

Graph techniques

David Gilbert

Bioinformatics Research Centre

www.brc.dcs.gla.ac.uk

Department of Computing Science, University of
Glasgow

Graph techniques

- Graphs intro
- Definition
- Paths, circuits, searching
- Breadth-first search
- Depth-first search

What do these pathways have in common?

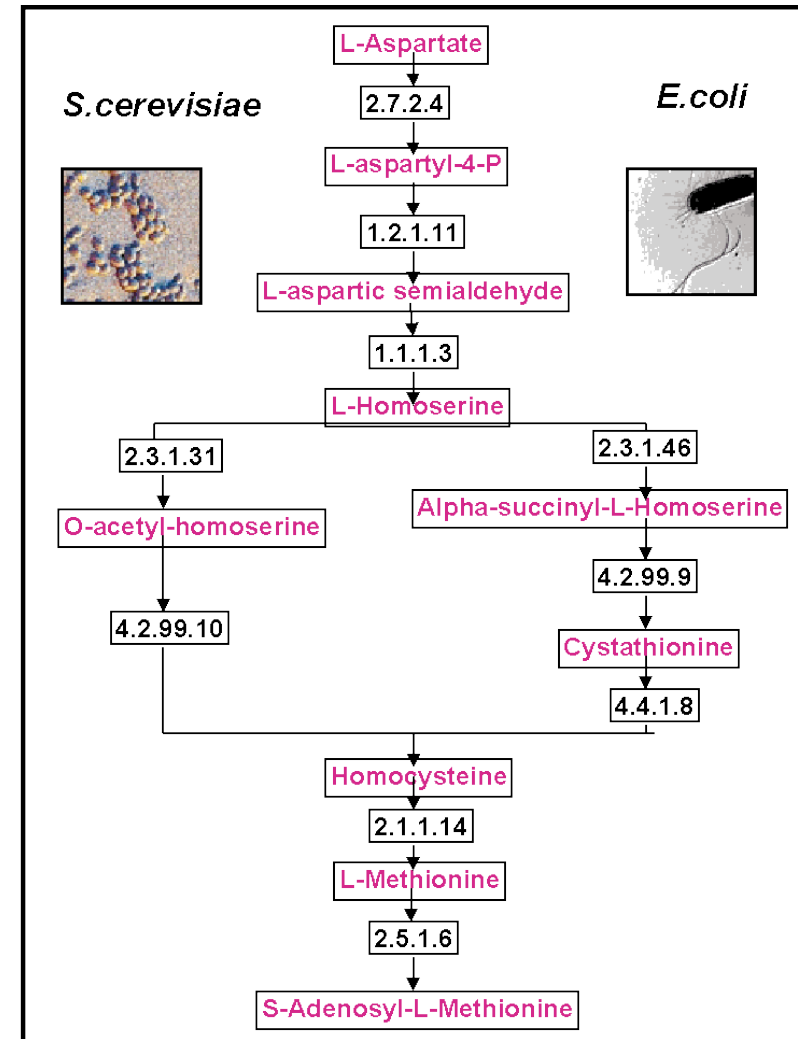
- They consist of objects connected by lines
 - The objects can be molecules, circles, boxes, reactions, other biochemical processes
 - The lines are usually arrows
 - Mathematically, they are graphs
- ⇒ Modelling pathways as directed graphs is the most straightforward approach

Pathway analysis

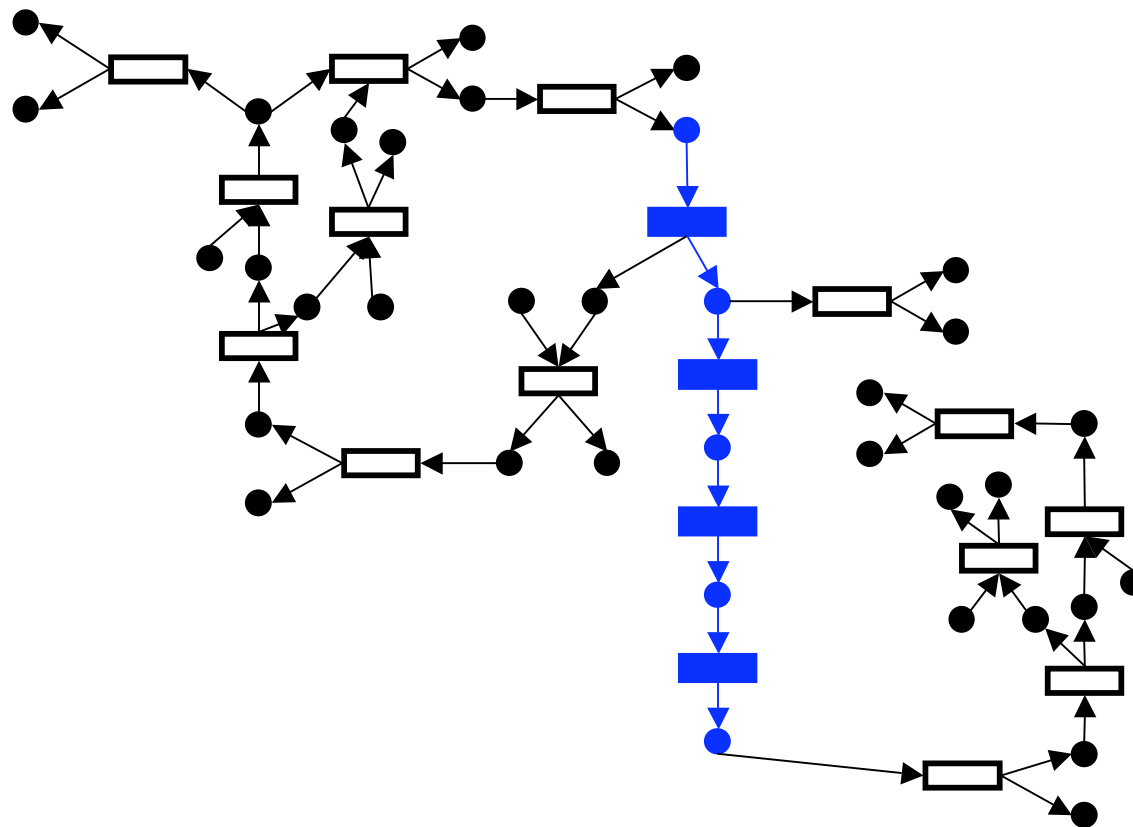
- What are the possible paths from entity A to entity B?
- How many paths, and of what lengths, lead from A to B?
- What is the average path distance between entities?
- Find all paths including a given set of entities
- Which genes are affected by a specific compound?
- Which pathways are affected if a given entity is missing or switched off?
- Compare pathways between two organisms or tissues, find common features or missing elements

Alternative Pathways

- Genome evolution
 - compare with known genome
 - infer to unknown genome
 - Missing enzymes
- Biotechnology
 - identification of alternative enzymes
 - identification of alternative pathways
 - identification of alternative substrates
 - identification of alternative products
- Pharmacology
 - non-homologous gene displacement
 - species-specific drug targets
- Identification of previously unknown genes



Reactions and compounds as graphs



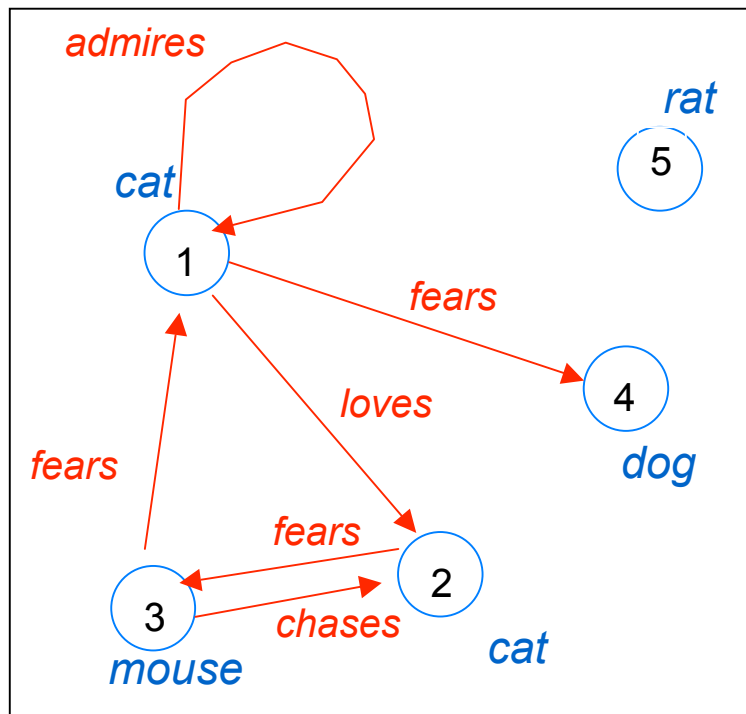
- 5,600 compounds (KEGG)
- ▭ 3,705 reactions (KEGG)
- 3,799 substrate → reaction (KEGG)
- 3,584 reaction → product (KEGG)

In *Escherichia coli*

- 4219 genes (KEGG)
- 850 enzymes (SwissAll)
- 159 pathways (EcoCyc)

Figure 6

Graph Theory (simple!)



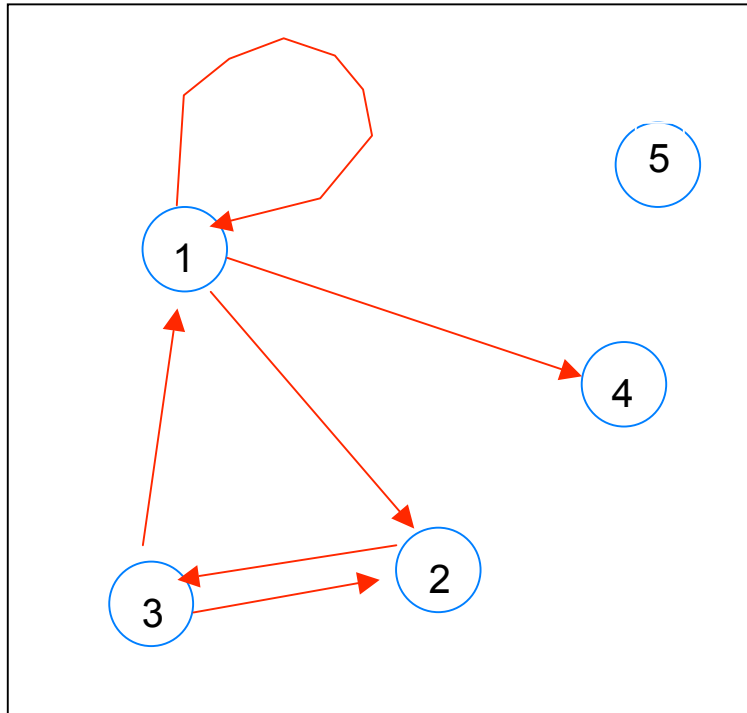
Graph = (V,A)

Vertices = { 1 , 2 , 3, 4 , 5 }

Arcs = { 1→2, 2→3, 3→2, 3→1, 1→4 , 1→1 }

Optionally label vertices & arcs

Example



?Maximal
paths?

Circuits

$$C1 = (1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1)$$

$$\text{len}(C1) = 3$$

$$C2 = (1 \rightarrow 1)$$

$$\text{len}(C2) = 1$$

Paths (some)

$$P1 = (2 \rightarrow 3, 3 \rightarrow 1)$$

$$P1 = (2 \rightarrow 3, 3 \rightarrow 1, 1 \rightarrow 4)$$

$$P3 = (2 \rightarrow 3, 3 \rightarrow 1, 1 \rightarrow 1)$$

Graphs

- A graph $G = (V, E)$
 - V = set of vertices, E = set of edges
 - *Dense* graph: $|E| \approx |V|^2$; *Sparse* graph: $|E| \approx |V|$
 - *Undirected graph*:
 - Edge (u,v) = edge (v,u)
 - No self-loops
 - *Directed graph*:
 - Edge (u,v) goes from vertex u to vertex v , notated $u \rightarrow v$
 - A *weighted graph* associates weights with either the edges or the vertices

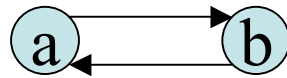
Terminology

- *Subgraph* G' of graph G if $G = (V, A)$, $G' = (V', A')$ and $V' \subset V$ and $A' = \{x \rightarrow y \mid x \rightarrow y \in A \text{ and } x, y \in V'\}$
- For arc $x \rightarrow y$ then x is its *initial* vertex and y the *terminal* vertex.
- Two vertices x, y *adjacent* if $x \neq y$ and $x \rightarrow y$ or $y \rightarrow x$

Paths and Circuits of a Graph

- *Path* = sequence of arcs
 $(x_1 \rightarrow x_2, x_2 \rightarrow x_3, x_3 \rightarrow x_4, \dots, x_{k-1} \rightarrow x_k)$
- Also can write $[x_1, x_2, x_3, \dots, x_k]$
- *Simple* if does not use the same arc twice, else *composite*
- *Elementary* if does not use same vertex twice
- Can be *finite* or *infinite*
- *Circuit* = path $[x_1, x_2, x_3, \dots, x_k]$ where initial vertex x_1 = terminal vertex x_k
- *Elementary circuit* if all vertices distinct apart from $x_1 = x_k$
- *Length* of path $(x_1 \rightarrow x_2, \dots, x_{k-1} \rightarrow x_k)$ is $k-1$
- *Loop* is circuit length=1, i.e. $(x_1 \rightarrow x_1)$

Small test network 1



circuits

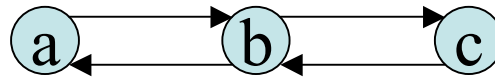
a -> b -> a

linear paths

a -> b

b -> a

Small test network 2



circuits

a -> b -> a

b -> c -> b

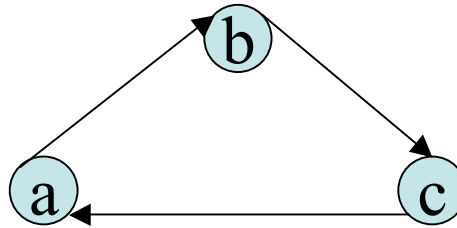
a -> b -> c -> b -> a

linear paths

a -> b -> c

c -> b -> a

Small test network 3



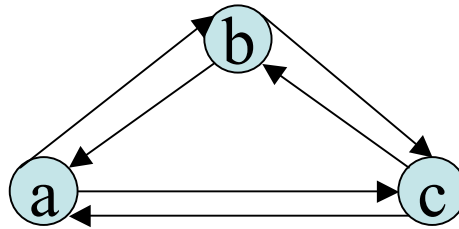
circuits

?

linear paths

?

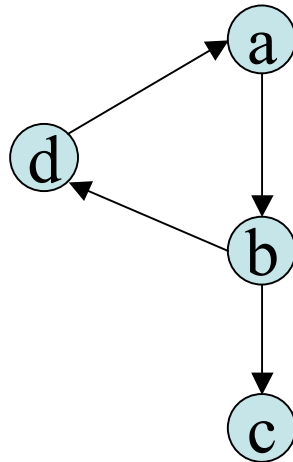
Small test network 4



circuits

linear paths

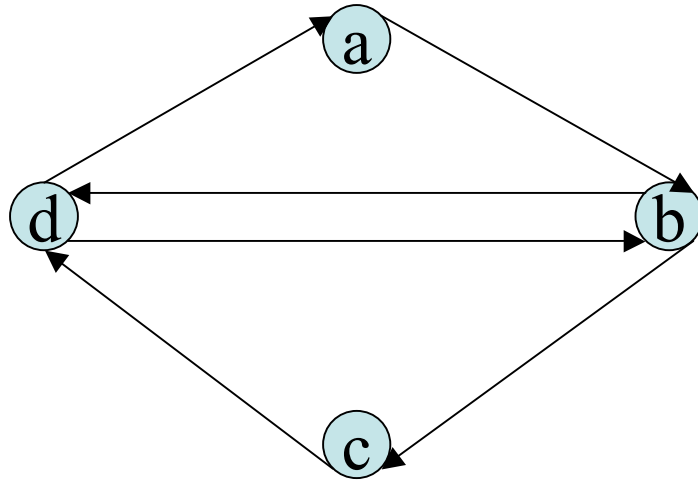
Small test network 5



circuits

linear paths

Small test network 11



linear paths

circuits

Representing Graphs

- Assume $V = \{1, 2, \dots, n\}$
- An *adjacency matrix* represents the graph as a $n \times n$ matrix A :
 - $A[i, j] = 1$ if edge $(i, j) \in E$ (or weight of edge)
 $= 0$ if edge $(i, j) \notin E$
 - Storage requirements: $O(V^2)$
 - A dense representation
 - But, can be very efficient for small graphs
 - Especially if store just one bit/edge
 - Undirected graph: only need one diagonal of matrix

Search strategies

- A search strategy is defined by picking the **order of node expansion**
- Strategies are evaluated along the following dimensions:
 - **completeness**: does it always find a solution if one exists?
 - **time complexity**: number of nodes generated
 - **space complexity**: maximum number of nodes in memory
 - **optimality**: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - *b*: maximum branching factor of the search tree
 - *d*: depth of the least-cost solution
 - *m*: maximum depth of the state space (may be ∞)

Graph Searching

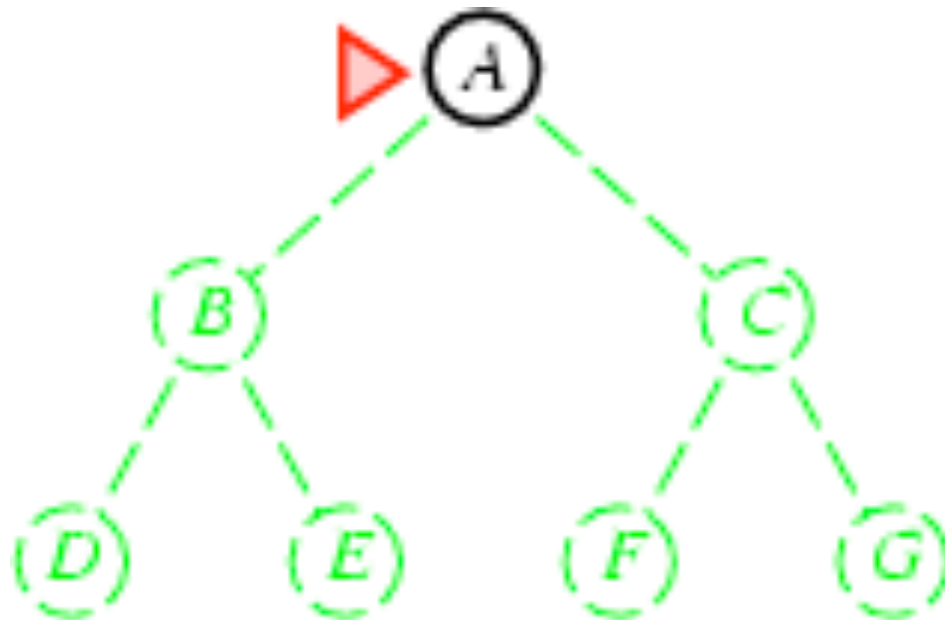
- Given: a graph $G = (V, E)$, directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
 - Pick a vertex as the root
 - Choose certain edges to produce a tree
 - Note: might also build a *forest* if graph is not connected

Breadth-First Search

- “Explore” a graph, turning it into a tree
 - One vertex at a time
 - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
 - Pick a *source vertex* to be the root
 - Find (“discover”) its children, then their children, etc.

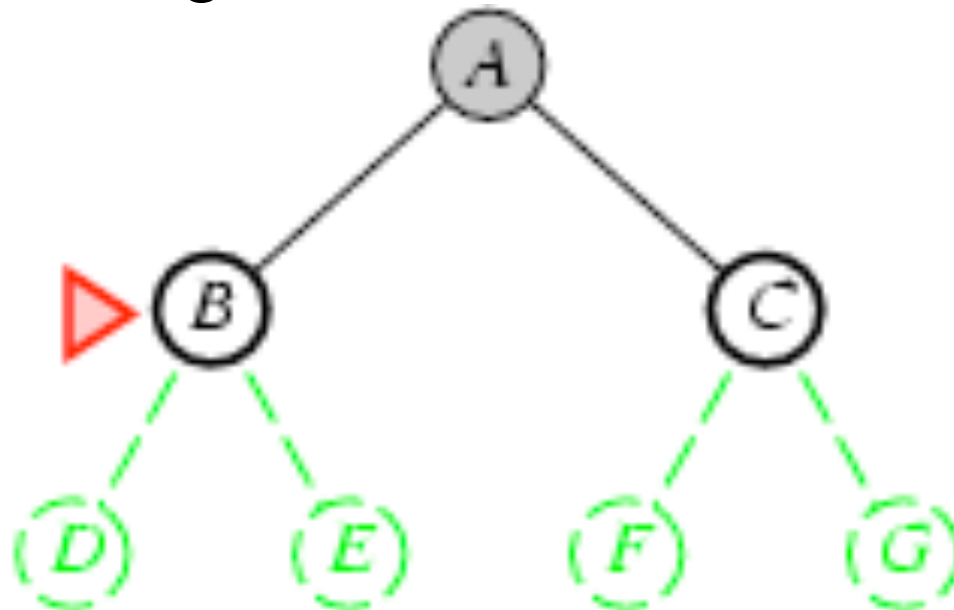
Breadth-first search

- Expand shallowest unexpanded node
- **Implementation:**
 - *fringe* is a FIFO queue, i.e., new successors go at end



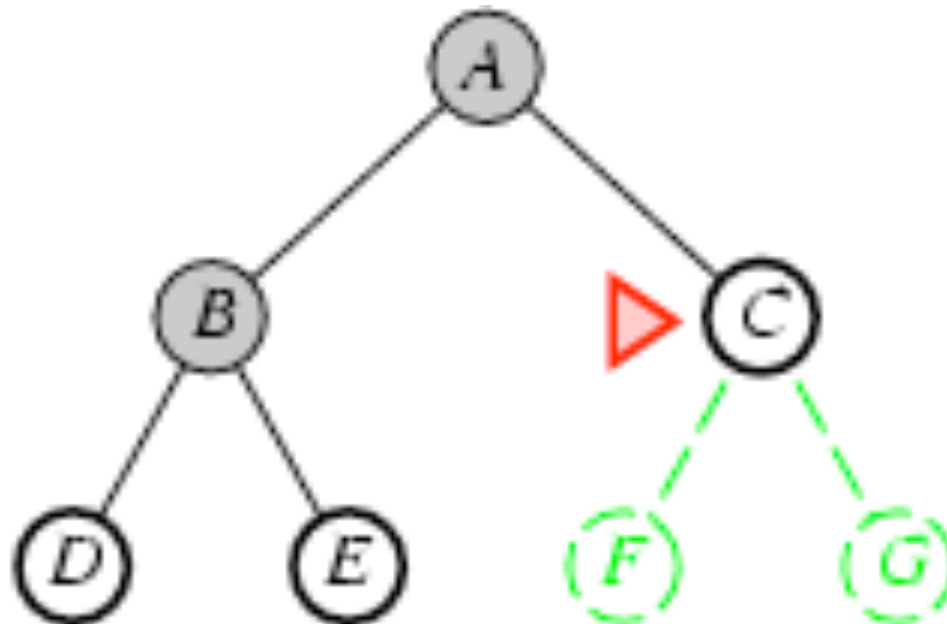
Breadth-first search

- Expand shallowest unexpanded node
- **Implementation:**
 - *fringe* is a FIFO queue, i.e., new successors go at end



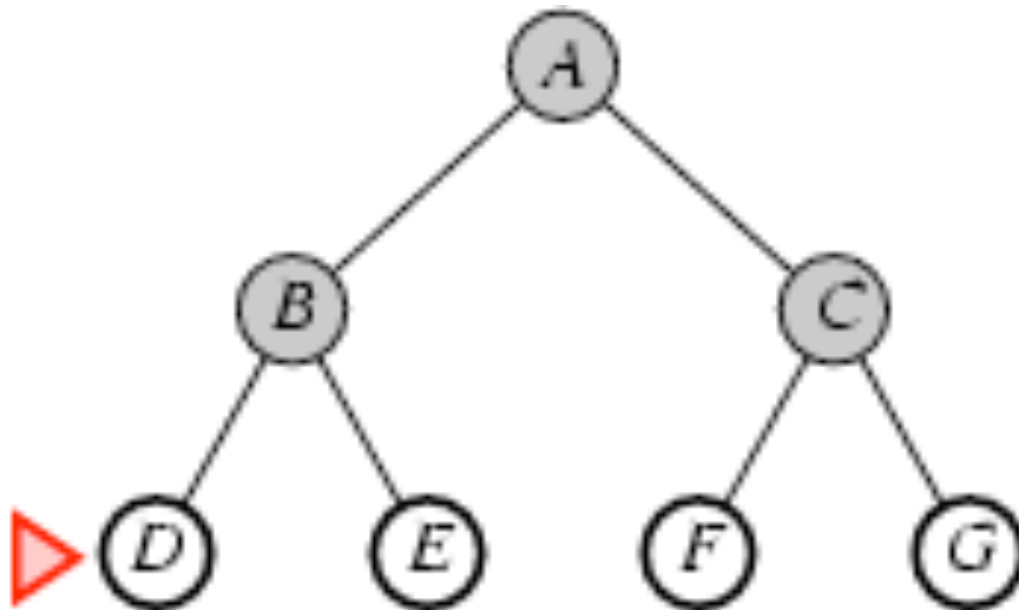
Breadth-first search

- Expand shallowest unexpanded node
- **Implementation:**
 - *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-first search

- Expand shallowest unexpanded node
- **Implementation:**
 - *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
 - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v , or ∞ if v not reachable from s
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G
 - Thus can use BFS to calculate shortest path from one vertex to another in $O(V+E)$ time

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\dots +b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)

- **Space** is the bigger problem (more than time)

Breadth-first search revisited

- Given a graph $G = (V, E)$ and a *source* vertex s
- BFS ‘discovers’ every vertex that is reachable from s
- Computes the distance from s to each vertex (smallest number of edges)
- Produces a ‘breadth-first tree’ with s as root and all reachable vertices
- For any vertex v reachable from s , the path in this tree from s to v corresponds to the shortest path from s to v in G
- Expands the ‘frontier’ between discovered and undiscovered vertices uniformly

BFS

- Each vertex is either white, grey, or black
- All start white, and may become grey and then black
- When a vertex is discovered, it becomes non-white
- All vertices adjacent to black ones are non-white
 - i.e. they have been discovered
- Vertices adjacent to grey ones can be white
 - i.e. they represent the frontier

Breadth-first tree

- Starts out with only root s
- When a white vertex v is discovered from an already discovered vertex u , the vertex v and the edge (u, v) are added to the tree
- u is the predecessor (or parent) of v
 - Each vertex can only have one parent

v

BFS - algorithm

- Assumes that $G = (V, E)$ is represented as an adjacency list
- Colour of vertex u is stored in the variable *colour*[u]
- Predecessor of u is stored in $p[u]$
- $p[u] = \text{NIL}$ if u has not been discovered yet
- Distance from the source s is stored in $d[u]$
- Uses a first-in, first-out queue for the grey vertices

BFS(G, s)

1. for each vertex $u \in V[G]$ (apart from s)
 - set $colour[u] = \text{WHITE}$
 - set $d[u] = \infty$
 - set $p[u] = \text{NIL}$
2. set $colour[s] = \text{GREY}$
3. set $d[s] = 0$
4. set $p[s] = \text{NIL}$
5. let Q be an empty queue ($Q = \emptyset$)
6. add s to Q
7. while $Q \neq \emptyset$
 - remove the first element u from Q
 - for each $v \in Adj[u]$ (the adjacency list of u)
 - if $colour[v] = \text{WHITE}$
 - set $colour[v] = \text{GREY}$
 - set $d[v] = d[u] + 1$
 - set $p[v] = u$
 - add v to the back of Q
 - set $colour[u] = \text{BLACK}$

BFS(G, s)

1. for each vertex $u \in V[G]$ (apart from s)
 - set $colour[u] = \text{WHITE}$
 - set $d[u] = \infty$
 - set $p[u] = \text{NIL}$
2. set $colour[s] = \text{GREY}$
3. set $d[s] = 0$
4. set $p[s] = \text{NIL}$
5. let Q be an empty queue ($Q = \emptyset$)
6. add s to Q
7. while $Q \neq \emptyset$
 - remove the first element u from Q
 - for each $v \in Adj[u]$ (the adjacency list of u)
 - if $colour[v] = \text{WHITE}$
 - set $colour[v] = \text{GREY}$
 - set $d[v] = d[u] + 1$
 - set $p[v] = u$
 - add v to the back of Q
 - set $colour[u] = \text{BLACK}$

Paints all vertices white, sets their distance to s to infinity, and sets their parent to NIL

BFS(G, s)

1. for each vertex $u \in V[G]$ (apart from s)

 set $colour[u] = \text{WHITE}$

 set $d[u] = \infty$

 set $p[u] = \text{NIL}$

Paints all vertices white, sets their distance to s to infinity, and sets their parent to NIL

2. set $colour[s] = \text{GREY}$

3. set $d[s] = 0$

4. set $p[s] = \text{NIL}$

Paints the source s grey and sets its distance from s to 0

5. let Q be an empty queue ($Q = \emptyset$)

6. add s to Q

7. while $Q \neq \emptyset$

 remove the first element u from Q

 for each $v \in Adj[u]$ (the adjacency list of u)

 if $colour[v] = \text{WHITE}$

 set $colour[v] = \text{GREY}$

 set $d[v] = d[u] + 1$

 set $p[v] = u$

 add v to the back of Q

 set $colour[u] = \text{BLACK}$

BFS(G, s)

1. for each vertex $u \in V[G]$ (apart from s)

 set $colour[u] = \text{WHITE}$

 set $d[u] = \infty$

 set $p[u] = \text{NIL}$

Paints all vertices white, sets their distance to s to infinity, and sets their parent to NIL

2. set $colour[s] = \text{GREY}$

Paints the source s grey and sets its distance from s to 0

3. set $d[s] = 0$

4. set $p[s] = \text{NIL}$

5. let Q be an empty queue ($Q = \emptyset$)

Creates an empty queue and adds s to it

6. add s to Q

7. while $Q \neq \emptyset$

 remove the first element u from Q

 for each $v \in Adj[u]$ (the adjacency list of u)

 if $colour[v] = \text{WHITE}$

 set $colour[v] = \text{GREY}$

 set $d[v] = d[u] + 1$

 set $p[v] = u$

 add v to the back of Q

 set $colour[u] = \text{BLACK}$

BFS(G, s)

1. for each vertex $u \in V[G]$ (apart from s)
 - set $colour[u] = \text{WHITE}$
 - set $d[u] = \infty$
 - set $p[u] = \text{NIL}$

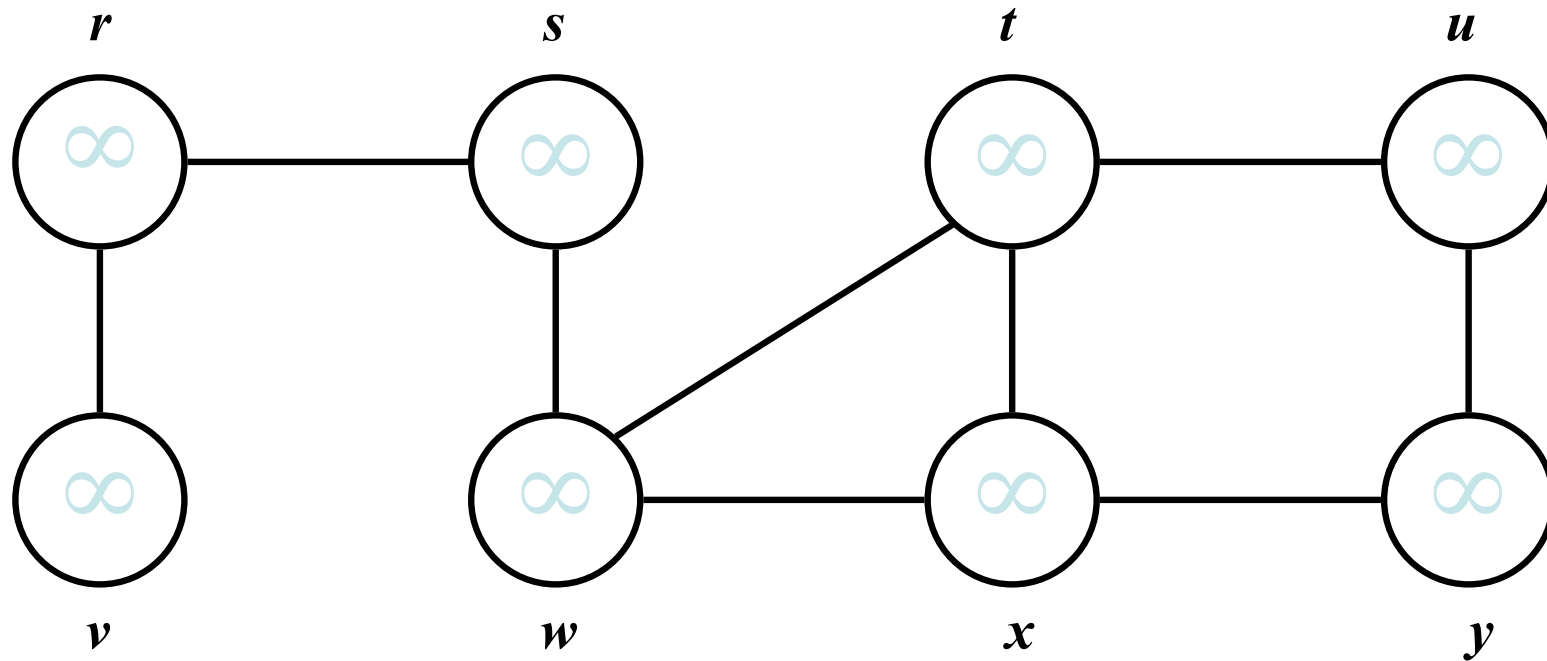
Paints all vertices white, sets their distance to s to infinity, and sets their parent to NIL
2. set $colour[s] = \text{GREY}$

Paints the source s grey and sets its distance from s to 0
3. set $d[s] = 0$
4. set $p[s] = \text{NIL}$
5. let Q be an empty queue ($Q = \emptyset$)

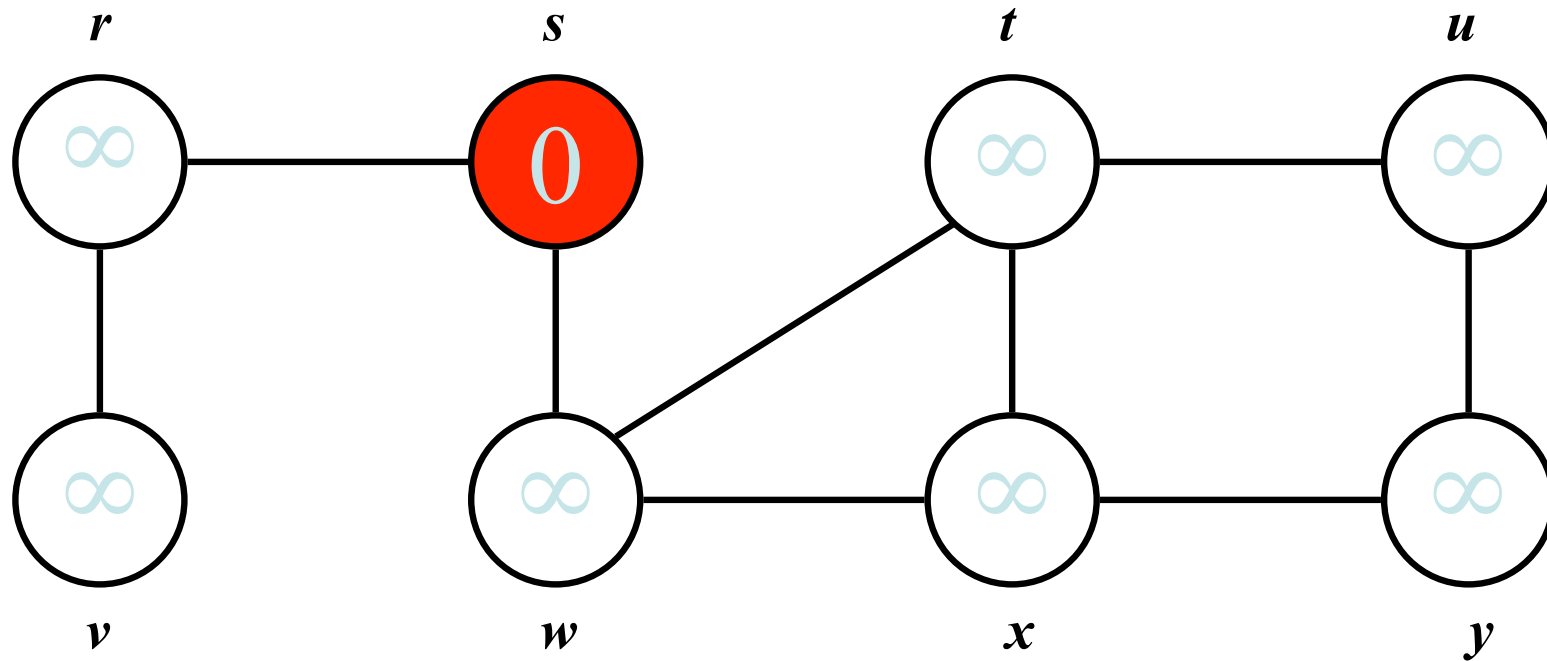
Creates an empty queue and adds s to it
6. add s to Q
7. while $Q \neq \emptyset$
 - remove the first element u from Q
 - for each $v \in Adj[u]$ (the adjacency list of u)
 - if $colour[v] = \text{WHITE}$
 - set $colour[v] = \text{GREY}$
 - set $d[v] = d[u] + 1$
 - set $p[v] = u$
 - add v to the back of Q
 - set $colour[u] = \text{BLACK}$

Proceeds while there are grey vertices
Removes the first vertex u from the queue, looks at each neighbour in turn
If neighbour is white, it is discovered and painted grey, its distance to s is incremented, its parent is stored and it is added to the back of the queue
Then u is painted black

Breadth-First Search: Example

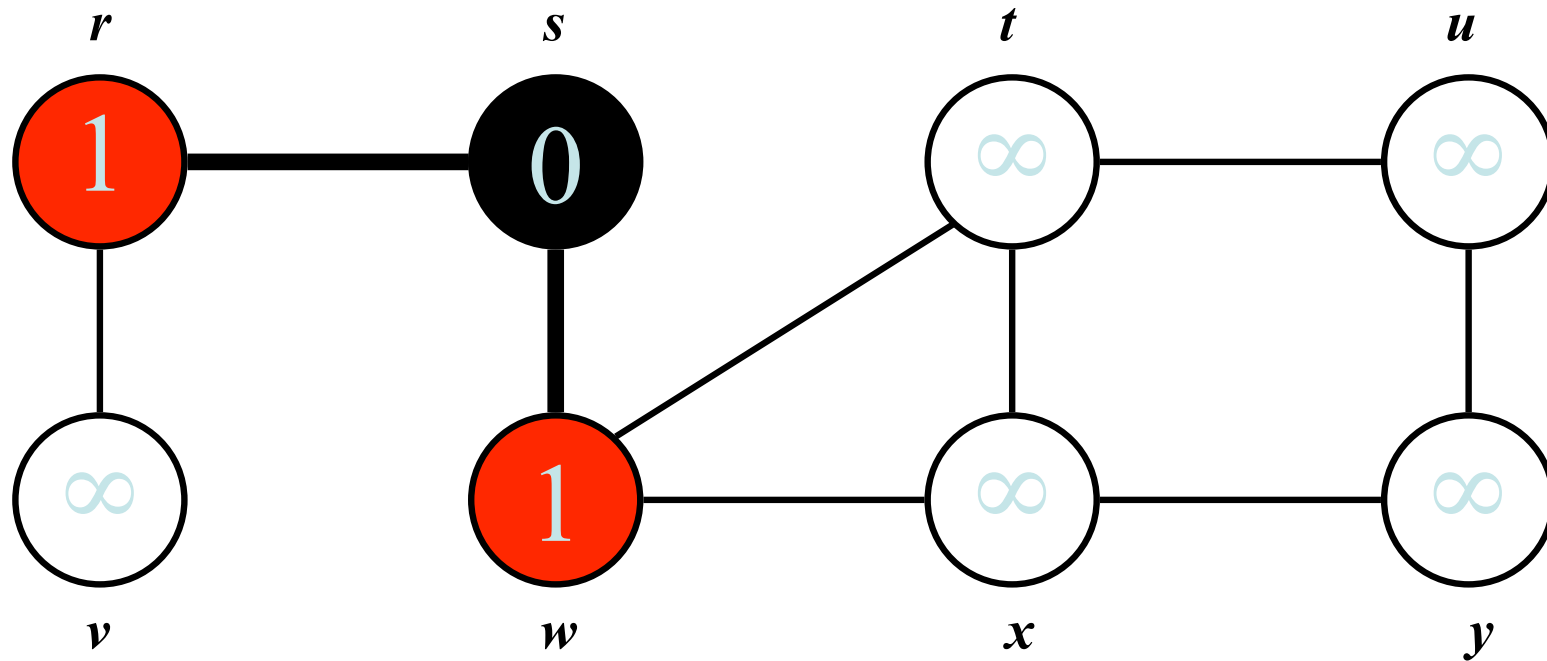


Breadth-First Search: Example



$Q:$ s

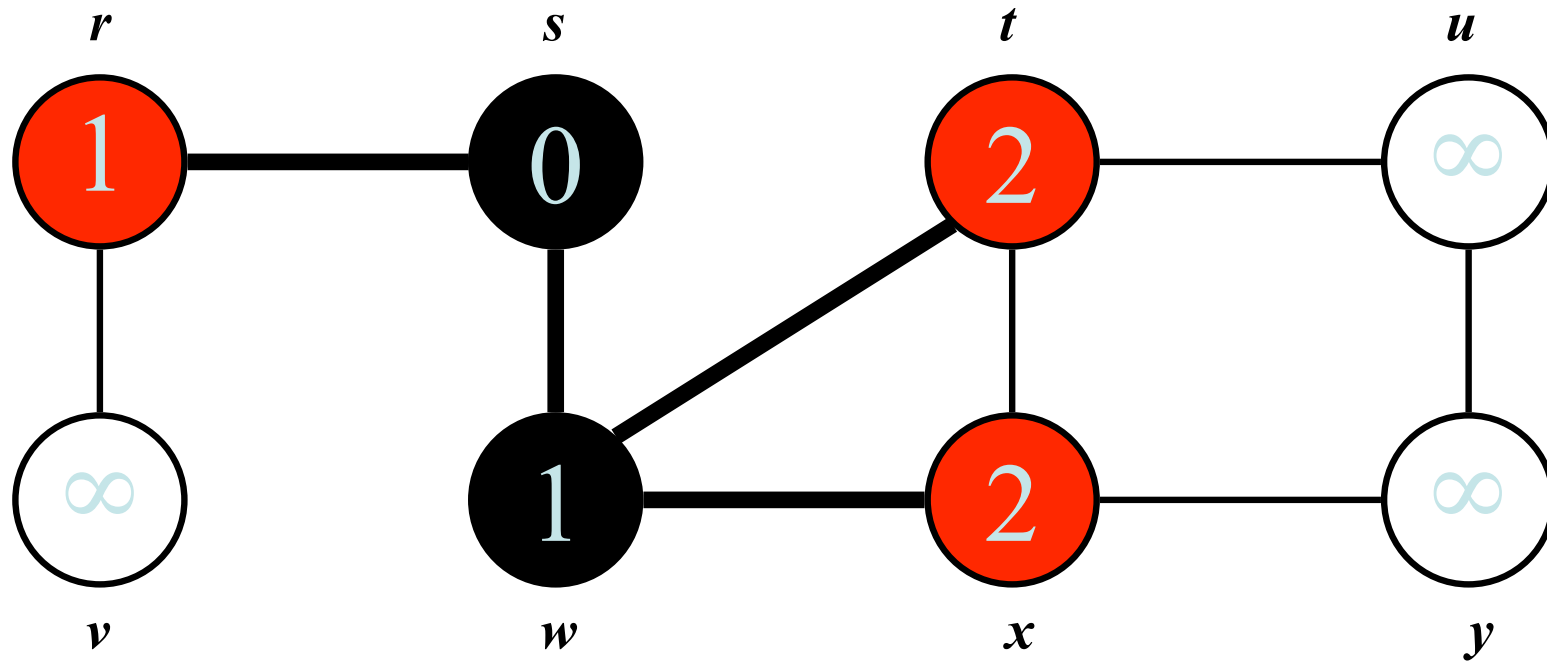
Breadth-First Search: Example



Q :

w	r
-----	-----

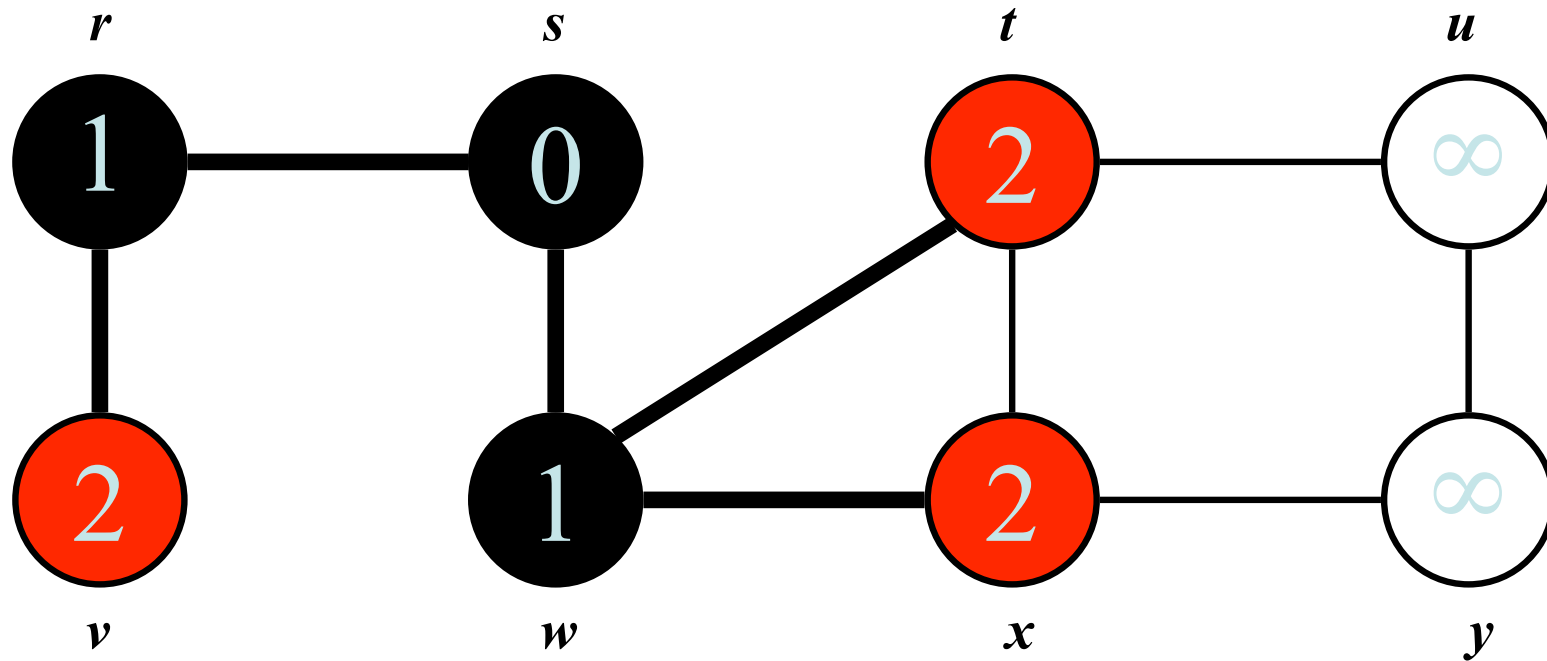
Breadth-First Search: Example



Q :

r	t	x
-----	-----	-----

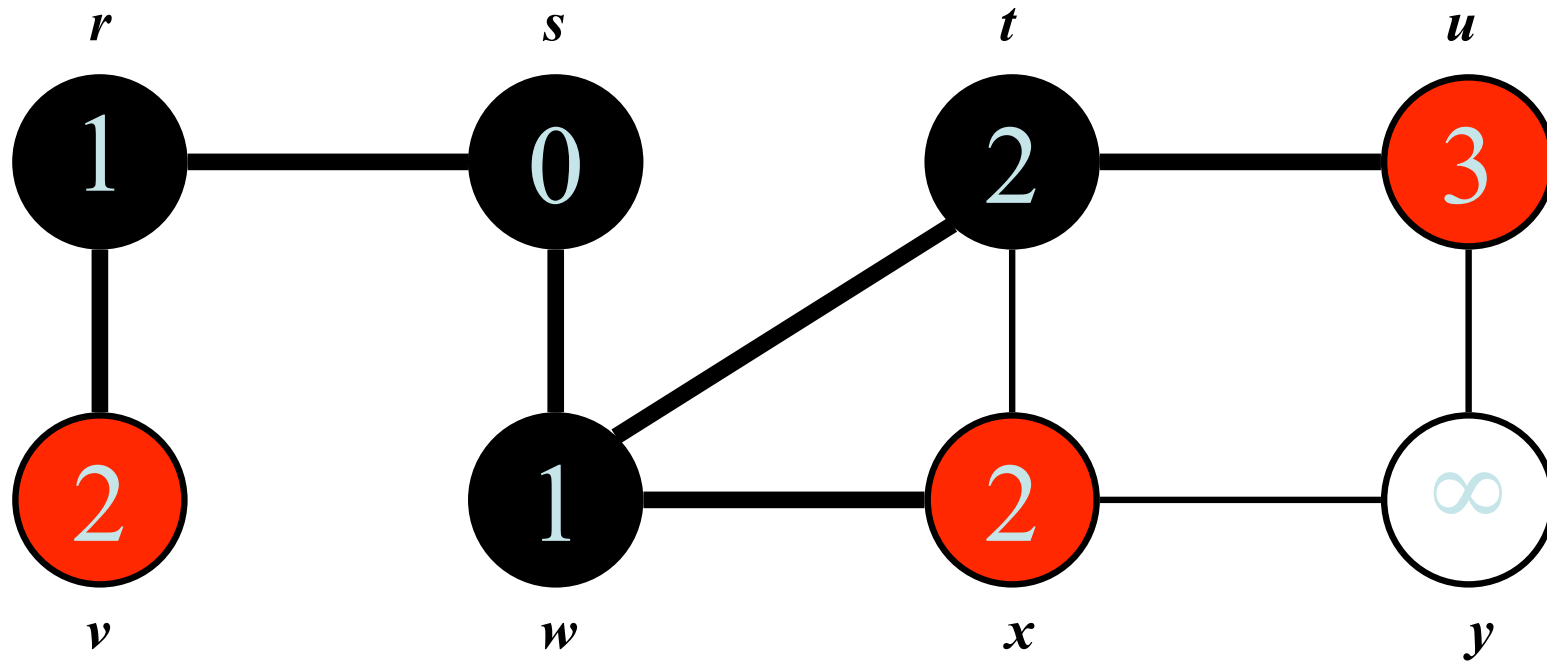
Breadth-First Search: Example



Q :

t	x	v
-----	-----	-----

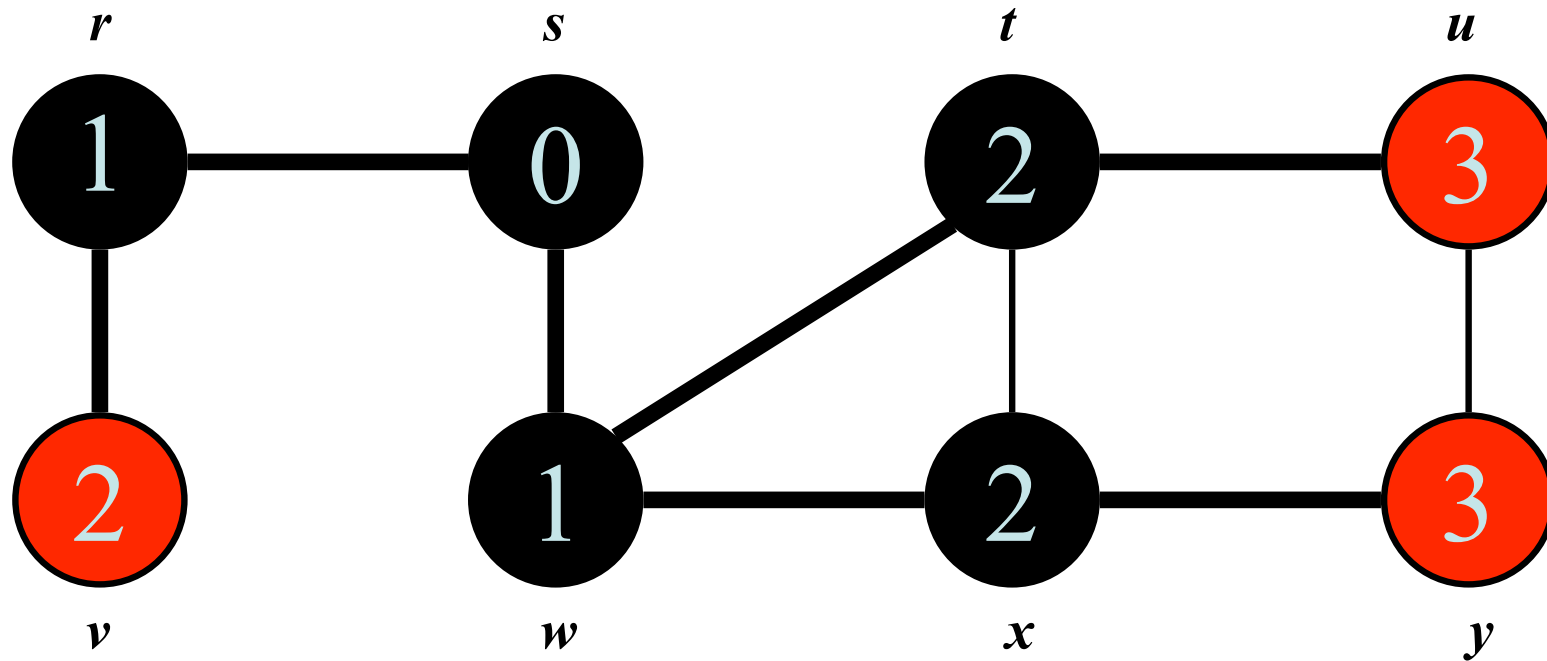
Breadth-First Search: Example



$Q:$

x	v	u
-----	-----	-----

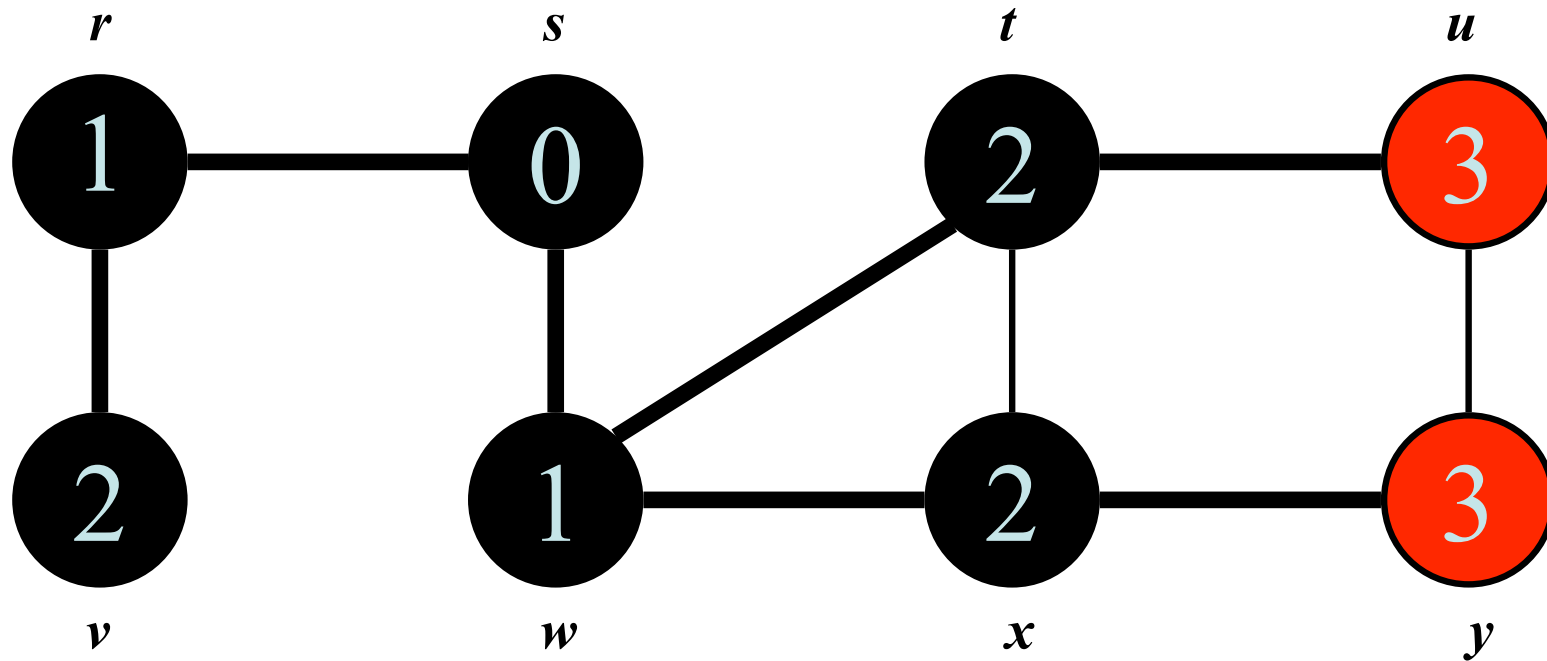
Breadth-First Search: Example



Q :

v	u	y
-----	-----	-----

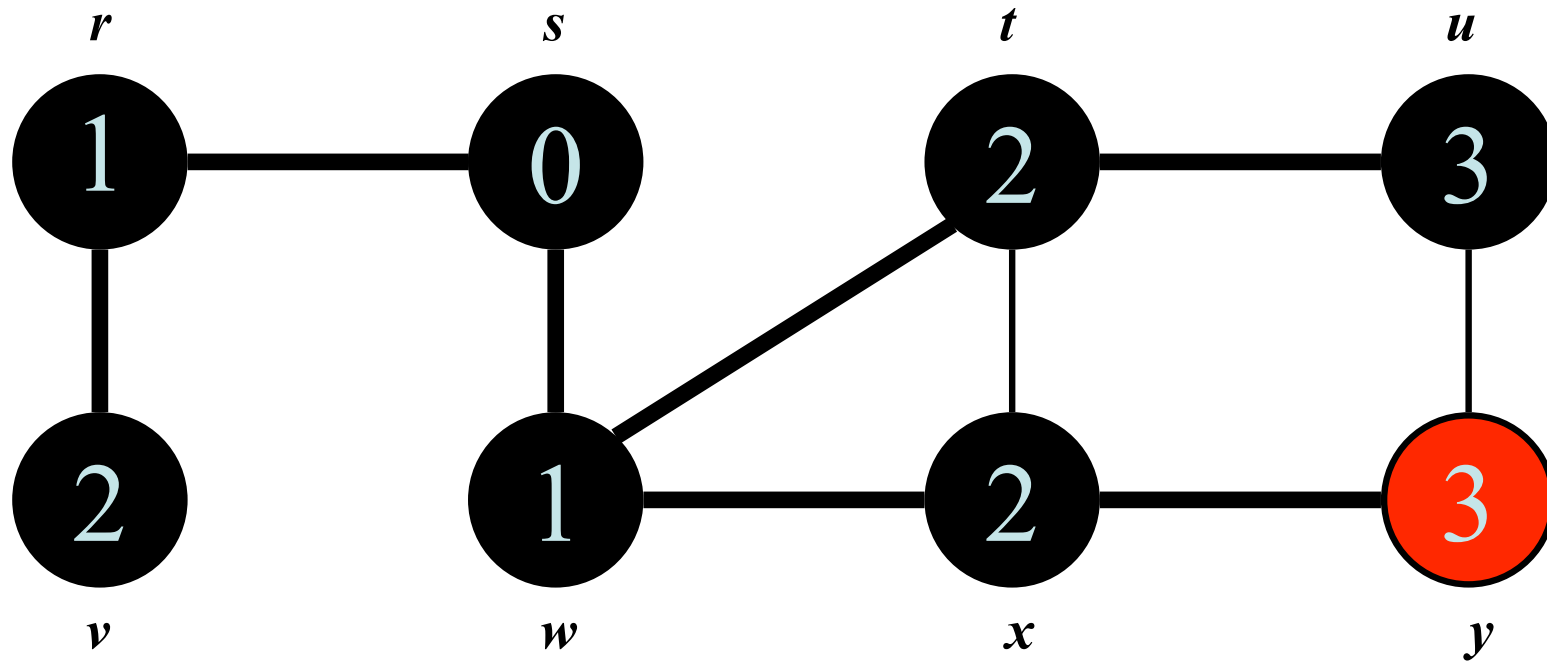
Breadth-First Search: Example



Q :

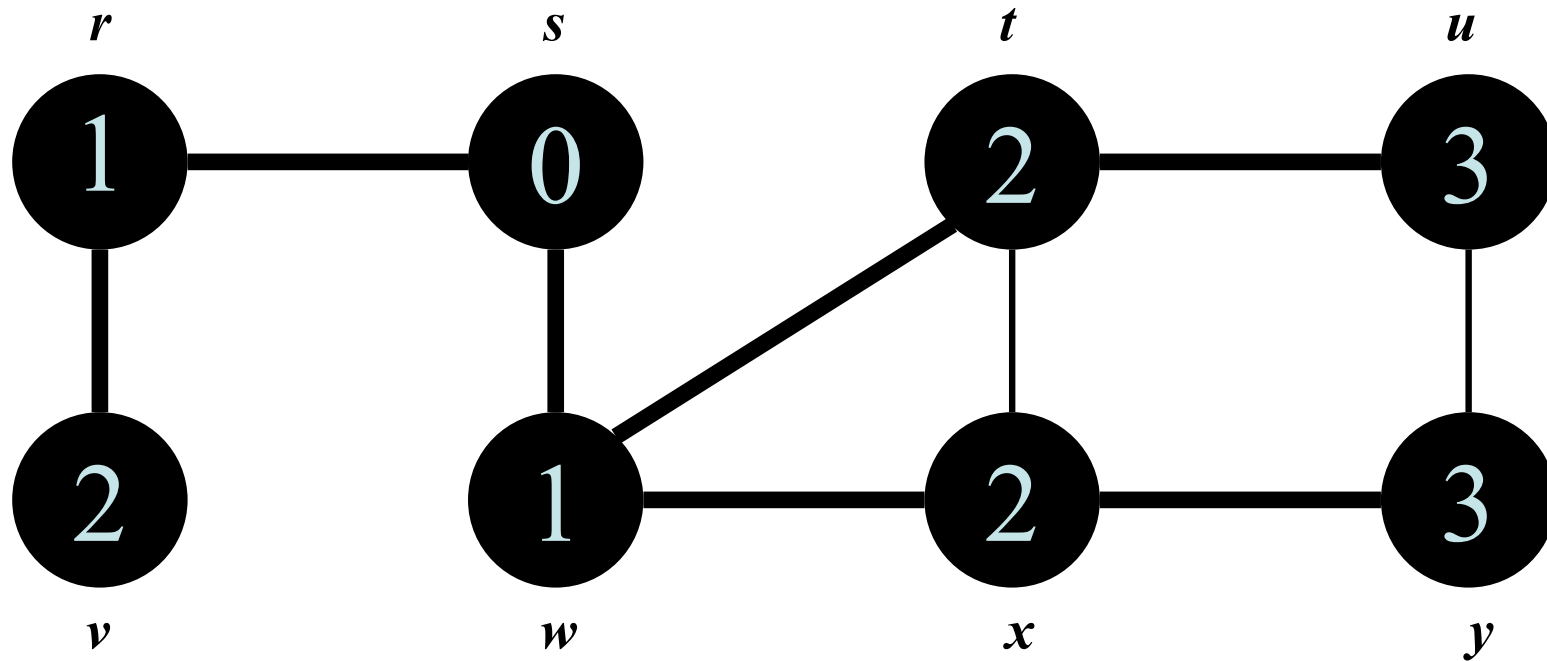
u	y
-----	-----

Breadth-First Search: Example



$Q:$ y

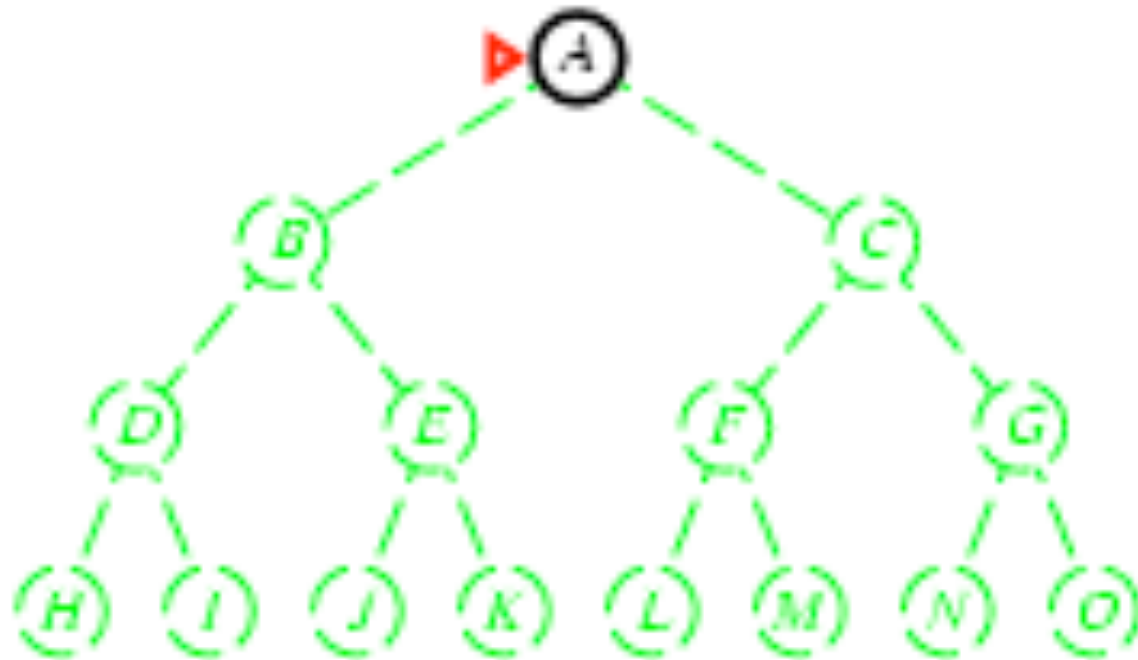
Breadth-First Search: Example



$Q: \emptyset$

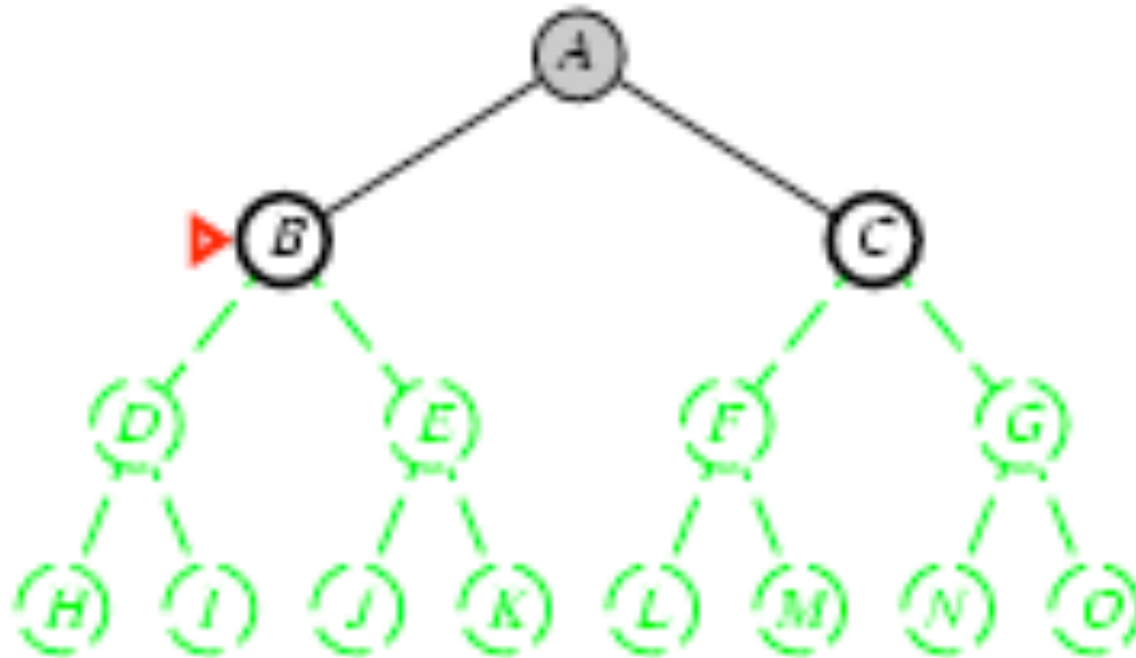
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



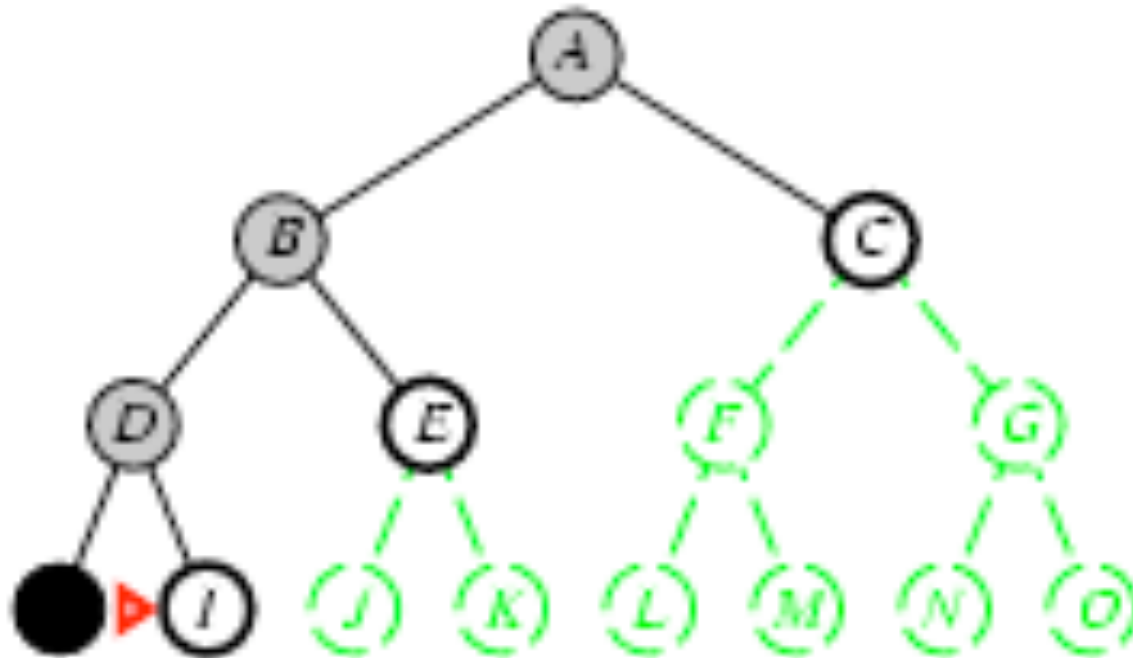
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



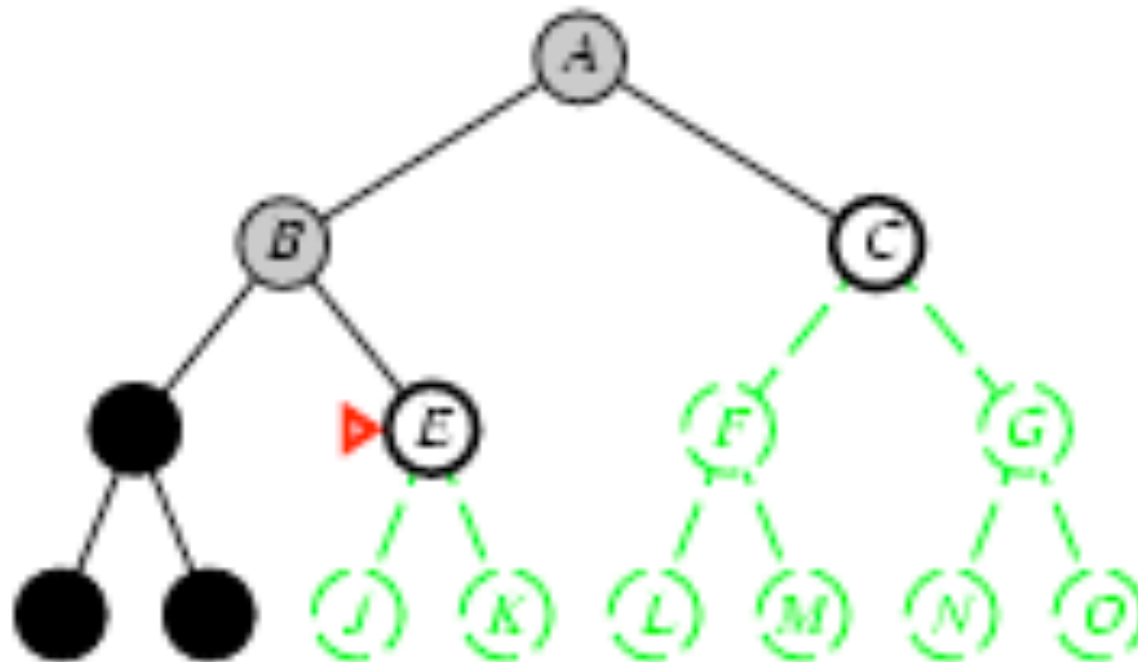
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



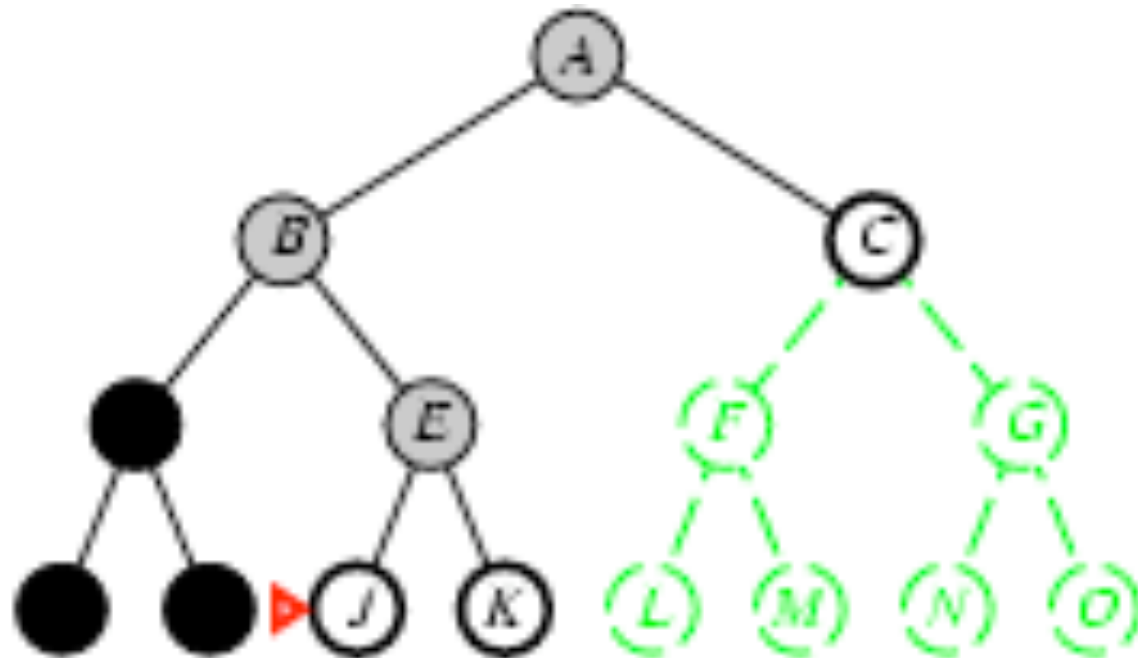
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



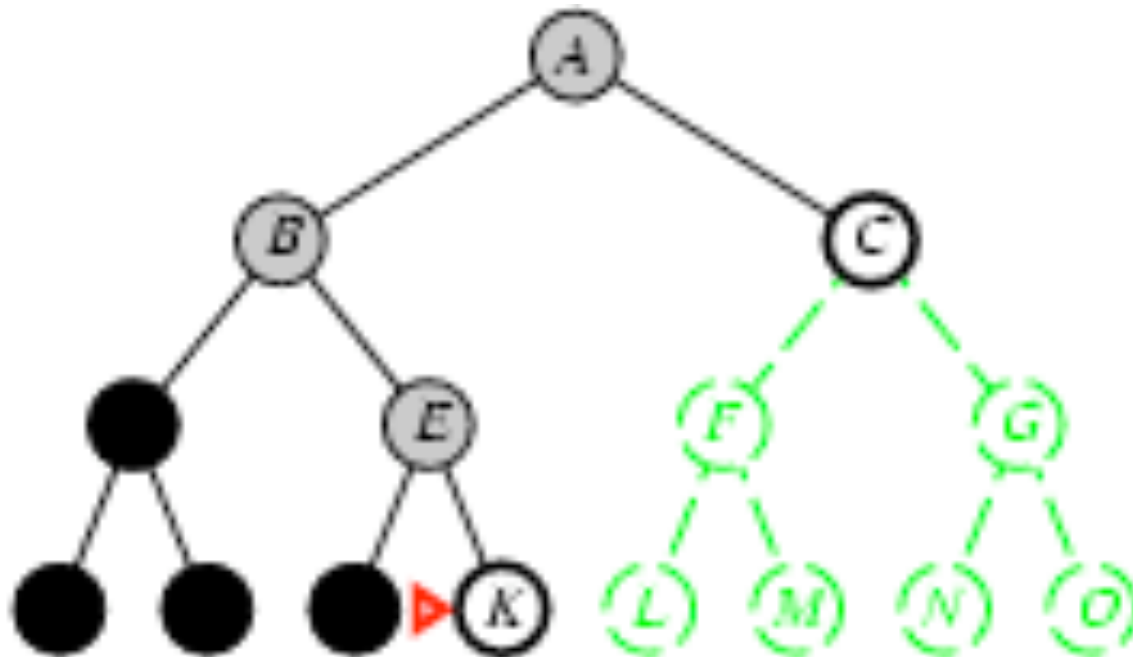
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



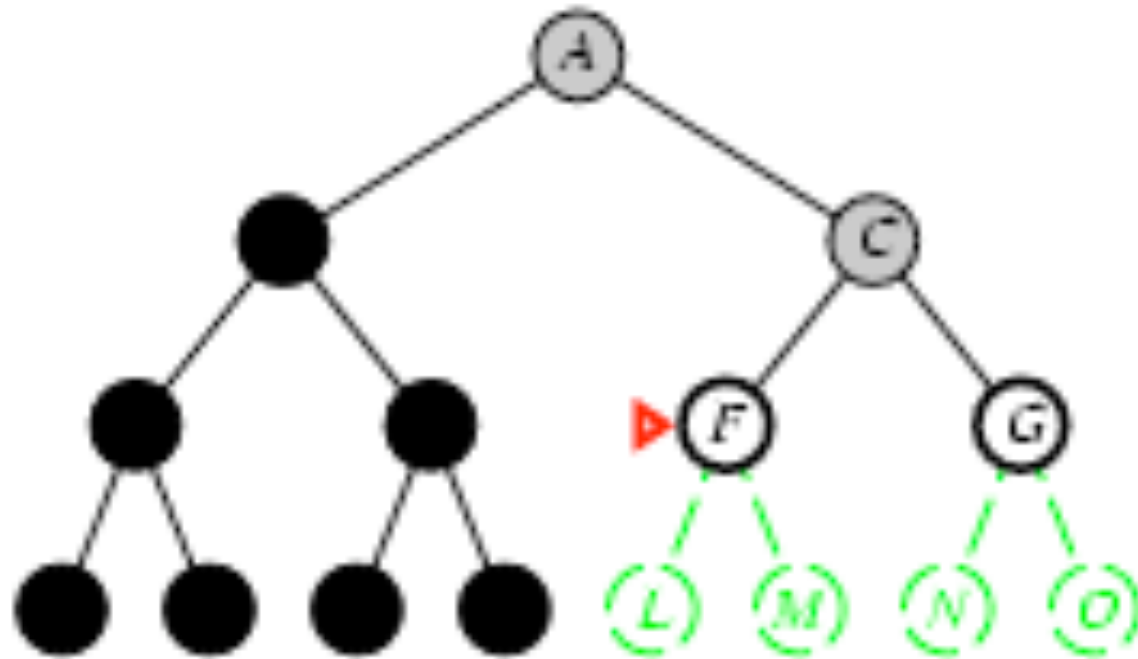
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



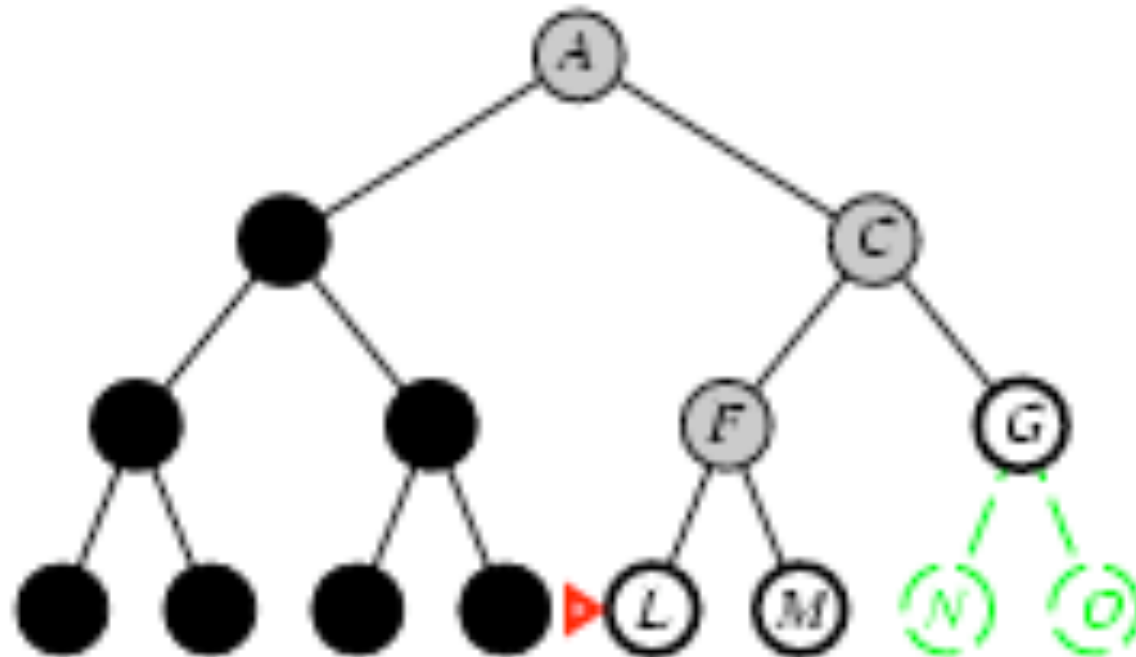
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



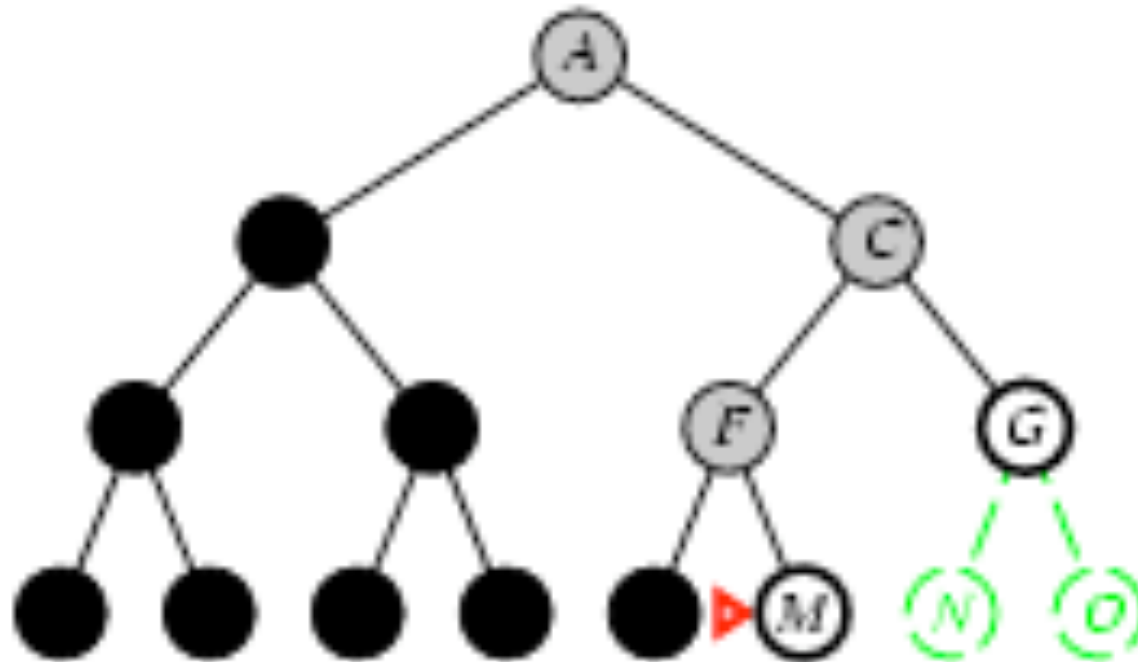
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
 - Modify to avoid repeated states along path
 - complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
 - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No

Depth-first search revisited

- Edges out of the most recently discovered vertex v are explored
- If v has no unexplored edges, the search 'backtracks' to explore edges leaving the vertex from which v has been discovered
- Process continues until all vertices which can be reached from the source are discovered
- If undiscovered vertices are left, one of them is chosen as new source and the process is repeated

DFS - algorithm

- Records the predecessor of each vertex
- Produces a ‘depth-first forest’ consisting of several (disjointed) trees
- Undiscovered vertices are white, greyed when discovered, and blackened when ‘finished’
 - when its adjacency list has been examined completely
- Each vertex v obtains two ‘timestamps’, the first one, $d[v]$ records when v is discovered, the second one, $f[v]$ when v is finished
- Vertex v is white before $d[v]$, grey between $d[v]$ and $f[v]$, and black thereafter

DFS(G)

1. for each vertex $u \in V(G)$
 - set $colour[u] = WHITE$
 - set $p[u] = NIL$
2. set $time = 0$
3. for each vertex $u \in V(G)$
 - if $colour[u] = WHITE$
 - do DFS-VISIT(u)

DFS(G)

1. for each vertex $u \in V(G)$

 set $colour[u] = WHITE$

 set $p[u] = NIL$

2. set $time = 0$

3. for each vertex $u \in V(G)$

 if $colour[u] = WHITE$

 do DFS-VISIT(u)

Paints all vertices white,
sets their parent to NIL

DFS(G)

1. for each vertex $u \in V(G)$

 set $colour[u] = WHITE$

 set $p[u] = NIL$

2. set $time = 0$

3. for each vertex $u \in V(G)$

 if $colour[u] = WHITE$

 do DFS-VISIT(u)

Paints all vertices white,
sets their parent to NIL

Resets the global time
counter

DFS(G)

1. for each vertex $u \in V(G)$

 set $colour[u] = WHITE$

 set $p[u] = NIL$

2. set $time = 0$

3. for each vertex $u \in V(G)$

 if $colour[u] = WHITE$

 do DFS-VISIT(u)

Paints all vertices white,
sets their parent to NIL

Resets the global time
counter

Visits each vertex in turn, if
a white one is found, it
visits all vertices starting
from there, using the
algorithm DFS-VISIT

DFS-VISIT(*u*)

1. set $colour[u] = GREY$
2. set $time = time + 1$
3. set $d[u] = time$
4. for each $v \in Adj[u]$
 - if $colour[v] = WHITE$
 - set $p[v] = u$
 - do DFS-VISIT(v)
5. set $colour[u] = BLACK$
6. set $time = time + 1$
7. set $f[u] = time$

DFS-VISIT(u)

1. set $colour[u] = GREY$
2. set $time = time + 1$
3. set $d[u] = time$
4. for each $v \in Adj[u]$
 - if $colour[v] = WHITE$
 - set $p[v] = u$
 - do DFS-VISIT(v)
5. set $colour[u] = BLACK$
6. set $time = time + 1$
7. set $f[u] = time$

A white vertex has been discovered, and its first timestamp is recorded

DFS-VISIT(*u*)

1. set $colour[u] = GREY$
2. set $time = time + 1$
3. set $d[u] = time$
4. for each $v \in Adj[u]$
 - if $colour[v] = WHITE$
 - set $p[v] = u$
 - do DFS-VISIT(v)
5. set $colour[u] = BLACK$
6. set $time = time + 1$
7. set $f[u] = time$

A white vertex has been discovered, and its first timestamp is recorded

Explore all edges originating from u , and recursively explore further

DFS-VISIT(*u*)

1. set $colour[u] = GREY$
2. set $time = time + 1$
3. set $d[u] = time$
4. for each $v \in Adj[u]$
 - if $colour[v] = WHITE$
 - set $p[v] = u$
 - do DFS-VISIT(*v*)
5. set $colour[u] = BLACK$
6. set $time = time + 1$
7. set $f[u] = time$

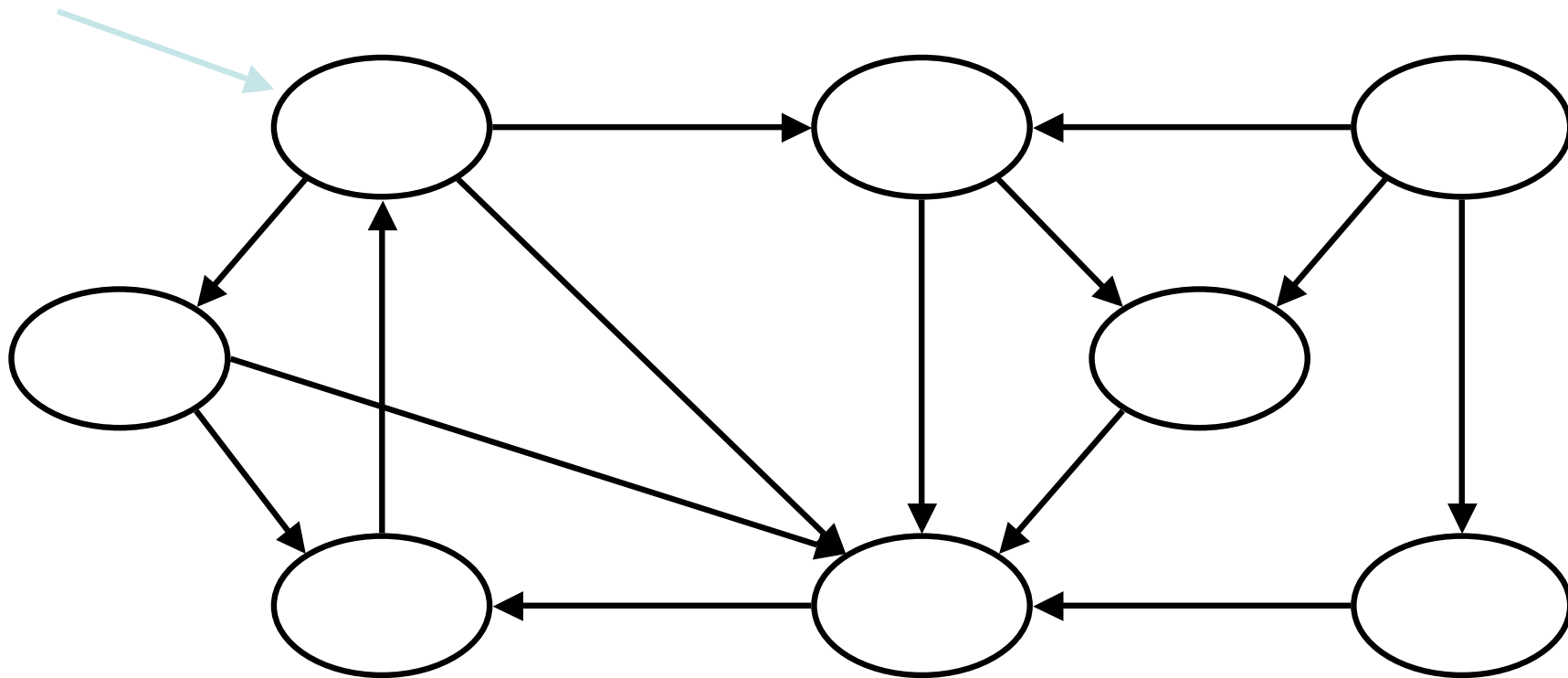
A white vertex has been discovered, and its first timestamp is recorded

Explore all edges originating from *u*, and recursively explore further

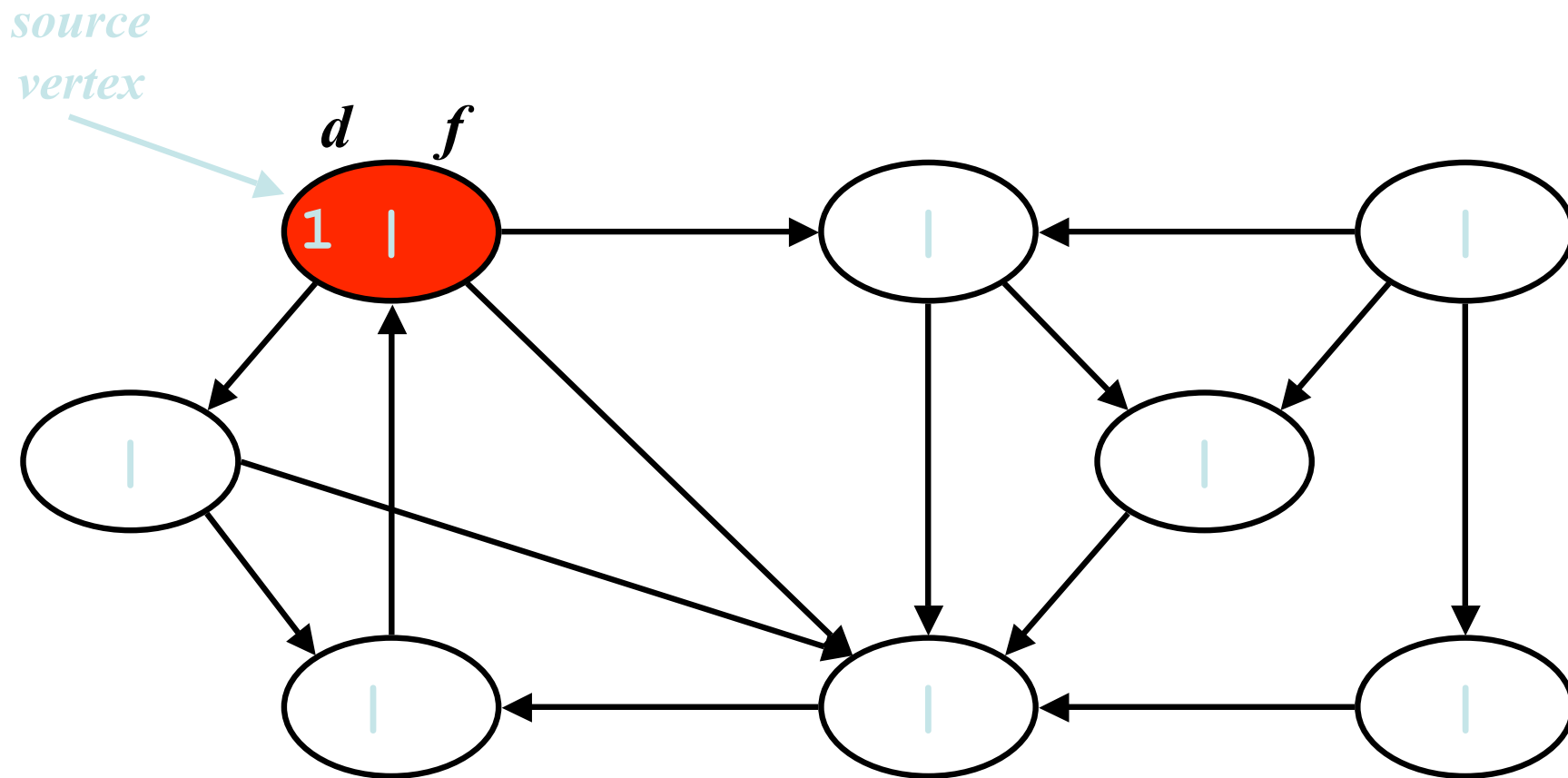
u is finished and blackened, its second timestamp is recorded

DFS Example

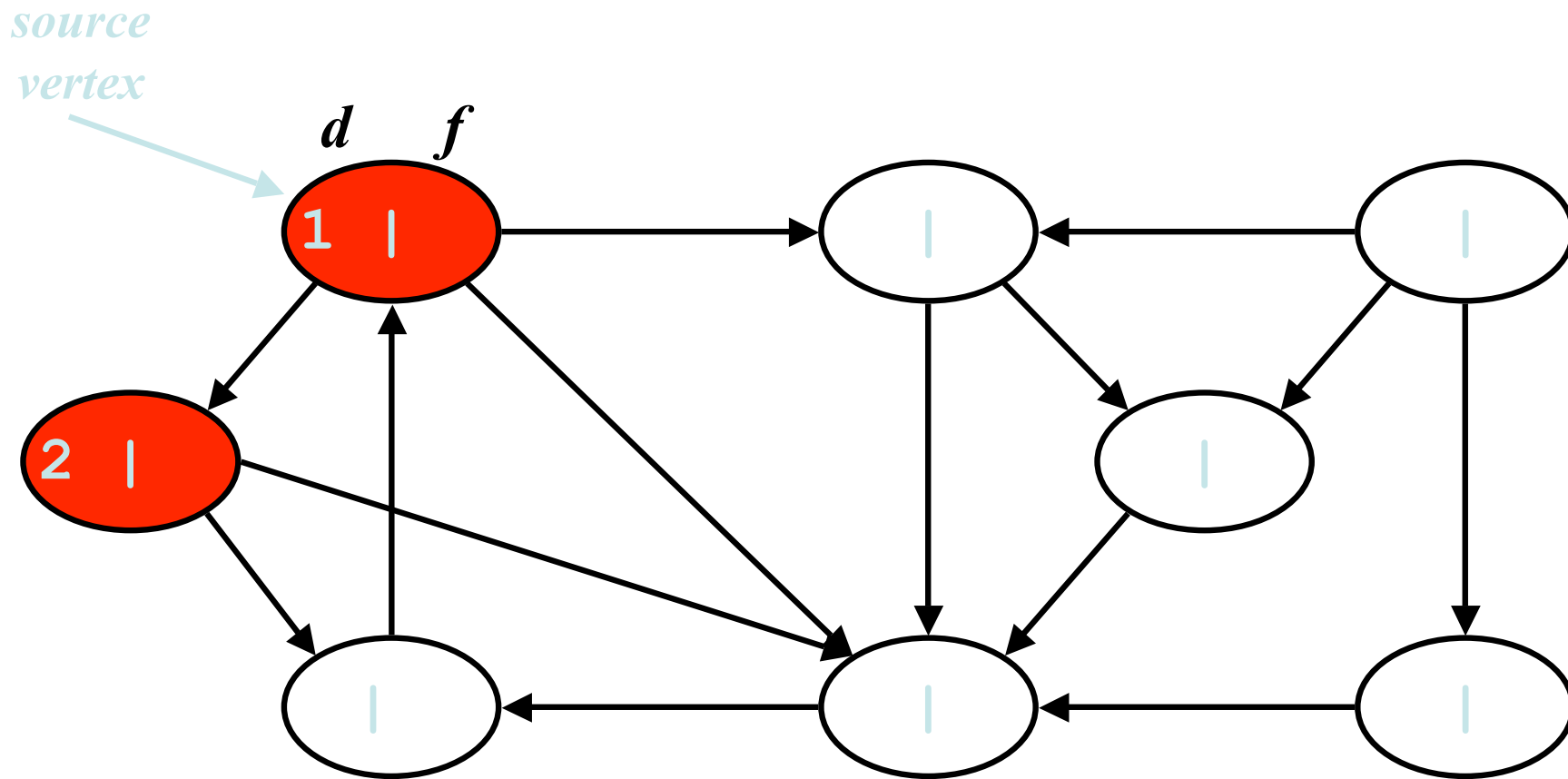
*source
vertex*



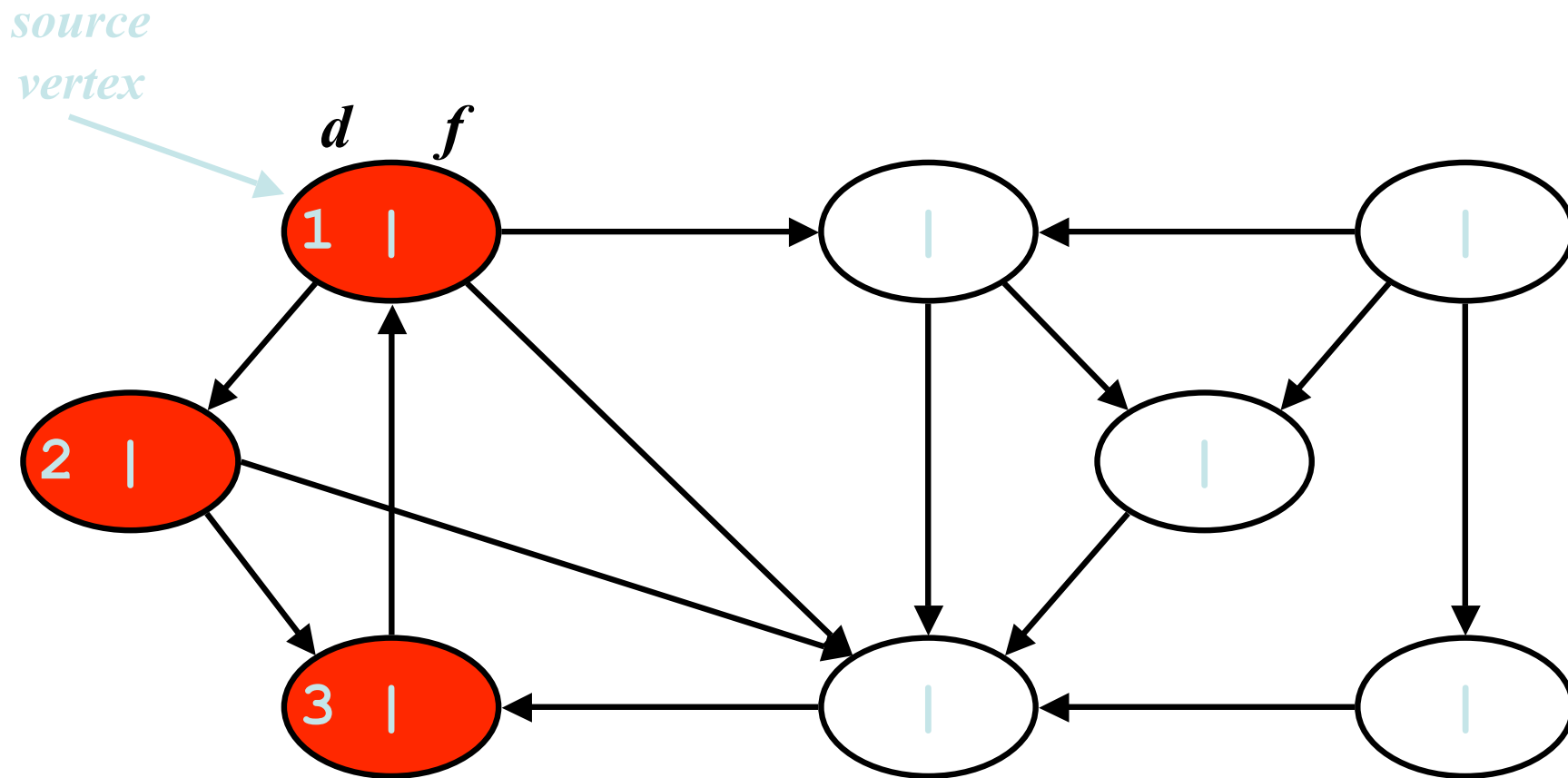
DFS Example



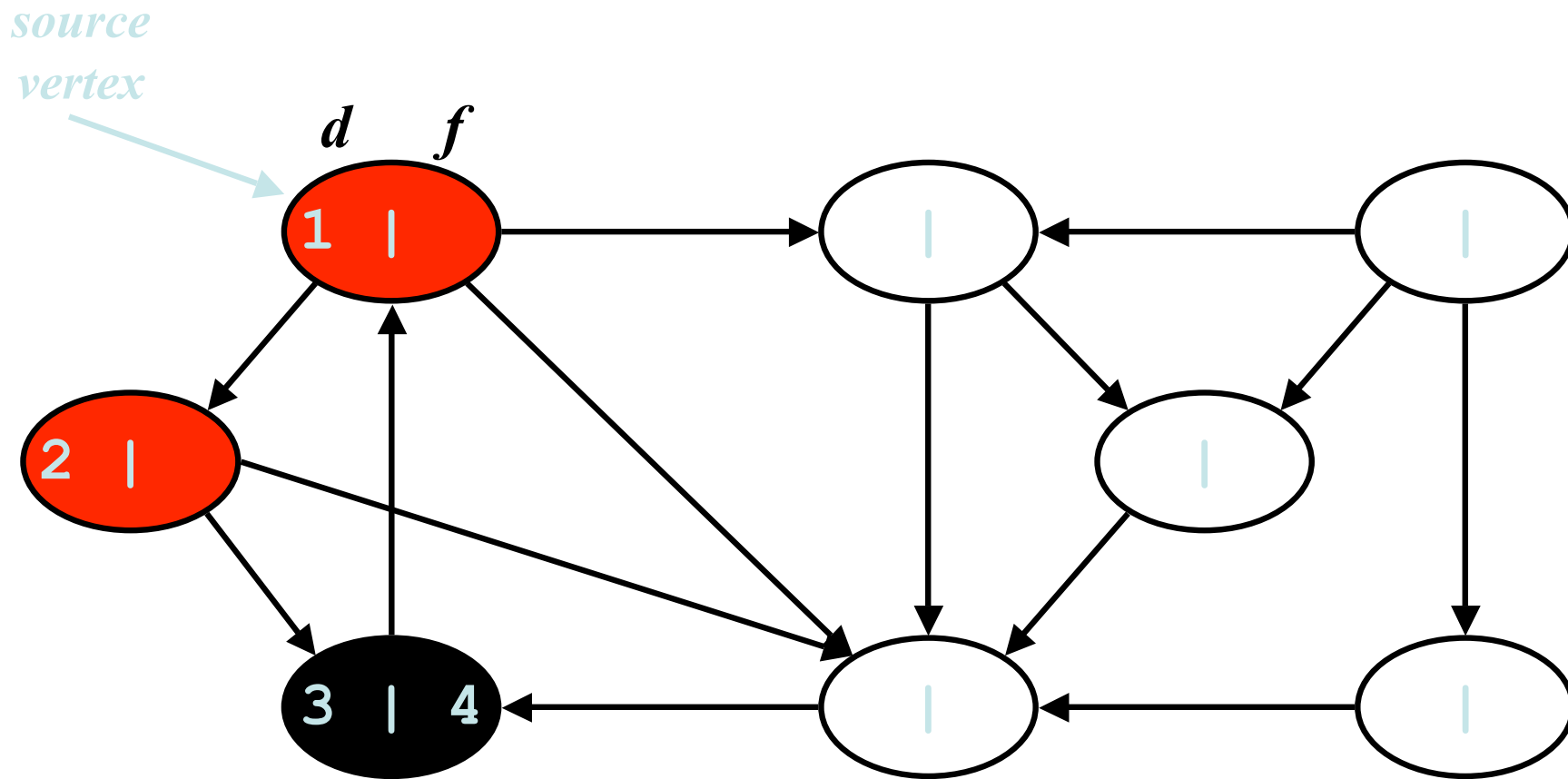
DFS Example



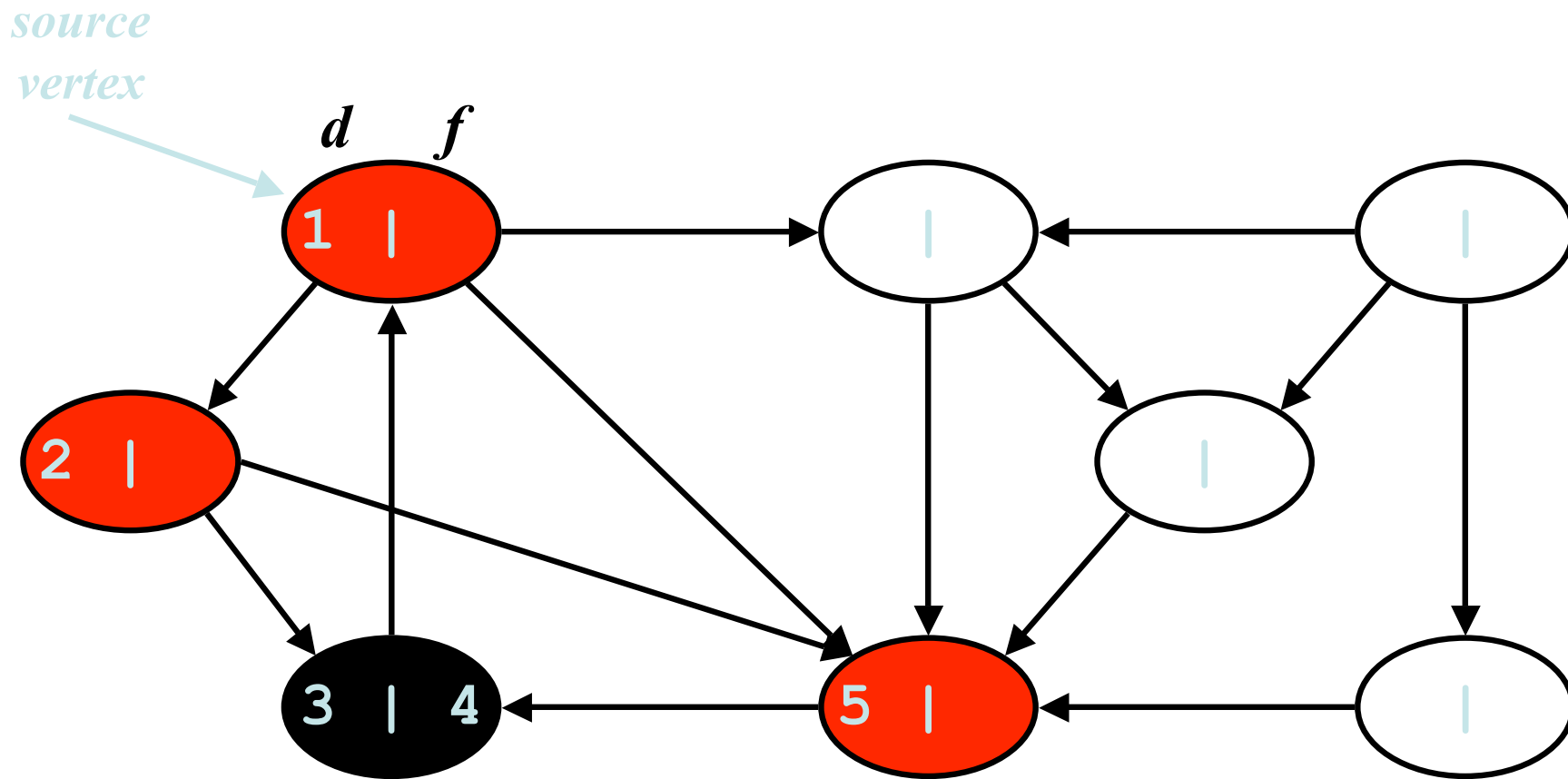
DFS Example



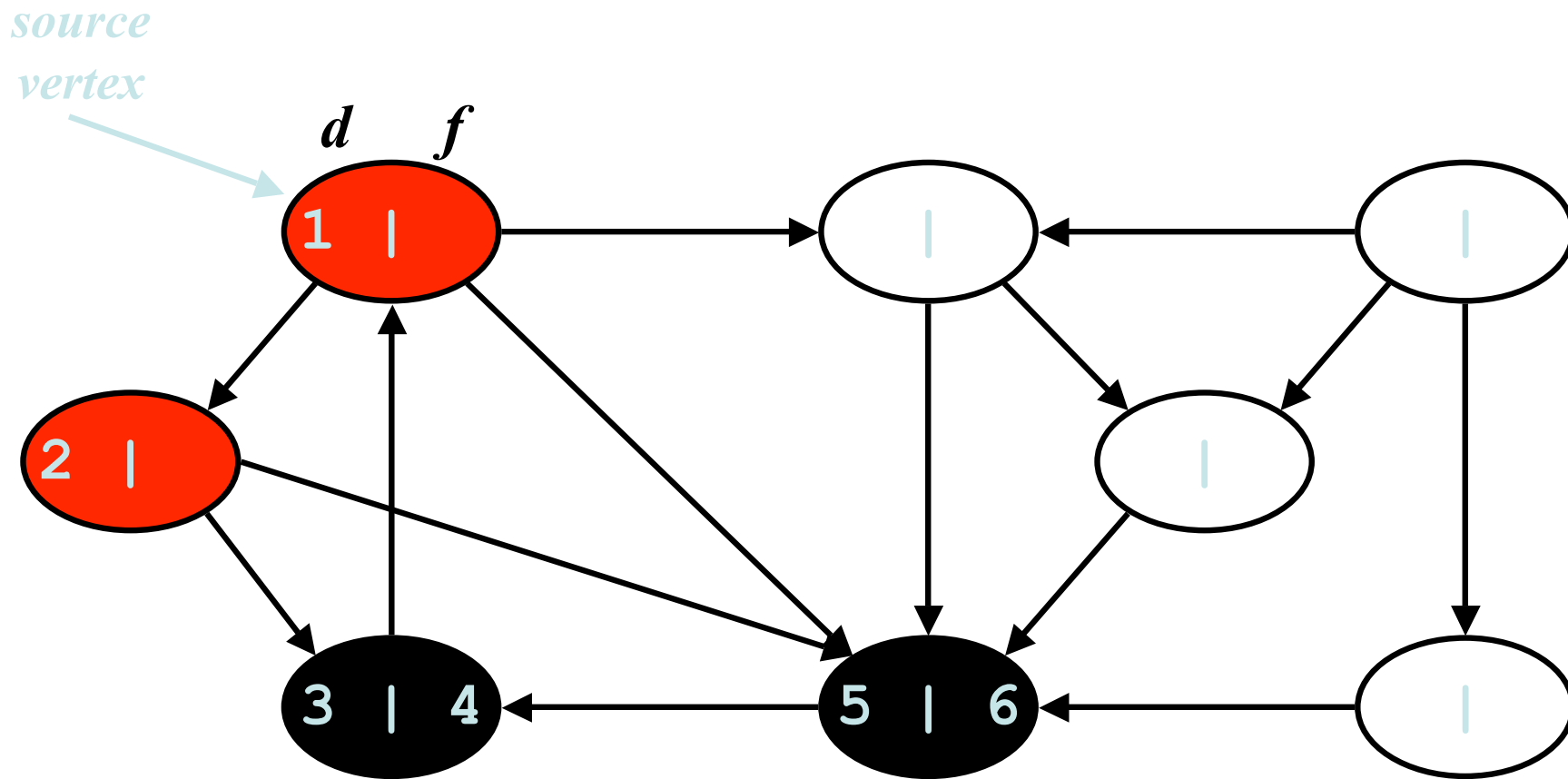
DFS Example



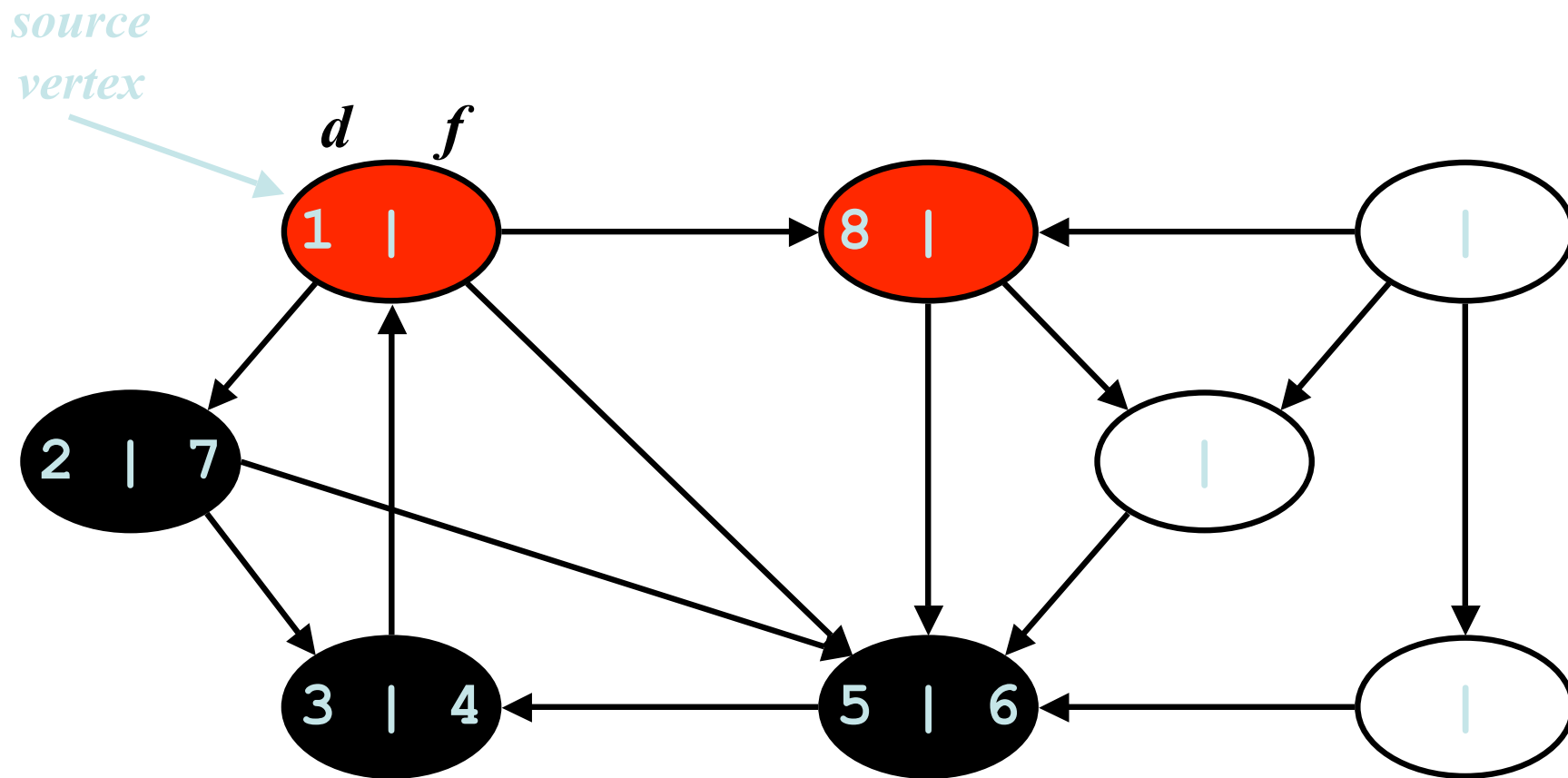
DFS Example



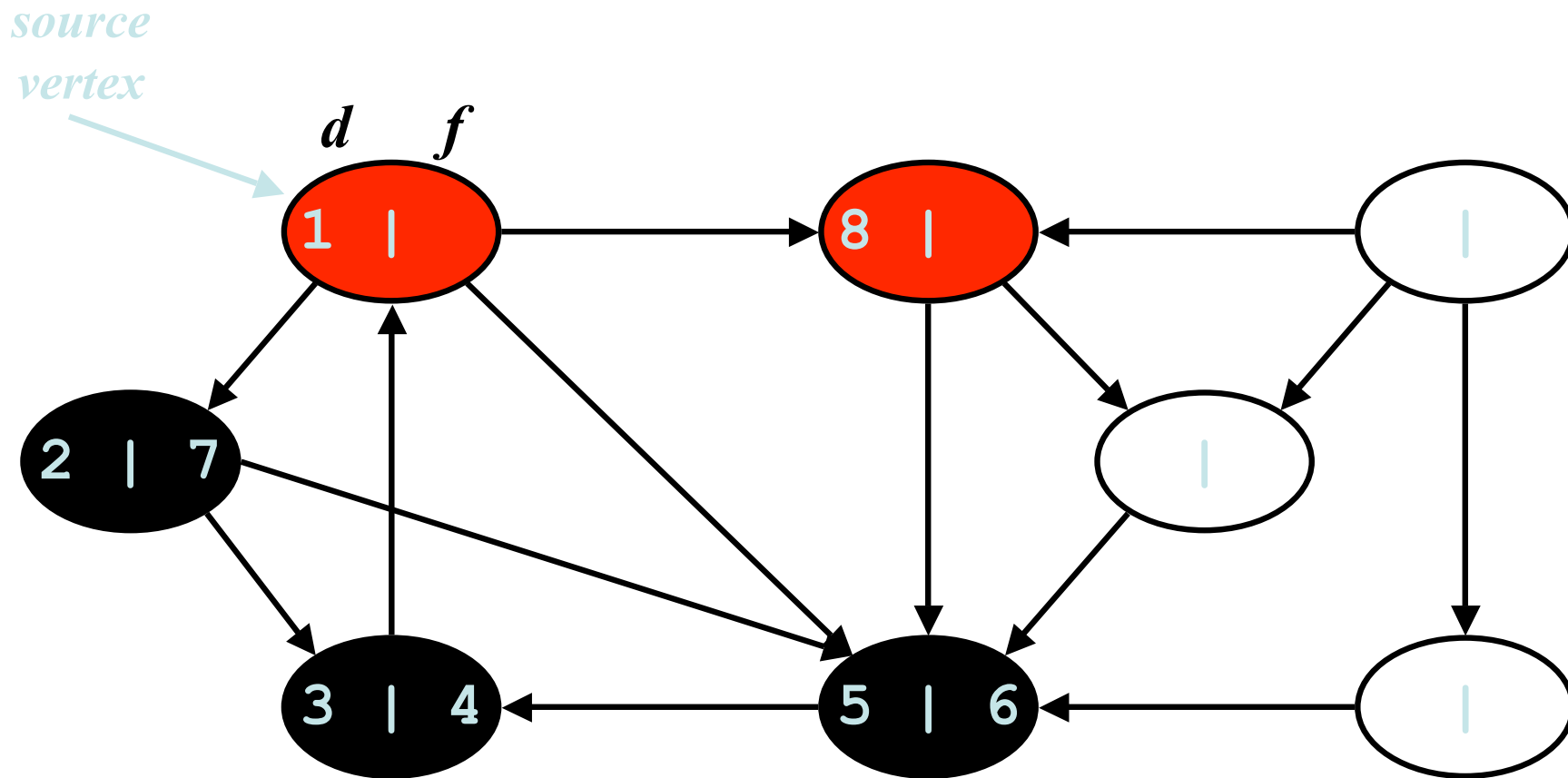
DFS Example



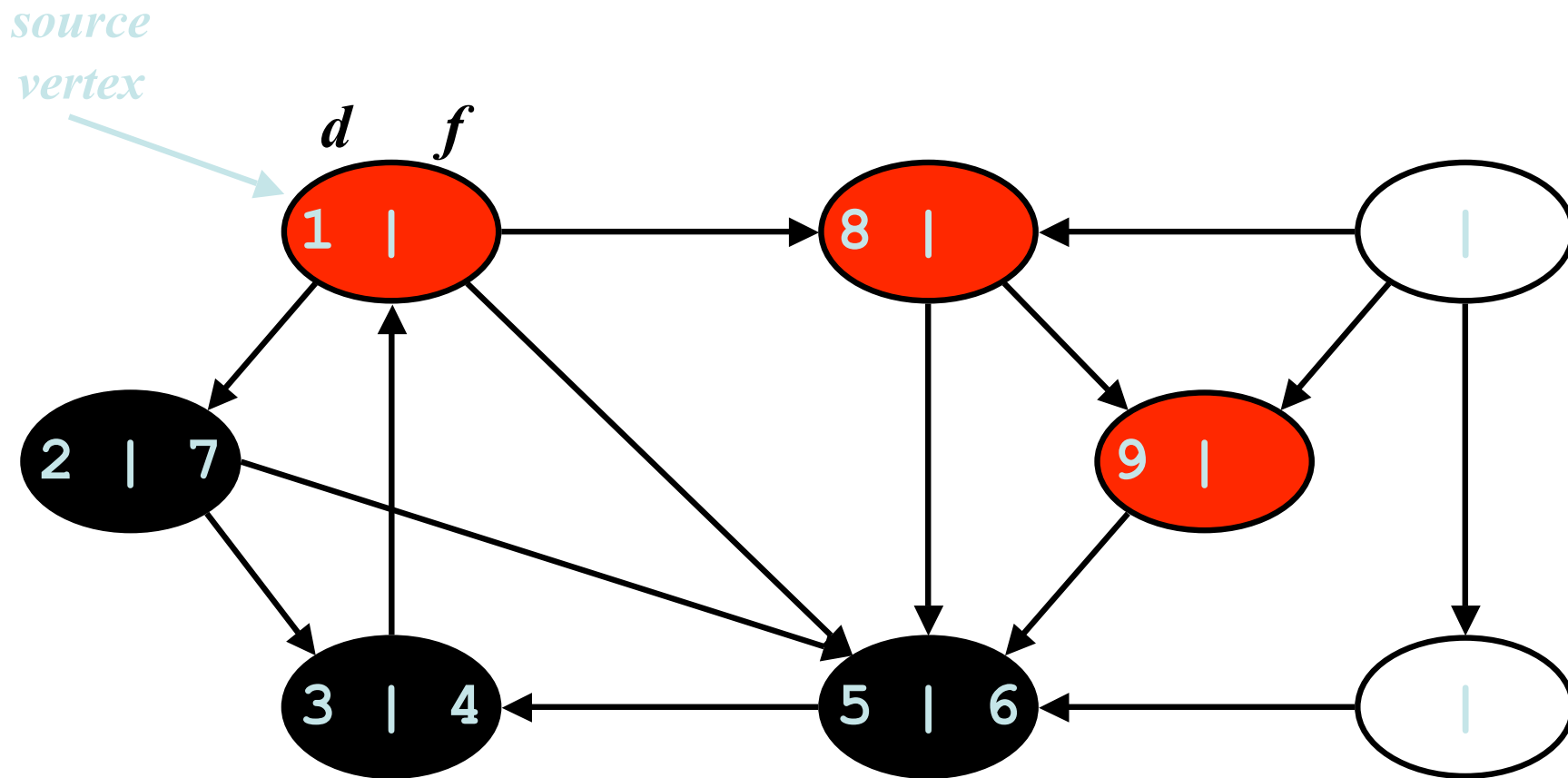
DFS Example



DFS Example

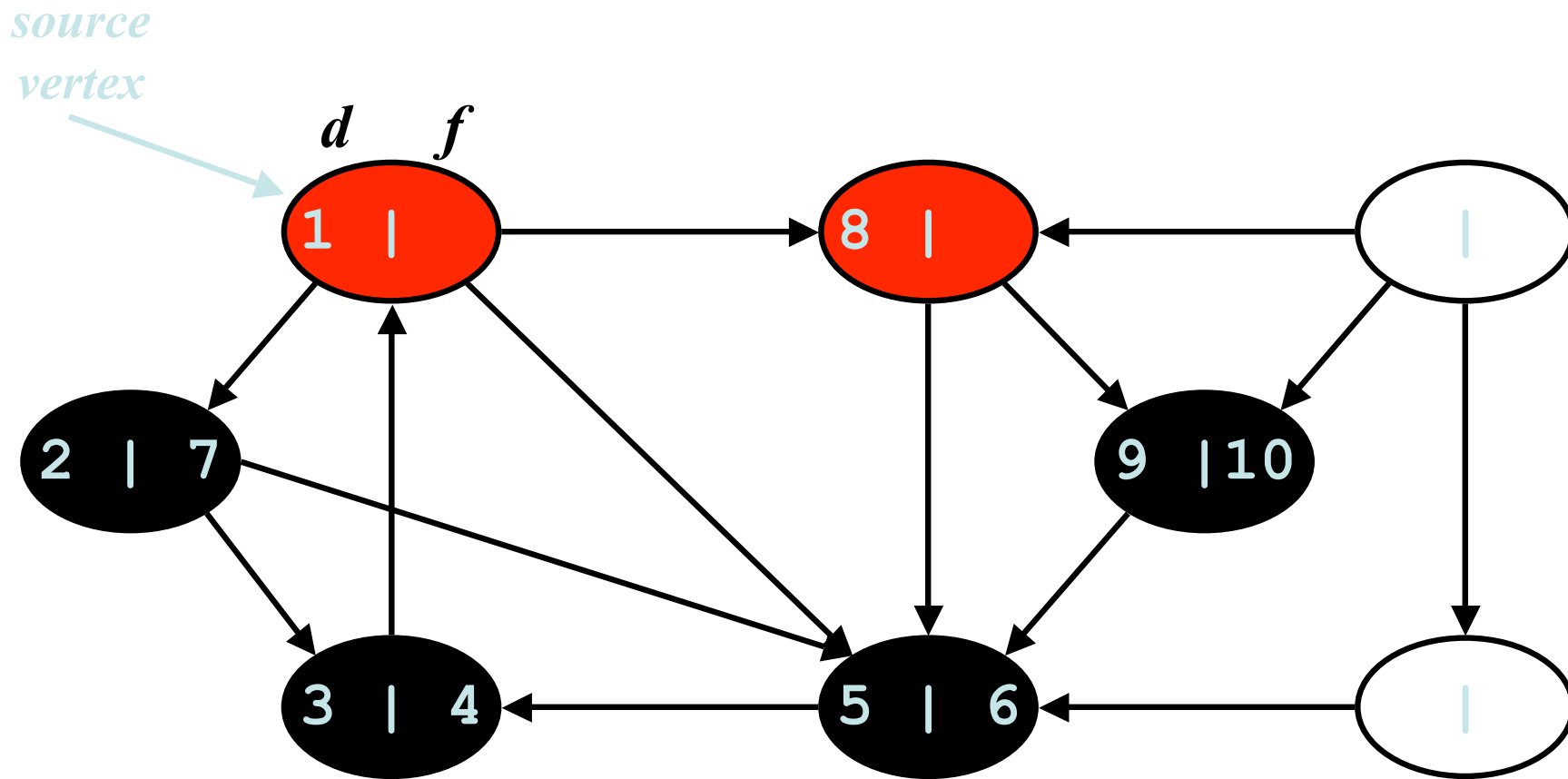


DFS Example

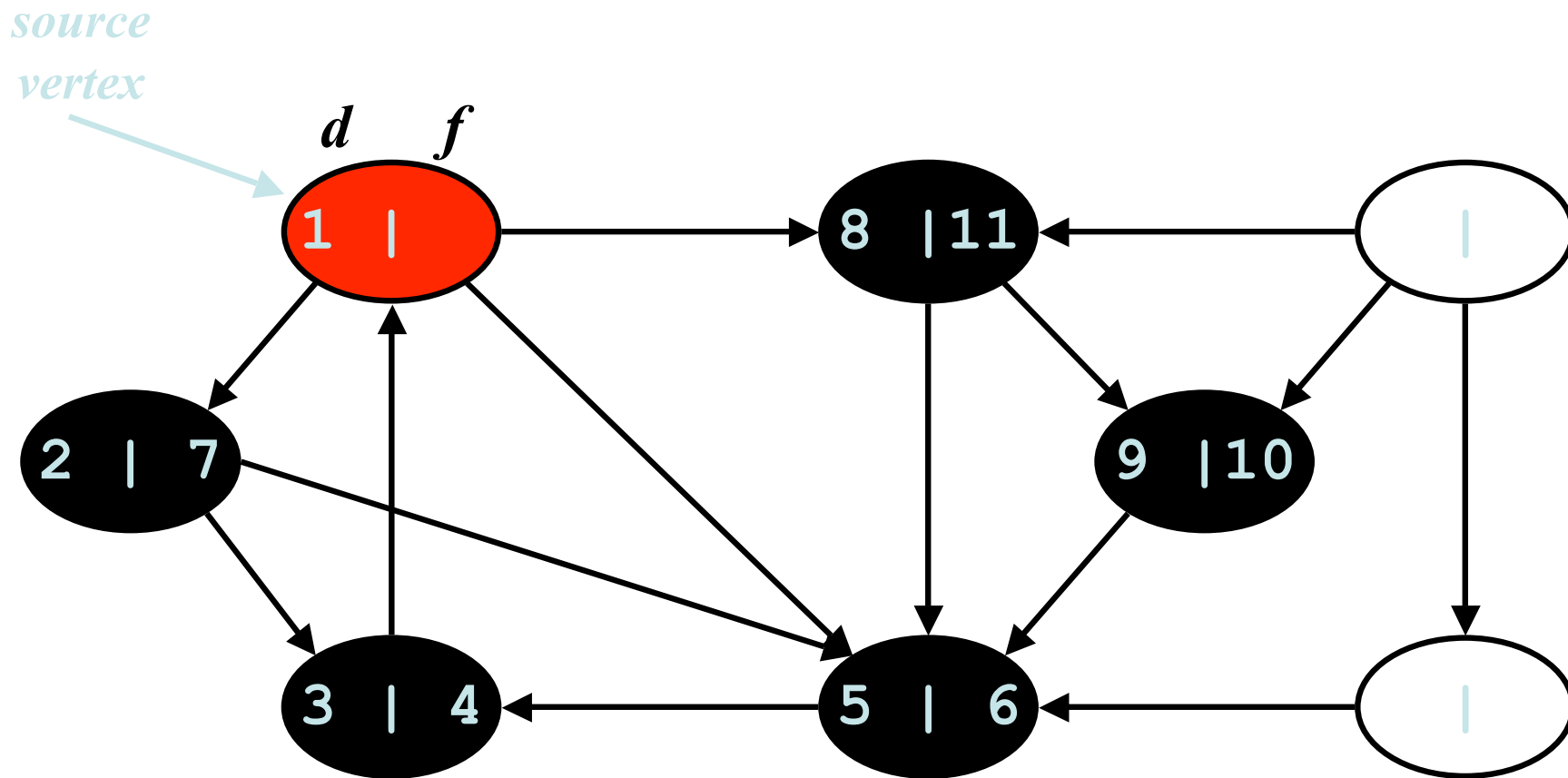


*What is the structure of the grey vertices?
What do they represent?*

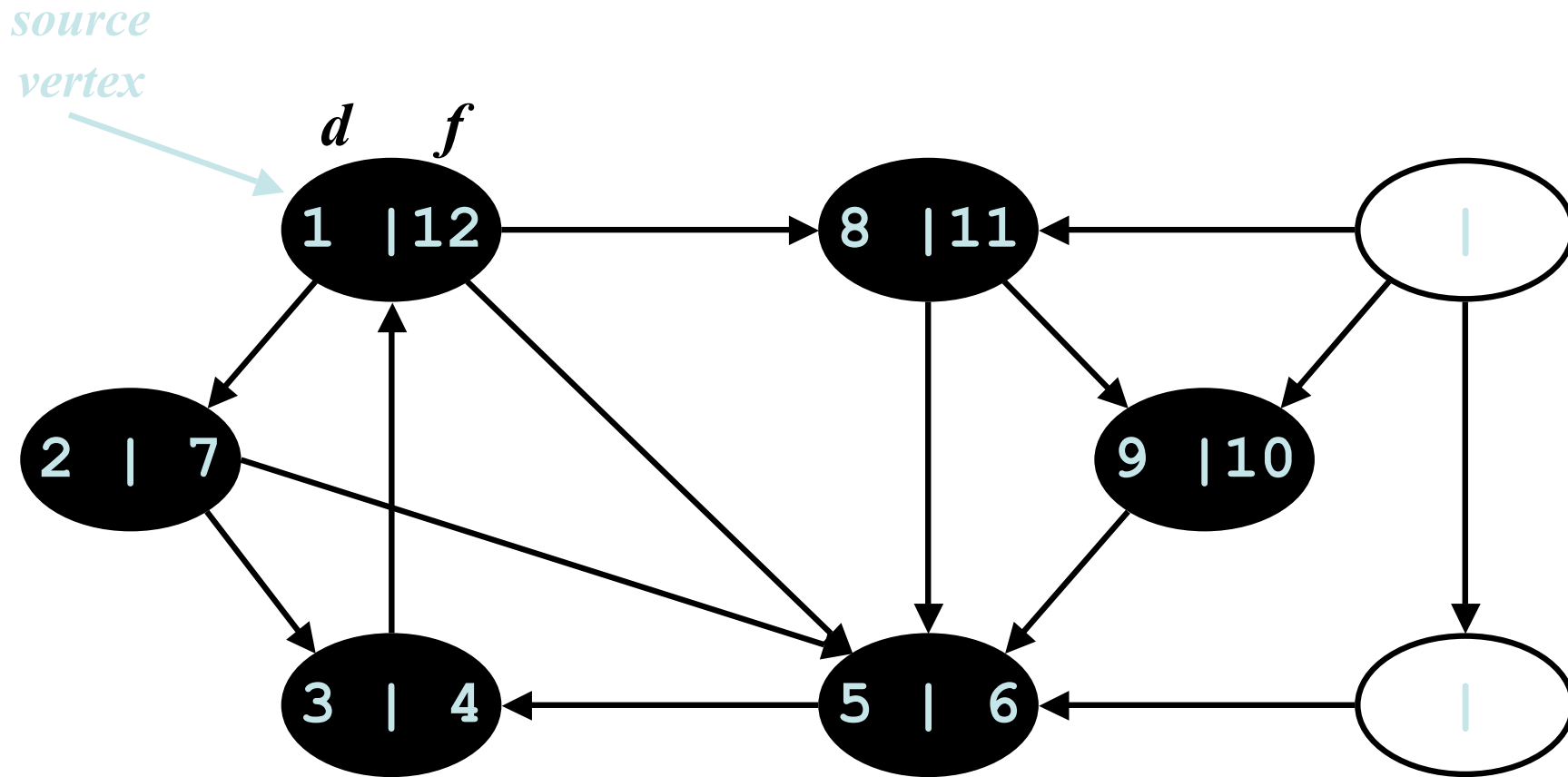
DFS Example



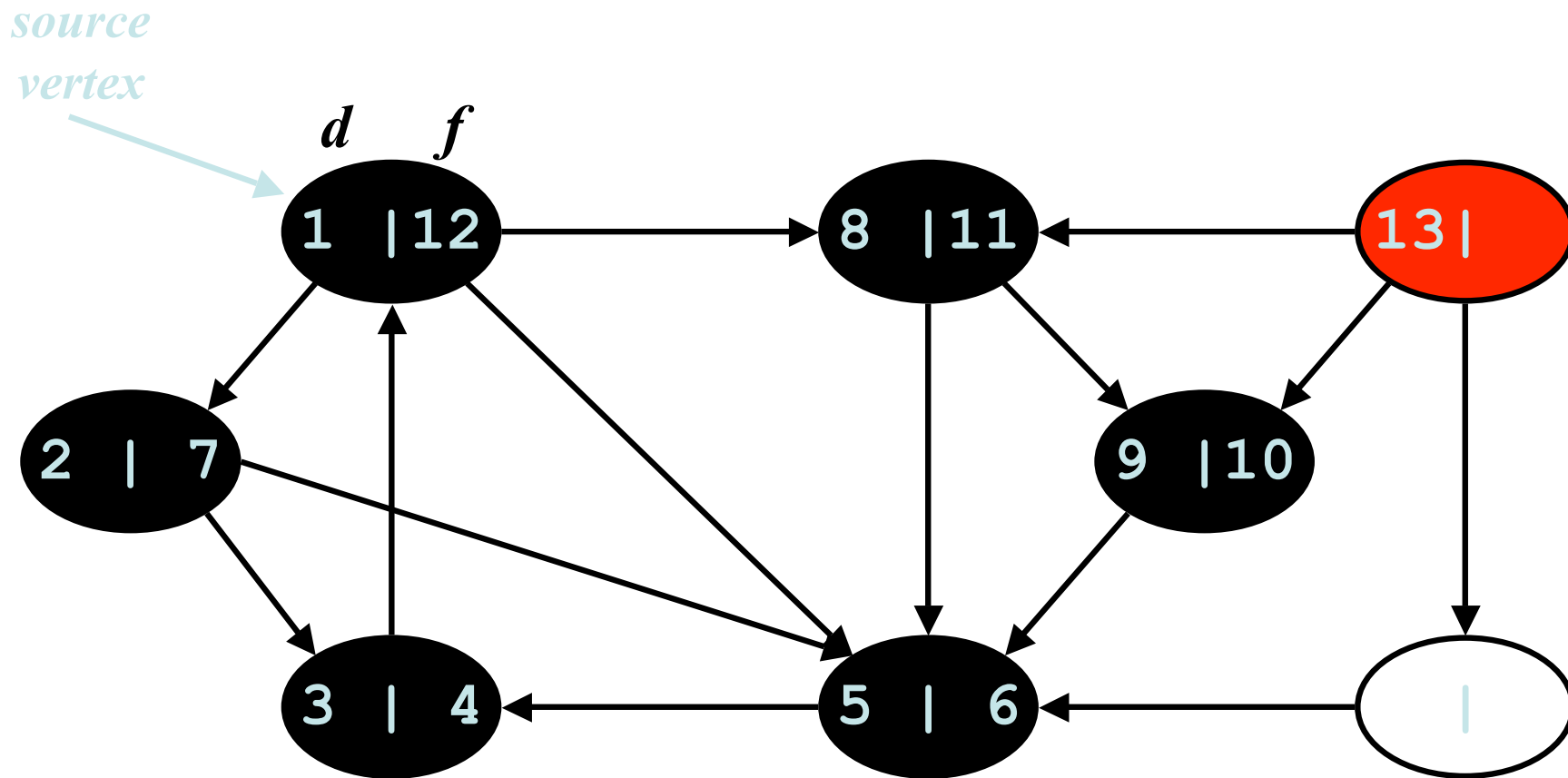
DFS Example



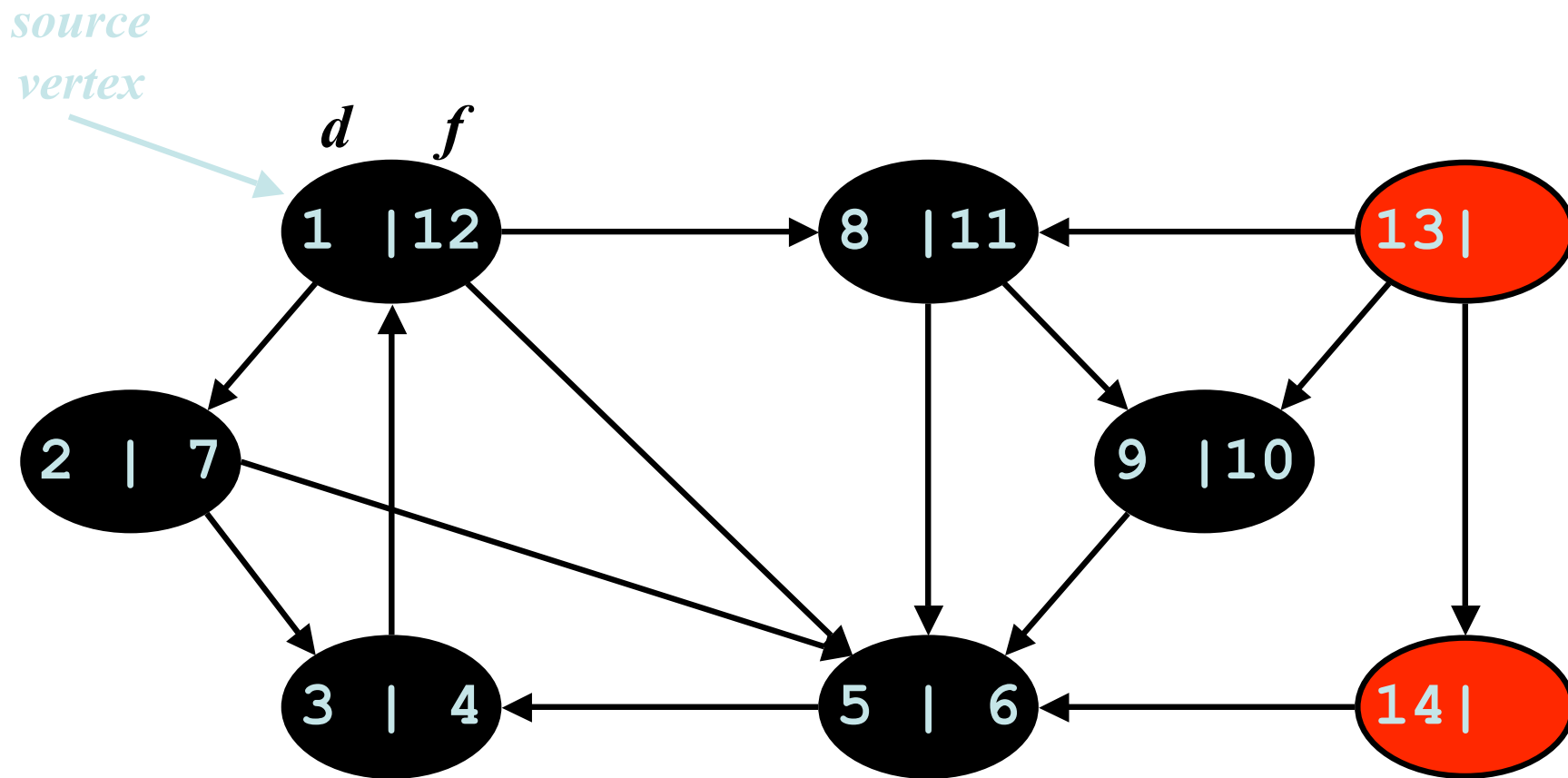
DFS Example



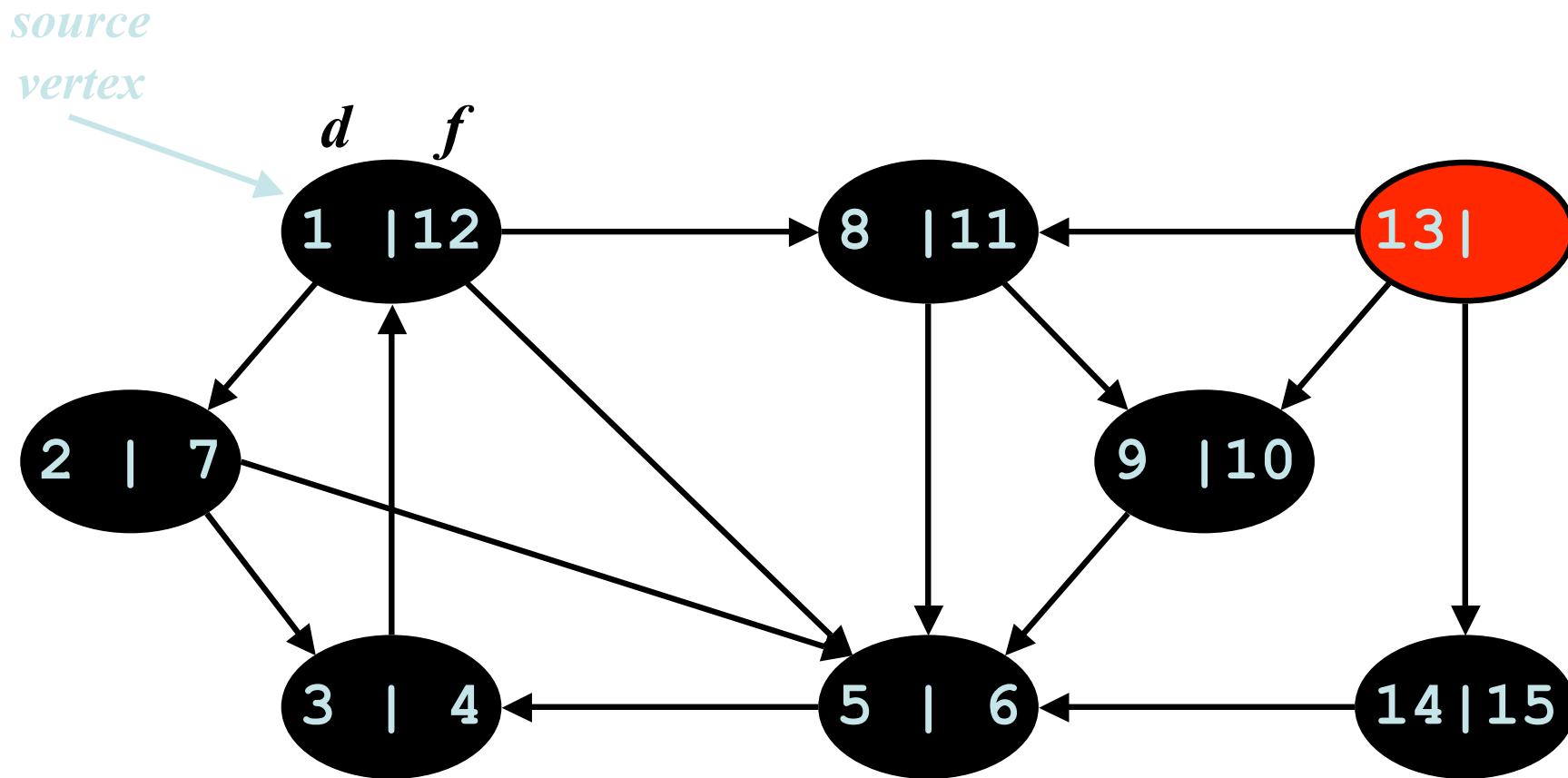
DFS Example



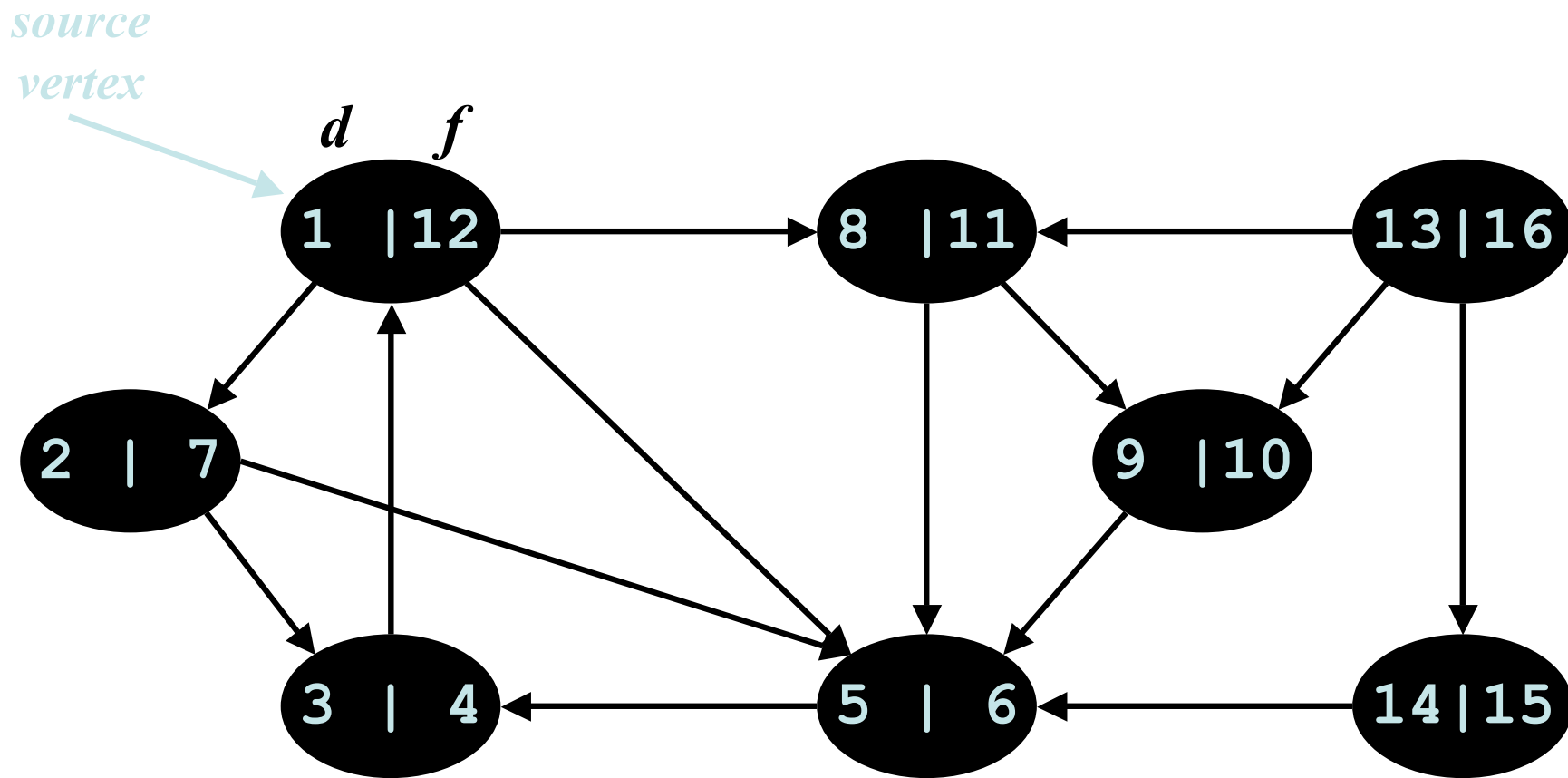
DFS Example



DFS Example



DFS Example



Depth-limited search

= depth-first search with depth limit l ,
i.e., nodes at depth l have no successors

- Recursive implementation:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result ← RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail-  
ure  
  inputs: problem, a problem  
  for depth  $\leftarrow$  0 to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)  
    if result  $\neq$  cutoff then return result
```

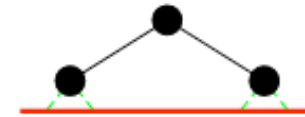
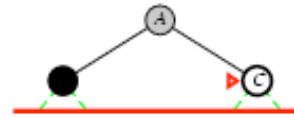
Iterative deepening search $l = 0$

Limit = 0



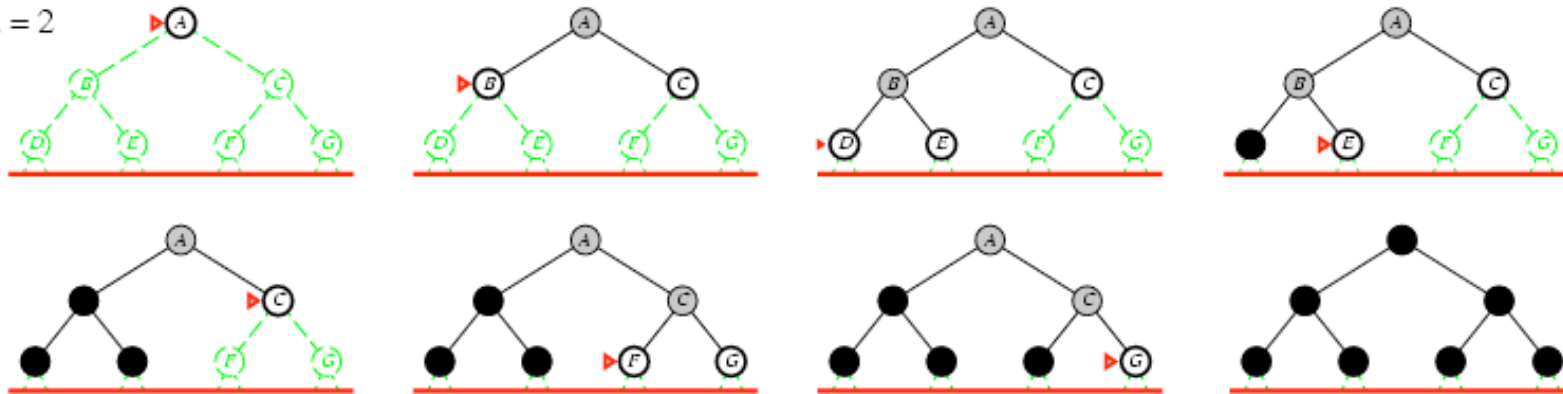
Iterative deepening search $l = 1$

Limit = 1



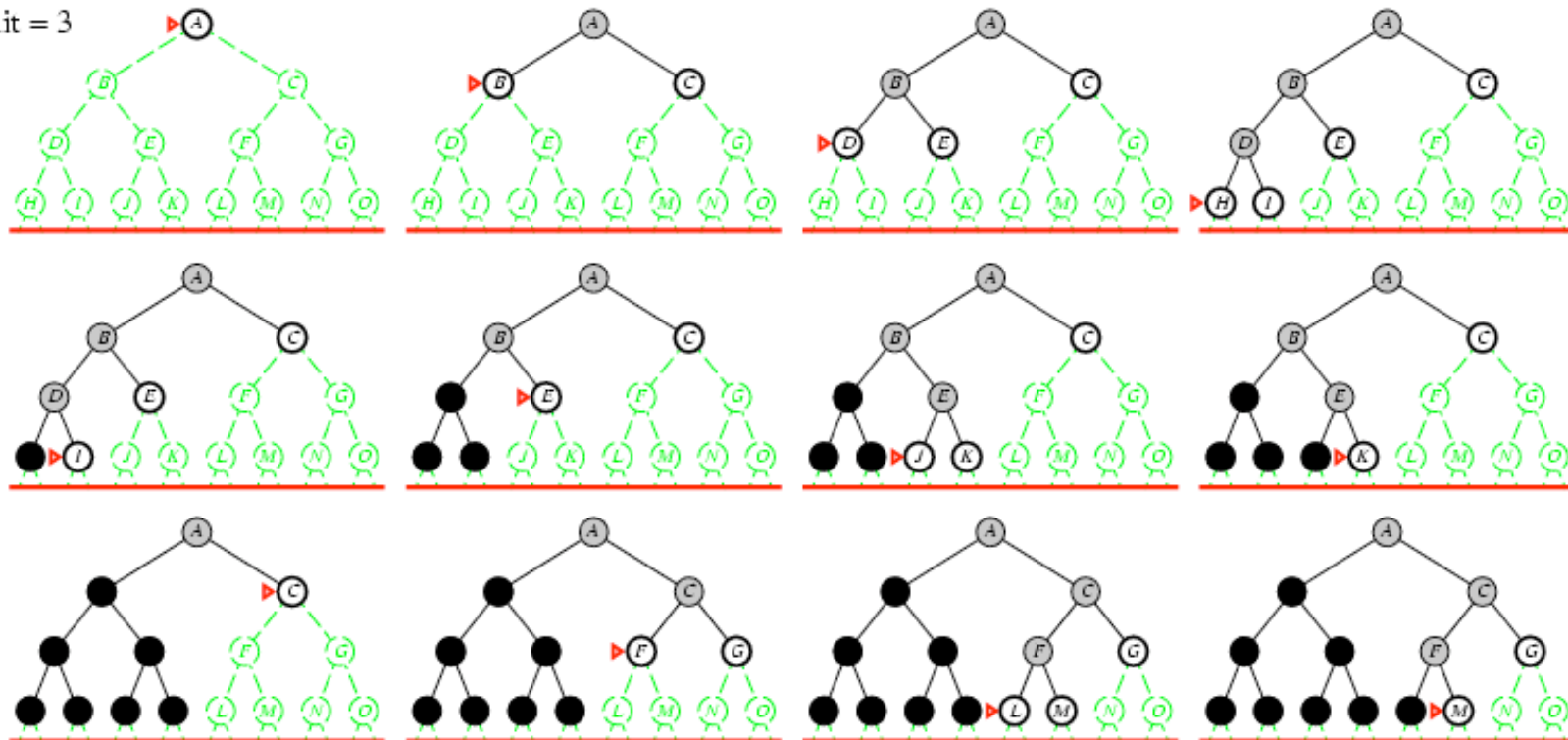
Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search / =3

Limit = 3



Properties of iterative deepening search

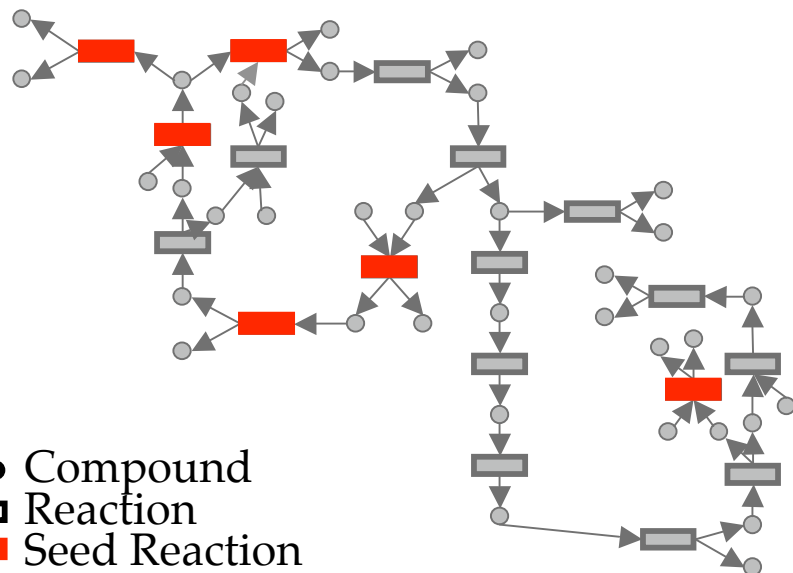
- Complete? Yes
- Time? $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

Summary of algorithms

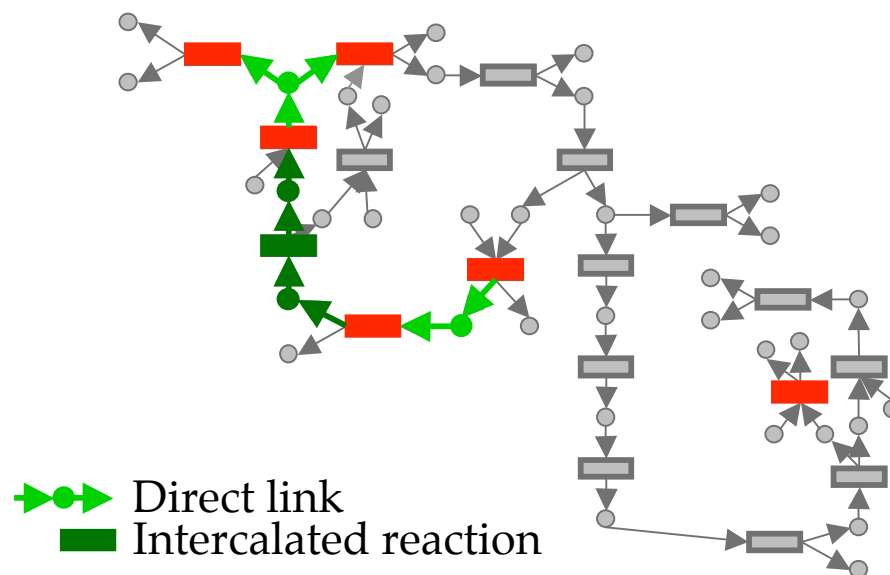
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Queries - subgraph extraction

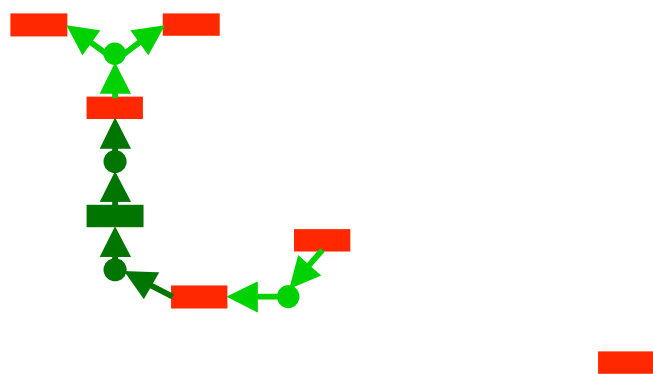
A. Seed reactions



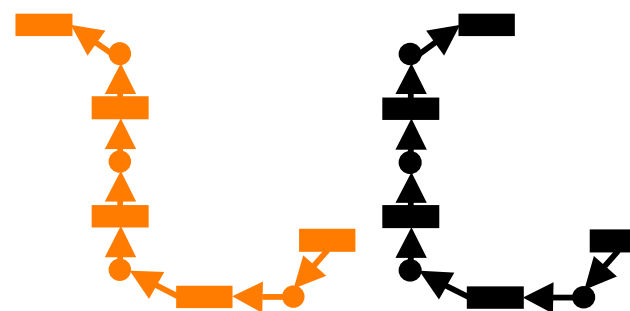
B. Reaction linking



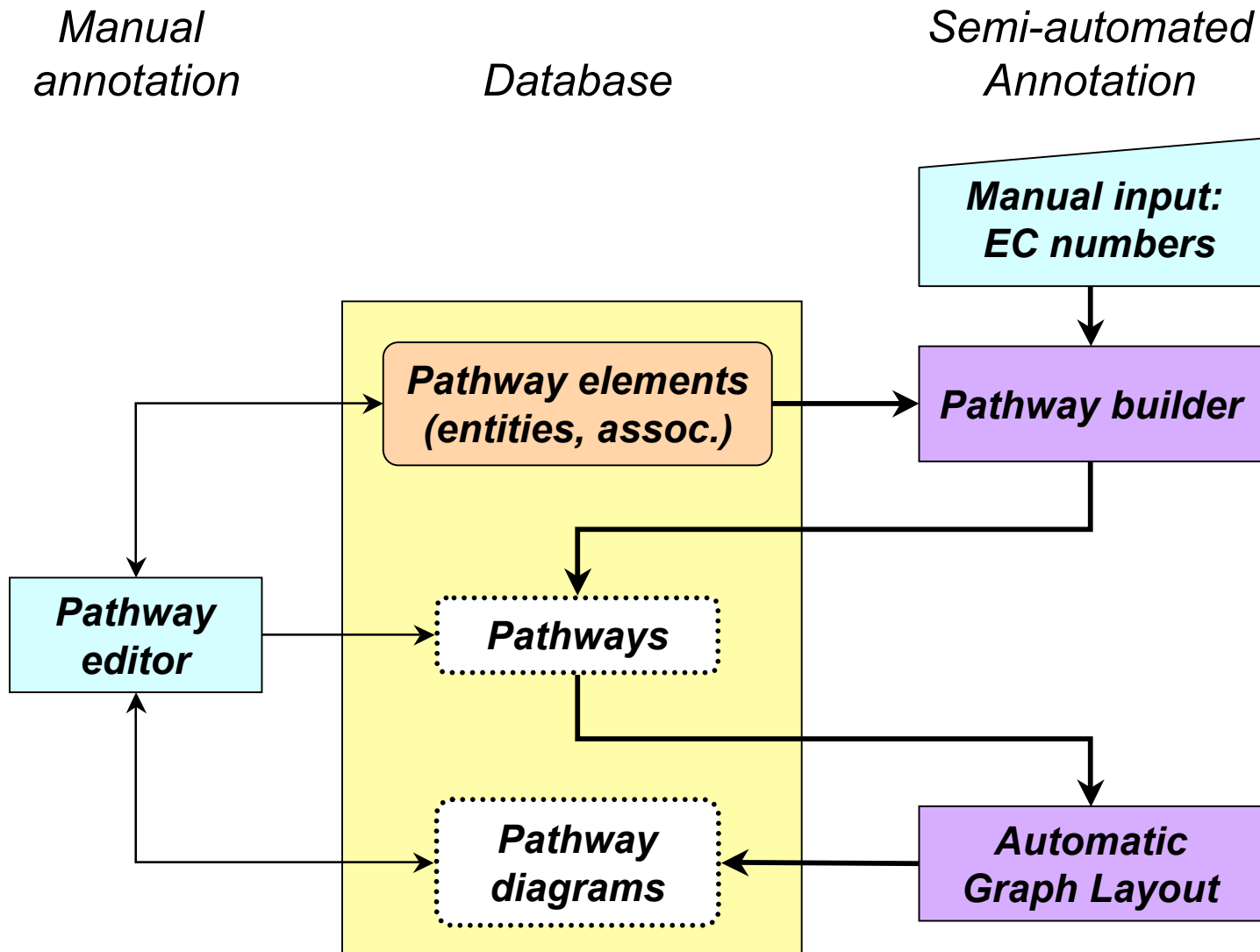
C. Subgraph extraction



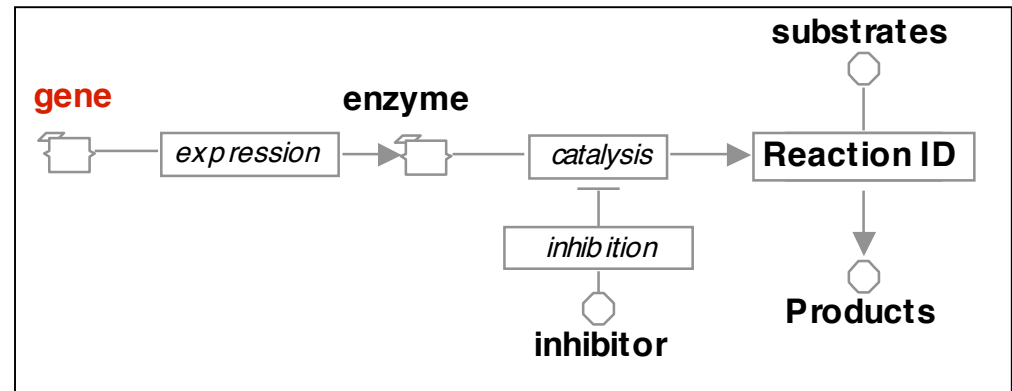
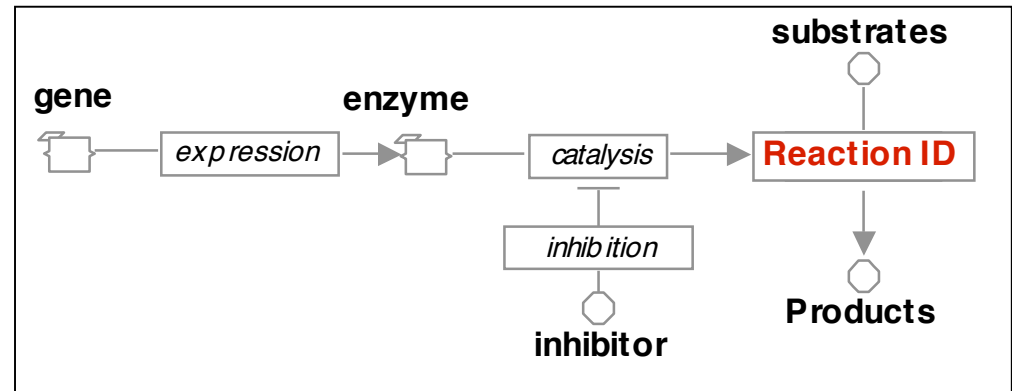
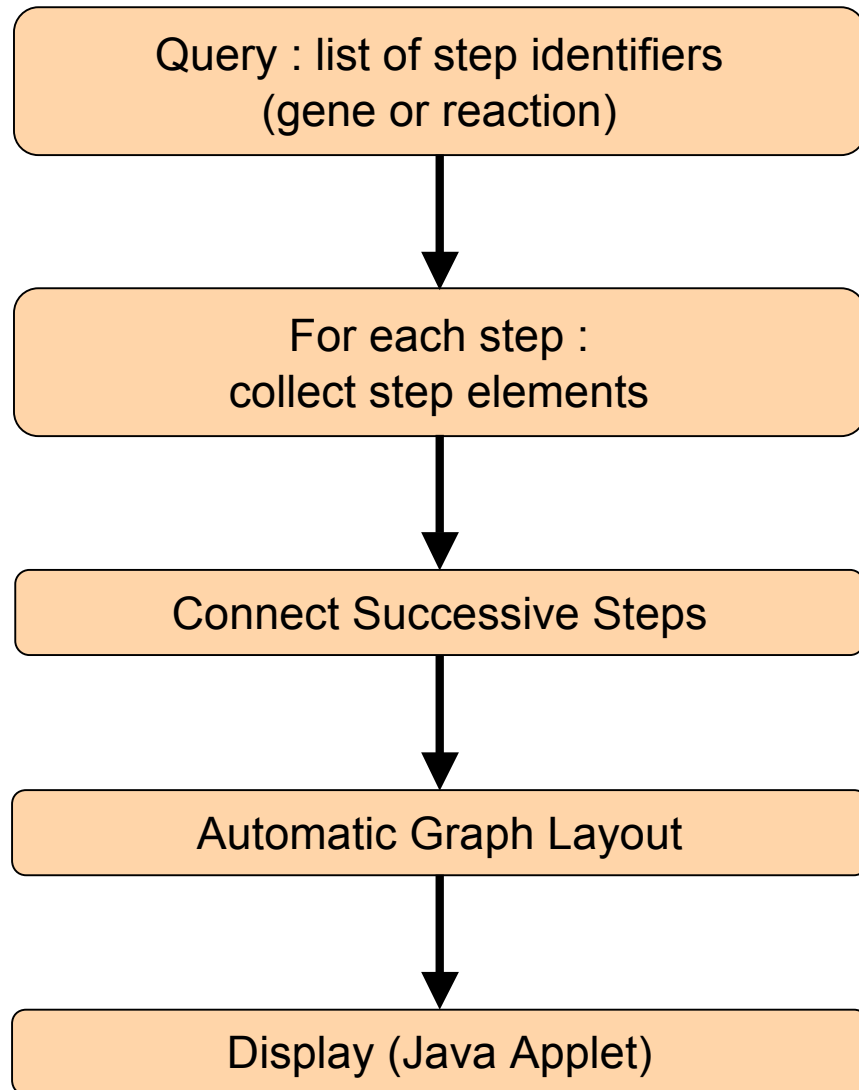
D. Linear Path Enumeration



Pathway Building : semi-automated annotation



Pathway builder program



Pathway builder result

Analysis of Regulatory Pathways - Netscape

File Edit View Go Communicator Help

Bookmarks Location: http://www.soi.city.ac.uk/~jvanheld/path_analysis/

Pathway Analysis

- [Project Home Page](#)
- [Warning](#)
- [Formula Listing](#)
- [Path finder](#)
- [Pathway Builder](#)
- [Data files](#)
- [Documentation](#)
- [Credits](#)
- [Feed-back](#)

Pathway graph builder result

```
graph LR; HDM3 --> Hom3p; Hom3p --> R2724[2.7.2.4]; R2724 --> LAspartate; R2724 --> ATP; R2724 --> ADP; R2724 --> P4LAspartate[4-Phospho-L-aspartate]; HDM2 --> Hom2p; Hom2p --> R12111[1.2.1.11]; R12111 --> NADPH; R12111 --> Orthophosphate; R12111 --> NADPplus[NADP+]; R12111 --> LAspartate4semialdehyde[L-Aspartate 4-semialdehyde]; HDM6 --> Hom6p; Hom6p --> R1113[1.1.1.3]; R1113 --> NADH_NADPH[NADH or NADPH]; R1113 --> NADplus_NADPplus[NAD+ or NADP+]; R1113 --> LHomoserine[L-Homoserine]; MET2 --> Met2p; Met2p --> R23131[2.3.1.31]; R23131 --> AcetylCoA[Acetyl-CoA]; R23131 --> CoA; R23131 --> OAcetylLhomoserine[O-Acetyl-L-homoserine];
```

Show edge labels

Applet NetGraph running

Further graph operations

- Sub-graph matching
 - (sub-graph isomorphism)
 - Pattern (graph motif) matching
- Graph comparison
 - (largest common subgraph)
- Both can be *weighted* (how?)

Summary

- Graphs intro
- Definition
- Paths, circuits, searching
- Breadth-first search
- Depth-first search