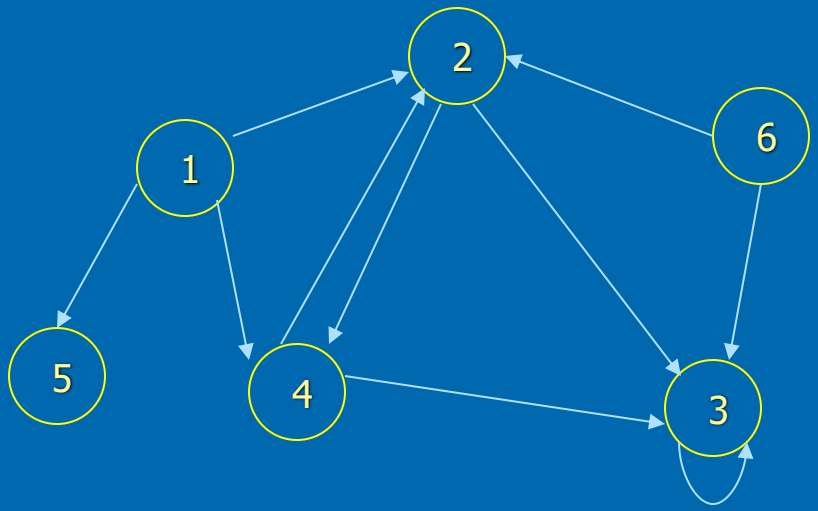
Lecture # 14

- Graphs
- Depth First Search (DFS)
- Breath First Search (BFS)

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 form 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.

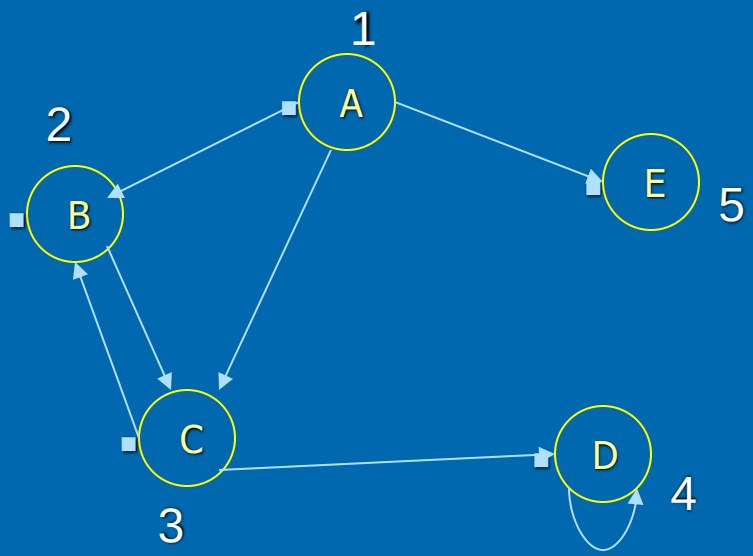
1. Construct an empty directed graph.
2. Check if it is empty
3. Destroy a directed graph
4. Insert a new node
5. Insert a directed edge between two existing nodes or from a node to itself
6. Delete a node and all directed edges to or from it.
7. Delete directed edge between two existing nodes
8. Search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

$$adj = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

- 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.

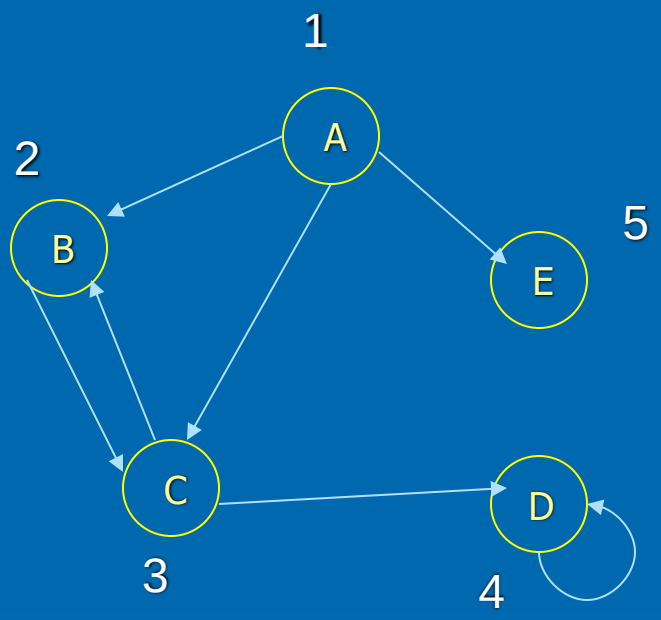
$$\text{adj} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{data} = \begin{pmatrix} A \\ B \\ C \\ D \\ E \end{pmatrix}$$

- 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.

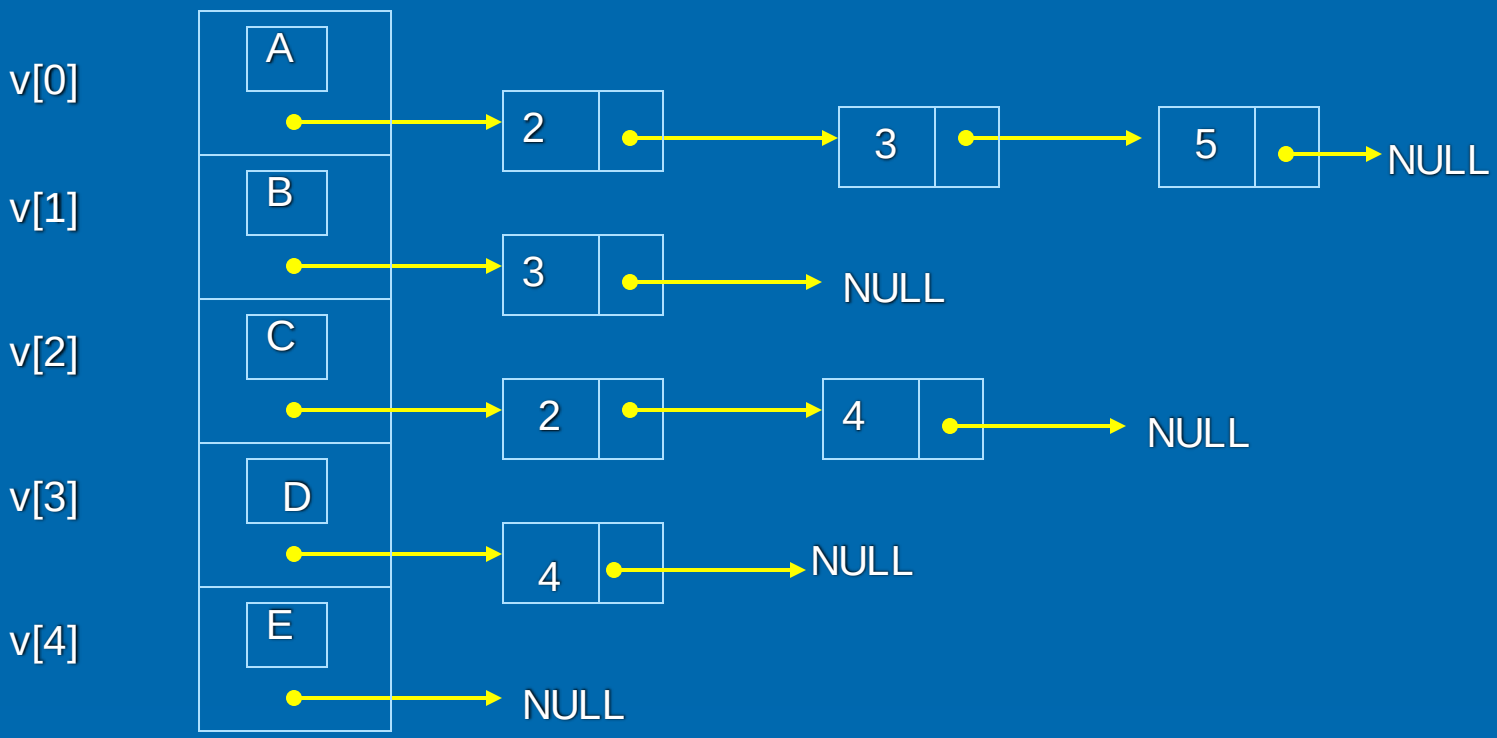
Adjacency List Representation

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

Analysis

- For both directed/undirecdted graphs, the adjacency list representation has the desirable property that the amount of memory it requires $\theta(V+E)$
- A potential disadvantage of the adjacency list representation is that there is no quicker way to determine if a given edge (u,v) is present in the graph than to search for v in the adjacency list $Adj[u]$. This disadvantage can be remedied by an adjacency matrix representation at the cost of using asymptotically more memory.

Analysis

- Adjacency matrix of a graph requires $\theta(V^2)$ memory, independent of the number of edges in the graph.

- Although the adjacency list representation is asymptotically at least as efficient as the adjacency matrix representation , the simplicity of adjacency matrix may make it preferable when graphs are reasonably small.

Searching and Traversing Digraphs

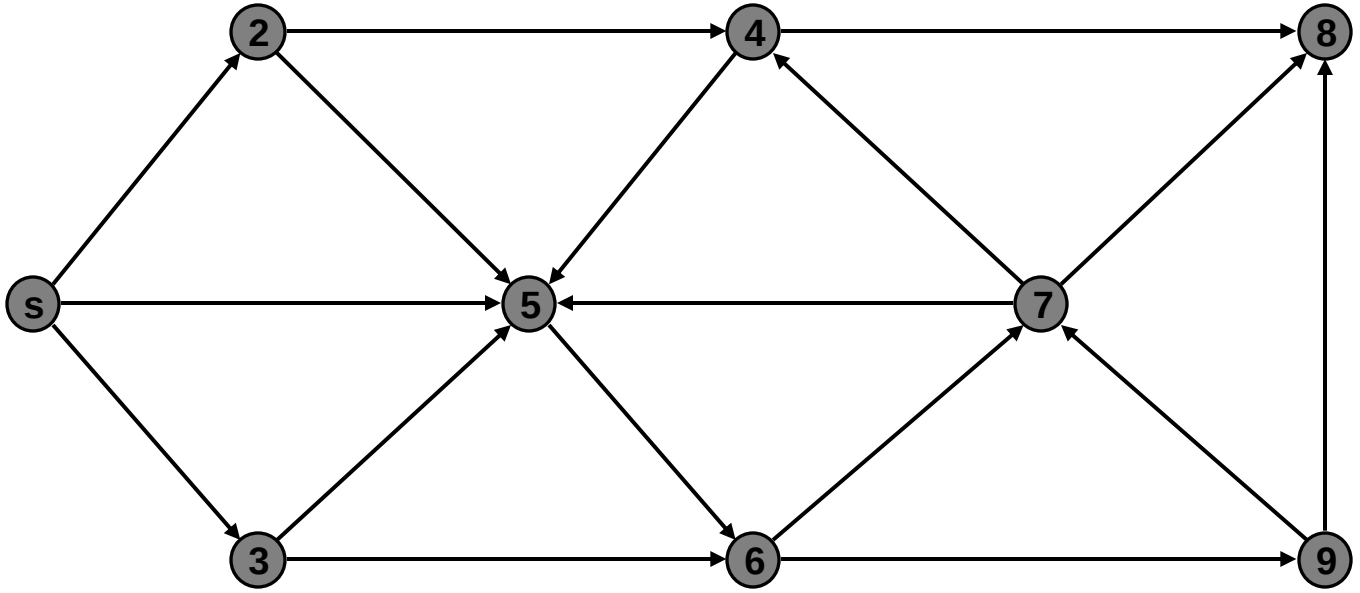
- Two common types of graph traversals are **Depth First Search** (DFS) and **Breadth First Search** (BFS).

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

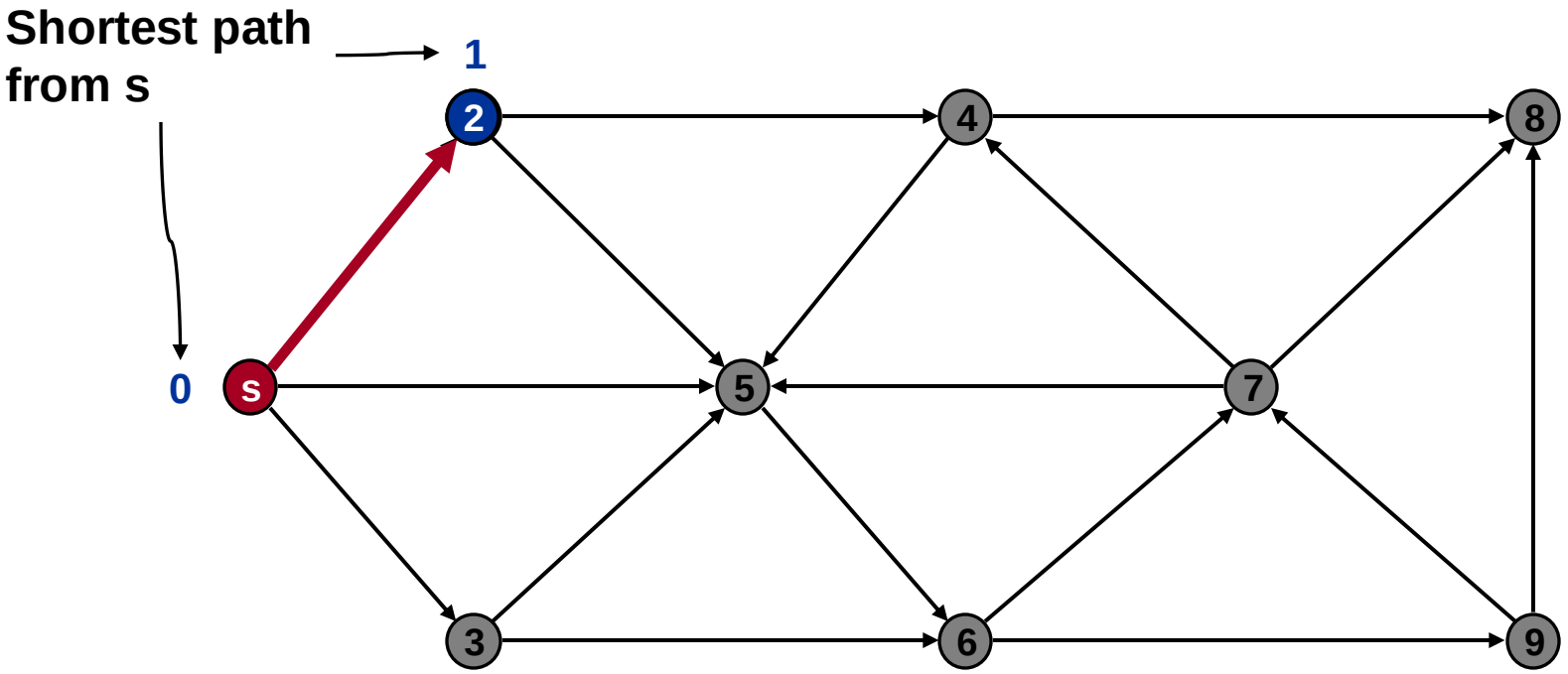
- The aim in both types of traversals is to visit each vertex of a graph *exactly once*.

- 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



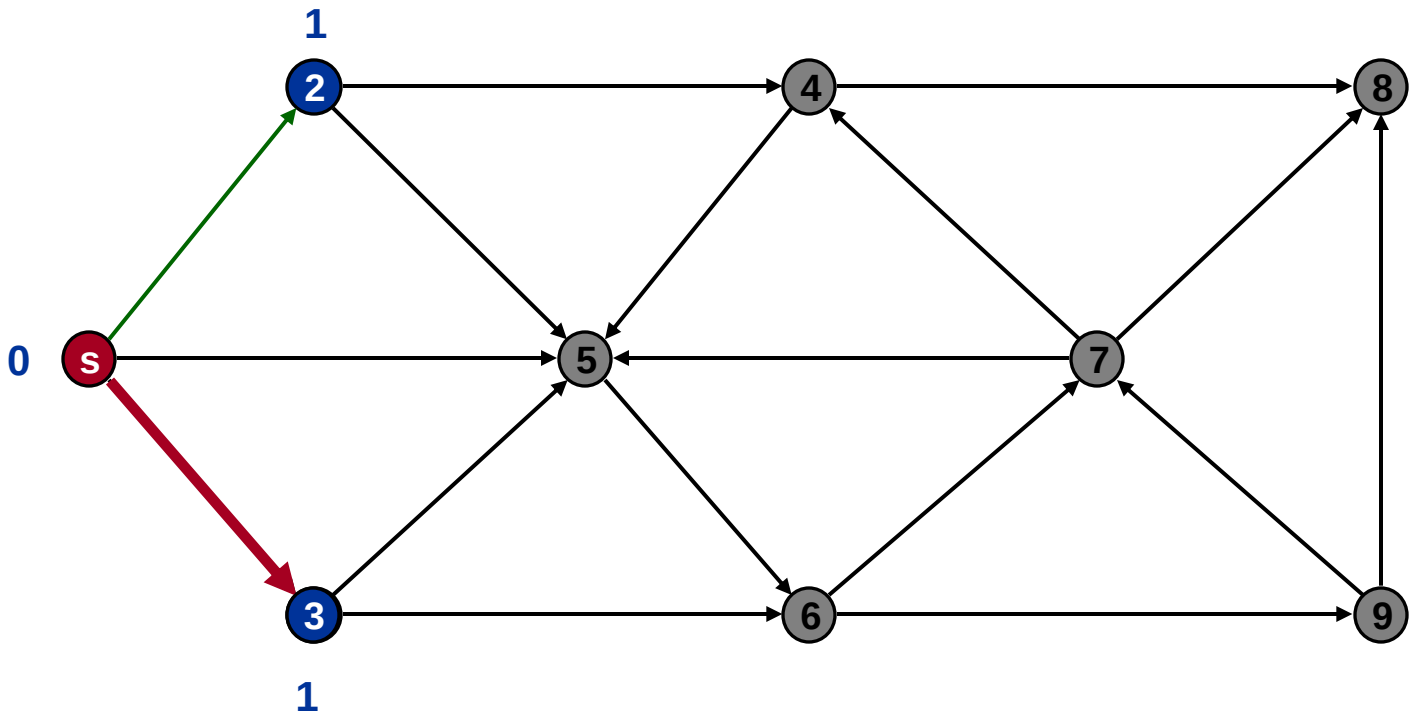
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: s

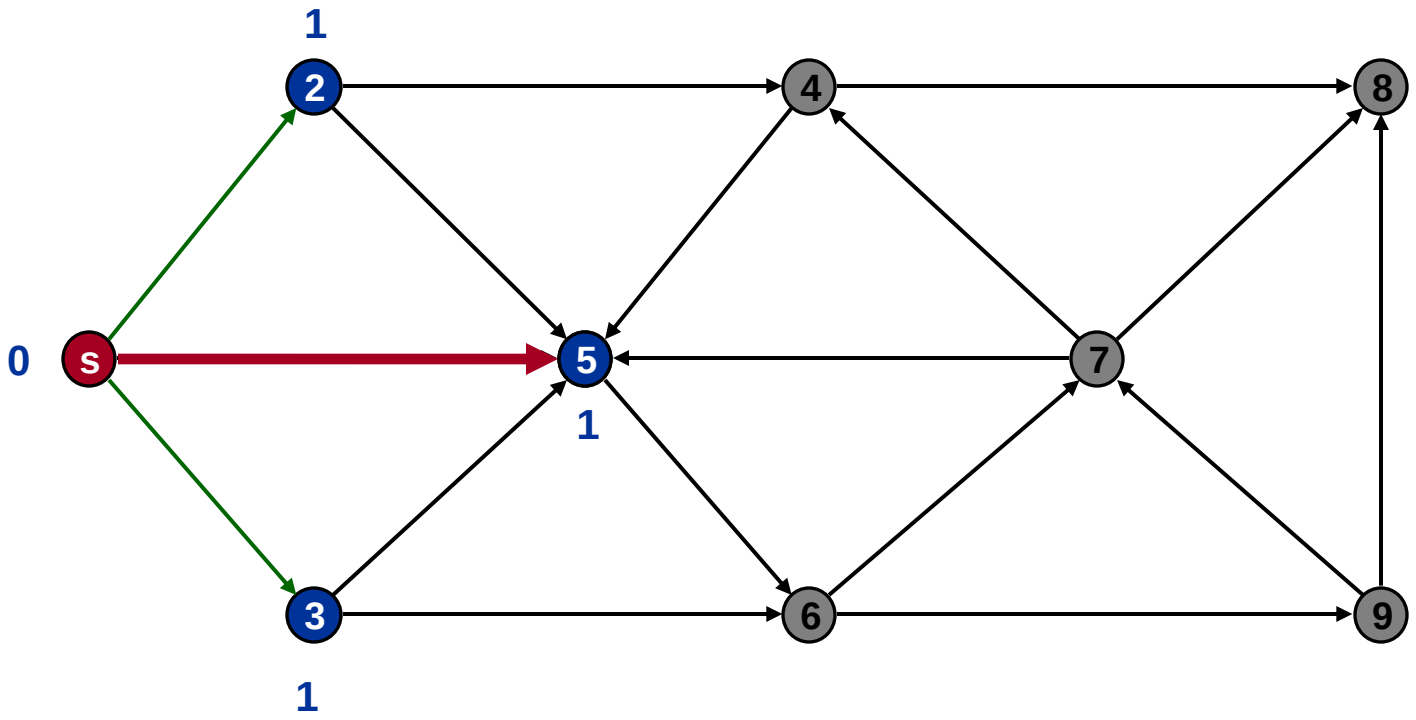
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: s 2

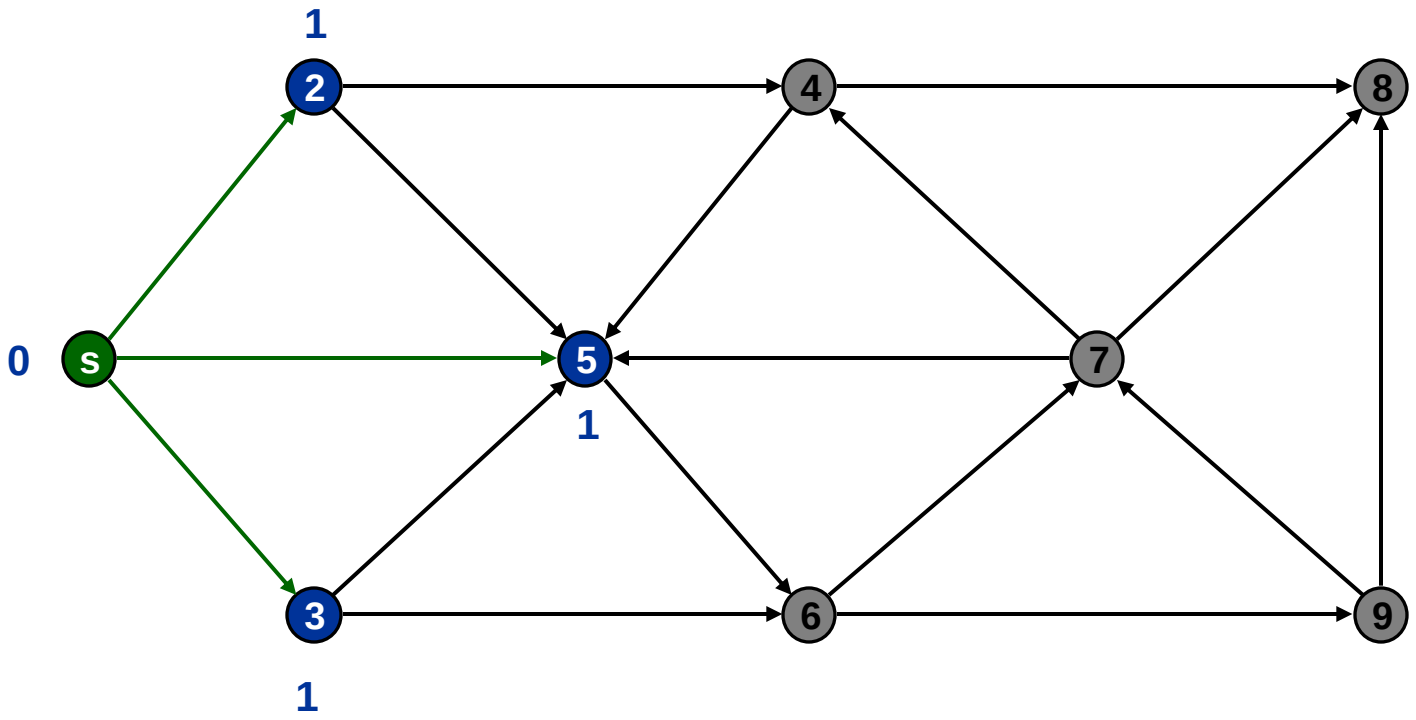
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: s 2 3

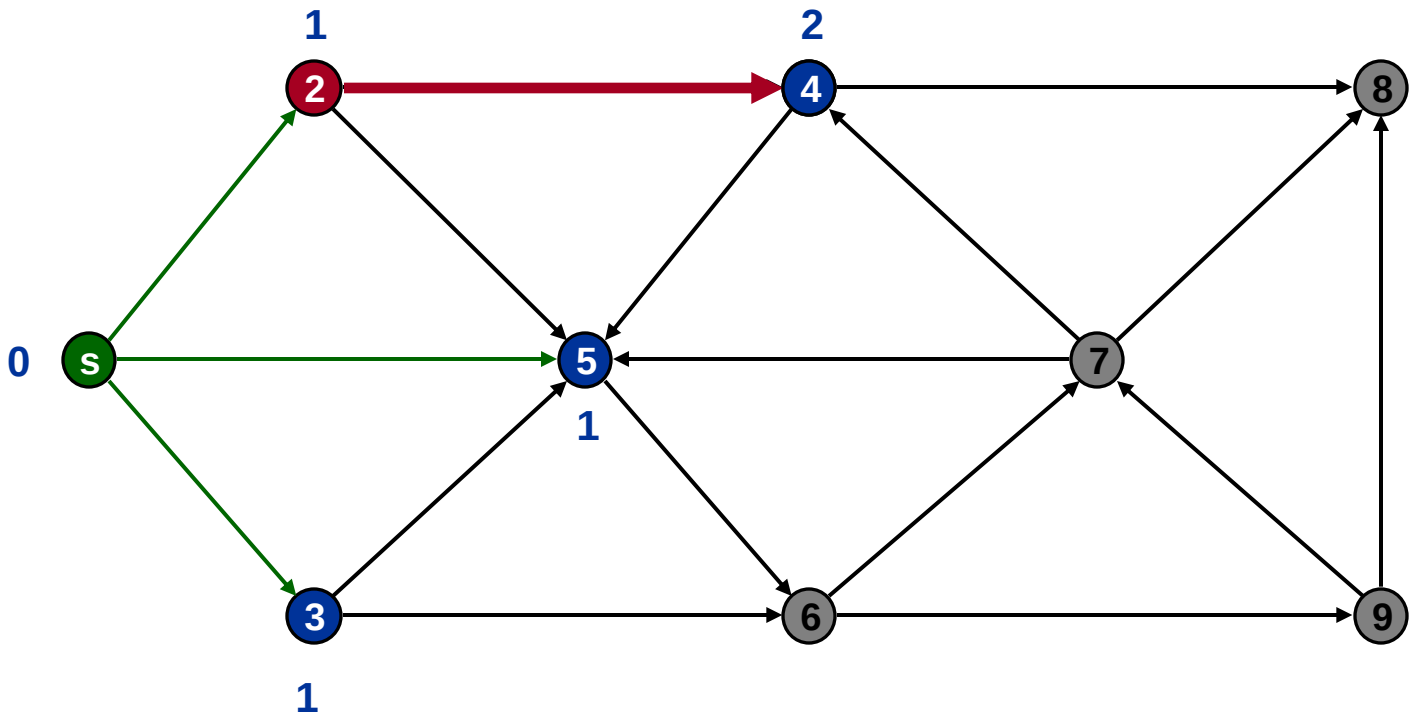
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5

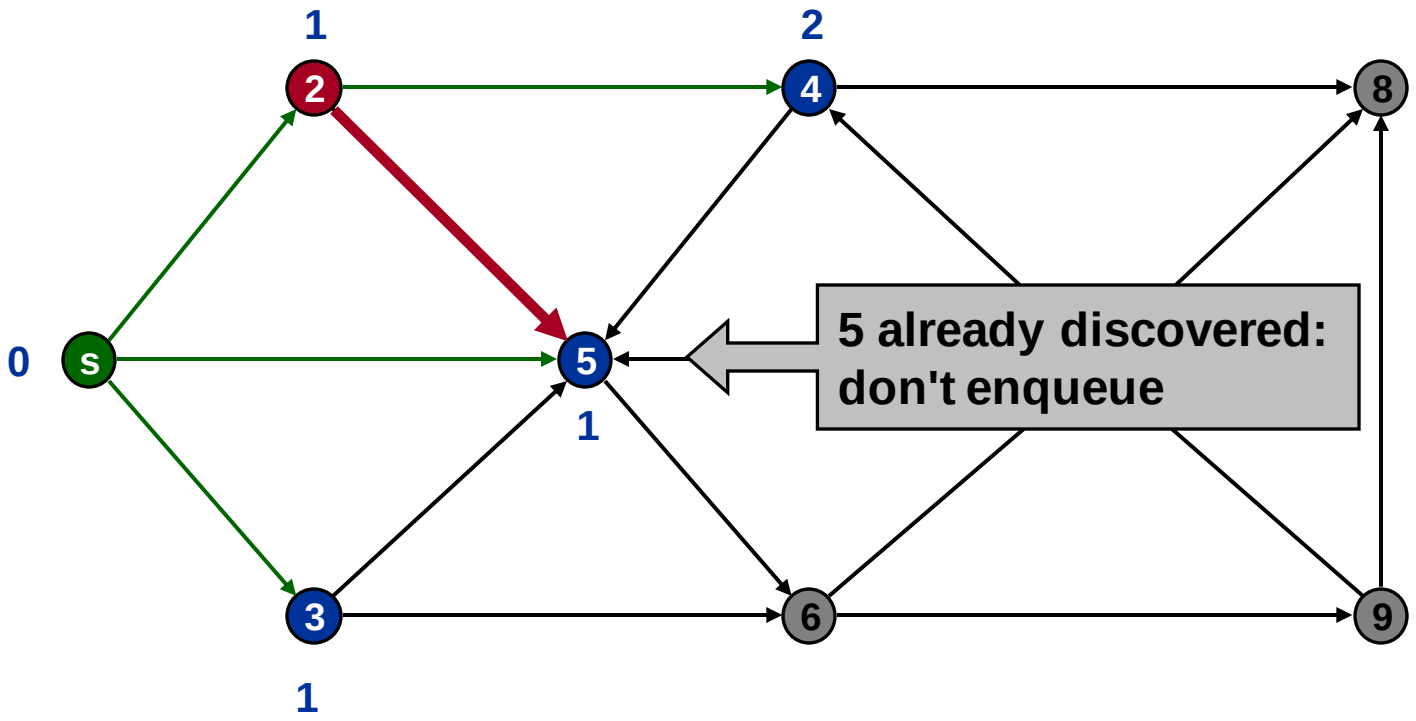
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5

Breadth First Search



Undiscovered

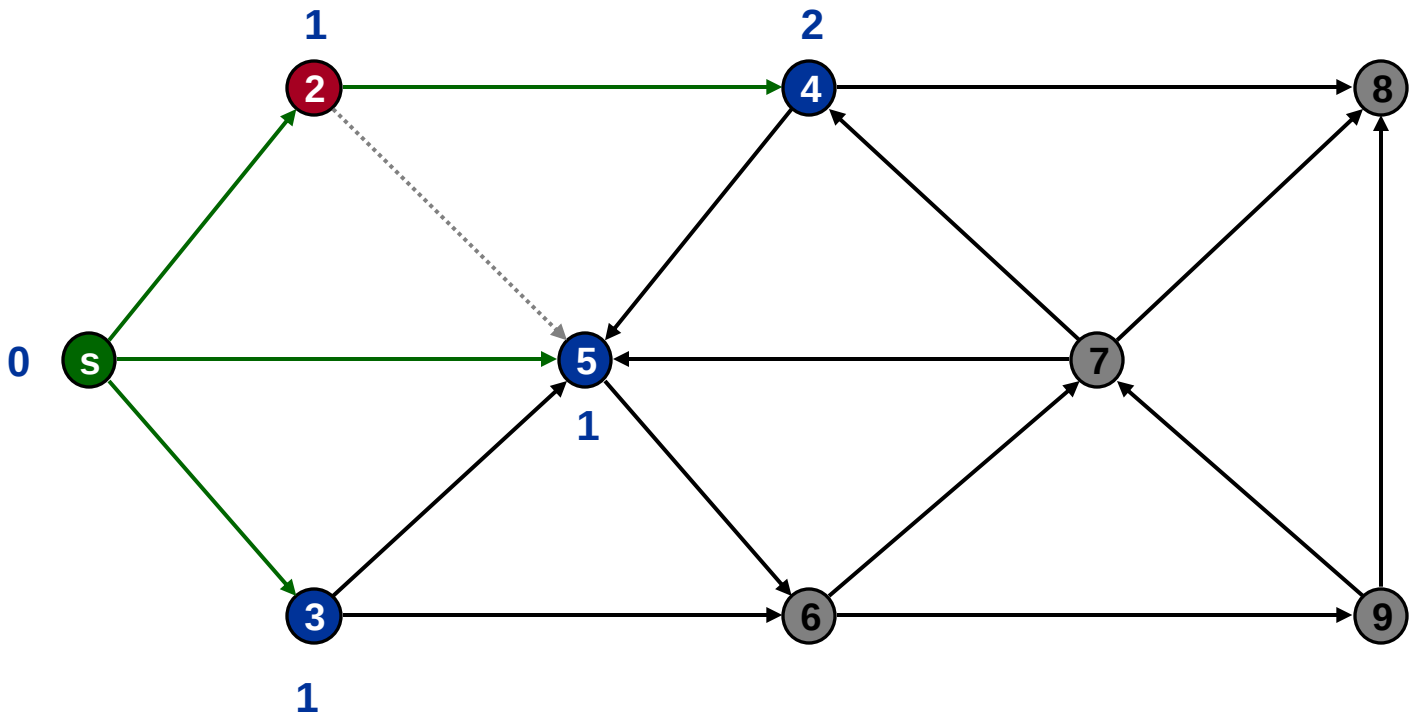
Discovered

Top of queue

Finished

Queue: 2 3 5 4

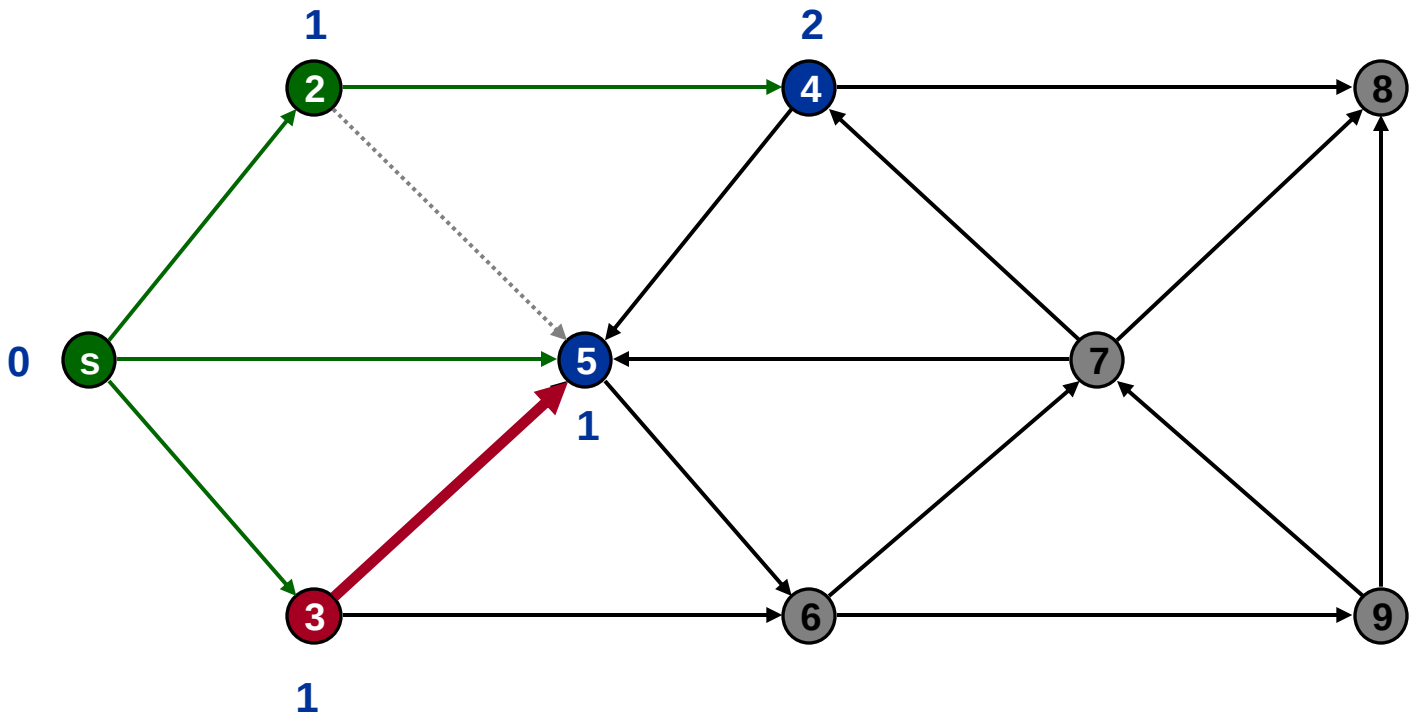
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5 4

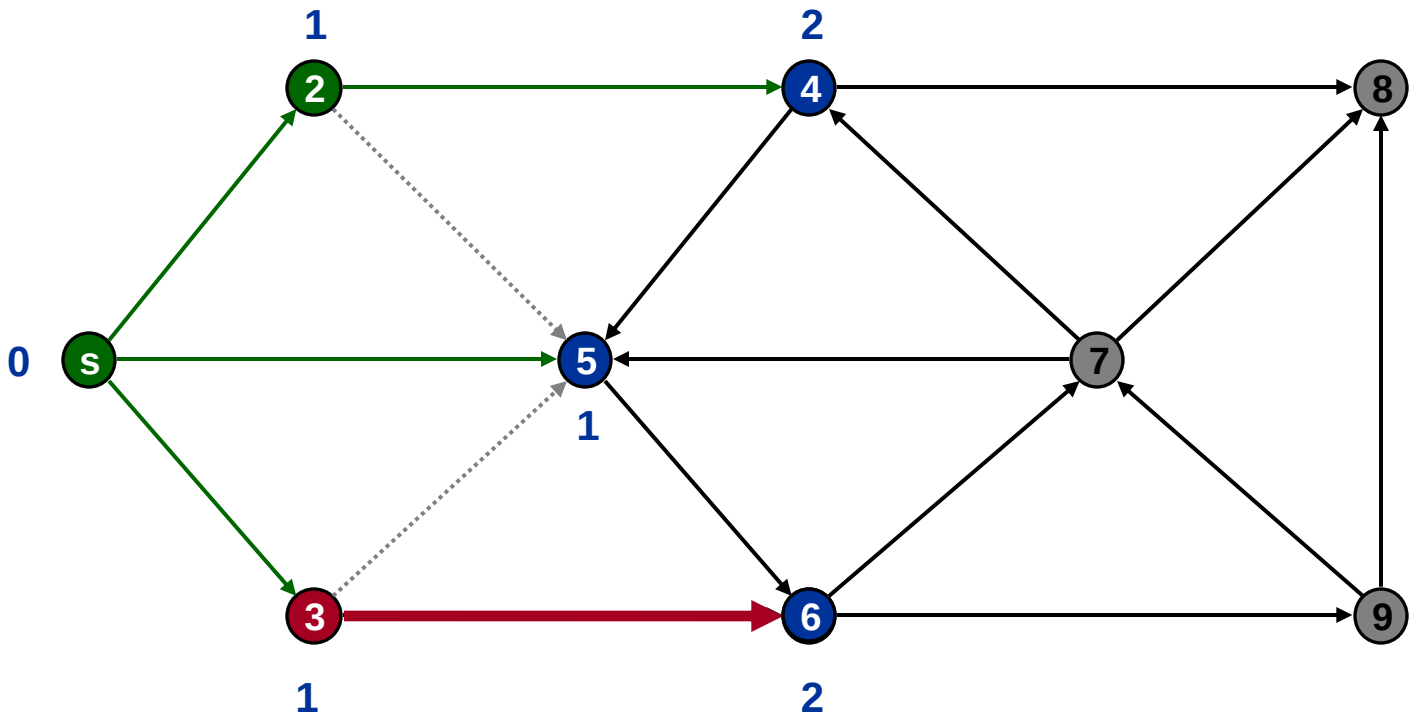
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4

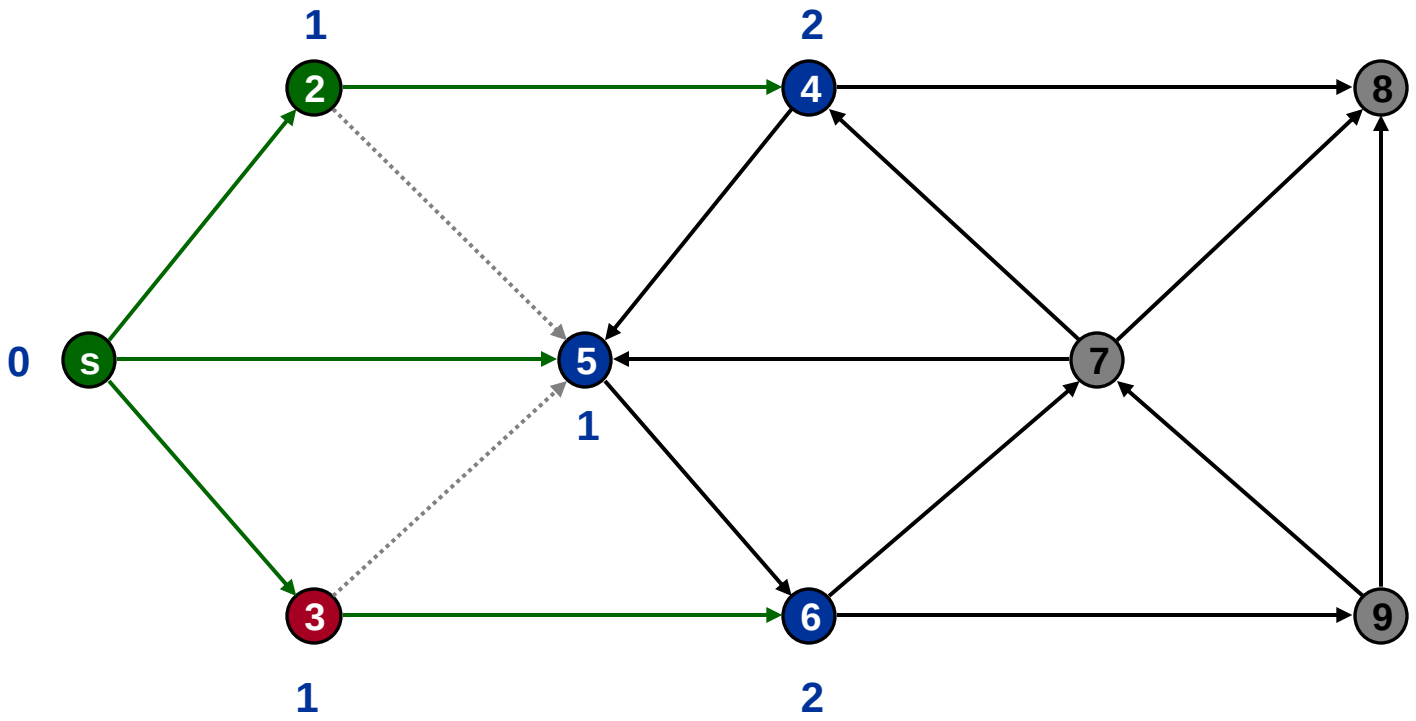
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4

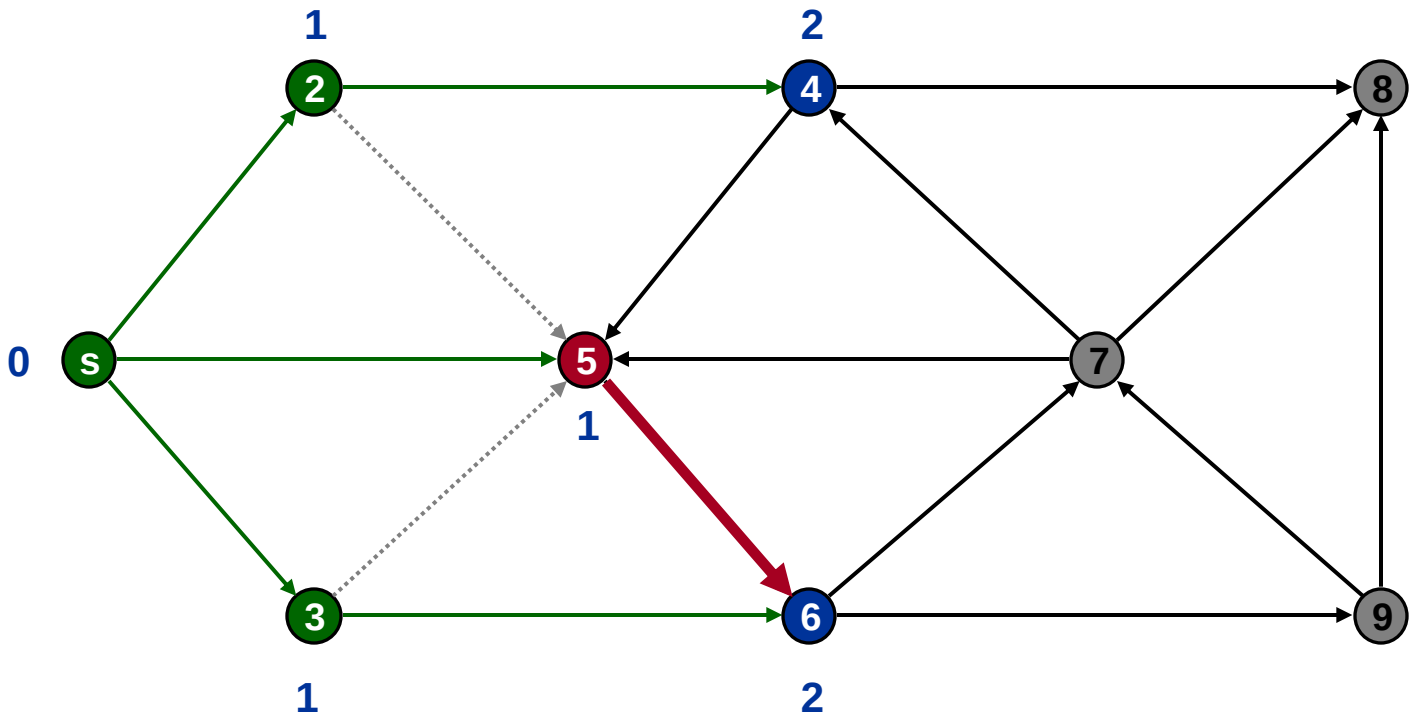
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4 6

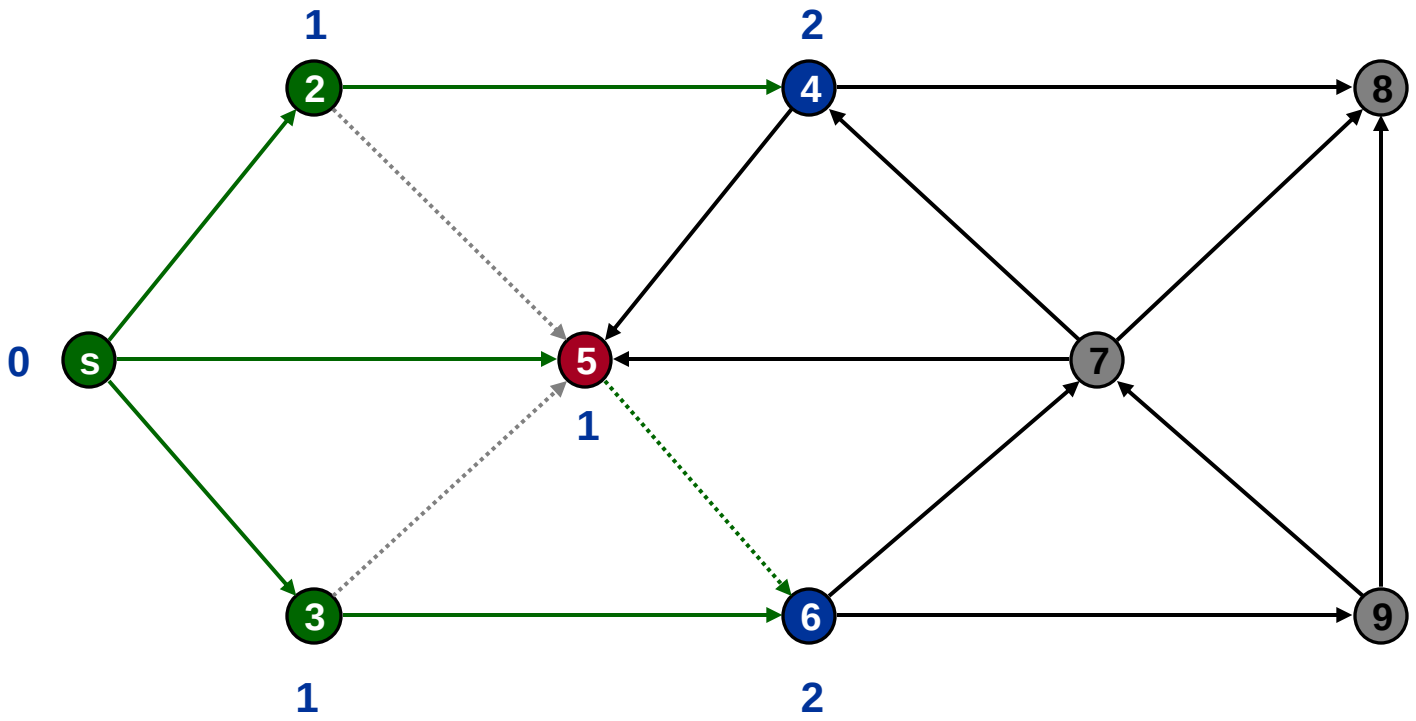
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 5 4 6

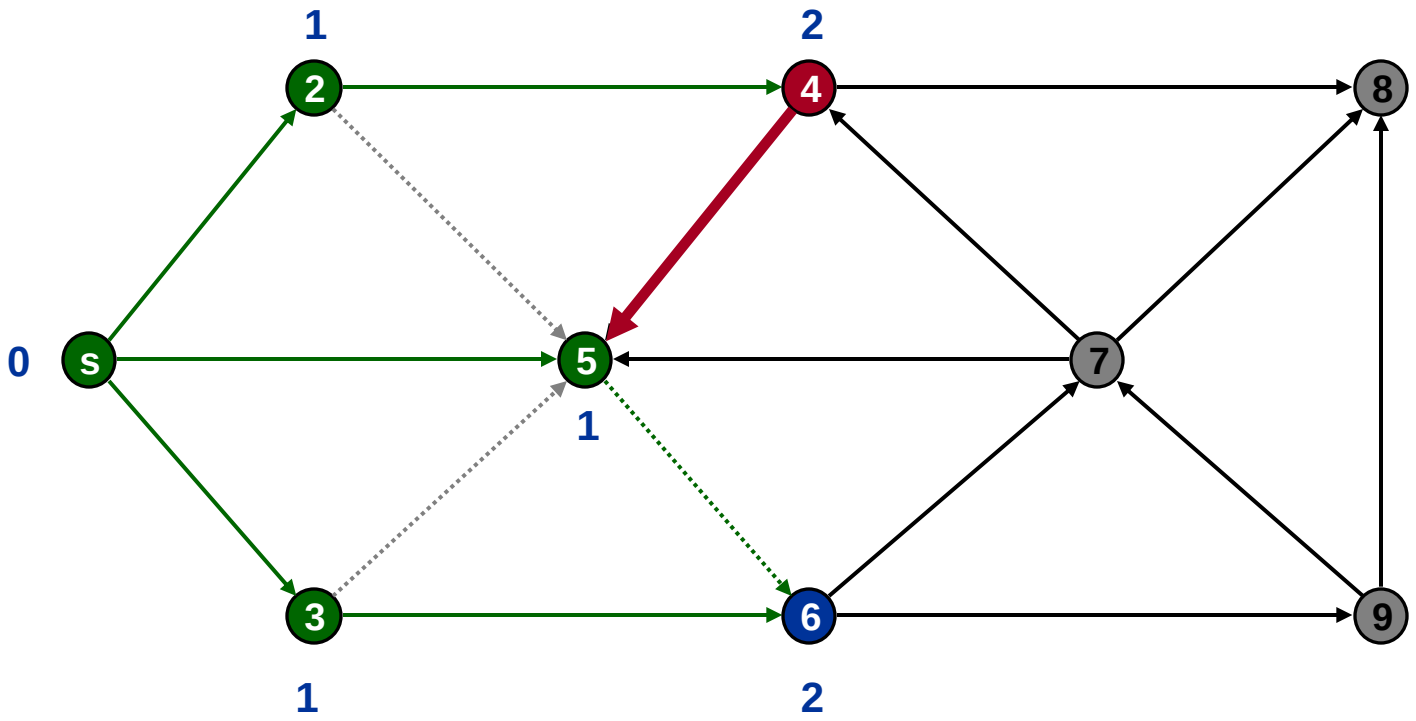
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 5 4 6

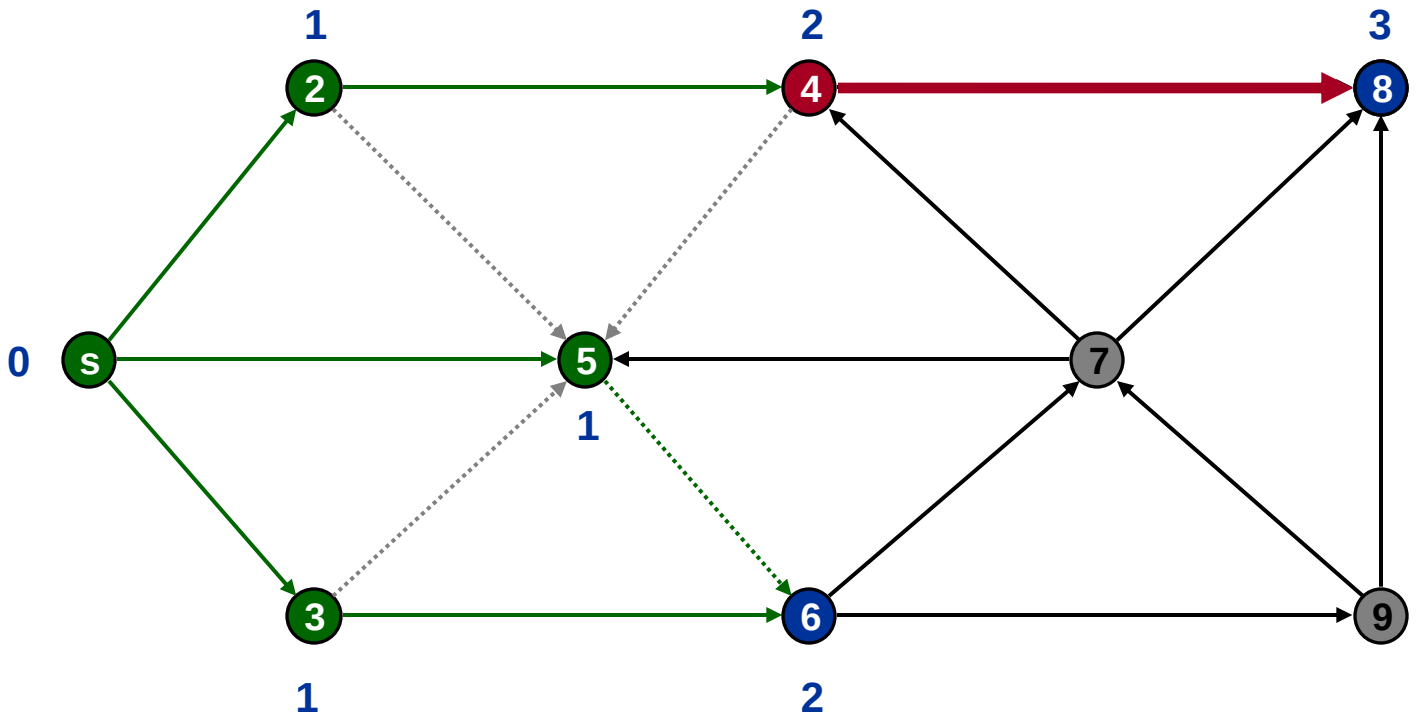
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6

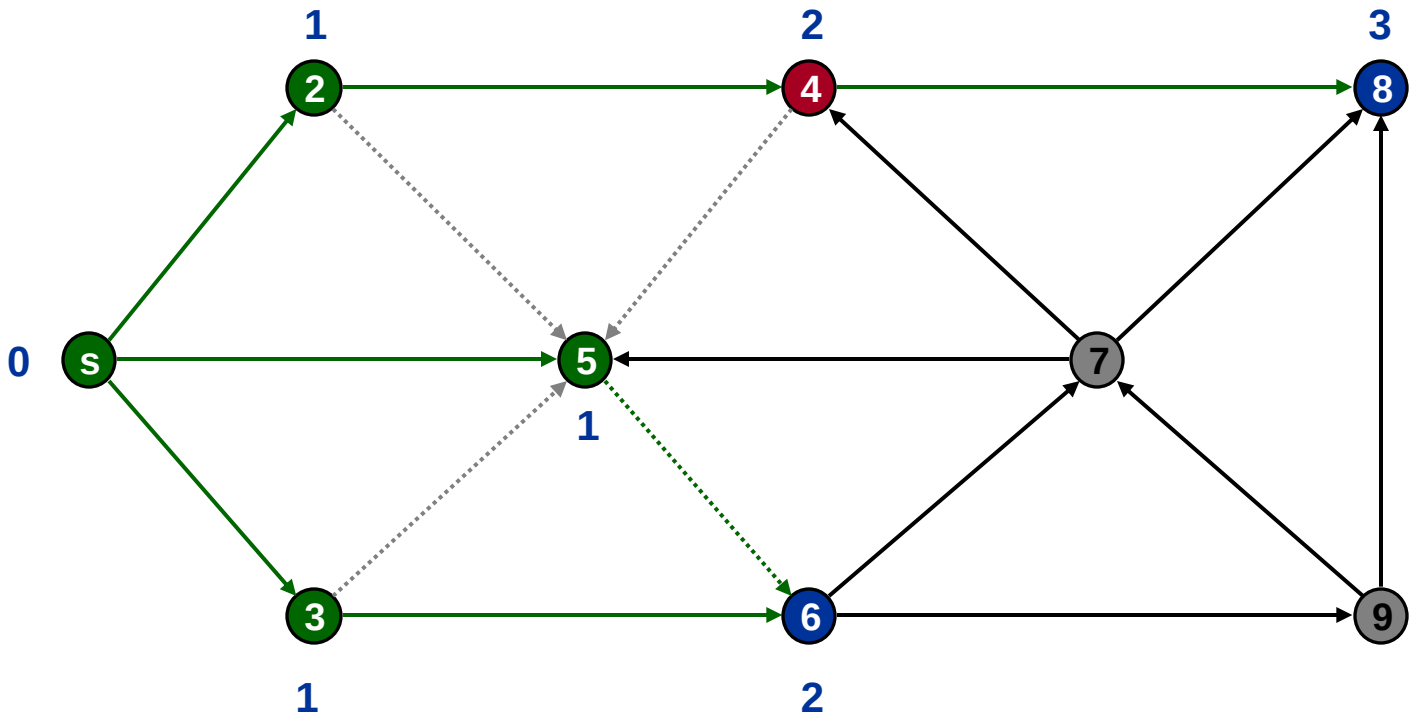
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6

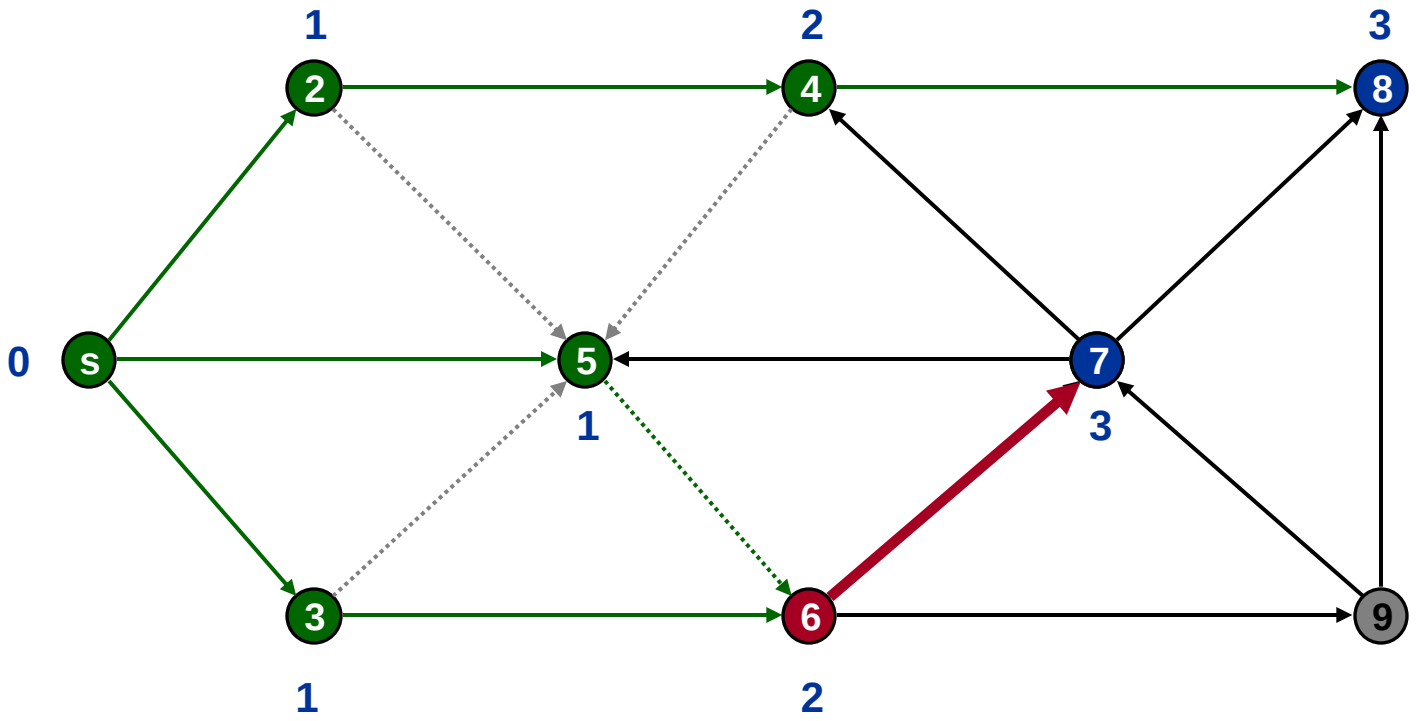
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6 8

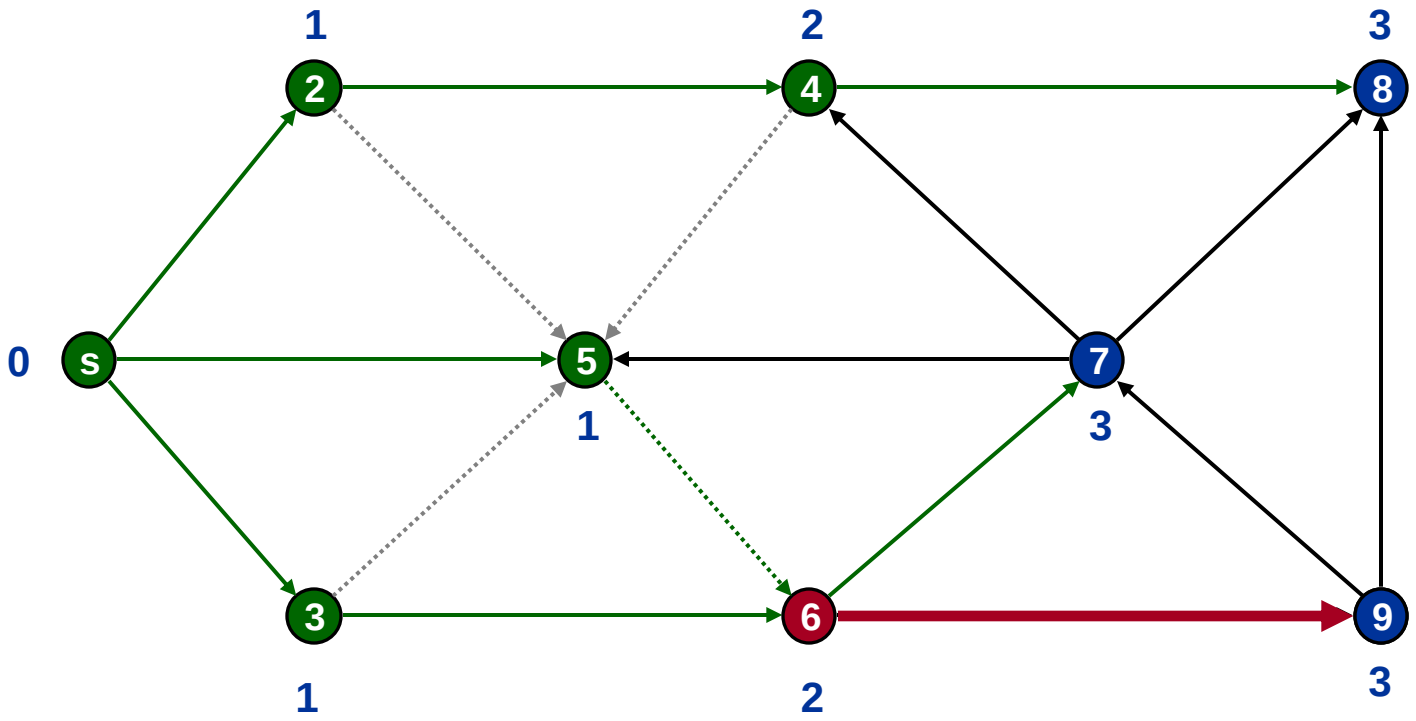
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8

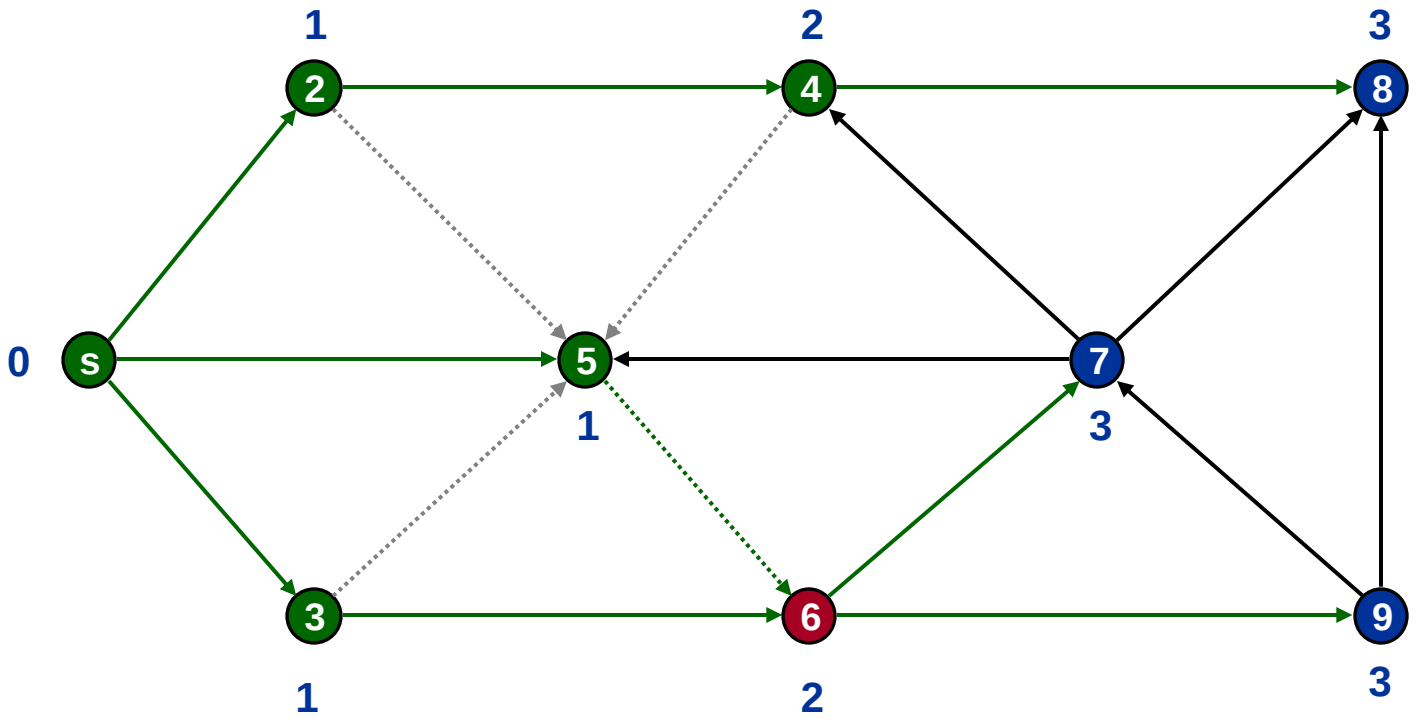
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7

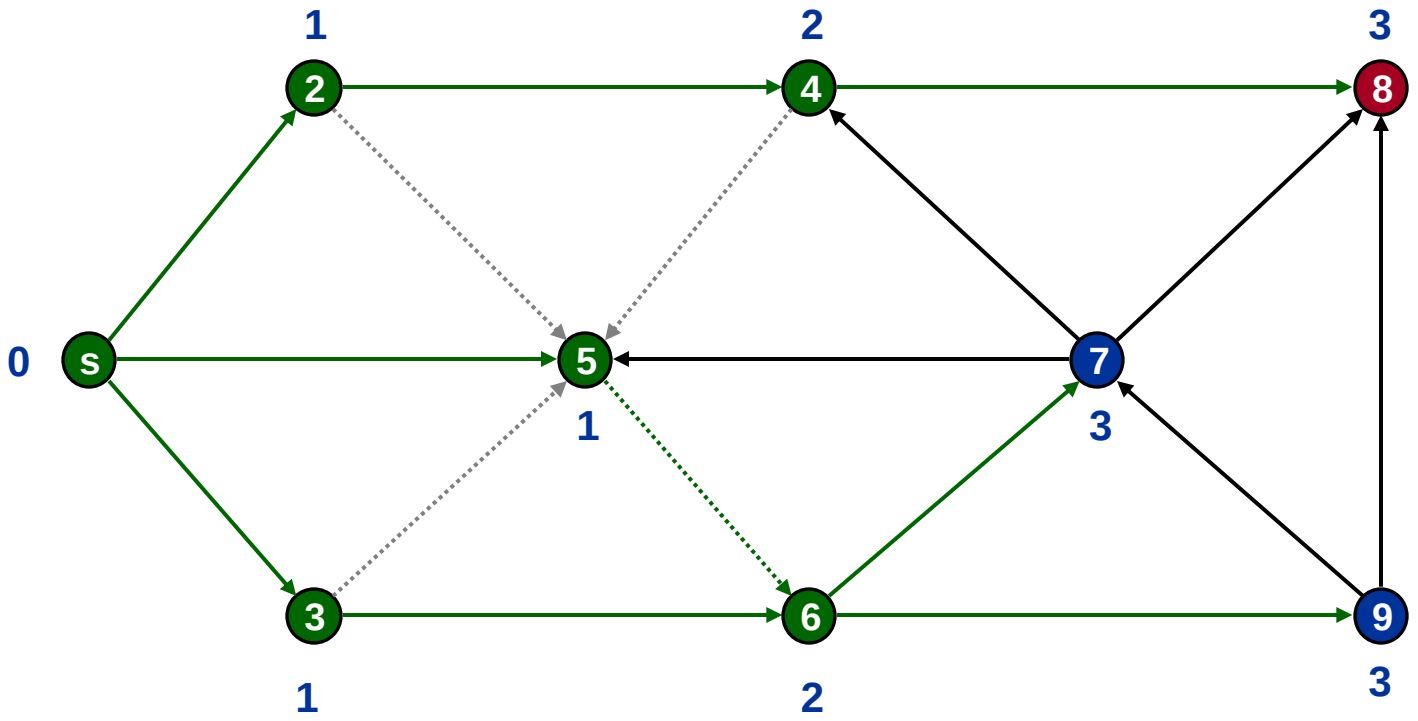
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7 9

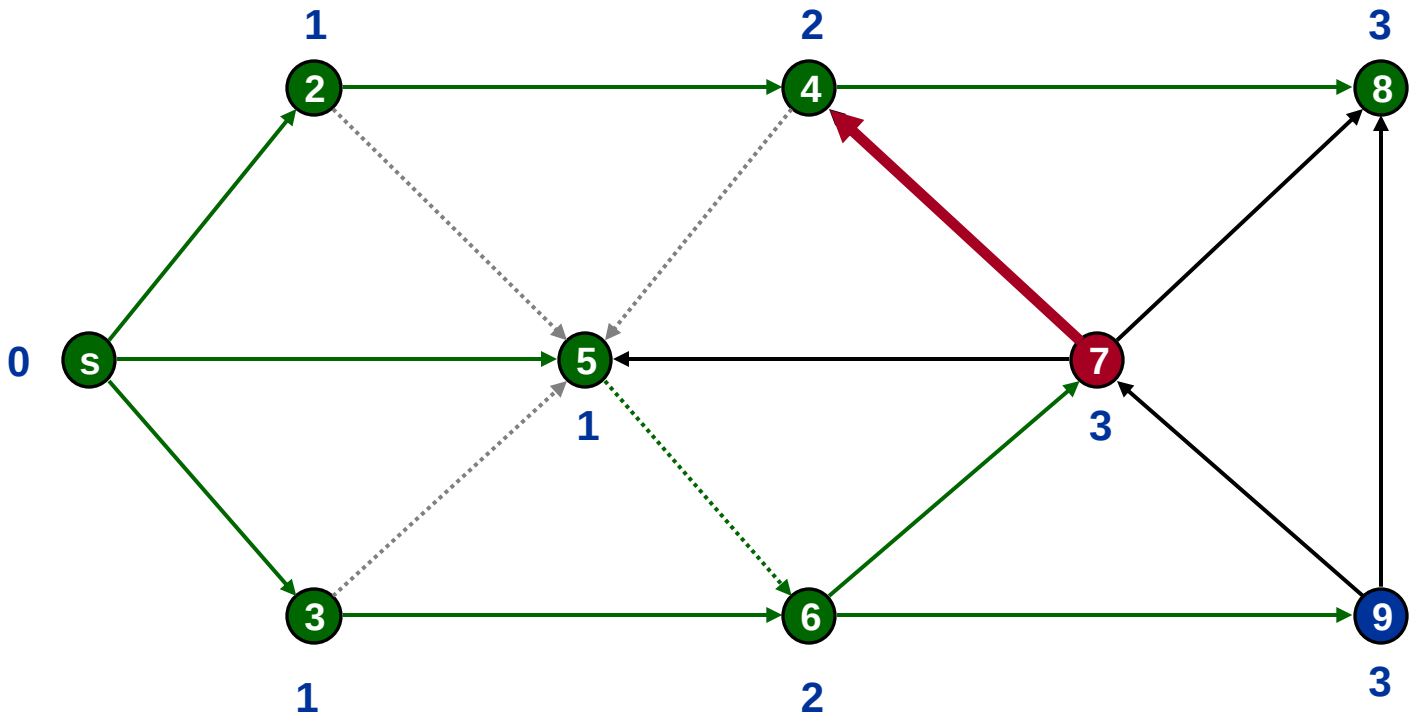
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 8 7 9

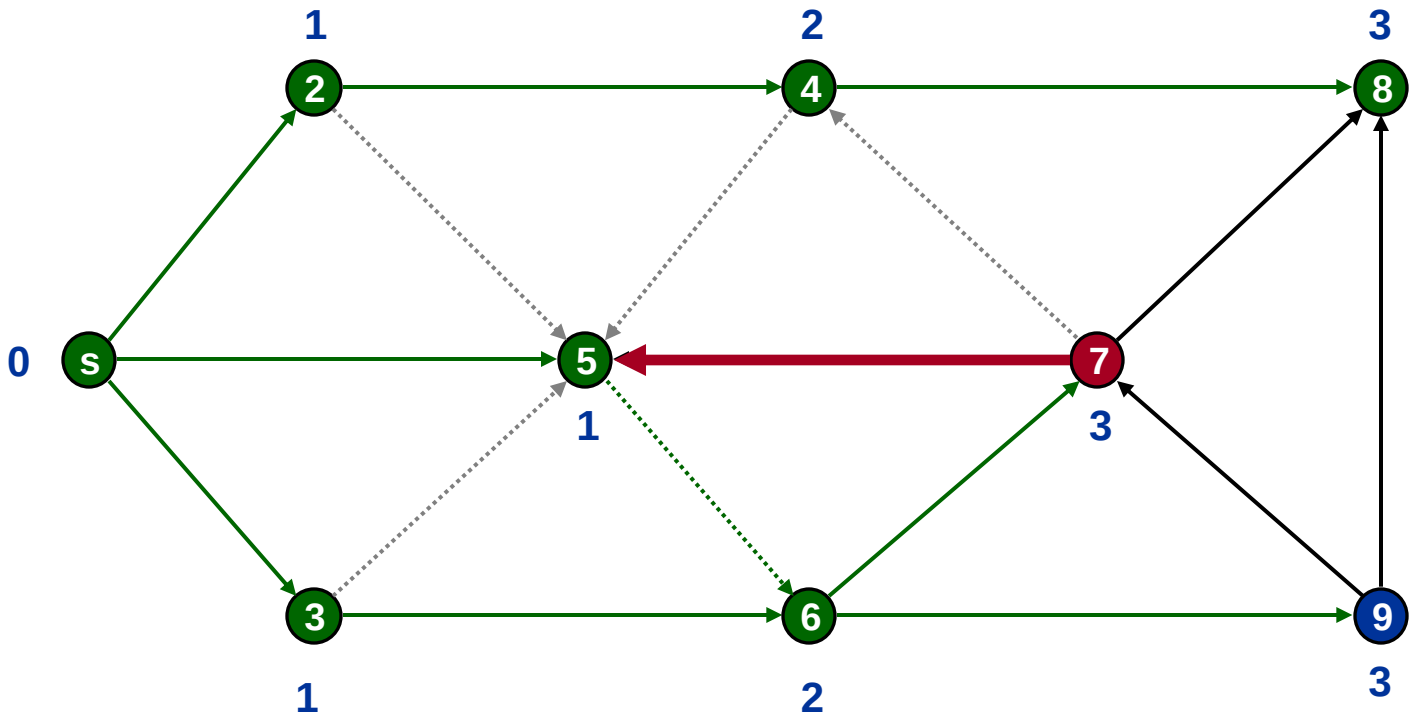
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

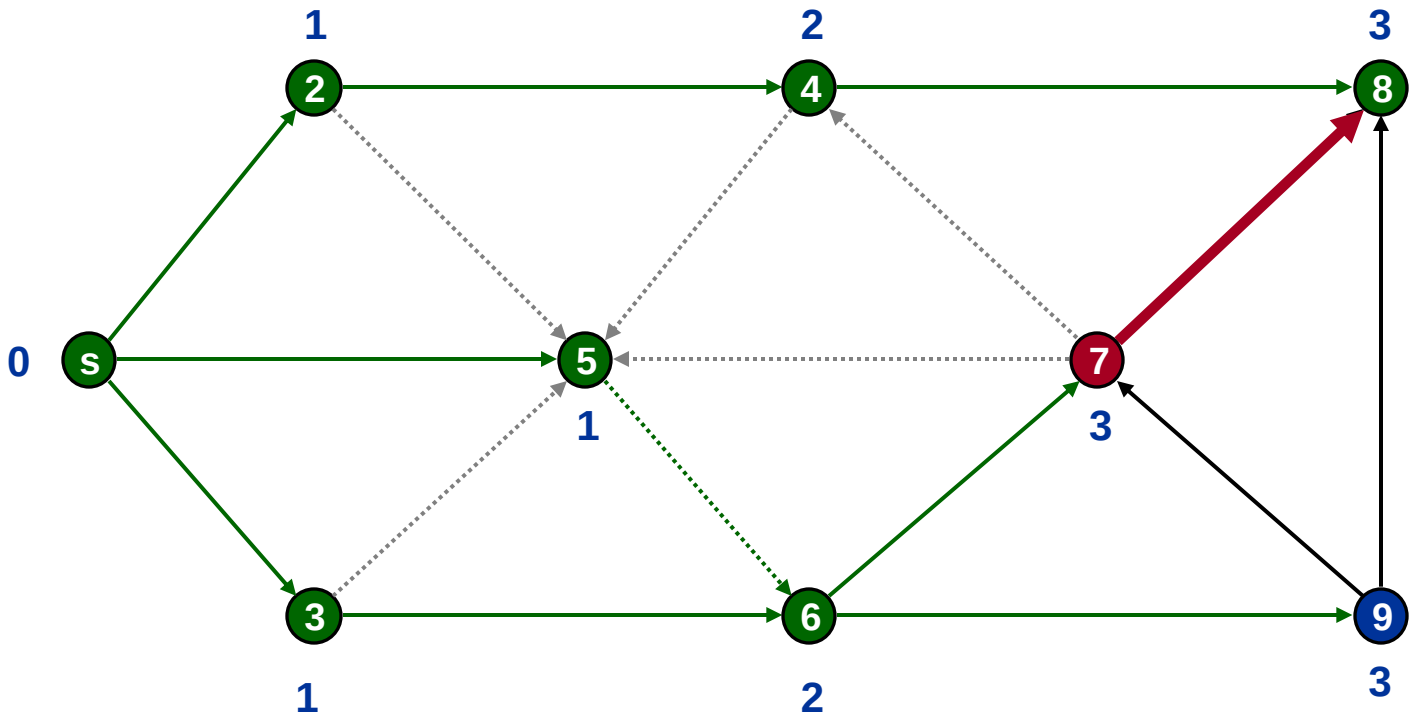
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

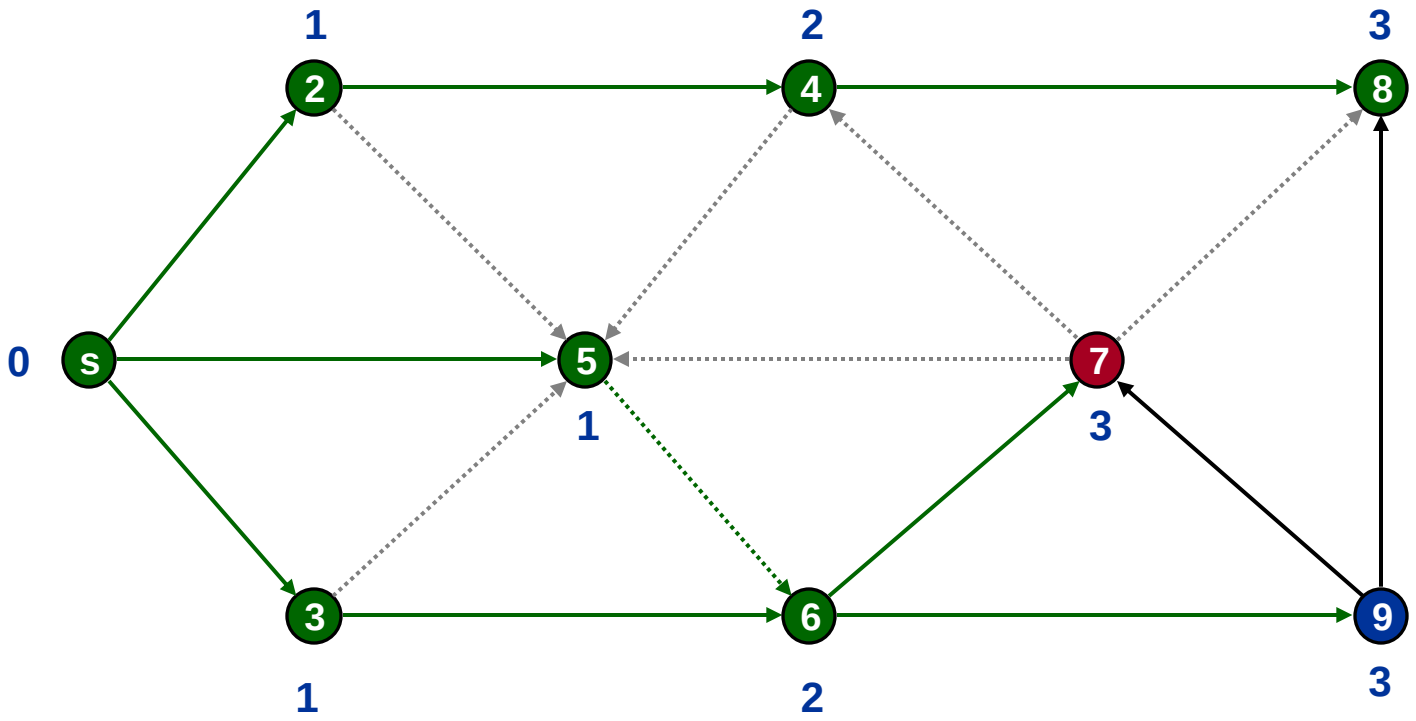
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

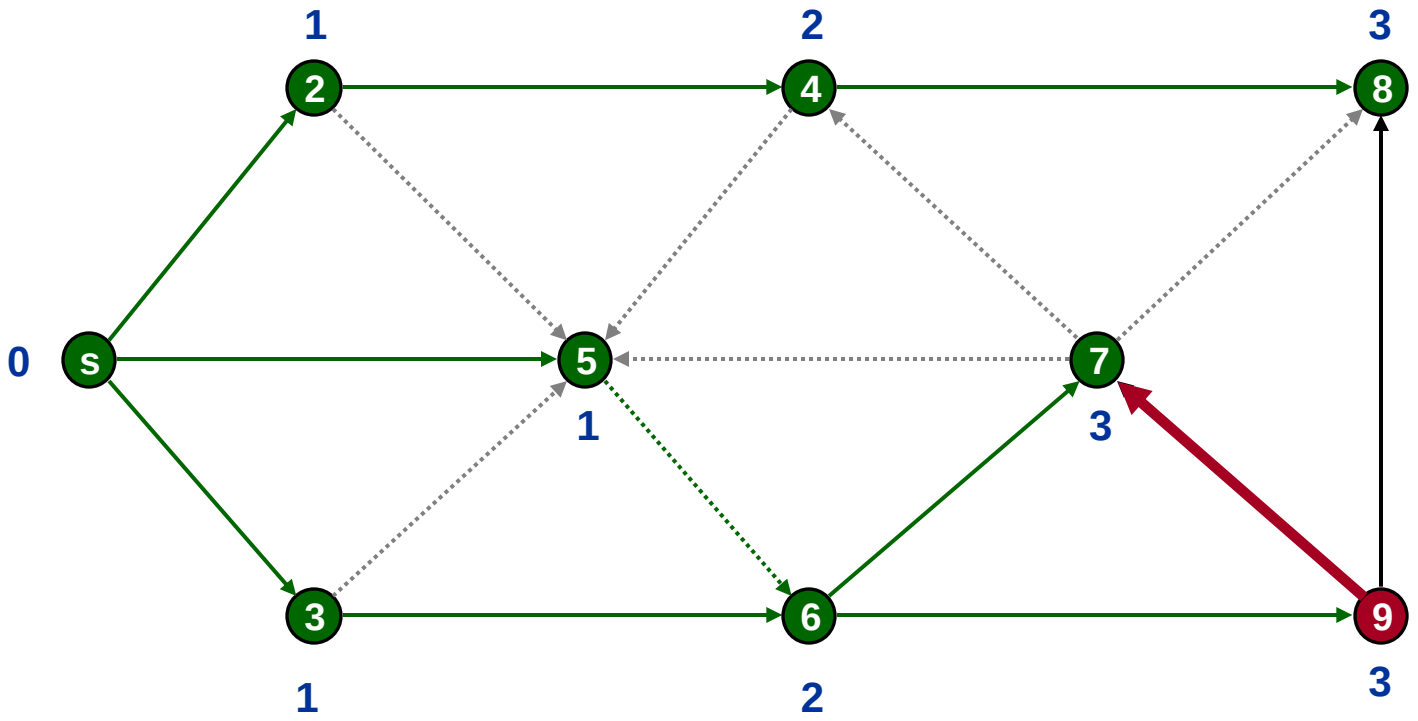
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

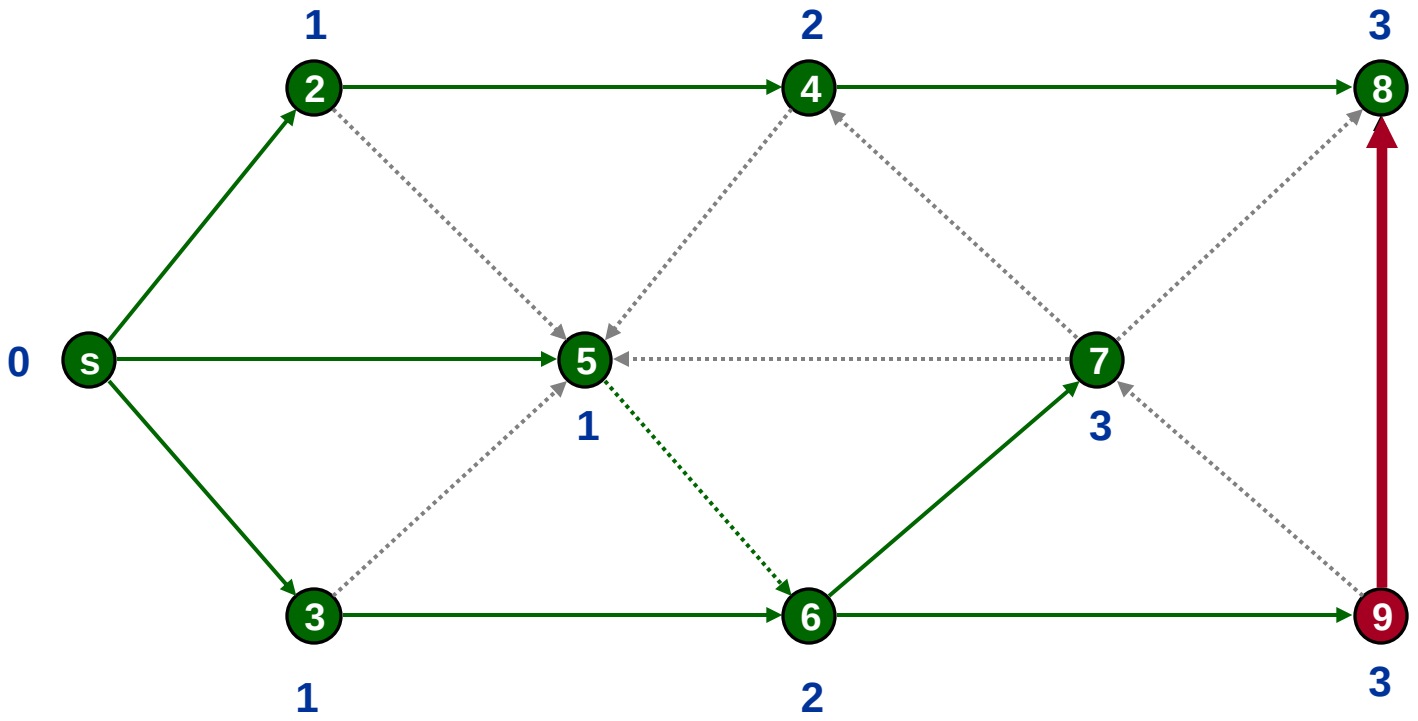
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

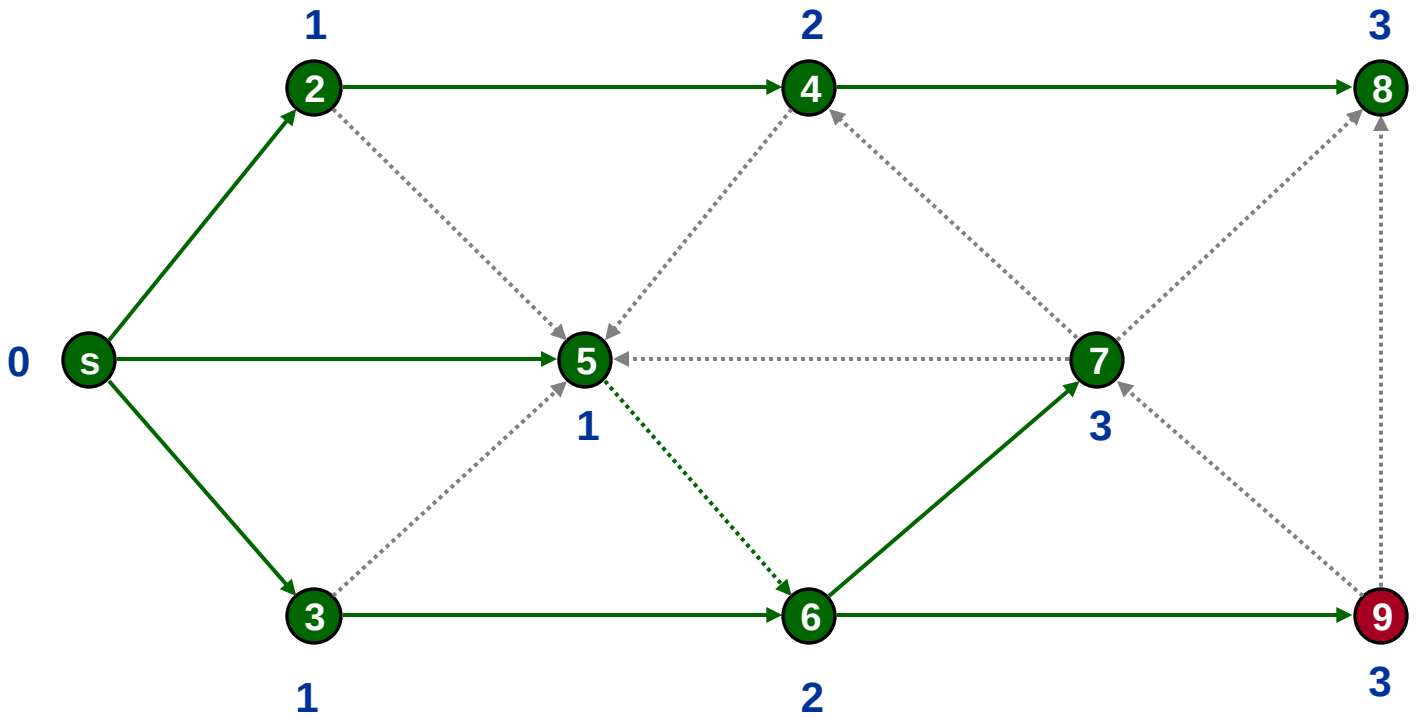
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

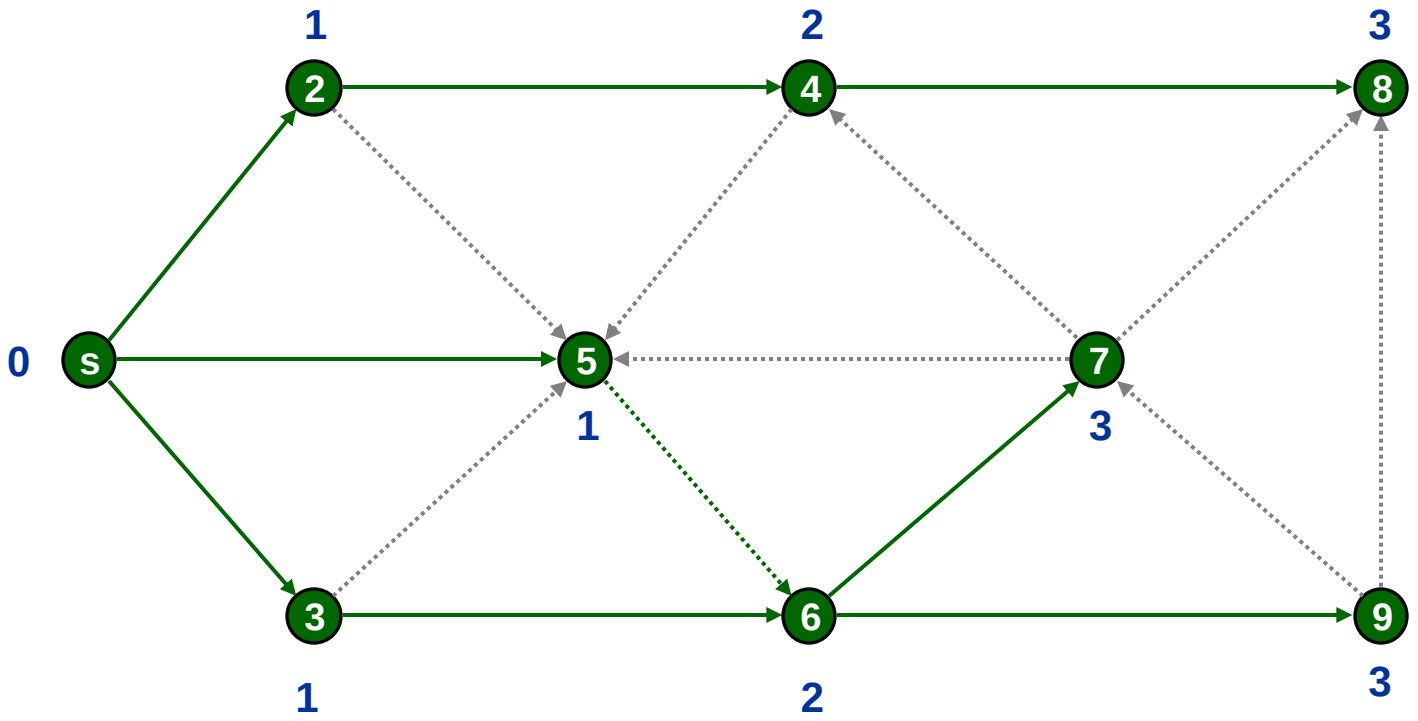
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

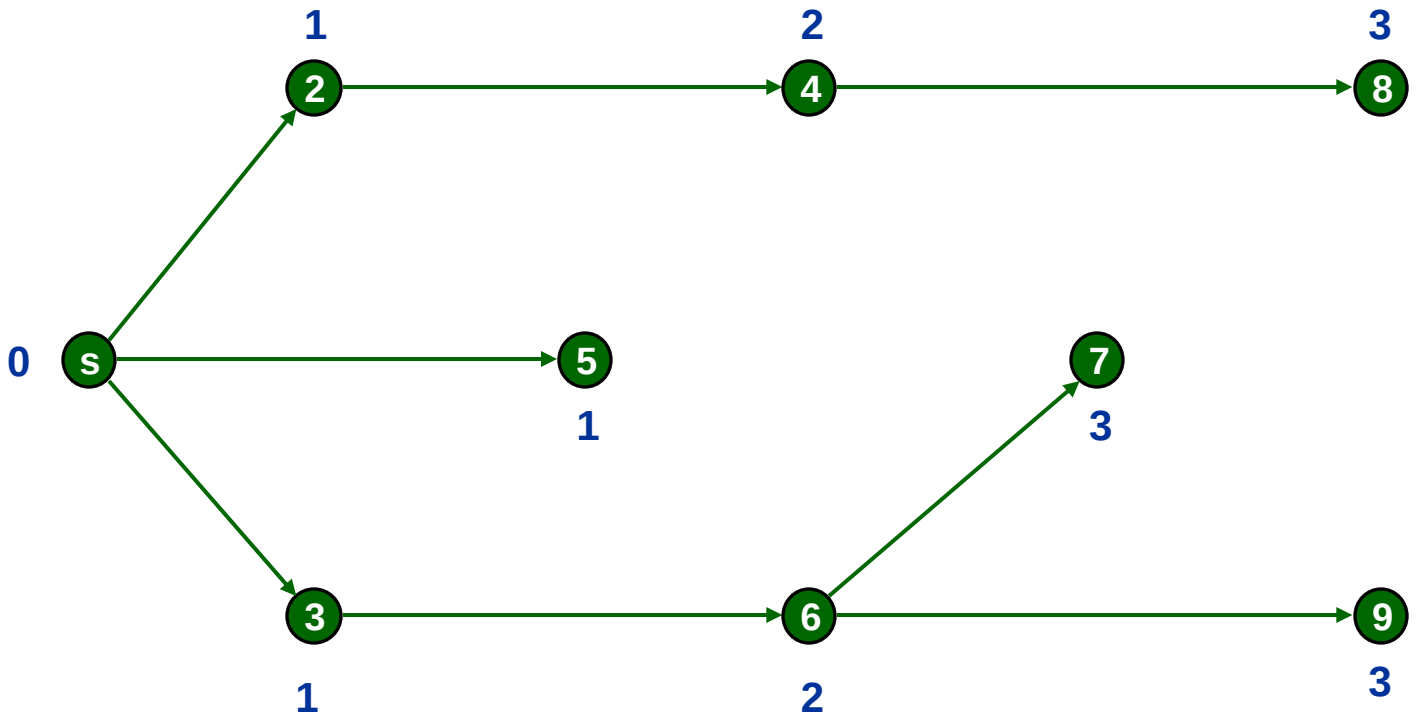
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

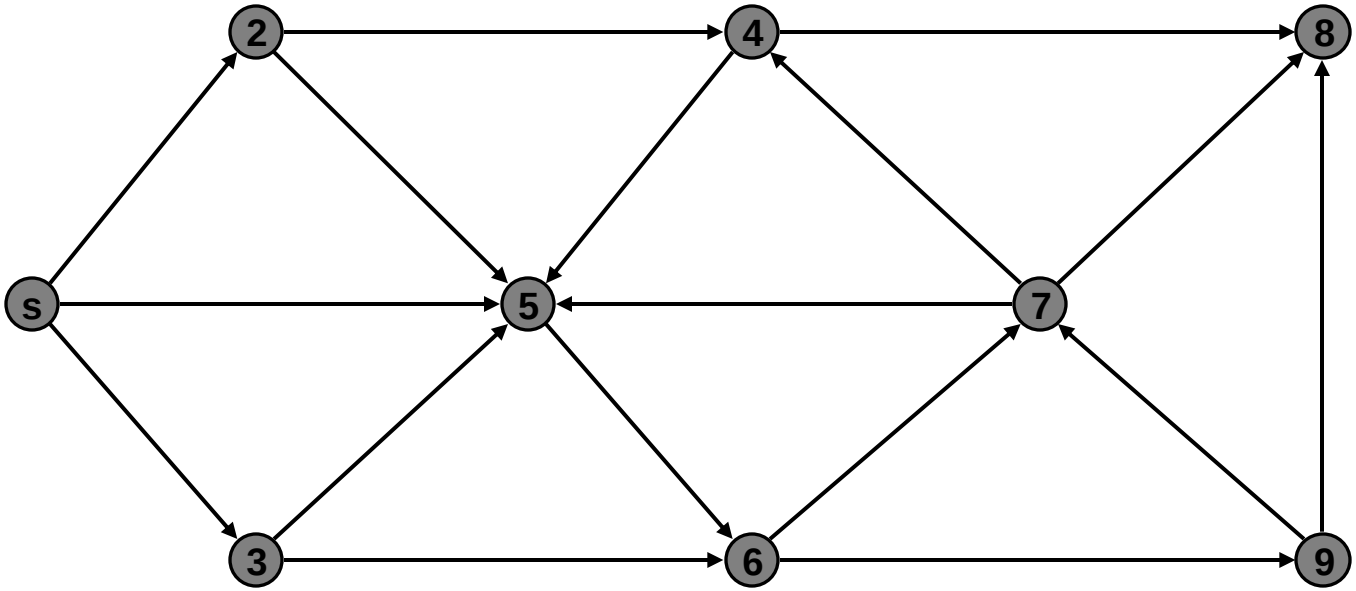
Queue:

Breadth First Search



Level Graph

Depth First Search



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

BFS/DFS

- Time Complexity:
- The operations of enqueuing and dequeuing takes $O(1)$ time, so total time devoted to queue operations is $O(V)$.
- Since sum of the lenghts of all adjacency lists is $\theta(E)$, the total time spent in scanning adjacency lists is $O(E)$.

- Total running time of BFS is $\theta(V+E)$.
- Same reasoning can be made for DFS.