Graph Traversal

Given a graph G = (V, E), and a distinguished vertex s, how do you visit each vertex v ∈ V exactly once? There are generally two ways for graph traversal.

Breadth-First-Search (BFS):

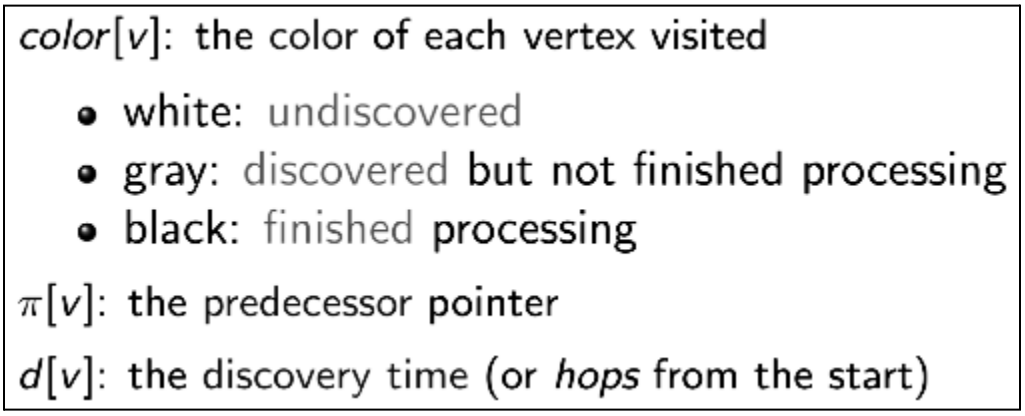
Given a graph G = (V, E) and a distinguished source vertex s, BFS search systematically explores the edges of G to discover every vertex that is reachable from s. It computes the distance from s to each reachable vertex, where the distance to a vertex v equals the smallest number of edges needed to go from s to v.

Breadth-first search (BFS) is named for how it explores a graph by expanding outward uniformly from the starting point. You can imagine it as discovering new nodes in waves that move outward from the source node. Starting from the source, BFS first finds all nodes directly connected to it. Then, it moves to the nodes that are 2 steps away, then those 3 steps away, and continues this process until all reachable nodes are found.

To keep track of nodes, BFS uses a queue that follows First In First Out (FIFO).

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white, and vertices not reachable from the source vertex 's' stay white the entire time. A vertex that is reachable from s is discovered the first time it is encountered during the search, at which time it becomes gray. The queue contains all the gray vertices. Eventually, all the edges of a gray vertex will be explored, so that all of its neighbors will be discovered. Once all of a vertex's edges have been explored, the vertex goes from gray to black.

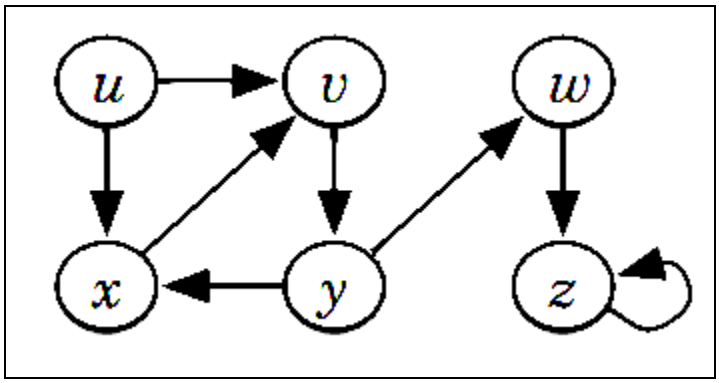
π[v] holds the predecessor or the parent through which we can track the path i.e. the path from A (source) to B (destination). d[v] holds the discovery time or the distance of a node from the source



Pseudo Code:

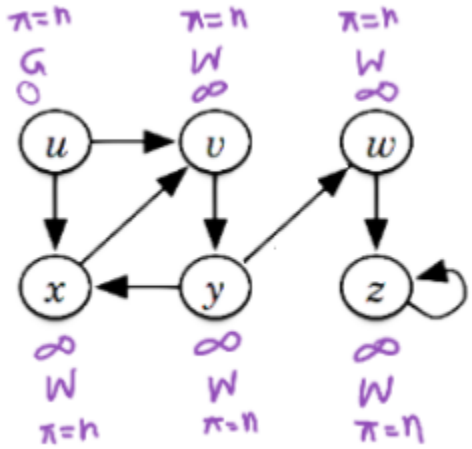
```
BFS(G, s) ▷ G = (V, E)
 1  for each vertex u ∈ V[G]
 2        do color[u] ← WHITE
 3           π[u] ← NULL
 4           d[u] ← ∞
 5  color[s] = GRAY; d[s] = 0
 6  Q = ∅; ENQUEUE(Q, s)
 7  while Q ≠ ∅
 8        do u = DEQUEUE(Q)
 9           for each vertex v ∈ Adj[u]
10                do if color[v] = WHITE
11                      then color[v] = GRAY
12                           d[v] = d[u] + 1
13                           π[v] = u
14                           ENQUEUE(Q, v)
15           color[u] = BLACK
```

Initial Graph:

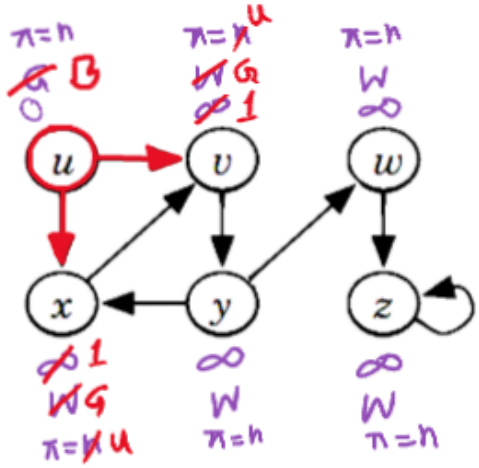


Simulation:

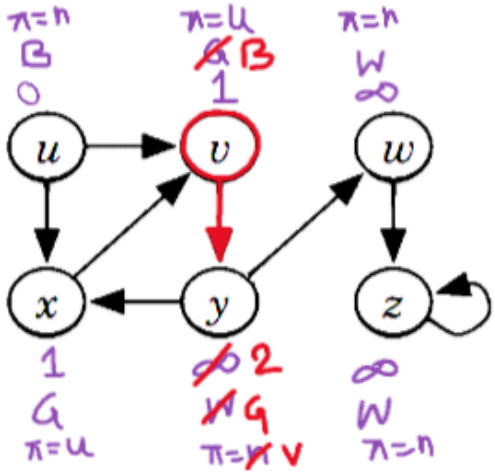
Step 1:



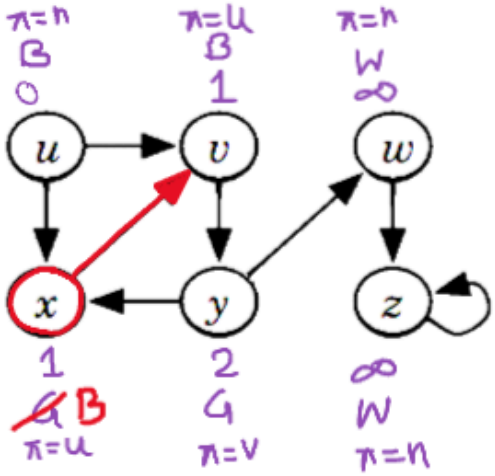
Step 2:



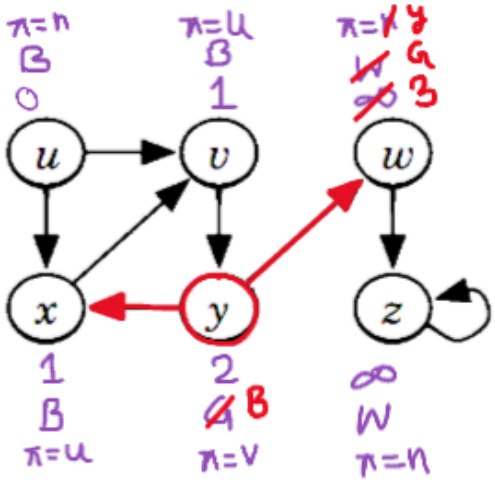
Step 3:



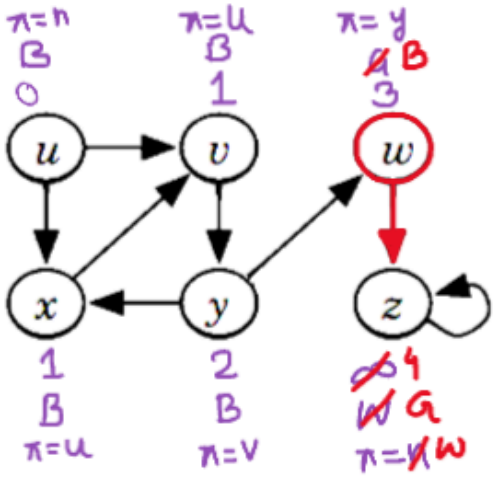
Step 4:



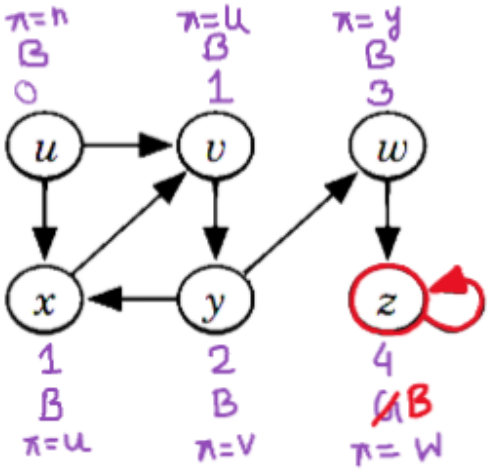
Step 5:



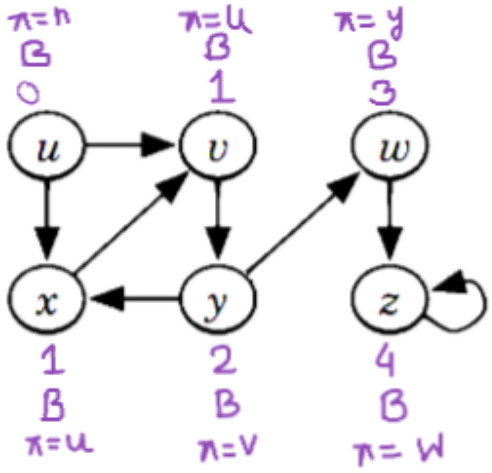
Step 6:



Step 7:



Final Graph:



Time Complexity Analysis:

If we traverse through the entire network, we will have to follow each edge. So the running time is at least O(number of edges). We keep a queue to track progress. Enqueing and dequeuing to the queue take constant time: O(1). Doing this for every vertex will take O(V) total. After dequeuing, we need to search all neighbors of a vertex. Considering we are using an adjacency list, it will take O(E) to scan all the neighbors. Hence, BFS search takes O(V+E).

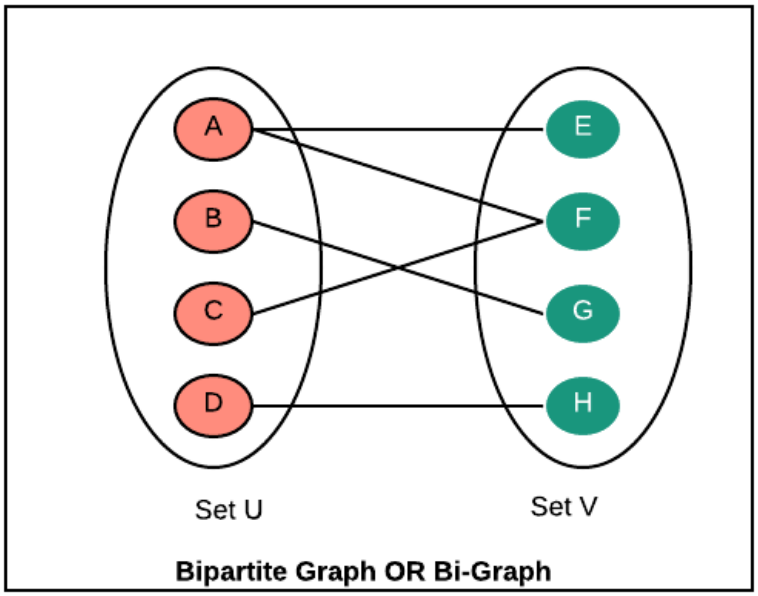
BFS Applications:

- Shortest Path:

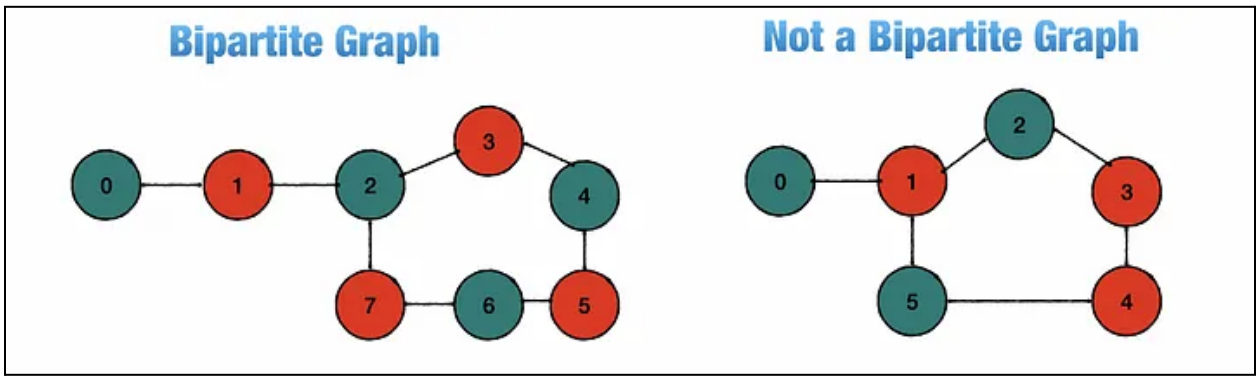
Using BFS, we can find the shortest path in an unweighted graph. Given the above pseudo-code and simulation, we can see how we can keep track of the parents and distances to find the shortest path from node A to node B when the graph has no weights.

- Bipartite/Bicolorable Graph:

A bipartite graph is a graph in which the vertices can be divided into two disjoint sets such that all edges connect a vertex in one set to a vertex in another. There are no edges between vertices in the disjoint sets.



A bipartite graph is a graph that can be colored using 2 colors such that no adjacent nodes have the same color. Any linear graph with no cycle is always a bipartite graph. With a cycle, any graph with an even cycle length can also be a bipartite graph. So, any graph with an odd cycle length can never be a bipartite graph.



Let's see how to implement this using BFS. It is the same as BFS, only changes are that we need to keep track of the color. Let color = -1 (uncolored), 0 (green), and 1 (red). While traversing using BFS, if we find the color of the adjacent node and the current node is the same, we return as False else we assign the opposite color to the adjacent/neighboring nodes

```
Bipartite(G, s):

    for each vertex u ∈ V[G]:
        color[s] = -1
    color[s] = 0
    queue=[ ]
    queue.enqueue(s)
    while queue is not empty:
        u = queue.dequeue()
        for each vertex v ∈ Adj[u]:
            if color[v] == -1:
                color[v] = 1 - color[u]  #Assigning opposite color
                queue.enqueue(v)
            elif color[v] == color[u]:
                return "Not Bipartite"
    return "Bipartite"
```

Depth-First-Search (DFS):

Given a graph G = (V, E) and a distinguished source vertex s, DFS explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered. This process continues until all vertices that are reachable from the original source vertex have been discovered. If any undiscovered vertices remain, then DFS selects one of them as a new source, repeating the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

DFS colors the vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree so that these trees are disjoint. Besides creating a depth-first forest, DFS also timestamps each vertex. Each vertex v has two timestamps: the first timestamp v:d records when v is first discovered (and grayed), and the second timestamp v:f records when the search finishes examining v's adjacency list (and blackens v). These timestamps provide important information about the structure of the graph and are generally helpful in reasoning about the behavior of DFS.

$color[v]$: the color of each vertex visited

- white: undiscovered
- gray: discovered but not finished processing
- black: finished processing

$\pi[v]$: the predecessor pointer

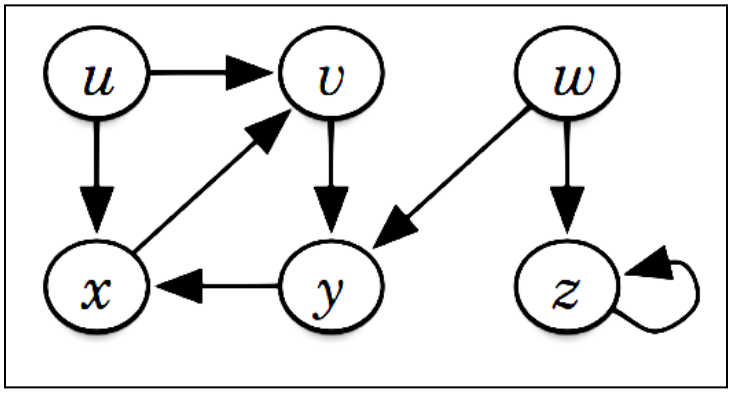
$d[v]$: the discovery time (or *hops* from the start)

$f[v]$: the finish time – when processing of v and all its descendants have finished.

Pseudo Code:

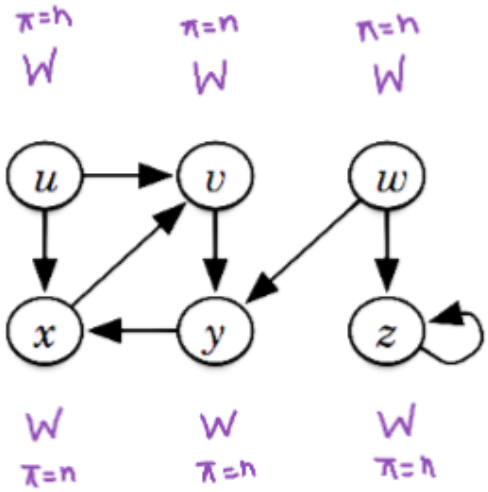
```
DFS(G) ▷ G = (V, E)
1  for each vertex u ∈ V[G]
2        do color[u] ← WHITE
3           π[u] ← NULL
4  time ← 0
5  for each vertex u ∈ V[G]
6        do if color[u] = WHITE
7              then DFS-VISIT(u)

DFS-VISIT(G, u)
1  color[u] ← GRAY
2  time ← time +1
3  d[u] ← time
4  for each vertex v ∈ Adj[u]
5        do if color[v] = WHITE
6              then π[v] ← u
7                   DFS-VISIT(v)
8  color[u] ← BLACK
9  time ← time +1
10 f[u] ← time
```

Initial Graph:

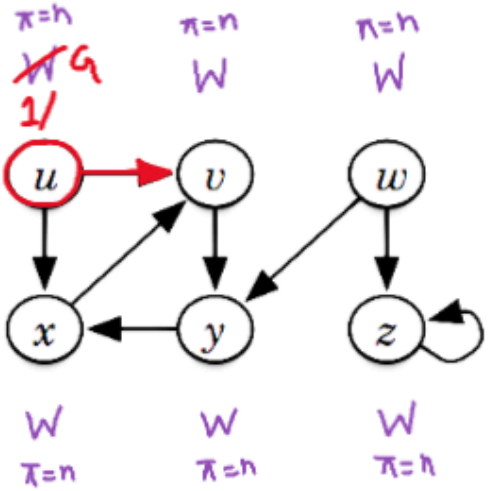


Simulation:

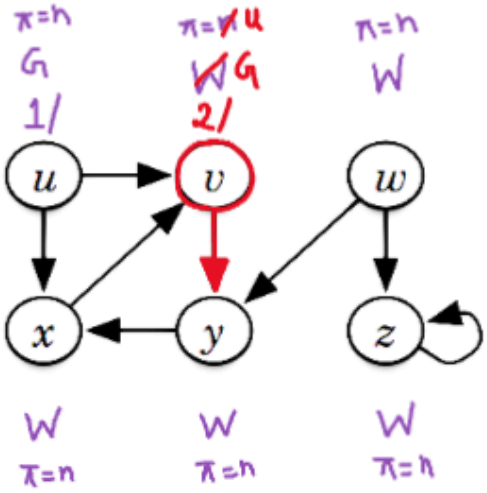
Step 1:



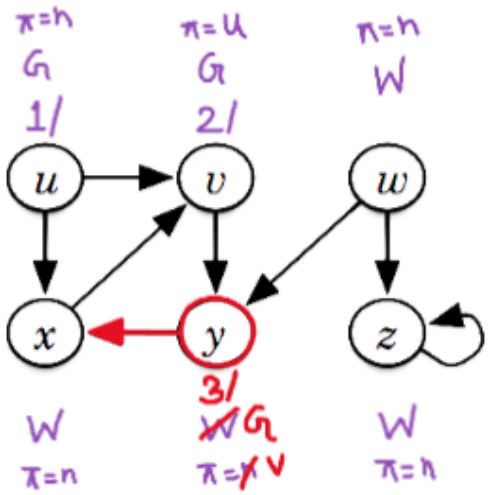
Step 2:



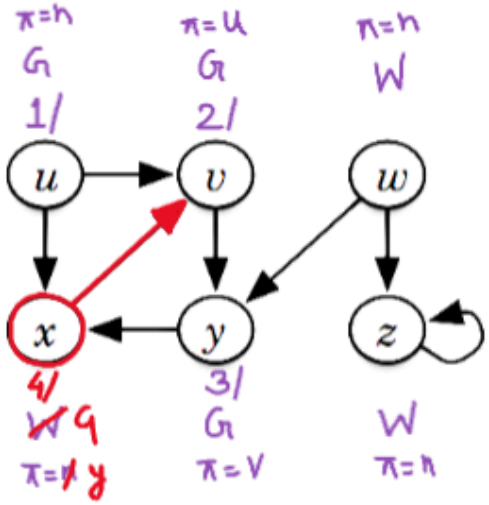
Step 3:



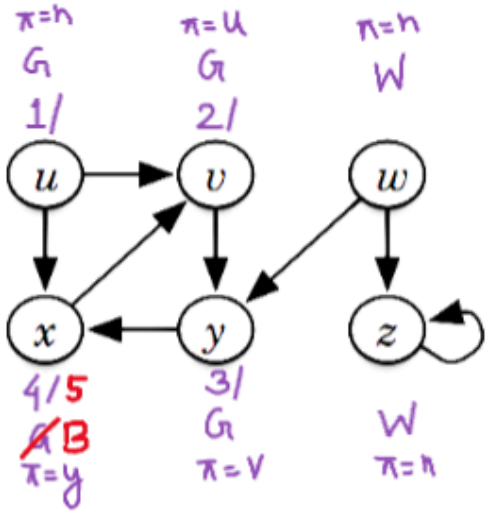
Step 4:



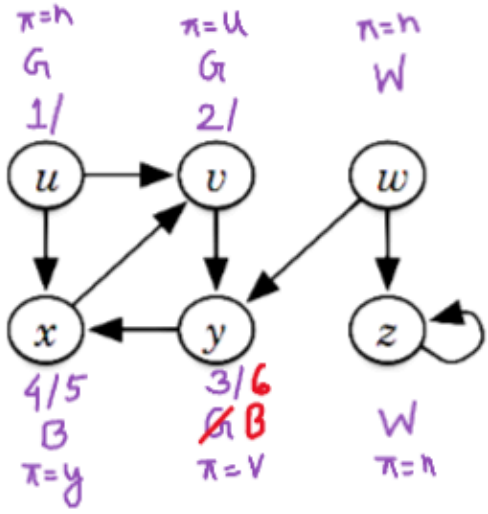
Step 5:



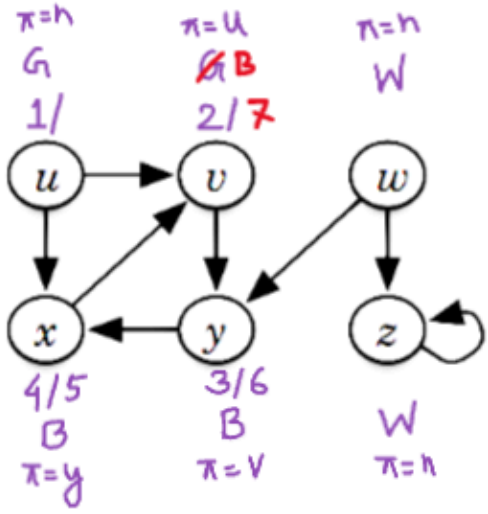
Step 6:



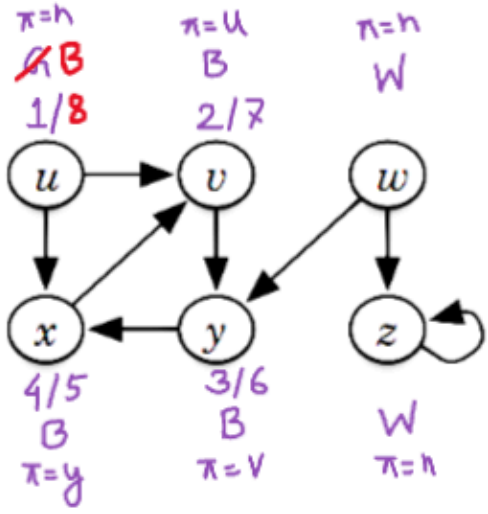
Step 7:



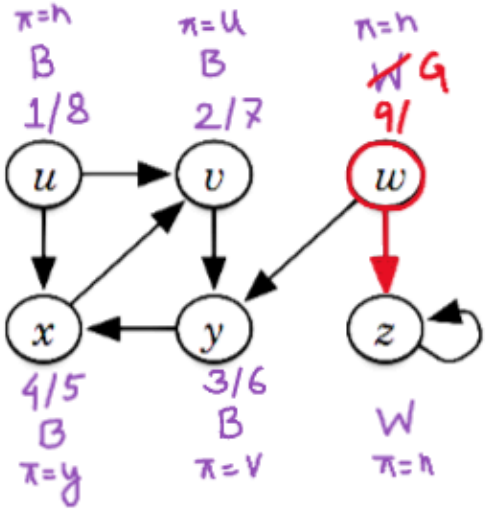
Step 8:



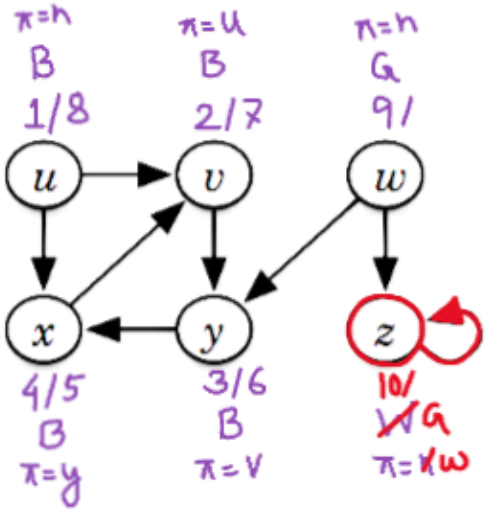
Step 9:



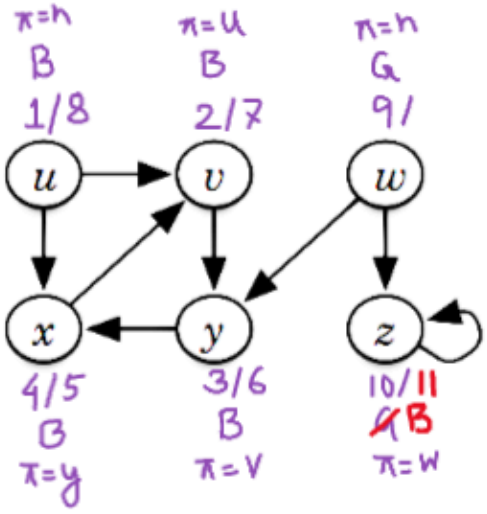
Step 10:



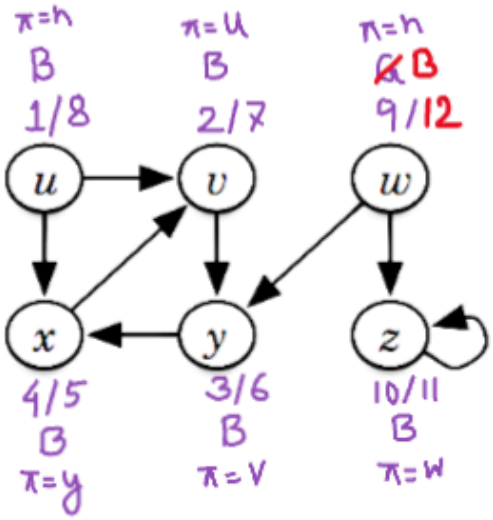
Step 11:



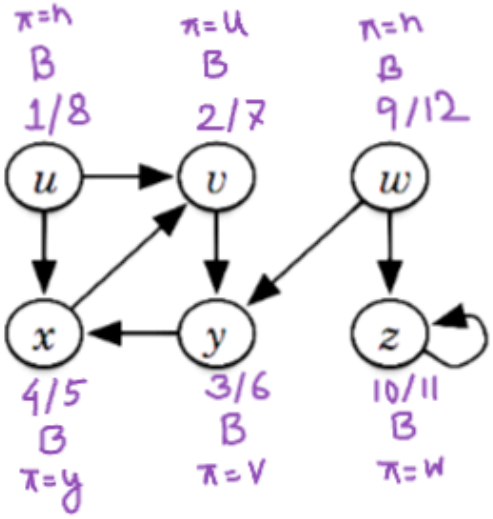
Step 12:



Step 13:



Final Graph:



Time Complexity Analysis:

For each vertex, DFS-VISIT is called, which takes O(V) time, exclusive of the time to execute the calls to DFS-VISIT. In DFS-VISIT, we need to search all neighbors of a vertex. Considering we are using an adjacency list as a graph, it will take O(E) to scan all the neighbors. Hence, DFS search takes O(V+E).

Stack and Queue in BFS and DFS:

```
BFS(G, s):

  visited[s] = True
  queue.enqueue(s)
  while queue is not empty:
    u = queue.dequeue()
    print(u)
    for each vertex v ∈ Adj[u]:
      if not visited[v]:
        visited[v] = True
        queue.enqueue(v)
```

```
DFS(G, s):

  visited[s] = True
  stack.push(s)
  while stack is not empty:
    u = stack.pop()
    print(u)
    for each vertex v ∈ Adj[u]:
      if not visited[v]:
        visited[v] = True
        stack.push(v)
```

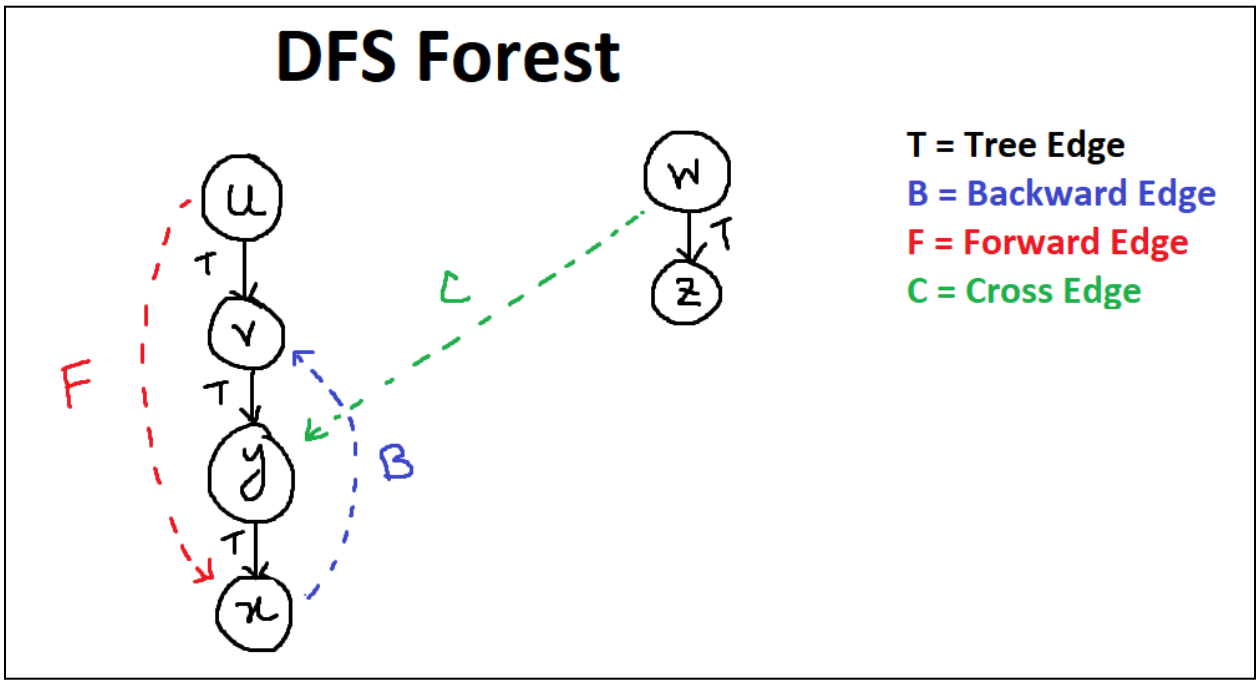
DFS Applications:

- Edge Classification:

After performing DFS, there are 4 types of edges:

1) Tree Edge - edge generated during DFS
2) Back Edge - edge from a node to one of its ancestors in a DFS tree
3) Forward Edge - edge from a node to one of its descendants in a DFS tree
4) Cross Edge - edge connecting nodes from different branches

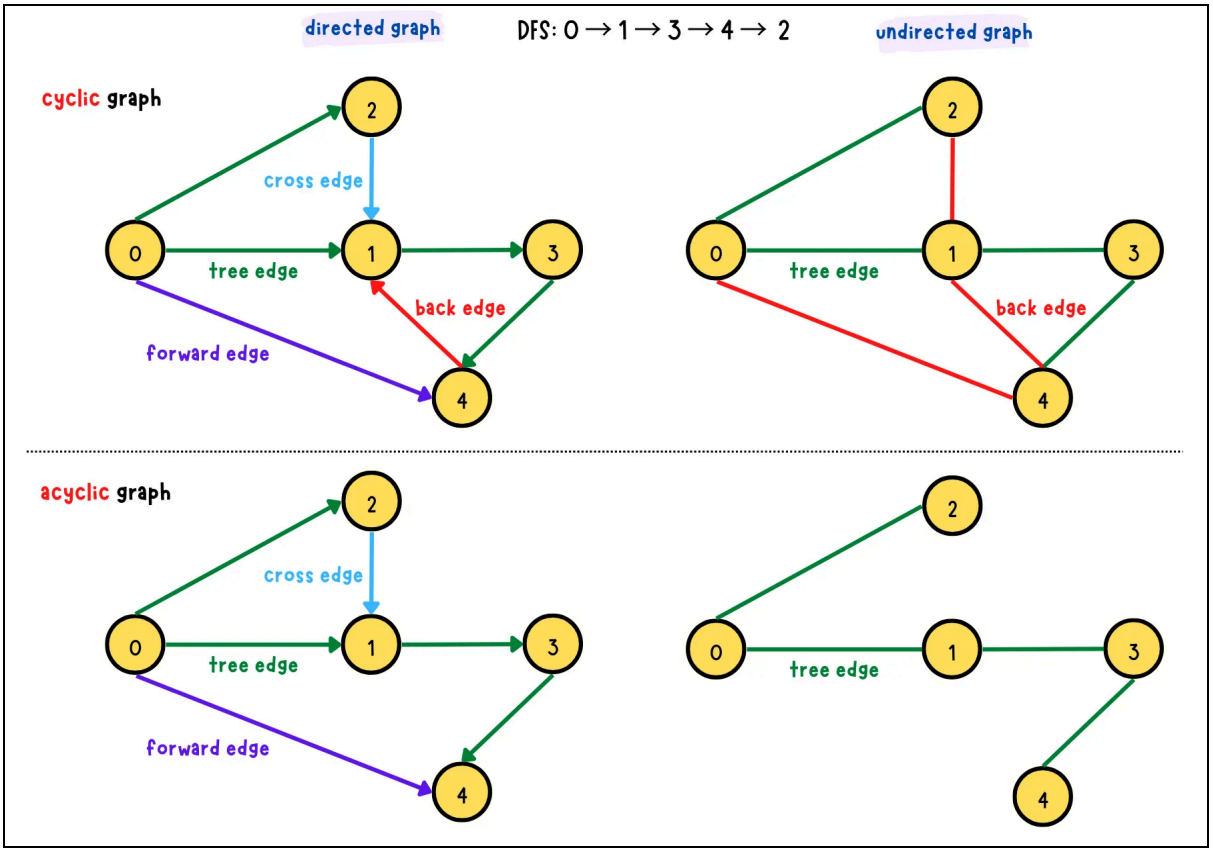
After applying DFS on the previous graph, a DFS forest will be generated:



For each edge (u –> v) in the graph:

Tree/Foward: u.start < v.start < v.finish < u.finish
Back: v.start < u.start < u.finish < v.finish
Cross: v.start < v.finish < u.start < u.finish

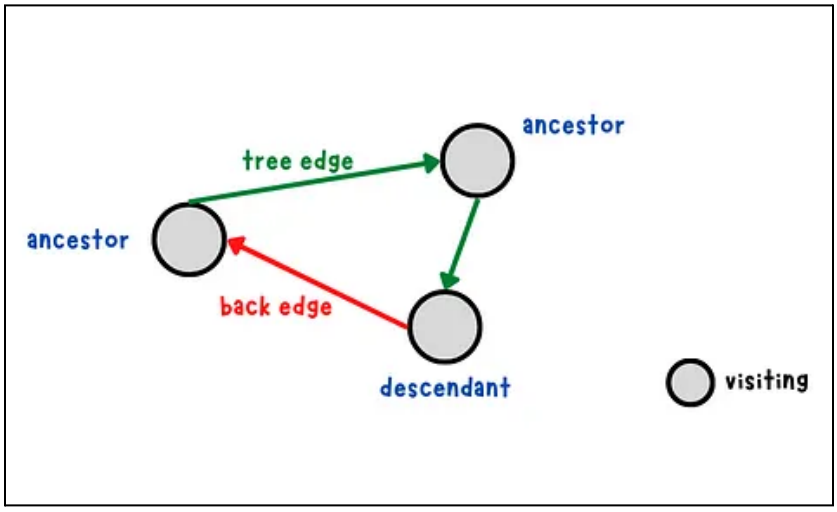
In a DFS of an undirected graph, every edge of G is either a tree edge or a back edge.



- Cycle Detection:

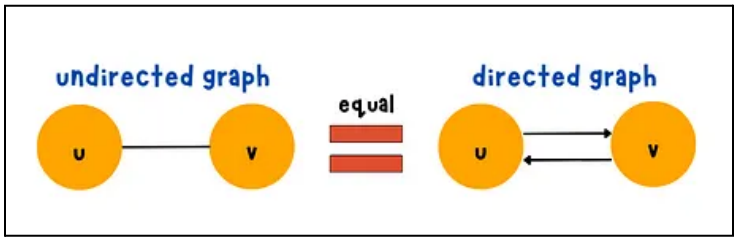
i) Detect Cycle in a Directed Graph:

White-Grey-Black coloring helps to detect a cycle. If a grey node has a descendant which is grey, it has a back edge, and hence, it is a cyclic graph.



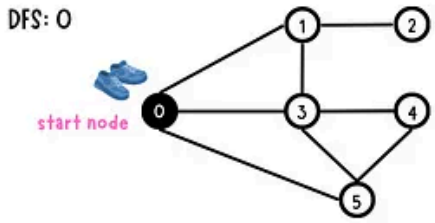
ii) Detect Cycle in an Undirected Graph:

White-Grey-Black coloring alone does not work in an undirected graph to detect a cycle as the edges are bidirectional.



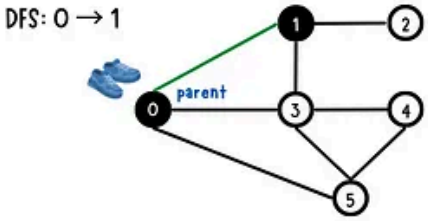
Using White-Grey-Black coloring would consider u and v to be a cycle. To avoid finding a cycle between two nodes connected by an edge, we need to remember a node's parent to prevent going back to the parent.

Start DFS traversal at the start node. Keep track of visited nodes by coloring it grey and the parent node of each node to avoid backtracking to them. If a grey node that is not a parent node is encountered, a cycle is detected.



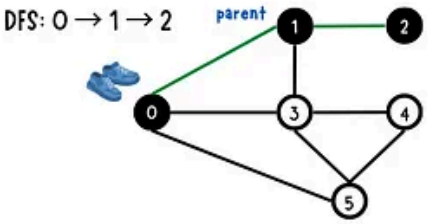
node 0 is not visited → visit node 0 → mark node 0 as visited

visited = {0}
parent = None



explore node 0's descendant: node 1
node 1 is not visited → visit node 1 → mark node 1 as visited

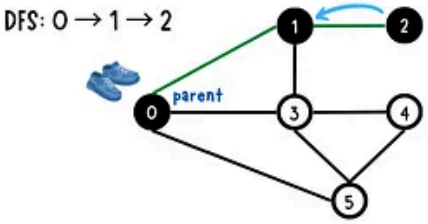
visited = {0, 1}
parent= 0



explore node 1's descendant: node 0
node 0 == parent → skip

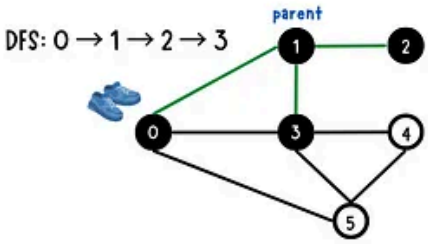
explore node 1's descendant: node 2
node 2 is not visited → visit node 2 → mark node 2 as visited

visited = {0, 1, 2}
parent = 1



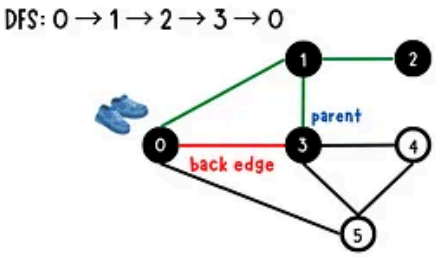
explore node 2's descendant: node 1
node 1 == parent → skip
→ return to its parent, node 1

visited = {0, 1, 2}
parent = 0



explore node 1's descendant: node 3
node 3 is not visited → visit node 3 → mark node 3 as visited

visited = {0, 1, 2, 3}
parent = 1



explore node 3's descendant: node 0
node 0 has been visited → a cycle is detected