Logistics

- Project proposal grades will be posted to UBLearns after the class
  - All grades will be on UBLearns
- Project biweekly meetings
  - Just two so far!
  - Reply on Piazza!
- Homework 1 is out after class
  - Due next Monday, before class

Graph traversal

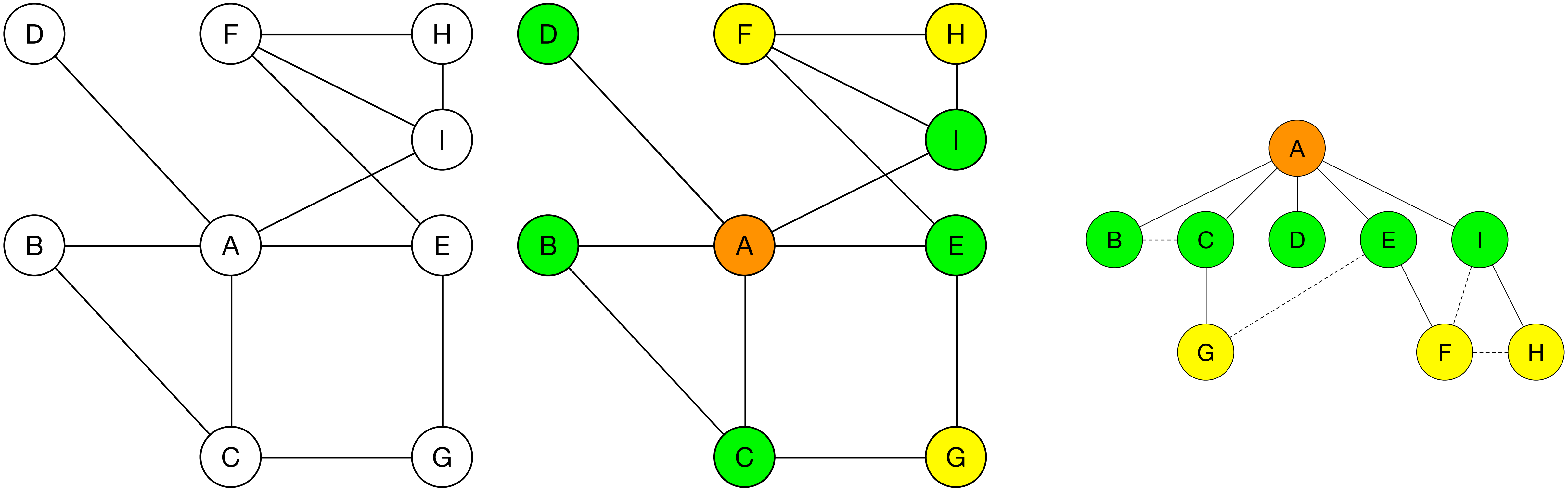
Fundamental building block

- Graph traversal is part of many important tasks
  - Connected components
  - Tree/Cycle detection
  - Articulation vertex finding
- Real-world applications
  - Peer-to-peer networks: Discover what’s around
    - Broadcasting
  - Web crawlers
  - Notion of proximity
  - GPS navigation systems
  - Garbage collection

Today

- Breadth First Search (BFS)
- Depth First Search (DFS)
- How to leverage characteristics of real-world networks?
- Applications of DFS
  - Detecting cycles
  - Topological sort
  - Finding SCCs

Breadth-first search



How do you implement BFS?

- Input: Graph G (n=|V|, m=|E|) and vertex u
- Output: Levels of vertices, parents of vertices

- Level of u is 0, parent of u is -1
- Initially k=0 (current level)

How do you implement BFS?

- Input: Graph G (n=|V|, m=|E|) and vertex u
- Output: Levels of vertices, parents of vertices

- Level of u is 0, parent of u is -1
- Initially k=0 (current level)
- Repeat the following
  - Find all vertices with level k and put them in set N
    - Stop if no vertex found
  - For each vertex v in N
    - Check each of its neighbors
    - If no level assigned yet, set its level as k+1 and its parent as v
  - Increment k

How do you implement BFS?

- Input: Graph G (n=|V|, m=|E|) and vertex u
- Output: Levels of vertices, parents of vertices

- Level of u is 0, parent of u is -1
- Initially k=0 (current level)
- Repeat the following
  - Find all vertices with level k and put them in set N
    - Stop if no vertex found
  - For each vertex v in N
    - Check each of its neighbors
    - If no level assigned yet, set its level as k+1 and its parent as v
  - Increment k

```
Complexity?
–  Initialize level,
   parent arrays: O(n)
–  Scanning level
   information: O(n)
   –  r levels: O(rn)
–  Checking neighbors:
   O(m)

–  Total: O(n+rn+m)
–  Worst case
   –  r is n (chain)
   –  O (m + n²)
```

Can we do better?

- Aim for the most expensive part
  - Scanning level information; O(rn)
  - For real-world networks, what's the expected value of r?

- We can store the vertices in the next level
  - So, one array for the current level vertices, another for the next
  - Combine them, just make first come-first serve
    - That's called queue

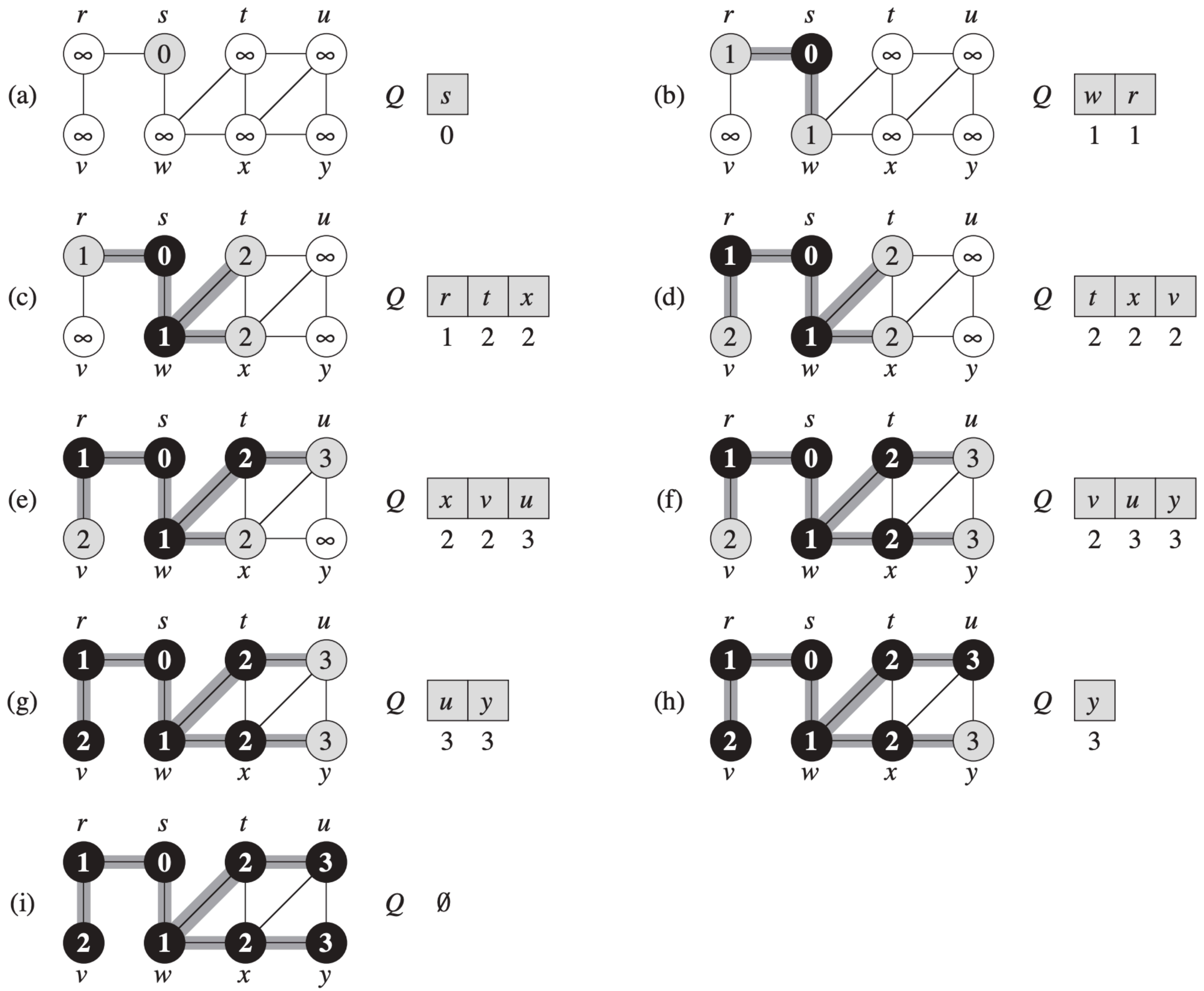
  - So, complexity becomes O(m)

$O(n^2)$ to $O(m)$ if you store the frontier

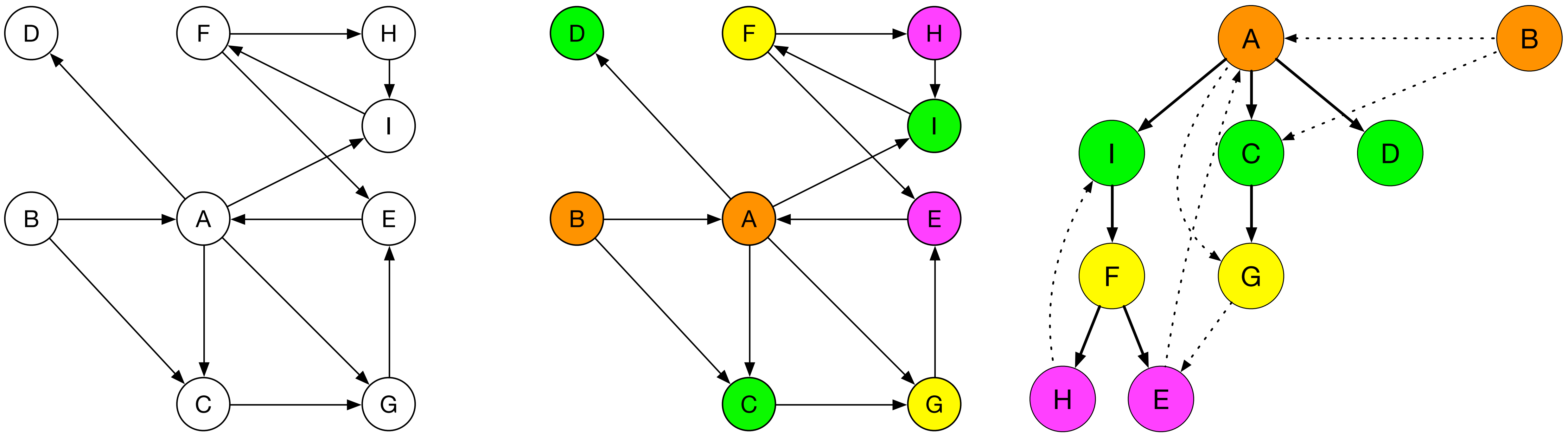
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = Ø
ENQUEUE(Q, s)
while Q ≠ Ø
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

O(n^2) to O(m) if you store the frontier

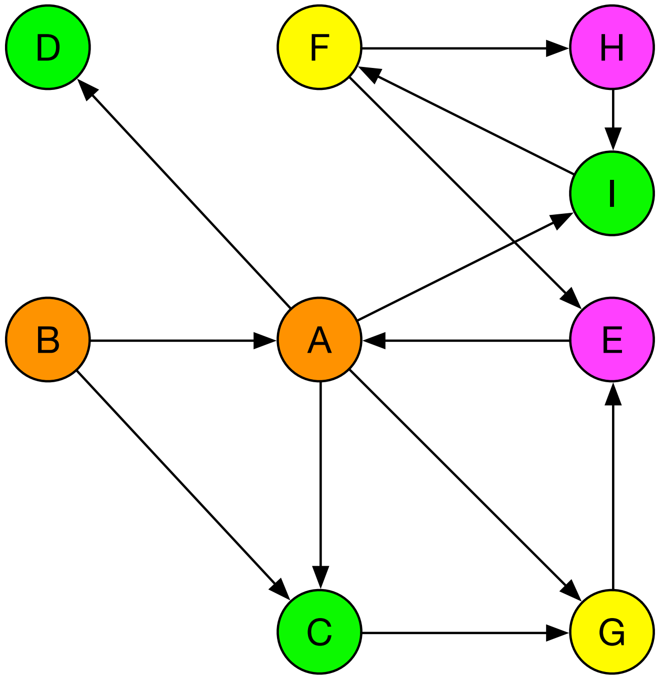
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

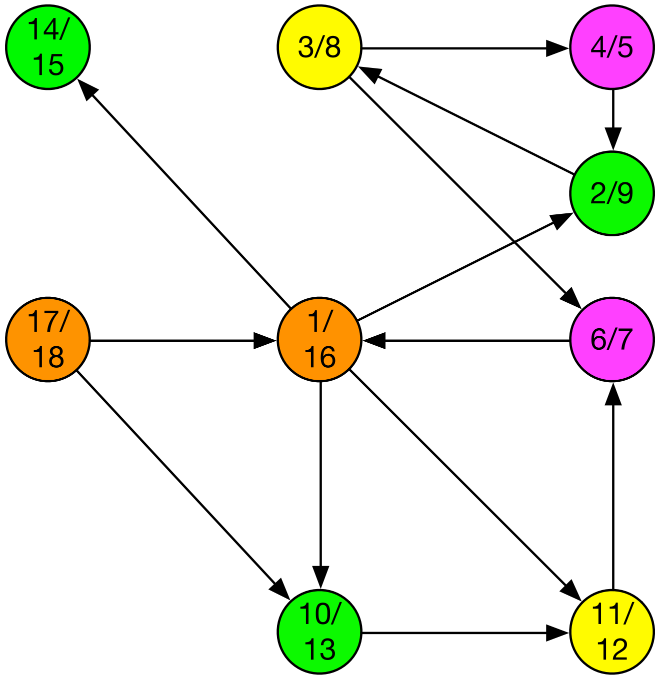


Depth-first search



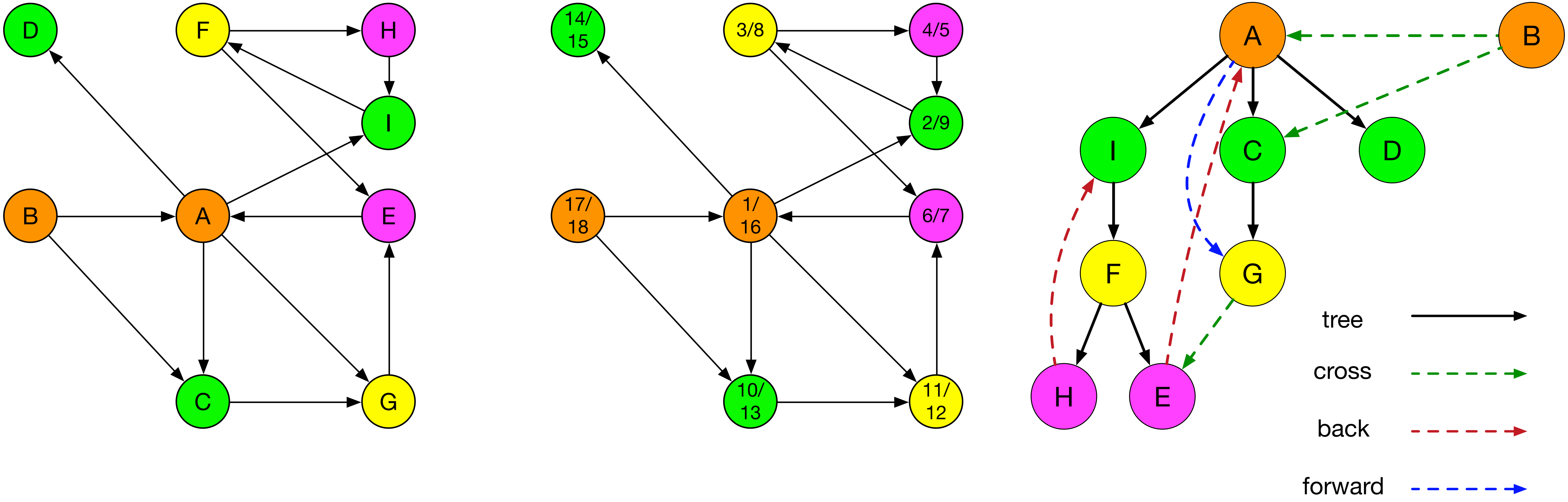
Depth-first search



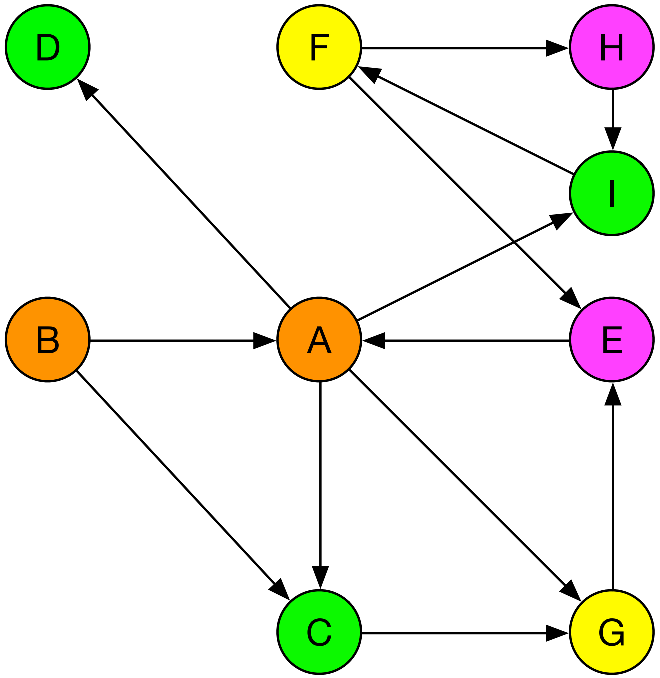


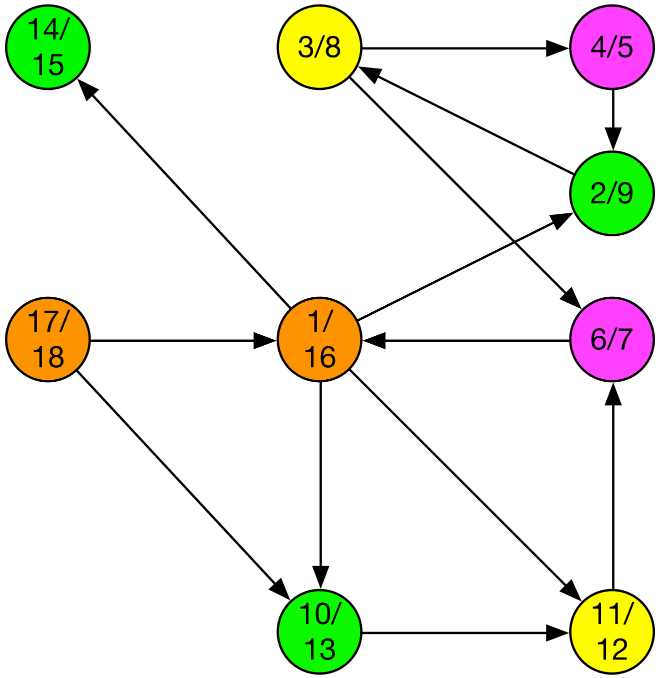
- Starting and finishing times

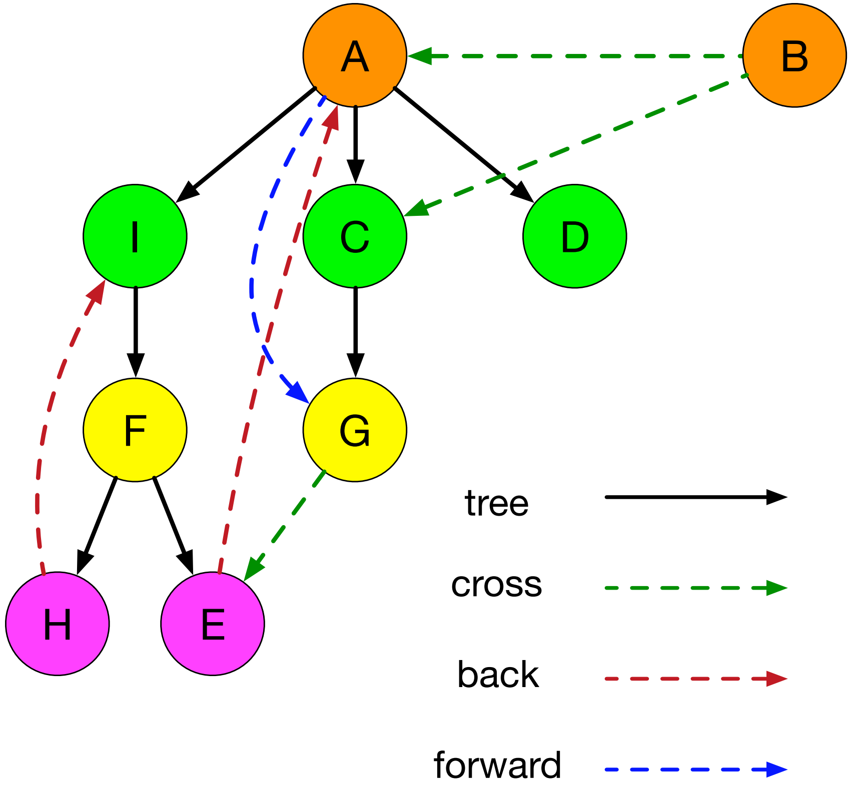
Depth-first search



Depth-first search







- What types of edges can be observed in BFS?

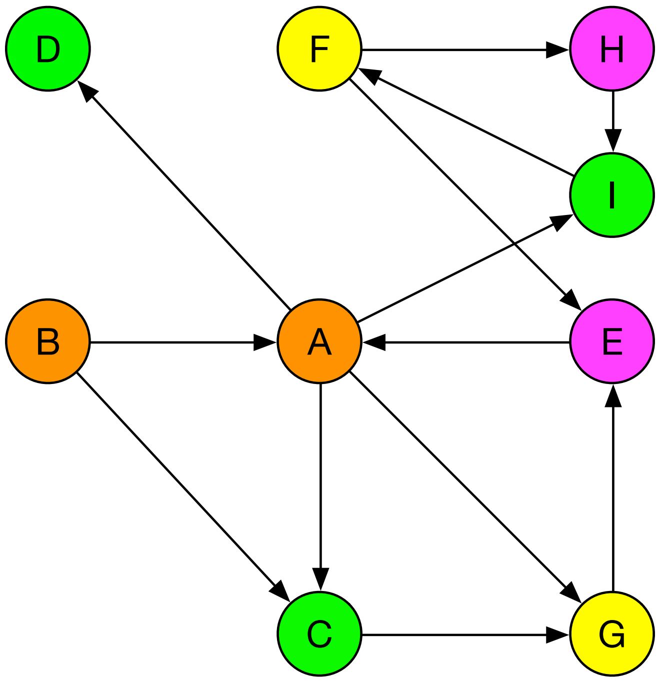
How to implement?

- Notice repetitive pattern
  - Going deep
- Recursion
- Define function DFS

- DFS(G, u, levels, k)
  - Return if levels[u] is assigned
  - Set levels[u] = k
  - For all vertices v which are neighbor to u
    - DFS (G, v, levels, k+1)

Recursion is stack

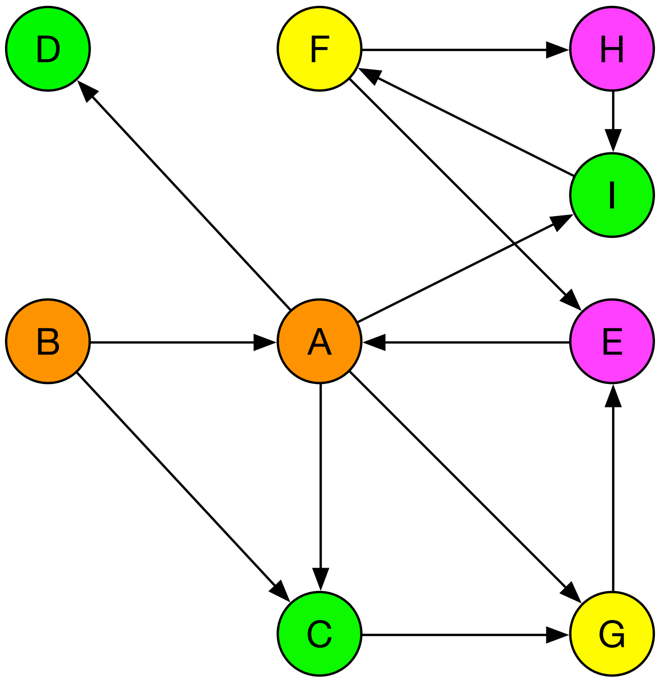
- First in Last Out (FILO)
- Any recursion can be performed by a stack
  - Indeed, CPU literally does that!
- Start from vertex A



- Complexity?

Recursion is stack

- First in Last Out (FILO)
- Any recursion can be performed by a stack
  - Indeed, CPU literally does that!
- Start from vertex A



- Complexity?
  - O(m)

BFS vs. DFS

- Which ones sounds more natural?

- Which one seems to be easier to be parallelized ?
  - Watch for race conditions

- Queue vs. stack

Sometimes you need multiple BFSs

- Coarse-level parallelism
  - Each BFS by a processing unit

- Betweenness centrality
- Closeness centrality
- One-to-all shortest paths

- Next week!

How to adapt your algorithm for real-world data?

- BFS is more popular (Graph500)

How to adapt your algorithm for real-world data?

- BFS is more popular (Graph500)
- What characteristic of real-world networks can be used to optimize BFS?
  - Sparseness?
  - Diameter?

How to adapt your algorithm for real-world data?

- BFS is more popular (Graph500)
- What characteristic of real-world networks can be used to optimize BFS?
  - Sparseness?
  - Diameter?

- Direction-Optimizing Breadth-First Search, SC 2012
  - By Scott Beamer, Krste Asanovic, and David Patterson
  - http://www.scottbeamer.net/pubs/beamer-sc2012.pdf

Queue-based BFS

```
function breadth-first-search(vertices, source)
    frontier ← {source}
    next ← {}
    parents ← [-1,-1,. . . -1]
    while frontier ≠ {} do
        top-down-step(vertices, frontier, next, parents)
        frontier ← next
        next ← {}
    end while
    return tree
```

Fig. 1. Conventional BFS Algorithm

```
function top-down-step(vertices, frontier, next, parents)
    for v ∈ frontier do
        for n ∈ neighbors[v] do
            if parents[n] = -1 then
                parents[n] ← v
                next ← next ∪ {n}
            end if
        end for
    end for
```

Fig. 2. Single Step of Top-Down Approach

Queue-based BFS

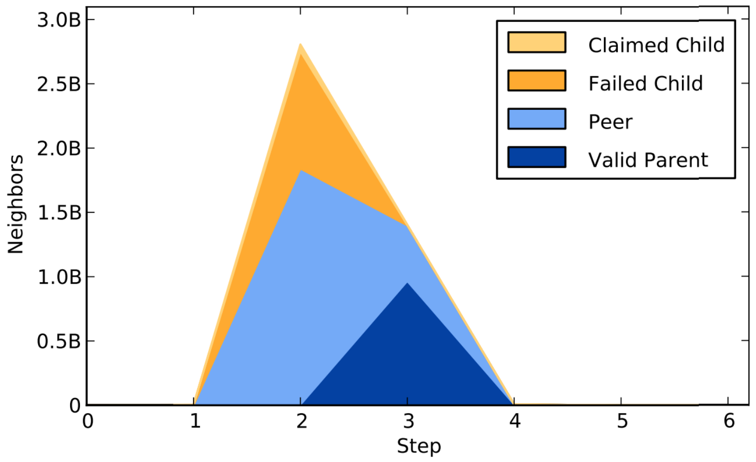
function breadth-first-search(vertices, source)
    frontier $\leftarrow$ {source}
    next $\leftarrow$ {}
    parents $\leftarrow$ [-1,-1,. . . -1]
    while frontier $\neq$ {} **do**
        top-down-step(vertices, frontier, next, parents)
        frontier $\leftarrow$ next
        next $\leftarrow$ {}
    end while
    return tree

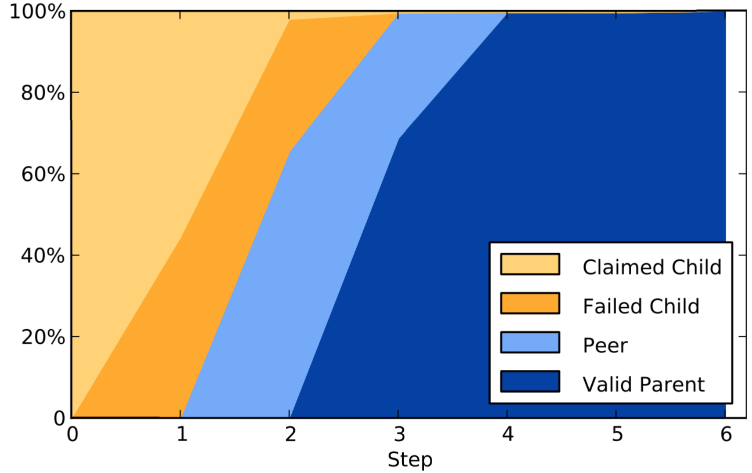
Fig. 1. Conventional BFS Algorithm

function top-down-step(vertices, frontier, next, parents)
    for v $\in$ frontier **do**
        for n $\in$ neighbors[v] **do**
            if parents[n] = -1 **then**
                parents[n] $\leftarrow$ v
                next $\leftarrow$ next $\cup$ {n}
            end if
        end for
    end for

Fig. 2. Single Step of Top-Down Approach

- Let's analyze the frontier in each step





Avoid redundant work in intermediate steps

- Very simple idea
- Check unvisited vertices!
- Bottom-up

- Easier to parallelize, no atomic

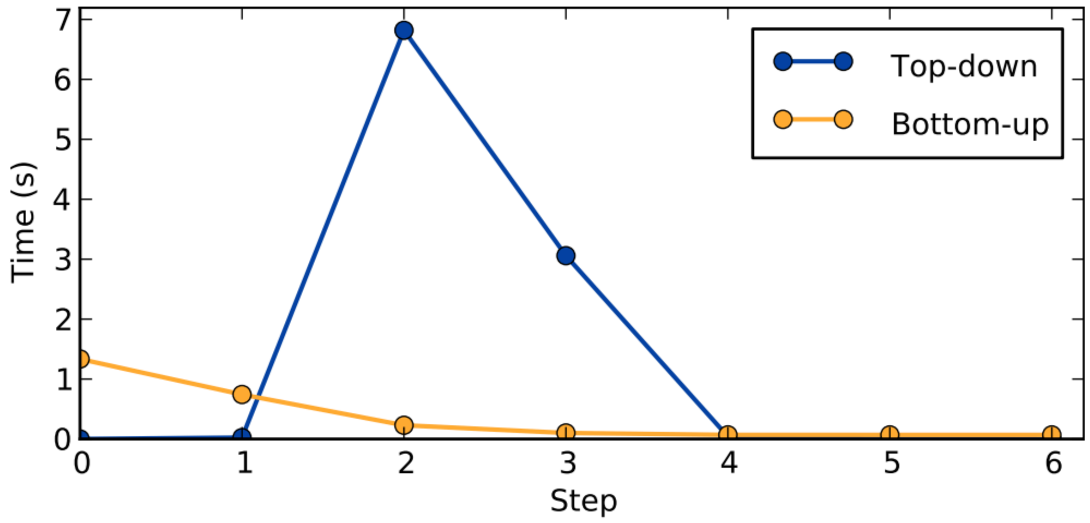
- For directed networks, this requires inverted graph

```
function bottom-up-step(vertices, frontier, next, parents)
    for v ∈ vertices do
        if parents[v] = -1 then
            for n ∈ neighbors[v] do
                if n ∈ frontier then
                    parents[v] ← n
                    next ← next ∪ {v}
                    break
                end if
            end for
        end if
    end for
```

Fig. 5. Single Step of Bottom-Up Approach

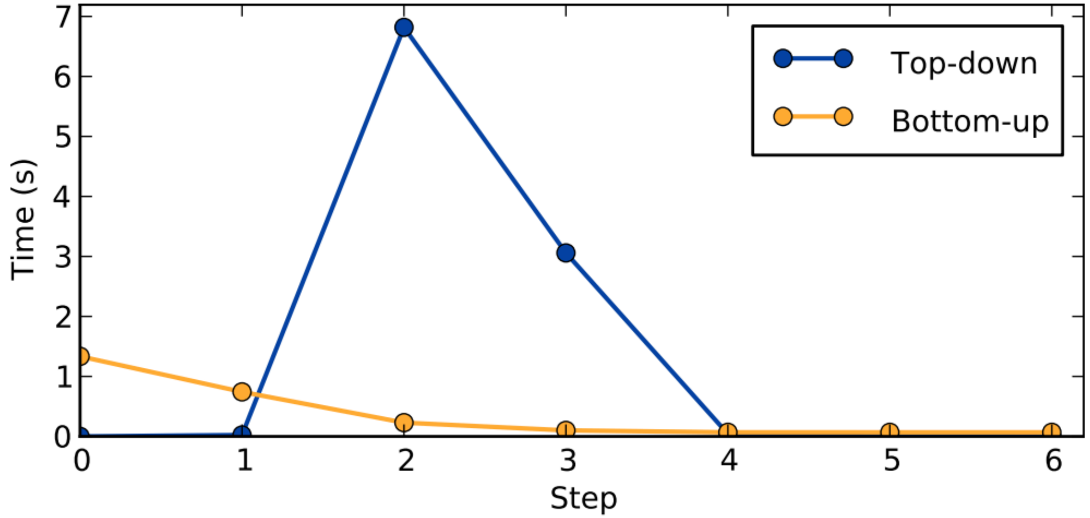
Hybrid approach

- Bottom-up is more effective when frontier is large
    - Unvisited set is small

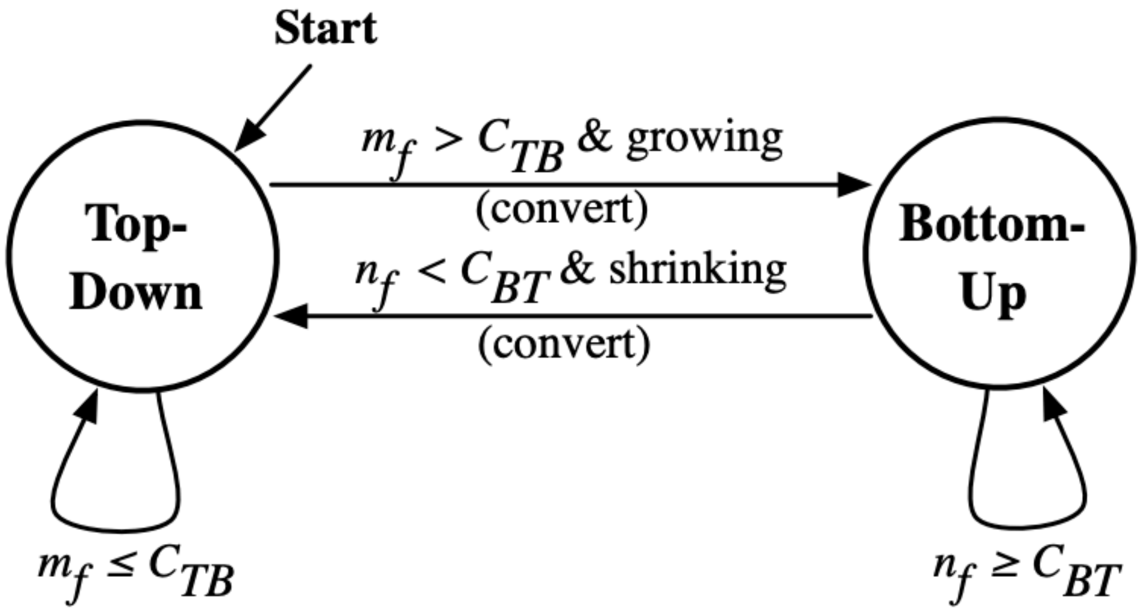


Hybrid approach

- Bottom-up is more effective when frontier is large
  - Unvisited set is small



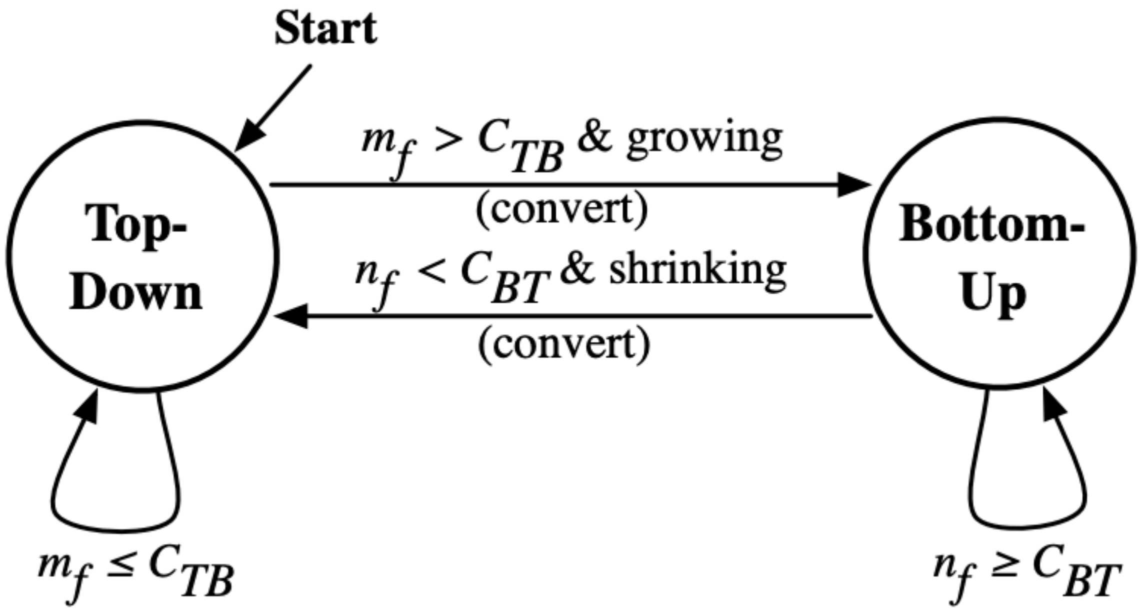
- Start with top-down
- Switch to bottom-up wher frontier gets large
- Switch back to top-down when frontier gets small



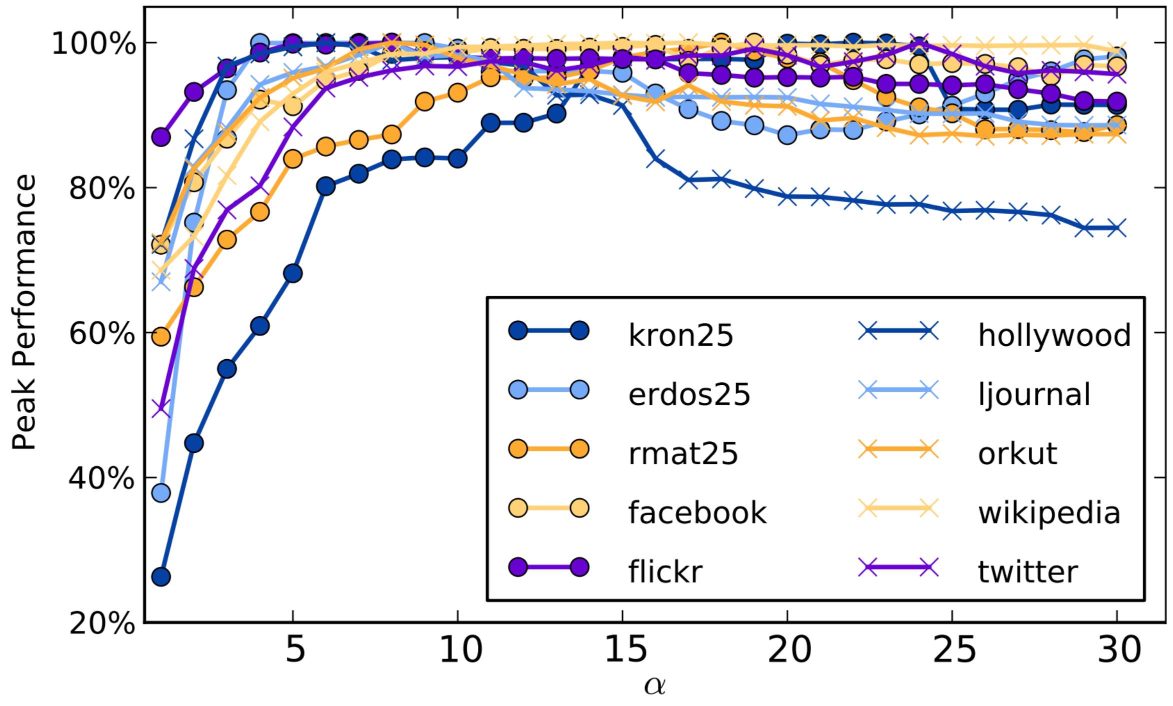
Hybrid approach

- m_f = edges adjacent to frontier
- m_u = edges adjacent to unvisited vertices
- n_f = vertices in frontier
- n = all vertices

- $m_f > \frac{m_u}{\alpha} = C_{TB}$
- $n_f > \frac{n}{\beta} = C_{BT}$

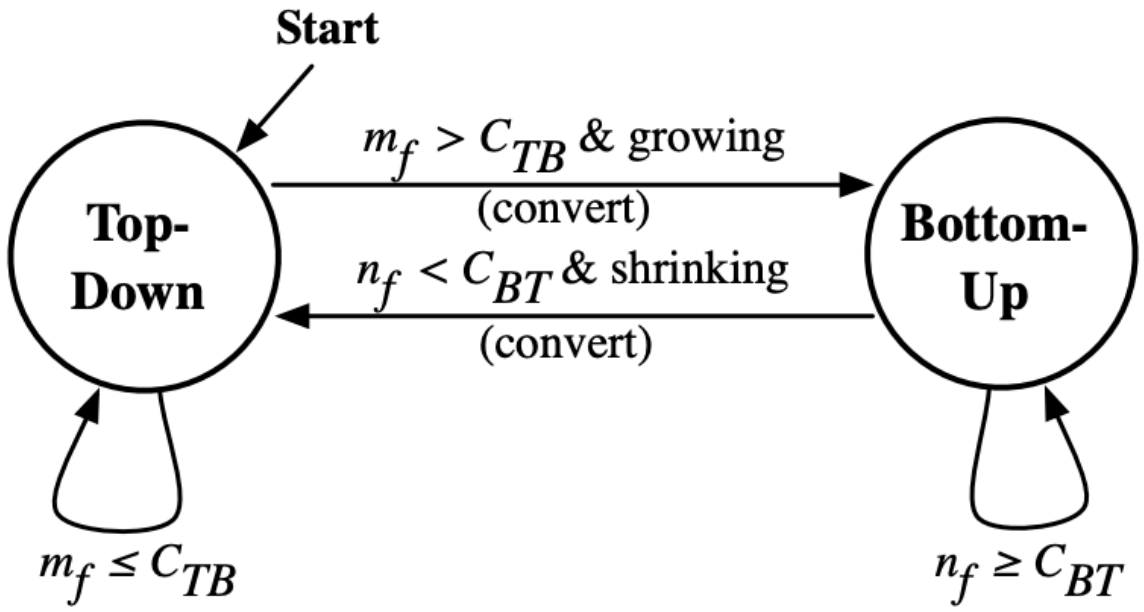


Optimizing α and β

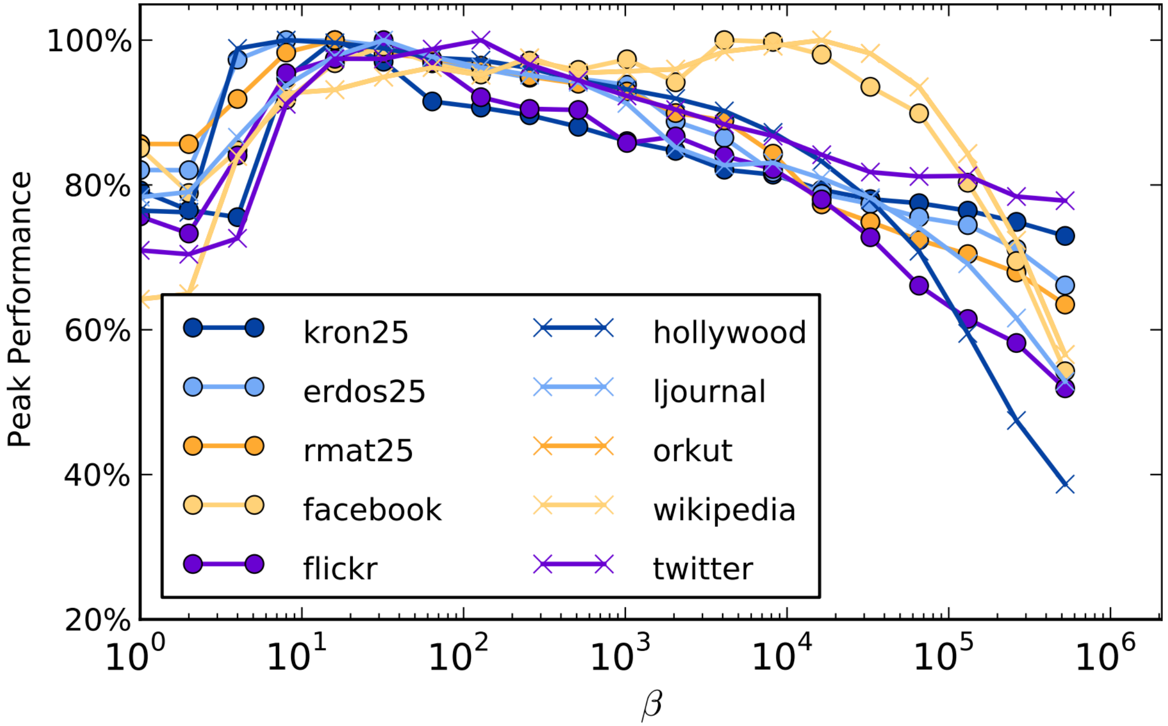


- Experimentally;
  - $\boldsymbol{\alpha}$ =14 is ideal

- $m_f > \frac{m_u}{\alpha} = C_{TB}$
- $n_f > \frac{n}{\beta} = C_{BT}$

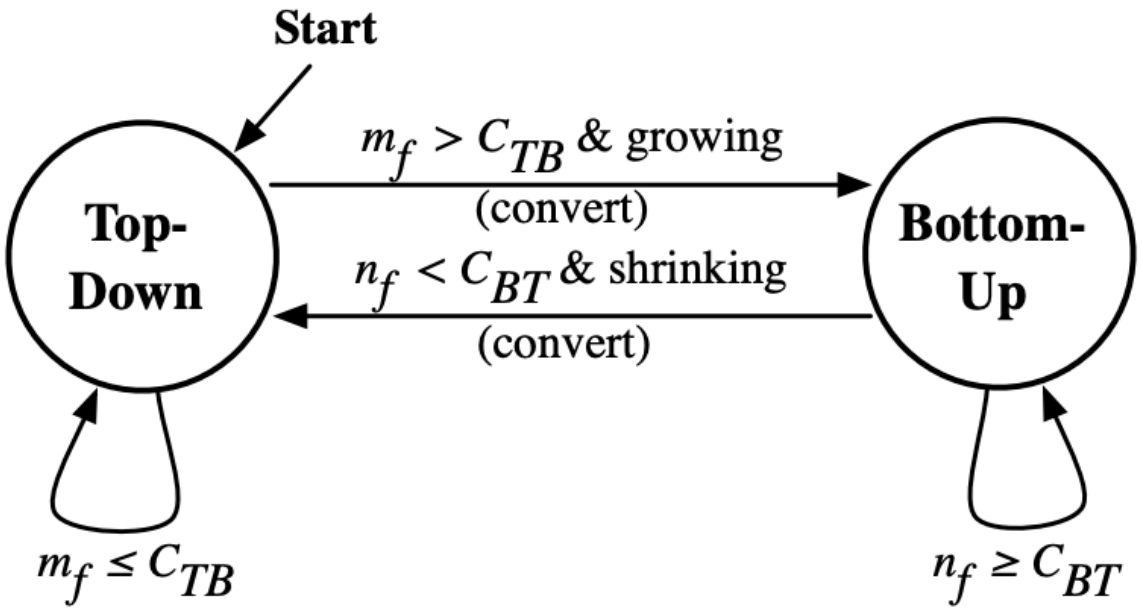


Optimizing α and β



- Experimentally;
  - $\boldsymbol{\beta}$ = 24 is ideal

- $m_f > \dfrac{m_u}{\alpha} = C_{TB}$
- $n_f > \dfrac{n}{\beta} = C_{BT}$



Hybrid is faster

- Significant speedup w.r.t. state-of-the-art

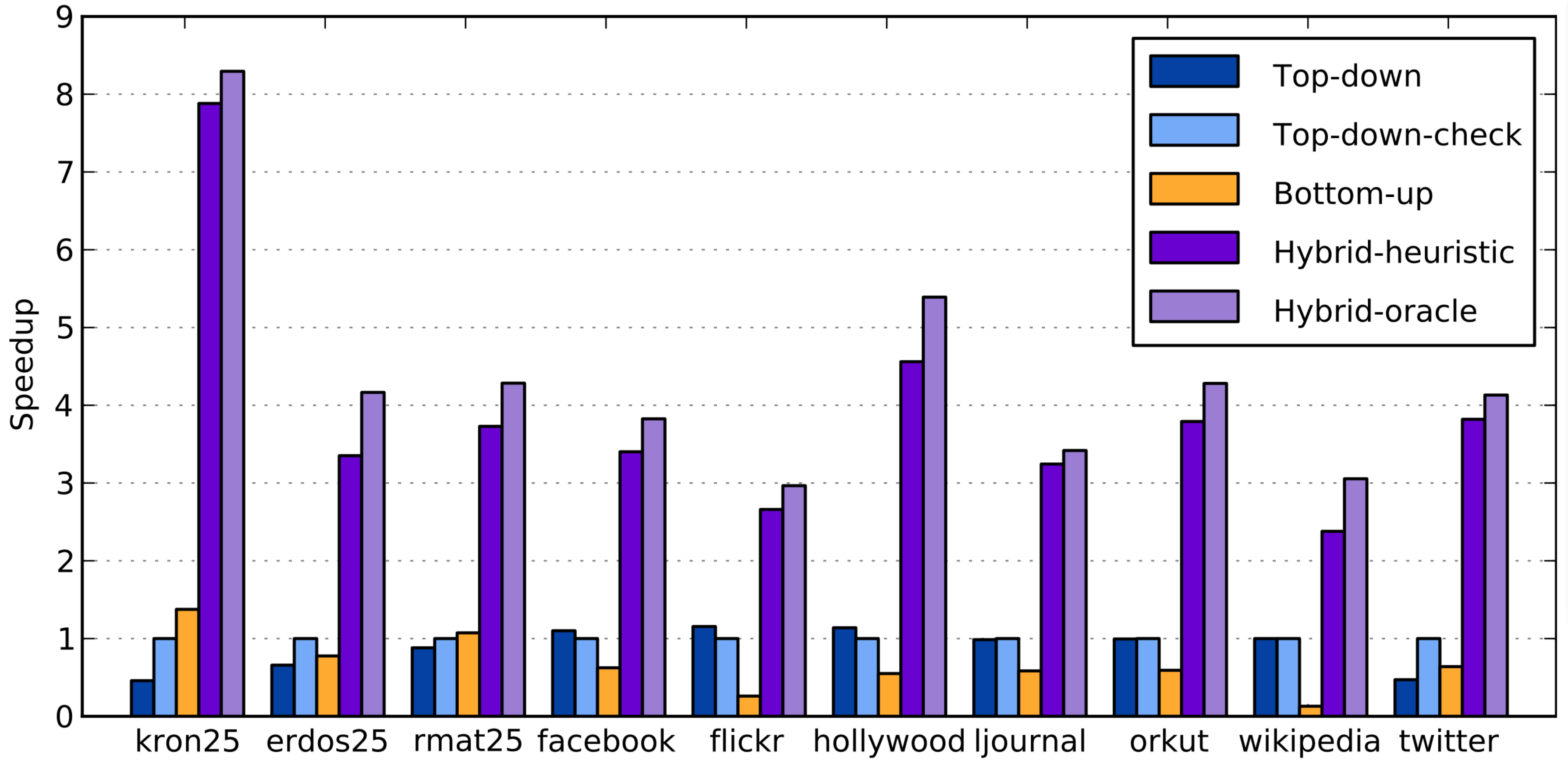


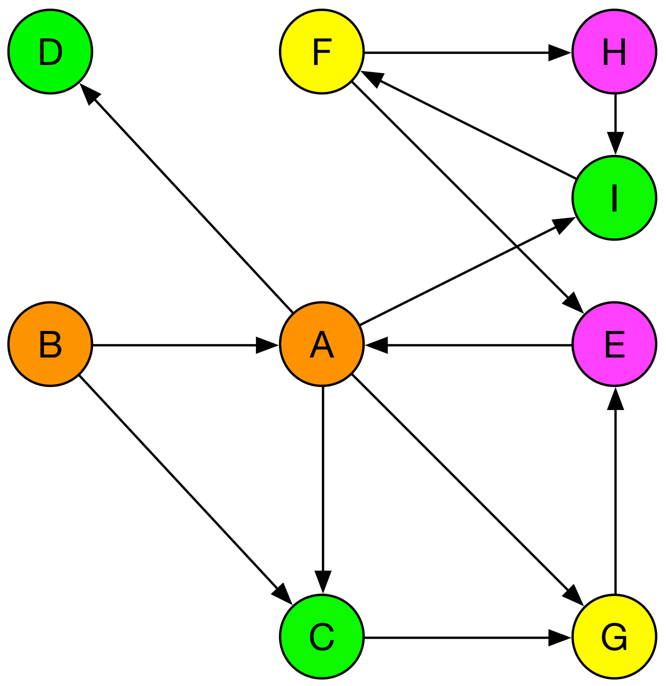
Fig. 10. Speedups on the 16-core machine relative to *Top-down-check*.

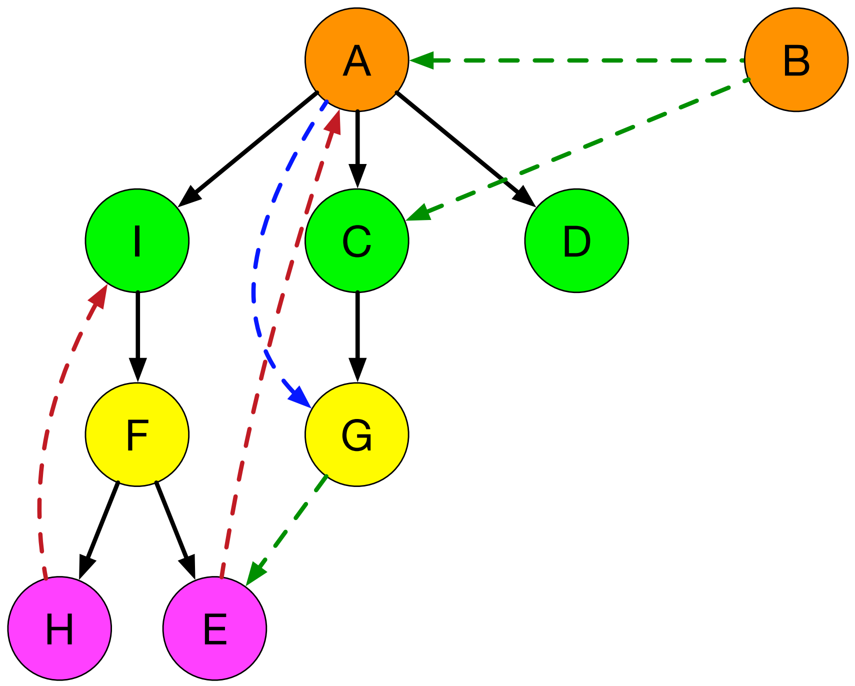
Back to DFS

- Three applications:
  - Cycle detection
  - Topological sort
  - Finding strongly connected components

Finding cycles by DFS

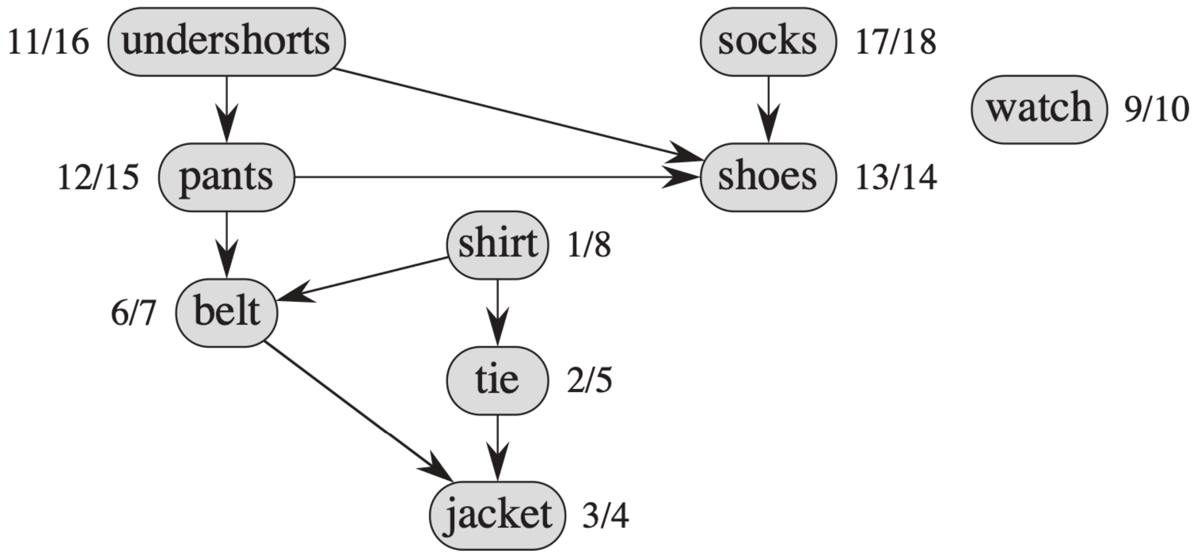
- Just make a DFS from a vertex
- Check for the back edges!

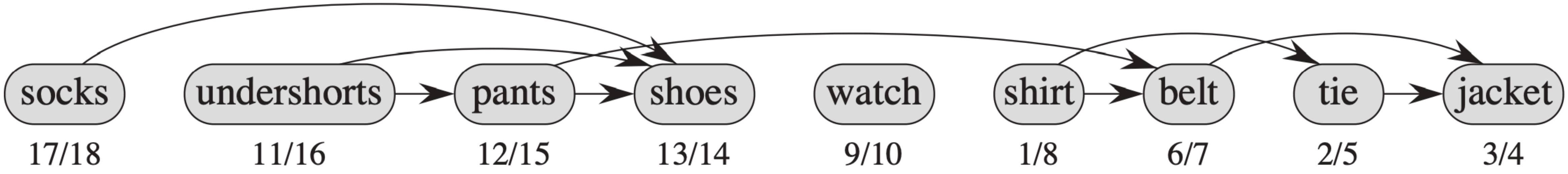




Topological Sort

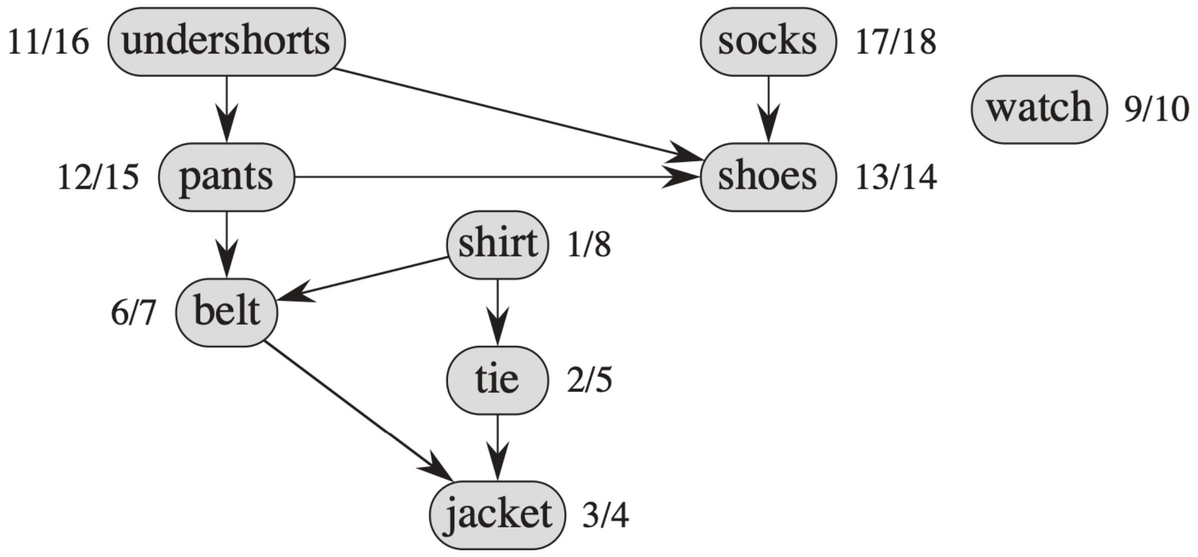
- Determine a partial ordering
  - Call DFS
  - Find finishing times
  - Report in the reverse order
- What kind of graph is this?

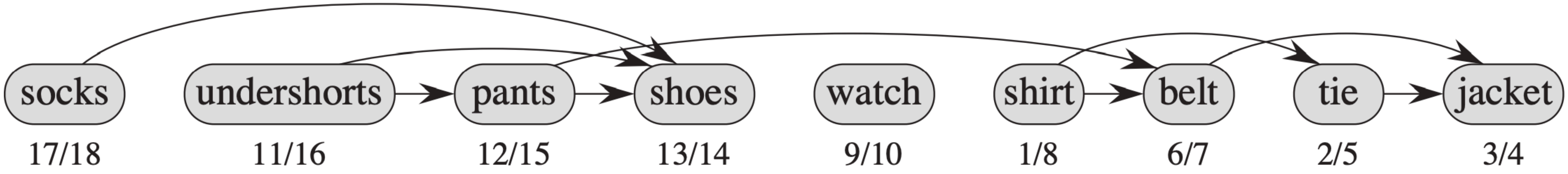




Topological Sort

- Determine a partial ordering
  - Call DFS
  - Find finishing times
  - Report in the reverse order
- What kind of graph is this?
  - DAG (directed acyclic graph)
  - Meaningless otherwise



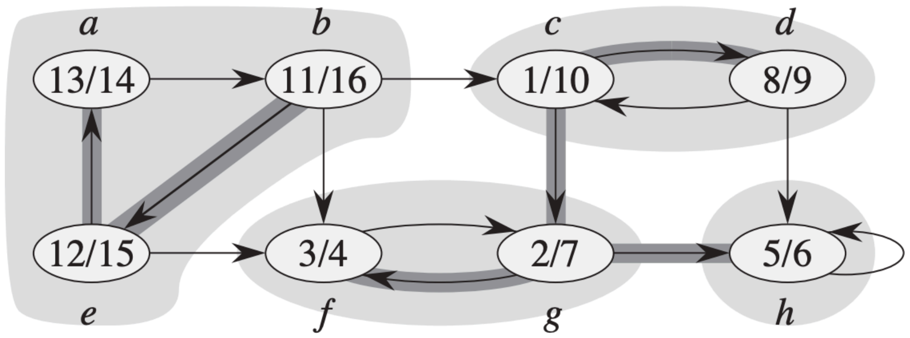


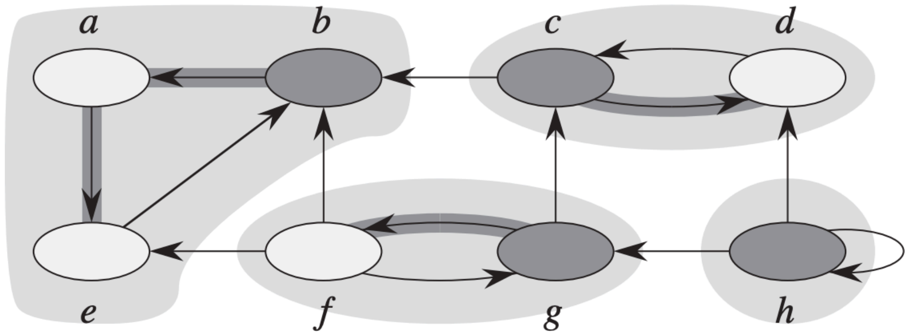
Finding SCCs by DFS

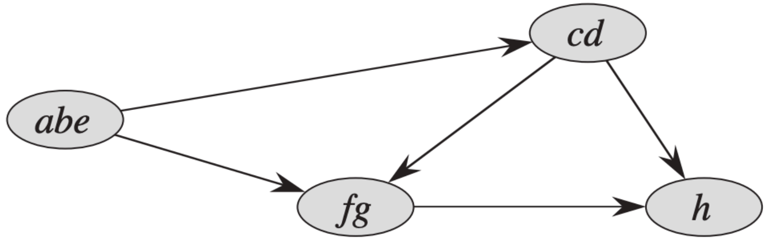
- ?

Finding SCCs by DFS

- By using starting and finishing times
- Remember: any directed graph is a DAG of its SCCs
- G^T is the transpose of G
  - Directions reversed
- Use DFS(G) and DFS(G^T)







Finding SCCs by DFS

- Find DFS (G) to find finishing times of all vertices
  - Can start from any node
- Compute G^T
- Find DFS (G^T), but
  - Consider the vertices in decreasing order of their finishing times found above
- Output each component in depth-first forest as an SCC

Example

- Started from node A
  - Start/finish times found
- Directions is reversed

- B
- AEGCFIH
- D