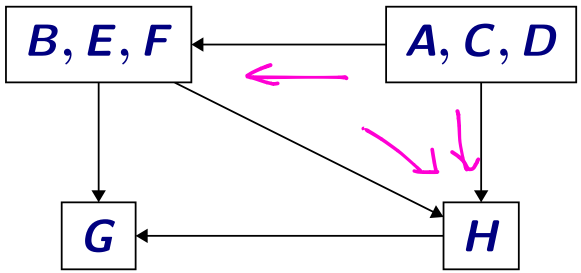
Running time: $O(n(n+m))$.

Question: Can we do it in $O(n+m)$ time?

Structure of a Directed Graph



Graph G



Graph of SCCs G^{SCC}

Reminder

G^{SCC} is created by collapsing every strong connected component to a single vertex.

Proposition

For a directed graph G, its meta-graph G^{SCC} is a DAG.

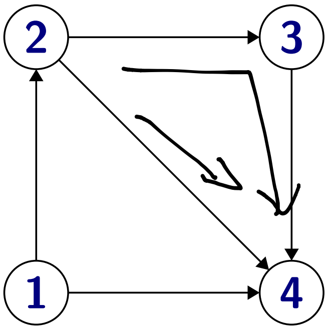
Part III

Directed Acyclic Graphs

Directed Acyclic Graphs

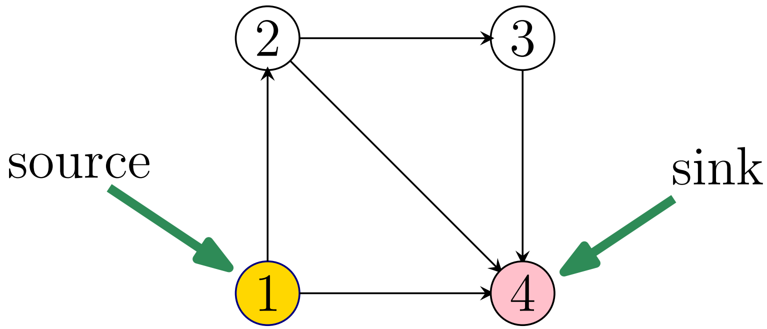
Definition

A directed graph G is a **directed acyclic graph** (DAG) if there is no directed cycle in G.



2 3 4 ×

Sources and Sinks



Definition

1. A vertex u is a **source** if it has no in-coming edges.
2. A vertex u is a **sink** if it has no out-going edges.

Simple DAG Properties

Proposition

Every DAG G *has at least one source and at least one sink.*

Simple DAG Properties

Proposition

Every DAG G *has at least one source and at least one sink.*

Proof.

Let $P = v_1, v_2, \ldots, v_k$ be a longest path in G. Claim that v_1 is a source and v_k is a sink.

Simple DAG Properties

Proposition

Every DAG G *has at least one source and at least one sink.*

Proof.

Let $P = v_1, v_2, \ldots, v_k$ be a longest path in G. Claim that v_1 is a source and v_k is a sink.
Suppose not. Then v_1 has an incoming edge which either creates a cycle or a longer path both of which are contradictions. Similarly if v_k has an outgoing edge. $\square$

$$\begin{tikzcd} v_0 \arrow[d, bend right] & & \\ v_1 \arrow[r] & v_2 \arrow[r] & v_3 \end{tikzcd}$$

Simple DAG Properties

Proposition

Every DAG G has at least one source and at least one sink.

Proof.

Let $P = v_1, v_2, \ldots, v_k$ be a longest path in G. Claim that v_1 is a source and v_k is a sink.
Suppose not. Then v_1 has an incoming edge which either creates a cycle or a longer path both of which are contradictions. Similarly if v_k has an outgoing edge. $\square$

1. G is a DAG if and only if G^{rev} is a DAG.

Simple DAG Properties

Proposition

Every DAG *G has at least one source and at least one sink.*

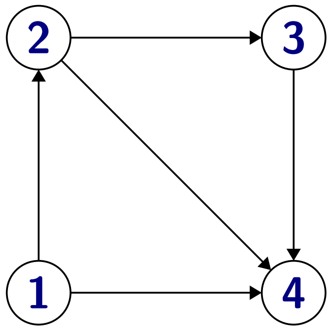
Proof.

Let $P = v_1, v_2, \ldots, v_k$ be a longest path in G. Claim that v_1 is a source and v_k is a sink.
Suppose not. Then v_1 has an incoming edge which either creates a cycle or a longer path both of which are contradictions. Similarly if v_k has an outgoing edge. □

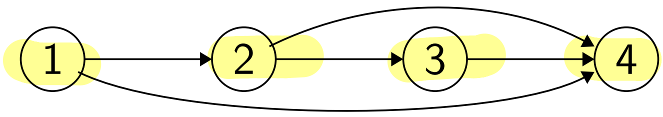
1. G is a DAG if and only if G^{rev} is a DAG.
2. G is a DAG if and only if each node is in its own strong connected component.

Formal proofs: exercise.

Topological Ordering/Sorting



Graph G



Topological Ordering of G

Definition

A **topological ordering**/**topological sorting** of $G = (V, E)$ is an ordering $\prec$ on V such that if $(u, v) \in E$ then $u \prec v$.

Informal equivalent definition:

One can order the vertices of the graph along a line (say the x-axis) such that all edges are from left to right.

DAGs and Topological Sort

Lemma

A directed graph G can be topologically ordered iff it is a DAG.

Need to show both directions.

DAGs and Topological Sort

Lemma

A directed graph G can be topologically ordered if it is a DAG.

Proof.

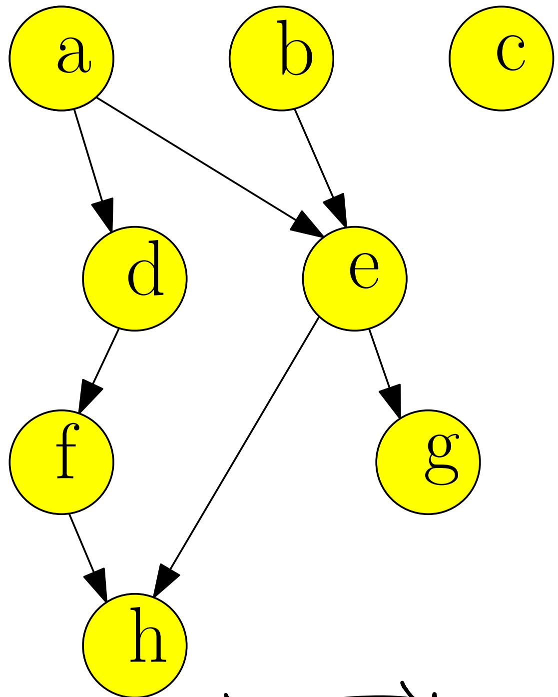
Consider the following algorithm:

1. Pick a source u, output it.
2. Remove u and all edges out of u.
3. Repeat until graph is empty.

Exercise: prove this gives toplogical sort. □

Exercise: show algorithm can be implemented in $O(m+n)$ time.

Topological Sort: Example



a b c d e f g h

b c a d e f g h

DAGs and Topological Sort

Lemma

A directed graph G can be topologically ordered only if it is a DAG*.*

Proof.

Suppose G is not a DAG and has a topological ordering $\prec$. G has a cycle $C = u_1, u_2, \ldots, u_k, u_1$.

Then $u_1 \prec u_2 \prec \ldots \prec u_k \prec u_1$!

That is... $u_1 \prec u_1$.

A contradiction (to $\prec$ being an order).

Not possible to topologically order the vertices. $\square$

DAGs and Topological Sort

Note: A DAG G may have many different topological sorts.

Question: What is a DAG with the most number of distinct topological sorts for a given number n of vertices?

Question: What is a DAG with the least number of distinct topological sorts for a given number n of vertices?