Complexity Theory and Algorithms

Graph Algorithms

DISCLAIMER

The presentation is an amalgamation of information obtained from textbooks and different internet resources and is intended for educational purposes only and does not replace independent professional judgement, statements of fact and opinions.

BOOKS

List of Books Referred.

1. Gilles Brassard and Paul Bratley, Fundamentals of Algorithmics, PHI Publication.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein, Introduction to Algorithms, PHI Publication.
3. Ellis Horowitz, Sartaj Sahni,Sanguthevar Rajasekaran, Fundamentals of Computer Algorithms, University Press.

Topics Covered

- Introduction to Graph Theory
- BFS
- DFS
- Backtracking
- Branch and Bound

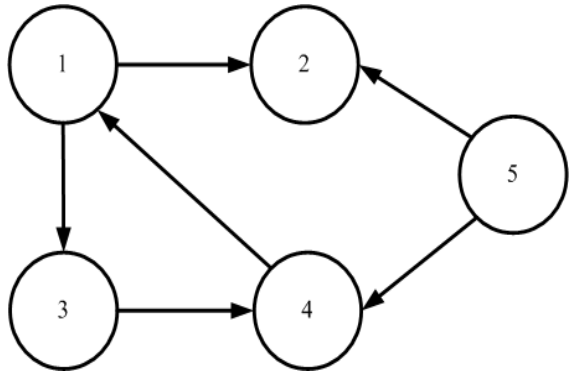
Introduction to GraphTheory

Graphs:Review

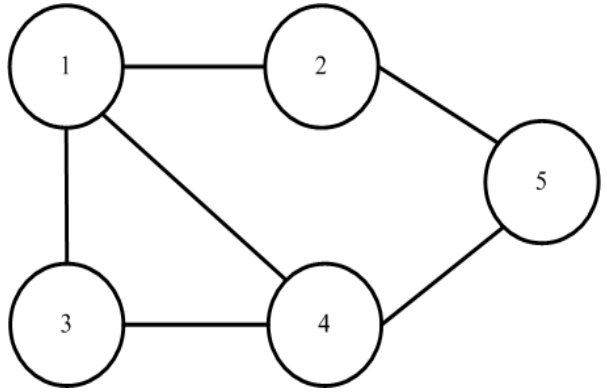
- A graph
  - A graph G consists of a nonempty set V called the set of nodes (points or vertices) of the graph and set E which is set of edges of the graph and a mapping from the set of edges E to a set of pairs of elements of V.
  - Denoted by G = (V, E)
    - V = set of vertices, E = set of edges
  - *Dense* graph: $|E| \approx |V|^2$; *Sparse* graph: $|E| \approx |V|$

Basic Types of Graph

- Directed graph:
  - If every edge (i,j) in E(G) of graph G is marked by a direction from i to j then the graph is called **directed graph**. It is also called **digraph.**

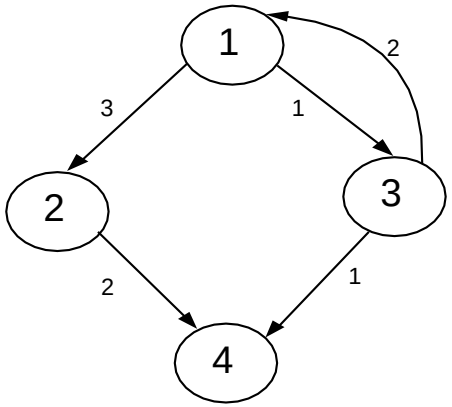


- Undirected graph:
  - A graph in which every edge is undirected is called undirected graph. i.e. the edges are bidirectional.
  - edge (u,v) = edge (v,u)



Basic Types of Graph

- A *weighted graph* associates weights with either the edges or the vertices



Graph Terminology

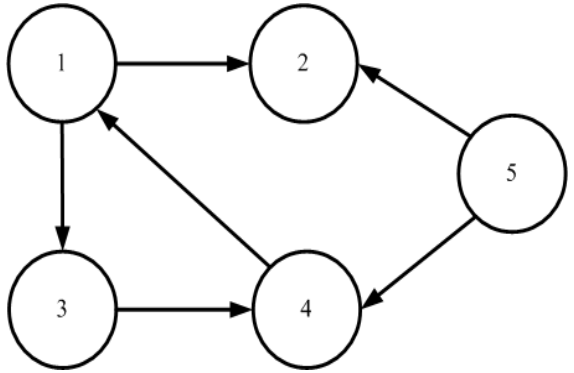
- The **degree** of a vertex in an **undirected graph** is the number of edges that leave/enter the vertex.
- The **degree** of a vertex in a **directed graph** is the same, but we distinguish between in-degree and out-degree. Degree = in-degree + out-degree.

Representing Graphs

- Adjacency-Matrix
  - Consider Graph G(V,E) and Assume V = {1, 2, …, *n*}
  - An adjacency matrix represents the graph as a *n* x *n* matrix A:

    A[*i*, *j*] = 1 if edge $(i, j) \in E$ (or weight of edge)

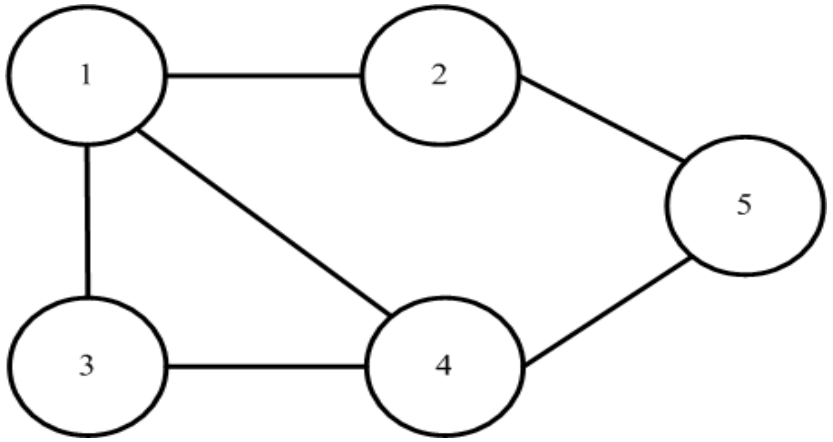
    = 0 if edge $(i, j) \notin E$



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

Representing Graphs

Adjacency-Matrix



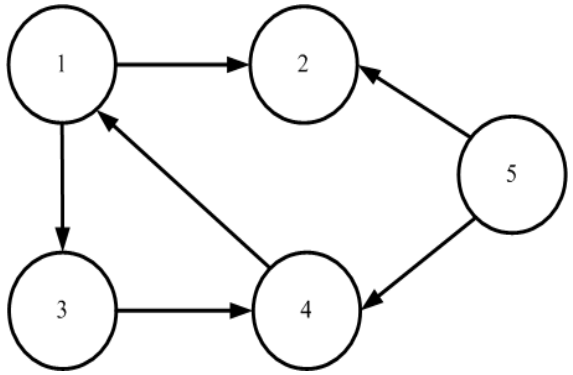
	1	**2**	**3**	**4**	**5**
1	0	1	1	1	0
2	1	0	0	0	1
3	1	0	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

- Storage requirements: $\Theta(V^2)$
  - Efficient if a graph is dense
  - Undirected graph: only need to store upper or lower triangular matrix

Representing Graphs

Adjacency-List

There is a one list for each node (vertex) of graph G and each list contains adjacent nodes.



1	→	**2**	**3**	\
2	→	\		
3	→	4	\	
4	→	1	\	
5	→	2	4	\

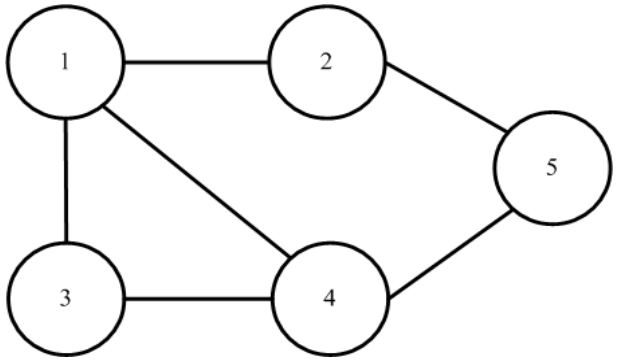
If G is directed graph, the sum of length of all adjacency lists is|E| and for undirected graph it's 2|E|.

Amount of memory requires(for directed or undirected) is

Θ (V + E)

Representing Graphs

Adjacency-List



1	→	2	3	4	\
2	→	1	5	\	
3	→	1	4	\	
4	→	1	3	5	\
5	→	2	4	\	

If G is undirected graph, the sum of length of all adjacency lists is 2|E|.

Amount of memory requires(for directed or undirected) is

Θ (V + E)

Representing Graphs

- Summary
  - Adjacency matrix representation takes $\Theta(V^2)$ regardless of number of edges in the graph.
  - Whereas space requirement for Adjacency list representation
    - is $\Theta (V + E)$
  - It is easy to find all vertices adjacent to a given vertex in an **adjacency list** representation. Simply read its adjacency list.
  - With an **adjacency matrix** you must instead scan over an entire row.
  - How ever it's easy to determine edge between two vertices with adjacency matrix.
  - Whereas **adjacency list** requires time proportional to the number of edges associated with the two vertices.

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

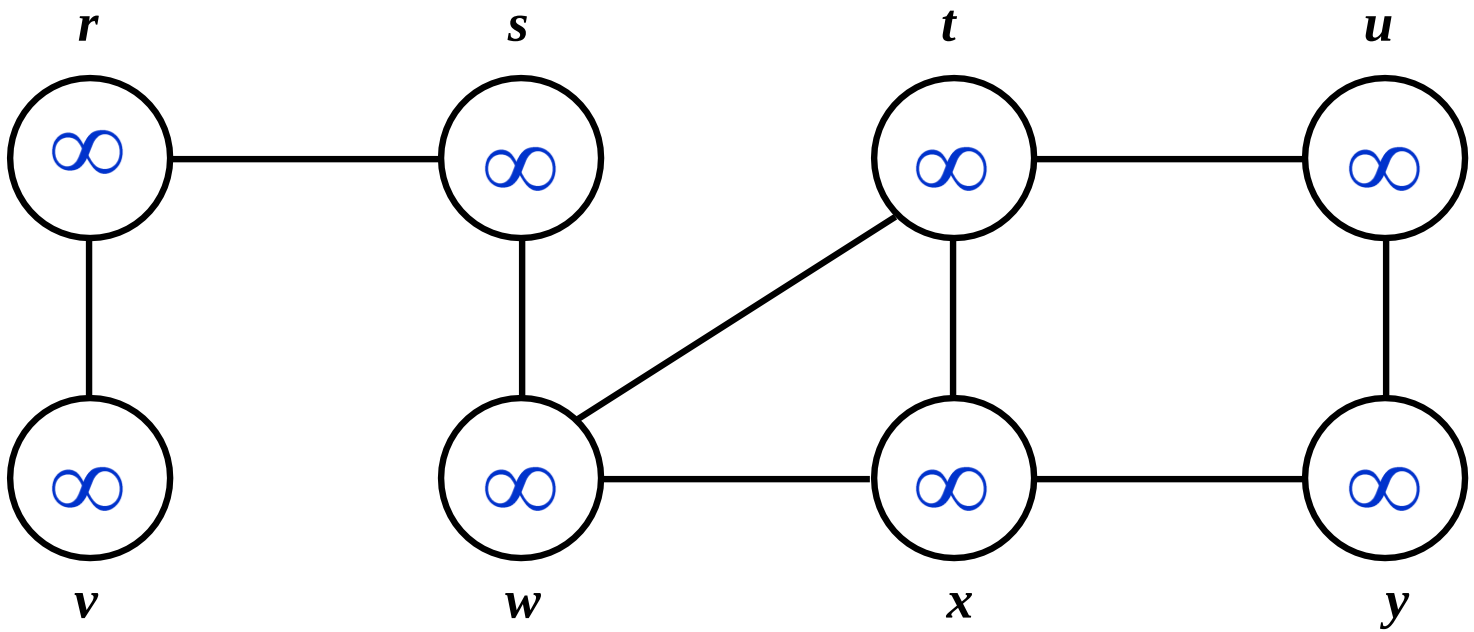
Breadth-First Search

- "Explore" a graph, turning it into a tree
  - The order of search is across levels.
  - The root is examined first; then children of the root ; then children of nodes visited in first phase. etc…

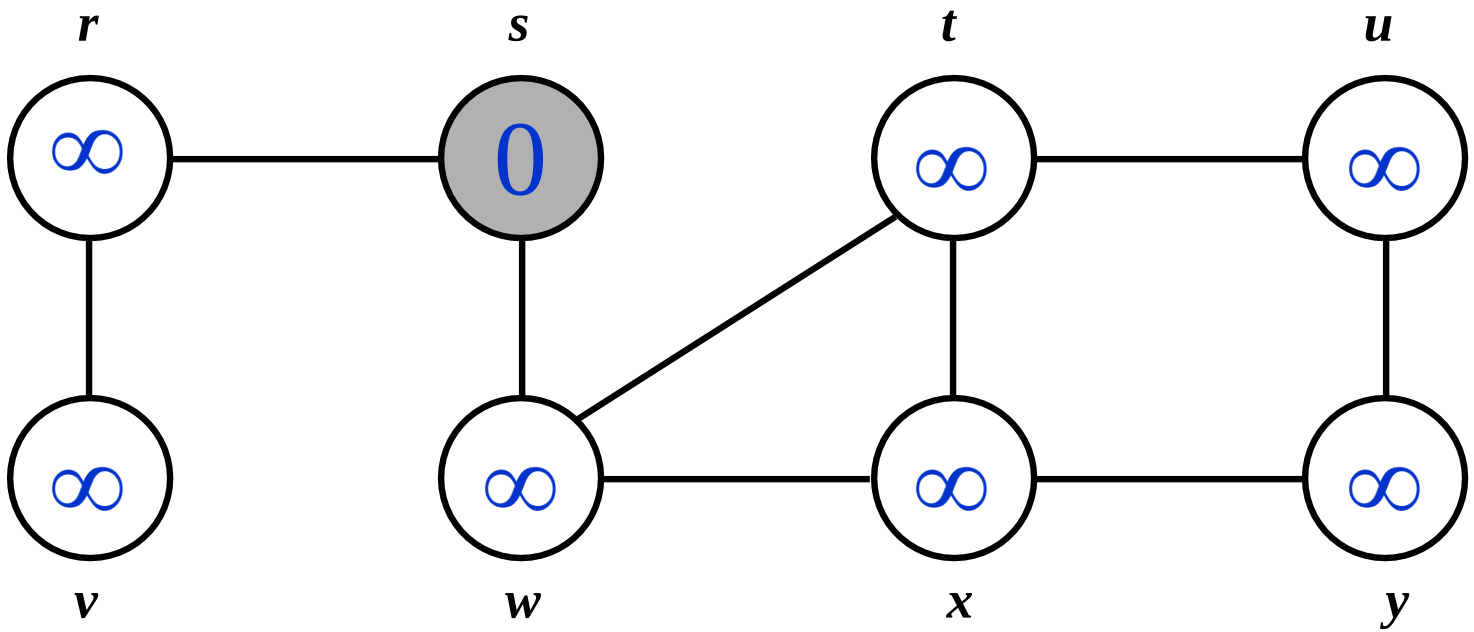
Breadth-First Search

- we will associate vertex “colors” to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

Breadth-First Search: Example

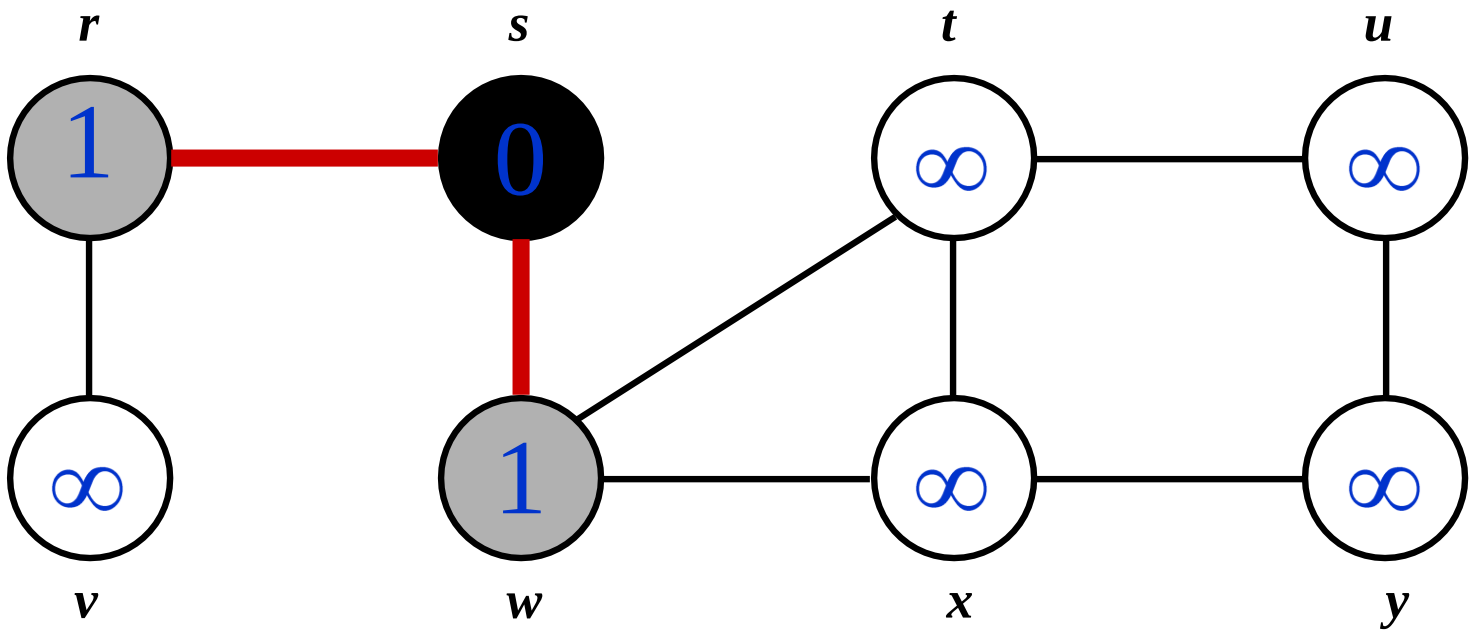


Breadth-First Search: Example



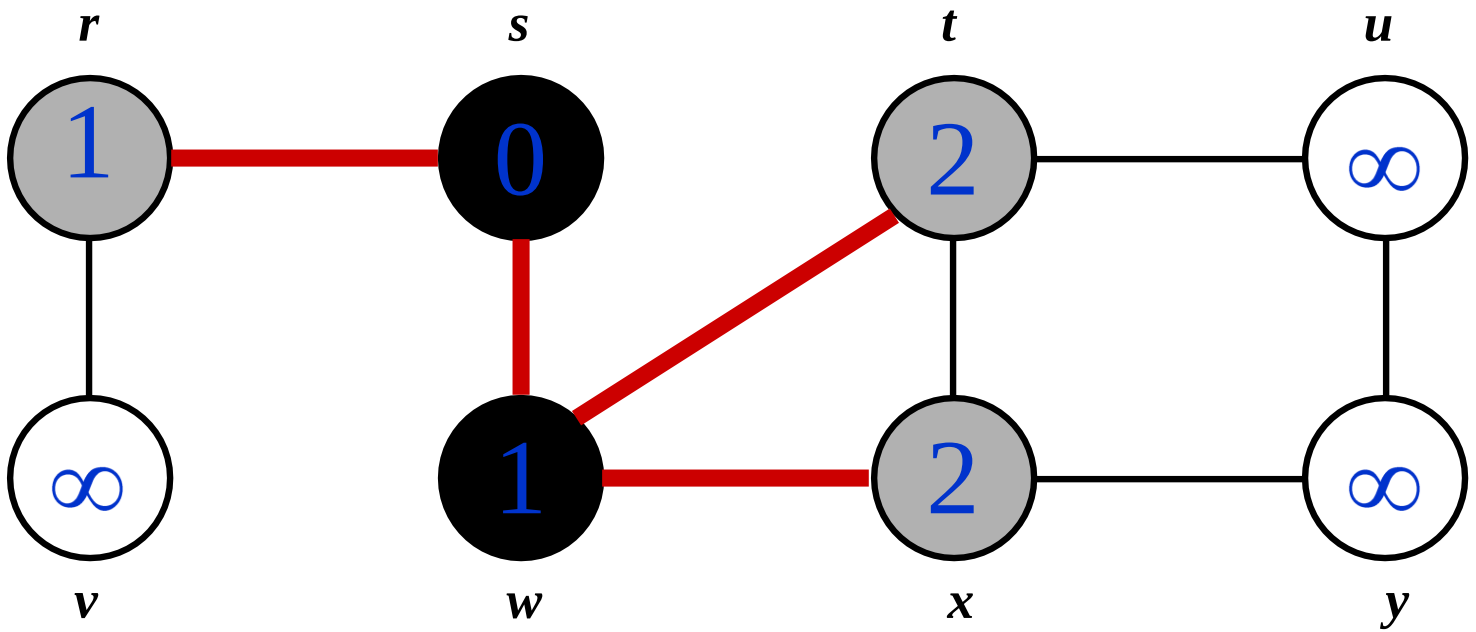
Q: $\boxed{s}$

Breadth-First Search: Example



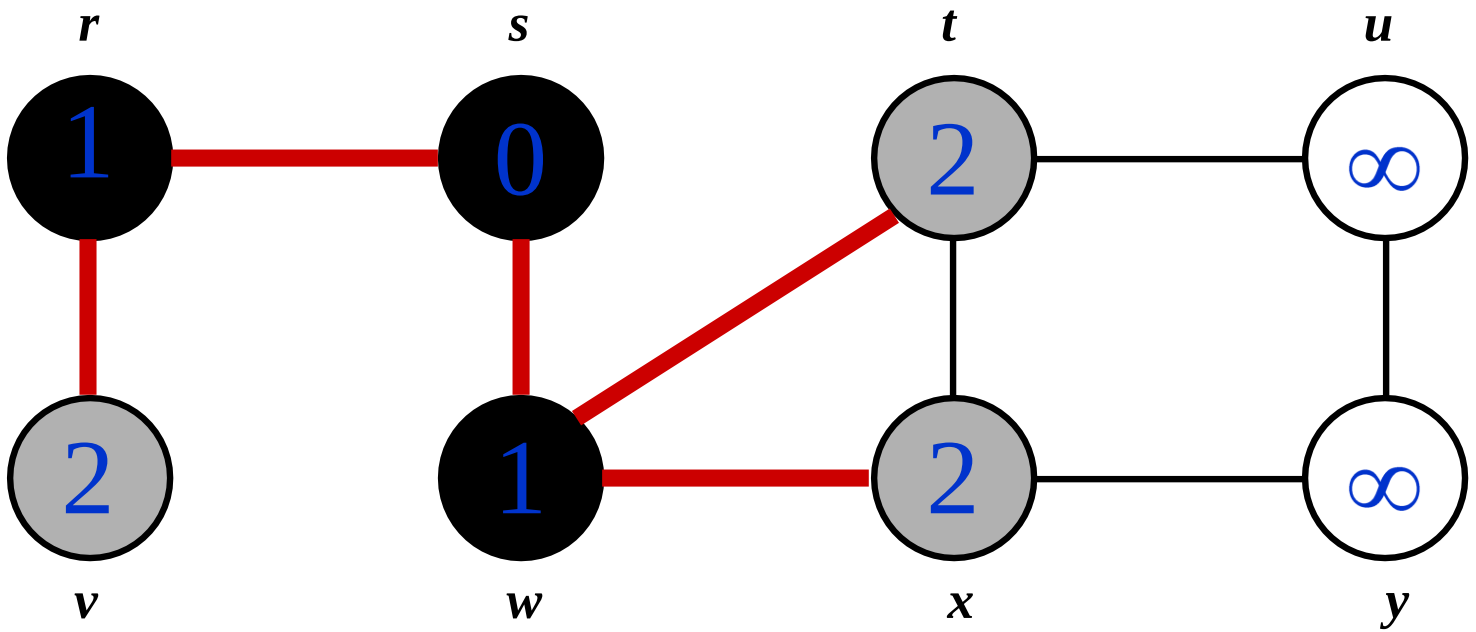
Q: | *w* | *r* |

Breadth-First Search: Example



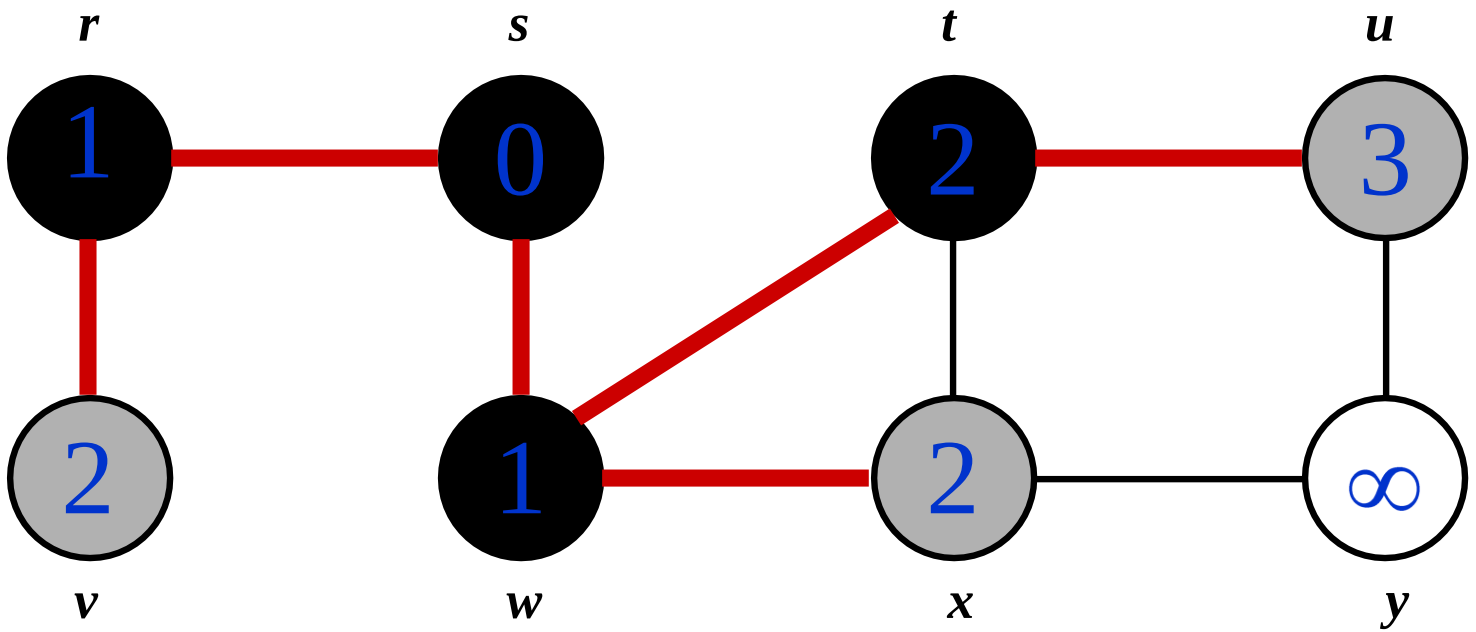
Q: | *r* | *t* | *x* |

Breadth-First Search: Example



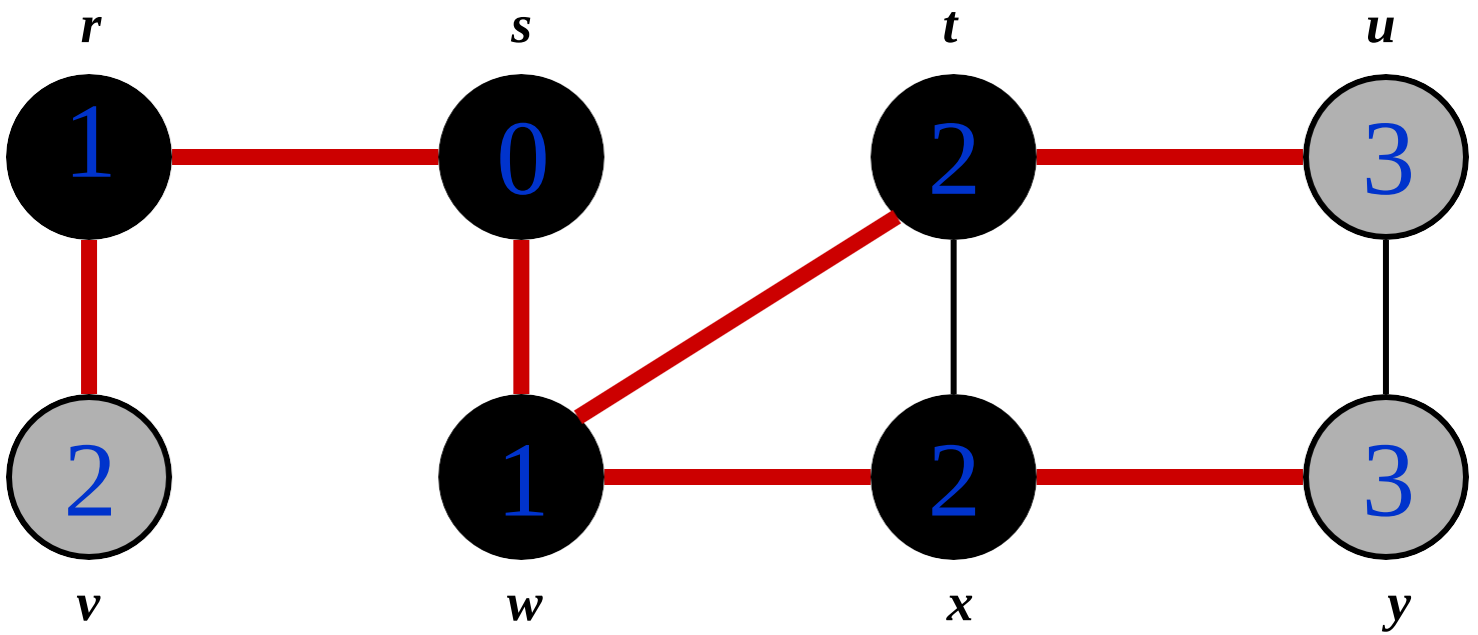
Q: | ***t*** | ***x*** | ***v*** |

Breadth-First Search: Example



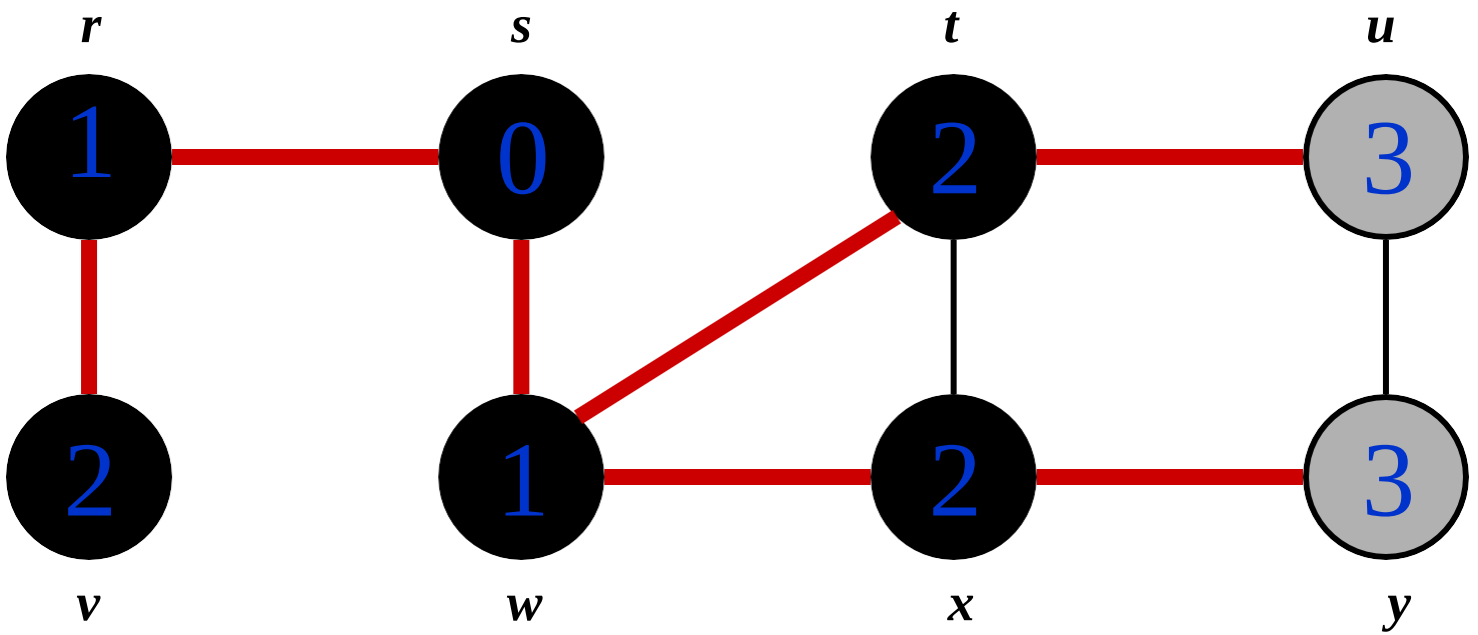
Q: | x | v | u |

Breadth-First Search: Example



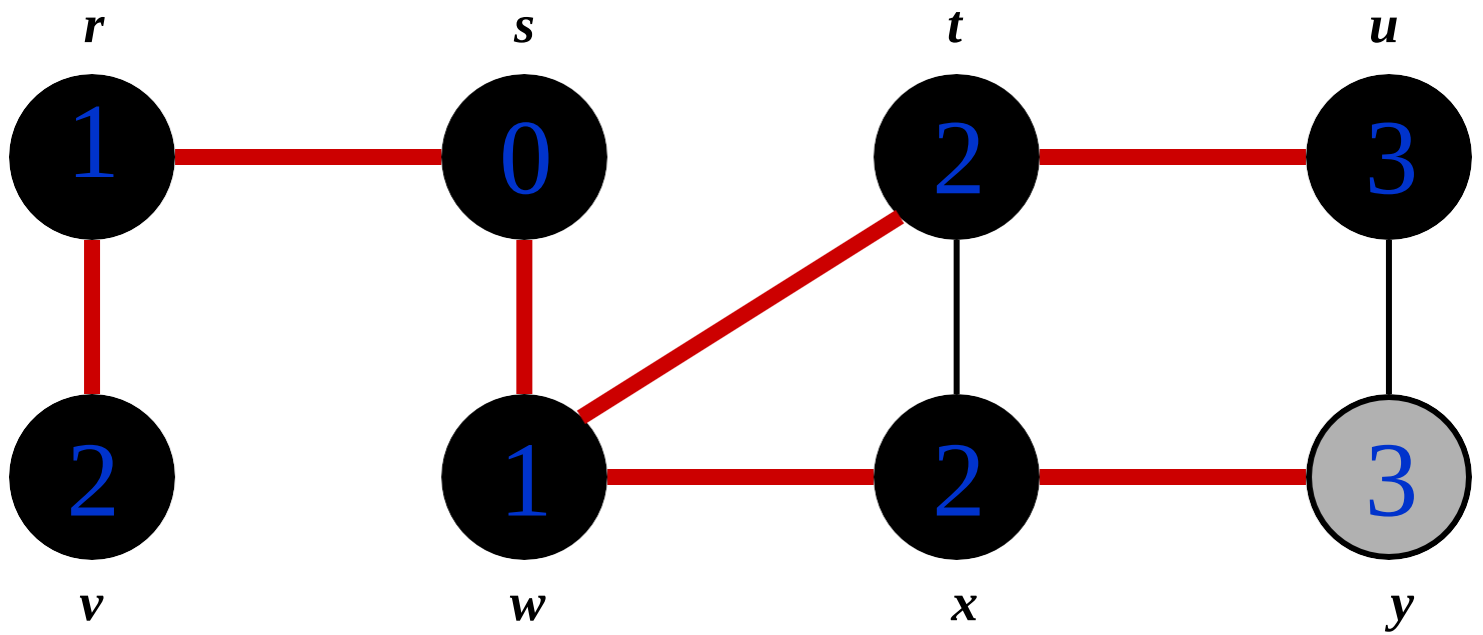
Q: | *v* | *u* | *y* |

Breadth-First Search: Example



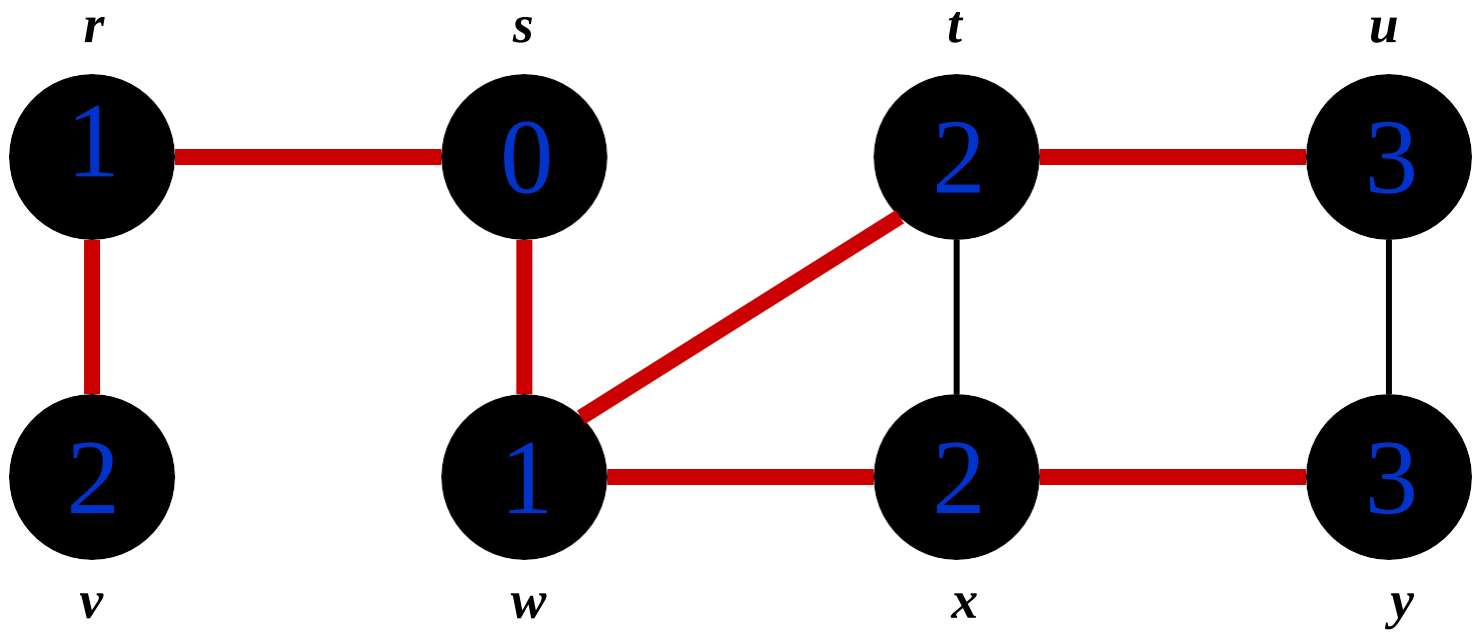
Q: | *u* | *y* |

Breadth-First Search: Example



Q: | *y* |

Breadth-First Search: Example



Q: ***Ø***

BFS: The Code

```
BFS(G, s) {
    initialize vertices;
    Q = {s};
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

Touch every vertex: O(V)

u = every vertex, but only once

So v = every vertex that appears in some other vert's adjacency list

What will be the running time?

Total running time: O(V+E)

Activate
Go to Sett

Depth-First Search

- *Depth-first search* is another strategy for exploring a graph
  - Explore "deeper" in the graph whenever possible
  - Edges are explored out of the most recently discovered
    - vertex *v* that still has unexplored edges
  - When all of *v*'s edges have been explored, backtrack to the vertex from which *v* was discovered

Depth-First Search

- Vertices initially colored white
- Then colored gray when discovered
- Then black when finished

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

What does `u->d` represent?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

What does `u->f` represent?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

Will all vertices eventually be colored black?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

What will be the running time?

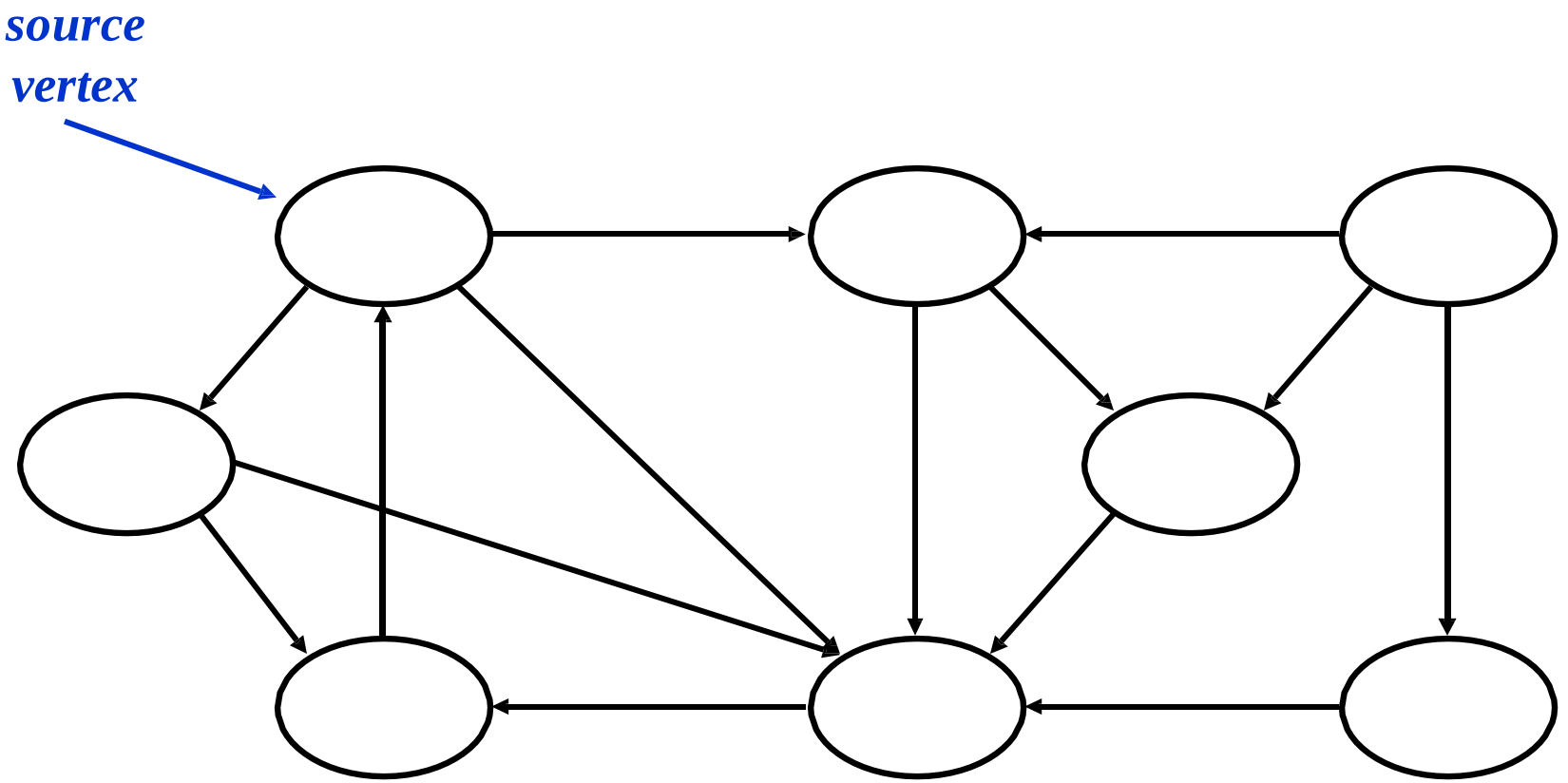
Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

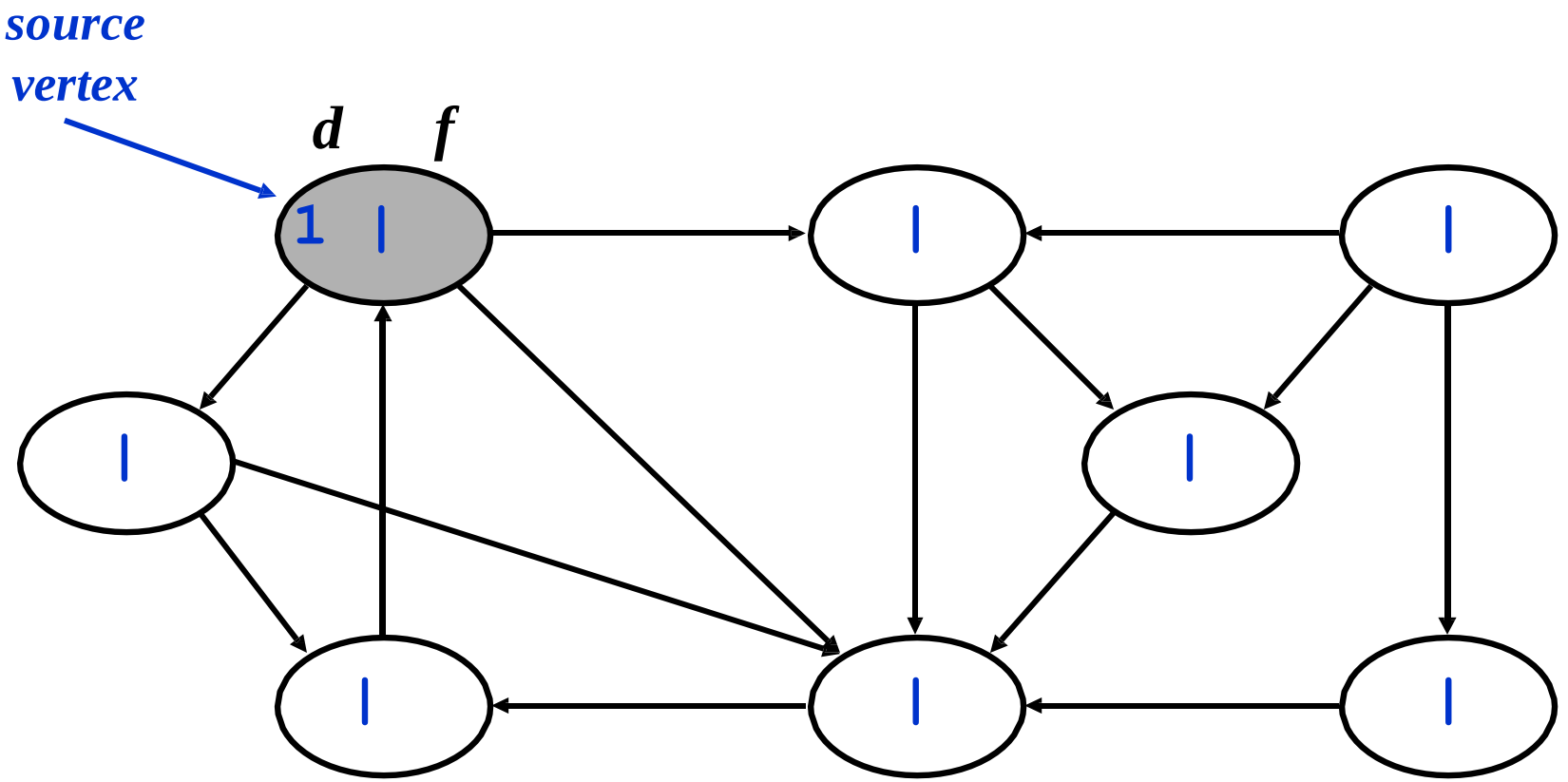
```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

So, running time of DFS = O(V+E)

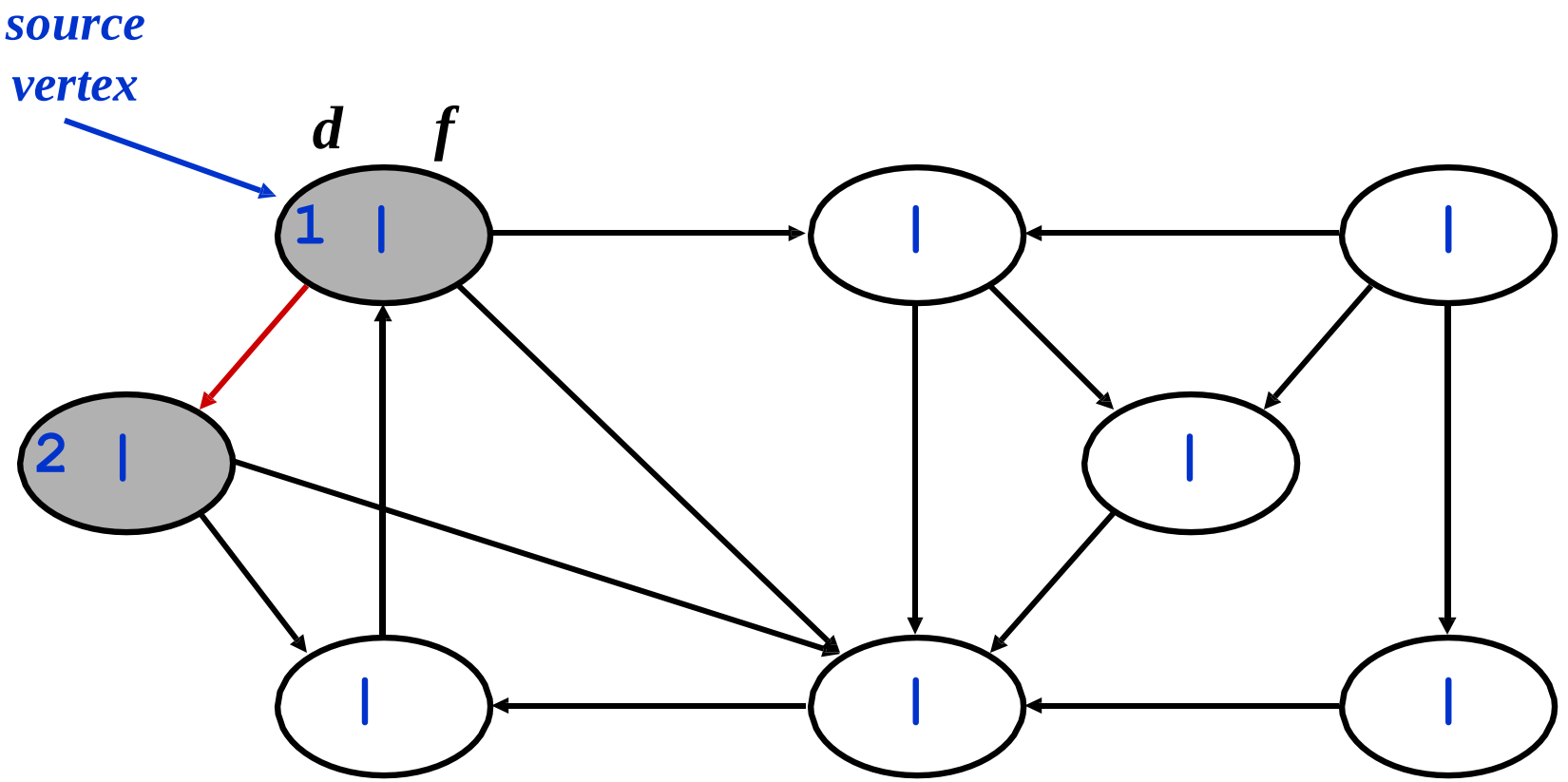
DFS Example



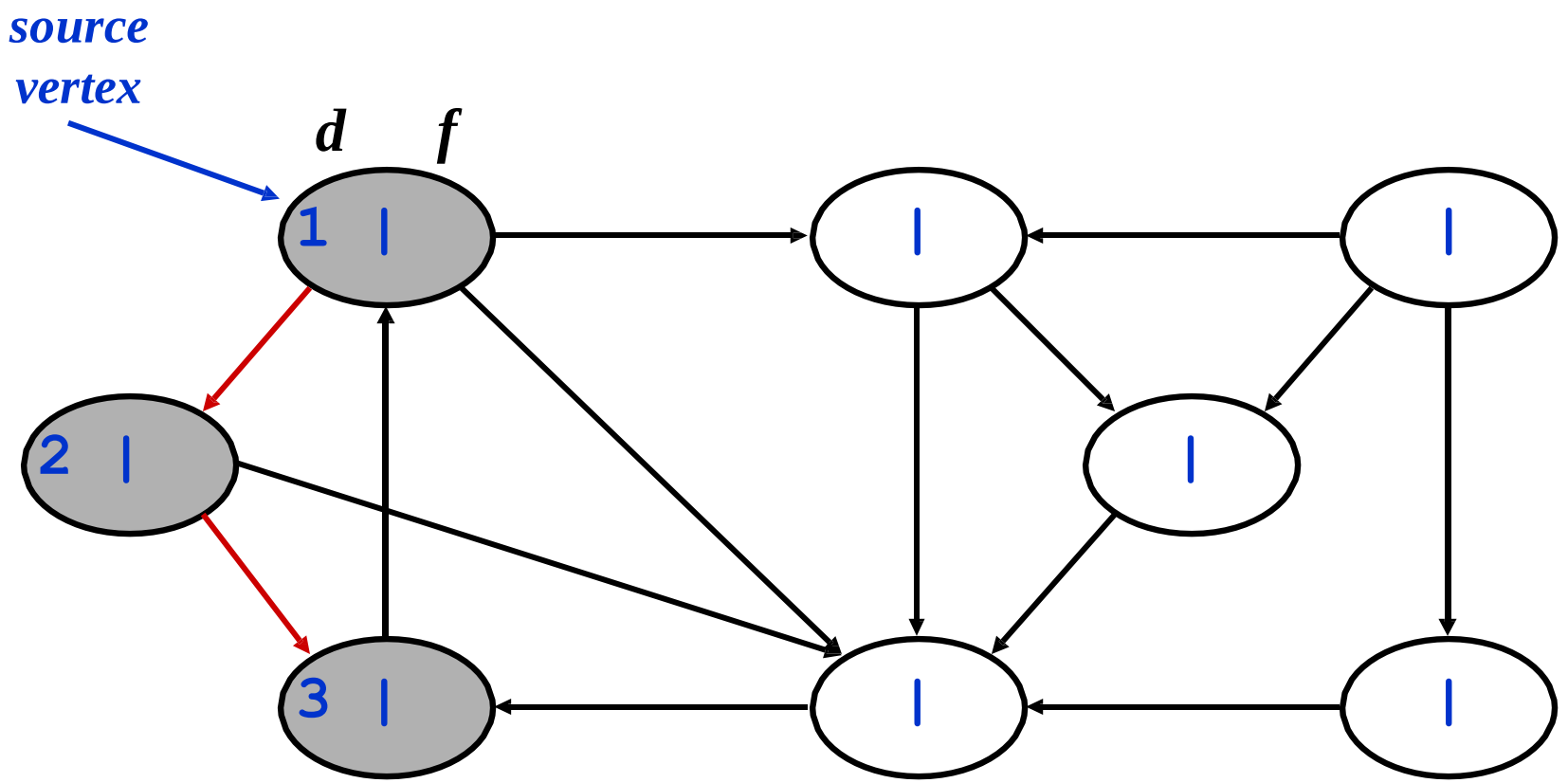
DFS Example



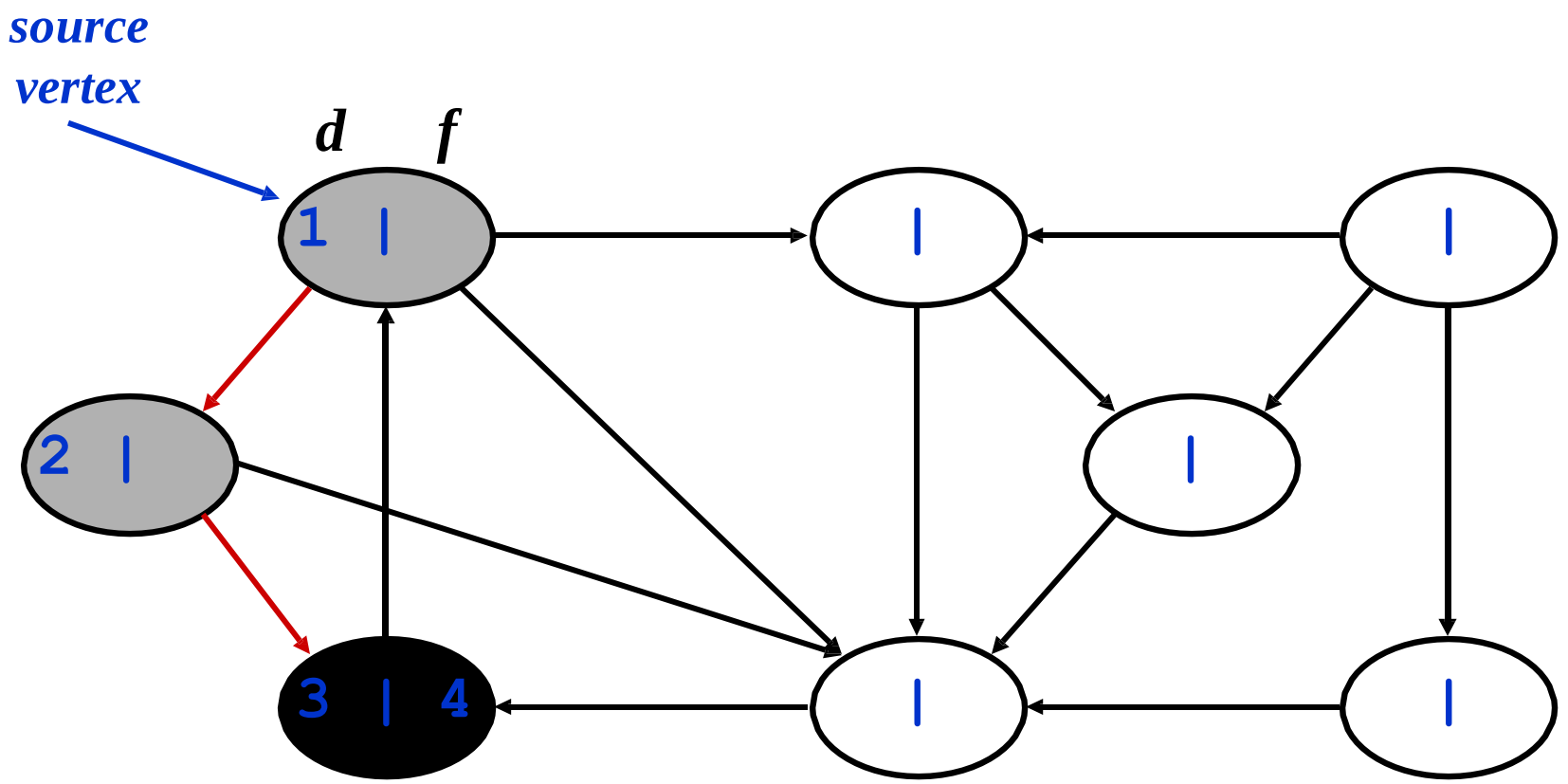
DFS Example



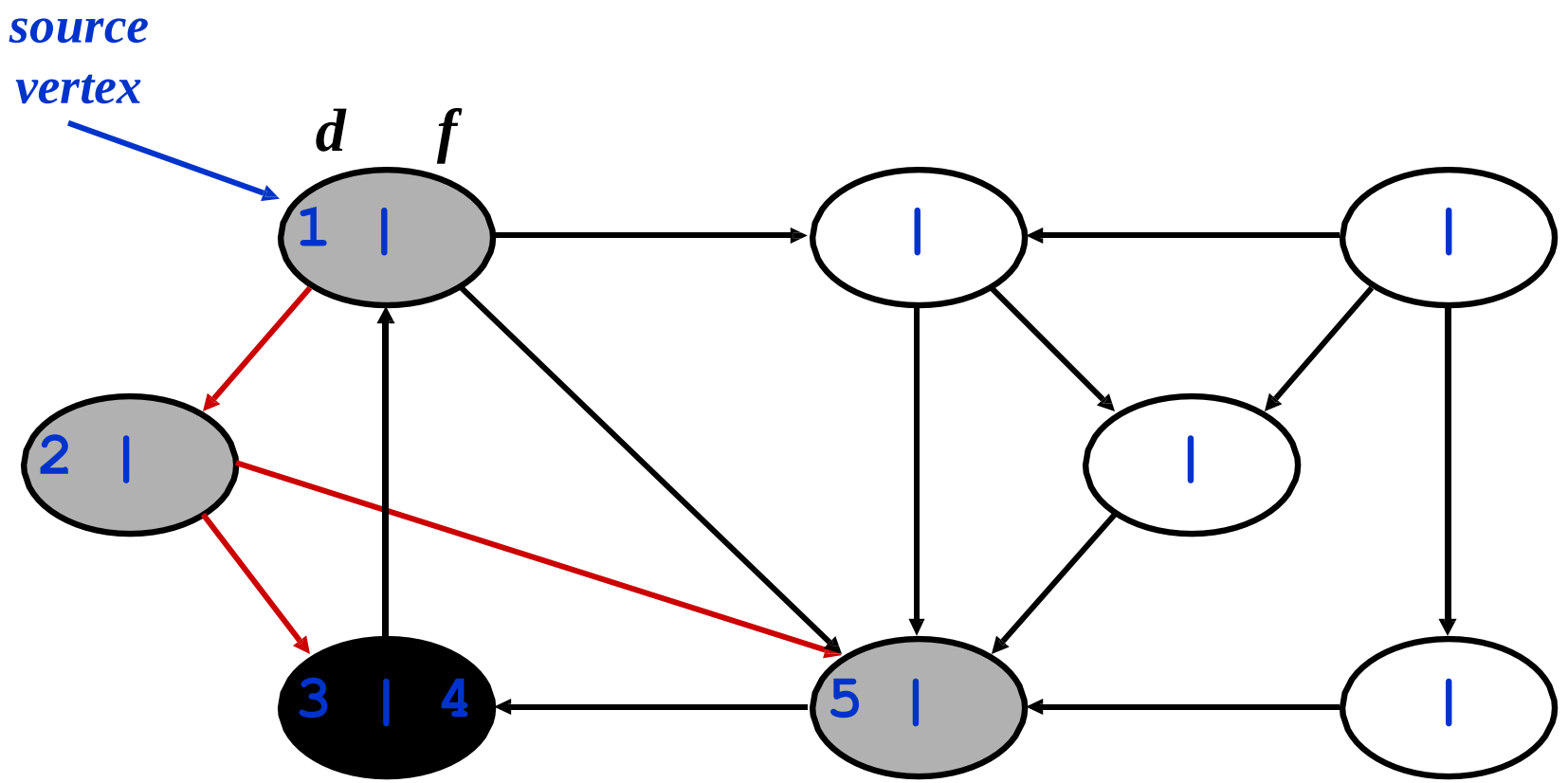
DFS Example



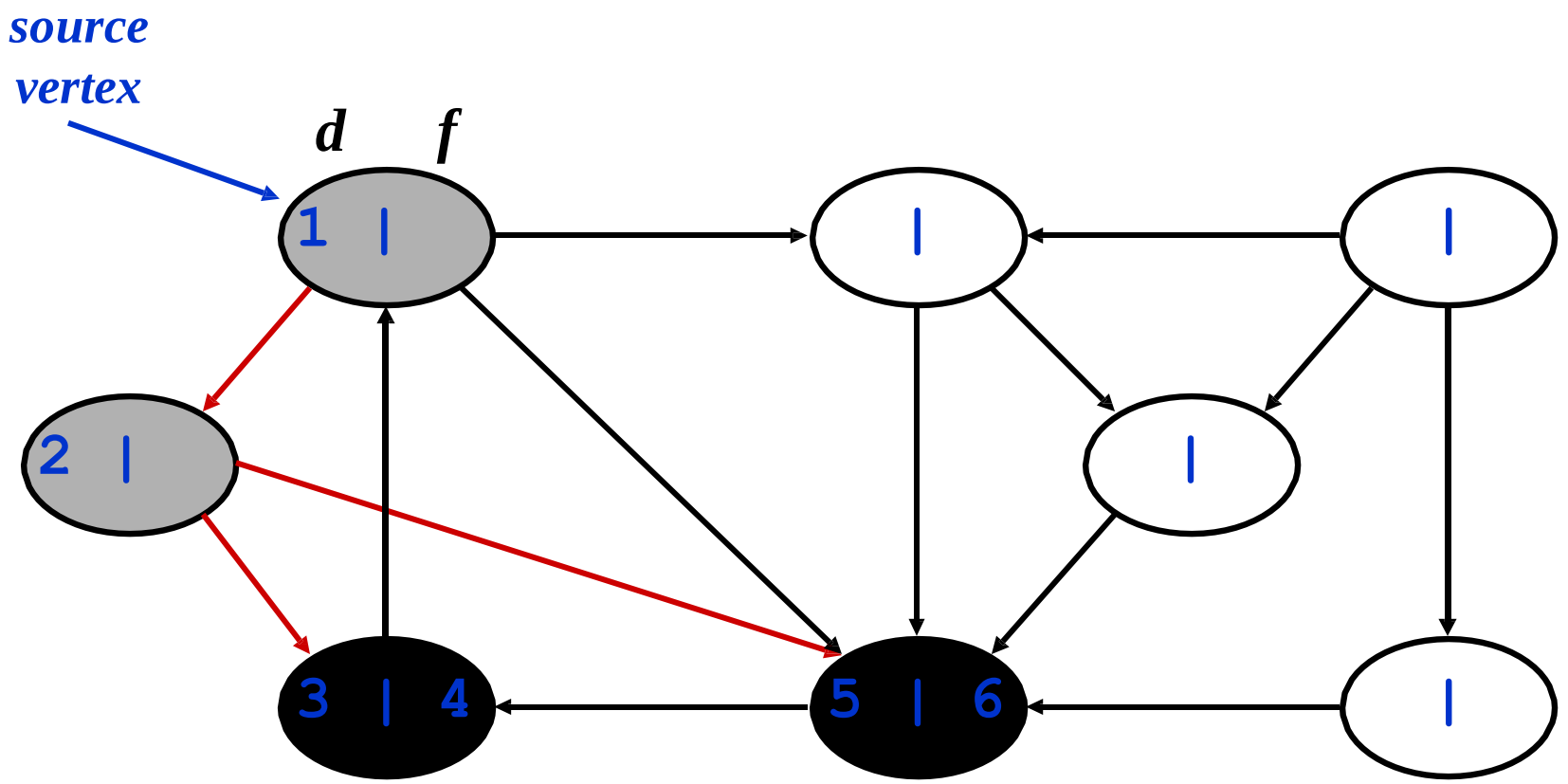
DFS Example



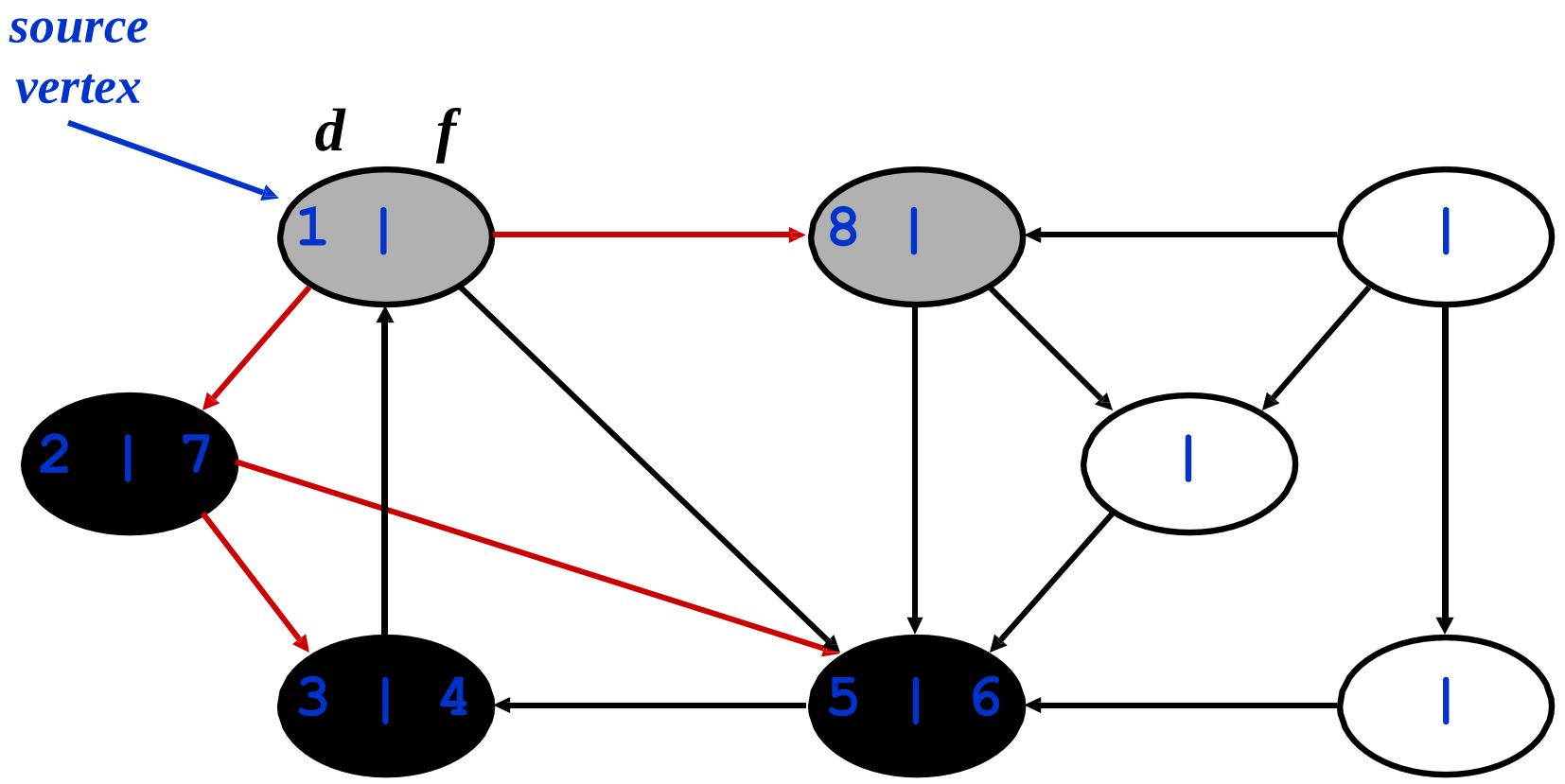
DFS Example



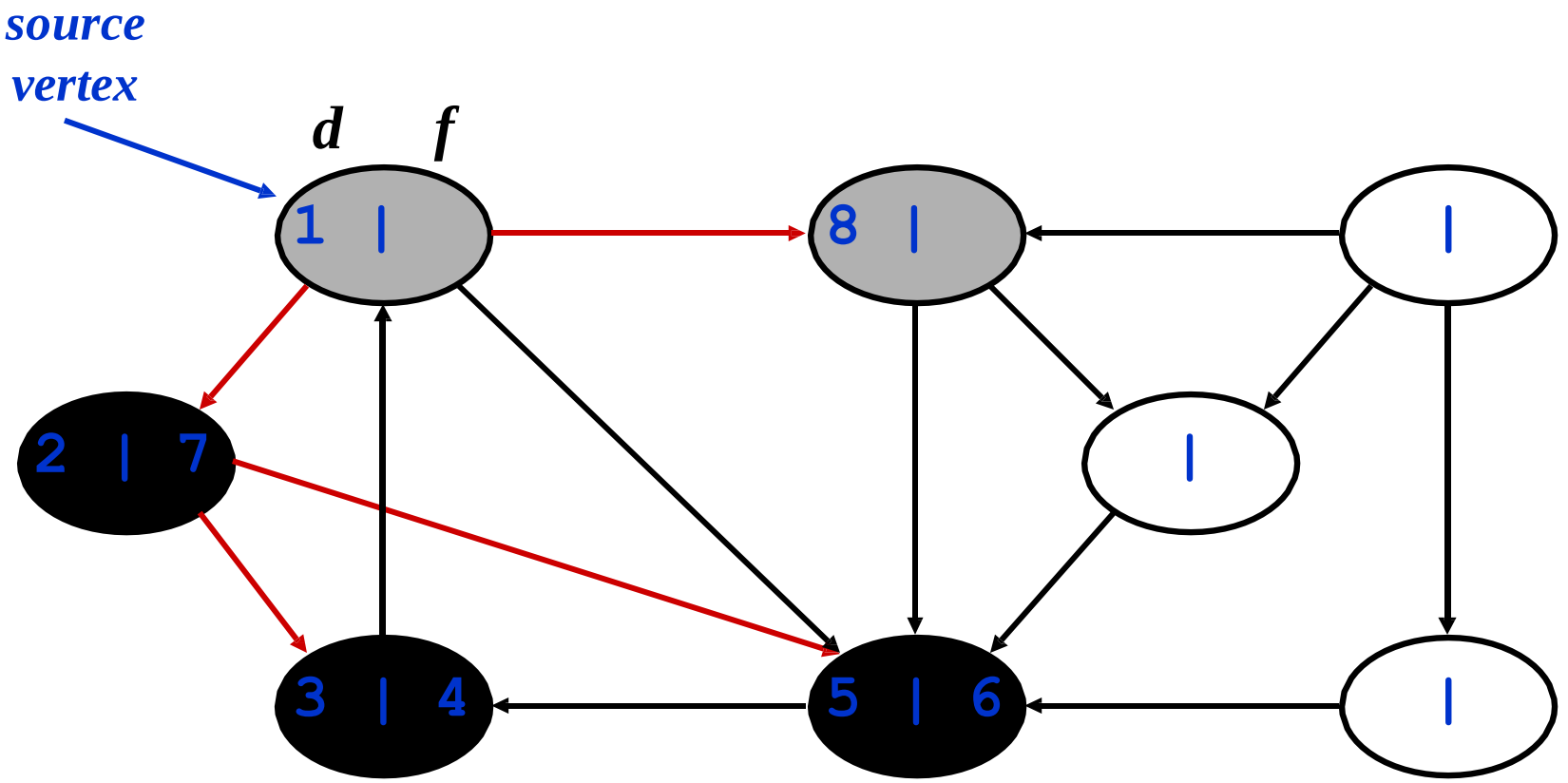
DFS Example



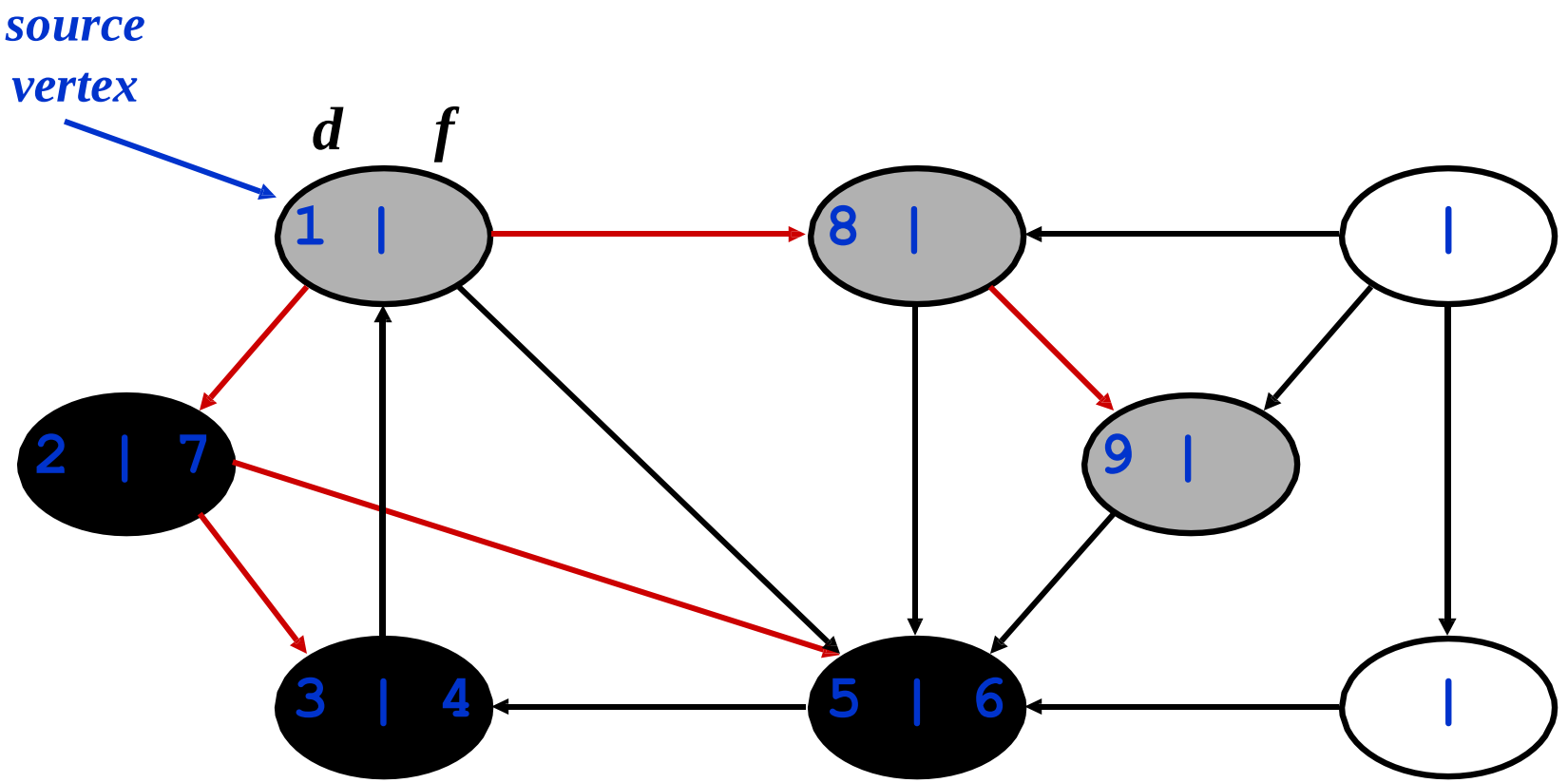
DFS Example



DFS Example

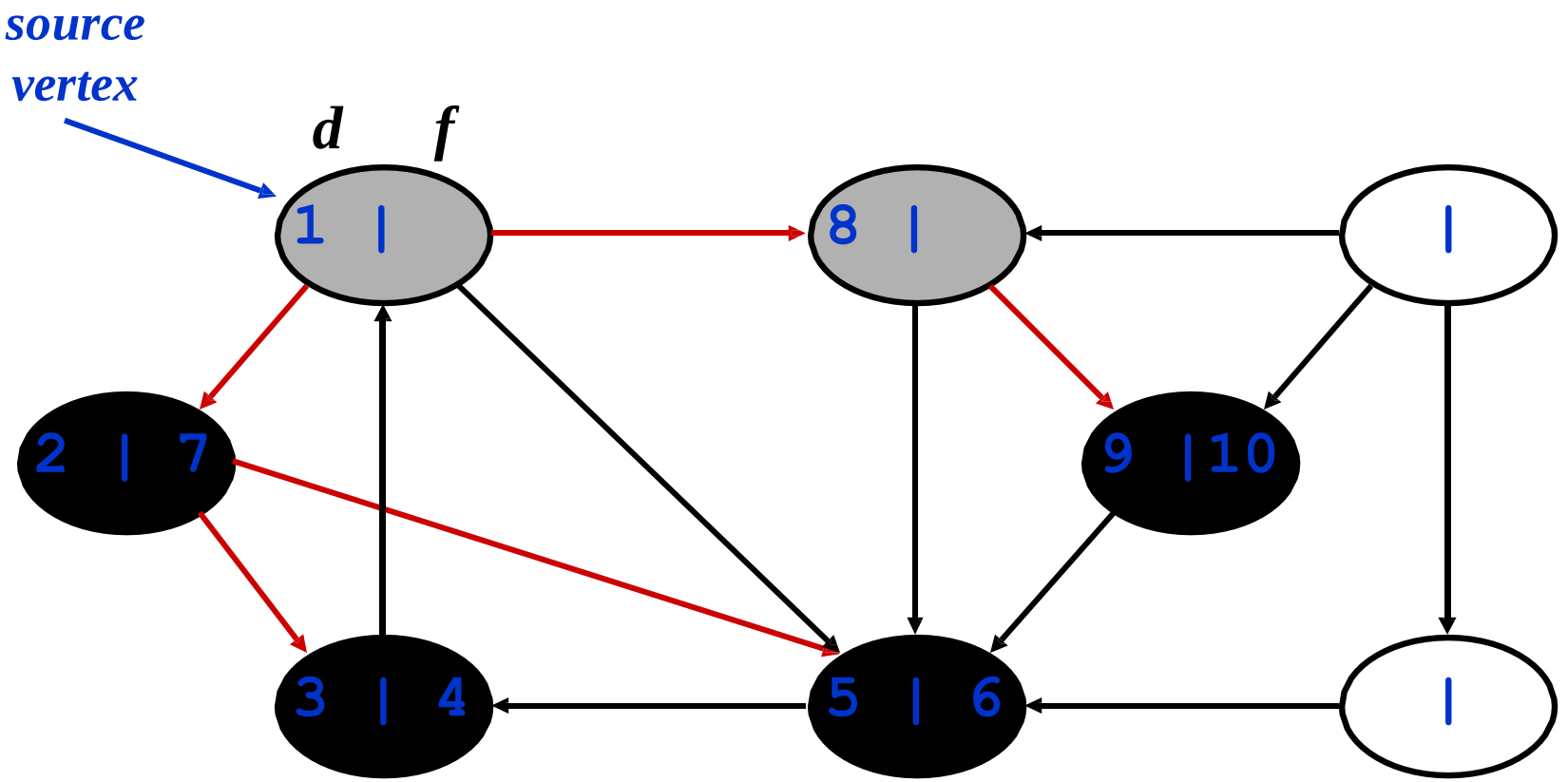


DFS Example

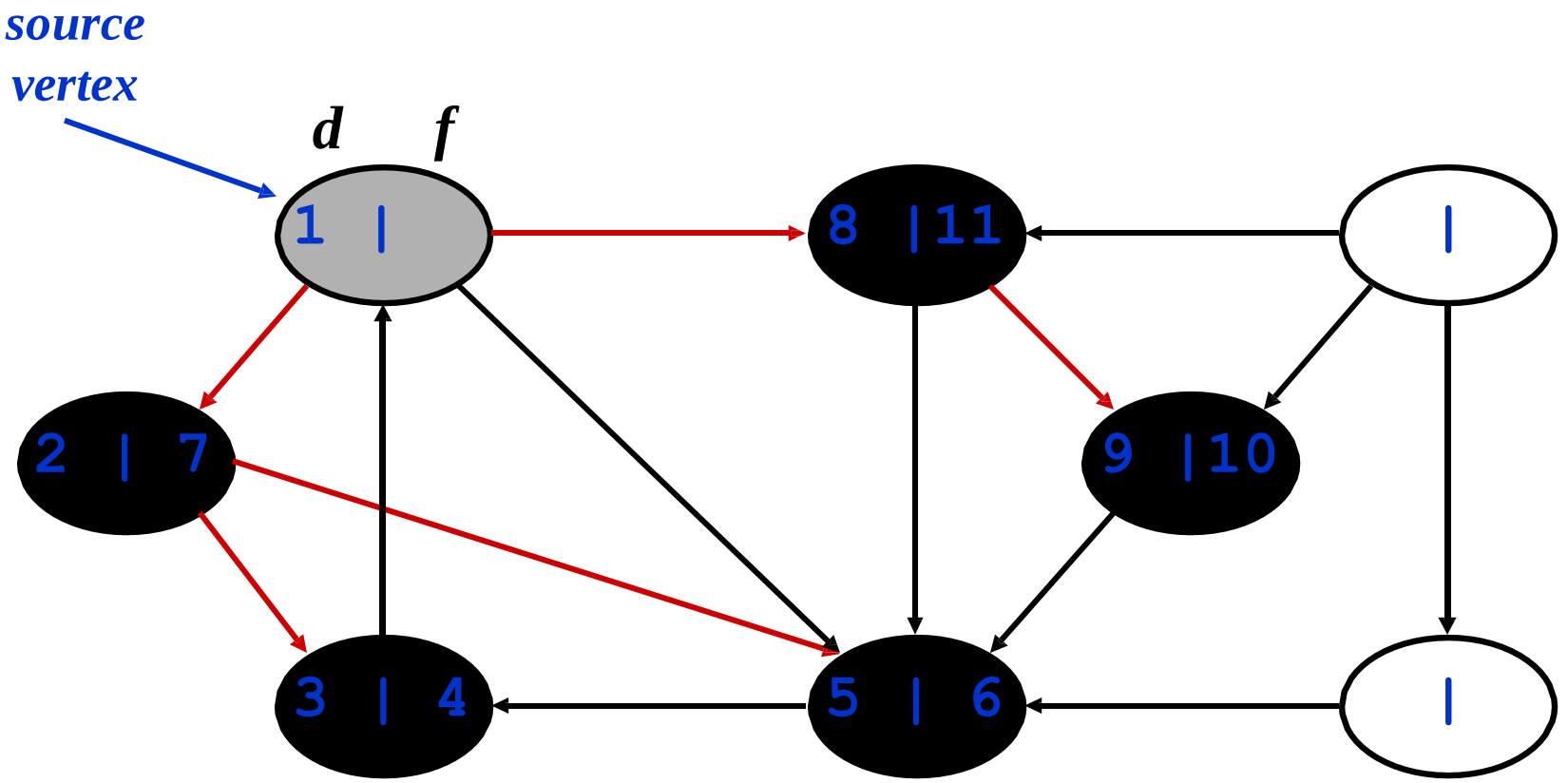


What is the structure of the grey vertices?
What do they represent?

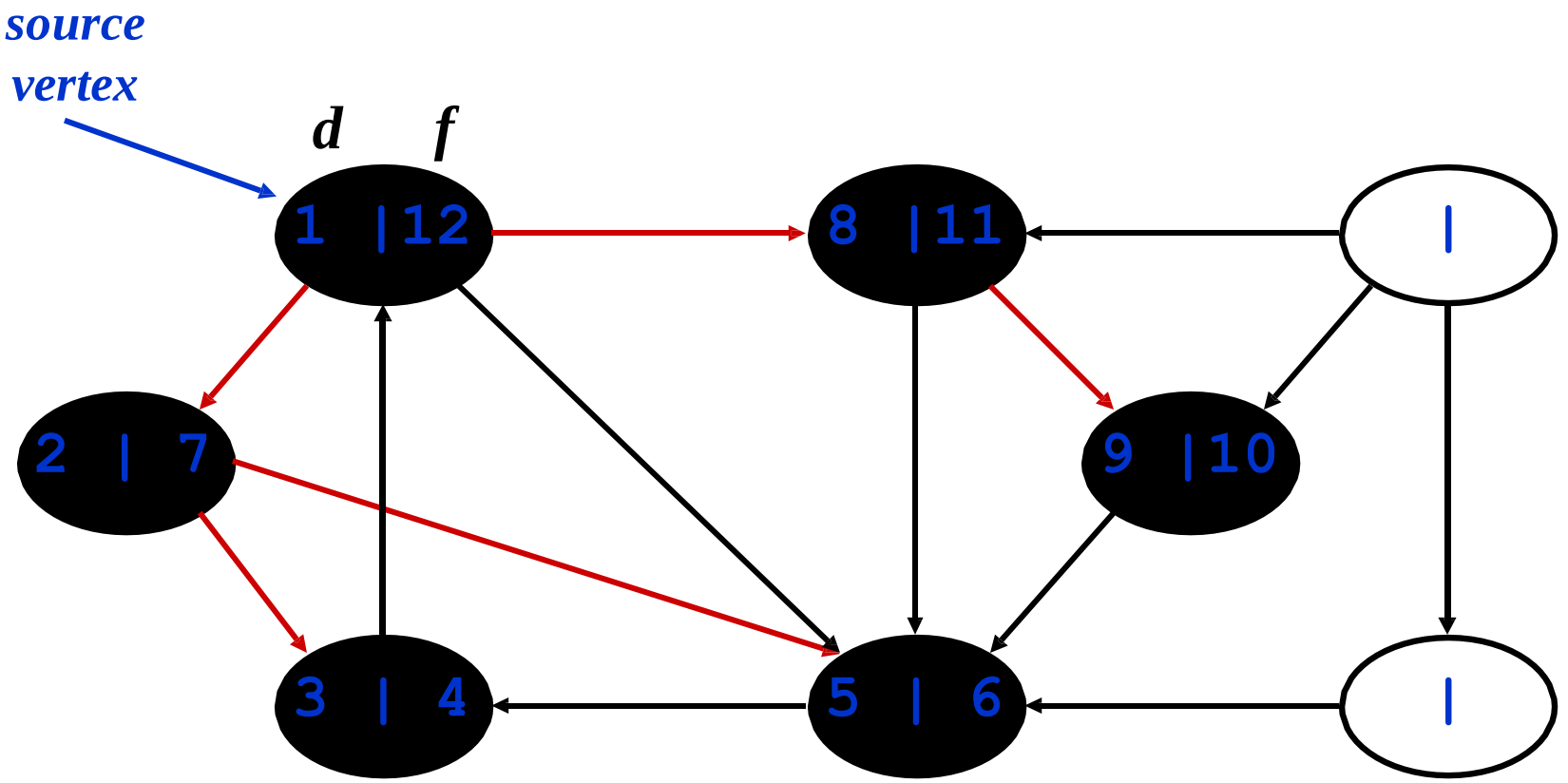
DFS Example



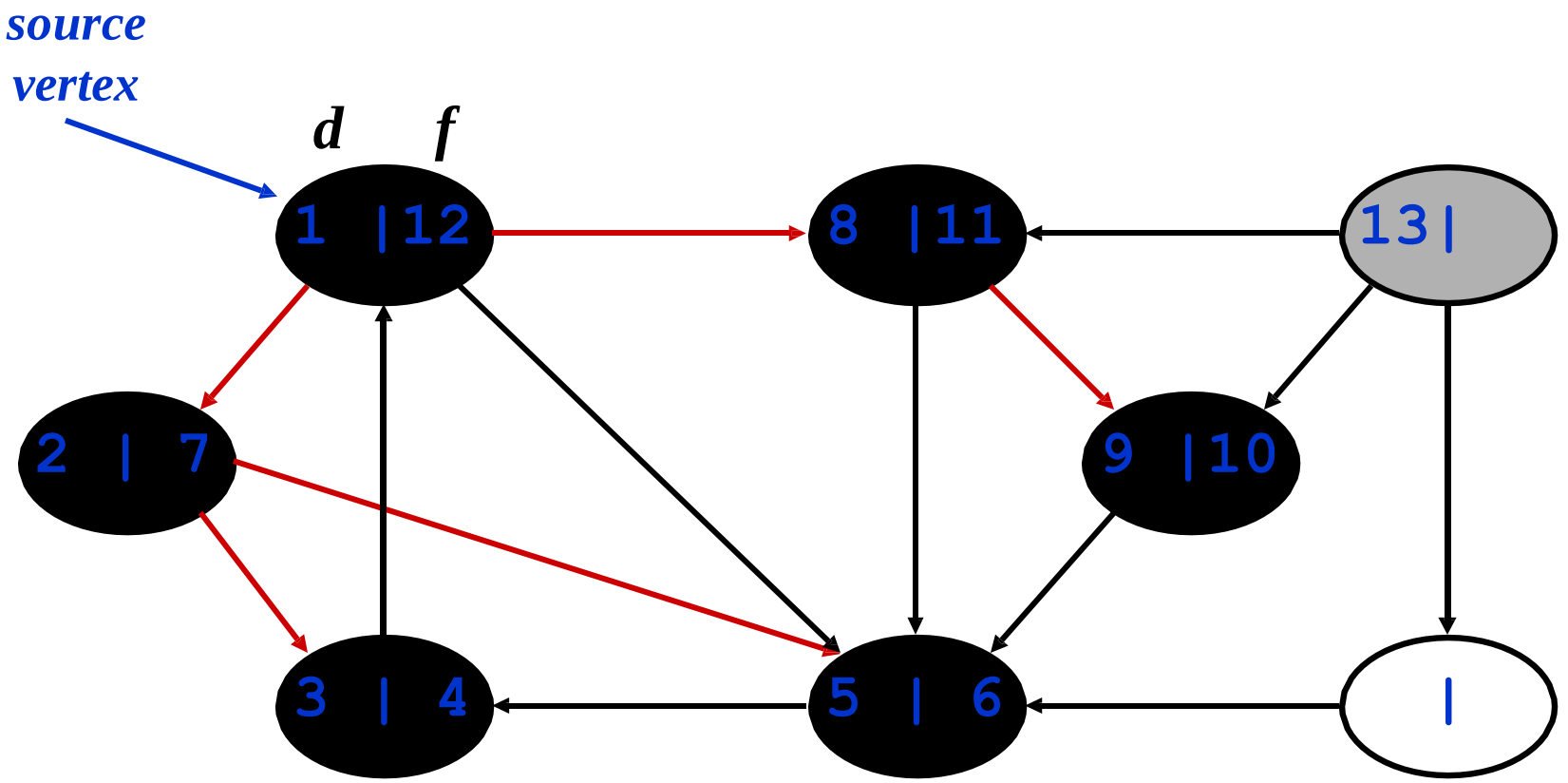
DFS Example



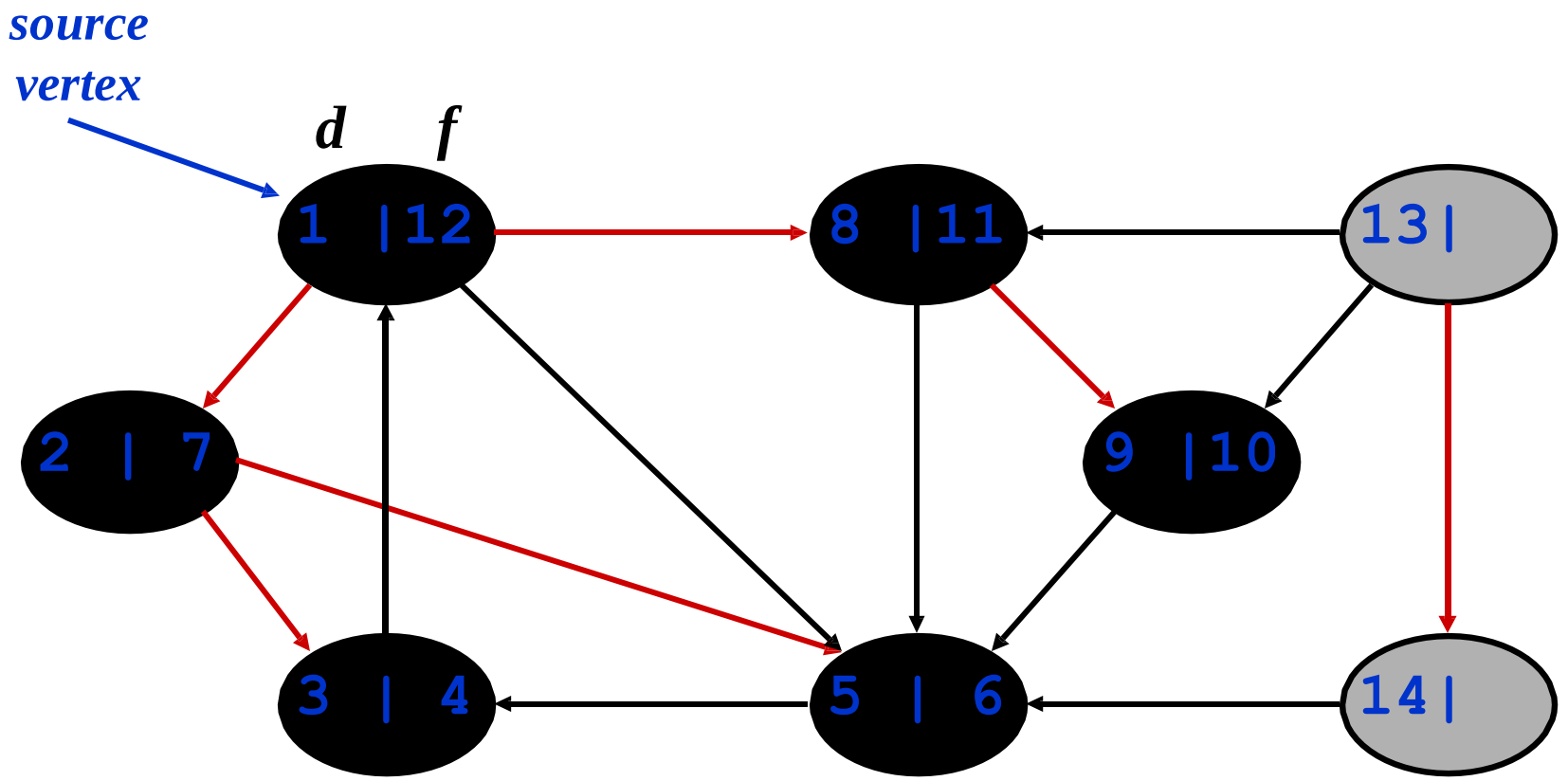
DFS Example



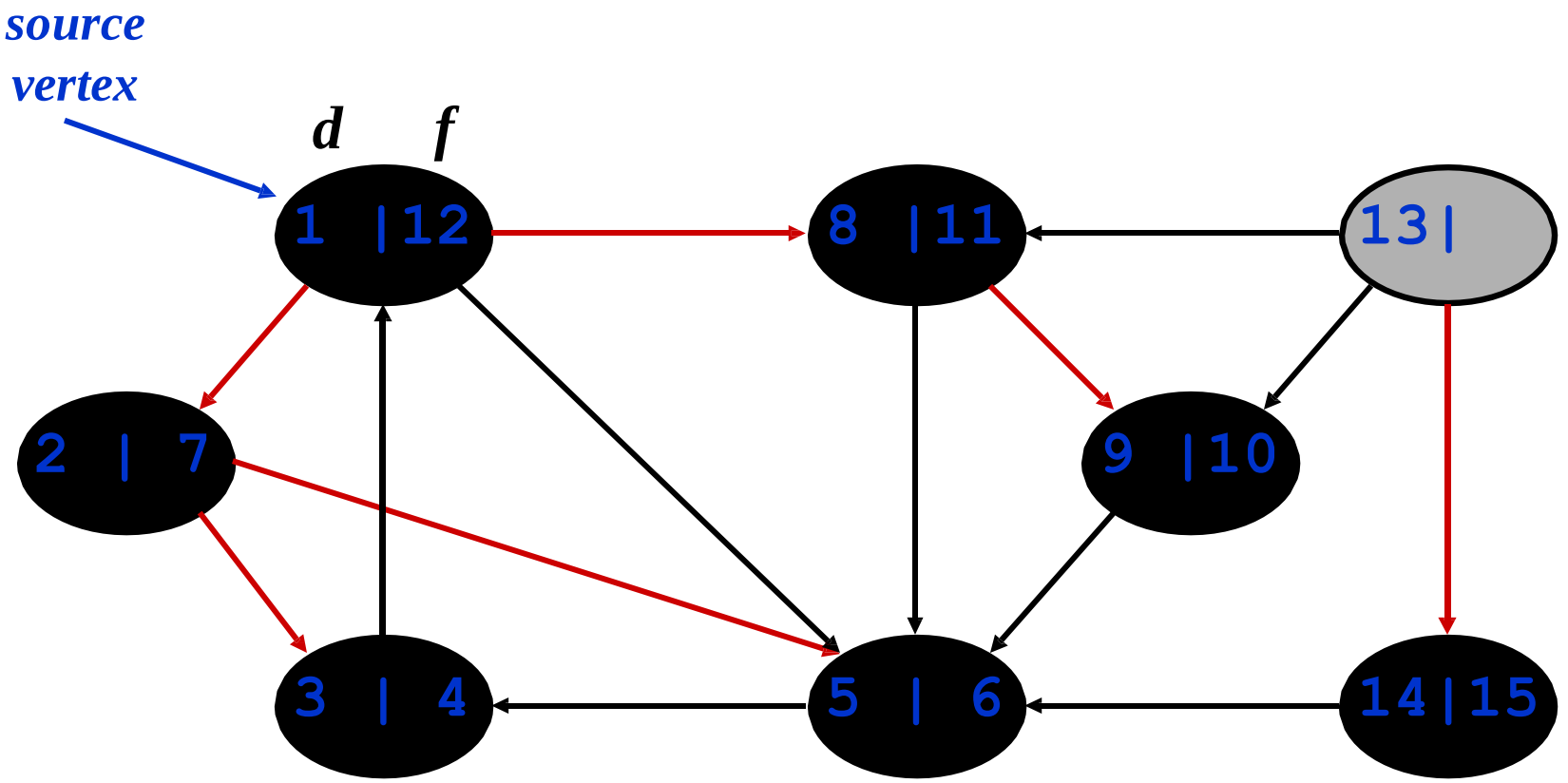
DFS Example



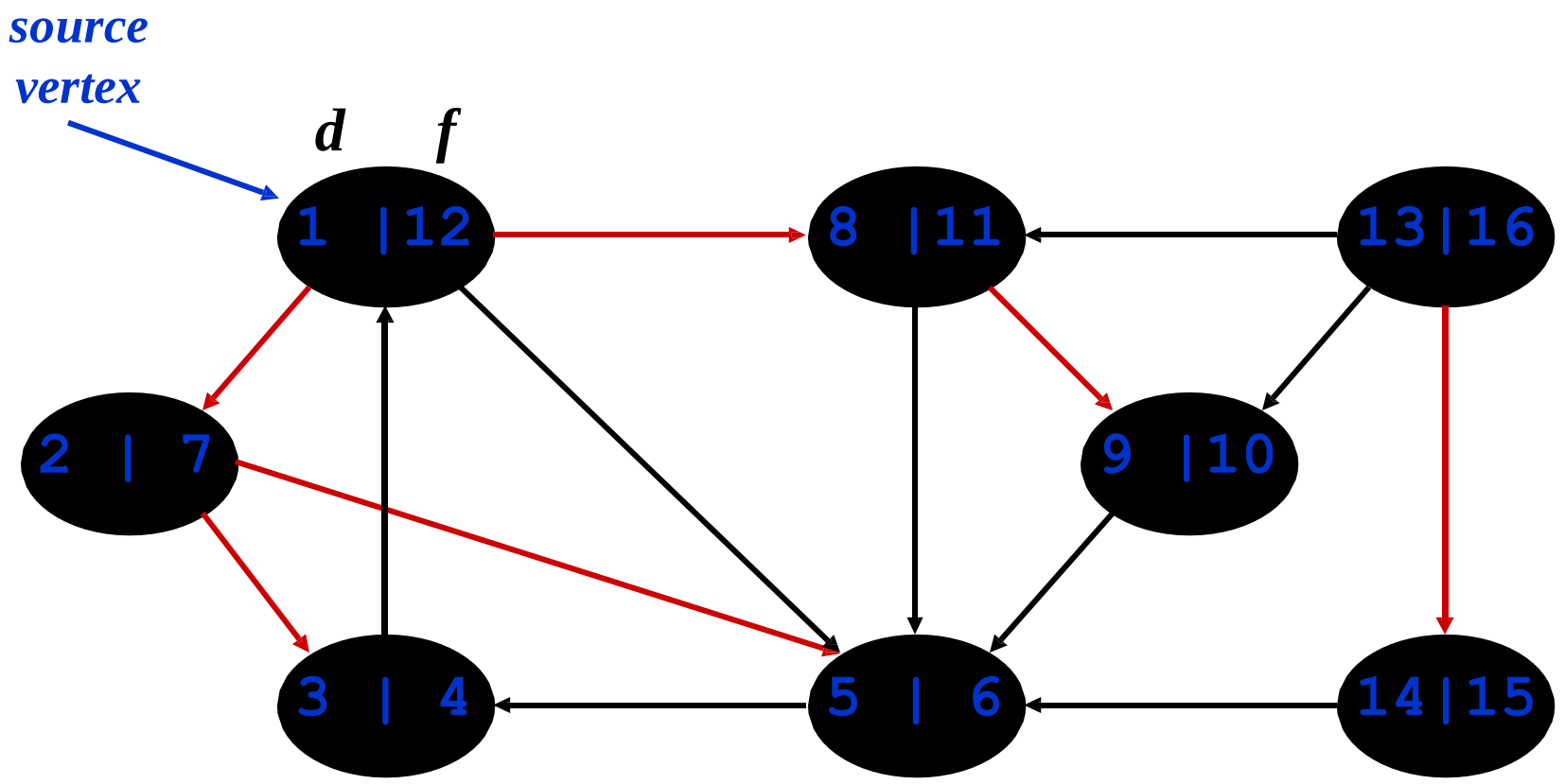
DFS Example



DFS Example



DFS Example



Backtracking

Backtracking

- Many problems translate to searching for a specific node, path or pattern in the associated graph.
- If the graph contains a large number of nodes, and particularly if it is infinite, it may be wasteful or infeasible to build it explicitly in computer storage before applying one of the search techniques we have discussed so far.
- In such a situation we use an implicit graph for which description of its nodes and edges is available and relevant parts of it can be built as the search progress.
- In this way, computing time is saved whenever the search succeeds before the entire graph has been constructed.
- The economy in storage space can also be dramatic, especially when nodes that have already been searched can be discarded, making room for subsequent nodes to be explored.

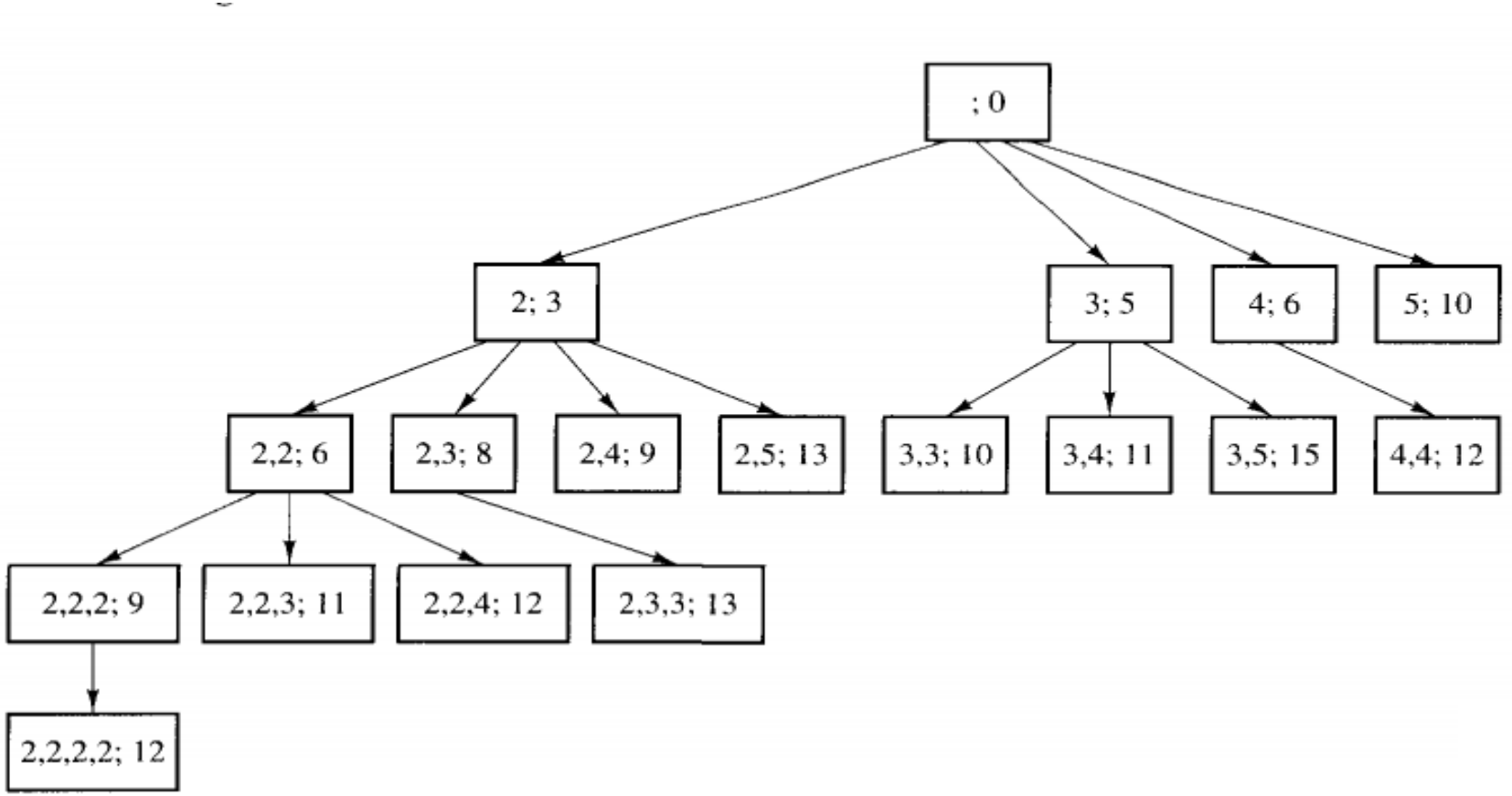
Backtracking

- If the graph involved is infinite, such a technique offers us the only way of exploring it.
- In its basic form, backtracking resembles a depth-first search in a directed graph.
- The graph concerned is usually a tree, or at least it contains no cycles.
- Whatever its structure, the graph exists only implicitly.

Backtracking

- **0/1 Knapsack Problem:**
  - We have certain number of objects and a knapsack.
  - Specifically we have n types of object (not n objects), and that an adequate number of objects of each type are available.
  - Instance:
    - Four types of objects, whose weights are respectively 2, 3, 4 and 5 units, and whose values are 3, 5, 6, and 10. The knapsack can carry a maximum of 8 units of weight.

0/1 Knapsack Problem:



0/1 Knapsack Problem:

- Here, a node such as (2,3; 8) corresponds to a partial solution of the problem.
- The figures to the left of the semicolon are the weights of the objects we have decided to include, and the figure to the right is the current value of the knapsack.
- Moving down from a node to one of its children corresponds to deciding which kind of object to put into the knapsack next.
- We may agree to load objects into the knapsack in order of increasing weight.
- This is not essential, and indeed any other order – by decreasing weight, for example, or by value-would work just as well.

0/1 Knapsack Problem:

- Initially the partial solution is empty.
- The backtracking algorithm explores the tree in a DFS manner, constructing nodes and partial solution as it goes.
- In the example, the first node visited is (2; 3), the next is (2, 2; 6), the third is (2, 2, 2; 9) and the fourth (2, 2, 2, 2; 12).
- As each new node is visited, the partial solution is extended.
- After visiting these four nodes, the dfs is blocked: node (2, 2, 2, 2: 12) has no unvisited successors (indeed no successors at all), since adding more items to this partial solution would violate the capacity constraint.
- Since this partial solution may turn out to be the optimal solution to our instance, we memorize it.

0/1 Knapsack Problem:

- The DFS now backs up to look for other solution. At each step back up the tree, the corresponding item is removed from the partial solution.

- In the example, the search first backs up to (2, 2, 2; 9), which also has no unvisited successor; one step further up the tree, however, at node (2, 2; 6), two successors remain to be visited.

- After exploring nodes (2, 2, 3; 11) and (2, 2, 4; 12), neither of which improves on the solution previously memorized, the search backs up one stage further, and so on.

0/1 Knapsack Problem:

- Exploring the tree in this way, (2, 3, 3; 13) is found to be a better solution than the one we have , and later (3, 5; 15) is found to be better still. Since no other improvement is made before the search ends, this is the optimal solution to the instance.

0/1 Knapsack Problem:

Programming the algorithm is straightforward, and illustrates the close relation between recursion and depth-first search. Suppose the values of n and W, and of the arrays $w[1..n]$ and $v[1..n]$ for the instance to be solved are available as global variables. The ordering of the types of item is unimportant. Define a function *backpack* as follows.

```
function backpack(i, r)
    {Calculates the value of the best load that can
     be constructed using items of types i to n
     and whose total weight does not exceed r}
    b ← 0
    {Try each allowed kind of item in turn}
    for k ← i to n do
        if w[k] ≤ r then
            b ← max(b, v[k] + backpack(k, r − w[k]))
    return b
```

Branch and Bound

Introduction

What is Branch and Bound?

- **Branch and bound** (**BB** or **B&B**) is a general algorithm for finding optimal solutions of various optimization problems.

- Like Backtracking, Branch and Bound is a technique for Exploring an implicit directed graph. Again this graph is Usually acyclic or even a tree.

- Branch and bound is a systematic method for solving optimization problems

- B&B is a rather general optimization technique that applies where the greedy method and dynamic programming fail.

- Branch and bound uses auxiliary computation to decide at each instant which node should explored next, and a priority list to hold those nodes that have been generated but not yet explored.

How Branch and Bound Work

- Starting by considering the root node and applying a lower-bounding and upper-bounding procedure to it
- If the bounds match, then an optimal solution has been found and the algorithm is finished
- If they do not match, then algorithm runs on the child nodes.

Application of Branch and Bound

- This approach is used for a number of NP-hard problems, such as
- Assignment Problem
- Traveling salesman problem (TSP)
- Knapsack problem
- Integer programming
- Nonlinear programming
- Maximum satisfiability problem (MAX-SAT)
- Nearest neighbor search (NNS)
- Cutting stock problem
- False noise analysis (FNA)

Assignment Problem

- In the Assignment Problem, n agent are to be assigned n task, each agent having exactly one task to perform. If agent i, (1<=i<=n), is assigned task j,(1<=j<=n), then the cost of performing this particular task will be Cij.

- The problem is to assign agents to tasks so as to minimize the total cost of executing n tasks.

For Example…..

	1	2	3
a	**4**	7	**3**
b	2	**6**	1
c	**3**	9	**4**

•Suppose Three agent a , b, c are to be assigned task 1, 2, 3 and Cost matrix is as above

•If we allocate task 1 to agent a, task 2 to agent b, task 3 to agent c, then our total cost will be 4+6+4=14, while if we allot task 3 to agent a, task 2 to agent b and task 1 to agent c then cost is only 3+6+3= 12. In this particular example, the optimal assignment is a→2, b→3 ,c→1, whose cost is 7+1+3= 11.

For Example…..

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

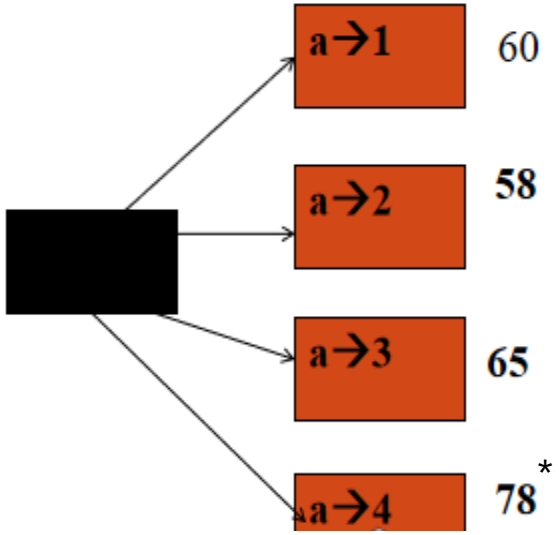
•Suppose we take Upper Bound then obtain then answer a→1 ,b→2, c→3, d→4 is one possible solution whose cost is 11+15+19+28=73. the optimal Solution to the problem cannot more than this. Another Possible Solution is a→4 , b→3, c→2, d→1 whose cost is by adding the elements in in the other diagonal of cost matrix, is given 40+13+17+17=87.

In this Case the second solution is no improvement over the first. To obtain a lower bound on the solution, we can argue that whoever execute task 1, the cost will be at least 11, whoever execute task 2 the cost will be at least 12, and so on. Thus adding Smallest element in each column gives us lower bound on the answer. In this example, this is 11+12+13+22=58. So From Above observation we know that our observation lies between [58..73]

	1	2	3	4
a	**11**	**12**	18	40
b	14	15	**13**	**22**
c	11	17	19	23
d	17	14	20	28

- To Solve the Problem By Branch and Bound, We explore a tree whose nodes correspond to partial assignment. At the root of the tree, no assignment have been made. Subsequently, at each level we fix the assignment of one or more agent.

- Suppose For Example that, we Starting from the root, we decide the assignment of agent a. So there are Four Way to doing this, there are Four branches from the root.



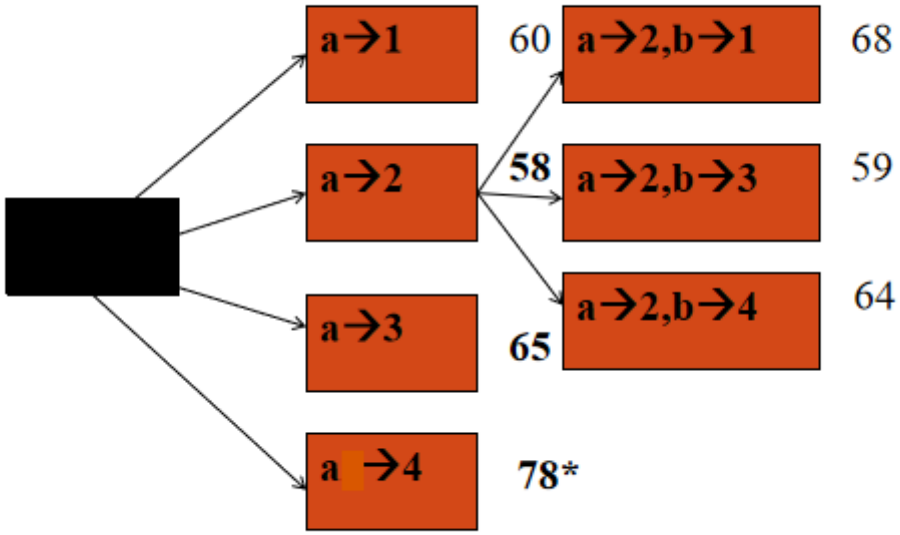
	1	2	3	4
a	**11**	12	18	40
b	14	15	**13**	**22**
c	11	17	19	23
d	17	**14**	20	28

- For Example, with this partial assignment task a will cost 11, task 2 will executed by b, c, d; so the lowest possible cost is 14 . Thus a lower bound on any Solution obtained by computing the partial assignment a→1. is **11+14+13+22= 60**. Similarly for the node a→2, task 2 will be executed by agent a at cost of 12, while task 1,3,4 will be executed by agent b, c and d at minimum cost of **11+12+13+22=58**. Similarly for the node a→3, minimum cost of **11+14+18+22=65**. for the node a→4, the minimum cost of **11+14+13+40=78**.
- But we know the optimal solution cannot be exceed 73, so there is no point in exploring the node a→4 any further, so it cannot be optimal, it is "dead". The other three node are alive. Node a→2 has the smallest lower bound. Therefore it look more promising then other , this is the one to explore next.

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

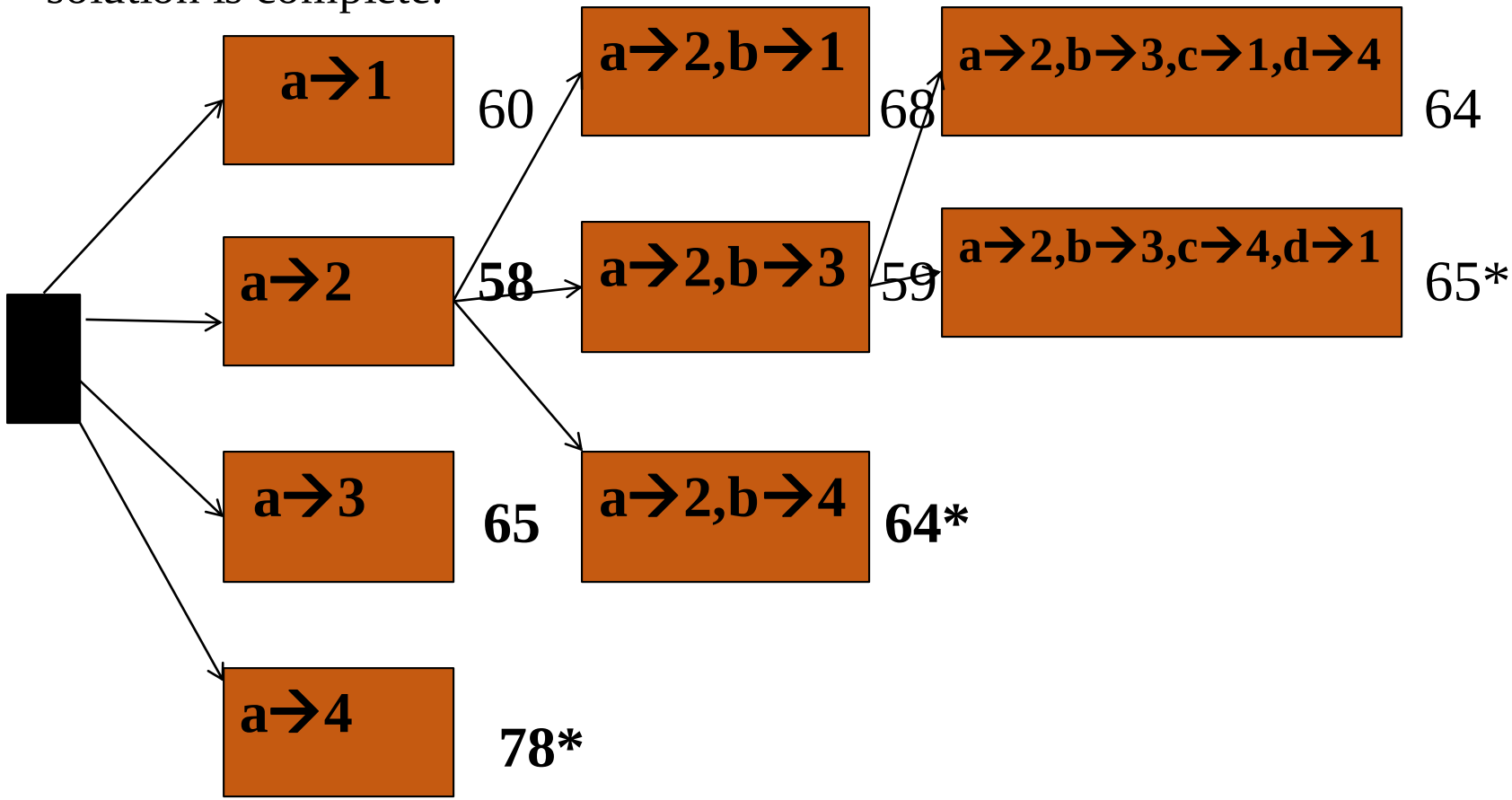
[Assign Agent b]

Now we Explore node at a→2, b→1, task 1 will cost 14 and task 2 will cost 12. The remaining execute by c or d. so the simplest possible cost for task 3 is 19, and for task 4 is 23. Hence lower bound for possible solution is **14+12+19+23= 68**. Similarly for a→2, b→3 the possible solution is **12+13+11+23=59**. Same for a→2, b→4 the possible solution is **12+22+11+19=64**.



[After Assigning agent b]

- The More promising node in the tree is now a→2,b→3 with lower bound of 59. Now we continue to exploring the node starting from this node, we fix another partial assignment c, when we assign the assignment of a , b and c fixed, however, we no longer have any choice about how we assign d, so the solution is complete.

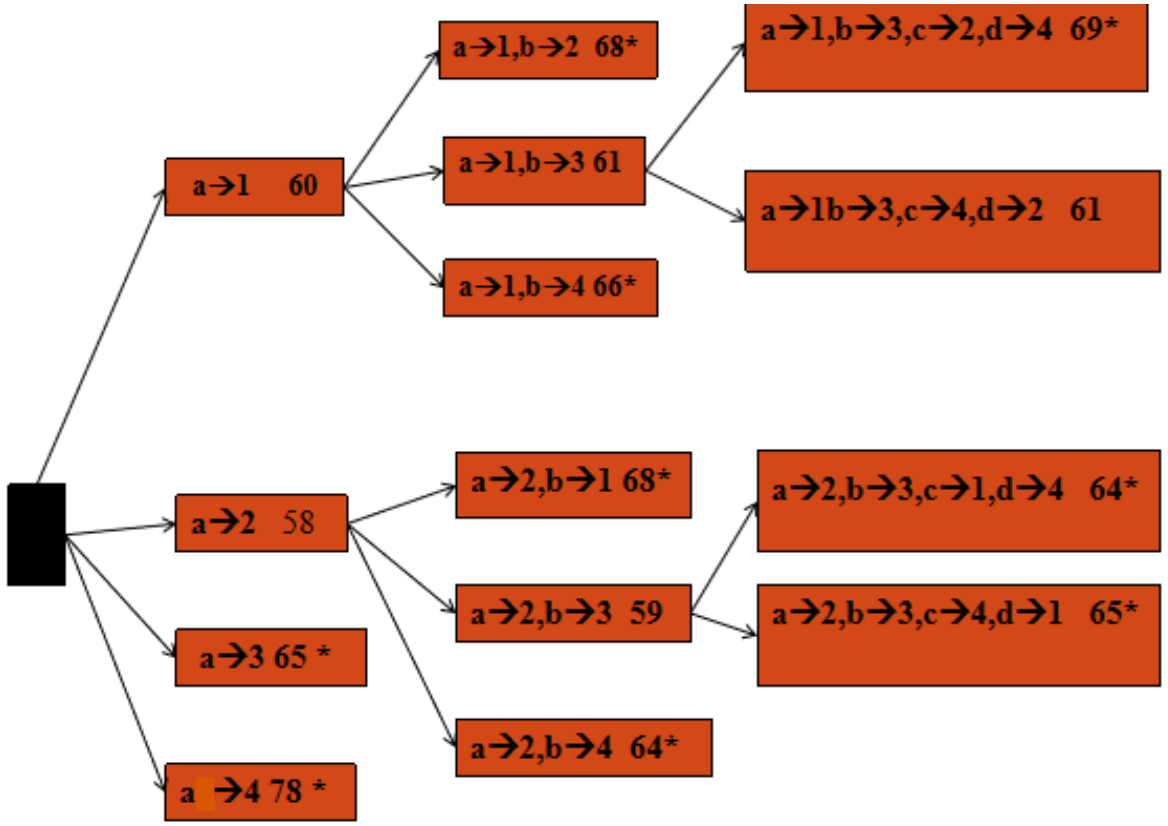


[After Assigning agent c]

- Now we get new upper bound, we can remove nodes a→3 and a→2, b→1 from further consideration, as indicated by asterisks. If we only want one solution, we can eliminate node a→2,b→4 as well.

- The Node still remaining is a→1. Proceeding after two step we obtain the final situation.
- Now expanding a→1, make a→1,b→2 fix we get 11+15+19+23=68
- For a→1,b→3 fix we get 11+17+13+23=61
- For a→1,b→4 fix we get 11+22+14+19=66
- Now a→1,b→3 is more promising so we expand it
- For a→1,b→3,c→2,d→4 fix we get 11+13+17+28=69
- For a→1,b→3,c→4,d→2 fix we get 11+13+23+14=61
- At the remaining unexplored nodes, the lower bound is greater than 61, so there is no point in exploring them further.
- The solution is therefore the optimal solution.

The Tree Completely Explored




Thank You!