Lecture 13: Basic Graph Algorithms

Michael Dinitz

October 8, 2024
601.433/633 Introduction to Algorithms

Introduction

Next 3-4 weeks: graphs!

- Super important abstractions, used all over the place in CS
- Most of my research is in graph algorithms (particularly when graph represents computer/communication network)
- Great course on Graph Theory in AMS

Today: review of basic graph algorithms from Data Structures, possibly one or two new

- Going to move pretty quickly, since much review: see CLRS for details!

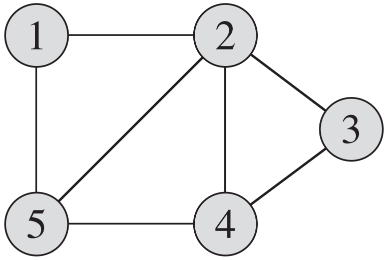
Basic Definitions

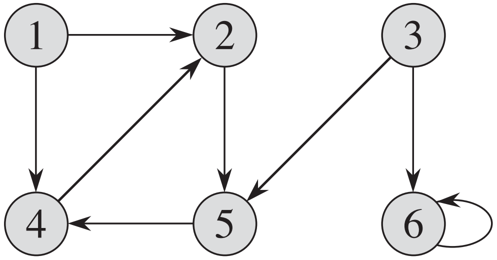
Definition

A graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ is a pair where $\boldsymbol{V}$ is a set and $\boldsymbol{E} \subseteq \binom{\boldsymbol{V}}{2}$ (unordered pairs) or $\boldsymbol{E} \subseteq \boldsymbol{V} \times \boldsymbol{V}$ (ordered pairs).

Notation:

- Elements of $\boldsymbol{V}$ are called *vertices* or *nodes*
- Elements of $\boldsymbol{E}$ are called *edges* or *arcs*.
- If $\boldsymbol{E} \subseteq \binom{\boldsymbol{V}}{2}$ then graph is *undirected*, if $\boldsymbol{E} \subseteq \boldsymbol{V} \times \boldsymbol{V}$ graph is *directed*
- $|\boldsymbol{V}| = \boldsymbol{n}$ and $|\boldsymbol{E}| = \boldsymbol{m}$ (usually)
- So "size of input" $= \boldsymbol{n} + \boldsymbol{m}$





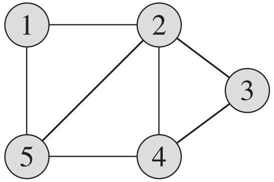
Representations

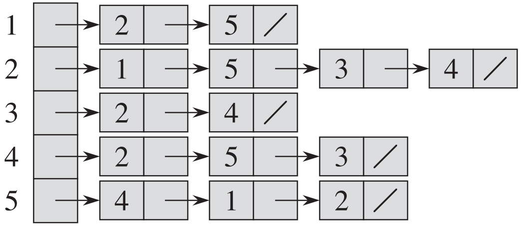
Adjacency List:

- Array A of length n
- $A[v]$ is linked list of vertices *adjacent* to v (edge from u to v)

Adjacency Matrix:

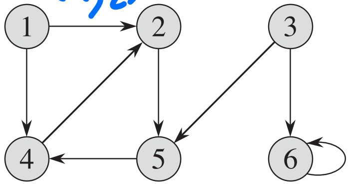
- $A \in \{0,1\}^{n \times n}$
- $A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$

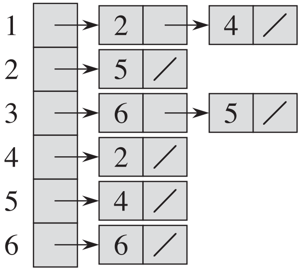




	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(1,2)





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Representations (cont'd)

Adjacency List:

- Pros:

Representations (cont'd)

Adjacency List:

- Pros:
  - $O(n + m)$ space
  - Can iterate through edges adjacent to v very efficiently

Representations (cont'd)

Adjacency List:

- Pros:
  - $O(n + m)$ space
  - Can iterate through edges adjacent to v very efficiently
- Cons:

Representations (cont'd)

Adjacency List:

- Pros:
  - $O(n + m)$ space
  - Can iterate through edges adjacent to v very efficiently
- Cons:
  - Hard to check of an edge exists: $O(d(u))$ or $O(d(v))$ (where $d(v)$ is the degree of v: # edges with v as endpoint)

Representations (cont'd)

Adjacency List:

- Pros:
  - $O(n + m)$ space
  - Can iterate through edges adjacent to v very efficiently
- Cons:
  - Hard to check of an edge exists: $O(d(u))$ or $O(d(v))$ (where $d(v)$ is the degree of v: # edges with v as endpoint)

Adjacency Matrix:

- Pros:

Representations (cont'd)

Adjacency List:

- Pros:
  - $O(n+m)$ space
  - Can iterate through edges adjacent to v very efficiently
- Cons:
  - Hard to check of an edge exists: $O(d(u))$ or $O(d(v))$ (where $d(v)$ is the degree of v: # edges with v as endpoint)

Adjacency Matrix:

- Pros:
  - Can check if $e = (u, v)$ an edge in $O(1)$ time

Representations (cont'd)

Adjacency List:

- Pros:
  - $O(n+m)$ space
  - Can iterate through edges adjacent to v very efficiently
- Cons:
  - Hard to check of an edge exists: $O(d(u))$ or $O(d(v))$ (where $d(v)$ is the degree of v: # edges with v as endpoint)

Adjacency Matrix:

- Pros:
  - Can check if $e = (u, v)$ an edge in $O(1)$ time
- Cons:

Representations (cont'd)

Adjacency List:

- Pros:
    - $O(n + m)$ space
    - Can iterate through edges adjacent to v very efficiently
- Cons:
    - Hard to check of an edge exists: $O(d(u))$ or $O(d(v))$ (where $d(v)$ is the degree of v: # edges with v as endpoint)

Adjacency Matrix:

- Pros:
    - Can check if $e = (u, v)$ an edge in $O(1)$ time
- Cons:
    - Takes $\Theta(n^2)$ space: if m small, lots wasted!
    - Iterating through edges incident on v takes time $\Theta(n)$, even if $d(v)$ small.

Representations (cont'd)

Adjacency List:

- Pros:
  - $O(n+m)$ space
  - Can iterate through edges adjacent to v very efficiently
- Cons:
  - Hard to check of an edge exists: $O(d(u))$ or $O(d(v))$ (where $d(v)$ is the degree of v: $\#$ edges with v as endpoint)

Adjacency Matrix:

- Pros:
  - Can check if $e = (u, v)$ an edge in $O(1)$ time
- Cons:
  - Takes $\Theta(n^2)$ space: if m small, lots wasted!
  - Iterating through edges incident on v takes time $\Theta(n)$, even if $d(v)$ small.

This class: adjacency list unless otherwise specified.

Representations (cont'd)

Adjacency List:

- Pros:
    - $O(n+m)$ space
    - Can iterate through edges adjacent to v very efficiently
- Cons:
    - Hard to check of an edge exists: $O(d(u))$ or $O(d(v))$ (where $d(v)$ is the degree of v: # edges with v as endpoint)

Adjacency Matrix:

- Pros:
    - Can check if $e = (u, v)$ an edge in $O(1)$ time
- Cons:
    - Takes $\Theta(n^2)$ space: if m small, lots wasted!
    - Iterating through edges incident on v takes time $\Theta(n)$, even if $d(v)$ small.

This class: adjacency list unless otherwise specified.

Any way to improve these?

Representations (cont'd)

Adjacency List:

- Pros:
  - $O(n+m)$ space
  - Can iterate through edges adjacent to v very efficiently
- Cons:
  - Hard to check of an edge exists: $O(d(u))$ or $O(d(v))$ (where $d(v)$ is the degree of v: # edges with v as endpoint)

Adjacency Matrix:

- Pros:
  - Can check if $e=(u,v)$ an edge in $O(1)$ time
- Cons:
  - Takes $\Theta(n^2)$ space: if m small, lots wasted!
  - Iterating through edges incident on v takes time $\Theta(n)$, even if $d(v)$ small.

This class: adjacency list unless otherwise specified.

Any way to improve these?

- Replace adjacency *list* with adjacency *structure*: Red-black tree, hash table, etc.
- Not traditional, doesn't gain us much, and more complicated. But better!

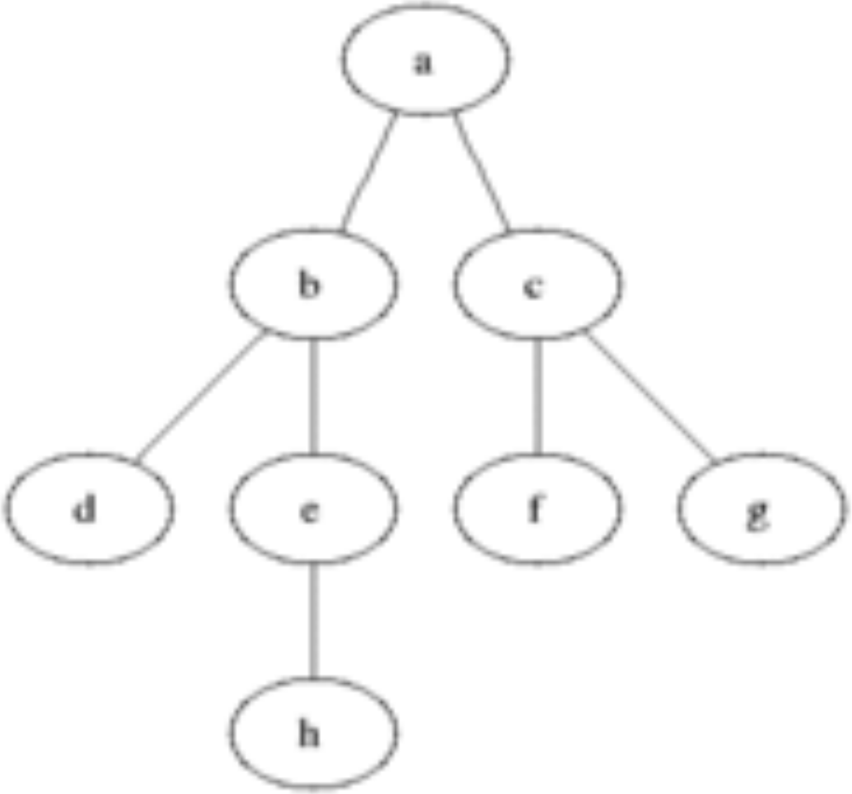
Breadth-First Search (BFS)

BFS Definition

Idea: explore graph in *levels* or *layers* from source s

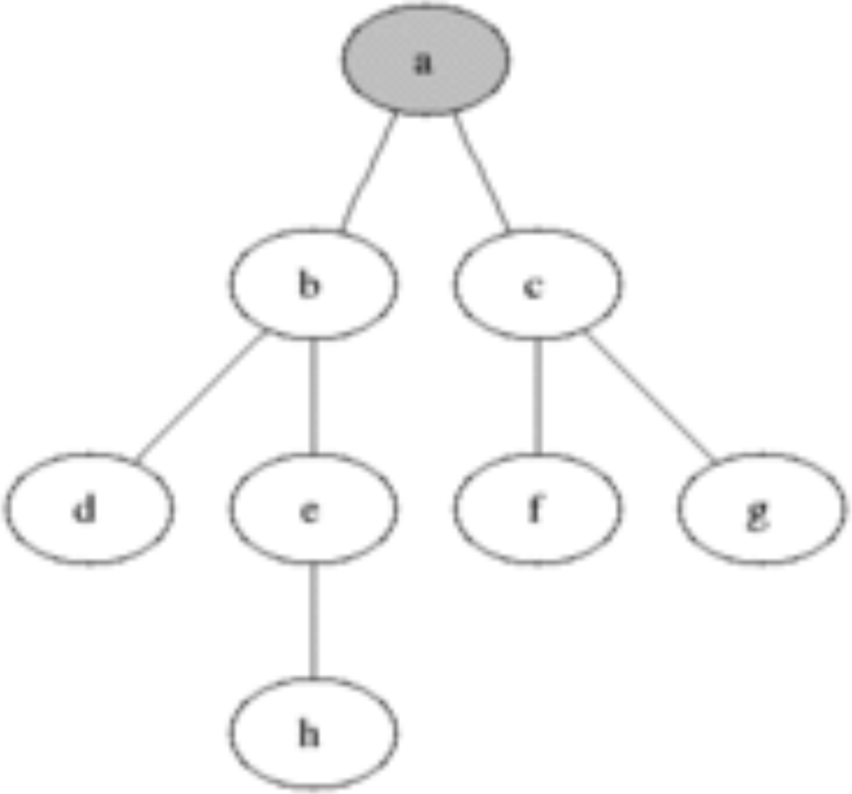
BFS Definition

Idea: explore graph in *levels* or *layers* from source ***s***



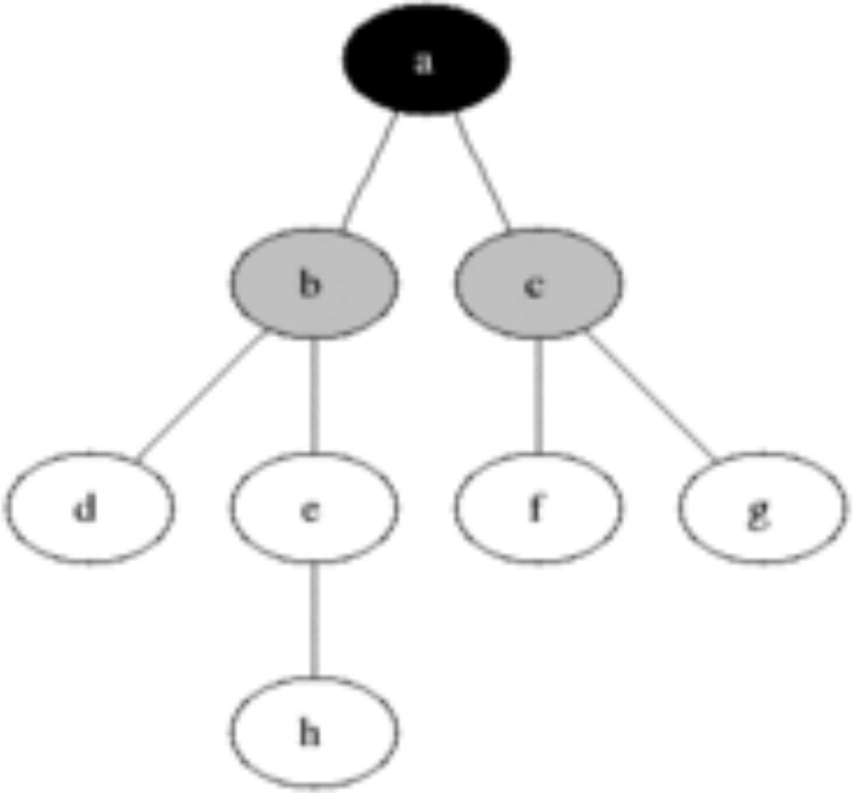
BFS Definition

Idea: explore graph in *levels* or *layers* from source s



BFS Definition

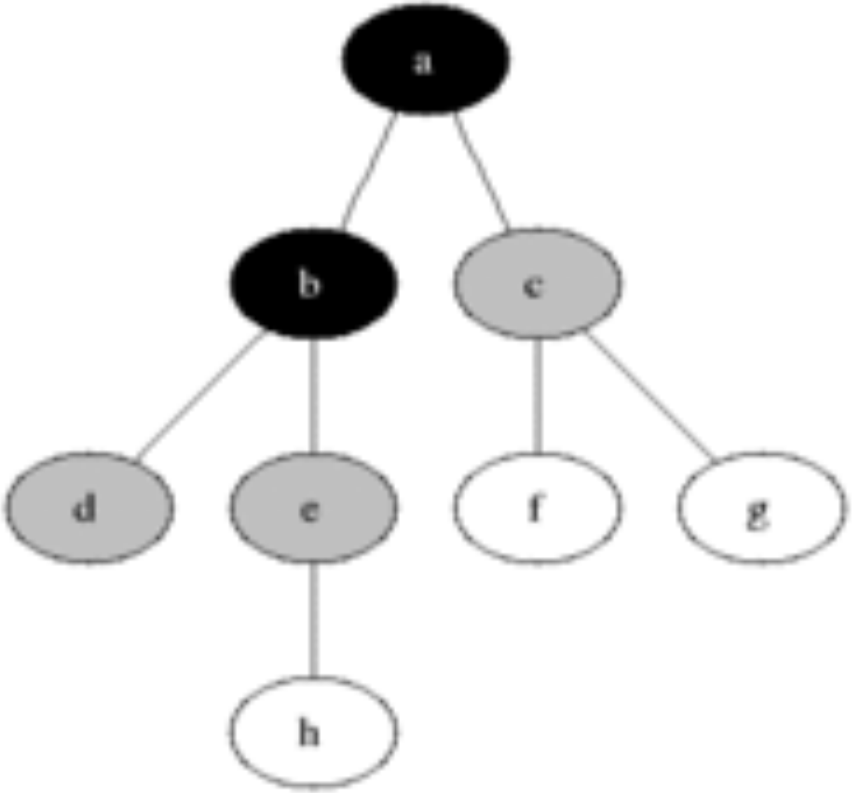
Idea: explore graph in *levels* or *layers* from source s



b
c

BFS Definition

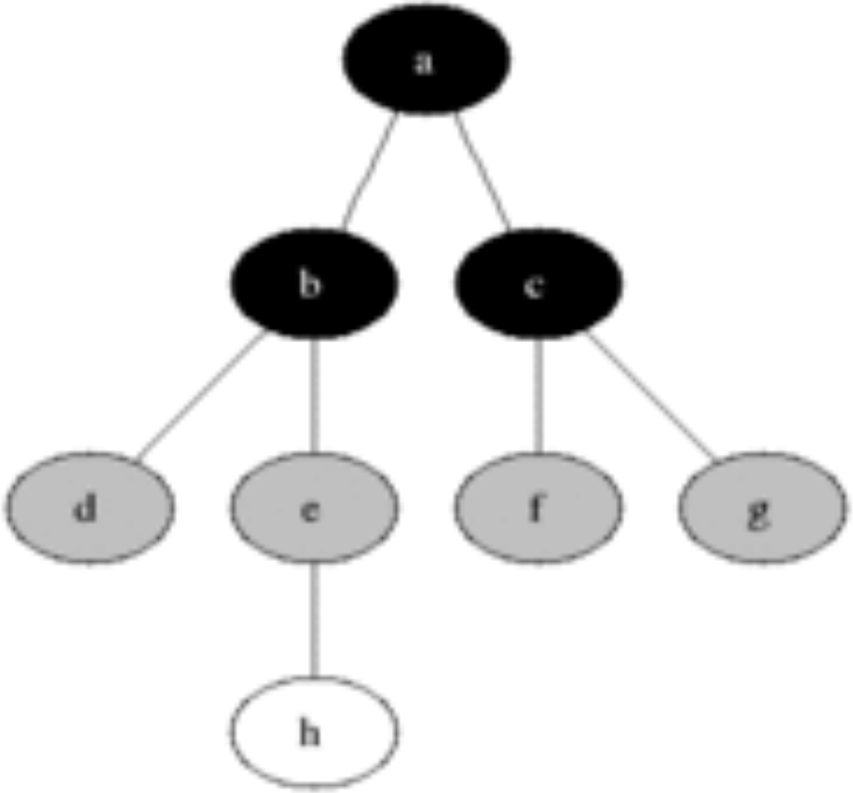
Idea: explore graph in *levels* or *layers* from source ***s***



c
d
e

BFS Definition

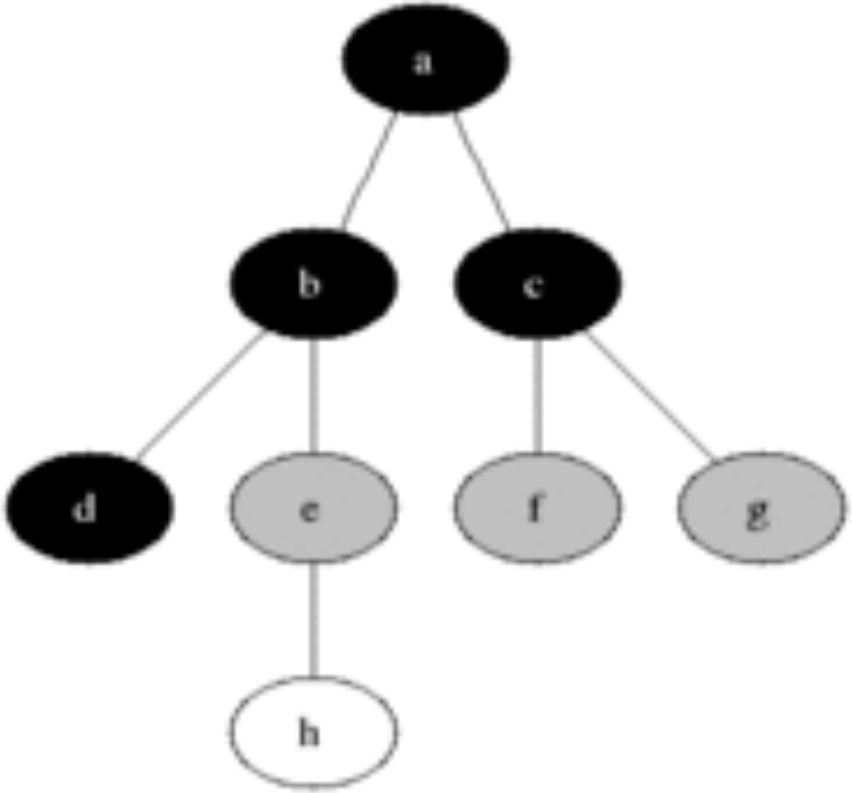
Idea: explore graph in *levels* or *layers* from source ***s***



d
e
f
g

BFS Definition

Idea: explore graph in *levels* or *layers* from source s



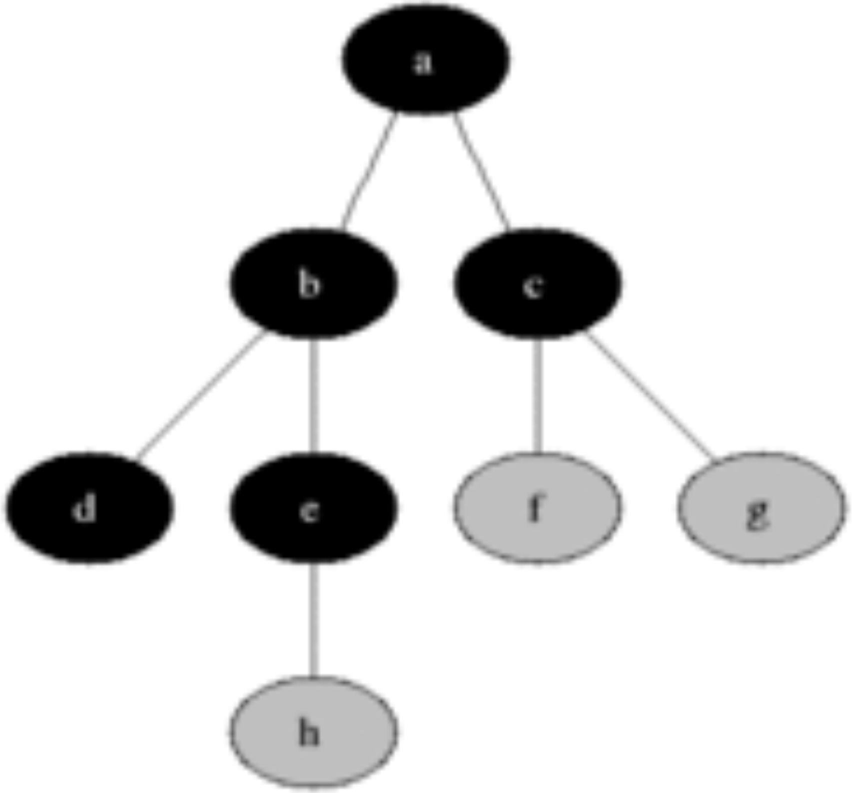
e

f

g

BFS Definition

Idea: explore graph in *levels* or *layers* from source ***s***



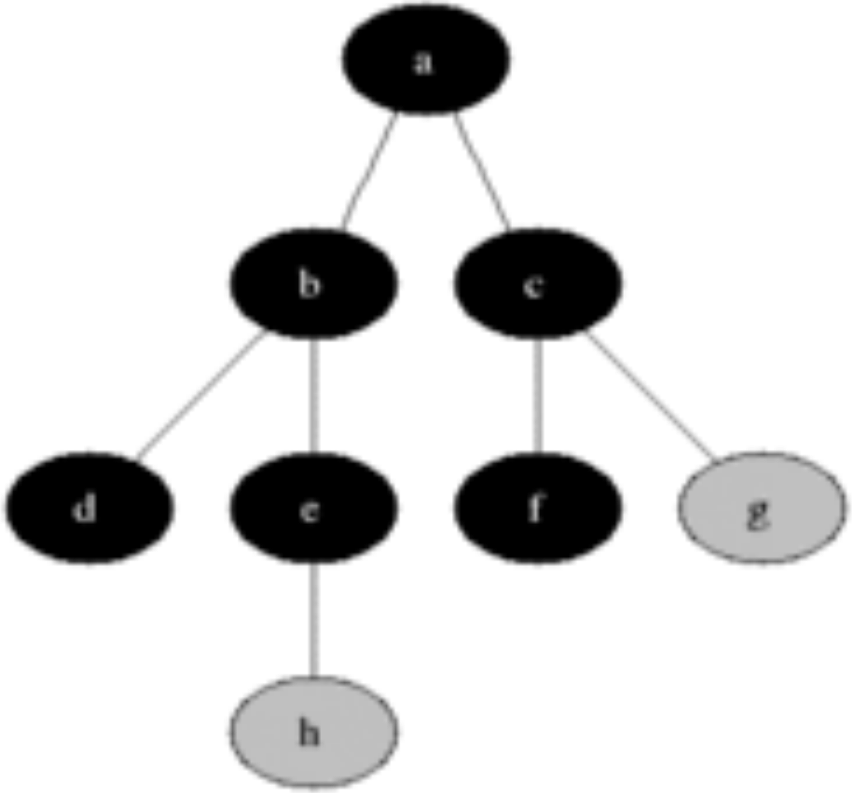
f

g

h

BFS Definition

Idea: explore graph in *levels* or *layers* from source s

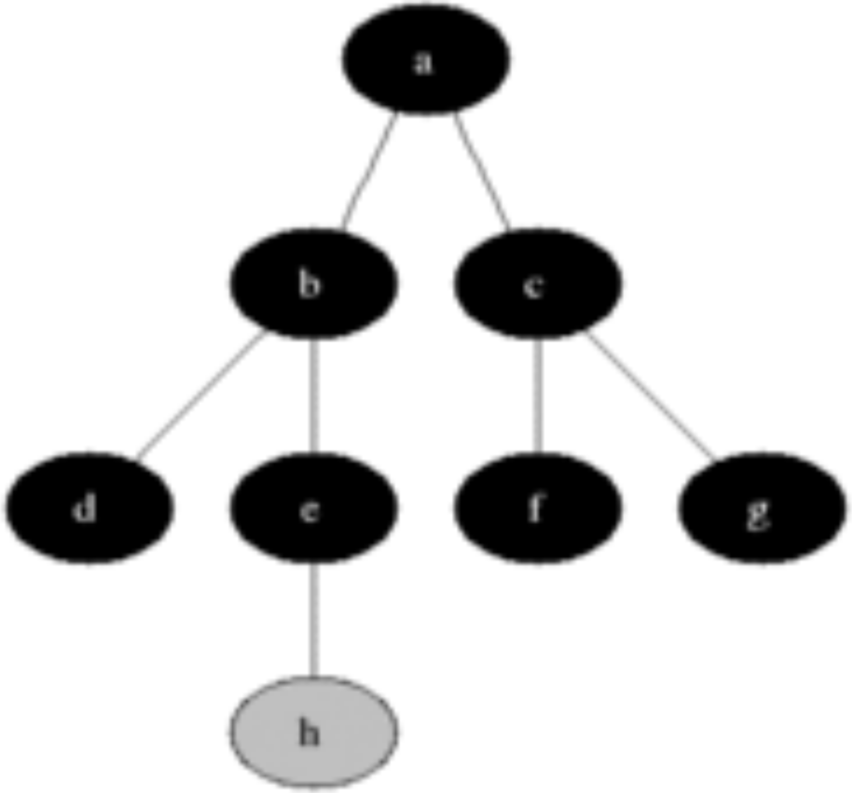


g

h

BFS Definition

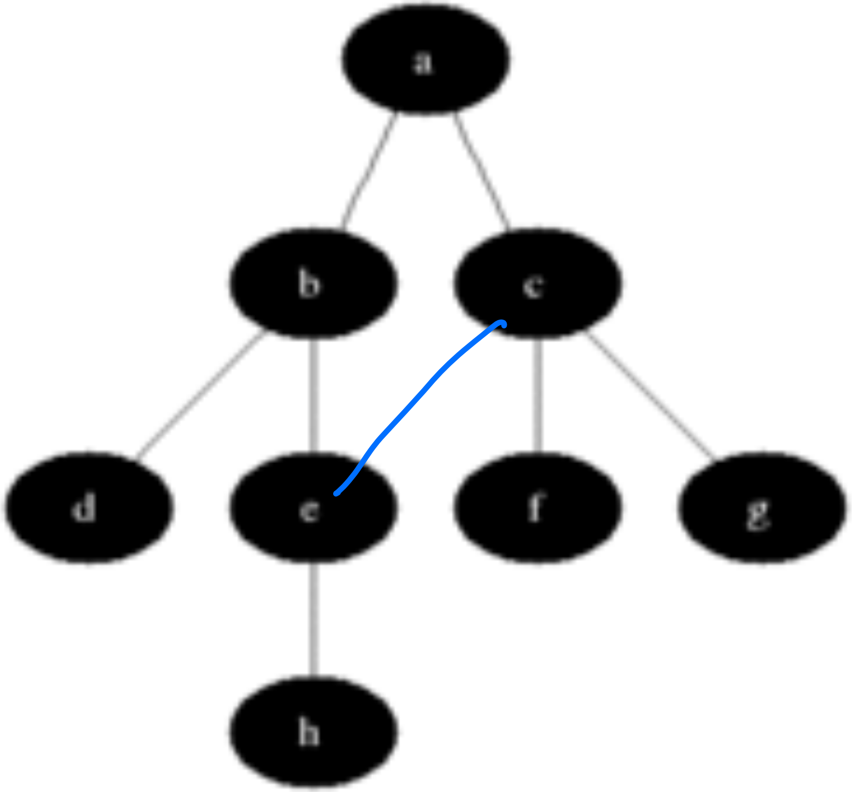
Idea: explore graph in *levels* or *layers* from source s



4

BFS Definition

Idea: explore graph in *levels* or *layers* from source s



BFS Pseudocode

Idea: explore with a queue (FIFO)

```
BFS($G = (V, E), s$) {
    Set $mark(s) = True$;
    Set $mark(v) = False$ for all $v \in V \setminus \{s\}$;
    $Enqueue(s)$;
    while(queue not empty) {
        $v = Dequeue();$
        forall neighbors $u$ of $v$ {
            if($mark(u) == False$) {
                $mark(u) = True$;
                $Enqueue(u)$;
            }
        }
    }
}
```

BFS Pseudocode

Idea: explore with a queue (FIFO)

```
BFS($G = (V, E), s$) {
    Set $mark(s) = True$;
    Set $mark(v) = False$ for all $v \in V \setminus \{s\}$;
    $Enqueue(s)$;
    while(queue not empty) {
        $v = Dequeue();$
        forall neighbors $u$ of $v$ {
            if($mark(u) == False$) {
                $mark(u) = True$;
                $Enqueue(u)$;
            }
        }
    }
}
```

Running Time:

BFS Pseudocode

Idea: explore with a queue (FIFO)

```
BFS($G = (V, E), s$) {
    Set $mark(s) = True$;
    Set $mark(v) = False$ for all $v \in V \setminus \{s\}$;
    $Enqueue(s)$;
    while(queue not empty) {
        $v = Dequeue();$
        forall neighbors $u$ of $v$ {
            if($mark(u) == False$) {
                $mark(u) = True$;
                $Enqueue(u)$;
            }
        }
    }
}
```

Running Time: $O(n + m)$

BFS Pseudocode

Idea: explore with a queue (FIFO)

```
BFS($G = (V, E), s$) {
    Set $mark(s) = True$;
    Set $mark(v) = False$ for all $v \in V \setminus \{s\}$;
    $Enqueue(s)$;
    while(queue not empty) {
        $v = Dequeue()$;
        forall neighbors $u$ of $v$ {
            if($mark(u) == False$) {
                $mark(u) = True$;
                $Enqueue(u)$;
            }
        }
    }
}
```

Running Time: $O(n+m)$

- $O(n)$ for initialization
- $O(m)$ for main while loop
  - Examine every edge twice: when each endpoint dequeued
  - Or (equivalent): Adjacency list scanned only when vertex dequeued

$$\sum_{v \in V} d(v) = O(m)$$

BFS Pseudocode

Idea: explore with a queue (FIFO)

```
BFS($G = (V, E), s$) {
    Set $mark(s) = True$;
    Set $mark(v) = False$ for all $v \in V \setminus \{s\}$;
    $Enqueue(s)$;
    while(queue not empty) {
        $v = Dequeue();$
        forall neighbors $u$ of $v$ {
            if($mark(u) == False$) {
                $mark(u) = True$;
                $Enqueue(u)$;
            }
        }
    }
}
```

Running Time: $O(n + m)$

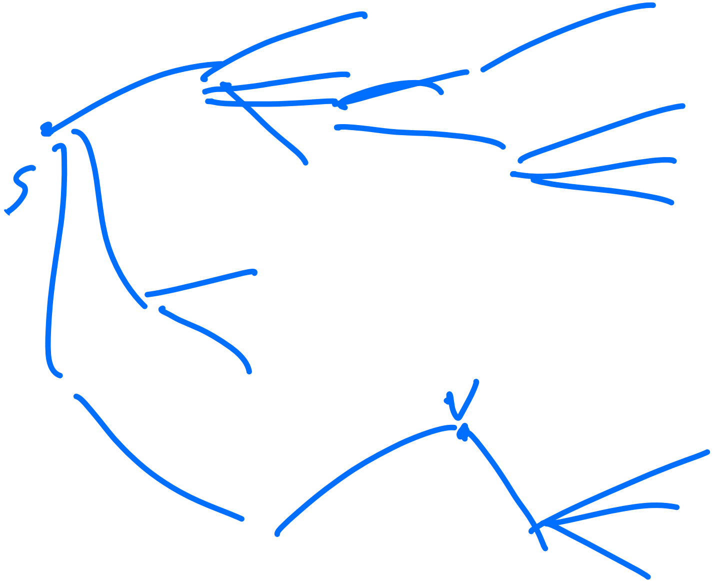
- $O(n)$ for initialization
- $O(m)$ for main while loop
  - Examine every edge twice: when each endpoint dequeued
  - Or (equivalent): Adjacency list scanned only when vertex dequeued

Note: edges that cause a node to be enqueued form a tree!

Correctness / Shortest Paths

Definition: Distance $d(u, v)$ from u to v is min # edges in any path from u to v

Theorem (informal): BFS(s) gives shortest paths from s to all other nodes



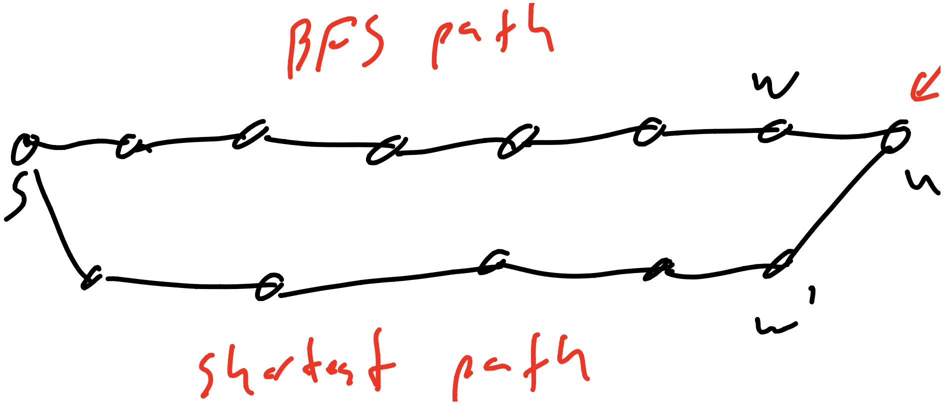
Correctness / Shortest Paths

Definition: Distance $d(u, v)$ from u to v is min # edges in any path from u to v

> **Theorem (informal):** BFS(s) gives shortest paths from s to all other nodes

Proof Sketch:

Assume false for contradiction, let u be closest node to s where BFS(s) doesn't give shortest path



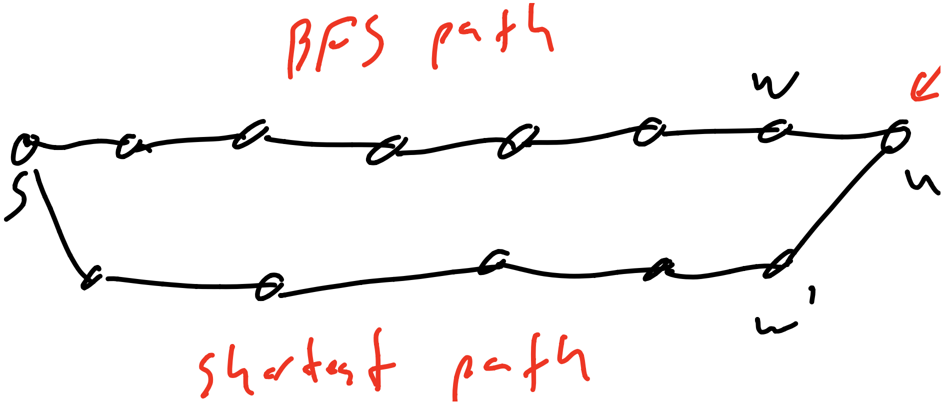
Correctness / Shortest Paths

Definition: Distance $d(u, v)$ from u to v is min # edges in any path from u to v

Theorem (informal): BFS(s) gives shortest paths from s to all other nodes

Proof Sketch:

Assume false for contradiction, let u be closest node to s where BFS(s) doesn't give shortest path



$d(s, w') < d(s, w)$

$\Longrightarrow$ w' dequeued before w (since w' has correct distance by def of u)

$\Longrightarrow$ u will be enqueued from w', not w. Contradiction.

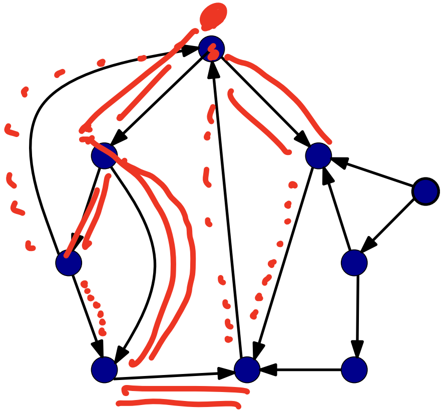
Depth-First Search (DFS)

DFS: Definition

Intuition: Instead of exploring wide (breadth), explore far (deep): just keep walking until see a node we've already seen, then backtrack!

```
Init: for each $v \in V$, $mark(v) = False$;

DFS($v$) {
    $mark(v) = True$;
    for each edge $(v,u) \in A[v]$ {
        if $mark(u) == False$ then DFS($u$);
    }
}
```

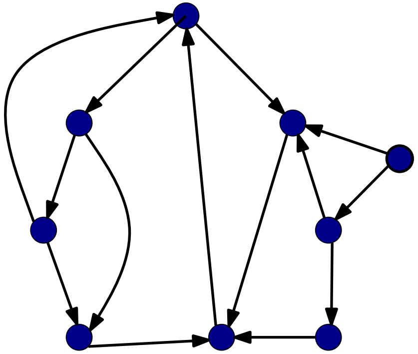


DFS: Definition

Intuition: Instead of exploring wide (breadth), explore far (deep): just keep walking until see a node we've already seen, then backtrack!

```
Init: for each $v \in V$, $mark(v) = False$;

DFS($v$) {
    $mark(v) = True$;
    for each edge $(v, u) \in A[v]$ {
        if $mark(u) == False$ then DFS($u$);
    }
}
```



Running time:

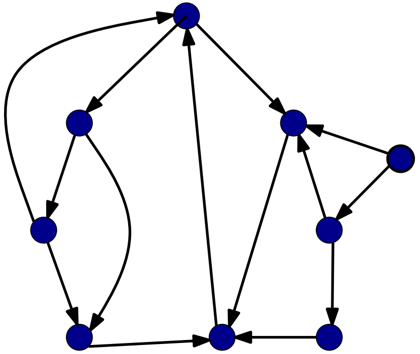
DFS: Definition

Intuition: Instead of exploring wide (breadth), explore far (deep): just keep walking until see a node we've already seen, then backtrack!

$O(n)$

```
Init: for each $v \in V$, $mark(v) = False$;

DFS($v$) {
    $mark(v) = True$;
    for each edge $(v, u) \in A[v]$ {
        if $mark(u) == False$ then DFS($u$);
    }
}
```



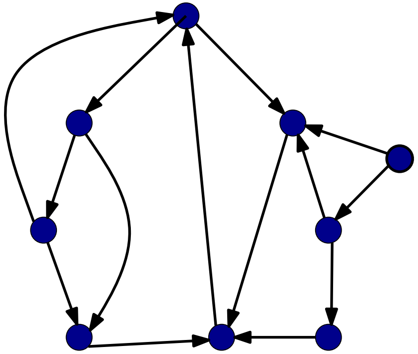
Running time: $O(m + n)$

DFS: Definition

Intuition: Instead of exploring wide (breadth), explore far (deep): just keep walking until see a node we've already seen, then backtrack!

```
Init: for each $v \in V$, $mark(v) = False$;

DFS($v$) {
    $mark(v) = True$;
    for each edge $(v, u) \in A[v]$ {
        if $mark(u) == False$ then DFS($u$);
    }
}
```



Running time: $O(m + n)$

- $O(n)$ initialization
- Every edge considered at most twice

DFS: Correctness

Definition: u is *reachable* from v if there is a path $v = v_0, v_1, \dots, v_k = u$ such that $(v_i, v_{i+1}) \in E$ for all $i \in \{0, 1, \dots, k-1\}$.

Theorem

When DFS(v) terminates, it has visited (marked) all nodes that are reachable from v.

Proof.

Suppose u reachable from v but not marked when DFS(v) terminates.

DFS: Correctness

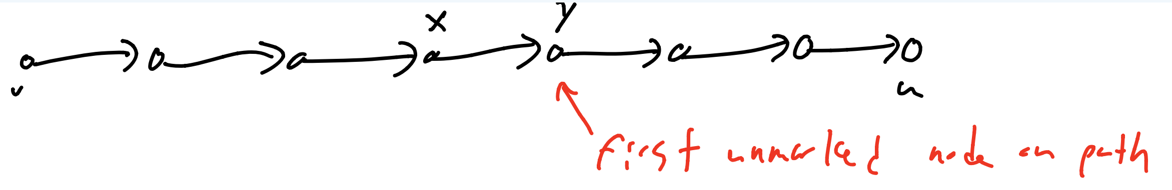
Definition: u is *reachable* from v if there is a path $v = v_0, v_1, \ldots, v_k = u$ such that $(v_i, v_{i+1}) \in E$ for all $i \in \{0, 1, \ldots, k-1\}$.

Theorem

When DFS(v) terminates, it has visited (marked) all nodes that are reachable from v.

Proof.

Suppose u reachable from v but not marked when DFS(v) terminates.



DFS: Correctness

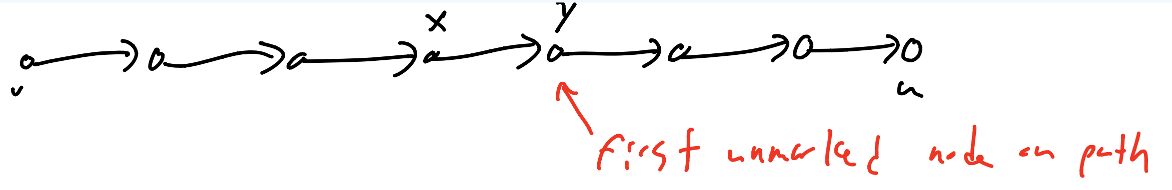
Definition: u is *reachable* from v if there is a path $v = v_0, v_1, \ldots, v_k = u$ such that $(v_i, v_{i+1}) \in E$ for all $i \in \{0, 1, \ldots, k-1\}$.

Theorem

When DFS(v) terminates, it has visited (marked) all nodes that are reachable from v.

Proof.

Suppose u reachable from v but not marked when DFS(v) terminates.



x is marked so DFS(x) must have been called

DFS: Correctness

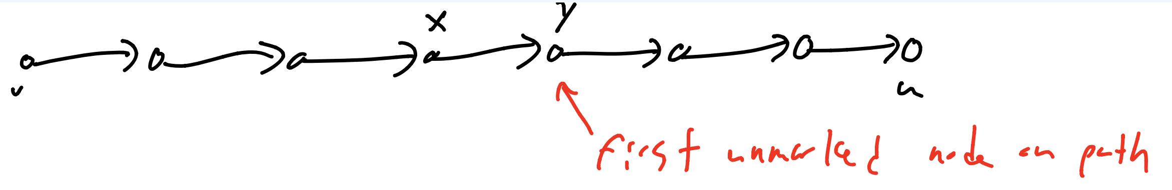
Definition: $\boldsymbol{u}$ is *reachable* from $\boldsymbol{v}$ if there is a path $\boldsymbol{v} = \boldsymbol{v}_0, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_k = \boldsymbol{u}$ such that $(\boldsymbol{v}_i, \boldsymbol{v}_{i+1}) \in \boldsymbol{E}$ for all $\boldsymbol{i} \in \{0, 1, \ldots, \boldsymbol{k} - 1\}$.

Theorem

When DFS($\boldsymbol{v}$) terminates, it has visited (marked) all nodes that are reachable from $\boldsymbol{v}$.

Proof.

Suppose $\boldsymbol{u}$ reachable from $\boldsymbol{v}$ but not marked when DFS($\boldsymbol{v}$) terminates.



$\boldsymbol{x}$ is marked so DFS($\boldsymbol{x}$) must have been called

$\implies$ $\boldsymbol{y}$ was either marked or DFS($\boldsymbol{y}$) called and it became marked.

DFS: Correctness

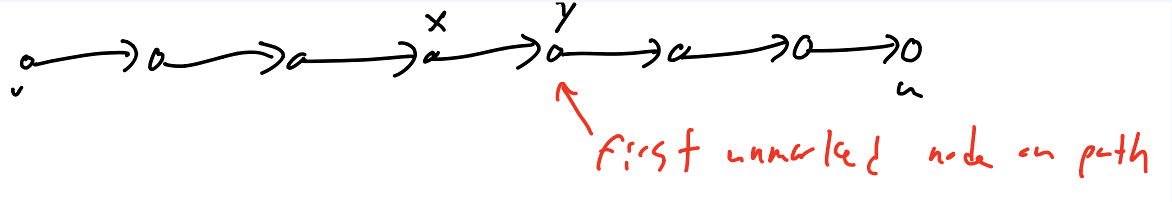
Definition: u is *reachable* from v if there is a path $v = v_0, v_1, \ldots, v_k = u$ such that $(v_i, v_{i+1}) \in E$ for all $i \in \{0, 1, \ldots, k-1\}$.

Theorem

When DFS(v) terminates, it has visited (marked) all nodes that are reachable from v.

Proof.

Suppose u reachable from v but not marked when DFS(v) terminates.



x is marked so DFS(x) must have been called

$\Longrightarrow$ y was either marked or DFS(y) called and it became marked.

Contradiction. □

Graph variant

After DFS(v), node marked if and only if reachable from v.

Might want to continue until all nodes marked.

```
DFS($G$) {
    for all $v \in V$, set $mark(v) = False$;
    while there exists an unmarked node $v$ {
        DFS($v$);
    }
}
```

Timestamps

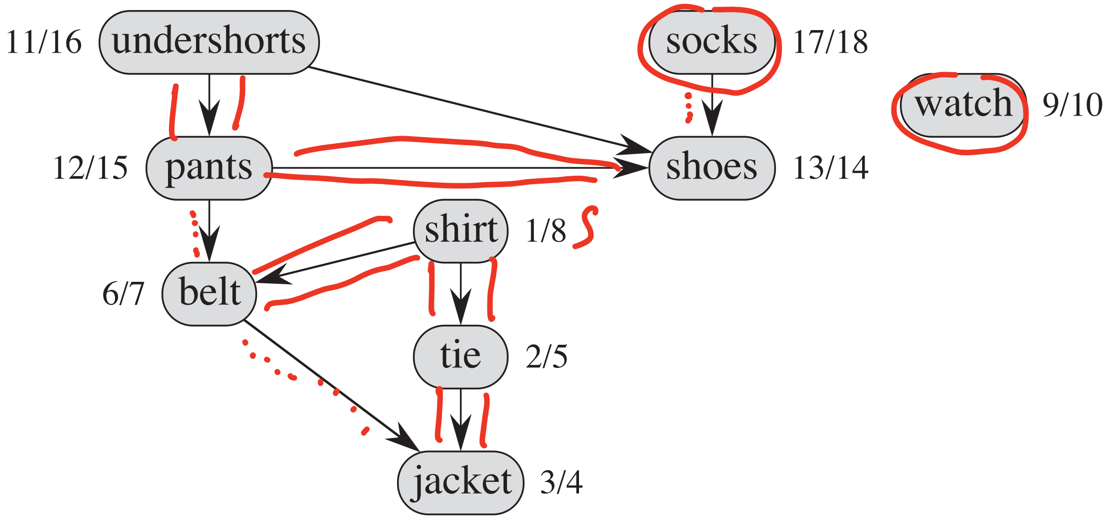
Explicitly keep track of “start” and “finishing” times

- Replaces ***mark***

```
DFS($G$) {
    $t = 0$;
    for all $v \in V$ {
        $start(v) = 0$;
        $finish(v) = 0$;
    }
    while $\exists v \in V$ with $start(v) = 0$ {
        DFS($v$);
    }
}
```

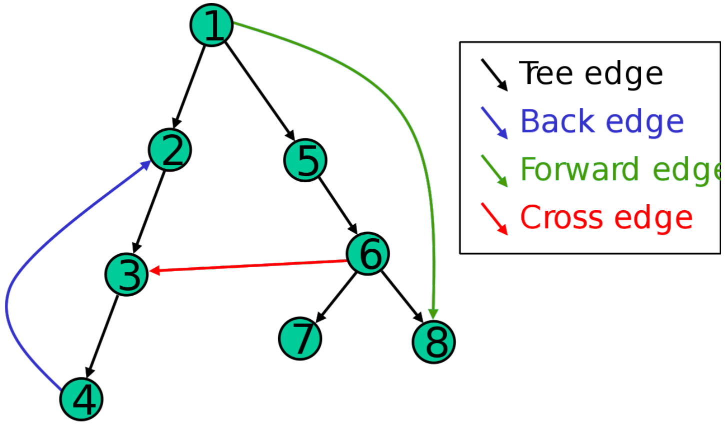
```
DFS($v$) {
    $t = t + 1$;
    $start(v) = t$;
    for each edge $(v, u) \in A[v]$ {
        if $start(u) == 0$ then DFS($u$);
    }
    $t = t + 1$;
    $finish(v) = t$;
}
```

Timestamp Example



Edge Types

DFS naturally gives a spanning forest: edge (v, u) if DFS(v) calls DFS(u)



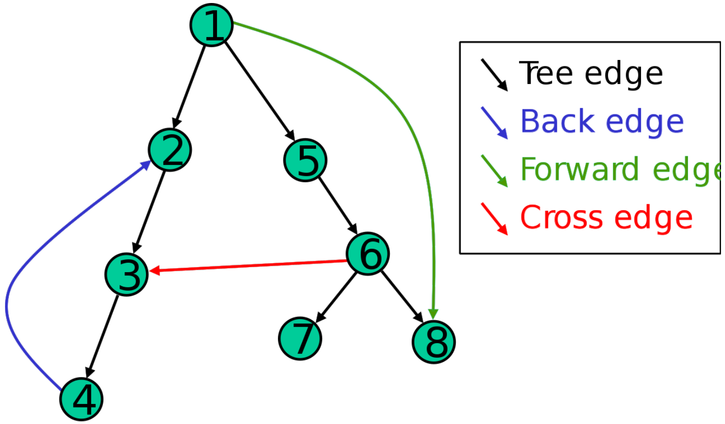
Forward Edges: (v, u) such that u descendent of v (includes tree edges)

Back Edges: (v, u) such that u an ancestor of v

Cross Edges: (v, u) such that u neither a descendent nor an ancestor of v

Edge Types

DFS naturally gives a spanning forest: edge $(\boldsymbol{v}, \boldsymbol{u})$ if DFS($\boldsymbol{v}$) calls DFS($\boldsymbol{u}$)



Forward Edges: $(\boldsymbol{v}, \boldsymbol{u})$ such that $\boldsymbol{u}$ descendent of $\boldsymbol{v}$ (includes tree edges)

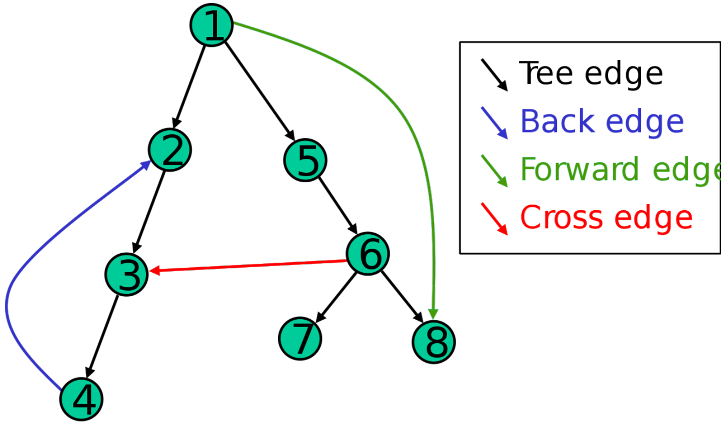
$$\boldsymbol{start}(\boldsymbol{v}) < \boldsymbol{start}(\boldsymbol{u}) < \boldsymbol{finish}(\boldsymbol{u}) < \boldsymbol{finish}(\boldsymbol{v})$$

Back Edges: $(\boldsymbol{v}, \boldsymbol{u})$ such that $\boldsymbol{u}$ an ancestor of $\boldsymbol{v}$

Cross Edges: $(\boldsymbol{v}, \boldsymbol{u})$ such that $\boldsymbol{u}$ neither a descendent nor an ancestor of $\boldsymbol{v}$

Edge Types

DFS naturally gives a spanning forest: edge $(\boldsymbol{v}, \boldsymbol{u})$ if DFS($\boldsymbol{v}$) calls DFS($\boldsymbol{u}$)



Forward Edges: $(\boldsymbol{v}, \boldsymbol{u})$ such that $\boldsymbol{u}$ descendent of $\boldsymbol{v}$ (includes tree edges)

$$\textit{start}(v) < \textit{start}(u) < \textit{finish}(u) < \textit{finish}(v)$$

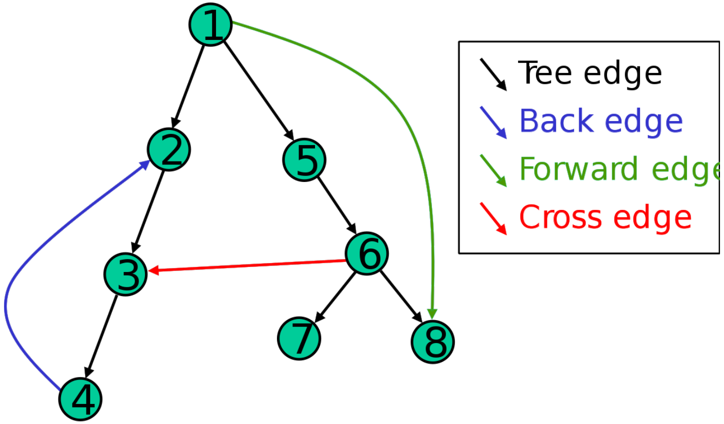
Back Edges: $(\boldsymbol{v}, \boldsymbol{u})$ such that $\boldsymbol{u}$ an ancestor of $\boldsymbol{v}$

$$\textit{start}(u) < \textit{start}(v) < \textit{finish}(v) < \textit{finish}(u)$$

Cross Edges: $(\boldsymbol{v}, \boldsymbol{u})$ such that $\boldsymbol{u}$ neither a descendent nor an ancestor of $\boldsymbol{v}$

Edge Types

DFS naturally gives a spanning forest: edge (v, u) if DFS(v) calls DFS(u)



Forward Edges: (v, u) such that u descendent of v (includes tree edges)

$$start(v) < start(u) < finish(u) < finish(v)$$

Back Edges: (v, u) such that u an ancestor of v

$$start(u) < start(v) < finish(v) < finish(u)$$

Cross Edges: (v, u) such that u neither a descendent nor an ancestor of v

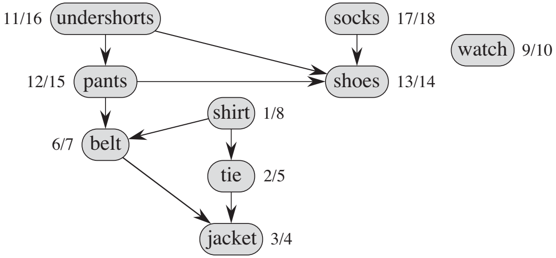
$$start(u) < finish(u) < start(v) < finish(v)$$

Topological Sort

Definitions

Definition

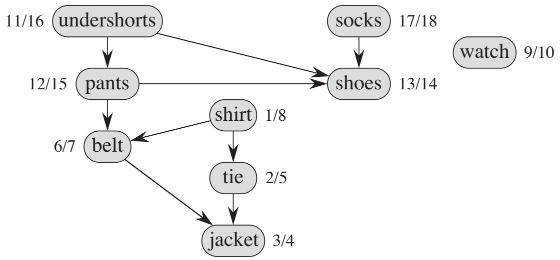
A directed graph G is a *Directed Acyclic Graph (DAG)* if it has no directed cycles.



Definitions

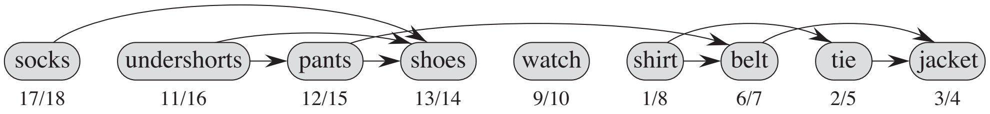
Definition

A directed graph G is a *Directed Acyclic Graph (DAG)* if it has no directed cycles.



Definition

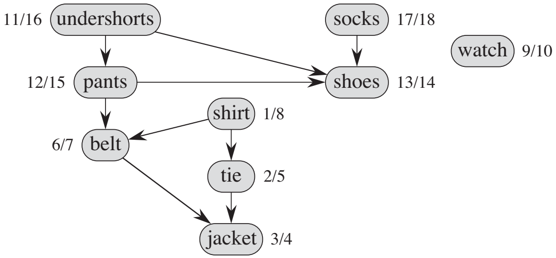
A *topological sort* $v_1, v_2, \ldots, v_n$ of a DAG is an ordering of the vertices such that all edges are of the form (v_i, v_j) with $i < j$.



Definitions

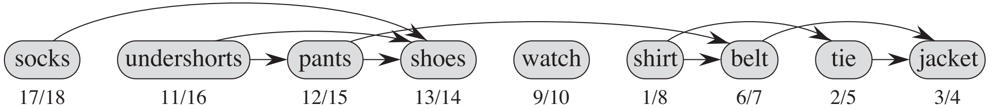
Definition

A directed graph G is a *Directed Acyclic Graph (DAG)* if it has no directed cycles.



Definition

A *topological sort* $v_1, v_2, \ldots, v_n$ of a DAG is an ordering of the vertices such that all edges are of the form (v_i, v_j) with $i < j$.



Q: Can we always topological sort a DAG? How fast?

Topological Sort

Algorithm (informal): Run DFS(G). When DFS(v) returns, put v at beginning of list

Topological Sort

Algorithm (informal): Run DFS(G). When DFS(v) returns, put v at beginning of list

```
DFS($G$) {
    $list \to head = NULL$;
    $t = 0$;
    for all $v \in V$ {
        $start(v) = 0$;
        $finish(v) = 0$;
    }
    while $\exists v \in V$ with $start(v) = 0$ {
        DFS($v$);
    }
}
```

```
DFS($v$) {
    $t = t + 1$;
    $start(v) = t$;
    for each edge $(v, u) \in A[v]$ {
        if $start(u) == 0$ then DFS($u$);
    }
    $t = t + 1$;
    $finish(v) = t$;
    $temp = list \to head$;
    $list \to head = v$;
    $list \to head \to next = temp$;
}
```

Characterizing DAGs

Theorem

A directed graph $\boldsymbol{G}$ *is a DAG if and only if DFS(*$\boldsymbol{G}$*) has no back edges.*

Characterizing DAGs

Theorem

A directed graph $\boldsymbol{G}$ *is a DAG if and only if DFS(*$\boldsymbol{G}$*) has no back edges.*

Proof.

Only if ($\Rightarrow$): contrapositive. If $\boldsymbol{G}$ has a back edge:

Characterizing DAGs

Theorem

A directed graph $\boldsymbol{G}$ is a DAG if and only if DFS($\boldsymbol{G}$) has no back edges.

Proof.

Only if ($\Rightarrow$): contrapositive. If $\boldsymbol{G}$ has a back edge: Directed cycle! Not a DAG.

Characterizing DAGs

Theorem

A directed graph $\boldsymbol{G}$ *is a DAG if and only if DFS(*$\boldsymbol{G}$*) has no back edges.*

Proof.

Only if (⇒): contrapositive. If $\boldsymbol{G}$ has a back edge: Directed cycle! Not a DAG.

If (⇐): contrapositive. If $\boldsymbol{G}$ has a directed cycle $\boldsymbol{C}$:

Characterizing DAGs

Theorem

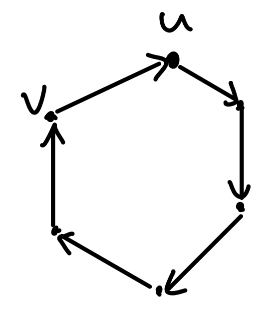
A directed graph G is a DAG if and only if DFS(G) has no back edges.

Proof.

Only if ($\Rightarrow$): contrapositive. If G has a back edge: Directed cycle! Not a DAG.

If ($\Leftarrow$): contrapositive. If G has a directed cycle C:

- Let $u \in C$ with minimum start value, v predecessor in cycle
- All nodes in C reachable from $u \implies$ all nodes in C descendants of u
- (v, u) a back edge



☐

Topological Sort Analysis

Correctness: Since G a DAG, never see back edge

$\implies$ Every edge (v, u) out of v a forward or cross edge

$\implies$ $\textit{finish}(u) < \textit{finish}(v)$

$\implies$ u already in list when v added to beginning

Topological Sort Analysis

Correctness: Since G a DAG, never see back edge

$\implies$ Every edge (v, u) out of v a forward or cross edge

$\implies$ $\mathit{finish}(u) < \mathit{finish}(v)$

$\implies$ u already in list when v added to beginning

Running Time: Same as DFS! $O(m + n)$