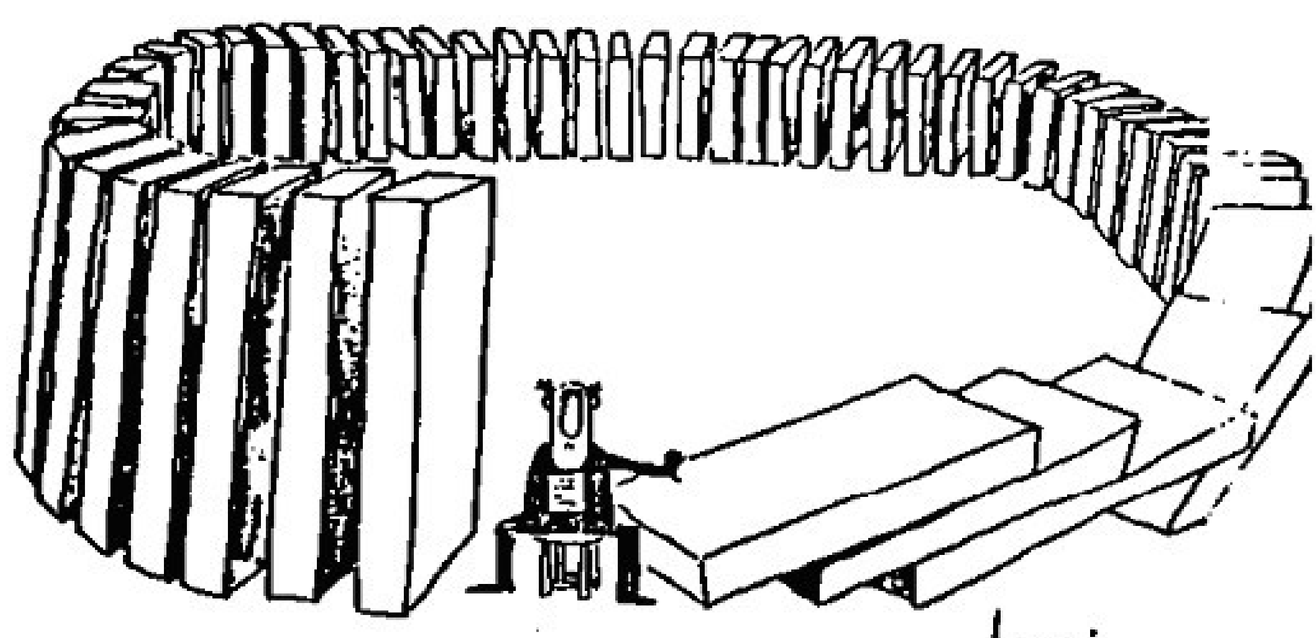
Unweighted directed graphs



Drawing by Lewin; © 1976 The New Yorker Magazine, Inc.

Announcements

Midterm & gradescope

- will get an email today to register (username name is your email)
- tests should appear next Tuesday (nothing there now)

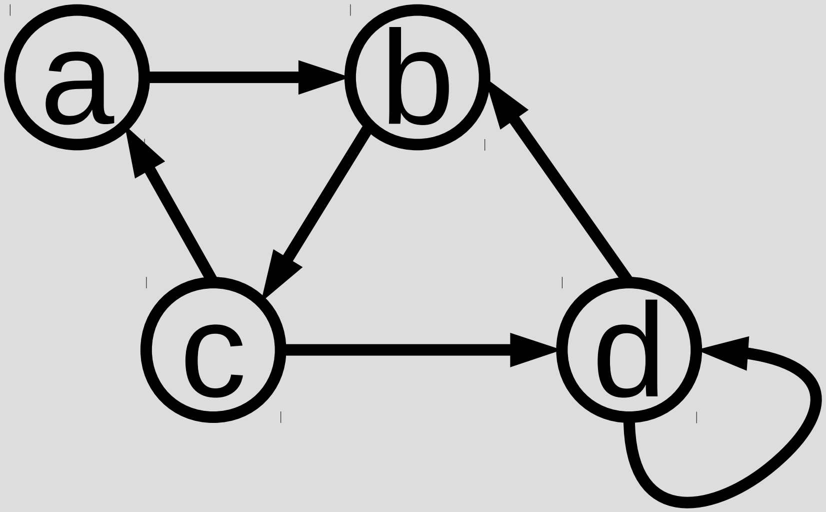
Graph

A directed graph G is a set of edges and vertices: G = (V, E)

Two common ways to represent a graph:

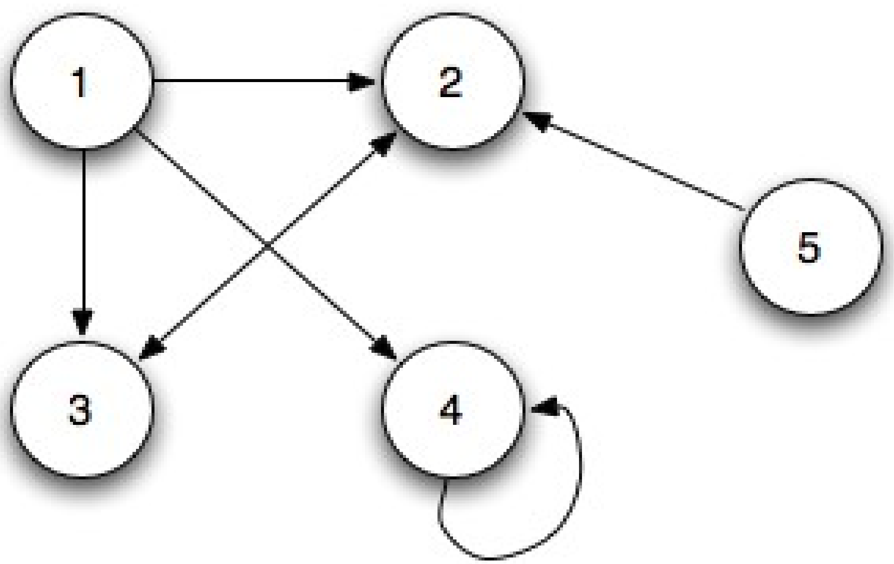
-Adjacency matrix

-Adjacency list



Graph

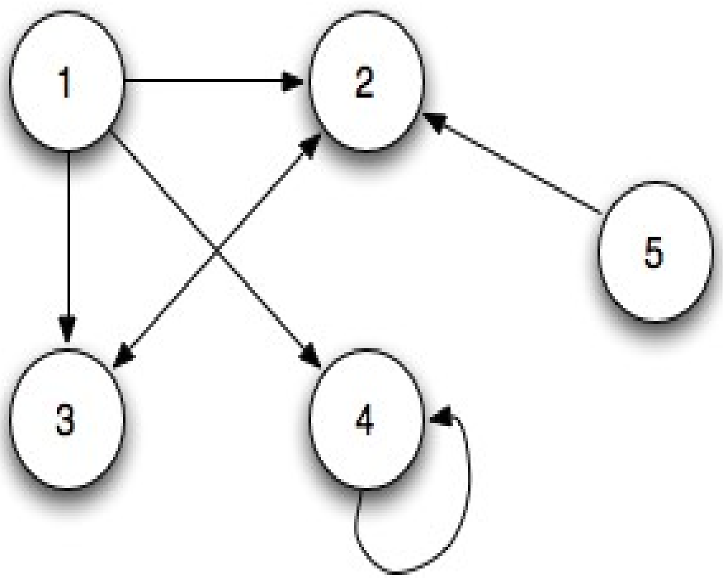
An adjacency matrix has a 1 in row i and column j if you can go from node i to node j



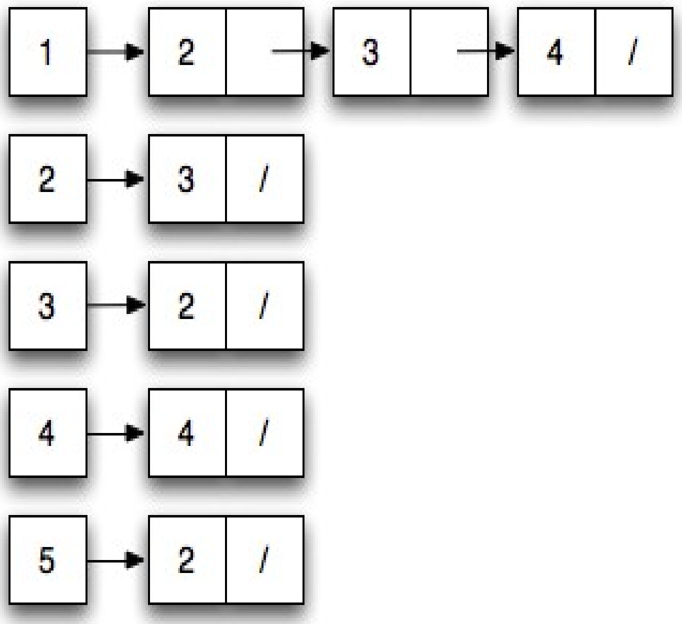
	1	2	3	4	5
1	0	1	1	1	0
2	0	0	1	0	0
3	0	1	0	0	0
4	0	0	0	1	0
5	0	1	0	0	0

Graph

An adjacency list just makes lists out of each row (list of edges out from every vertex)



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



Graph

Difference between adjacency matrix and adjacency list?

Graph

Difference between adjacency matrix and adjacency list?

Matrix is more memory $O(|V|^2)$, less computation: $O(1)$ lookup

List is less memory $O(E+V)$ if sparse, more computation: O(branch factor)

Graph

Adjacency matrix, $A=A^1$, represents the number of paths from row node to column node in 1 step

Prove: A^n is the number of paths from row node to column node in n steps

Graph

Proof: Induction

Base: $A^0 = I$, 0 steps from i is i

Induction: (Assume A^n, show A^{n+1})

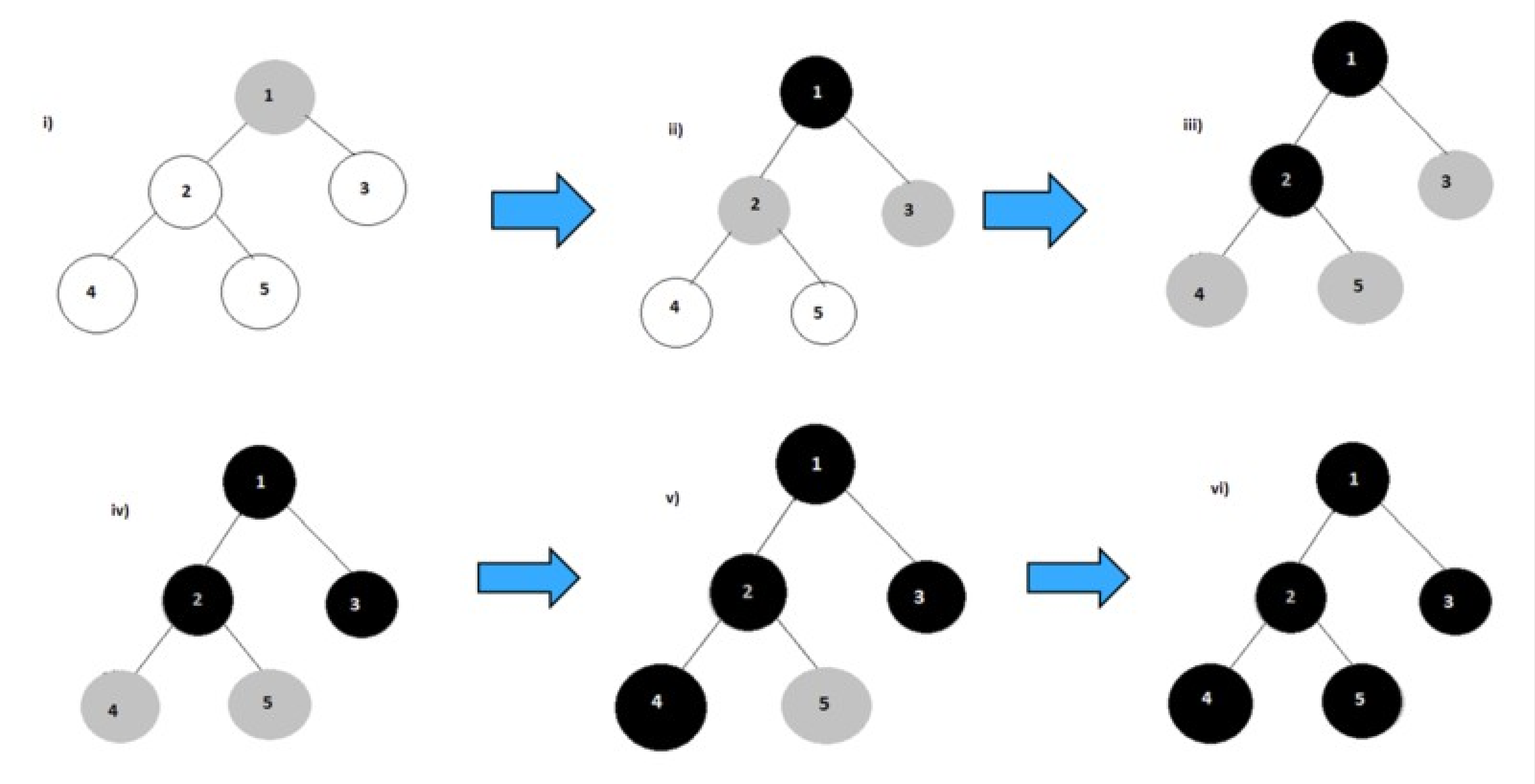
Let $a^n_{i,j}$ = i^{th} row, j^{th} column of A^n

Then $a^{n+1}_{i,j} = \sum_k a^n_{i,k} \, a^1_{k,j}$

This is just matrix multiplication

Breadth First Search Overview

Create first-in-first-out (FIFO) queue to explore unvisited nodes

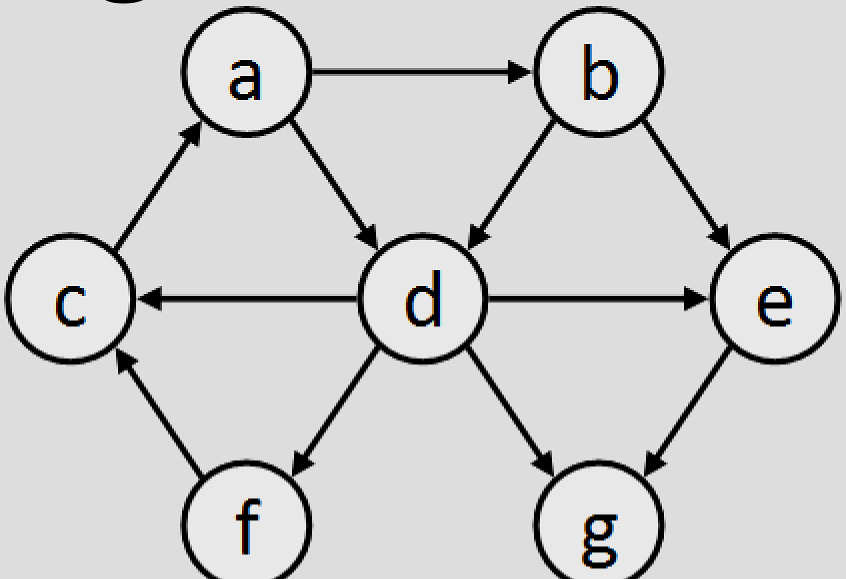


https://www.youtube.com/watch?v=nI0dT288VLs

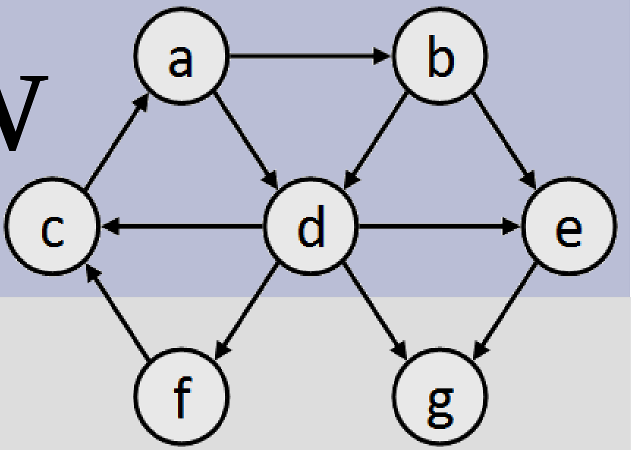
Breadth First Search Overview

Consider the graph below

Suppose we wanted to get from “a” to “c” using breadth first search



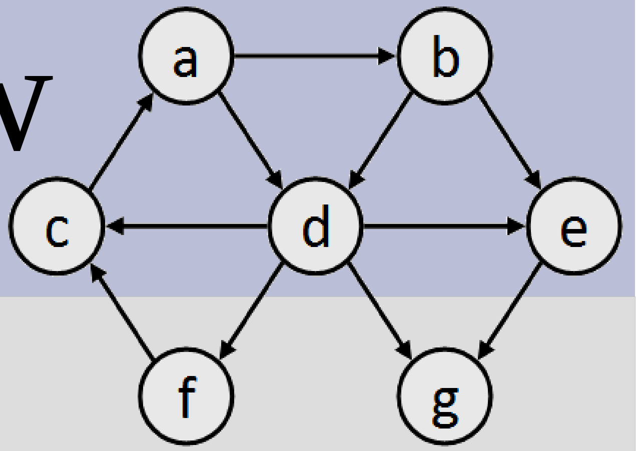
BFS Overview



To keep track of which nodes we have seen, we will do:

White nodes = never seen before
Grey nodes = nodes in Q
Black nodes = nodes that are done
To keep track of who first saw nodes I will make red arrows (π in book)

BFS Overview



First, we add the start to the queue, so Q = {a}

Then we will repeatedly take the left-most item in Q and add all of its neighbors (that we haven't seen yet) to the Q on the right

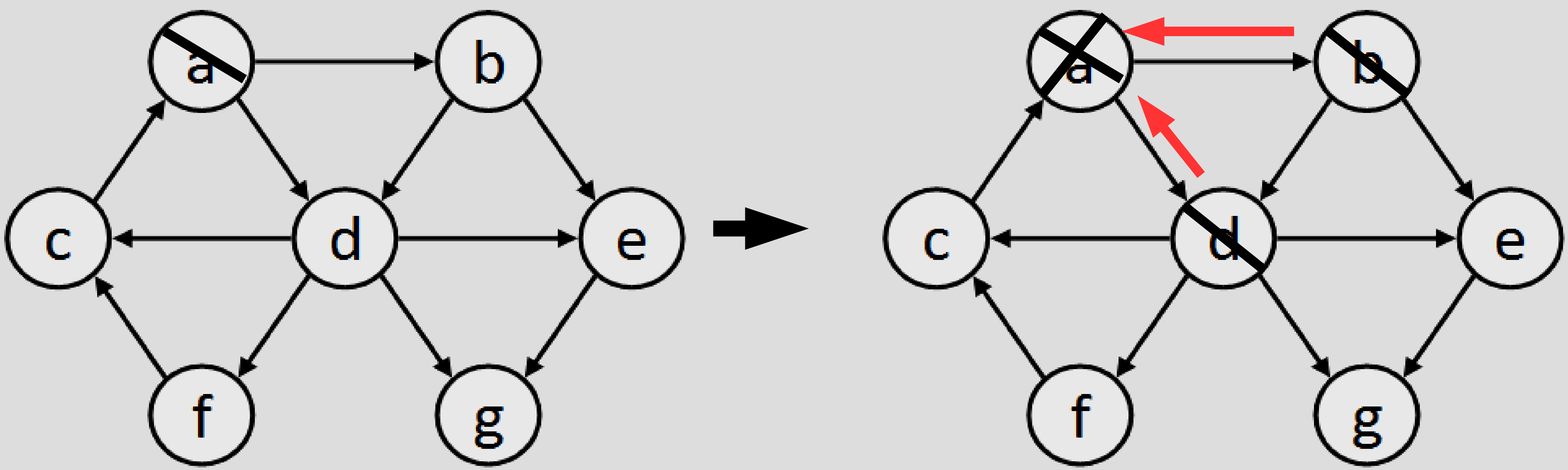
BFS Overview

Q = {a}

Left-most = a

White neighbors = b & d

New Q = {b, d}



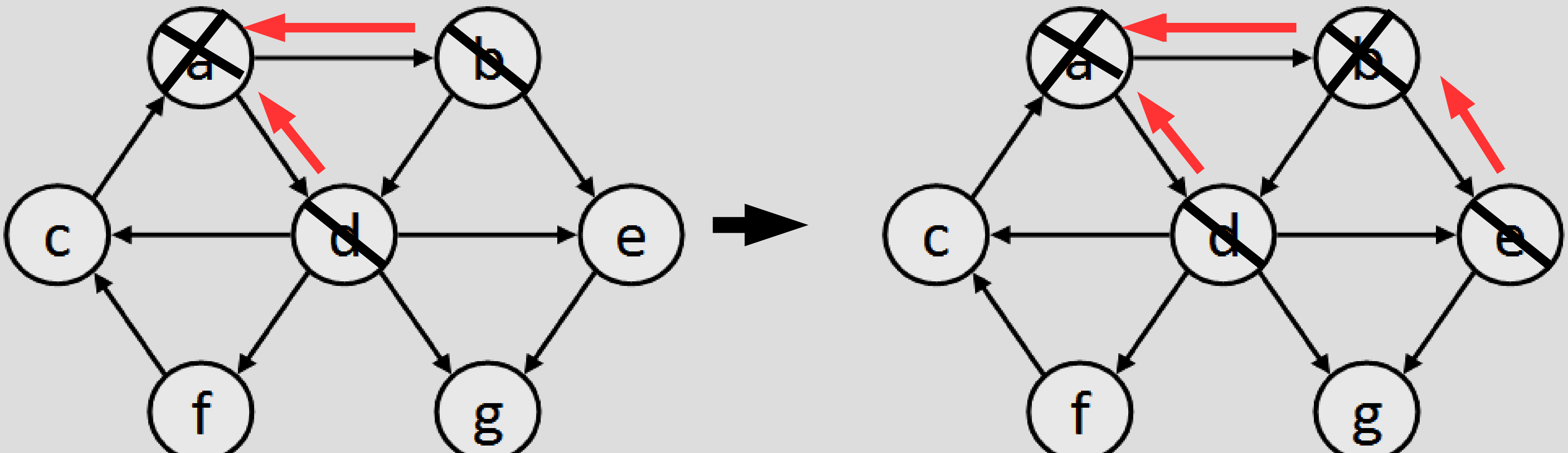
BFS Overview

Q = {b, d}

Left-most = b

White neighbors = e

New Q = {d, e}



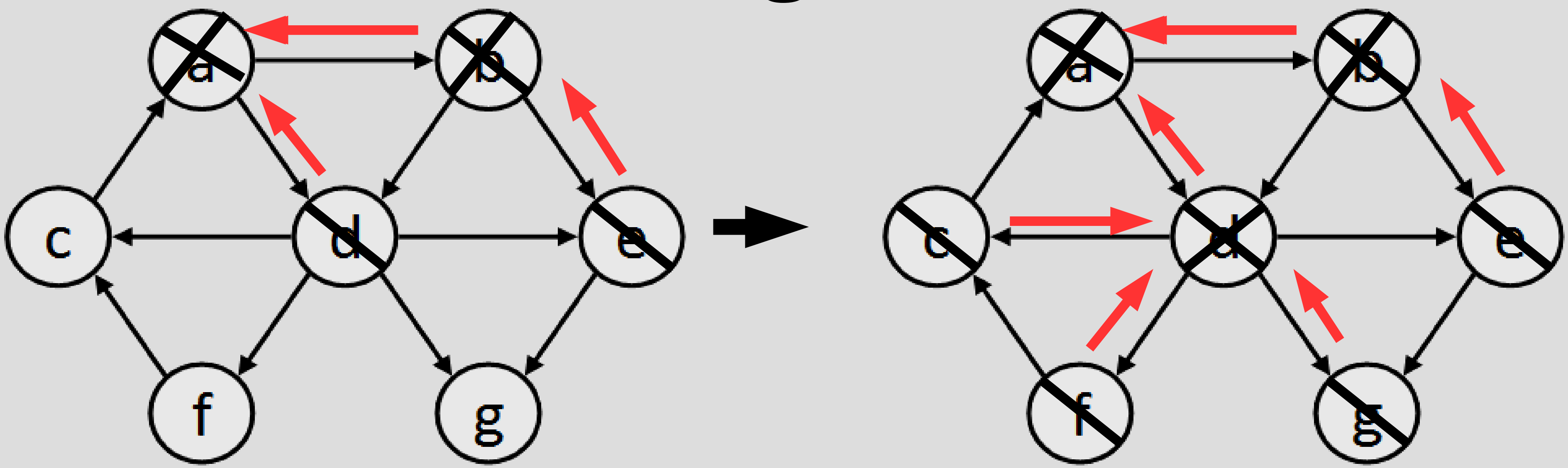
BFS Overview

Q = {d, e}

Left-most = d

White neighbors = c & f & g

New Q = {e, c, f, g}



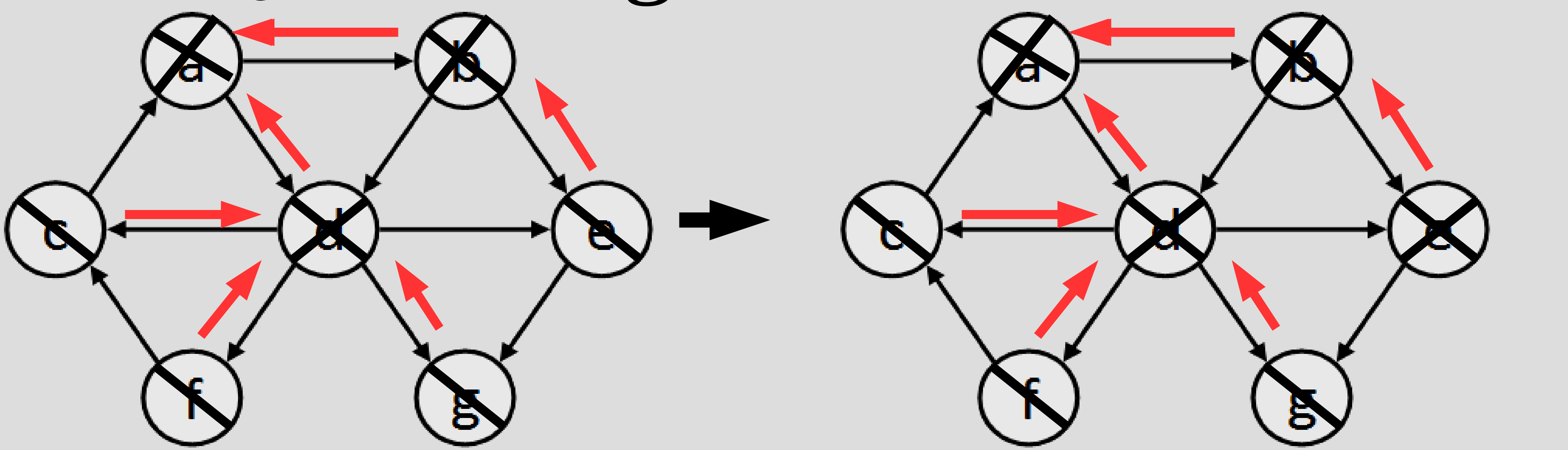
BFS Overview

Q = {e, c, f, g}

Left-most = e

White neighbors = (none)

New Q = {c, f, g}

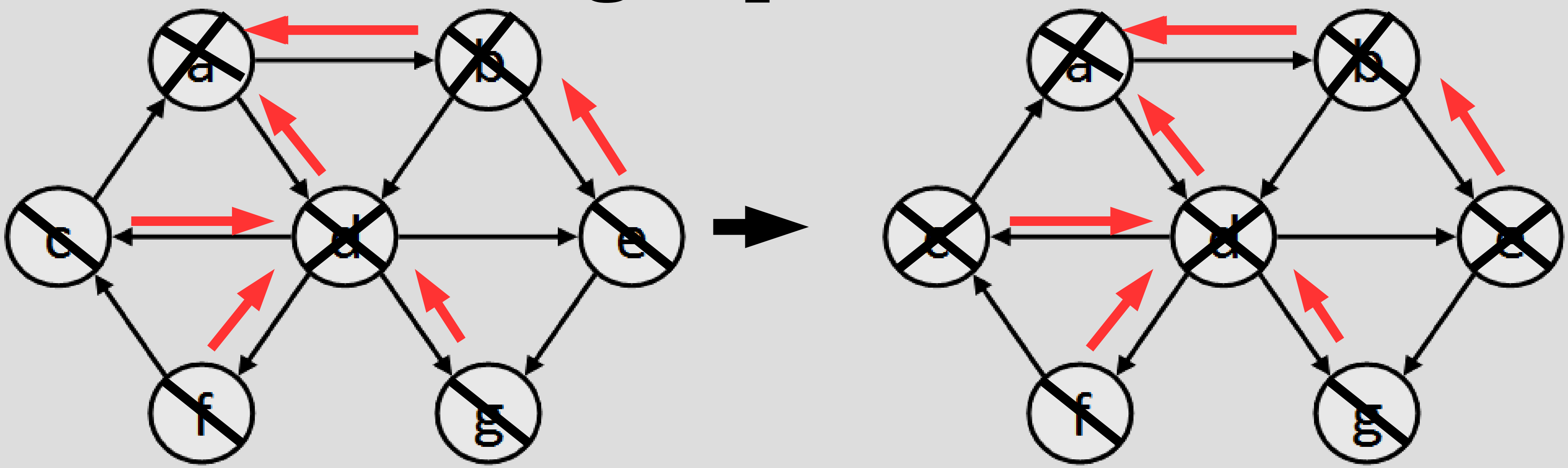


BFS Overview

Q = {c, f, g}

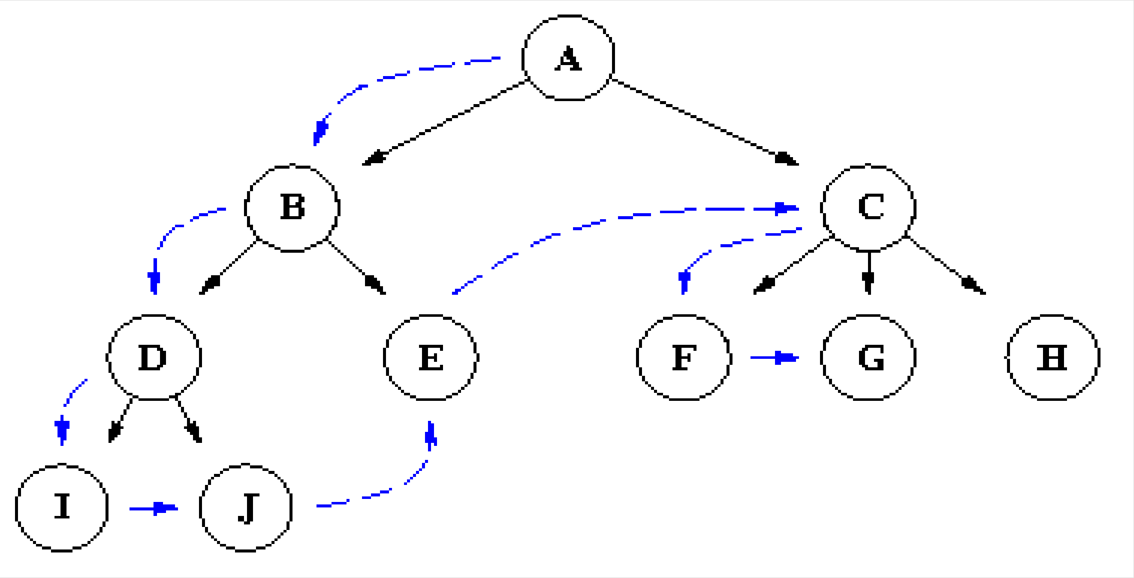
Left-most = c

Done! We found c, backtrack on red arrows to get path from “a”



Depth First Search Overview

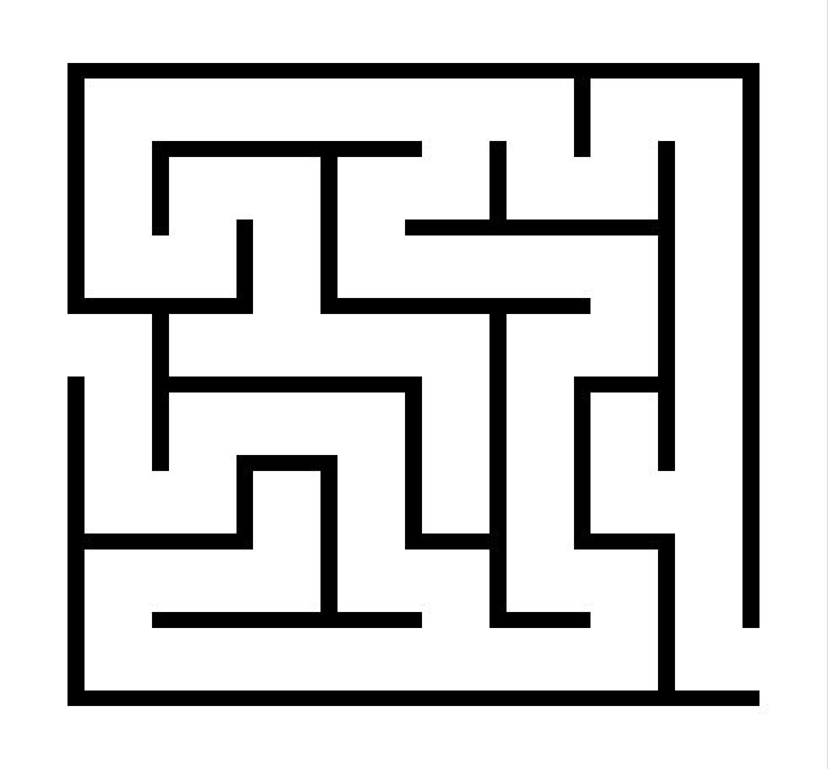
Create first-in-last-out (FILO) queue to explore unvisited nodes



Depth First Search Overview

You can solve mazes by putting your left-hand on the wall and following it

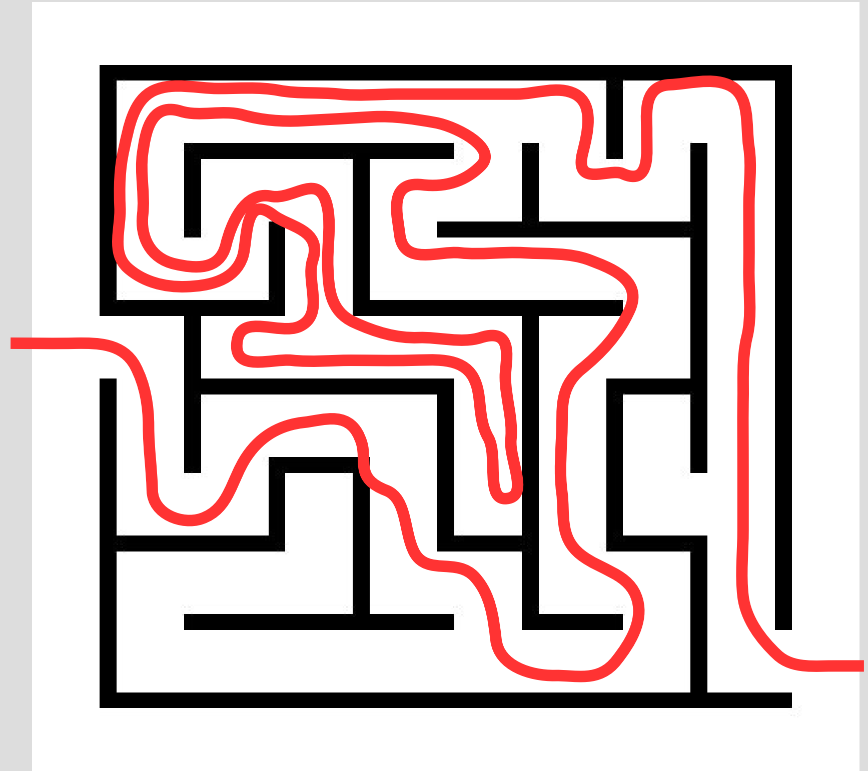
(i.e. left turns at every intersection)



Depth First Search Overview

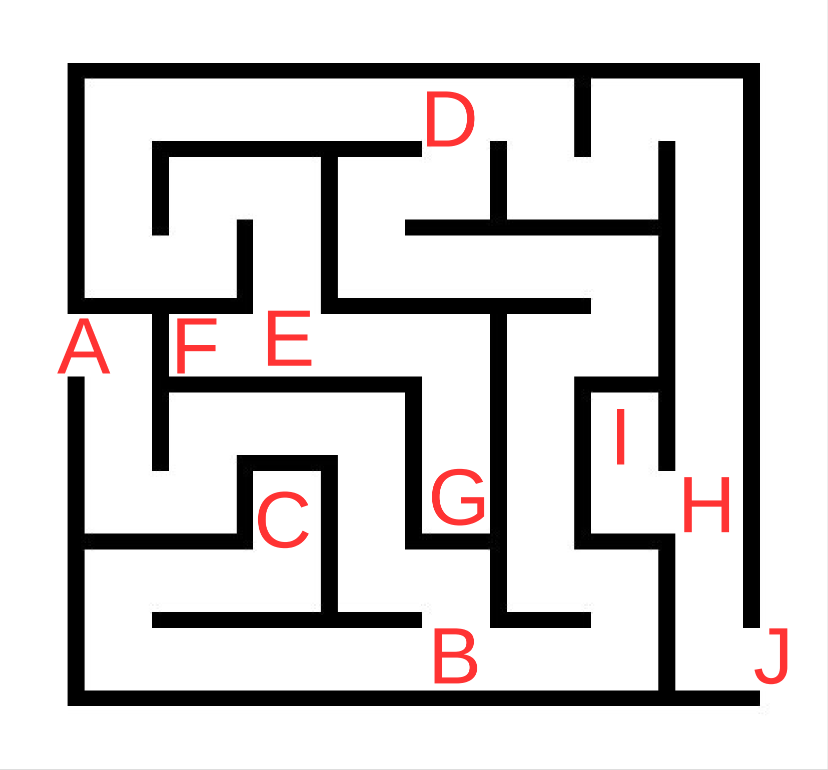
You can solve mazes by putting your left-hand on the wall and following it

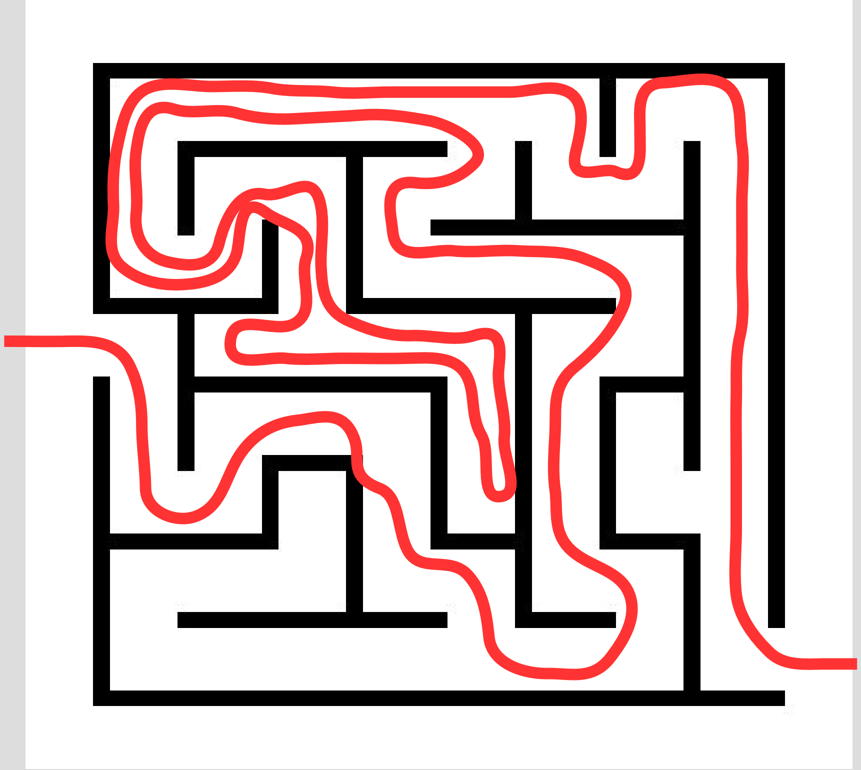
(i.e. left turns at every intersection)



Depth First Search Overview

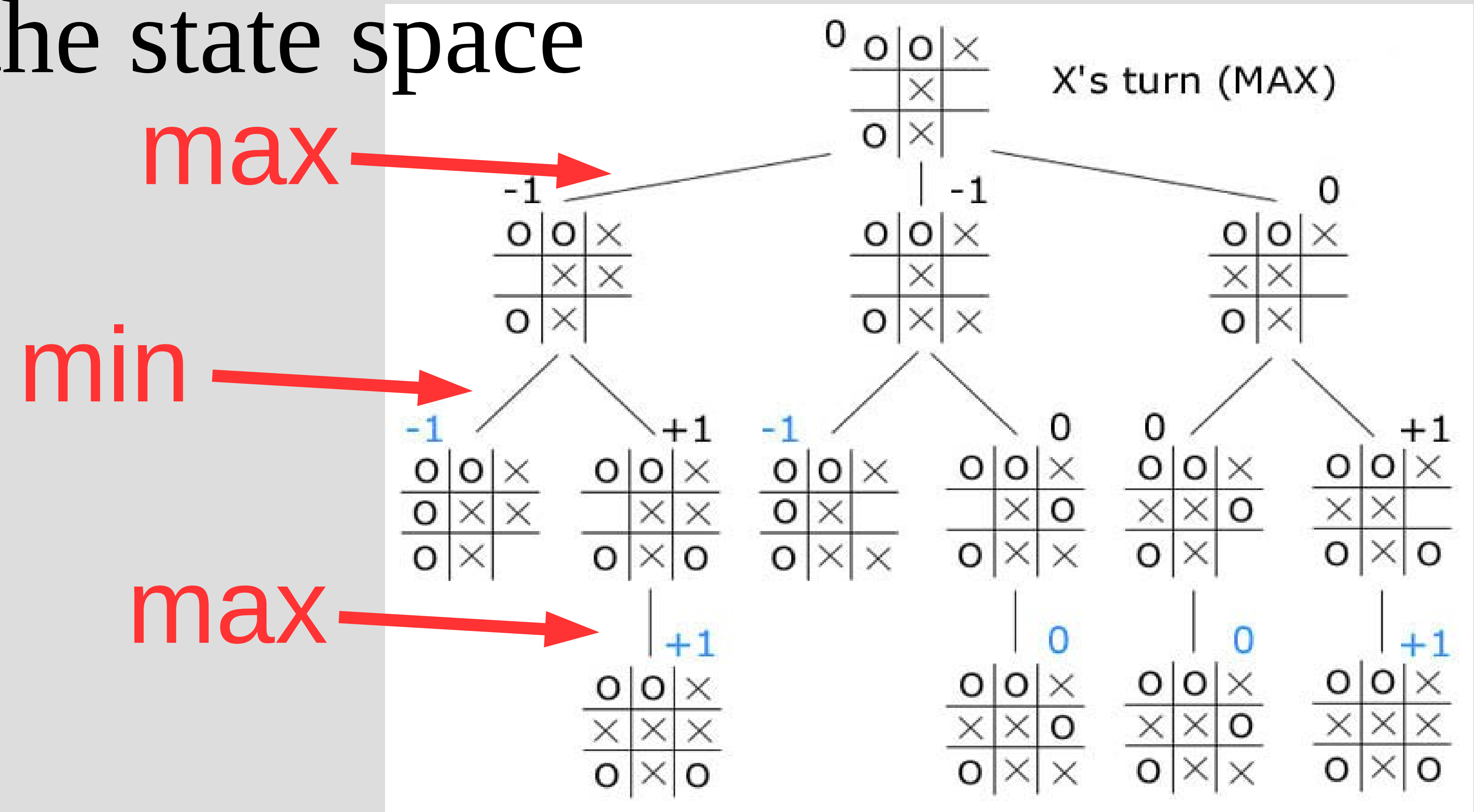
This is actually just depth first search





BFS and DFS in trees

Solve problems by making a tree of the state space



BFS and DFS in trees

Often times, fully exploring the state space is too costly (takes forever)

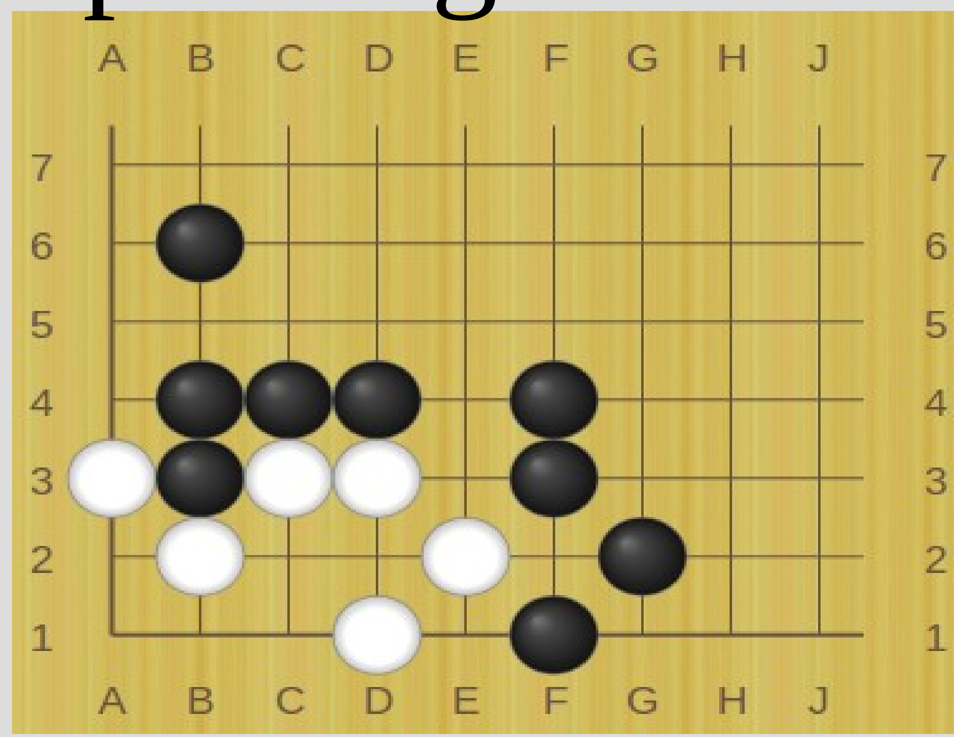
Chess: 10^{47} states (tree about 10^{123})
Go: 10^{171} states (tree about 10^{360})
At 1 million states per second...
Chess: 10^{109} years (past heat death
Go: 10^{346} years of universe)

BFS and DFS in trees

BFS prioritizes “exploring”

DFS prioritizes “exploiting”

White to move



Black to move

BFS and DFS in trees

BFS benefits?

DFS benefits?

BFS and DFS in trees

BFS benefits?

-if stopped before full search, can evaluate best found

DFS benefits?

-uses less memory on complete search

BFS and DFS in graphs

BFS: shortest path from origin to any node

DFS: find graph structure

Both running time of $O(V+E)$

Breadth first search

```
BFS(G,s) // to find shortest path from s
for all v in V
    v.color=white, v.d=∞,v.π=NIL
s.color=grey, v.d=0
Enqueue(Q,s)
while(Q not empty)
    u = Dequeue(Q,s)
    for v in G.adj[u]
        if v.color == white
            v.color=grey, v.d=u.d+1, v.π=u
            Enqueue(Q,v)
    u.color=black
```

Breadth first search

Let $\delta(s,v)$ be the shortest path from s to v

After running BFS you can find this path as: $v.\pi$ to $(v.\pi).\pi$ to ... s

(pseudo code on p. 601, recursion)

BFS correctness

Proof: contradiction

Assume $\delta(s,v) \neq v.d$

$v.d \geq \delta(s,v)$ (Lemma 22.2, induction)

Thus $v.d > \delta(s,v)$

Let u be previous node on $\delta(s,v)$

Thus $\delta(s,v) = \delta(s,u)+1$

and $\delta(s,u) = u.d$

Then $v.d > \delta(s,v) = \delta(s,u)+1 = u.d+1$

BFS correctness

$v.d > \delta(s,v) = \delta(s,u)+1 = u.d+1$

Cases on color of v when u dequeue,
all cases invalidate top equation

Case white: alg sets $v.d = u.d + 1$

Case black: already removed
thus $v.d \leq u.d$ (corollary 22.4)

Case grey: exists w that dequeued v,
$v.d = w.d+1 \leq u.d+1$ (corollary 22.4)

Depth first search

DFS can be implemented with BFS

We will mark both a start (colored grey) and finish (colored black) times

This helps us quantify properties of graphs

Depth first search

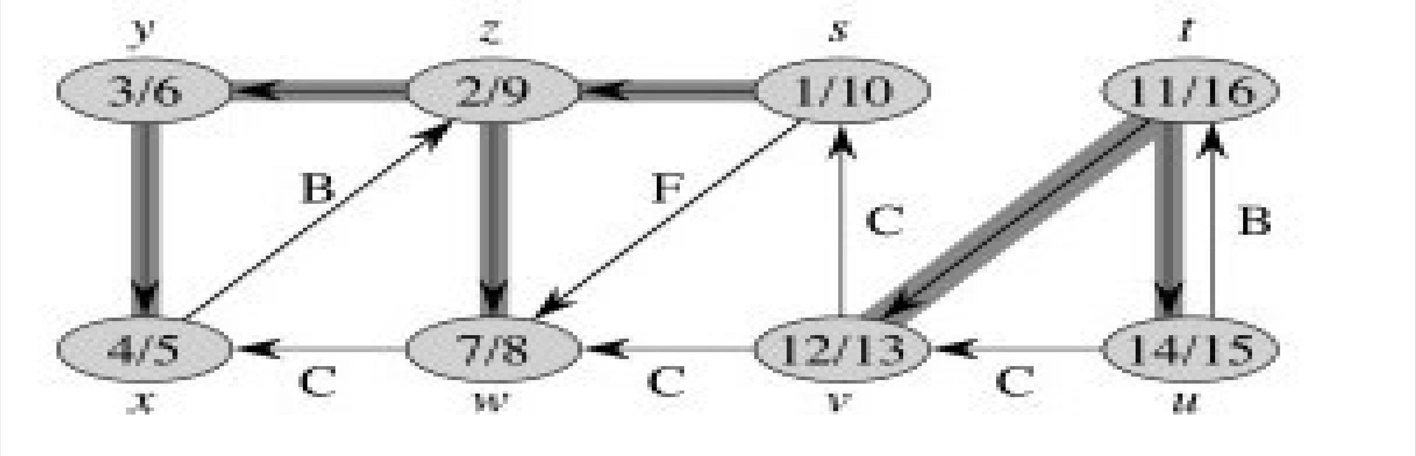
```
DFS(G)
for all v in V
   v.color=white, v.π=NIL
time=0
for each v in V
   if v.color==white
      DFS-Visit(G,v)
```

Depth first search

```
DFS-Visit(G,u)
time=time+1
u.d=time, u.color=grey
for each v in G.adj[u]
   if v.color == white
      v.π=u
      DFS-Visit(G,v)
u.color=black, time=time+1, u.f=time
```

Depth first search

Edge markers:



Consider edge u to v

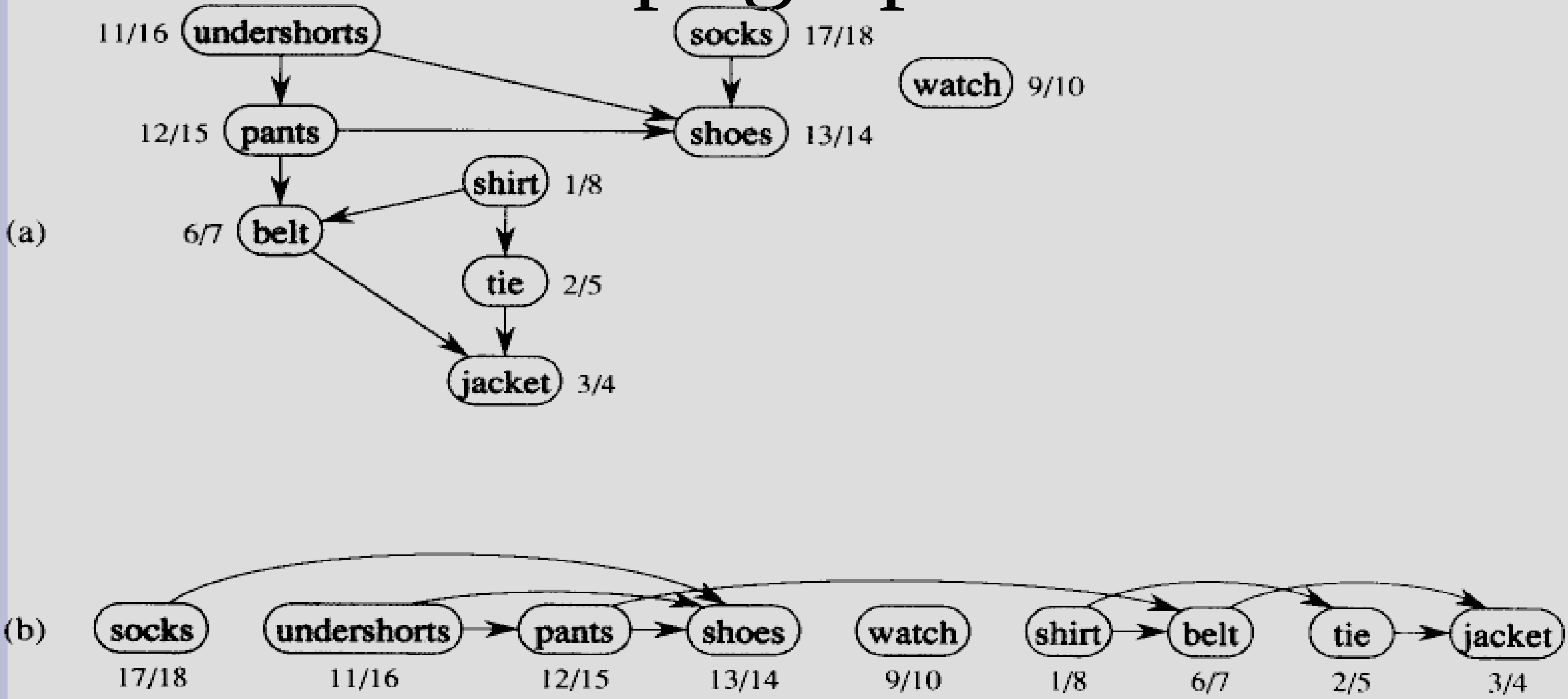
B = Edge to grey node (u.f < v.f)

F = Edge to black node (u.f > v.f)

C = Edge to black node (u.d > v.f)

Depth first search

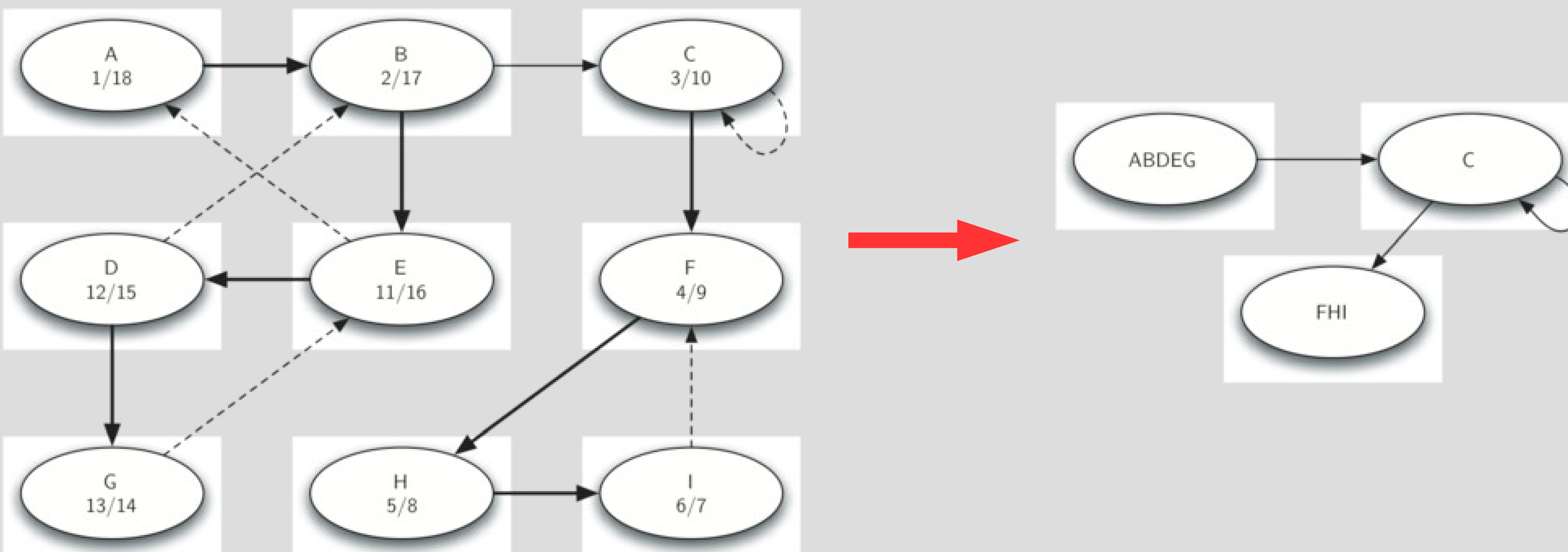
DFS can do topographical sort



Run DFS, sort in decreasing finish time

Depth first search

DFS can find strongly connected components



Depth first search

Let G^T be G with edges reversed

Then to get strongly connected:

1. DFS(G) to get finish times
2. Compute G^T
3. DFS(G^T) on vertex in decreasing finish time
4. Each tree in forest SC component