Elementary Graph Algorithms

CLRS 20

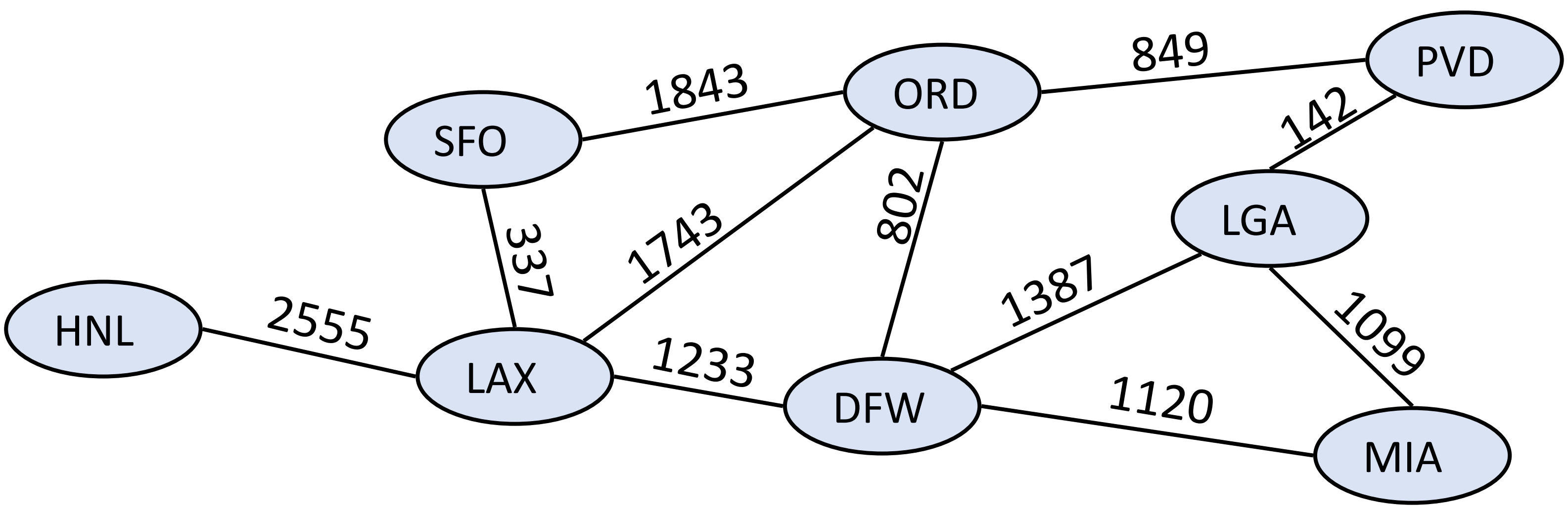
(+ many supplemental material)

Graph

- A **graph** is a pair ***(V, E)*** where
  - ***V*** is a set of **vertices**
  - ***E*** is a collection of pairs of vertices, called **edges**
  - Vertices and edges are positions and store elements
  - Edges can be directed (an ordered pair) or undirected (unordered)

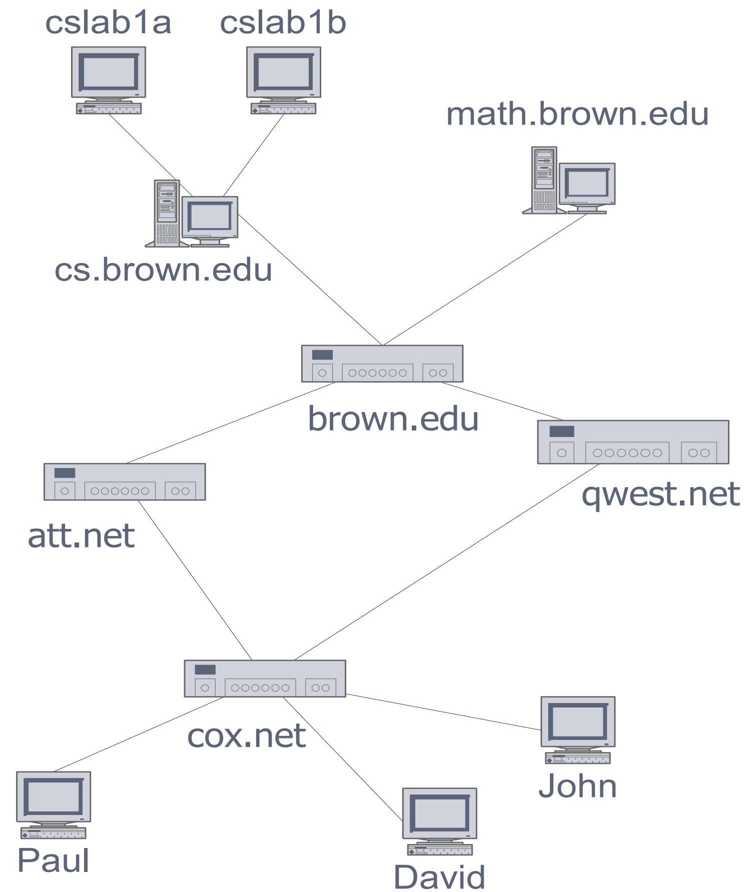
Example:

- A vertex represents an airport and stores the three-letter airport code
- An edge represents a flight route between two airports and stores the mileage of the route



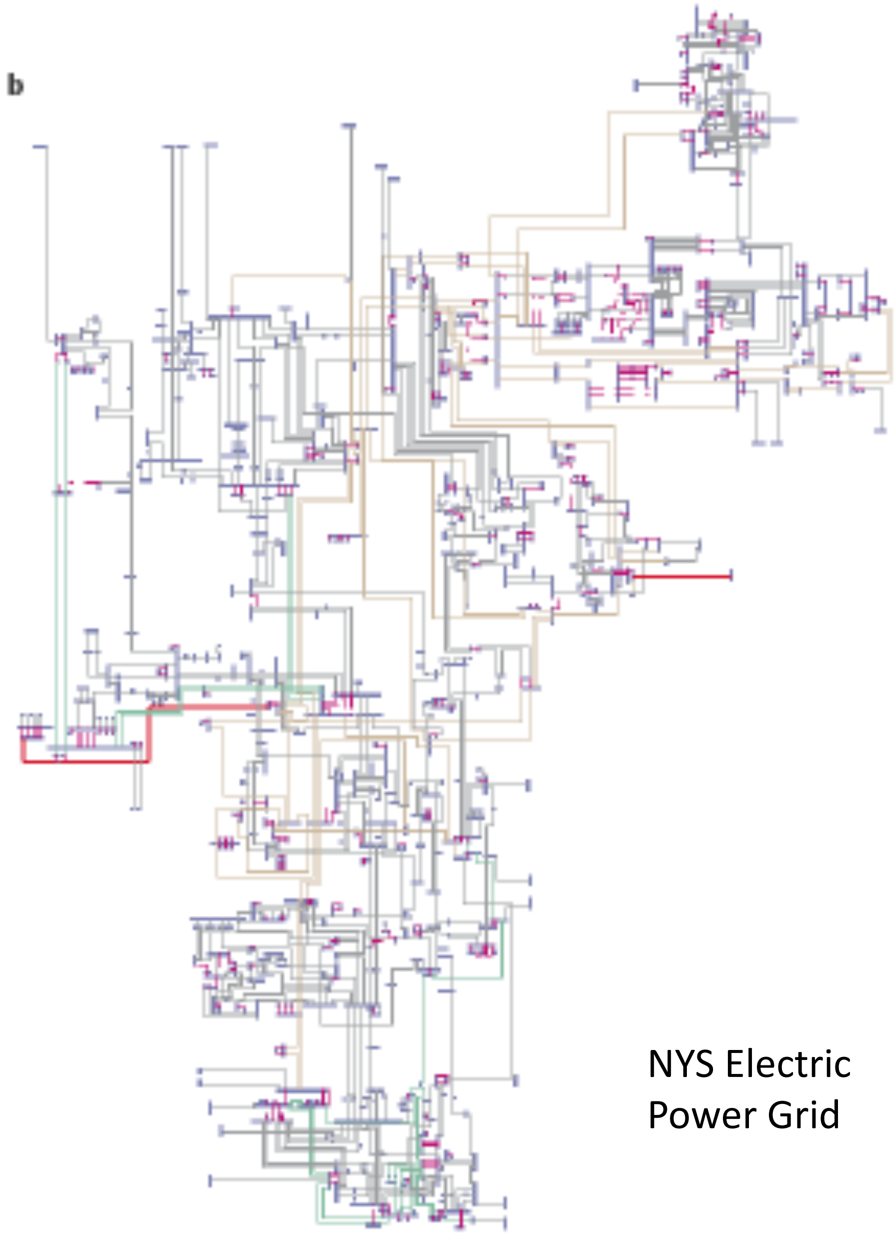
Applications & uses

- Electronic circuits
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases (Entity-relationship diagram)



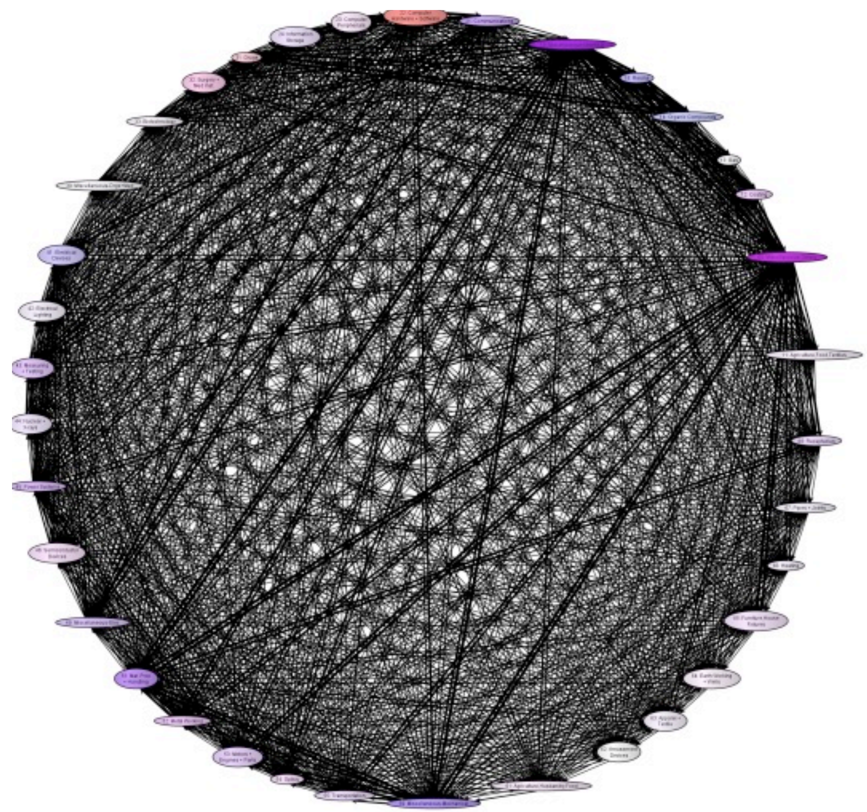
- Other
  - Spread (of disease, of disinformation)
  - Facility location
  - Finding influential nodes (social network, terrorist network)
  - Routing traffic (cell phone towers, internal traffic, car traffic)

Real-world networks



NYS Electric
Power Grid

Utility Patent network
1972-1999
(3 Million patents)

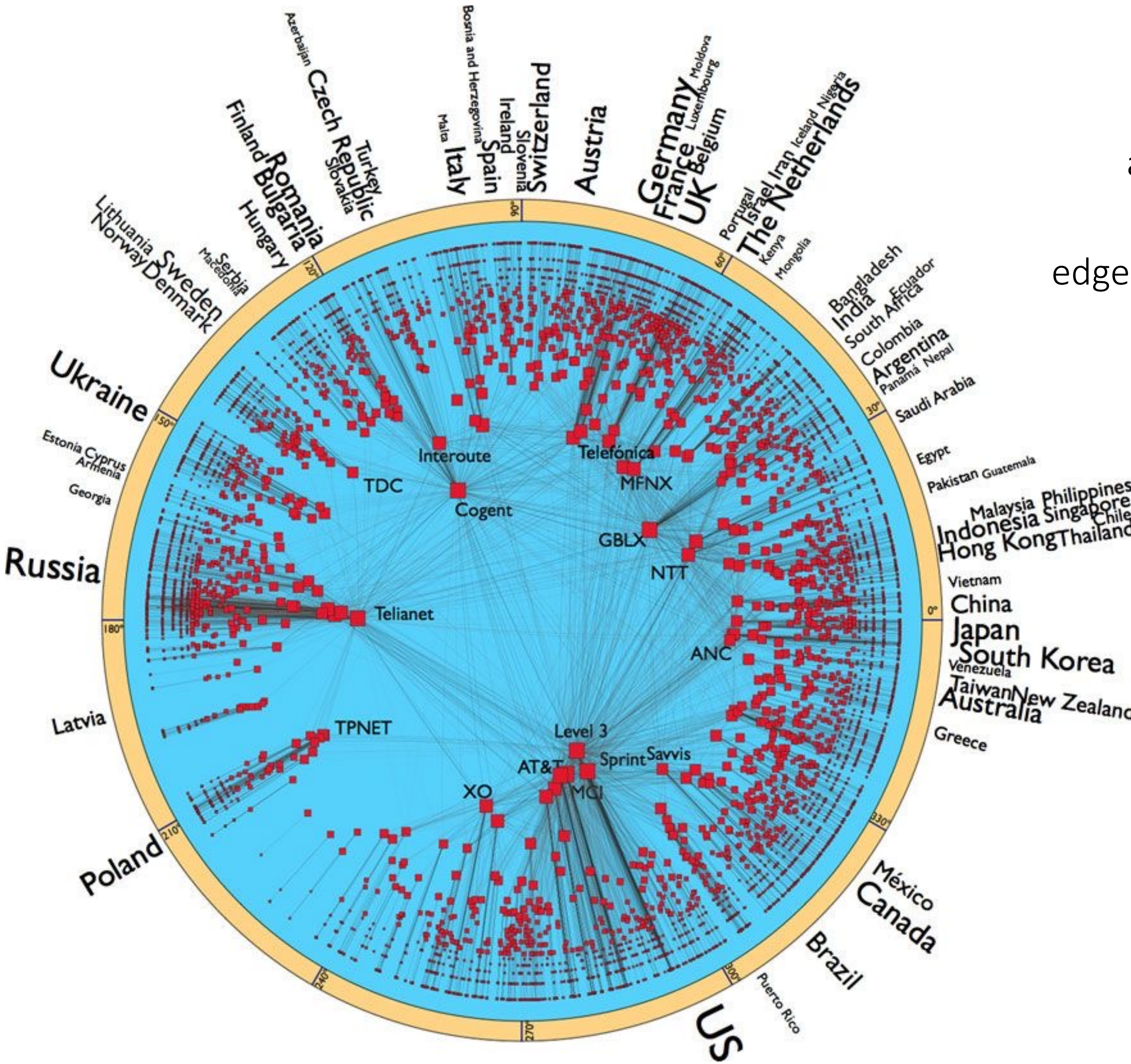


Real-world networks

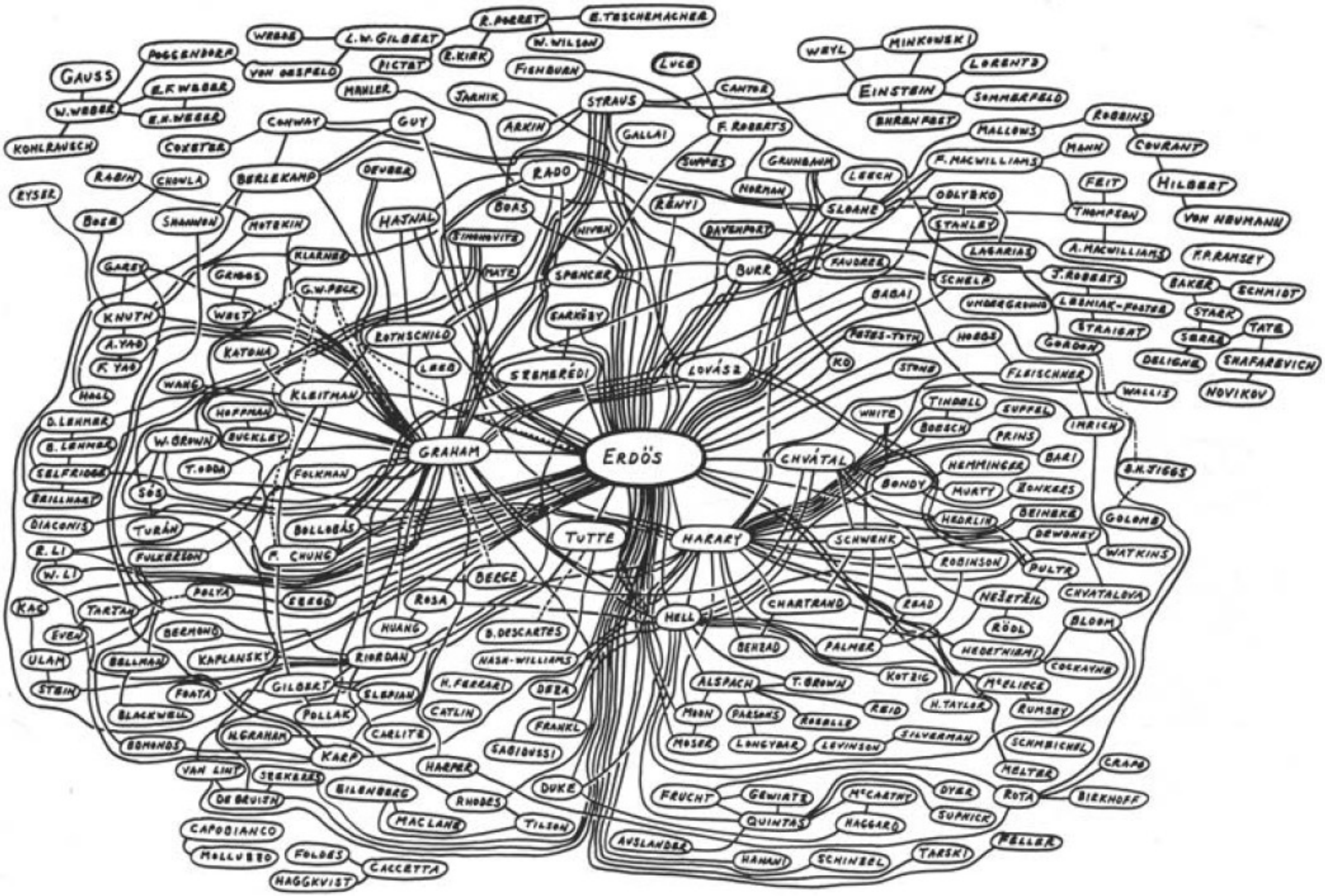
Internet (AS-level)

nodes n = 23,752 autonomous systems

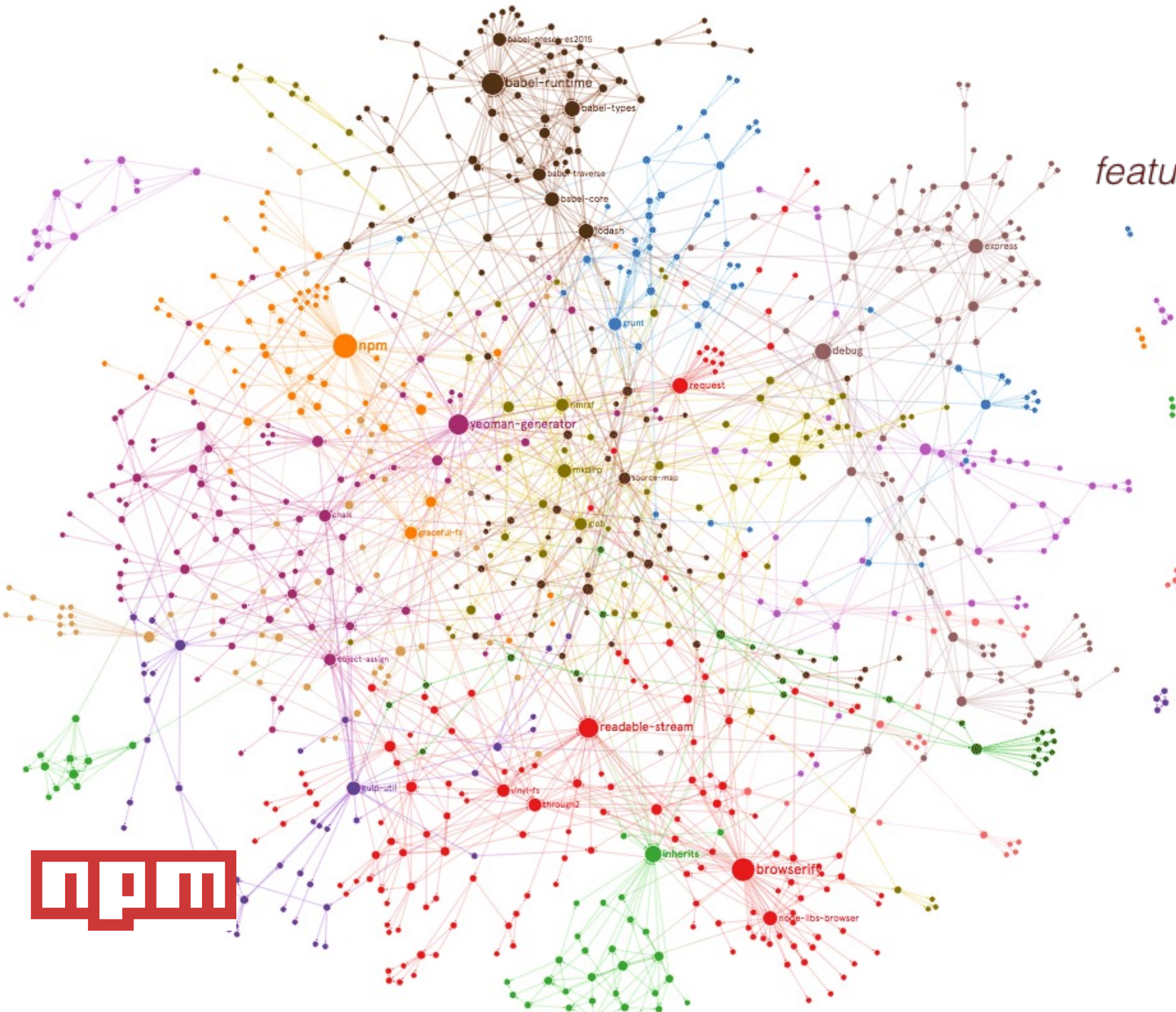
edges m = 58,416 AS links

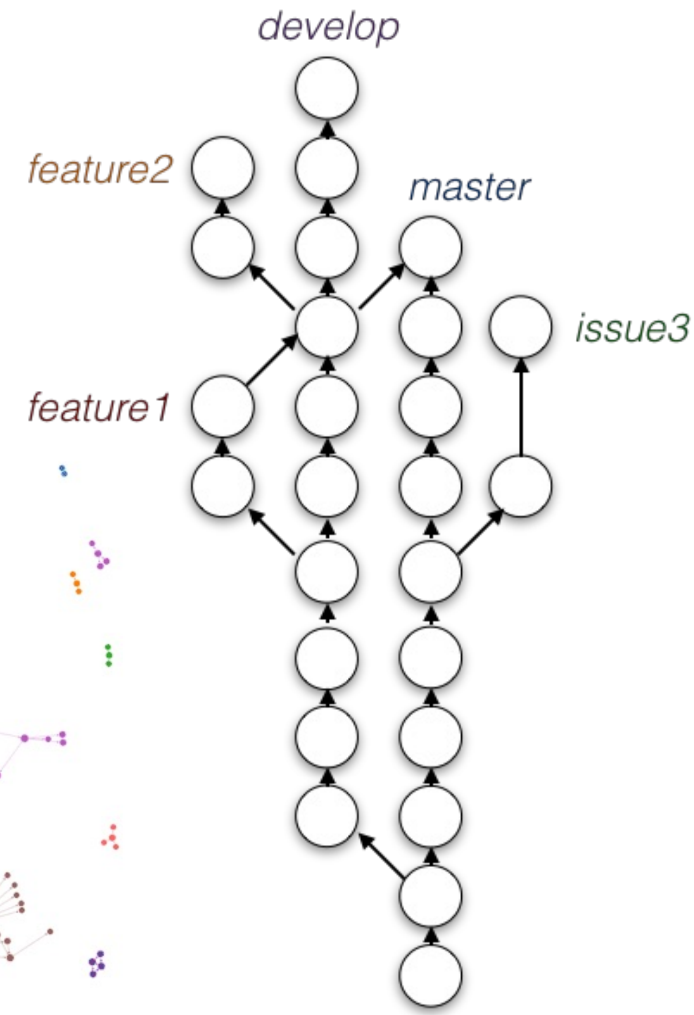


Real-world networks

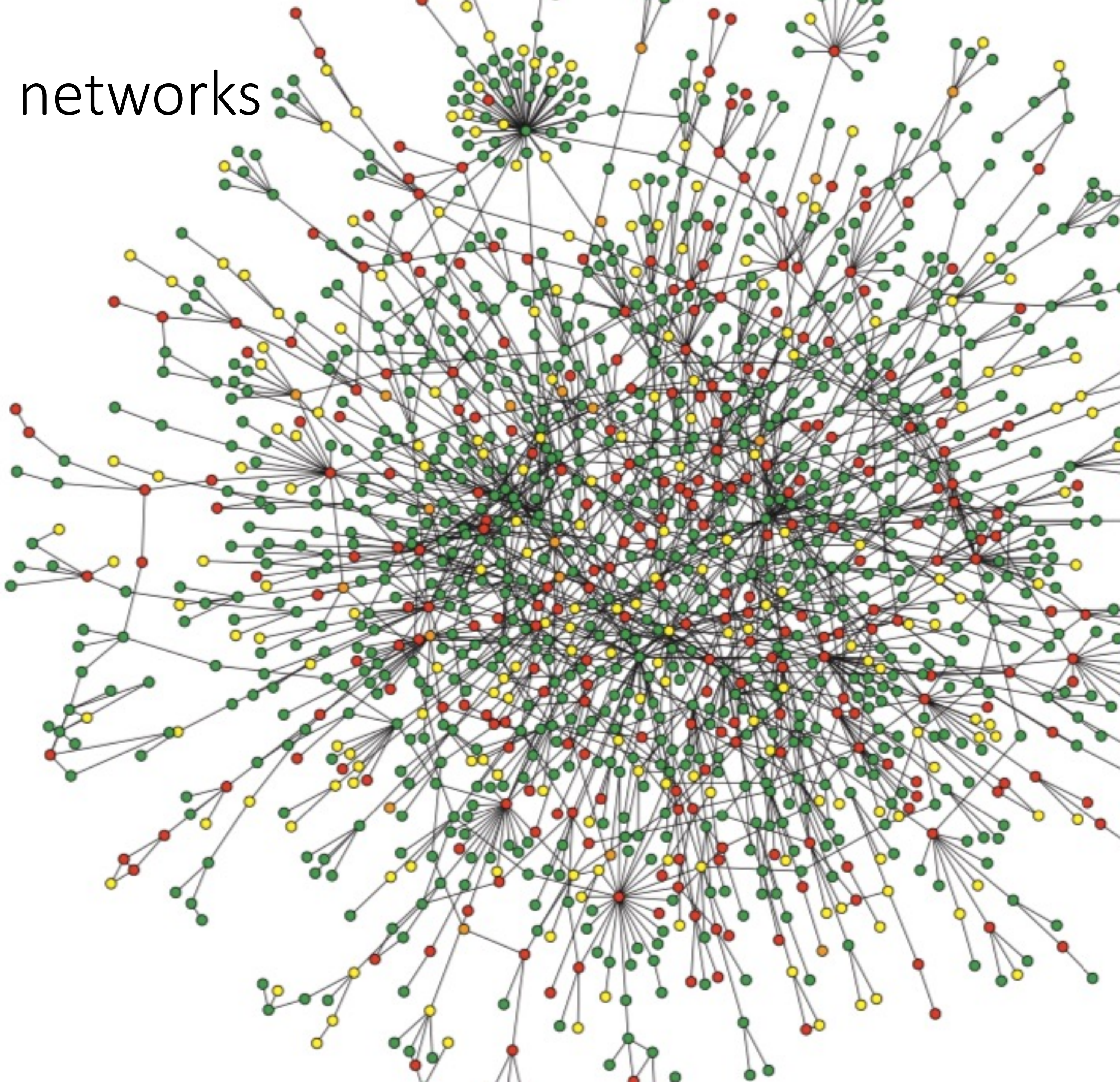


Real-world networks



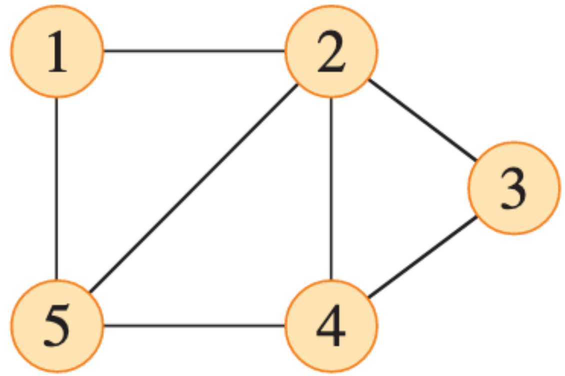


Real-world networks

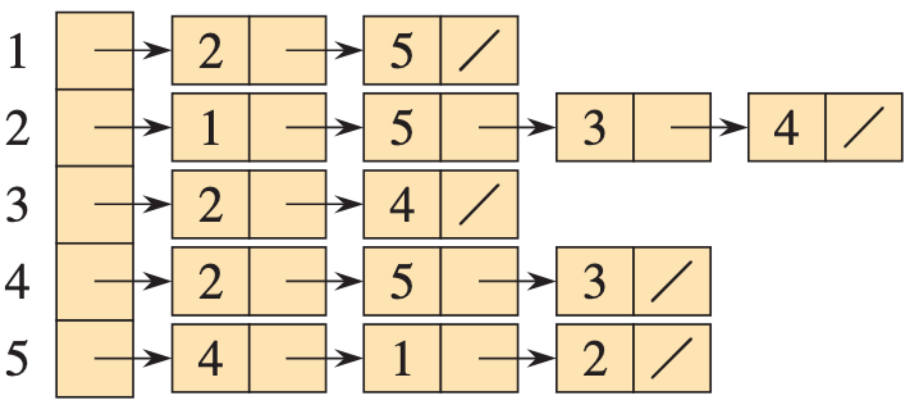


Protein interaction network

Ex: undirected graph G with 5 vertices & 7 edges



Adjacency list representation



Adjacency matrix representation

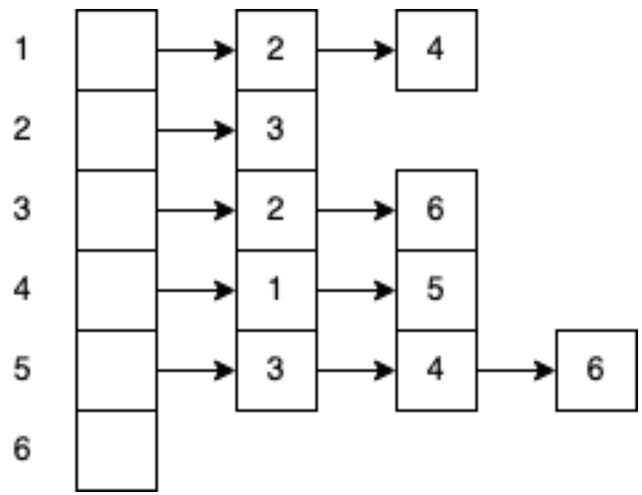
$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

Space: $O(n+m)$
Good for **sparse** graphs, i.e., $m = O(n)$

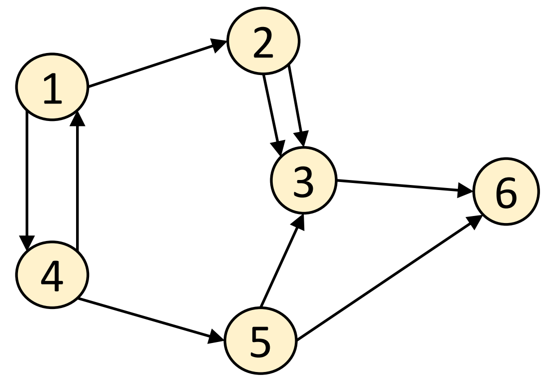
Space: $O(n^2)$
Good for **dense** graphs, i.e., $m = O(n^2)$

Ex: Consider the following adjacency list representation

Adjacency list representation



1. Draw the corresponding graph.



2. Write the adjacency matrix representation

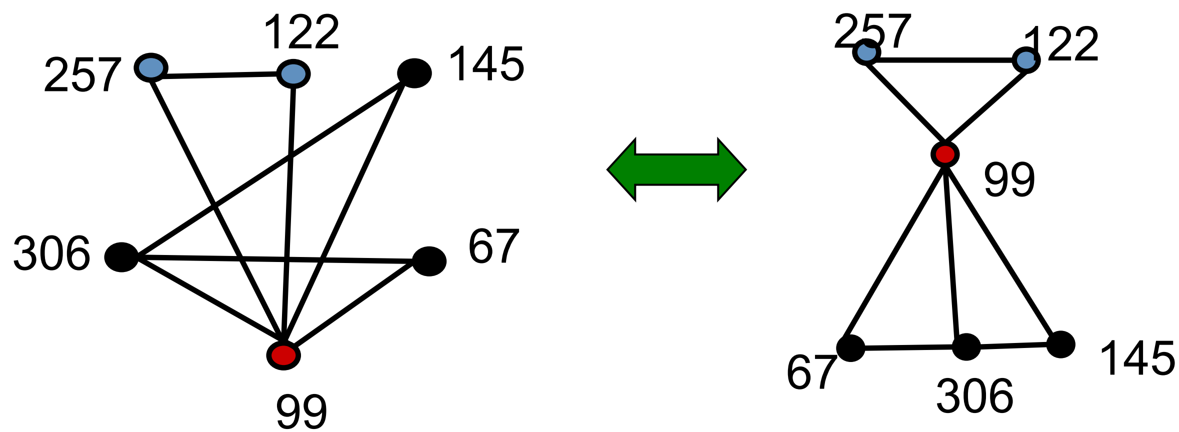
	1	2	3	4	5	6
1	0	**1**	0	**1**	0	0
2	0	0	**1**	0	0	0
3	0	**1**	0	0	0	**1**
4	**1**	0	0	0	**1**	0
5	0	0	**1**	**1**	0	**1**
6	0	0	0	0	0	0

Graph isomorphism

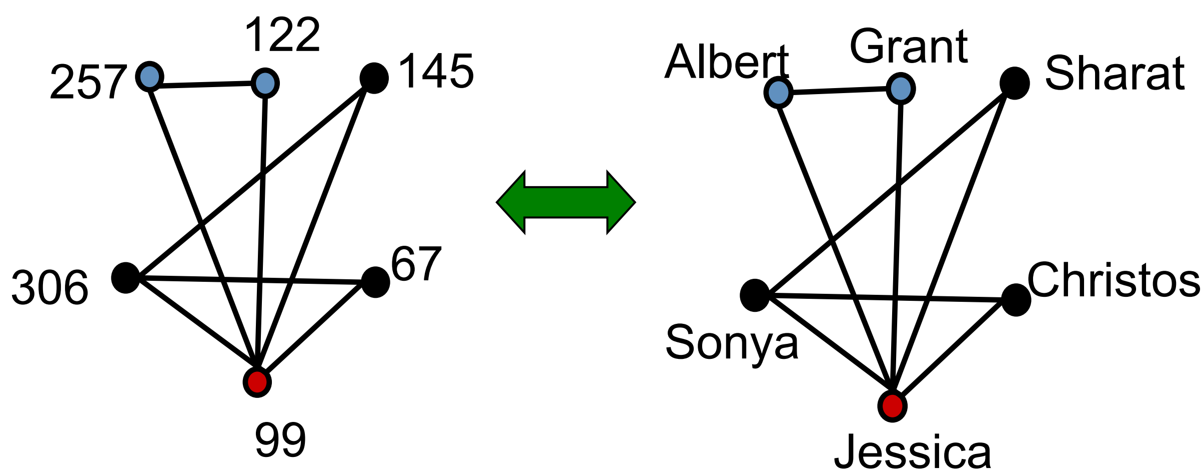
A graph G_1 is **isomorphic** to graph G_2 if there is an edge-preserving vertex matching.

- Graphs are the same, but may be drawn differently or labeled differently

Same graph
(different *drawings*)



Same graph
(different *labels*)



Breadth-first search (BFS)

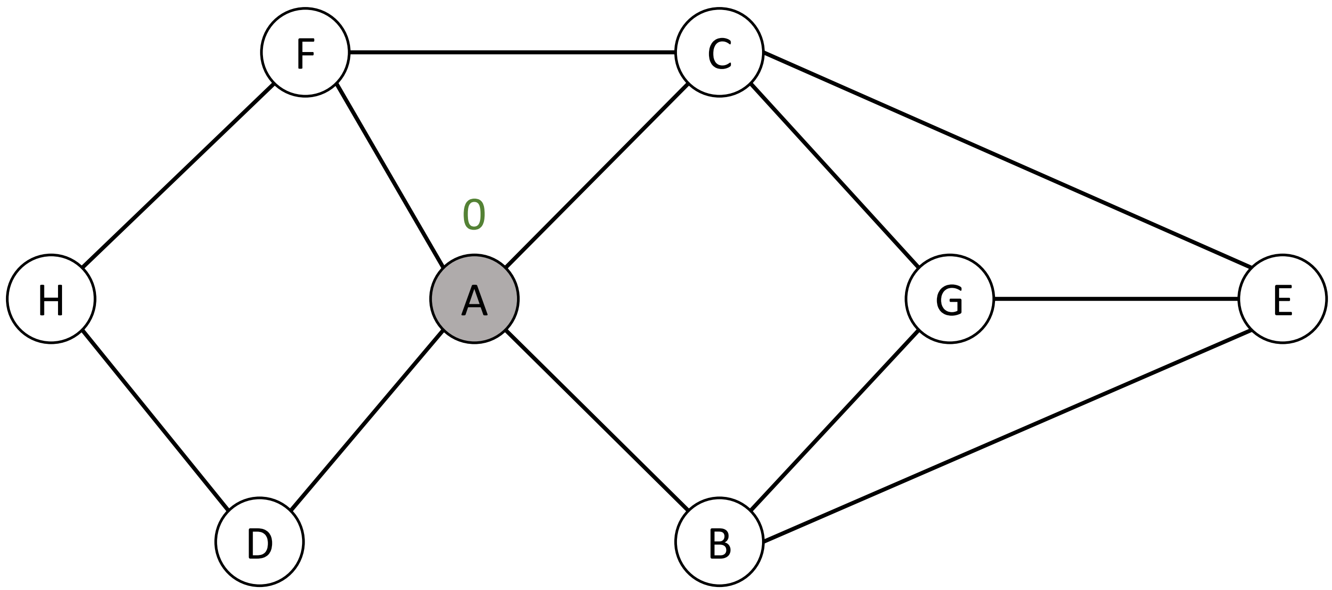
- Breadth-first search (BFS) is a general technique for traversing a graph.
  - visits all vertices and edges
  - on a graph with n vertices and m edges takes $O(n + m)$ time
  - Produces a breadth-first tree consisting of vertices reachable from the starting point

- BFS can be further extended to solve other graph problems
  - determine whether G is connected
  - compute the connected components of G
  - compute a spanning forest of G
  - find and report a path with the minimum number of edges between two given vertices
  - find a simple cycle, if there is one

Breadth-first search (BFS)

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each vertex v in G.Adj[u]     // search the neighbors of u
        if v.color == WHITE           // is v being discovered now?
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)             // v is now on the frontier
    u.color = BLACK                   // u is now behind the frontier
```

Ex: BFS on a graph

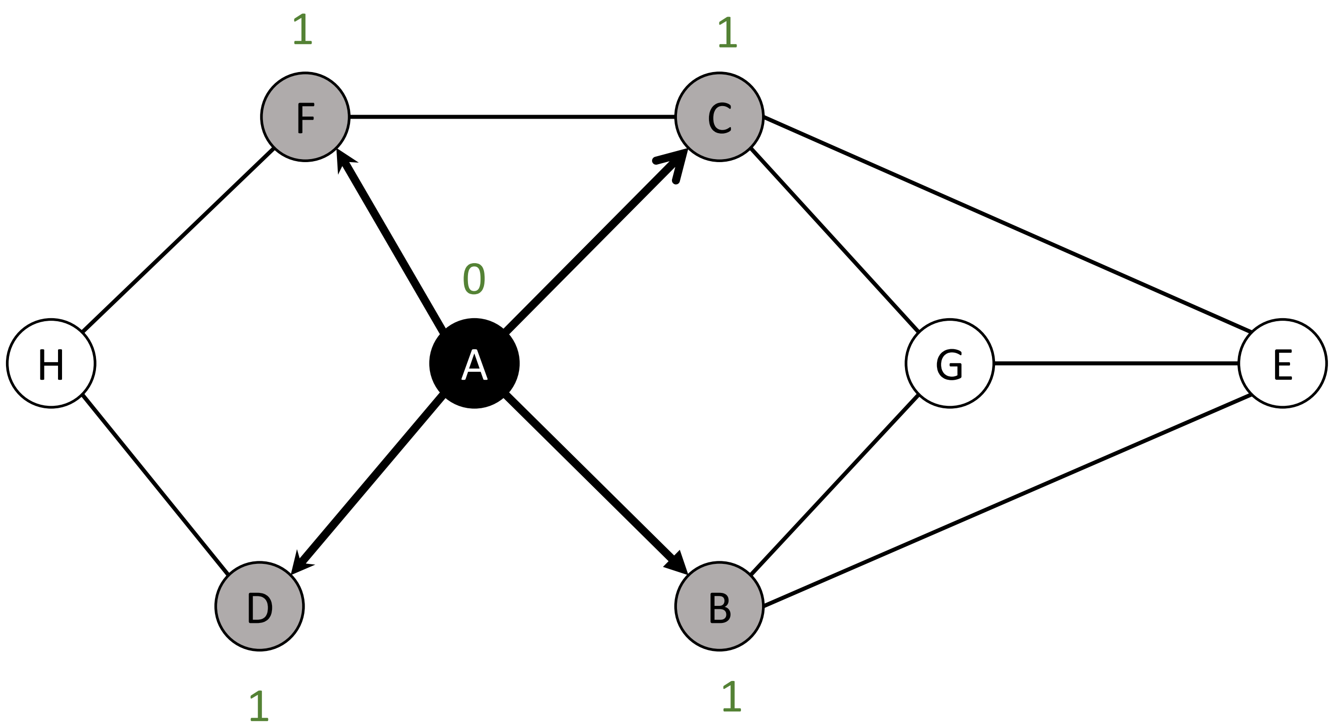


Queue: A

distance d

u → v indicates that $v.\pi = u$

Ex: BFS on a graph

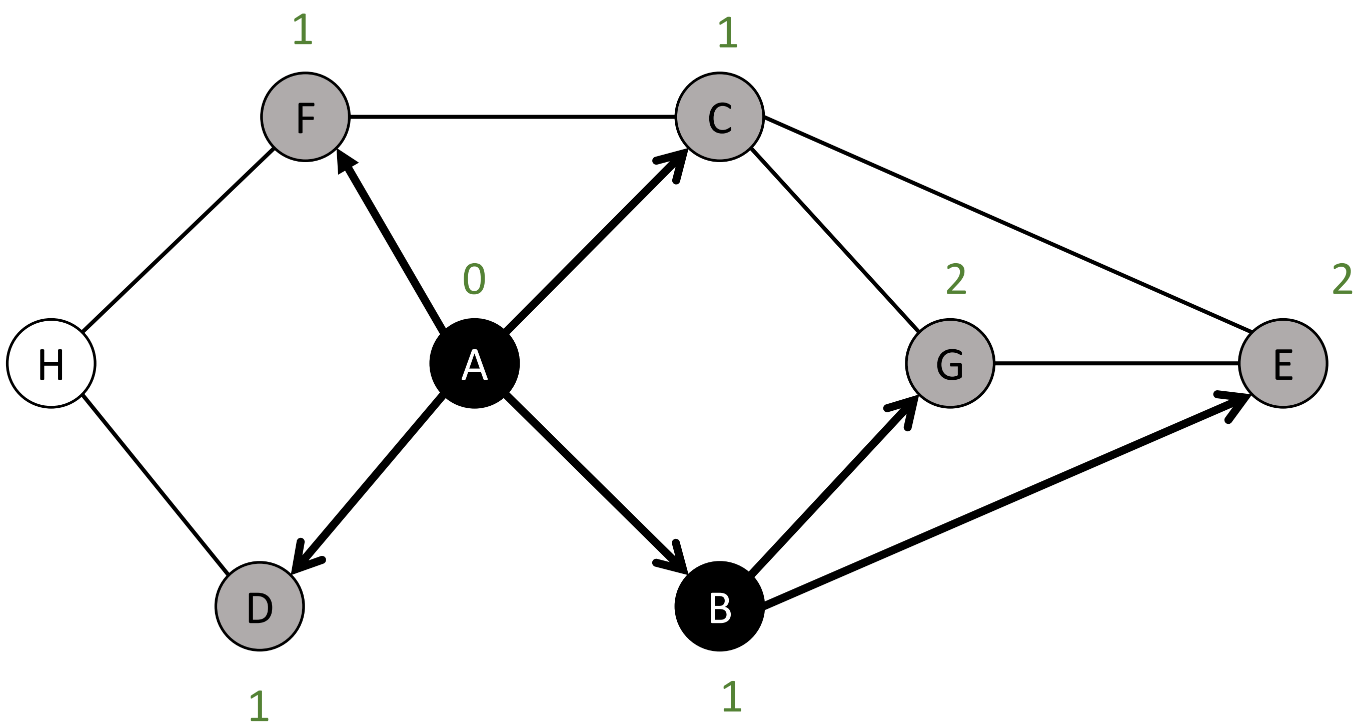


Queue: A, B, C, D, F

distance d

u → v indicates that $v.\pi = u$

Ex: BFS on a graph

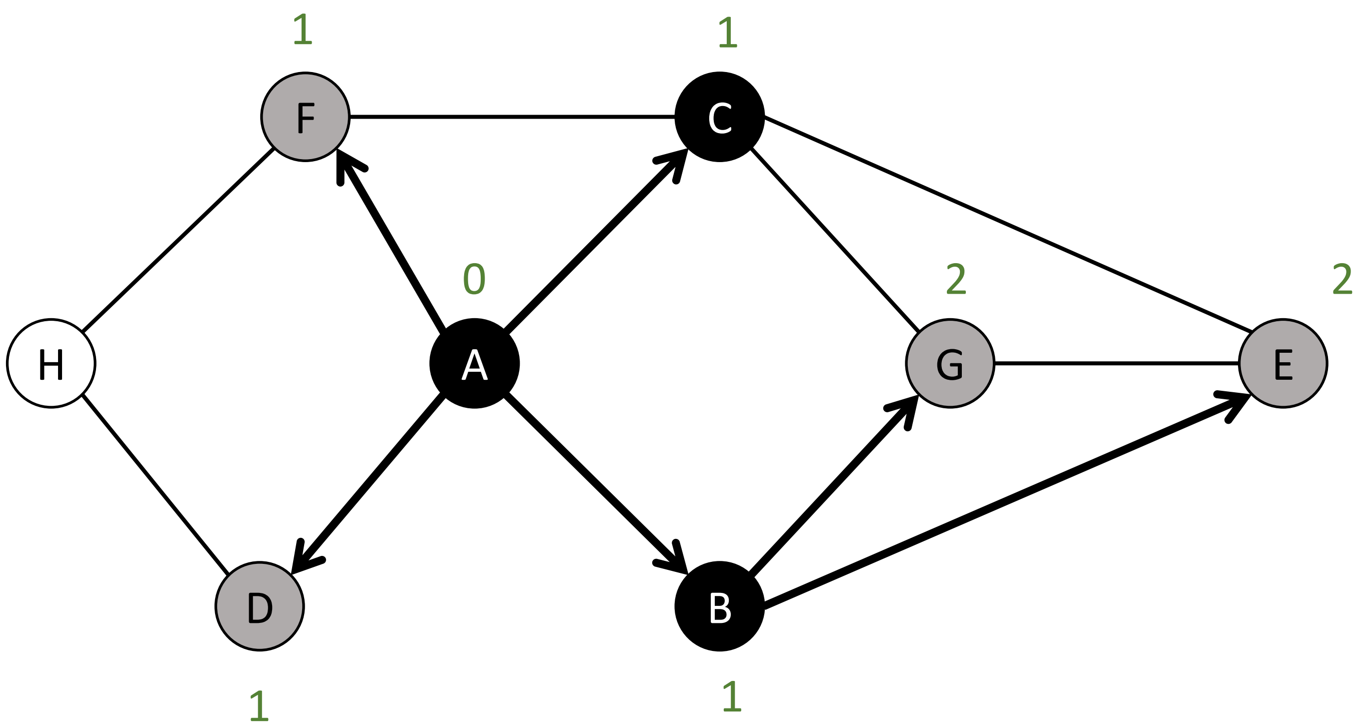


Queue: ~~A~~, ~~B~~, C, D, F, G, E

distance d

u → v indicates that $v.\pi = u$

Ex: BFS on a graph

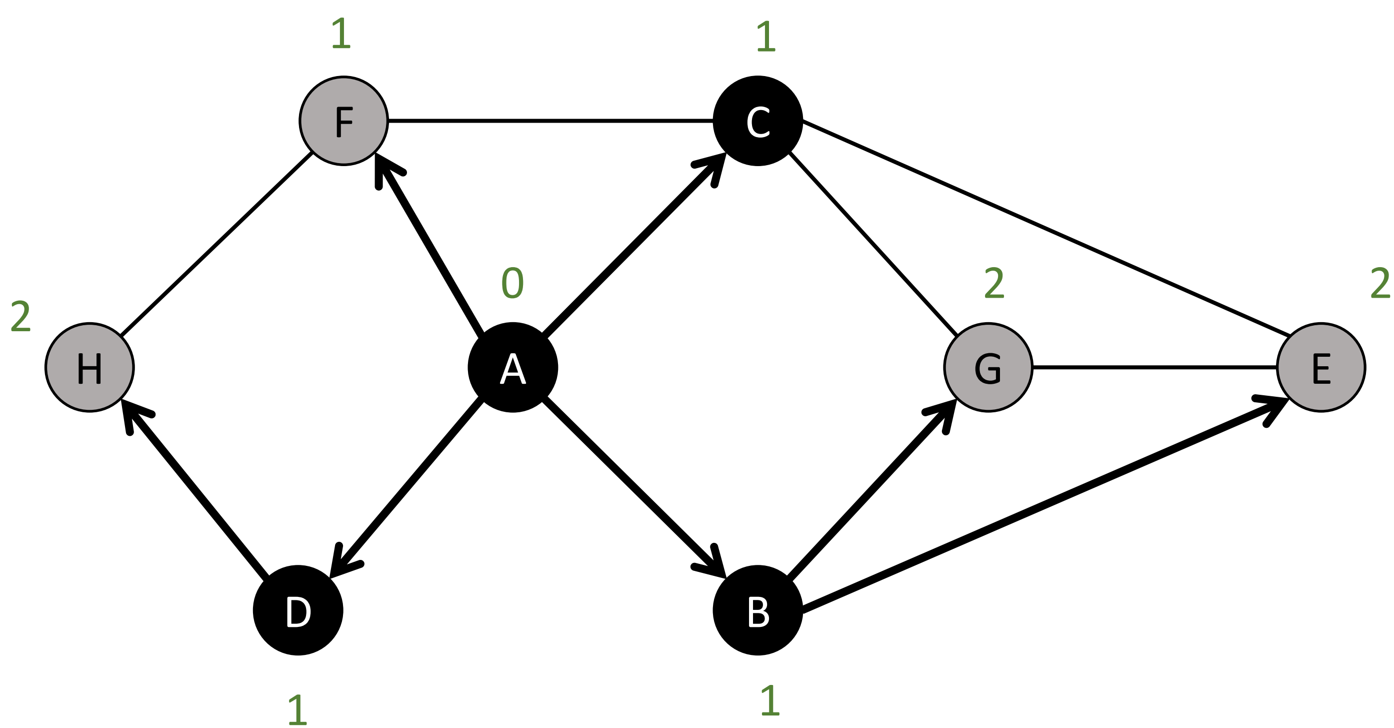


Queue: A, B, C, D, F, G, E

distance d

u → v indicates that $v.\pi = u$

Ex: BFS on a graph

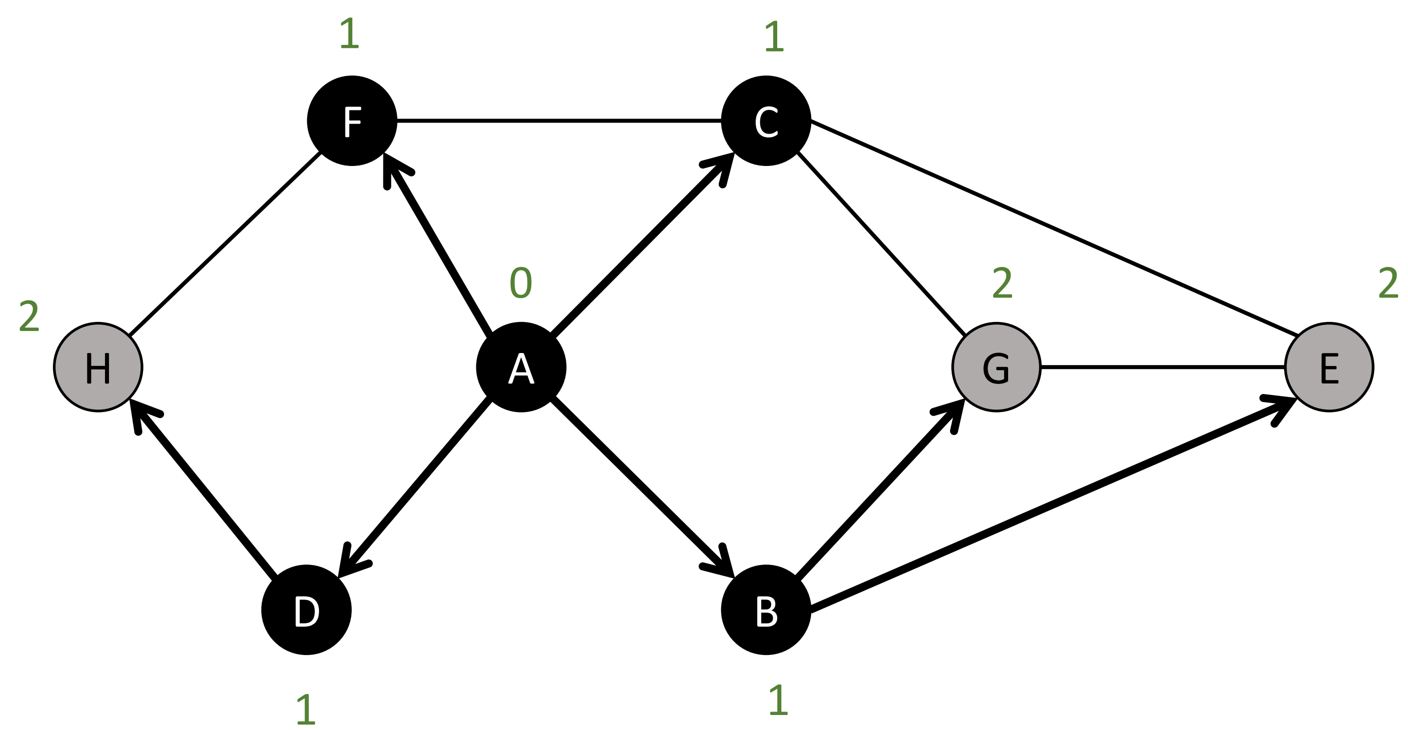


Queue: ~~A~~, ~~B~~, ~~C~~, ~~D~~, F, G, E, H

distance d

u → v indicates that $v.\pi = u$

Ex: BFS on a graph

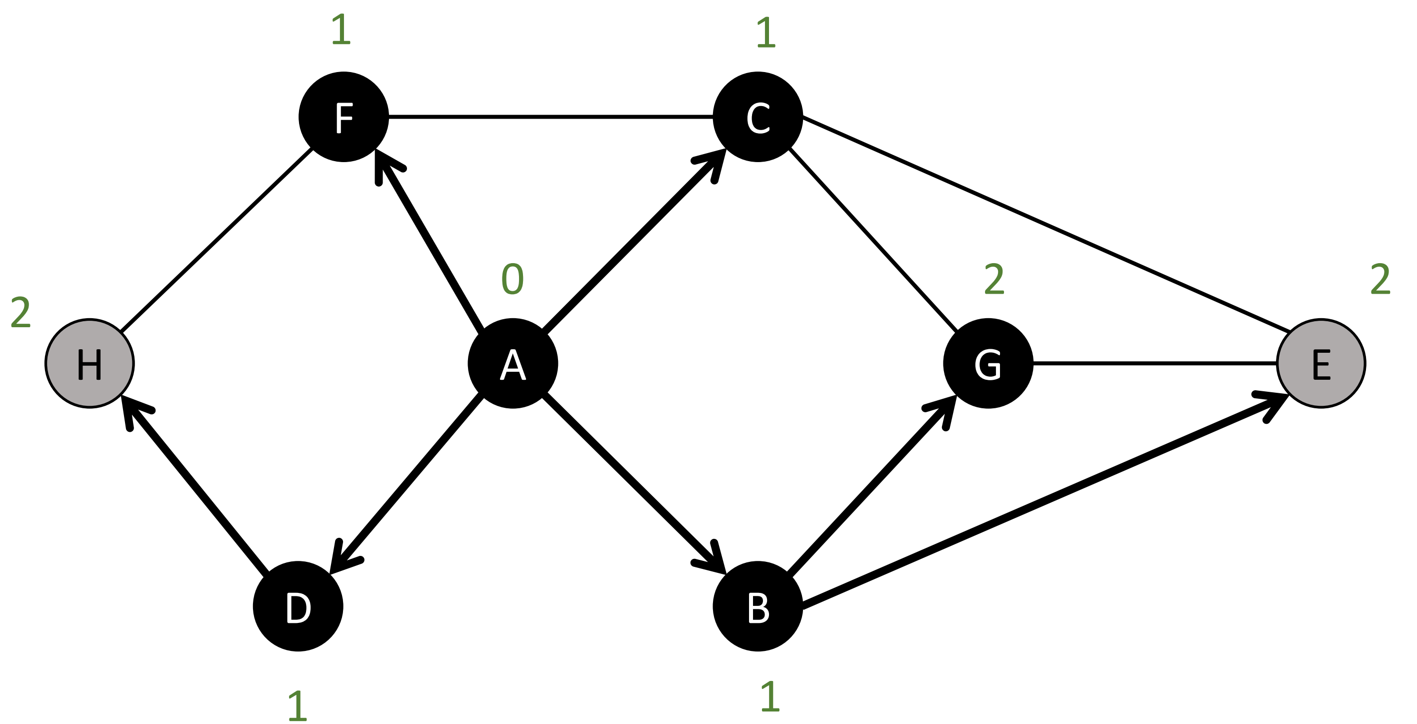


Queue: A, B, C, D, F, G, E, H

distance d

u → v indicates that $v.\pi = u$

Ex: BFS on a graph

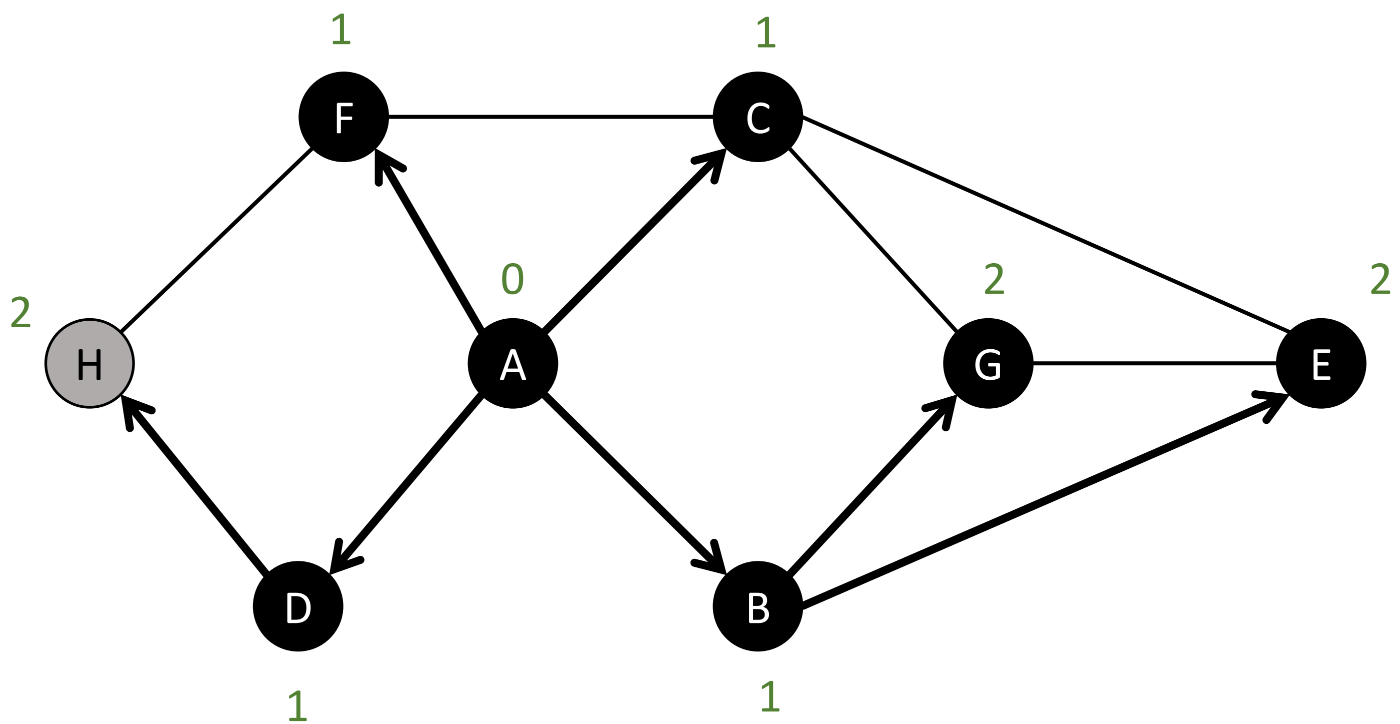


Queue: ~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~F~~, ~~G~~, E, H

distance d

u → v indicates that $v.\pi = u$

Ex: BFS on a graph

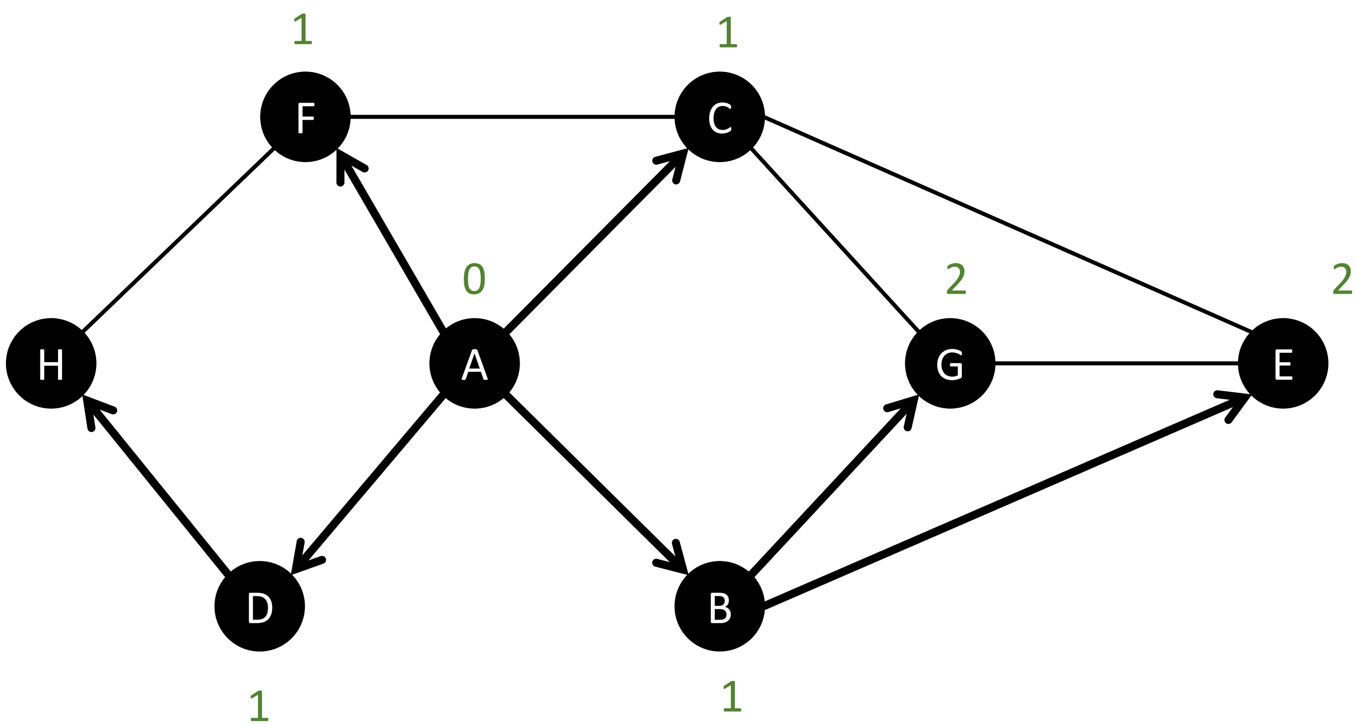


Queue: ~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~F~~, ~~G~~, ~~E~~, H

distance d

u → v indicates that $v.\pi = u$

Ex: BFS on a graph



Queue: ~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~F~~, ~~G~~, ~~E~~, ~~H~~

distance d

u → v indicates that $v.\pi = u$

Depth-first search (DFS)

- Depth-first search (DFS) is a general technique for traversing a graph.
  - visits all vertices and edges
  - on a graph with n vertices and m edges takes $O(n + m)$ time
  - Produces a depth-first tree consisting of vertices reachable from the starting point

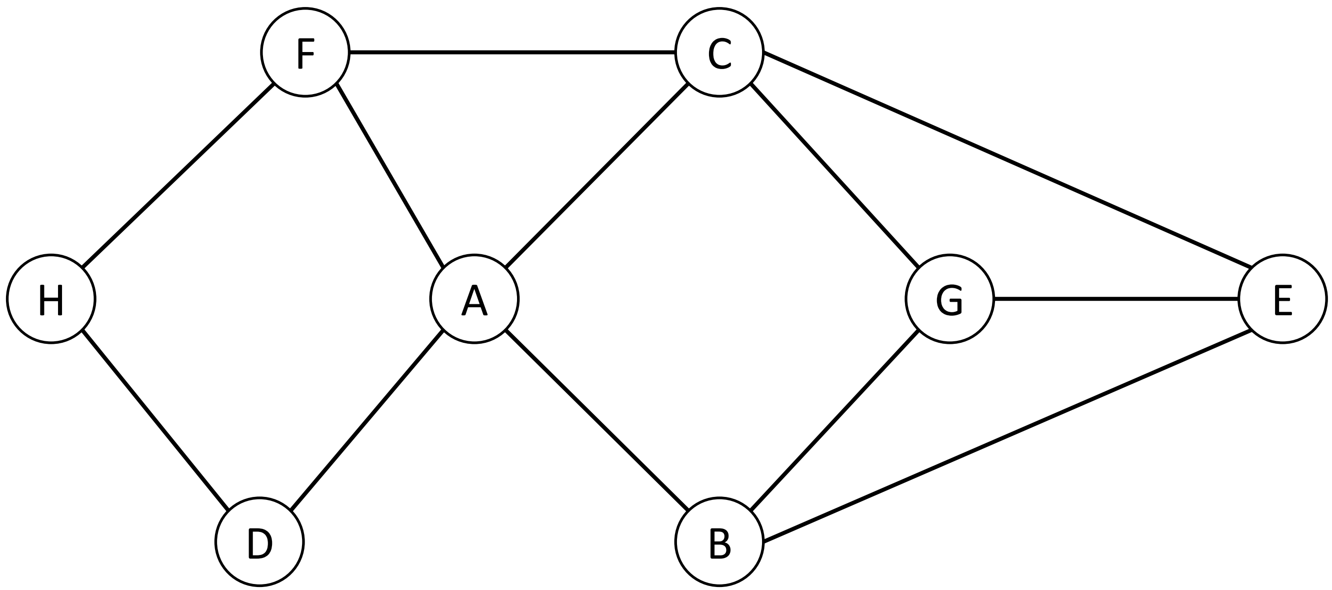
- DFS can be further extended to solve other graph problems
  - determine whether G is connected
  - compute the connected components of G
  - compute a spanning forest of G
  - find and report a path between two given vertices
  - find a simple cycle, if there is one

Depth-first search (DFS)

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1                     // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each vertex v in G.Adj[u]       // explore each edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
time = time + 1
u.f = time
u.color = BLACK                     // blacken u; it is finished
```

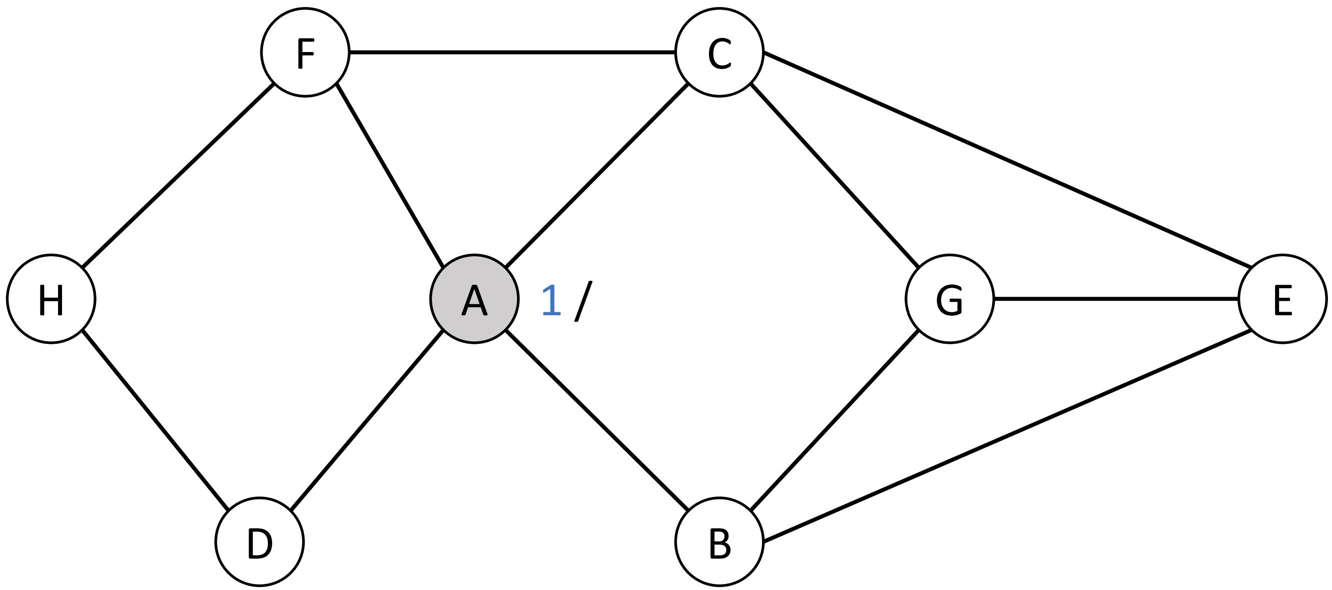
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

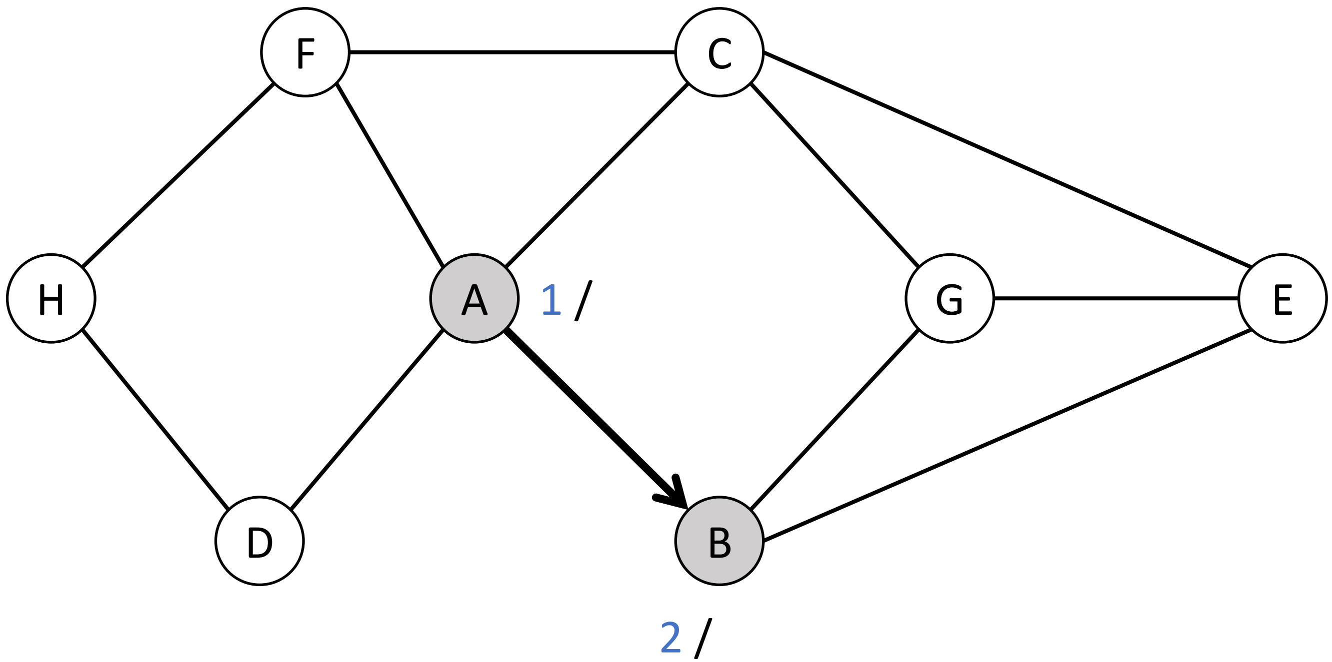
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

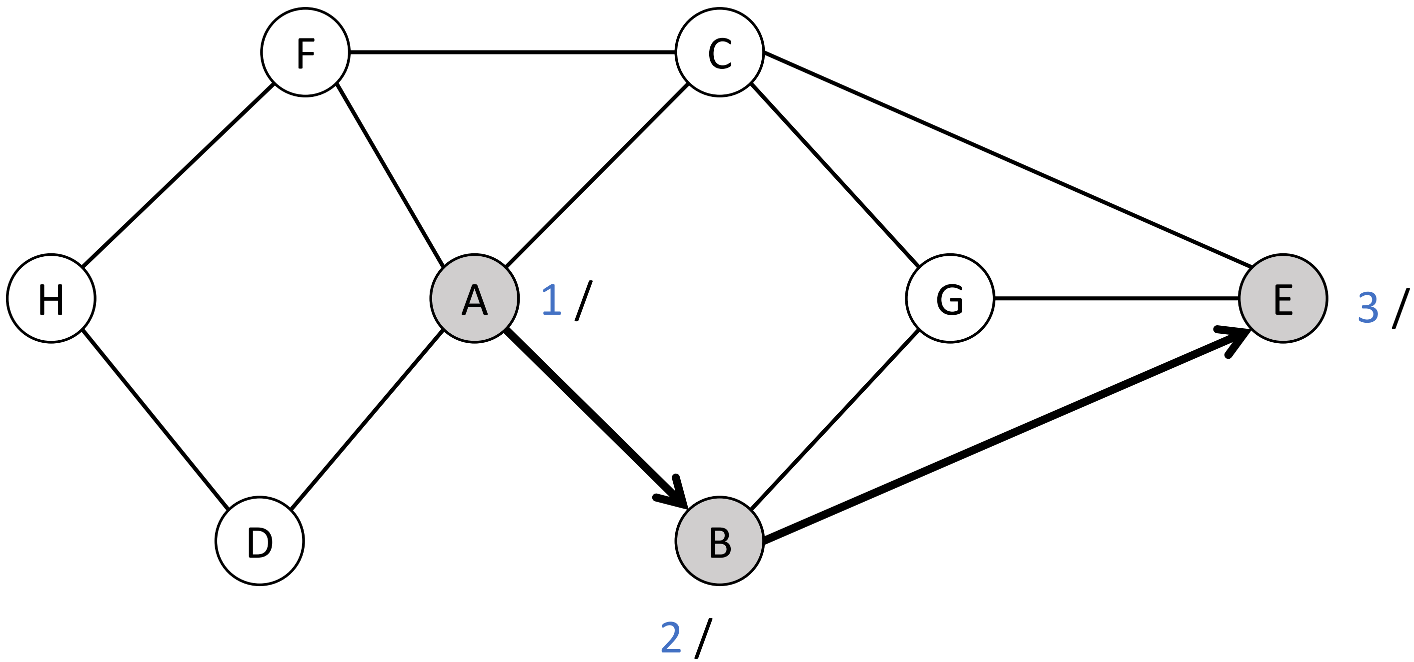
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

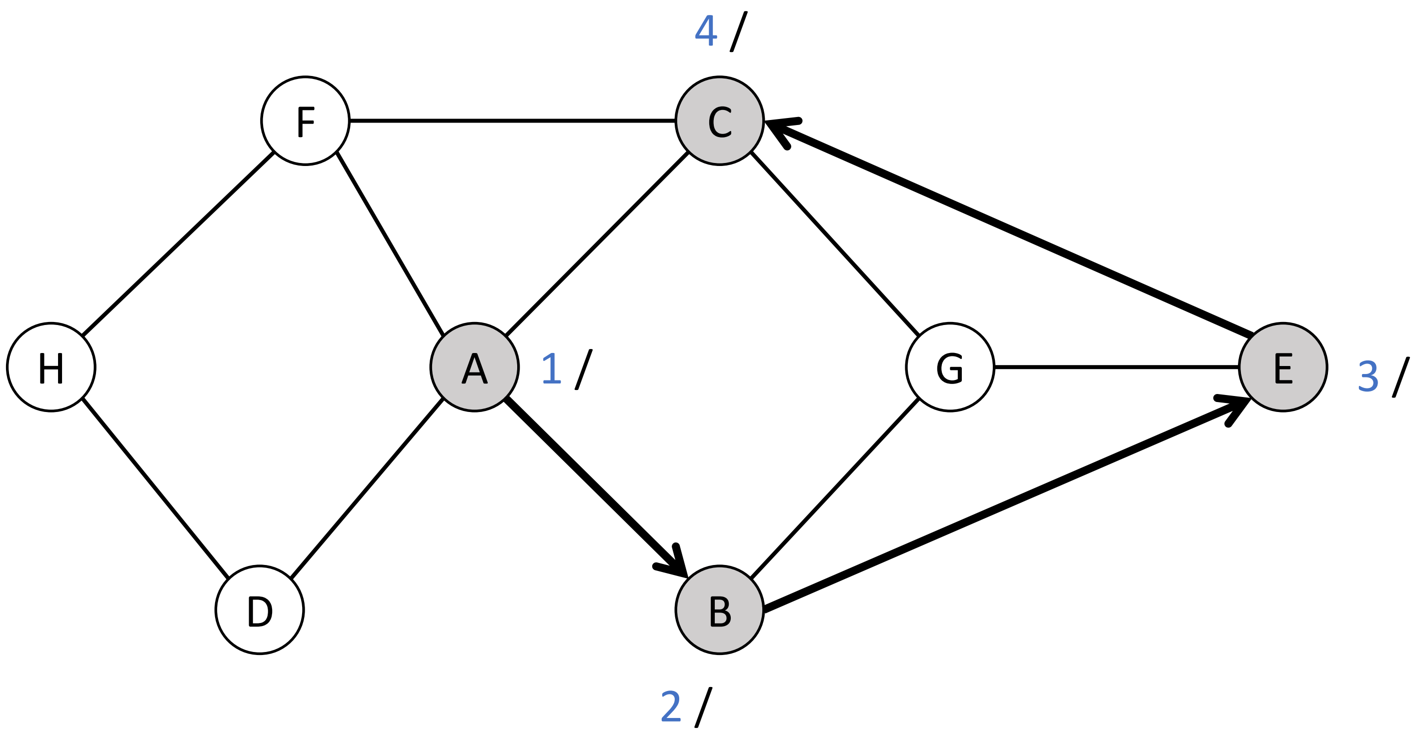
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

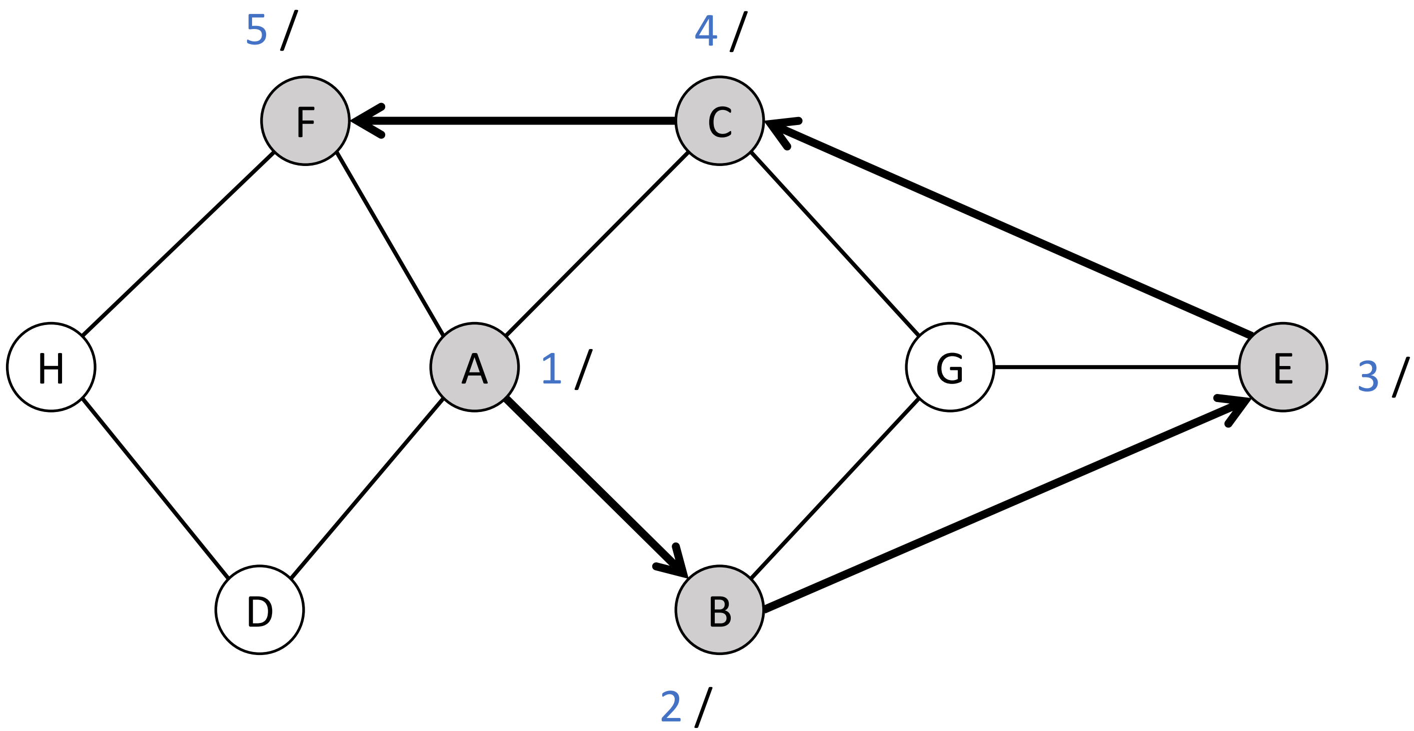
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

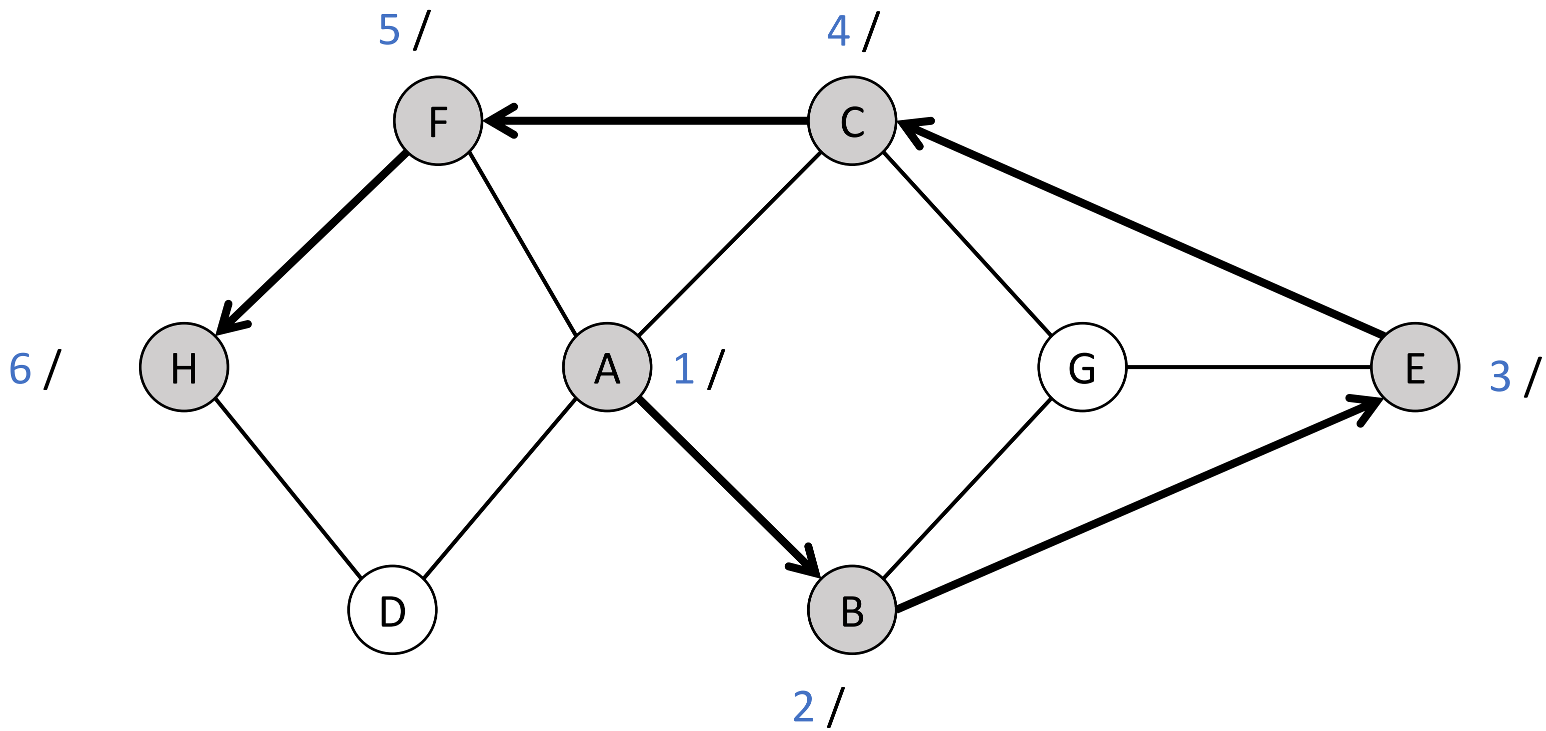
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

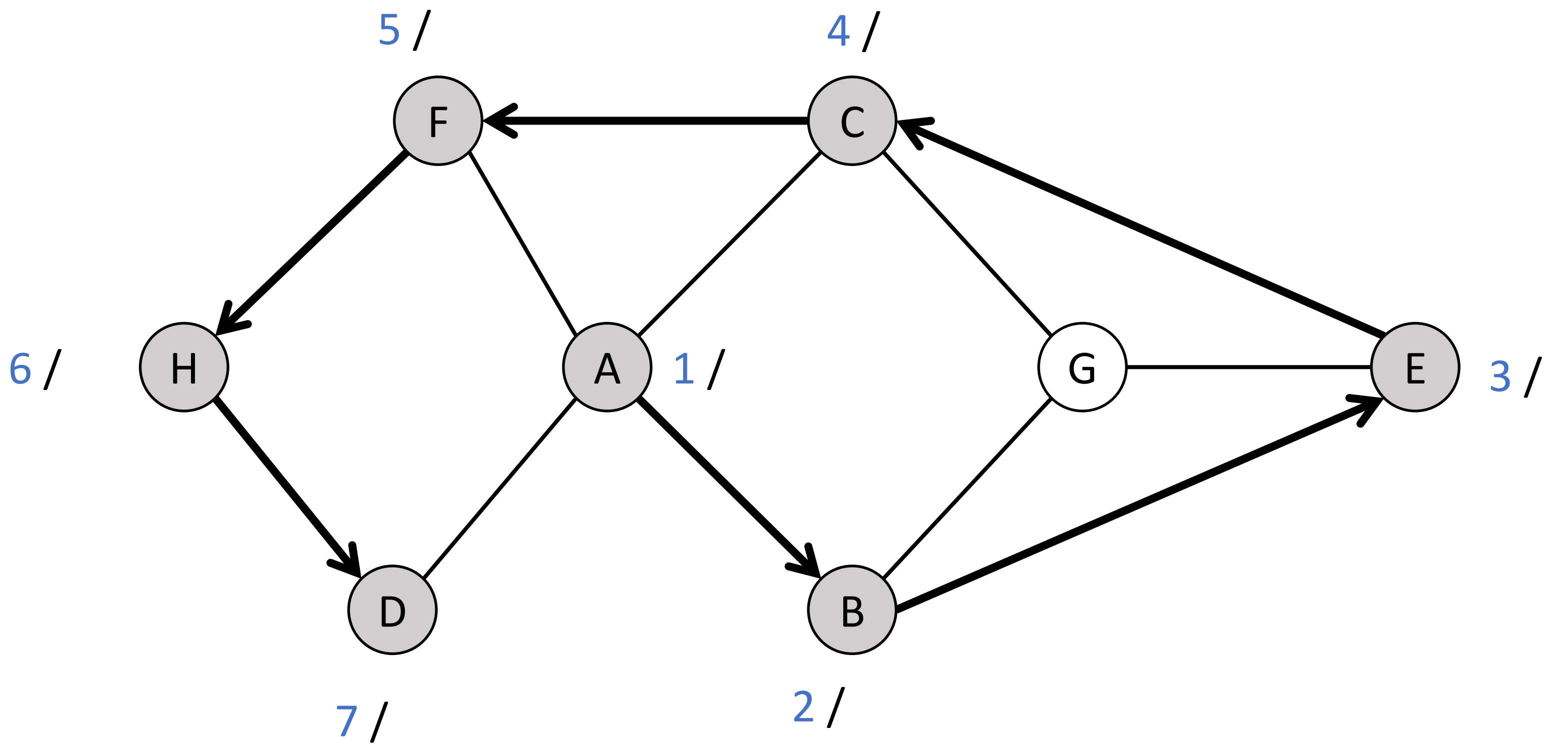
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

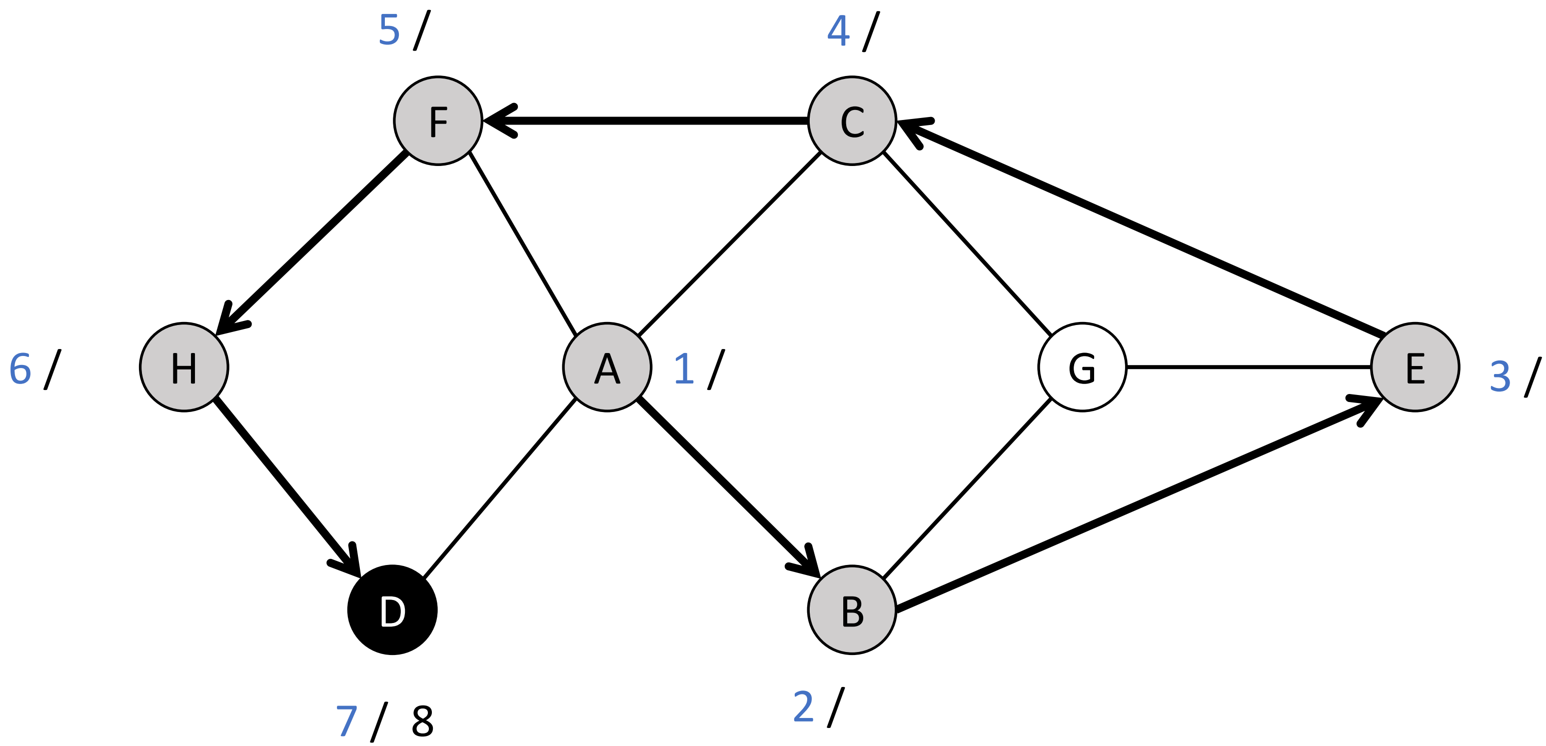
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

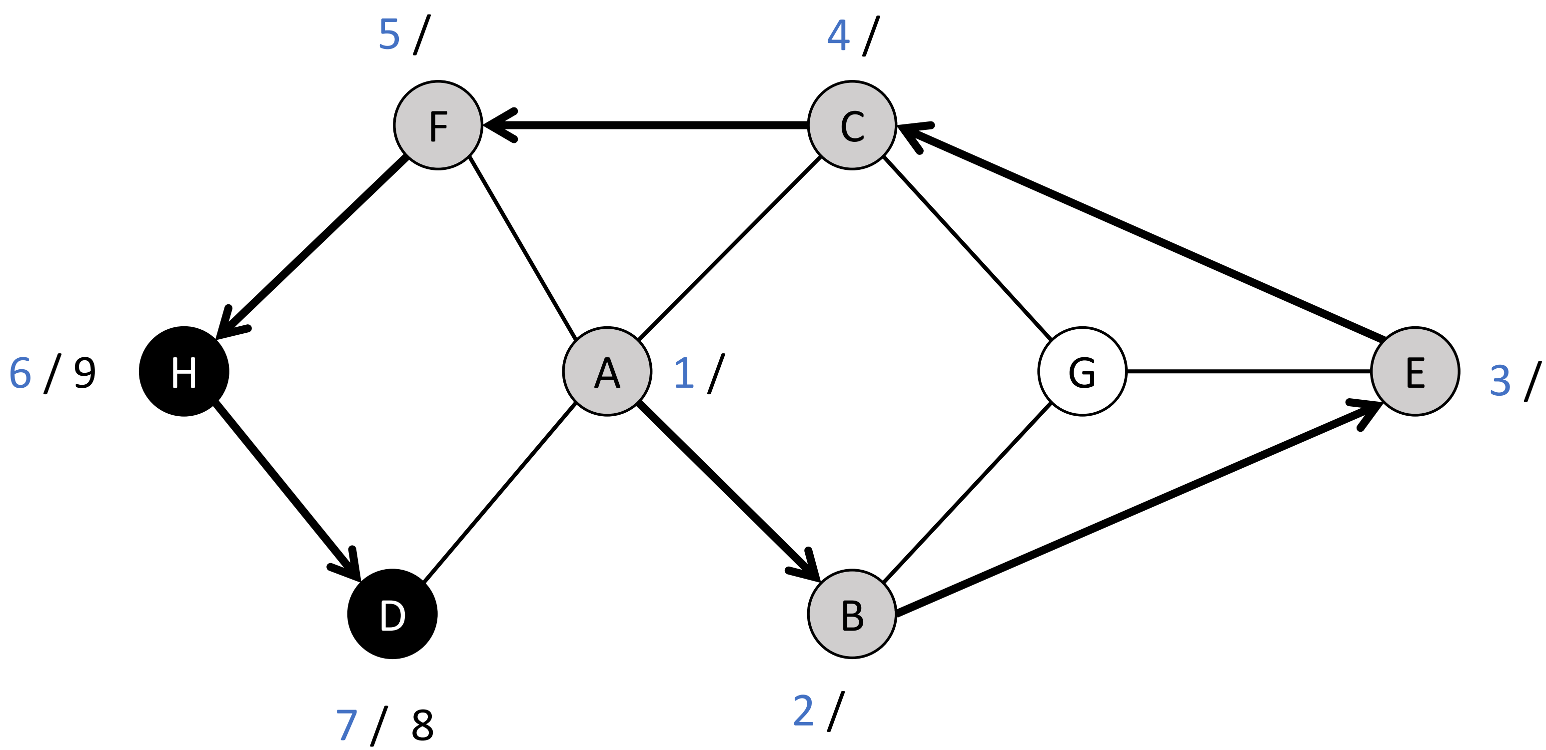
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

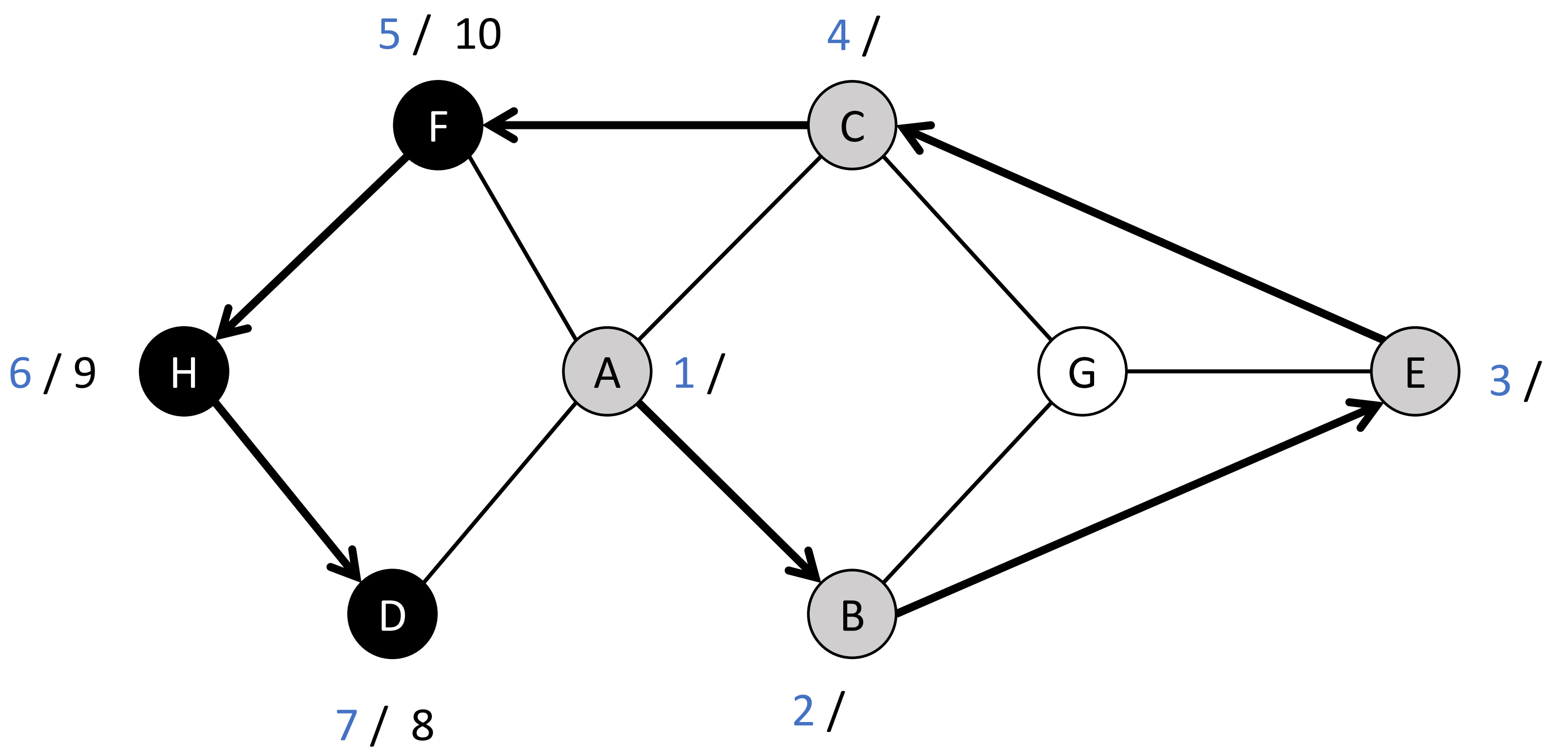
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

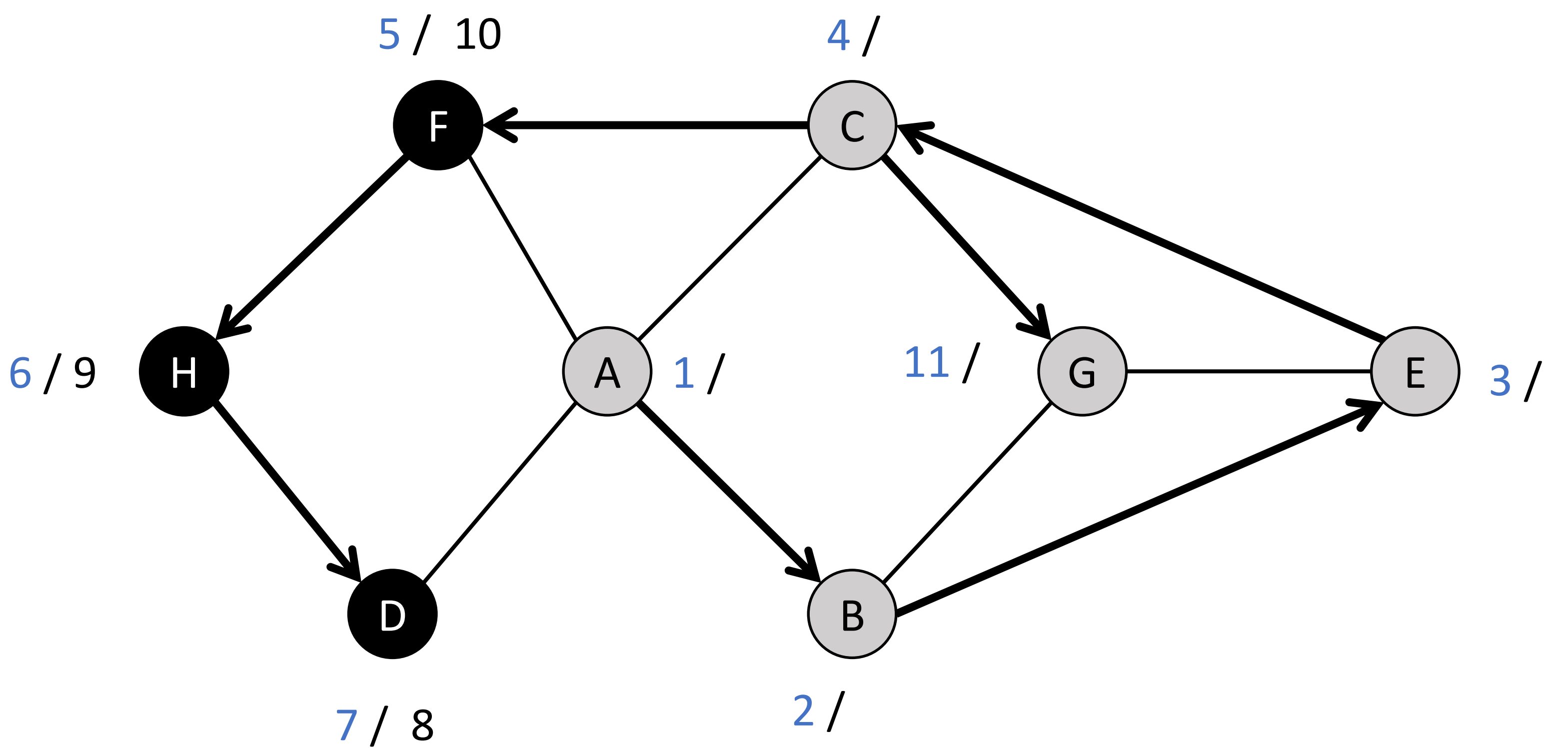
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

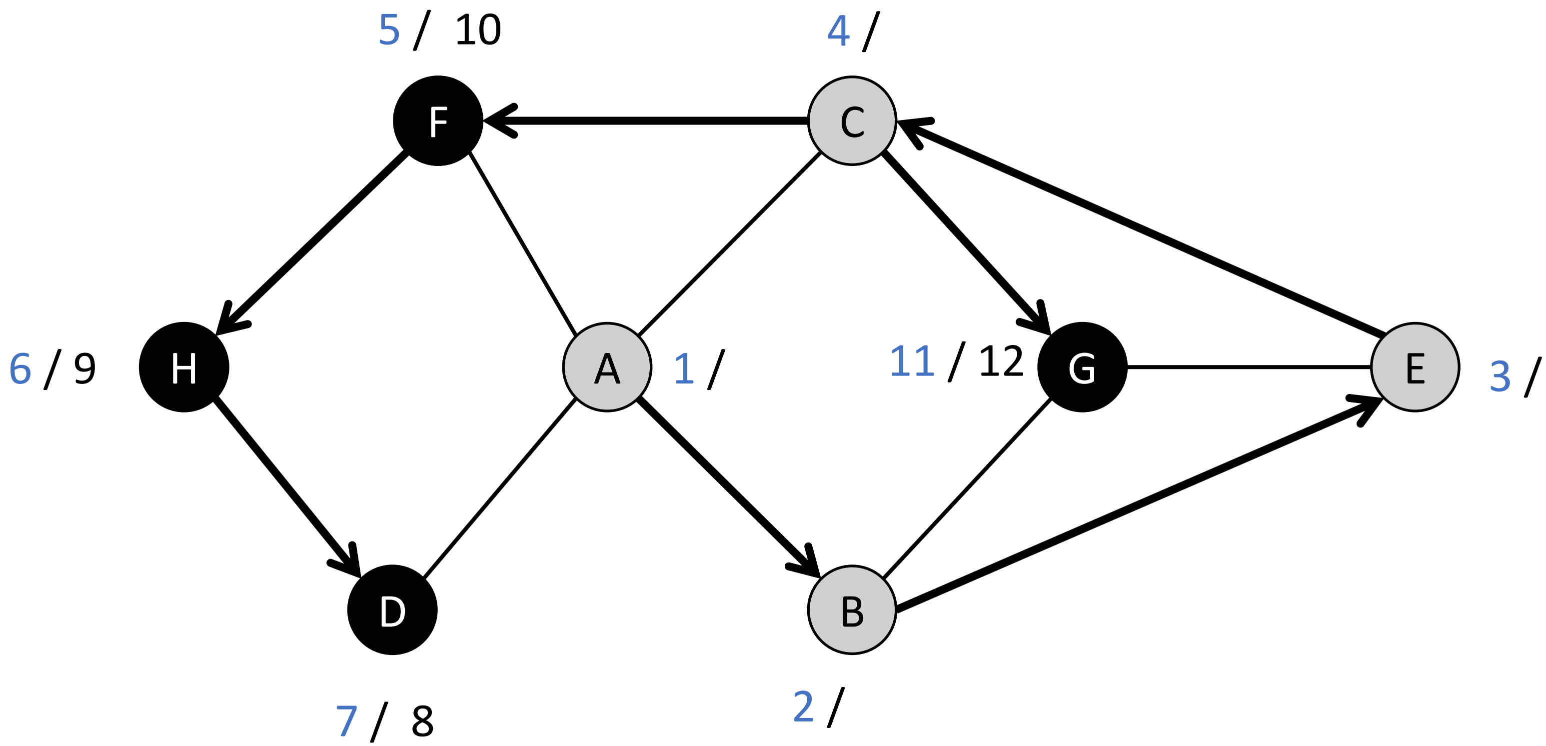
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

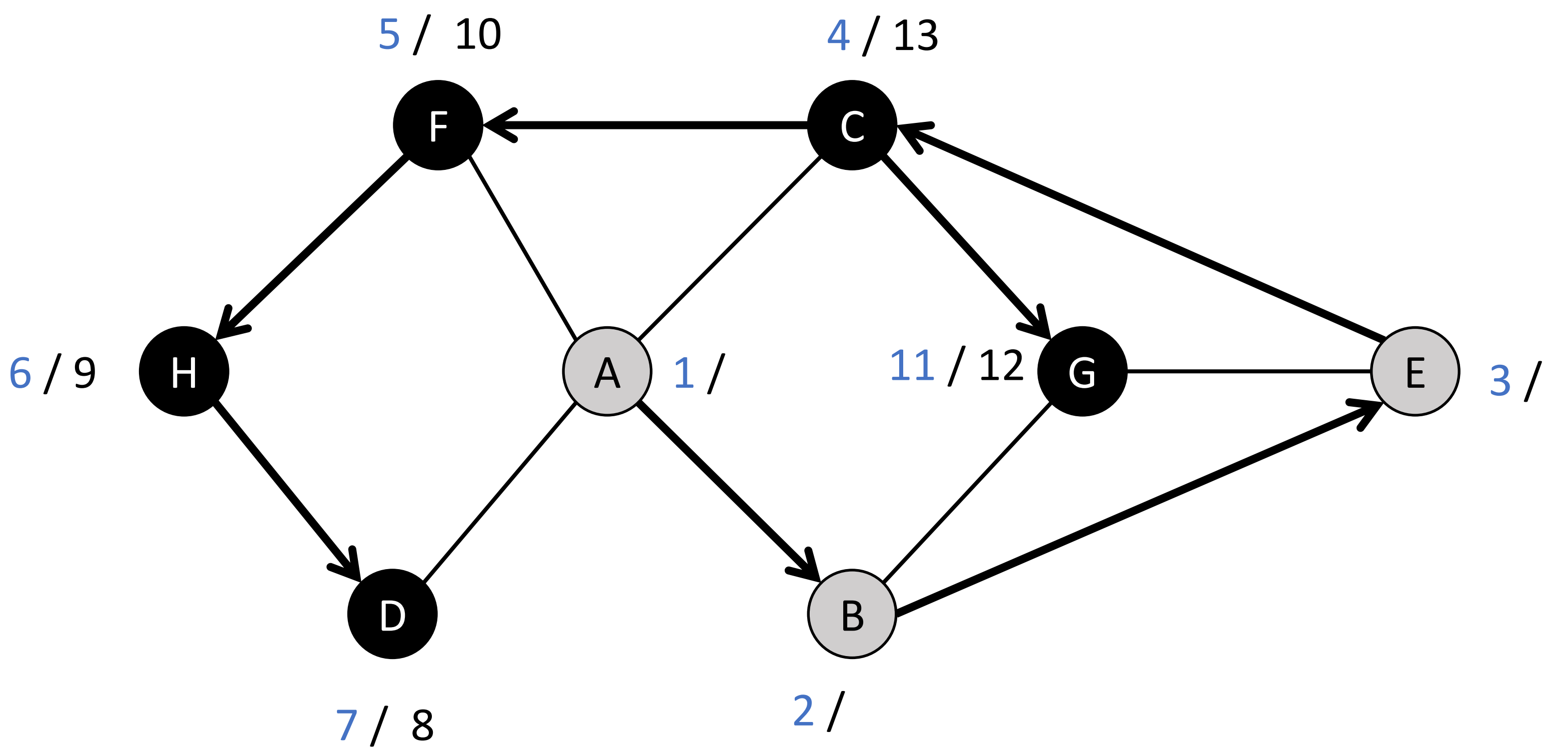
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

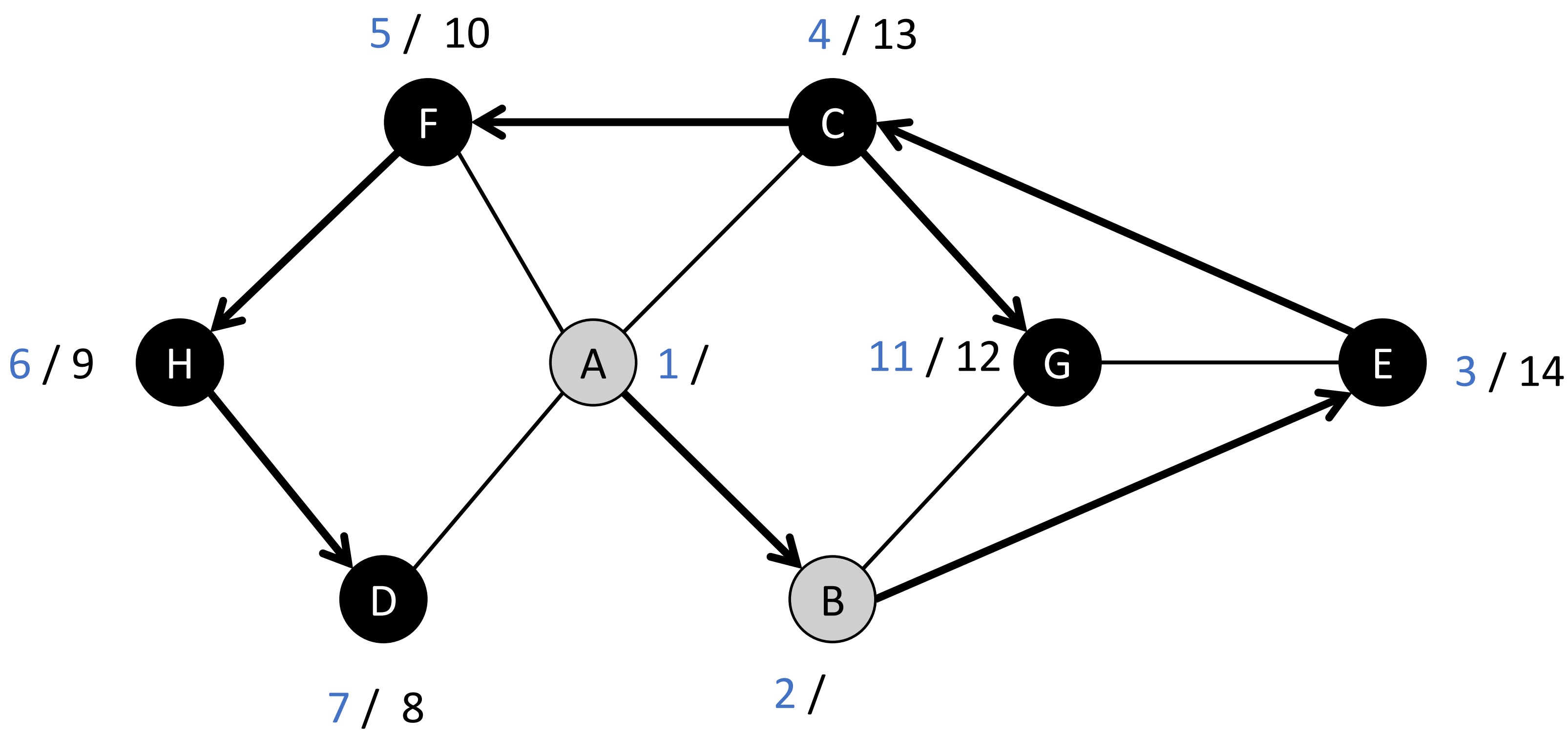
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

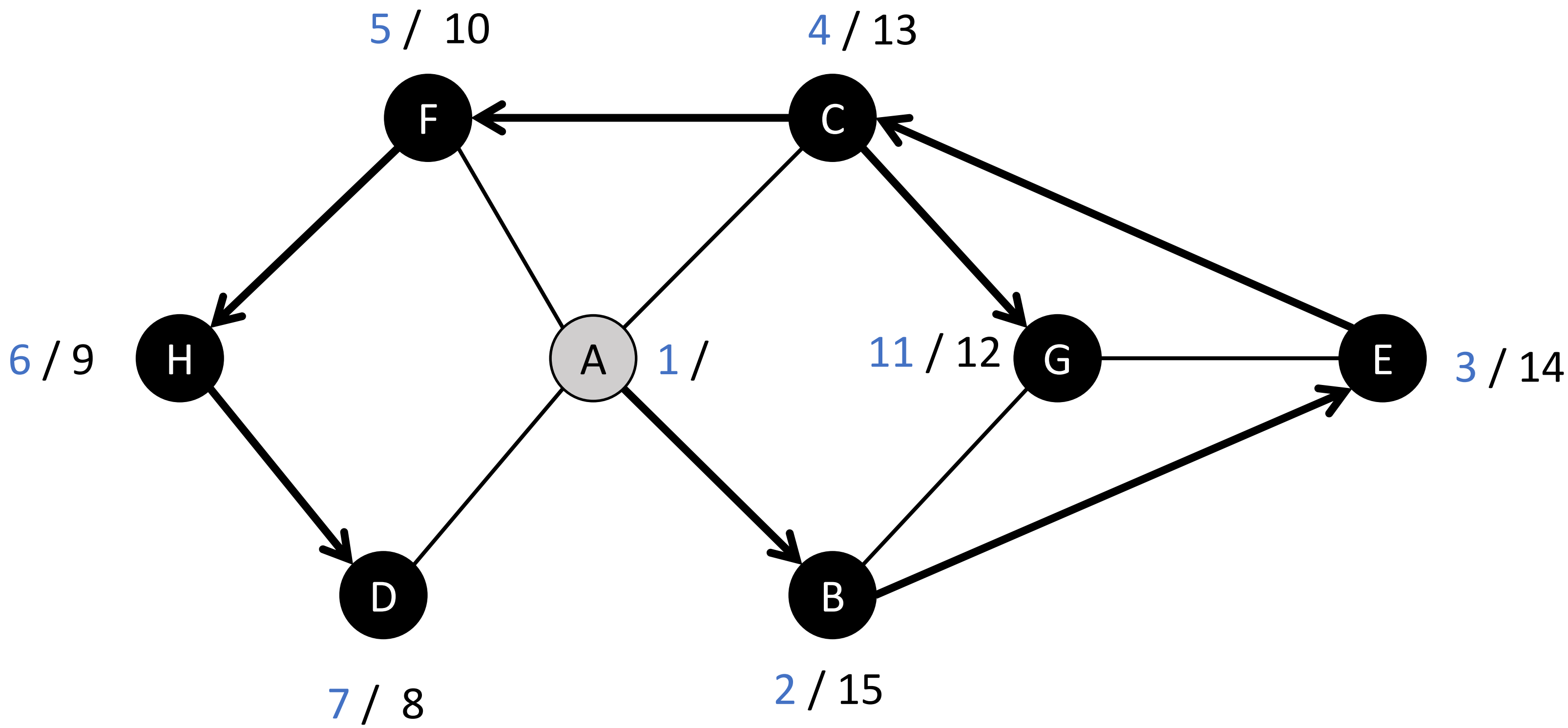
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

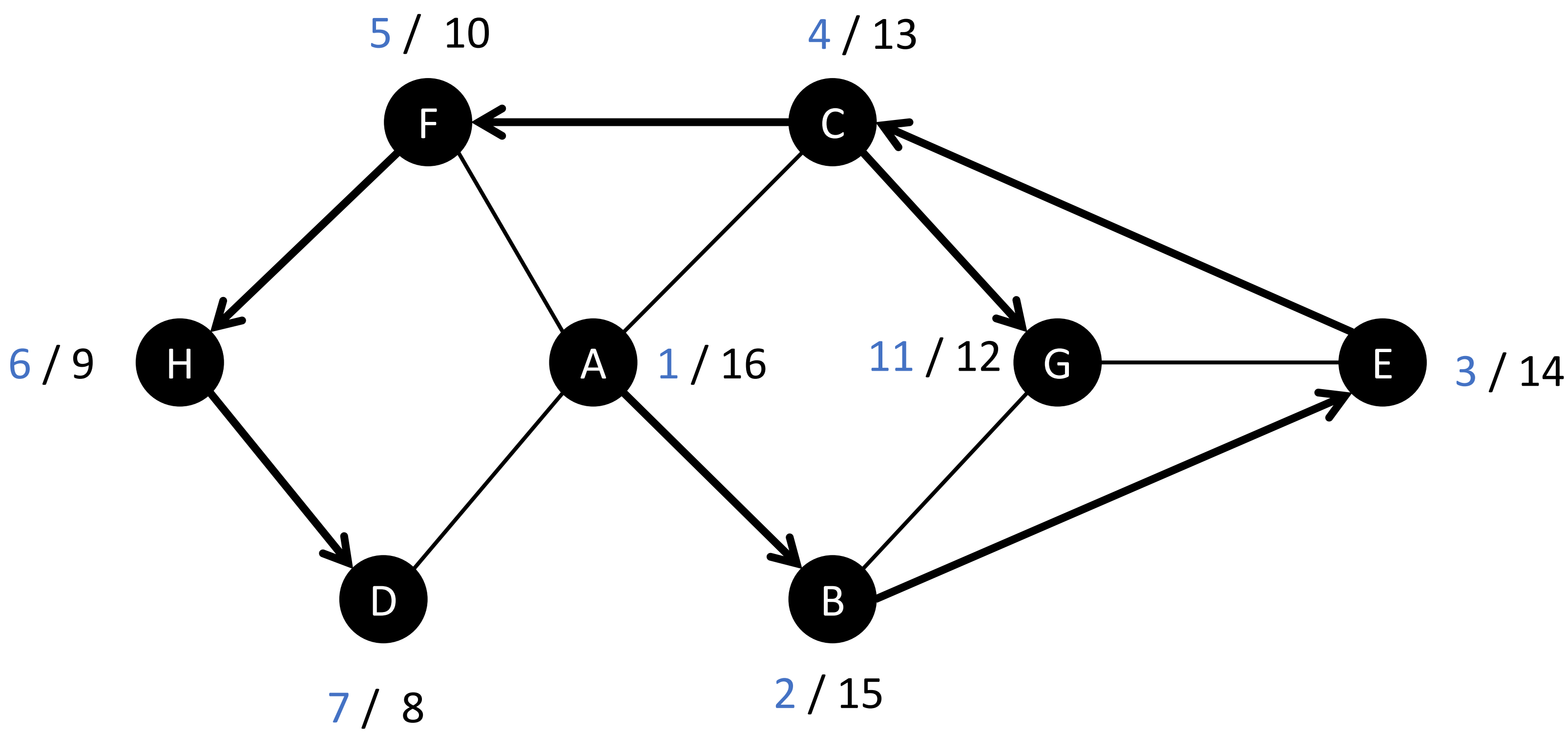
Ex: DFS on a graph



time in / time out
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

Ex: DFS on a graph



time in / time out

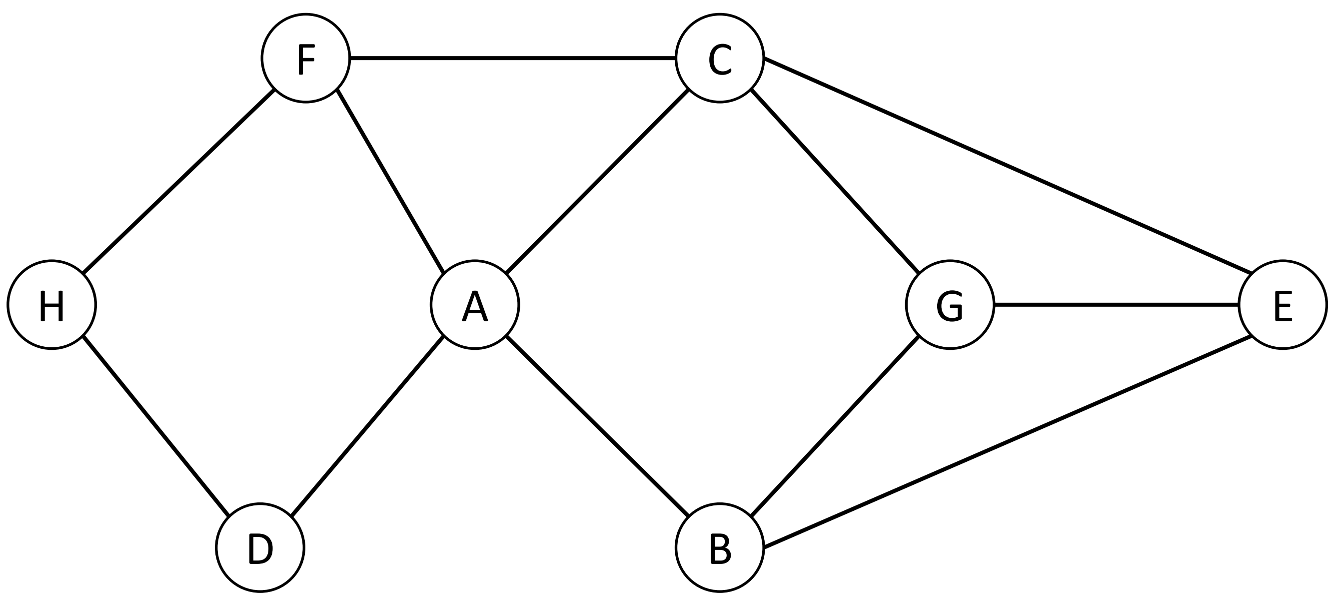
u → v indicates that $v.\pi = u$

When given a choice, for consistency we will pick vertices which occur earliest alphabetically

BFS / DFS Forest

Consider only those edges used to discovery a new vertex. We obtain a tree which spans each connected component of the original graph.

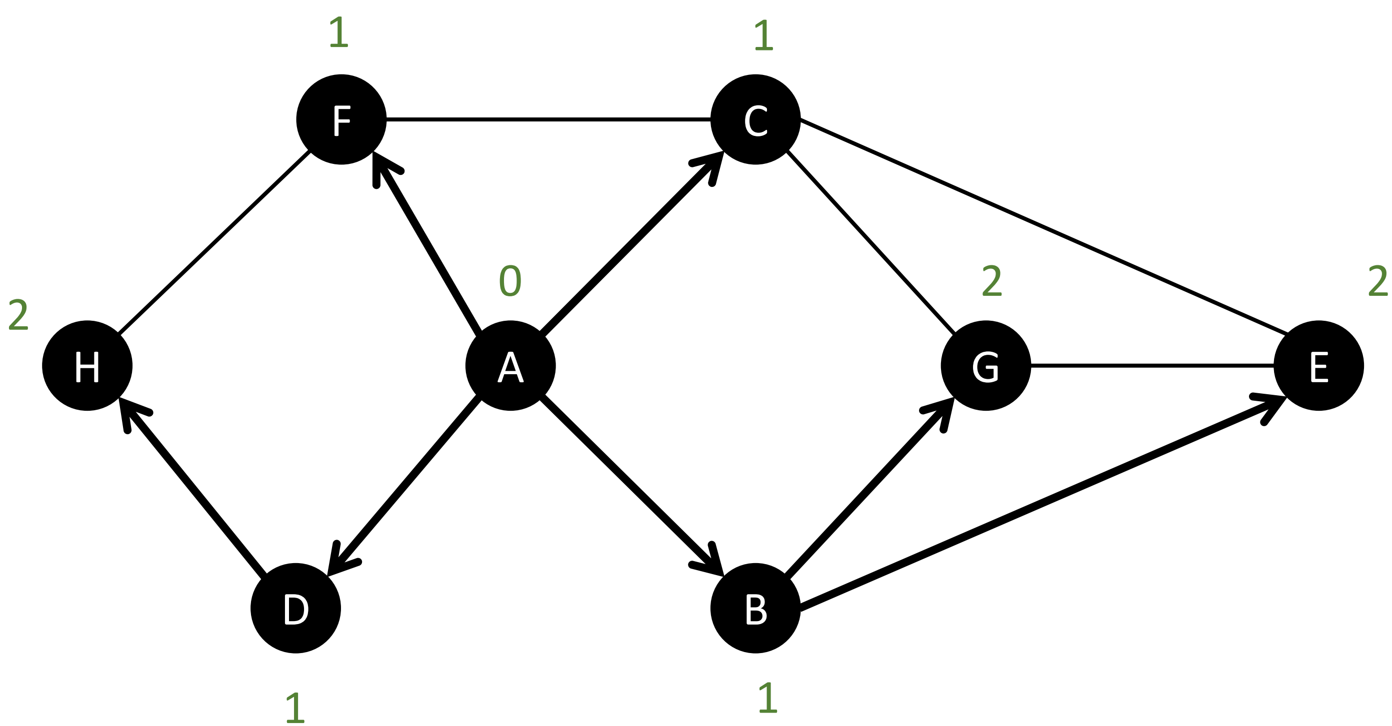
Original Graph:



BFS / DFS Forest

Consider only those edges used to discovery a new vertex. We obtain a tree which spans each connected component of the original graph.

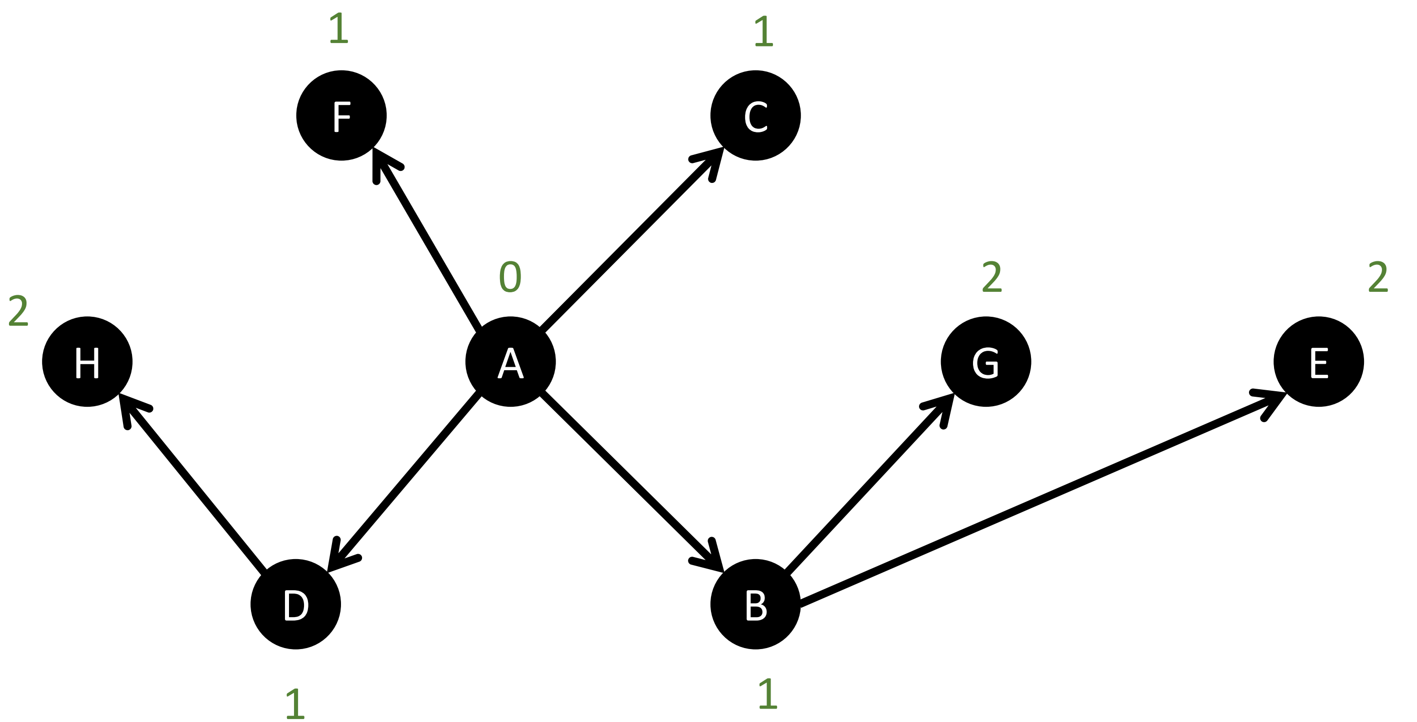
Graph after BFS traversal:



BFS / DFS Forest

Consider only those edges used to discovery a new vertex. We obtain a tree which spans each connected component of the original graph.

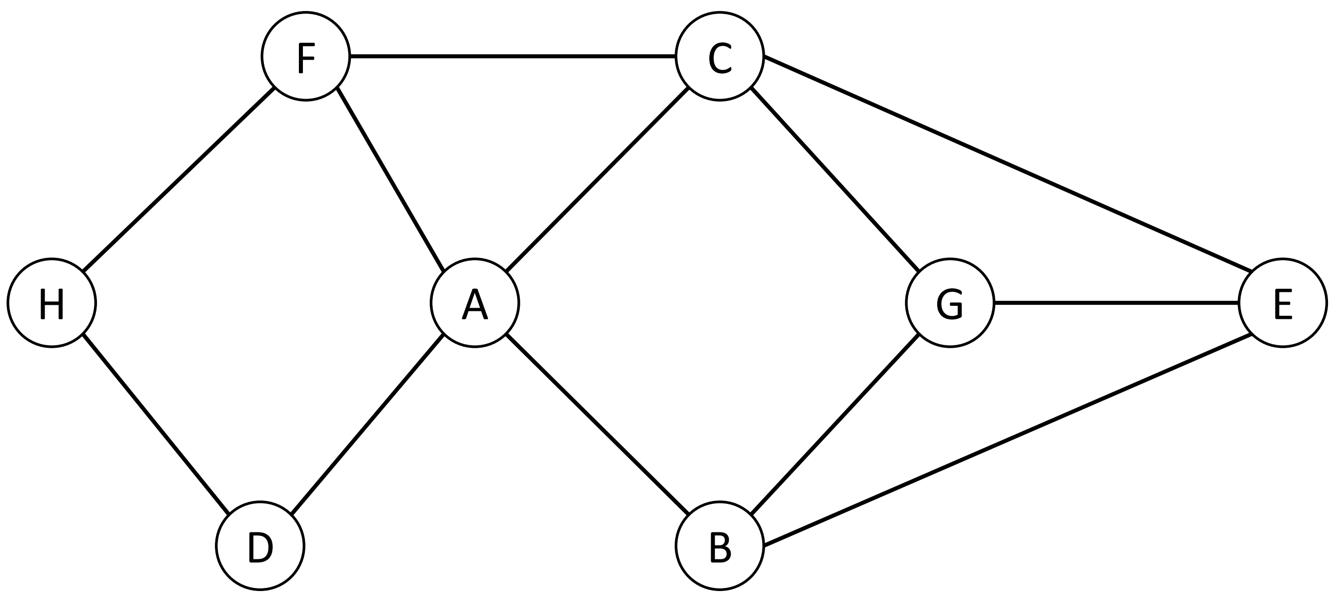
BFS tree:



BFS / **DFS** Forest

Consider only those edges used to discovery a new vertex. We obtain a tree which spans each connected component of the original graph.

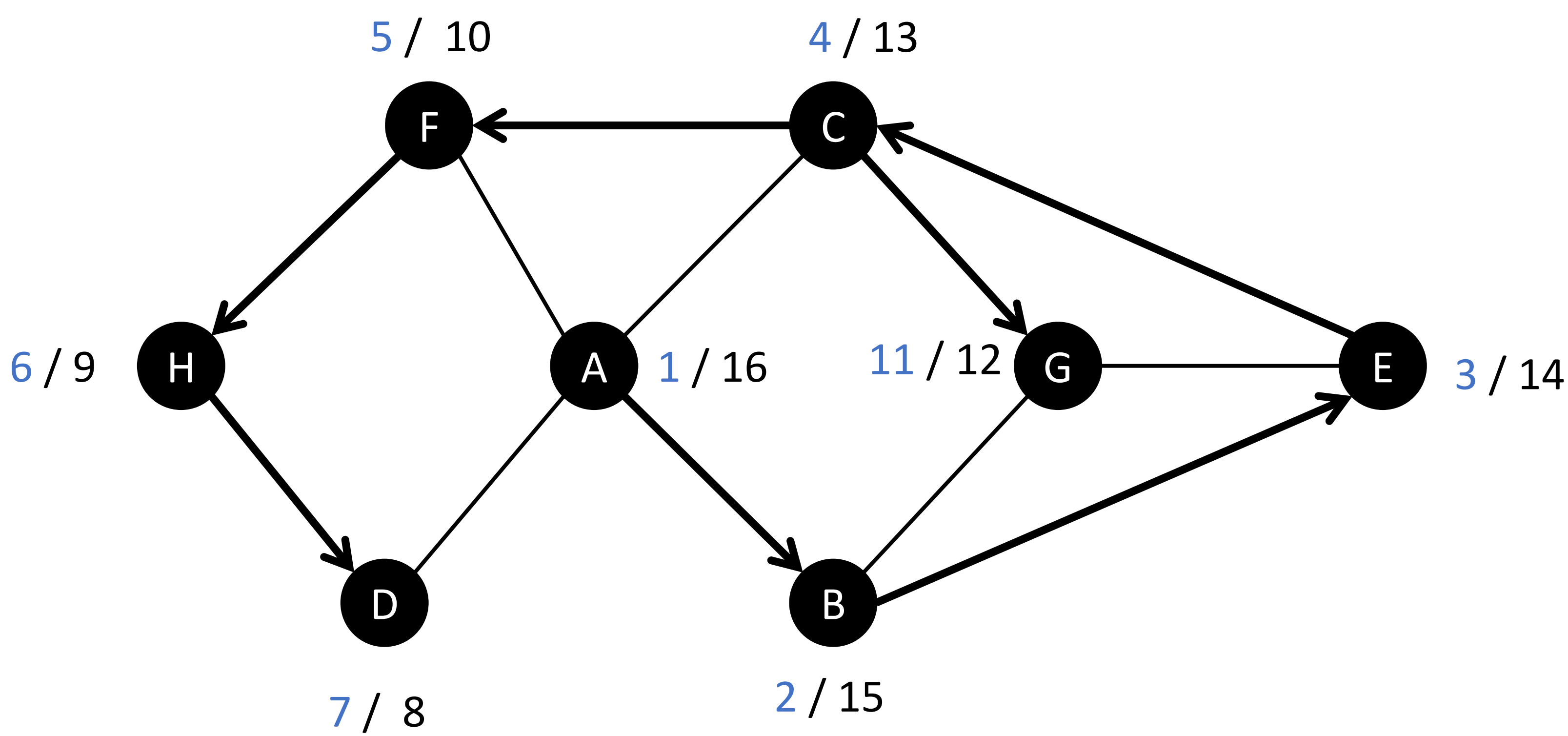
Original Graph:



BFS / **DFS** Forest

Consider only those edges used to discovery a new vertex. We obtain a tree which spans each connected component of the original graph.

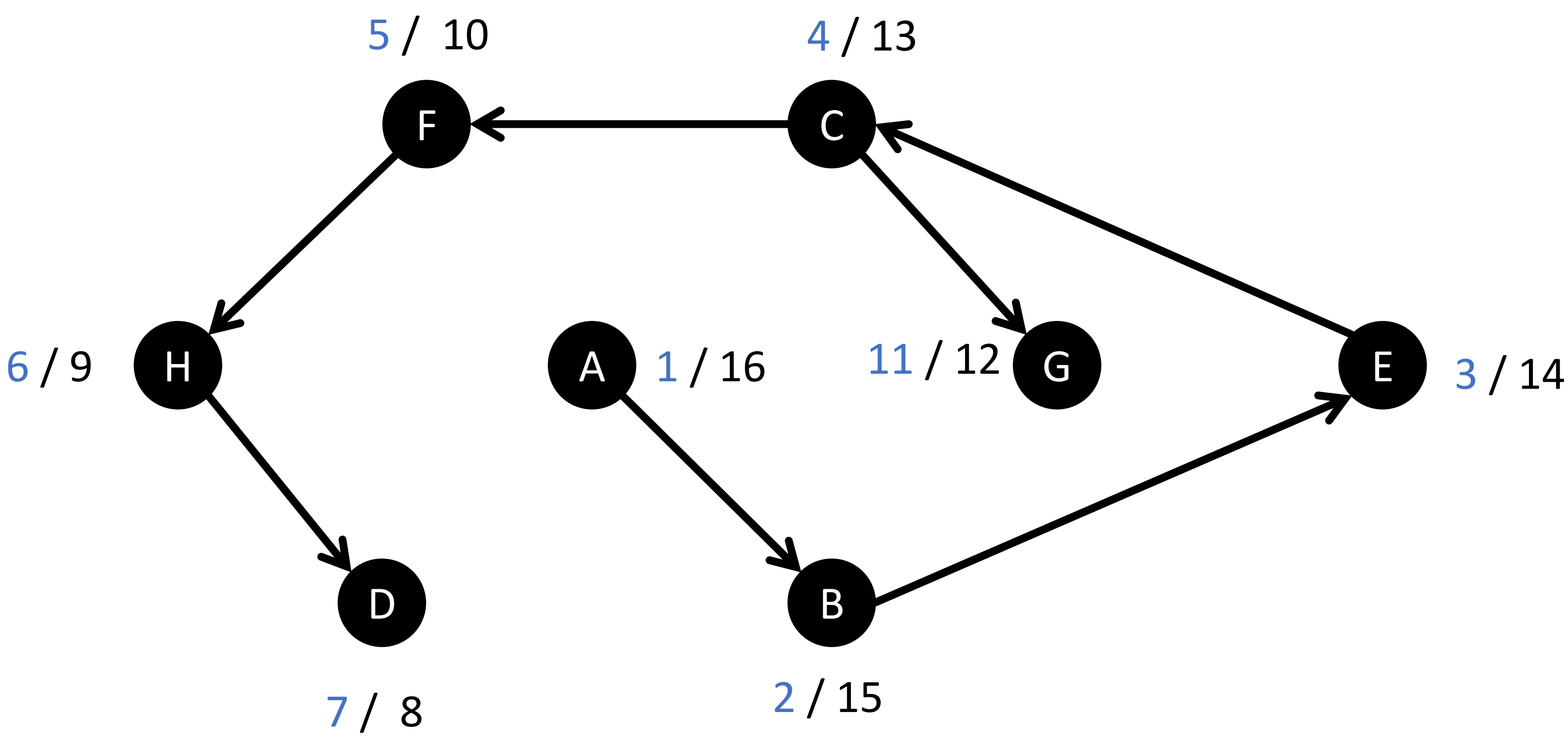
Graph after DFS traversal:



BFS / **DFS** Forest

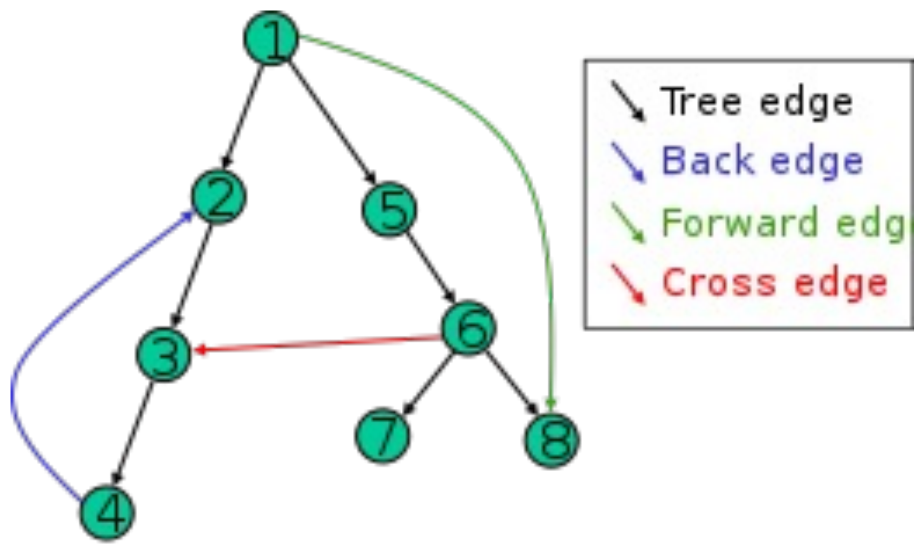
Consider only those edges used to discovery a new vertex. We obtain a tree which spans each connected component of the original graph.

DFS tree:



Classification of edges

The traversals can classify each edge (u,v). Consider the resulting graph drawn like a tree rooted at the starting vertex.



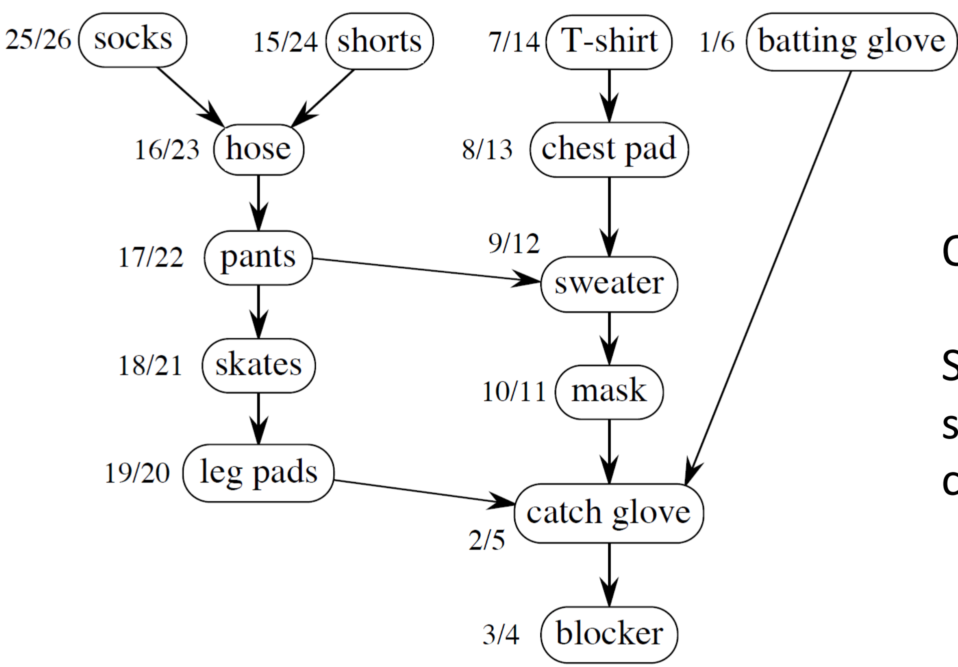
- **Tree edge**: the dark black edges (→) used to discovery a new vertex v from u (that is, $v.\pi = u$)
- **Back edge**: v is an ancestor of u
- **Forward edge**: v is at a descendent level of u
- **Cross edge**: v is neither an ancestor nor descendent of u in the BFS/DFS tree

Topological Sort

A **topological sort** of a directed acyclic graph (DAG) is a linear ordering of all its vertices such that if G contains an edge *(u,v)*, then *u* appears before *v* in the ordering.

Approach: use DFS; as a vertex is finished (time 'out' marked), put it in the front of the list

Ex: DAG of dependencies for putting on goalie equipment for ice hockey



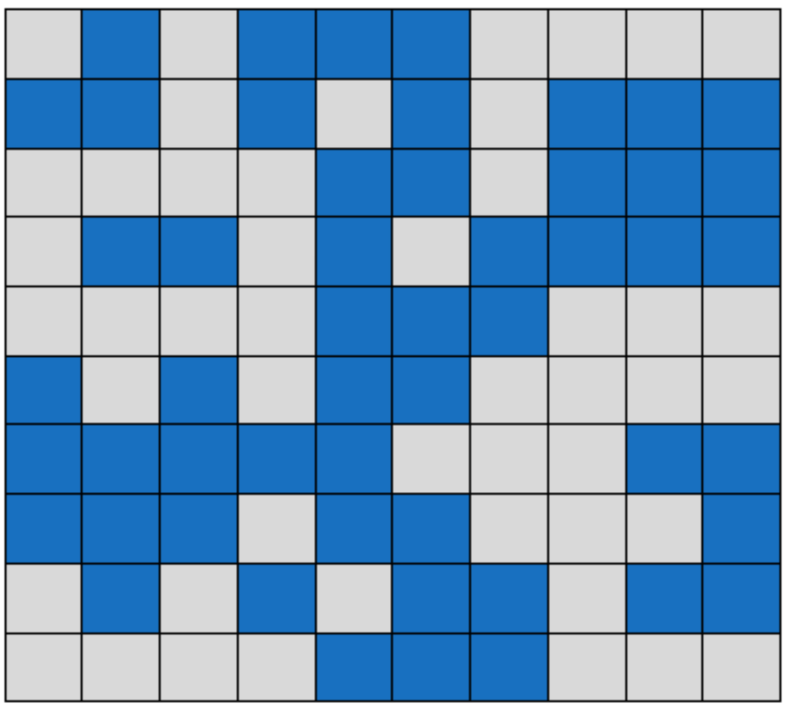
One topological order:

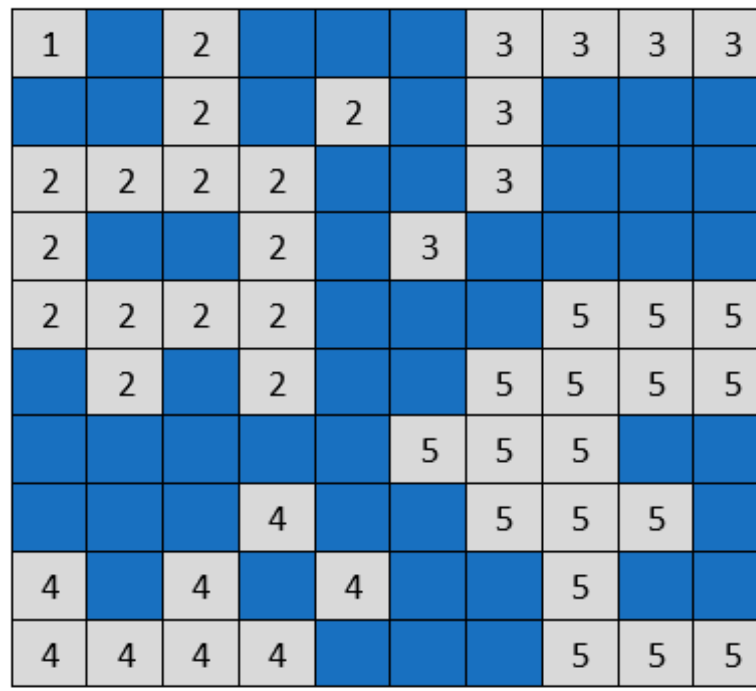
Socks, Shorts, Hose, Pants, Skates, Leg pads, T-shirt, Chest pad, Sweater, Mask, batting glove, catch glove, blocker

Other: islands

Given a binary matrix where 0 represents water and 1 represents land, and connected ones form an island, count the total islands.

For example, consider the left 10x10 image, where blue is water and land is grey. There are a total of five islands present. They are marked by the numbers 1-5 in the right image.





Other: positivity

Given an M by N matrix of integers where each cell can contain a negative, zero, or a positive value, determine the minimum number of passes required to convert all negative values in the matrix positive.

Only a non-zero positive value at cell `(i , j)` can convert a negative value to present at its adjacent cells `(i-1 , j)`, `(i+1 , j)`, `(i,  j-1)`, and `(i,  j+1)`, i.e., up, down, left and right.

For example, the following matrix needs 3 passes, as demonstrated:

-1	-9		-1	
-8	-3	-2	9	-7
2			-6	
	-7	-3	5	-4

Input Matrix

-1	-9		1	
8	-3	2	9	7
2			6	
	-7	3	5	4

After end of Pass 1

1	-9		1	
8	3	2	9	7
2			6	
	7	3	5	4

After end of Pass 2

1	9		1	
8	3	2	9	7
2			6	
	7	3	5	4

After end of Pass 3

Other: knight problem

Given a chessboard, find the shortest distance (minimum number of steps) taken by a knight to reach a given destination from a given source.

For example, given as input an 8x8 board, a source (7,0), and destination (0,7), the output would produce the minimum steps required is 6 (as illustrated by the following figure.