DFS, Cycle Detection, and Topological Sort

Lecture 16

March 25, 2025

Recall WFS

```
WFS(G,u):
    Initialize array Found[1..n] to all FALSE
    Set Found[u] = TRUE
    Initialize ToExplore, S as {u}
    while ToExplore is non-empty do
        Remove a node x from ToExplore
        for each edge (x,y) in Adj(x) do
            if Found[y] = FALSE
                Found[y] ← TRUE
                Set y.parent = x
                Add y to ToExplore and S
    Output S
```

Recall WFS

```
WFS(G,u):
    Initialize array Found[1..n] to all FALSE
    Set Found[u] = TRUE
    Initialize ToExplore, S as {u}
    while ToExplore is non-empty do
        Remove a node x from ToExplore
        for each edge (x,y) in Adj(x) do
            if Found[y] = FALSE
                Found[y] ← TRUE
                Set y.parent = x
                Add y to ToExplore and S
    Output S
```

BFS: Implement ToExplore with a queue (FIFO)

DFS: Implement ToExplore with a stack (LIFO)

Uses for WFS

WFS can be used to:

- Find if u is connected to v
- Find the connected component containing u
- Find / count *all* connected components in G

Uses for WFS

WFS can be used to:

- Find if u is connected to v
- Find the connected component containing u
- Find / count *all* connected components in G

BFS can be used to

- Find the path from u to v that uses the fewest edges

Uses for WFS

WFS can be used to:

- Find if u is connected to v
- Find the connected component containing u
- Find / count *all* connected components in G

BFS can be used to

- Find the path from u to v that uses the fewest edges

Focus of today: uses for DFS, especially in *directed* graphs

Part I

Understanding DFS

Recursive DFS

Easier to analyze DFS if we phrase it as a recursive algorithm.

```
Global array Visited[1..n], init all False
DFS(G,u):
    Set Visited[u] = TRUE
    for each edge (u, v) in Adj(u) do
        if Visited[v] = FALSE
            v.parent = u
            DFS(G, v)
```

Recursive DFS

Easier to analyze DFS if we phrase it as a recursive algorithm.

```
Global array Visited[1..n], init all False
DFS(G,u):
    Set Visited[u] = TRUE
    for each edge (u, v) in Adj(u) do
        if Visited[v] = FALSE
            v.parent = u
            DFS(G, v)
```

Note: recursion stack takes the place of `ToExplore`

Pre- and Post-Ordering

Useful to know what order DFS processes vertices in, so store when DFS on u starts and finishes.

```
Global array Visited[1..n], init all False
Global integer clock, init to 1
DFS(G,u):
    u.pre = clock, clock = clock + 1
    Set Visited[u] = TRUE
    for each edge (u, v) in Adj(u) do
        if Visited[v] = FALSE
            v.parent = u
            DFS(G, v)
    u.post = clock, clock = clock + 1
```

Pre- and Post-Ordering

Useful to know what order DFS processes vertices in, so store when DFS on u starts and finishes.

```
Global array Visited[1..n], init all False
Global integer clock, init to 1
DFS(G,u):
    u.pre = clock, clock = clock + 1
    Set Visited[u] = TRUE
    for each edge (u, v) in Adj(u) do
        if Visited[v] = FALSE
            v.parent = u
            DFS(G, v)
    u.post = clock, clock = clock + 1
```

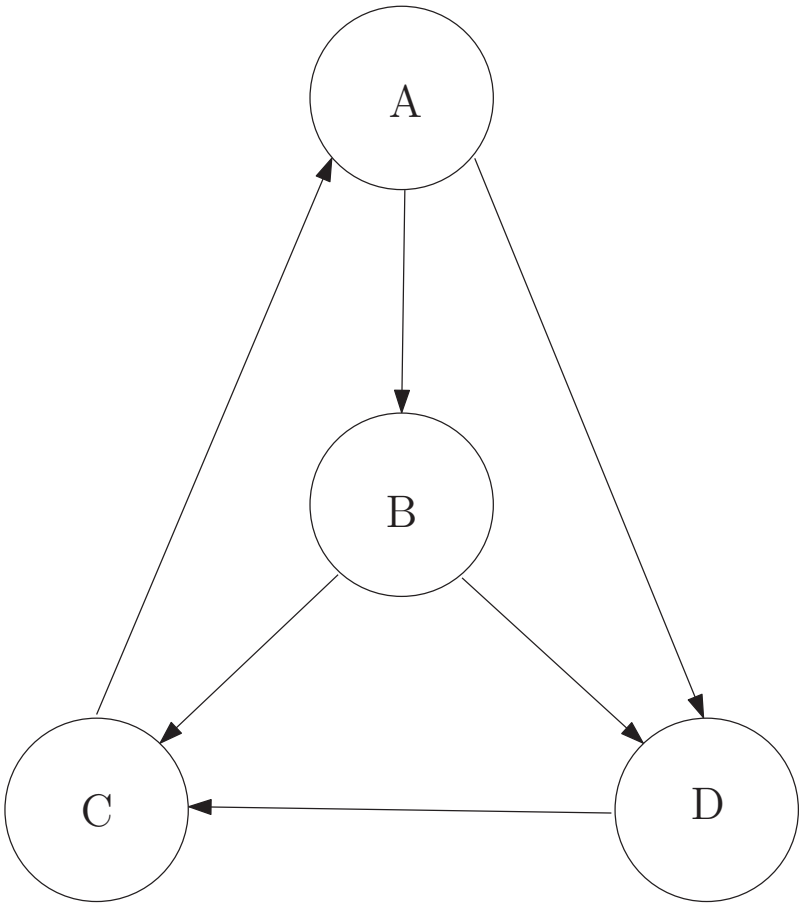
pre and post each define an ordering on the vertices.

DFSAll

To get pre- or post-ordering on the whole graph, continue to run DFS on any vertex we haven't seen yet.

```
Global array Visited[1..n], init all False
Global integer clock, init to 1
DFSAll(G):
    for each vertex u do
        if Visited[u] = FALSE
            DFS(G, u)
```

Example of DFS with `pre` and `post`



`clock` =

Recursion stack:

Using `pre` and `post`

`pre` and `post` values can tell us a lot about the structure of a graph.

Using `pre` and `post`

`pre` and `post` values can tell us a lot about the structure of a graph.

Say we have an edge $(u, v) \in E$. What does it mean if $u.\texttt{pre} < v.\texttt{pre} < v.\texttt{post} < u.\texttt{post}$?

Using `pre` and `post`

`pre` and `post` values can tell us a lot about the structure of a graph.

Say we have an edge $(u, v) \in E$. What does it mean if $u.\texttt{pre} < v.\texttt{pre} < v.\texttt{post} < u.\texttt{post}$?

What if $v.\texttt{pre} < u.\texttt{pre} < u.\texttt{post} < v.\texttt{post}$?

Using `pre` and `post`

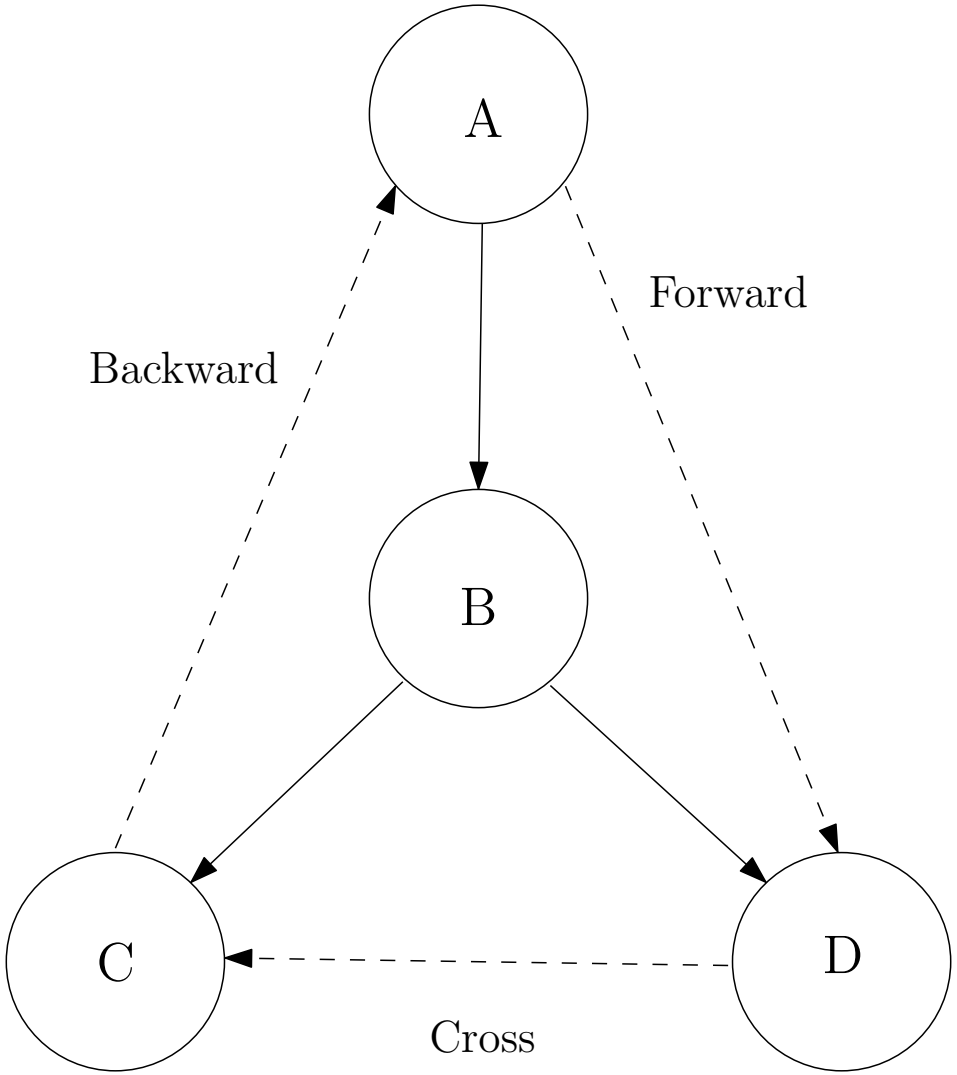
`pre` and `post` values can tell us a lot about the structure of a graph.

Say we have an edge $(u, v) \in E$. What does it mean if $u.\texttt{pre} < v.\texttt{pre} < v.\texttt{post} < u.\texttt{post}$?

What if $v.\texttt{pre} < u.\texttt{pre} < u.\texttt{post} < v.\texttt{post}$?

What if $v.\texttt{pre} < v.\texttt{post} < u.\texttt{pre} < u.\texttt{post}$?

A Thousand Words



No Other Cases

The three cases we considered are the only possible ones!

No Other Cases

The three cases we considered are the only possible ones!

For *any* vertices u and v, the intervals $[u.\texttt{pre}, u.\texttt{post}]$ and $[v.\texttt{pre}, v.\texttt{post}]$ are either nested or disjoint.

No Other Cases

The three cases we considered are the only possible ones!

For *any* vertices u and v, the intervals $[u.\texttt{pre}, u.\texttt{post}]$ and $[v.\texttt{pre}, v.\texttt{post}]$ are either nested or disjoint.

If $(u, v) \in E$, cannot have $u.\texttt{pre} < u.\texttt{post} < v.\texttt{pre} < v.\texttt{post}$.

Application: Finding Cycles

Given `pre` and `post` values from a run of DFS:

- How can we check if an *undirected* graph has a cycle?

- How can we check if a *directed* graph has a cycle?

Cycle Finding Algorithms

```
CheckUndirCycle(G):
    DFSAll(G)
    for each edge uv do
        if uv is not a tree edge
            return True
    return False
```

```
CheckDirCycle(G):
    DFSAll(G)
    for each edge (u, v) do
        if (u, v) is a backward edge
            return True
    return False
```

Cycle Finding Algorithms

```
CheckUndirCycle(G):
    DFSAll(G)
    for each edge uv do
        if uv is not a tree edge
            return True
    return False
```

```
CheckDirCycle(G):
    DFSAll(G)
    for each edge (u, v) do
        if (u, v) is a backward edge
            return True
    return False
```

Efficiency?

Cycle Finding Algorithms

```
CheckUndirCycle(G):
    DFSAll(G)
    for each edge uv do
        if uv is not a tree edge
            return True
    return False
```

```
CheckDirCycle(G):
    DFSAll(G)
    for each edge (u, v) do
        if (u, v) is a backward edge
            return True
    return False
```

Efficiency? $O(V + E)$

Exercise: modify these to output a cycle if one exists.

Part II

DAGs and Topological Sort

DAG Definition

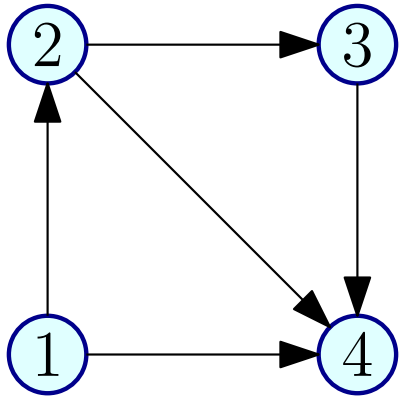
Definition

A **directed acyclic graph** (DAG) is a directed graph with no directed cycles.

DAG Definition

Definition

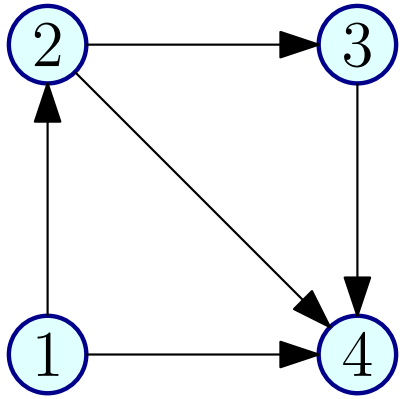
A **directed acyclic graph** (DAG) is a directed graph with no directed cycles.



DAG Definition

Definition

A **directed acyclic graph** (DAG) is a directed graph with no directed cycles.



Commonly used to model dependencies or rankings.

Sources and Sinks

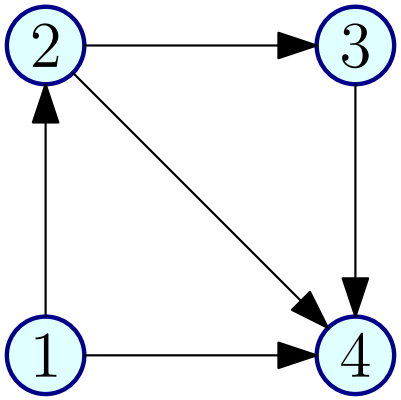
Definition

- A vertex is a **source** if it has no incoming edges.
- A vertex is a **sink** if it has no outgoing edges.

Sources and Sinks

Definition

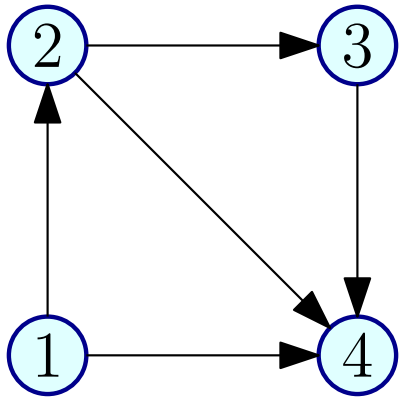
- A vertex is a **source** if it has no incoming edges.
- A vertex is a **sink** if it has no outgoing edges.



Sources and Sinks

Definition

- A vertex is a **source** if it has no incoming edges.
- A vertex is a **sink** if it has no outgoing edges.



Note: it is possible for a vertex to be both a source and a sink! (Requires it to have no edges at all.)

Sources and Sinks in DAGs

Proposition

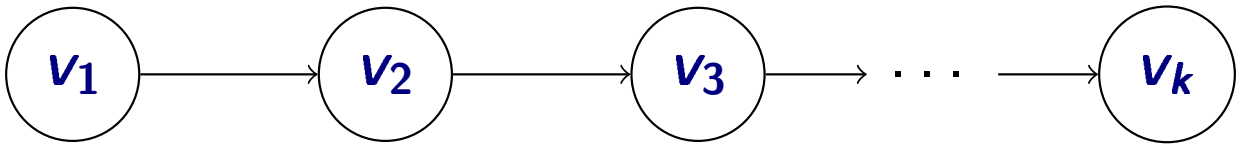
Every DAG has at least one source and at least one sink.

Sources and Sinks in DAGs

Proposition

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

Let $(v_1, v_2, \ldots, v_k)$ be the *longest* path in our DAG.

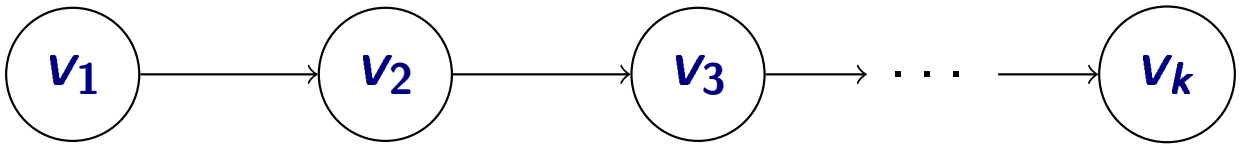


Sources and Sinks in DAGs

Proposition

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

Let $(v_1, v_2, \ldots, v_k)$ be the *longest* path in our DAG.



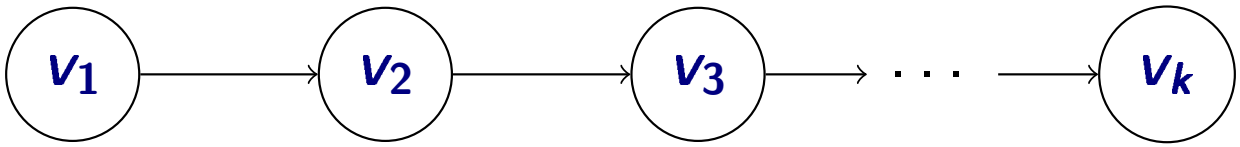
Claim: v_1 is a source.

Sources and Sinks in DAGs

Proposition

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

Let $(v_1, v_2, \ldots, v_k)$ be the *longest* path in our DAG.



Claim: v_1 is a source.

Claim: v_k is a sink.

Topological Order

Definition

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

Topological Order

Definition

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

Informal equivalent definition:

A way to arrange V along a line such that all edges go left to right.

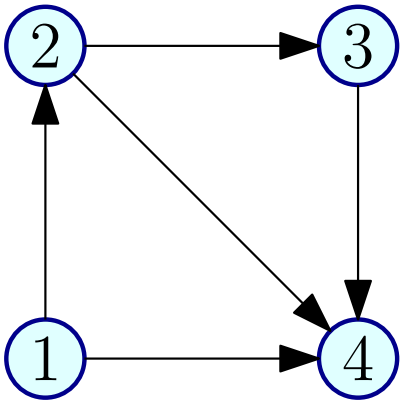
Topological Order

Definition

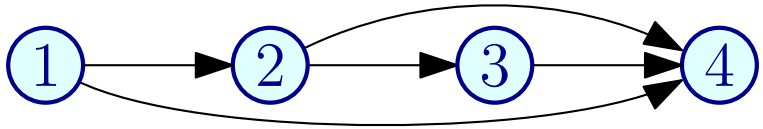
A **topological order** of a directed graph $G = (V, E)$ is an ordering $\prec$ on V such that if $(u, v) \in E$ then $u \prec v$.

Informal equivalent definition:

A way to arrange V along a line such that all edges go left to right.



Graph G



Topological Ordering of G

Finding a Topological Order

Proposition

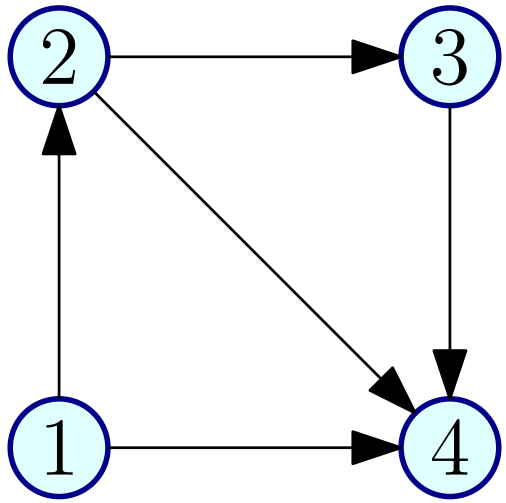
Every DAG has at least one topological order.

Finding a Topological Order

Proposition

Every DAG has at least one topological order.

How would you find a topological order of a DAG?



Naïve Topological Order

```
NaïveTopOrder(G):
    order = []
    while G is non-empty
        Find a sink v in G
        Remove v from G
        order = [v] + order
    return order
```

Naïve Topological Order

```
NaïveTopOrder(G):
    order = []
    while G is non-empty
        Find a sink v in G
        Remove v from G
        order = [v] + order
    return order
```

Efficiency?

Naïve Topological Order

```
NaïveTopOrder(G):
    order = []
    while G is non-empty
        Find a sink v in G
        Remove v from G
        order = [v] + order
    return order
```

Efficiency? $O(V(V+E))$

Can we do better?

Topological Order from `post`

Idea: reduce the time we spend repeatedly finding sinks.

Topological Order from `post`

Idea: reduce the time we spend repeatedly finding sinks.

Observation 1: In a DAG, the vertex with the smallest `post` value is always a sink.

Topological Order from `post`

Idea: reduce the time we spend repeatedly finding sinks.

Observation 1: In a DAG, the vertex with the smallest `post` value is always a sink.

Observation 2: Removing a sink from a DAG doesn't change the *order* of `post` values.

Topological Order from `post`

Idea: reduce the time we spend repeatedly finding sinks.

Observation 1: In a DAG, the vertex with the smallest `post` value is always a sink.

Observation 2: Removing a sink from a DAG doesn't change the *order* of `post` values.

```
TopOrder(G):
    DFSAll(G)
    order = []
    for all vertices v in decreasing order of v.post
        order = [v] + order
    return order
```

Topological Order from `post`

Idea: reduce the time we spend repeatedly finding sinks.

Observation 1: In a DAG, the vertex with the smallest `post` value is always a sink.

Observation 2: Removing a sink from a DAG doesn't change the *order* of `post` values.

```
TopOrder(G):
    DFSAll(G)
    return V in decreasing order of post
```

Topological Order from `post`

Idea: reduce the time we spend repeatedly finding sinks.

Observation 1: In a DAG, the vertex with the smallest `post` value is always a sink.

Observation 2: Removing a sink from a DAG doesn't change the *order* of `post` values.

```
TopOrder(G):
    DFSAll(G)
    return V in decreasing order of post
```

Efficiency?

No Topological Order With Cycles

Proposition

If a directed graph $G = (V, E)$ has a cycle, G does not have a topological order.

No Topological Order With Cycles

Proposition

If a directed graph $G = (V, E)$ has a cycle, G does not have a topological order.

In particular, this means that you should *never* invoke the topological sort algorithm on a non-DAG!

Problems and Algorithms From Today

- Cycle checking in *undirected* graphs
  - Use DFS (or WFS), $O(V + E)$ time
- Cycle checking in *directed* graphs
  - Use DFS, $O(V + E)$ time
- Finding a topological ordering in a DAG
  - Use DFS `post`-order, $O(V + E)$ time