Graphs

CSC-114 Data Structure and Algorithms

Slides credit: Ms. Saba Anwar, CIIT Lahore

Outline

- Non-Linear Data Structures
  - Graphs
    - Traversal
      - BFS
      - DFS
    - Topological Sort
    - Shortest Path
      - Dijkstra's Algorithm

Graph Traversal

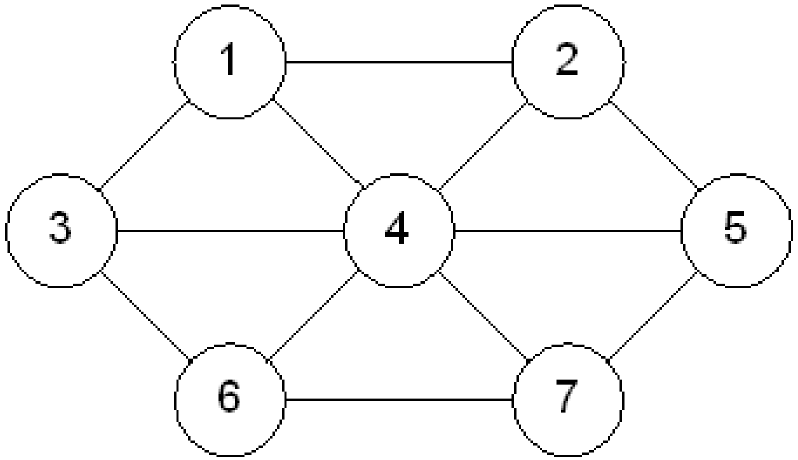
- Visiting all vertices one bye one
- Just like we did in tree but remember tree has no cycles, so its bit different in graph scenario
  - We need to keep track of nodes that have been visited/discovered
- And if graph is disconnected
  - We cannot traverse each vertex from just a single vertex
- Two ways:
  - Breadth First Traversal
    - Visit nodes that are closest to starting node before other nodes
  - Depth First Traversal
    - Visit recursively one side before going back to other side

Breadth First Search

Given a graph G and a start vertex, breadth first search systematically explores edges of G to **discover** vertices that are reachable from start vertex.

It is named as breadth first because as it traverse the vertices at distance K before the vertices at distance k+1, where k represents number of edges.

BFS order of vertices From 1: 1 , 2, 3, 4, 5, 6, 7



Vertices at edge distance of 1are visited before vertices at edge distance of 2.

So, BFS discovers each vertex by exploring minimum possible number of edges.

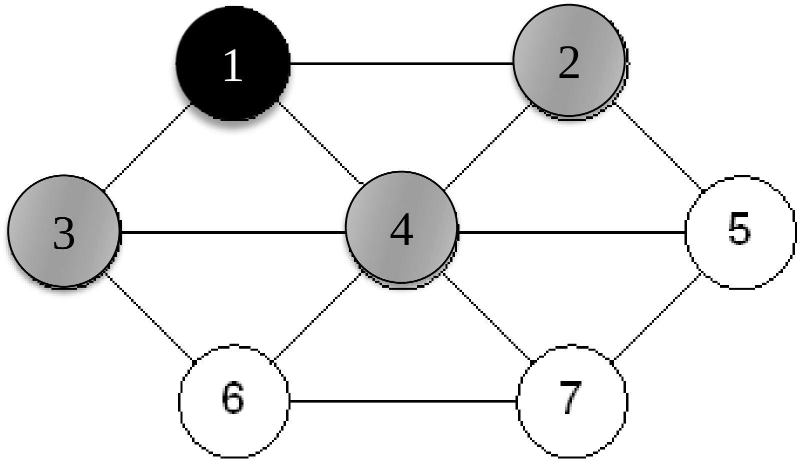
Breadth First Search

Progress can be tracked by maintaining a state for each vertex. Vertices can be in three distinguished states:

- Not discovered
- Partially discovered: a vertex is discovered first time but not fully explored.
- Finished/Fully explored: vertex that has been fully explored, all its adjacent nodes have been discovered.

Tri-Coloring

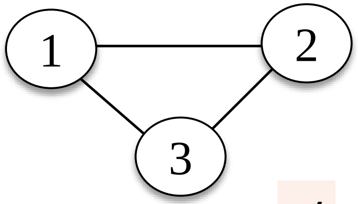
- Vertices are given colors according to state
  - White : un discovered
  - Grey: partially discovered
  - Black: finish/fully explored



Breadth First Search

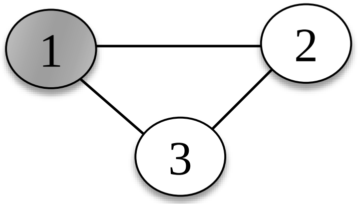
BFS: from 1

All vertices are undiscovered at start.



∅

Mark start vertex as discovered

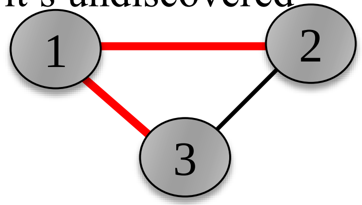


1

Go to its adjacency list→ 1 edge distance

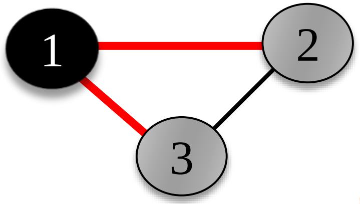
first node is 2 and its un-discovered

2nd node is 3 and it’s undiscovered



2 | 3

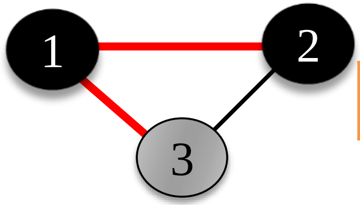
1’s adjacency list is fully explored now.



Go to 2’s adjacency list.→ 2 edge distance

2 | 3

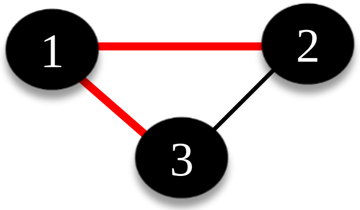
Its vertices are already discovered



3

Go to 3’s adjacency list.→ 2 edge distance

Its vertices are already discovered



Graph is fully explored

∅

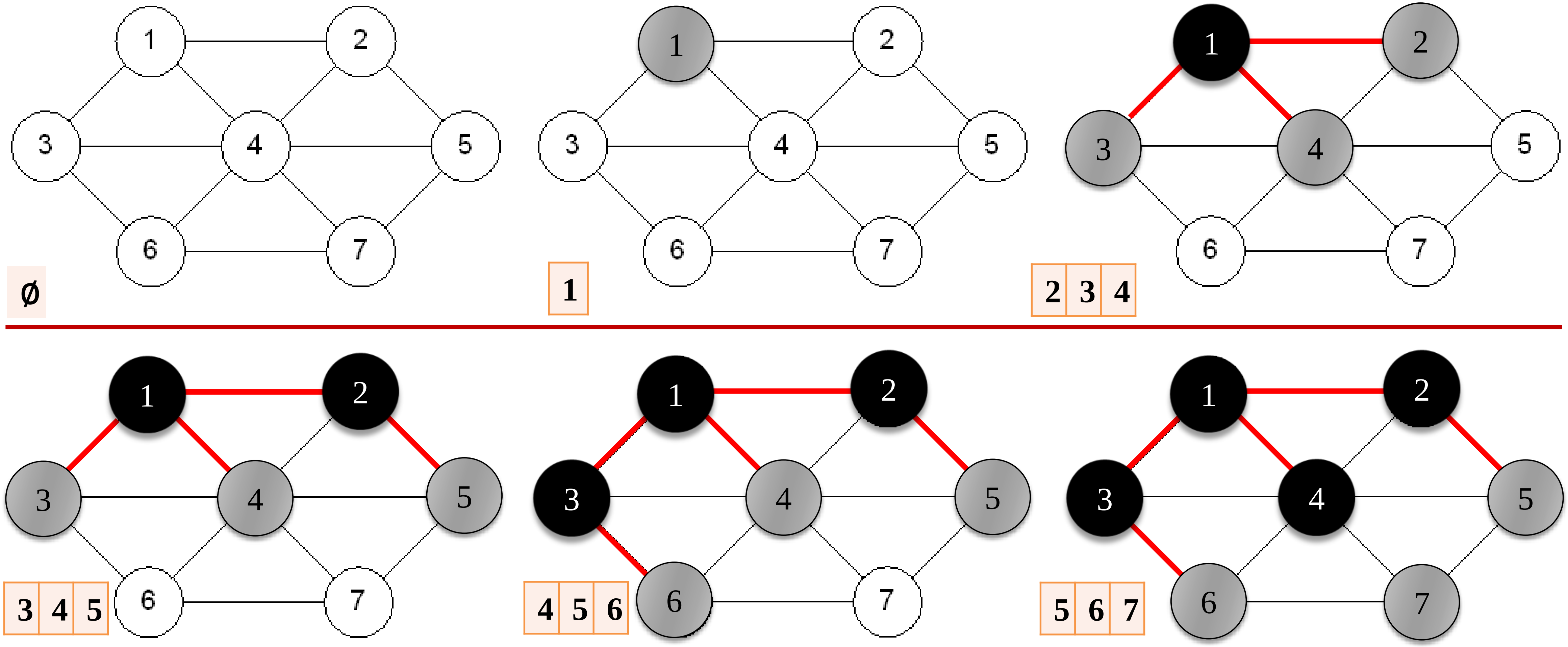
Breadth First Search (BFS)

Algorithm: BFS(G, start)

```
Input: Graph and start vertex of graph.
Output: list of vertices reachable from start in order of their discovery time
Steps:
Q = new Queue()
For each vertex v in G
  color[v]=white
color[start]=grey
Q.enqueue(start)
While Q is not empty
     u= Q.dequeue()
     print(u)
     For each vertex v adjacent to u
        if color[v] is white          //undiscovered
            color[v]=grey             //discovered
            Q.enqueue(v)
        End if
   End For
   color[u]=black                     //fully explored or finished
 End While
```

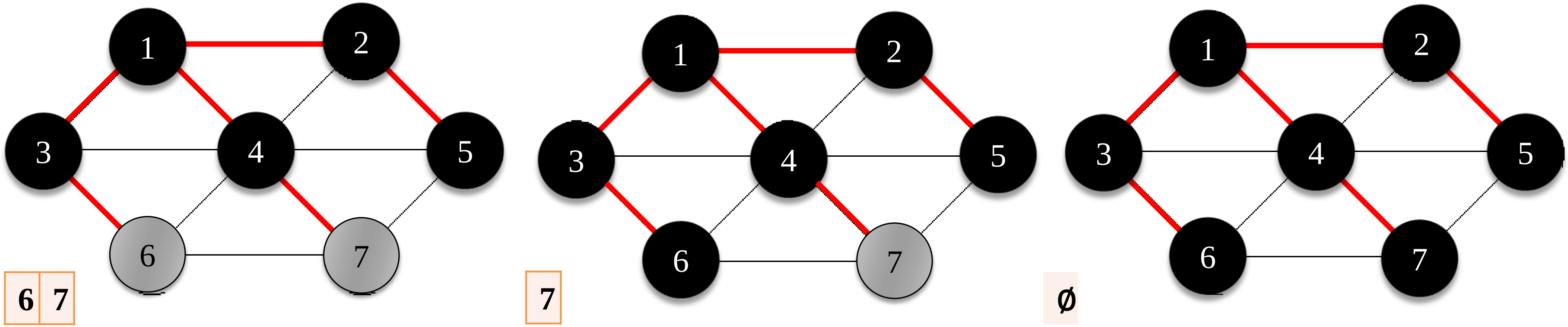
Breadth First Search

BFS: start vertex=1



Breadth First Search

BFS: start vertex=1



BFS Order: 1 2 3 4 5 6 7

Breadth First Search (Alternate Approach)

Rather than using tri-coloring, just maintain two states.

- If a vertex is undiscovered
  - It will be marked as visited=false
- If a vertex is pushed to queue→ it is discovered
  - Mark it visited = true
- If a vertex is popped off→ it is fully explored
  - No marking

Algorithm: BFS(G, start)

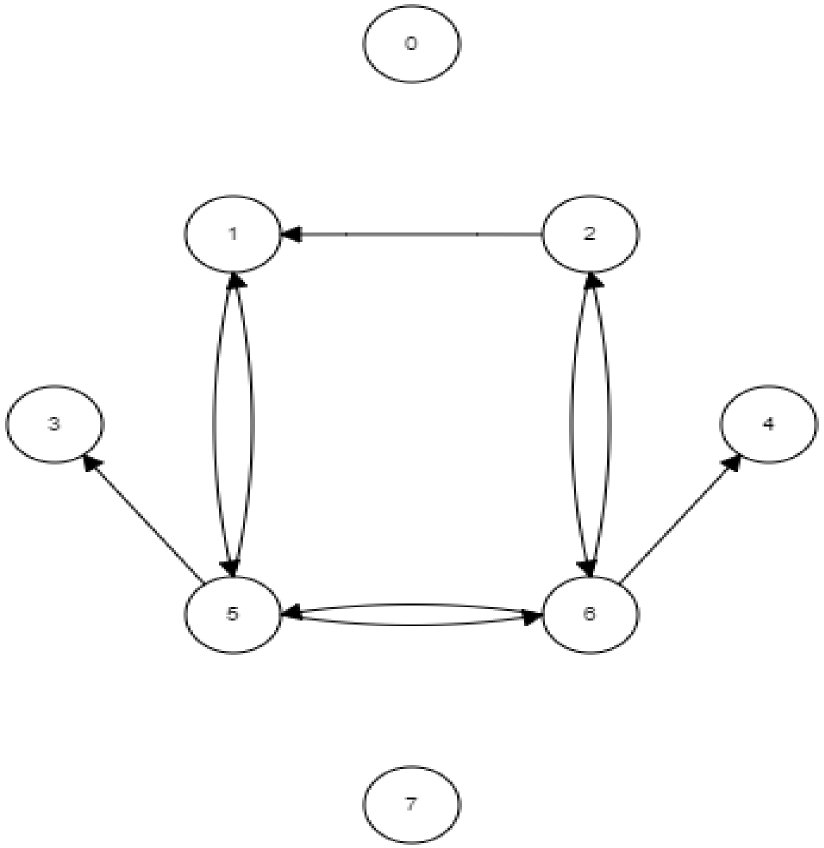
```
Input: Graph and start vertex of graph.
Output: list of vertices reachable from start
Steps:
  Q = new Queue()
  For each vertex v in G
    visited[v]=false
  visited[start]=true
    Q.enqueue(start)
  While Q is not empty
        u= Q.dequeue()
        print(u)
        For each node v adjacent to u
           if !(visited[v])
                visited[v]=true
                Q.enqueue(v)
            End if
      End For
  End While
```

Breadth First Search

Start vertex=1

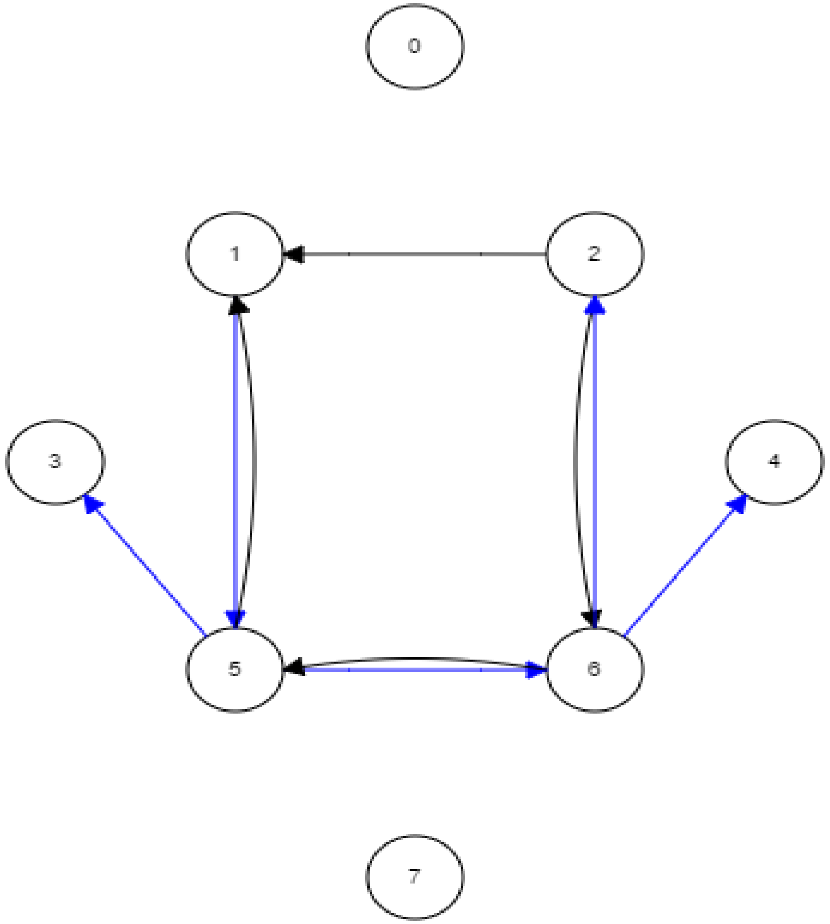
Visited

0	f
1	f
2	f
3	f
4	f
5	f
6	f
7	f



Visited

0	f
1	f
2	T
3	T
4	T
5	T
6	T
7	f

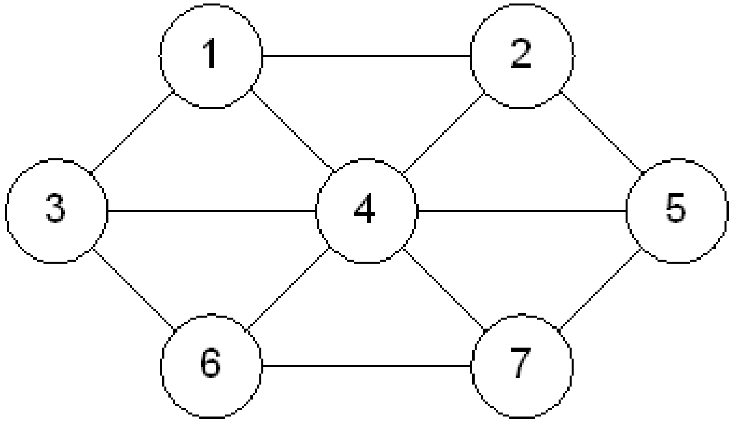


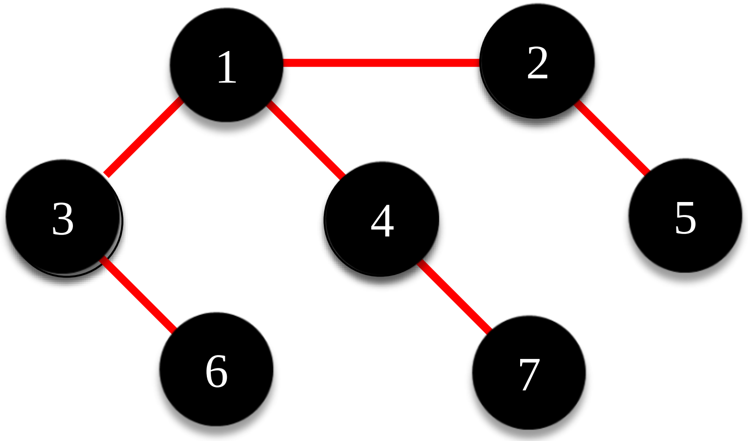
Order: 1, 5, 3, 6, 2, 4

Breadth First Search

BFS Tree

If graph is connected, by discarding the edges that were not explored during BFS, we can get a BFS tree of graph





BFS-Shortest Paths

Shortest Path in Non-Weighted Graph:

- Let say we want to find shortest paths to each vertex from source vertex?
- How BFS can help?

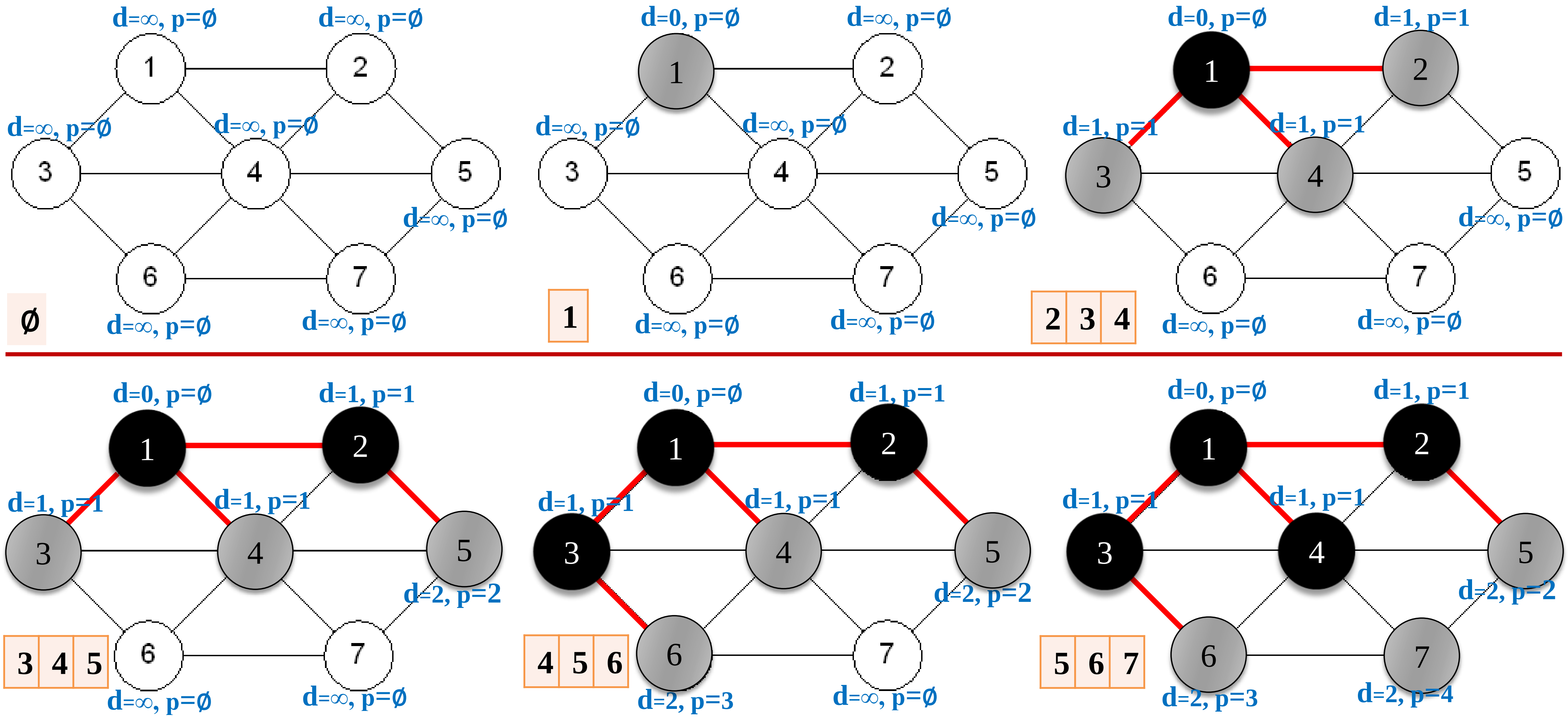
- Every edge is considered as having unit weight, means edges are equal in terms of cost.
- BFS discovers each vertex by exploring minimum possible number of edges.
  - That is shortest path
  - Path length= sum of edges

What information needs to be maintained?

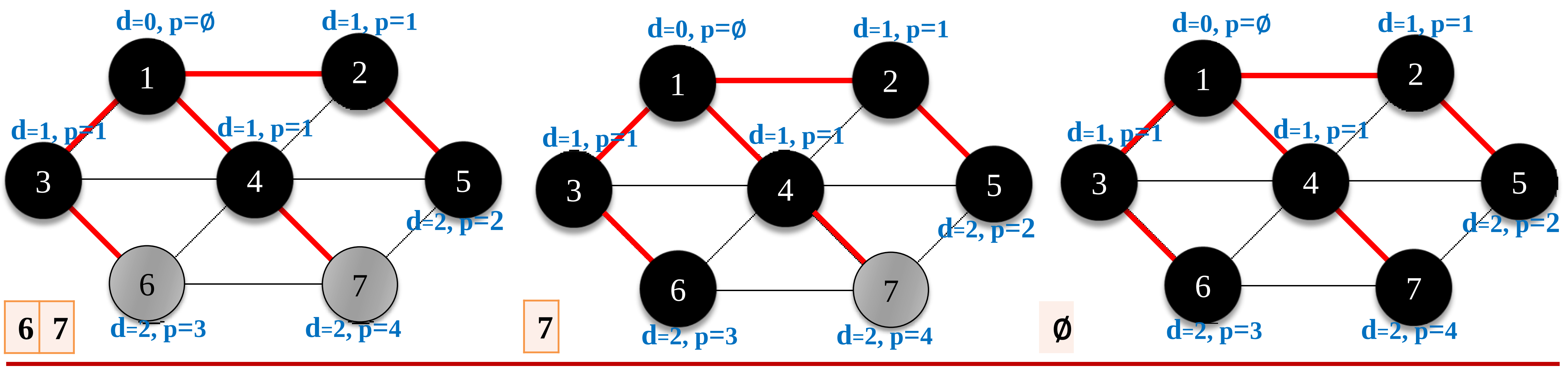
- Record distance→ Path Length

- Record parent/prev vertex of discovered vertex→ It is the vertex from which you discovered the current vertex.
  - To track the path from start vertex to destination vertex

BFS Shortest Paths



BFS Shortest Paths



Back track paths from destination to start:

- 2-1
- 3-1
- 4-1
- 5-2-1
- 6-3-1
- 7-4-1

Shortest Path(Non-Weighted Graph)

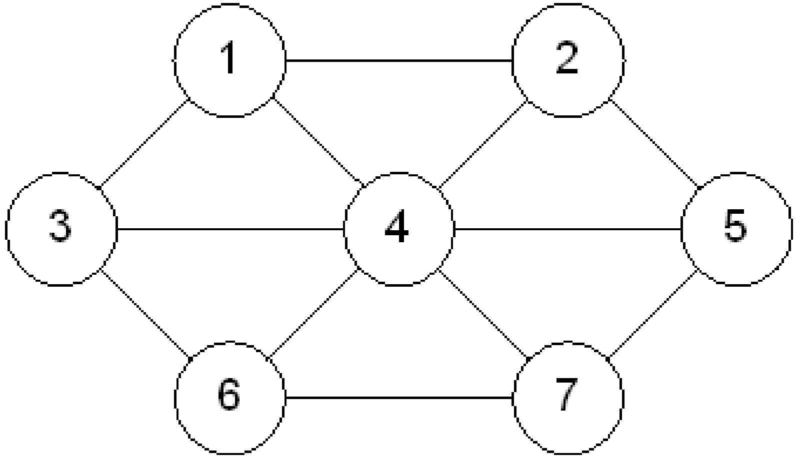
Algorithm: BFS(G, start)

```
Input: Graph and start vertex of graph.
Output: list of vertices reachable from start along with their paths and distances
Steps:
Q = new Queue()
For each vertex v in G
  d[v]=infinity
  p[v]=null
End For
d[start]=0
Q.enqueue(start)
While Q is not empty
     u= Q.dequeue()
     For each node v adjacent to u
        if d[v] is infinity     //undiscovered
             d[v]=d[u]+1
             p[v]=u
             Q.enqueue(v)
         End if
   End For
 End While
```

Depth First Search

From start vertex traverse the vertices at one path in depth till last reachable vertex before traversing other paths. Then go back to last visited node to explore other paths recursively un till all reachable vertices have been traversed.

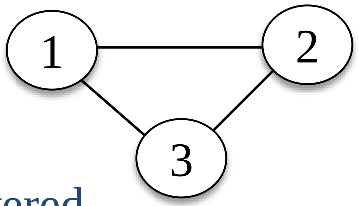
DFS order of nodes from 1: 1 2 4 3 6 7 5



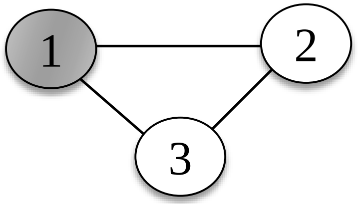
DFS as it name suggest discover a vertex in depth. When it first discovers a vertex, it will go to its adjacent vertices [adjacency list] and before discovering all its adjacent vertices at once (like BFS does) it will just discover first (undiscovered) vertex, and then will go to this vertex's adjacent vertices. And process goes one.

Depth First Search

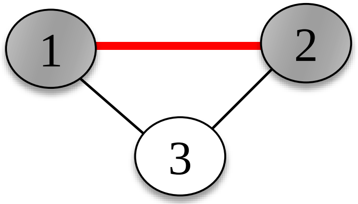
DFS: from 1



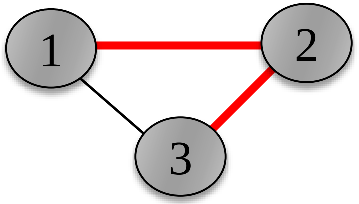
Mark start vertex as discovered



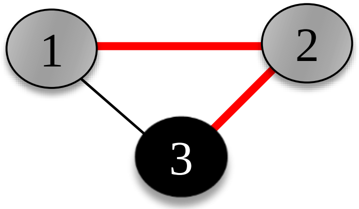
Go to its adjacency list, first node is 2 and its un-discovered



Go to adjacency list of 2, 1 is already discovered, discover 3



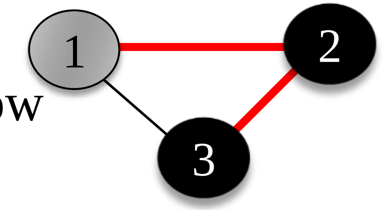
3's adjacency list is fully explored now.



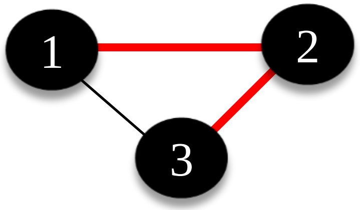
We discovered 3 from 2→parent

Go back to 2

Its fully explored now



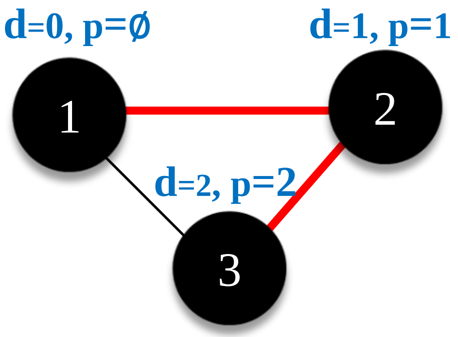
- Repeat process of going back, till start vertex



Depth First Search

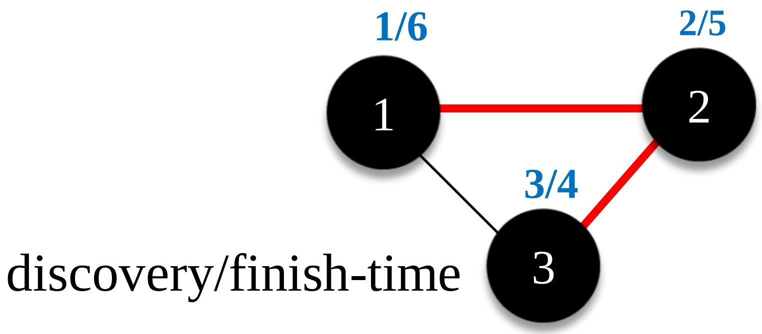
Difference from BFS:

- 3 is discovered from 2 and not 1
- Before discovering all adjacent vertices of 1, DFS has went to adjacent vertices of 2



Two different time stamps of visit?

- Discovery time: Vertex becomes grey
- Explore/Finish time: Vertex becomes black



Depth First Search

Algorithm: DFS(G, start)

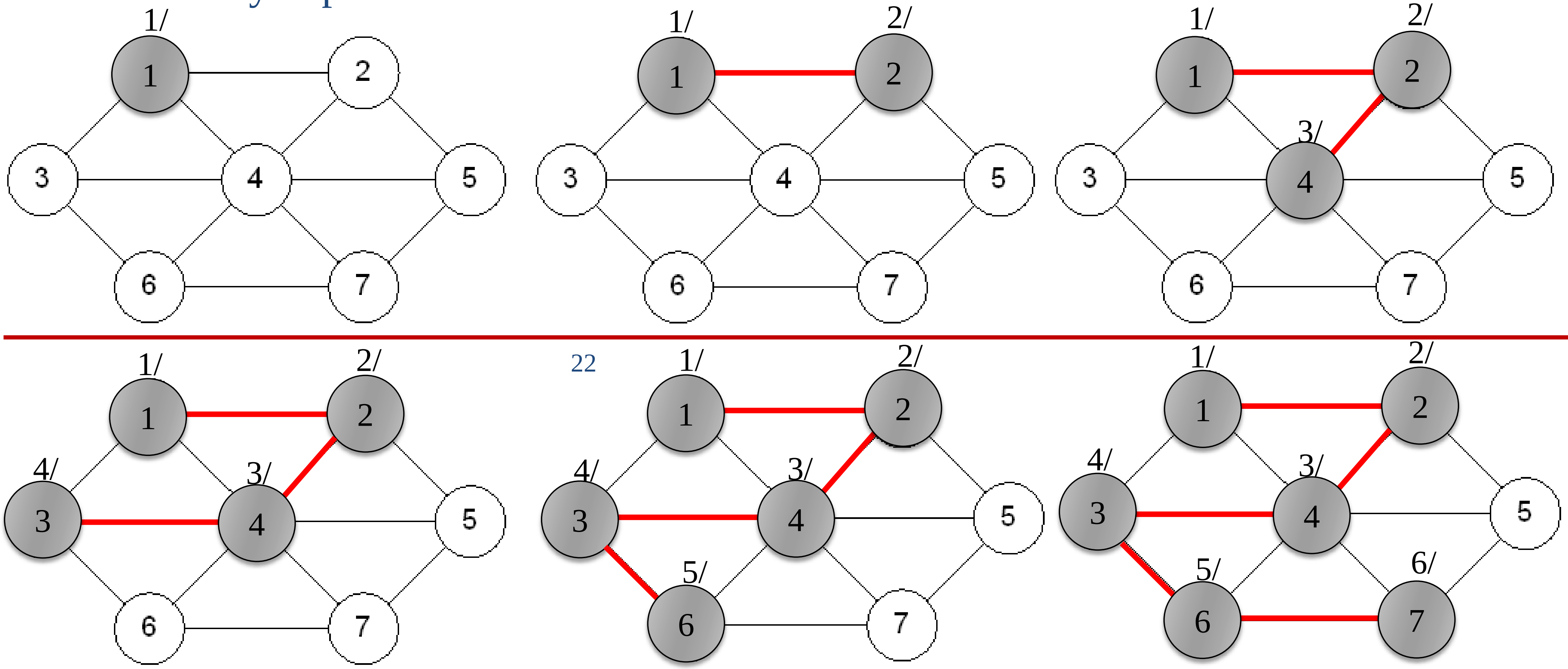
```
Input: Graph and start vertex of graph.
Output: list of vertices reachable from start with their discovery and finish time labels
Steps:
1.  For each vertex v in G
2.    color[v]=white
3.  DFS_Visit(start)
4.  time=0                      //a global timer
```

DFS_Visit (start)

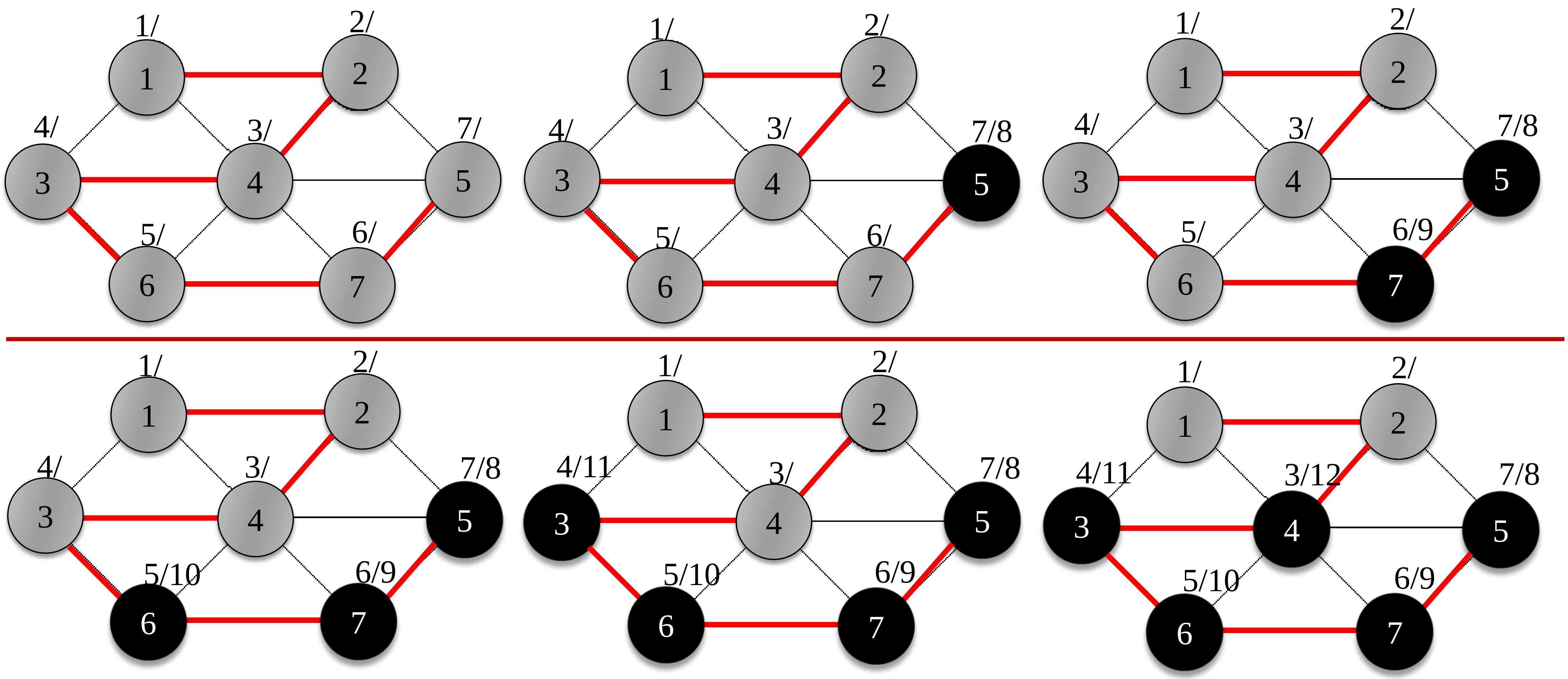
```
1.  color[start]=grey   //discovered
2.  d[start]=time+1            //discovery time
3.  time=time+1
4.  print(start)
5.     for each node v adjacent  to start
6.        if color[v]=white
7.             DFS_Visit (v)
8.  color[start]=black  //finished
9.  f[start]=time+1            //finish time
10. time=time+1
```

Depth First Search

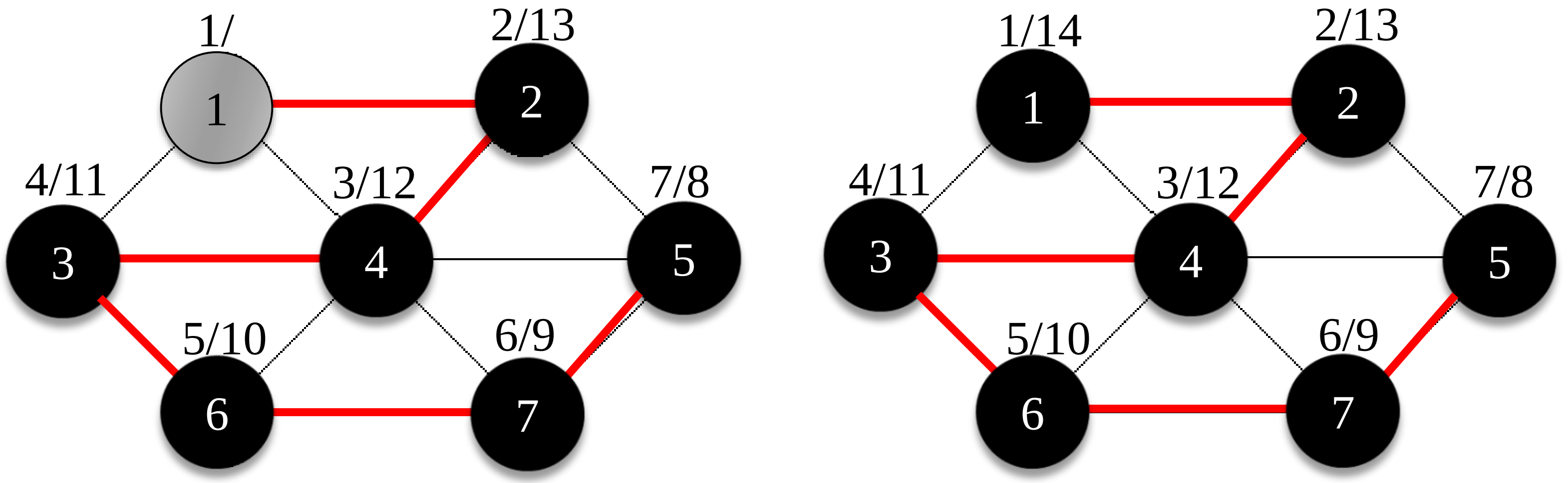
Discovery/explore time



Depth First Search



Depth First Search

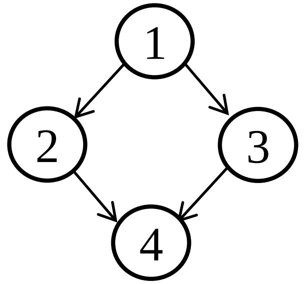


DFS Oder: 1 2 4 3 6 7 5

- PreOrdering: vertices in order of discovery time
  - 1 2 4 3 6 7 5
- PostOrdering: vertices in order of their finish time
  - 5 7 6 3 4 2 1

PreOrdering and Postordering are not reverse of each other.

- Check on the graph at right.

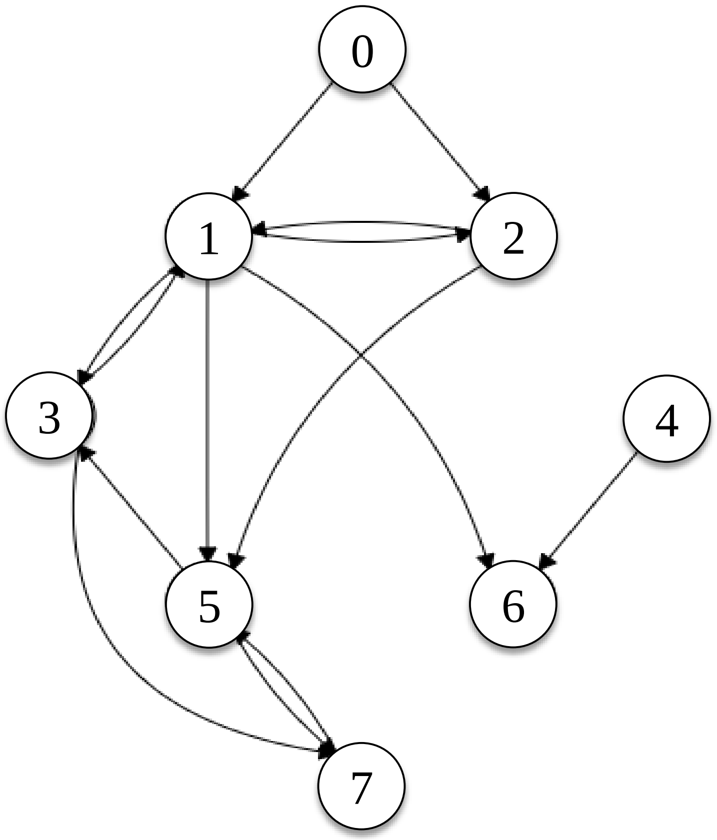


Depth First Search

Start vertex=1

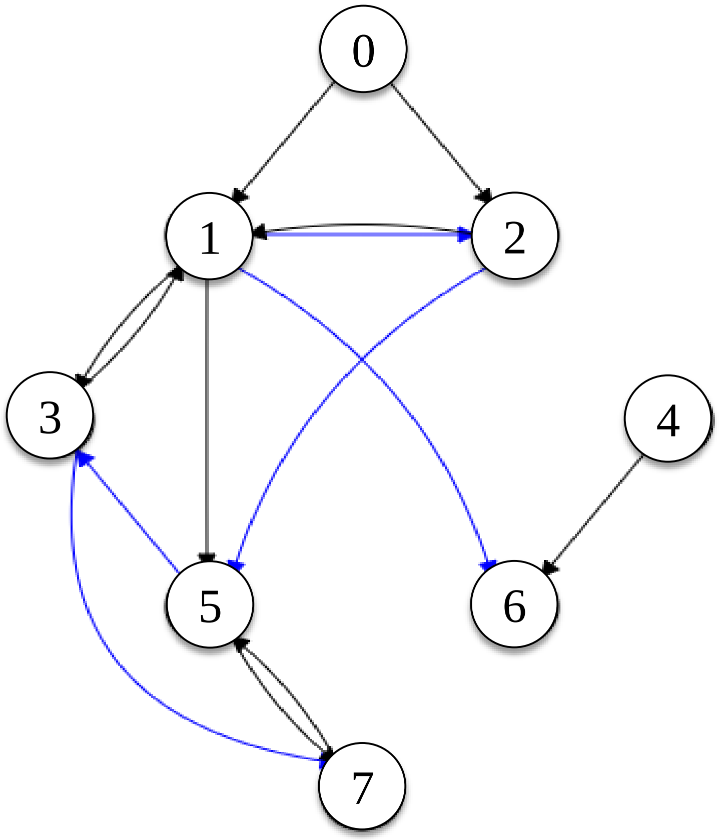
Visited

0	f
1	f
2	f
3	f
4	f
5	f
6	T
7	f



Visited

0	f
1	T
2	T
3	T
4	f
5	T
6	T
7	T



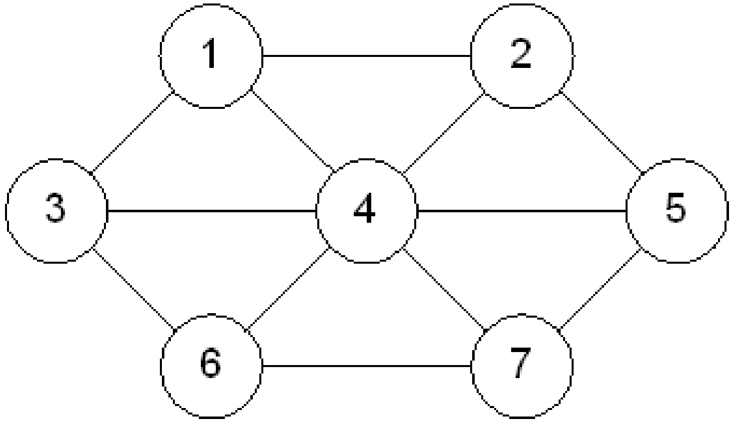
without tri/coloring

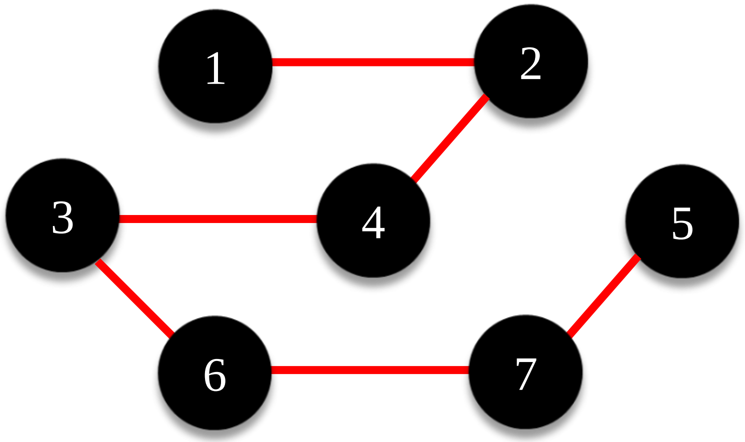
Order: 1, 2, 5, 3, 7, 6

Depth First Search

DFS Tree

If a graph is connected, then by discarding the un-explored edges, resultant set of vertices and edges forms a tree





Multiple trees are possible depending upon choice of start vertex

And how adjacent vertices are picked

Useful Links

A different approach

- http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/breadthSearch.htm
- http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/depthSearch.htm

Visualization

- https://www.cs.usfca.edu/~galles/visualization/BFS.html
- https://www.cs.usfca.edu/~galles/visualization/DFS.html

BFS vs. DFS

- Complexity?
  - Adjacency List?
  - Adjacency Matrix?
- Helps to solve certain problems?
  - Cycle detection
    - both
  - Reachability of vertices
    - both
  - Is the graph connected?
    - both
  - Shortest path in non-weighted graphs
    - BFS

Shortest Path

- Given a graph, find the shortest path from start node to end node
- Applications:
  - Going from one location to other
  - Routing phone calls from one network to other network
  - GPS applications for navigation
  - And many more
- Single source vs. All pair shortest paths
  - SSSP: single source shortest paths
    - Shortest path from one vertex to all other vertices
  - APSP: all pair shortest paths
    - Shortest path from all vertices to all vertices

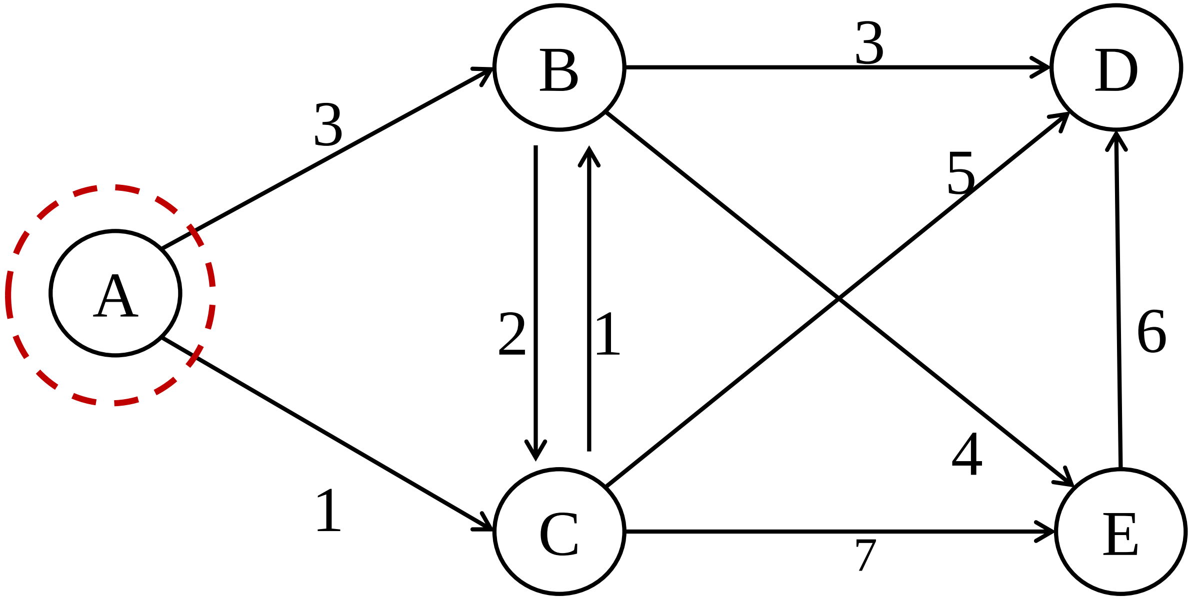
Shortest Path

Weighted Graph?

Shortest path between A to other nodes

B:	[A, B]=	2	D:	[A, C, B, D] = 5
C:	[A, C]=	1	E:	[A, C, B, E] = 6

How to find?



Shortest Path

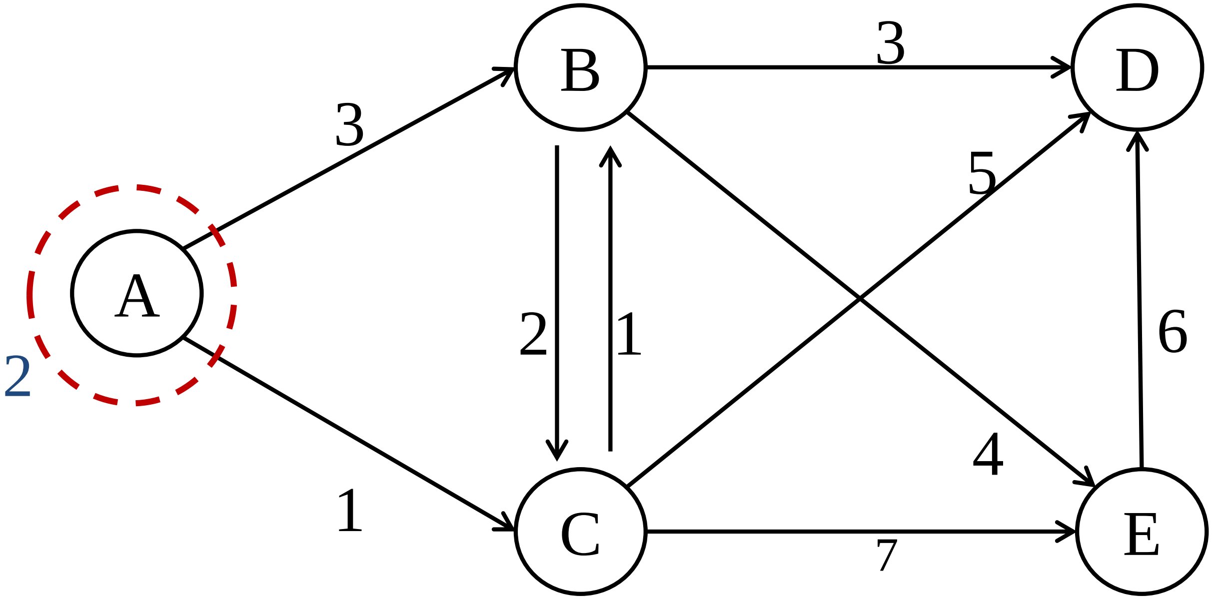
Weighted Graph?

Shortest path between A to other nodes

B:	[A, B]=	2	D:	[A, C, B, D] = 5
C:	[A, C]=	1	E:	[A, C, B, E] = 6

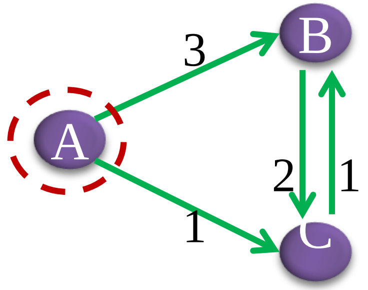
How to find?

- Can we use BFS still?
- BFS shortest path from A to B is 3
- Actual shortest path from A to B is 2
- So simple BFS will not work



Shortest Path

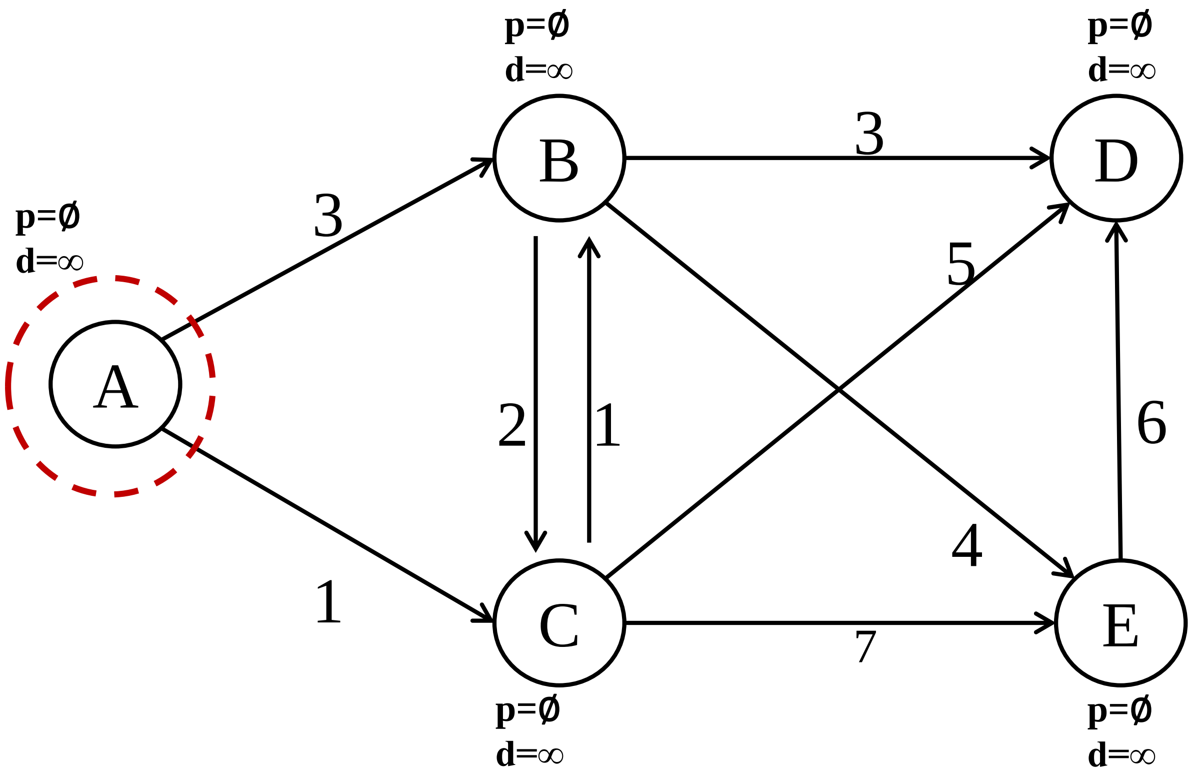
- Modify BFS Shortest Path Algorithm to handle weights
  - Instead of finalizing distance of a nodes when we simply visit them, we should only finalize node (i.e. pop out of the queue) which has *minimum* distance so far.
    - So, a priority queue is needed instead of queue, to pop out the node with minimum distance so far from start node.
    - The node that is popped out has been finalized but nodes inside queue will continue to get updated when ever we can visit them from popped out nodes.
- See the example:
  - A is start vertex, it will be pushed to queue with distance 0 (The queue will contain distance info as well.)
  - Pop the queue and check adjacent nodes, B and C will be pushed to queue with distance 3 and 1.
  - Next, instead of popping out B which comes first order wise, we will pop C
    - Because it’s distance is 1 from A
  - Now B is also adjacent node of C, so we will see if its distance can be updated which was 3 when it was discovered from A.
    - it will become 2 now
  - Keep on updating distances of adjacent nodes, when a node is popped out.
    - Untill queue becomes empty



Dijkstra's Algorithm

Step 1: Initialize Graph

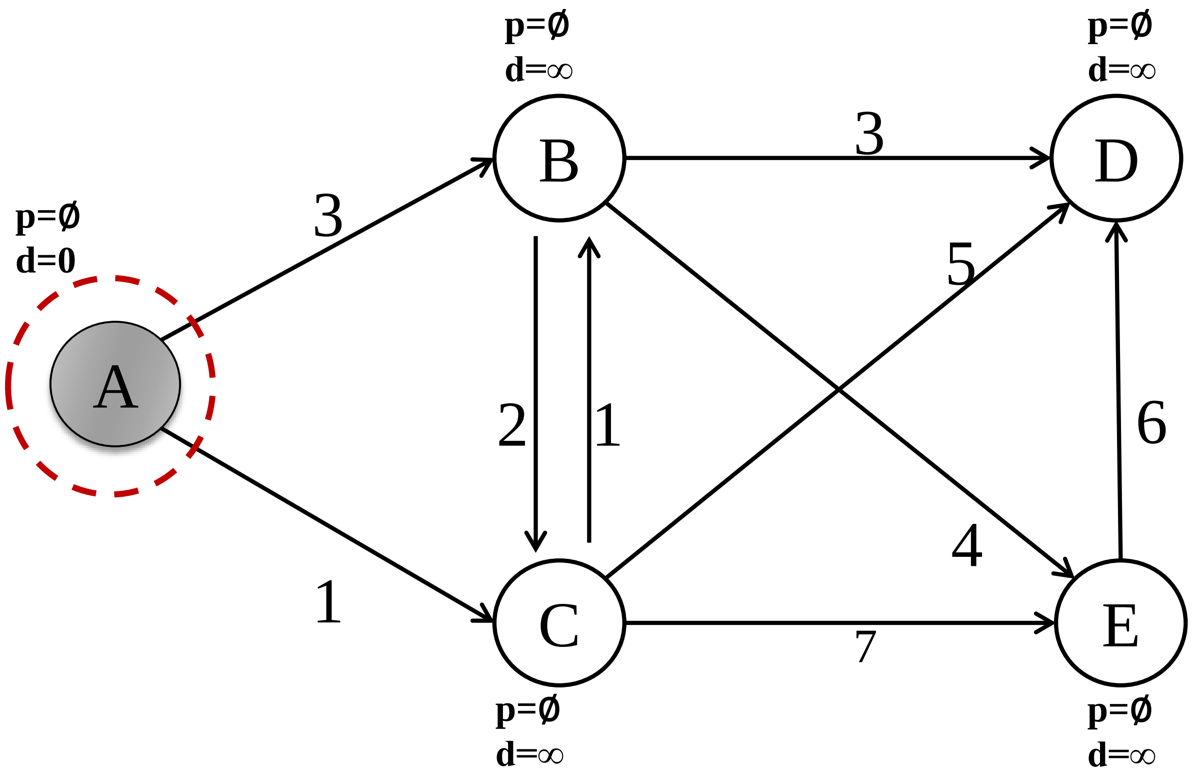
```
Mark all node's prev pointer to NULL
Mark all node's distance to infinity
```



Dijkstra’s Algorithm

STEP 2:

```
Initialize distance of start node 0
Push start node to queue
```



A				
0				

Dijkstra's Algorithm

Now B and C are adjacent nodes of A, they will be pushed to queue, with their respective distances. Distance is calculated as:

d[v]=d[u]+weight(u,v)

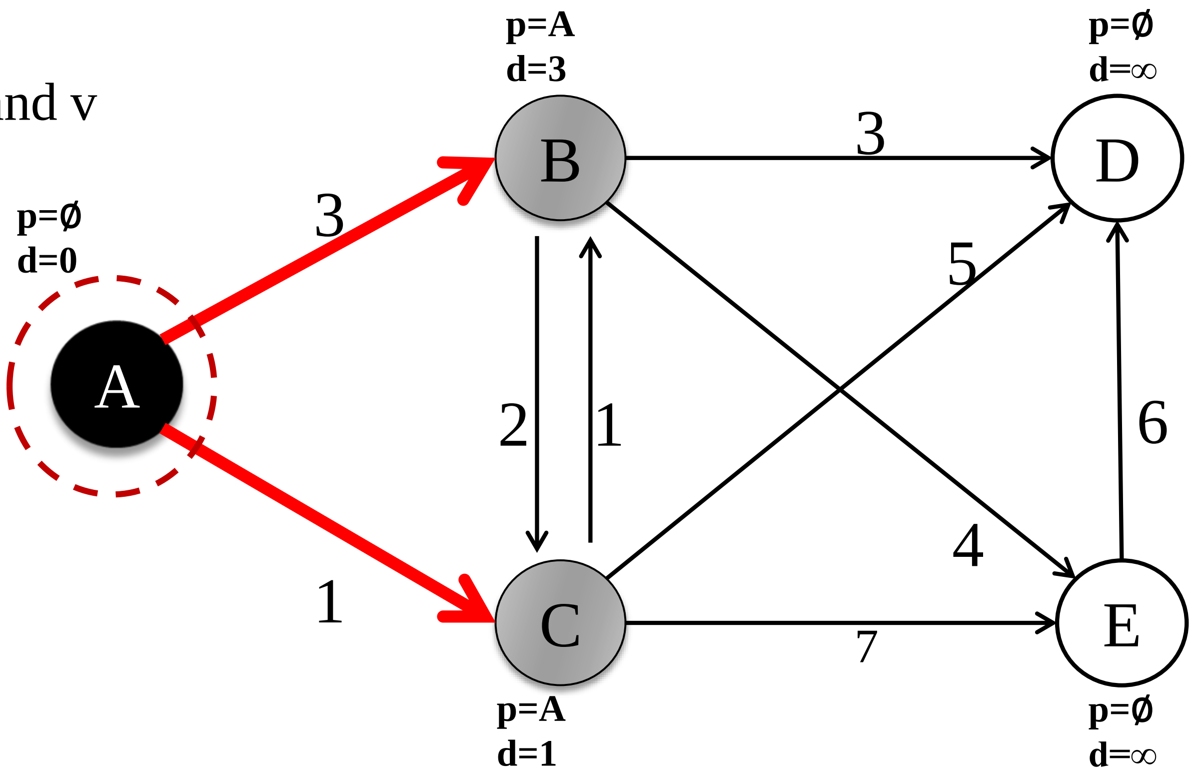
u and v are two connected nodes,

weight(u,e) is edge weight between u and v

- Prev pointer will point to A

A				
0				

B	C			
3	1			



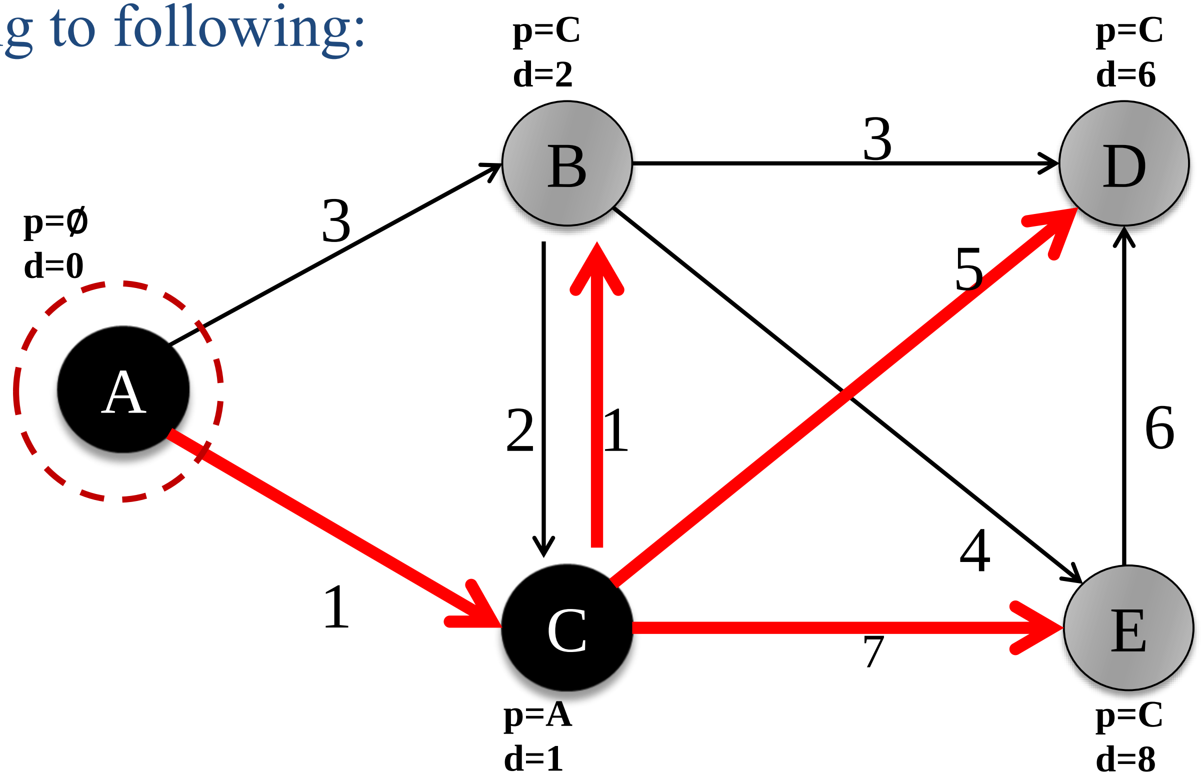
Dijkstra's Algorithm

C has minimum distance so far, so it will be popped out.

- Two new nodes D and E are pushed to queue, and B's distance and prev pointer is updated
- Node's distance is updated according to following:
  - If d[v] > d[u]+weight(u,v)
    - d[v]=d[u]+weight(u,v)
    - p[v]=u

B	C			
3	1			

B	D	E		
2	6	8		



Dijkstra’s Algorithm

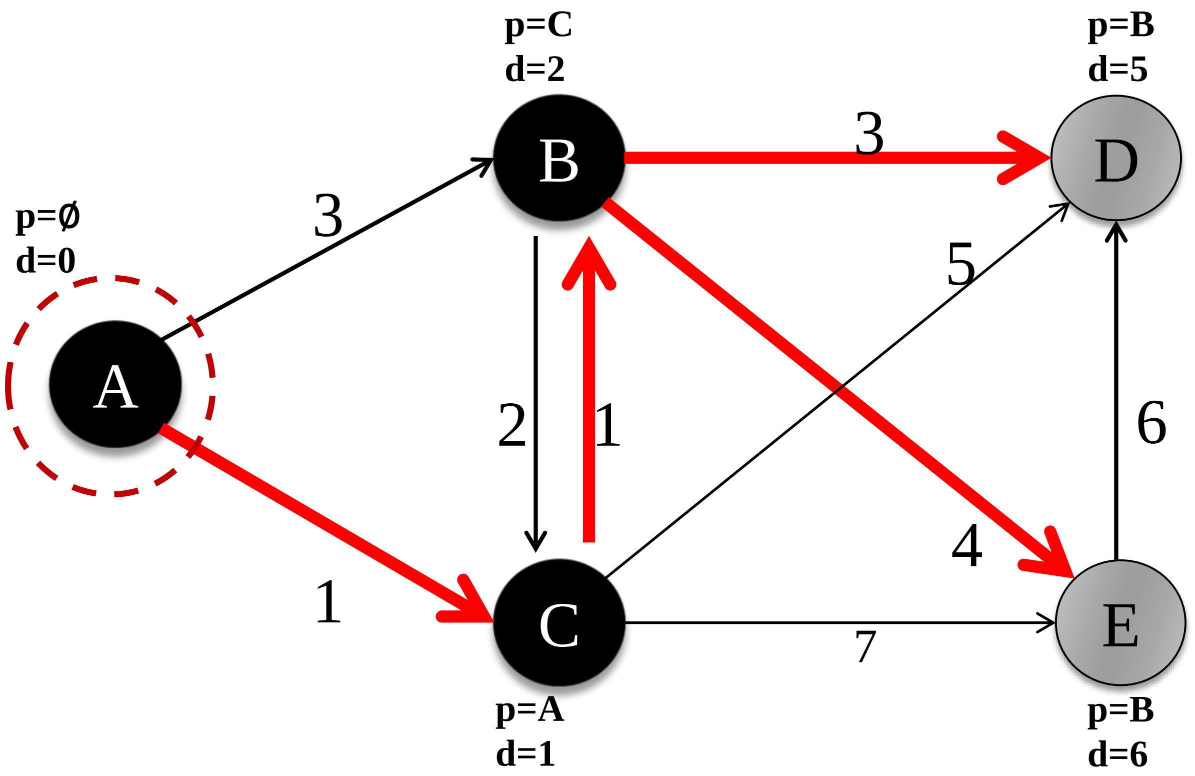
Now B will be popped out, as it has minimum distance of 2

Is it possible to reduce distance of D and E, as they are adjacent nodes of B

If Yes, update them

B	D	E		
2	6	8		

D	E			
5	6			



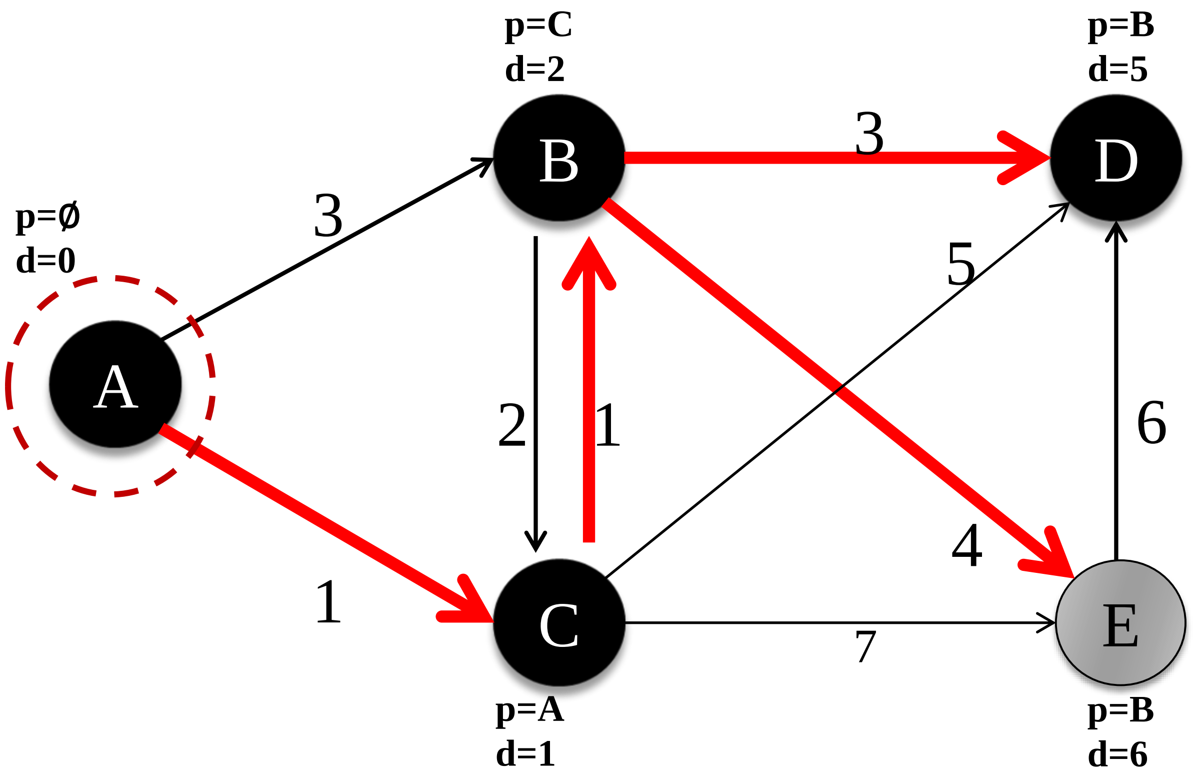
Dijkstra’s Algorithm

Now D will be popped out, as it has minimum distance of 5

No node will be updated

D	E			
5	6			

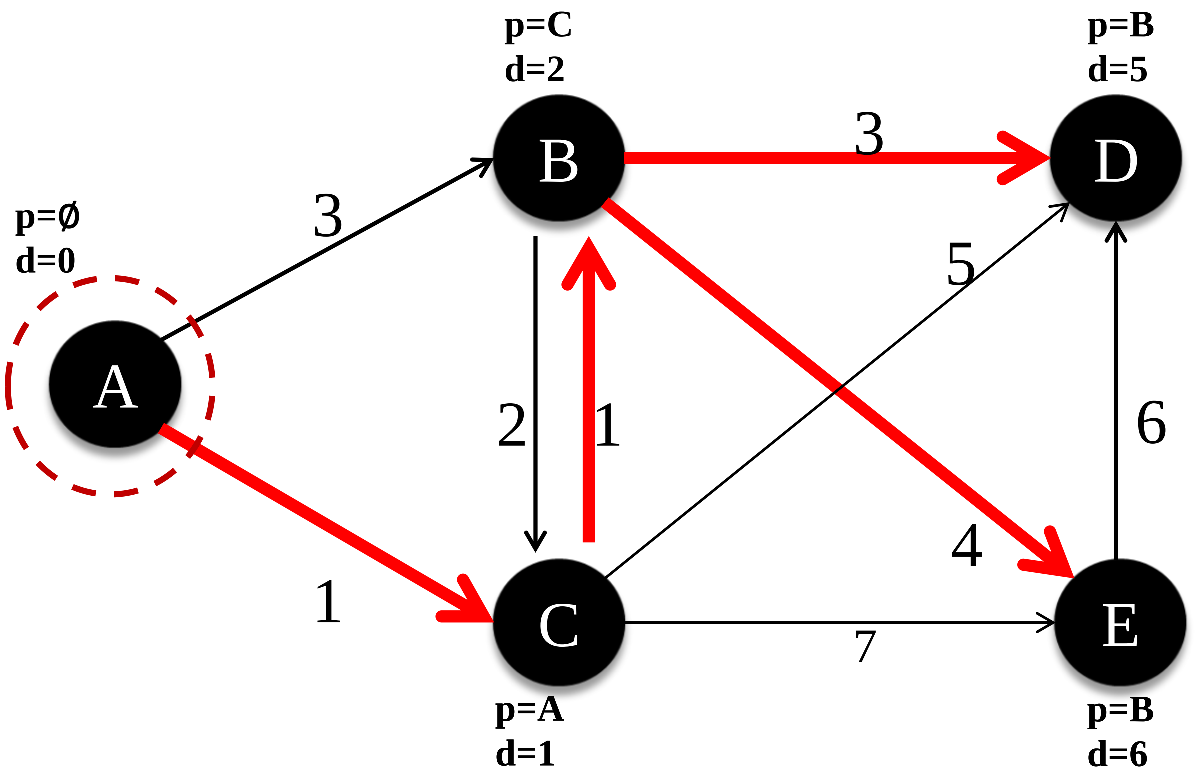
E				
6				



Dijkstra's Algorithm

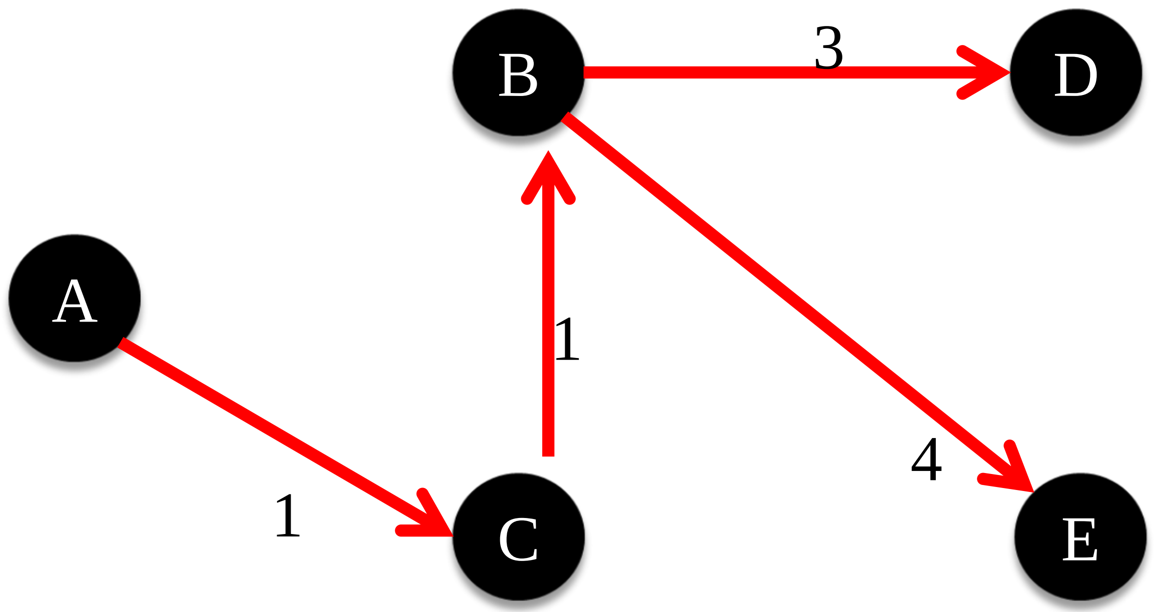
Finally pop E

Queue has become empty and all distance and paths have been found



Dijkstra’s Algorithm

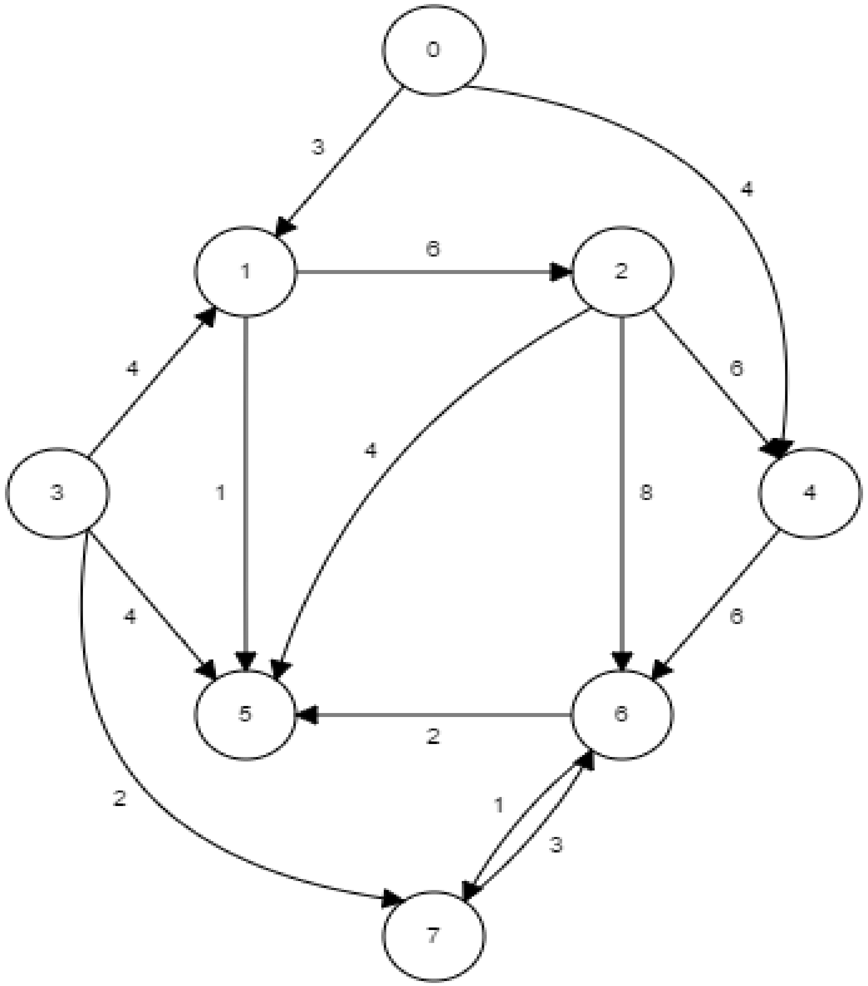
Dijkstra’s Shortest Path Tree



Directed Graph

Start vertex: 1

Vertex	Cost	Path	
0	INF	-1	No Path
1	0	-1	1
2	6	1	1 2
3	INF	-1	No Path
4	12	2	1 2 4
5	1	1	1 5
6	14	2	1 2 6
7	15	6	1 2 6 7



Path is alternate for parent

Cost is alternate for distance

Dijkstra’s Shortest Path Algorithm

Algorithm: DIJKSTRA(G, start)

```
Input: Graph G and start vertex of graph G
Steps:
PQ = new PriorityQueue(V) //where V is number of vertices
d[start] = 0
For each node v in G         //initialization of all nodes
   if start != v
       d[v]= infinity
    end if
    p[v]=null
    PQ.add(v,d[v])
End For
while PQ is not empty
   u = PQ.removeMin()
   For each node v adjacent to u that is in PQ //updating distances of adjacent nodes
       if d[v] > d[u]+ cost(u,v) //cost mean edge weight between u and v
            d[v] = d[u] + cost(u,v)   // update distance with new value
            p[v] = u                  // update prev pointers to maintain path
            PQ.update_Priority(v,d[v]) // assumes that v is already in PQ
        End if
   End For
End While
return d[] and p[]
```

Time Complexity?

- Depends upon implementation of priority queue
  - With array/ linked list implementation
    - removeMin() will take O(V) time
    - Priority update at each distance update→ O(1) time
  - With binary heap implementation
    - removeMin() will take O(logV) time
    - Priority update at each distance update→ O(logV) time

- Time complexity with array or list as priority queue will be $O(V^2)$
- Time complexity with heap as priority queue will be O(E+ VlogV)
- For details please visit:
  - http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
  - https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/lec20/lec20.htm

Dijkstra’s Shortest Path Algorithm

Run Dijkstra’s algorithm on following graph, taking 0 as start node.