Lecture 10 – Graph Algorithms II

AIAA 5037 Advanced Algorithms and Data Structures

Ying Sun, AI Thrust

Outline

- Topological Sort
- Breath-First Search
- Depth-First Search
- Strongly Connected Component
- Single-Source Shortest Path
  - Dynamic Programming for DAG
  - Bellman-Ford Algorithm
  - Dijkstra's Algorithm
  - Application: Difference Constraints Problem

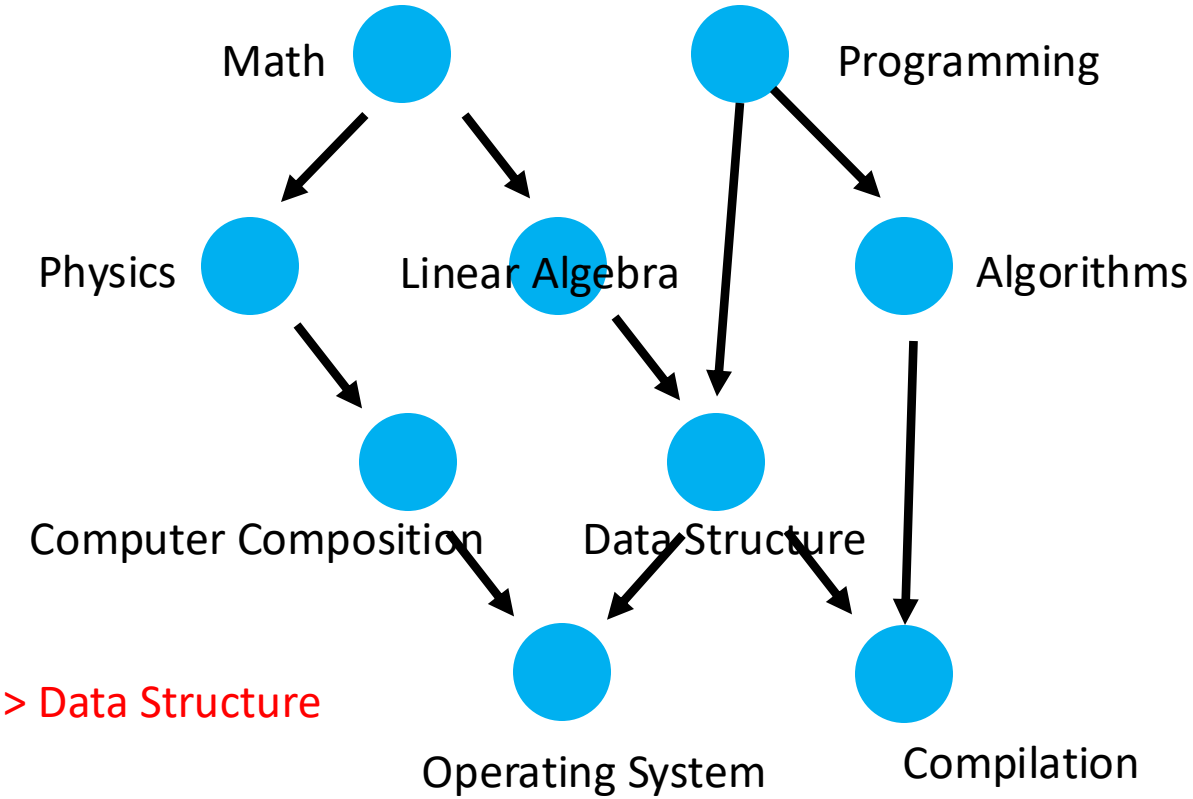
Topological Sort

Topological Sort

Given a directed acyclic graph $G =< V, E >$. Find an order $O =< o_1, o_2, \ldots, o_V >$ satisfying there does not exist pair $< i, j >$ that $i < j$ and $< o_j, o_i > \in E$

Example:

- Course Selection
- Manufacturing Workflows



Intuitive understanding

- Find an order of doing things that respect the given pair-wise dependencies

Math > Physics > Programming > Linear Algebra > Algorithms > Data Structure > Computer Composition > Operating System > Compilation

Optimal Substructure

Given a topological sort $O =< o_1, o_2, \ldots, o_v >$ for graph $G =< V, E >$, deleting o_1 generates a topological sort of a graph $G' =< V', E' >$, where

$$V' = V - \{o_1\},$$

$$E' = \{< u', v' > | < u', v' > \in E, u' ! = o_1, v' ! = o_1\}$$

Proof: obvious, for the original order, there is no $< i, j >$ that $i < j$ and $< o_j, o_i > \in E$, deleting a node does not change of pair-wise relationship of the rest of the nodes

Recurrence for Optimized Value

Given a topological sort $O =< o_1, o_2, \dots, o_v >$ for graph $G =< V, E >$, deleting o_1 generates a topological sort of a graph $G' =< V', E' >$, where

$$V' = V - \{o_1\},$$

$$E' = \{< u', v' > | < u', v' > \in E, u'! = o_1, v'! = o_1\}$$

Recurrence: $f(G)$ - whether you can find a topological sort for G

$$f(G) = \max_{u \in V} f\big(g(G,u)\big)\mathbb{I}\{|\{< v,u > | v \in V - \{u\}, < v,u > \in E\}| = 0\}$$

Complexity: exponential subproblems (subgraphs)

Greedy Intuition: choose a node that does not have precedents

Is it optimal?

—— For any node without precedents, there exists a topological order starting with it

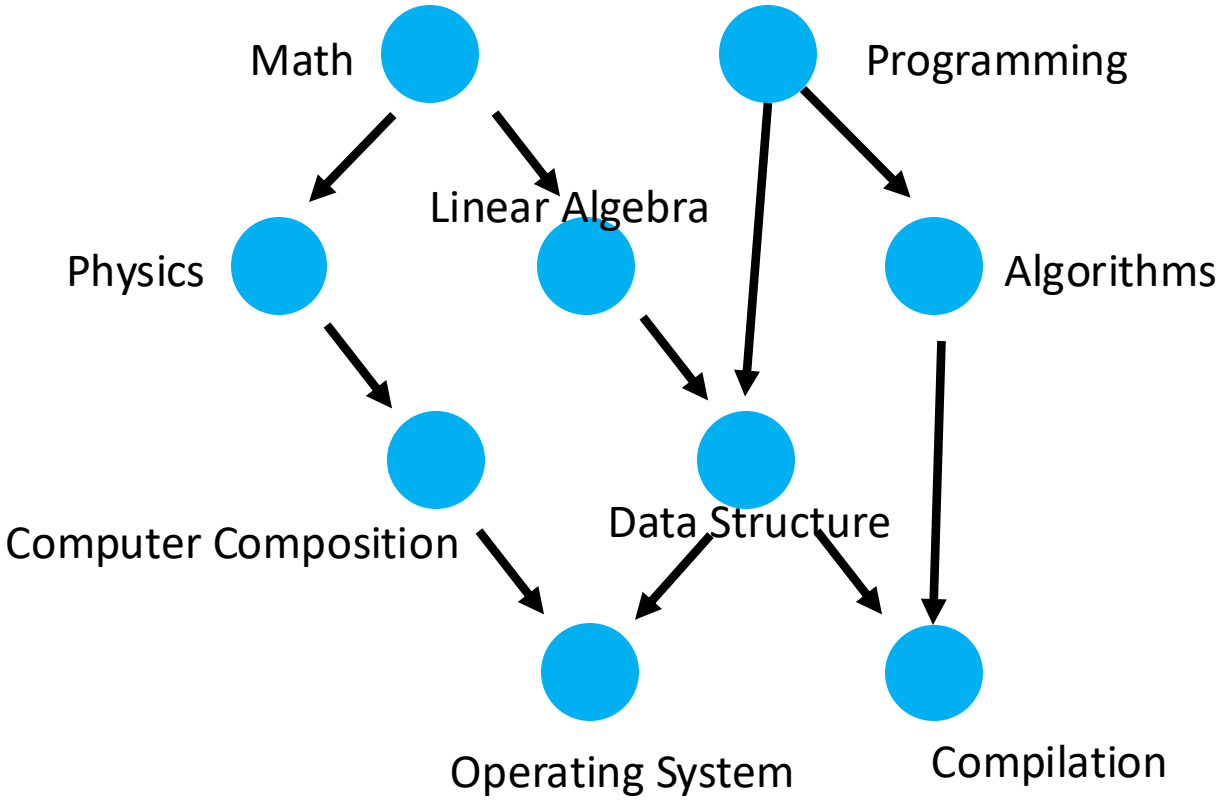
Greedy Choice Property

For any node (u) without precedents, there exists a topological order starting with it

Proof:

- Given any topological sort $O =< o_1, \dots o_k, u, o_{k+2}, \dots o_n >$ where u does not appear as the first, $O' =< u, o_1, \dots o_k, o_{k+2}, \dots o_n >$ is a topological sort

Kahn Algorithm



Math Physics Programming Linear Algebra Algorithms Data Structure Computer Composition Operating System Compilation

Kahn Algorithm

Greedily delete a node that has no predecessor (in-degree is 0)

1. Record the in-degree of each node
2. Pick the node whose in-degree is 0 (delete and put to the list)
3. Update the in-degree of the successors
4. Repeat 2-3 until all the nodes have been picked

```
deg := each vertices' in-degree
Q := all vertices with in-degree=0
ordering := {}
while Q is not empty:
    u = Q.pop()
    ordering.append(v)
    for v ∈ {v'| < u, v' >∈  E}:
        deg[v] -= 1
        if deg[v] is 0: push v to Q
```

Complexity

- Each node will be inserted to the queue once and deleted once – $O(V)$
- Edges from a node are traversed when it is deleted: $O(E)$ in total
- Total time complexity: $O(V + E)$
- Auxiliary Space complexity: $O(V)$

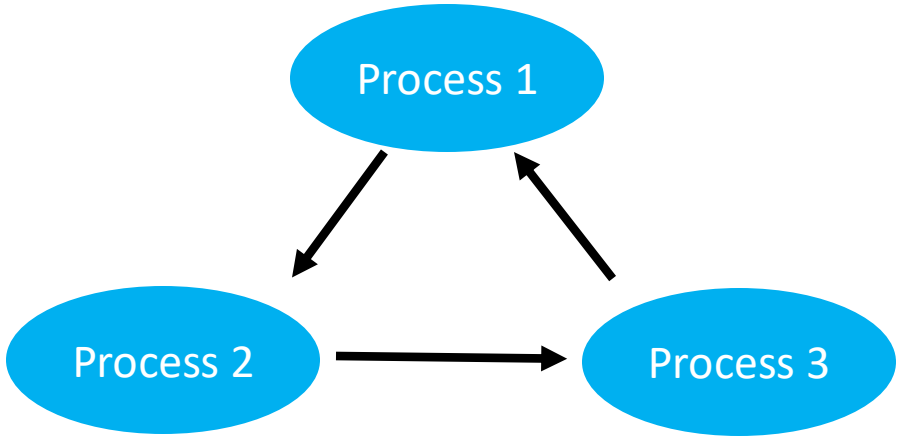
```
deg := each vertices' in-degree
Q := all vertices with in-degree=0
ordering := {}
while Q is not empty:
    u = Q.pop()
    ordering.append(v)
    for v ∈ {v'| < u, v' >∈ E}:
        deg[v] -= 1
        if deg[v] is 0: push v to Q
```

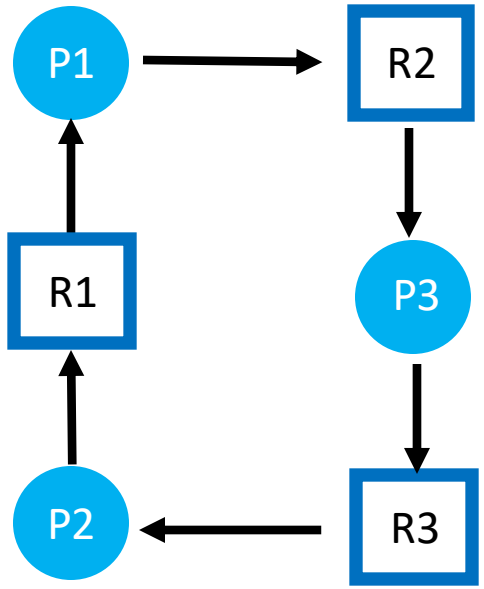
Extension: Cycle Detection

Given a directed graph $G =< V, E >$, detect whether there is cycle in G

- Example applications: detecting deadlocks in Operating Systems
- Solution: a graph with cycle is non-empty when the topological sort algorithm stops

```
deg := each vertices' in-degree
Q := all vertices with in-degree=0
while Q is not empty:
    u = Q.pop()
    for v ∈ {v'| < u, v' >∈ E}:
        deg[v] -= 1
        if deg[v] is 0: push v to Q
for v in V:
    if deg[v] != 0: return "has cycle!"
return "no cycle."
```





Resource Allocation Graph

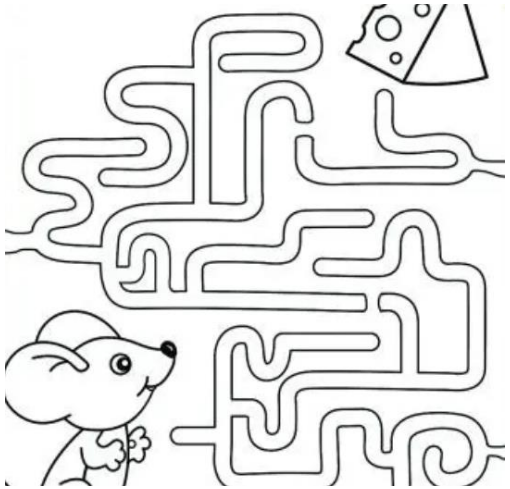
Graph Search

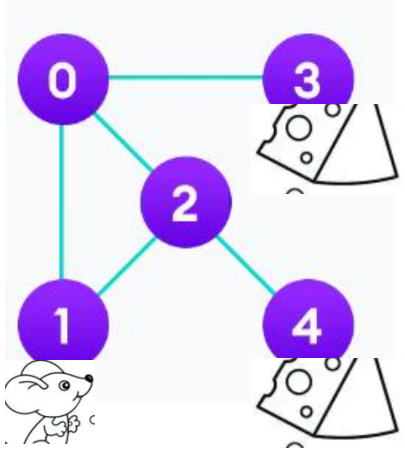
Graph Search

Problem: Given a maze as a graph $G = < V, E >$, where node $v \in V$ means rooms and edges $e \in E$ means connection between rooms, some room contain some cheese $c[v]$. Given a starting point s of the mouse , what is the maximum number of cheese the mouse can get?

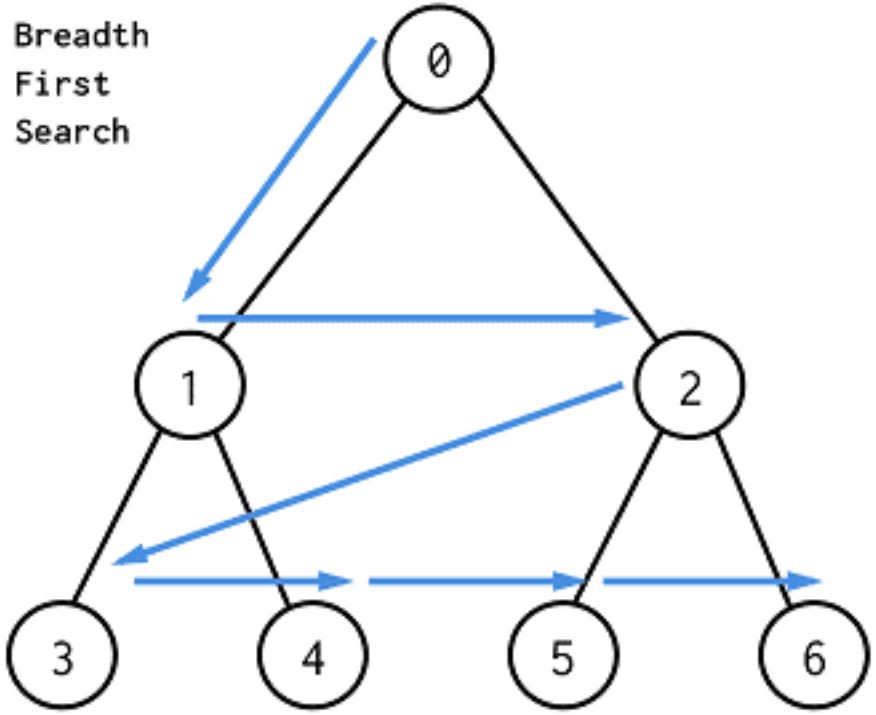
Graph search/traversal

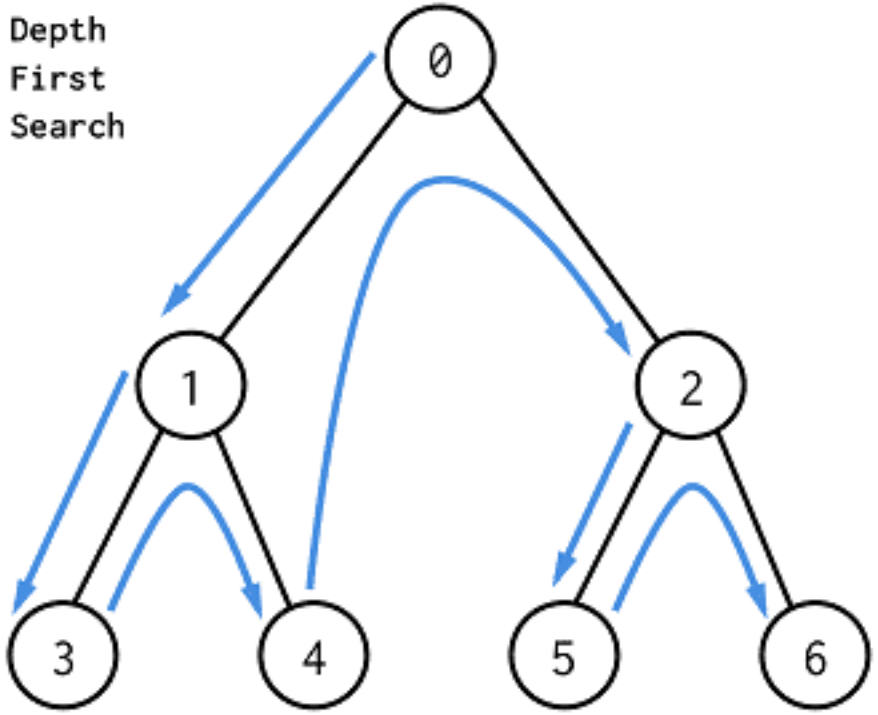
- Visit edges and nodes for general discovery or explicit search
- Common in graph algorithms





Basic Graph Search Algorithms





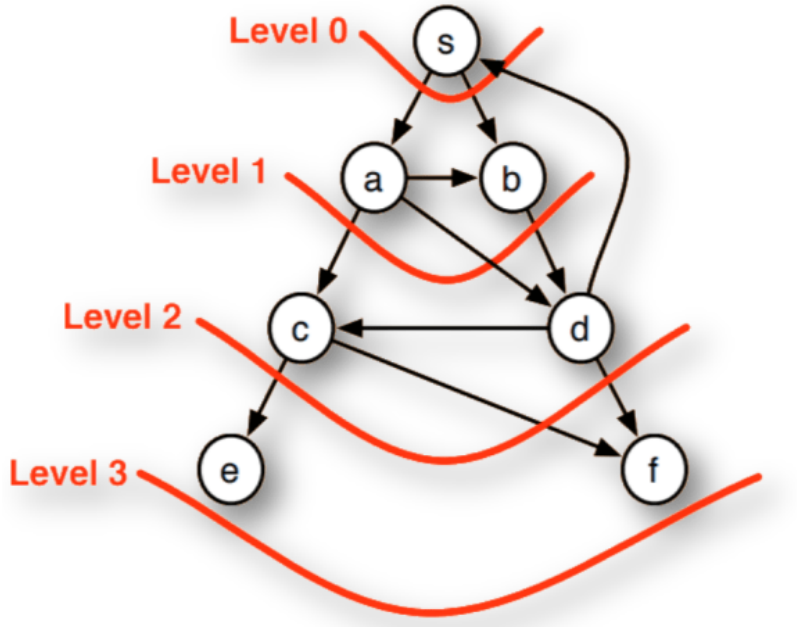
Breadth-First Search

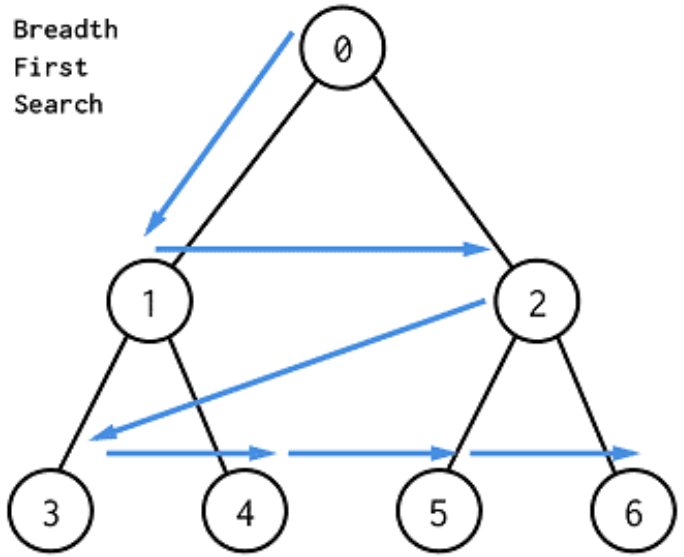
Breadth-First Search

Search a graph data structure for a node

- Starts at the root of the graph
- Breadth-first: finish visiting all nodes you can reach now before moving to the next level

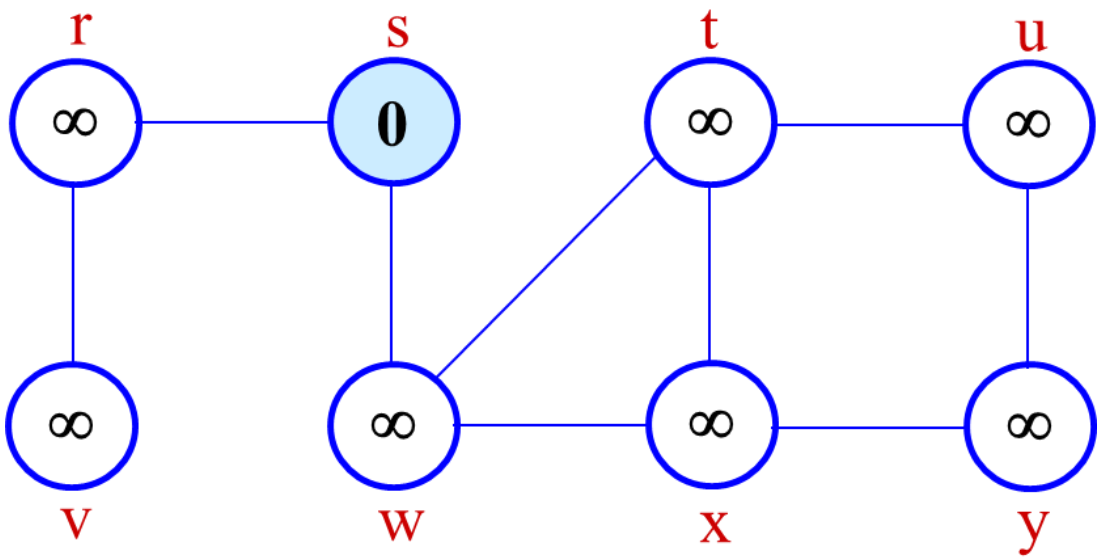
Example: grab the general ideas of various knowledge before going deep into any of them





Breadth-First Search

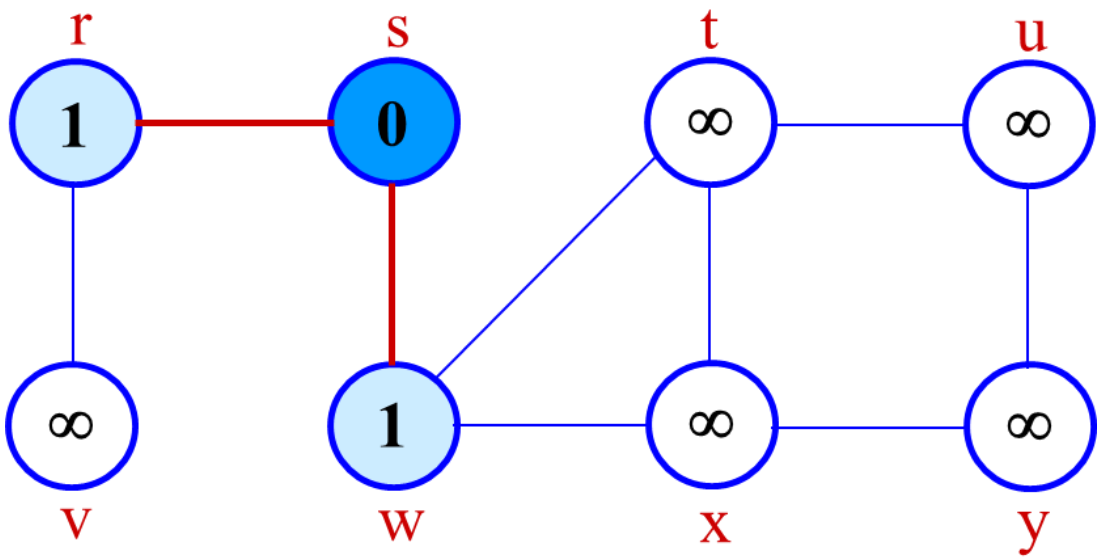
- Track of the nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	s
	0

Breadth-First Search

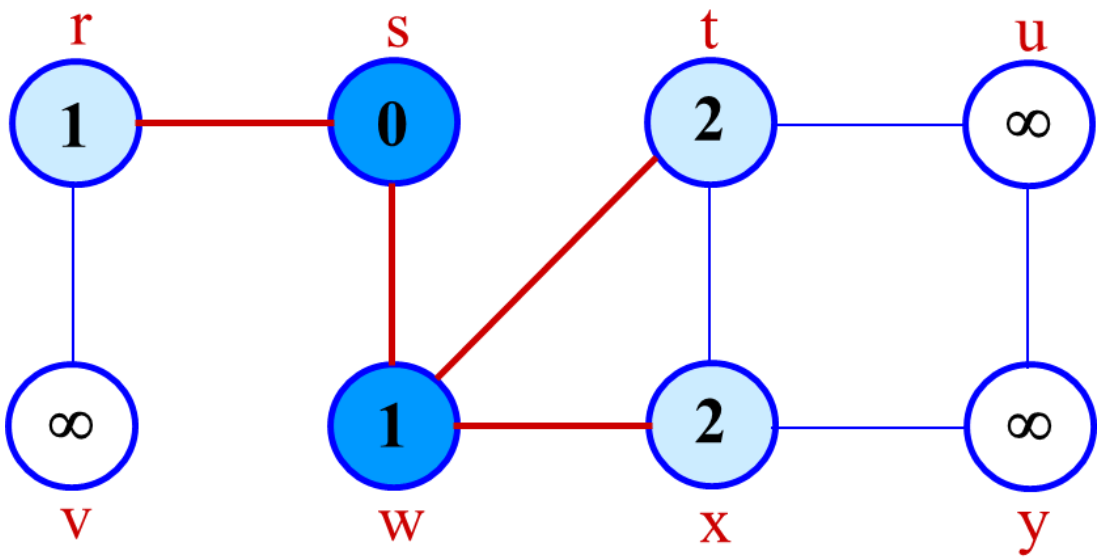
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	w	r
	1	1

Breadth-First Search

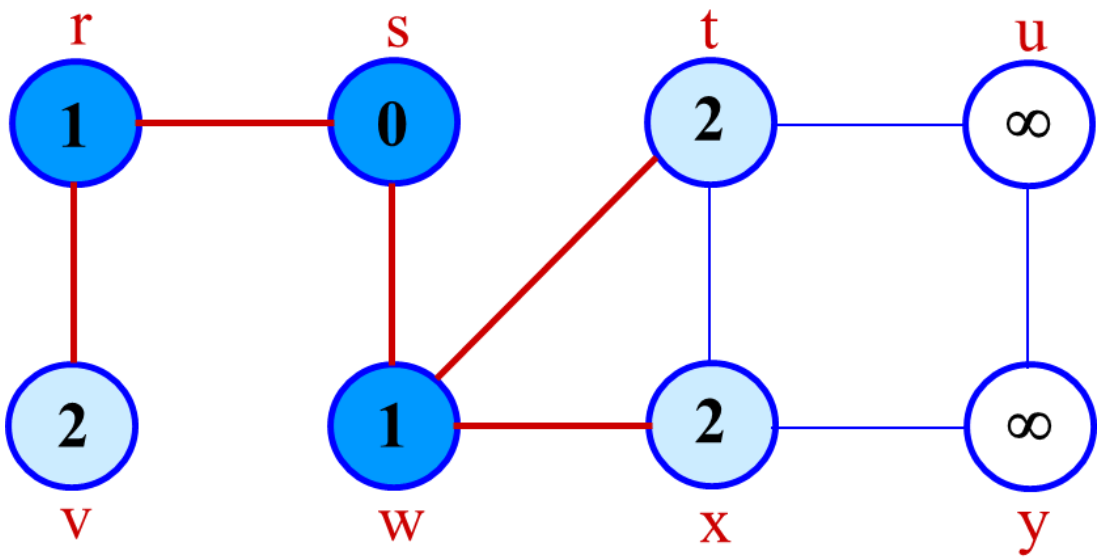
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	r	t	x
	1	2	2

Breadth-First Search

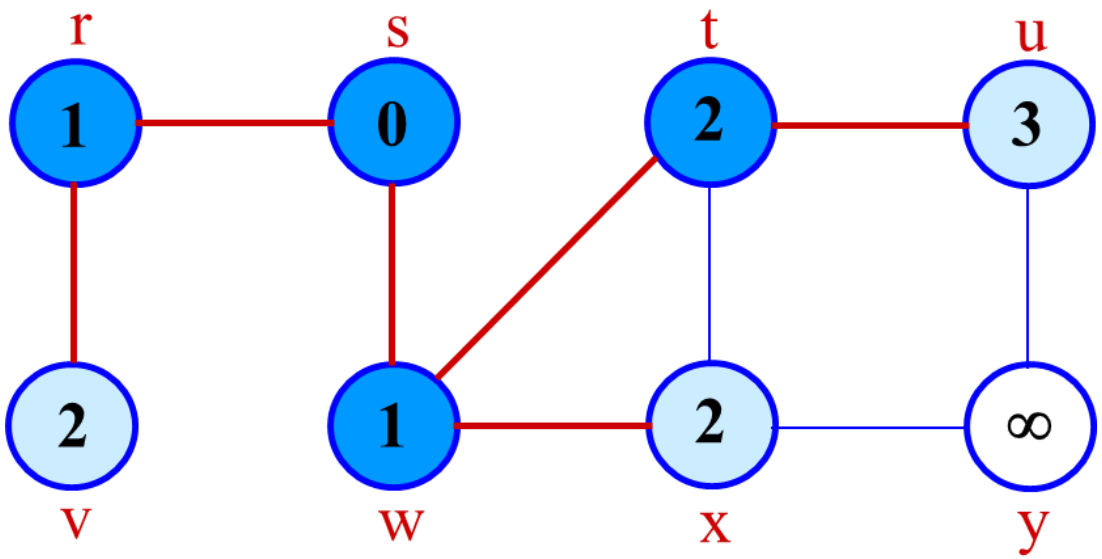
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	t	x	v
	2	2	2

Breadth-First Search

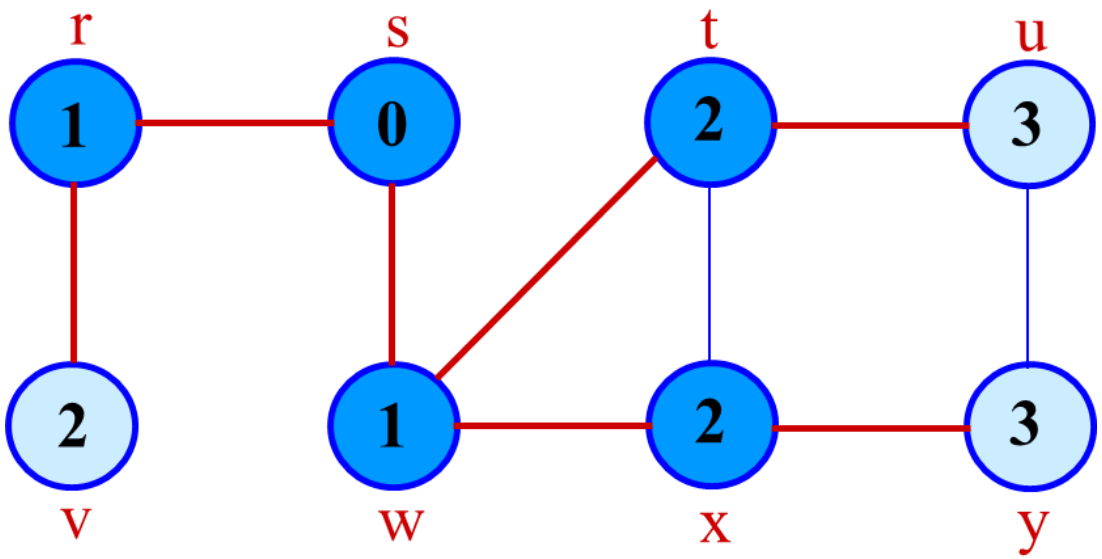
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	x	v	u
	2	2	3

Breadth-First Search

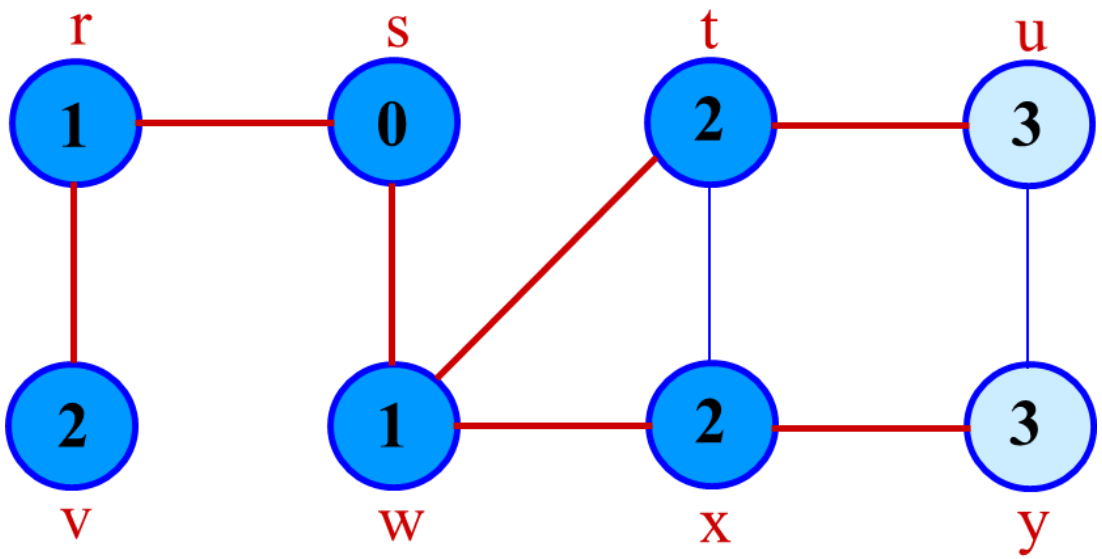
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	v	u	y
	2	3	3

Breadth-First Search

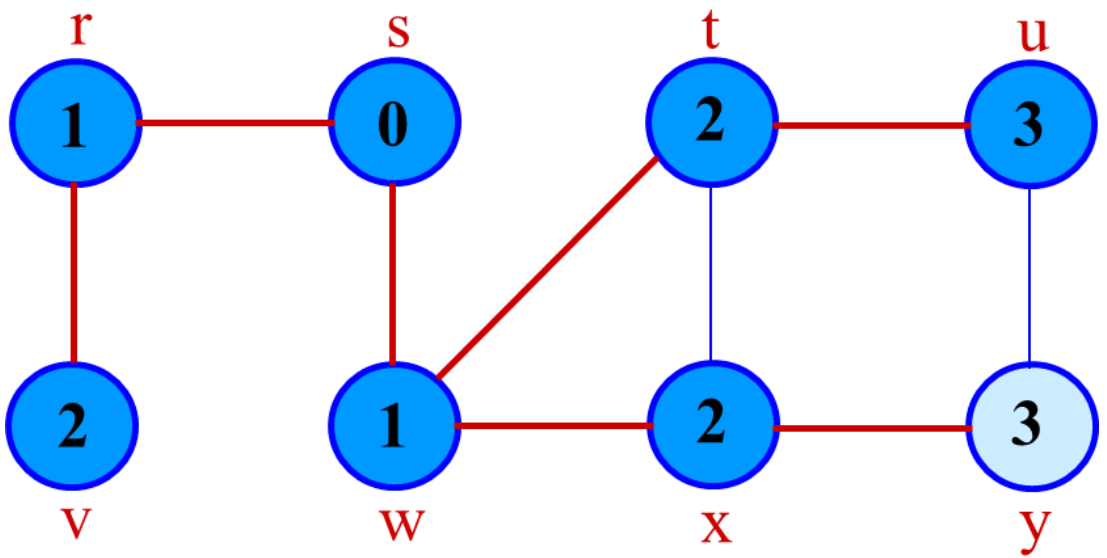
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	u	y
	3	3

Breadth-First Search

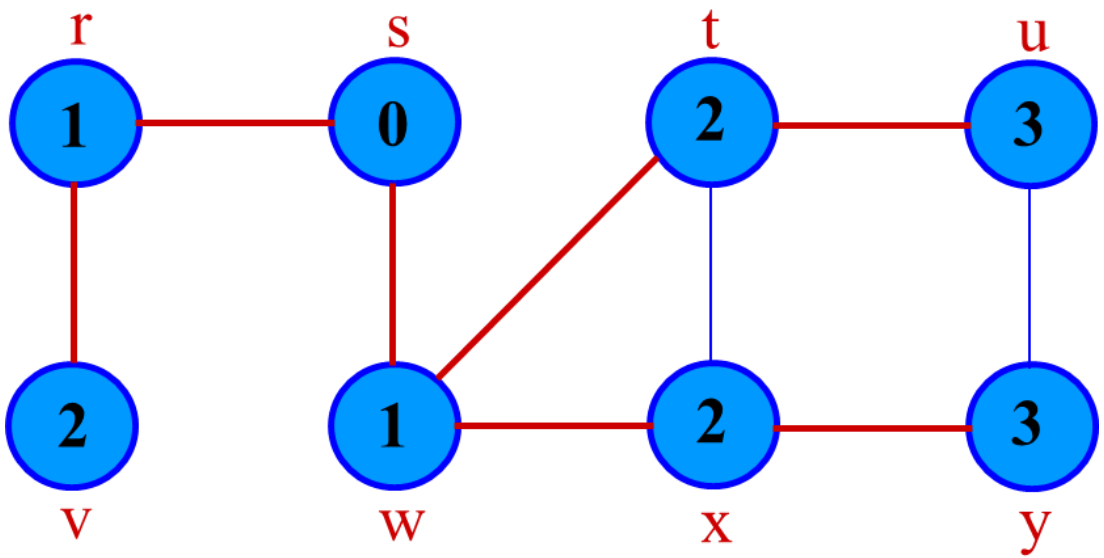
- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped



Q:	y
	3

Breadth-First Search

- Track of the child nodes that were encountered but not yet explored with a queue
- Nodes that has been visited before are skipped

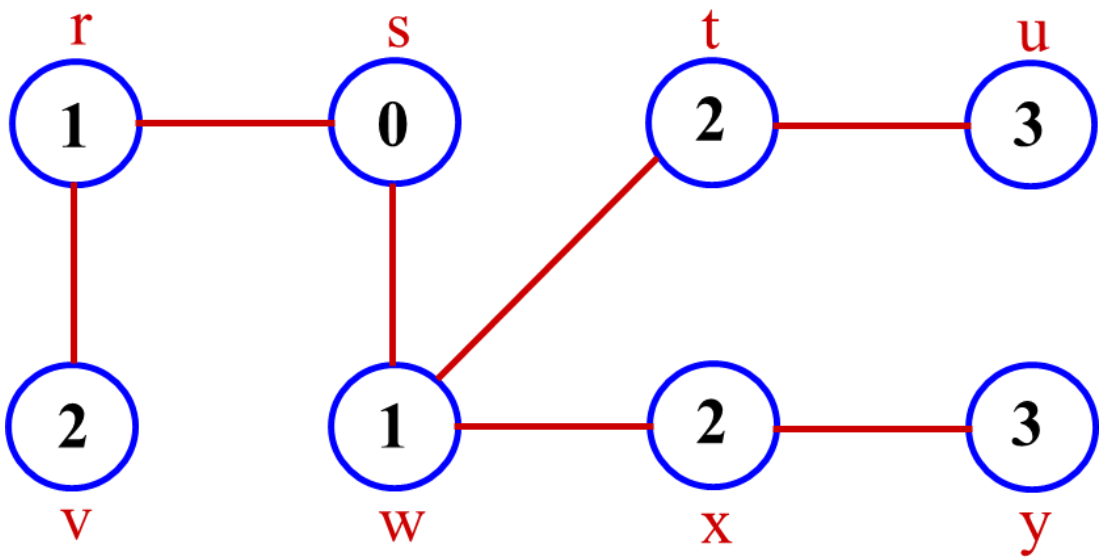


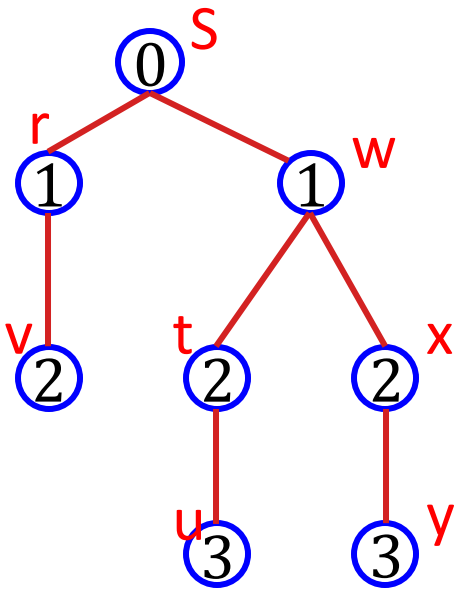
Q: ∅

BFS Tree

An edge (u, v) is a tree edge if v is first discovered when traversing edge (u, v)

Tree edges define the BFS tree, contains all reachable vertices





BF Tree

Breadth-First Search

Notations

- $v.d$: the depth of v in the bfs tree
- $v.\pi$: the parent node of v in the bfs tree

Time Complexity

- **Initialization:** $O(V)$
- **Traversal Loop**
  - Vertex: enqueued and dequeued at most once, $O(V)$
  - Edge: Edges from each node traversed once, $O(E)$
- **Total:** $\boldsymbol{O(V+E)}$

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Breadth-First Search

Theorem (Correctness of breadth-first search)

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. During its execution, (1) BFS discovers every vertex $v \in V$ that is reachable from the source s, and (2) upon termination, $v.d = \delta(s, v)$. Moreover, (3) for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

Notations

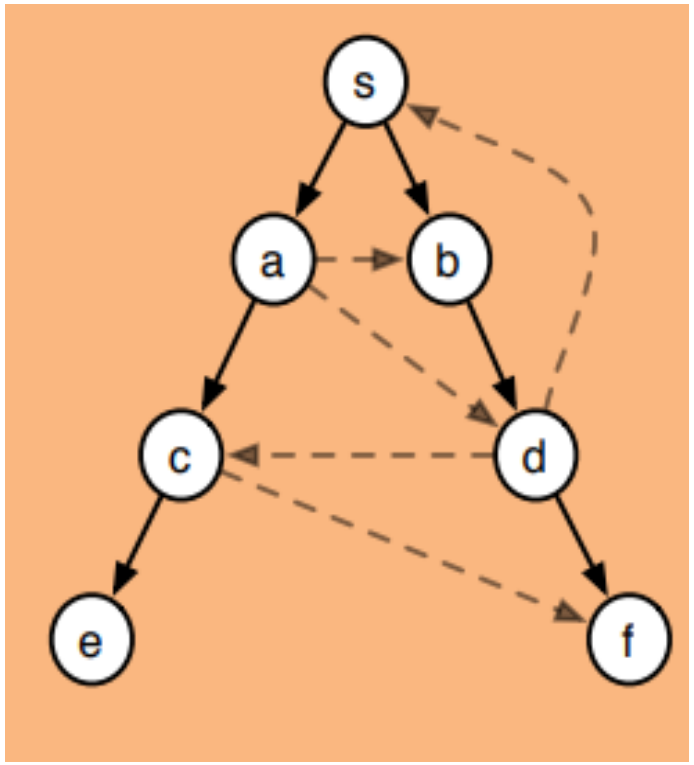
- $\delta(s, v)$: the shortest path distance from s to v (minimum number of edges)
  - $\delta(s, v) = \infty$ if no path exists

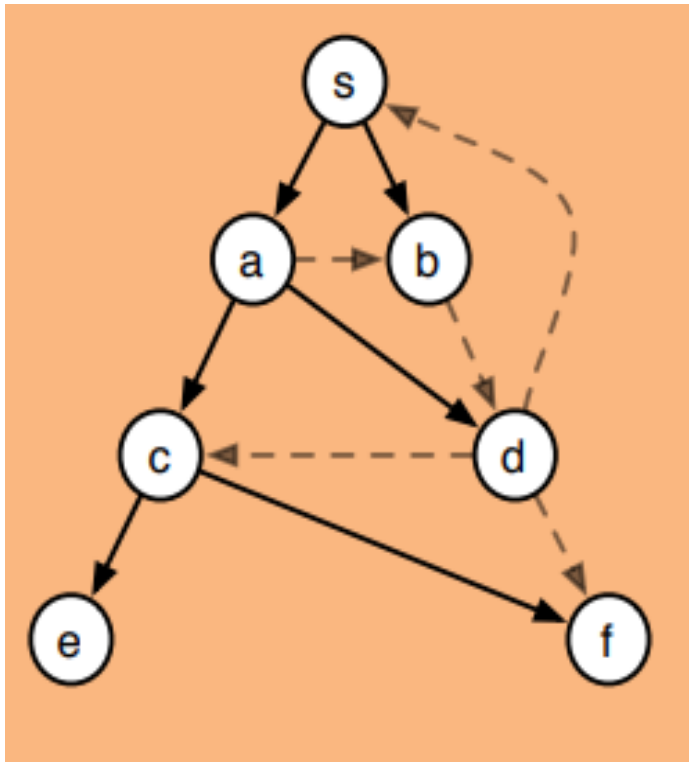
Proof: mathematical induction, $\delta(s, v) = 1$ then $v.d = 1$, $\delta(s, v) = 2$ then $v.d = 2$, …

Breadth-First Search

Each path from source to a vertex is the shortest path between them on the graph

Example: an undirected graph and two possible BFS trees with distances from s





Graph Search: Example

Given an array of non-negative integers arr, you are initially positioned at start index of the array. When you are at index i, you can jump to i + arr[i] or i - arr[i], check if you can reach to any index with value 0.

Notice that you can not jump outside of the array at any time.

Example:

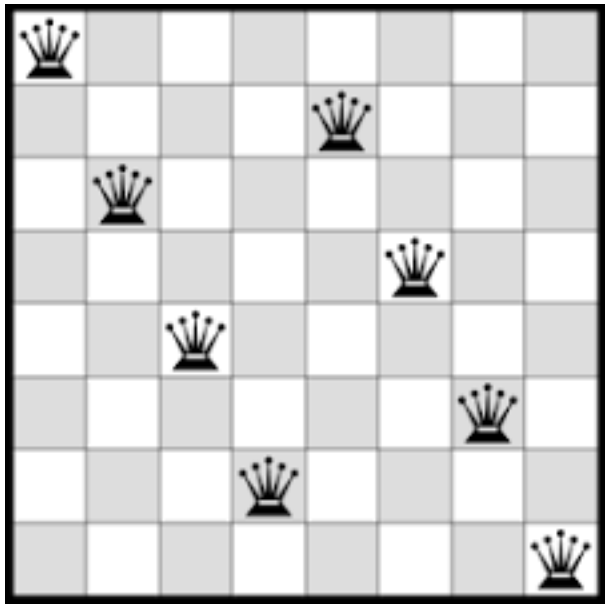
Input: arr = [4,2,3,0,3,1,2], start = 5

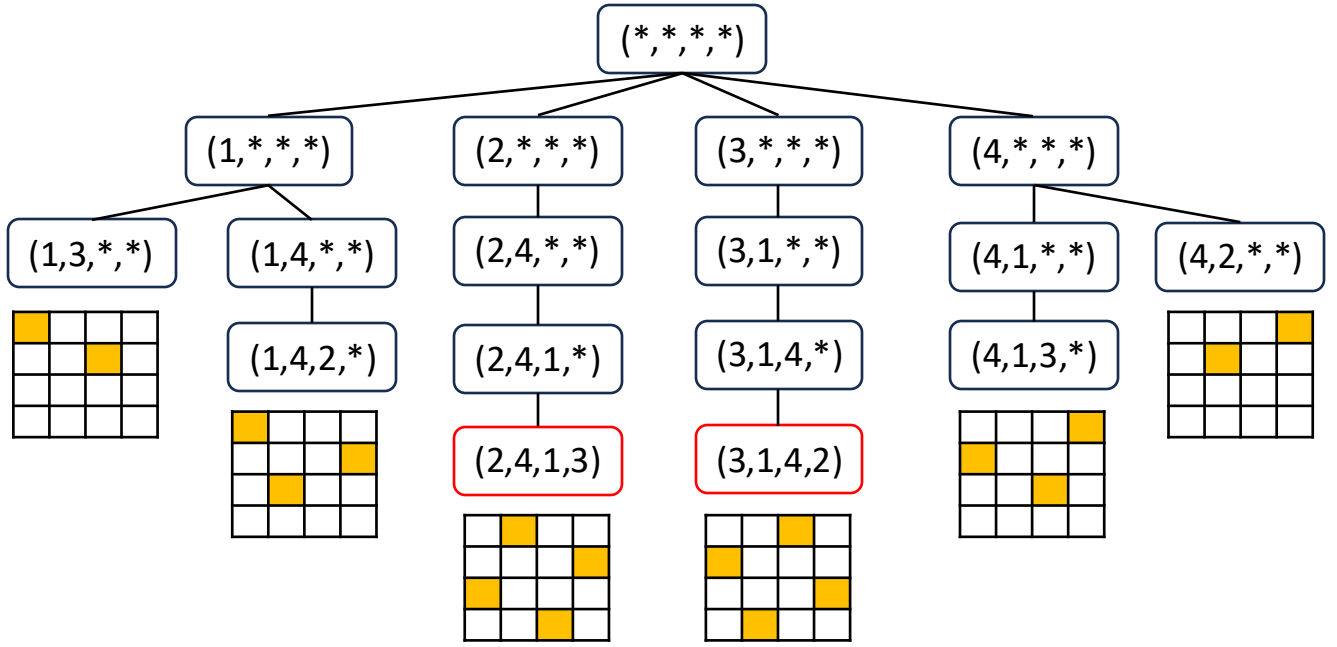
Output: true

Explanation: All possible ways to reach at index 3 with value 0 are: index 5 -> index 4 -> index 1 -> index 3 index 5 -> index 6 -> index 4 -> index 1 -> index 3

Eight Queens Problem?

- Find all the ways to place eight chess queens on an 8x8 chessboard so that no two queens threaten each other (cannot share the same row, column, or diagonal)
- First posed in the mid-19th century





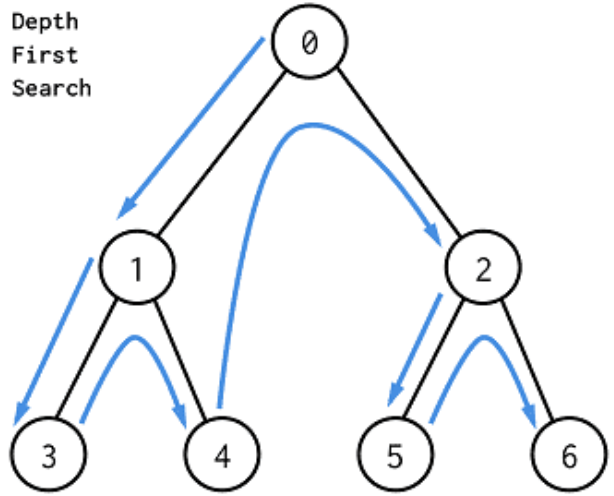
Depth-First Search

Depth-First Search

Another strategy for exploring a graph, searches "**deeper**" in the graph whenever possible

- Explore the first unexplored edge from the most recently discovered vertex v
- When all edges of v have been explored, "backtracks" to the precedent vertex
- Repeat until all vertices reachable from the source are discovered

Exactly backtracking, but on an explicitly defined graph



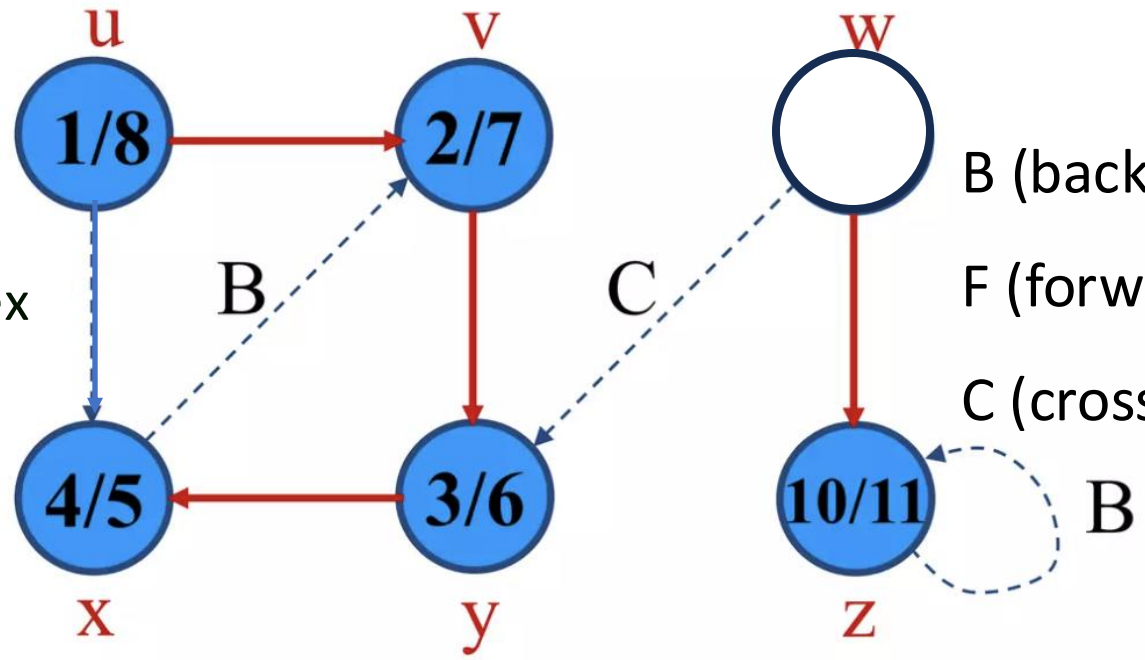
Depth-First Search

Another strategy for exploring a graph, searches "**deeper**" in the graph whenever possible

- Explore the first unexplored edge from the most recently discovered vertex v
- When all edges of v have been explored, "backtracks" to the precedent vertex
- Repeat until all vertices reachable from the source are discovered

DFS numbers:

Timestamps when the vertex is *discovered* and *finished*



B (back edges): to ancestor

F (forward edges): to descendant

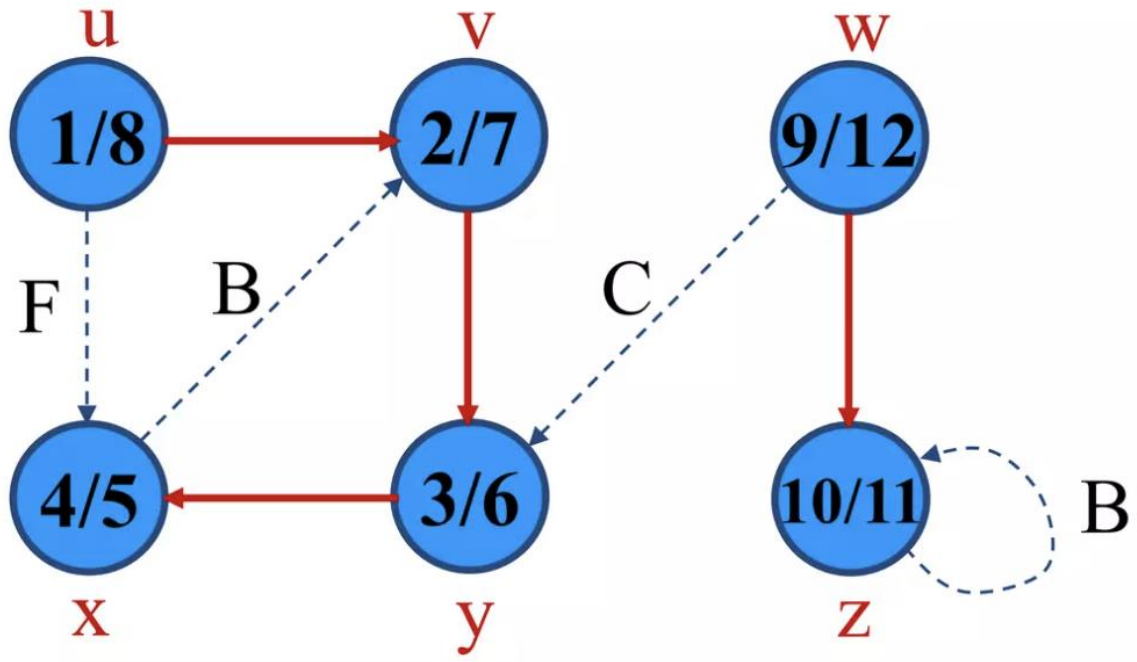
C (cross edges): other cases

Depth-First Search

Another strategy for exploring a graph, searches "**deeper**" in the graph whenever possible

- Explore the first unexplored edge from the most recently discovered vertex v
- When all edges of v have been explored, "backtracks" to the precedent vertex
- Repeat until all vertices reachable from the source are discovered

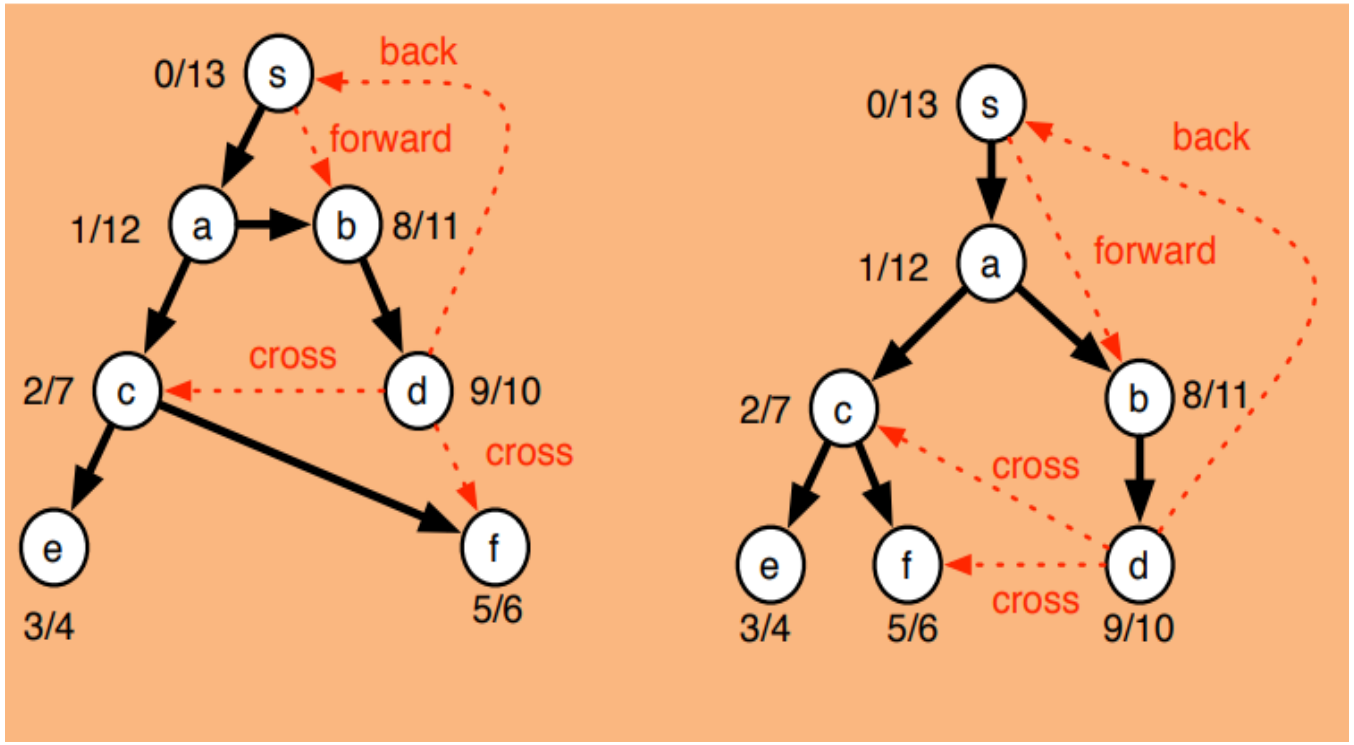
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

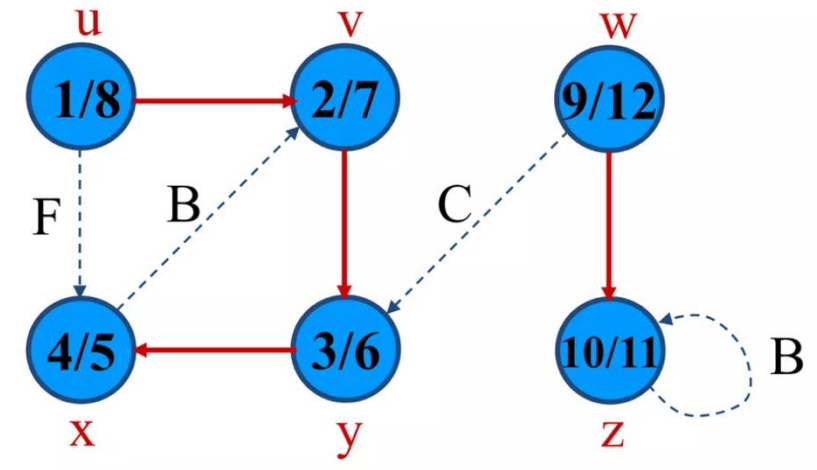



```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

DFS Tree

- An edge (u, v) is a tree edge if v is first discovered when traversing edge (u, v)
- Tree edges define the DFS tree, contains all reachable vertices





B (back edges): point to ancestor

F (forward edges): point to descendant

C (cross edges): other cases

Time Complexity

- Each vertex is discovered and finished once: $O(V)$
- Each edge is explored once: $O(E)$

Total complexity: $O(V + E)$

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1   time = time + 1
2   u.d = time
3   u.color = GRAY
4   for each v ∈ G.Adj[u]
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   u.color = BLACK
9   time = time + 1
10  u.f = time
```

Parenthesis Theorem

Theorem 22.7 (Parenthesis Theorem)

In a DFS forest for $G = (V, E)$, for any u, v *pair*, exactly one of the following holds:

1. $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint and neither u nor v is a descendant of the other.
2. $u.d < v.d < v.f < u.f$ and v is a descendant of u.
3. $v.d < u.d < u.f < v.f$ and u is a descendant of v.

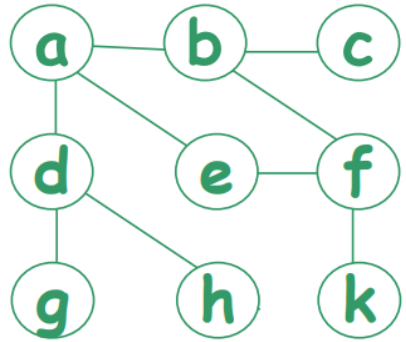
u.d < v. d < u. f < v. f cannot happen

- Like parentheses:
  - OK: ( ) [ ] ([ ]) [( )]; Not OK: ([ )] [( ])

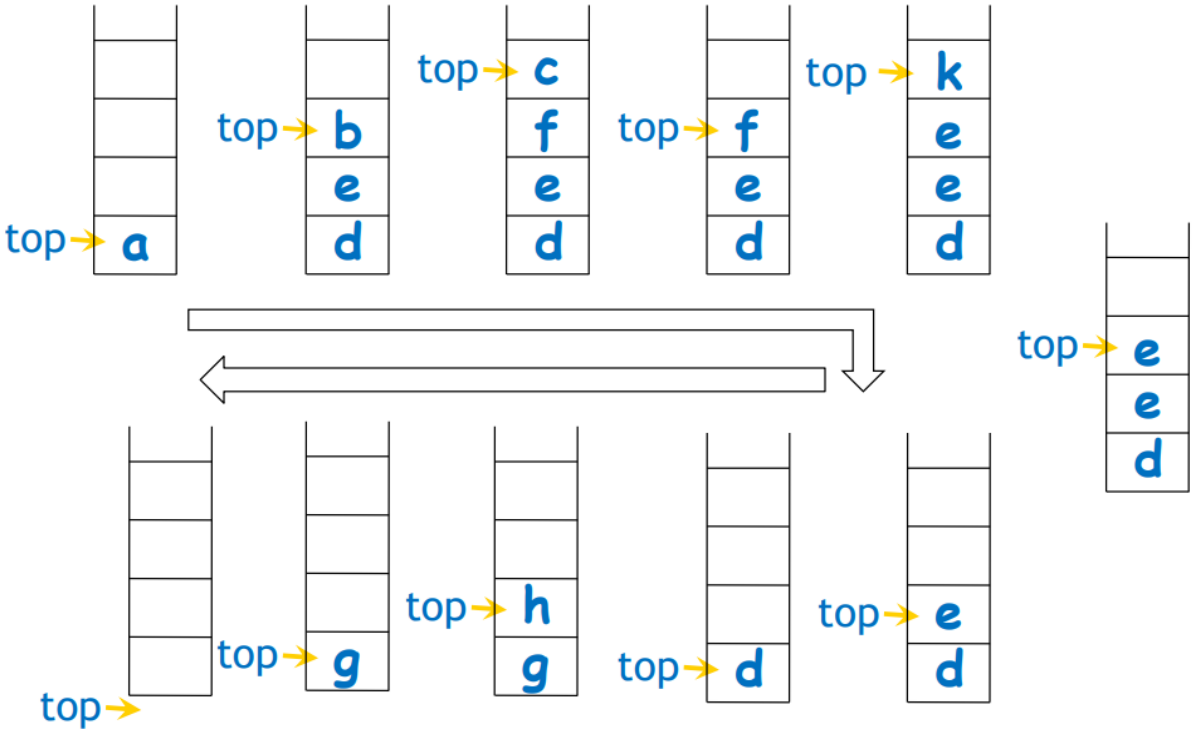
Corollary (Nesting of descendants' intervals): v is a descendant of u i.f.f $u.d < v.d < v.f < u.f$

DFS from a Stack View

- BFS explore nodes following FIFO order (Queue)
- DFS explore nodes following LIFO order (Stack)

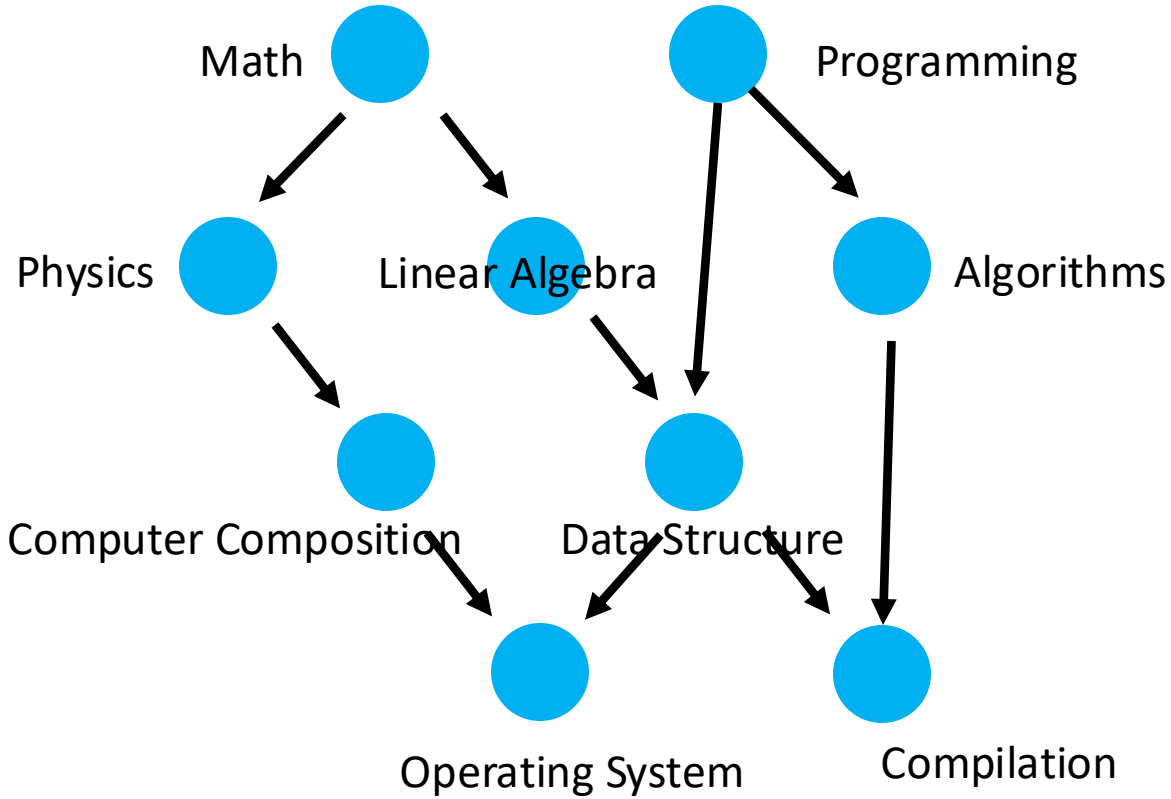


a, b, c, f, k, e, d, h, g



Topological Sort with DFS?

Given a directed acyclic graph $G =< V, E >$. Find an order $O =< o_1, o_2, \ldots, o_V >$ satisfying there does not exist pair $< i, j >$ that $i < j$ and $< o_j, o_i > \in E$

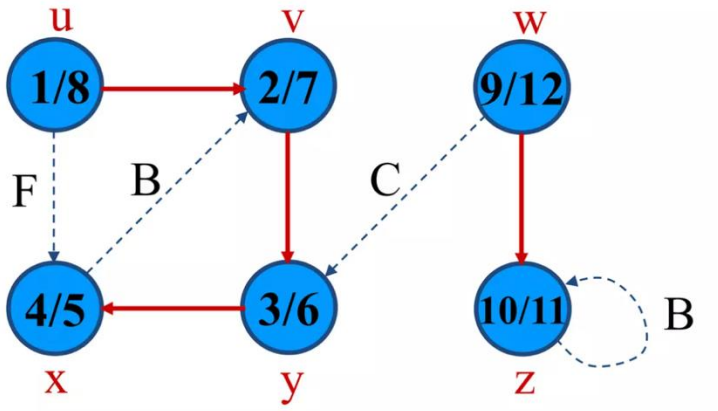


Topological Sort with DFS

Topological sort can be obtained by sorting v with decreasing order of $v.f$

Proof:

- In dfs, each vertex cannot finish before all the nodes reachable from it are finished

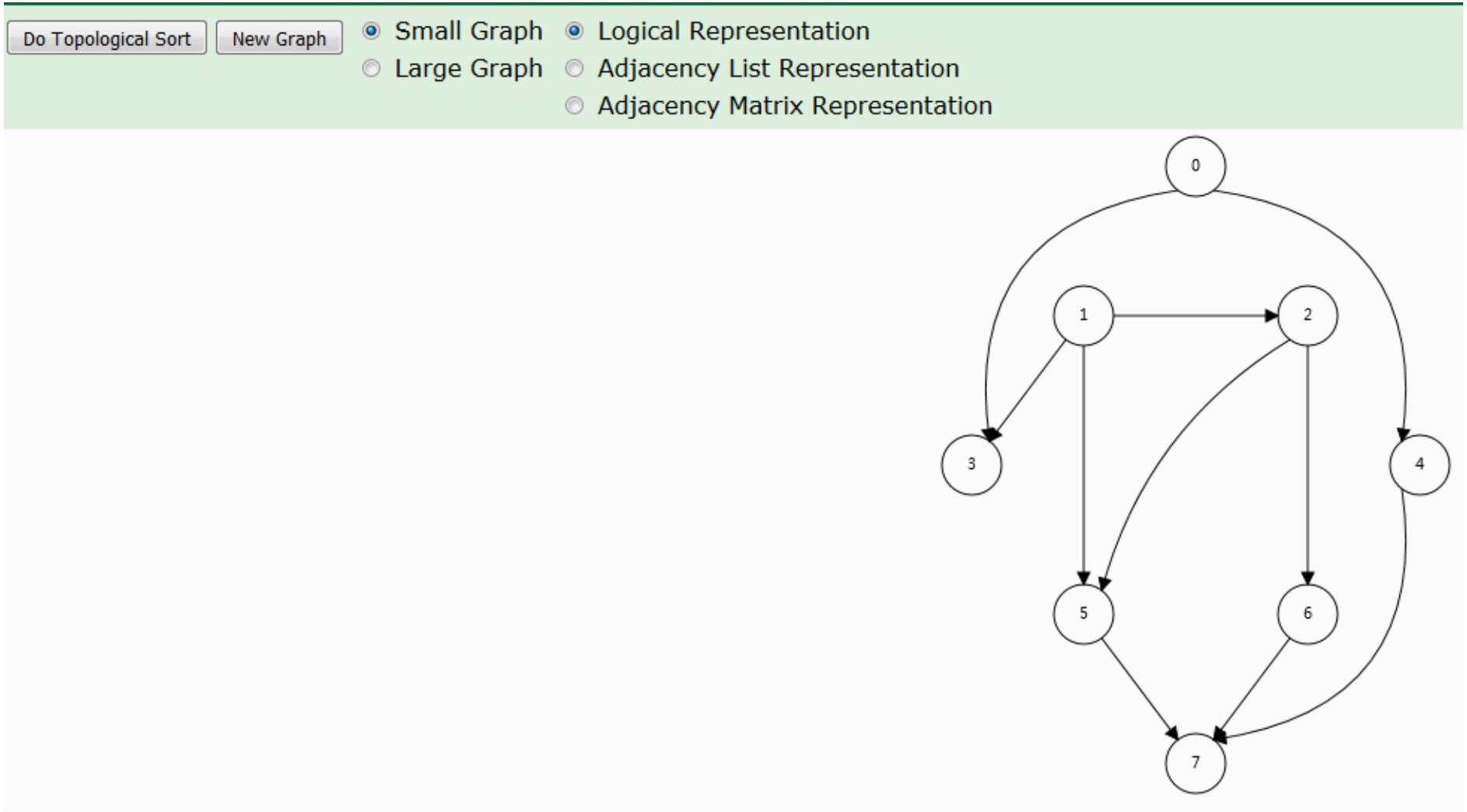


$$w \to z \to u \to v \to y \to x$$

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Topological Sort with DFS



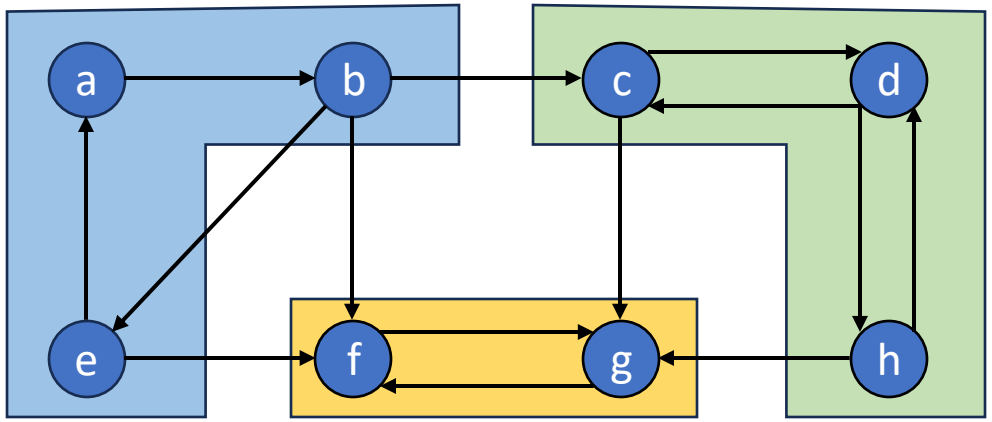
Strongly Connected Components

Strongly Connected Components

Problem: Given a directed graph $G = (V, E)$, find the *strongly connected components.*

Definitions:

- *Strongly connected subgraph:* every vertex is reachable from every other vertex
- *Strongly connected component*: a maximized (极大) strongly connected subgraph
  - Cannot add more nodes to the subgraph and make it still strongly connected
  - Forms a partition of G into strongly connected subgraphs



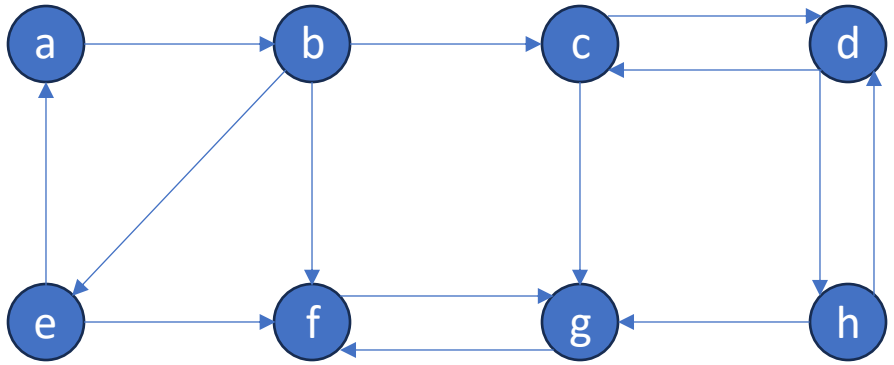
Strongly Connected Components

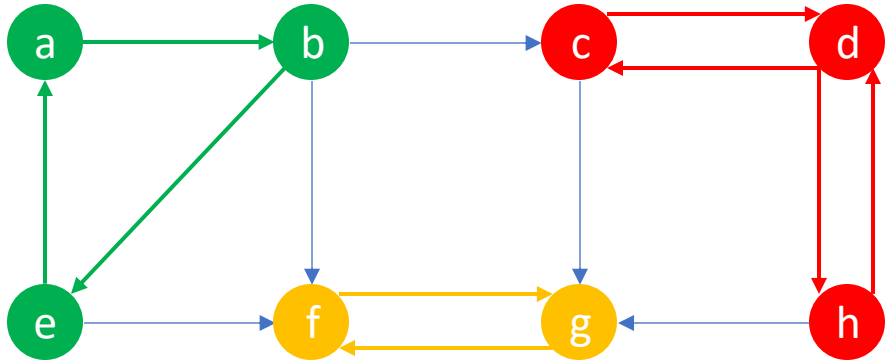
Problem: given a directed graph $G = (V, E)$, find the *strongly connected components.*

Directly adopt DFS?

- *DFS(g)*
- ~~*DFS(f)*~~
- *DFS(h)*
- *DFS(a)*
- ~~*DFS(d)*~~
- ~~*DFS(c)*~~
- ~~*DFS(b)*~~
- ~~*DFS(e)*~~

Seems correct, always correct?





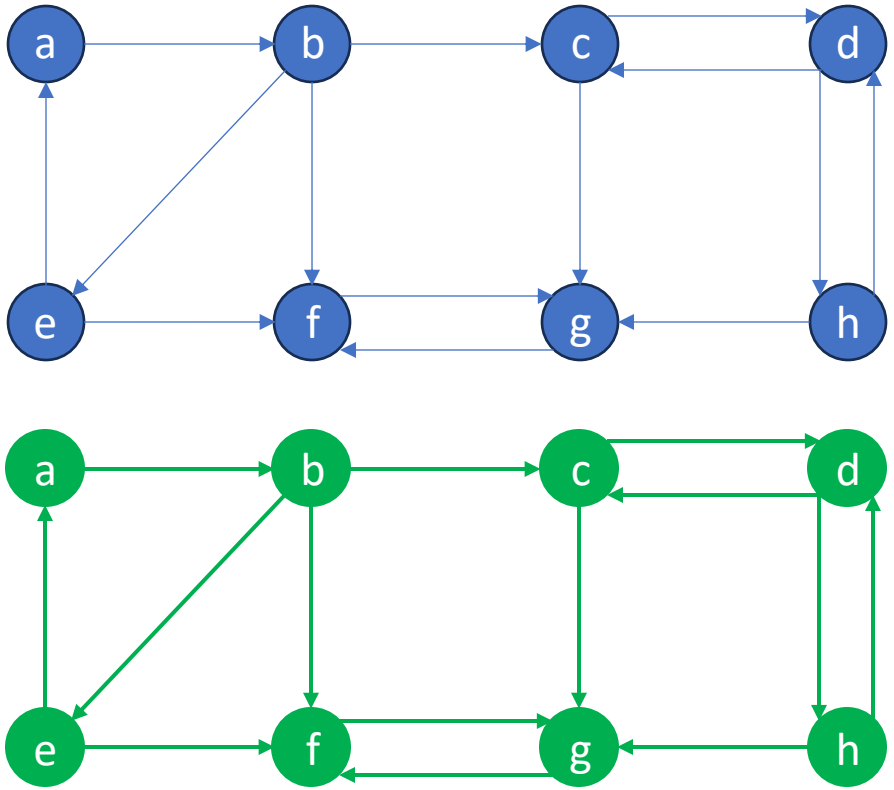
Strongly Connected Components

Problem: given a directed graph $G = (V, E)$, find the *strongly connected components.*

Another order of DFS

- *DFS(a)*
- ~~*DFS(f)*~~
- ~~*DFS(h)*~~
- ~~*DFS(a)*~~
- ~~*DFS(d)*~~
- ~~*DFS(c)*~~
- ~~*DFS(b)*~~
- ~~*DFS(e)*~~

The order matters!



Strongly Connected Components

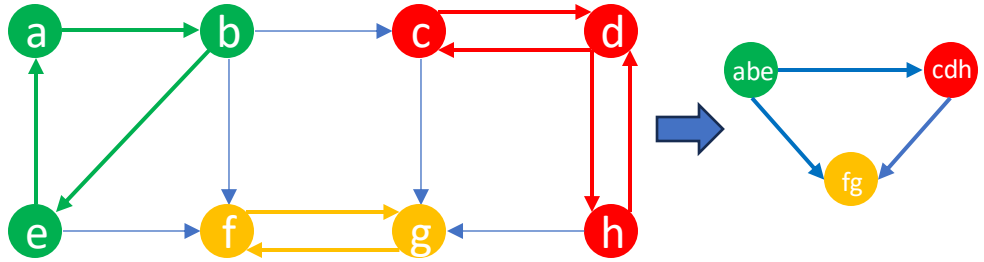
Observe each components as a whole and decide the order

Definition (component graph): Given a G with strongly connected components $C_1, C_2, \dots, C_k$, its component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ represents the connection between components. Specifically, $V^{SCC} = \{v_1, v_2, \dots, v_k\}$ and v_i represents C_i. There is an edge $(v_i, v_j) \in E^{SCC}$ if there is $< x \in C_i, y \in C_j > \in E$

Key property: a component graph is a Directed Acyclic Graph

- Proof: obviously, a cycle will make all nodes in the two components reach each other. These two components are no longer strongly connected maximized subgraphs

Intuition: Decide the order of DFS with topological sort on the component graph



Strongly Connected Components

Notation (for a set of vertices $U \subseteq V$): $d(U) = \min_{u \in U} u.d$ and $f(U) = \max_{u \in U} u.f$.

Lemma: Let C_i and C_j be distinct strongly connected components in directed graph $G = (V, E)$. If there is an edge $< u \in C_i, v \in C_j > \in E$. Then $f(C_i) > f(C_j)$.

Proof:

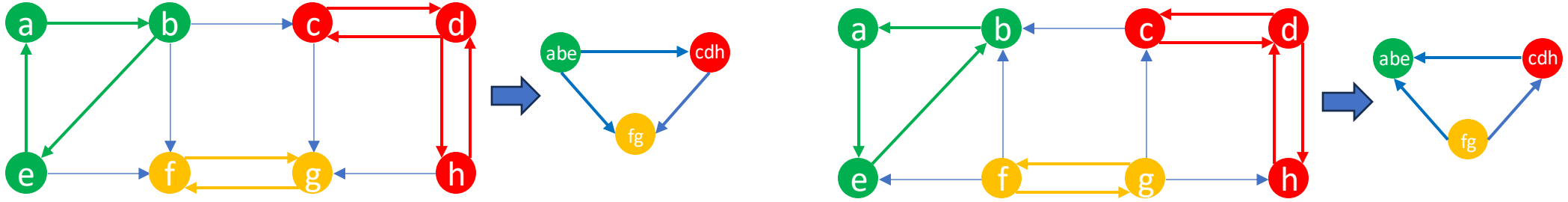
- If $d(C_i) < d(C_j)$:C_i is first discovered in dfs, meaning nodes in C_j are descendants of a node in C_i (denoted as x, possibly u). Since descendants finish earlier than ancestors, $f(C_j) = \max_{y \in C_i} y.f < x.f \leq f(C_i)$.
- If $d(C_i) > d(C_j)$: since C_i and C_j are distinct strongly connected components, with edge $< u, v >$ pointing from C_i to C_j, there cannot be a path pointing from C_j to C_i. Therefore, nodes in C_i are not reachable during DFS in C_j. i.e., C_j finishes before C_i is discovered. $f(C_j) < d(C_i) < f(C_i)$

Sort components in the finish time decreasing order produces a topological sort

Strongly Connected Components

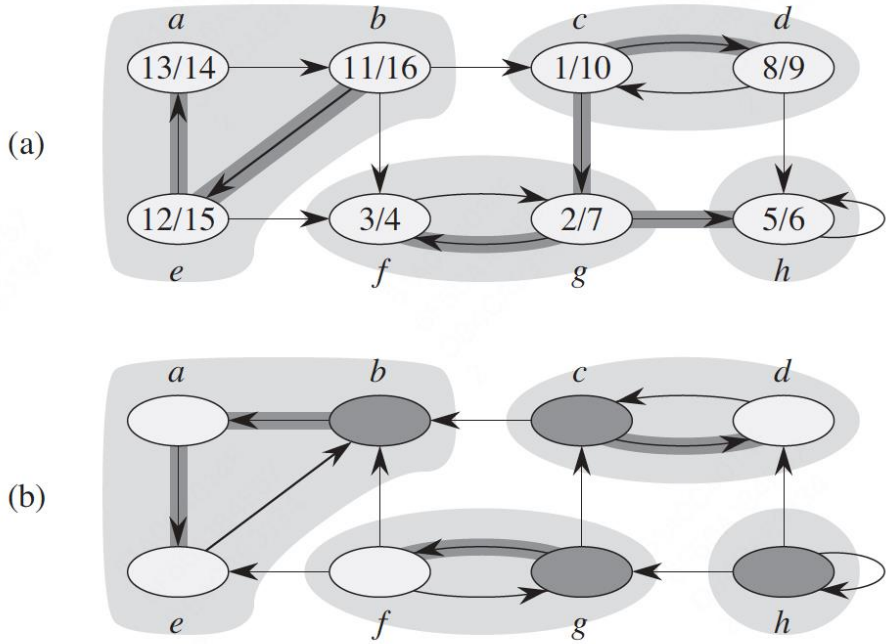
Sort components in the finish time decreasing order to produce a topological sort

- Problem: composition nodes of each component are unknown to us
- Observation: the last finished node belongs to the last finished component
  - Find and delete the nodes in this component.
  - The next last finished nodes help finding the second last finished component
- Still a problem: the last ended component can reach other components
- Solution: reverse all the edges!

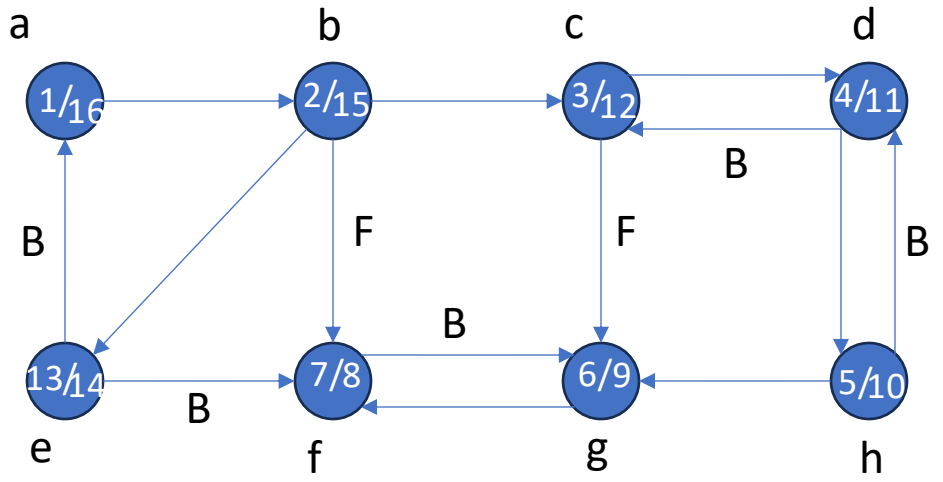


Kosaraju Algorithm

1. DFS in any order to find the finishing time of all the nodes
2. Transpose the graph
3. DFS in finishing time decreasing order



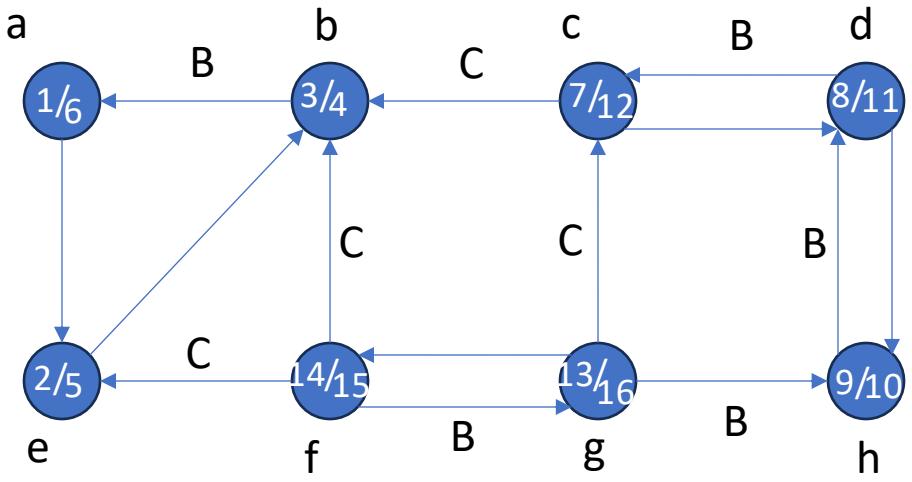
Kosaraju Algorithm: Example



Finishing time decreasing order: a b e c d h g f

Kosaraju Algorithm: Example

Transpose the graph



Finishing time decreasing order: a b e c d h g f

Component 1: a b e

Component 2: c d h

Component 3: g h

Thank you!

AIAA 5037 Advanced Algorithms and Data Structures