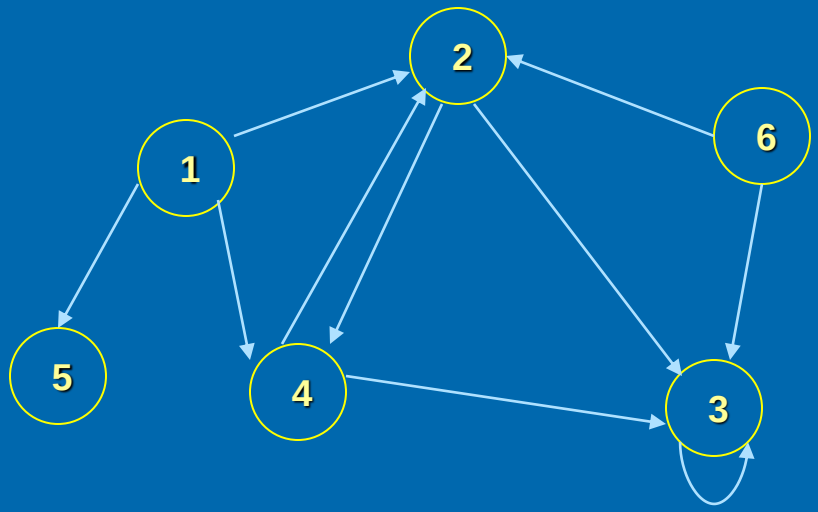
# Lecture # 19

# Directed Graphs ( Digraph )

- Directed graphs differ from trees in that they need not have a root node and there may be several paths from one vertex (element/node) to another.

- As a mathematical structure, a **directed graph** or **digraph** , like a tree consists of finite set of elements called vertices or nodes, together with finite set of directed arcs or edges that connect pair of vertices. or
- A finite set of elements called nodes or vertices , and a finite set of directed arcs or edges that connect pair of nodes.
- For example, a directed graph having six vertices numbered **1,2,3,4,5,6** and ten directed arcs joining vertices 1 to 2 , 1 to 4, 1 to 5, 2 to 3, 2 to 4, 3 to itself, 4 to 2, 4 to 3, 6 to 2 , 6 to 3, can be pictured as


2
1
6
5
4
3

- Trees are special kinds of directed graphs and are characterized by the fact that one of their nodes, the root , has no incoming arcs and every other node can be reached from the root by a unique path, i.e., by following one and only one sequence of consecutive arcs.

- In the preceding digraph, vertex 1 is “rootlike” node having no incoming arcs, but there are many different paths from vertex 1 to various other nodes. So that is not tree. For example , to vertex 3.

- The following description of a digraph as an ADT (abstract data type) includes some of the most common operations on digraphs.

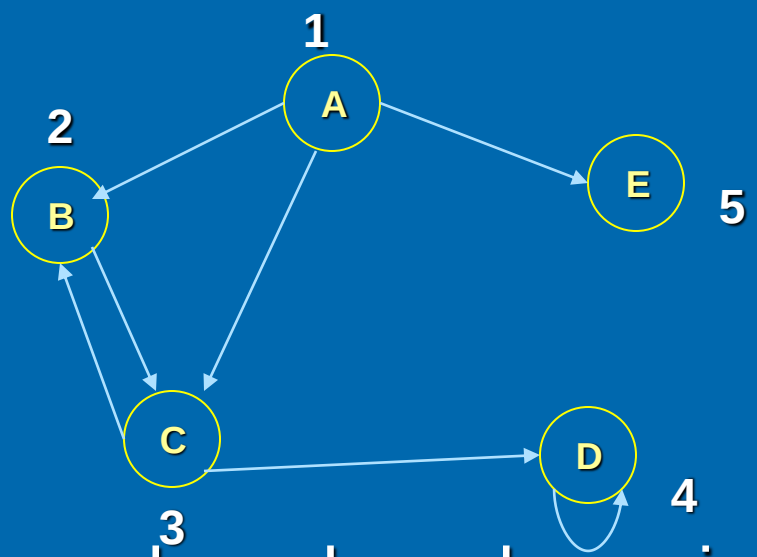
1. Construct an empty directed graph.
2. Check if it is empty
3. Destroy a directed graph
4. Insert a new node
5. Insert a directed edge between two existing nodes or from a node to itself
6. Delete a node and all directed edges to or from it.
7. Delete directed edge between two existing nodes
8. Search for a value in a node, starting from a given node.

# Adjacency Matrix Representation

- There are several common ways of implementing a directed graph using data structures already known to us. One of these is the adjacency matrix of the digraph.

- To construct it, we first number the vertices of the digraph 1,2…. n;  the adjacency matrix is the n x n matrix adj, in which the entry in row i and column j is 1 ( or true ) if vertex j is adjacent to vertex i ( i.e. if there is a directed arcs from vertex i to vertex j) , and is 0 (or false) otherwise.

# For example, the adjacency matrix for the digraph




- With nodes numbered as shown is

$$\text{Adj} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

- For a weighted graph in which some cost or weight is associated with each arc , the cost of an arc form vertex i to vertex j is used instead of 1 in the adjacency matrix.

- For example, with this representation it is easy to determine the in-degree and out-degree of any vertex which are the number of edges coming into or going out from that vertex, respectively. The sum of the entries in row i of the adjacency matrix is obviously the out-degree of the $i^{th}$ vertex and the sum of entries in the $i^{th}$ column is its in-degree.

- There are some deficiencies in this approach.
- One that it does not store the data items in the vertices of digraph., the letters A,B,C,D, and E. but we can find solution by having an extra array which store the inside data of nodes of Digraph as follows.

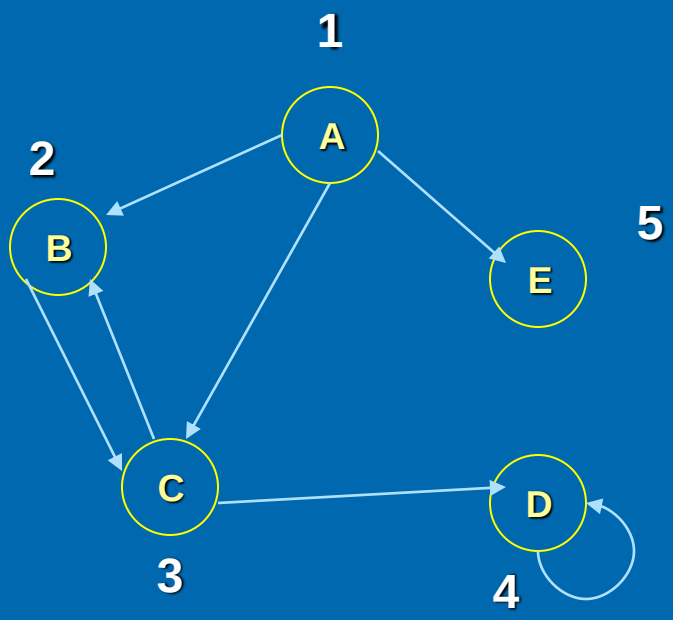
$$\text{Adj} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad \text{Data} = \begin{pmatrix} A \\ B \\ C \\ D \\ E \end{pmatrix}$$

- Another problem of adjacency matrix is that the matrix is sparse i.e. it has many 0 entries and thus considerable space is wasted of memory.

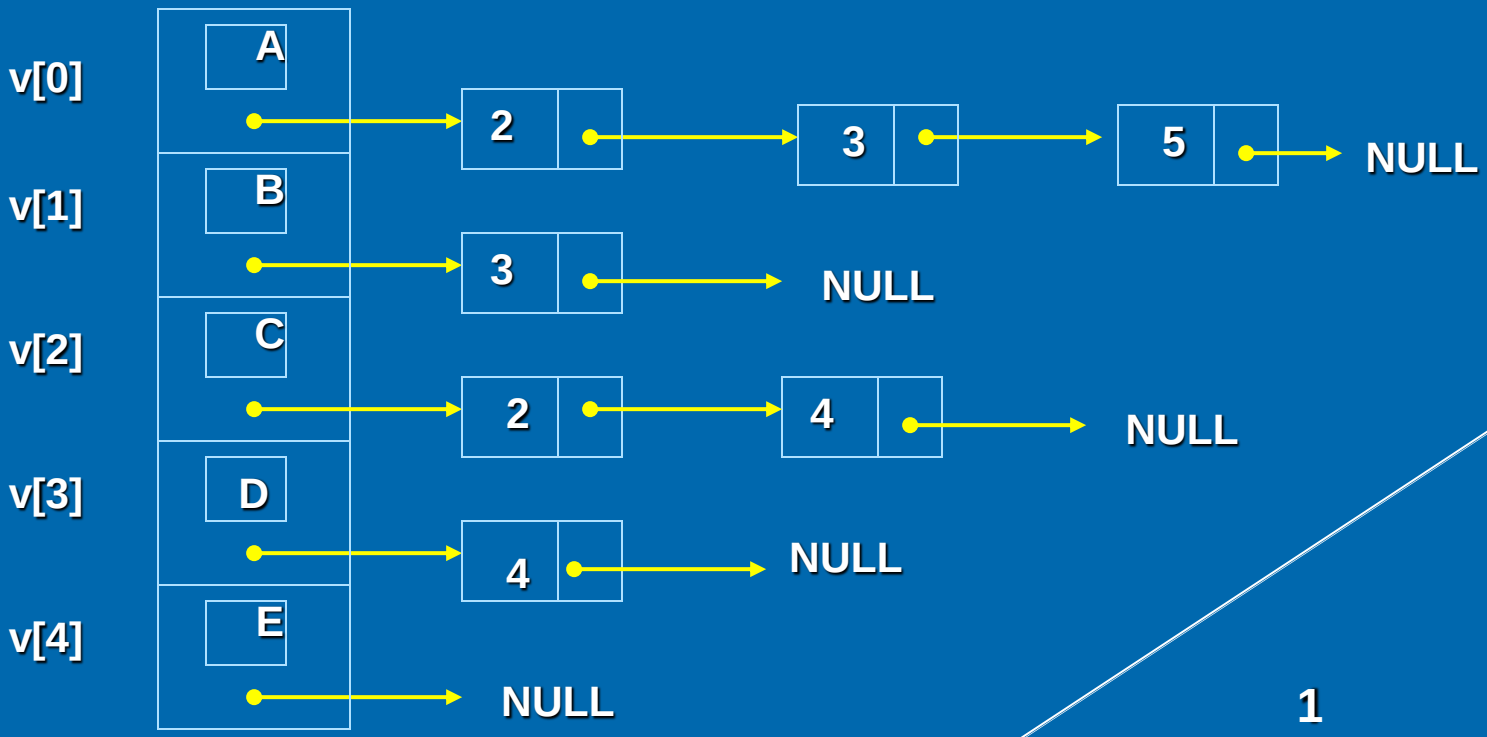
# Adjacency List Representation

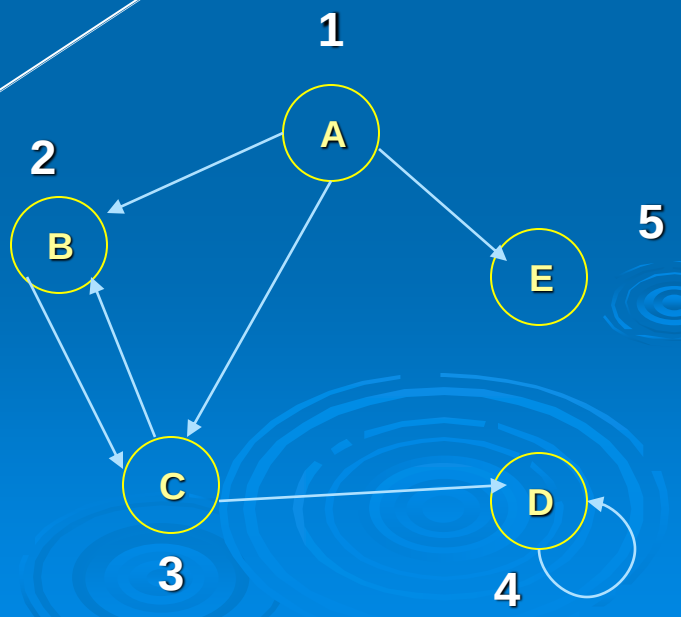
- we can eliminate both the problems by having Adjacency List Representation like as follows.

- Consider the Digraph




**For above the adjacency list representation is as follows**


v[0]
A
2
3
5
NULL
v[1]
B
3
NULL
v[2]
C
2
4
NULL
v[3]
D
4
NULL
v[4]
E
NULL


1
A
2
B
5
E
C
D
3
4

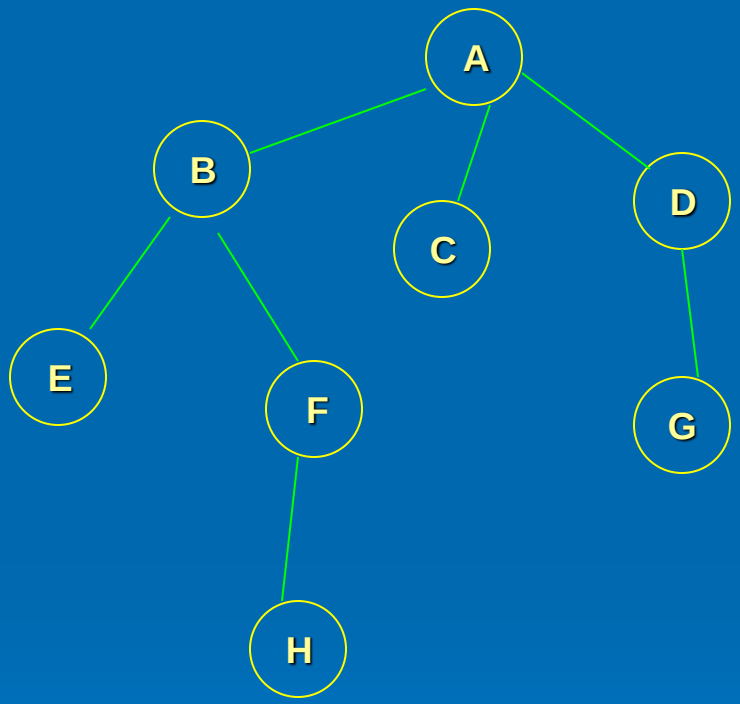
# Searching and Traversing Digraphs

- Traversal of a tree is always possible if we begin at the root, because every node is reachable via a sequence of consecutive arcs.

- However, in a general digraph there may not be node from which every other node can be reached and thus it may not possible to traverse the entire graph regardless of start vertex.

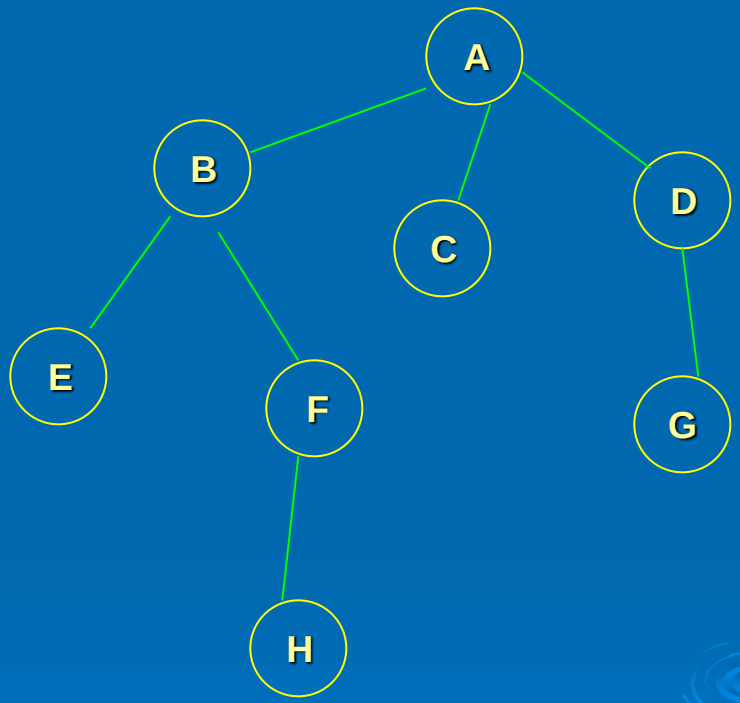
- Two standard methods of searching for vertices in digraph are Depth-first Search and Breadth-first Search.

- To illustrate DFS consider the following tree.



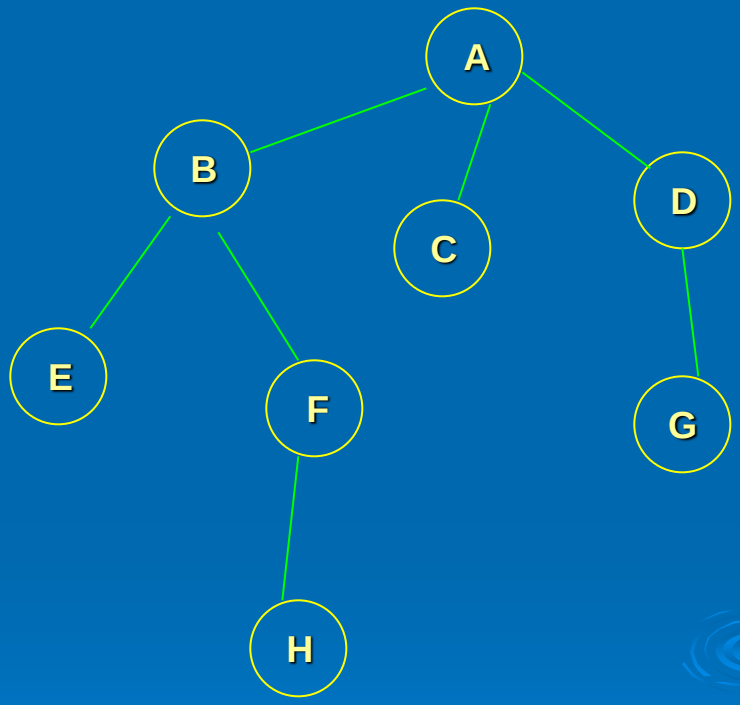
- Using DFS, we will go into depth of tree first at every node then move breath wise as follows

- A B E
- A B E F H
- A B E F H C
- A B E F H C D G



- Using BFS, we will go into Breath of tree first at every node then move Depth wise as follows

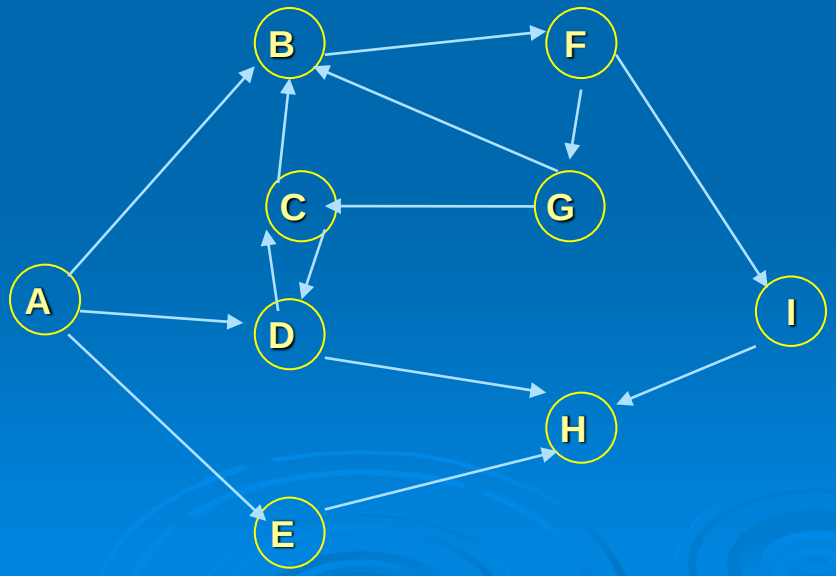
- A B C D
- A B C D E F G
- A B C D E F G H



# Depth-First Search

- A depth-first search of a general directed graph from a given start vertex is  similar to that for trees. We visit the start vertex and then follow directed arcs as “deeply” as possible to visit the vertices reachable from it that have not already been visited, backtracking when necessary.

- For example, a DFS of a following digraph beginning at vertex A might first visit vertices A, B, F, I and H. we then backtrack to I, the last node visited on this search path, but no unvisited nodes are reachable from it.
- So we back track to F and from there visit G, from which we can reach vertices C and D ( as well as B F  I and H but they are already visited ).
- We backtrack to C but no unvisited node are reachable form it, so we backtrack G.
- Finding no unvisited nodes reachable form it, we backtrack to F and then to B; there are no unvisited nodes reachable from  either of these vertices.



- Finally, we backtrack to A, form which we can reach the last unvisited node E.
- So we have visited the vertices in the order

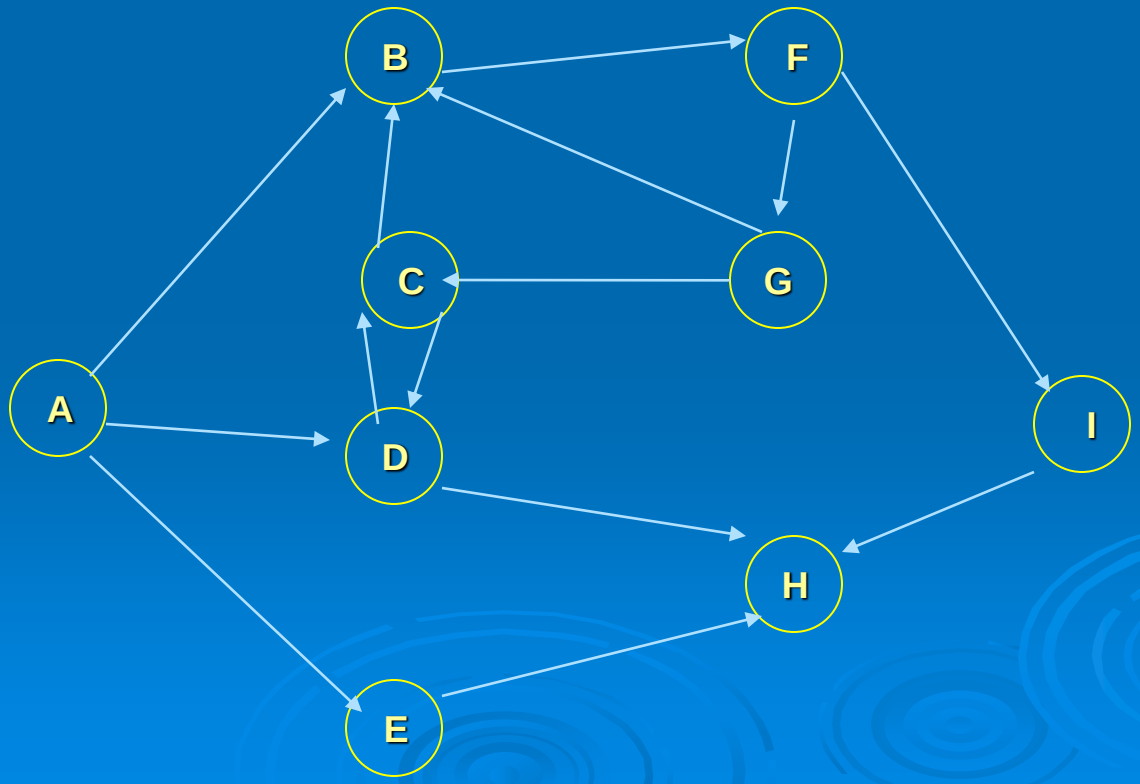
  A B F I H G C D E
- If we start DFS from some vertices, we may not be able to reach all of the other vertices. For example, a DFS starting at B will visit the vertices

  B F I H G C D

  but we cannot reach vertices A and E.

# Breath-First Search

- In a Breath-First Search of the preceding directed graph is as follows.



- Beginning at A , we visit A and then all those vertices adjacent to it

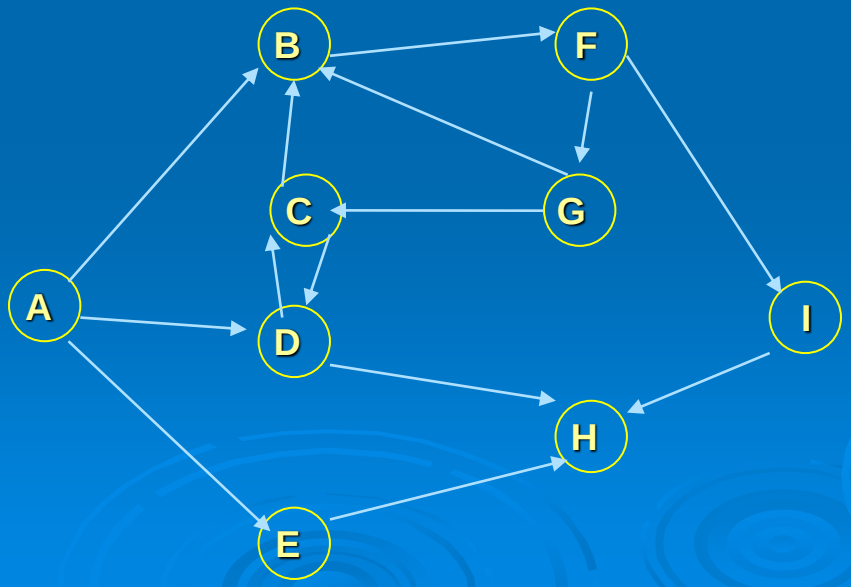
A, B, D, E

we then select B , the first vertex adjacent to A, and visit all unvisited vertices adjacent to it.

A, B, D, E, F , C , H

And then for E, the last vertex adjacent to A, for which only adjacent vertex is H, which was already visited.

- Having finished with B, D , and E the nodes adjacent to A we now repeat this approach for the unvisited vertices adjacent to them: F, then C, and then H.
- G and I are adjacent to F so we visit them

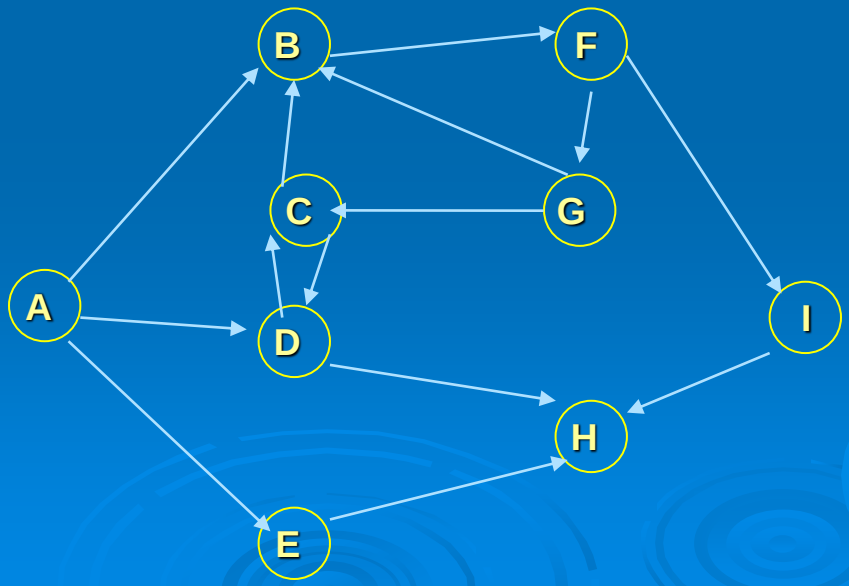


A, B, D, E, F , C , H, G , I

- A, B and D are adjacent to C but these vertices have already been visited, and no vertices are adjacent to H.
- Since all of the vertices have been visited, the search terminates.
- As was true for Depth-first Search, a Breath-first Search from some vertices may fail to locate all vertices. For example a Breath-first Search from B might first visit B and F, then G and I, followed by H and C, and finally D:

B F G I H C D

- Because A and E are not reachable from B, the search terminates.



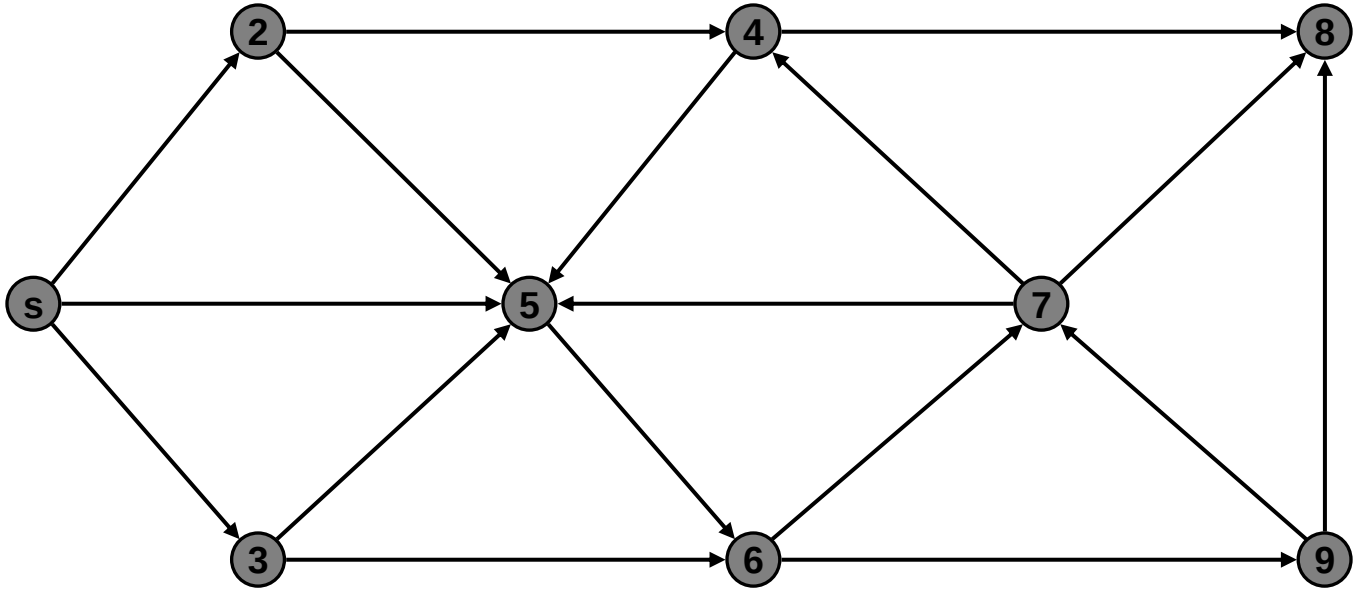
- Two common types of graph traversals are **Depth First Search** (DFS) and **Breadth First Search** (BFS).

- DFS is implemented with a stack, and BFS with a queue.

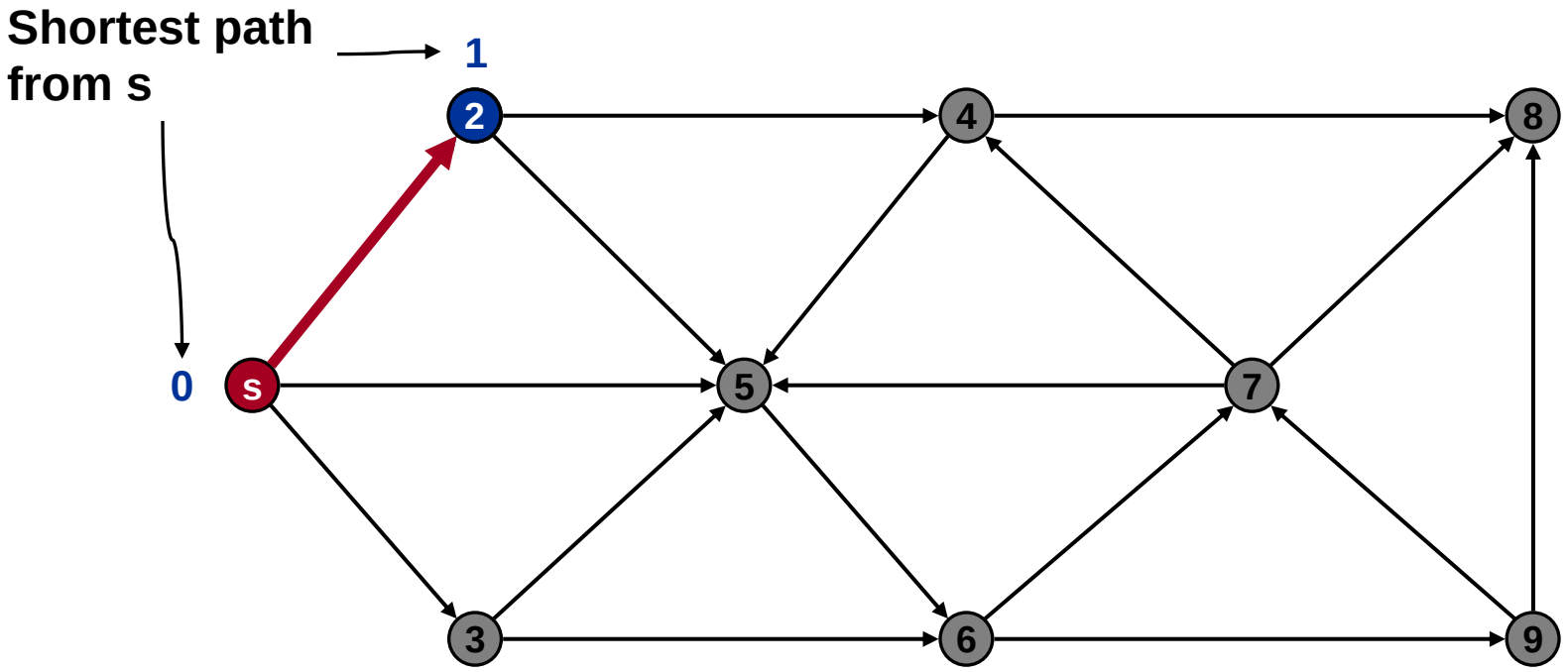
- The aim in both types of traversals is to visit each vertex of a graph *exactly once*.

- In DFS, you follow a path as far as you can go before backing up. With BFS, you visit all the neighbors of the current node before exploring further a nodes in the graph.

# Breadth First Search



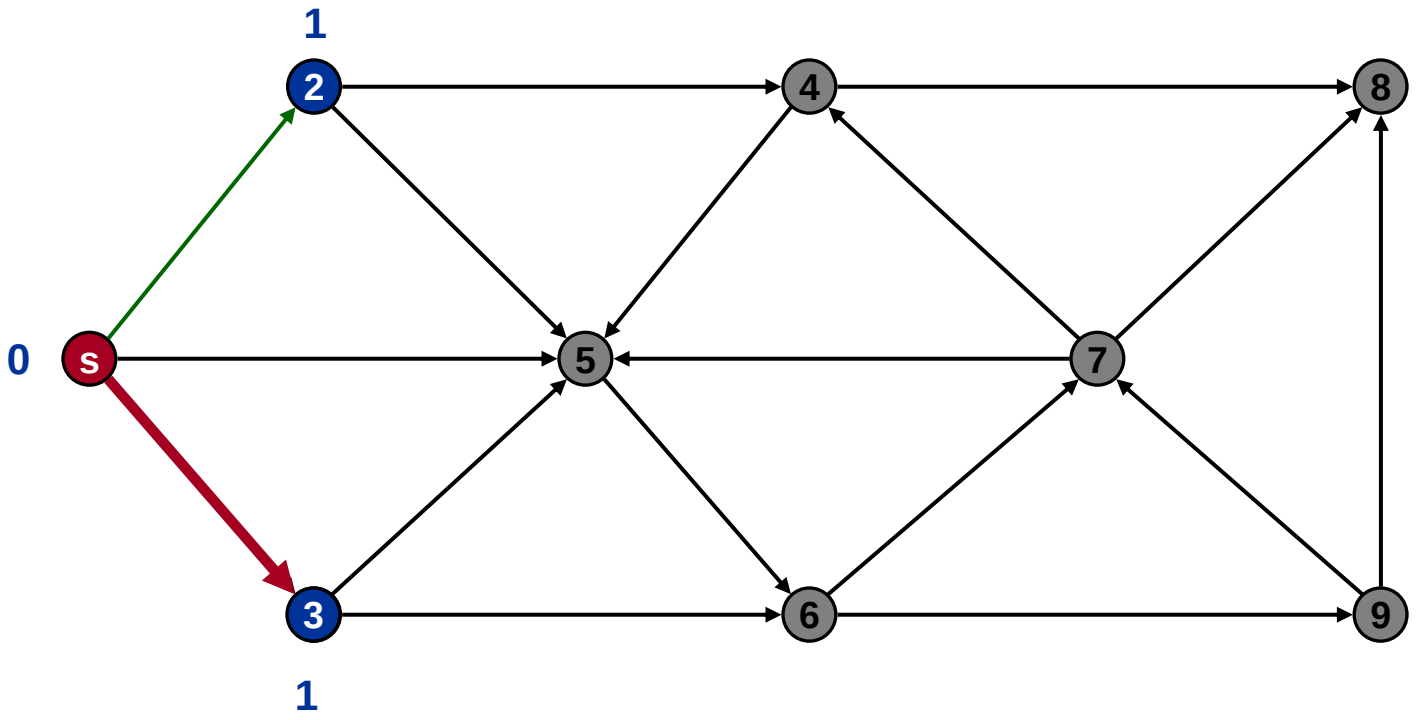
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: s**

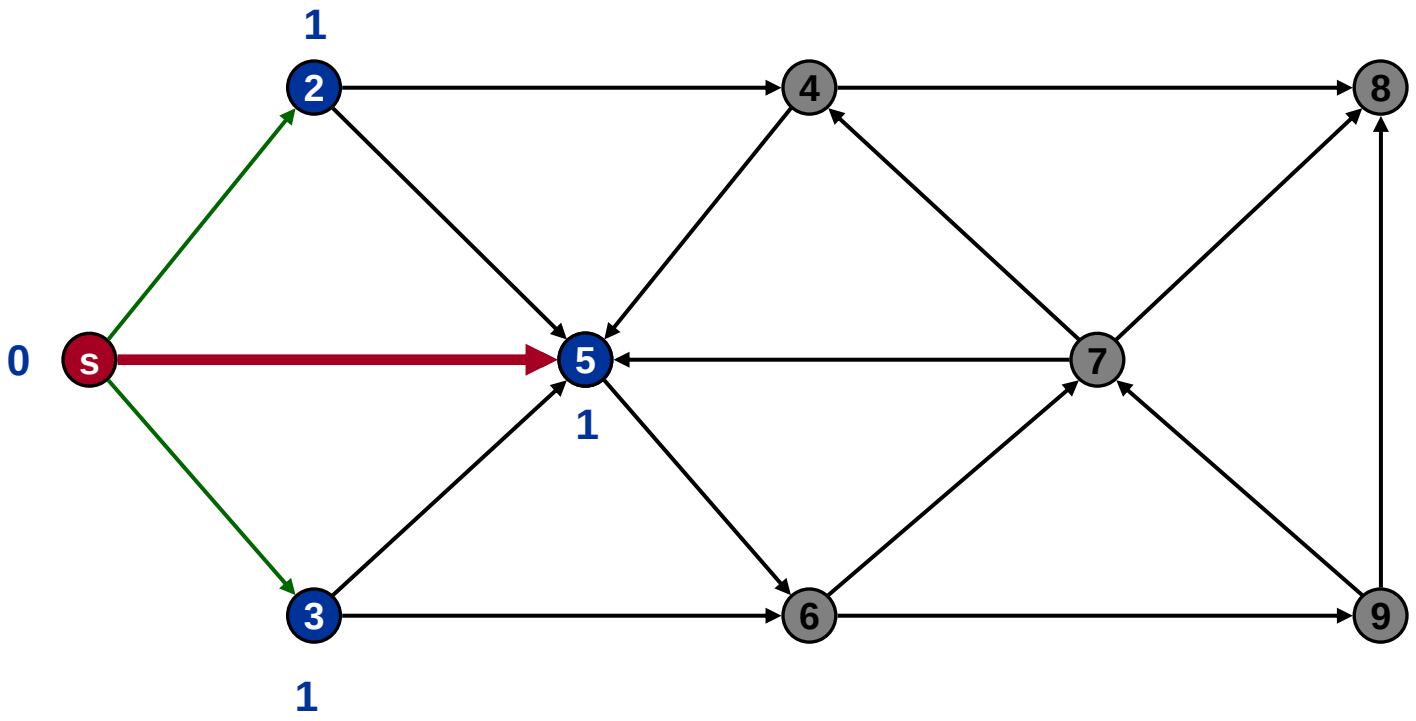
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: s 2**

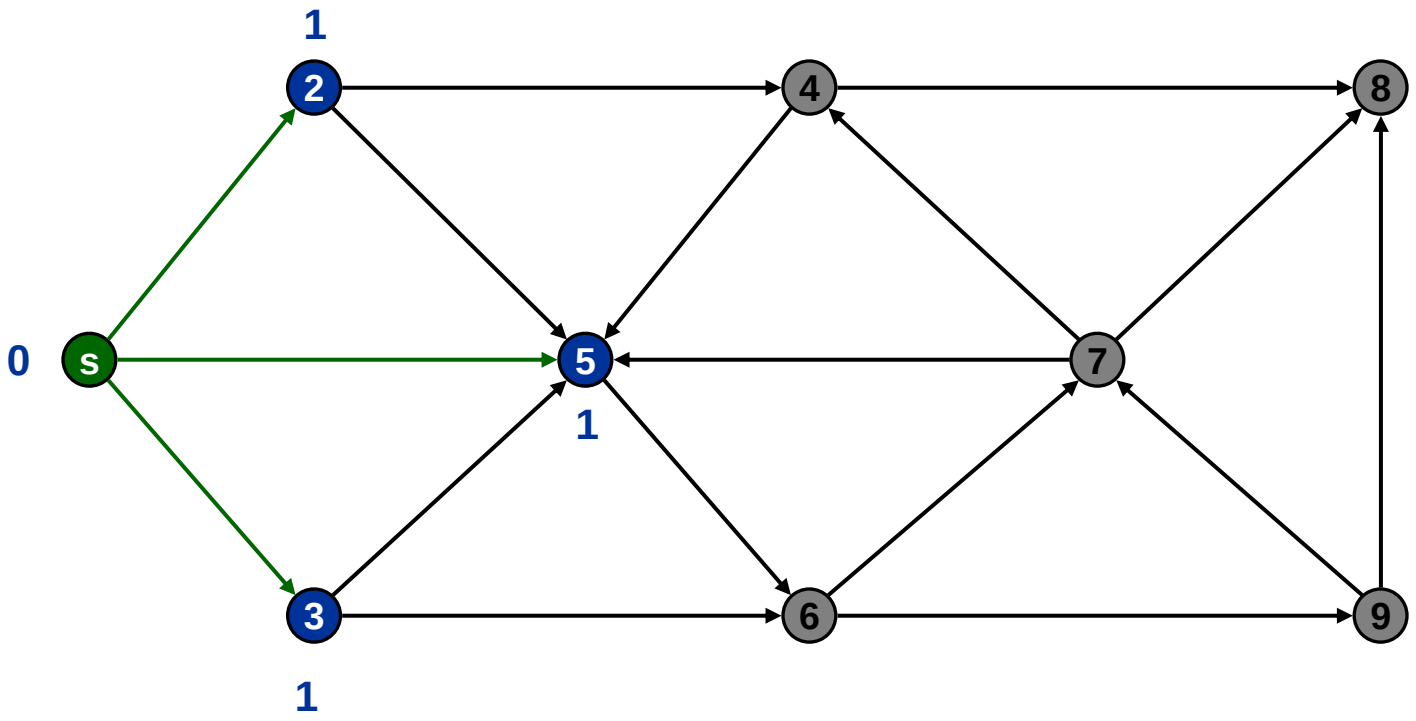
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: s 2 3**

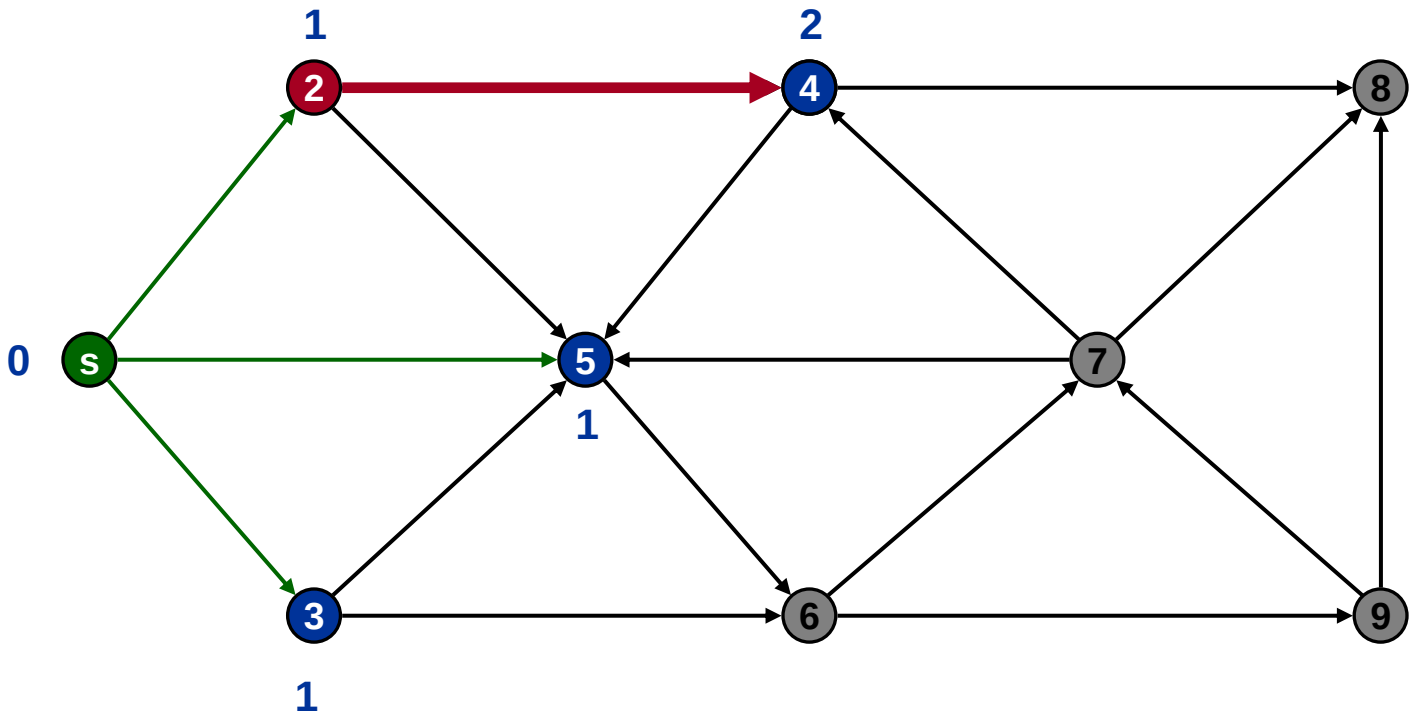
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: 2 3 5**

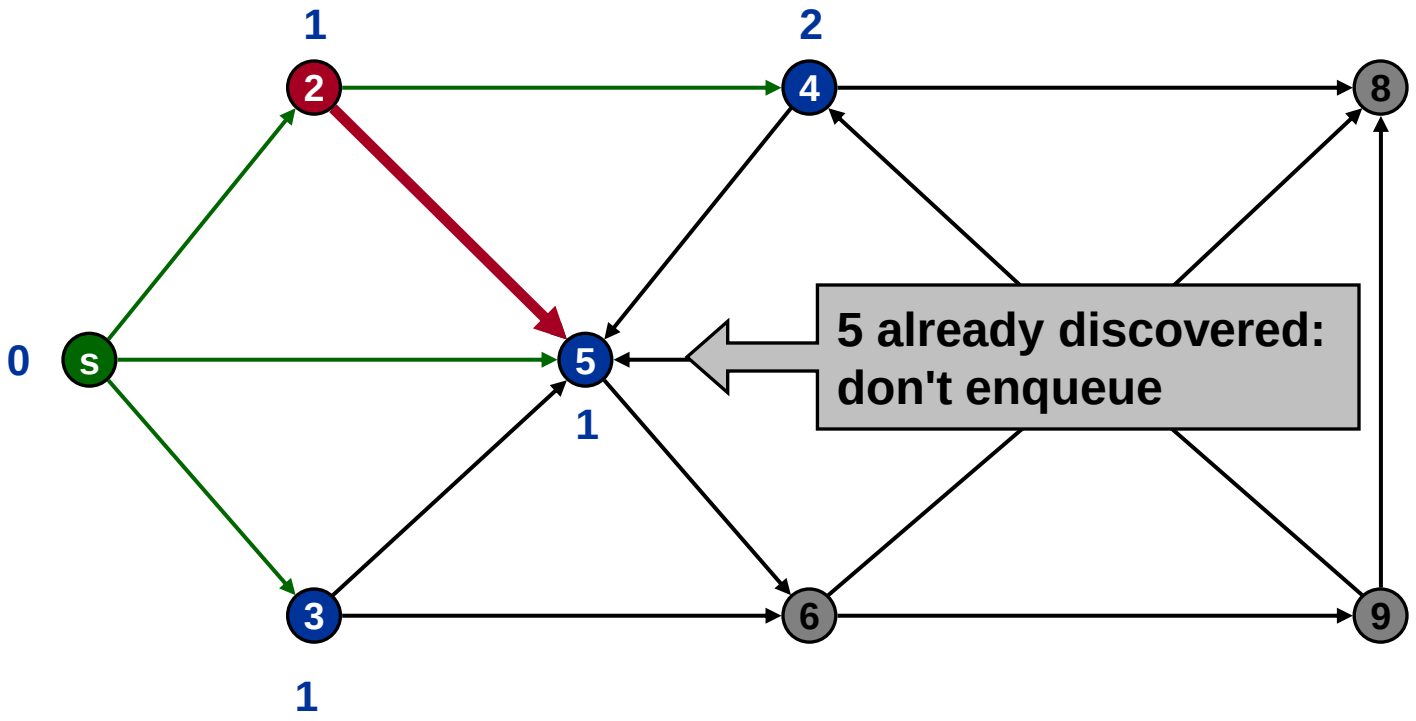
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: 2 3 5**

# Breadth First Search




Undiscovered

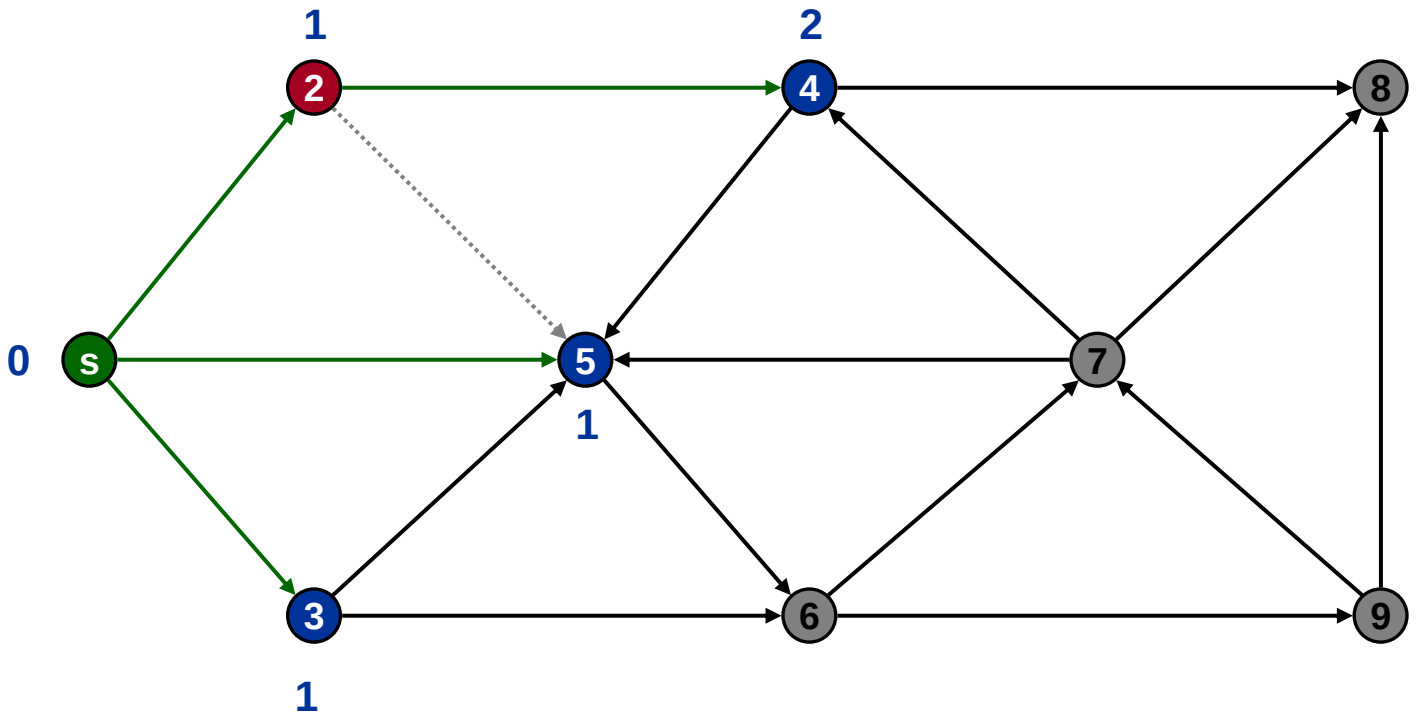
Discovered

Top of queue

Finished

Queue: 2 3 5 4

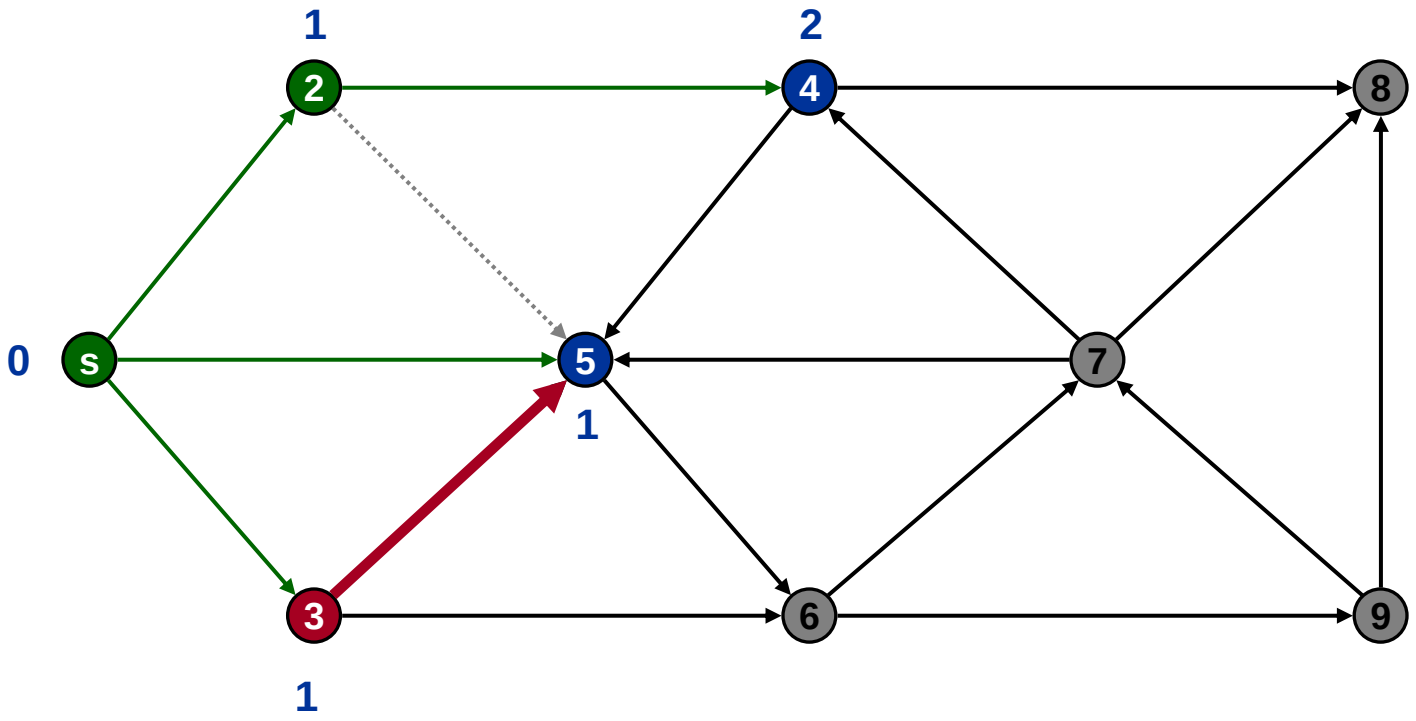
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: 2 3 5 4**

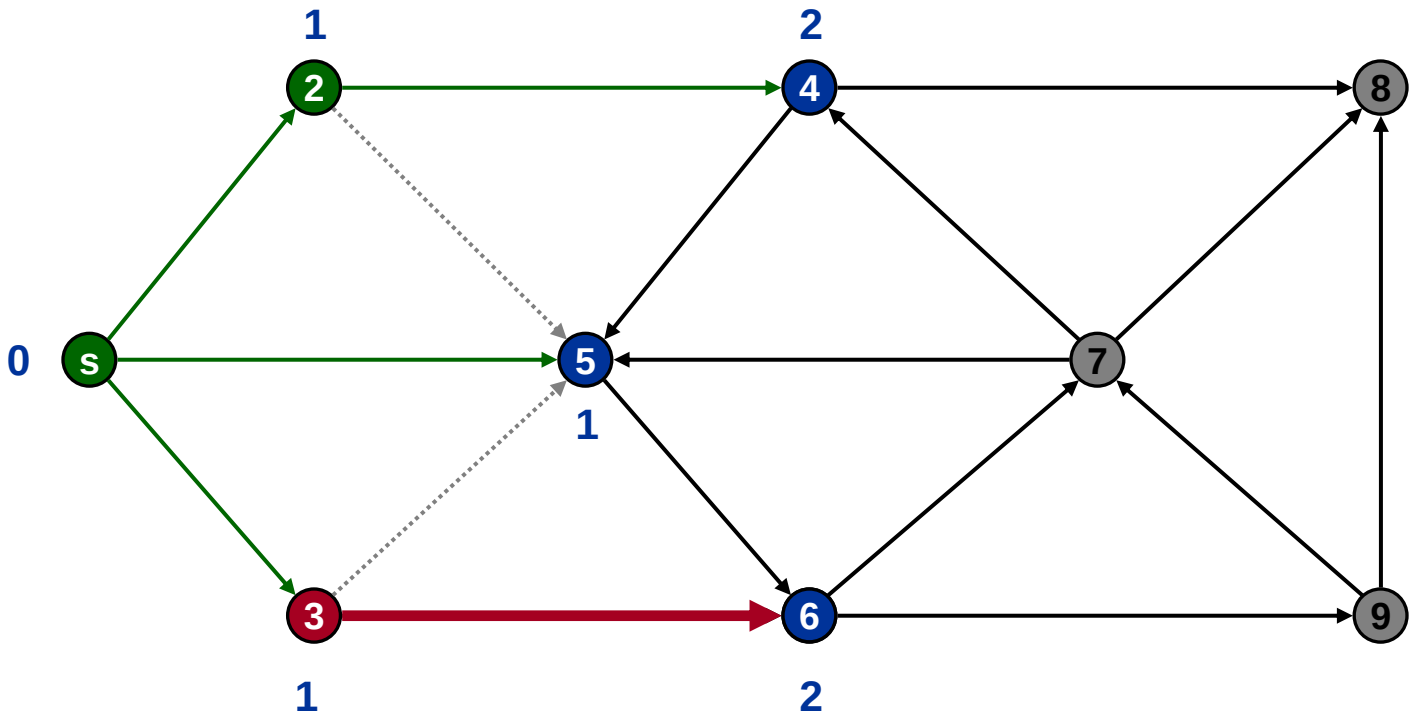
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: 3 5 4**

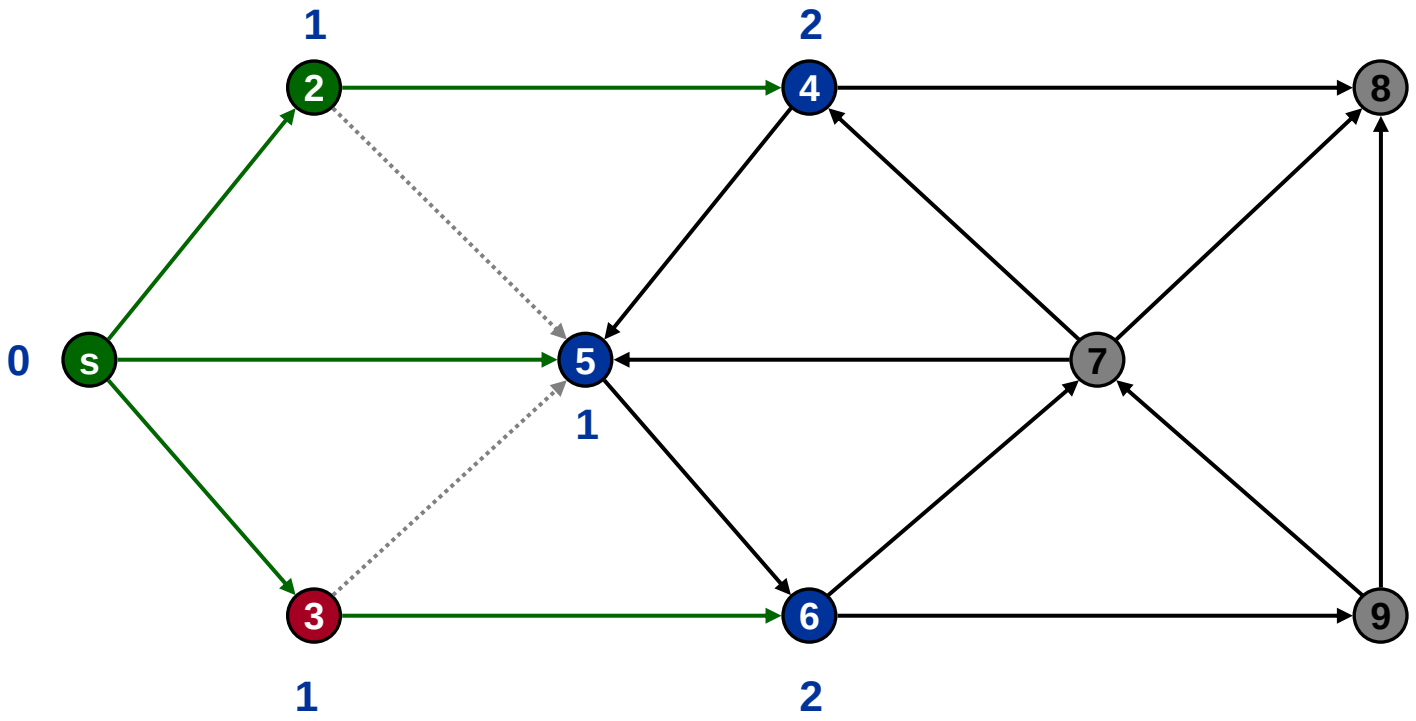
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 3 5 4**

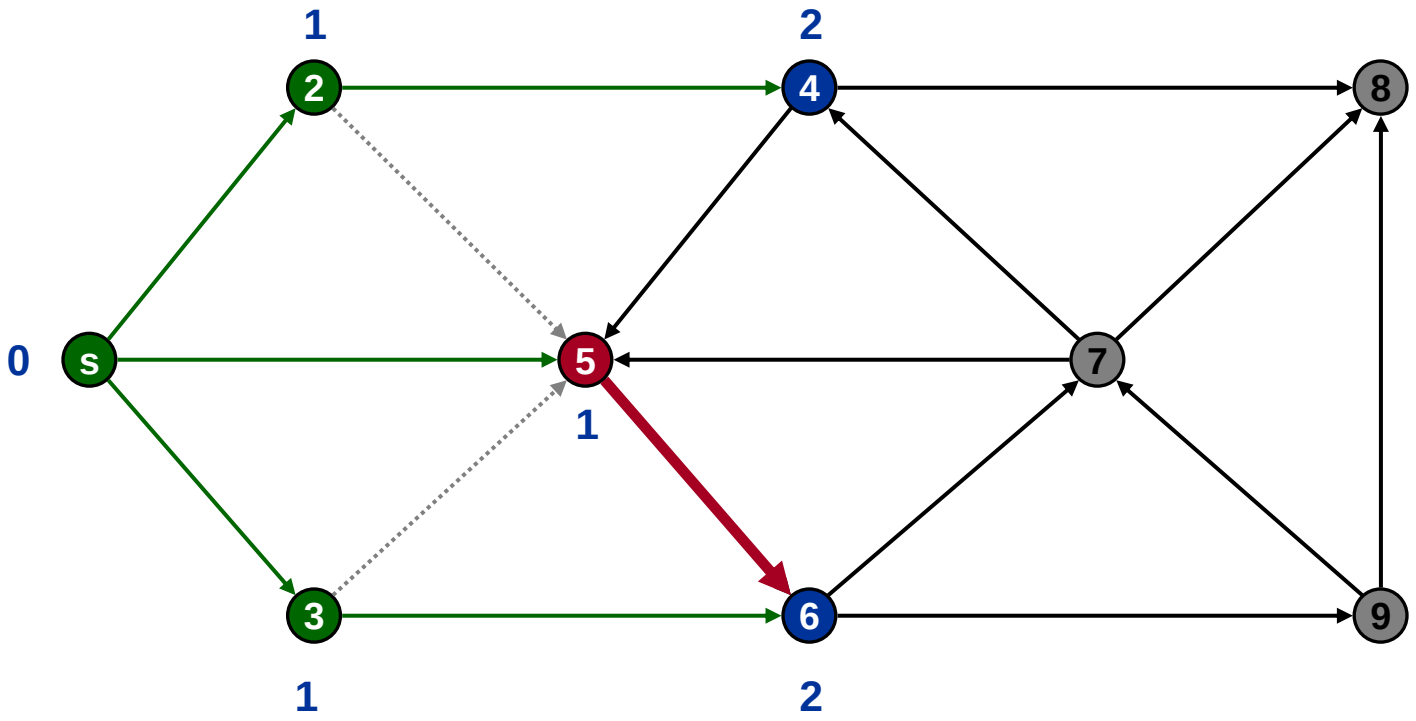
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: 3 5 4 6**

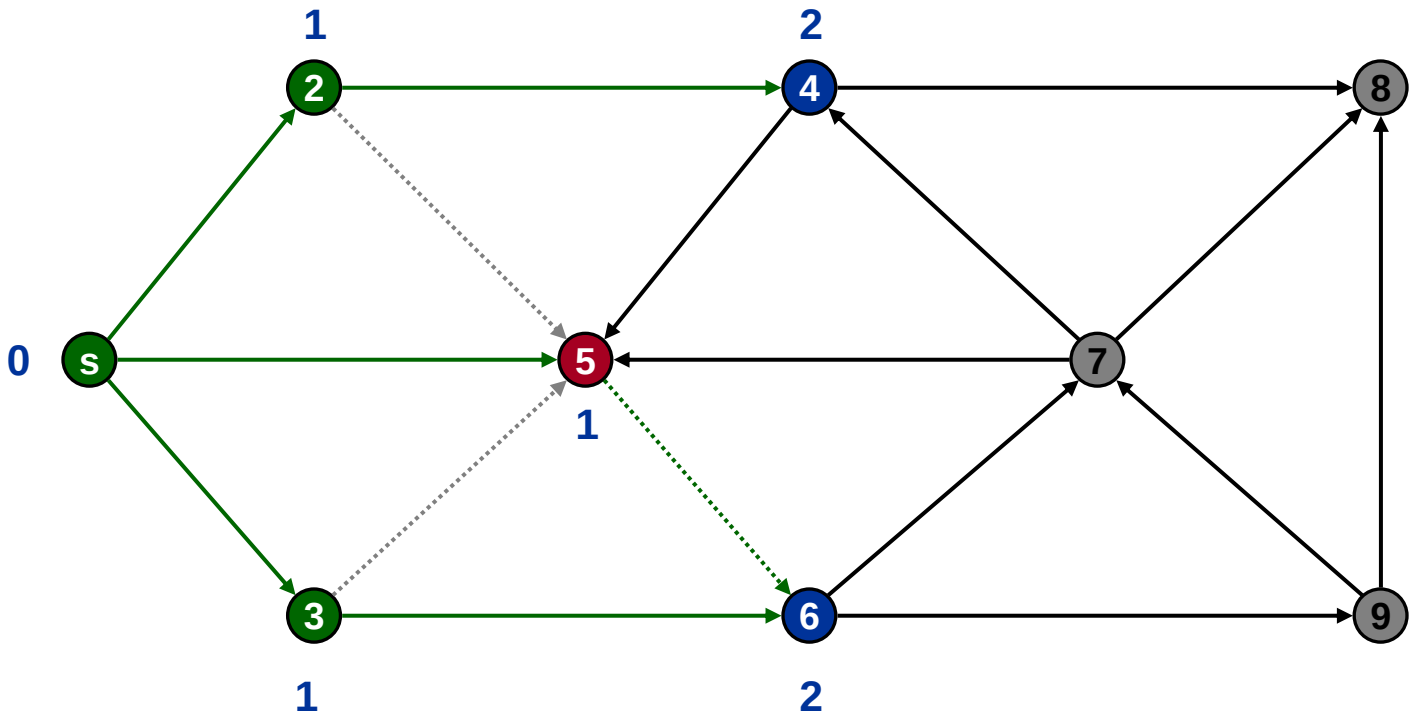
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 5 4 6**

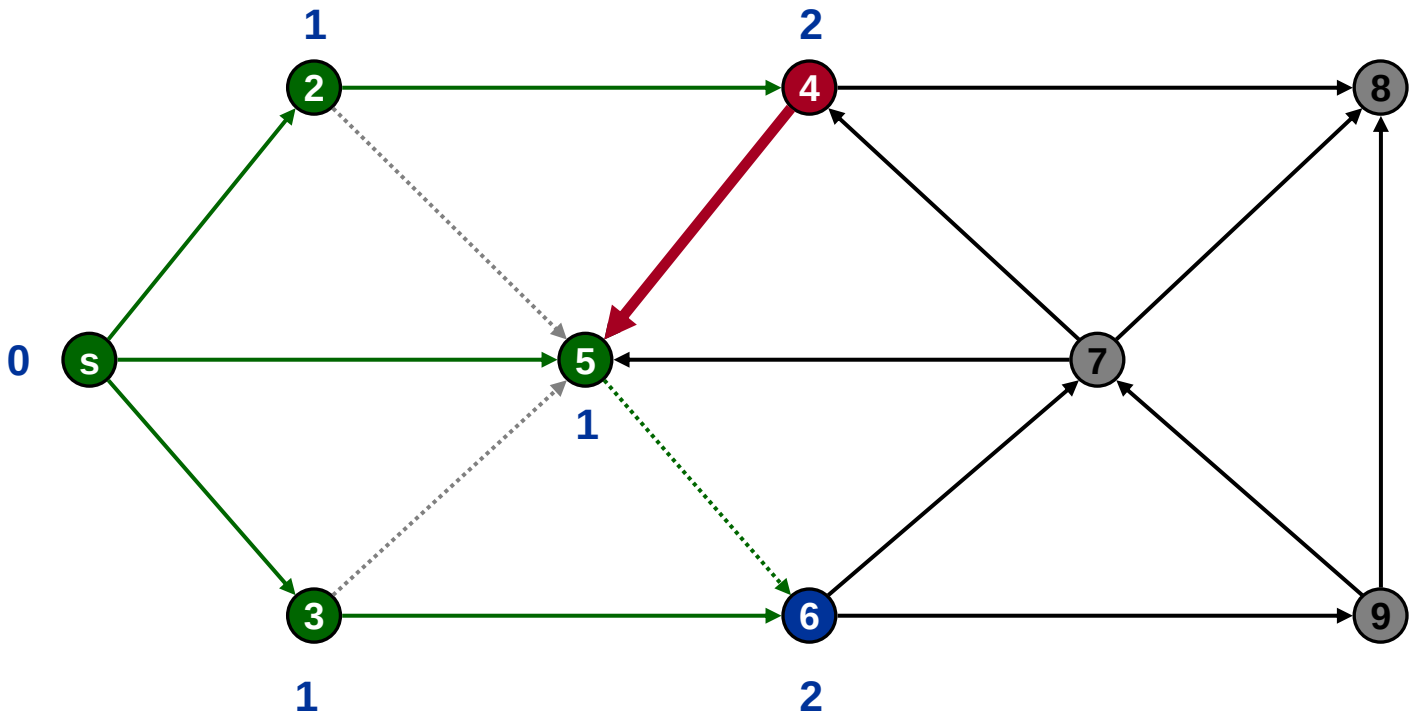
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: 5 4 6**

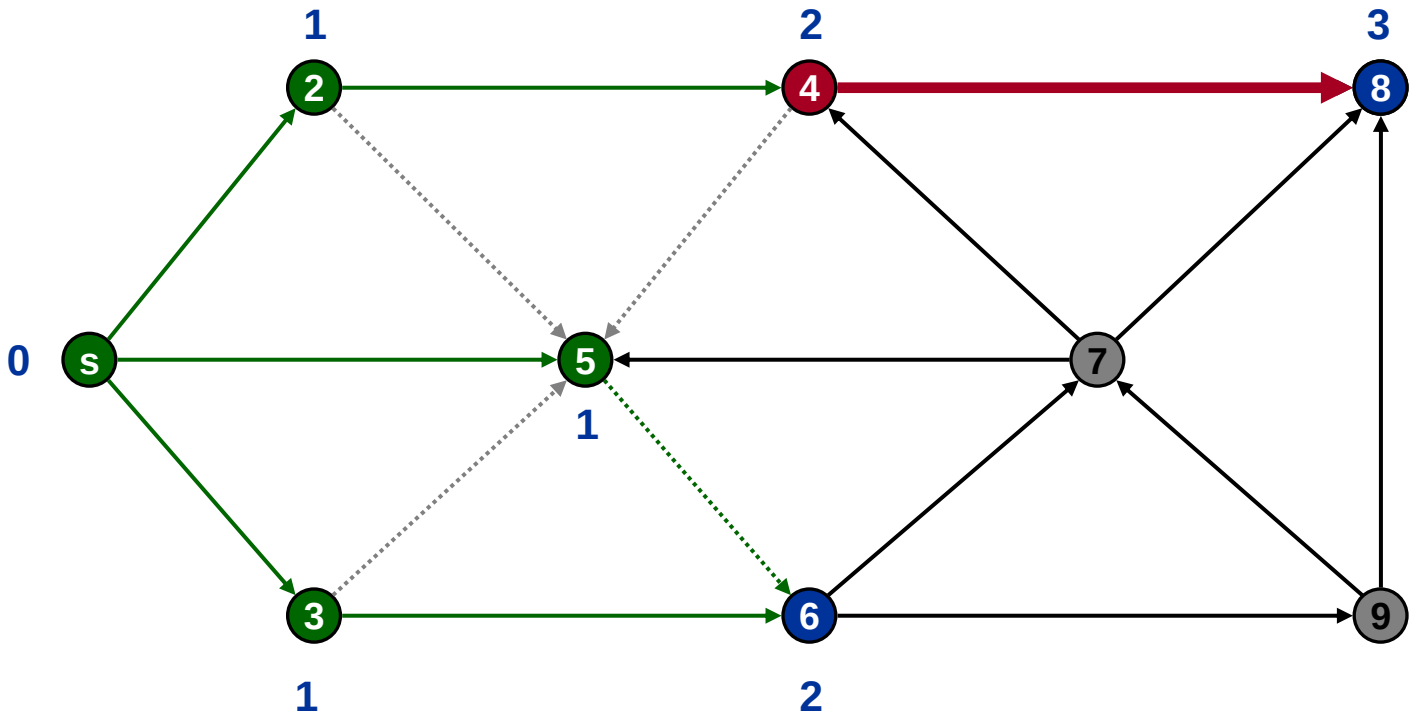
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: 4 6**

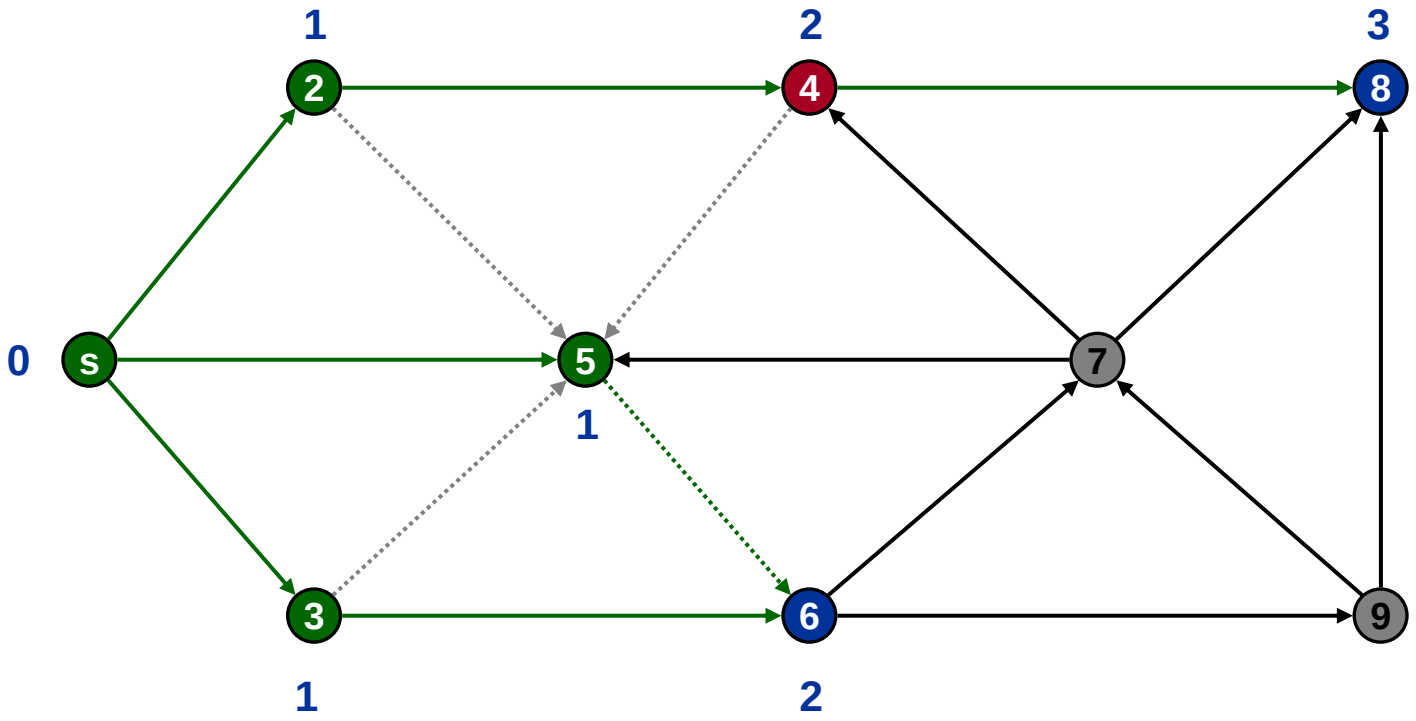
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 4 6**

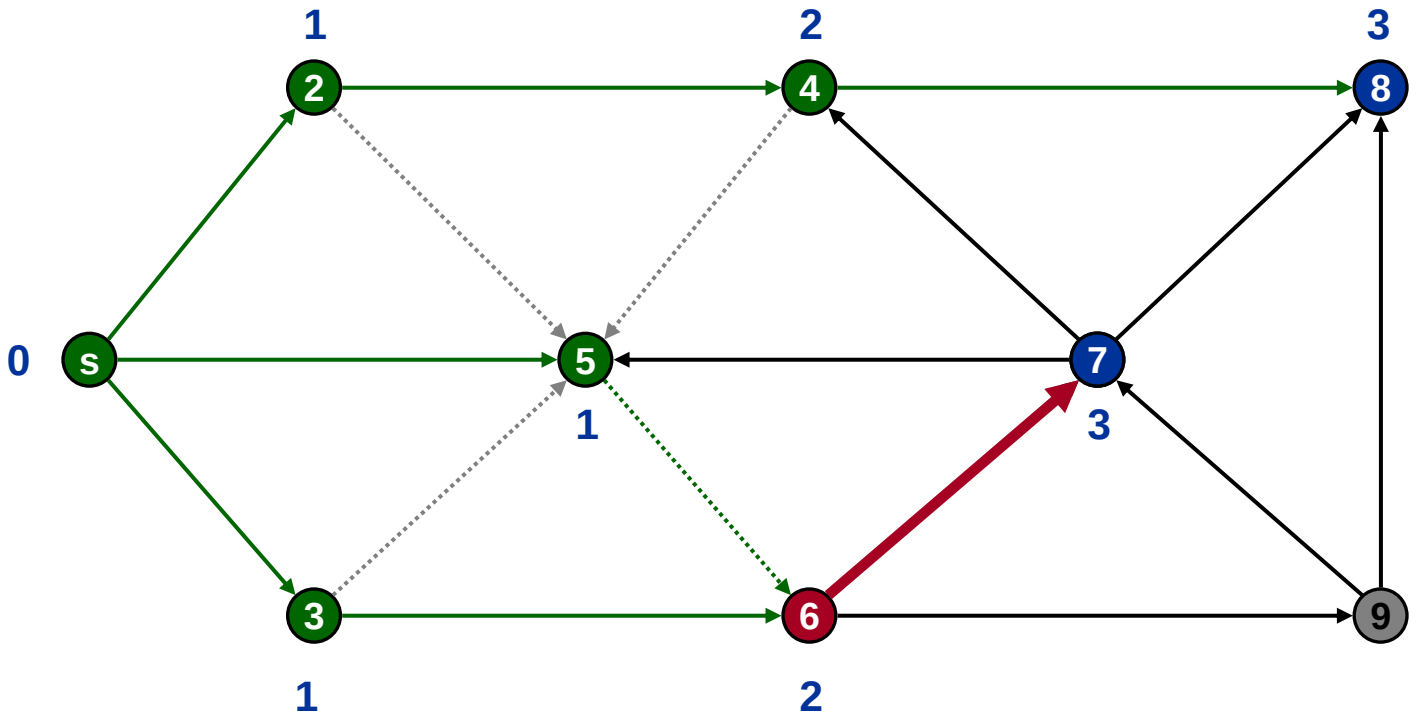
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: 4 6 8**

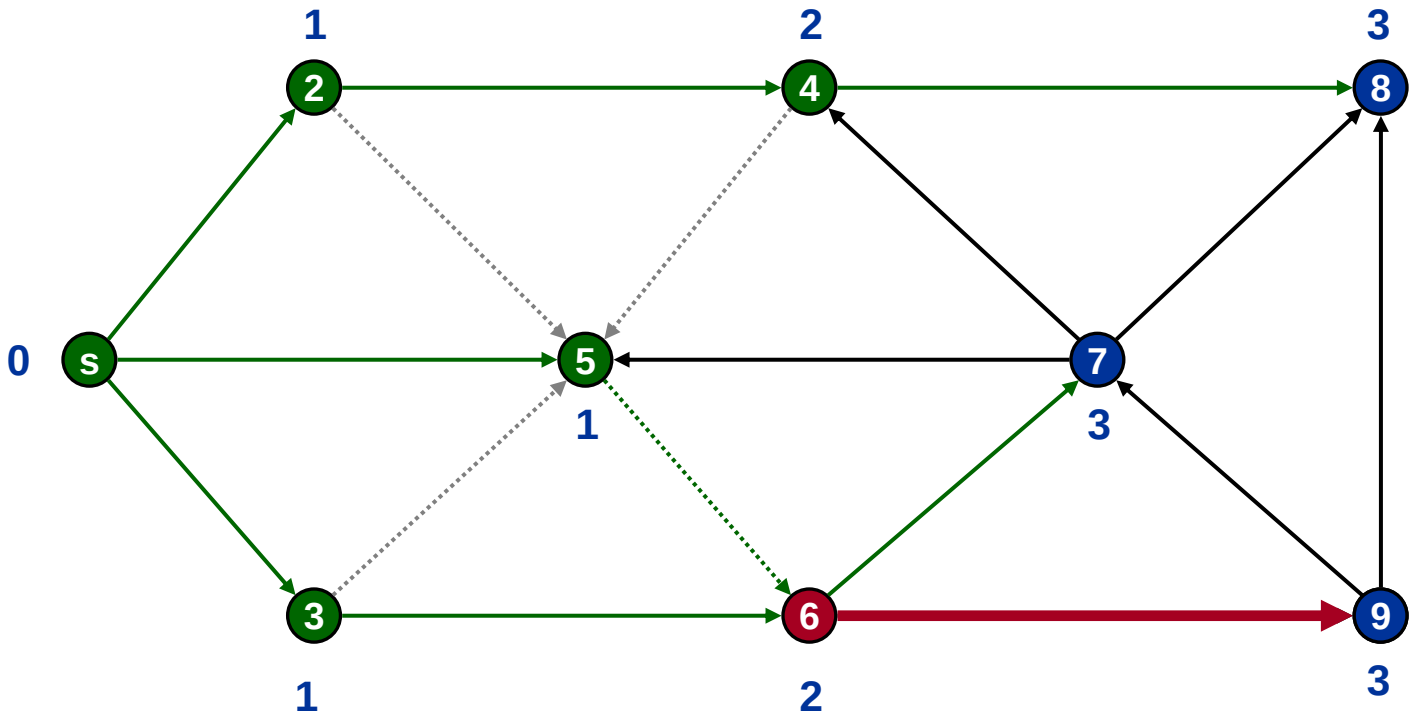
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 6 8**

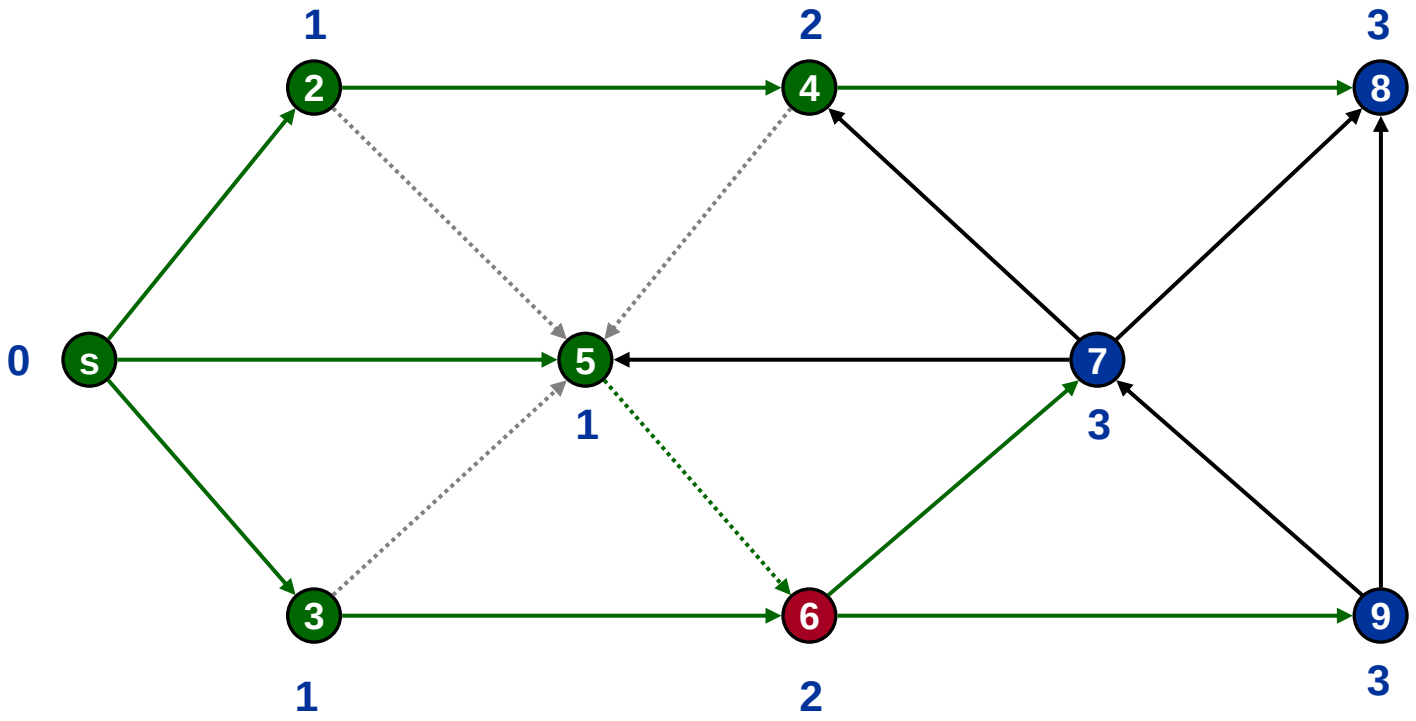
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 6 8 7**

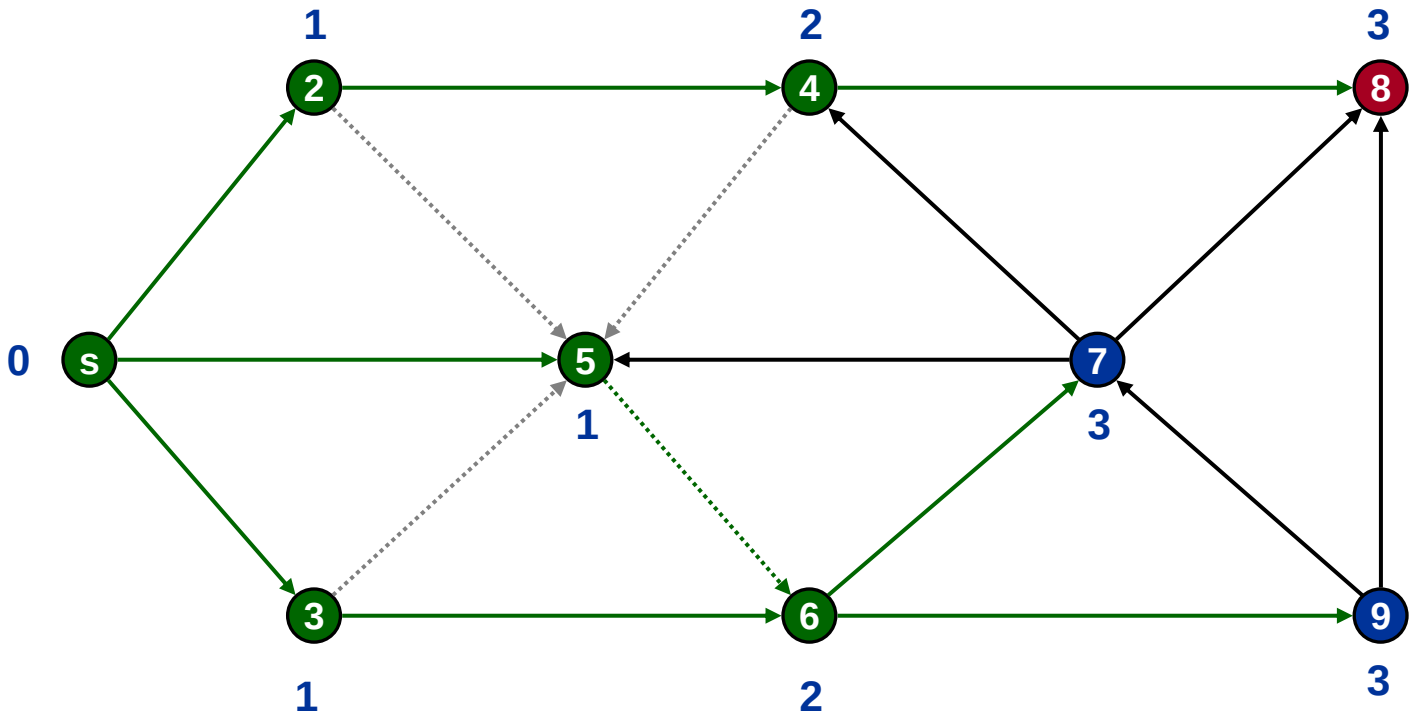
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 6 8 7 9**

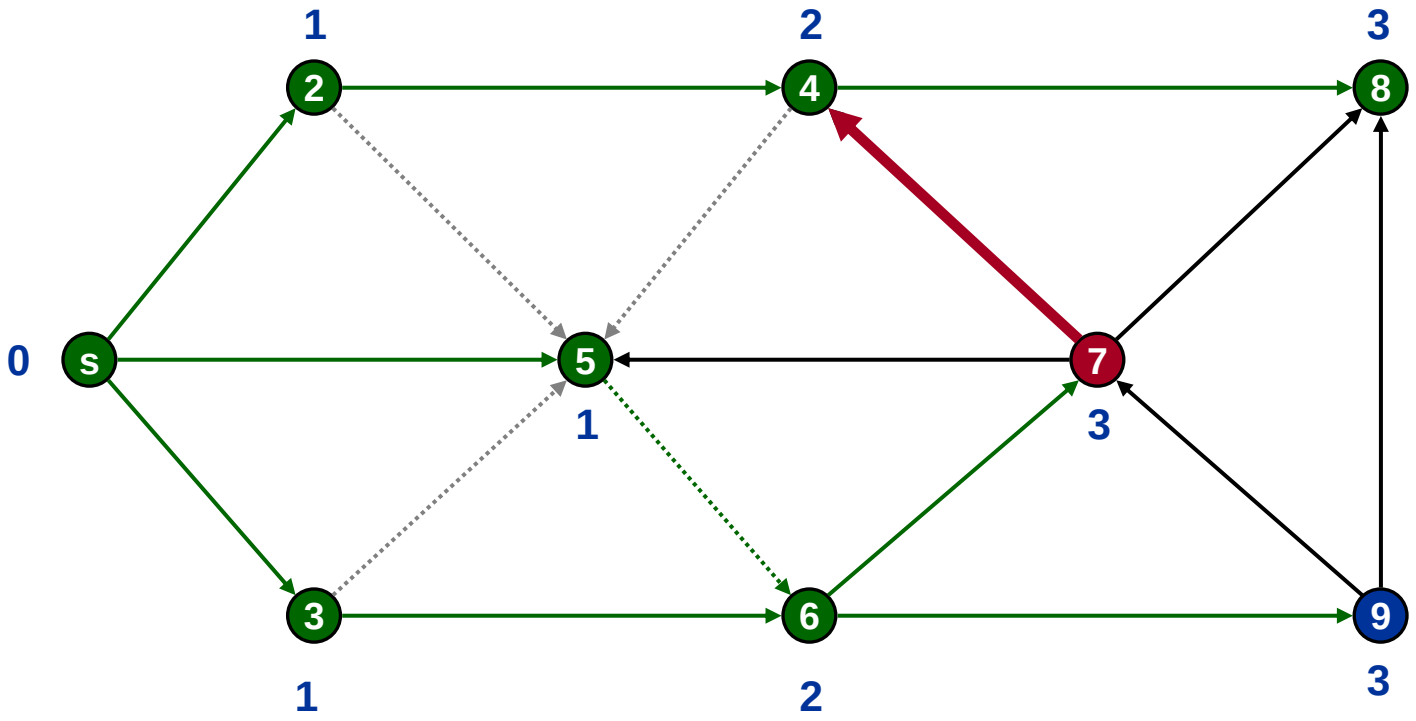
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 8 7 9**

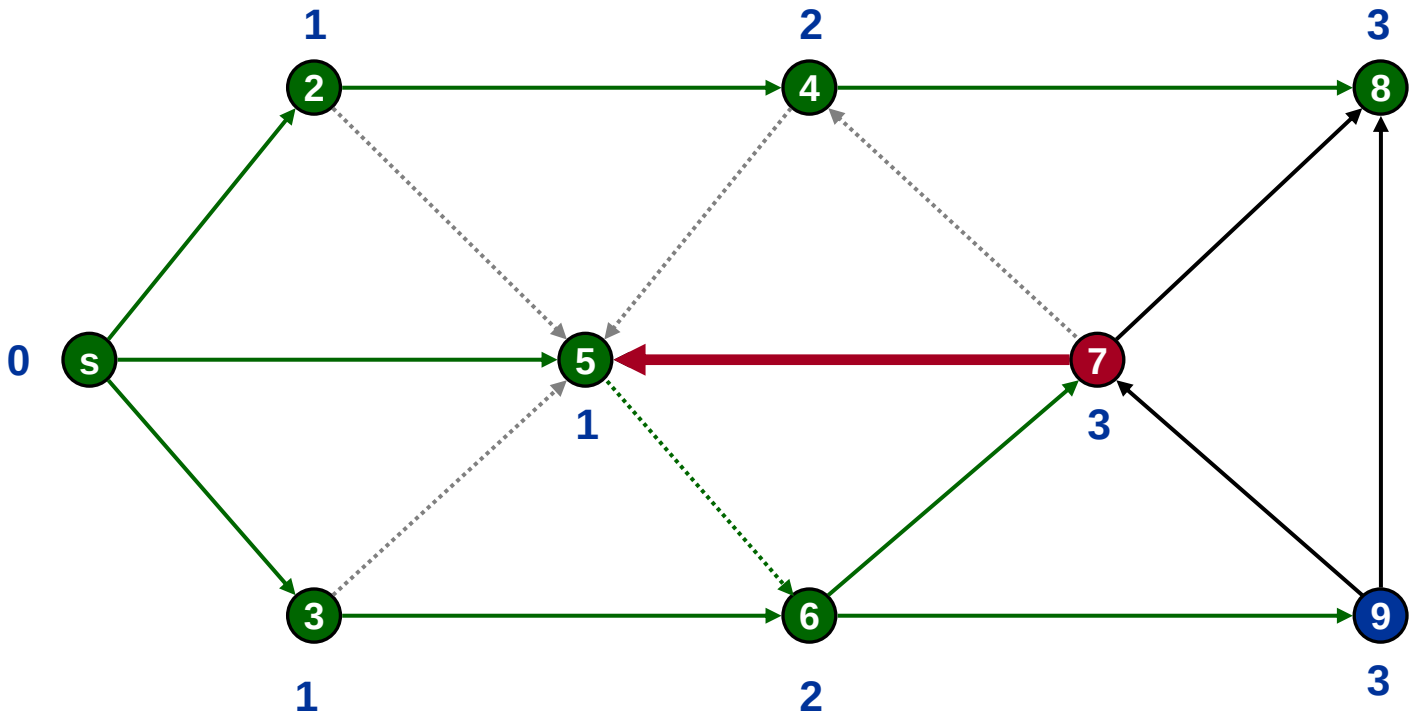
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 7 9**

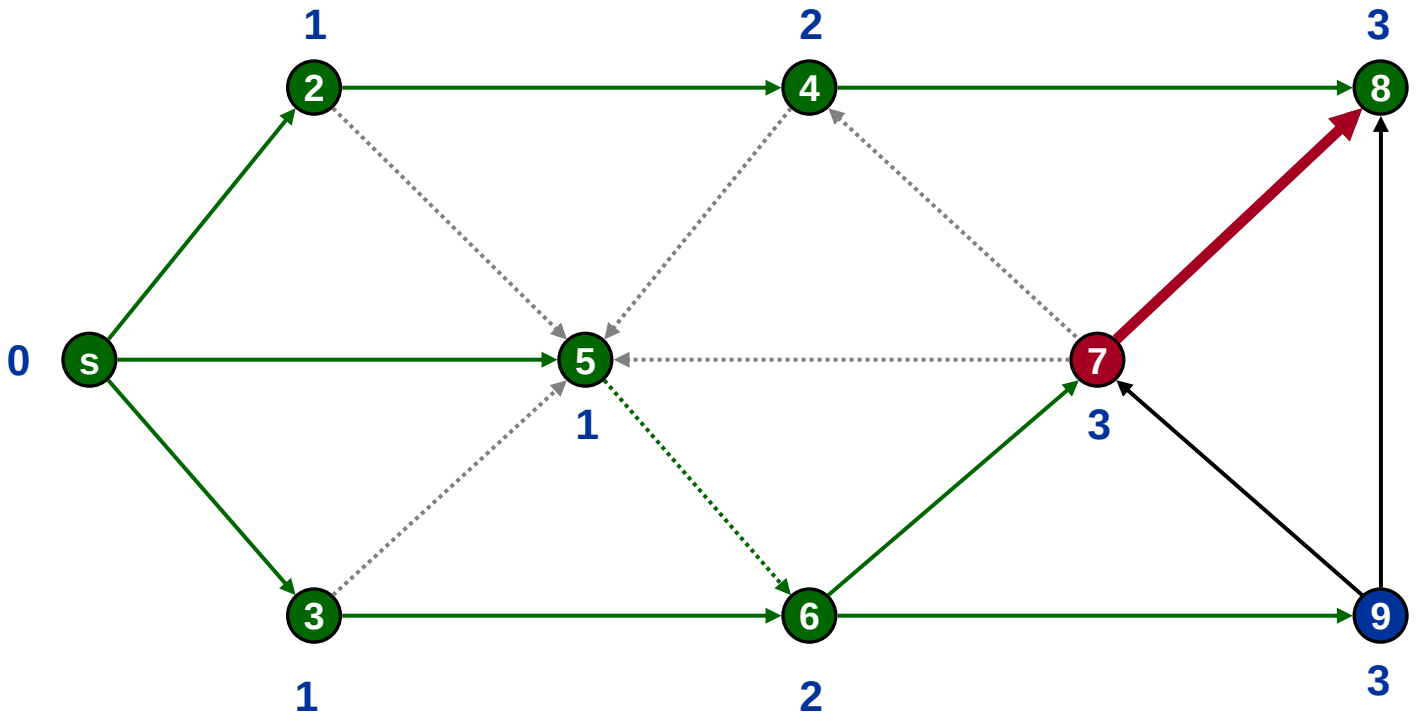
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 7 9**

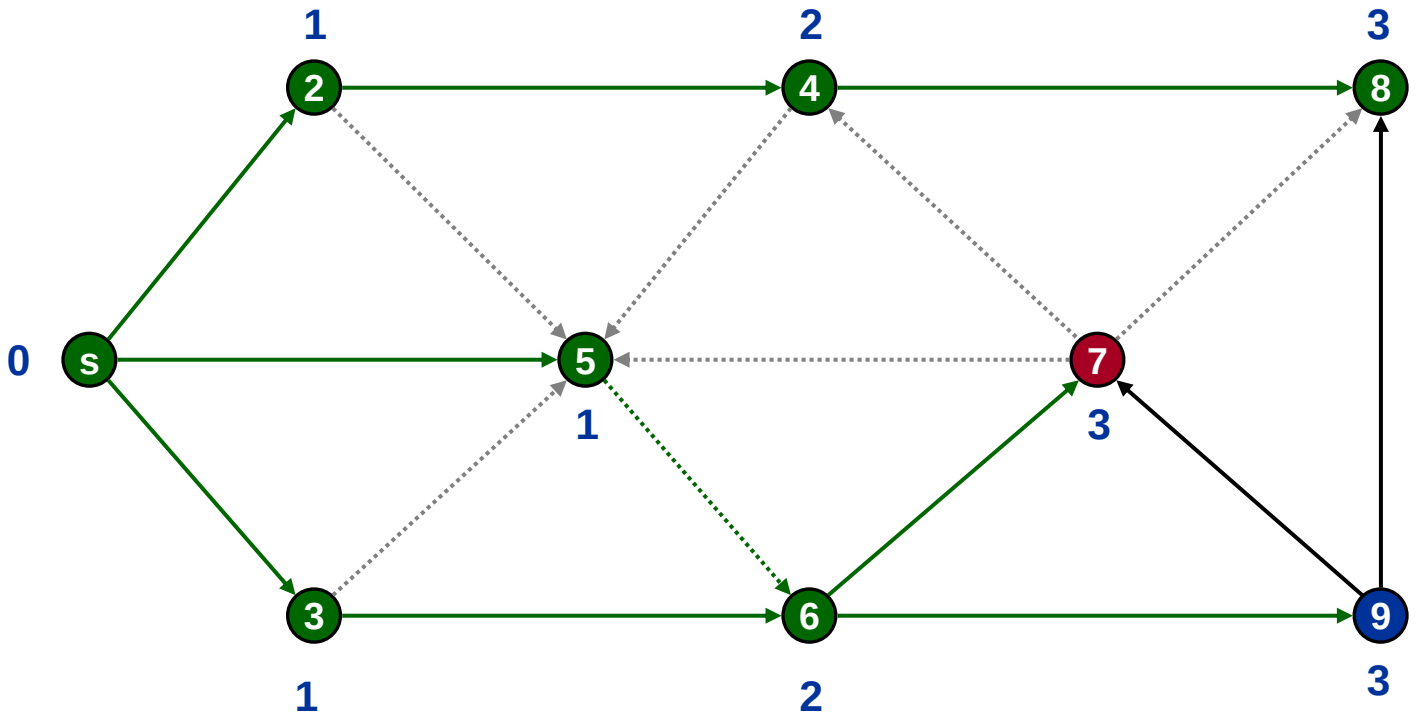
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 7 9**

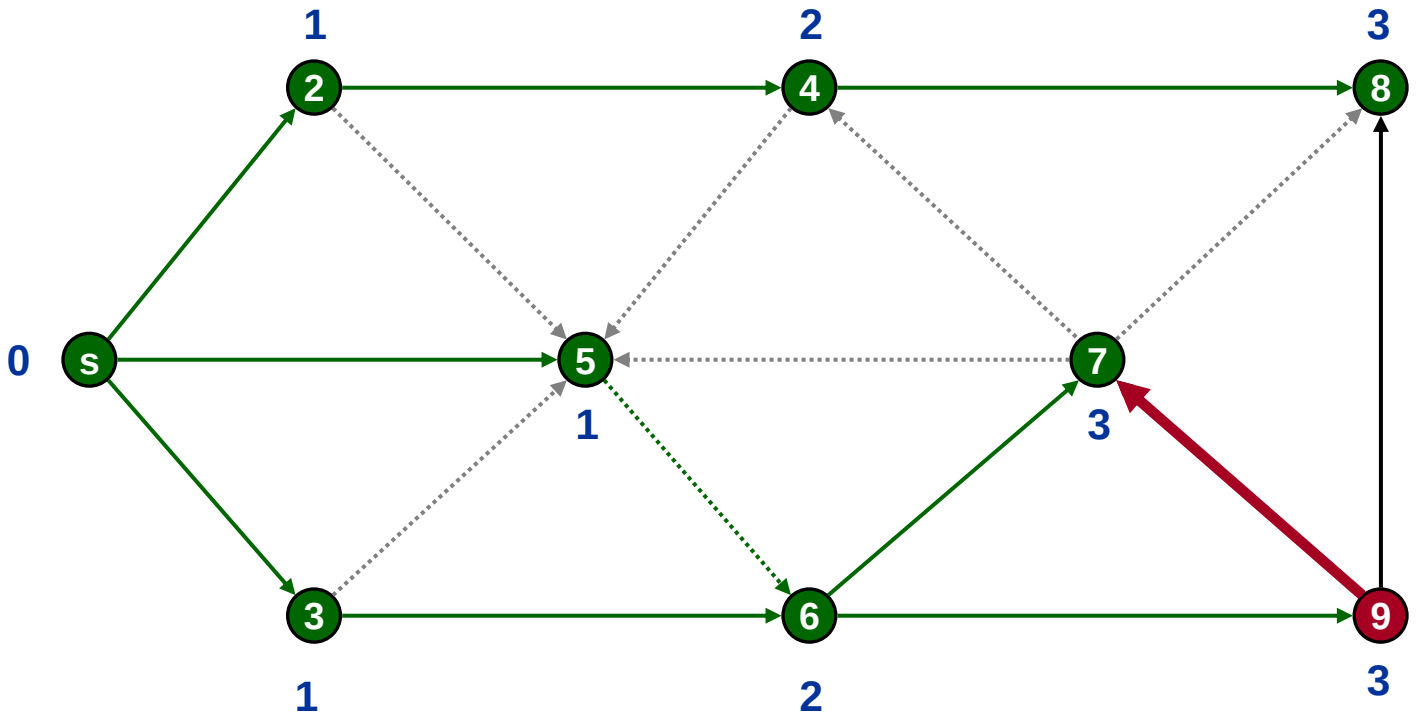
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 7 9**

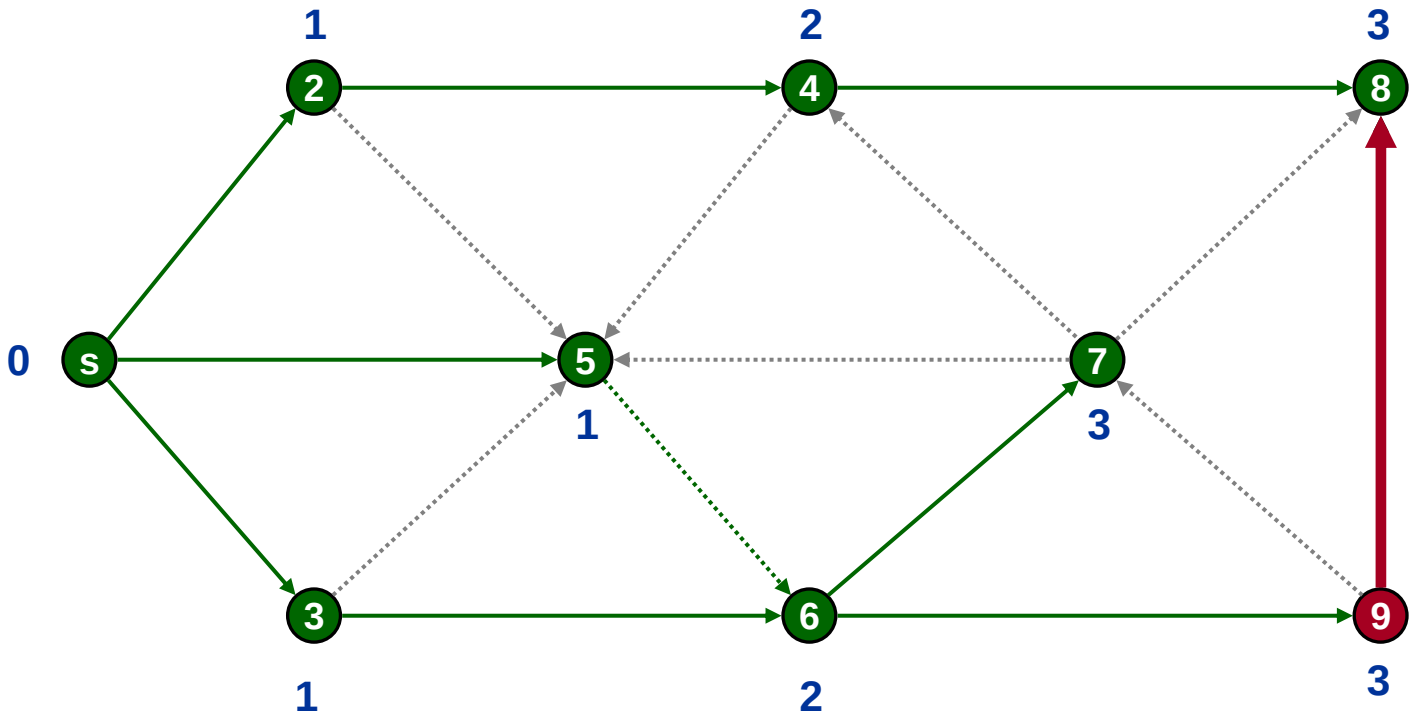
# Breadth First Search




| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 9**

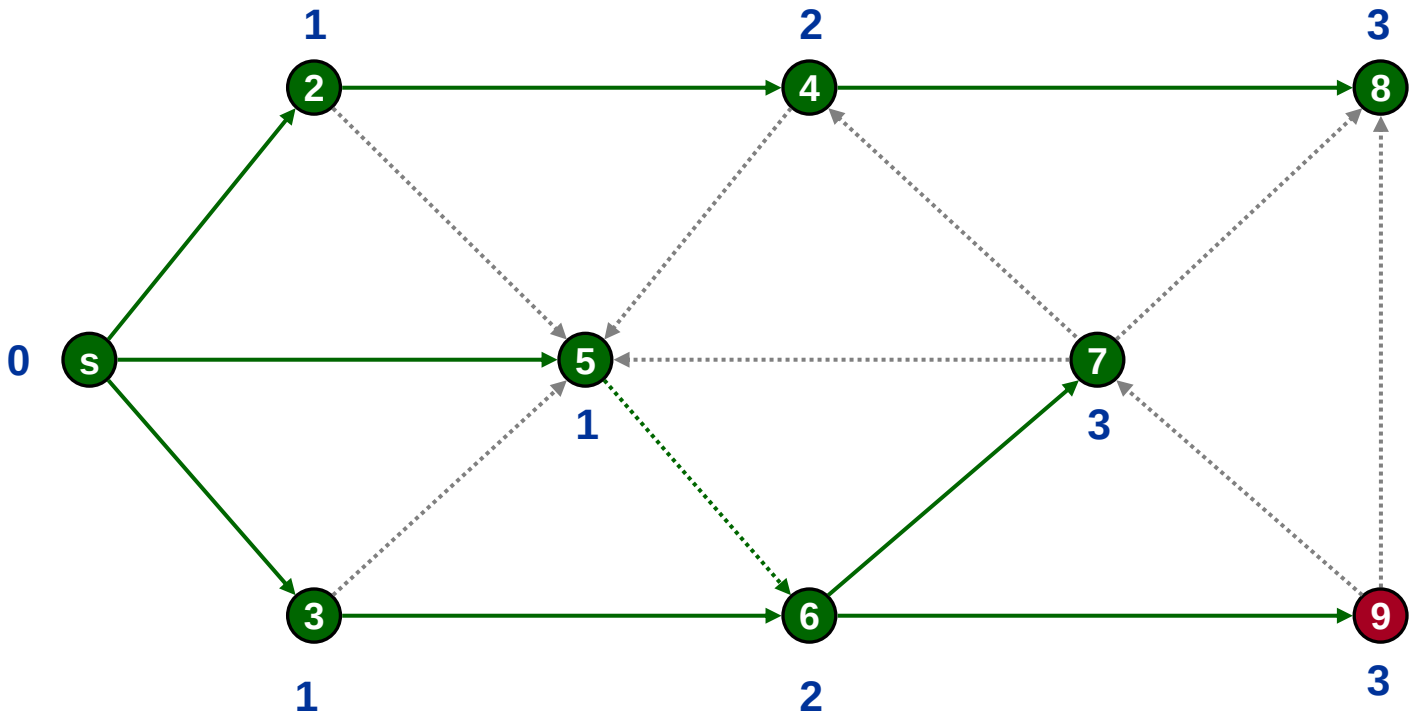
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue: 9**

# Breadth First Search

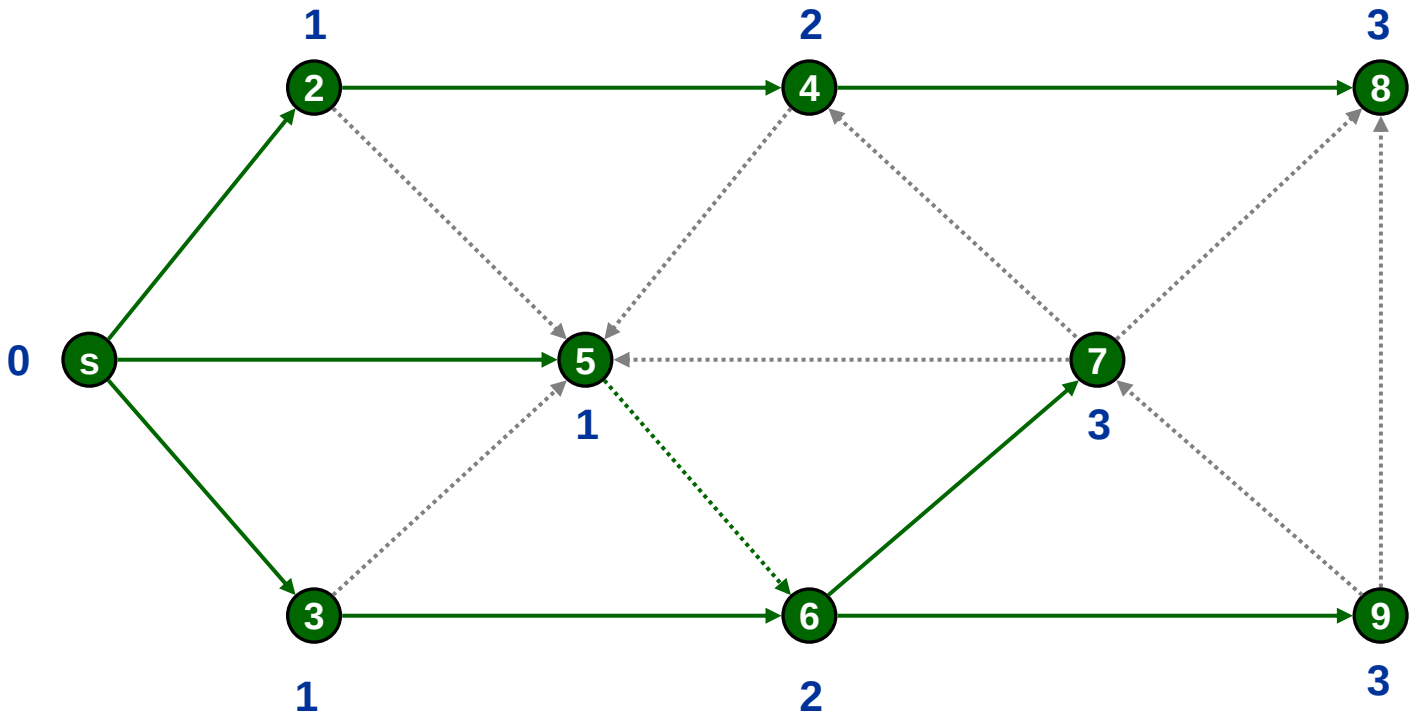



| Undiscovered |
| --- |
| Discovered |
| Top of queue |
| Finished |

**Queue: 9**

# Breadth First Search

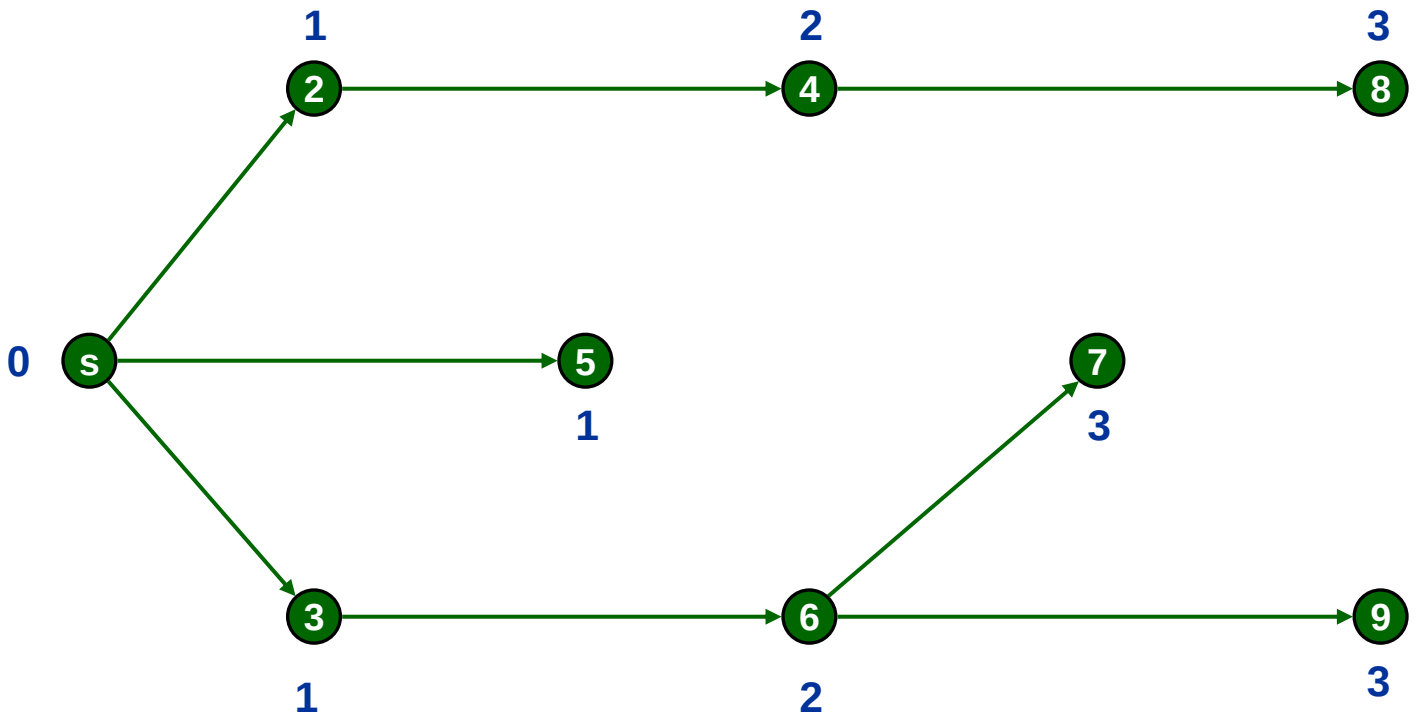
**Traversal**
s 2 3 5 4 6 8 7 9




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

**Queue:**

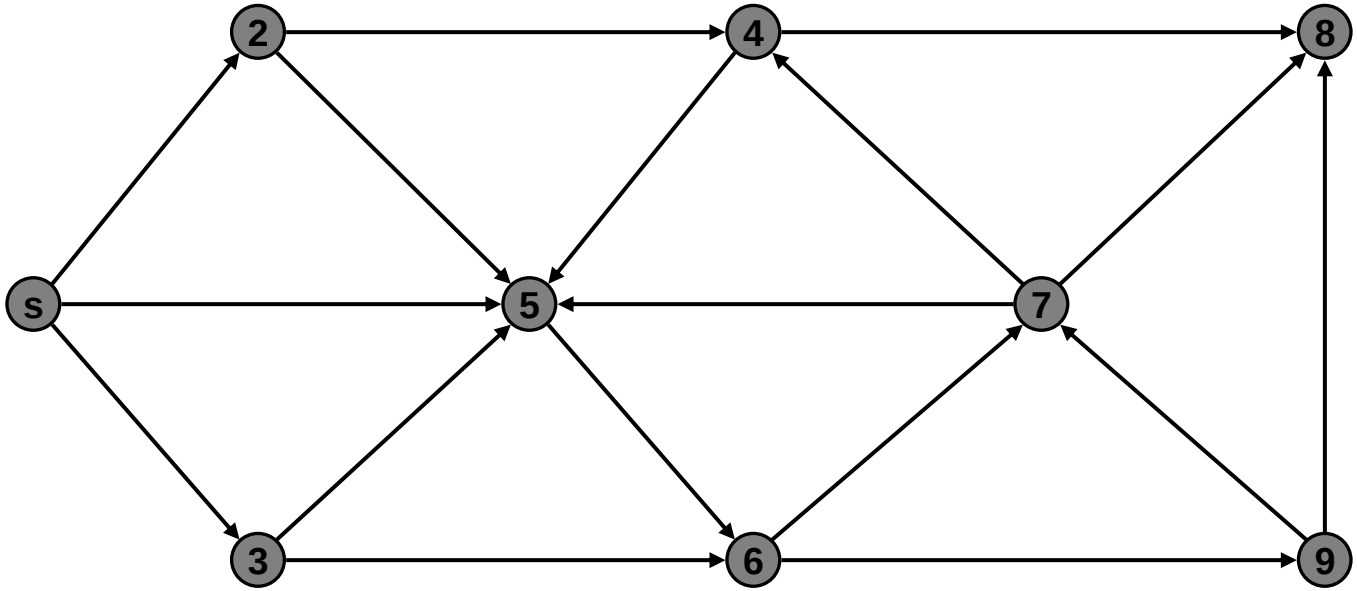
# Breadth First Search




**Level Graph**

# Depth First Search

**Traversal**
s 2 4 8 5 6 7 9 3




- **Apply DFS algorithm on the same graph using stack and see what is DFS Tree generated after exhausting whole stack.**

*Thank You……*