Problem Solving:

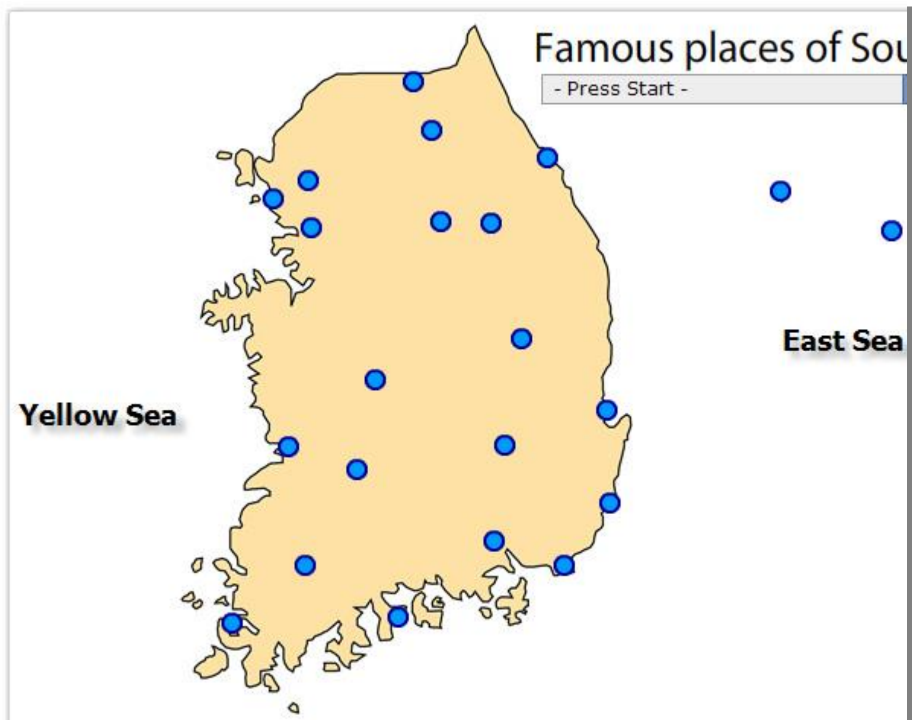
Graph, DFS, BFS

May 2019

Honguk Woo

Graph Usage

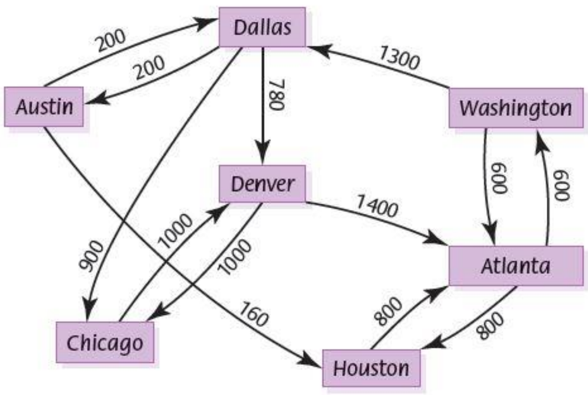
- I want to visit all the known famous places starting from Seoul ending in Seoul
- Knowledge: distances, costs
- Find the optimal (distance or cost) path



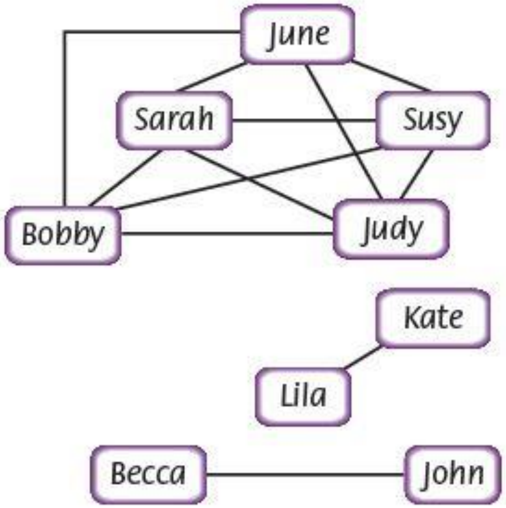
Graph Theory

- Many problems are mapped to graphs
  - traffic
  - VLSI circuits
  - social network
  - communication networks
  - web pages relationship

- Problems
  - how can a problem be represented as a graph?
  - how to solve a graph problem?



(b) Vertices: Cities
Edges: Direct flights



(a) Vertices: People
Edges: Siblings

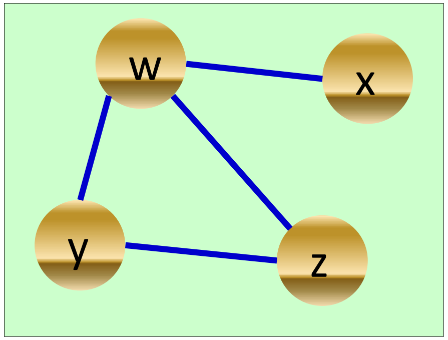
Graph Notation

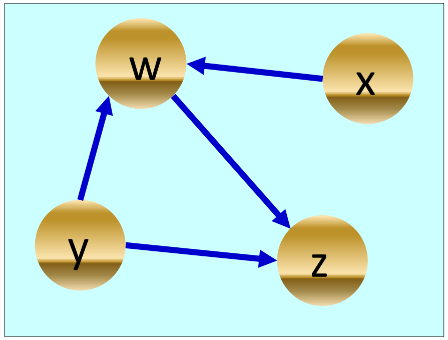
- A graph $G = (V, E)$
  - V is a set of vertices (nodes)
  - E is a set of edges
    - $E = (x, y)$ where $x, y \in V$
    - ordered or unordered pairs of vertices from V

- Examples
  - map
    - landmarks or cities are vertices
    - roads are edges
  - program analysis
    - a line of program statement is a vertices
    - the next line to be executed is connected through edge

Graphs

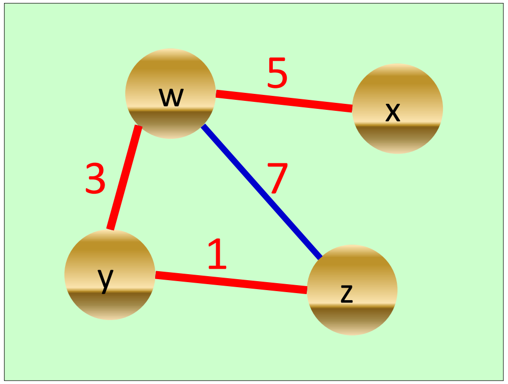
- A graph G = (V, E)
  - is undirected if edge (x, y) $\in$ E implies that (y, x) $\in$ E, too.
  - is directed if not

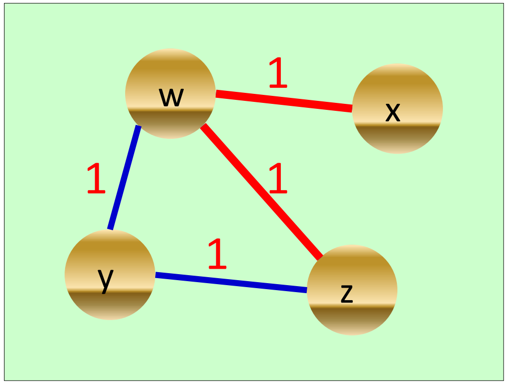




Graphs

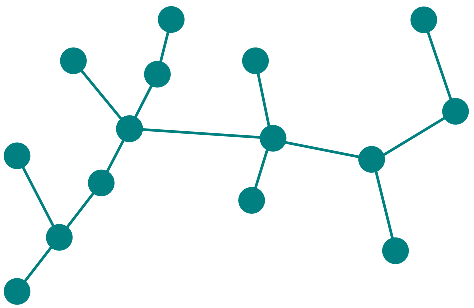
- weighted or unweighted



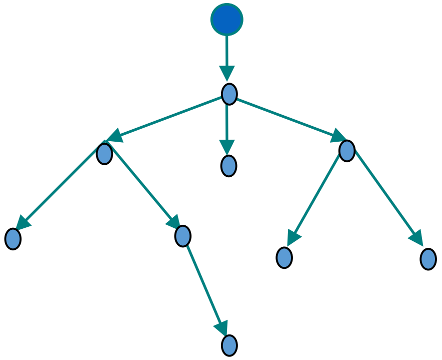


Graphs

- acyclic – a graph without any cycle
  - undirected (free tree)



  - directed (DAG)



Graph Representation : adjacency matrix

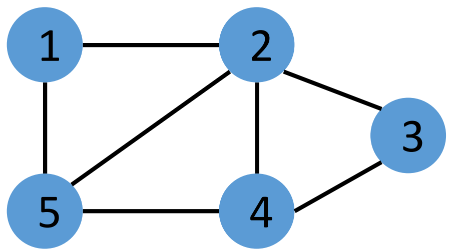
- G = (V, E), |V|=n and |E|=m

- **adjacency-matrix (인접행렬)**
  - n x n matrix M

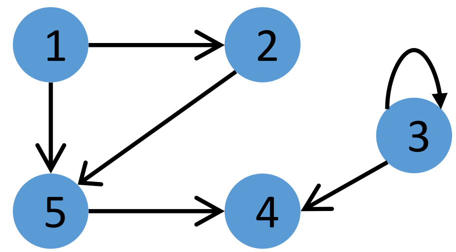
    $M[i, j] = 1$, if $(i, j) \in E$

    0, if $(i, j) \notin E$
  - good
    - easy to check if an edge (i, j) is in E
    - easy to add/remove edges
  - bad
    - space overhead if n >> m

Graph Representation : adjacency matrix Examples



	1	2	3	4	5
1	**0**	**1**	**0**	**0**	**1**
2	**1**	**0**	**1**	**1**	**1**
3	**0**	**1**	**0**	**1**	**0**
4	**0**	**1**	**1**	**0**	**1**
5	**1**	**1**	**0**	**1**	**0**

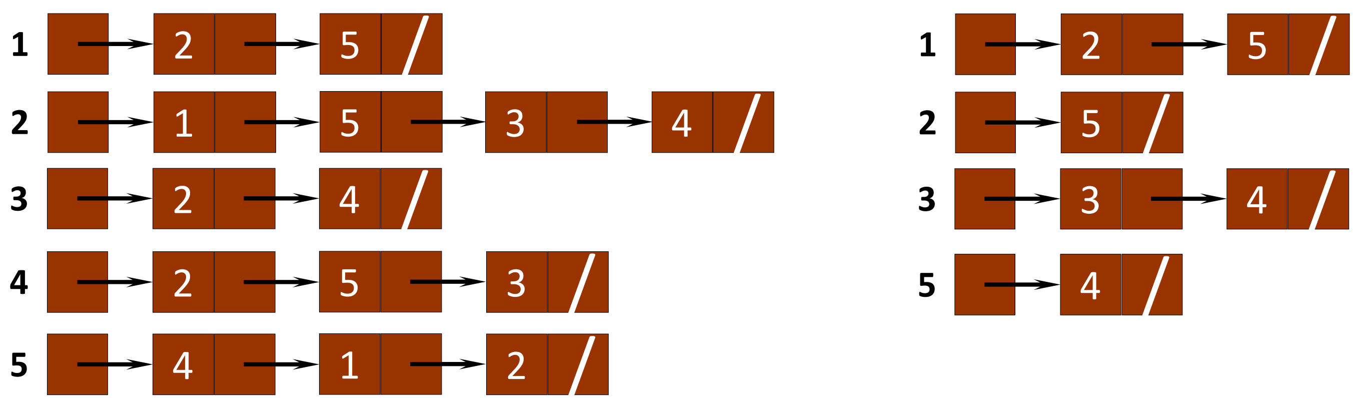


	1	2	3	4	5
1	**0**	**1**	**0**	**0**	**1**
2	**0**	**0**	**0**	**0**	**1**
3	**0**	**0**	**1**	**1**	**0**
4	**0**	**0**	**0**	**0**	**0**
5	**0**	**0**	**0**	**1**	**0**

- the City (Manhattan) – not so big area
  - 15 avenues and 200 streets
  - 3000 vertices and 6000 edges
  - 3000 X 3000 = 9,000,000 cells
- VLSI chip with 15 million transistors

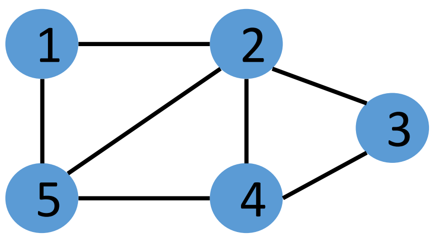
Graph Representation : adjacency list

- Adjacency list : an array of lists
  - array[i] denotes the list of vertices adjacent to the i^{th} vertex
  - Good : space efficient for sparse graphs
  - Bad : queries like whether there is an edge from vertex u to v are not efficient

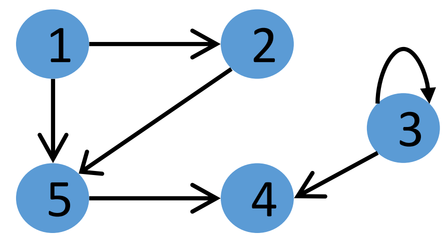


Graph Representation : mixed

- mixed version (adjacency lists in matrices)
  - use array instead of linked lists



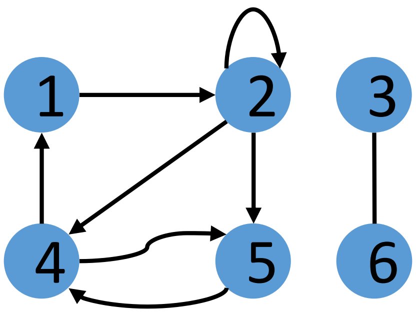
	1	2	3	4
1	2	5		
2	1	5	4	3
3	2	4		
4	2	5	3	
5	1	2	4	



	1	2
1	2	5
2	5	
3	3	4
5	4	

Terminologies - adjacency

- clear for undirected graph
- for directed graph
  - 2 is adjacent to 1 coz …
  - 1 is NOT adjacent to 2
  - make a formal definition
- if y is adjacent to x, we write $x \to y$
  - $1 \to 2$

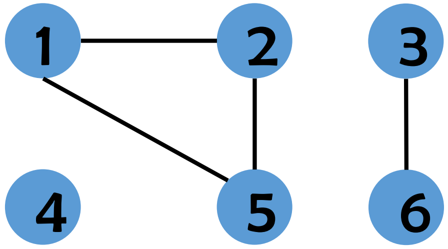


Terminologies - incident

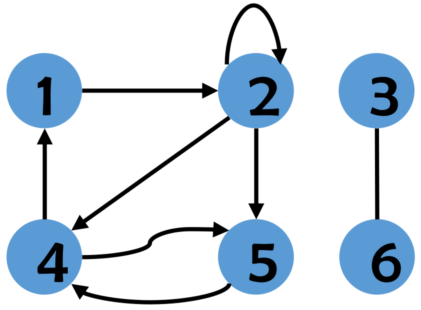
- directed ($x \rightarrow y$)
  - an edge (x, y) is incident from (or leaves) vertex x
  - and is incident to (or enters) vertex y

- undirected
  - an edge (x, y) is incident on vertices x and y
  - e.g., the edges incident on vertex 2: (1, 2), (2, 5)

Terminologies - degree



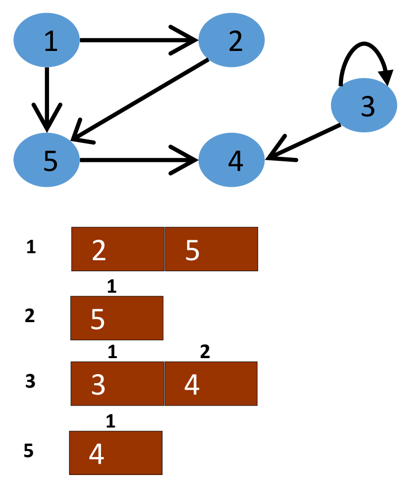
- **Degree** of a vertex
  - undirected
    - the number of edges incident on it.
      ex. vertex 2 in the graph has degree 2.
    - A vertex whose degree is 0,
      i.e., vertex 4 in the graph, is *isolated*.
  - directed
    - **out-degree** of a vertex : the number of edges leaving it
    - **in-degree** of a vertex : the number of edges entering it
    - degree of a vertex : its in-degree + out-degree
    - vertex 2 in the right graph
      - in-degree = 2
      - out-degree = 3
      - degree = 2+3 = 5



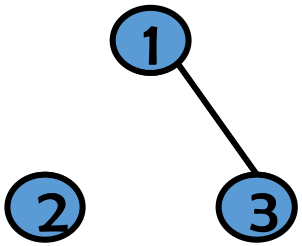
“Adjacency lists in matrices” structure

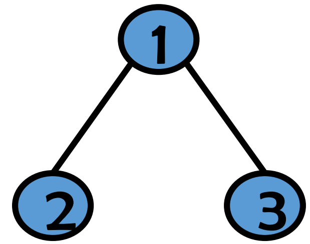
```
#define MAXV            100            /* maximum number of vertices */
#define MAXDEGREE       50             /* maximum vertex outdegree */

typedef struct {
        int edges[MAXV+1][MAXDEGREE];  /* adjacency info */
        int degree[MAXV+1];            /* outdegree of each vertex */
        int nvertices;                 /* number of vertices in graph */
        int nedges;                    /* number of edges in graph */
} graph;
```

- only if you know MAXDEGREE; otherwise, MAXV x MAXV



adding an edge - insert_edge(g, 1, 2, false)





	0	1	...		49
1					
2					
...					
100					

edges[101][50]

1	0
2	0
3	0
100	0

degree[101]

adding an edge : e.g., insert_edge(g, 1, 2, false)

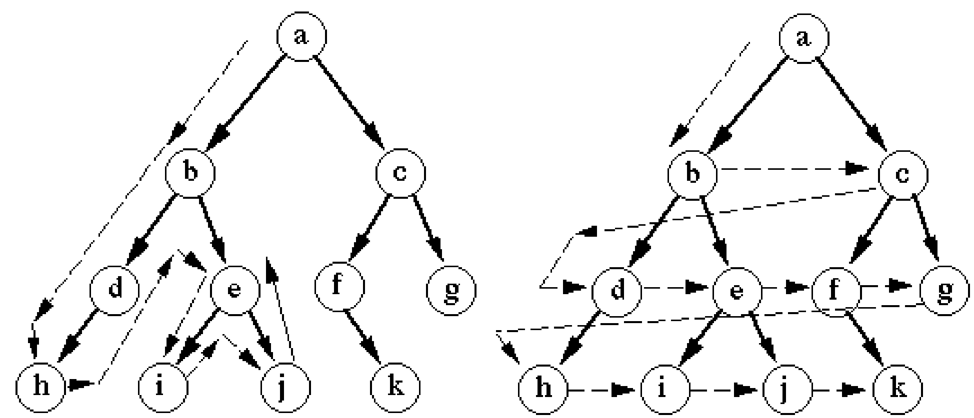
```
insert_edge(graph *g, int x, int y, bool directed)
{
        if (g->degree[x] > MAXDEGREE)
            printf("Warning: insertion(%d,%d) exceeds max degree\n",x,y);

        g->edges[x][g->degree[x]] = y;
        g->degree[x] ++;

        if (directed == FALSE)
                insert_edge(g,y,x,TRUE);
        else
                g->nedges ++;
```

Printing a graph

```
print_graph(graph *g)
{
        int i,j;                        /* counters */

        for (i=1; i<=g->nvertices; i++) {
                printf("%d: ",i);
                for (j=0; j<g->degree[i]; j++)
                        printf(" %d",g->edges[i][j]);
                printf("\n");
        }
}
```

Graph Traversal

- To visit vertices
  - all of them in a graph for completeness
  - exactly once for efficiency
- Two algorithms
  - Breadth First Search (BFS)
  - Depth First Search (DFS)
- Fundamental idea
  - mark the vertices visited before and don't explore again

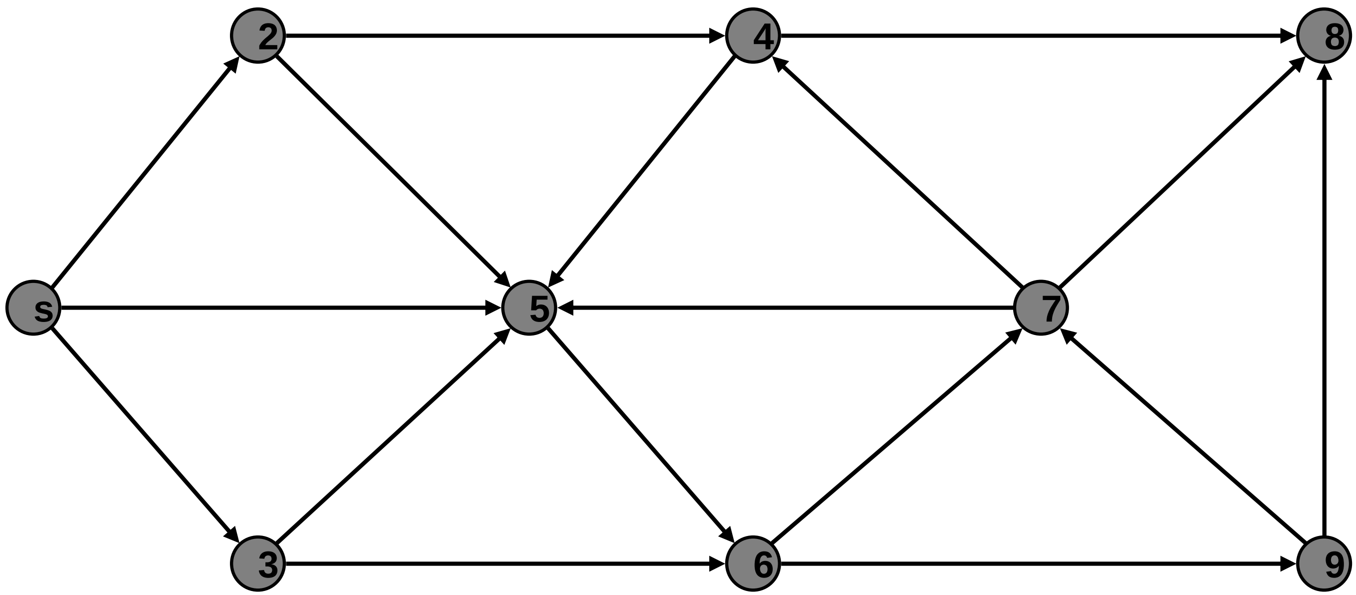


Depth-first search

Breadth-first search

Visit from source (root); once a vertex is discovered, it is placed on a queue (FIFO)

BFS



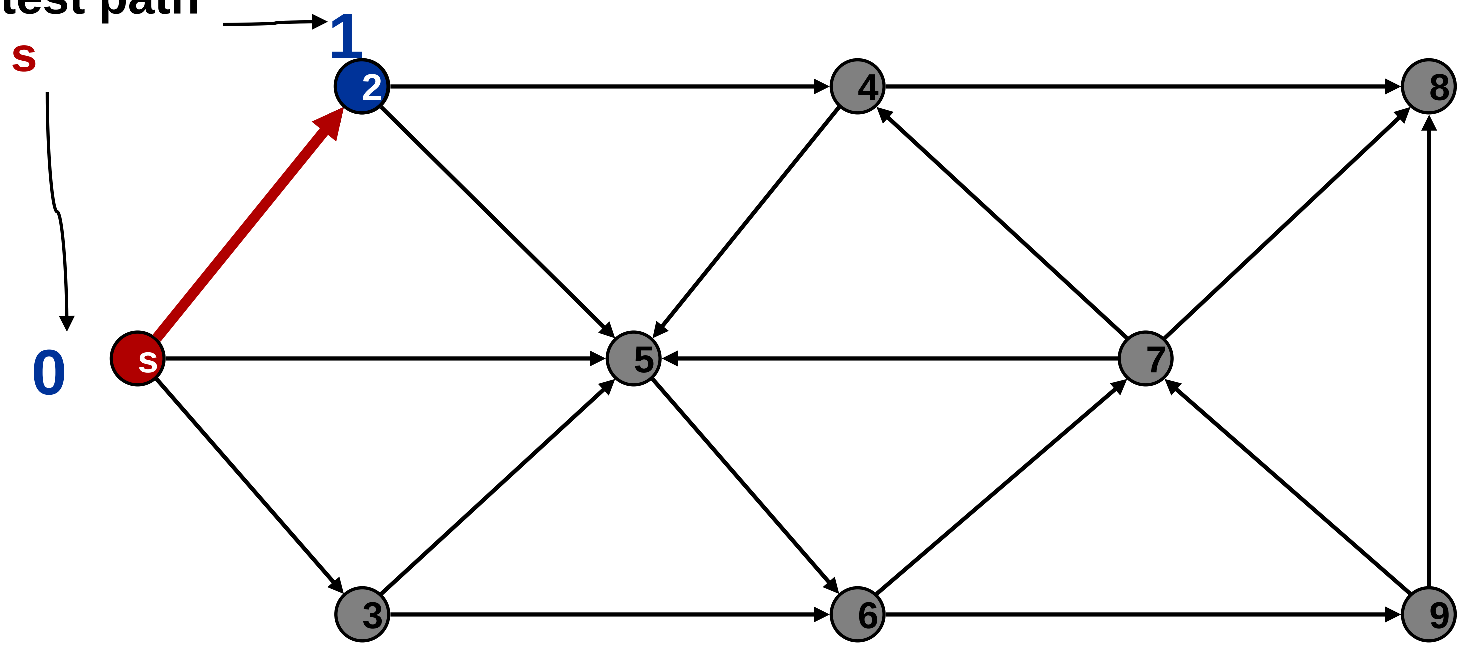
		0	1	…	49
s		2	3	5	
2		4	5		
3		5	6		
100					

edges[101][50]

1	3
2	2
3	2
100	0

degree[101]

Shortest path from s



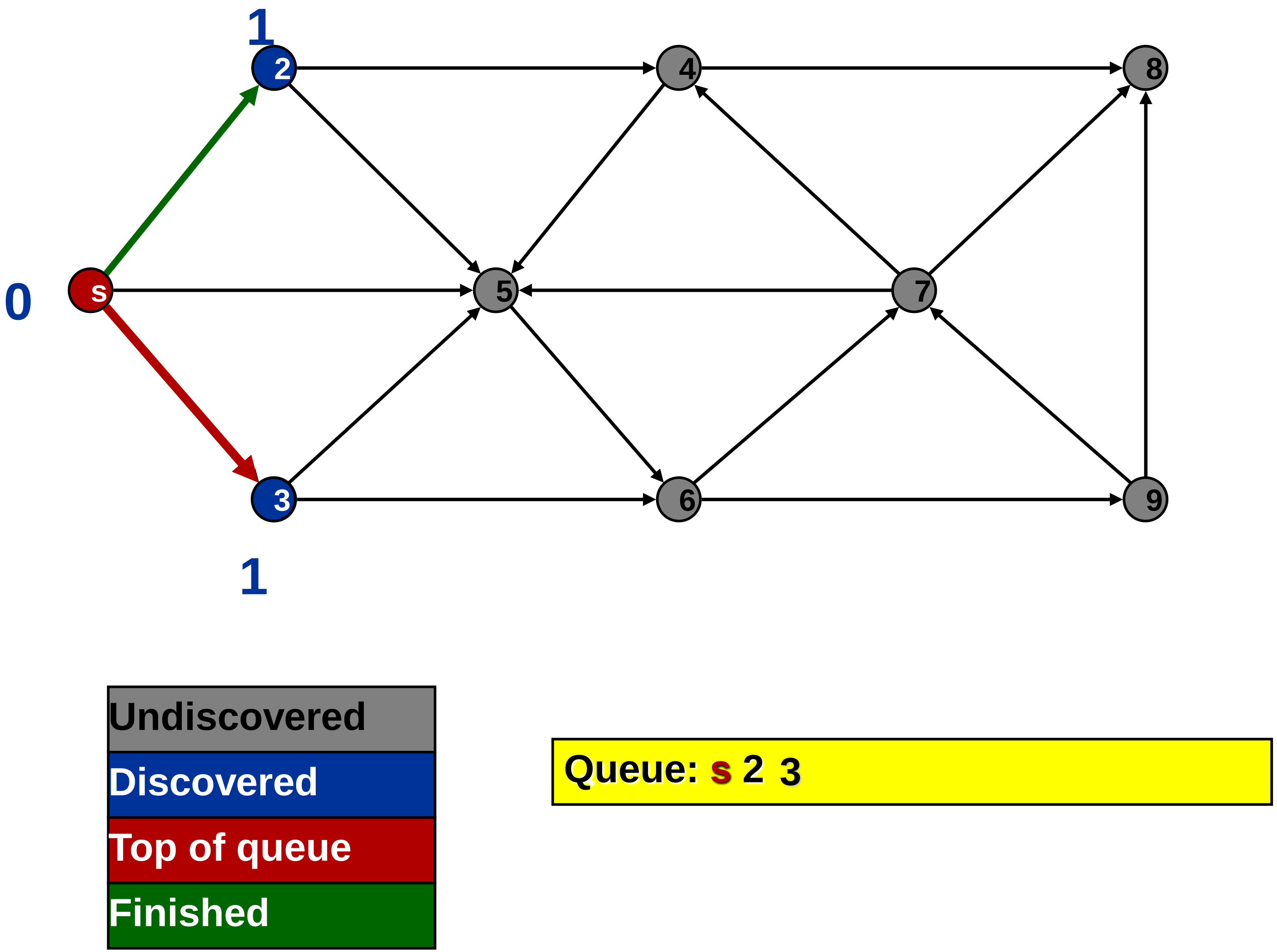
Undiscovered

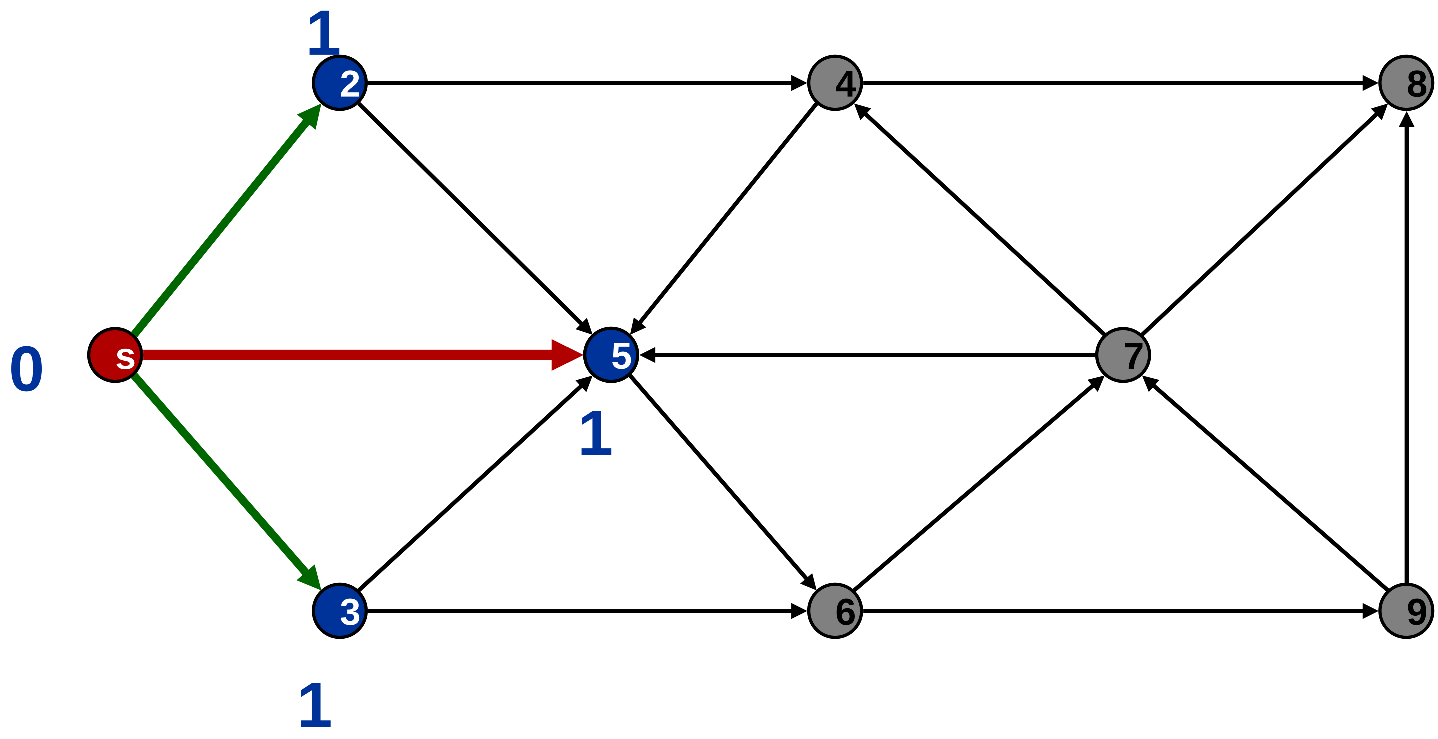
Discovered

Top of queue

Finished

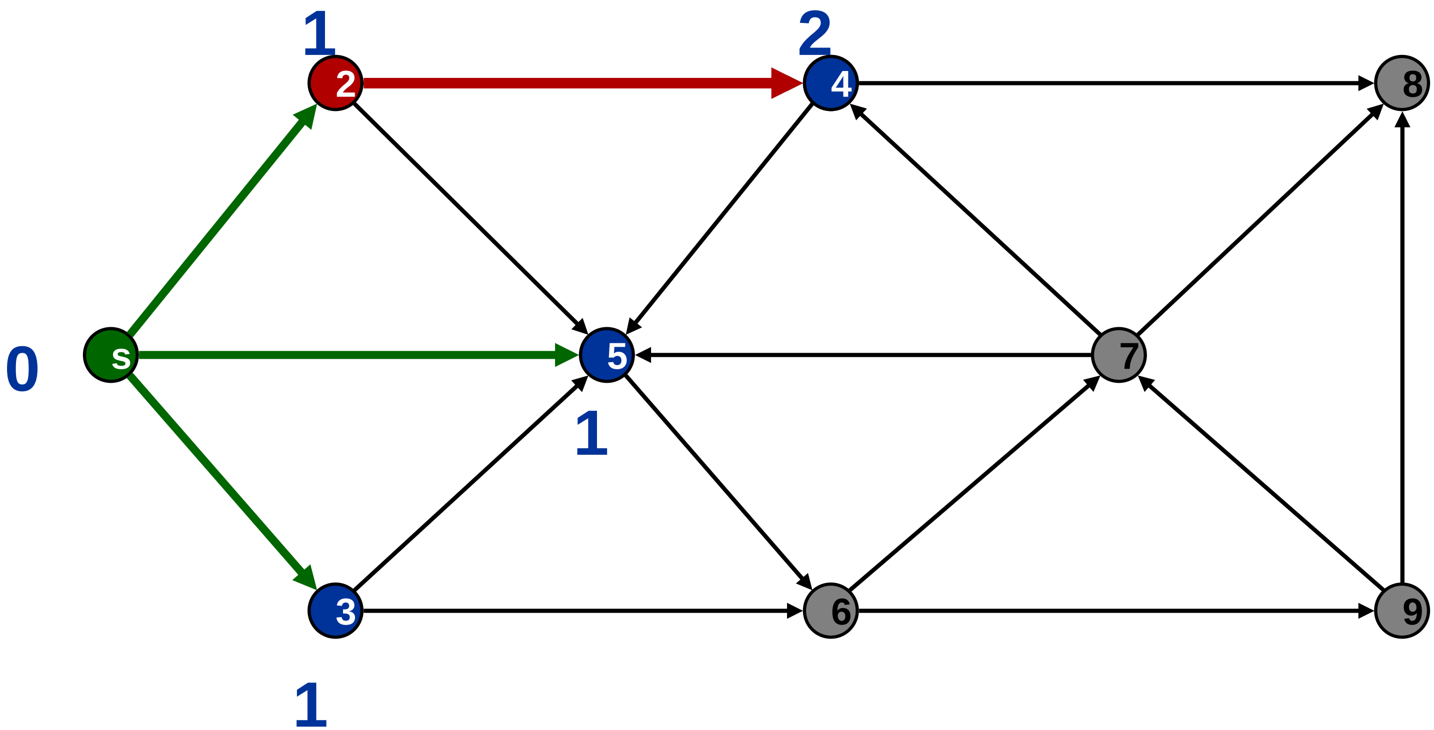
Queue: s 2


1
2
4
8
0
s
5
7
3
6
9
1
Undiscovered
Discovered
Top of queue
Finished
Queue: s 2 3



Undiscovered
Discovered
Top of queue
Finished

Queue: s 2 3 5



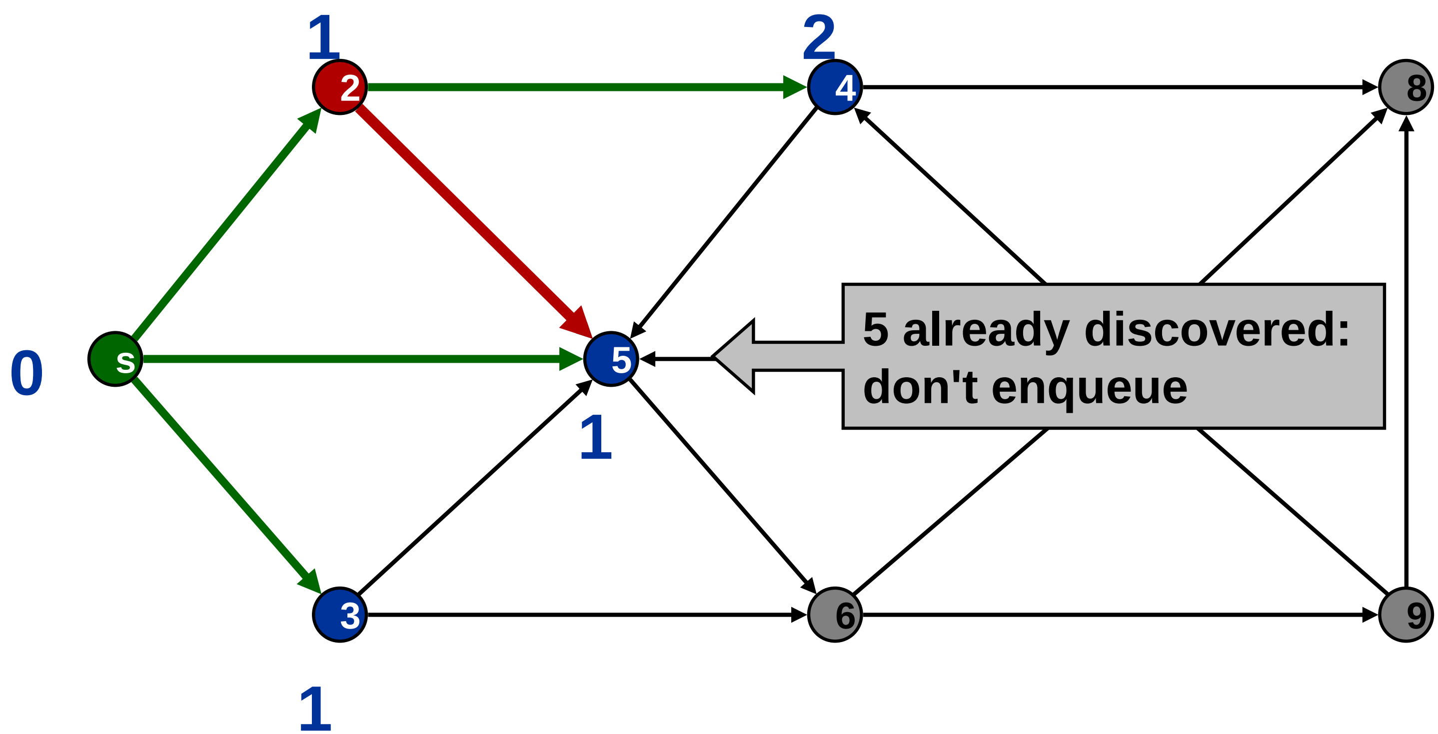
Undiscovered

Discovered

Top of queue

Finished

Queue: 2 3 5 4



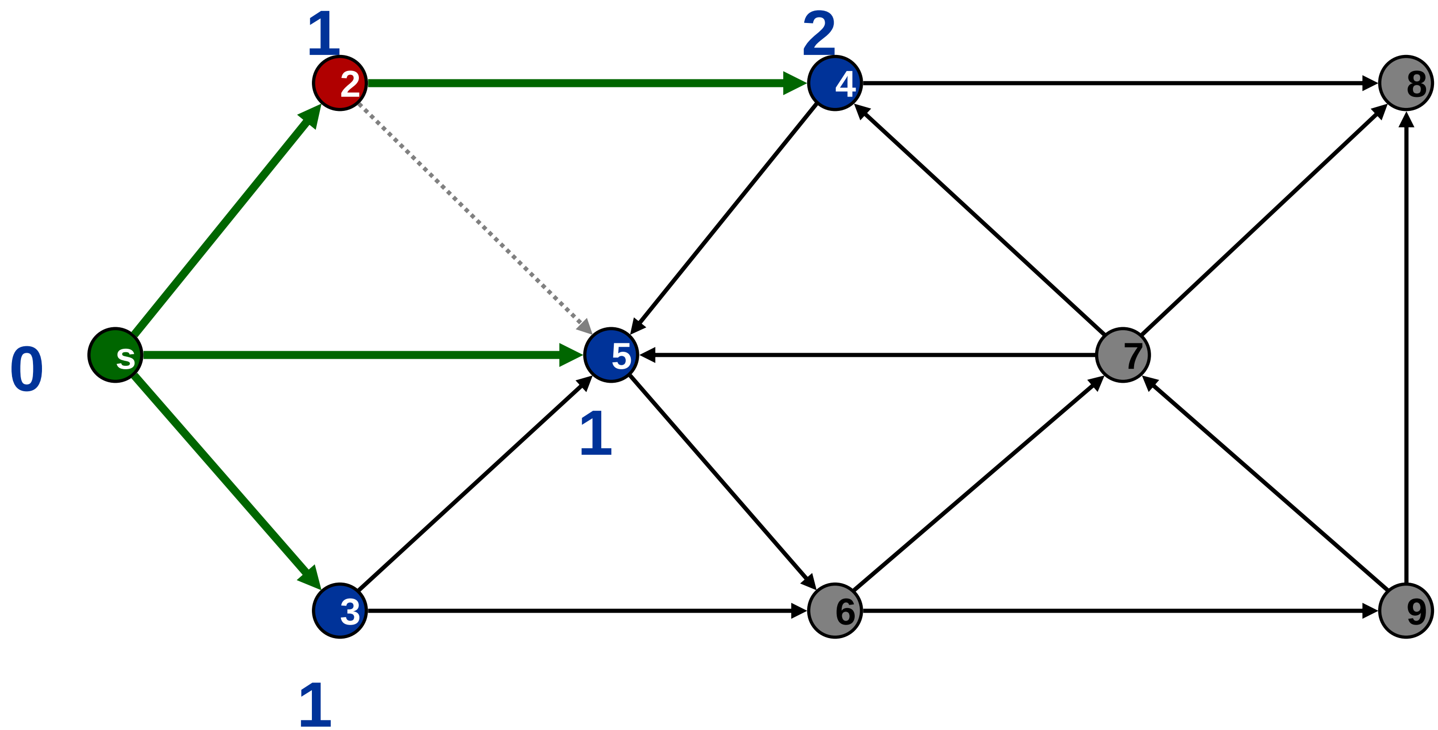
Undiscovered

Discovered

Top of queue

Finished

Queue: 2 3 5 4



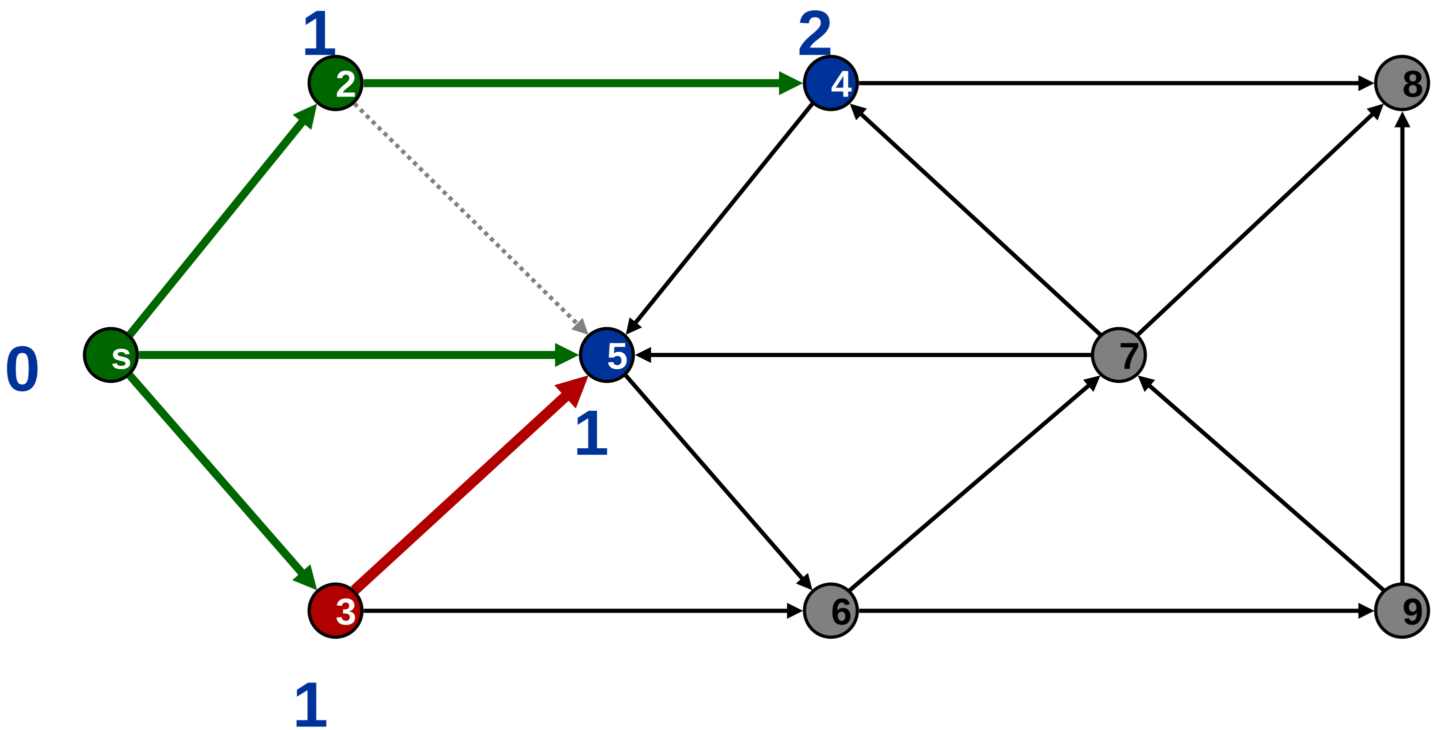
Undiscovered

Discovered

Top of queue

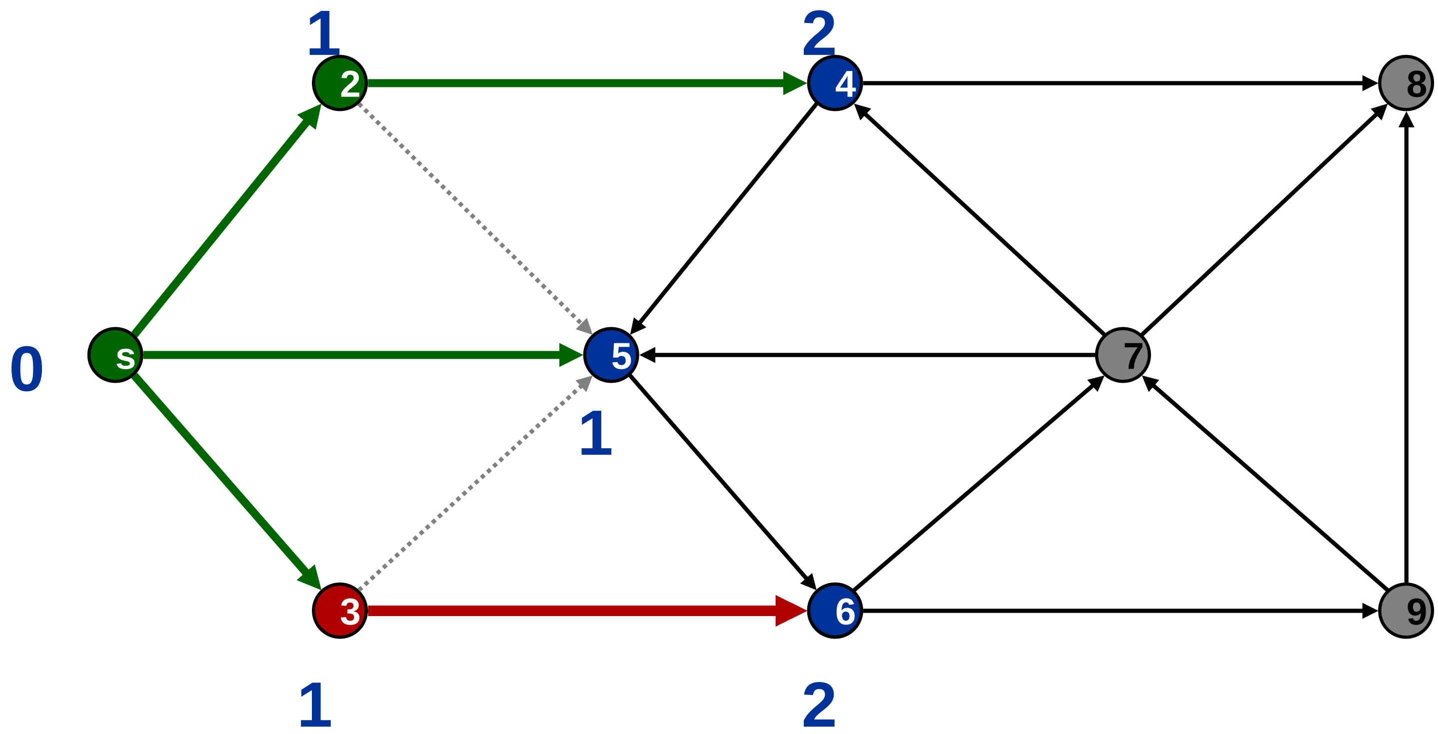
Finished

Queue: 2 3 5 4



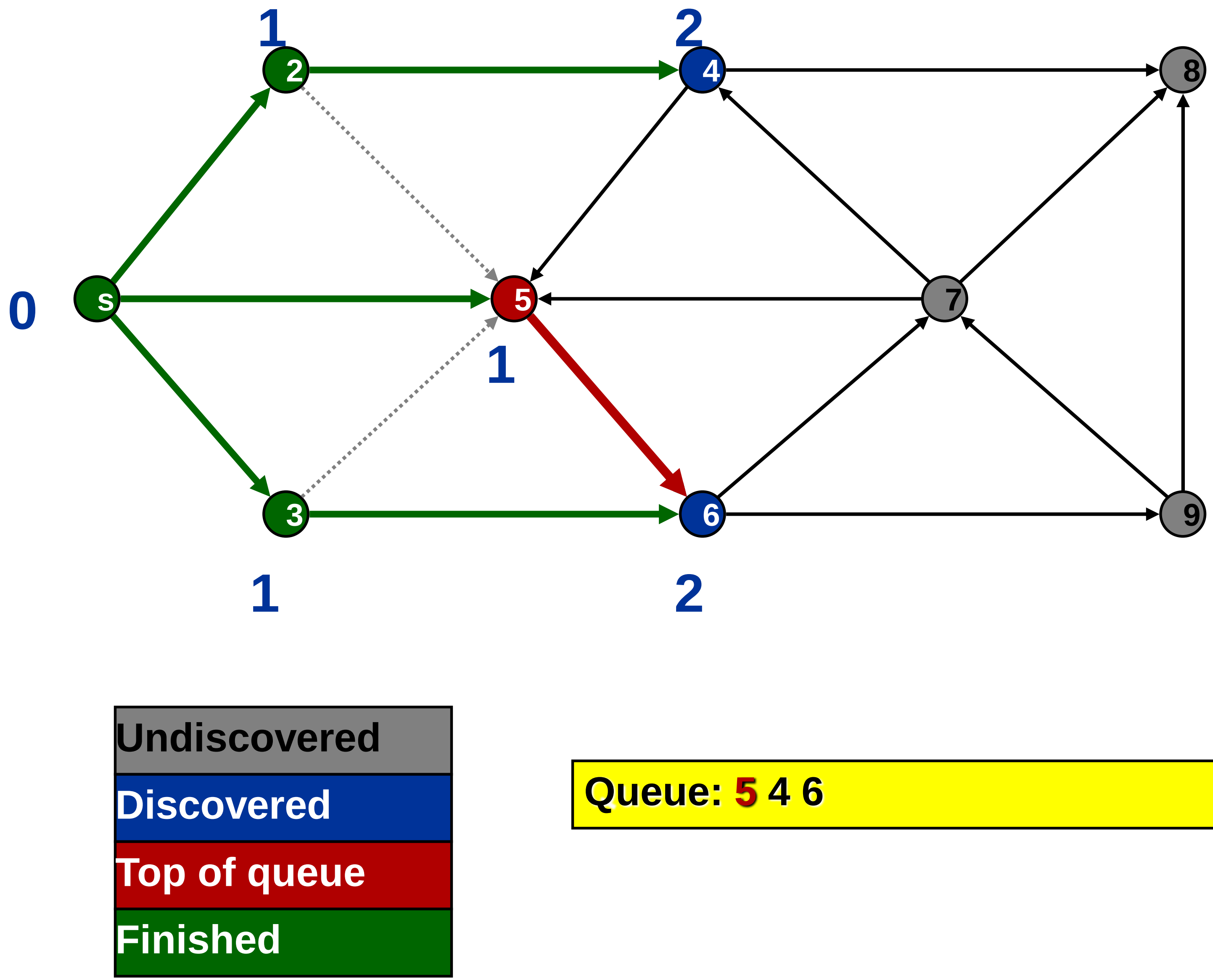
Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4



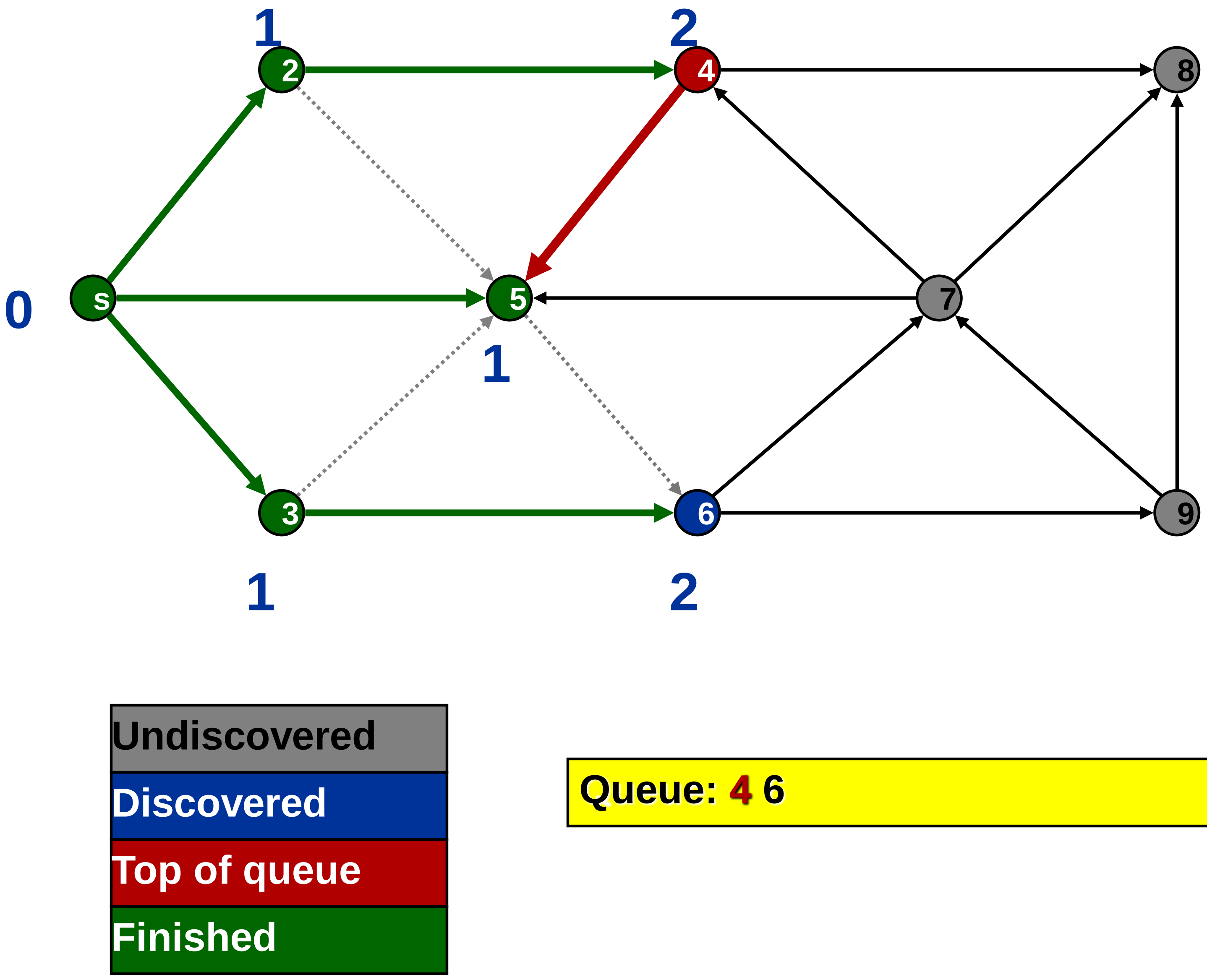
Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4 6



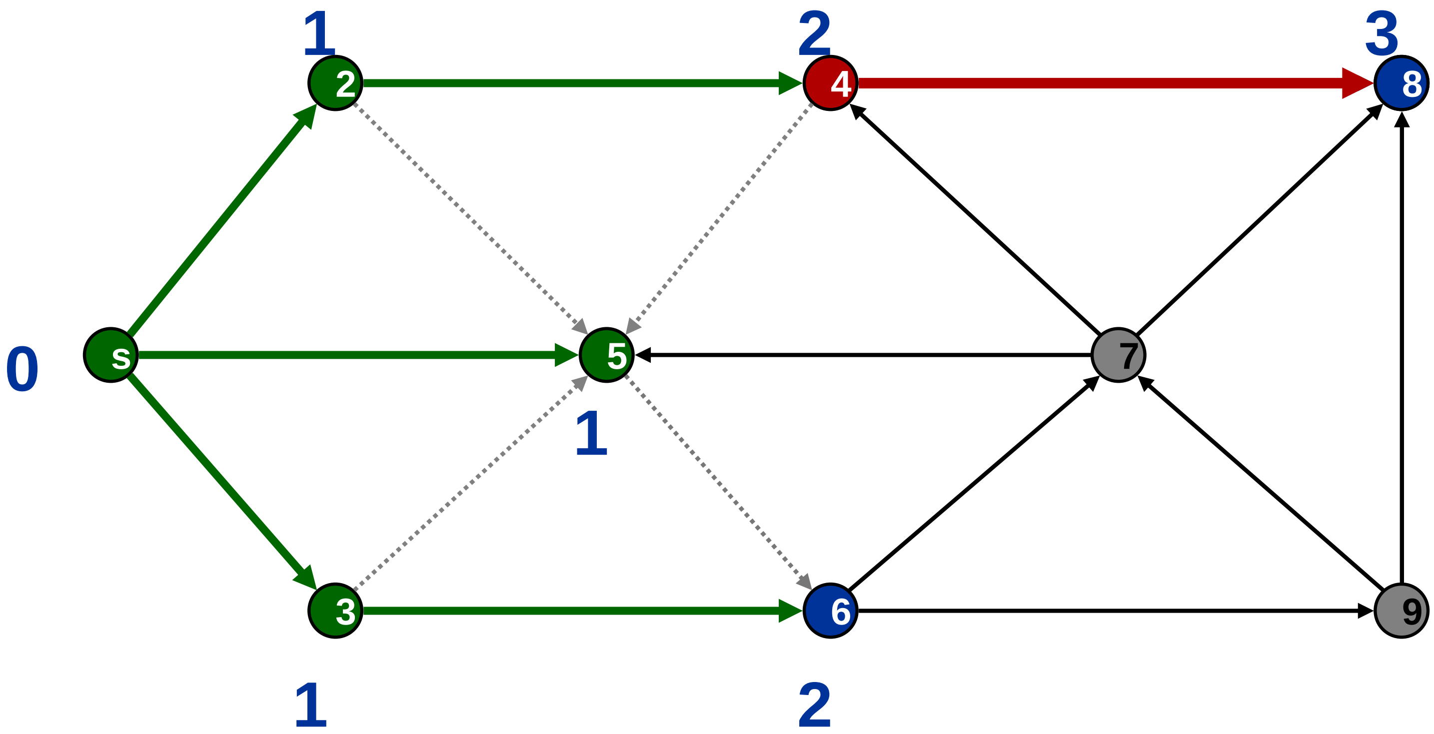
1
2
0
s
2
4
8
5
7
1
3
6
9
1
2
Undiscovered
Discovered
Top of queue
Finished

Queue: 5 4 6



1
2
2
4
8
0
s
5
7
1
3
6
9
1
2
Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6



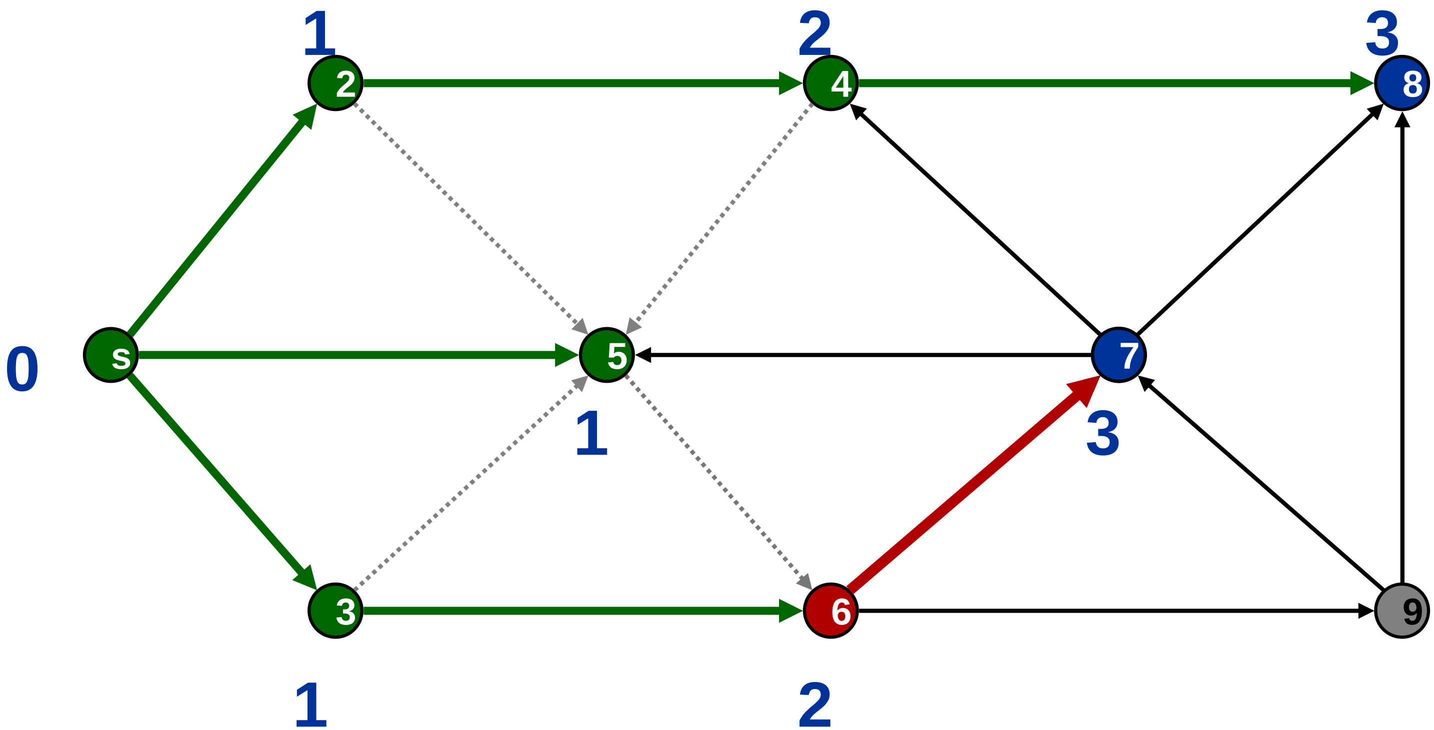
Undiscovered

Discovered

Top of queue

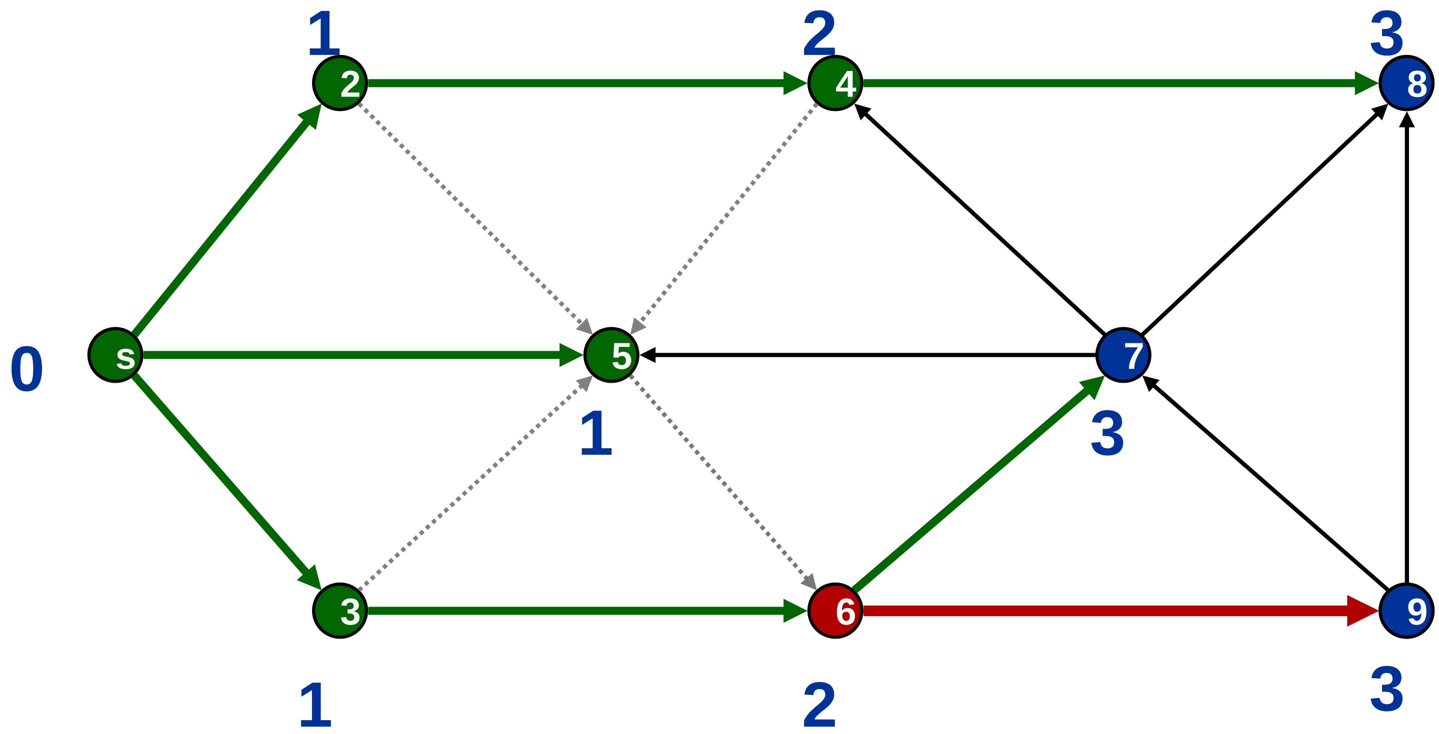
Finished

Queue: 4 6 8



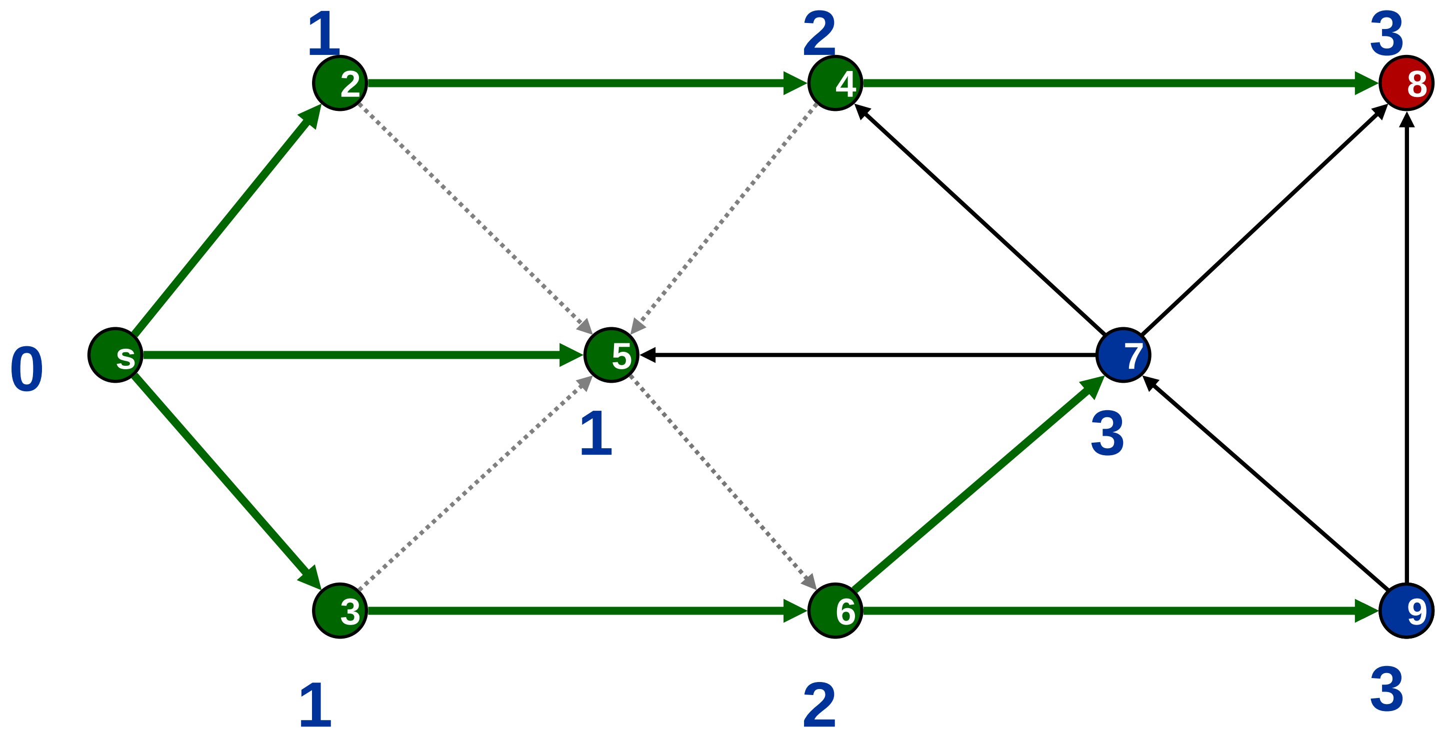
Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7 9



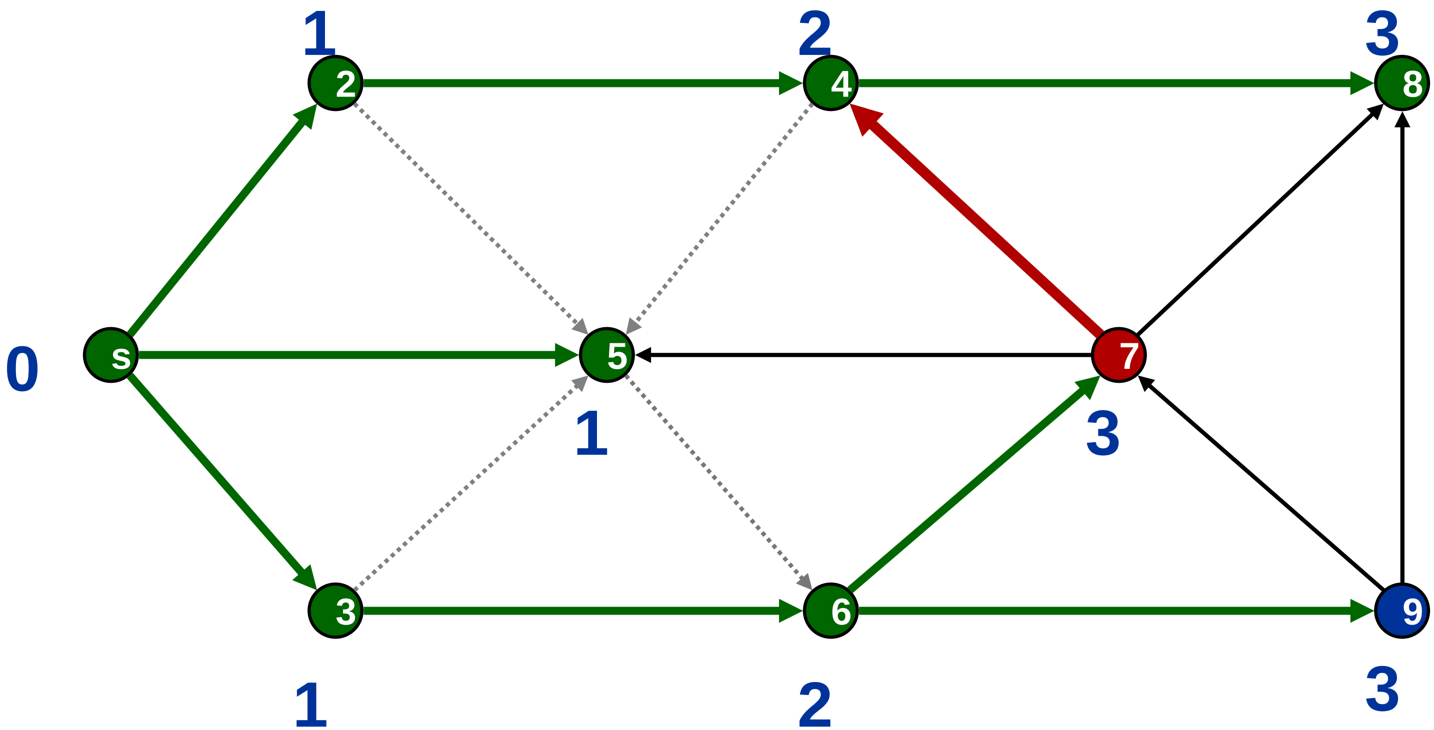
Undiscovered

Discovered

Top of queue

Finished

Queue: 8 7 9



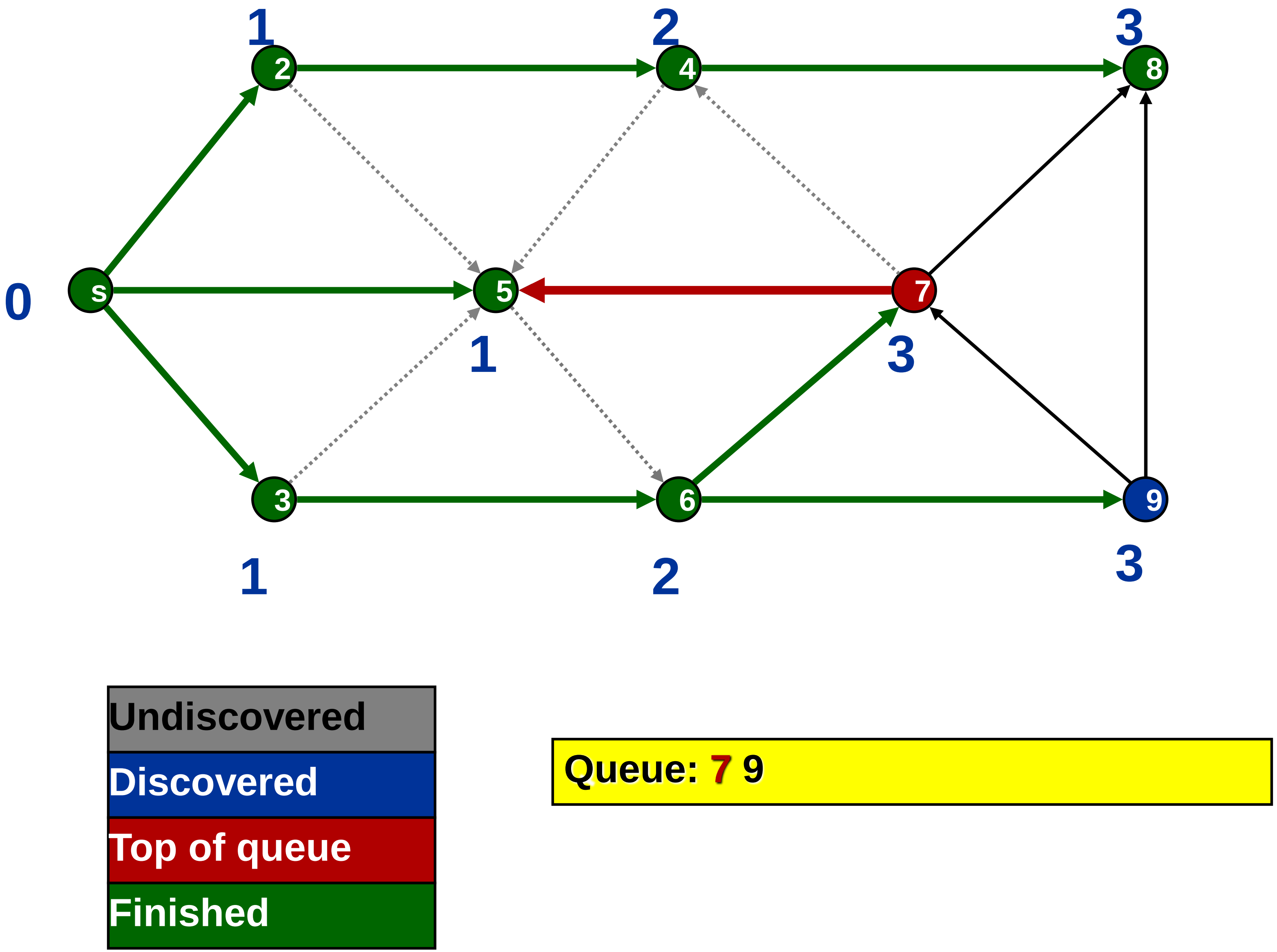
Undiscovered

Discovered

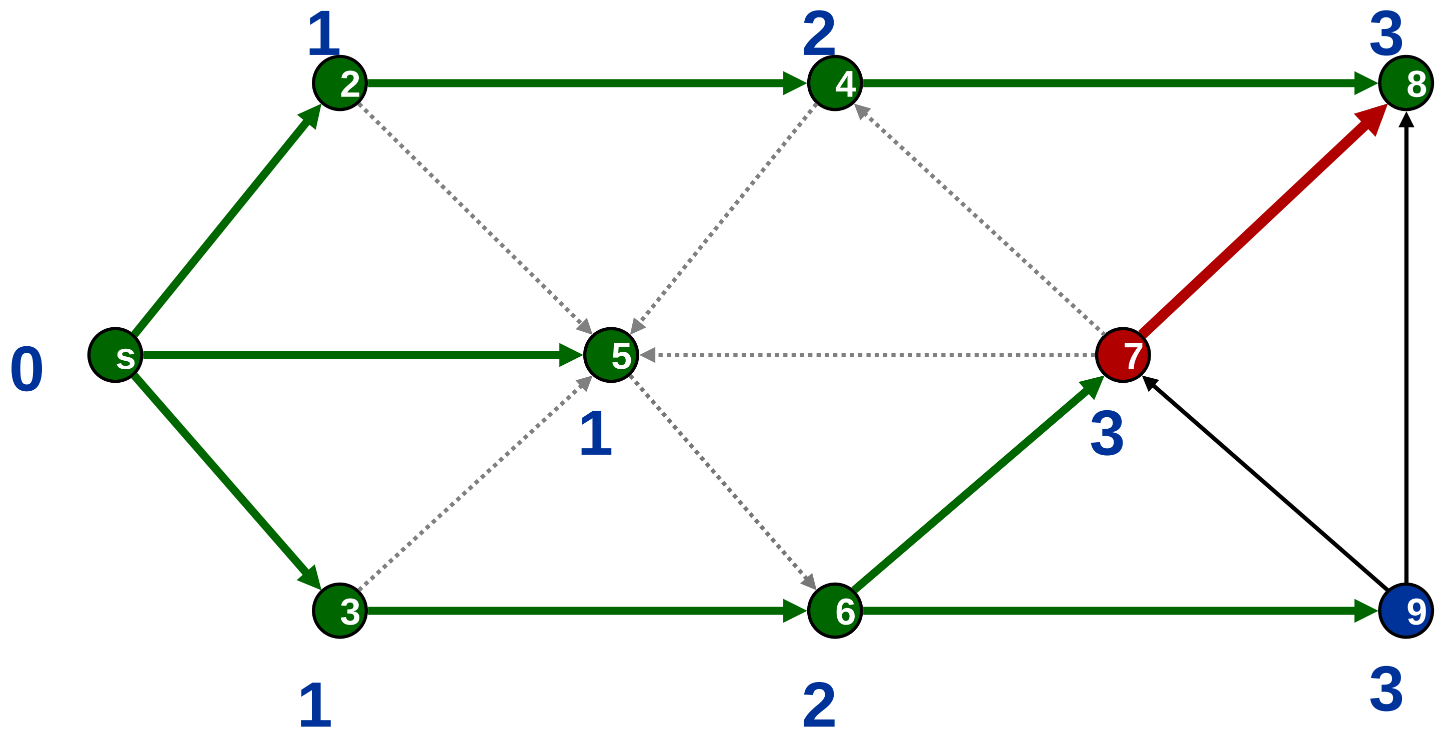
Top of queue

Finished

Queue: 7 9



0
1
2
3
s
2
4
8
5
7
1
3
3
6
9
1
2
3
Undiscovered
Discovered
Top of queue
Finished
Queue: 7 9



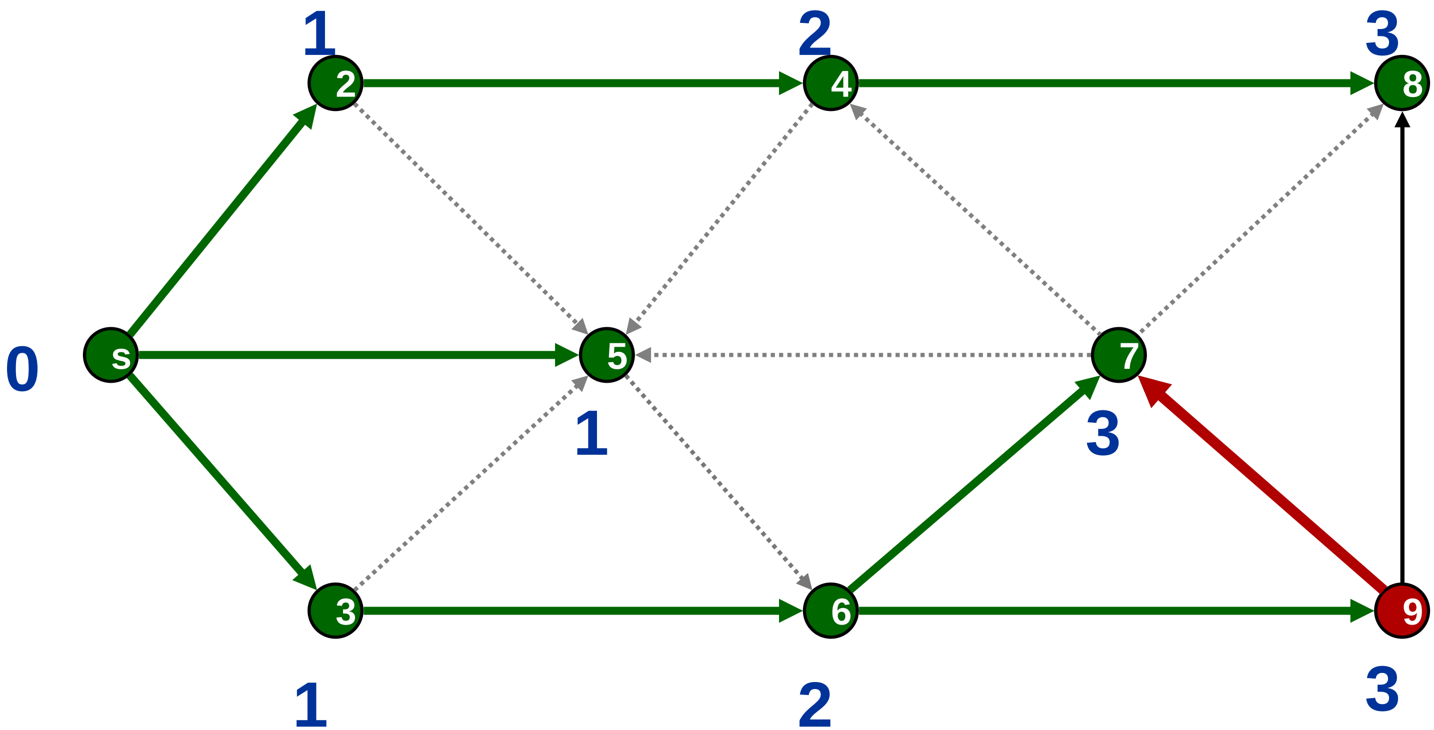
Undiscovered

Discovered

Top of queue

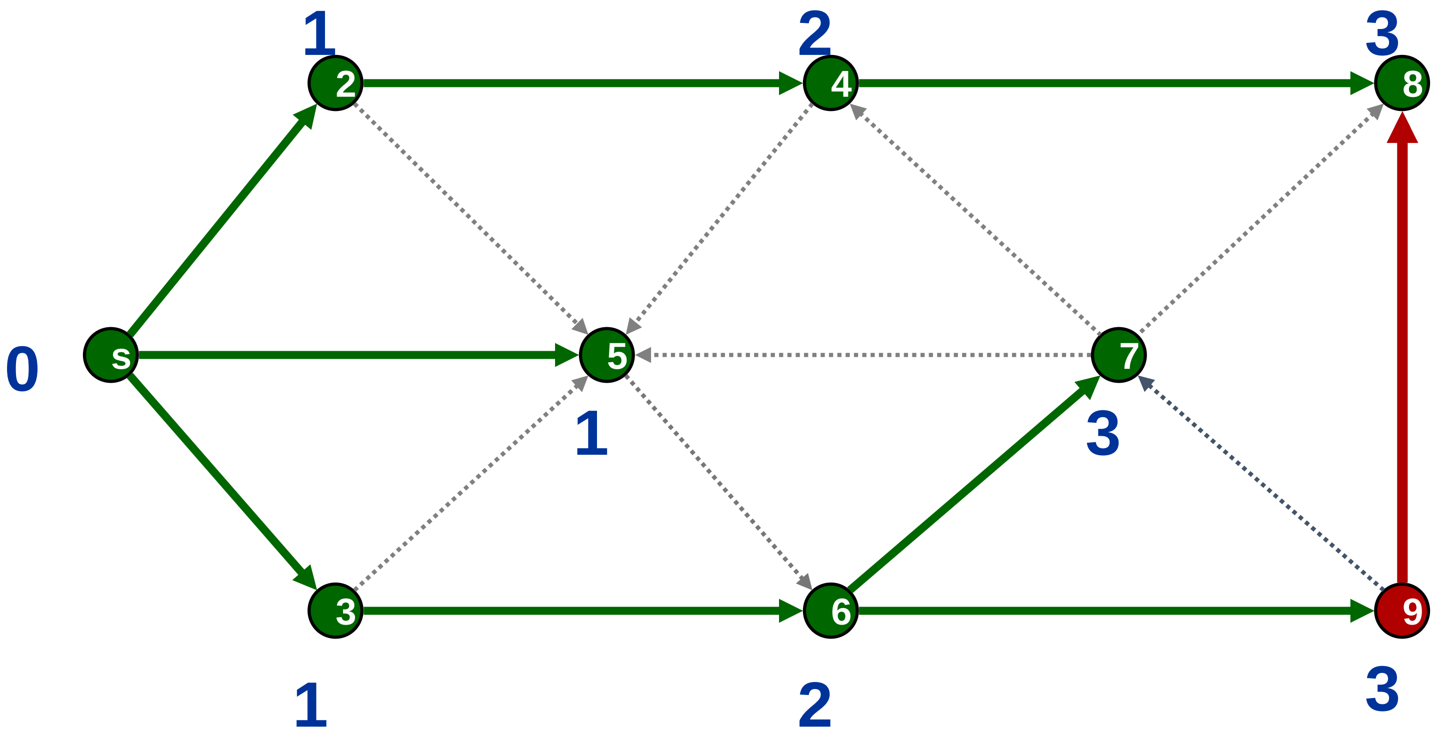
Finished

Queue: 7 9



Undiscovered
Discovered
Top of queue
Finished

Queue: 9



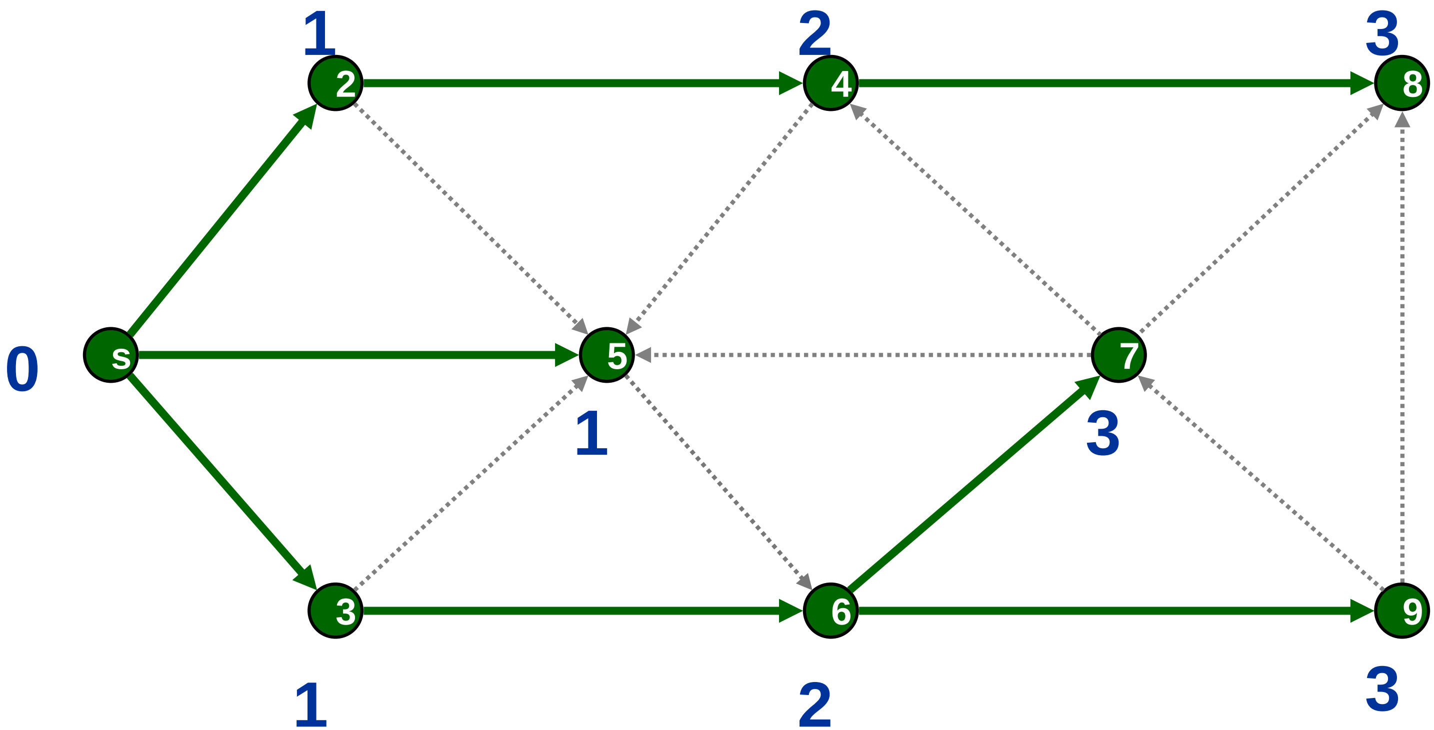
Undiscovered

Discovered

Top of queue

Finished

Queue: 9



Undiscovered

Discovered

Top of queue

Finished

Queue:

➔ Since Queue is empty, STOP!

BFS Algorithm

- Discovered vertex : visited before

```
bfs(graph *g, int start)
{
        queue q;                        /* queue of vertices to visit */
        int v;                          /* current vertex */
        int i;                          /* counter */

        init_queue(&q);
        enqueue(&q,start);
        discovered[start] = TRUE;

        while (empty(&q) == FALSE) {
                v = dequeue(&q);
                process_vertex(v);
                processed[v] = TRUE;
                for (i=0; i<g->degree[v]; i++)
                    if (valid_edge(g->edges[v][i]) == TRUE) {
                        if (discovered[g->edges[v][i]] == FALSE) {
                                enqueue(&q,g->edges[v][i]);
                                discovered[g->edges[v][i]] = TRUE;
                                parent[g->edges[v][i]] = v;
                        }
                        if (processed[g->edges[v][i]] == FALSE)
                                process_edge(v,g->edges[v][i]);
                    }
        }
}
```

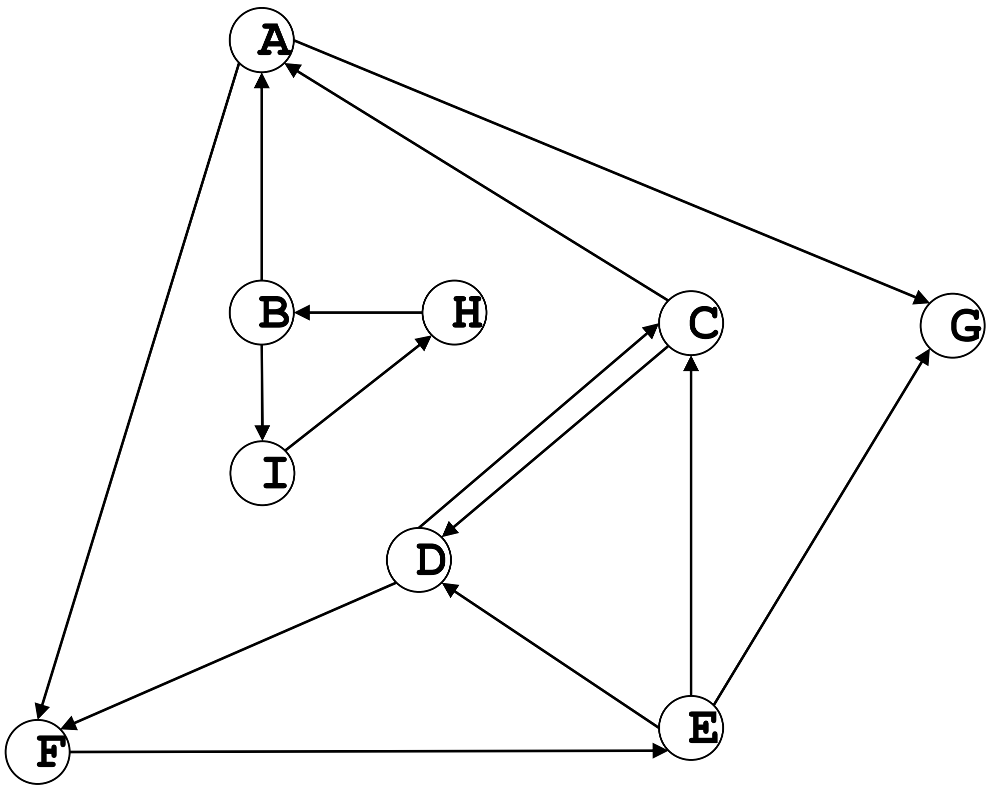
BFS for search

```
Breadth First Search(startVertex, endVertex)

Set found to FALSE
Enque(myQueue, startVertex)
WHILE (NOT IsEmpty(myQueue) AND NOT found)
       Deque(myQueue, tempVertex)
       IF (tempVertex equals endVertex)
              Write endVertex
              Set found to TRUE
       ELSE IF (tempVertex not visited)
              Write tempVertex
              Enque all unvisited vertexes adjacent with tempVertex
              Mark tempVertex as visited
IF (found)
       Write "Path has been printed"
ELSE
       Write "Path does not exist"
```

DFS

- Same idea as backtracking
  - go as deep as you can, backing up as soon as there is no unexplored possibility
  - Recursive algorithms; stack is an ideal candidate instead of queue

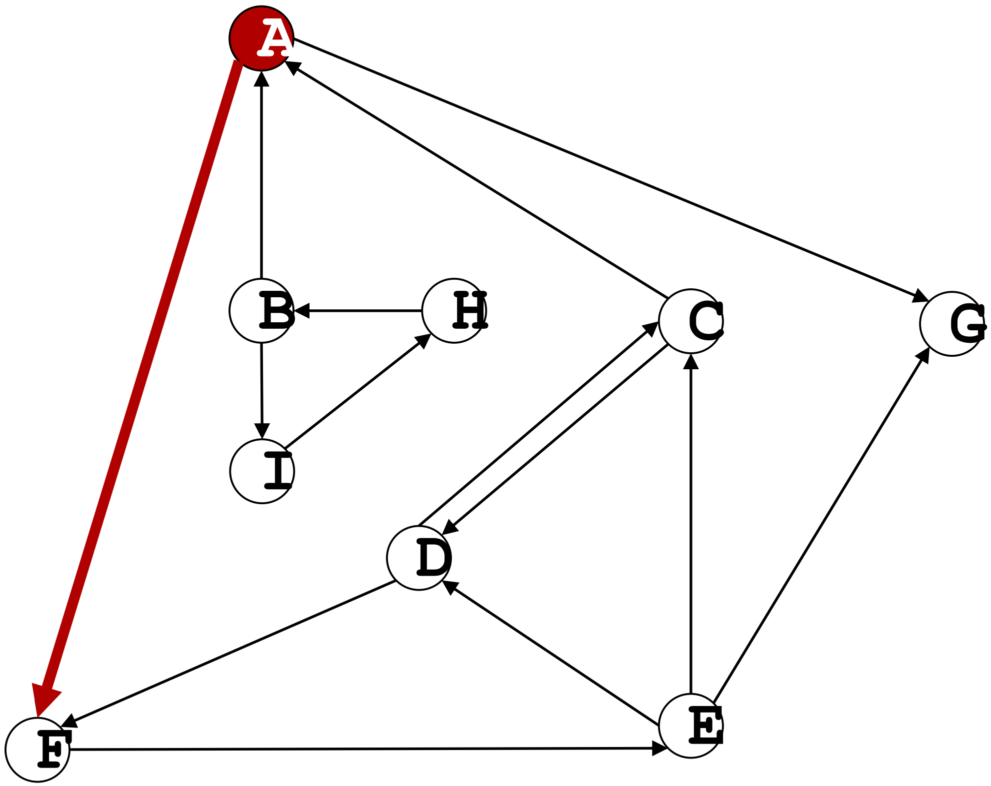
```
dfs(graph *g, int v)
{
        int i;                          /* counter */
        int y;                          /* successor vertex */

        if (finished) return;           /* allow for search termination */

        discovered[v] = TRUE;
        process_vertex(v);

        for (i=0; i<g->degree[v]; i++) {
                y = g->edges[v][i];
                if (valid_edge(g->edges[v][i]) == TRUE) {
                        if (discovered[y] == FALSE) {
                                parent[y] = v;
                                dfs(g,y);
                        } else
                                if (processed[y] == FALSE)
                                        process_edge(v,y);
                }
                if (finished) return;
        }

        processed[v] = TRUE;
}
```



Adjacency Lists

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

assume "left child first"



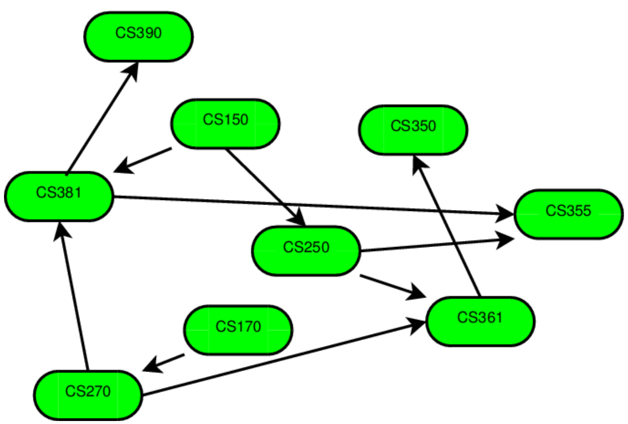
Function call stack:

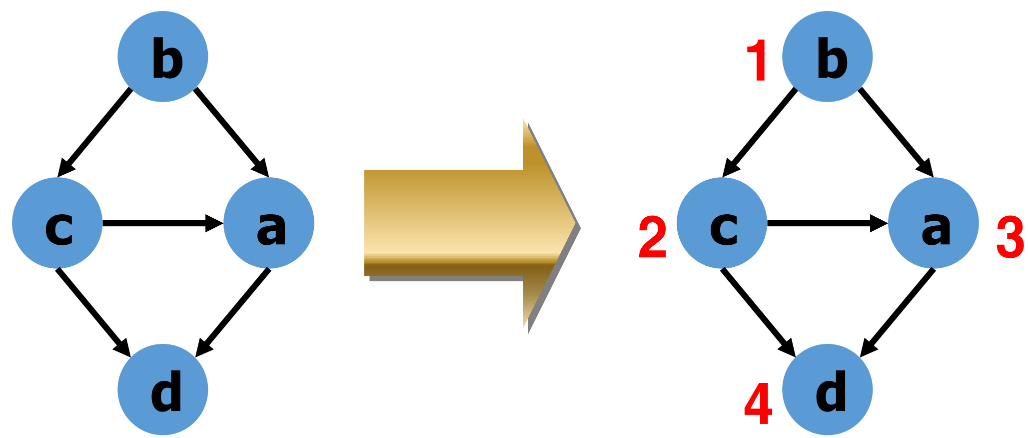
DFS with stack

```
Depth First Search(startVertex, endVertex)

Set found to FALSE
Push(myStack, startVertex)
WHILE (NOT IsEmpty(myStack) AND NOT found)
      Pop(myStack, tempVertex)
      IF (tempVertex equals endVertex)
            Write endVertex
            Set found to TRUE
      ELSE IF (tempVertex not visited)
            Write tempVertex
            Push all unvisited vertexes adjacent with tempVertex
            Mark tempVertex as visited
IF (found)
      Write "Path has been printed"
ELSE
      Write "Path does not exist")
```

Topological Sort

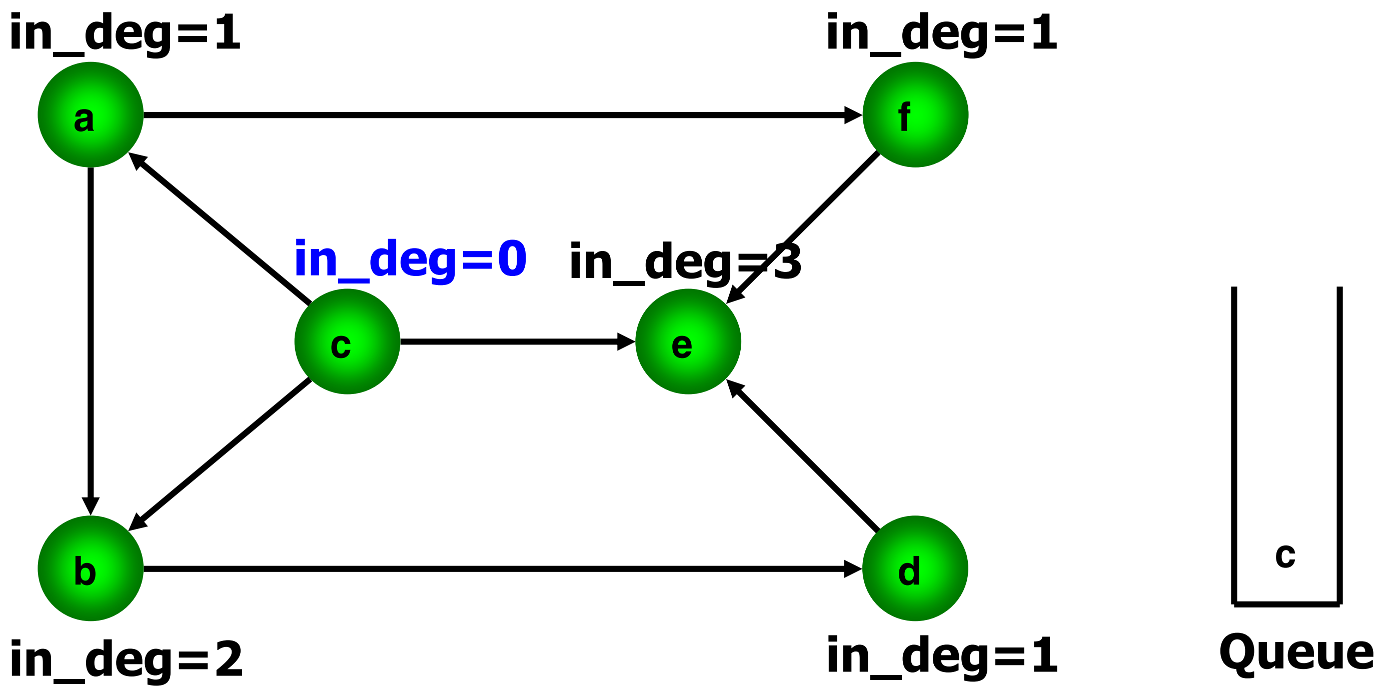
- A topological sort of a DAG (directed acyclic graph) **G** is a linear ordering of all its vertices such that if **G** contains a link (u,v), then node u appears before node v in the ordering
- Examples with precedence constraints
  - Library build, Pre-requisites in curriculum





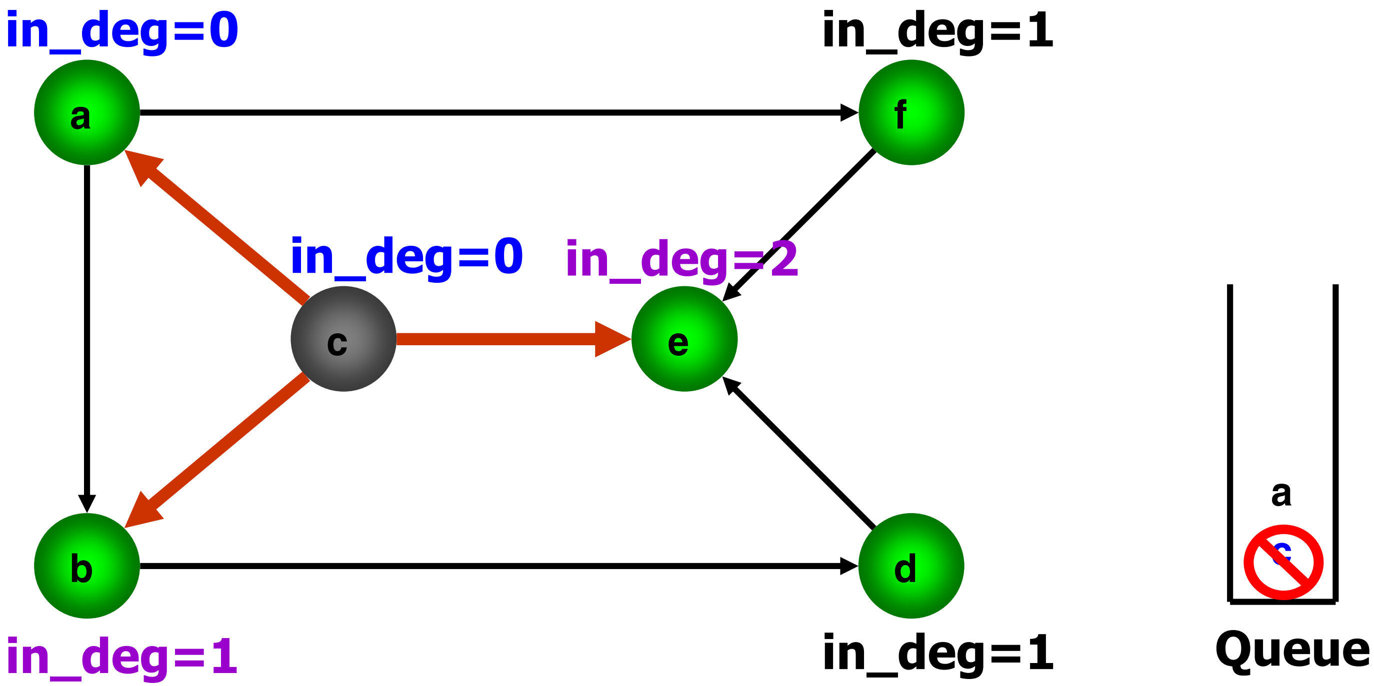
Algorithm Example

- find source nodes (**indegree = 0**)
  - if there is no such node, the graph is NOT DAG



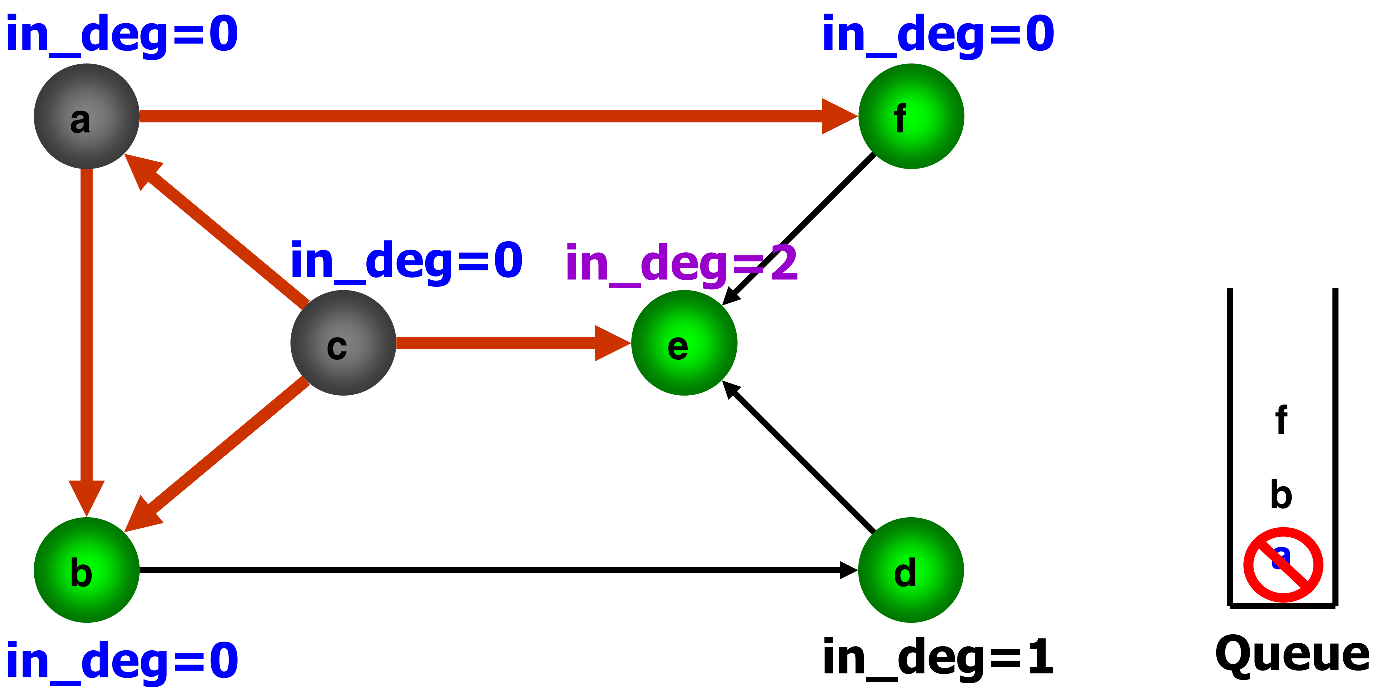
Sorted: –

- span **c**; decrement in_deg of a, b, e
  - store **a** in Queue since in_deg becomes 0



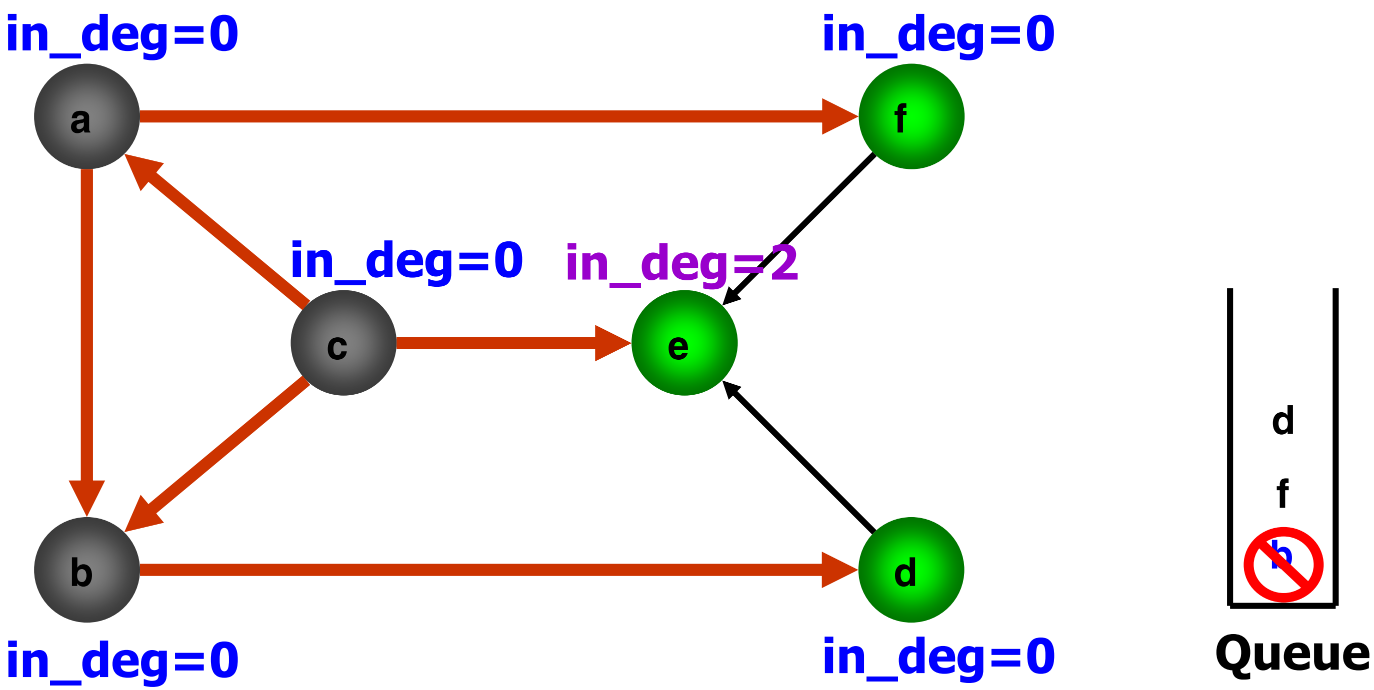
Sorted: c

- span a; decrement in_deg of b, f
  - store b, f in Queue since in_deg becomes 0



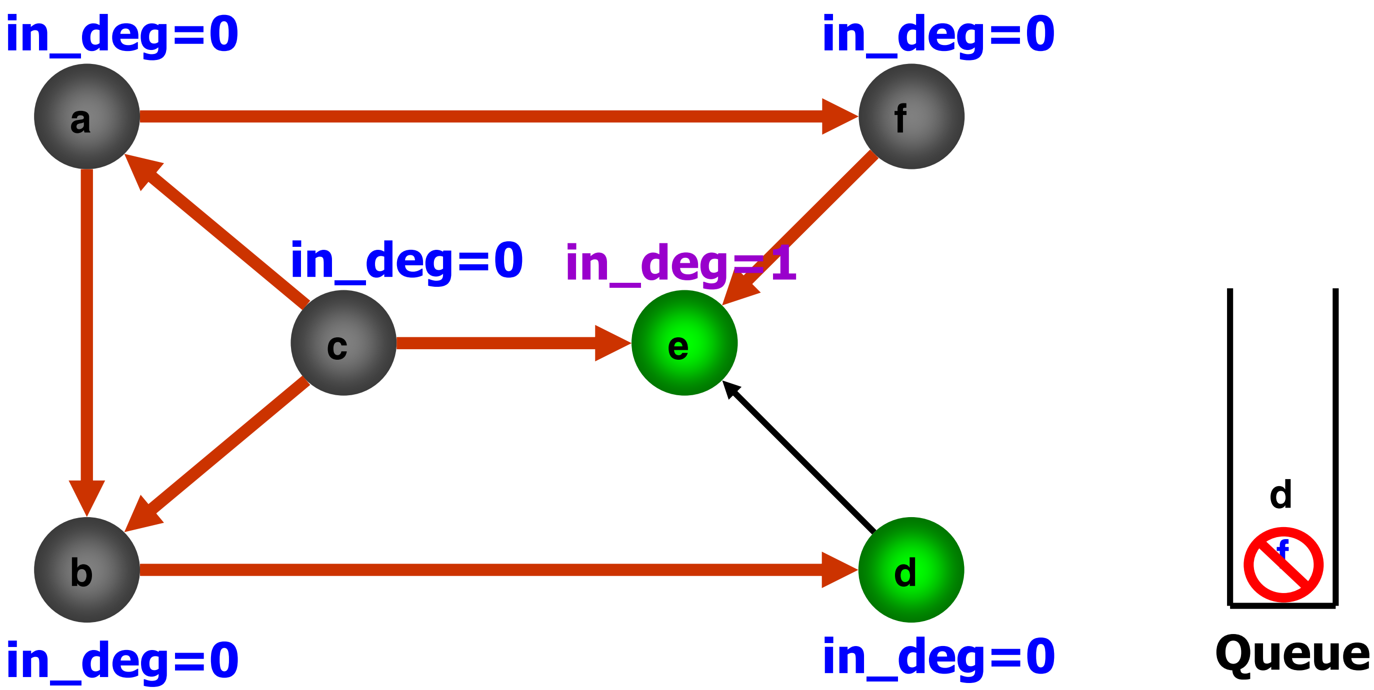
Sorted: c a

• span b; store d in Queue



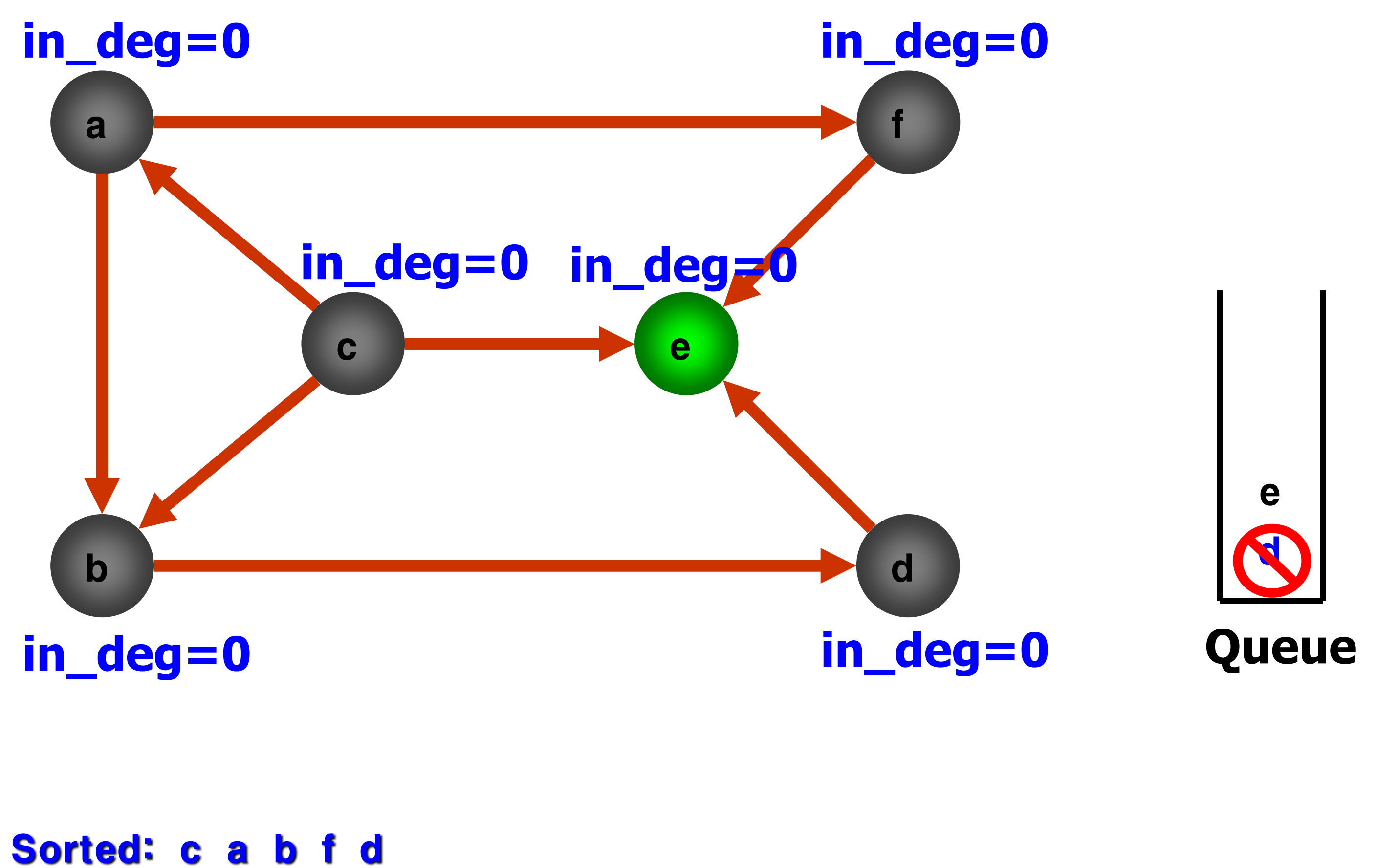
Sorted: c a b

- span f; decrement in_deg of e
  - no node with in_deg = 0 is found

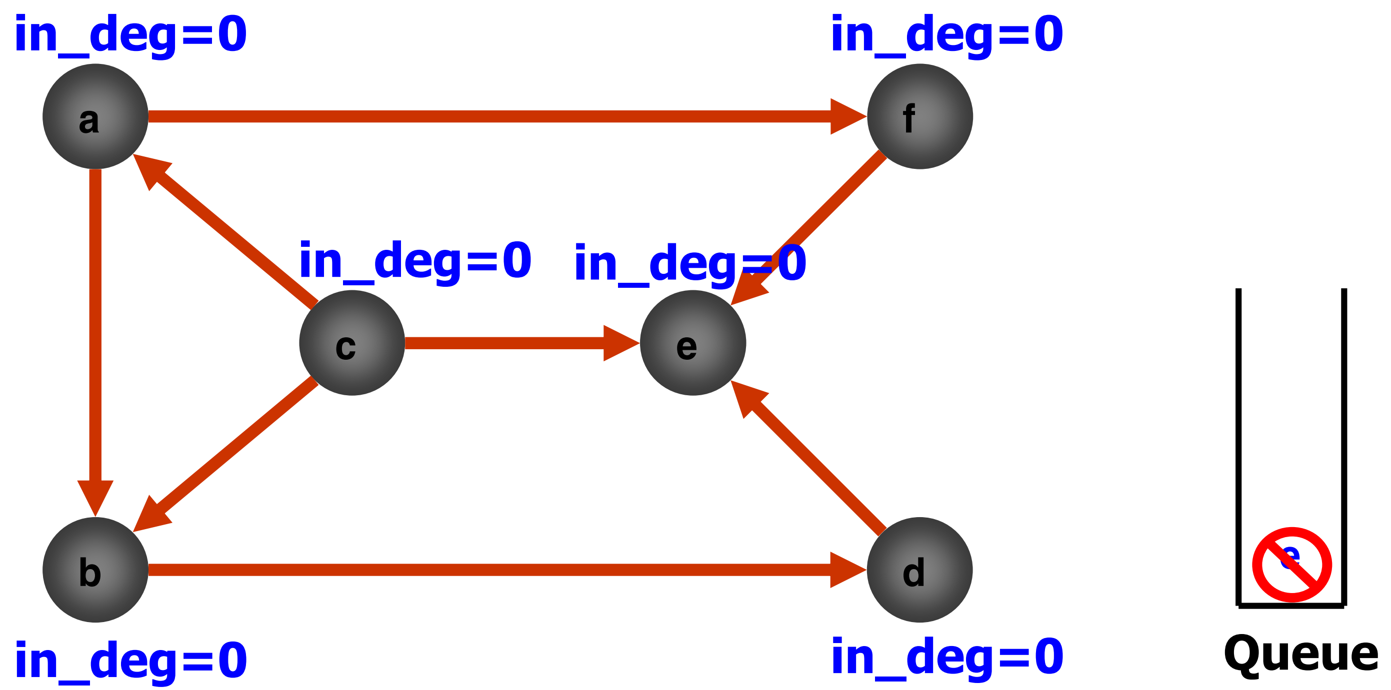


Sorted: c a b f

- span d; store e in Queue.



- span e; Queue is empty



Sorted: c a b f d e

Example Algorithm Summary

- Based on indegree of each vertex
  - if it is 0, this node is the first one in the sorted list
  - span this node
    - move this node from Queue to the sorted list
    - find nodes edged from this node
    - decrement indegrees of them
- It is so similar to BFS

```
topsort(graph *g, int sorted[])
{
        int indegree[MAXV];
        queue zeroin;
        int x, y;
        int i, j;

        compute_indegrees(g,indegree);
        init_queue(&zeroin);
        for (i=1; i<=g->nvertices; i++)
                if (indegree[i] == 0) enqueue(&zeroin,i);

        j=0;
        while (empty(&zeroin) == FALSE) {
                j = j+1;
                x = dequeue(&zeroin);
                sorted[j] = x;
                for (i=0; i<g->degree[x]; i++) {
                        y = g->edges[x][i];
                        indegree[y] --;
                        if (indegree[y] == 0) enqueue(&zeroin,y);
                }
        }

        if (j != g->nvertices)
                printf("Not a DAG -- only %d vertices found\n",j);
}
```

```
compute_indegrees(graph *g, int in[])
{
        int i,j;                        /* counters */

        for (i=1; i<=g->nvertices; i++) in[i] = 0;

        for (i=1; i<=g->nvertices; i++)
                for (j=0; j<g->degree[i]; j++)
                        in[ g->edges[i][j] ] ++;
}
```

입력차수 '0'인 노드에서 시작!

큐가 비워질 때 까지 루프내의 동작을 수행!

노드 y와 연결된 노드의 입력차수를 하나씩 감소!

입력차수가 '0'인 노드가 생성되면 큐에 저장!

Q : Bicoloring

- Decide whether a given connected graph can be bicolored, i.e., can the vertices be painted red and black such that no two adjacent vertices have the same color.
- To simplify the problem, you can assume the graph will be connected, undirected, and not contain self-loops (i.e., edges from a vertex to itself).

Input

The input consists of several test cases. Each test case starts with a line containing the number of vertices n, where $1 < n < 200$. Each vertex is labeled by a number from 0 to $n-1$. The second line contains the number of edges l. After this, l lines follow, each containing two vertex numbers specifying an edge.

An input with $n = 0$ marks the end of the input and is not to be processed.

Output

Decide whether the input graph can be bicolored, and print the result as shown below.

Sample Input

```
3
3
0 1
1 2
2 0
9
8
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0
```

Sample Output

```
NOT BICOLORABLE.
BICOLORABLE.
```