Graph

DR. MAMATA P. WAGH

Graphs

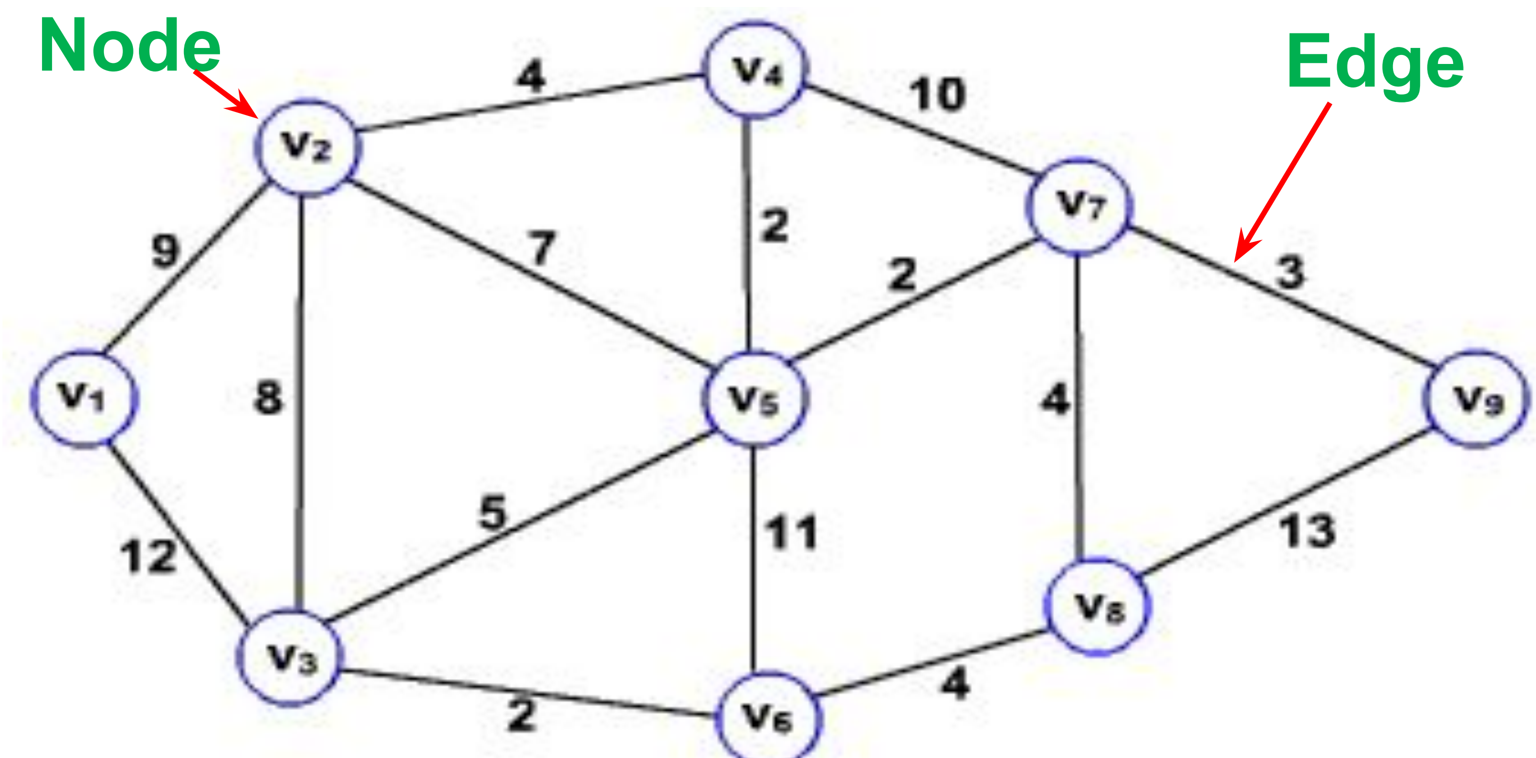
Topics to be Covered

1. **Graph terminology**
2. **Representation of graphs**
3. **Path matrix**
4. **BFS (breadth first search)**
5. **DFS (depth first search)**
6. **Topological sorting**
7. **Warshall's algorithm (shortest path algorithm.)**

Graph

1.Non linear Data structure
2.No of Nodes or vertices
3.Nodes connected with each other by different edges
4.Edges can be of different length

Q:- WHAT IS THE SHORTEST ROUTE BETWEEN CITY V1 and V9 ?



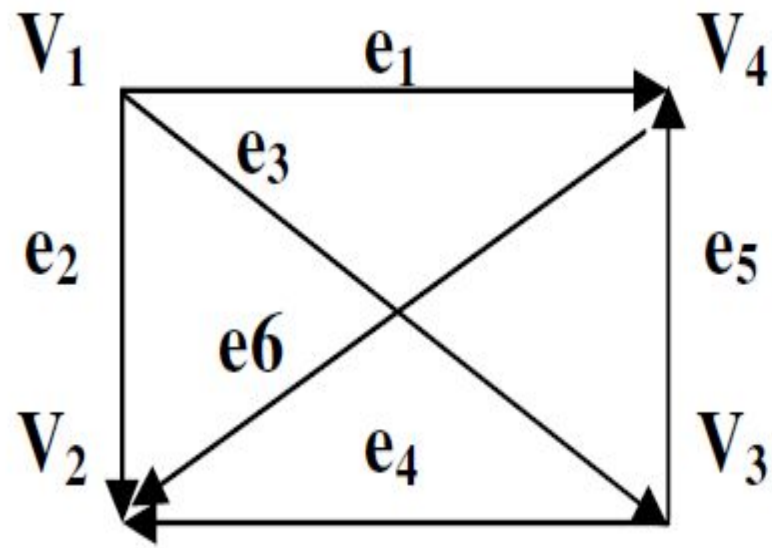
Definition:

- A graph G=(V,E) consists of a two finite non-empty sets
- A set V, called set of vertices or nodes.
- A set of E, called set of edges such that each edge e in E is identified with a unique pair of nodes in V, denoted by e=[u,v].
- Graphs may be either directed or undirected.

Example

$V=\{v_1, v_2, v_3, v_4\}$

$E=\{ (v_1, v_4), (v_1, v_2), (v_1, v_3), (v_3, v_2), (v_3, v_4), (v_4, v_2) \}$



Representation of graph

Two ways

1)sequential or matrix representation

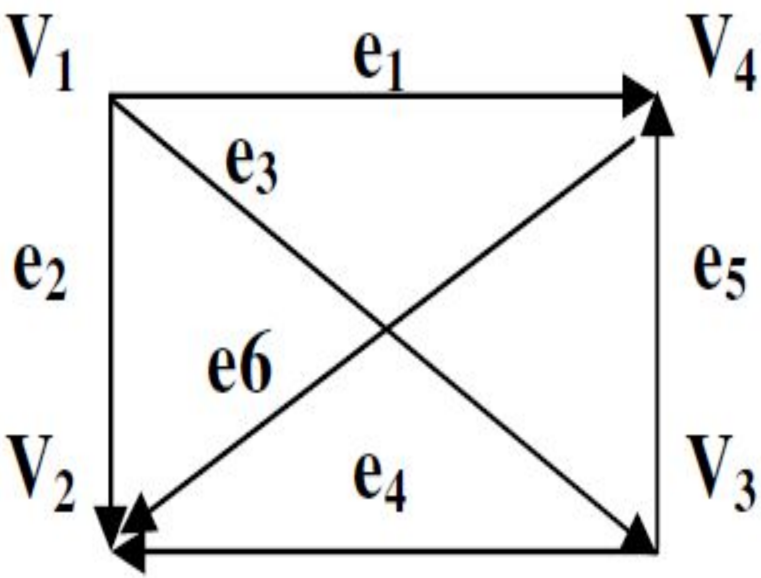
- Adjacency matrix
- Incidence matrix

2) Adjacent list or linked list representation.

Adjacency matrix

- one row and one column for each vertex.
- E_{ij} =Edge between node v_i and v_j .
- Elements are either 0 or 1.

- if (E_{ij} exists) then A_{ij} =1 Else A_{ij} =0



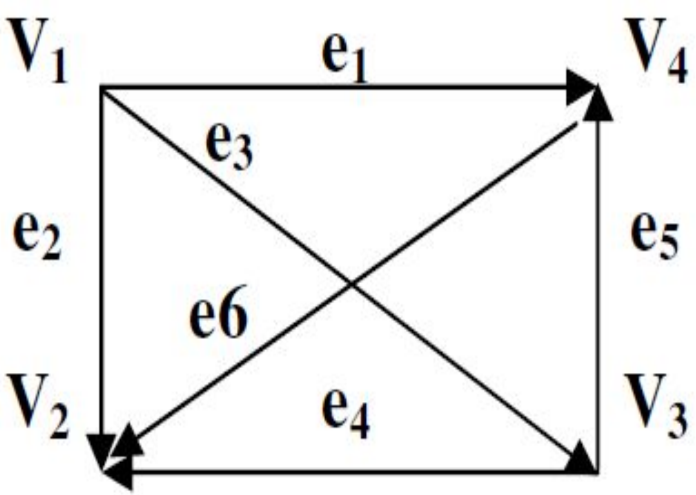
A=

	V_1	V_2	V_3	V_4
V_1	0	1	1	1
V_2	0	0	0	0
V_3	0	1	0	1
V_4	0	1	0	0

E 14=e1

Incidence matrix (I)

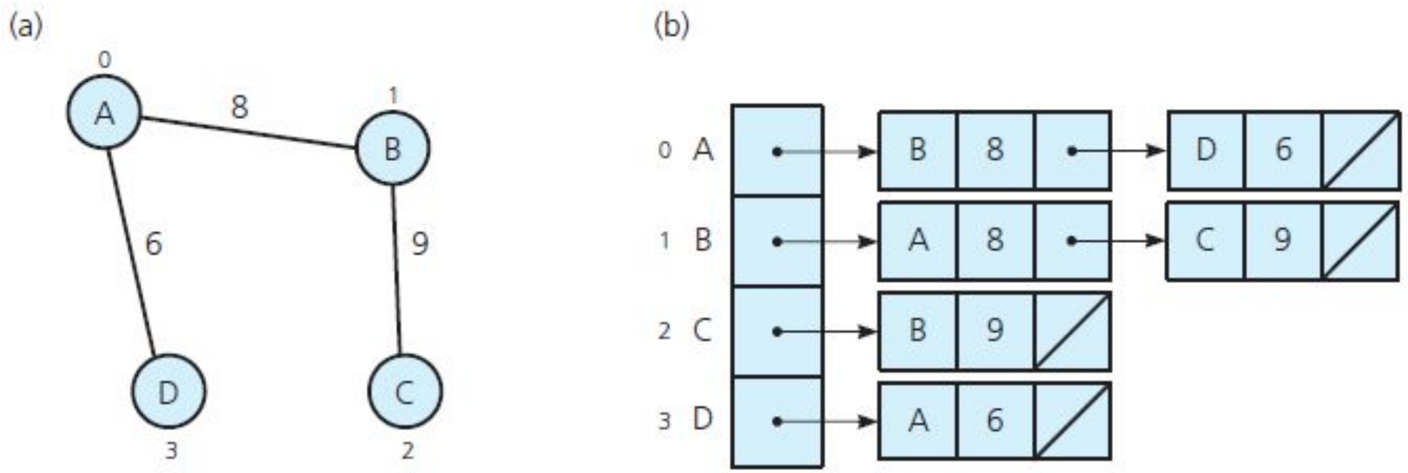
- Matrix which shows stating node and ending node of an edge in a graph.
- Rows are for nodes and column are for edges.
- Each column consists of one 1 and one -1 .
- **For a column $e_x = E_{ij}$, $I_{ix} = 1$ and $I_{jx} = -1$,** all other value 0's.



I=

	e_1	e_2	e_3	e_4	e_5	e_6
V_1	1	1	1	0	0	0
V_2	0	-1	0	-1	0	-1
V_3	0	0	-1	1	1	0
V_4	-1	0	0	0	-1	1

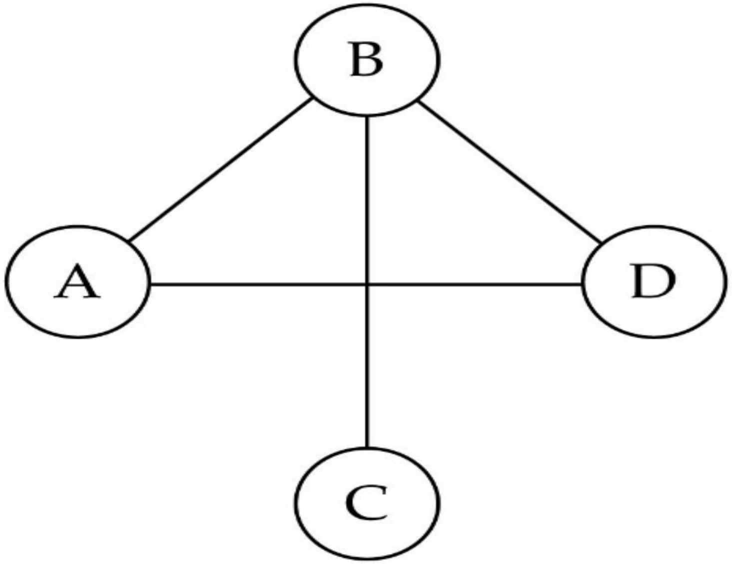
Adjacent list or linked list representation



- **FIGURE (a) A weighted undirected graph and (b) its adjacency list**

- **It is a linked list representation which shows the nodes and its adjacent nodes**
- **Each node acts as start node for its adjacent list.**
- **There is no order in arranging the adjacent nodes.**

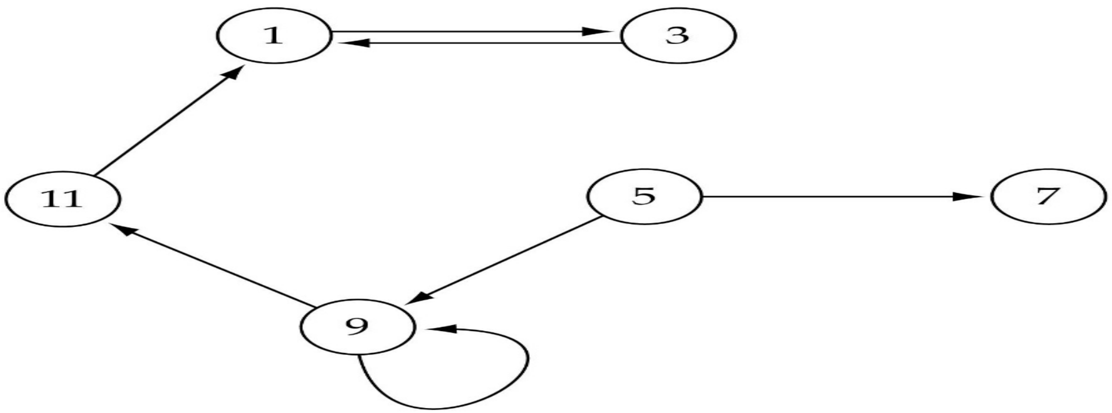
Directed vs. undirected graphs



- When the edges in a graph have no direction, the graph is called *undirected graph*
- $E_{AB}=E_{BA}$

Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

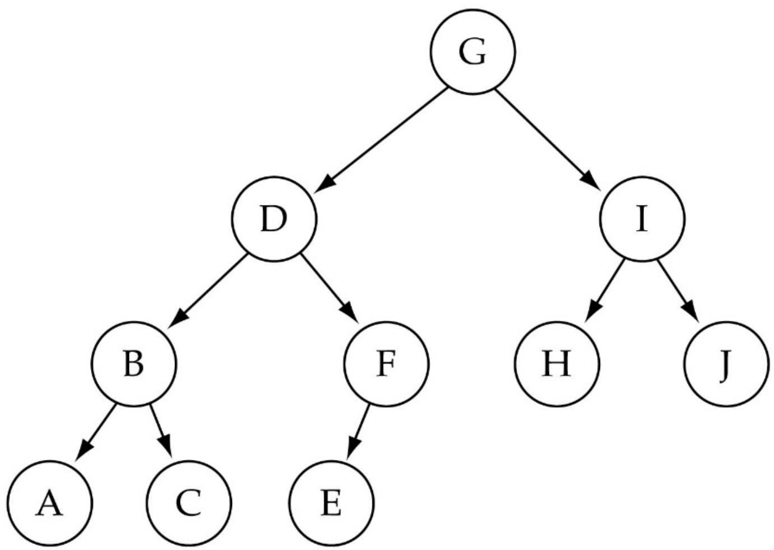


•*Warning*: if the graph is directed, the order of the vertices in each edge is important !!
• (11,1) is correct
• (1,11) is wrong
•(9,9) is called a self loop or loop

Trees vs graphs

- Trees are special cases of graphs!!

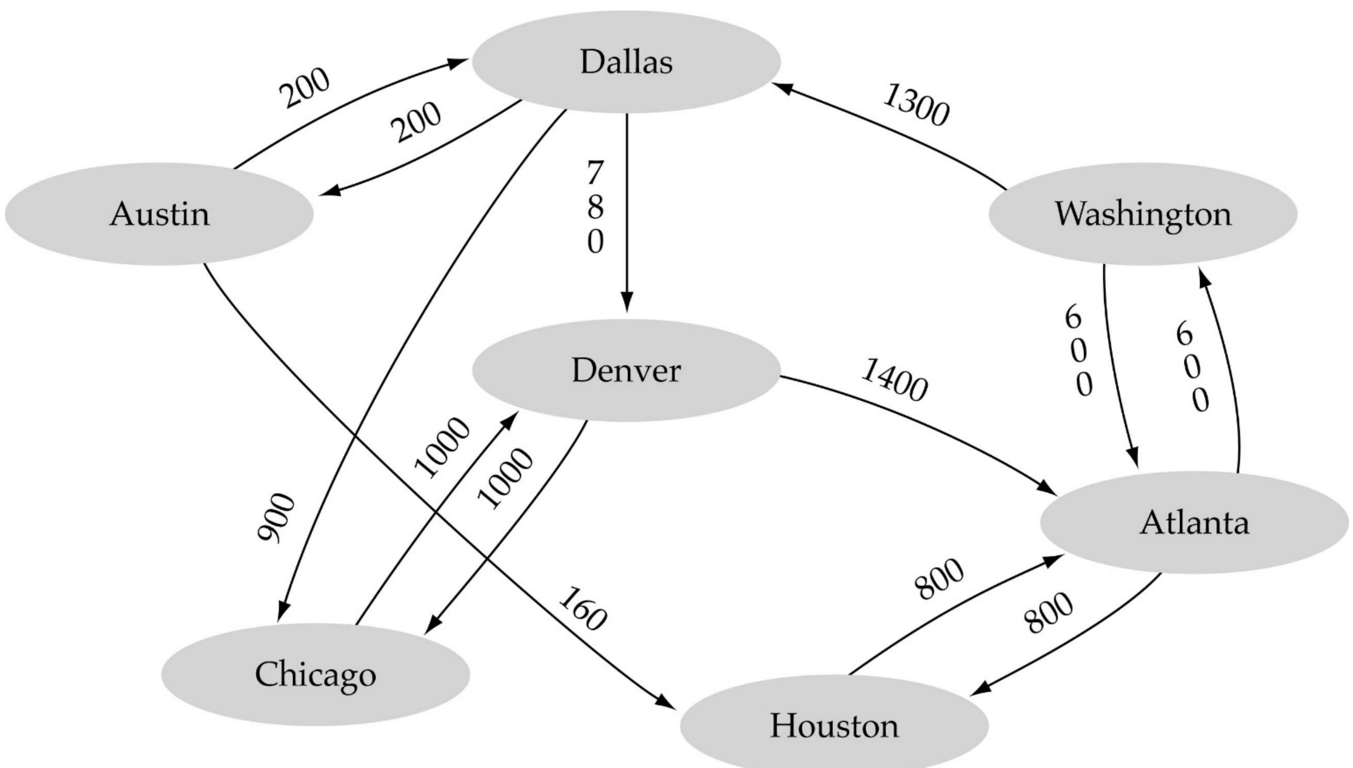
(c) Graph3 is a directed graph.



V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, J), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

Graph terminology (cont.)

- <u>Weighted graph</u>: a graph in which each edge carries a value

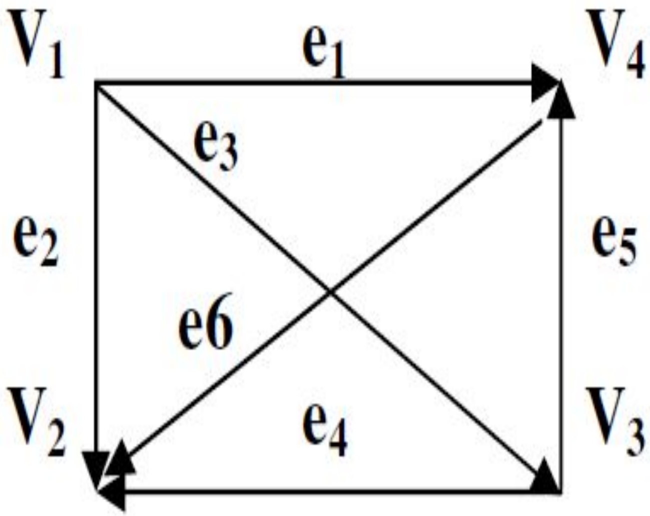


Degree of a graph

For an undirected graph , degree of a node is defined as the total no of edges originating from that node.

But for a directed graph a node has 2 types of degrees
(i)Out-degree:- The no of edges beginning at the node.
(ii)In-degree :- The no of edges ending at the node.

	In-degree	Out-degree
v1	0	3
v2	3	0
v3	1	2
v4	2	1



Degree of graph continues…….

For a directed graph like in previous slide

- Sum of in-degrees of all nodes= sum of the out-degrees.

If direction is ignored

- Degree = in-degree + out-degree
- Sum of degree of all node = 2 × total no of edges

Ex:- 12 = 2 × 6

- Source node :- The node having zero in-degree , but positive out degree.

Ex:- V1

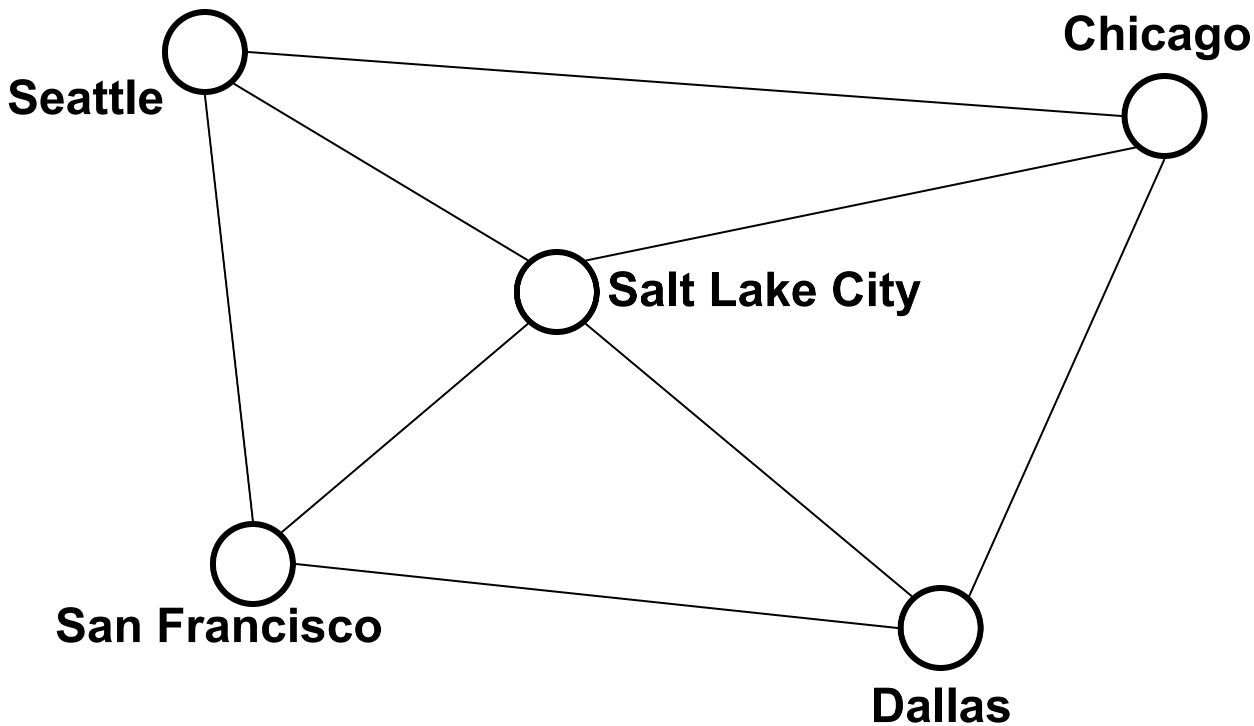
- Sink node :- The node having zero out-degree , but positive in-degree.

Ex:-V2

Paths and Cycles

A *path* is a list of vertices $\{v_1, v_2, \ldots, v_n\}$ such that $(v_i, v_{i+1}) \in E$ for all $0 \le i < n$.

A *cycle* is a path that begins and ends at the same node.

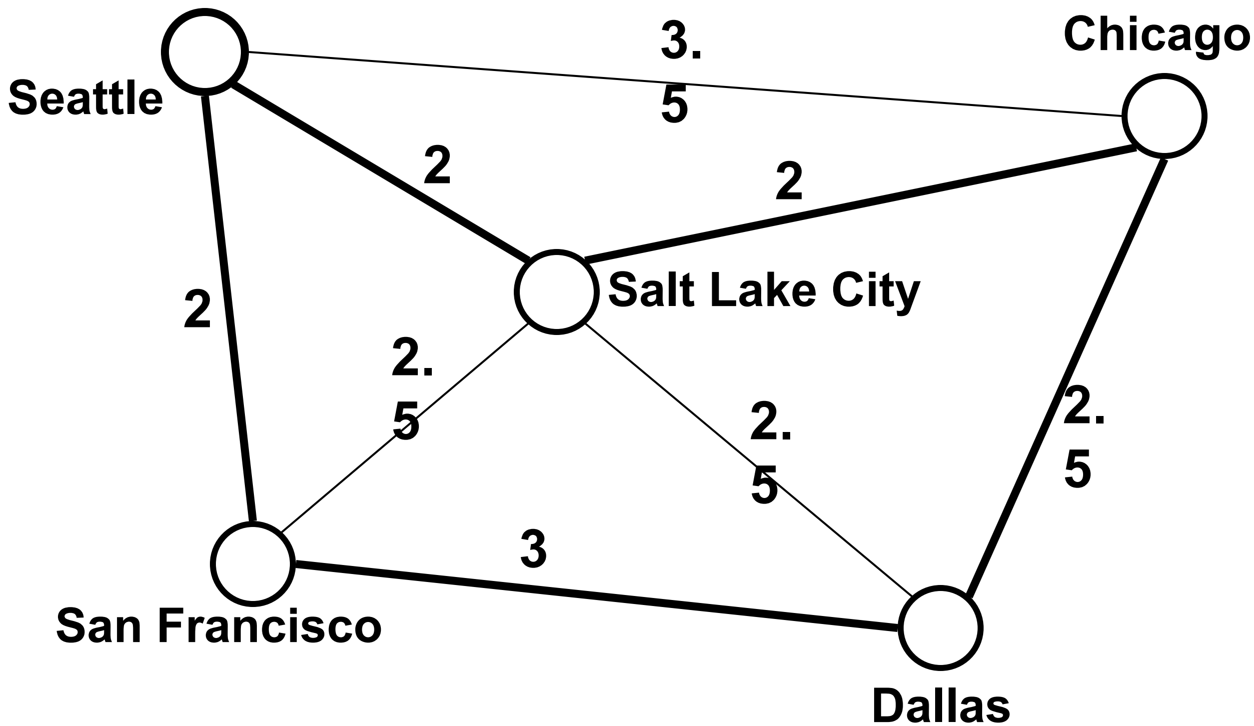


p = {Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle}

Path Length and Cost

Path length: the number of edges in the path

Path cost: the sum of the costs of each edge

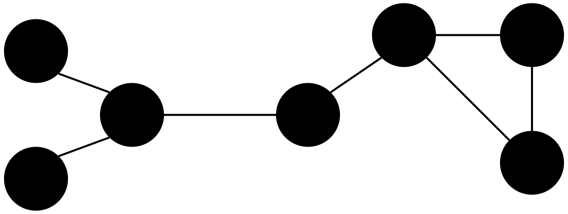


length(p) = 5

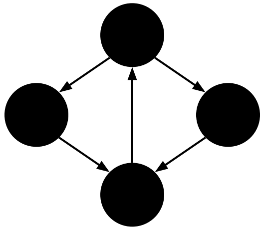
cost(p) = 11.5

Connectivity

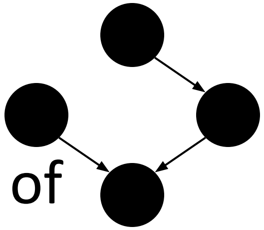
Undirected graphs are *connected* if there is a path between any two vertices



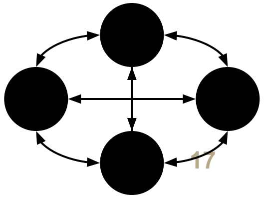
Directed graphs are *strongly connected* if there is a path from any one vertex to any other



Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*

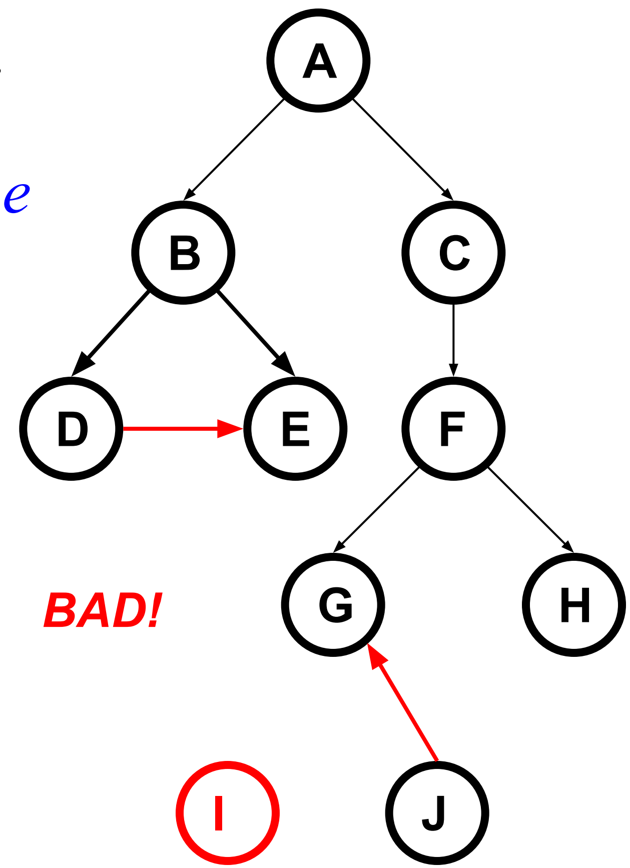


A *complete* graph has an edge between every pair of vertices.It contains n(n-1)/2 no of vertices.



Trees as Graphs

- Every tree is a graph with some restrictions:
  - the tree is *directed*
  - there are *no cycles* (directed or undirected)
  - there is a *directed path from the* root *to every node*
  - *Tree is example of an acyclic graph.*
  - ***Isolated Vertex***:-*It is a vertex not connected to any vertex. degree=0.Ex:- I*
  - ***Pendant Vertex***:-*The vertex having degree 1.Ex:-G,H.*



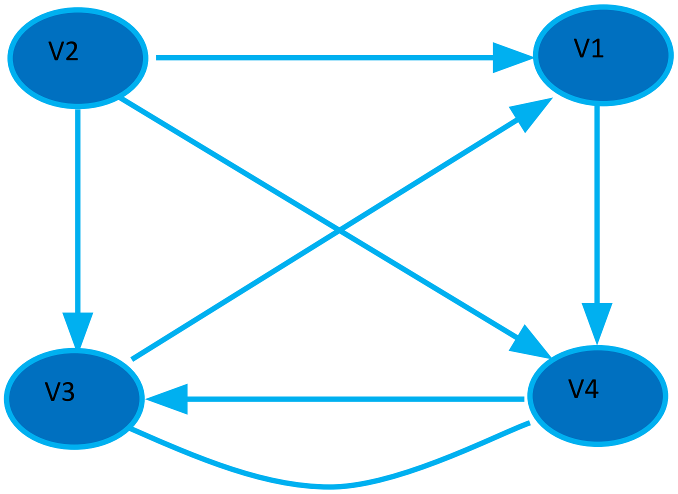
Path Matrix

Let G be a simple directed graph with m nodes, $v_1, v_2, v_3, \ldots\ v_m$. the path matrix or rechability matrix of G is the m-square matrix $P=(p_{ij})$defined as follows.

$$P_{ij} = \begin{cases} 1 & \text{if there is a path from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

P=

	V_1	V_2	V_3	V_4
V_1	1	0	1	1
V_2	1	0	1	1
V_3	1	0	1	1
V_4	1	0	1	1



Calculating Path Matrix

- Let A be the adjacency matrix and let $P=(p_{ij})$ be the path matrix of a digraph G. Then $p_{ij}=1$ if and only if there is a non-zero number in the ij entry of the matrix B_m where m is the no of nodes in the graph.

$B_m=A+A^2+A^3+....+A^m$

Consider the graph G with m=4 nodes in previous slide.

So $B_4=A+A^2+A^3+A^4.$

A=

	V_1	V_2	V_3	V_4
V_1	0	0	0	1
V_2	1	0	1	1
V_3	1	0	0	1
V_4	0	0	1	0

A^2 =

	V_1	V_2	V_3	V_4
V_1	0	0	1	0
V_2	1	0	1	2
V_3	0	0	1	1
V_4	1	0	0	1

A^3=

	V_1	V_2	V_3	V_4
V_1	1	0	0	1
V_2	1	0	2	2
V_3	1	0	1	1
V_4	0	0	1	1

A^4=

	V_1	V_2	V_3	V_4
V_1	0	0	1	1
V_2	2	0	2	3
V_3	1	0	1	2
V_4	1	0	1	1

B_4=

	V_1	V_2	V_3	V_4
V_1	1	0	2	3
V_2	5	0	6	8
V_3	3	0	3	5
V_4	2	0	3	3

P=

	V_1	V_2	V_3	V_4
V_1	1	0	1	1
V_2	1	0	1	1
V_3	1	0	1	1
V_4	1	0	1	1

Graph Traversal

Traversing a graph means visiting all the nodes in the graph exactly once.

There are mainly two methods of traversal

1. Breadth first Search (BFS)
2. Depth First Search (DFS)

The graph traversal starts with any arbitrary node, because there is no special node like root in tree.

 We will assume that while traversing ,each node will pass through three different states

- ready state
- waiting state
- visited state.

Breadth First Search (BFS)

<u>Algorithm</u>

1. All nodes are initialized as ready state and initialize queue to empty.
2. Begin with any node, which is ready state and put into queue. Mark the status of that node waiting.
3. While (Queue is not empty)

   do

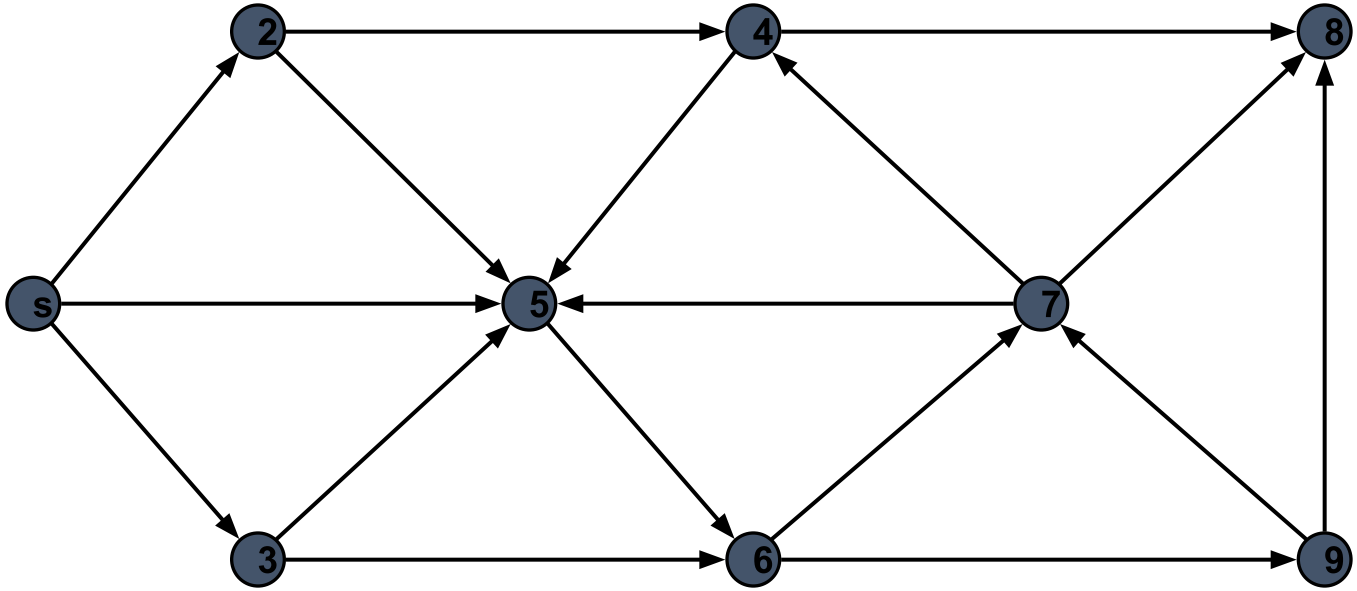
   Begin

4. Delete the first node K from the Queue and process it. Make the status of that node to visited.
5. Add all the adjacent nodes of K, which are in ready state to the rear side of the Queue and mark the status of those nodes to waiting.

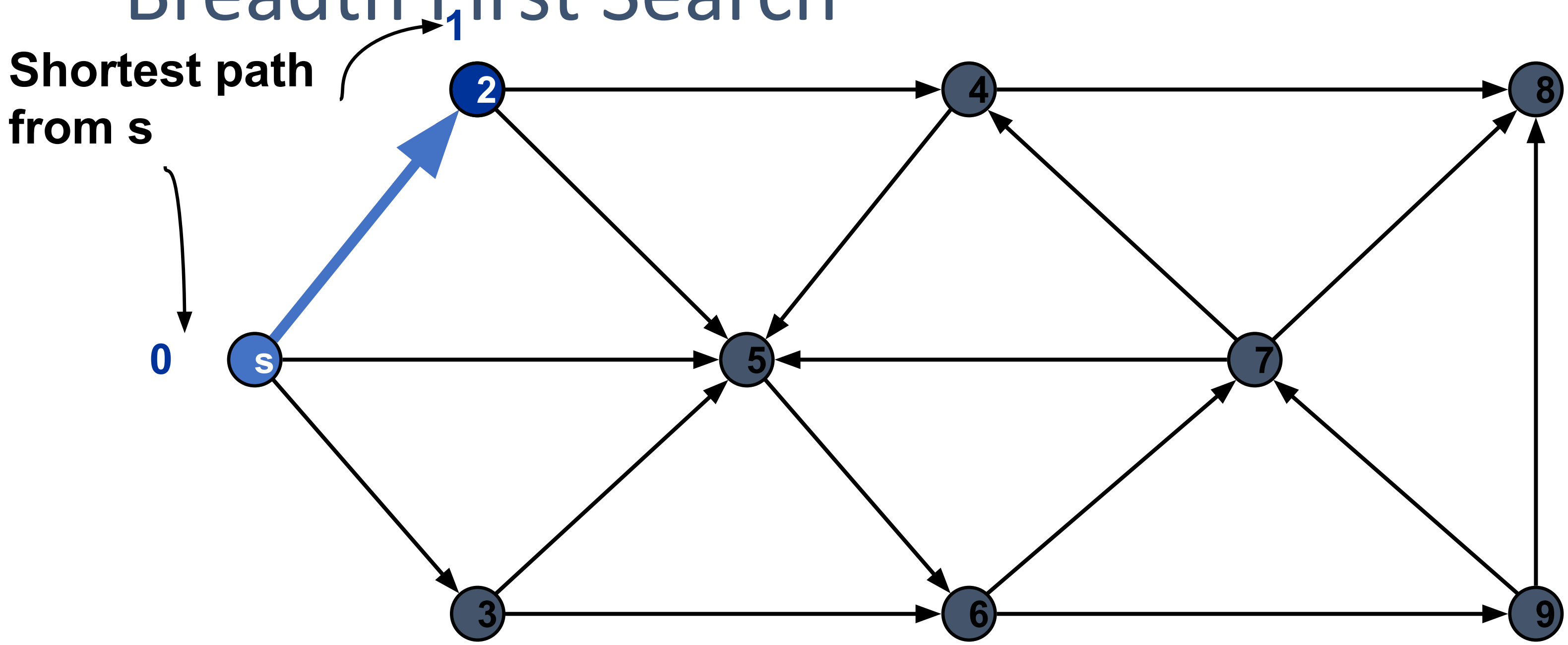
   End

6. If the graph still containing nodes, which are in ready state then go to step 2.
7. Return.

Breadth First Search



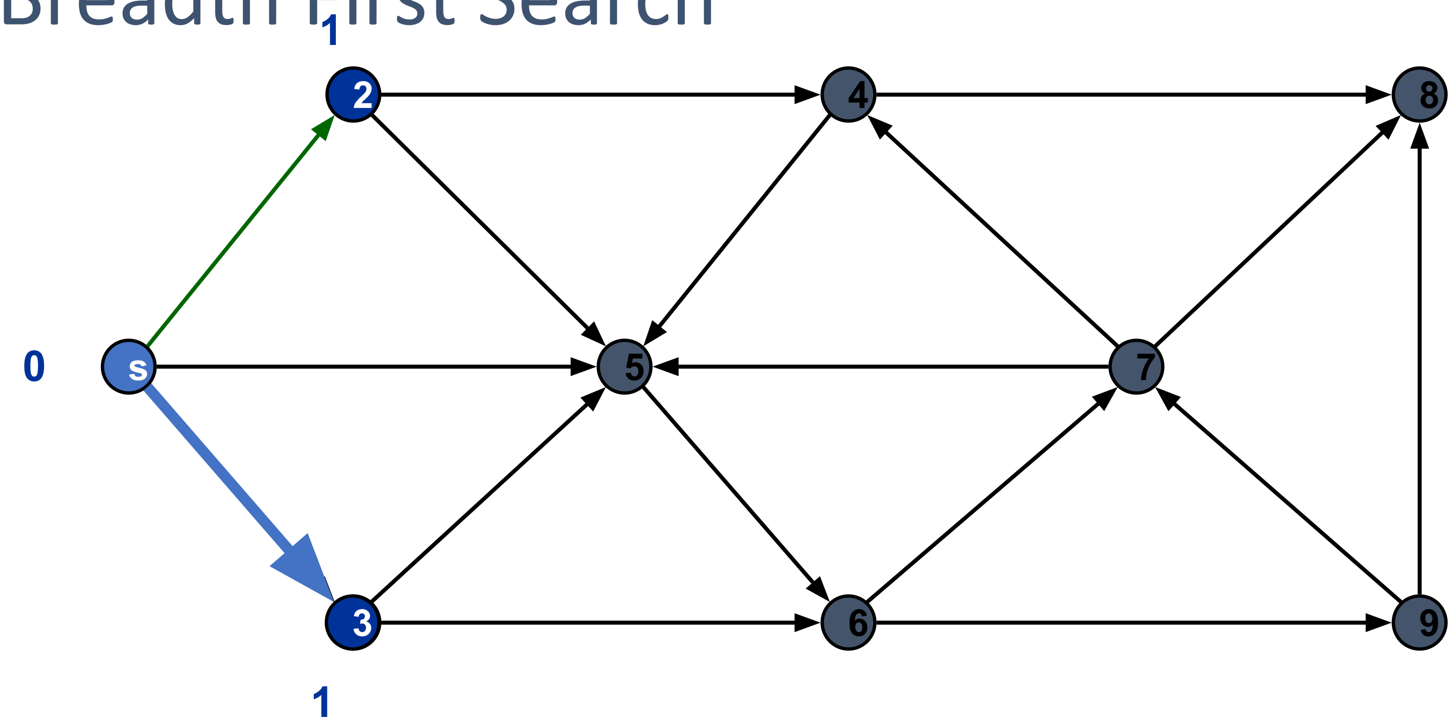
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: s

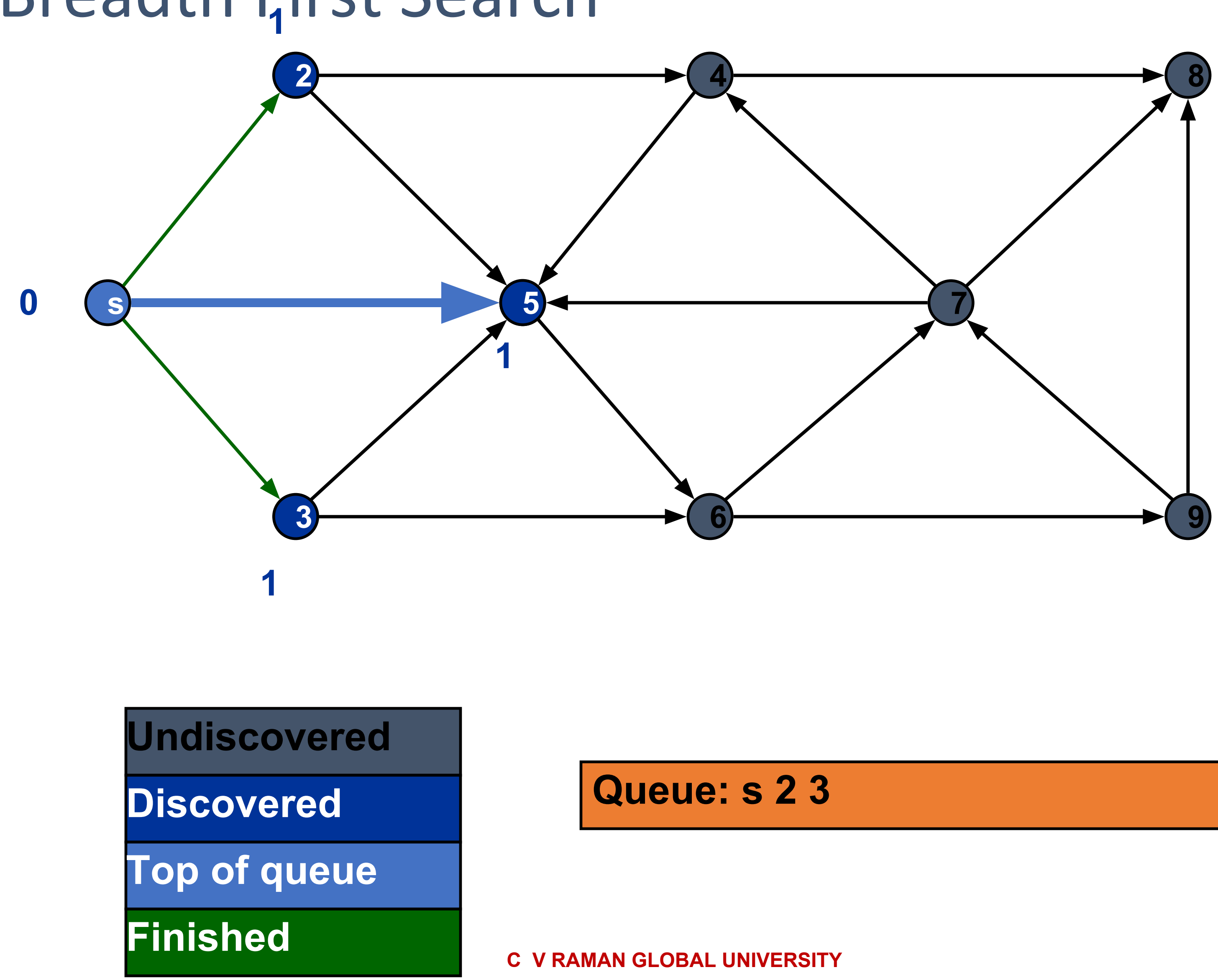
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

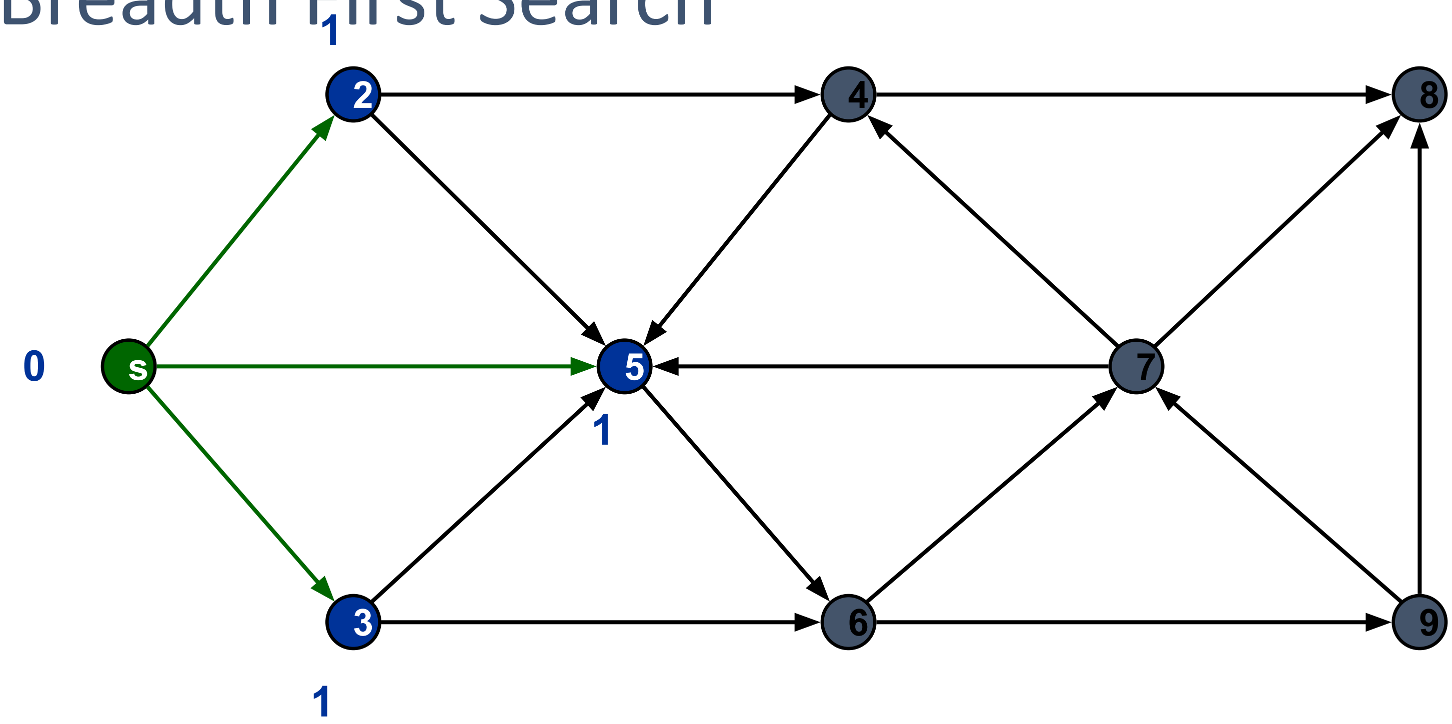
Queue: s 2

Breadth First Search



Queue: s 2 3

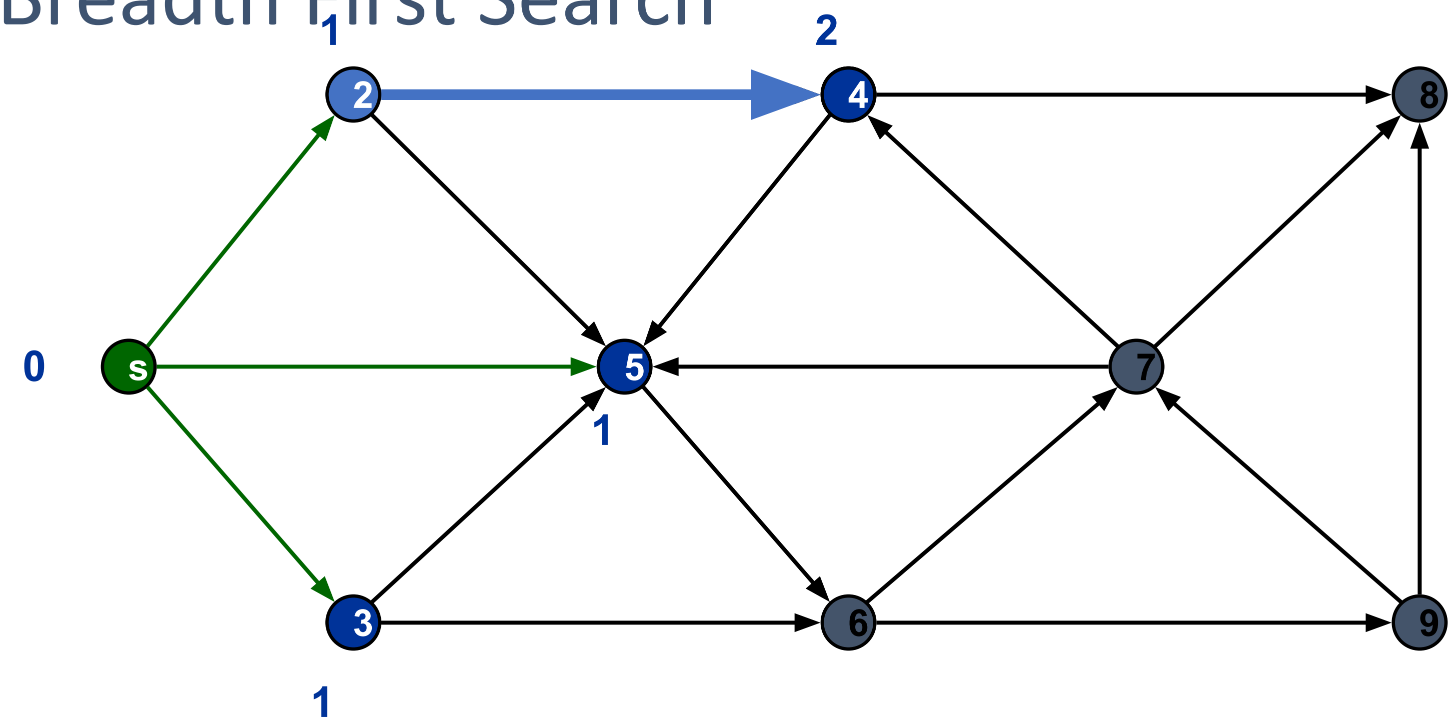
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5

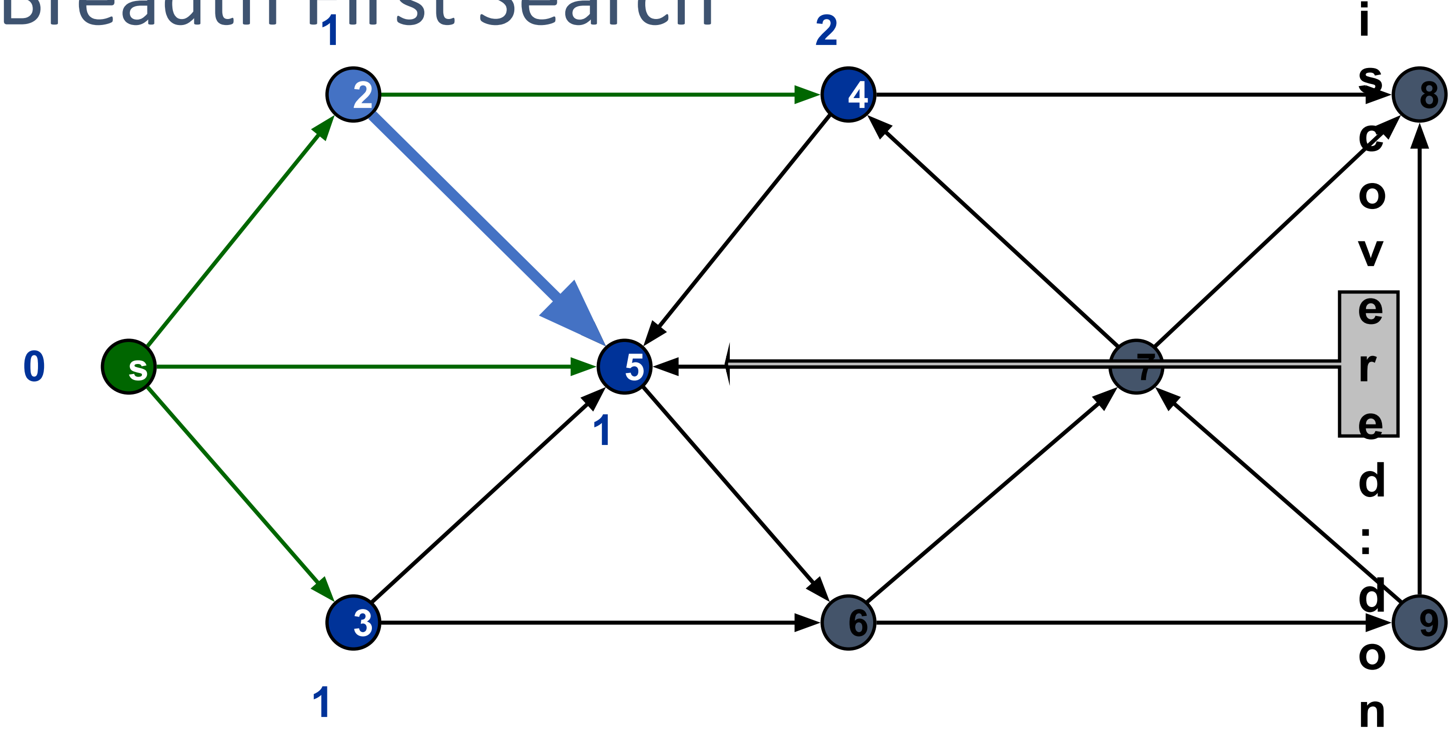
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5

Breadth First Search



Undiscovered

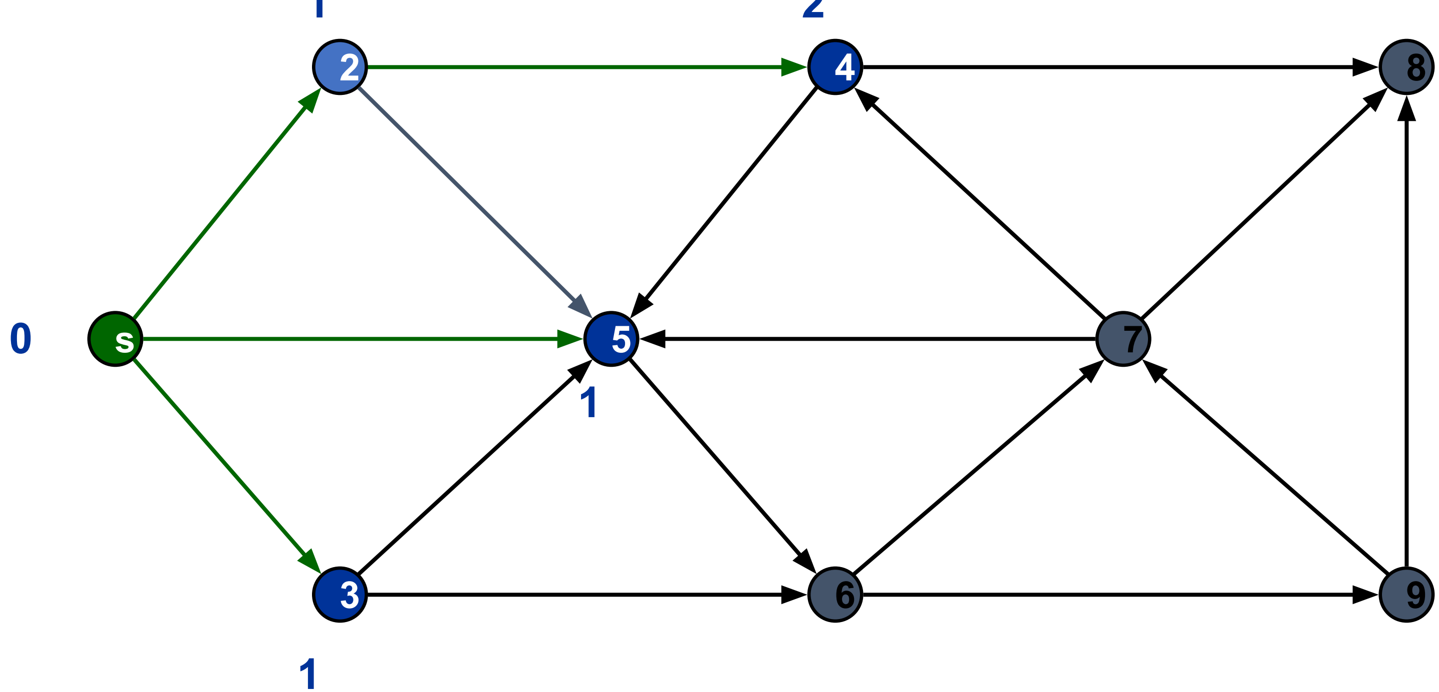
Discovered

Top of queue

Finished

Queue: 2 3 5 4

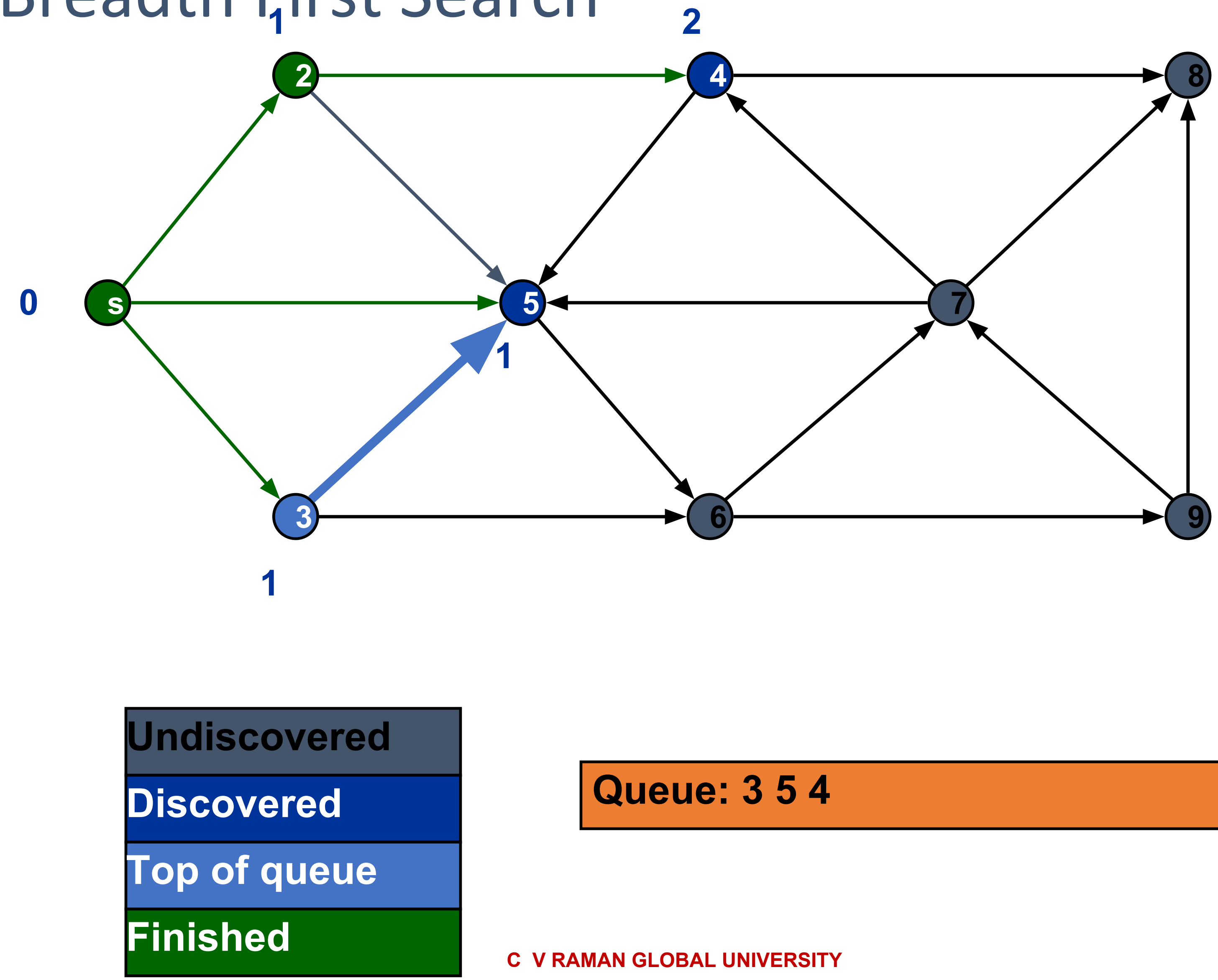
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

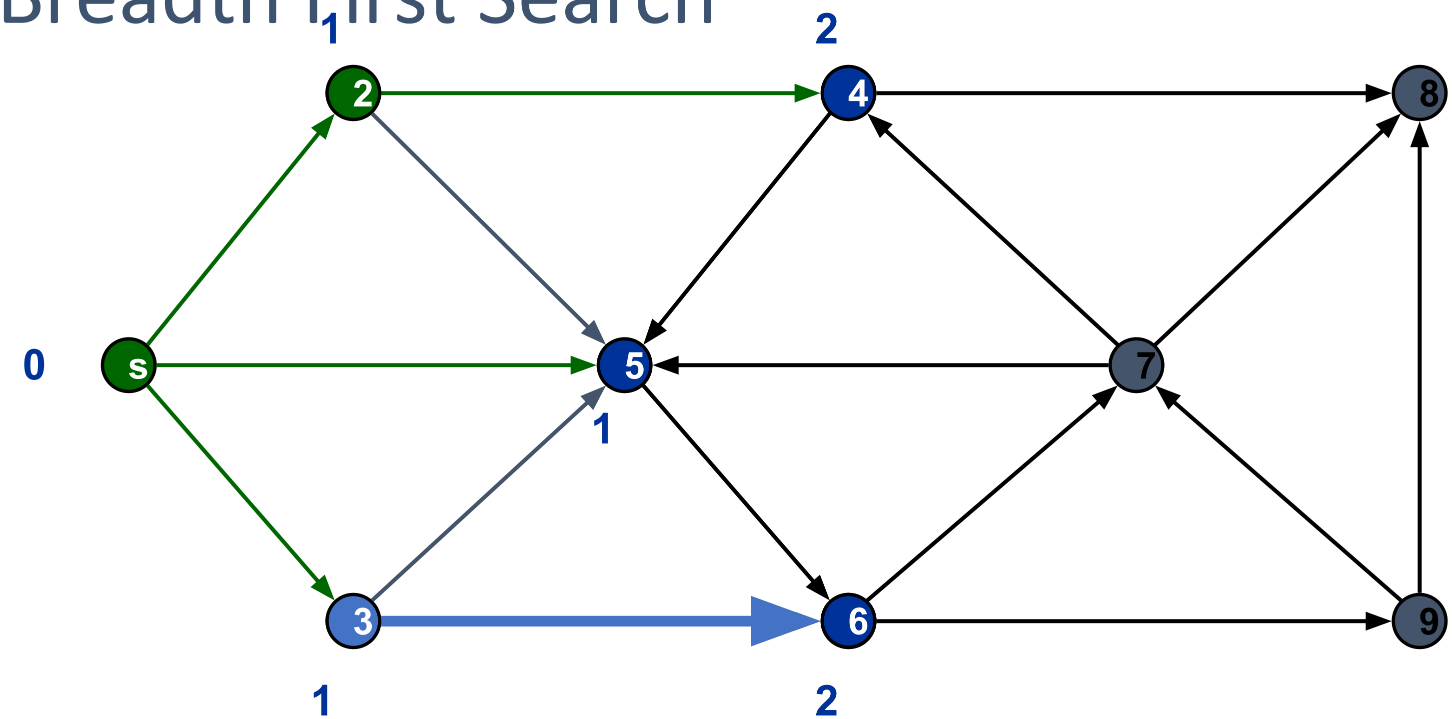
Queue: 2 3 5 4

Breadth First Search



Queue: 3 5 4

Breadth First Search



Undiscovered

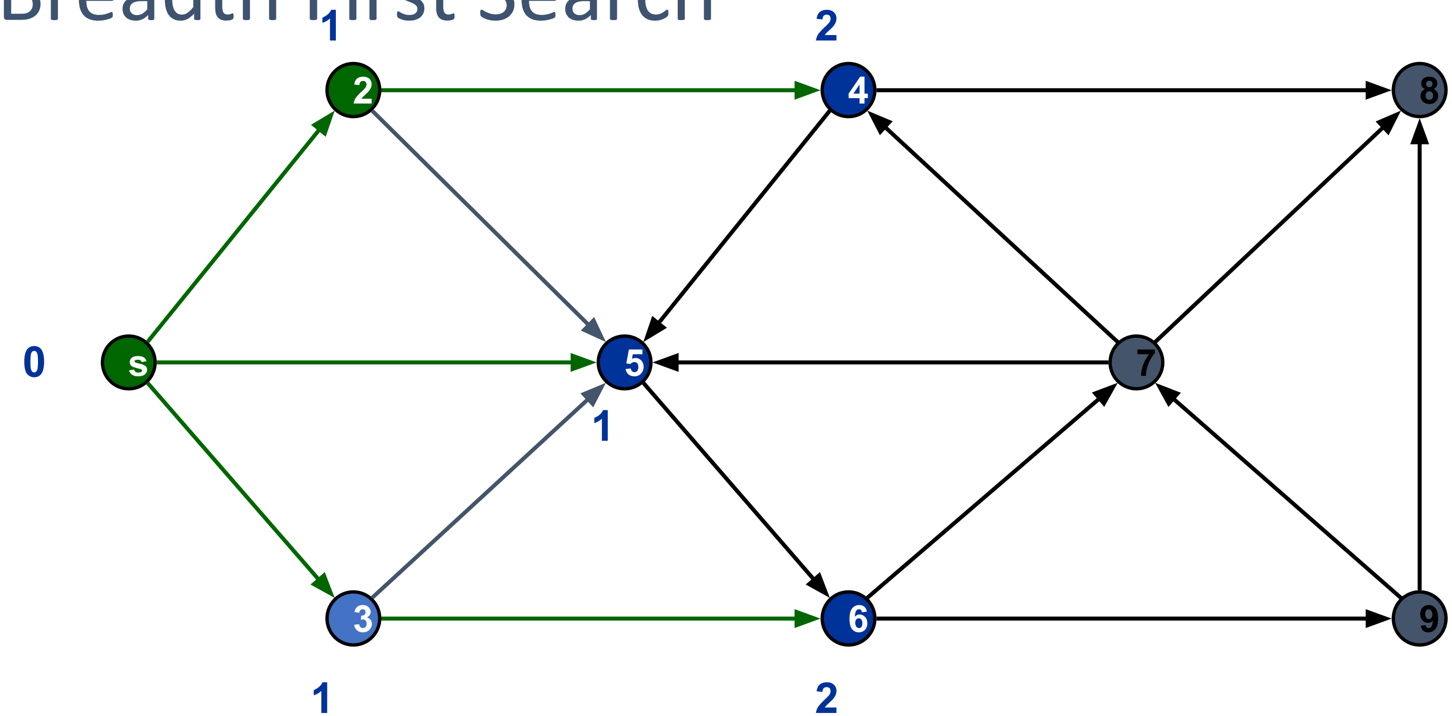
Discovered

Top of queue

Finished

Queue: 3 5 4

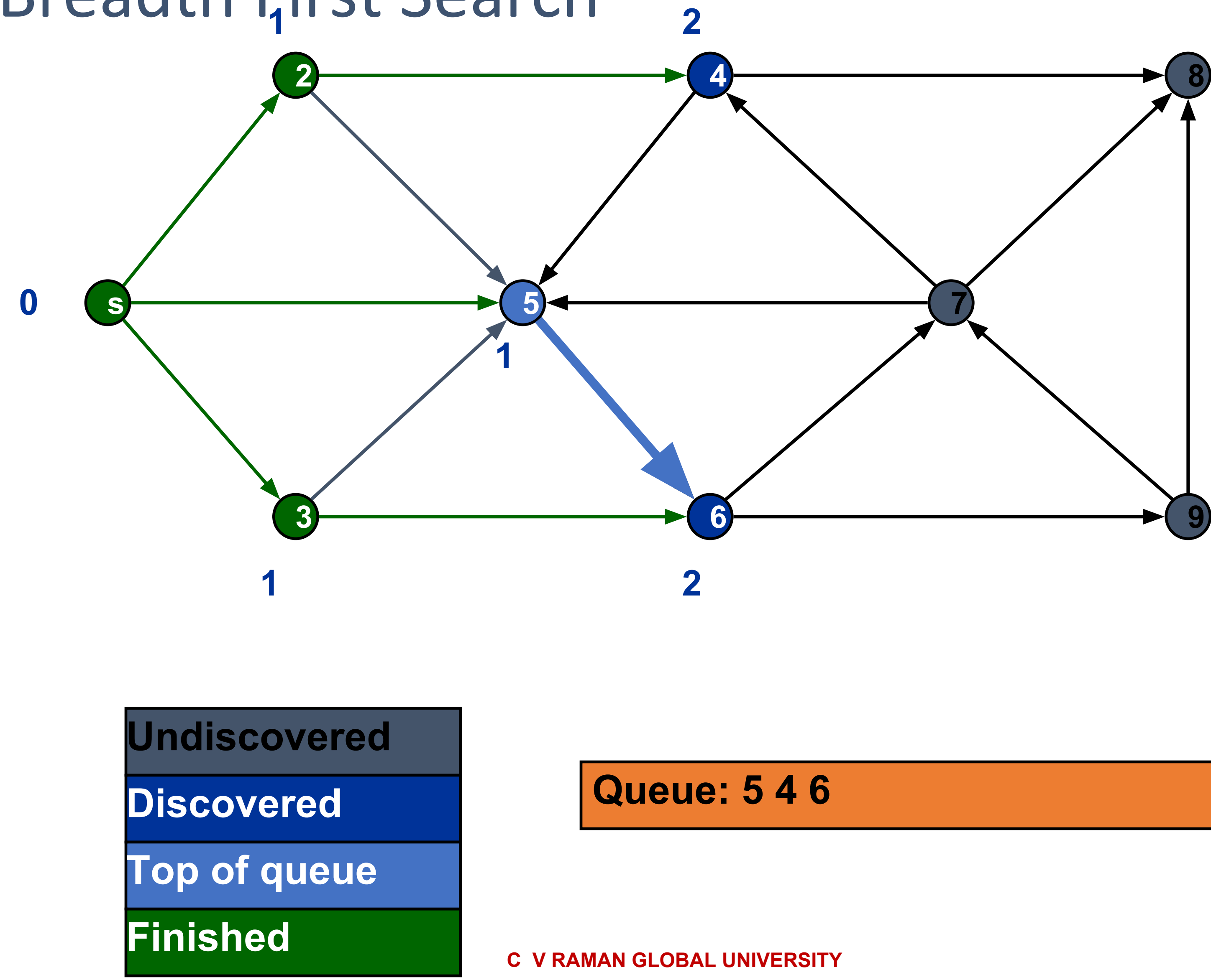
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

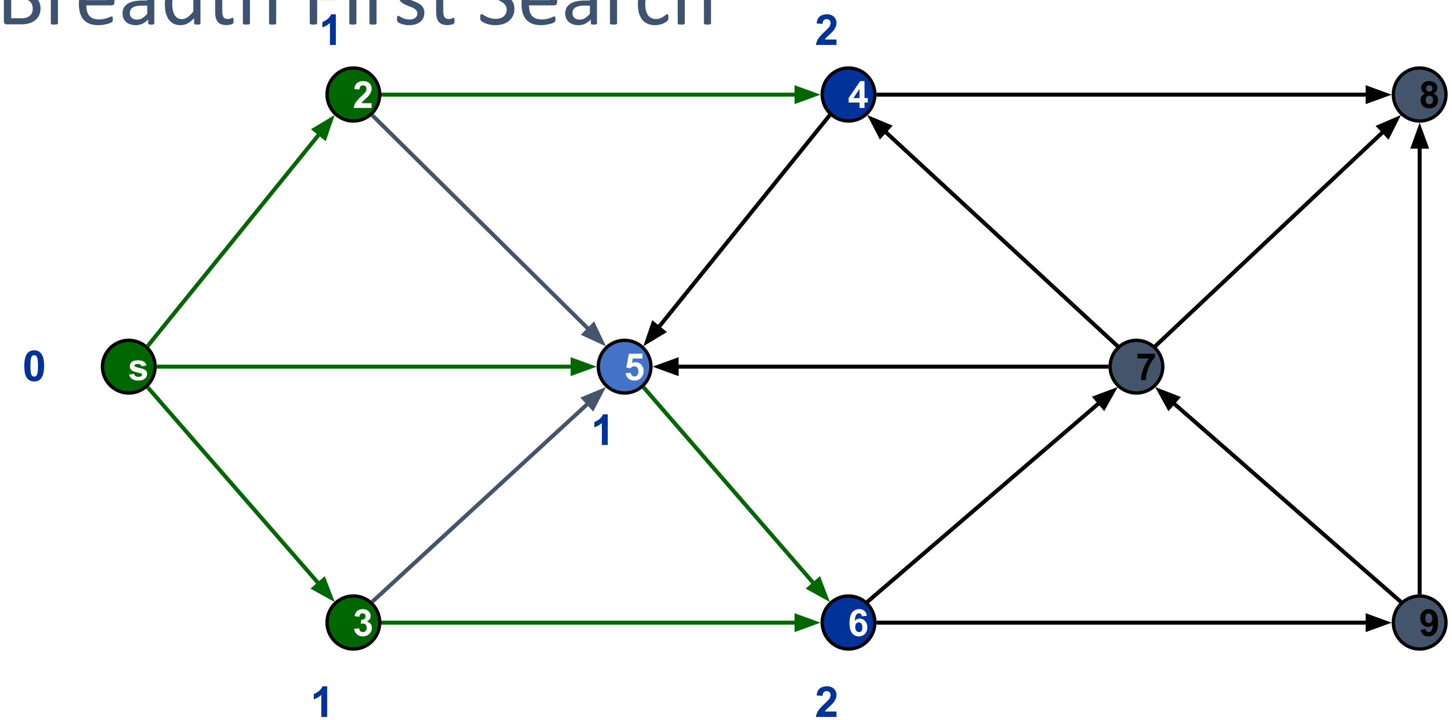
Queue: 3 5 4 6

Breadth First Search



Queue: 5 4 6

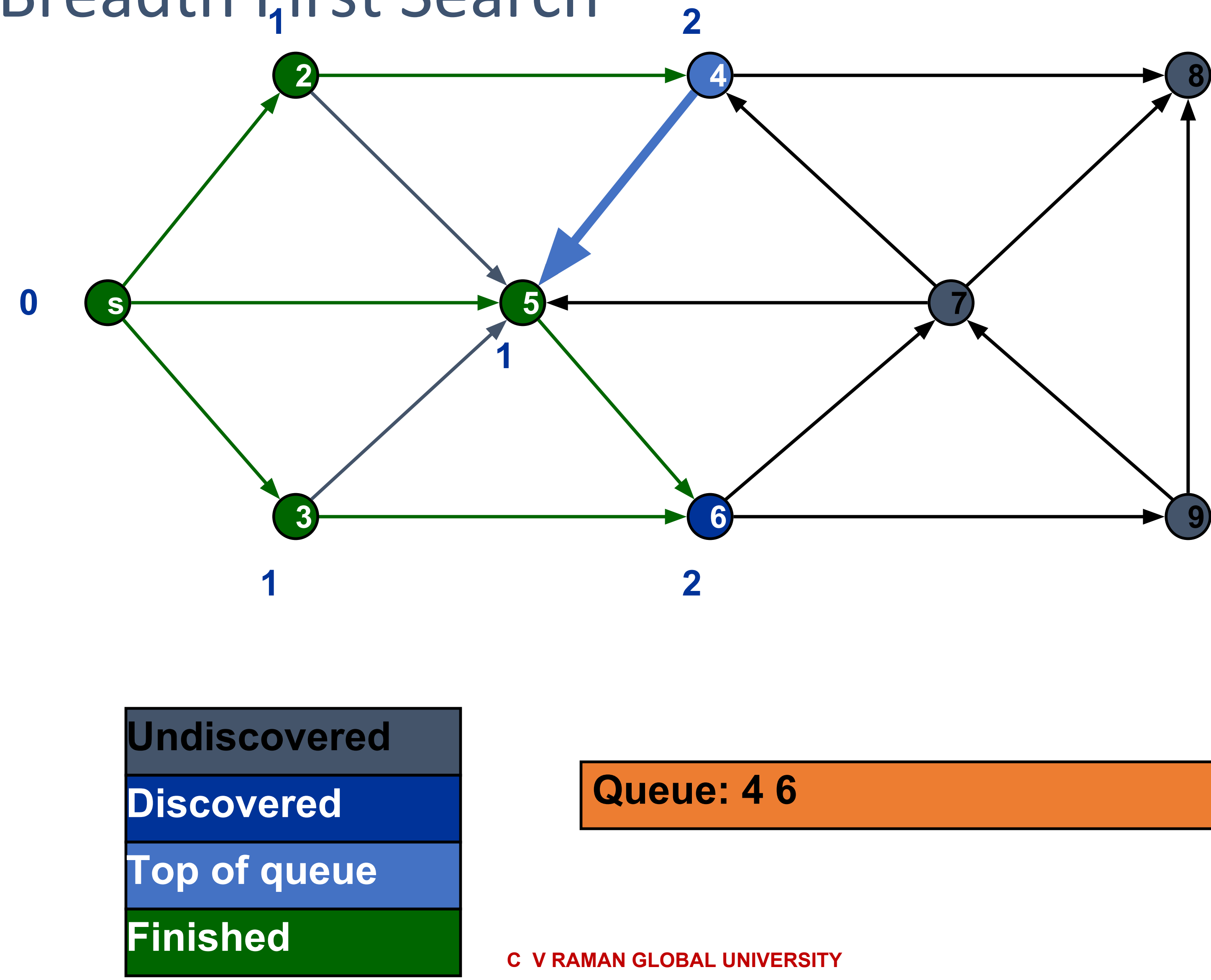
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

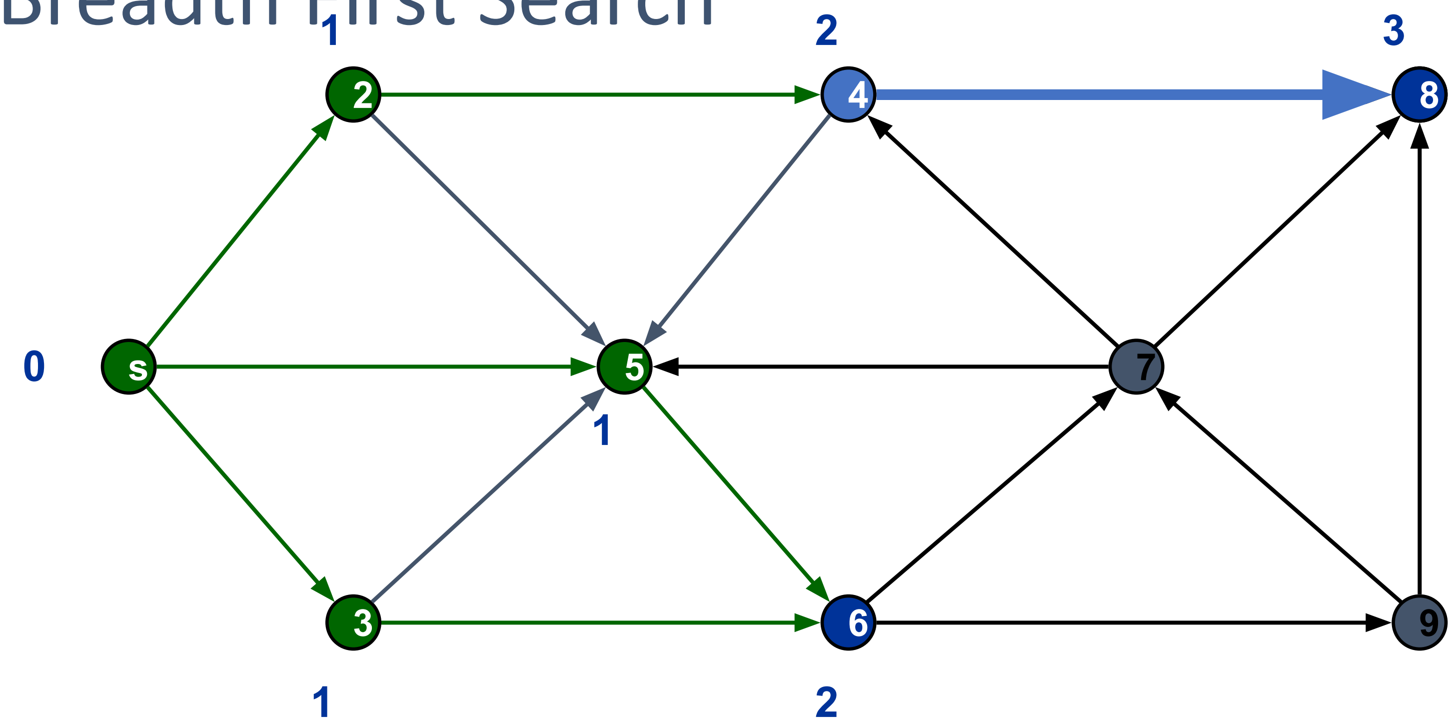
Queue: 5 4 6

Breadth First Search



Queue: 4 6

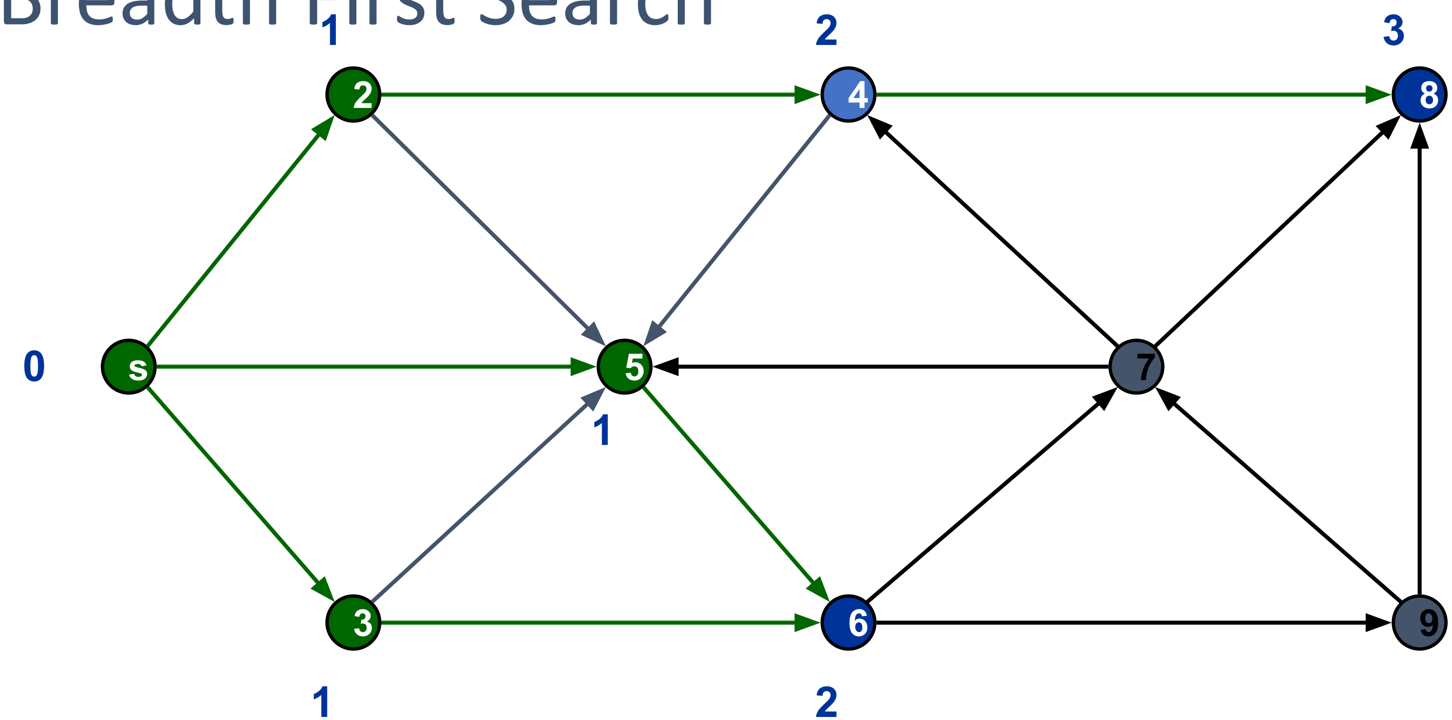
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6

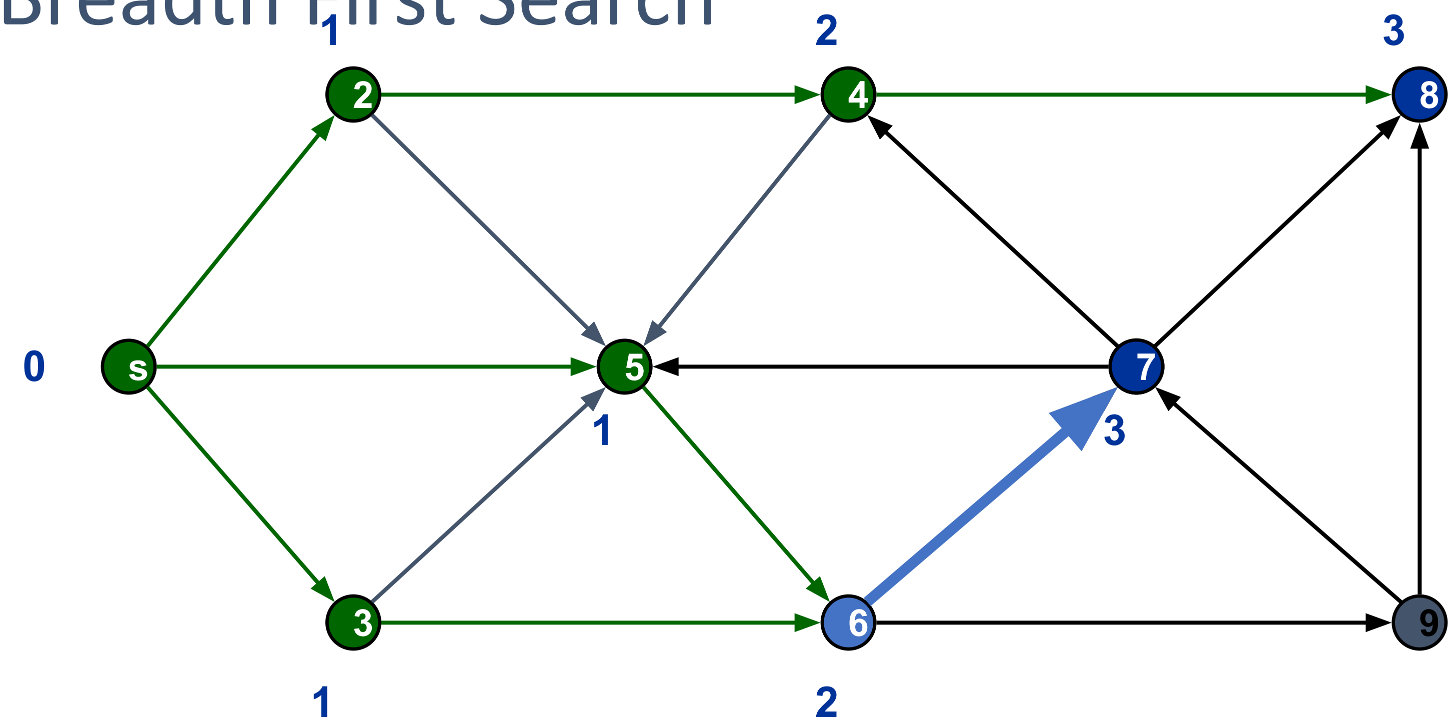
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6 8

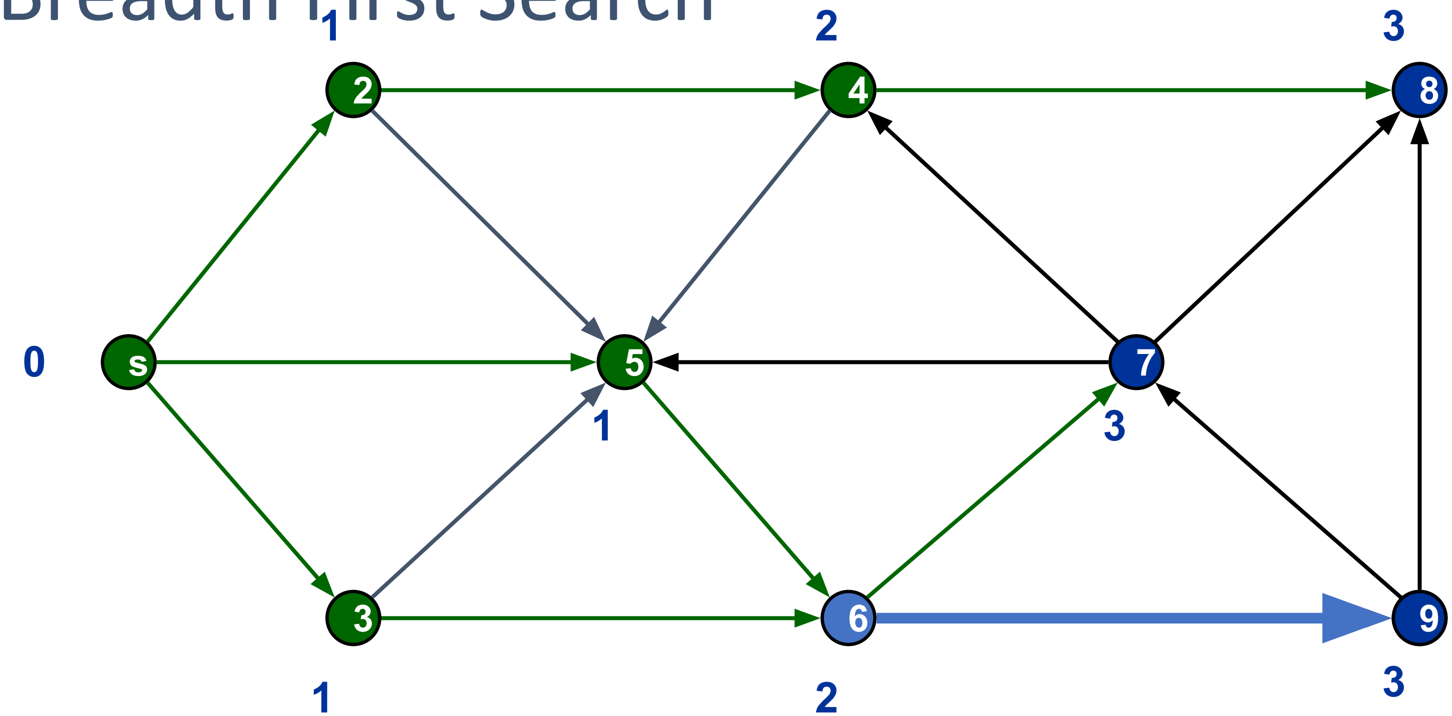
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8

Breadth First Search



Undiscovered

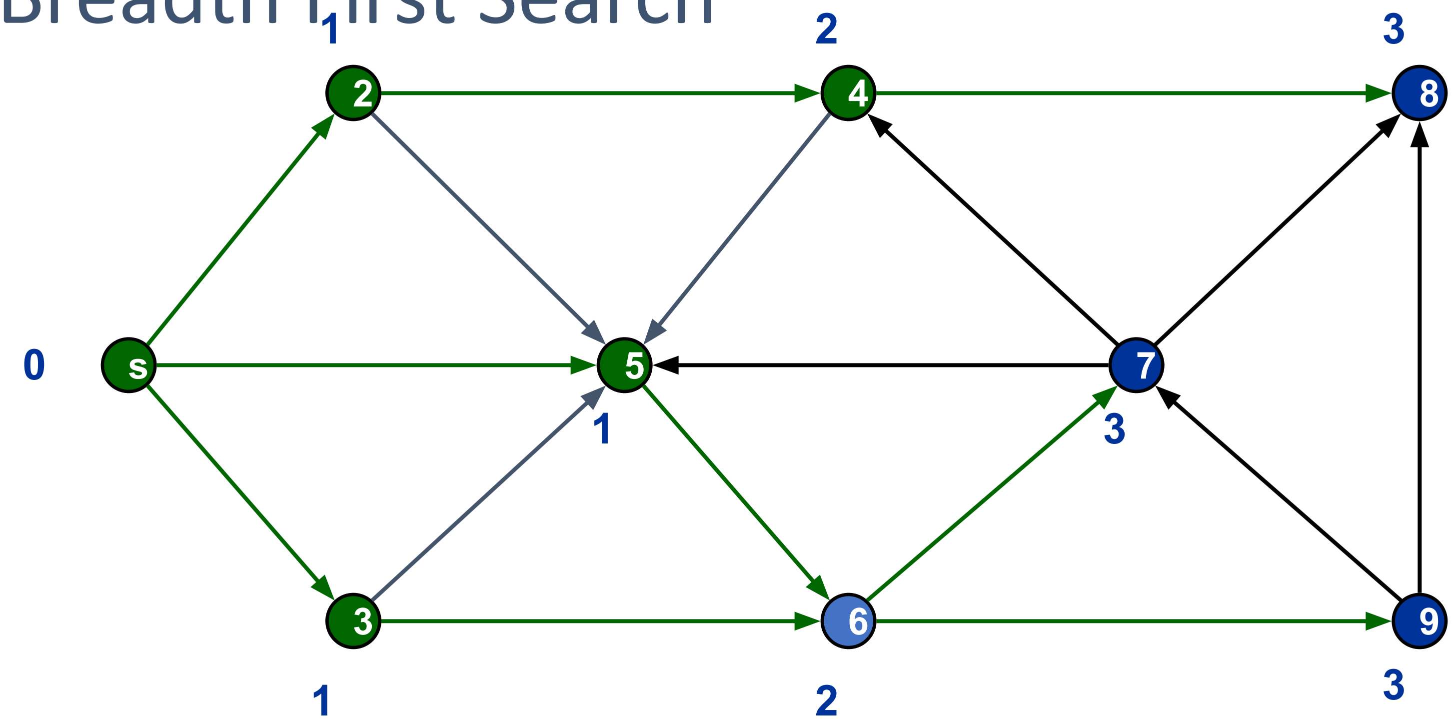
Discovered

Top of queue

Finished

Queue: 6 8 7

Breadth First Search



Undiscovered

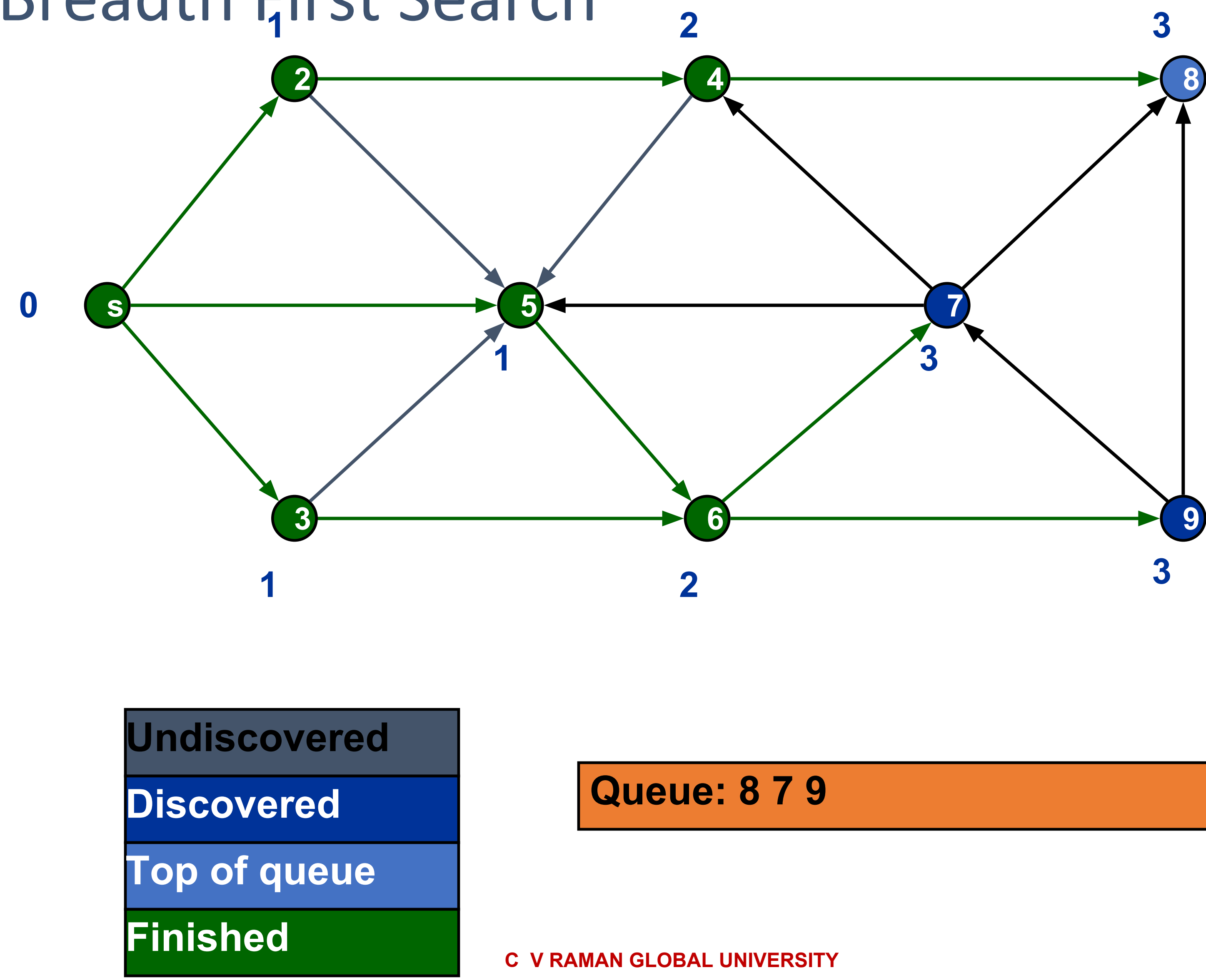
Discovered

Top of queue

Finished

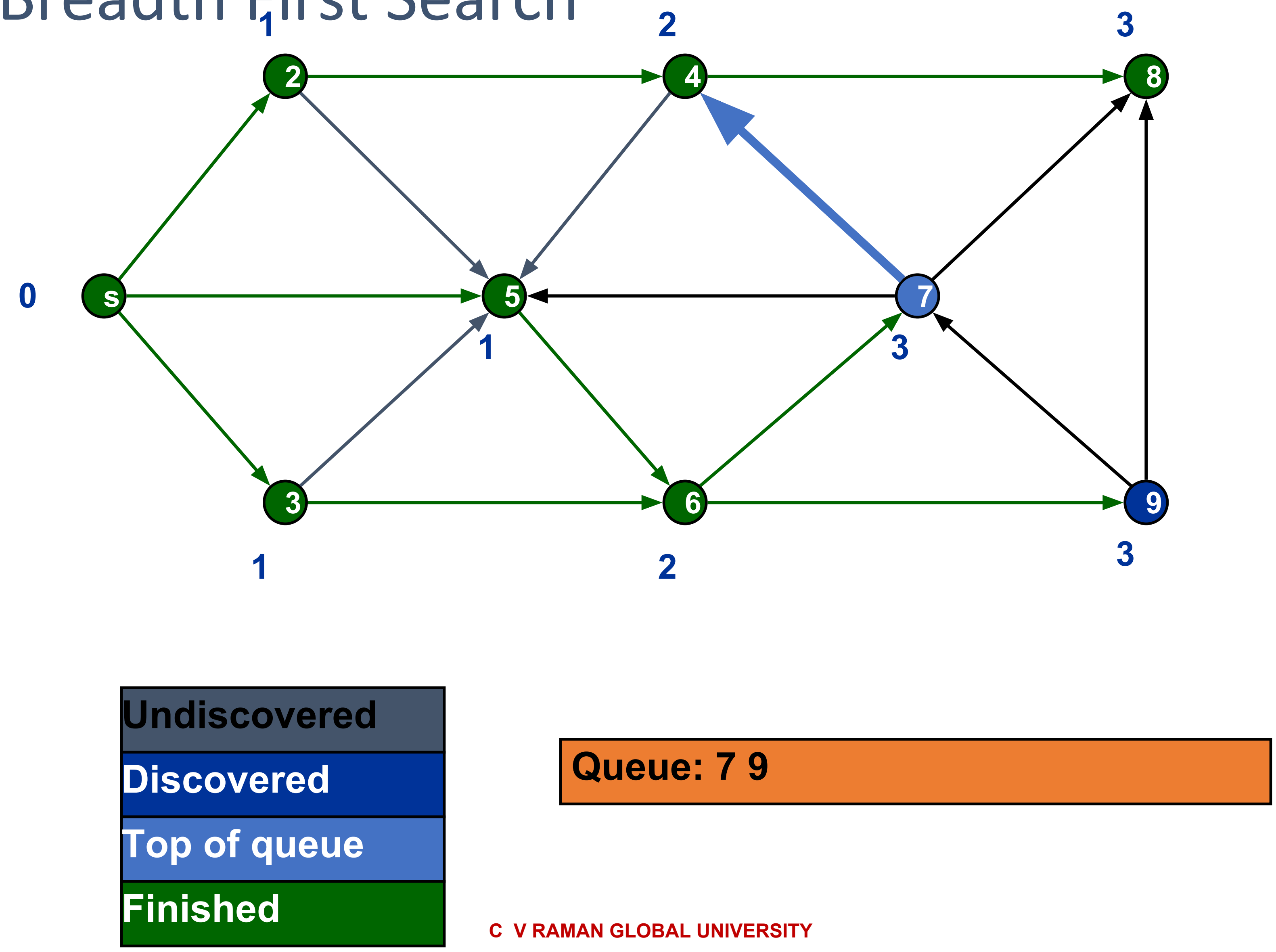
Queue: 6 8 7 9

Breadth First Search

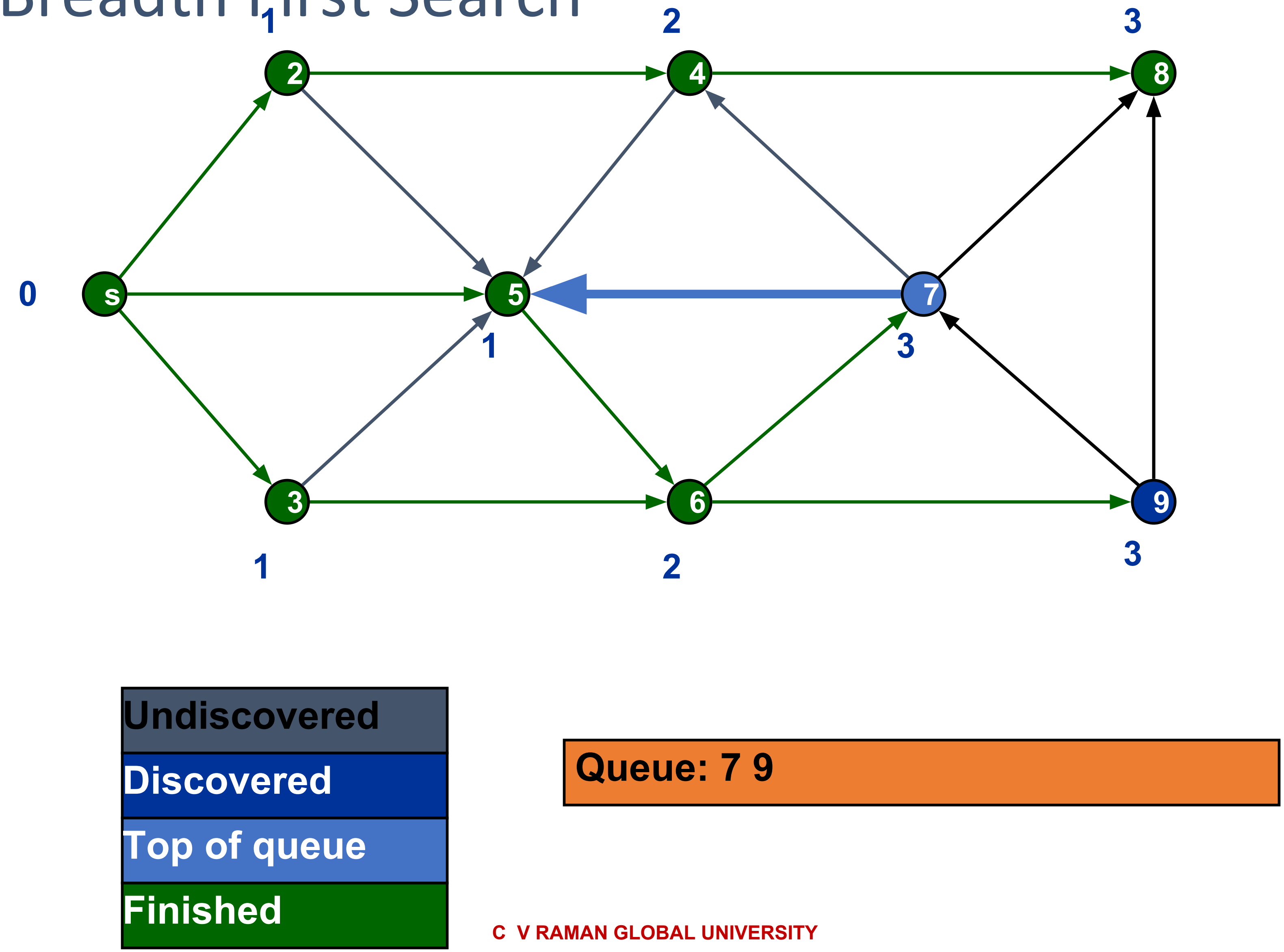


Queue: 8 7 9

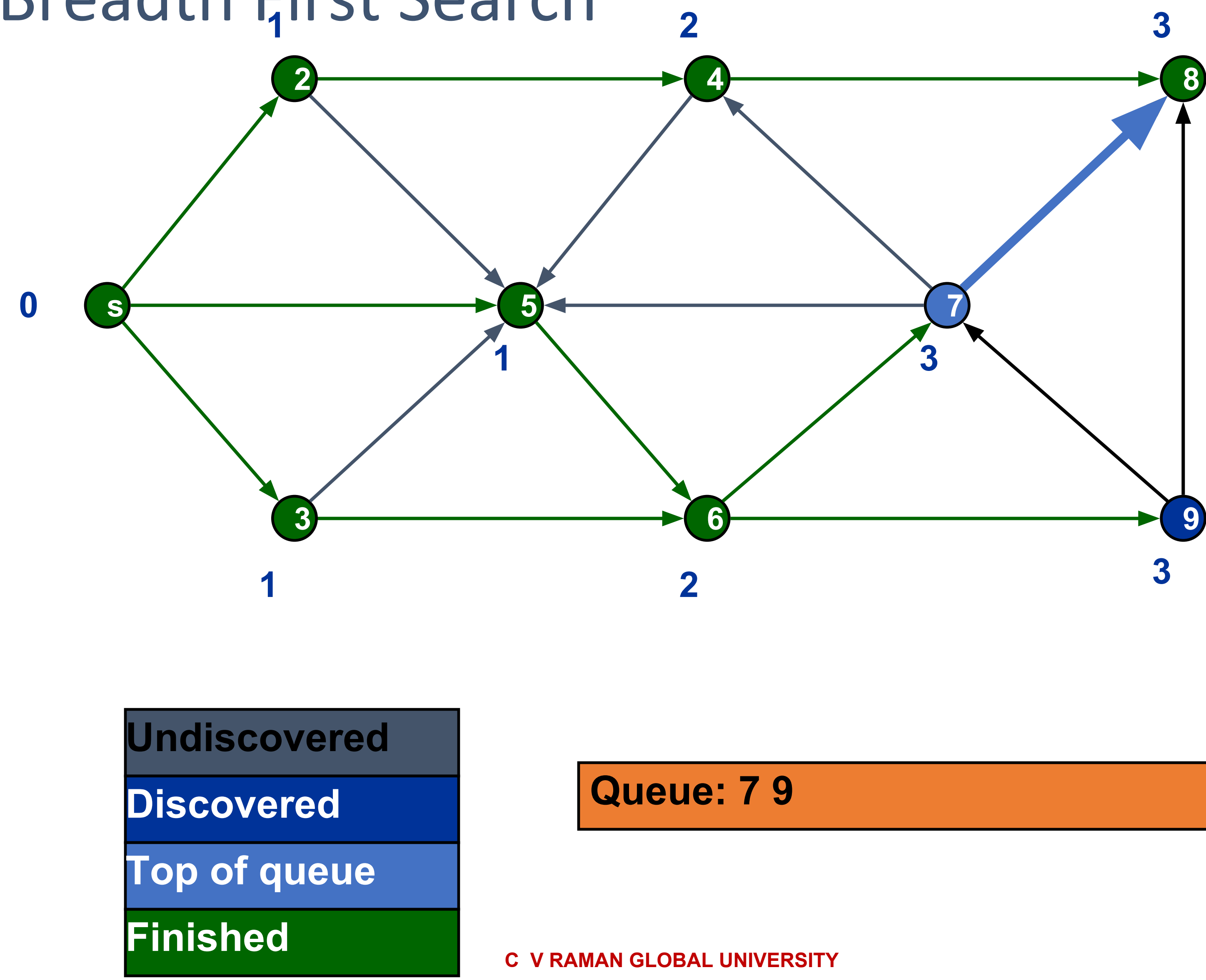
Breadth First Search



Breadth First Search

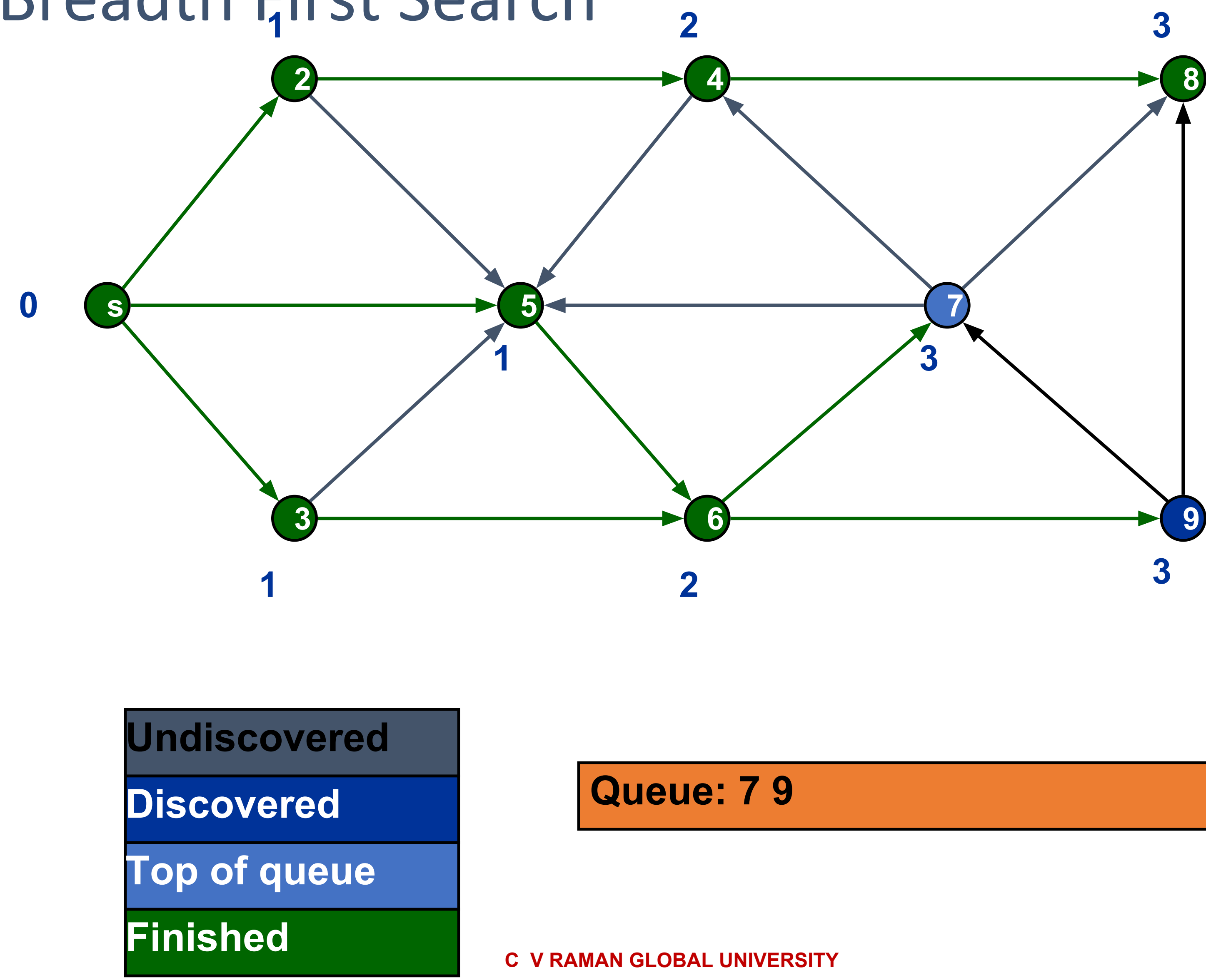


Breadth First Search



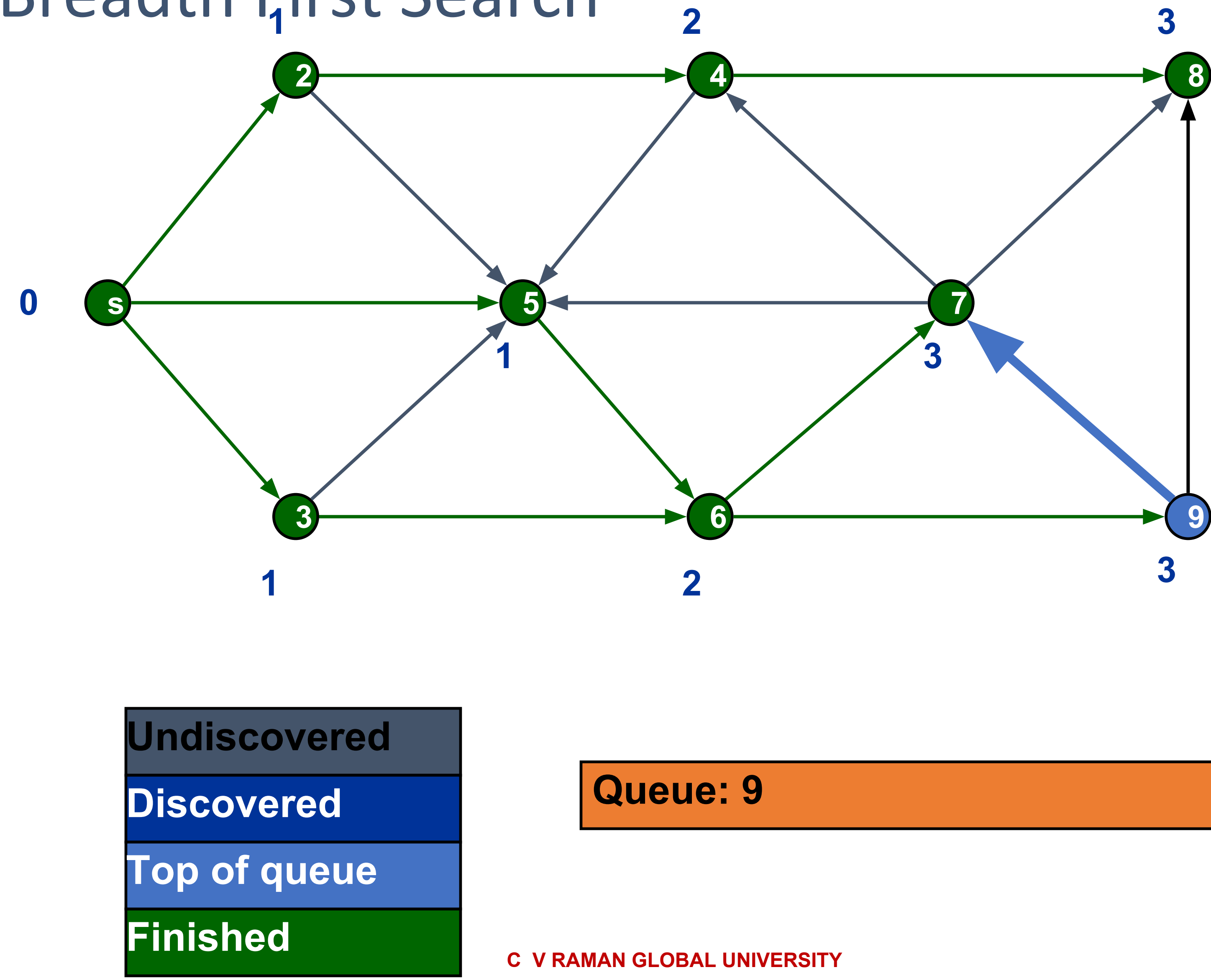
Queue: 7 9

Breadth First Search



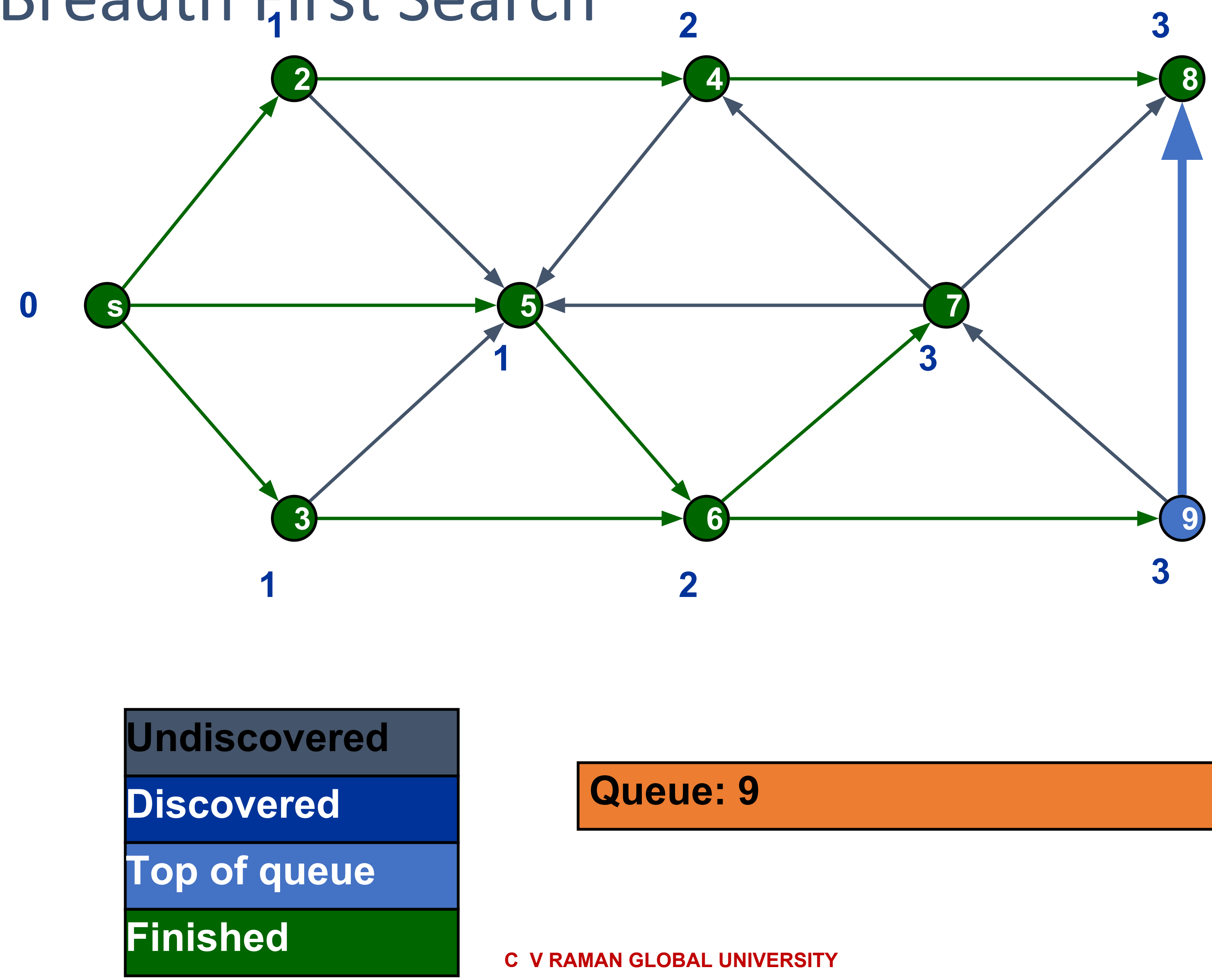
Queue: 7 9

Breadth First Search



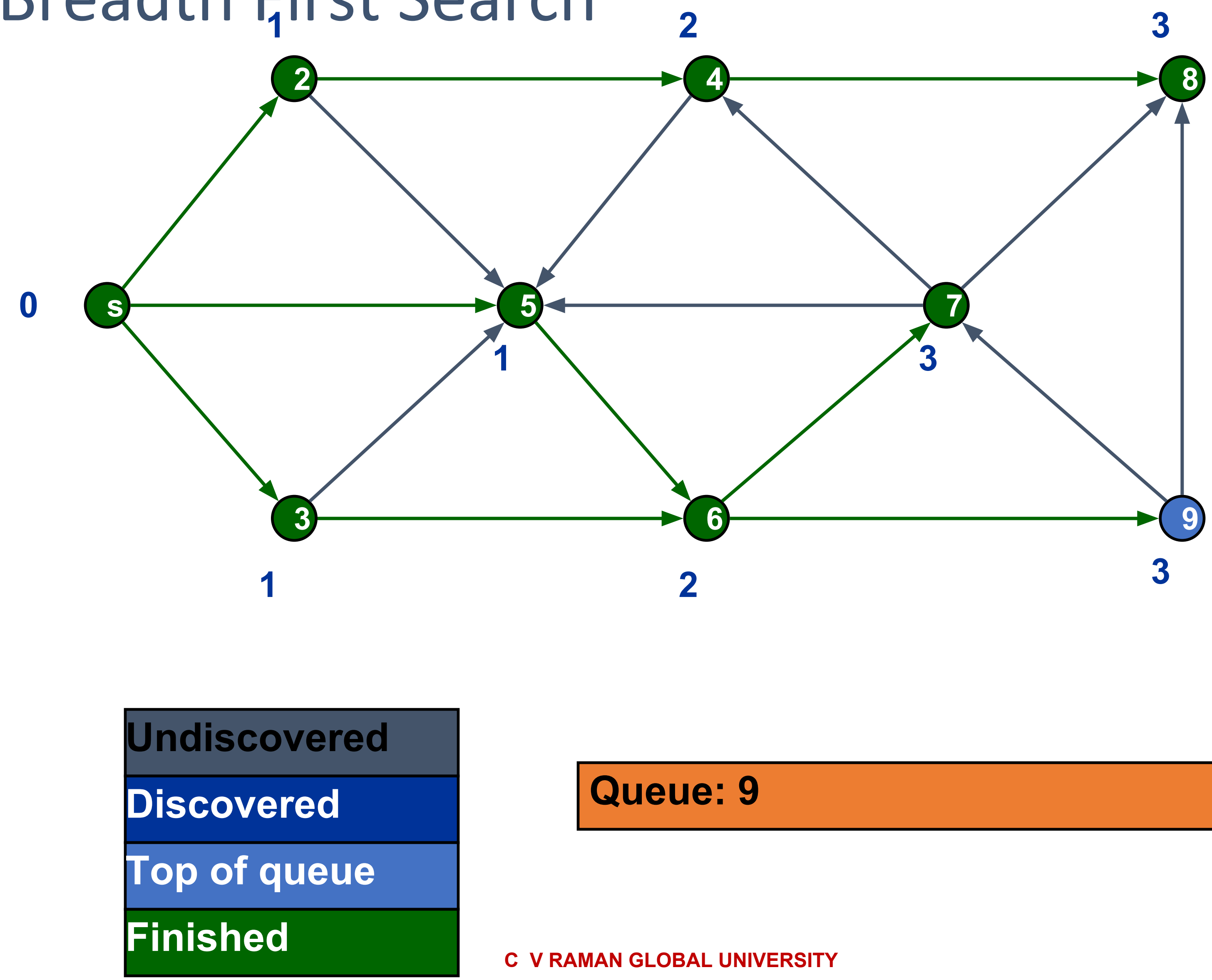
Queue: 9

Breadth First Search



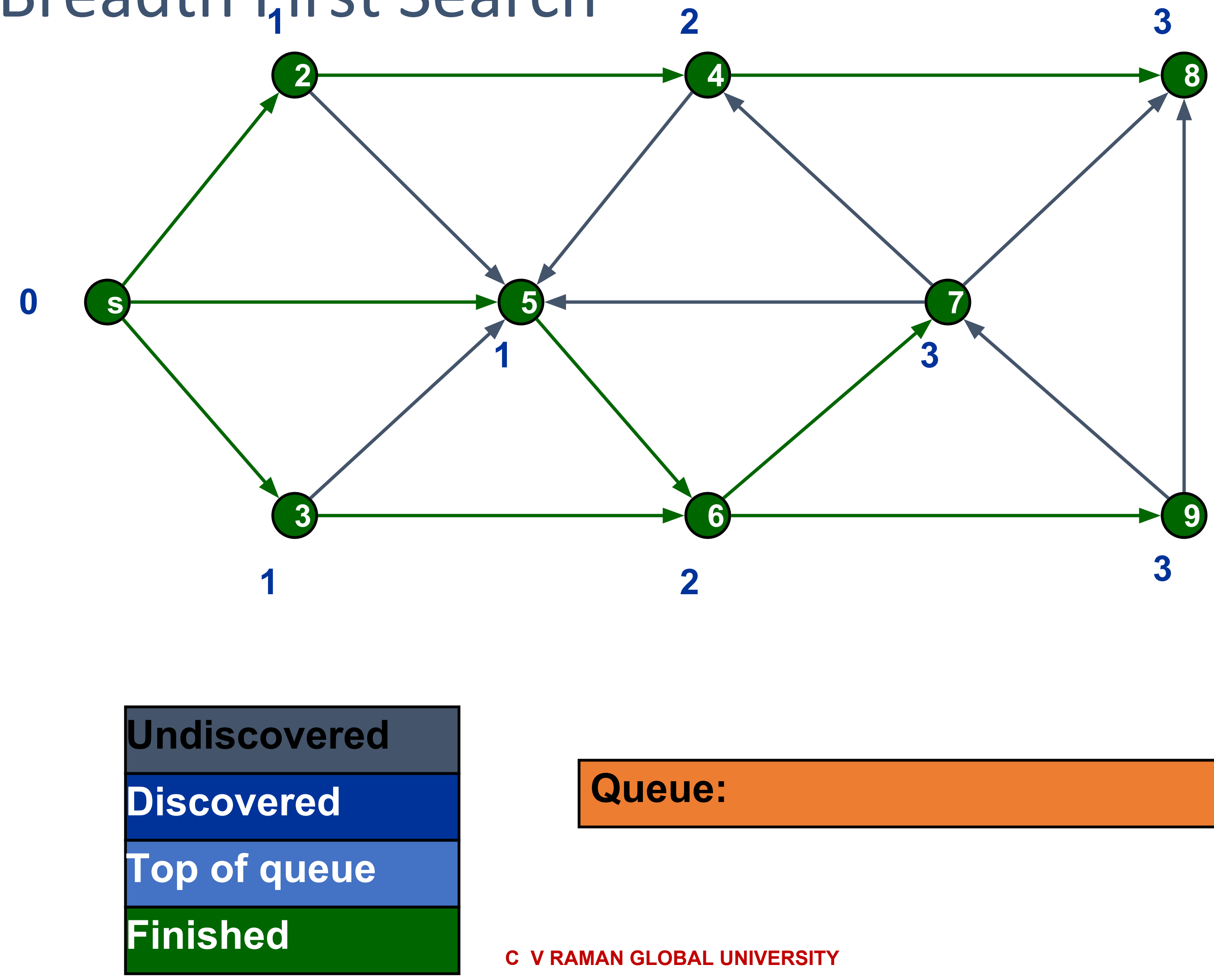
Queue: 9

Breadth First Search



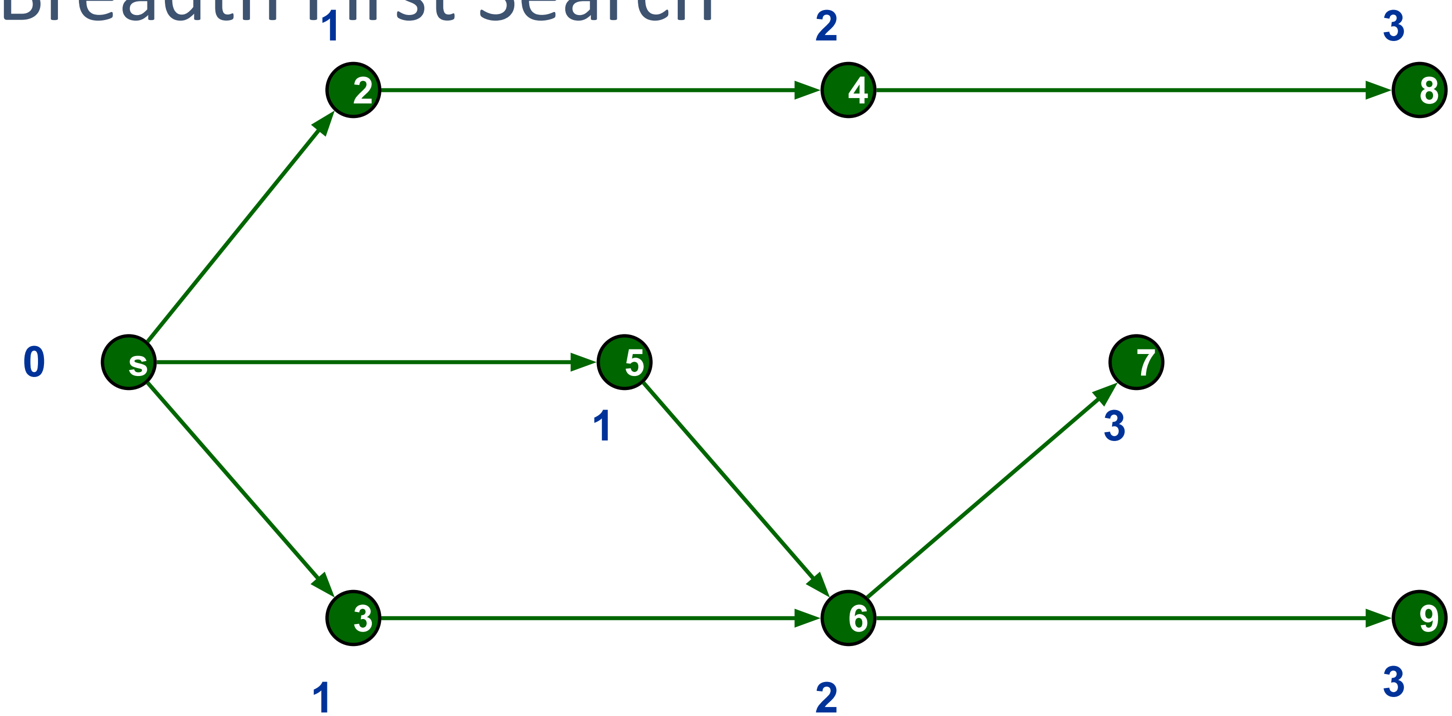
Queue: 9

Breadth First Search



Queue:

Breadth First Search



Level Graph

Directed Depth First Search

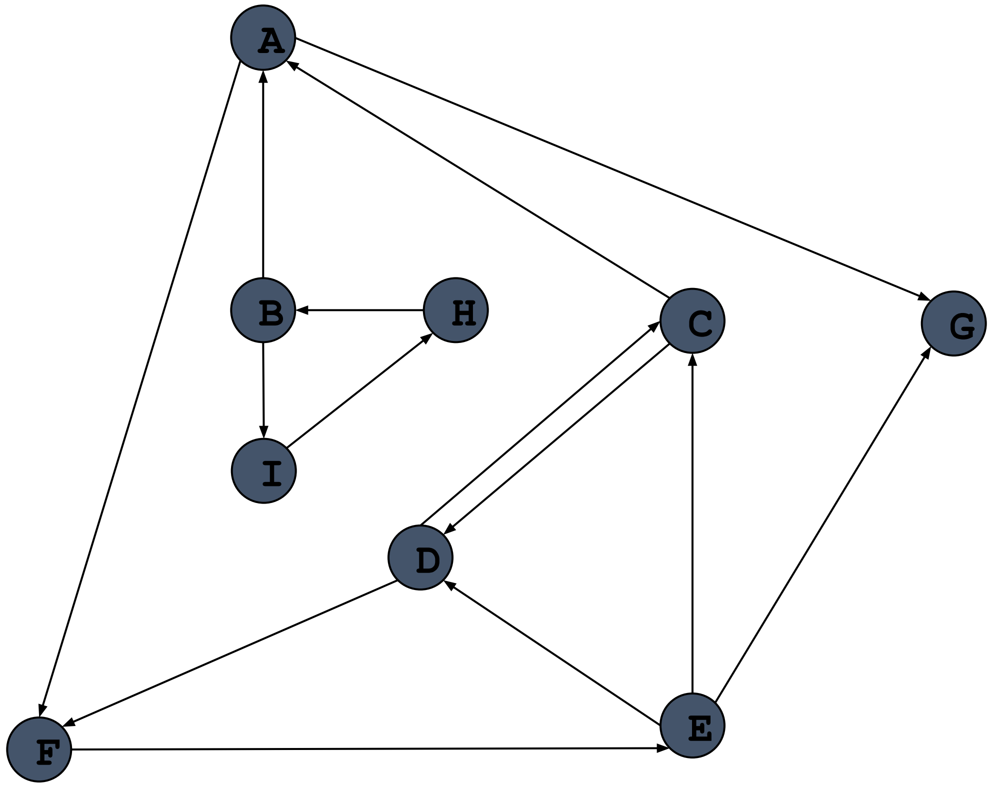
Algorithm :

1. Initialize all nodes to the ready state.
2. Push the starting node onto the STACK and change its status to the waiting state.
3. Repeat steps 4 and 5 until STACK is empty.
4. Pop the top node N of stack. Process N and change its status to the processed state.
5. Push onto STACK all the neighbours of N that are still in the ready state and change their status to the waiting state.

   End of step 3

6. Exit.

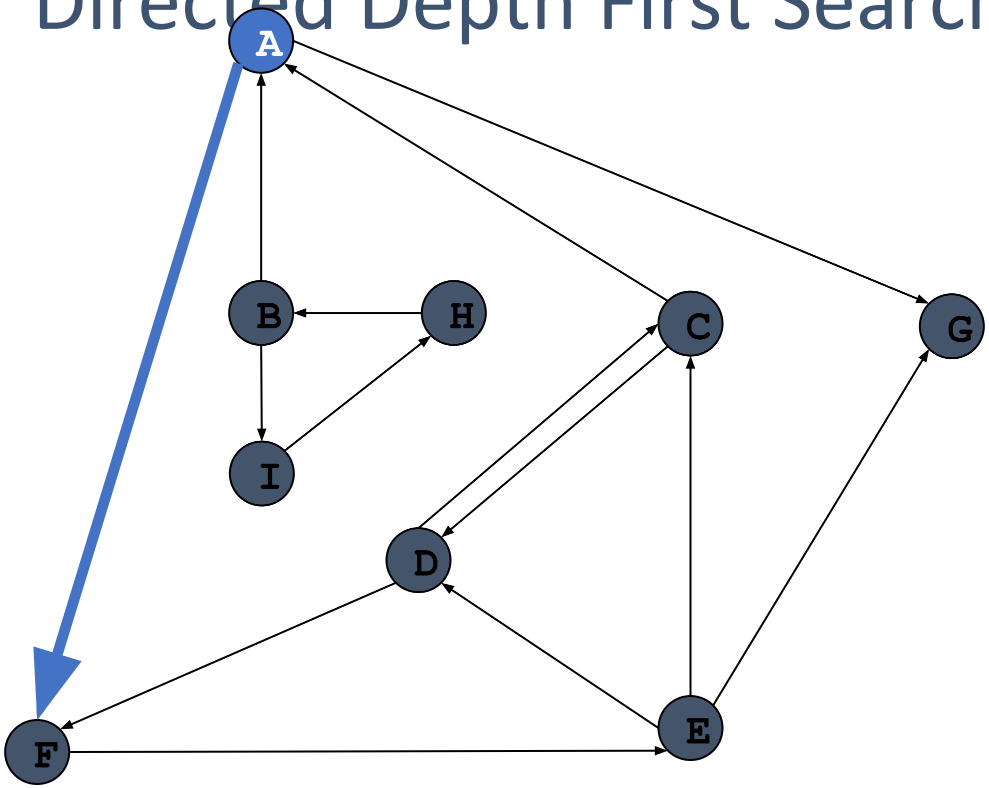
Directed Depth First Search



Adjacency Lists

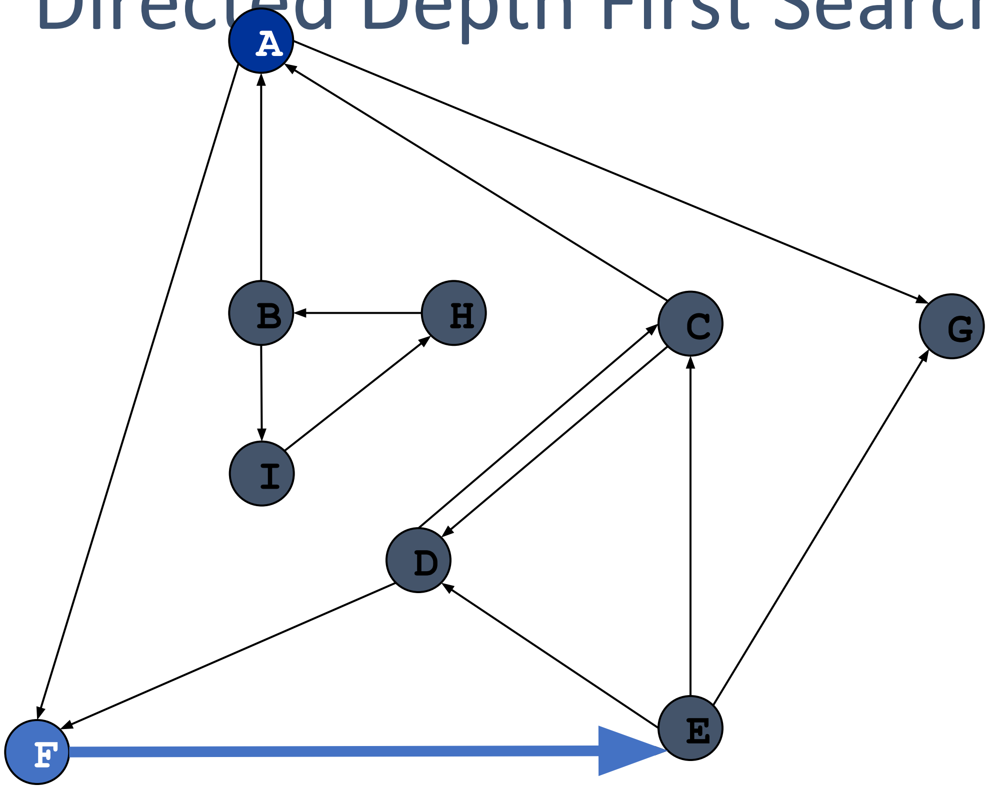
A: F G
B: A H
C: A D
D: C F
E: C D G
F: E
G:
H: B
I: H

Directed Depth First Search



Function call stack:

Directed Depth First Search



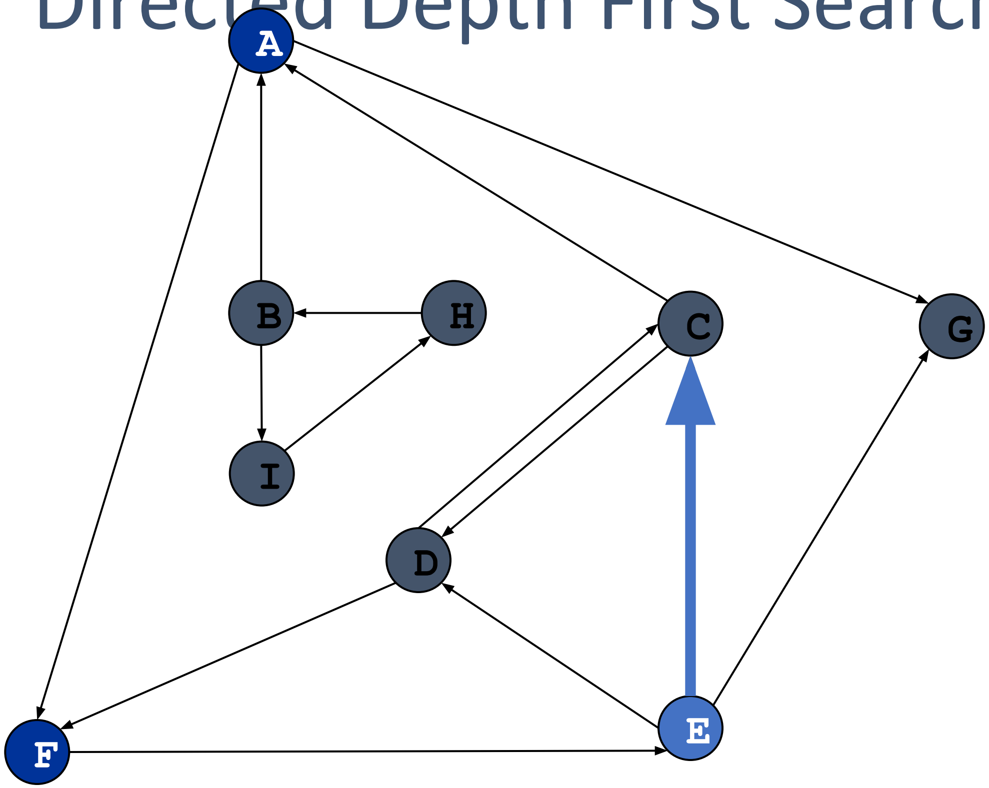
visit(F)

F-E

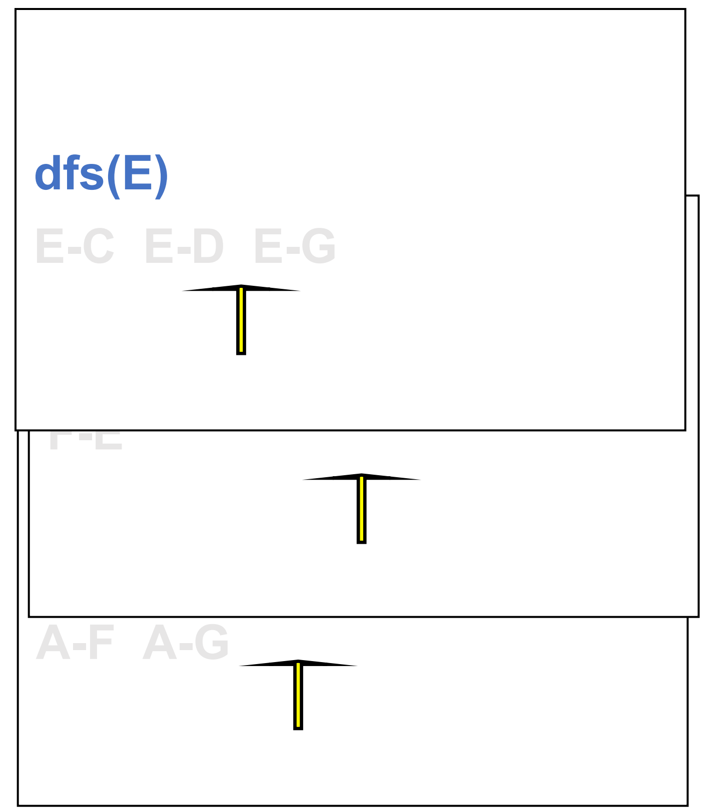
Function call stack:

C V RAM

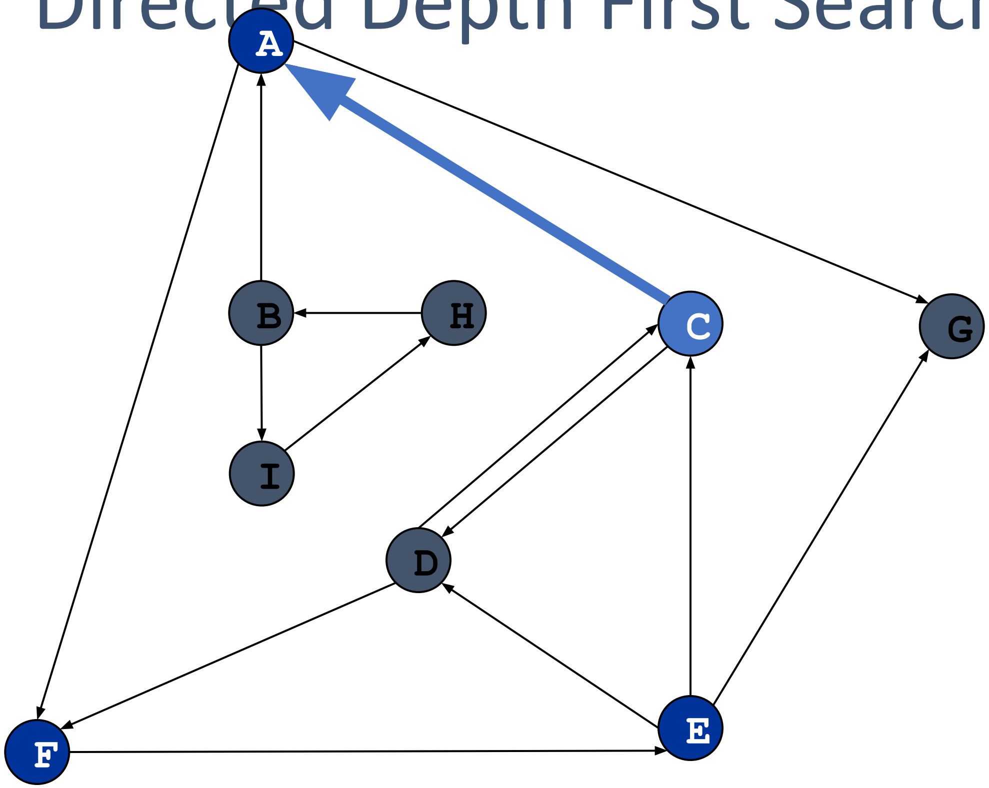
Directed Depth First Search

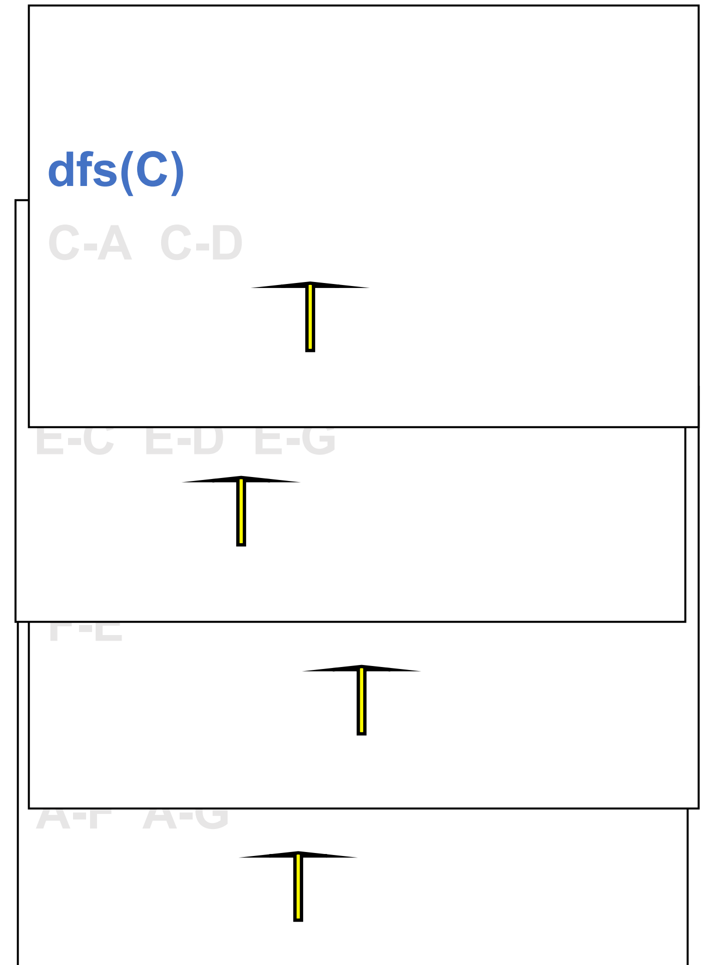


Function call stack:



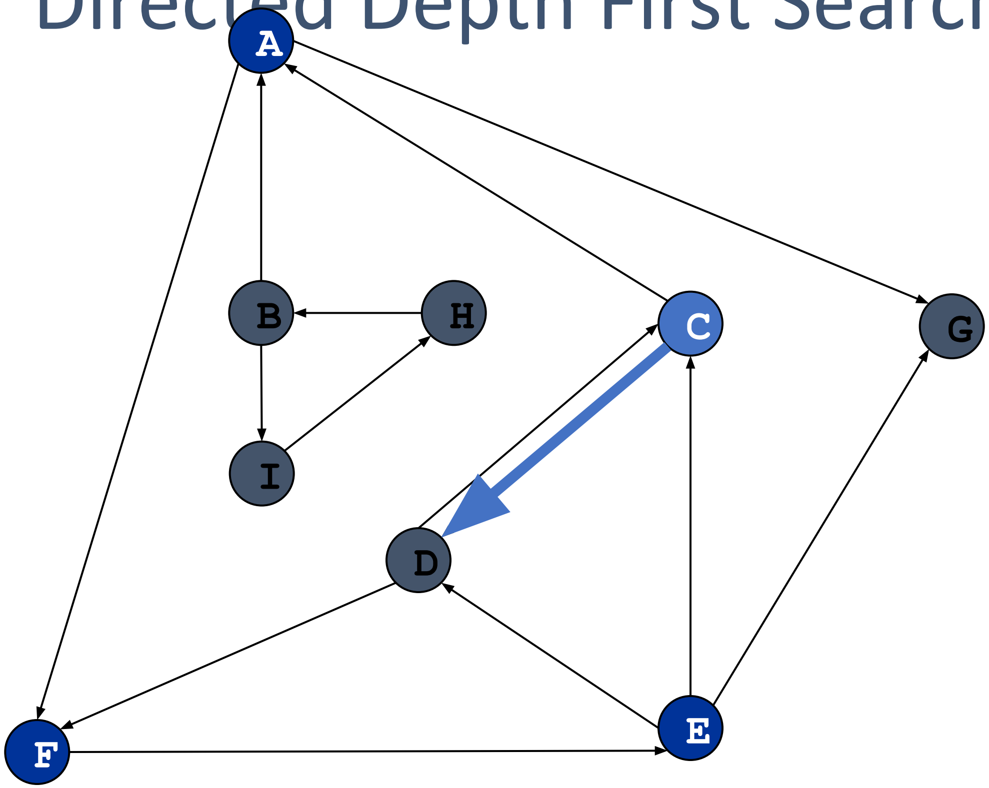
Directed Depth First Search



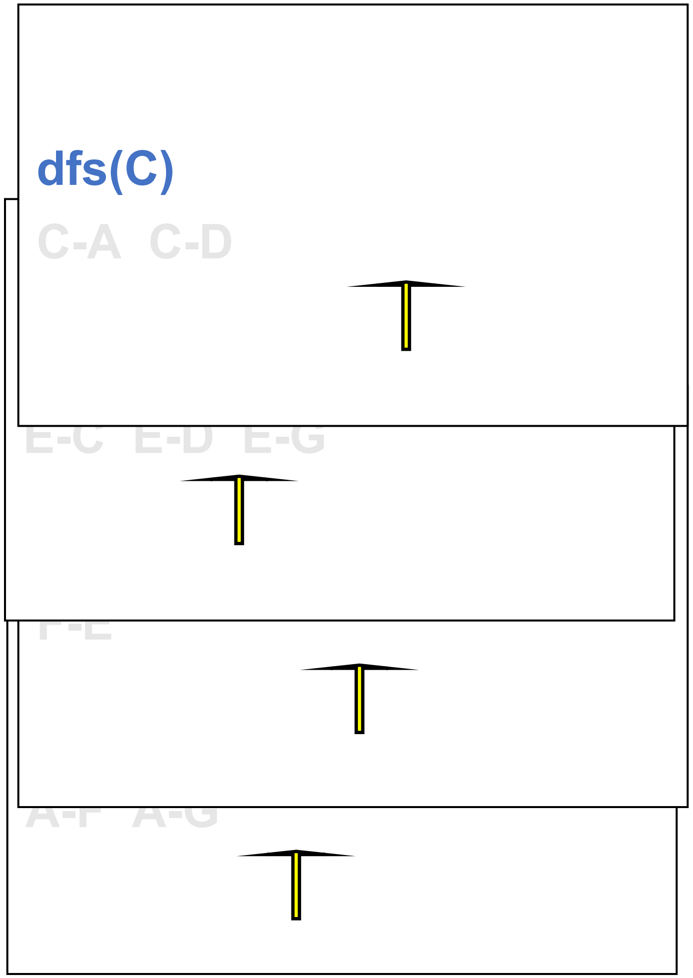


Function call stack:

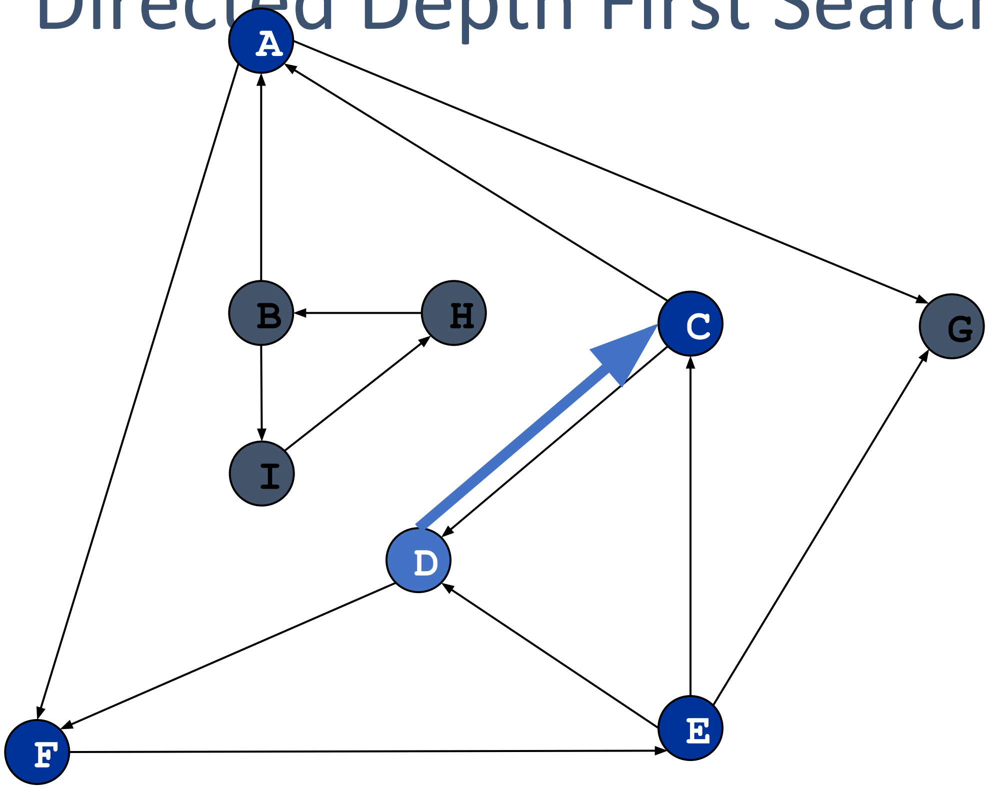
Directed Depth First Search

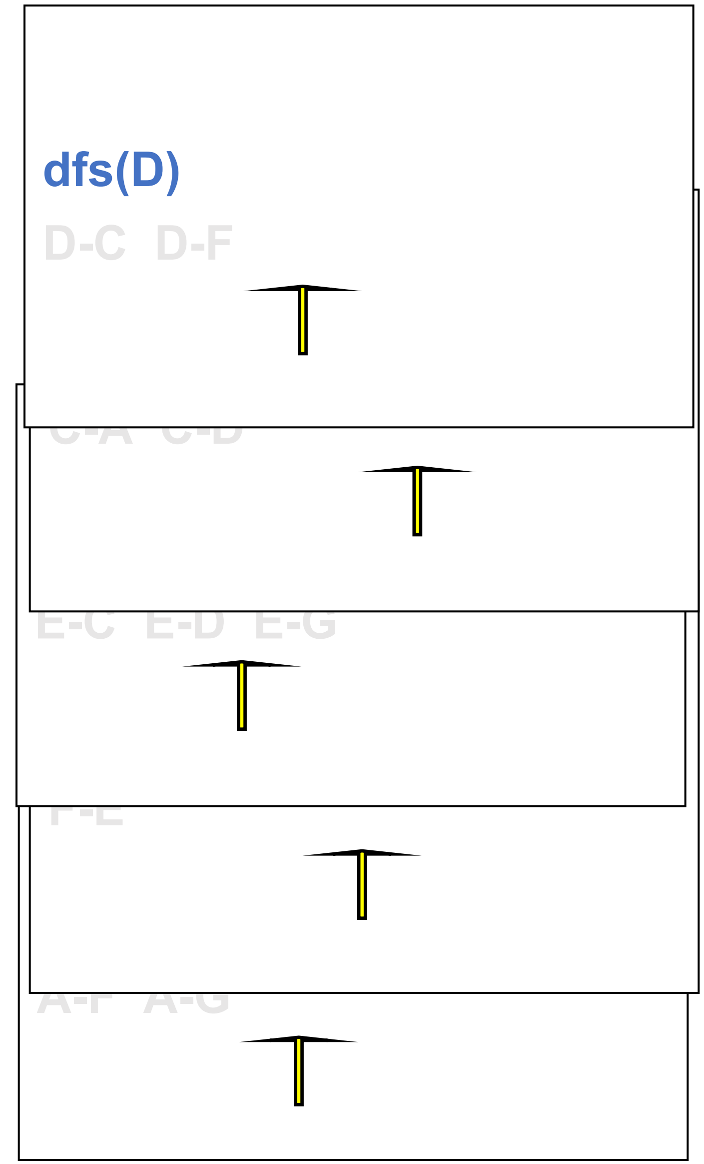


Function call stack:



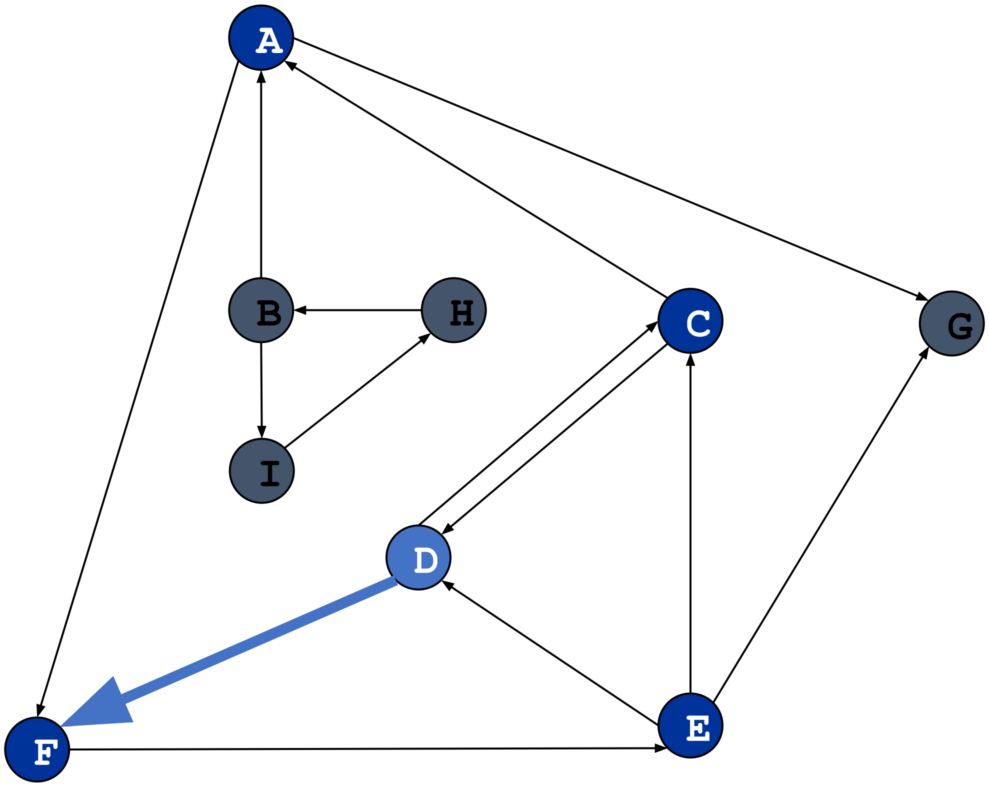
Directed Depth First Search

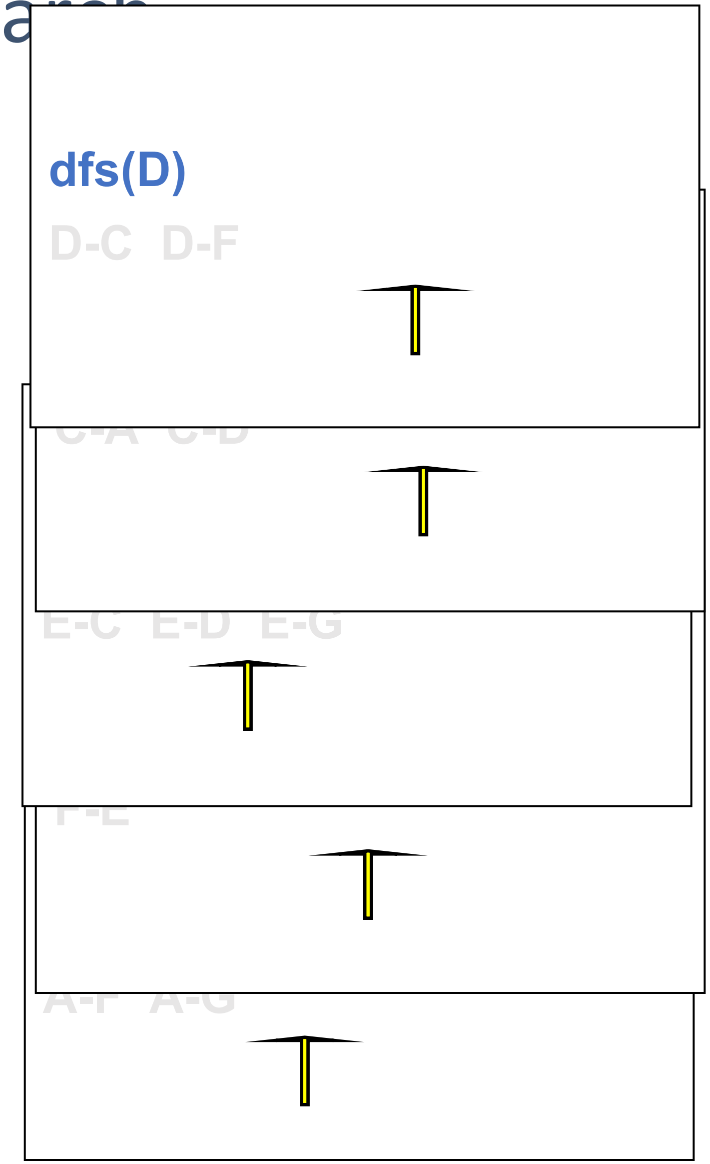




Function call stack:

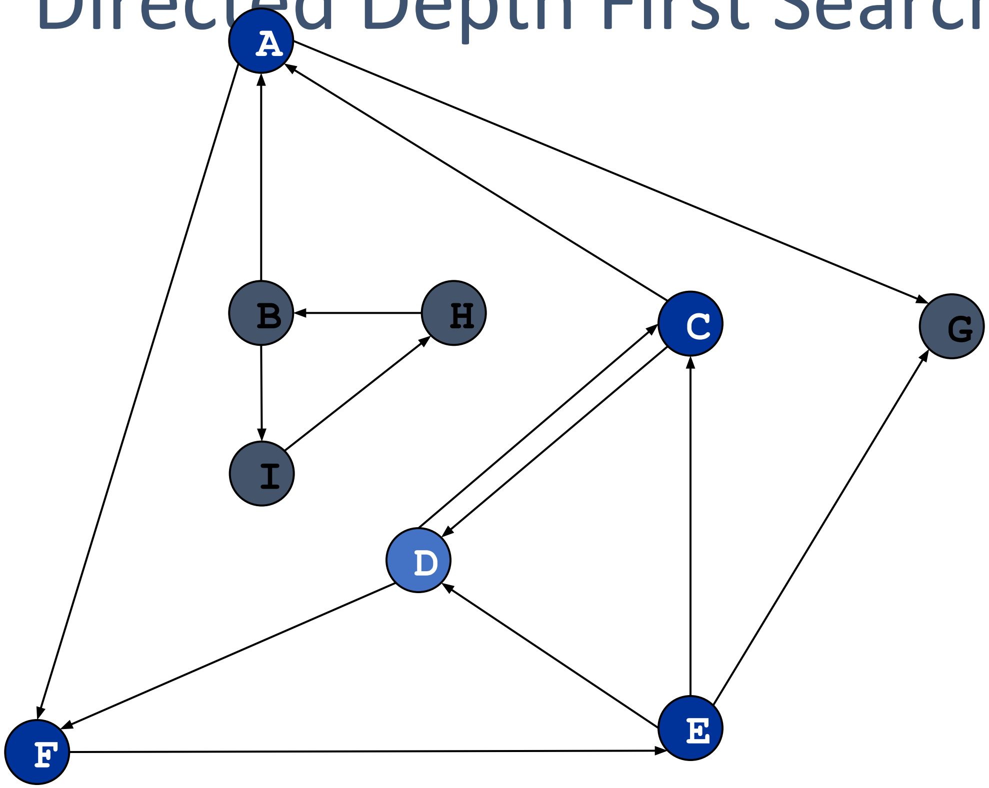
Directed Depth First Search



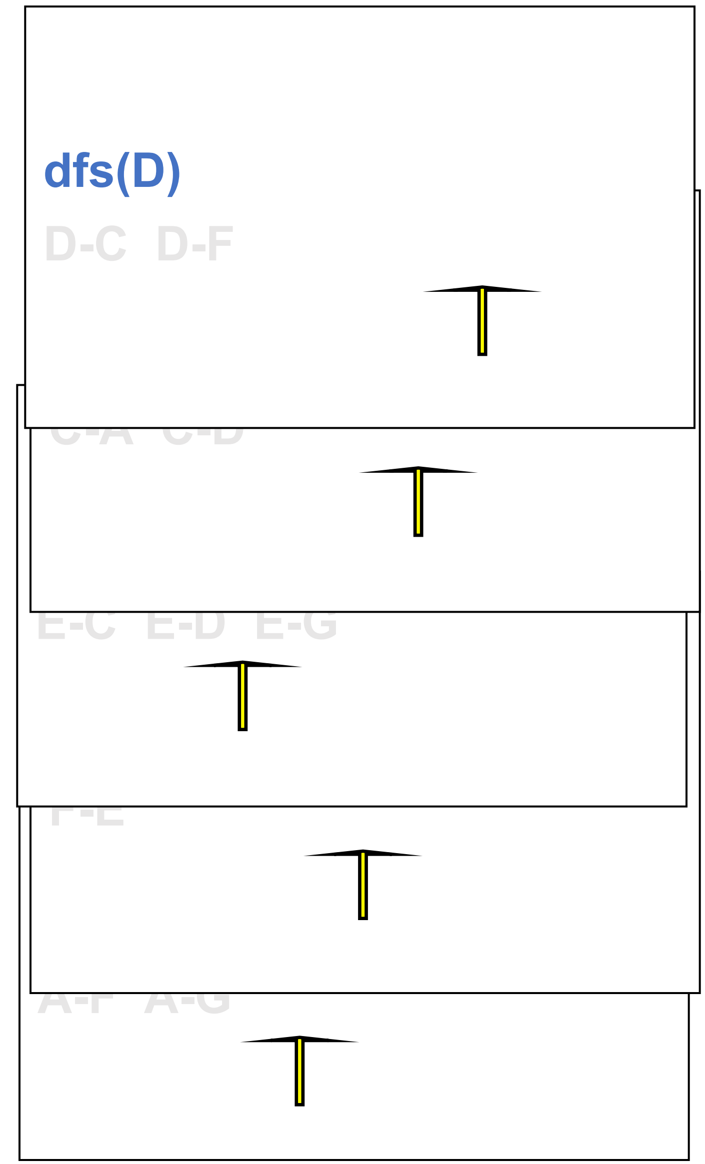


Function call stack:

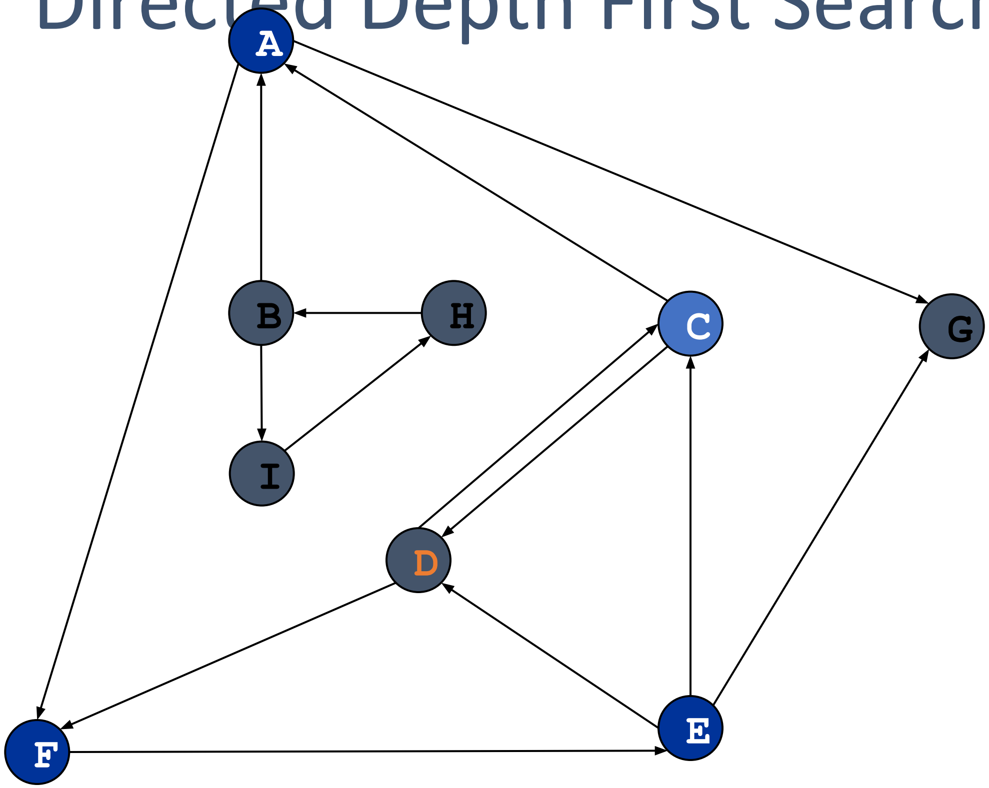
Directed Depth First Search

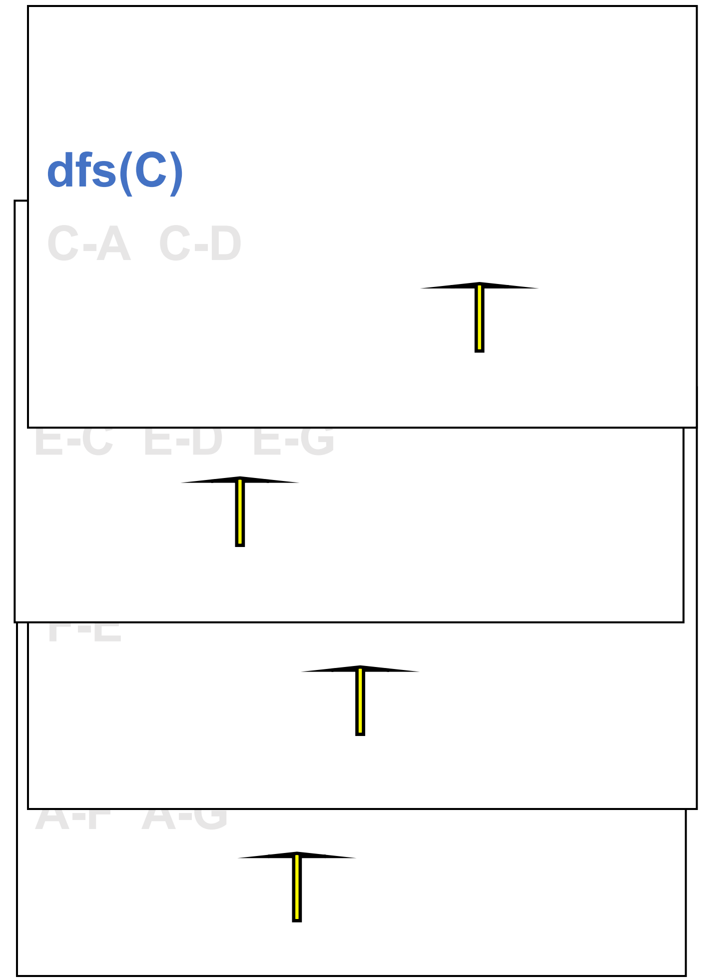


Function call stack:



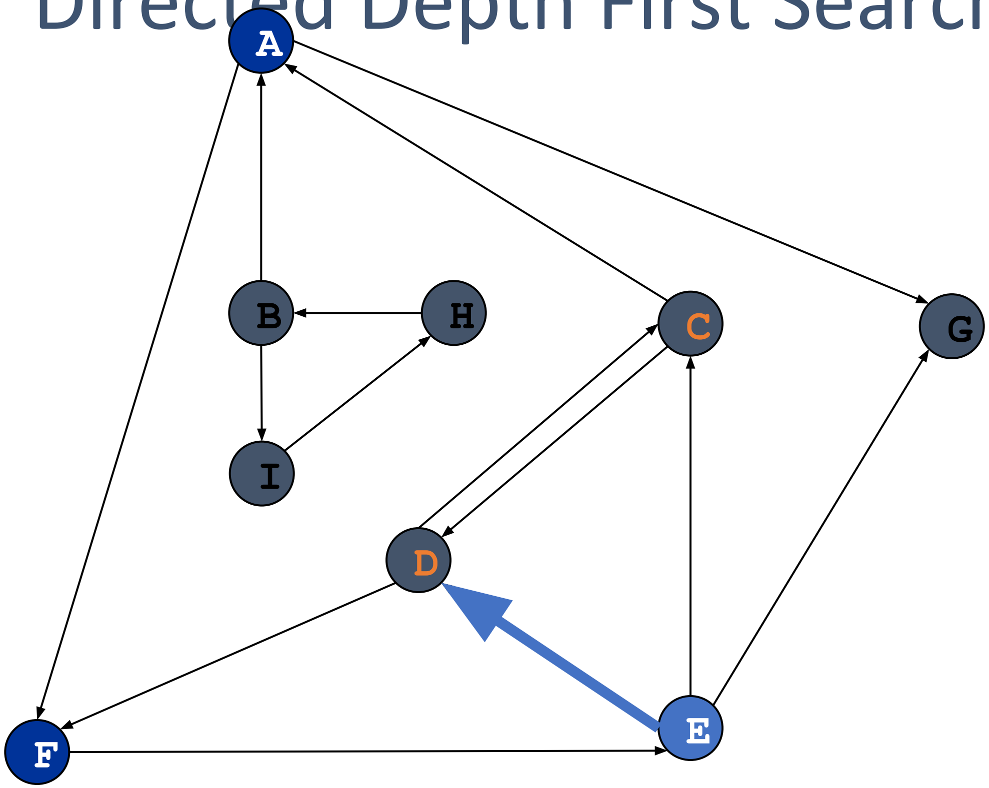
Directed Depth First Search

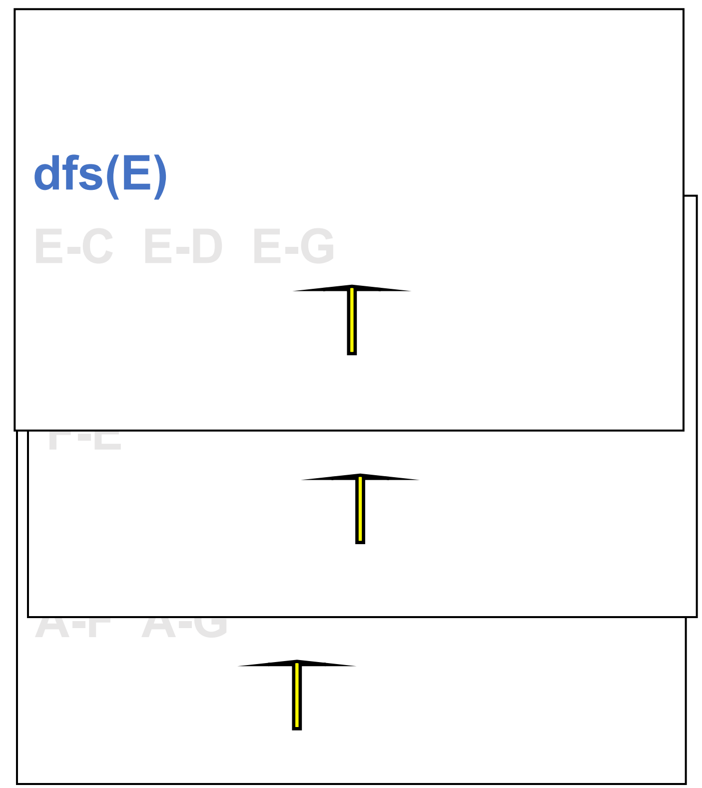




Function call stack:

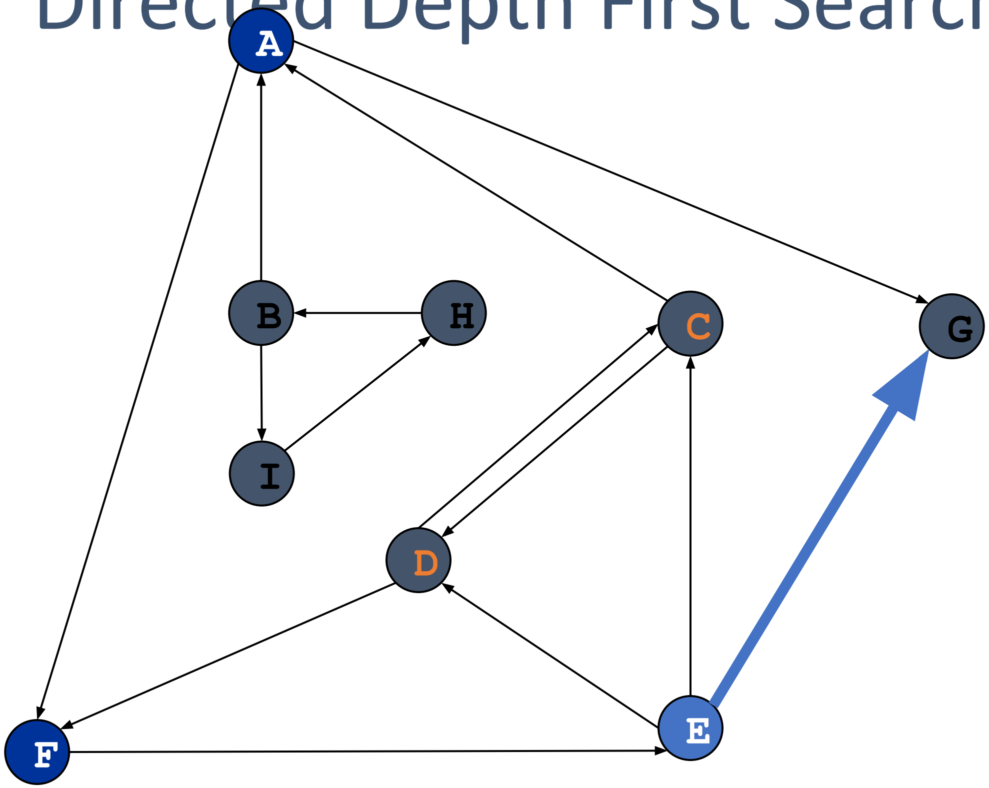
Directed Depth First Search



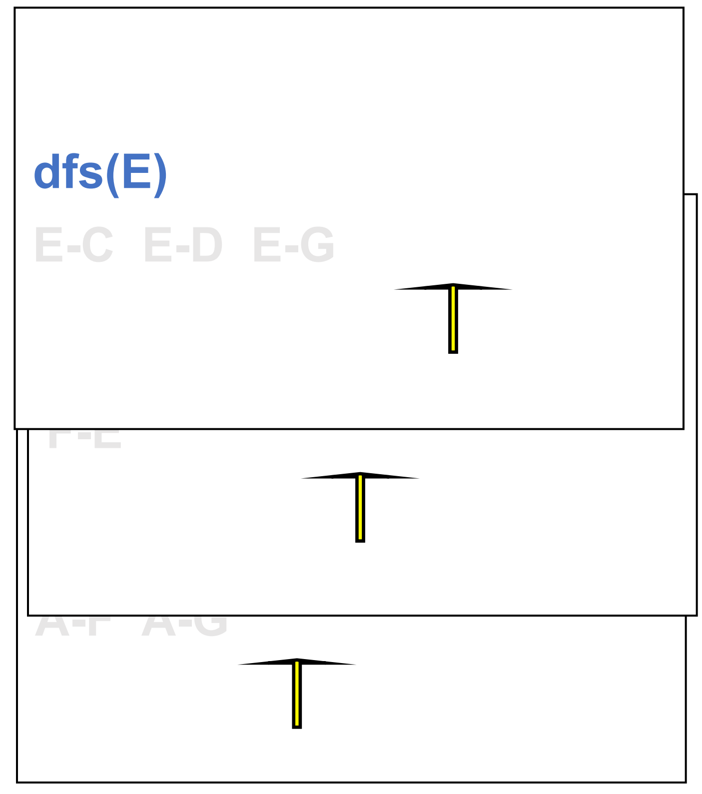


Function call stack:

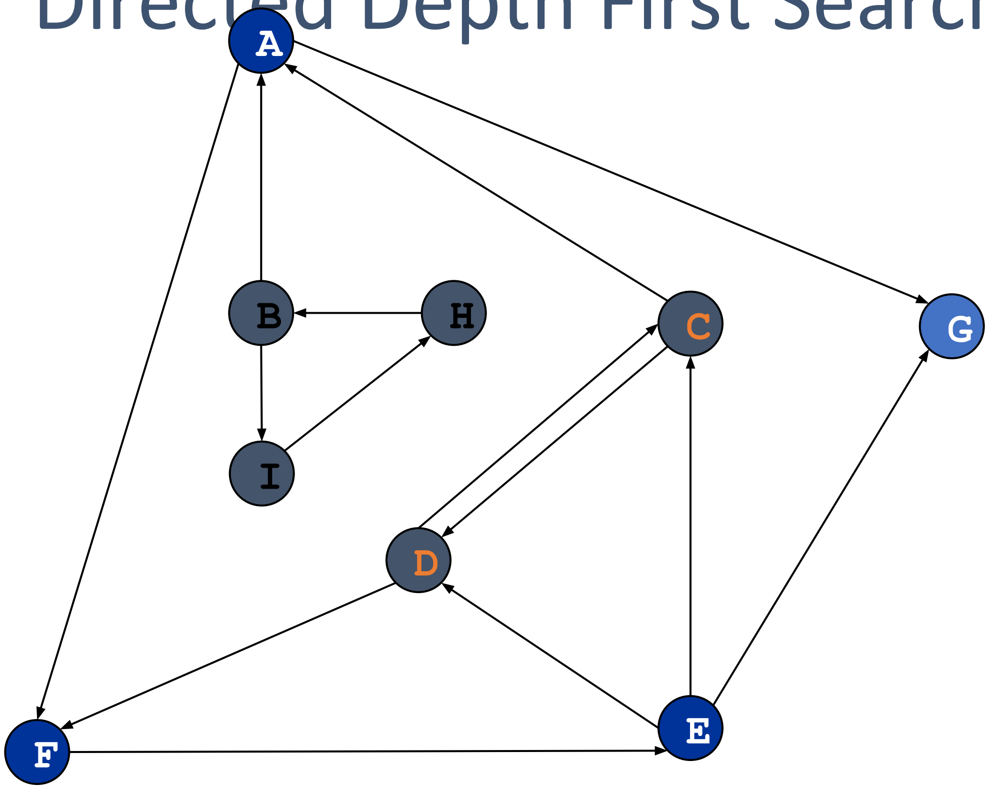
Directed Depth First Search



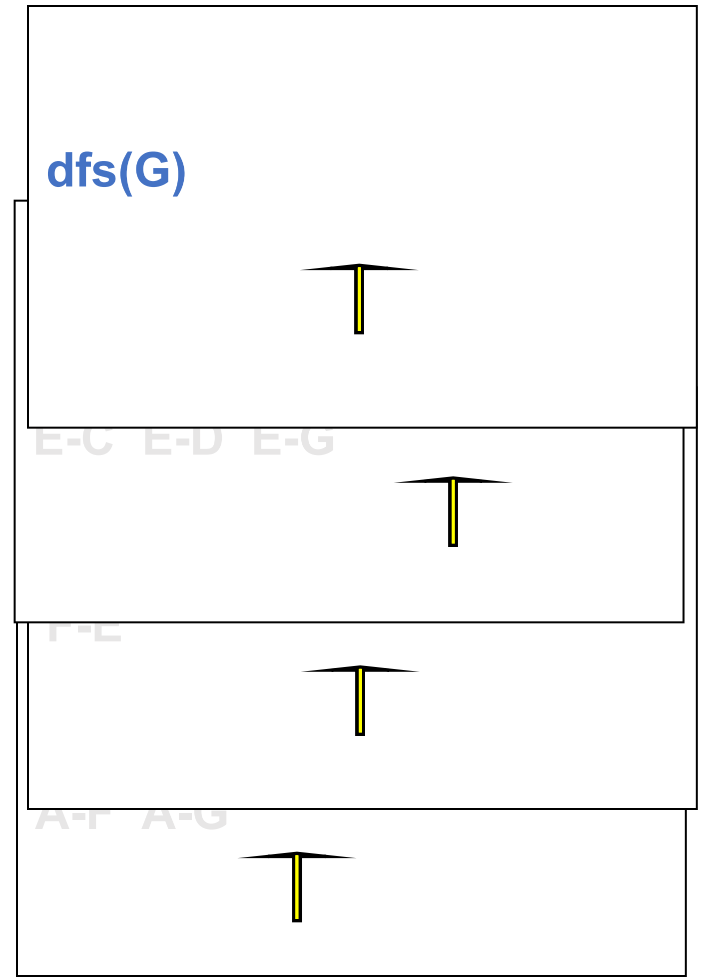
Function call stack:



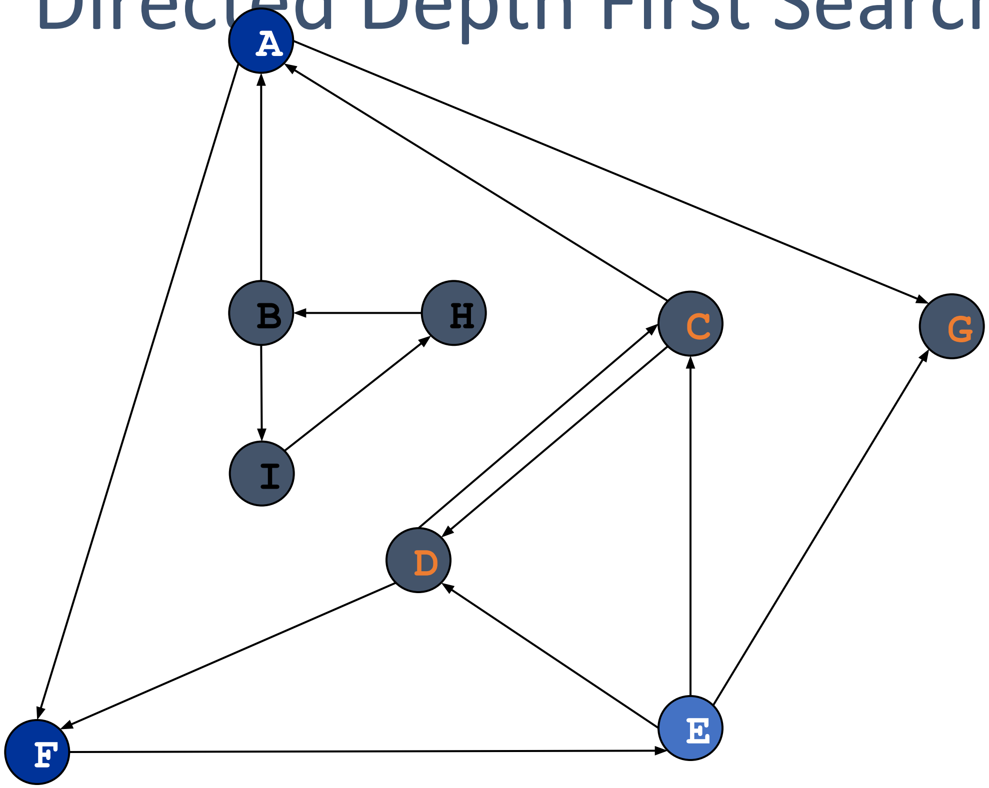
Directed Depth First Search



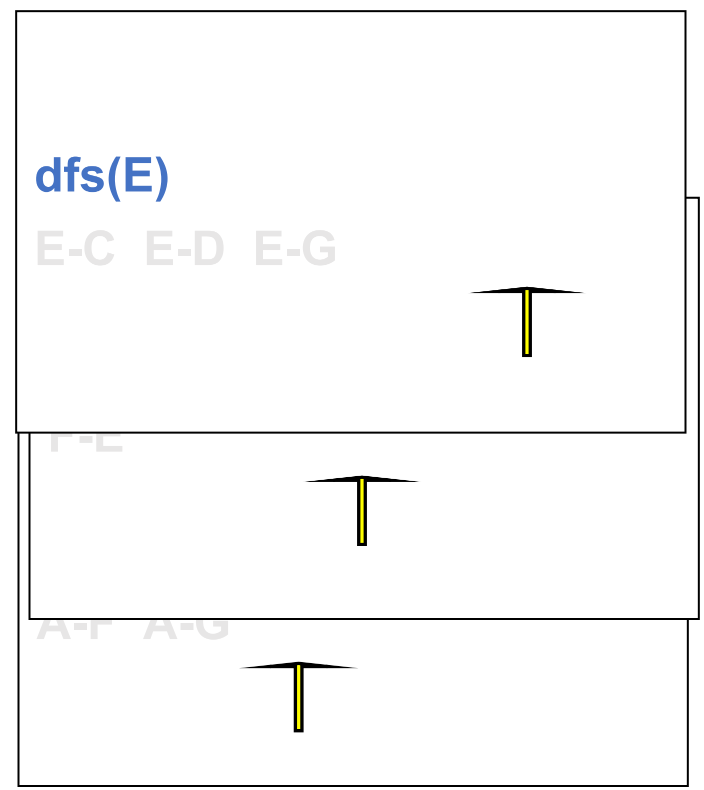
Function call stack:



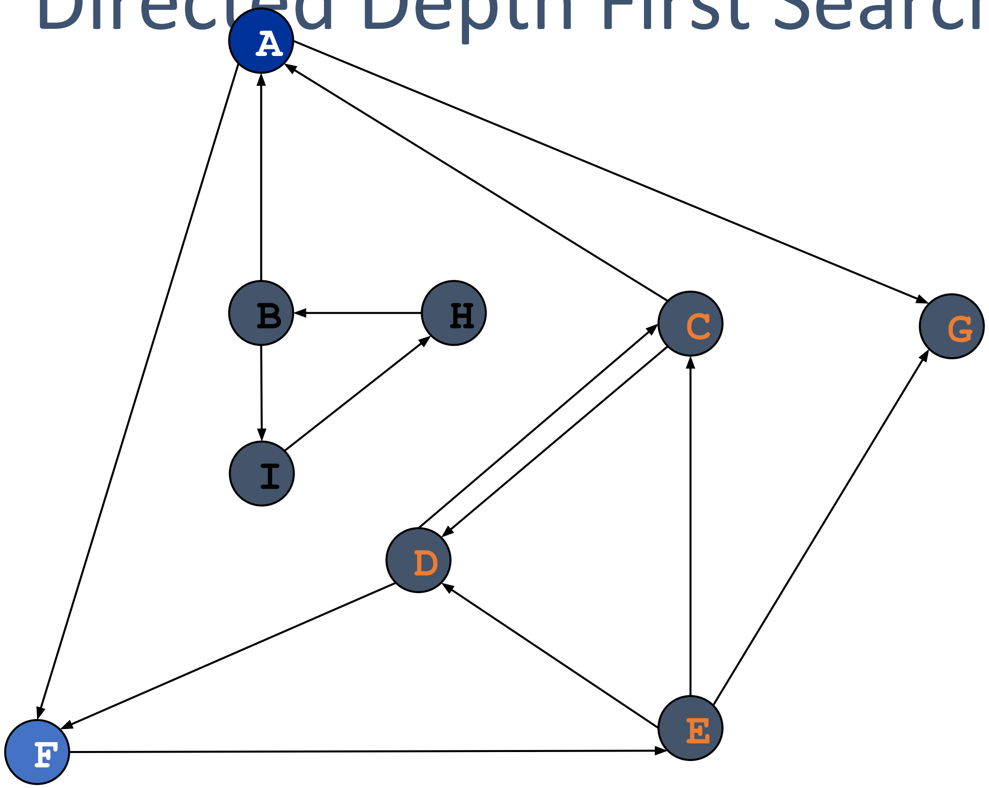
Directed Depth First Search



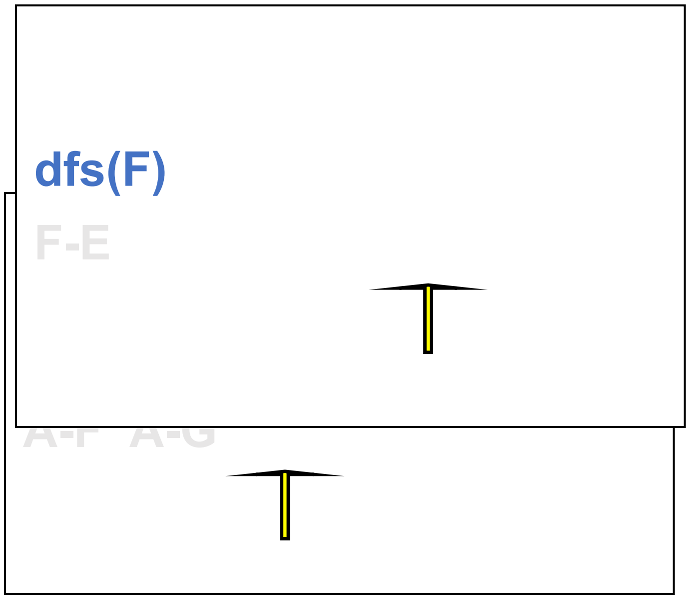
Function call stack:



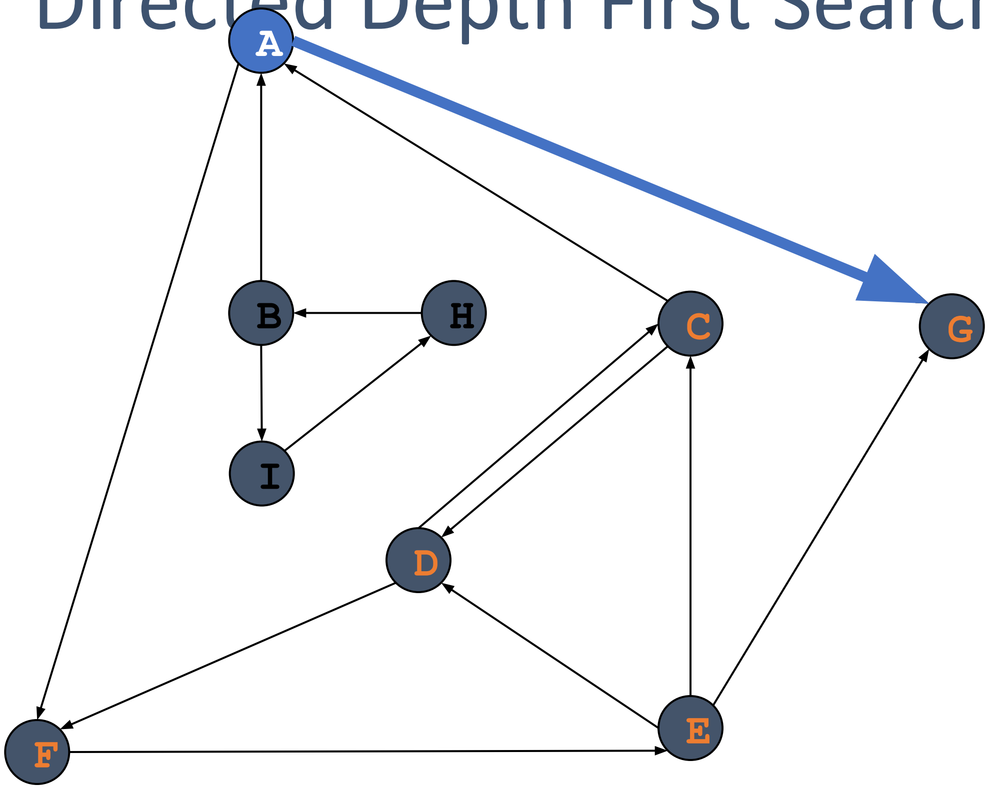
Directed Depth First Search



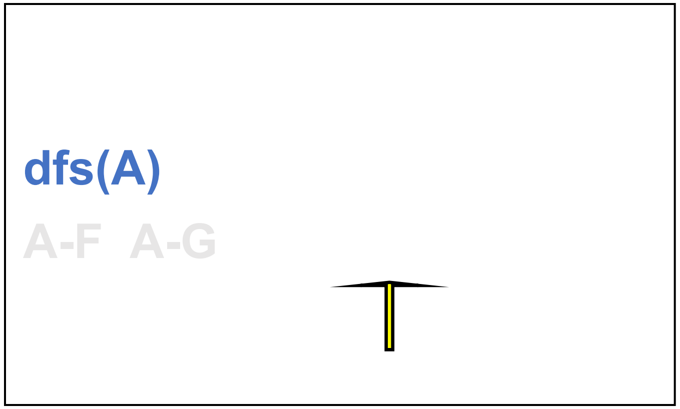
Function call stack:



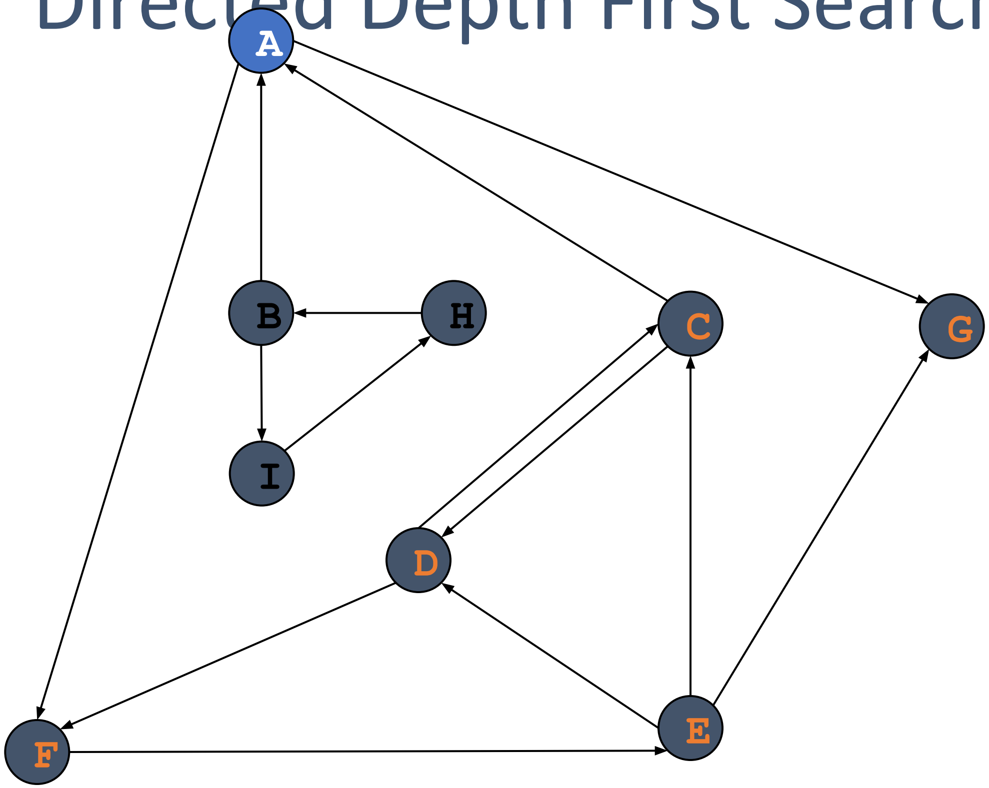
Directed Depth First Search



Function call stack:

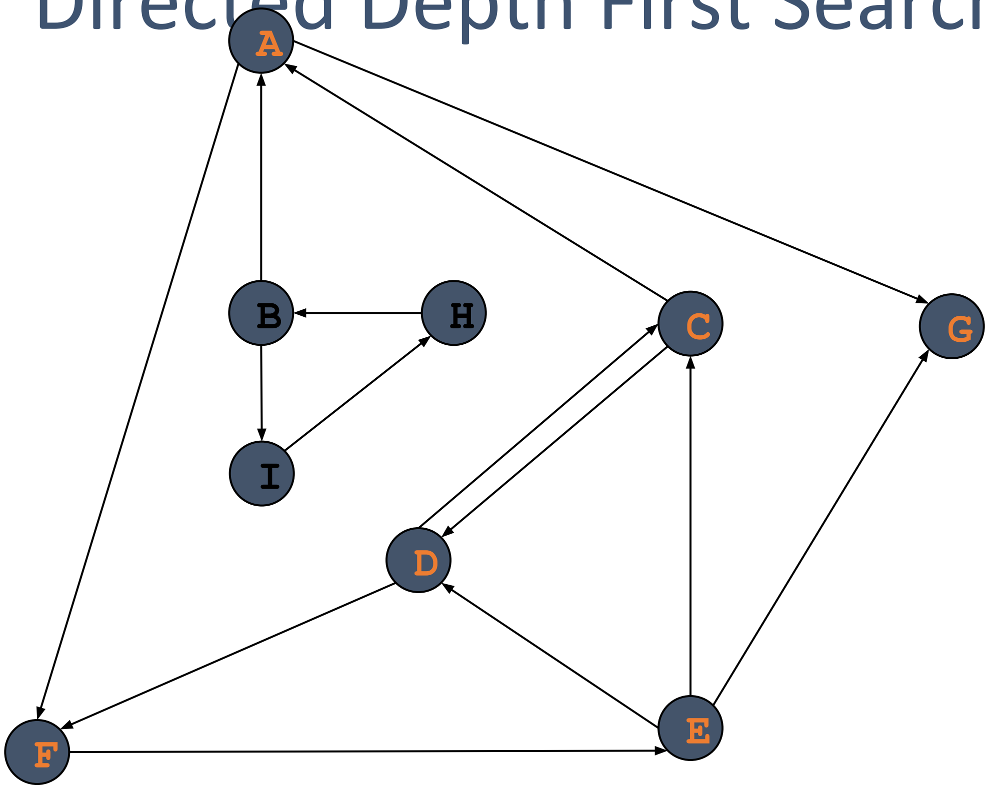


Directed Depth First Search



Function call stack:

Directed Depth First Search



Nodes reachable from A: A, C, D, E, F, G

Shortest-Path Algorithm
(Taking A weighted graph G with M nodes of matrix W. The algorithm finds a matrix Q such that Q[I,J] is the length of a shortest path from nodes V_i to V_j

Step-1 : Repeat for I,J = 1,2…M [initializes Q]

If W[I,J]=0 then set Q[I,J]=infinity;

[infinity may be taken a big number i.e.9999]

Else

Set Q[I,J]=W[I,J]

End loop

Step-2:Repeat steps 3 and 4 for K=1,2…M [update Q]

Step-3: Repeat step-4 for I=1,2…M

Step-4: Repeat for J=1,2…M

Set Q[I,J]=MIN(Q[I,J], (Q[I,K]+Q[K,J]) [finding minimum value]

End loop.

End of step-3 loop .

End of step-2 loop.

Step-5 Exit.

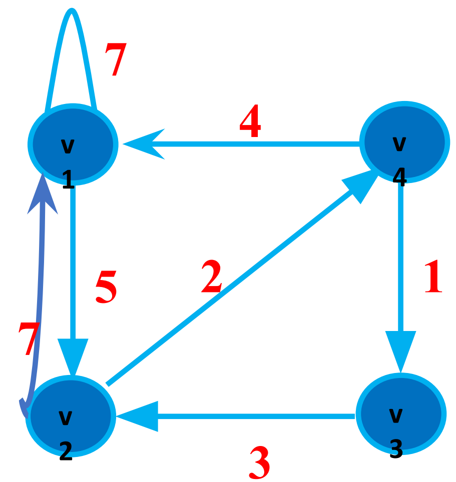
Warshall's shortest path algorithm Example

W =

	V_1	V_2	V_3	V_4
V_1	7	5	0	0
V_2	7	0	0	2
V_3	0	3	0	0
V_4	4	0	1	0

Q_0 =

	V_1	V_2	V_3	V_4
V_1	7	5	999	999
V_2	7	999	999	2
V_3	999	3	999	999
V_4	4	999	1	999



Q_1 =

	V_1	V_2	V_3	V_4
V_1	7	5	999	999
V_2	7	12	999	2
V_3	999	3	999	999
V_4	4	9	1	999

Q_2 =

	V_1	V_2	V_3	V_4
V_1	7	5	999	7
V_2	7	12	999	2
V_3	10	3	999	5
V_4	4	9	1	11

Q_3 =

	V_1	V_2	V_3	V_4
V_1	7	5	999	7
V_2	7	12	999	2
V_3	10	3	999	5
V_4	4	4	1	6

Q_4 =

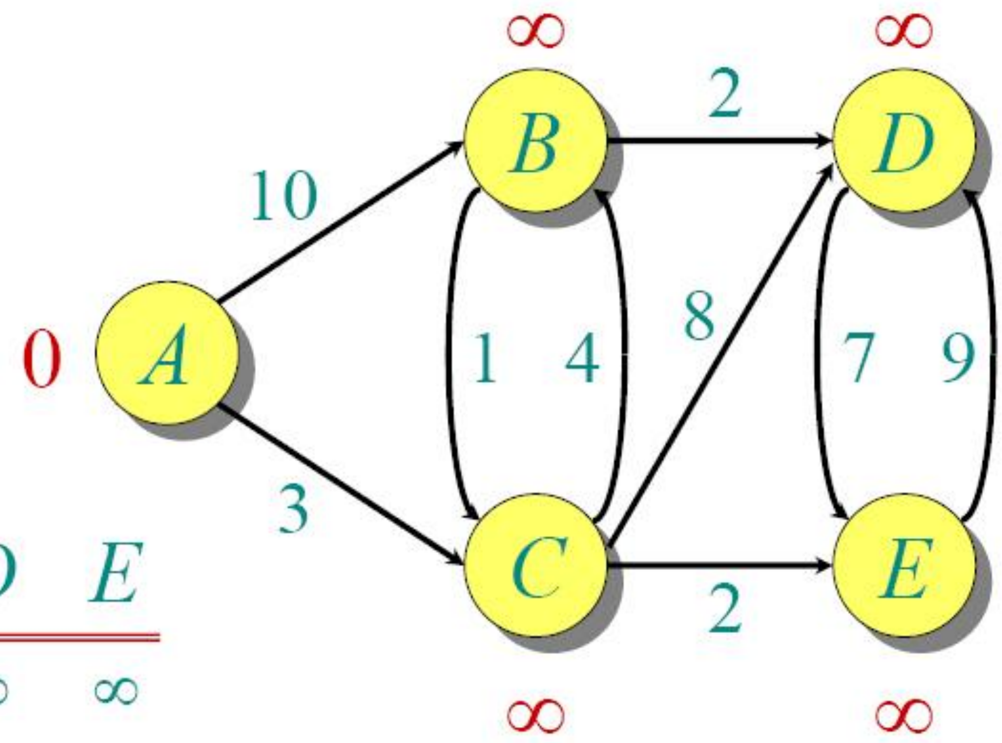
	V_1	V_2	V_3	V_4
V_1	7	5	8	7
V_2	7	11	3	2
V_3	9	3	6	5
V_4	4	4	1	6

Dijkstra's algorithm - Pseudocode

```
dist[s] ←0                              (distance to source vertex is zero)
for  all v ∈ V–{s}
        do  dist[v] ←∞                  (set all other distances to infinity)
S←∅                              (S, the set of visited vertices is initially empty)
Q←V                              (Q, the queue initially contains all vertices)
while Q ≠∅                       (while the queue is not empty)
do   u ← mindistance(Q,dist) (select the element of Q with the min. distance)
      S←S ∪ {u}                         (add u to list of visited vertices)
       for all v ∈ neighbors[u]
              do  if   dist[v] > dist[u] + w(u, v)           (if new shortest path found)
                         then      d[v] ←d[u] + w(u, v)       (set new value of shortest path)
                 (if desired, add traceback code)
return dist
```

Dijkstra Animated Example

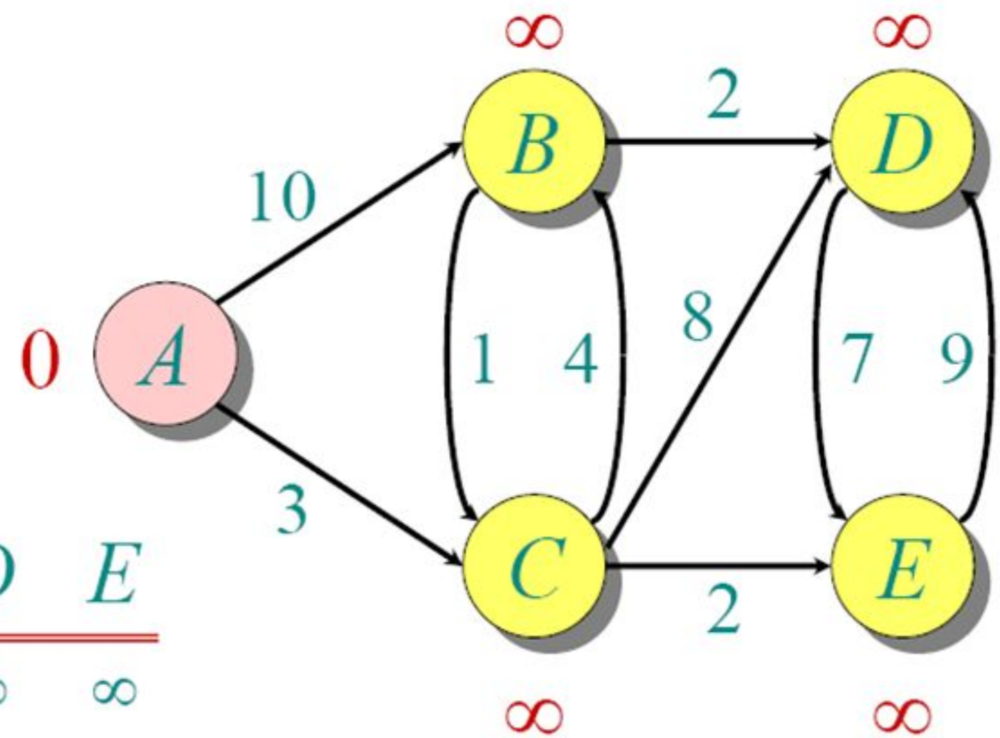
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

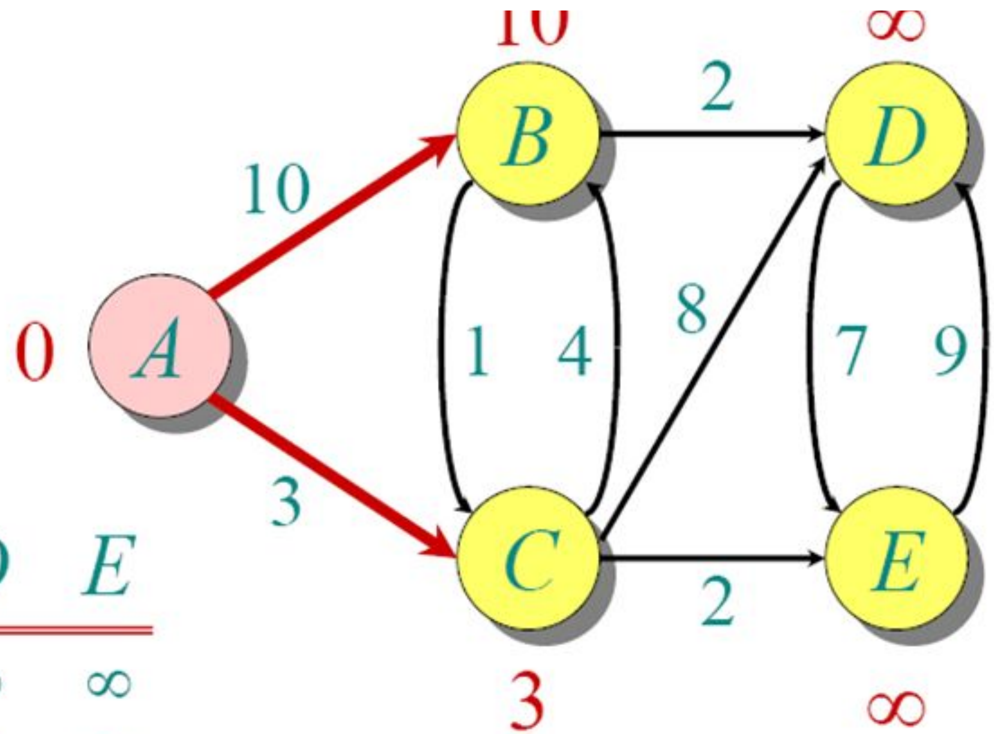
S: {}

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

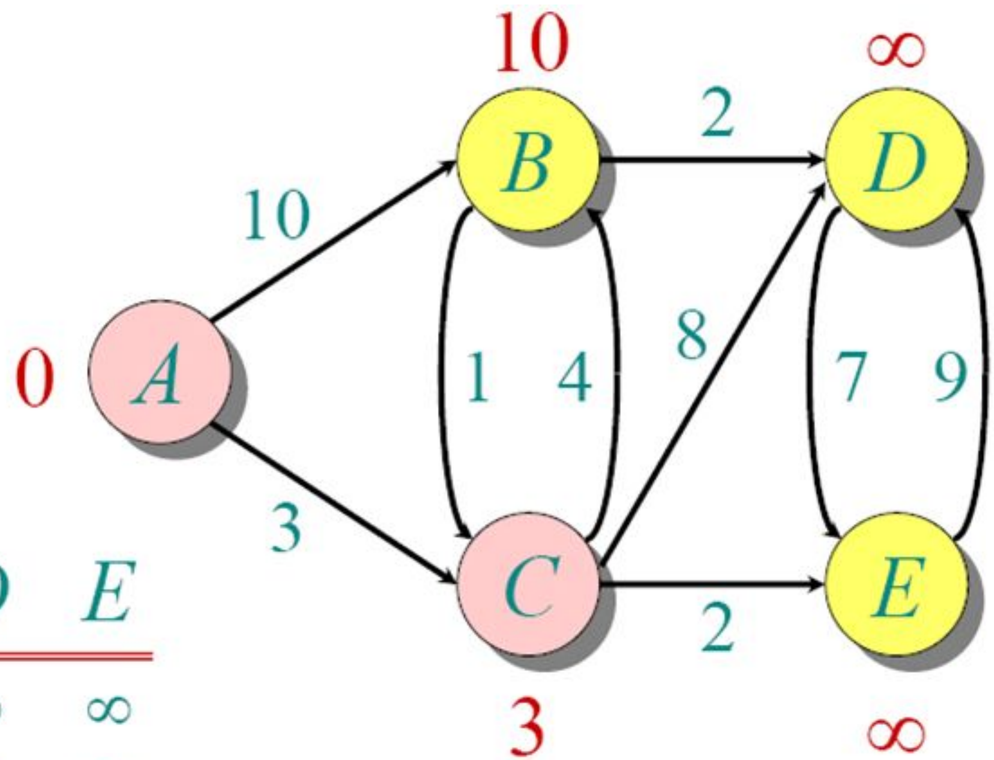
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

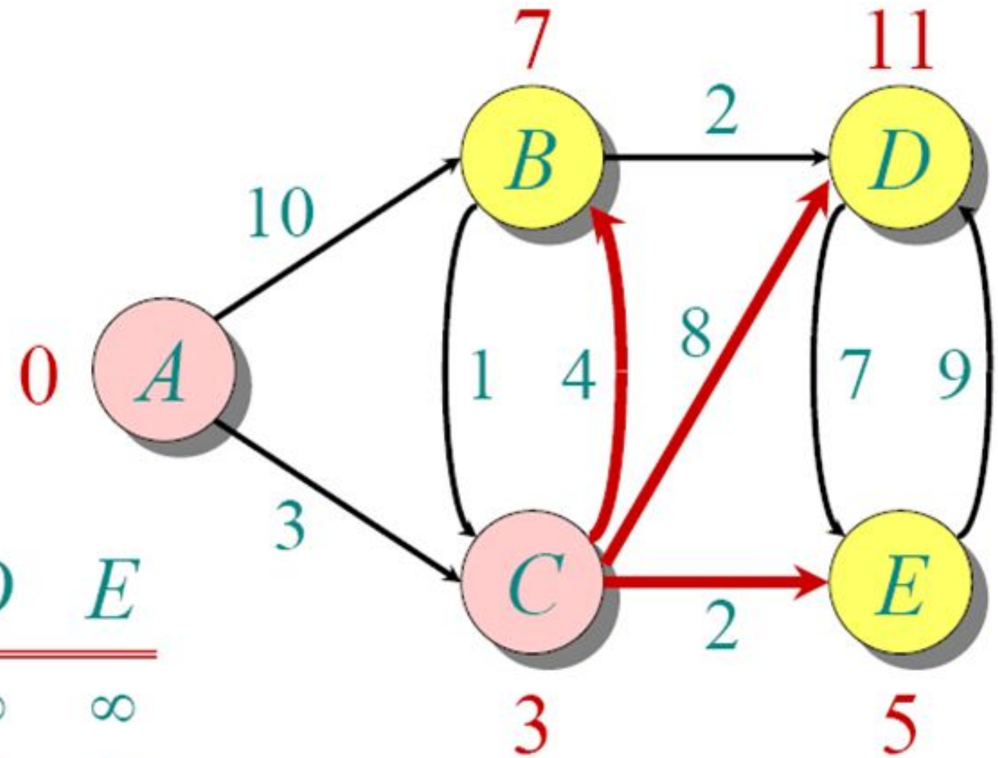
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

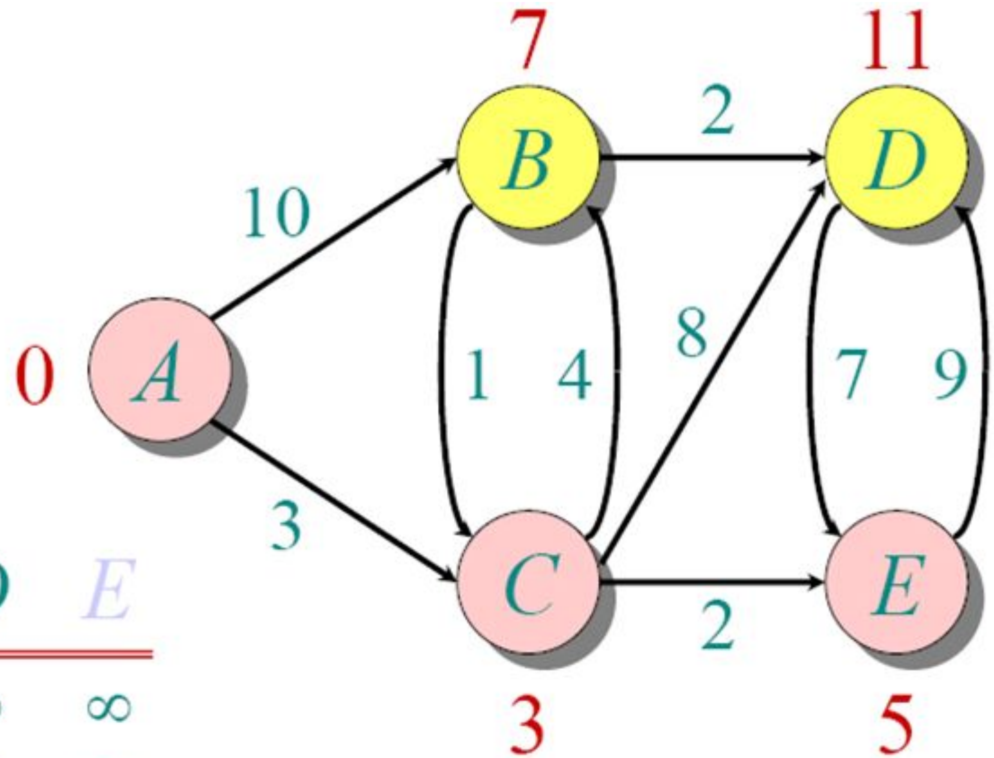
Dijkstra Animated Example



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	5

S: { A, C }

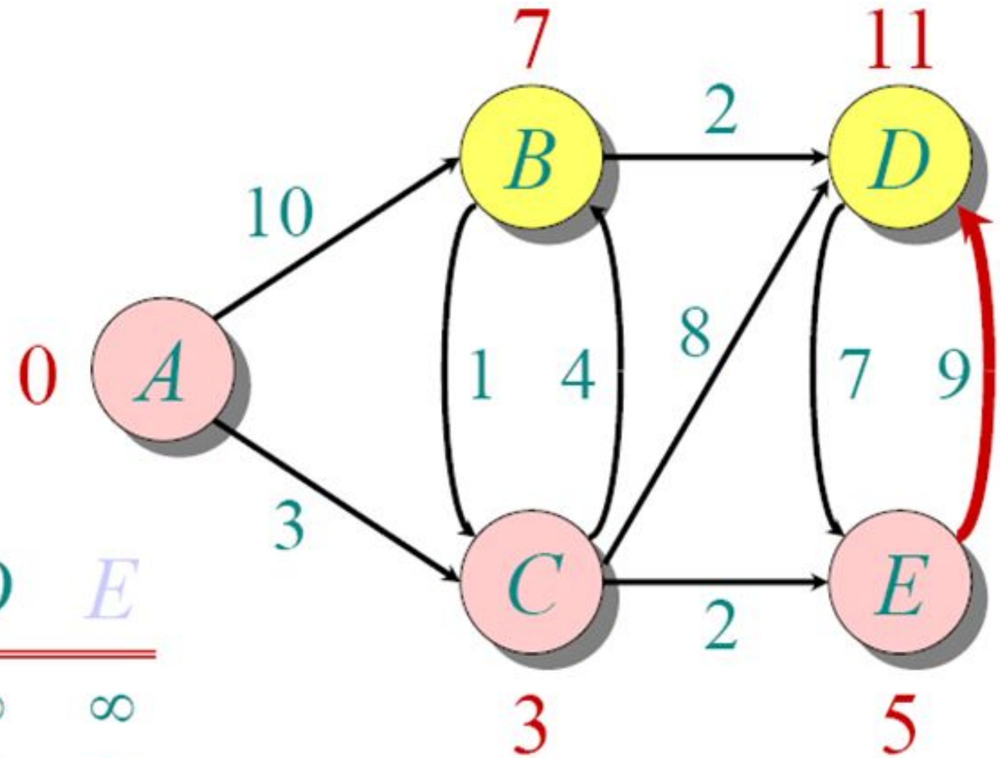
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

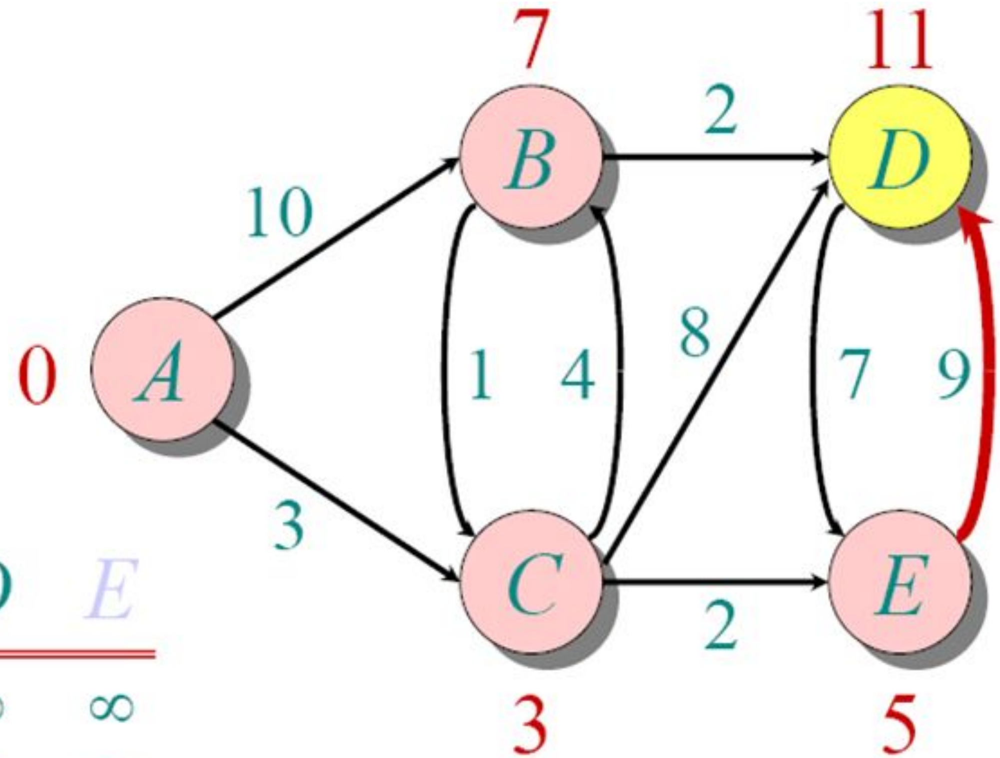
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

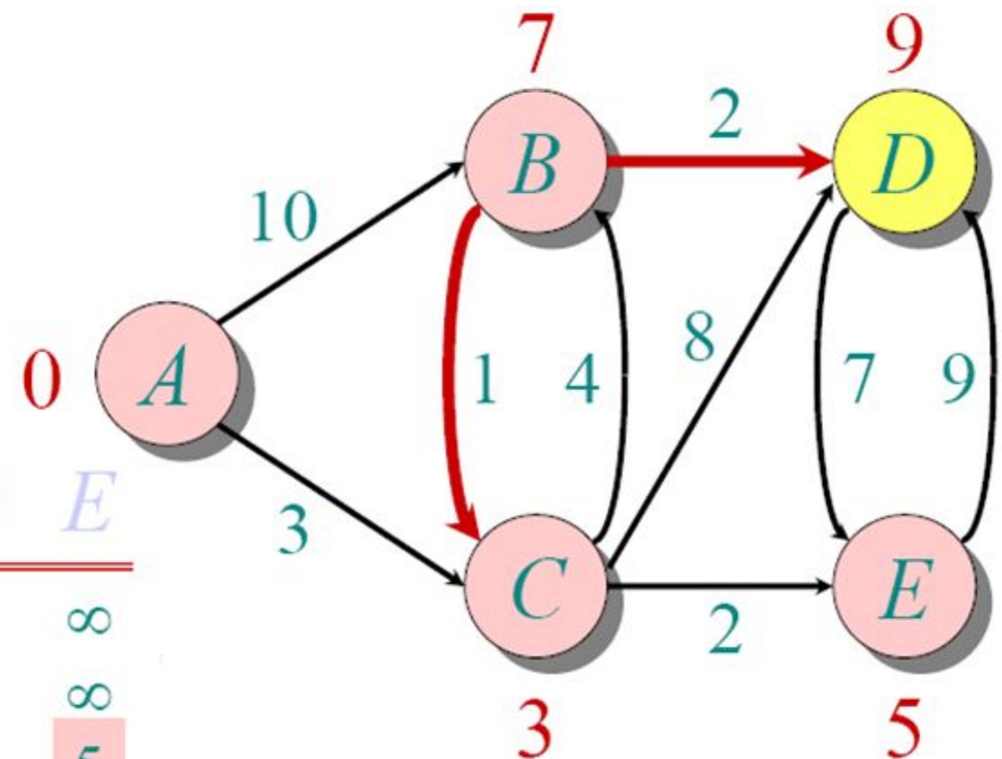
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

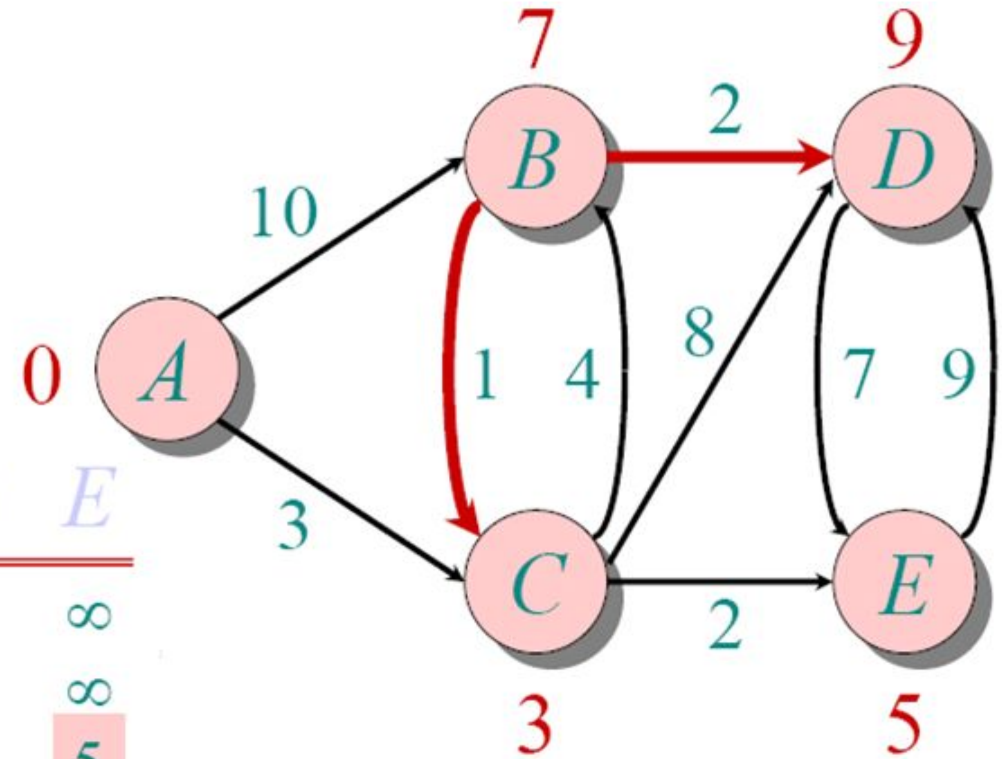
Dijkstra Animated Example



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				9	

S: { A, C, E, B }

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { *A*, *C*, *E*, *B*, *D* }

Implementations and Running Times

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of

$O(|V|^2 + |E|)$

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

$O((|E|+|V|) \log |V|)$

Dijkstra's Algorithm - Why It Works

- As with all greedy algorithms, we need to make sure that it is a correct algorithm (e.g., it *always* returns the right solution if it is given correct input).

- A formal proof would take longer than this presentation, but we can understand how the argument works intuitively.

- If you can’t sleep unless you see a proof, see the second reference or ask us where you can find it.

DIJKSTRA'S ALGORITHM - WHY IT WORKS

- To understand how it works, we'll go over the previous example again. However, we need two mathematical results first:

- **Lemma 1**: Triangle inequality
  If $\delta(u,v)$ is the shortest path length between u and v,
  $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$
- **Lemma 2**:
  The subpath of any shortest path is itself a shortest path.

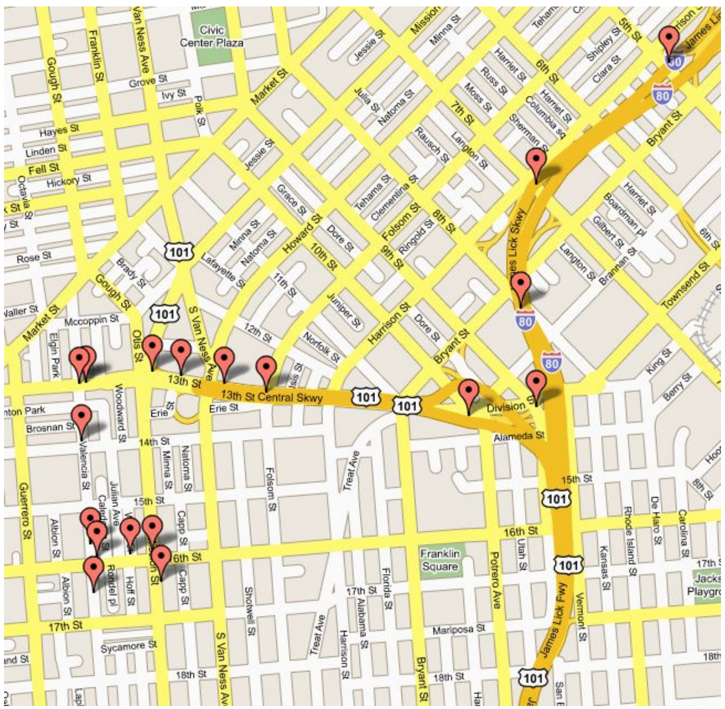
- The key is to understand why we can claim that anytime we put a new vertex in S, we can say that we already know the shortest path to it.
- Now, back to the example…

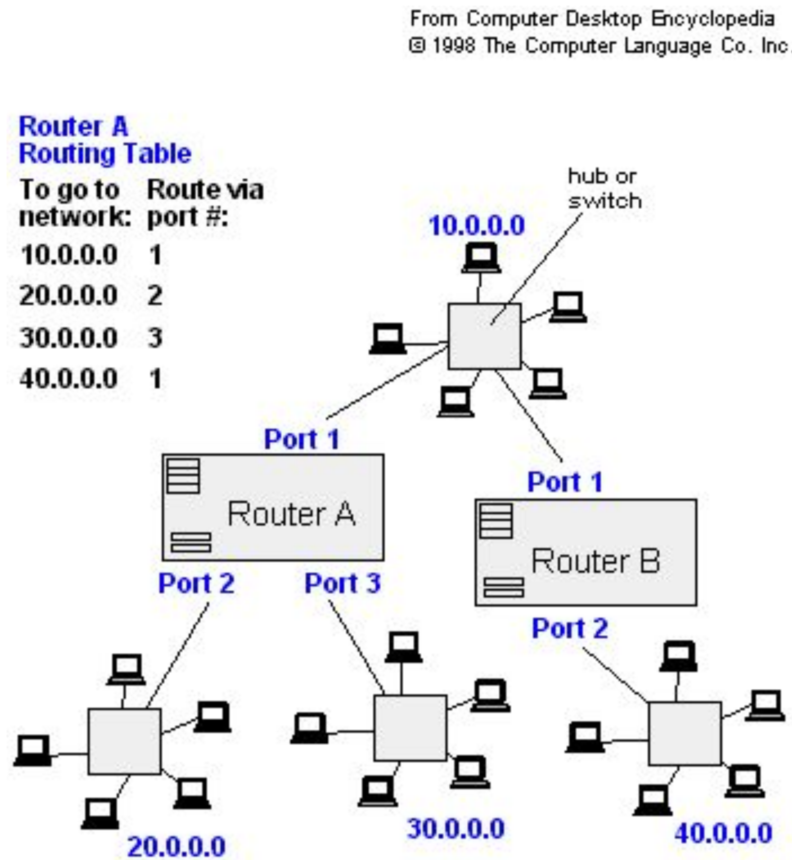
DIJKSTRA'S ALGORITHM - WHY USE IT?

- As mentioned, Dijkstra’s algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex *u* to every vertex using Dijkstra’s as it is to calculate the shortest path to some particular vertex *v*.
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra’s algorithm.

Applications of Dijkstra's Algorithm

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems





Applications of Dijkstra's Algorithm

- One particularly relevant this week: epidemiology
- Prof. Lauren Meyers (Biology Dept.) uses networks to model the spread of infectious diseases and design prevention and response strategies.
- Vertices represent individuals, and edges their possible contacts. It is useful to calculate how a particular individual is connected to others.
- Knowing the shortest path lengths to other individuals can be a relevant indicator of the potential of a particular individual to infect others.