Design and Analysis of Algorithms
BFS, DFS, and topological sort

From: Haidong Xue

Slides Provided By: Muhammad Atif Tahir

Presented by: Farrukh Salim Shaikh

Graph Traversal

Two techniques of graph traversing for both directed or undirected graphs:

- Breadth First Search (BFS) – Uses Queue
- Depth First Search (DFS) – Uses Stack

What are BFS and DFS?

- Two ambiguous terms: search, traversal
- Visit each of vertices once
  - E.g.: tree walks of a binary search tree
  - Traversal
  - Search
- Start from a vertex, visit all the **reachable** vertices
  - Search

What are BFS and DFS?

- To eliminate the ambiguity, in my class
- Search indicates
  - Start from a vertex, visit all the **reachable** vertices
- Traversal indicates
  - Visit each of vertices once
- However, in other materials, you may see some time “search” is considered as “traversal”

What are BFS and DFS?

- BFS
  - Start from a vertex, visit all the reachable vertices in a breadth first manner
- DFS
  - Start from a vertex, visit all the reachable vertices in a depth first manner
- BFS or DFS based traversal
  - Repeat BFS or DFS for unreachable vertices

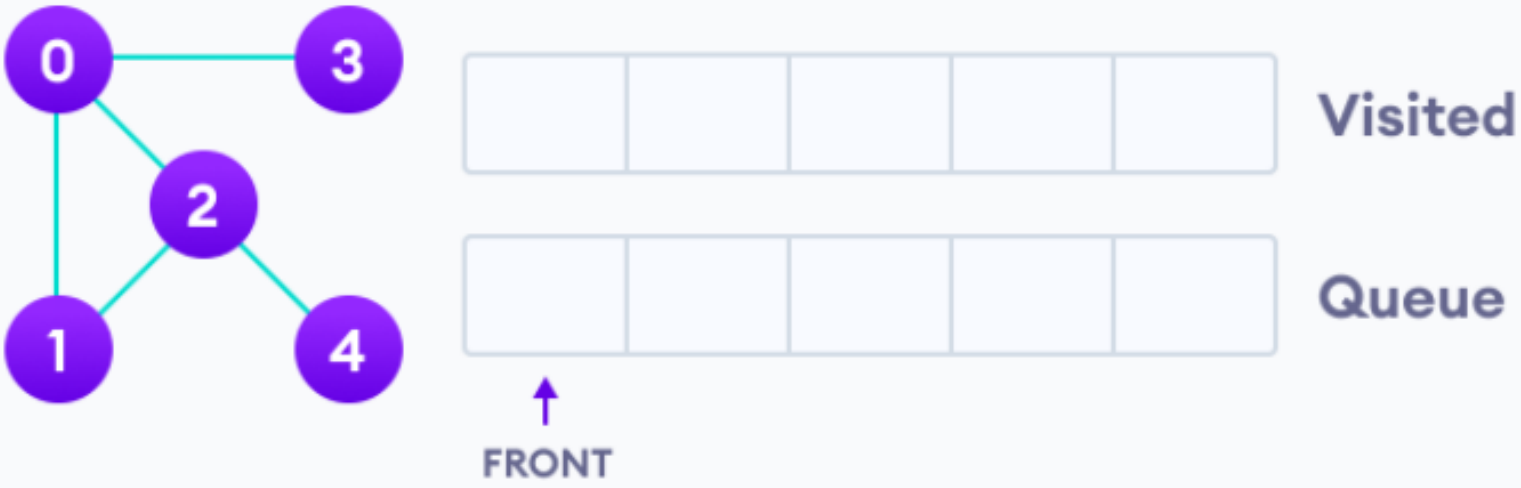
BFS, BF tree and shortest path

- Breadth-first search
  - From a source vertex s
  - Breadth-firstly explores the edges to discover every vertex that is reachable from s
- **BFS**(s)

```
visit(s);
queue.insert(s);
while( queue is not empty ){
    u = queue.extractHead();
    for each edge <u, d>{
        if(d has not been visited)
            visit(d);
            queue.insert(d);
    }
}
```

Graph Traversal : BFS

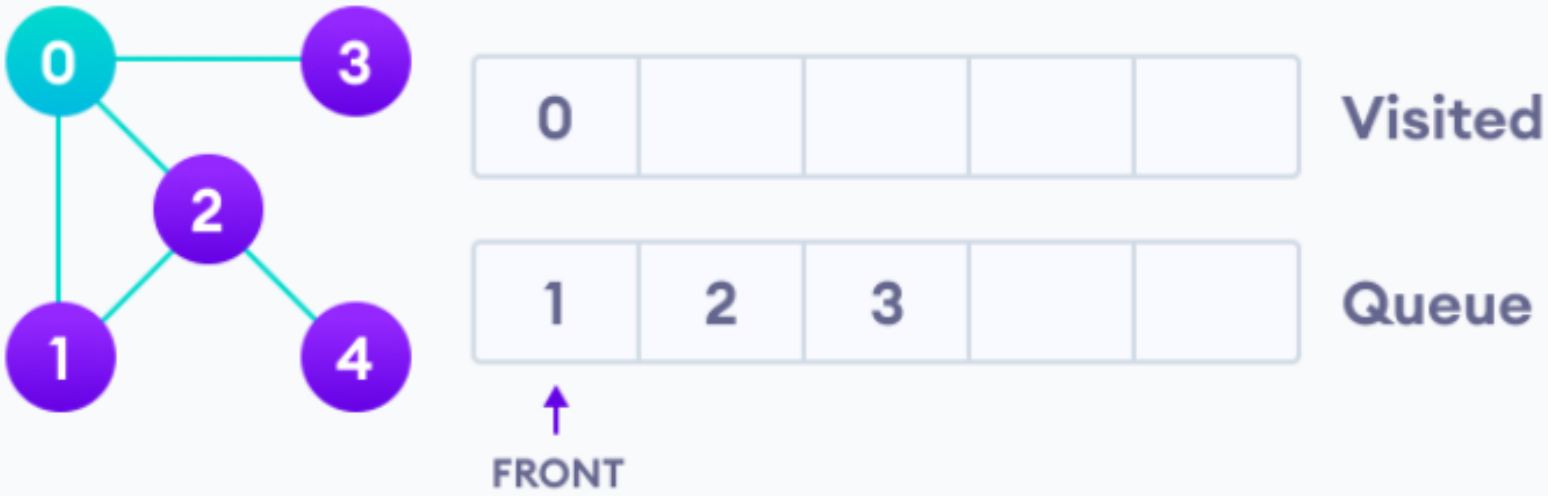
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



Undirected graph with 5 vertices

Graph Traversal : BFS

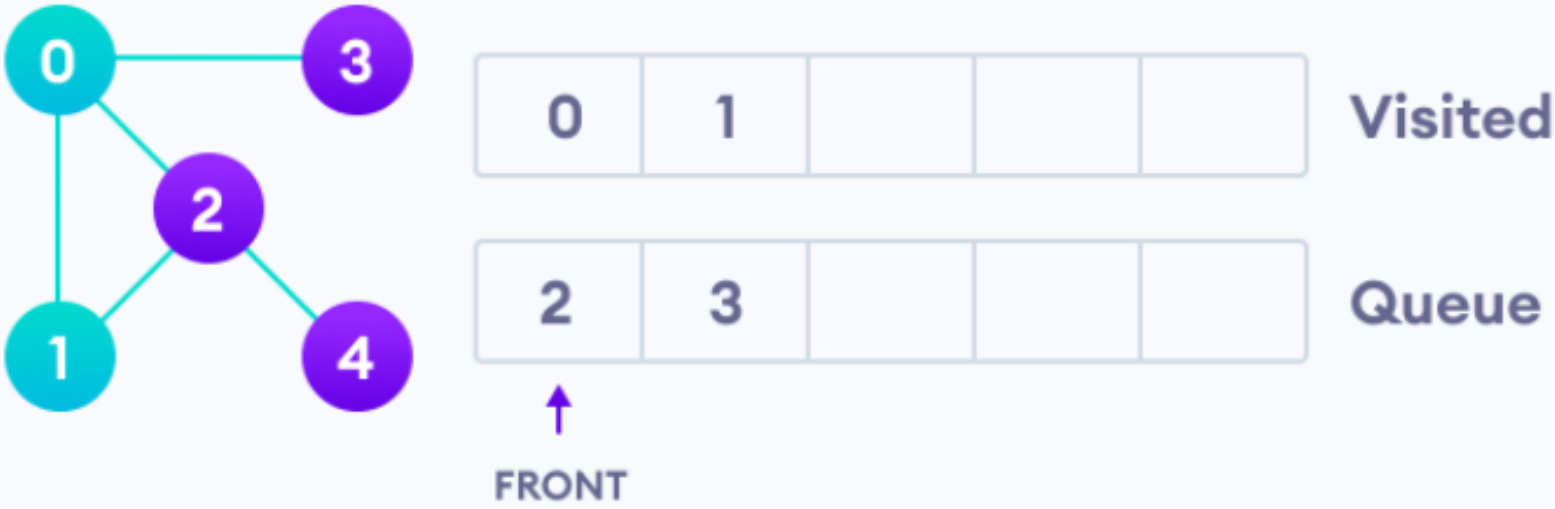
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



Visit start vertex and add its adjacent vertices to queue

Graph Traversal : BFS

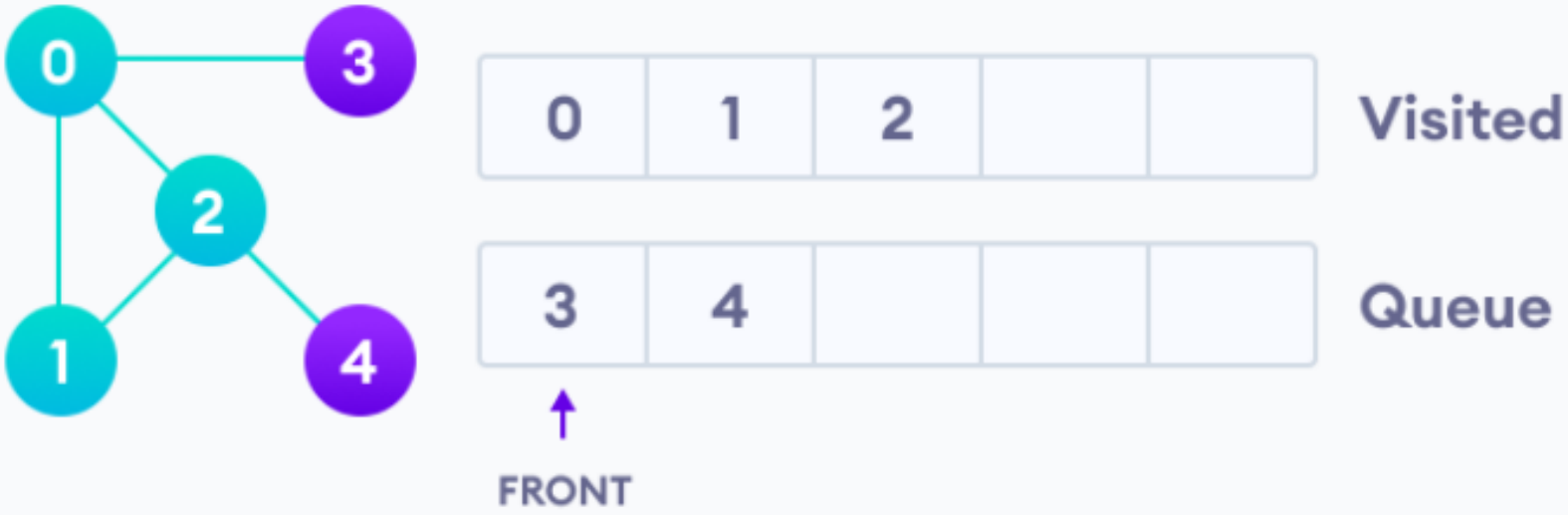
Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Visit the first neighbour of start node 0, which is 1

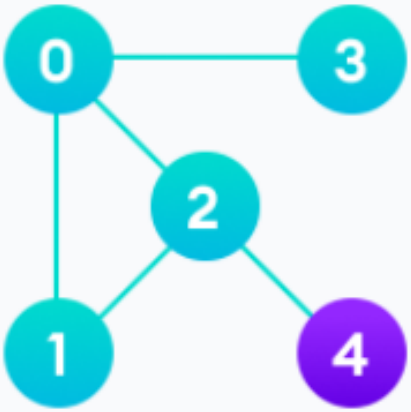
Graph Traversal : BFS

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



Visit 2 which was added to queue earlier to add its neighbours

Graph Traversal : BFS



0	1	2	3		Visited
4					Queue

FRONT

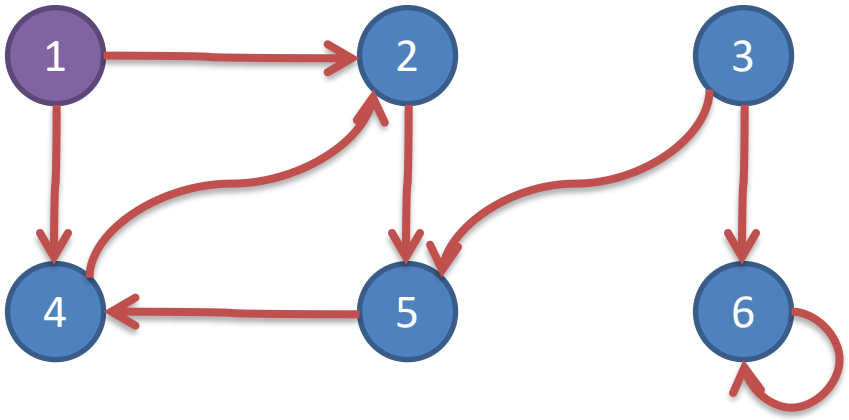
4 remains in the queue

Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.

Graph Traversal : BFS

BFS, BF tree and shortest path

BFS(1)



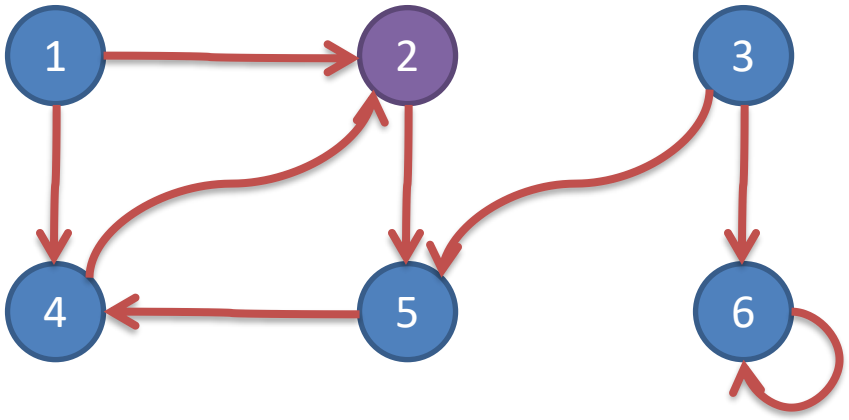
Queue: 1 4 2 5

```
BFS(s)
    visit(s);
    queue.insert(s);
    while( queue is not empty ){
        u = queue.extractHead();
        for each edge <u, d>{
            if(d has not been visited)
                visit(d);
                queue.insert(d);
        }
    }
```

Visit order: 1 4 2 5

BFS, BF tree and shortest path

BFS(2)



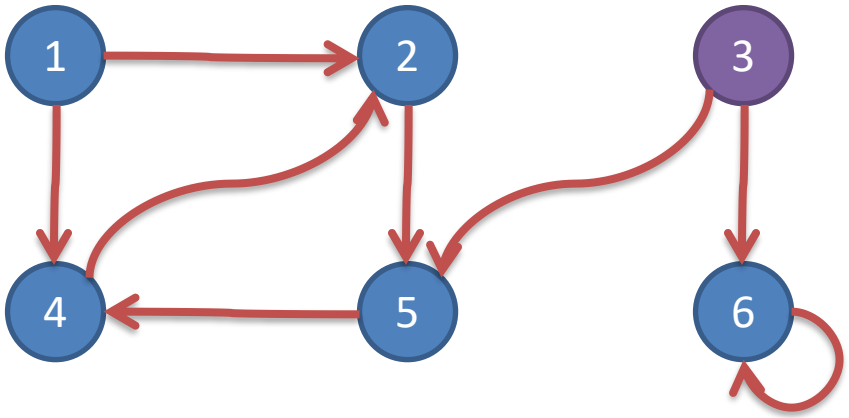
Queue: 2 5 4

```
BFS(s)
    visit(s);
    queue.insert(s);
    while( queue is not empty ){
        u = queue.extractHead();
        for each edge <u, d>{
            if(d has not been visited)
                visit(d);
                queue.insert(d);
        }
    }
```

Visit order: 2 5 4

BFS, BF tree and shortest path

BFS(3)



Queue: 3 6 5 4 2

Visit order: 3 6 5 4 2

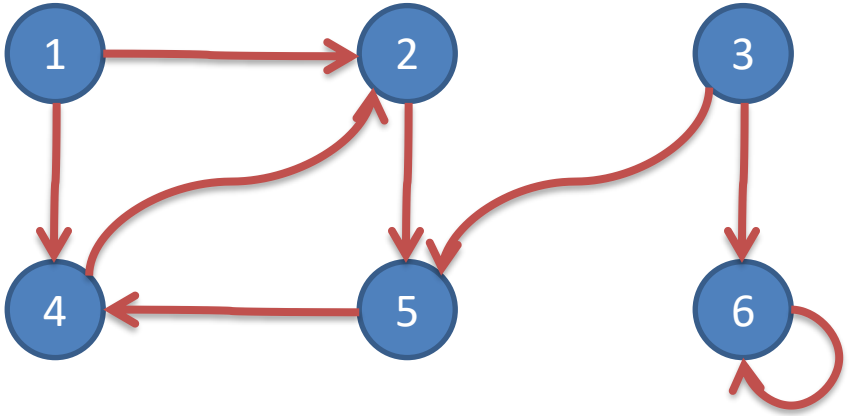
```
BFS(s)
    visit(s);
    queue.insert(s);
    while( queue is not empty ){
        u = queue.extractHead();
        for each edge <u, d>{
            if(d has not been visited)
                visit(d);
                queue.insert(d);
        }
    }
```

Note that: no matter visit 5 first or visit 6 first, they are BFS

BFS, BF tree and shortest path

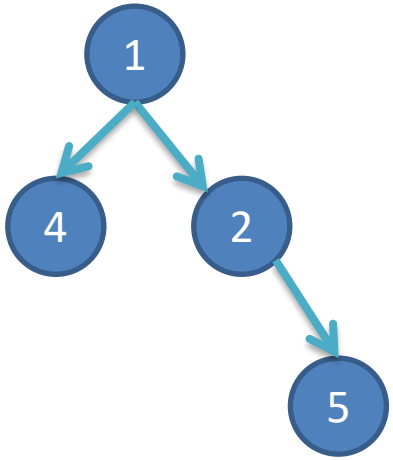
- Byproducts of BFS(s)
  - Breadth first tree
    - The tree constructed when a BFS is done
  - Shortest path
    - A path with minimum number of edges from one vertex to another
    - BFS(s) find out all the shortest paths from s to all its reachable vertices

BFS, BF tree and shortest path



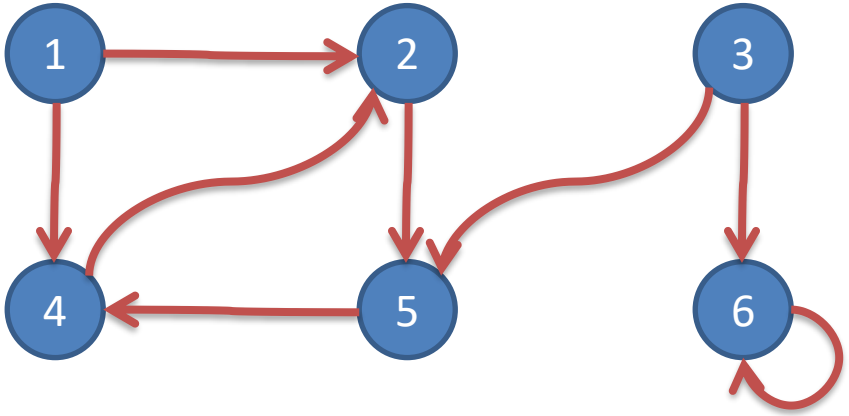
BFS(1): 1 4 2 5

BF Tree:



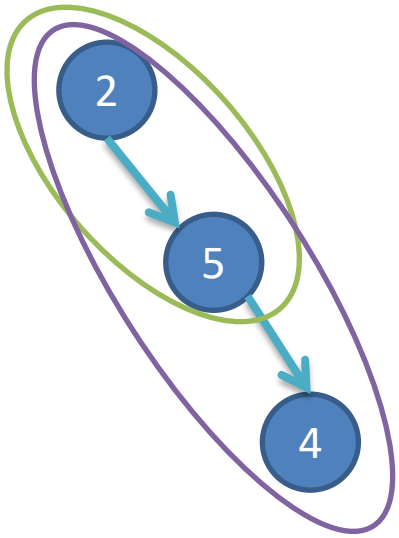
All shortest paths started from vertex 1 are found
e.g. 1 to 5

BFS, BF tree and shortest path



BFS(2): 2 5 4

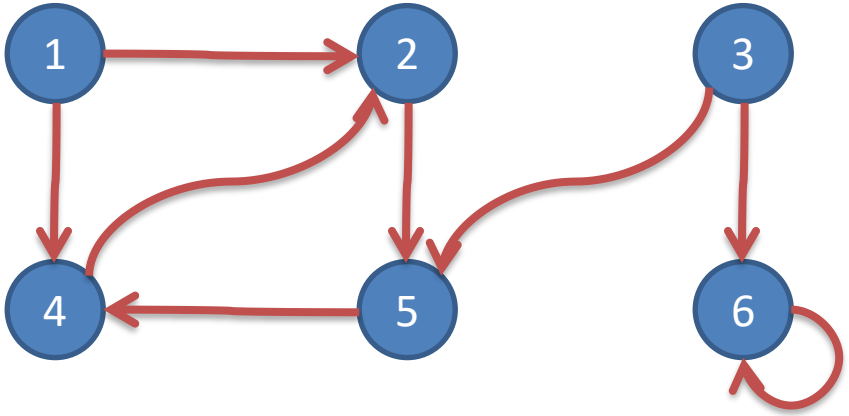
BF Tree:



Shortest 2 to 5

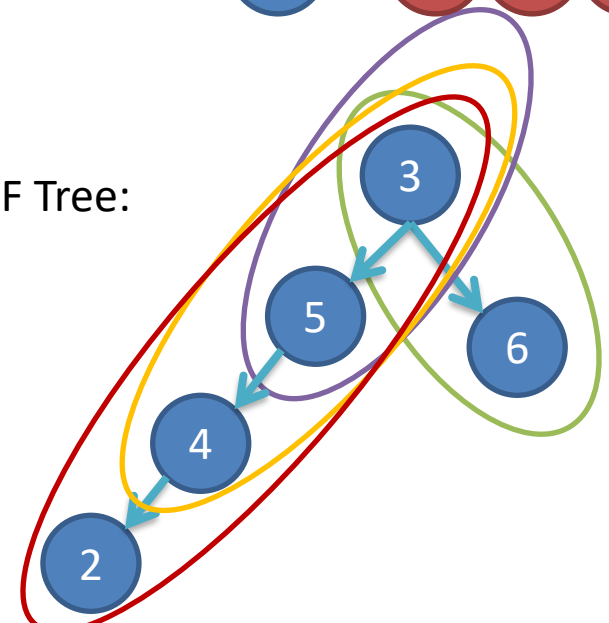
Shortest 2 to 4

BFS, BF tree and shortest path



BFS(3): 3 6 5 4 2

BF Tree:



Shortest 3 to 6

Shortest 3 to 5

Shortest 3 to 4

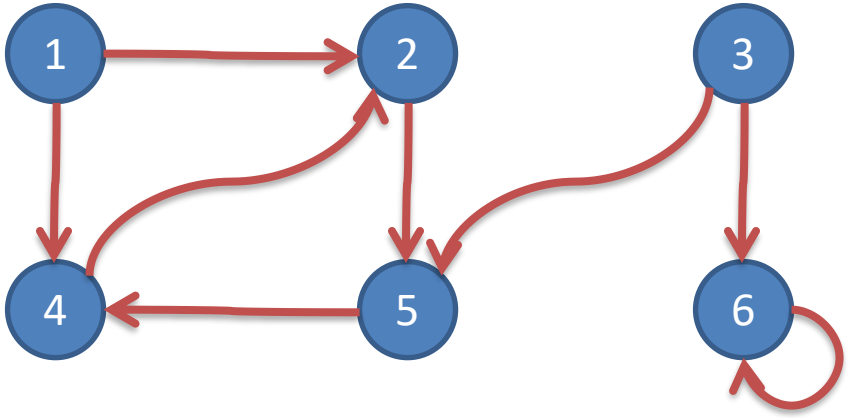
Shortest 3 to 2

BFS, BF tree and shortest path

- BFS Traversal
- **BFS_Traversal**(G)

```
for each v in G{
    if (v has not been visited)
        BFS(v);
}
```

BFS, BF tree and shortest path




```
BFS_Traversal(G)
for each v in G{
        if (v has not been visited)
                BFS(v);
}
```

Queue: 1 4 2 5 3 6

Visit order: 1 4 2 5 3 6

DFS, DF tree

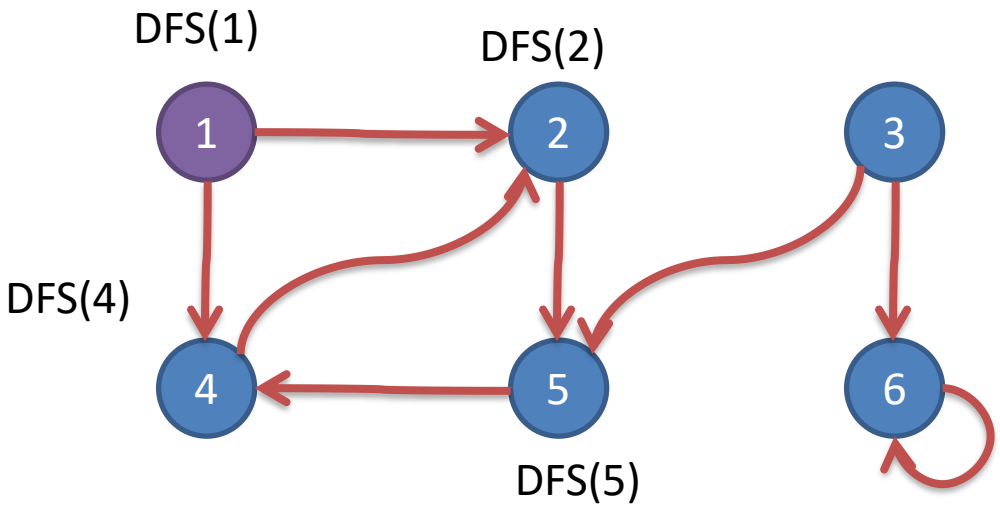
- Depth-first search
  - From a source vertex s
  - Depth-firstly search explores the edges to discover every vertex that is reachable from s
- **DFS**(s):

```
s.underDFS = true;    // grey
for each edge <s, d>{
    if(! d.underDFS and d has not been visited)
        DFS(d);
}
Visit(s); // black
```

From the deepest one to the current one

DFS, DF tree

DFS(1)

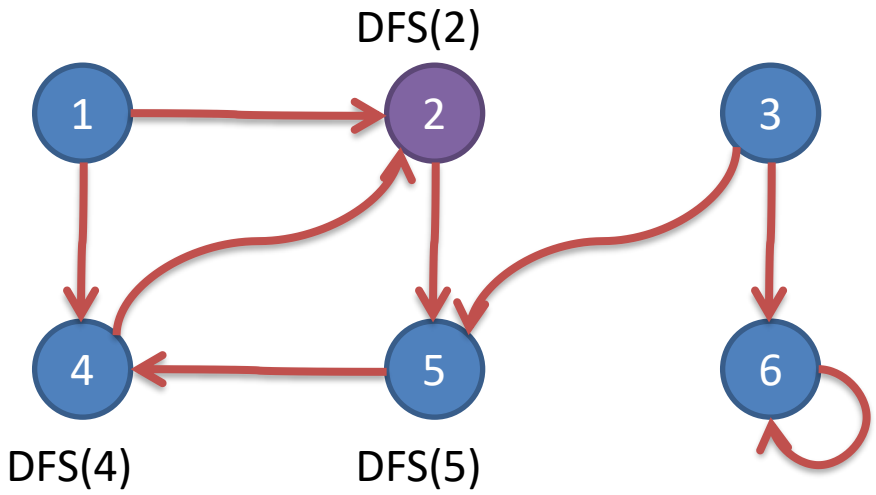



```
DFS(s):
    s.underDFS = true;
     for each edge <s, d>{
          if((! d.underDFS and d has
          not been visited)
           DFS(d);
    }
    Visit(s);
```

Visit order:

DFS, DF tree

DFS(2)

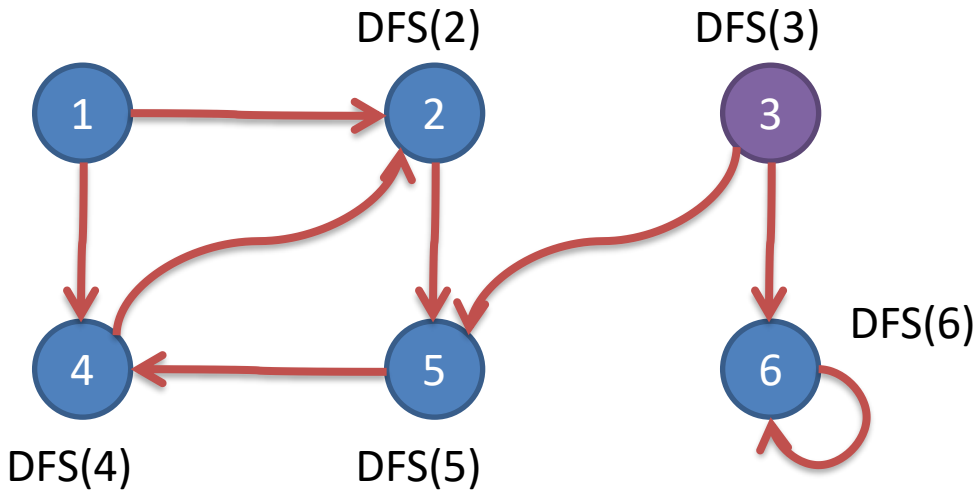



```
DFS(s):
    s.underDFS = true;
     for each edge <s, d>{
          if((! d.underDFS and d has
          not been visited)
               DFS(d);
     }
     Visit(s);
```

Visit order:

DFS, DF tree

DFS(3)




```
DFS(s):
    s.underDFS = true;
    for each edge <s, d>{
        if((!d.underDFS and d has
        not been visited)
            DFS(d);
    }
    Visit(s);
```

Visit order: 6 2 4 5 3

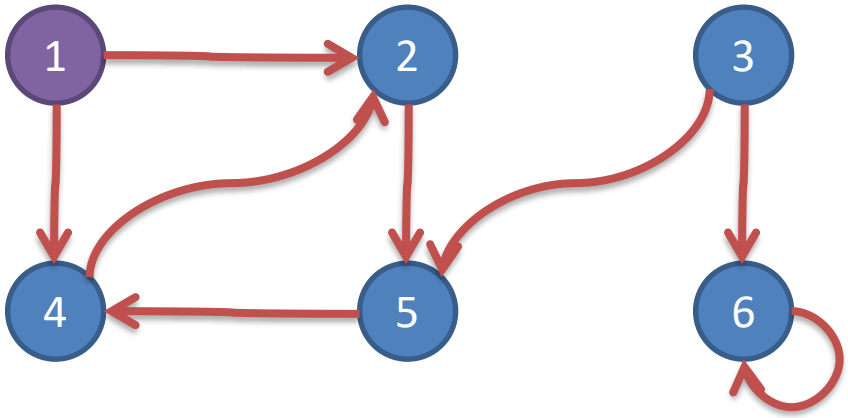
The reachable vertices are exactly the same with BFS, but with a different order

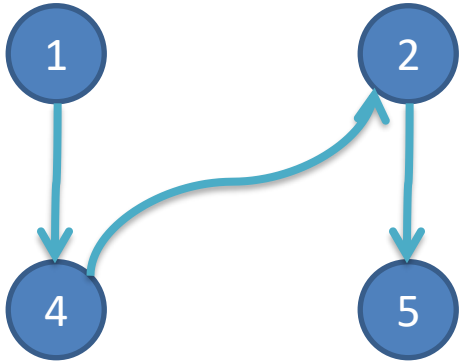
DFS, DF tree

- Depth first tree
  - The tree constructed when a DFS is done

DFS, DF tree

DF Tree **of DFS**(1)



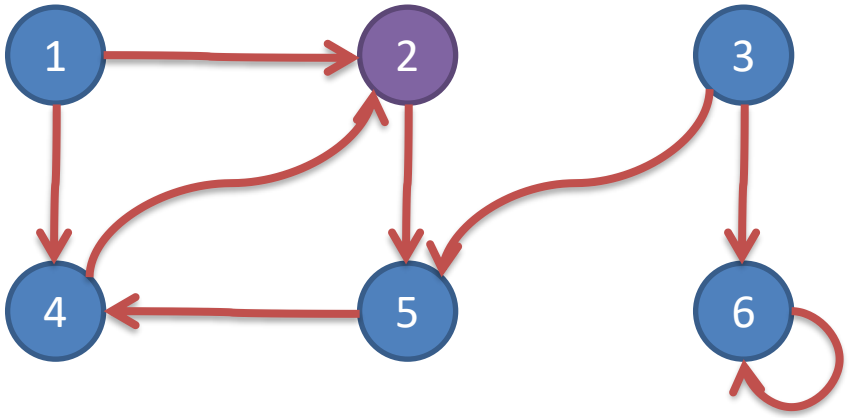


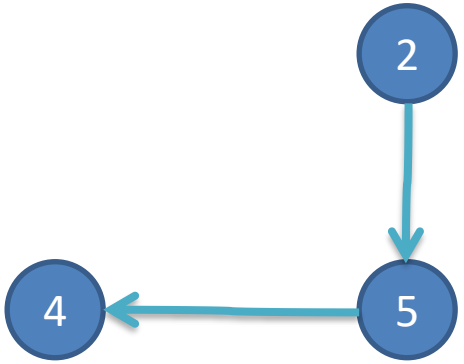
DF Tree

Visit order:

DFS, DF tree

DF Tree of **DFS**(2)



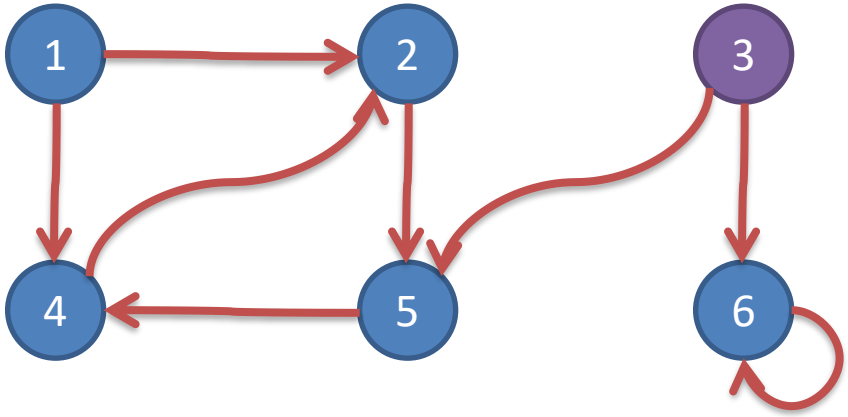


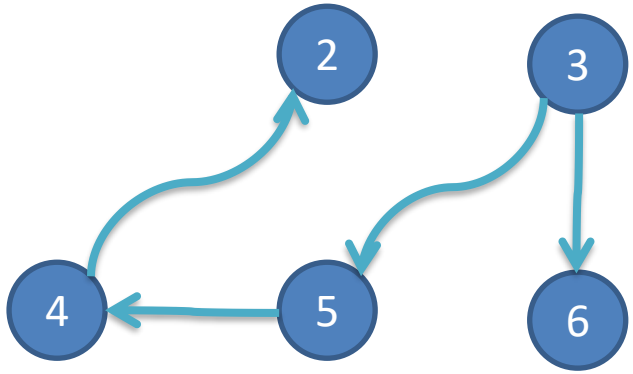
DF Tree

Visit order:

DFS, DF tree

DF Tree **of DFS**(3)





DF Tree

Visit order:

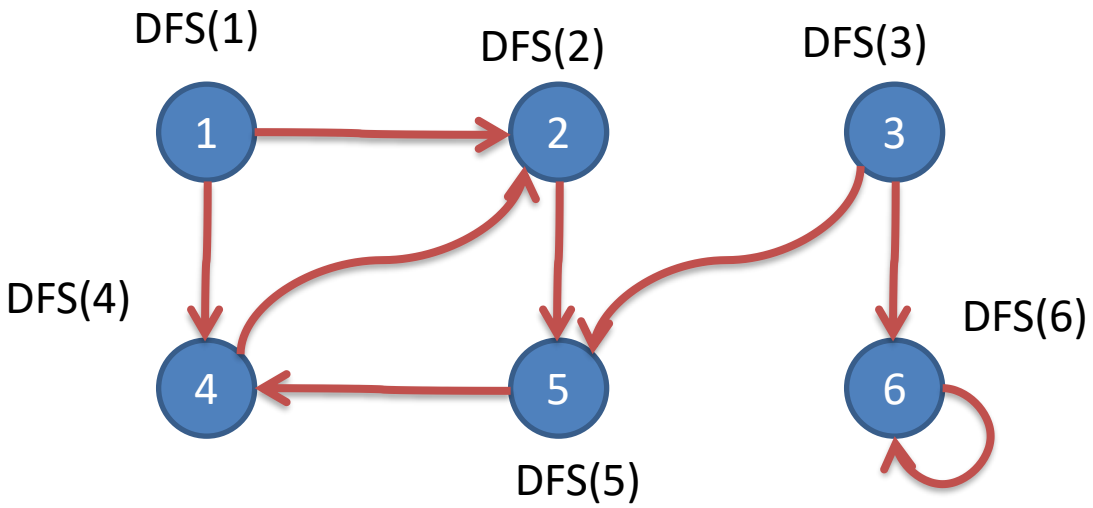
DFS, DF tree

- DFS Traversal // (The DFS in the textbook)
- **DFS_Traversal**(G)

```
for each v in G{
    if (v has not been visited)
        DFS(v);
}
```

DFS, DF tree

DFS_Traversal(G)

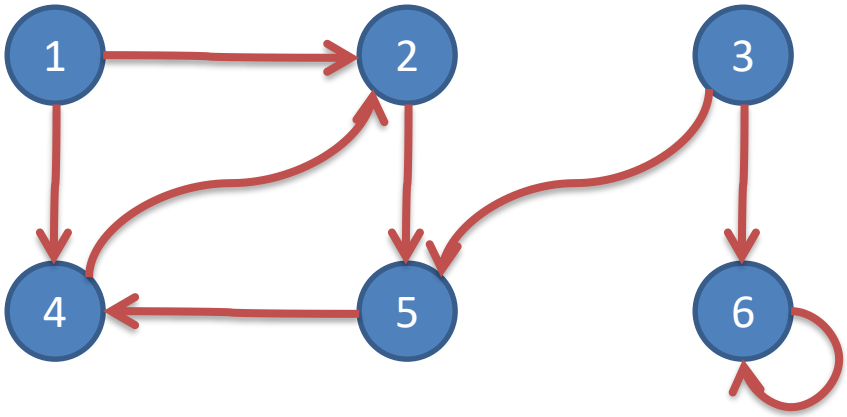



```
DFS_Traversal(G)
for each v in G{
        if (v has not been visited)
                DFS(v);
}
```

Visit order: 5 2 4 1 6 3

Topological sort

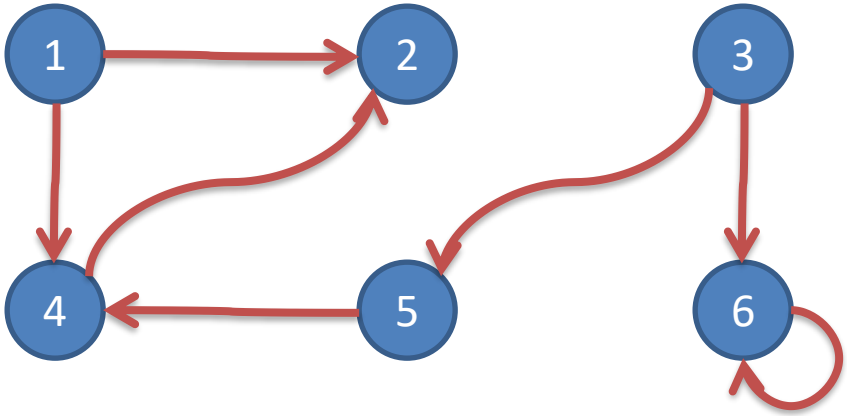
- DAG: directed acyclic graph
  - A graph without cycles



Dag? No

Topological sort

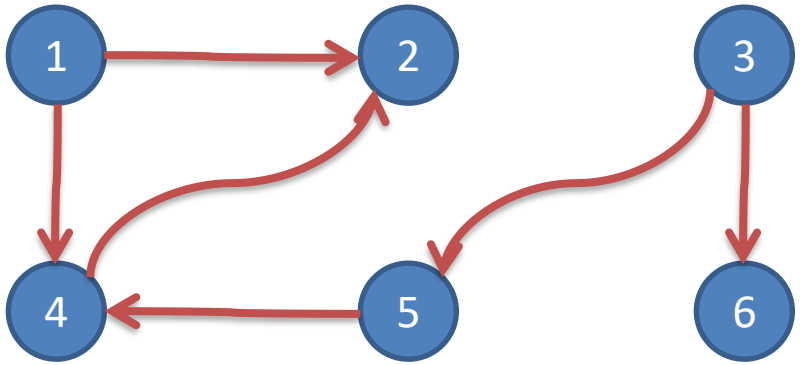
- DAG: directed acyclic graph
  - A graph without cycles



Dag? No

Topological sort

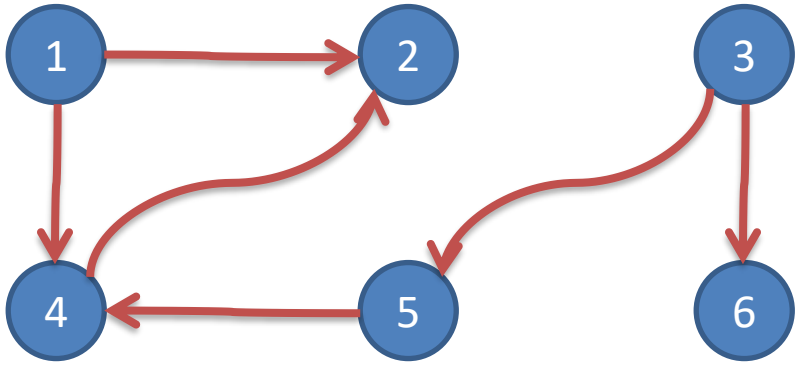
- Ordering in DAGs
  - If there is an edge <u, v>, then u appears before v in the ordering

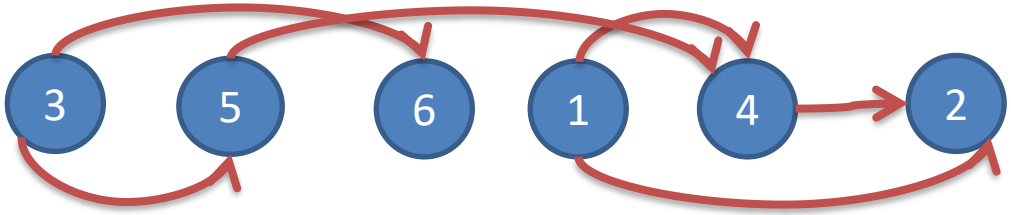


Dag? Yes

Topological sort

- Example



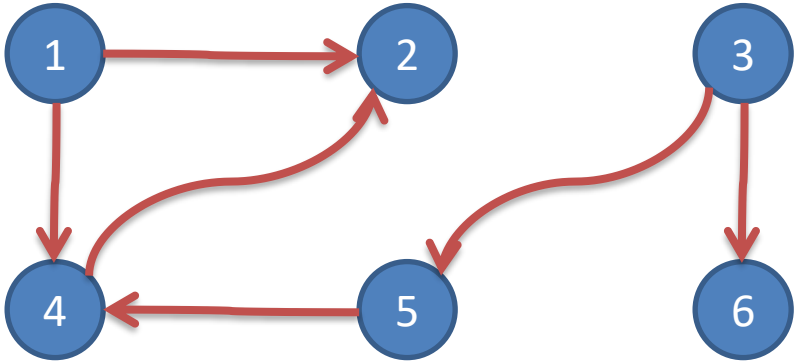


Put all the topological sorted vertices in a line, all edges go from left to right

Topological sort

- How to topological sort a dag?
- Just use **DFS_Traversal**
- The reverse order of DFS_Traversal is a topological sorted order

Topological sort



DFS_Traversal(G):