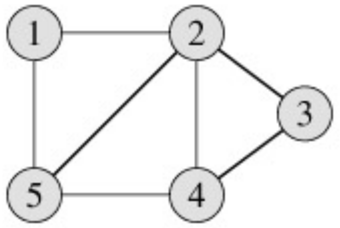
Graph Search:

Breadth-First Search (BFS)
Depth-First Search (DFS)

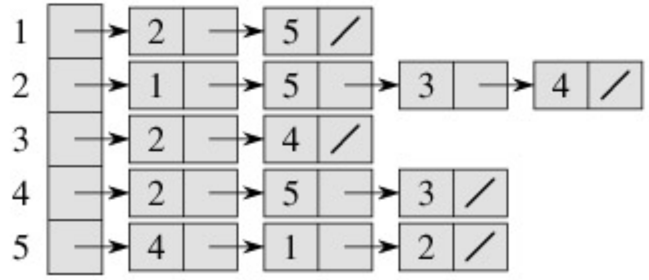
Breadth-First and Depth-First Search

- BFS Basic Algorithm
- BFS Complexity
- DFS Algorithm
- DFS Implementation
- Relation between BFS and DFS

Graph representation



(a)



(b)

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

(c)

graph

Adjacency list

Adjacency matrix

Graph Traversal

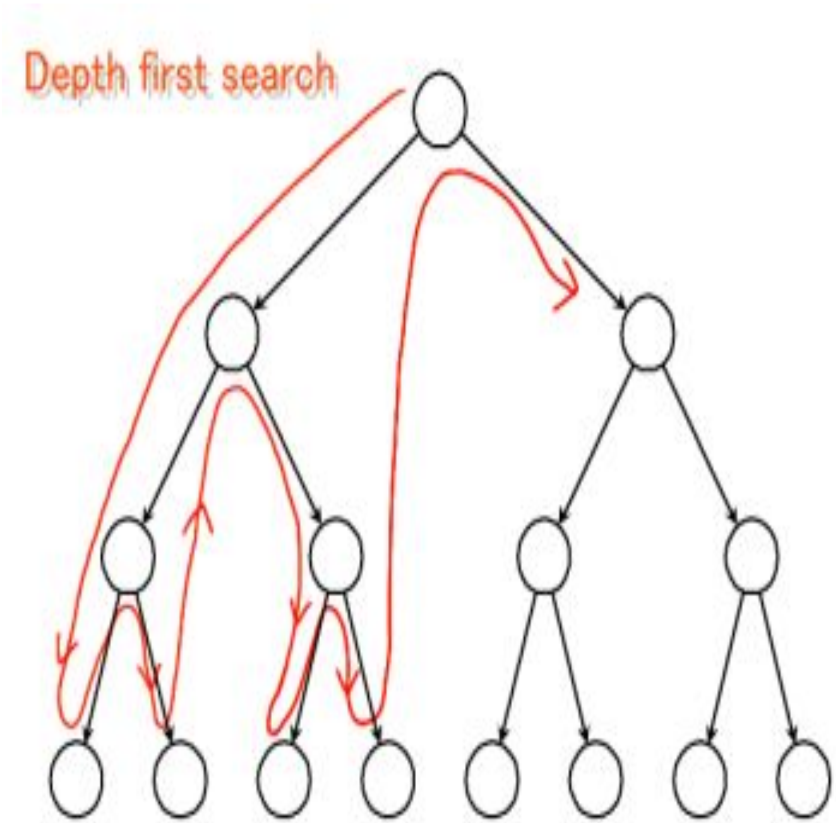
- **Problem:** Search for a certain node or traverse all nodes in the graph
- **Depth First Search**
  - Once a possible path is found, continue the search until the end of the path
- **Breadth First Search**
  - Start several paths at a time, and advance in each one step at a time

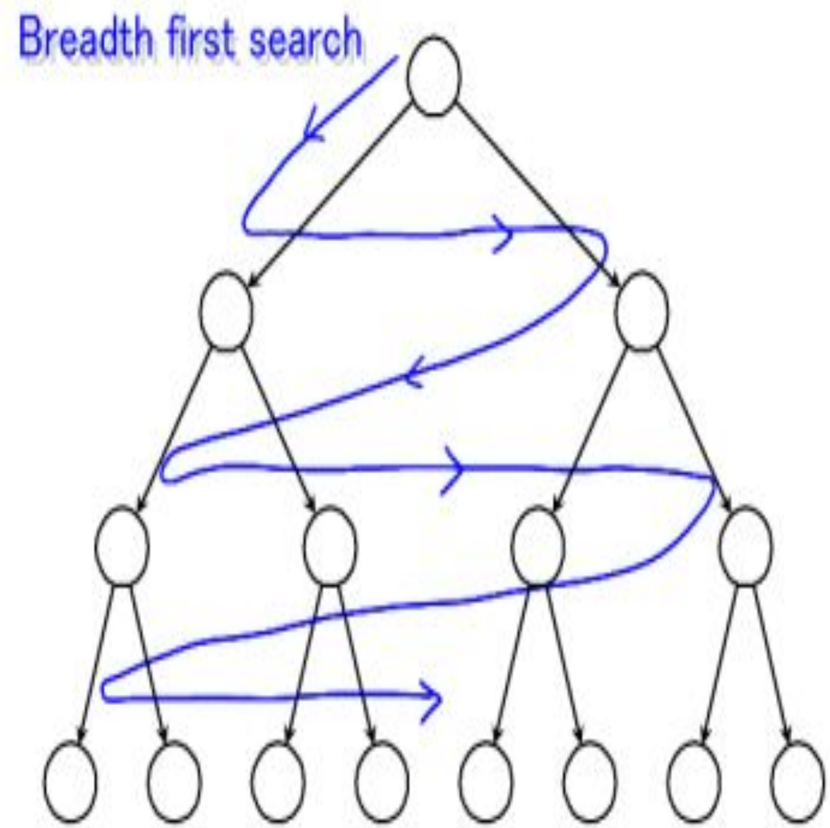
Graph Traversal (Contd.)

- In both DFS and BFS, the nodes of the undirected graph are visited in a systematic manner so that every node is visited exactly once.

- Both BFS and DFS give rise to a tree:
  - When a node x is visited, it is labeled as visited, and it is added to the tree.
  - If the traversal got to node x from node y, y is viewed as the parent of x, and x is the child of y.

Graph Traversals





Breadth-First Search

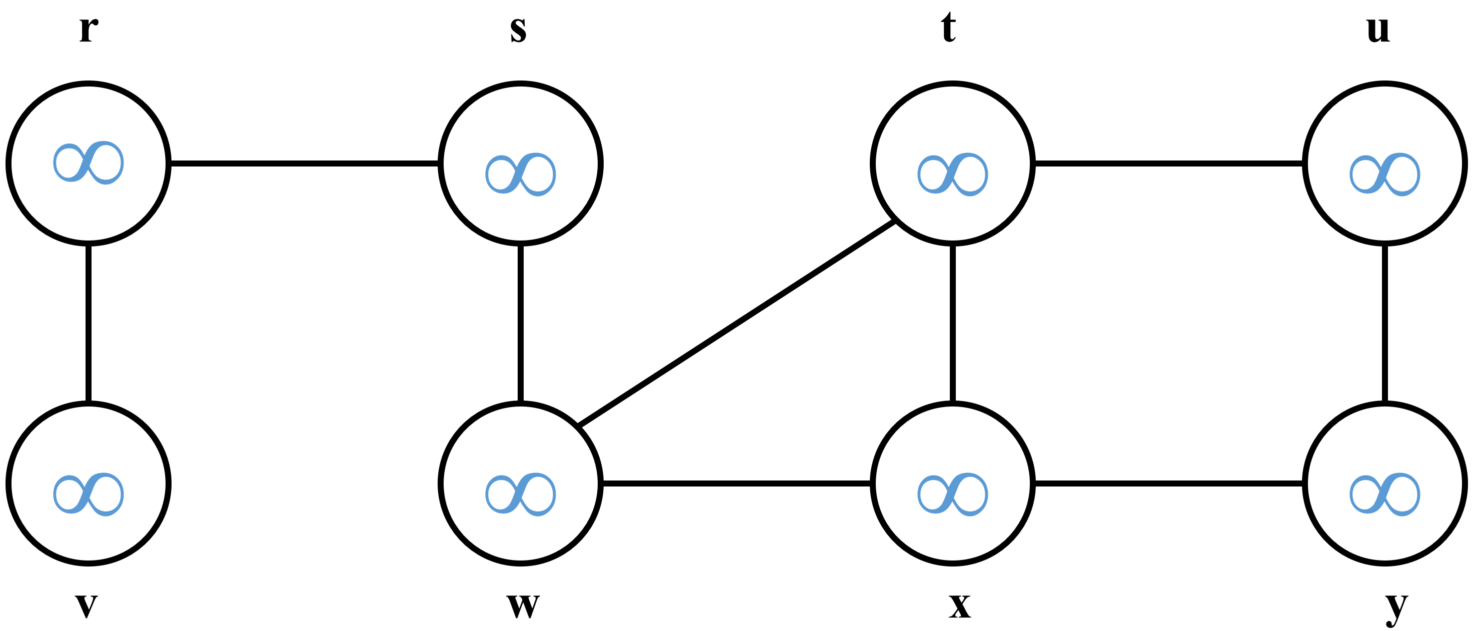
- **BFS follows the following rules:**
  - Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
  - For each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. The newly visited nodes from this level form a new level that becomes the next current level.
  - Repeat step 2 until no more nodes can be visited.
  - If there are still unvisited nodes, repeat from Step 1.

Implementation of BFS

- Observations:
  - the first node visited in each level is the first node from which to proceed to visit new nodes.

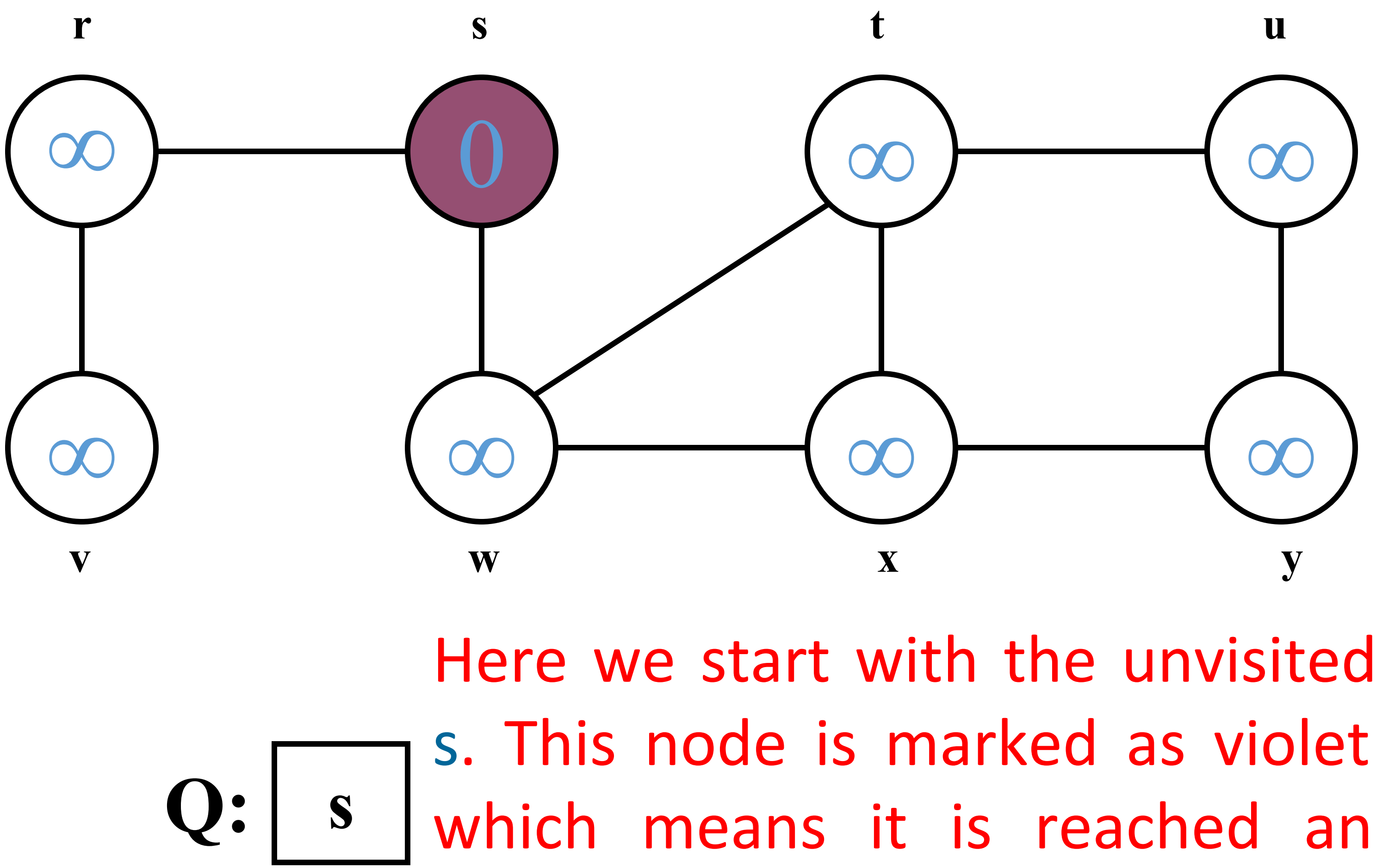
- This suggests that a ***queue*** is the proper data structure to remember the order of the steps.

Breadth-First Search: Example



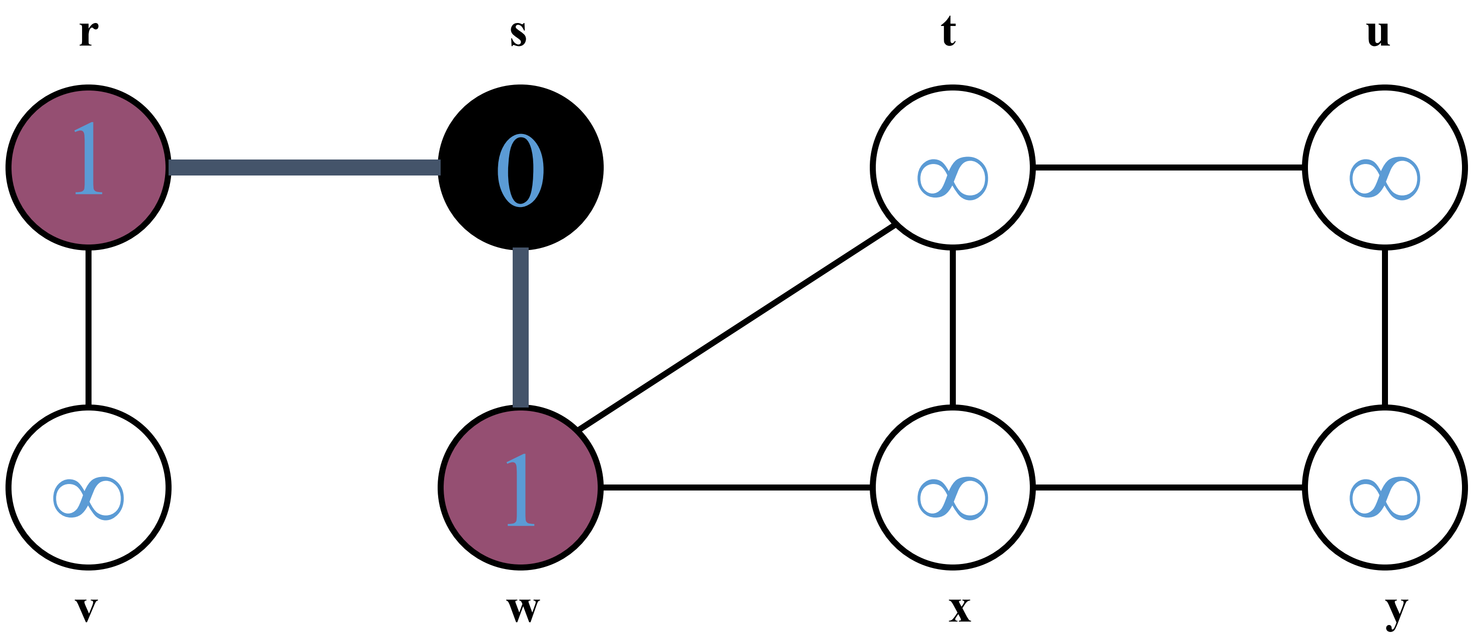
Symbol ∞ signifies the distance of the node from the root of the BFS tree. Since we do not have any visited node, all the nodes are marked as ∞. White color signifies the node as unvisited.

Breadth-First Search: Example



Here we start with the unvisited node s. This node is marked as violet color which means it is reached and the node is already in the queue.

Breadth-First Search: Example

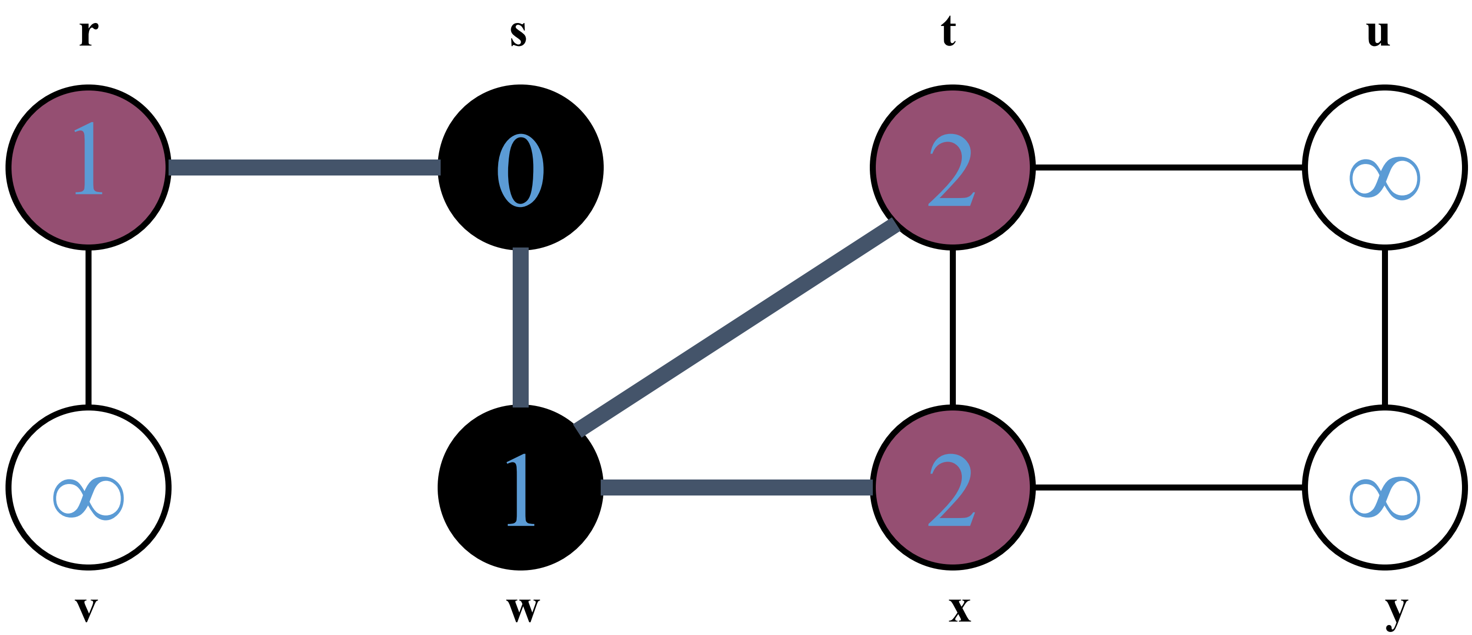


Q:

w	r

Here node s is deleted from the queue and mark as visited. This is shown as black color in the picture. The univisted neighbours of s are r and w. Both are inserted into the queue. Since node r is at the rear of the queue, it will be visited first.

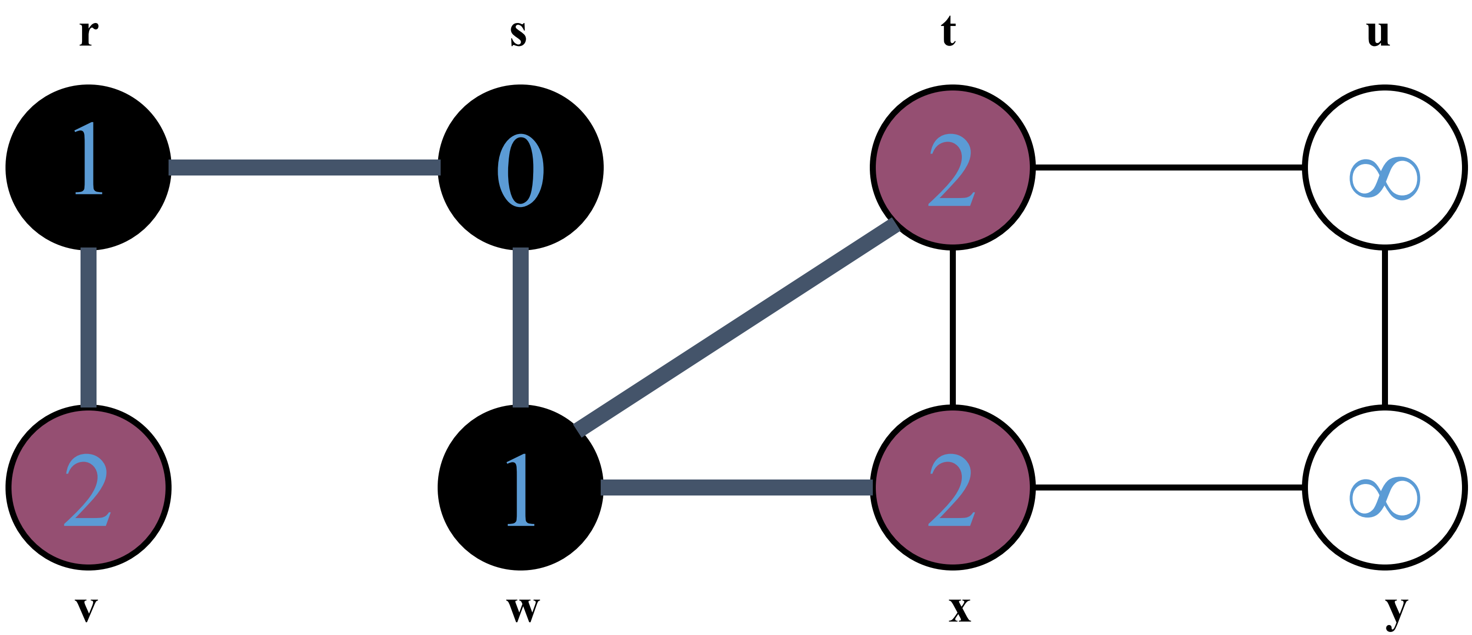
Breadth-First Search: Example



Q: | r | t | x |

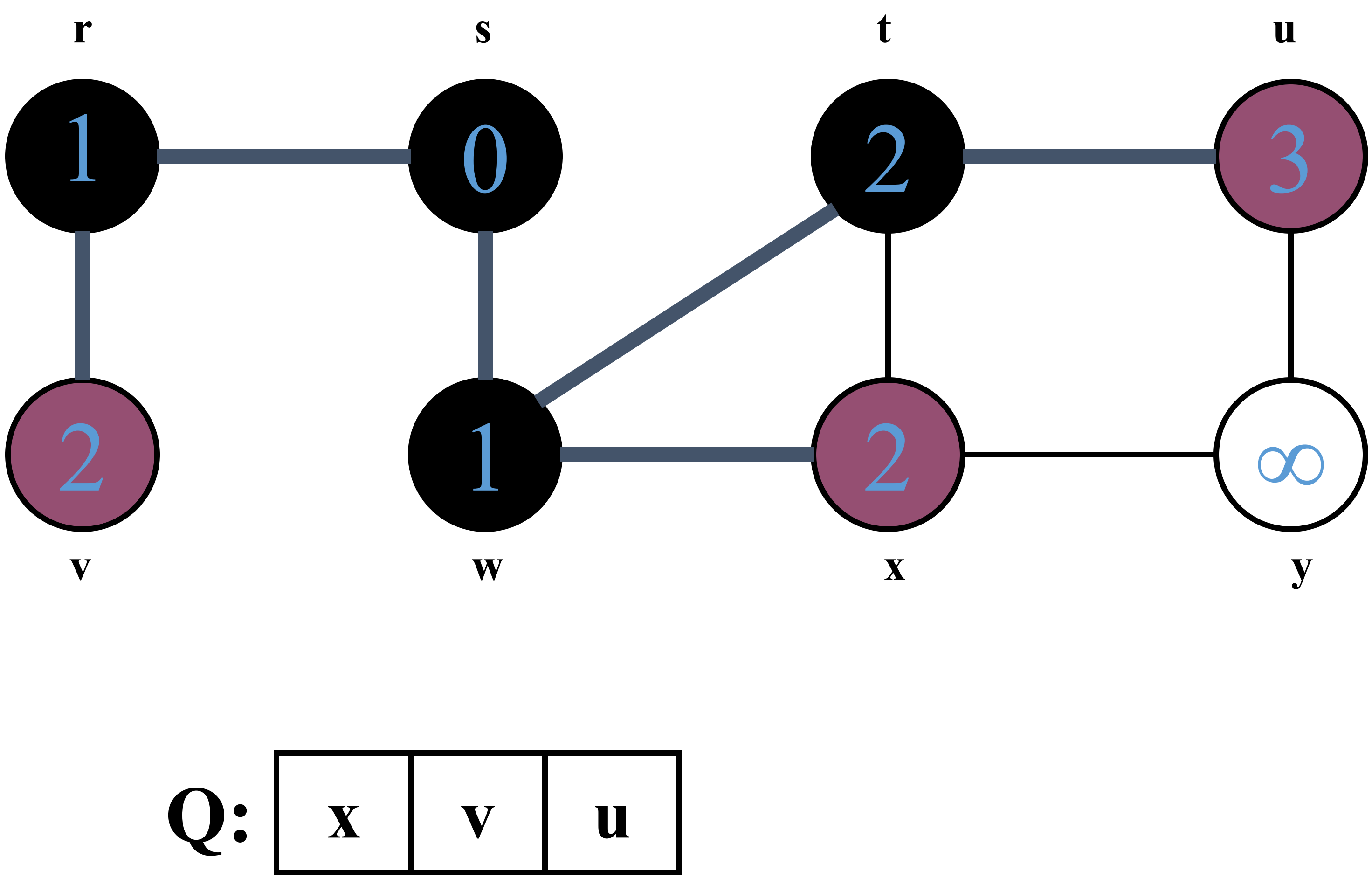
The univisted neighbours of w are t and x.

Breadth-First Search: Example

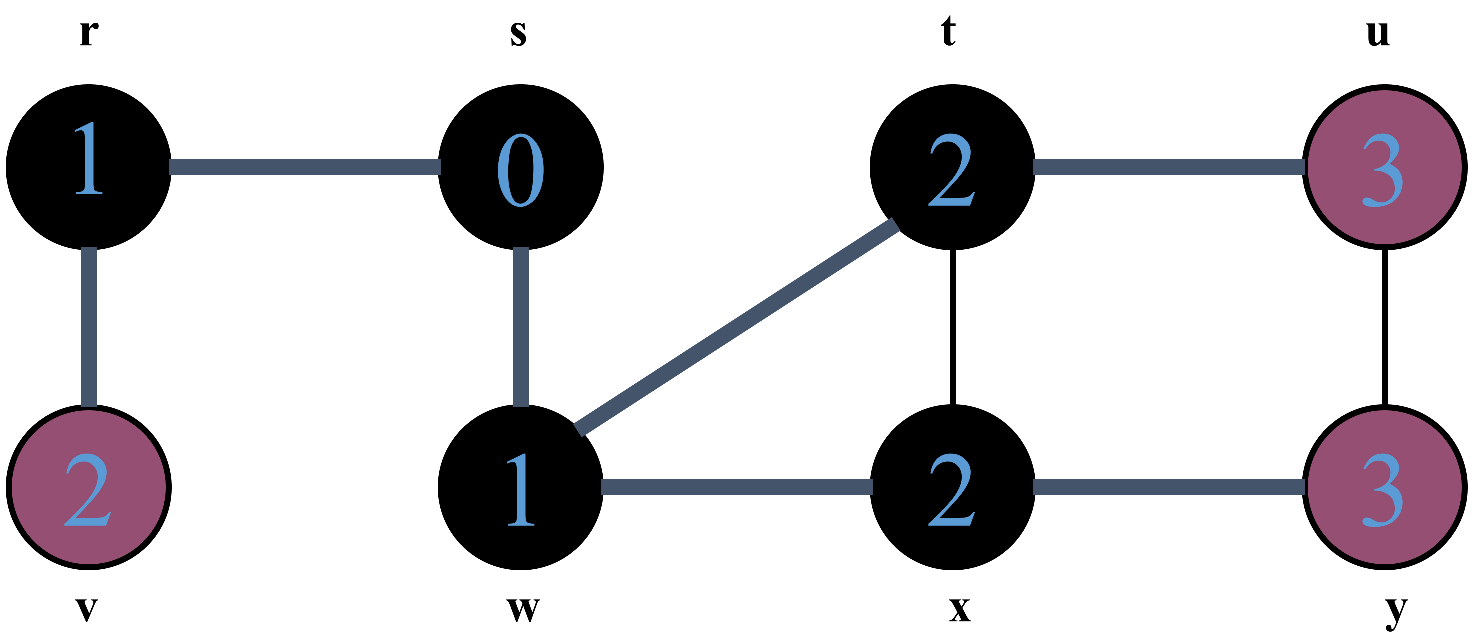


Q: | t | x | v |

Breadth-First Search: Example

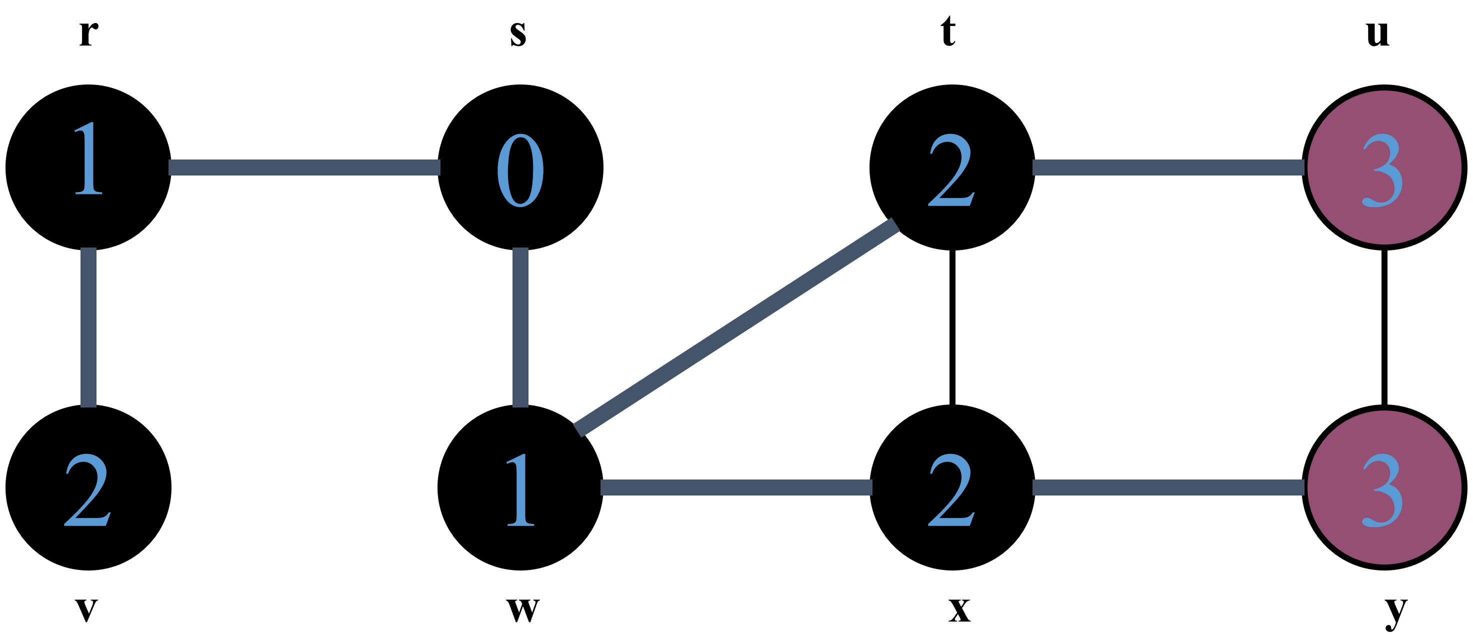


Breadth-First Search: Example



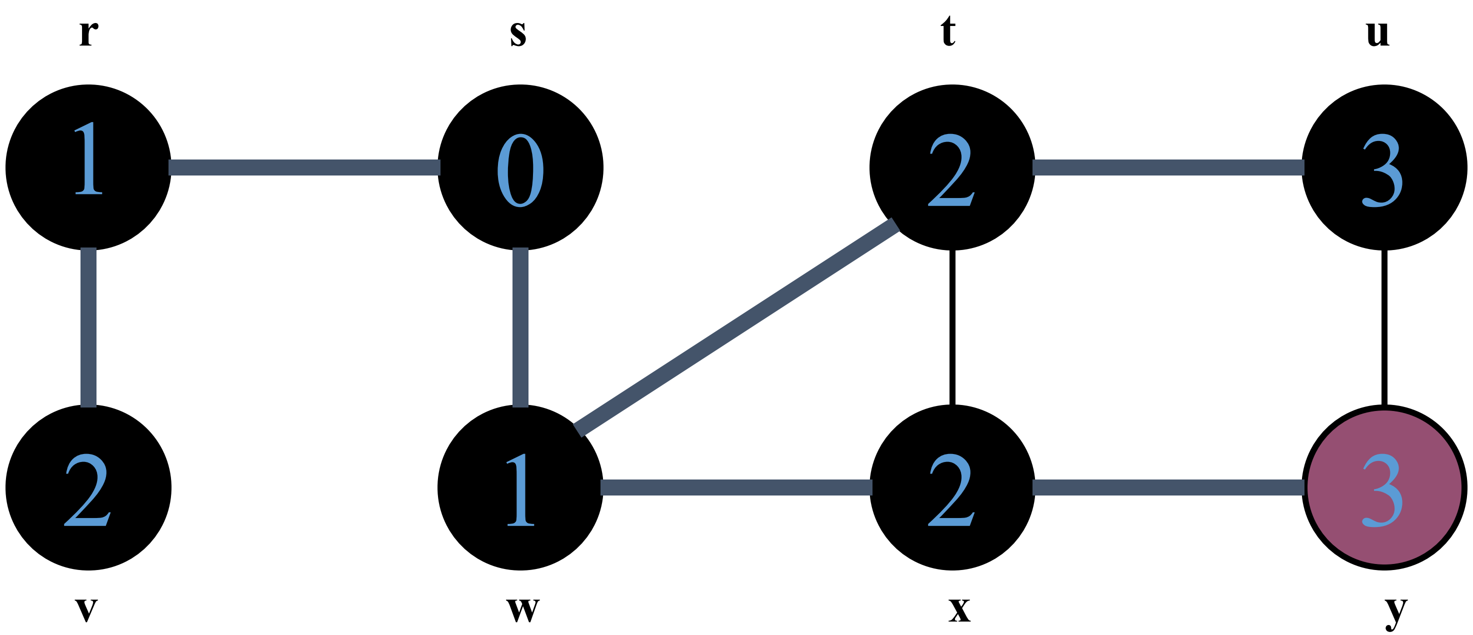
Q: | v | u | y |

Breadth-First Search: Example



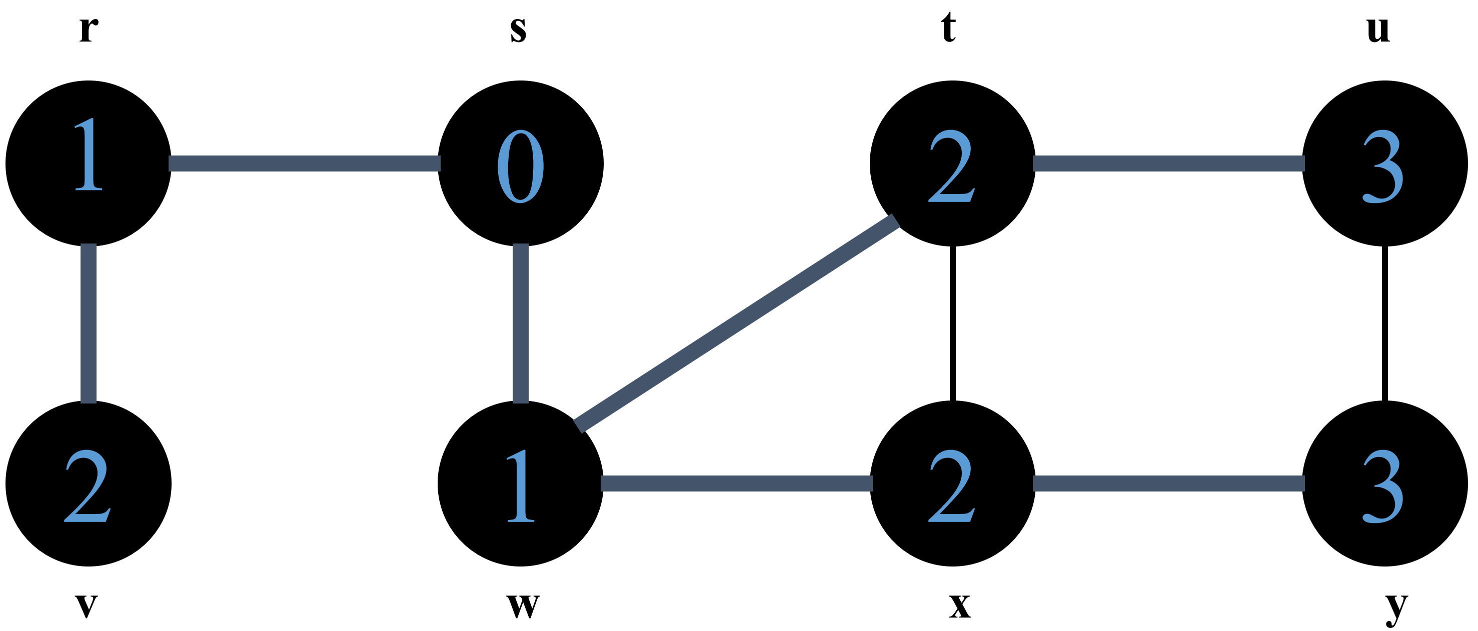
Q: | u | y |

Breadth-First Search: Example



Q: $\boxed{y}$

Breadth-First Search: Example



Q: Ø

BFS (Non Recursive algorithm)

BFS(V,E,s)

For all nodes except ***s***

$d[v]=\infty$

$d[\boldsymbol{s}]=0$

Q= s

While $Q \neq \emptyset$

u= Dequeue(Q)

for all adjacent nodes v of u

if $d[v]=\infty$

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

Enqueue(Q,v)

BFS (time complexity)

BFS(V,E,s)

For all nodes except *s*

    d[v]=∞

  d[*s*]=0

Q= Ø

While Q ≠ Ø

    u= Dequeue(Q)

    for all adjacent nodes v of u

        if d[v]= ∞

            then d[v]=d[u]+1

            Enqueue(Q,v)

O(|V|)

O(V^2) = |E|).

Overall complexity = O(|V|)+ |E|)

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 $\boldsymbol{v}$ that still has unexplored edges.
  - When all of ***v*'s** edges have been explored, backtrack to the vertex from which $\boldsymbol{v}$ was discovered.

Depth-First Search

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

DFS Algorithm

DFS(V,E)

```
for each vertex u in V[G]
    color[v] ← WHITE
    π[v] ← NIL
    time ← 0
for each vertex v in V[G]
    if color[v] ← WHITE
       then
          Depth_First_Search(v)
```

Depth_First_Search(v)

```
color[v] ← GRAY
time ← time + 1
d[v] ← time
for each vertex u adjacent to v
    if color[u] ← WHITE
        π[u] ← v
        Depth_First_Search(u)
color[v] ← BLACK
time ← time + 1
f[v] ← time
```

Explanation of the algorithm

- Create and maintain 4 variables for each vertex of the graph, *such as,* color[v], d[v], f[v], π[v].
- **<u>color[v]</u> :**
  - This variable represents the color of the vertex ‘v’ at the given point of time.
  - The possible values of this variable are- WHITE, GREY and BLACK.
  - WHITE color of the vertex signifies that it has not been discovered yet.
  - GREY color of the vertex signifies that it has been discovered and it is being processed.
  - BLACK color of the vertex signifies that it has been completely processed.
- **π<u>[v]</u> :** This variable represents the predecessor of vertex ‘v’.
- **<u>d[v]</u> :** This variable represents a timestamp when a vertex ‘v’ is discovered.
- **<u>f[v]</u> :** This variable represents a timestamp when the processing of vertex ‘v’ is completed.

Explanation of the algorithm

- For each vertex of the graph, initialize the variables as-
  - color[v] = WHITE
  - π[v] = NIL
  - time = 0   (Global Variable acting as a timer)
- Repeat the Depth_First_Search procedure until all the vertices of the graph become BLACK. Consider any white vertex 'v' and call the Depth_First_Search function on it.
- **DFS Time Complexity-** The total running time for Depth First Search is θ (V+E).

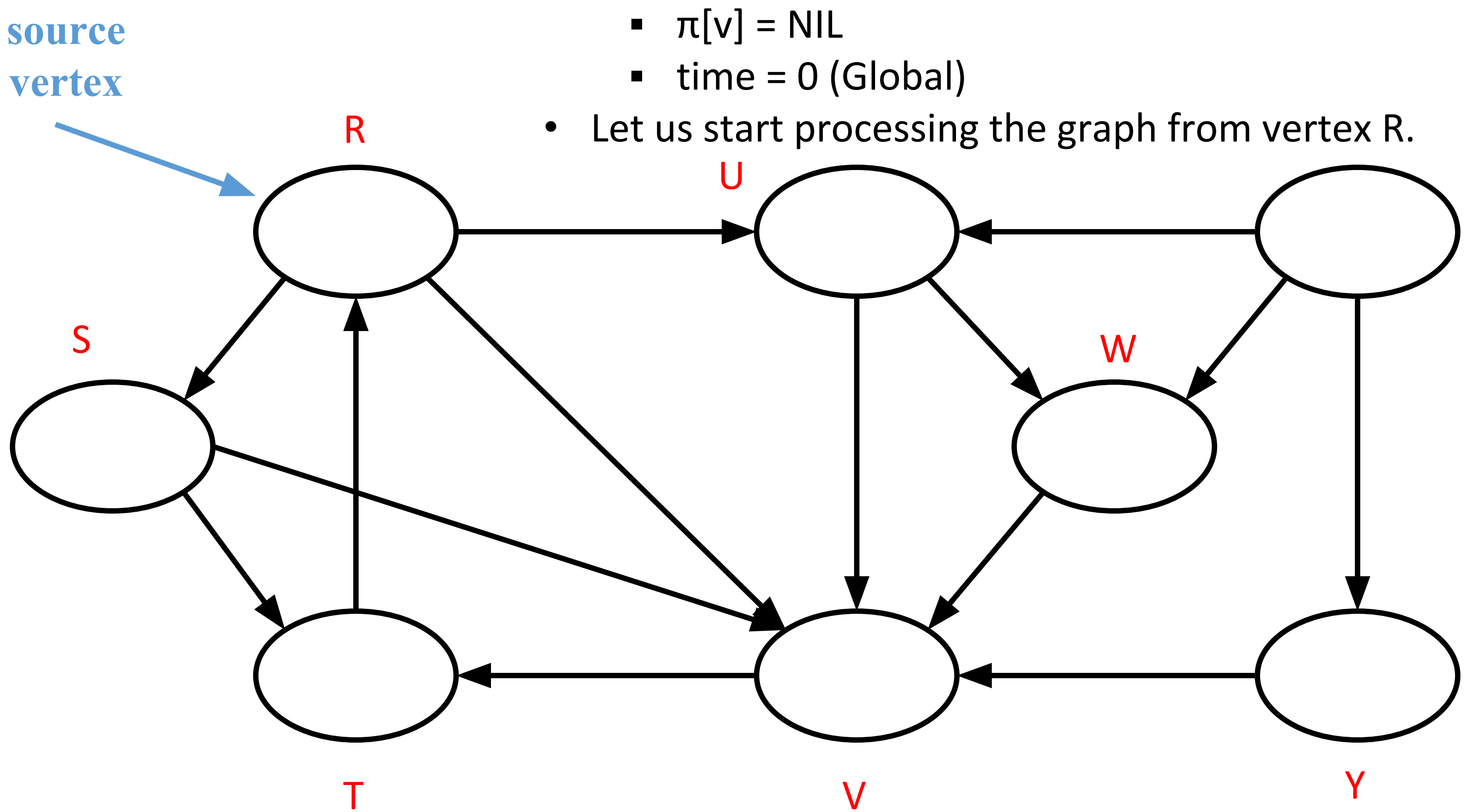
Types of Edges in DFS

- After a DFS traversal of any graph G, all its edges can be put in one of the following 4 classes:
  - ✔ Tree Edge, Back Edge, Forward Edge, Cross Edge
- **Tree edge**: A tree edge is an edge that is included in the DFS tree.
- **Back edge:** An edge from a vertex ‘u’ to one of its ancestors ‘v’ is called as a back edge. A self-loop is considered as a back edge. A back edge is discovered when
  - DFS tries to extend the visit from a vertex ‘u’ to vertex ‘v’ and
  - vertex ‘v’ is found to be an ancestor of vertex ‘u’ and grey at that time.
- **Forward edge:** An edge from a vertex ‘u’ to one of its descendants ‘v’ is called as a forward edge. A forward edge is discovered when
  - DFS tries to extend the visit from a vertex ‘u’ to a vertex ‘v’
  - And finds that color(v) = BLACK and d(v) > d(u).
- **Cross edge:** An edge from a vertex ‘u’ to a vertex ‘v’ that is neither its ancestor nor its descendant is called as a cross edge. A cross edge is discovered when
  - DFS tries to extend the visit from a vertex ‘u’ to a vertex ‘v’
  - And finds that color(v) = BLACK and d(v) < d(u).

DFS Example

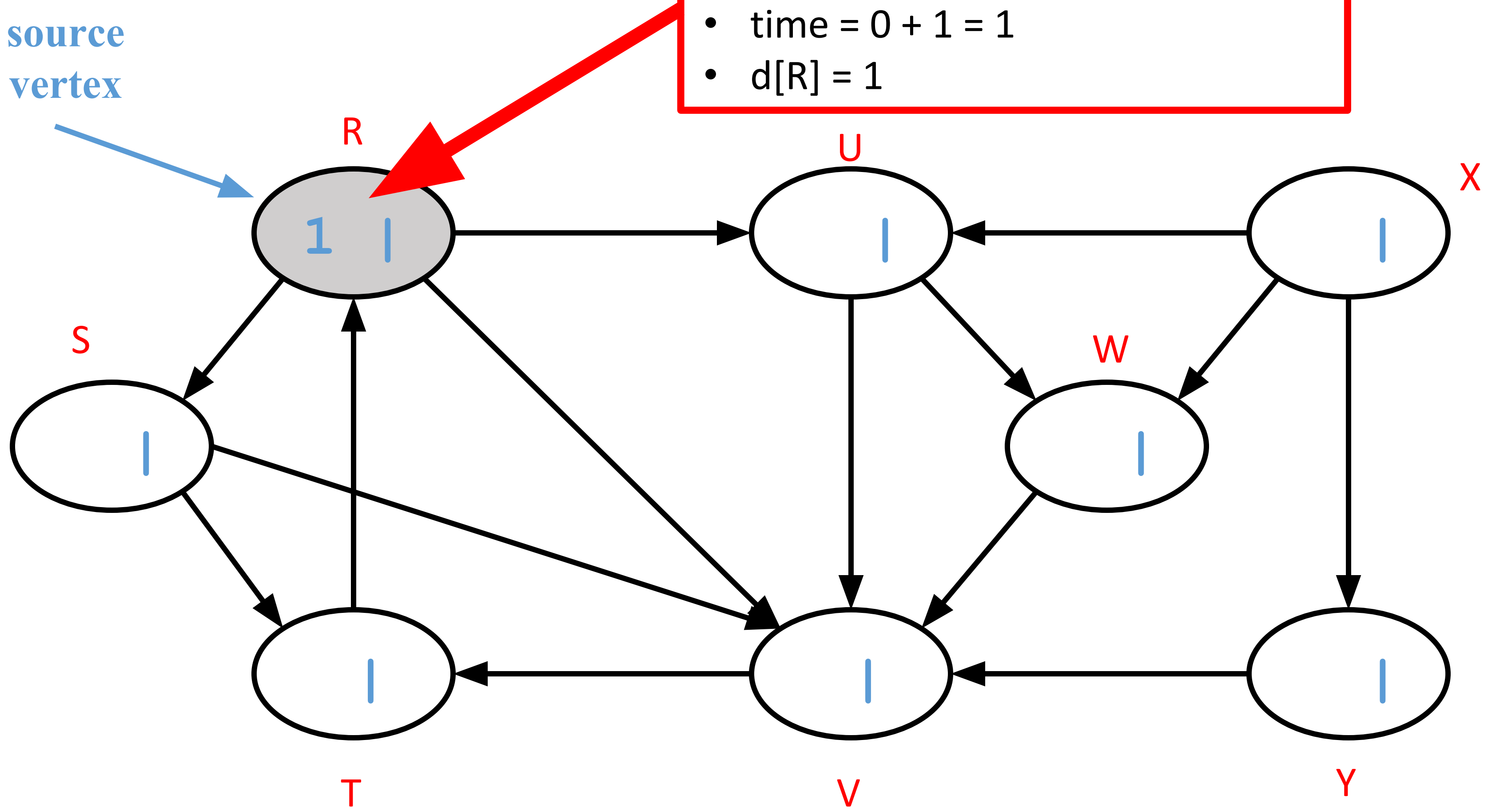
We will traverse the given graph using DFS.

- Initially for all the vertices of the graph, we set the variables as-
  - color[v] = WHITE
  - π[v] = NIL
  - time = 0 (Global)
- Let us start processing the graph from vertex R.



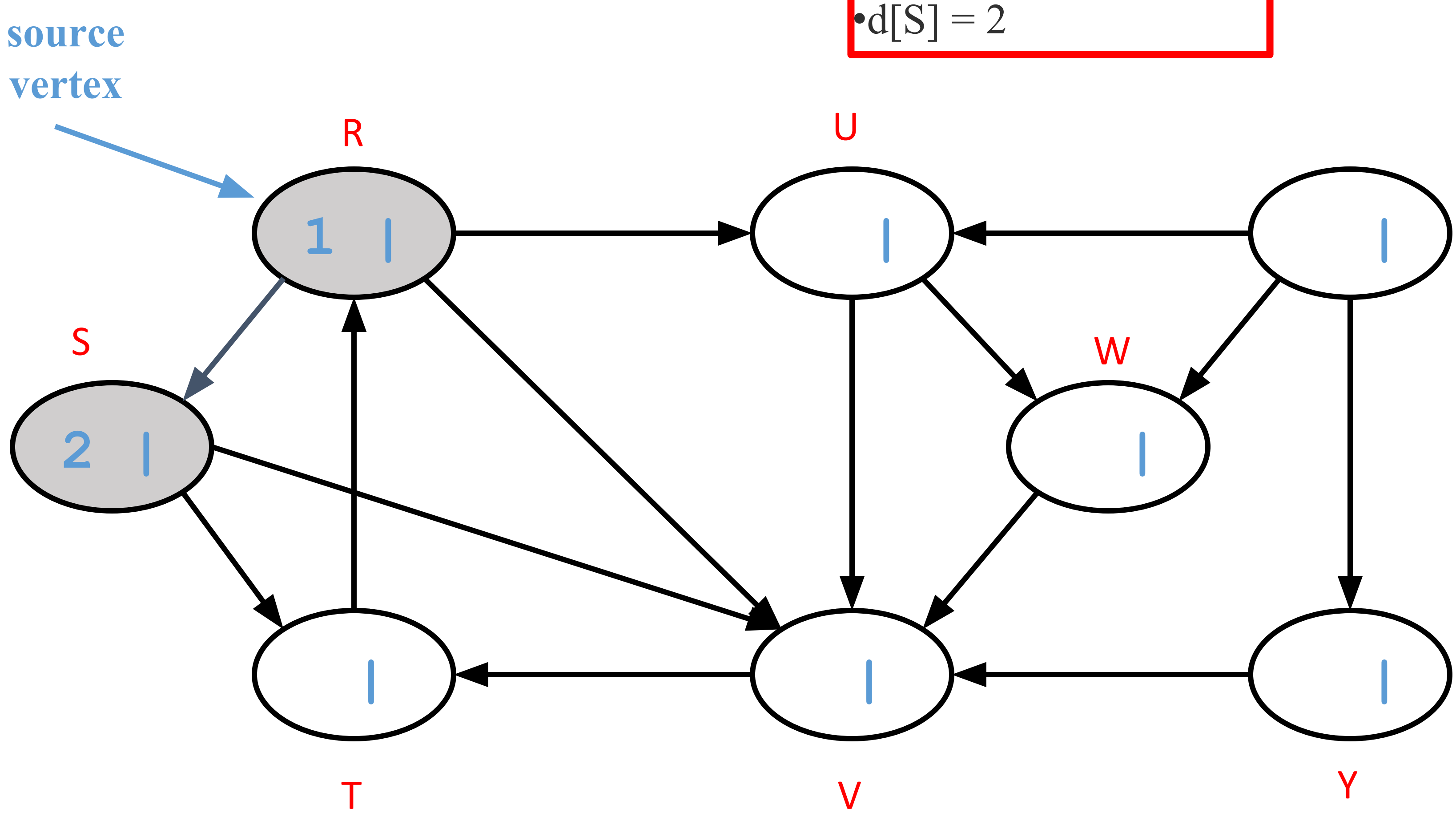
DFS Example

- color[R] = GREY
- time = 0 + 1 = 1
- d[R] = 1



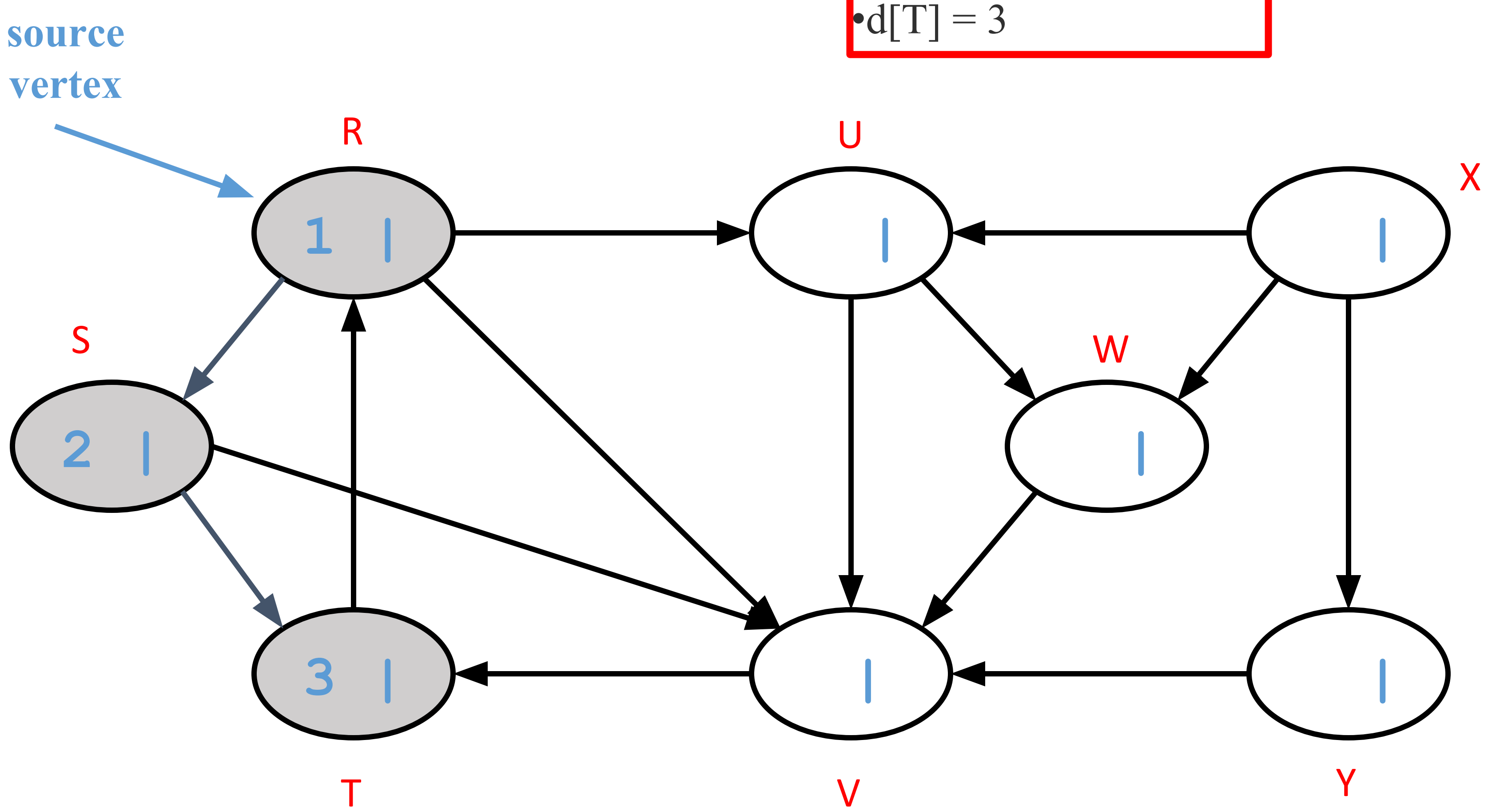
DFS Example

- $\pi[S] = R$
- $color[S] = GREY$
- $time = 1 + 1 = 2$
- $d[S] = 2$



DFS Example

- $\pi[T] = S$
- $color[T] = GREY$
- $time = 1 + 2 = 3$
- $d[T] = 3$



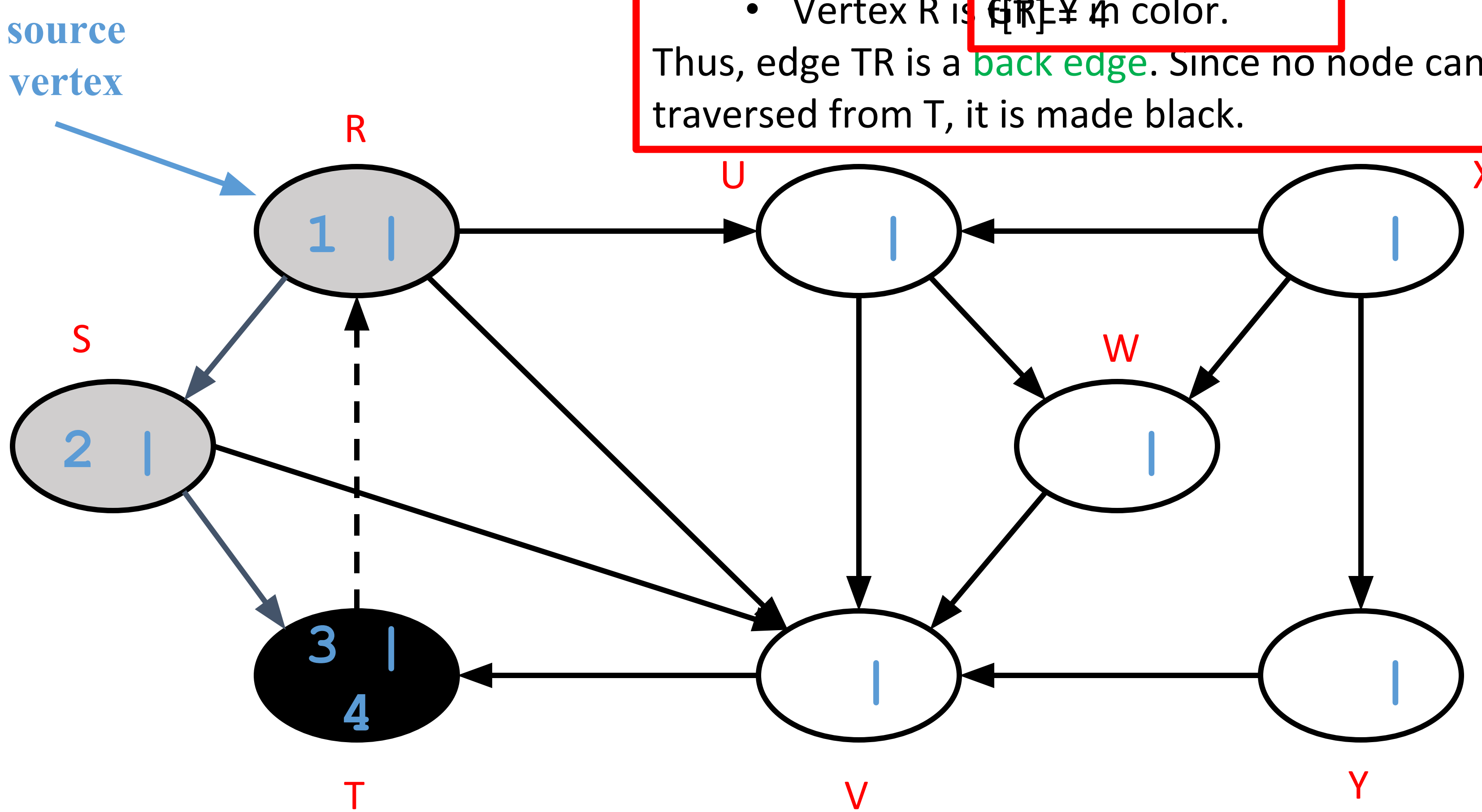
DFS Example

When DFS tries to extend the visit from vertex T to vertex R, it finds

- Vertex R is an ancestor of vertex T since it has already been discovered.
- Vertex R is GREY in color.

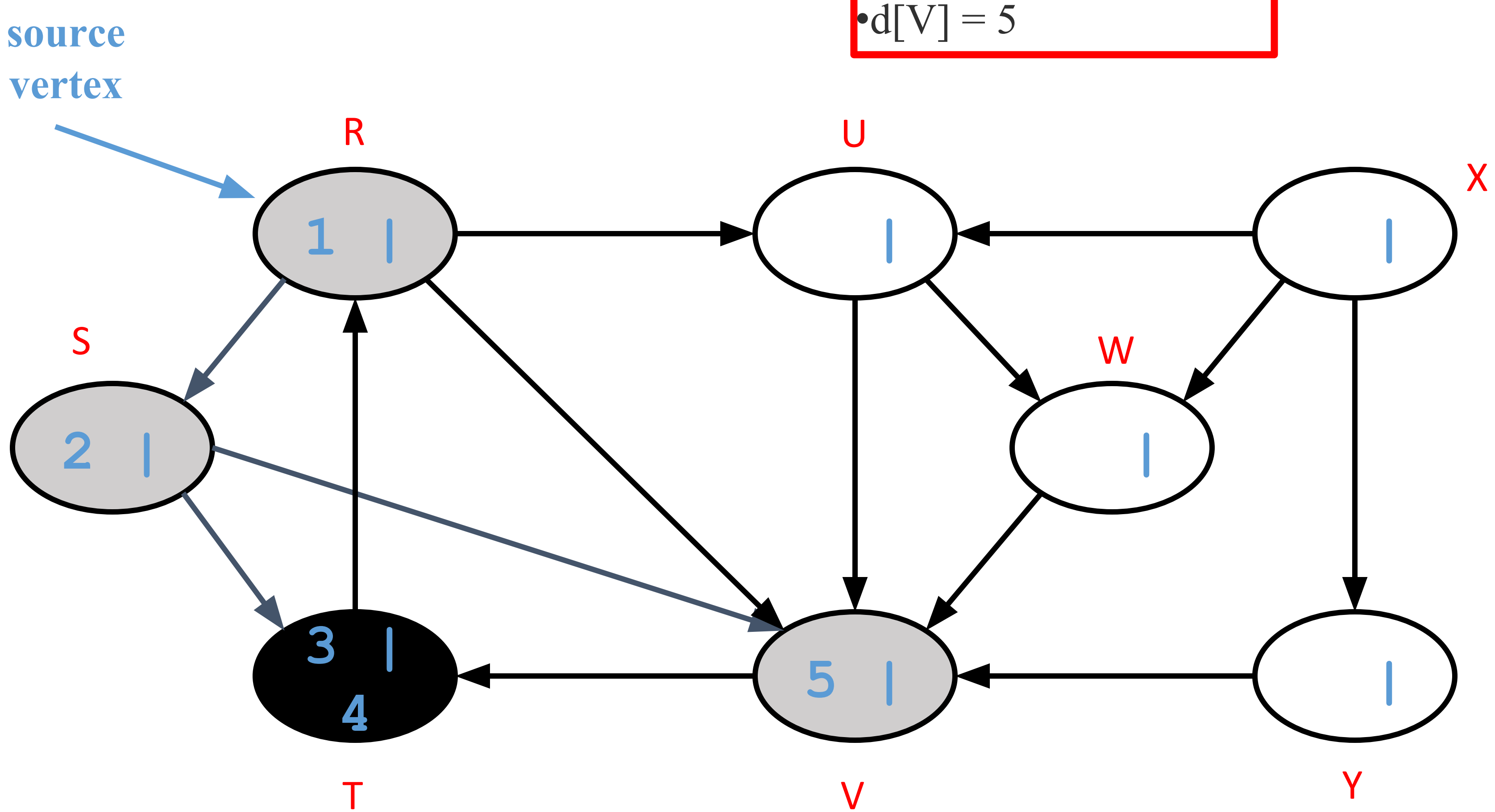
Thus, edge TR is a back edge. Since no node can be traversed from T, it is made black.

color[T] = BLACK
time = 3 + 1 = 4
f[T] = 4



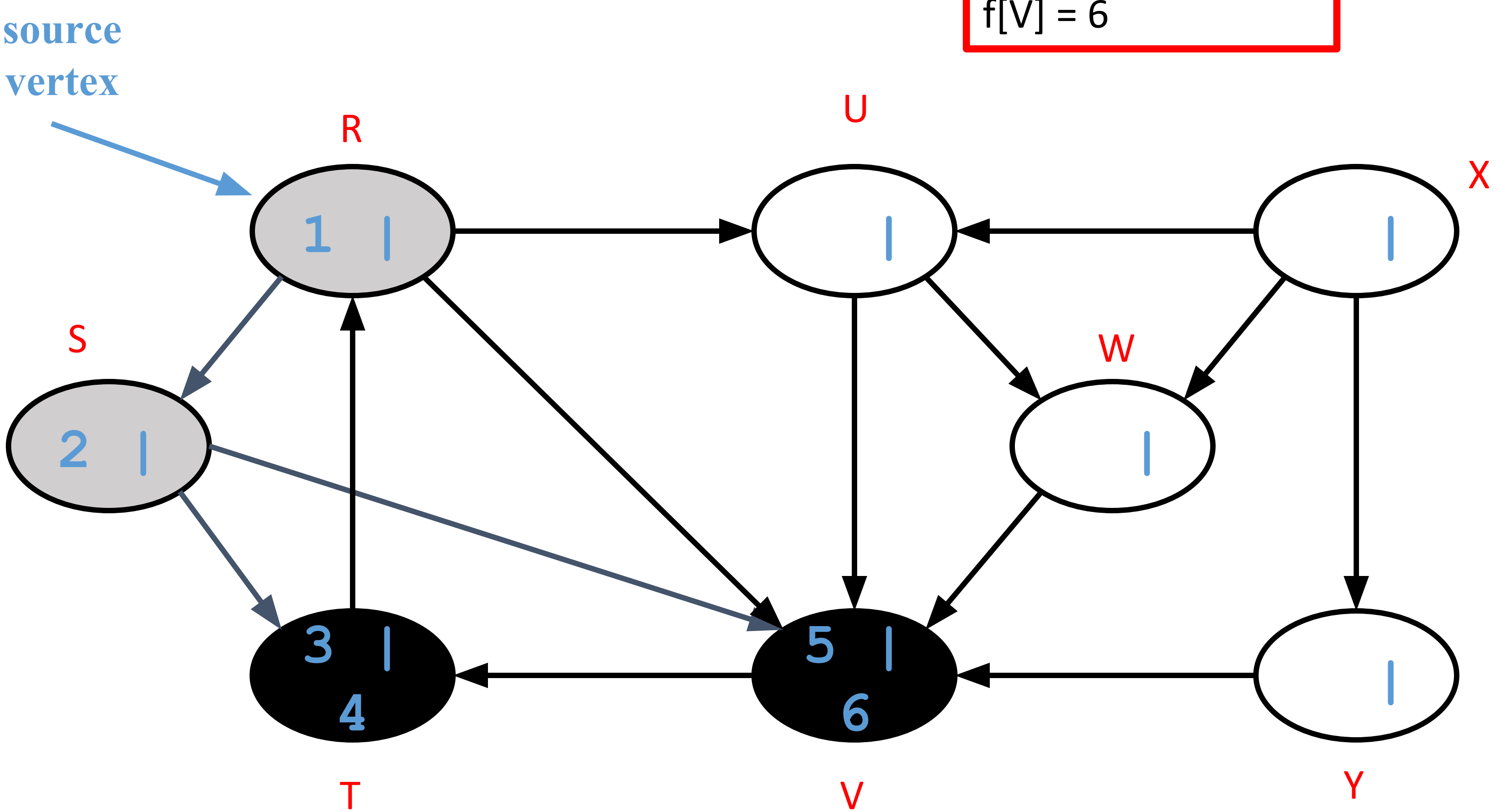
DFS Example

- $\pi[V] = S$
- $color[V] = GREY$
- $time = 4 + 1 = 5$
- $d[V] = 5$



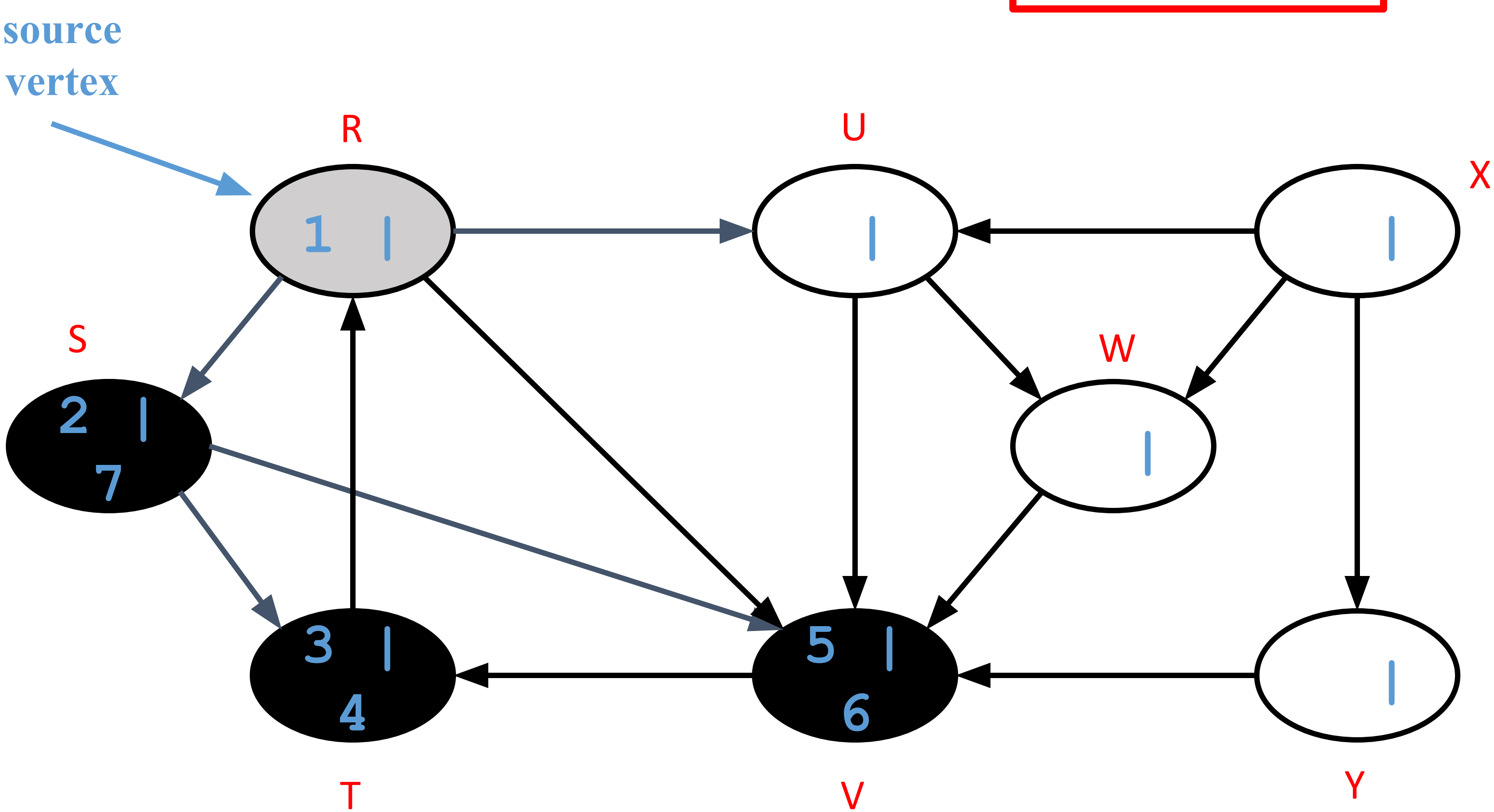
DFS Example

color[V] = BLACK
time = 5 + 1 = 6
f[V] = 6



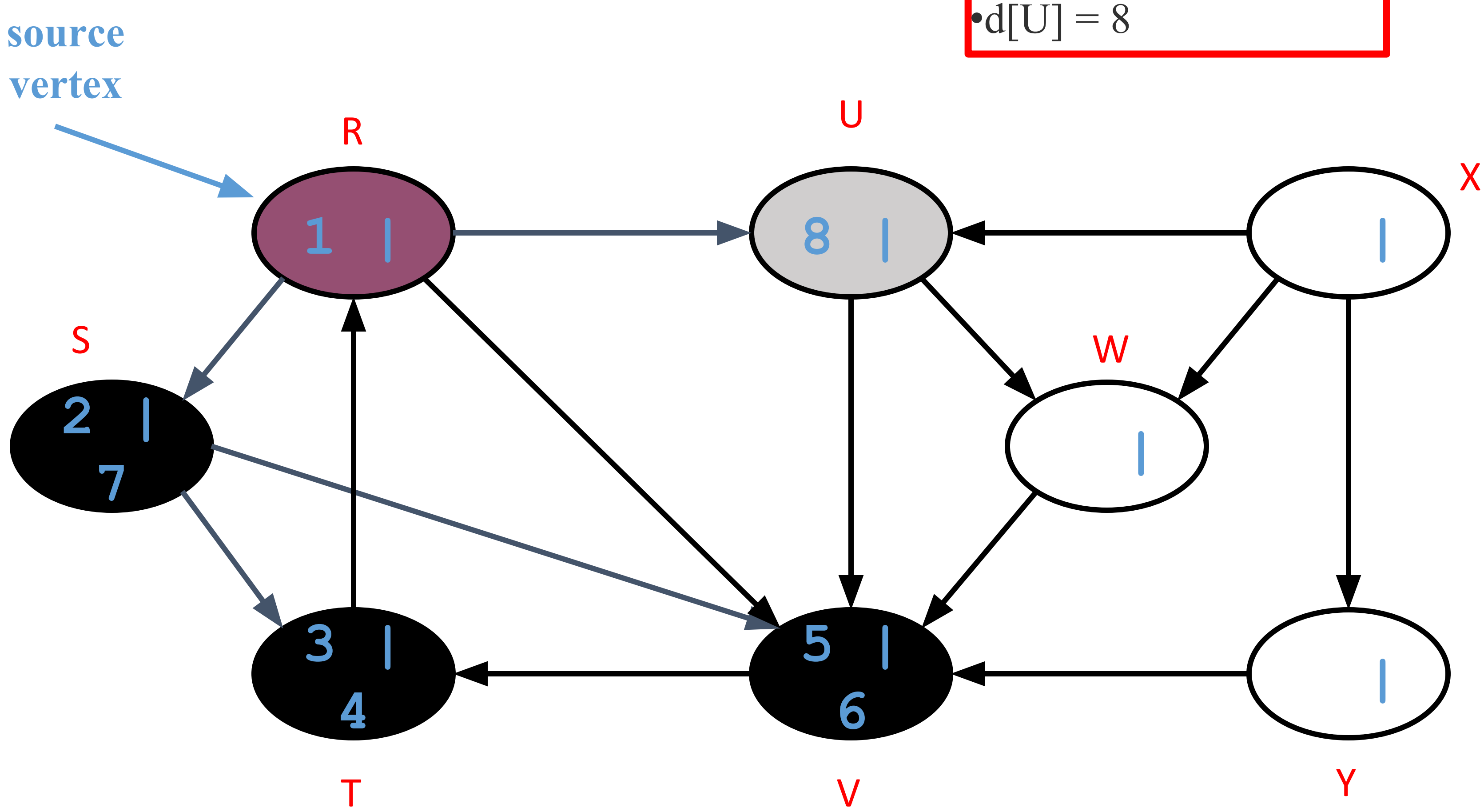
DFS Example

color[S] = BLACK
time = 6 + 1 = 7
f[S] = 7



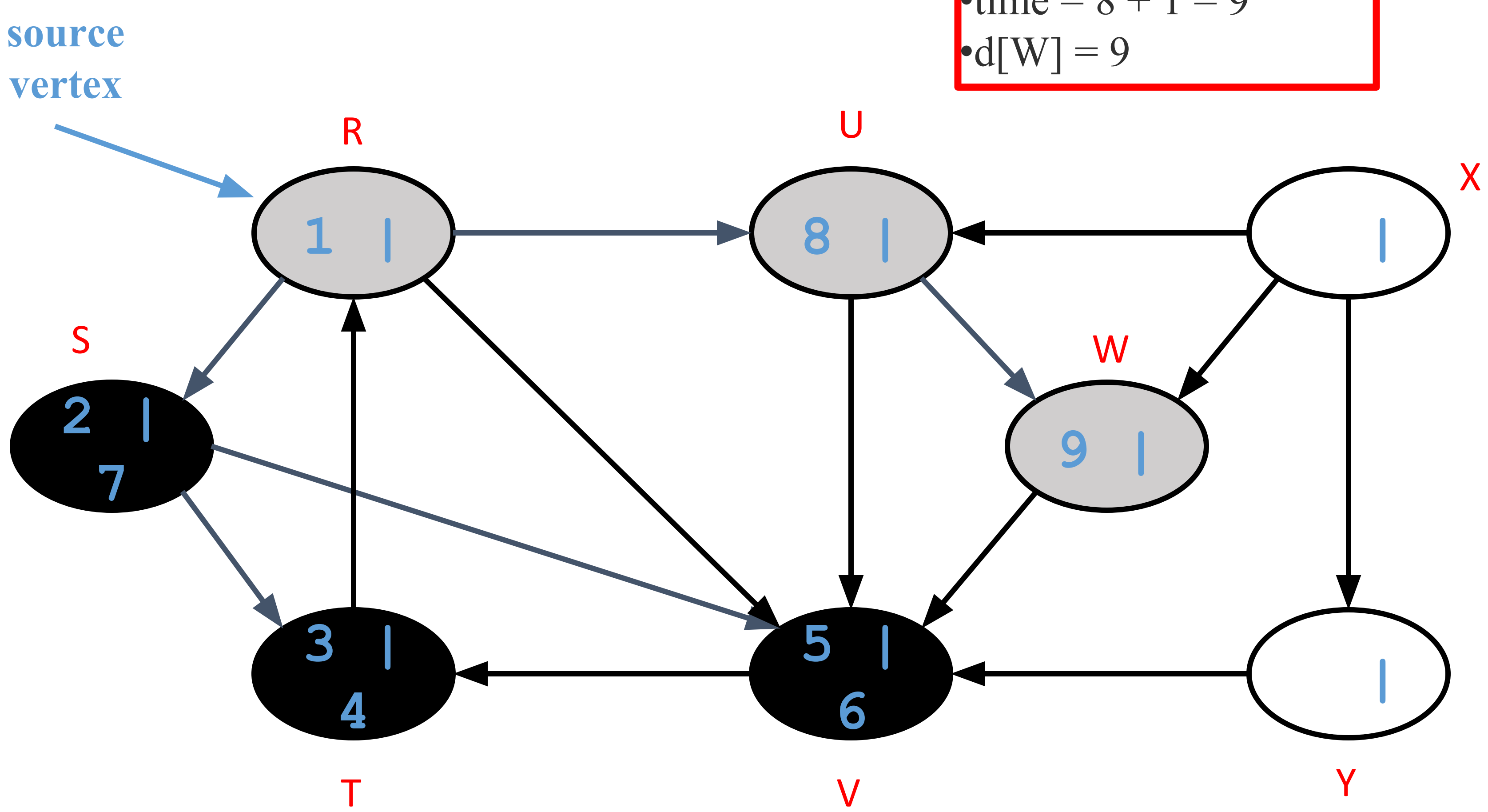
DFS Example

- $\pi[U] = R$
- $color[U] = GREY$
- $time = 7 + 1 = 8$
- $d[U] = 8$



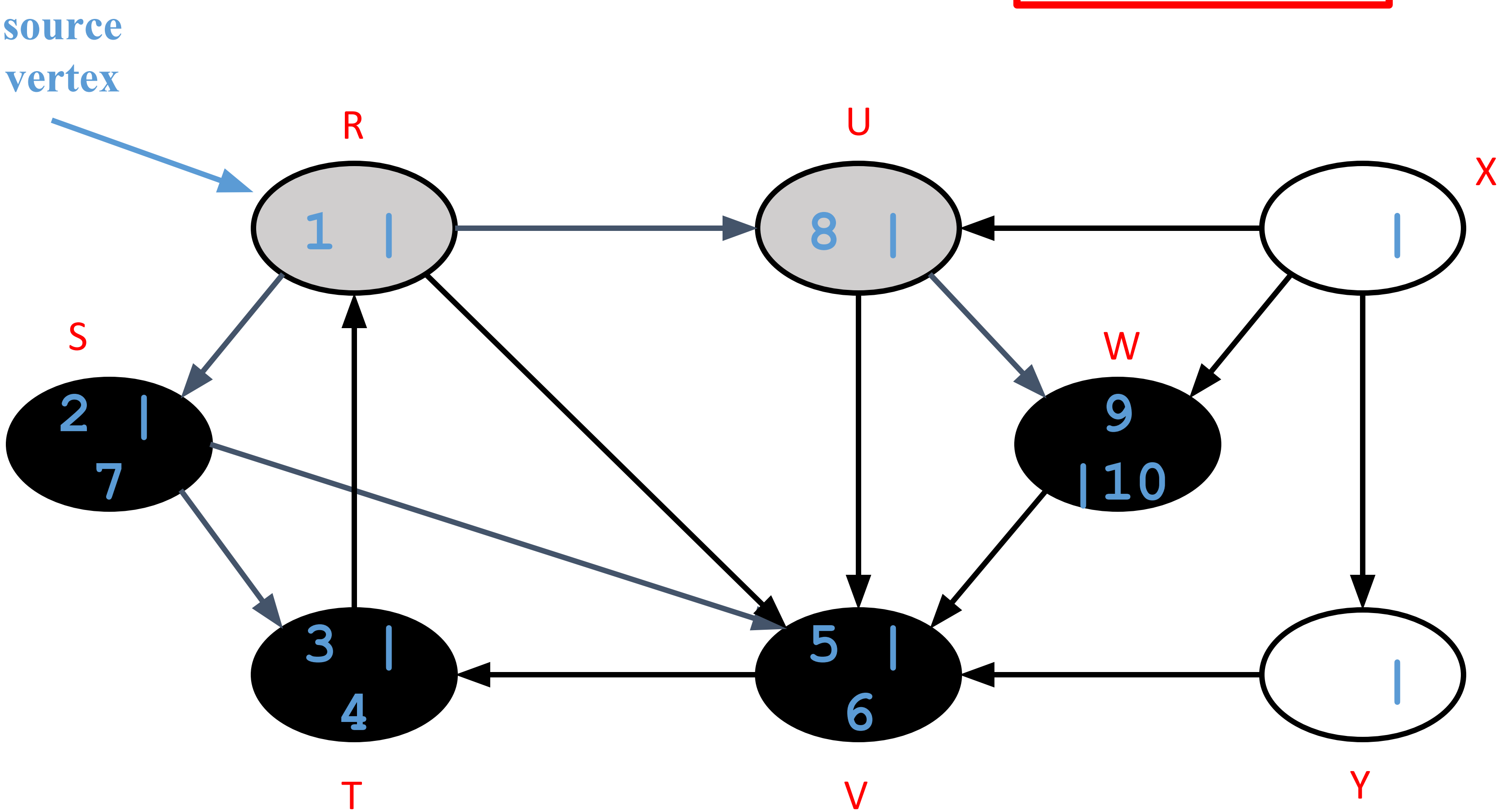
DFS Example

- $\pi[W] = U$
- color[W] = GREY
- time = 8 + 1 = 9
- d[W] = 9



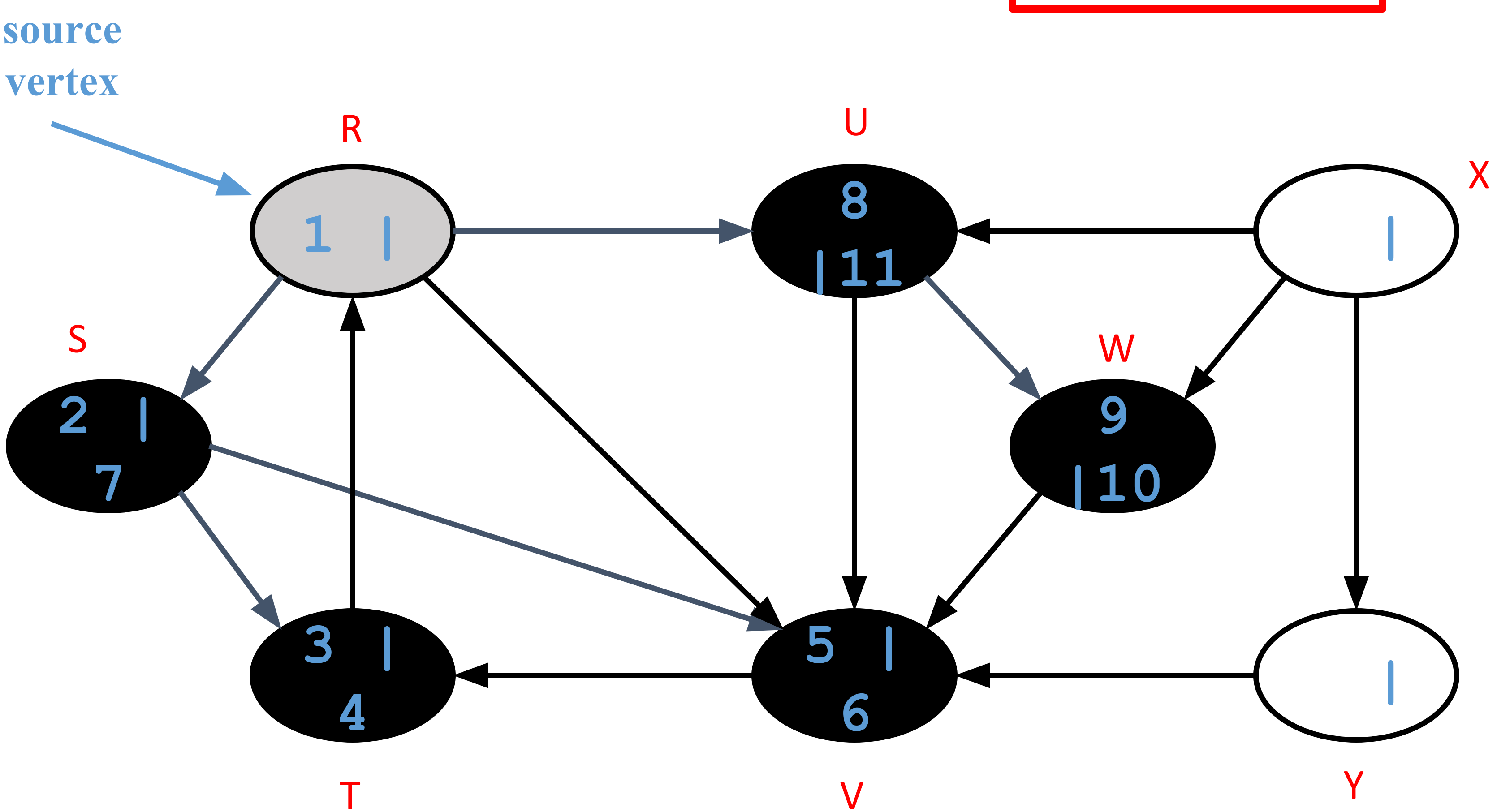
DFS Example

color[W] = BLACK
time = 9 + 1 = 10
f[S] = 10



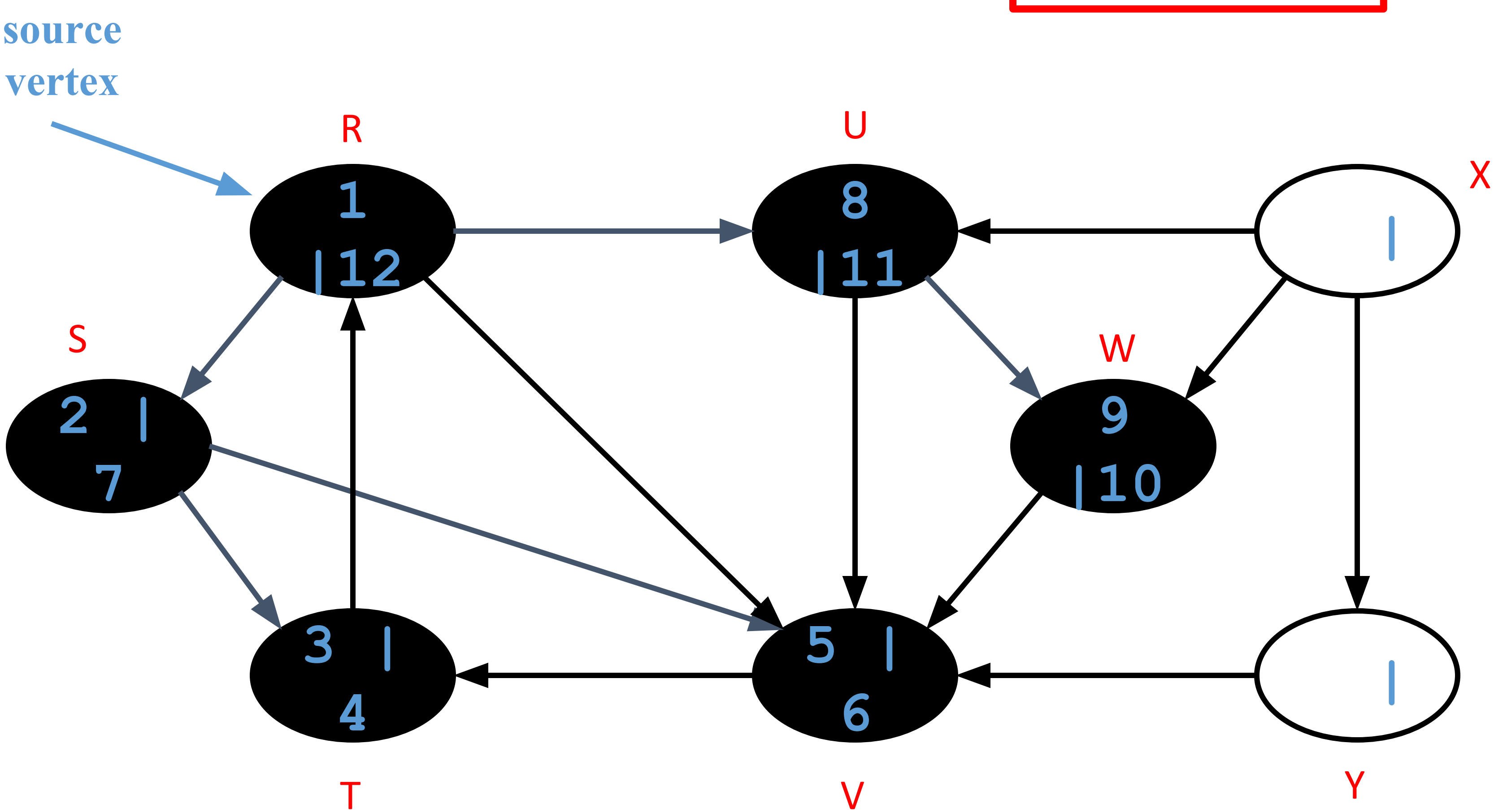
DFS Example

color[U] = BLACK
time = 10 + 1 = 11
f[S] = 11



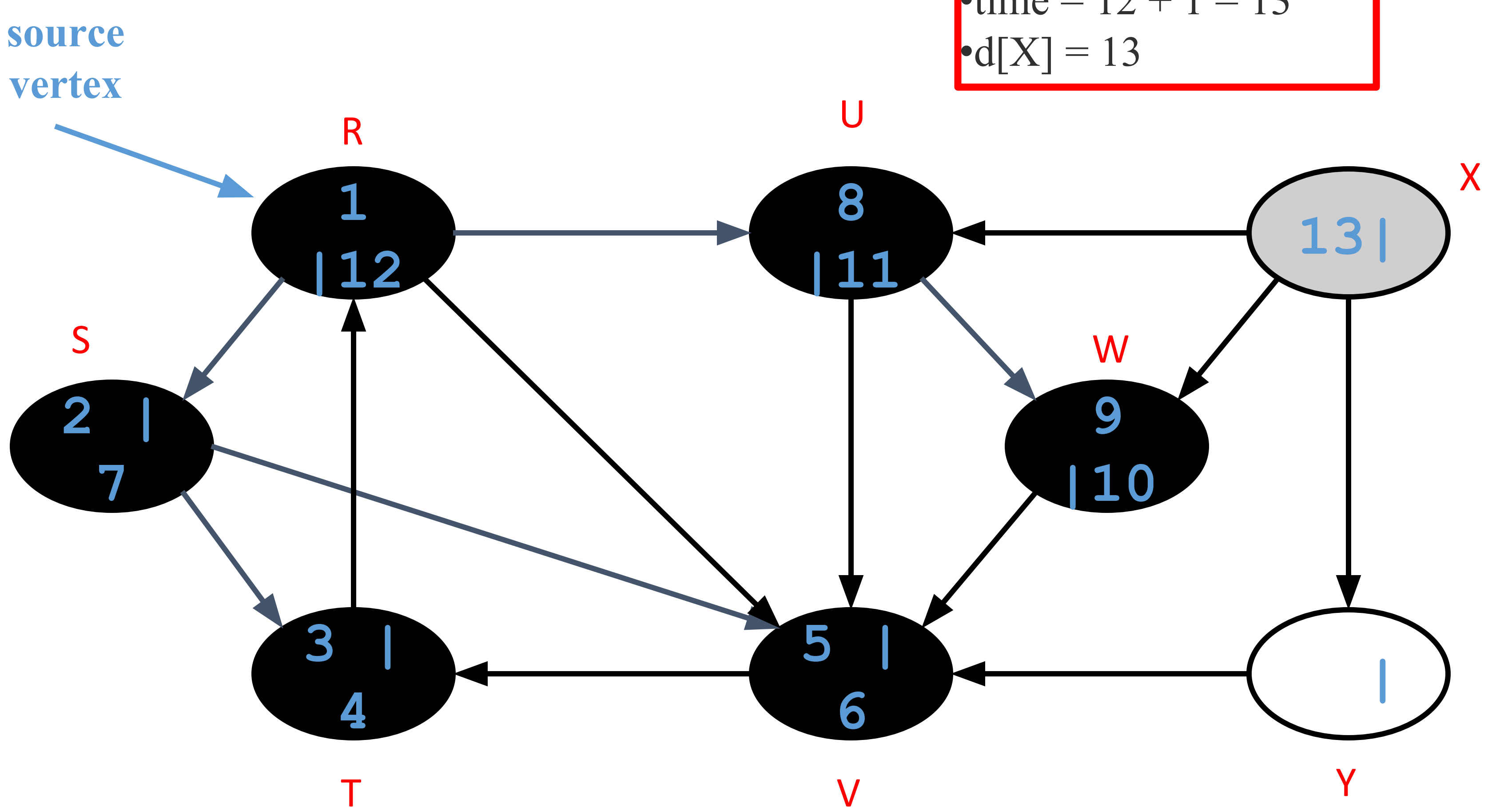
DFS Example

color[R] = BLACK
time = 11 + 1 = 12
f[S] = 12



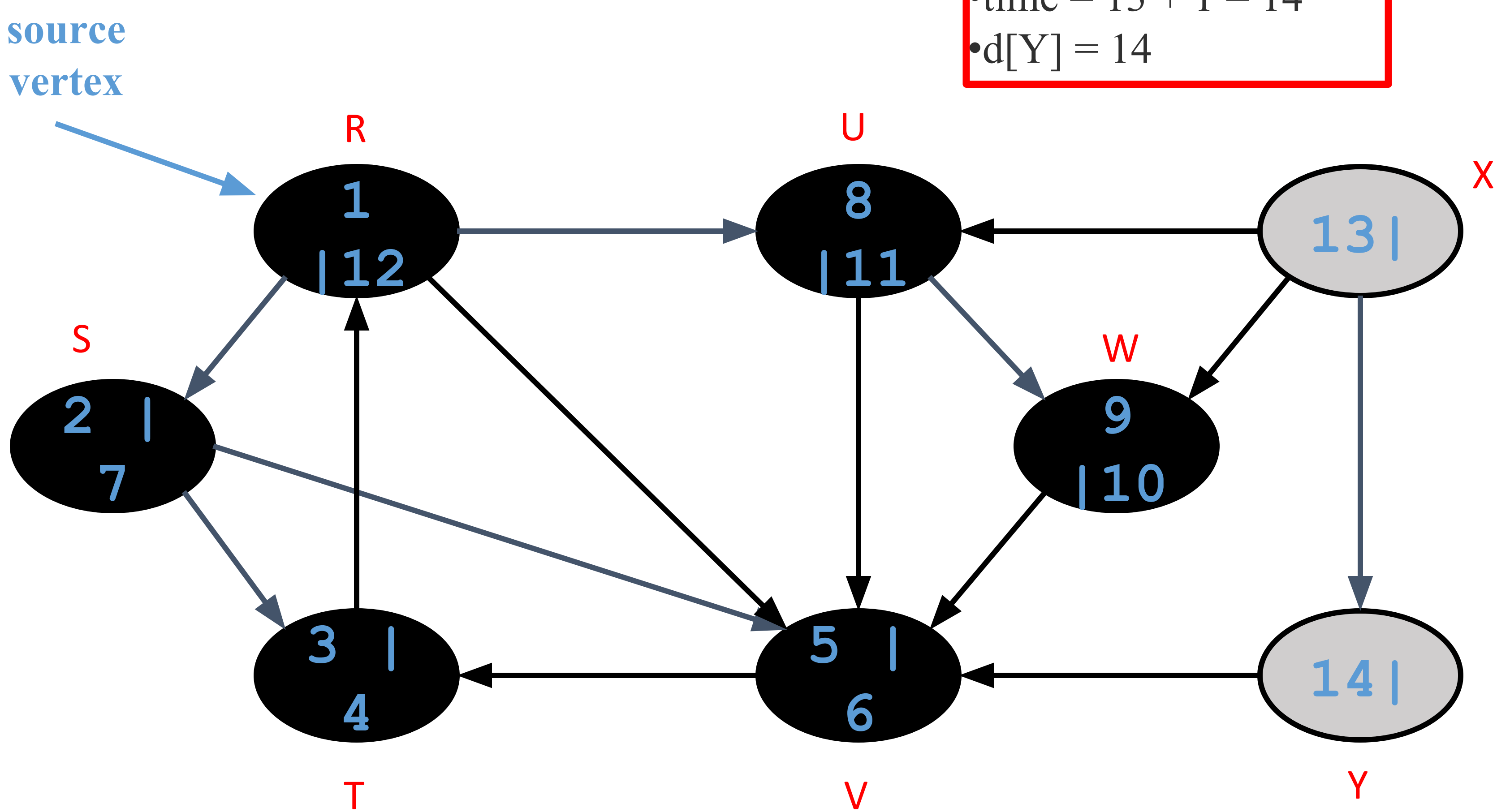
DFS Example

- $\pi[X] = X$
- color[X] = GREY
- time = 12 + 1 = 13
- d[X] = 13



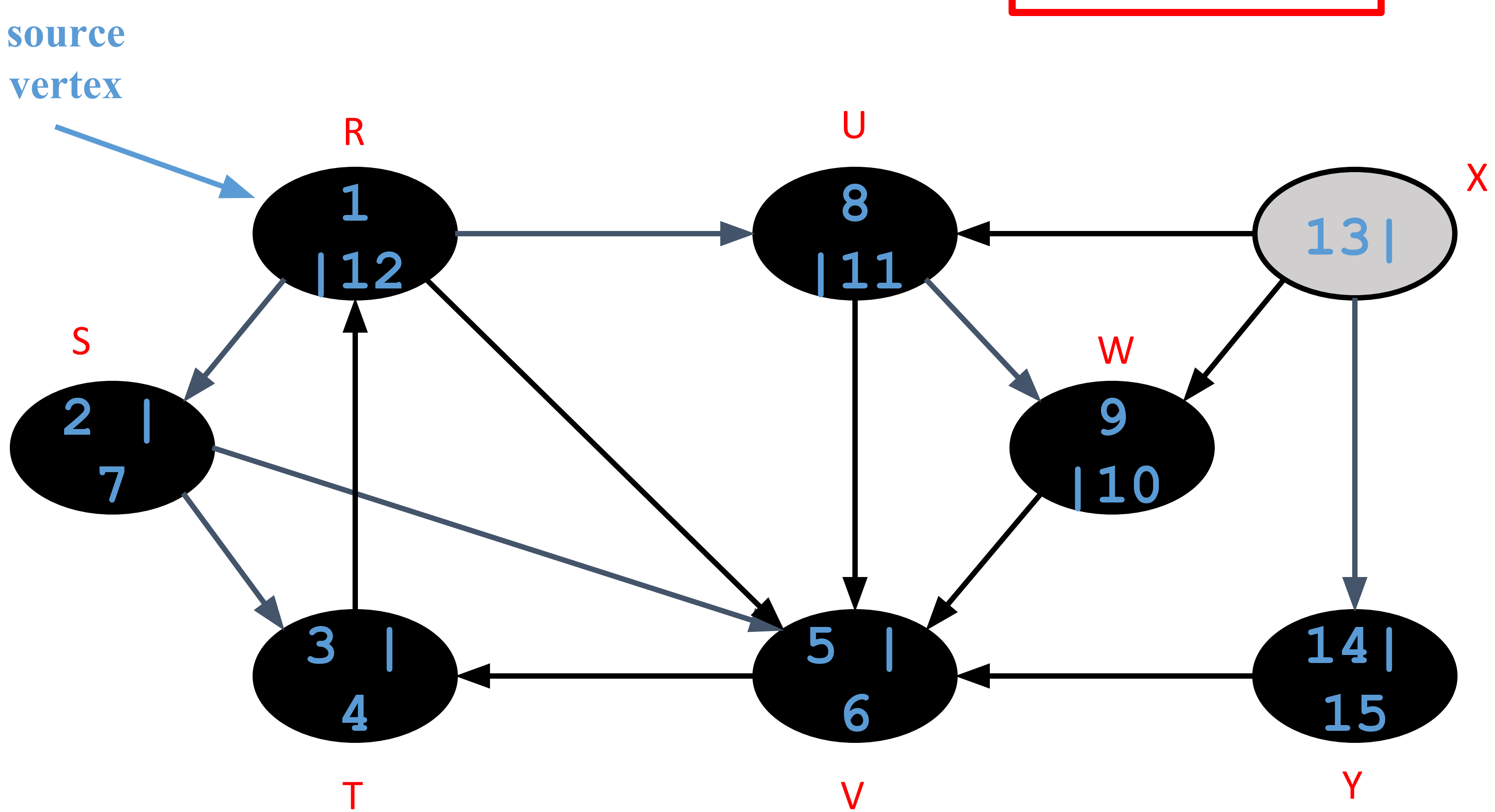
DFS Example

- $\pi[Y] = X$
- color[Y] = GREY
- time = 13 + 1 = 14
- d[Y] = 14



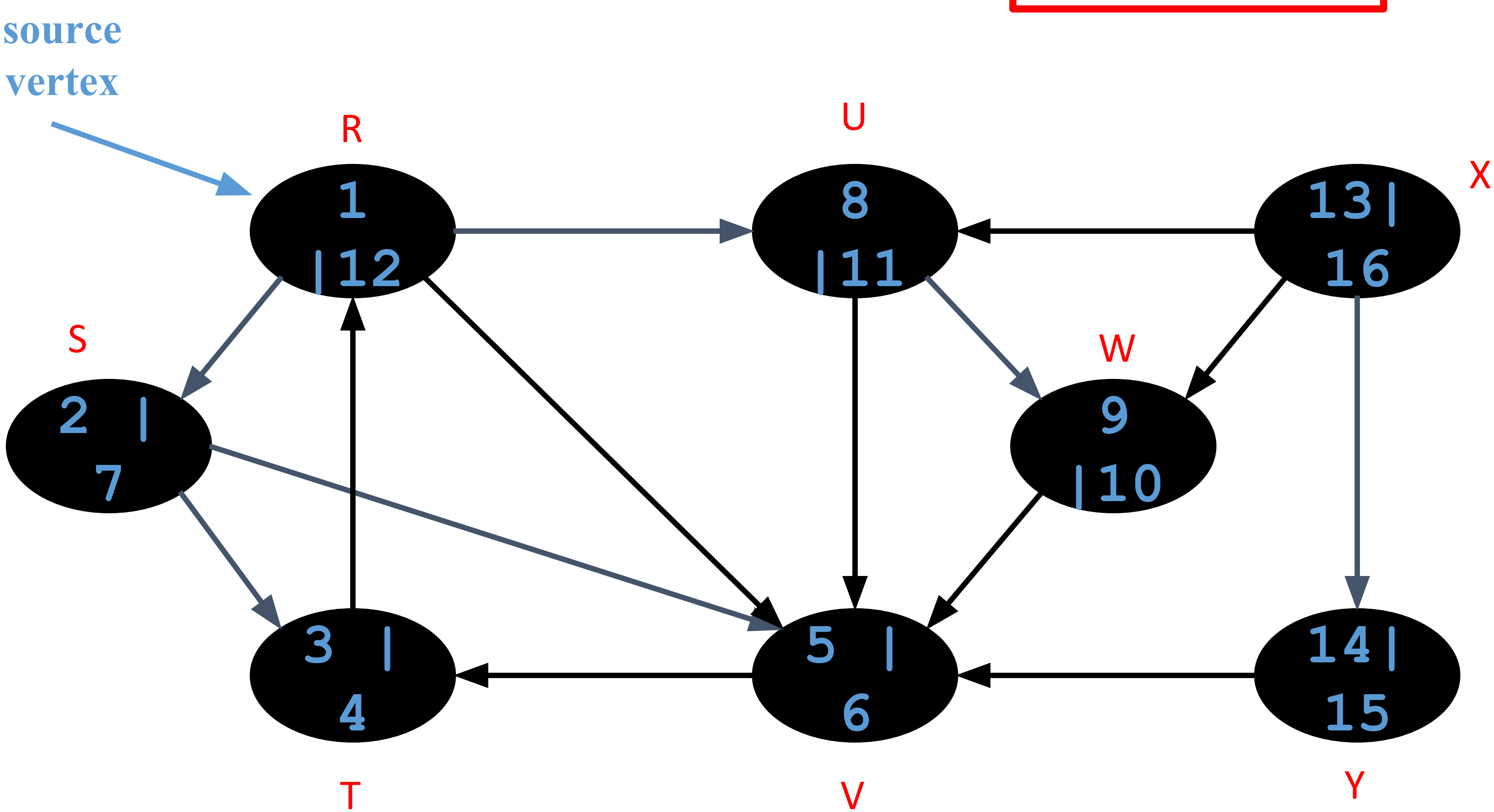
DFS Example

color[Y] = BLACK
time = 14 + 1 = 15
f[S] = 15



DFS Example

color[X] = BLACK
time = 15 + 1 = 16
f[S] = 16



DFS Iterative Algorithm

- DFS-iterative (G, x): *//Where G is graph and x is source vertex*
- let S be stack
- S.push(x) *//Inserting x in stack*
- mark x as *explored*.
- ***while*** (S is not empty):
  - v = S.top( ) *//Pop a vertex from stack to visit next. Here vertex v is visited.*
- S.pop( )
- *//Push all the neighbours of v in stack that are not explored*
- ***for*** all neighbours w of v in Graph G:

  if w is not explored :

  S.push(w)

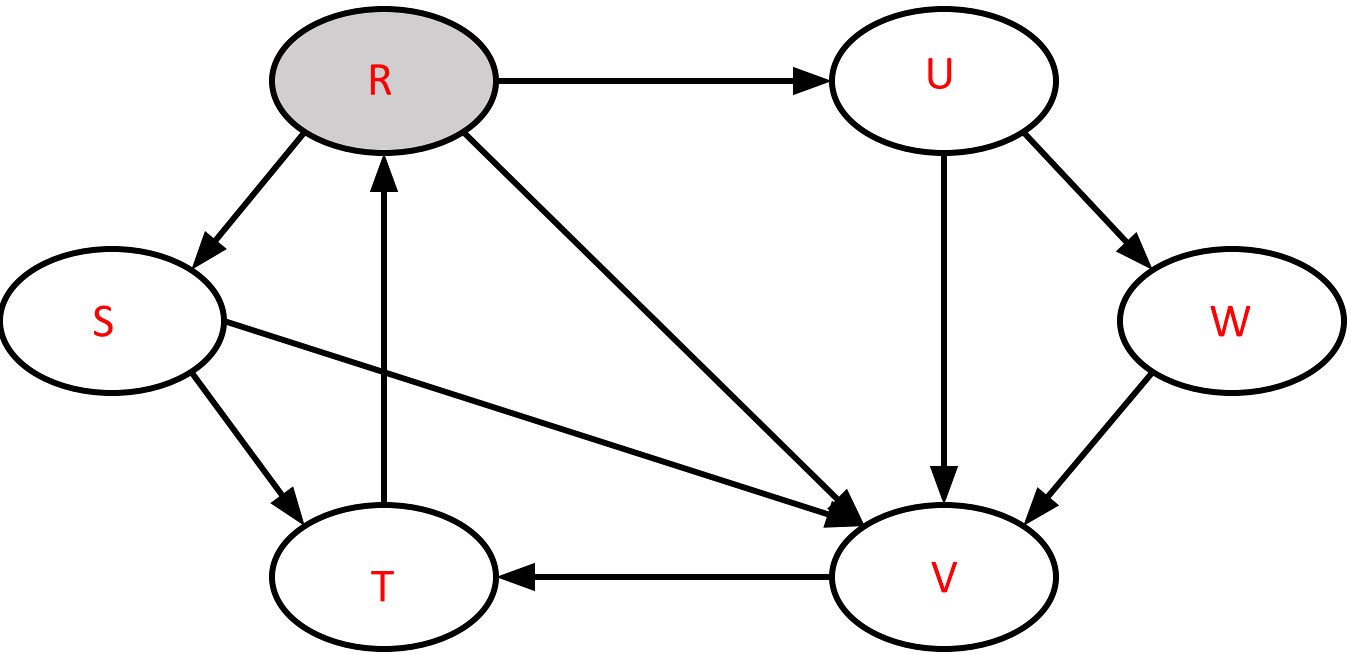
  mark w as explored

DFS Example

We will traverse the given graph using DFS.

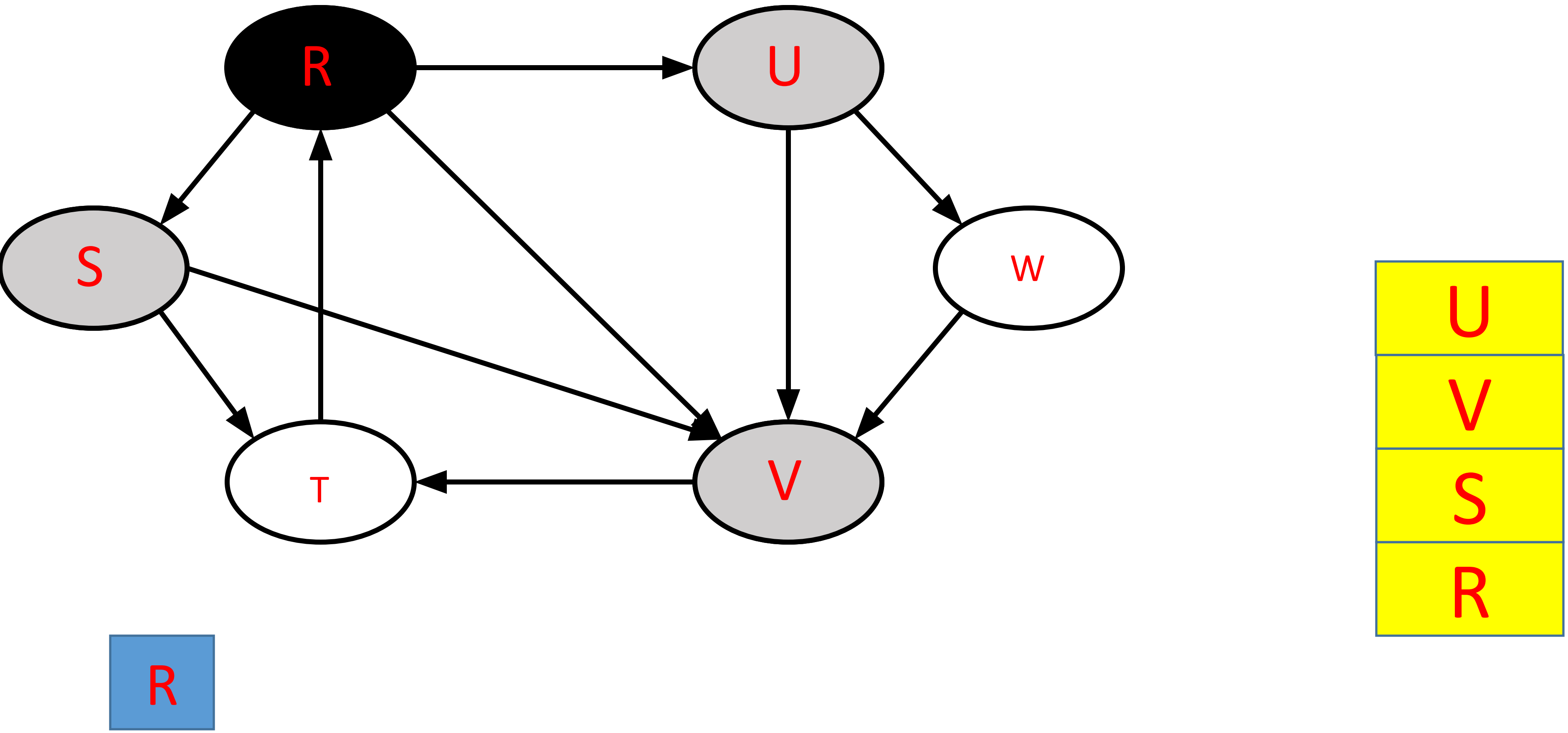
- Now POP node R from the stack, mark it as visited and push all the unexplored neighbours onto the stack.

- Let us start processing the graph from vertex R. Push the node R onto the stack.



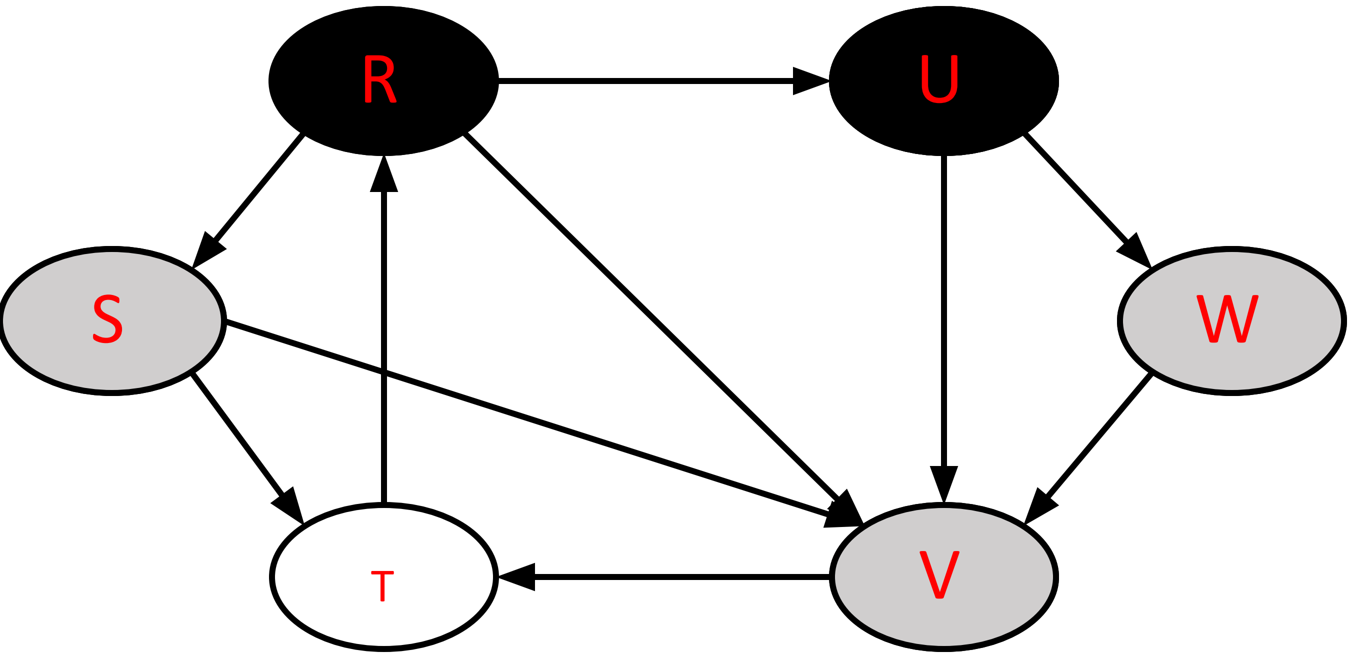
DFS Example

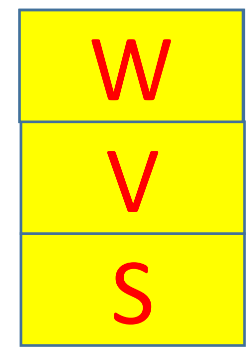
- Now POP node R from the stack, mark it as visited and push all the explored neighbours onto the stack. All the neighbours which are pushed onto the stack must be marked as gray.

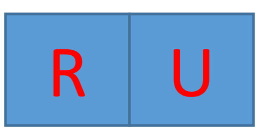


DFS Example

- Now POP node U from the stack, mark it as visited and push all the unexplored neighbours onto the stack. All the neighbours which are pushed onto the stack must be marked as gray.

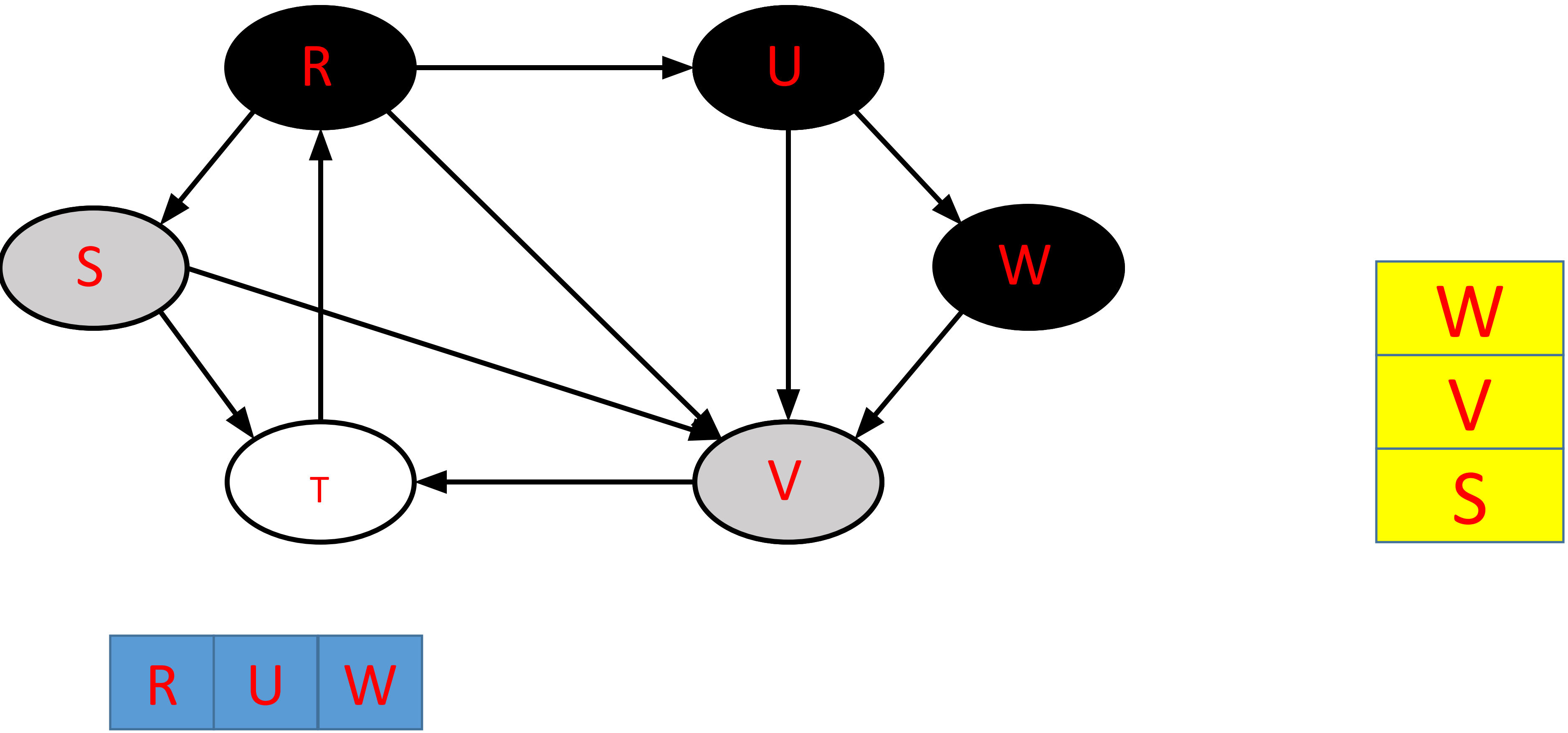






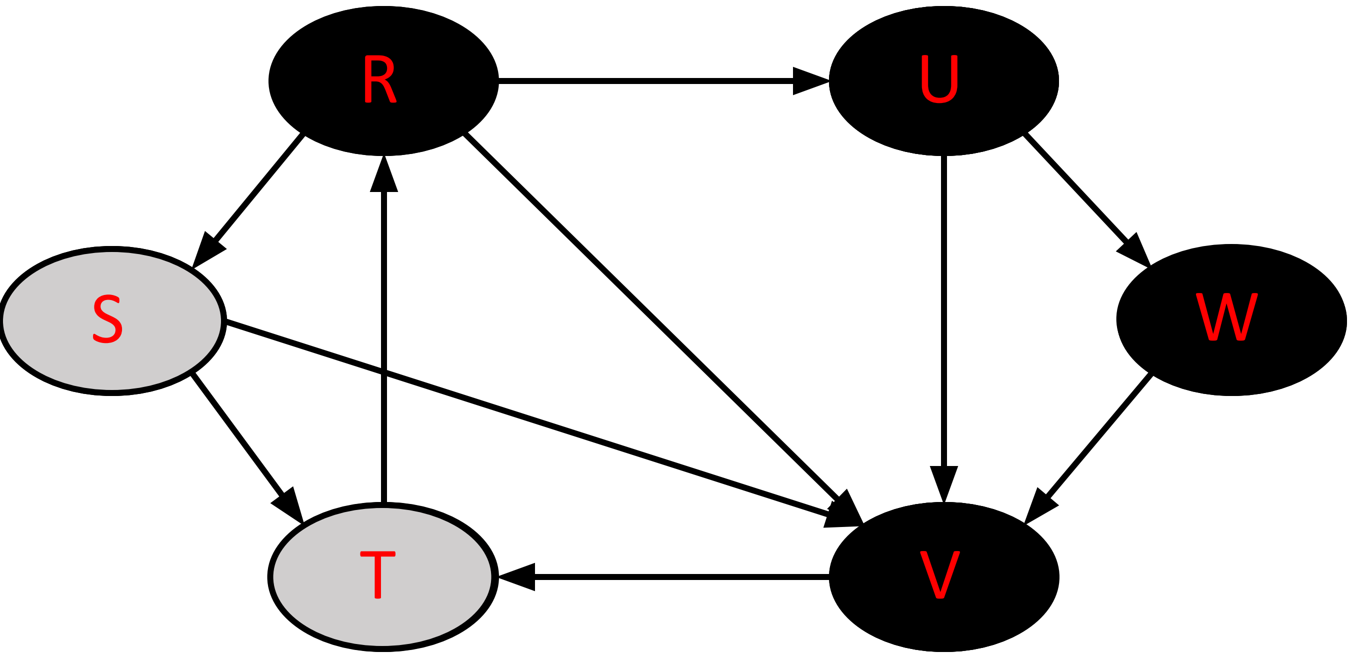
DFS Example

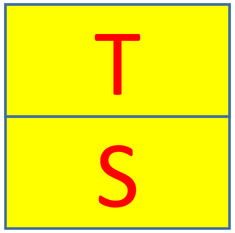
- Now POP node W from the stack, mark it as visited and push all the unexplored neighbours onto the stack. Here node W has no unexplored neighbours.

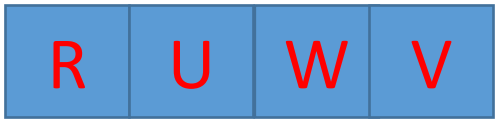


DFS Example

- Now POP node V from the stack, mark it as visited and push all the unexplored neighbours onto the stack. Here node V has one unexplored neighbour as node T.

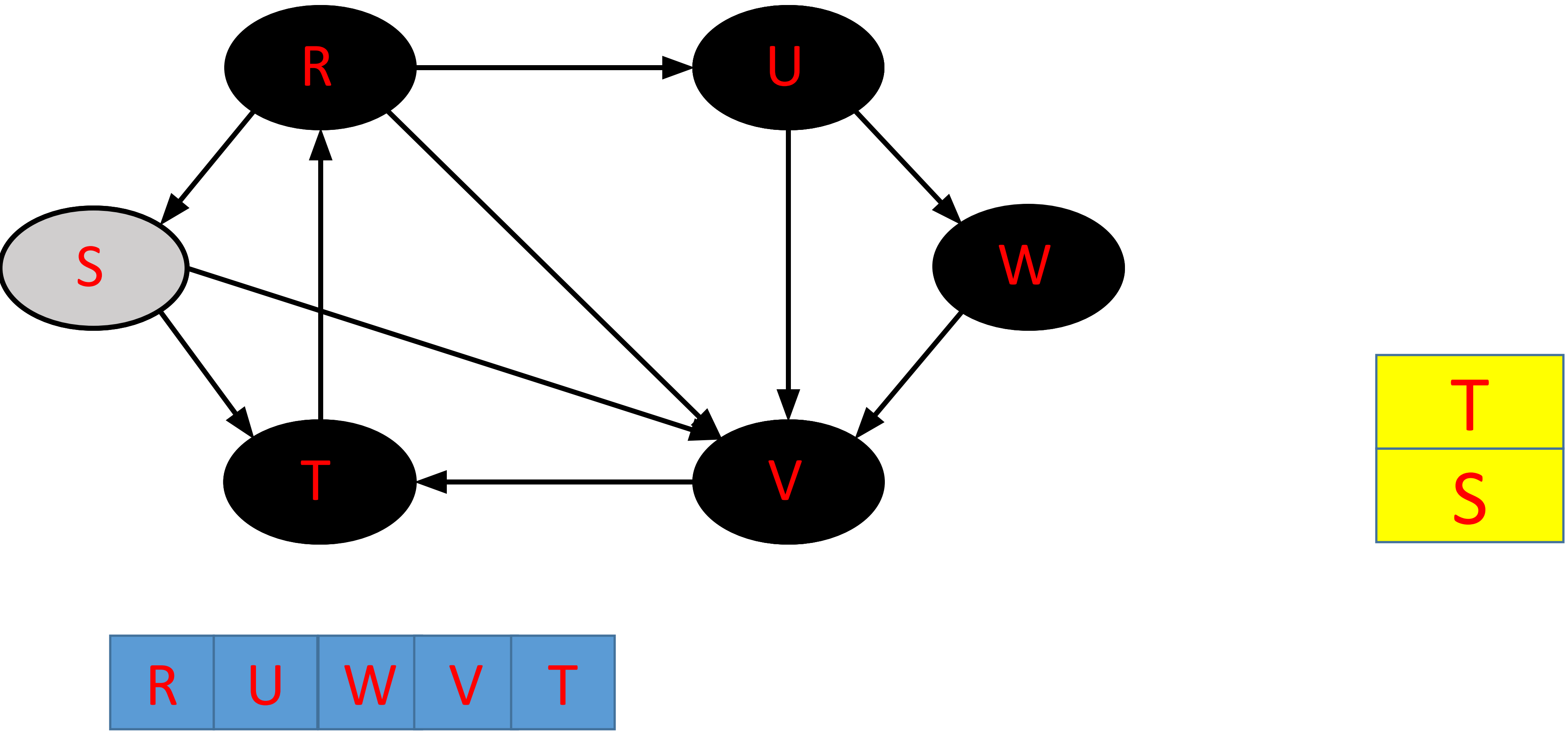






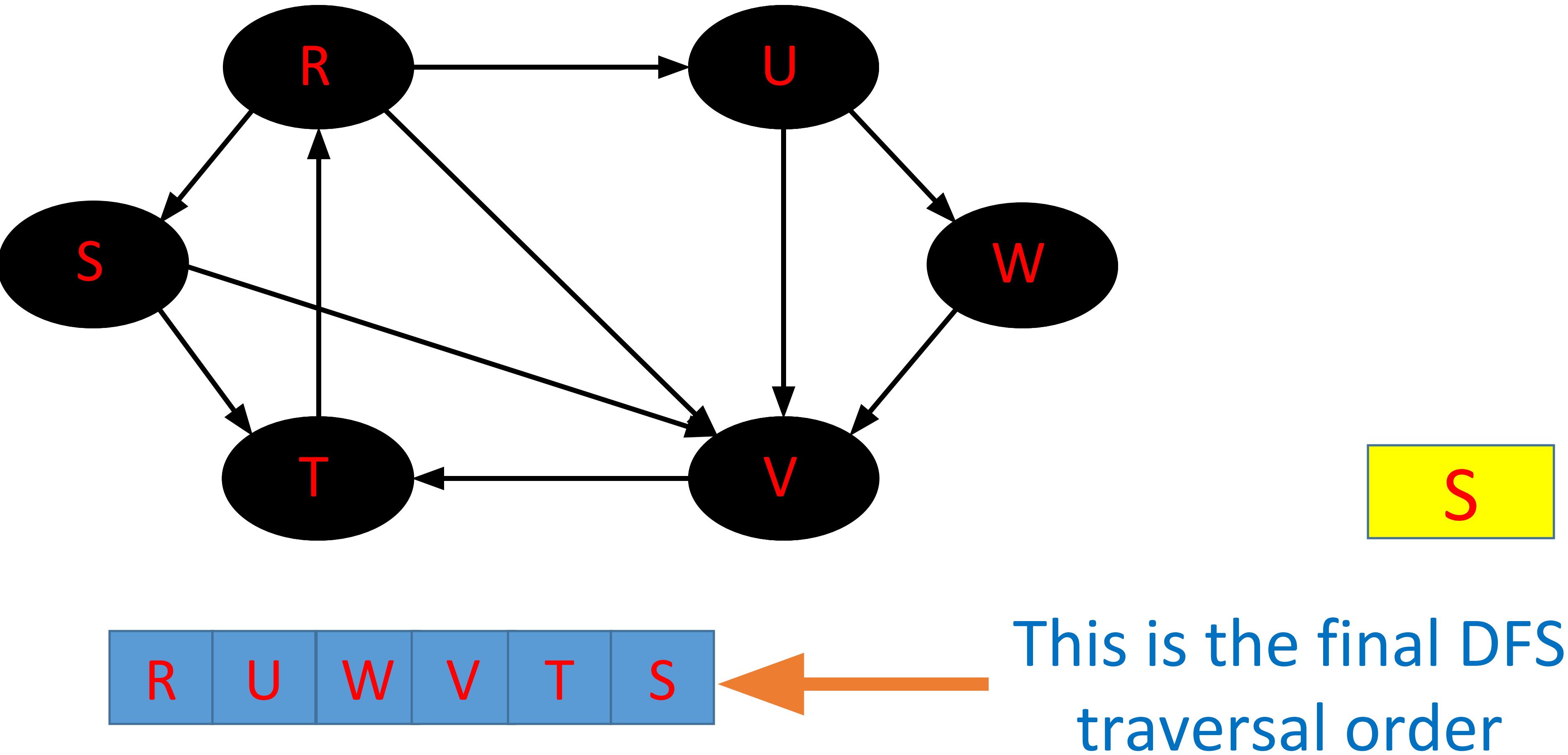
DFS Example

- Now POP node T from the stack, mark it as visited and push all the unexplored neighbours onto the stack. Here node T has no unexplored neighbours.



DFS Example

- Now POP node S from the stack, mark it as visited and push all the unexplored neighbours onto the stack. Here node S has no unvisited neighbours.



Applications of DFS

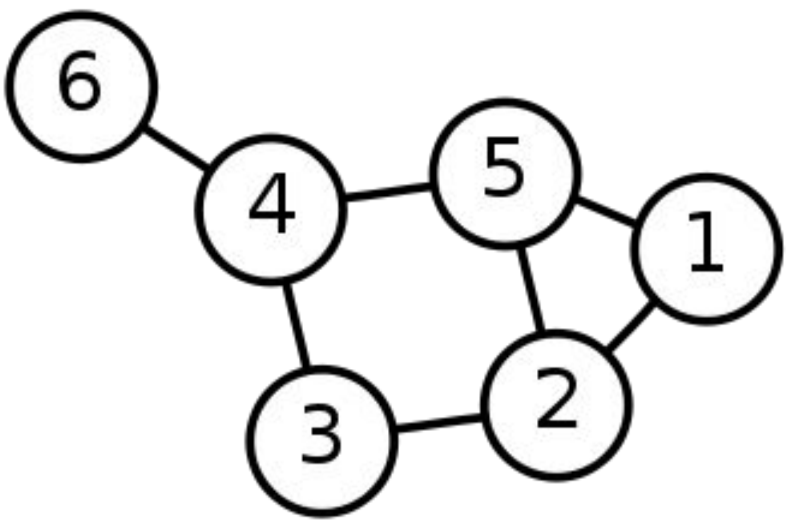
- **Trees: preorder traversal**

- **Graphs:**
  - **Connectivity**
  - **Biconnectivity – articulation points**
  - **Euler graphs**
  - **cycle detection in graphs**
  - **solving puzzles with only one solution, such as a maze or a sudoku puzzle**

Shortest Path Algorithms

Single-Source Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have **nonnegative weights**.

Input: Weighted graph $G=\{E,V\}$ and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

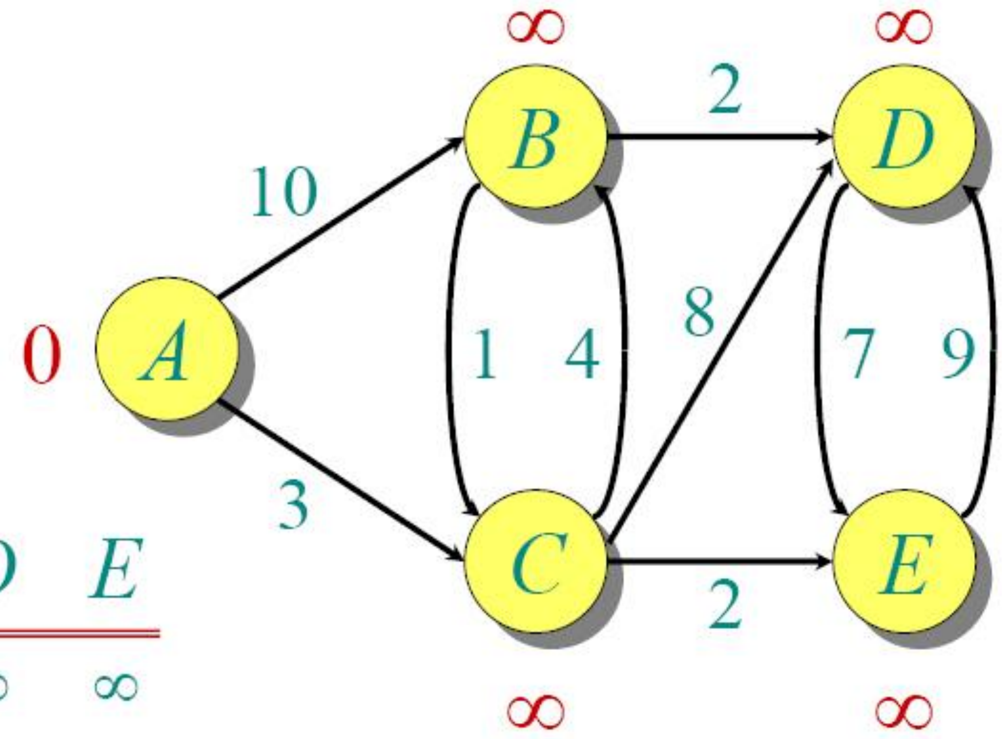
Dijkstra's algorithm - Pseudocode

```
dist[s] ←0                             (distance to source vertex is zero)
for  all v ∈ V–{s}
        do  dist[v] ←∞                 (set all other distances to infinity)
S←∅                                    (S, the set of visited vertices is initially empty)
Q←V                                    (Q, the queue initially contains all vertices)
while Q ≠∅                             (while the queue is not empty)
do   u ← mindistance(Q,dist)           (select the element of Q with the min. distance)
     S←S ∪ {u}                         (add u to list of visited vertices)
     for all v ∈ neighbors[u]
          do  if   dist[v] > dist[u] + w(u, v)        (if new shortest path found)
                   then      d[v] ←d[u] + w(u, v)     (set new value of shortest path)

return dist
```

Dijkstra Example

Initialize:

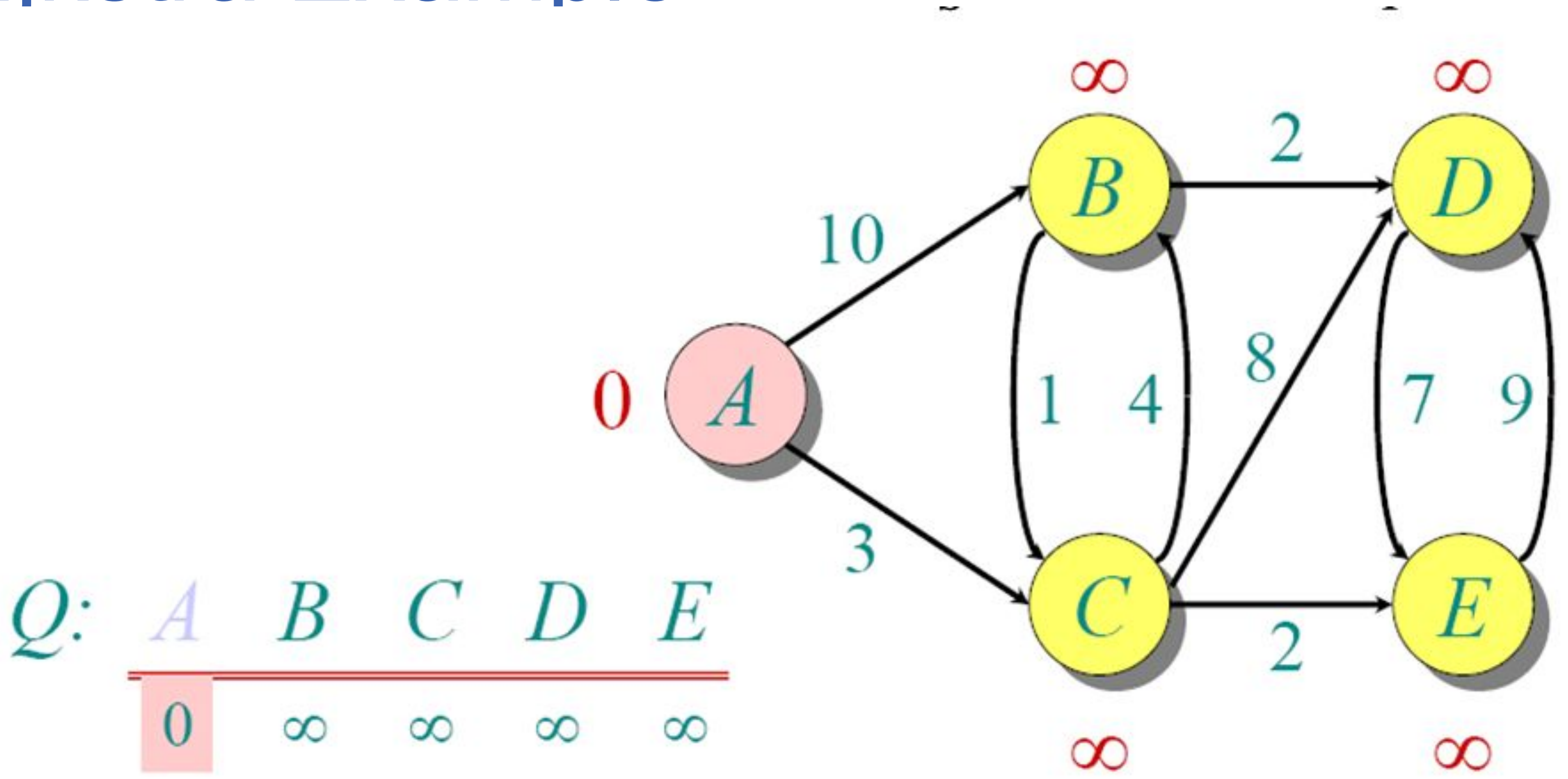


Q:

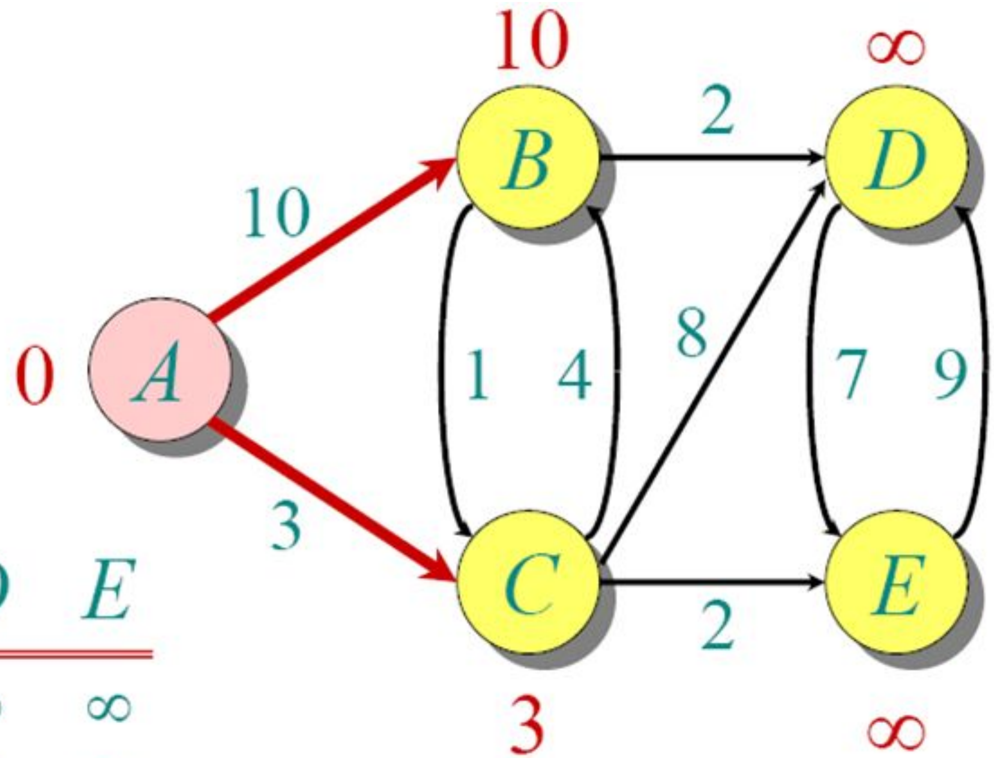
A	B	C	D	E
0	∞	∞	∞	∞

S: {}

Dijkstra Example



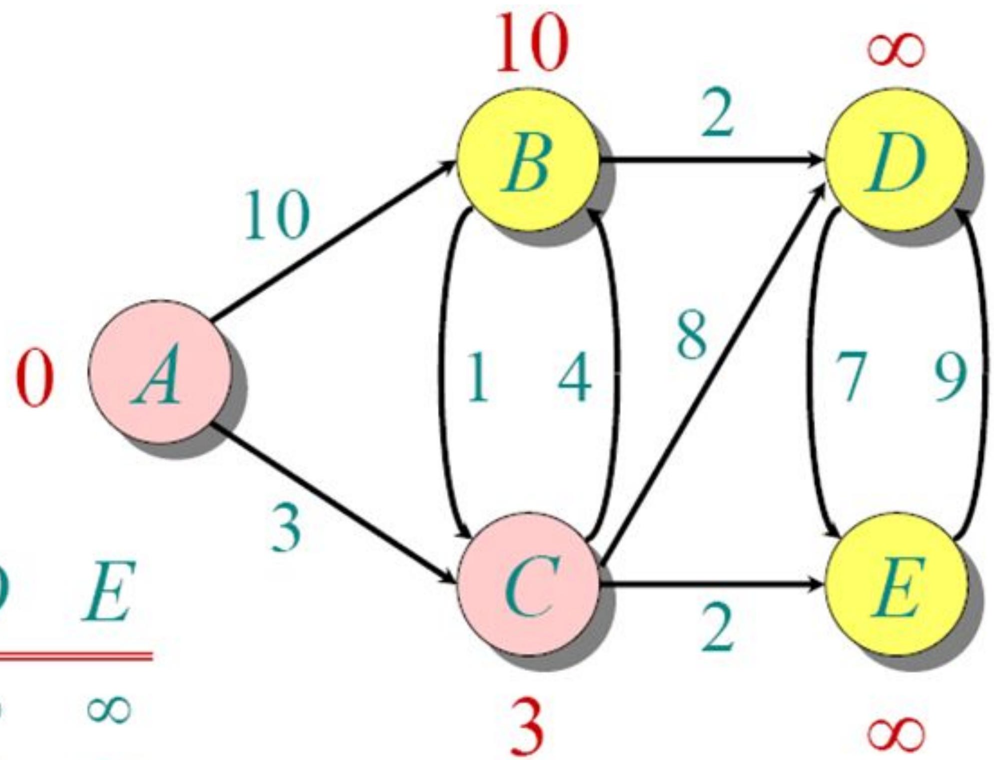
Dijkstra Example



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

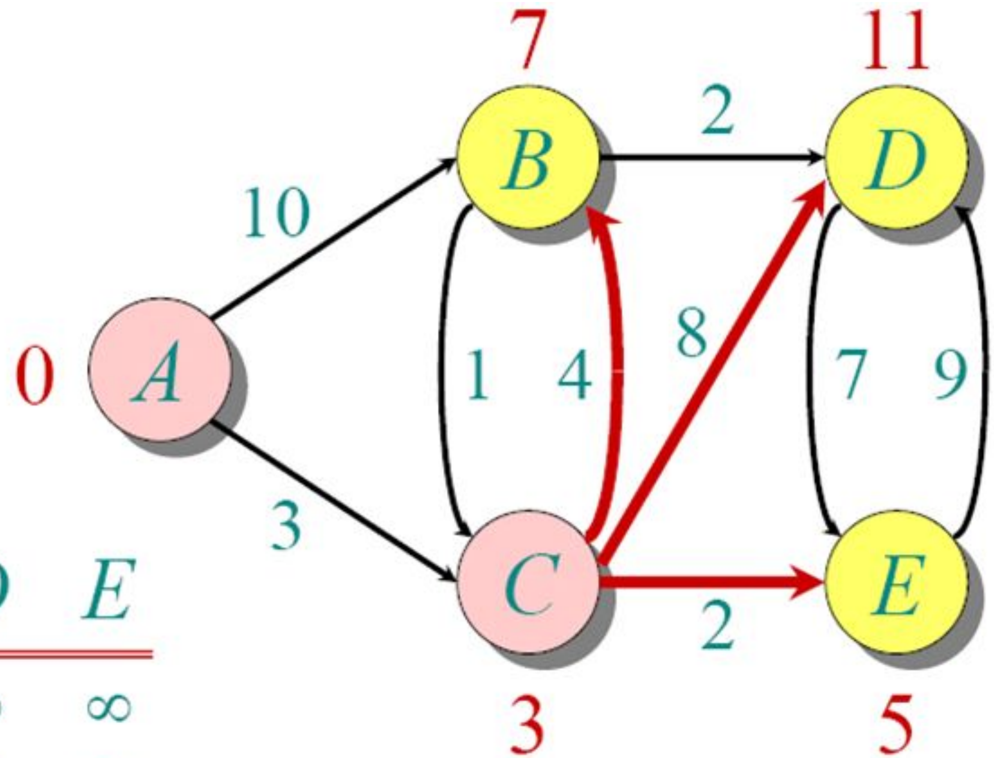
Dijkstra Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

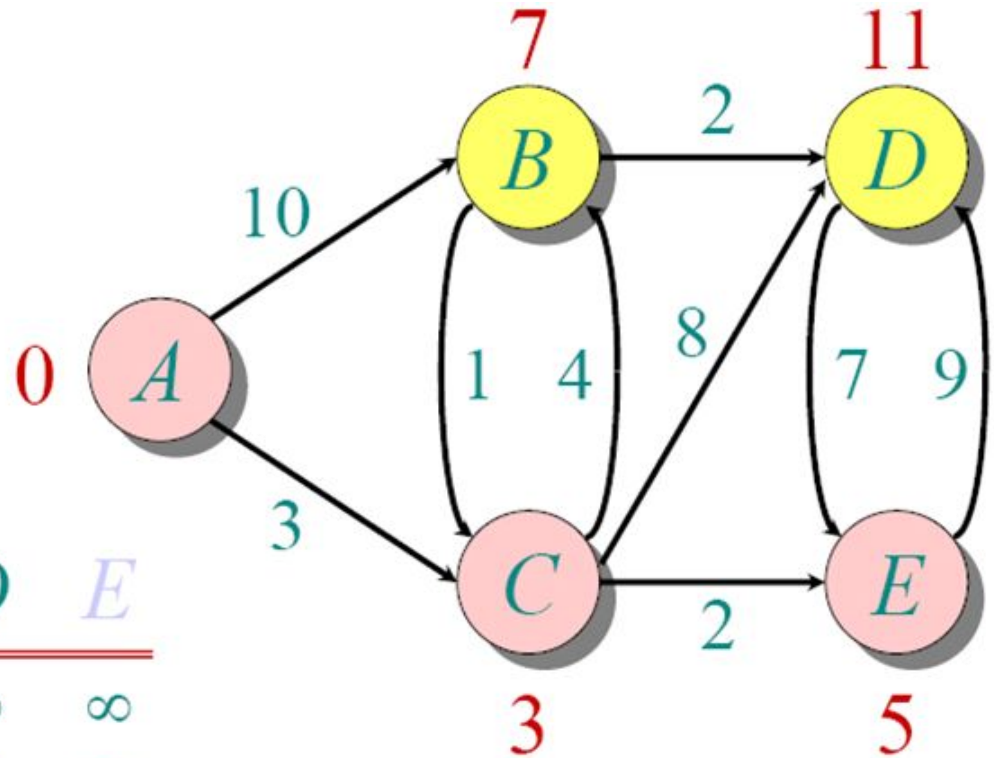
Dijkstra Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C }

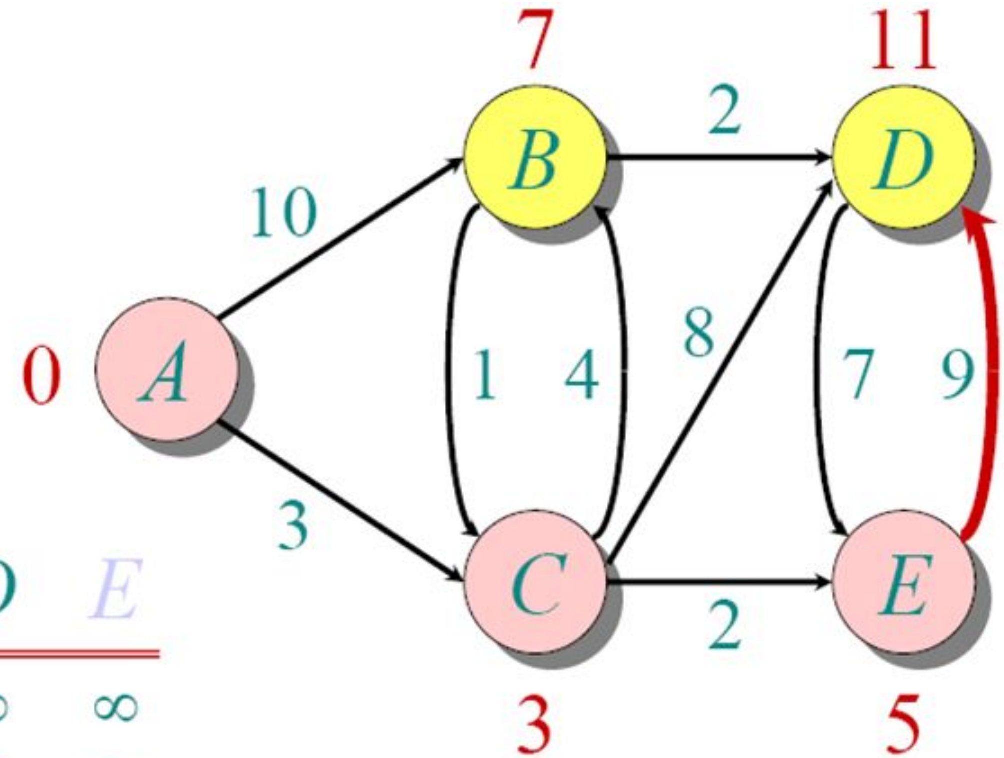
Dijkstra Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: $\{ A, C, E \}$

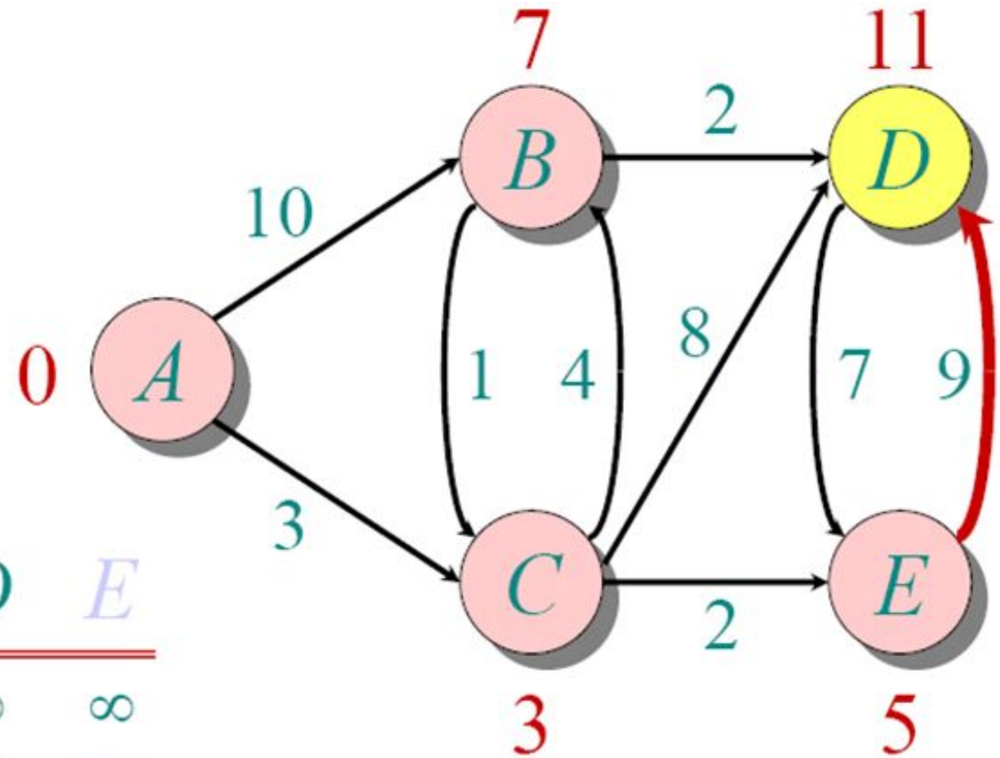
Dijkstra Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

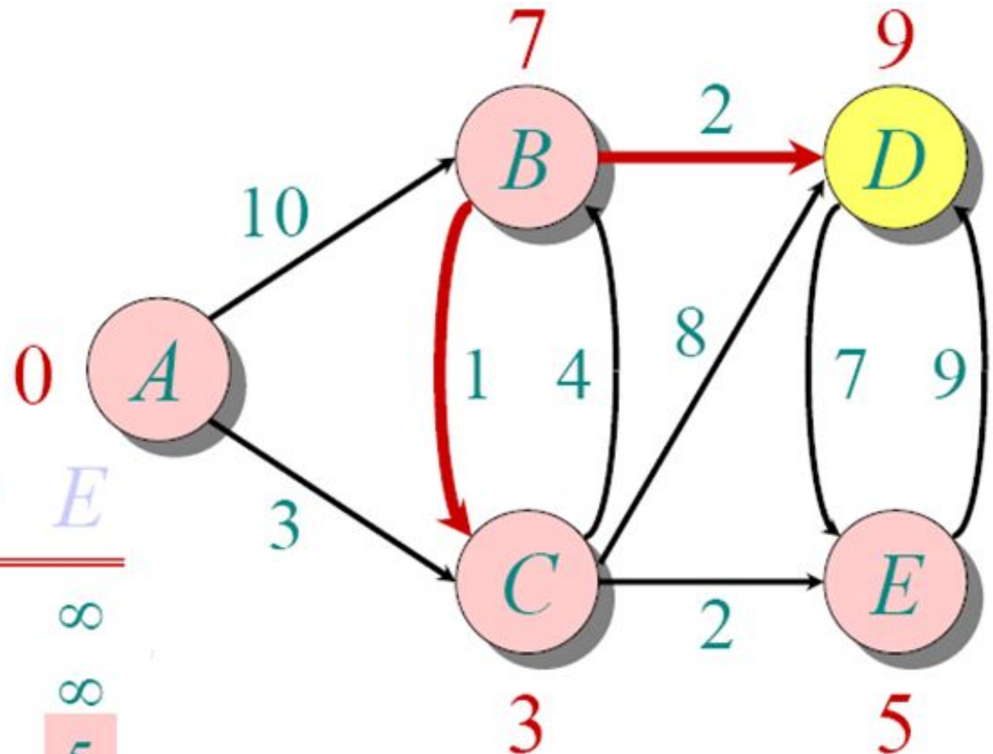
Dijkstra Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

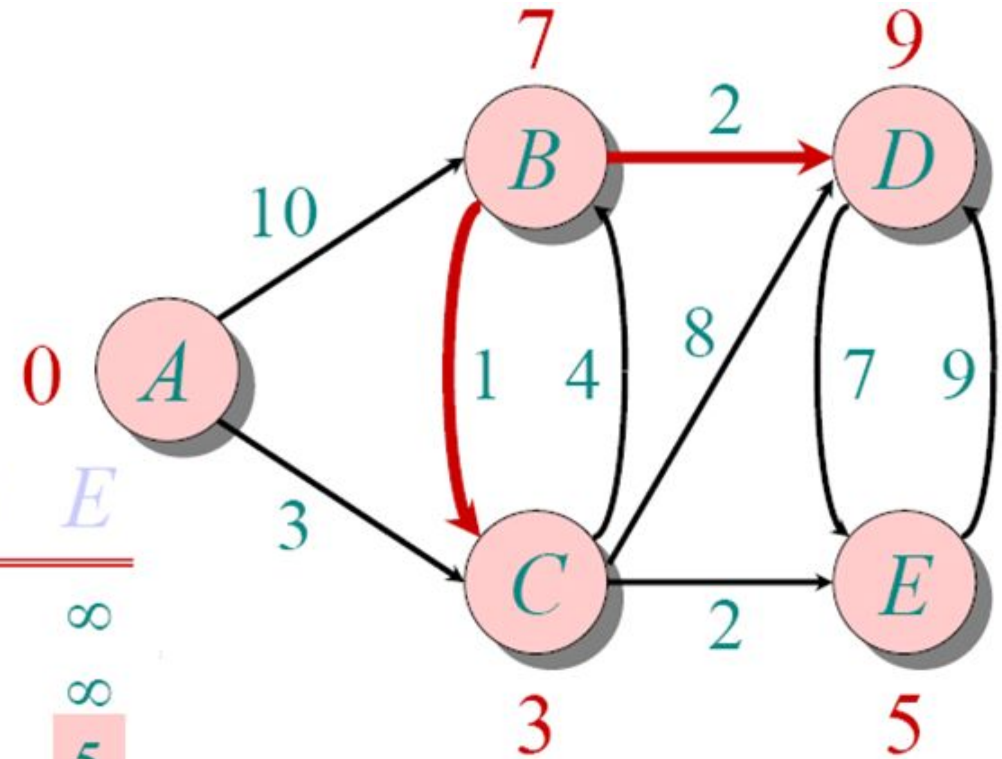
Dijkstra Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Dijkstra Example



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				$\boxed{9}$	

S: { A, C, E, B, D }

Dijkstra-Complexity

- Dijkstra's algorithm is structurally identical to BFS. However, it is slower because the priority queue is computationally more demanding than the constant-time pop and push of BFS.
- Runtime analysis using a binary heap:
- BuildPriorityQueue takes O(|V|).
- ExtractMin and Insert are executed |V| times each – once for each node, thus it takes O(|V|log|V|).
- In the worst case, DecreaseKey can be executed for every edge, thus |E| times, giving O(|E|log|V|). Overall we get O((|V|+|E|)log|V|)

Running Time Analysis of Dijkstra's Algorithm

- Look at different Q implementation, as did for Prim's algorithm

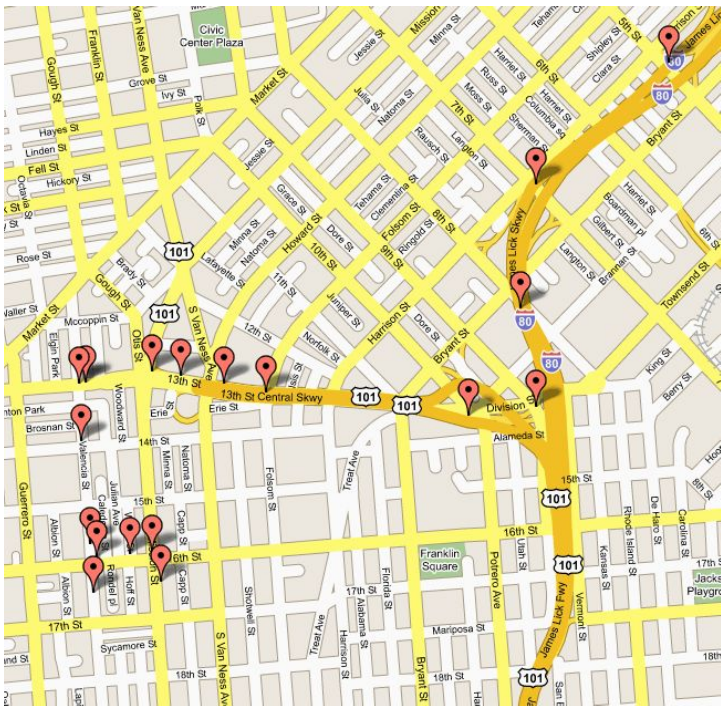
- Linear
  Unsorted Array: $O(V^2+E)$
- Binary Heap: $O(V\lg V+E\lg V) = O(E\lg V)$
- Fibonacci heap: $O(V\lg V+E)$

Dijkstra's Algorithm - Why use it?

- As mentioned, Dijkstra's algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex *u* to every vertex using Dijkstra's as it is to calculate the shortest path to some particular vertex *v*.
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra's algorithm.

Applications of Dijkstra's Algorithm

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.