Problem Solving Techniques 문제해결

Jinkyu Lee

Dept. of Computer Science and Engineering,
Sungkyunkwan University (SKKU)

Contents

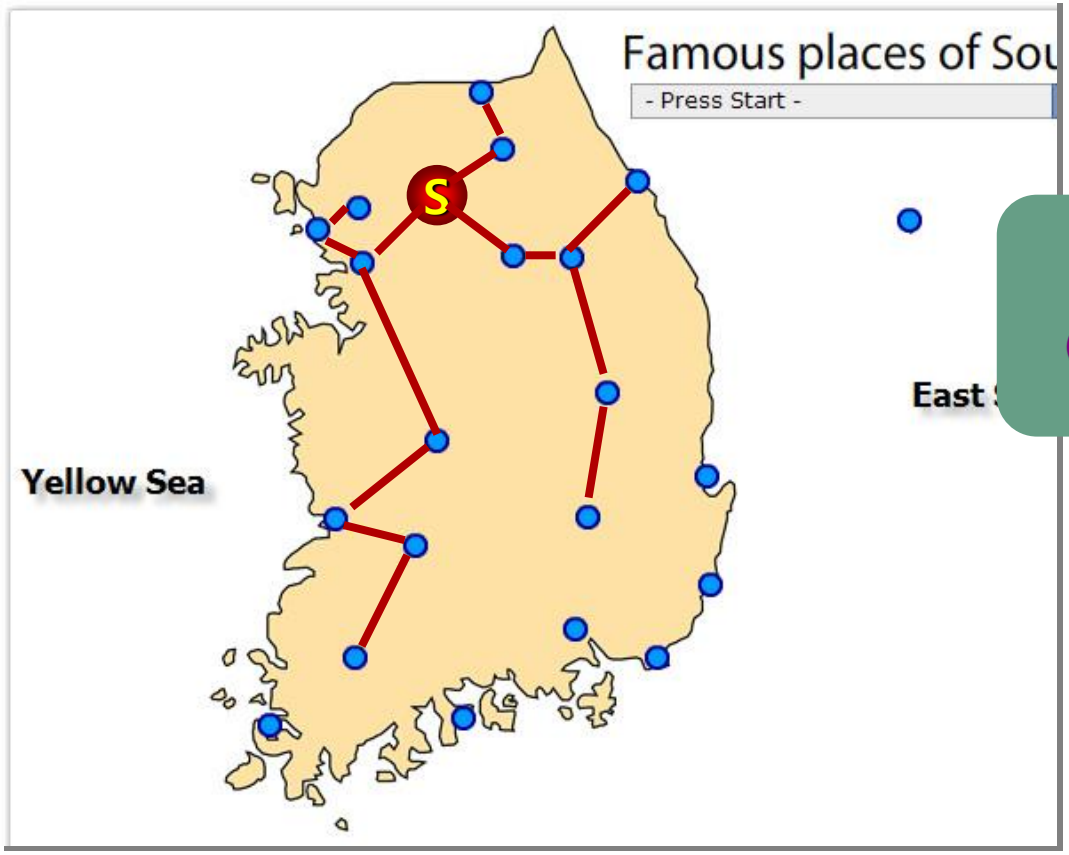
- Chapter 9 – Graph Traversal
  - Breadth-First Search (BFS)

Traversal

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

Breadth-first search (BFS)

- We would like to visit famous historic sites in Korea, starting from Seoul.
- In the previous slides, we investigated a proper Representation method.
- Now, we want to construct a tree that spans all the historic sites.
- How do we achieve the goal of this mission?



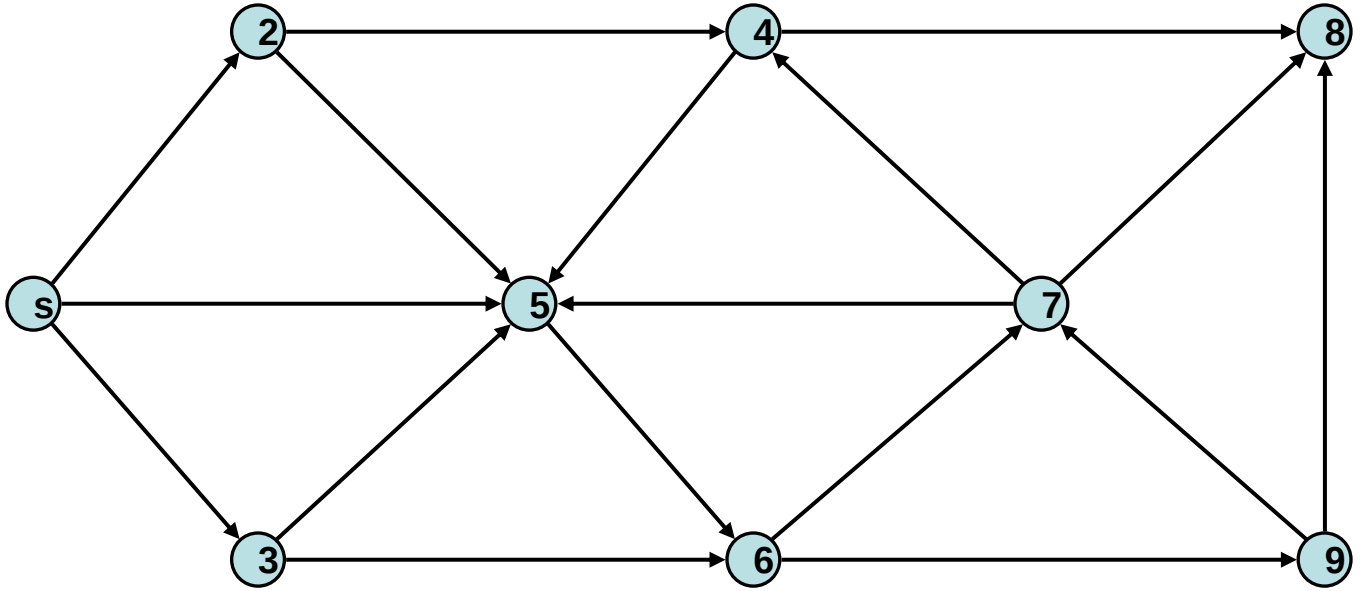
Breadth-first search (BFS)

- The (vertex or edge) visiting sequence is not important!
  - It is used where the visiting sequence is meaningless
- Applicable when finding the shortest path in the Unweighted graphs
- Given a graph ***G = (V, E)*** and a distinguished ***source*** vertex ***s***
  - BFS systematically explores the edges of G to *"discover"* every vertex that is reachable from ***s***.
  - BFS discovers all vertices at distance ***k*** from ***s*** before discovering any vertex at distance ***k+1***
    - **Here, distance denotes the smallest number of edges**

Breadth-first search (BFS)

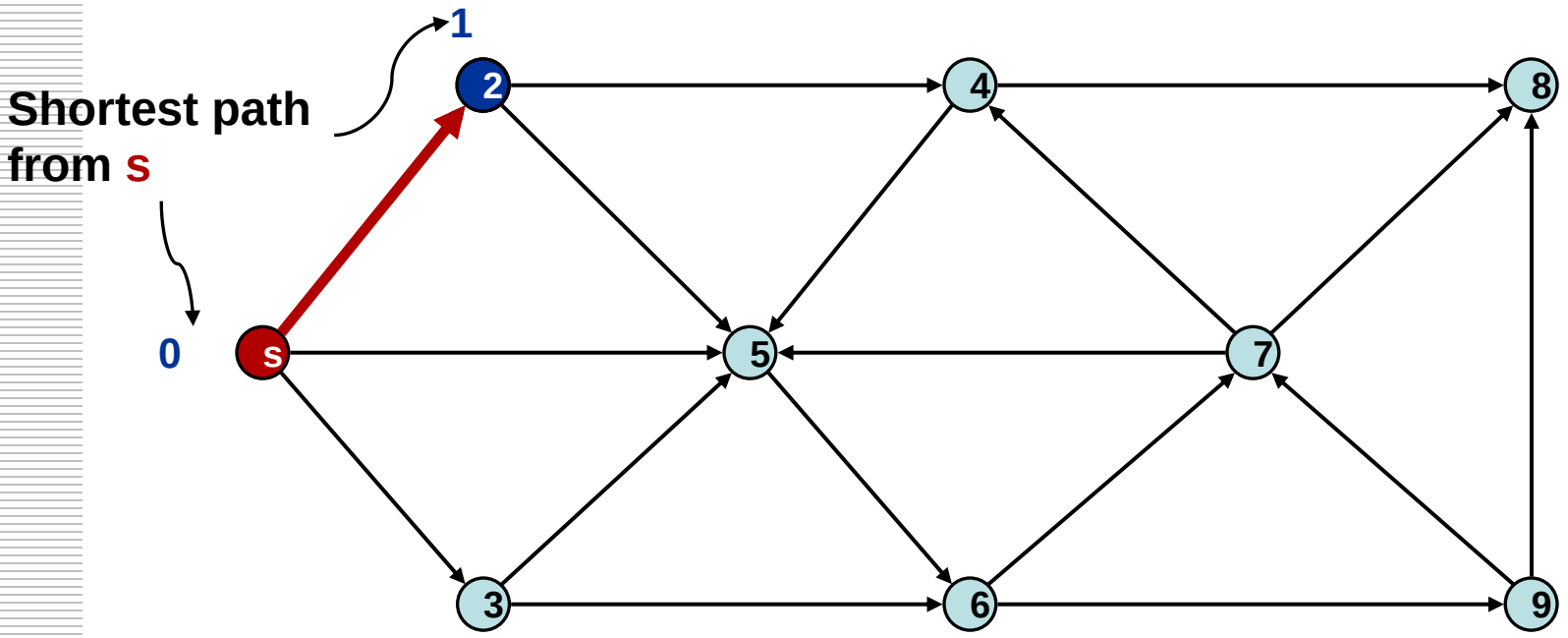
```
BFS(G, v)
{
        for each v ∈ V –{s}
                visited[v] ← NO;
        visited[s] ← YES;                ▷ s: Starting vertex
        enqueue(Q, s);                   ▷ Q: Queue
        while (Q ≠ {} ) {
                u ← dequeue(Q);
                for each v ∈ L(u)    ▷ L(u): Adjacency list of vertex u
                        if (visited[v] = NO) then
                                visited[v] ← YES;
                                enqueue(Q, v);
                        }

        }
}
```

Breadth-first search (BFS)



Let us apply the BFS
to the above graph!

Breadth-first search (BFS)



Undiscovered

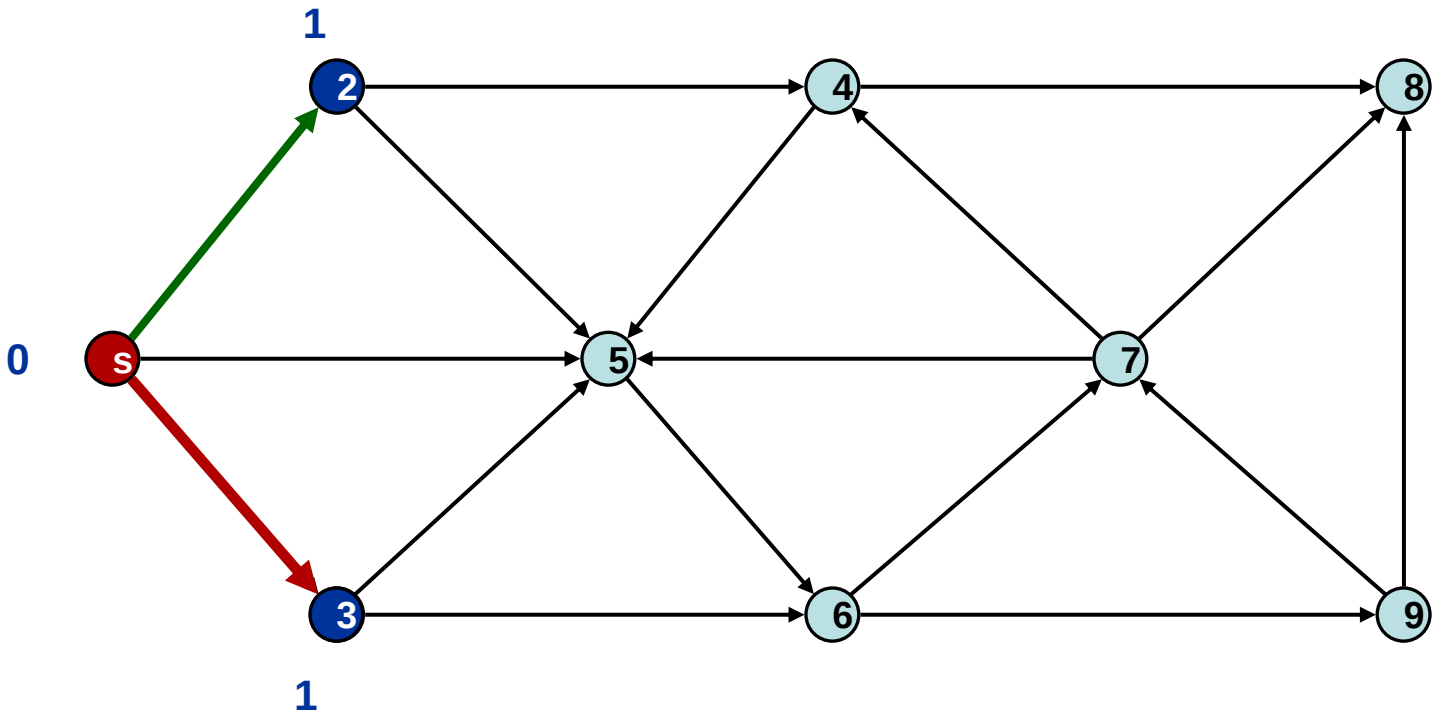
Discovered

Top of queue

Finished

Queue: s 2

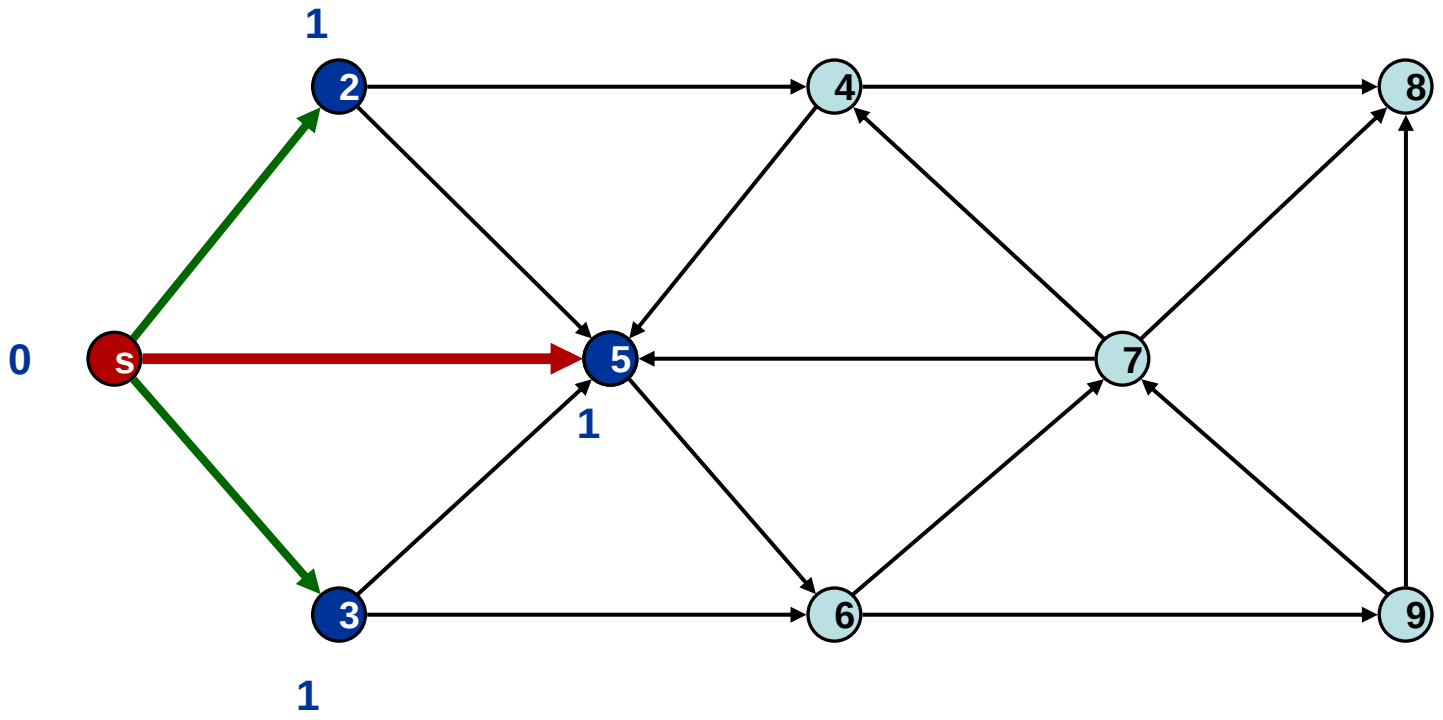
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: s 2 3

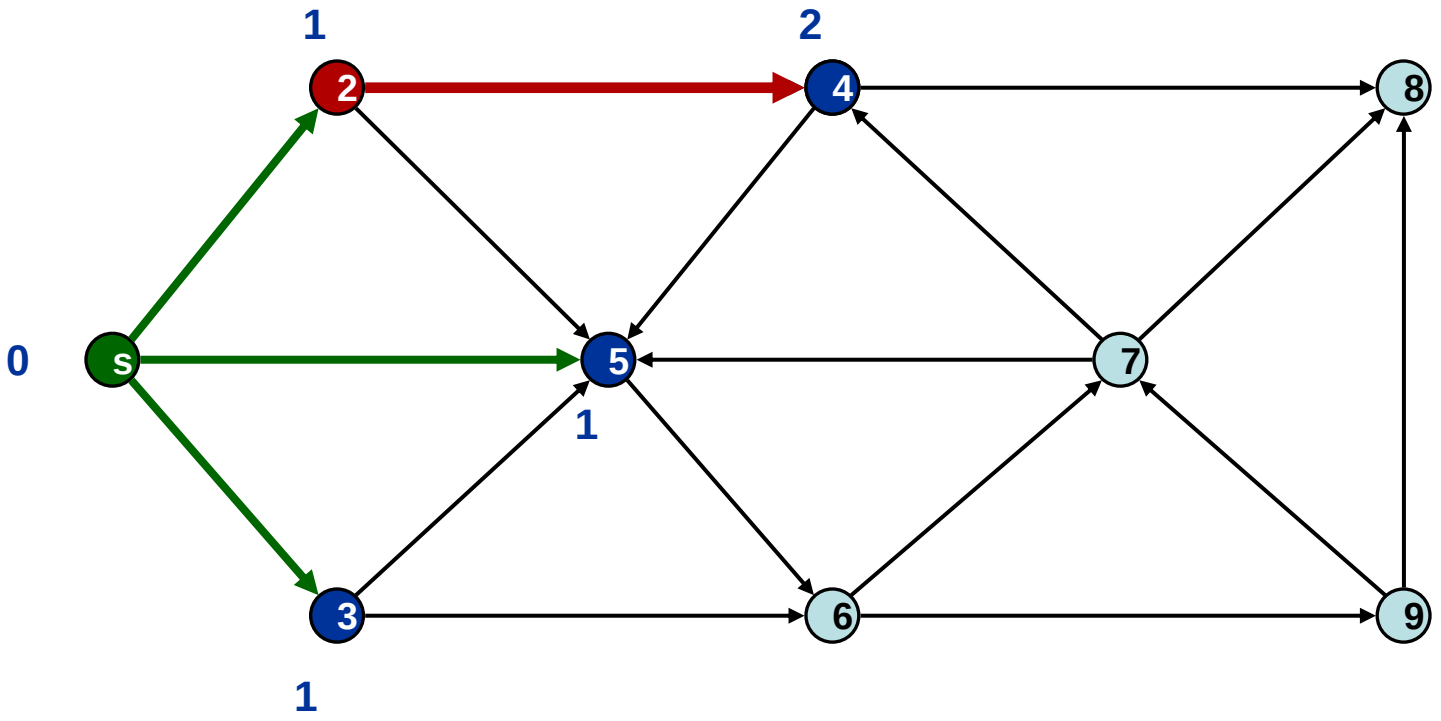
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: s 2 3 5

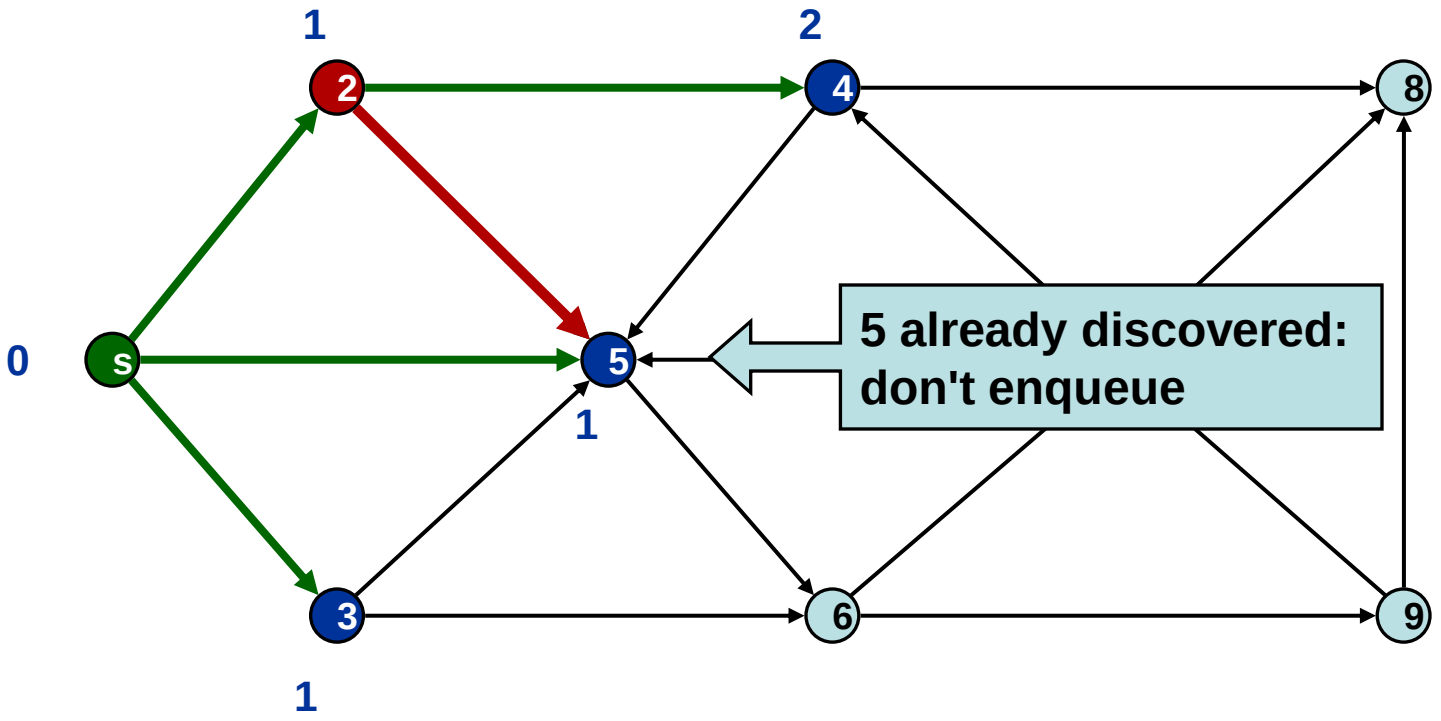
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5 4

Breadth-first search (BFS)



Undiscovered

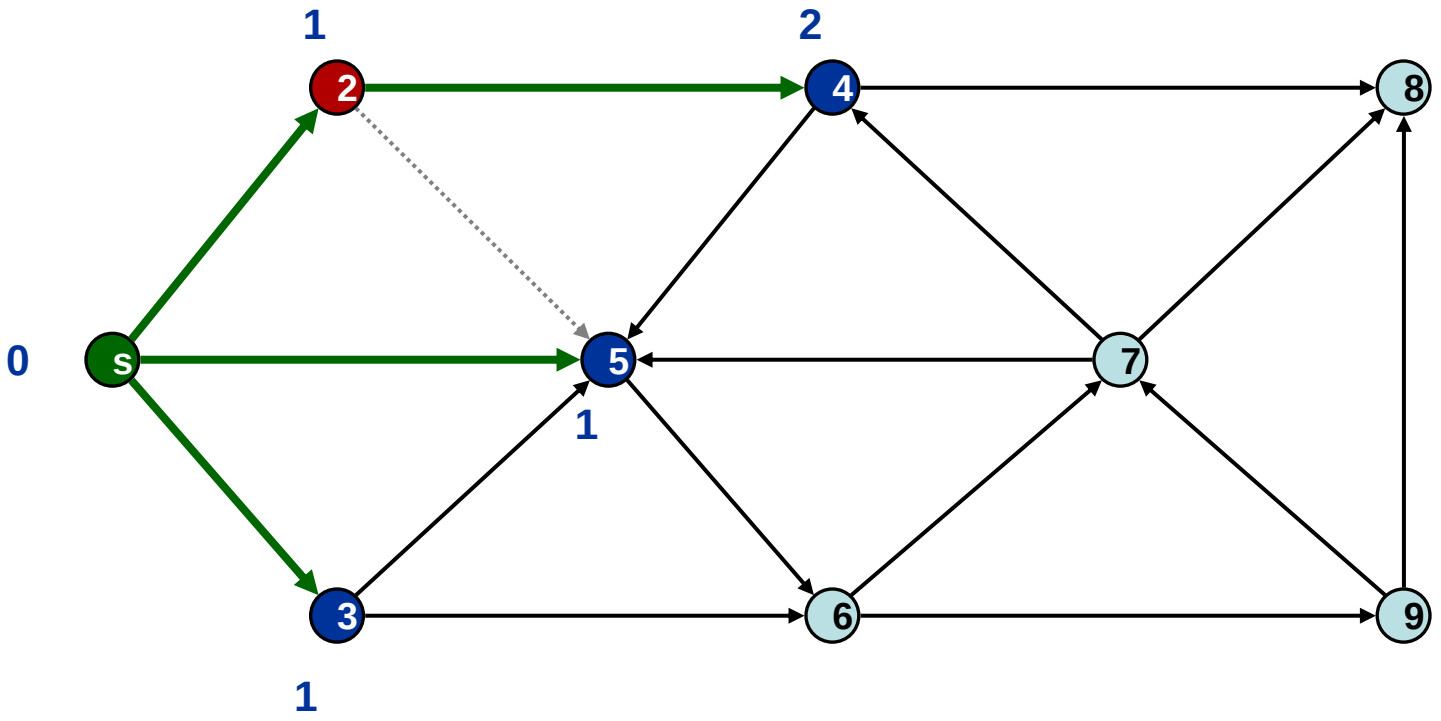
Discovered

Top of queue

Finished

Queue: 2 3 5 4

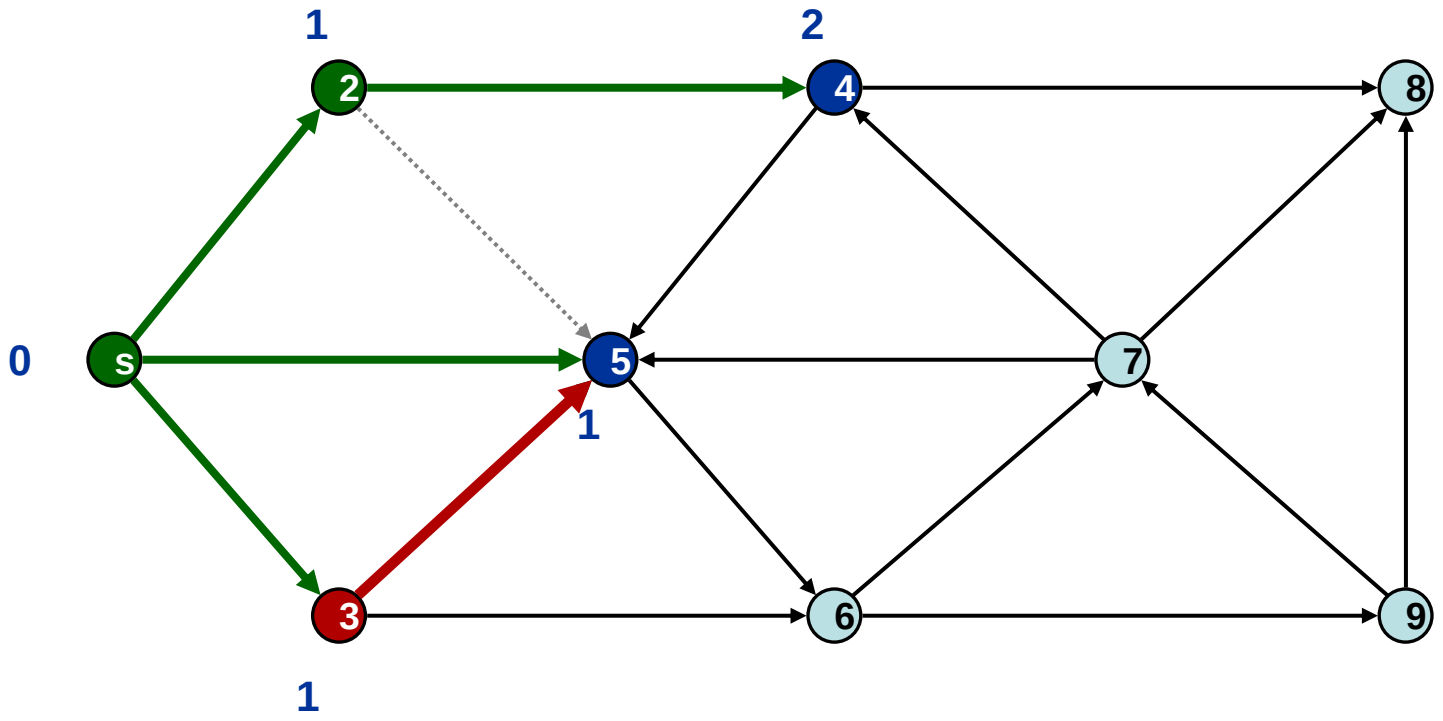
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5 4

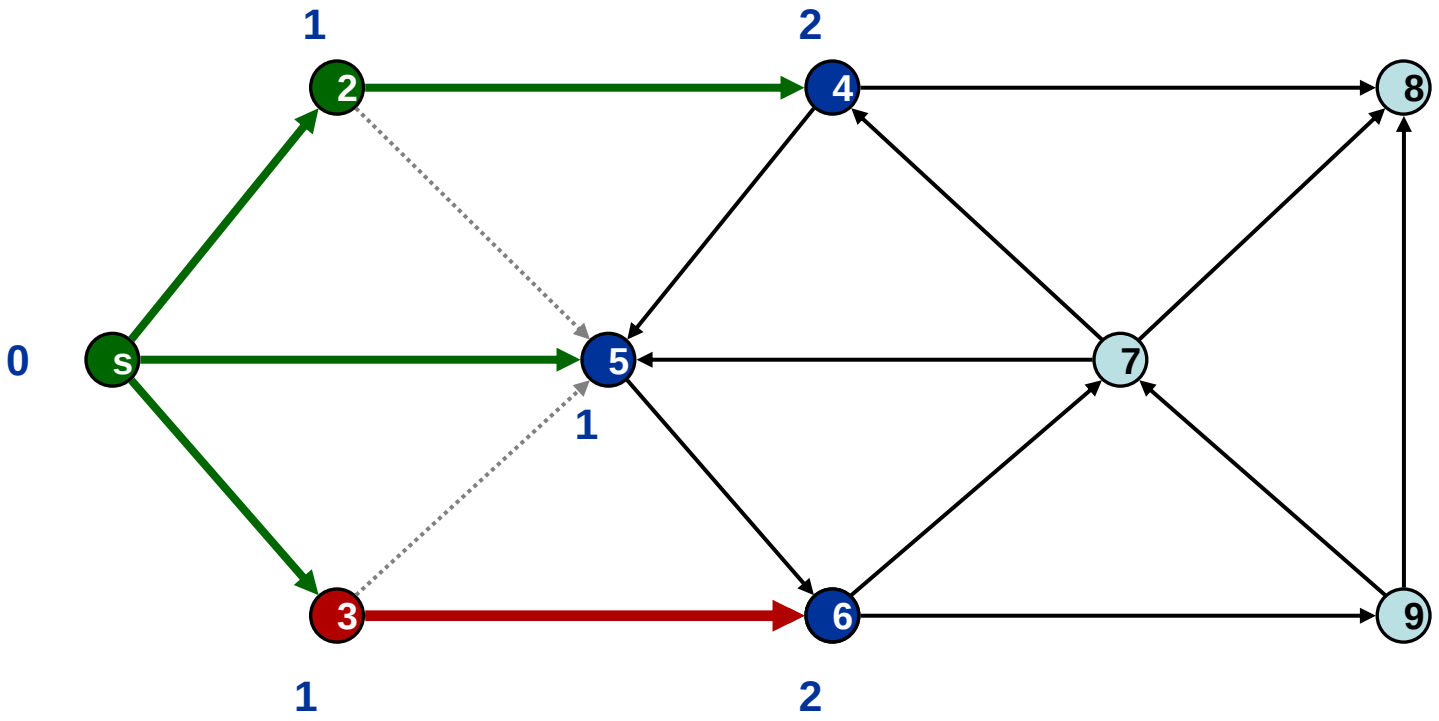
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4

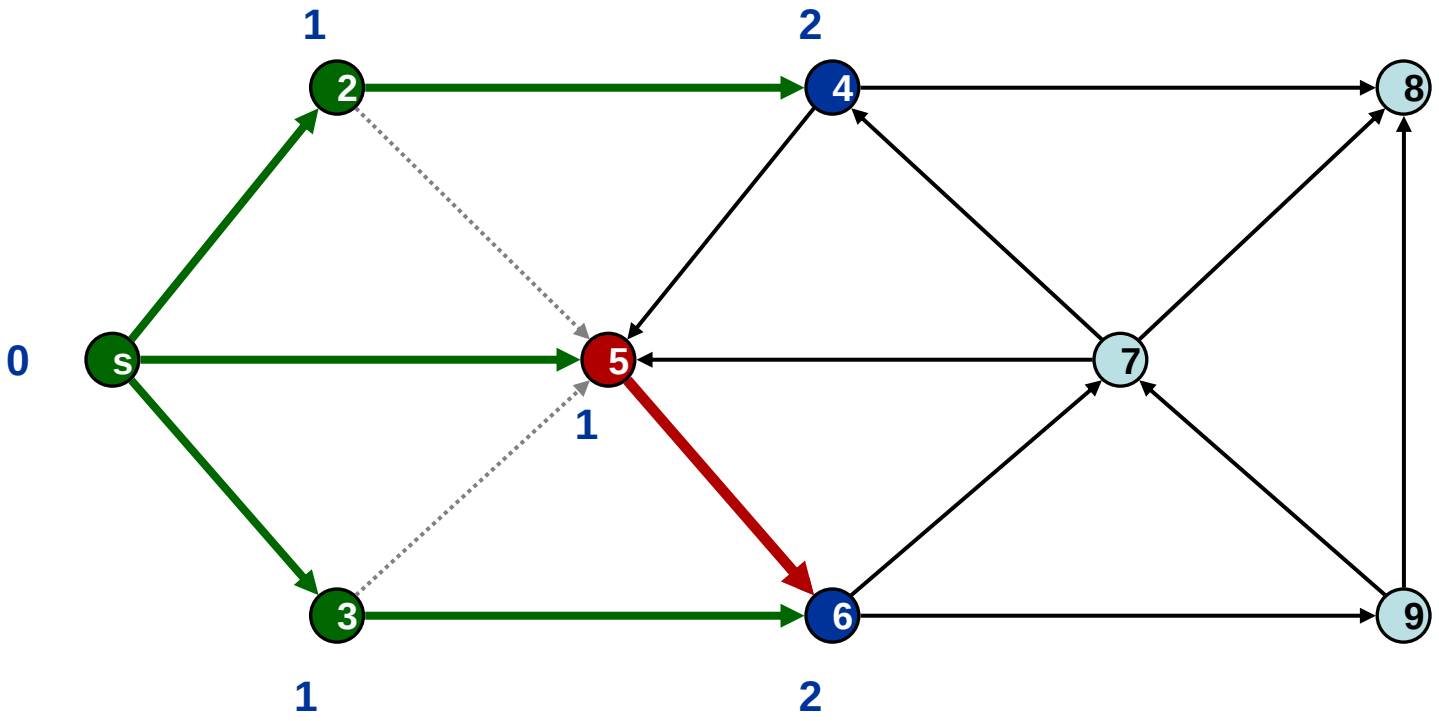
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4 6

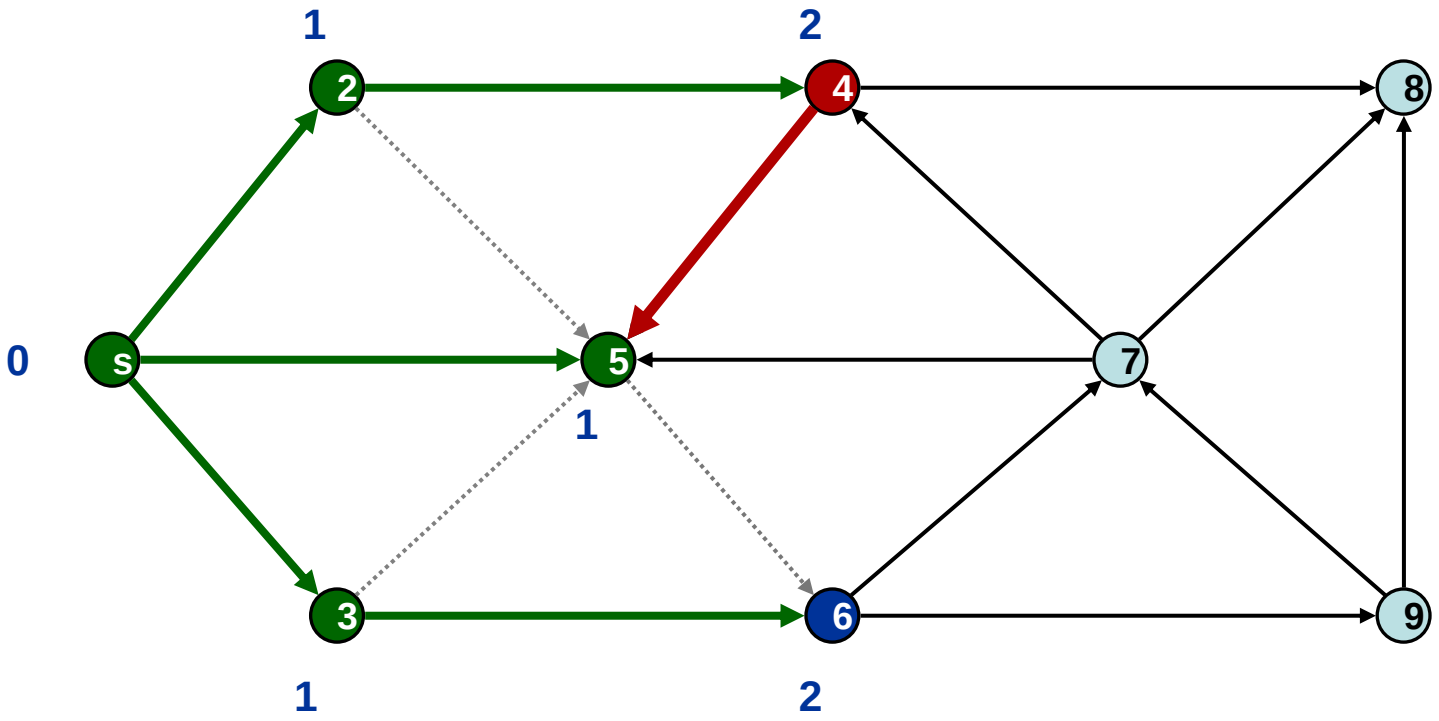
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 5 4 6

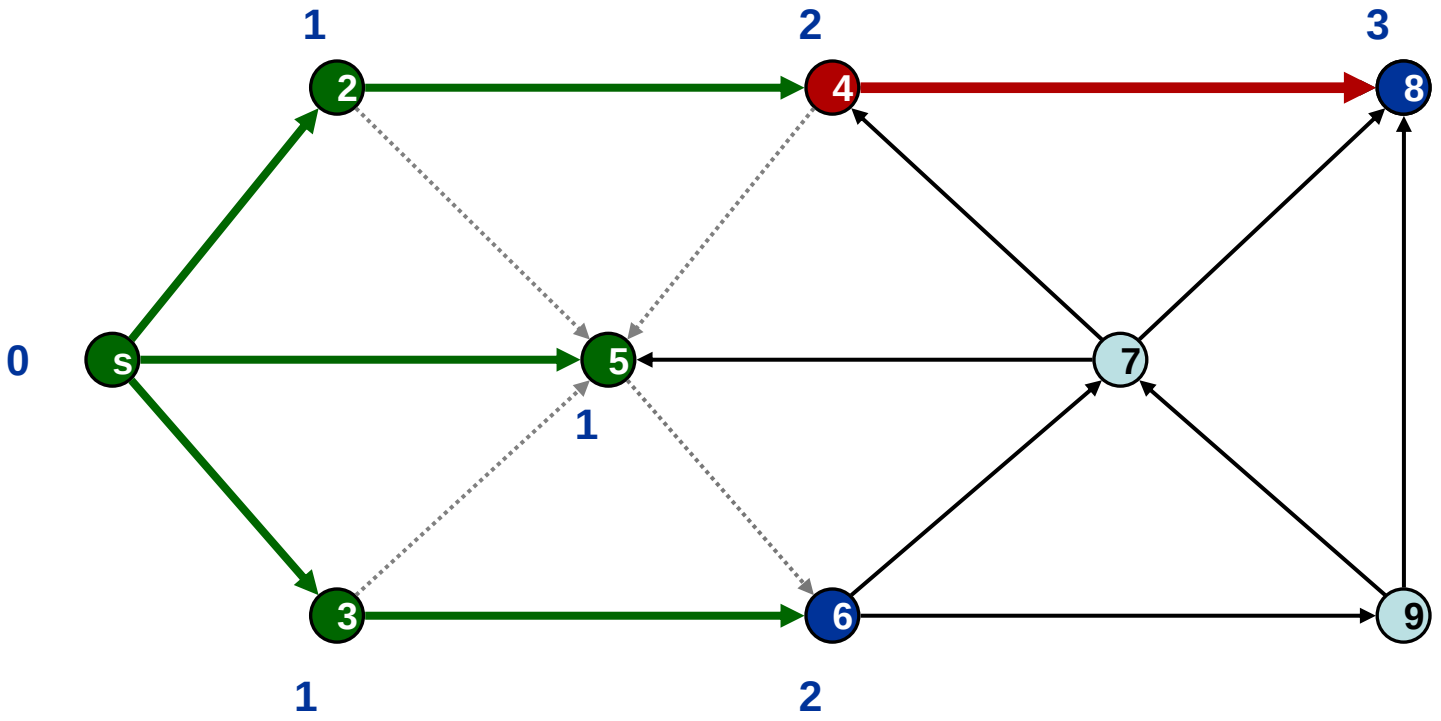
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6

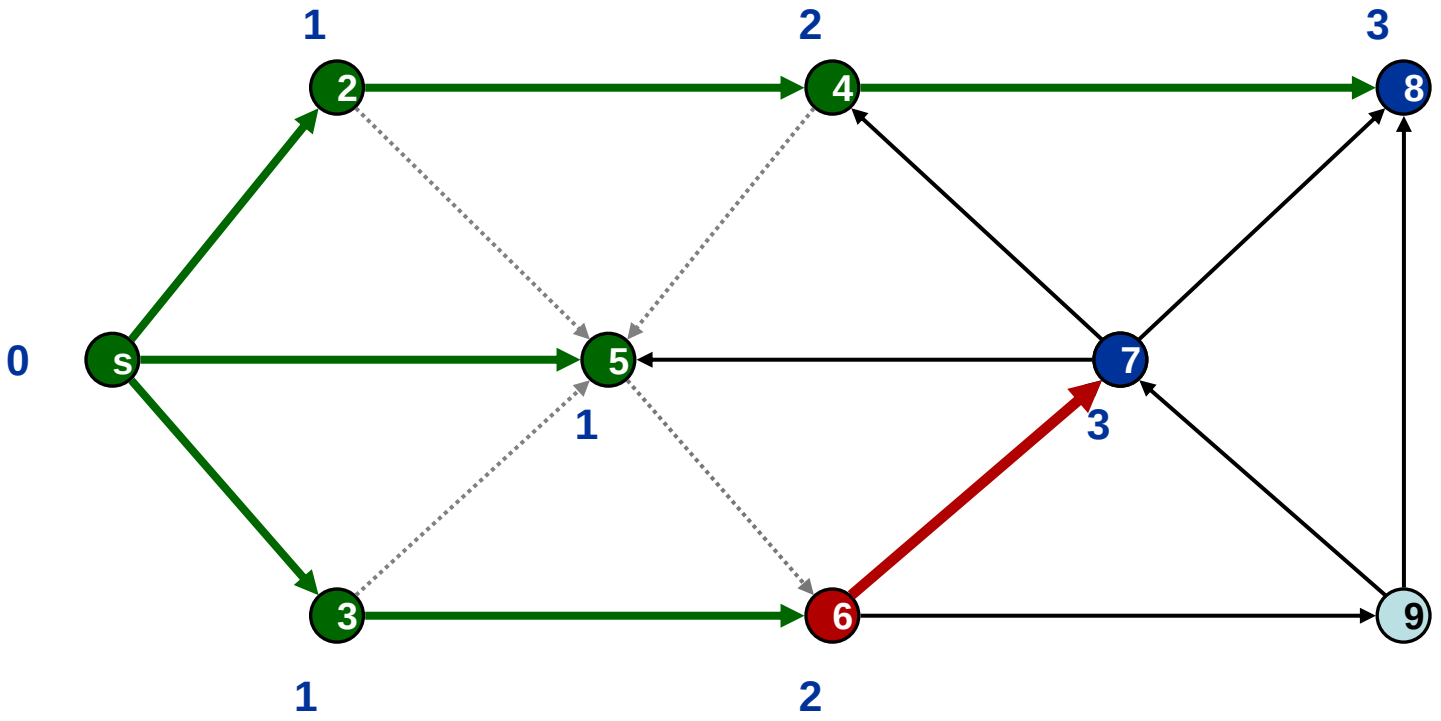
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6 8

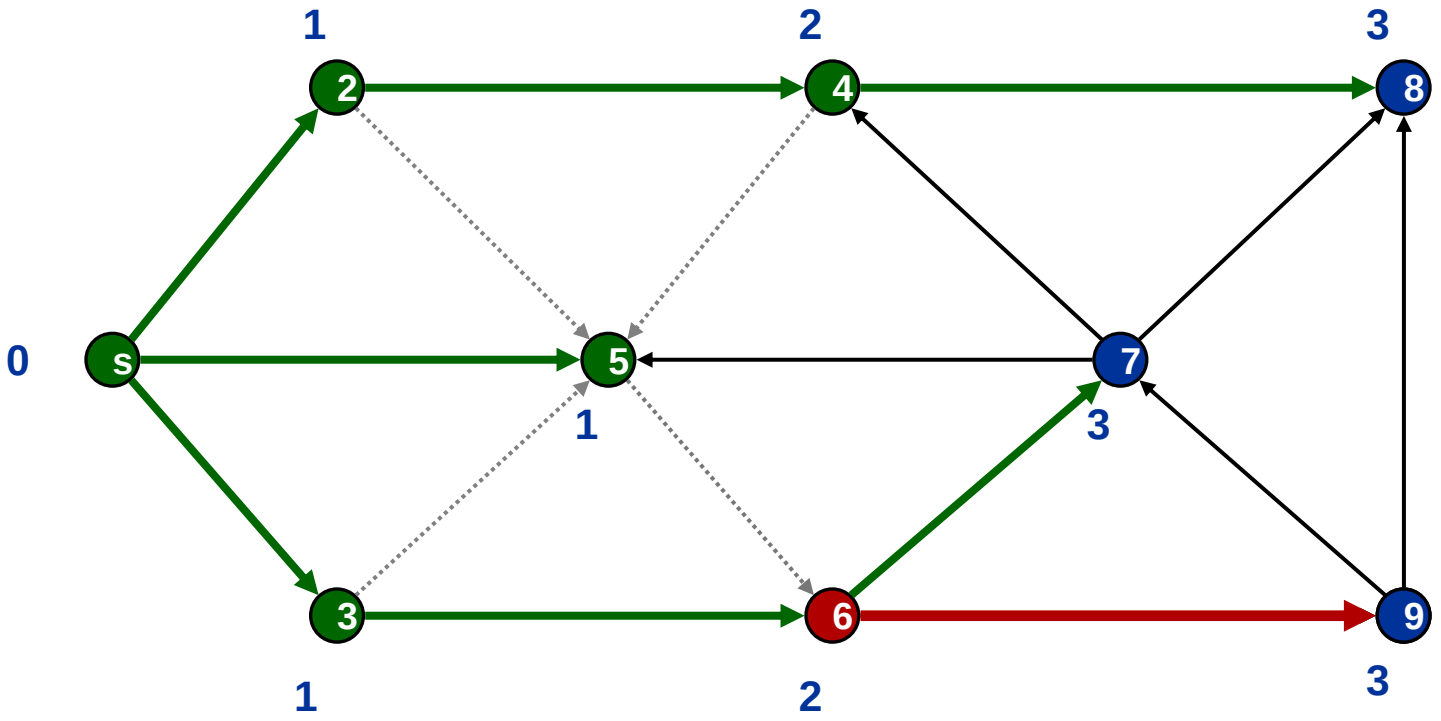
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7

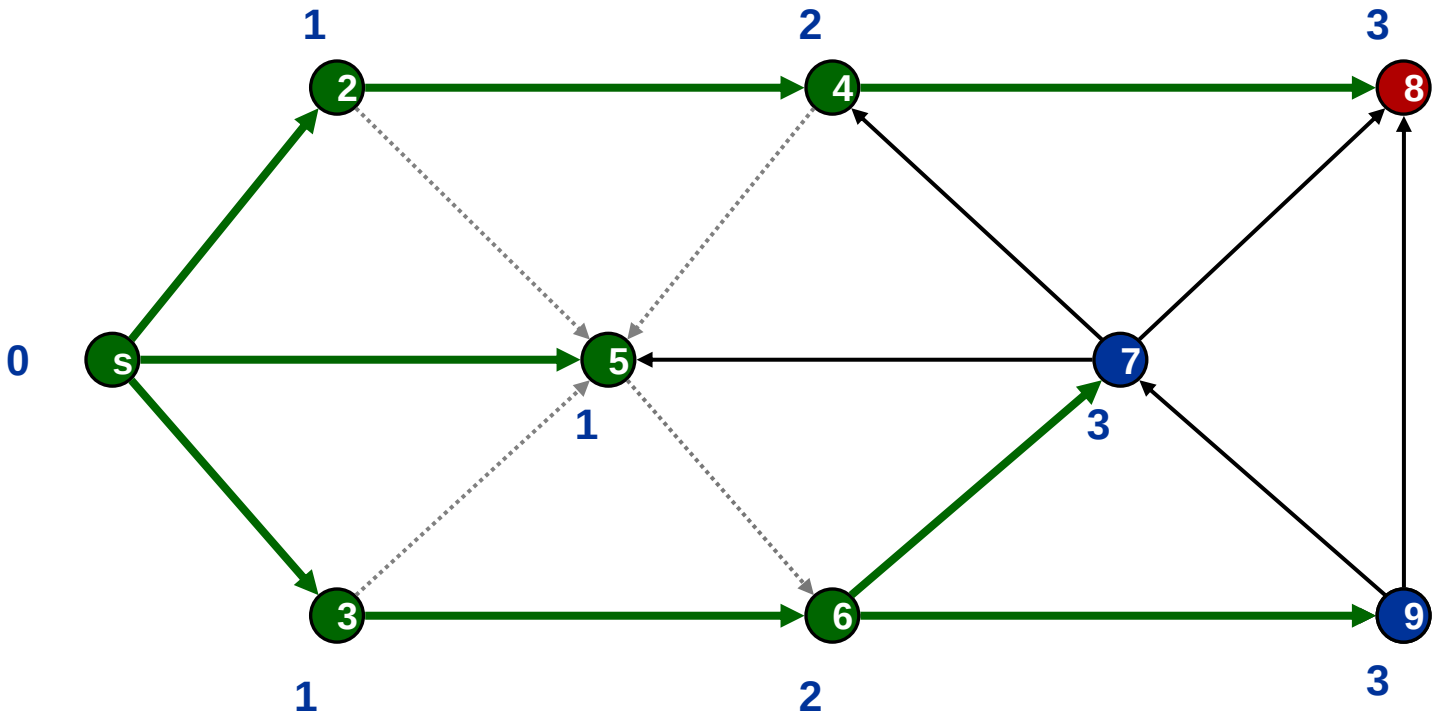
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7 9

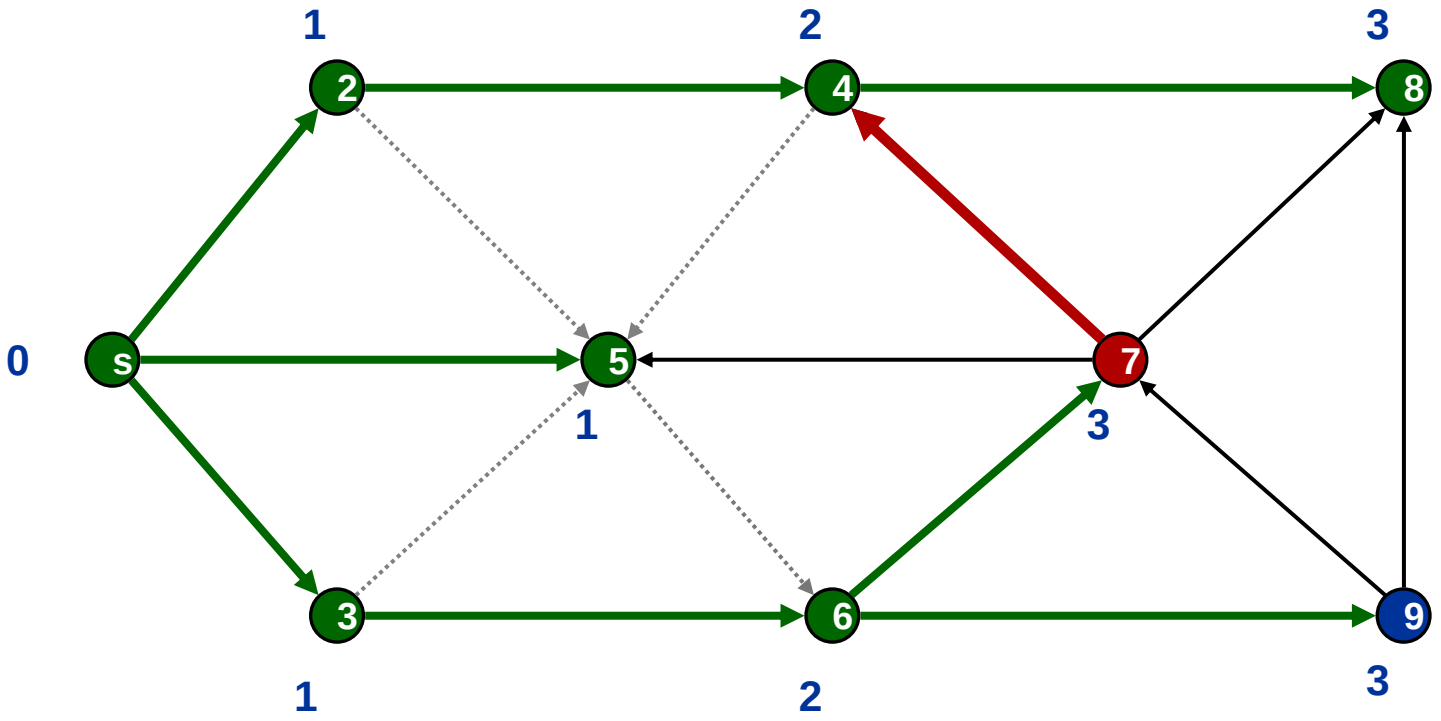
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 8 7 9

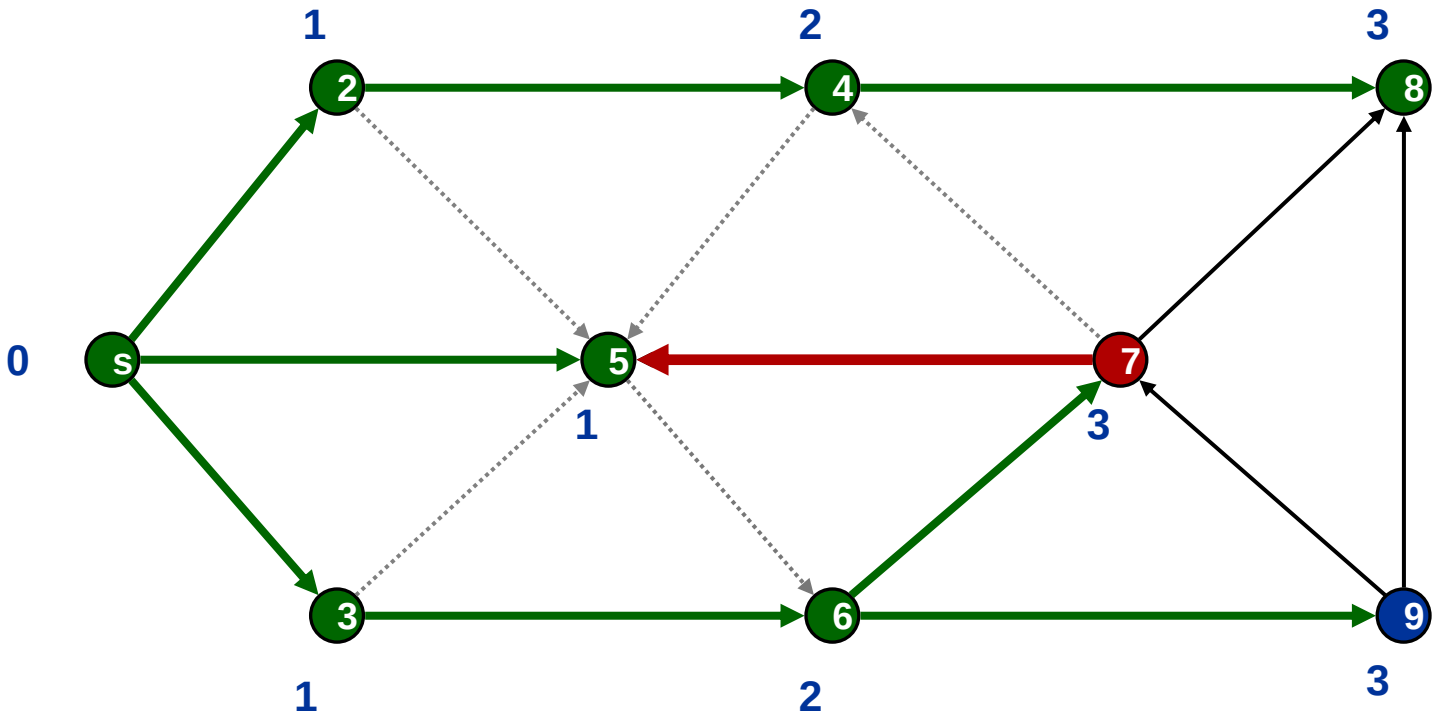
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

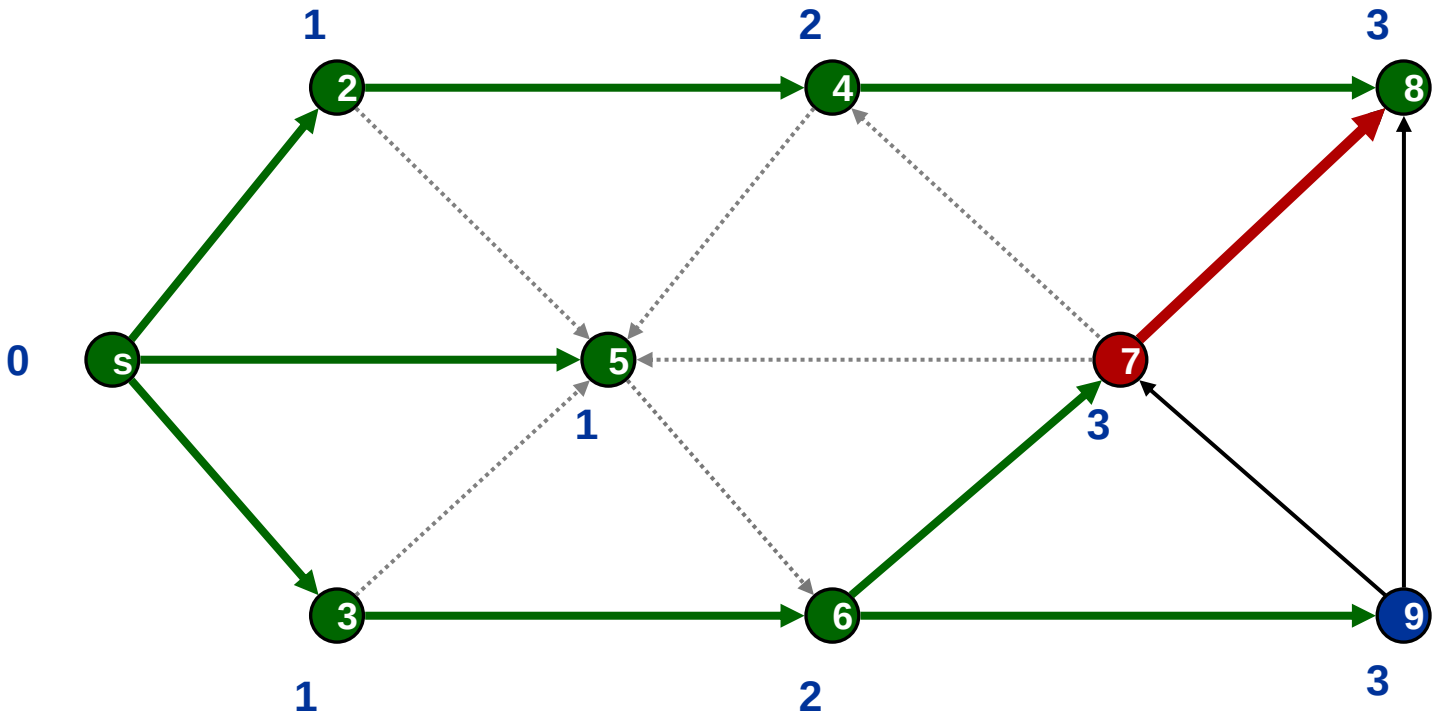
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

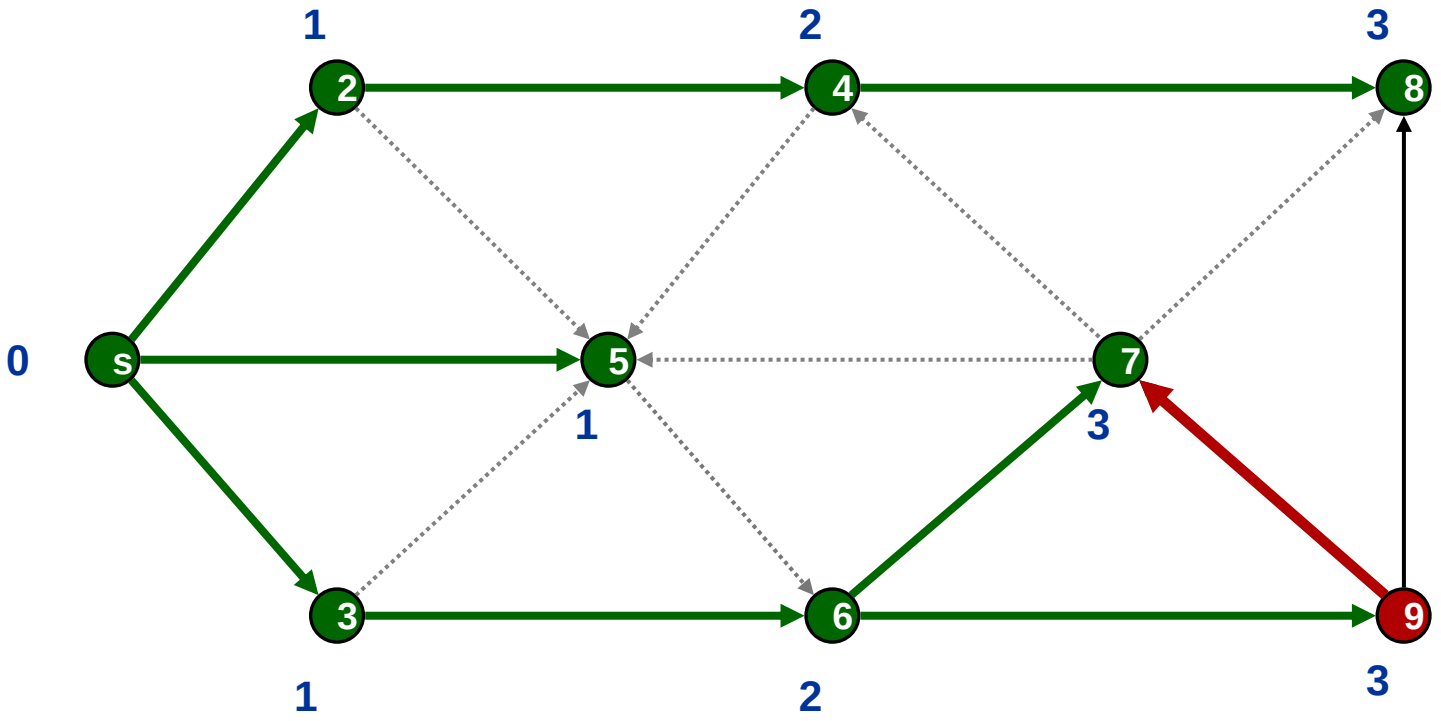
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

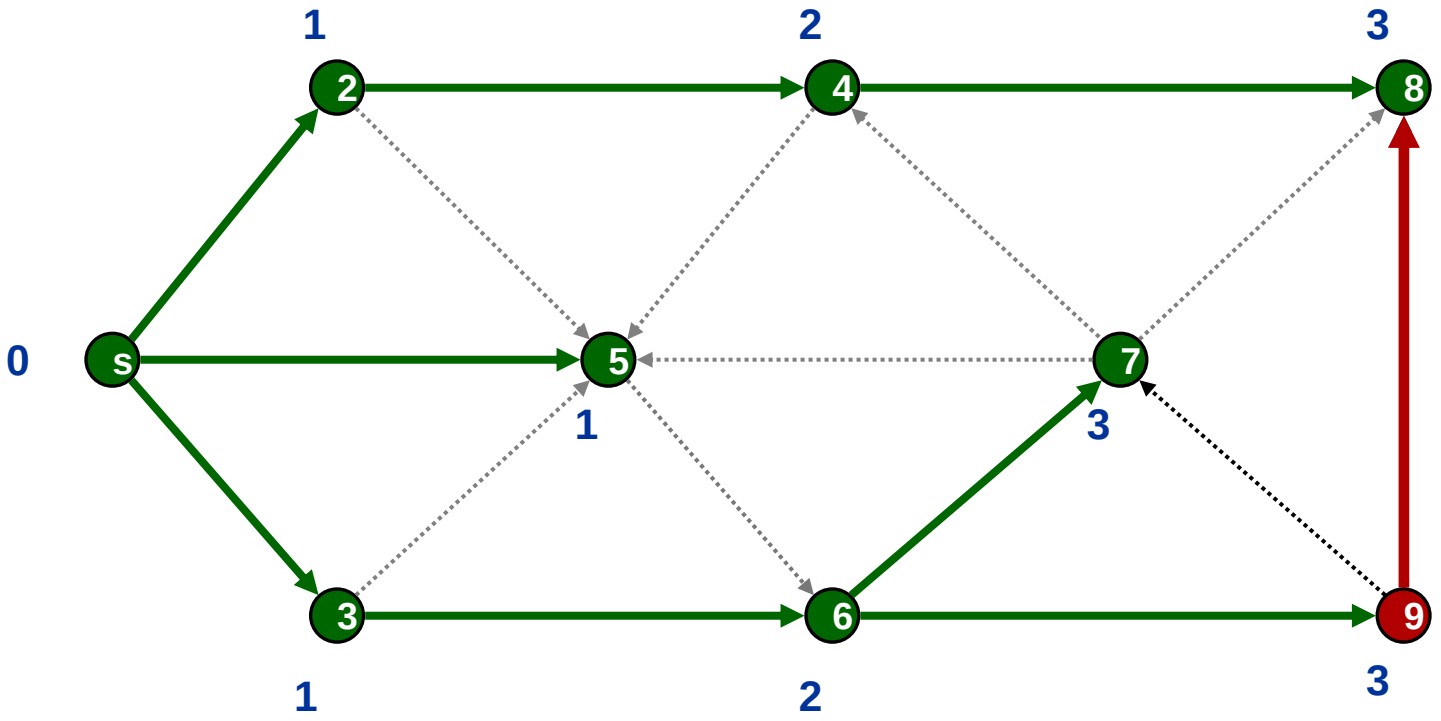
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

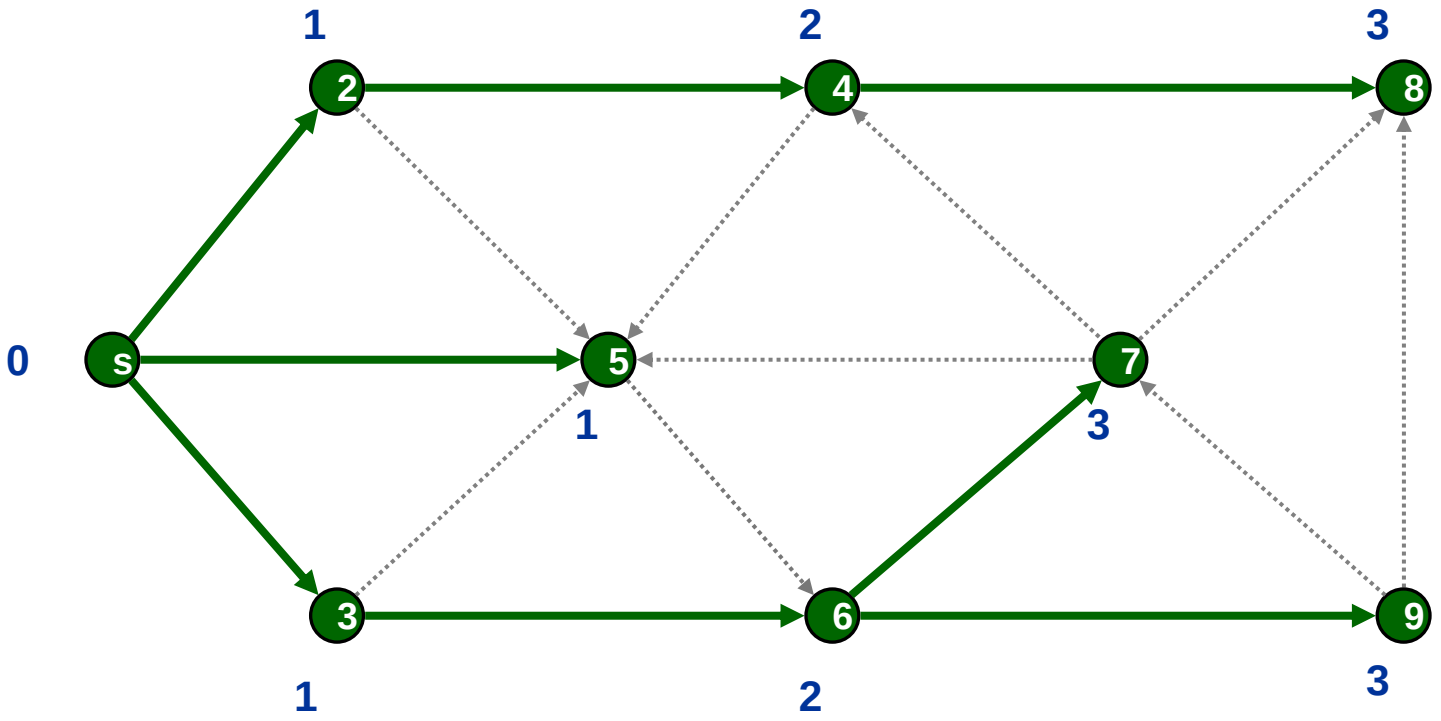
Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

Breadth-first search (BFS)



Undiscovered
Discovered
Top of queue
Finished

➔ Since Queue is empty, STOP!

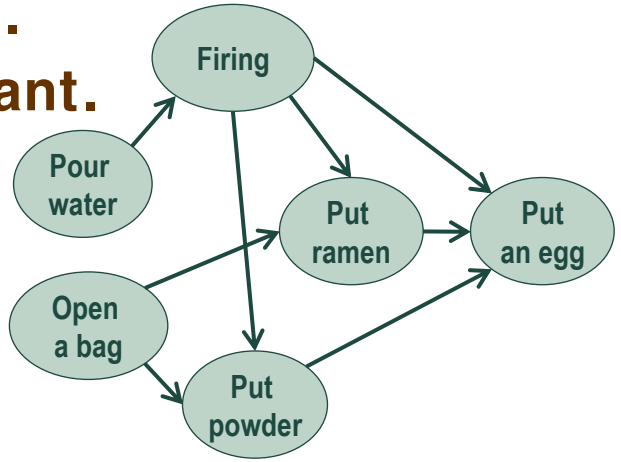
Contents

- Chapter 9 – Graph Traversal
  - Program design example

Topological Sorting: Example <1>

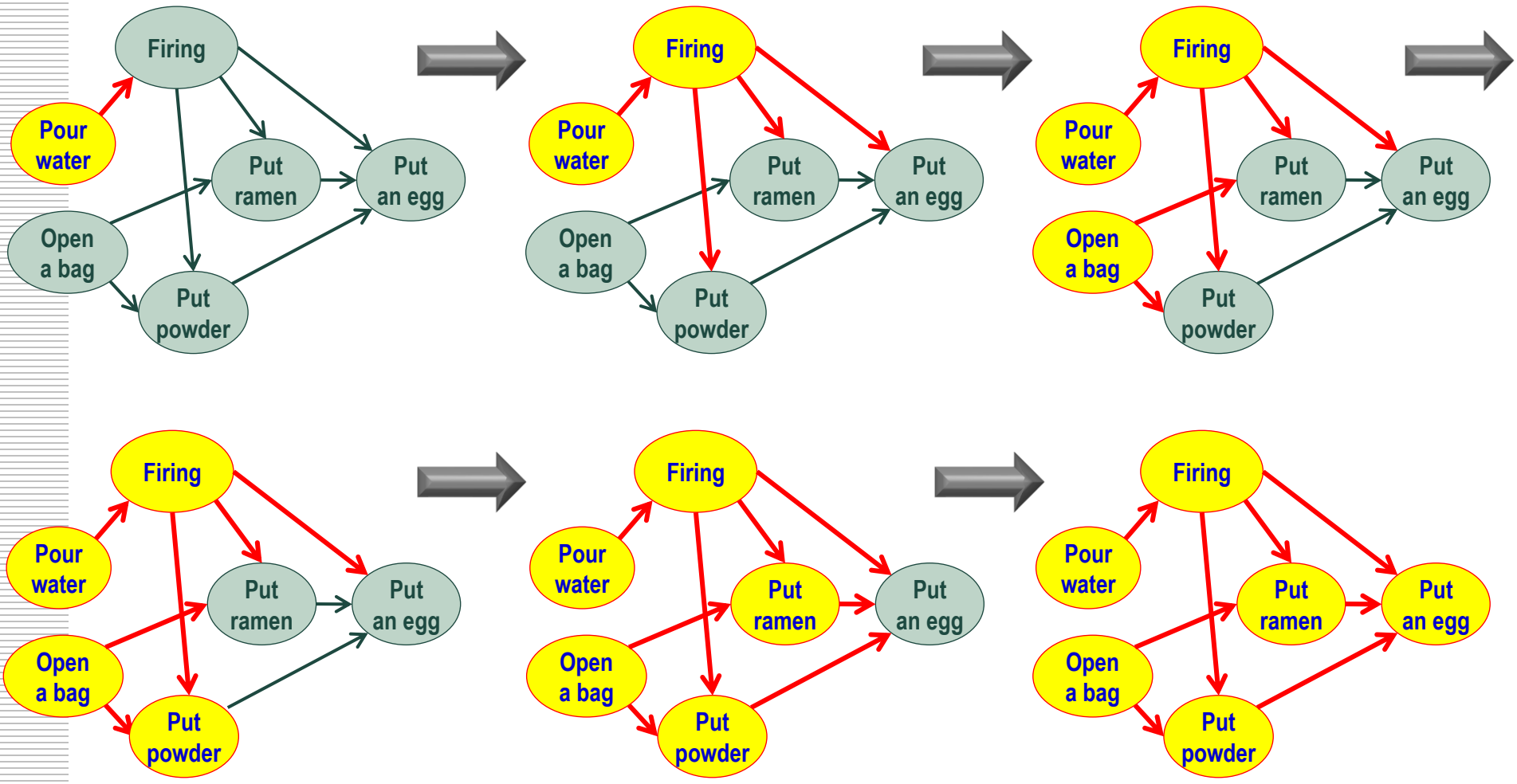
- You have never cooked ramen. But you are trying to do it, referring to a recipe as follows:
  - Open a bag
  - Pour water
  - Firing
  - Put ramen
  - Put powder
  - Put an egg

- One task is allowable at each instant.
- The order of cooking tasks is important.
- How to cook ramen in order?
  - Assume that the cooking tasks are represented by this graph.



Topological Sorting: Example <2>

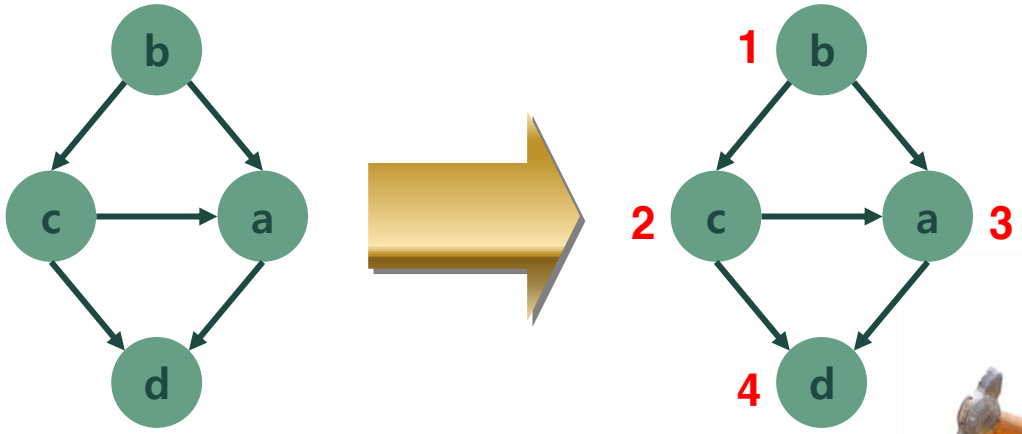
- Their order can be known by the topological sorting.



Topological Sorting: Concept

Definition

- A *topological sort* of a DAG *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

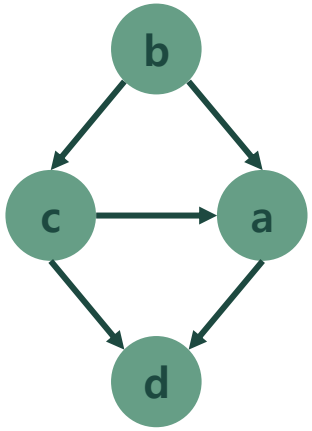


An example

- Task (or Job) Scheduling
  - This can be modeled by a DAG, in which tasks are nodes and their dependencies are represented by edges.
  - The process of sub-tasks is dependent on that of ancestors

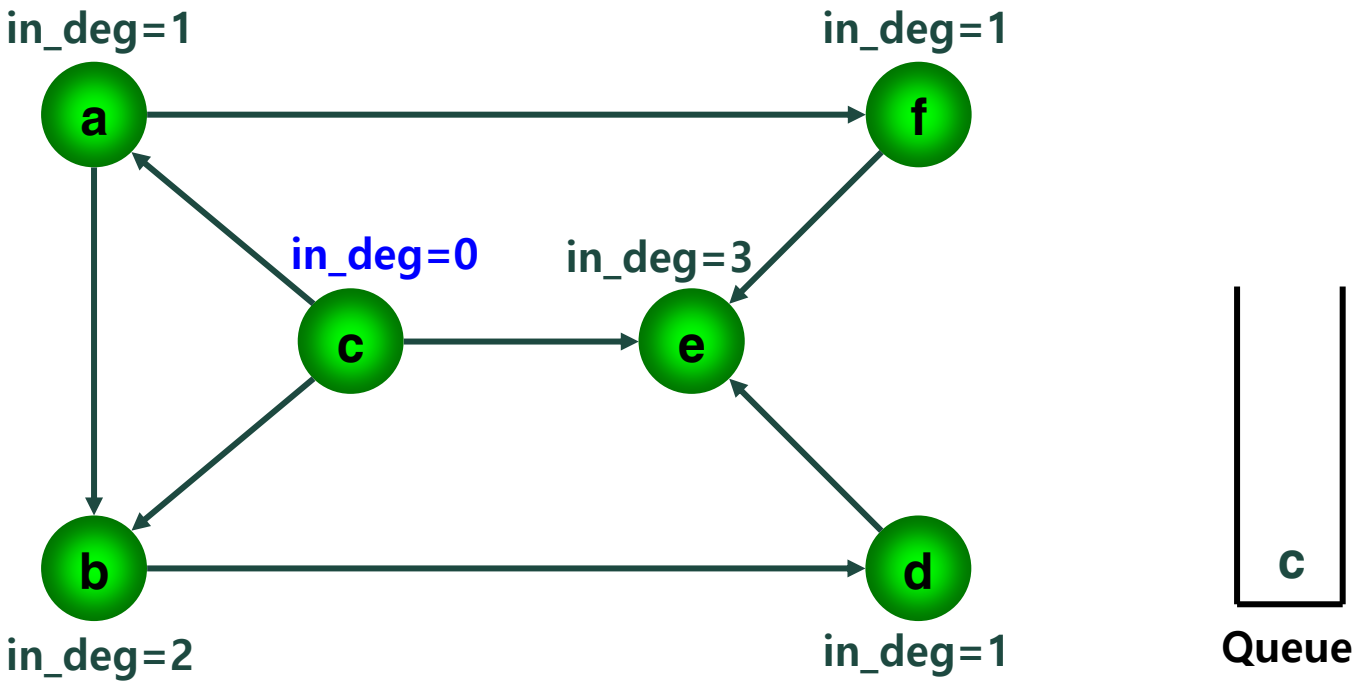
Topological Sorting: Method 〈0〉

- **Compute each node's indegree**
  - **What is the meaning of the indegree of a vertex?**
    - If the number is 0?
    - If the number is 1?
    - If the number is 2?



Topological Sorting: Method <1>

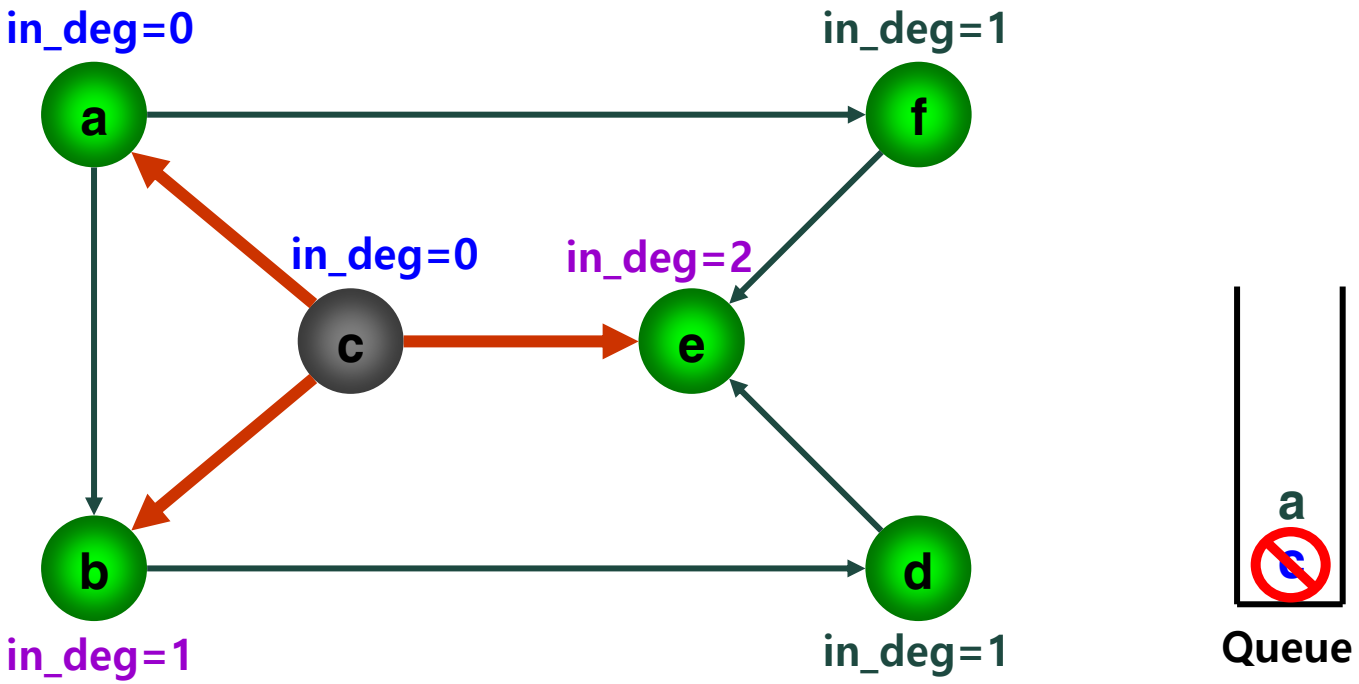
- Compute each node's indegree
  - Starting after storing the node 'c' into Queue since c's indegree is '0'



Sorted: -

Topological Sorting: Method <2>

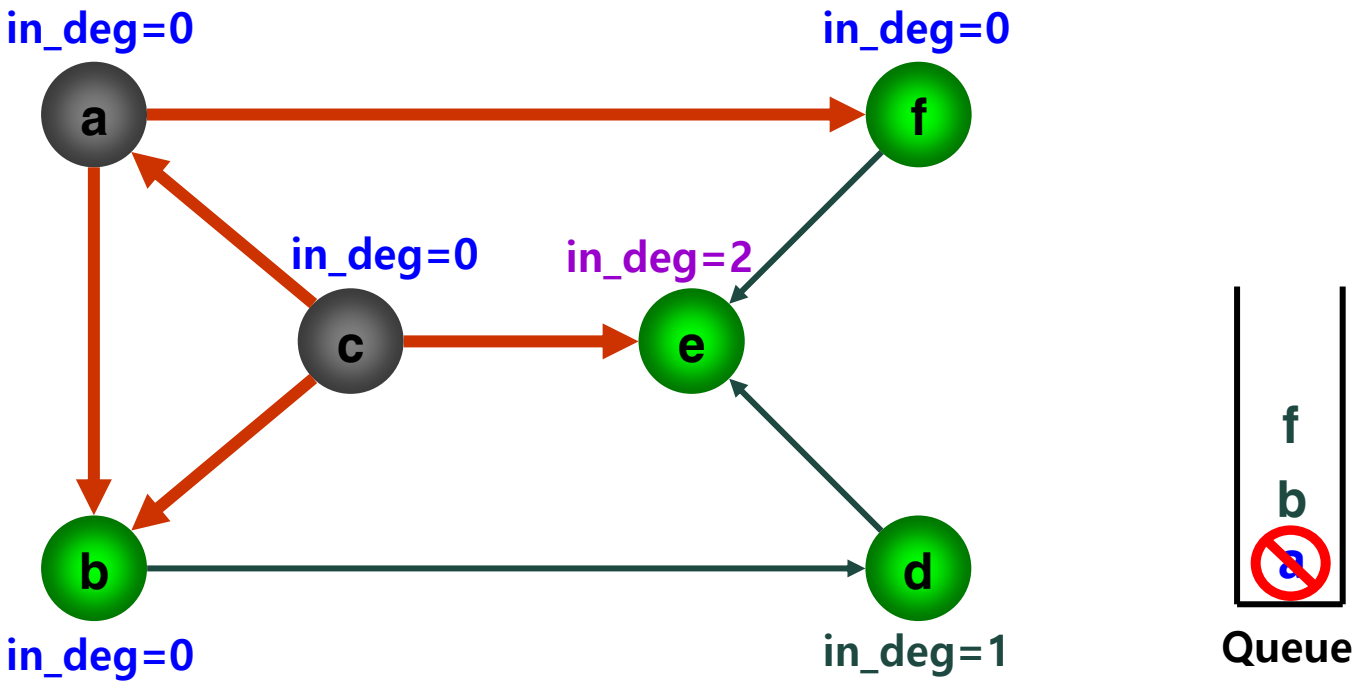
- While spanning from the node c, every indegree is decremented by 1
  - Since the node a's indegree is 0, 'a' is stored into Queue



Sorted: c

Topological Sorting: Method <3>

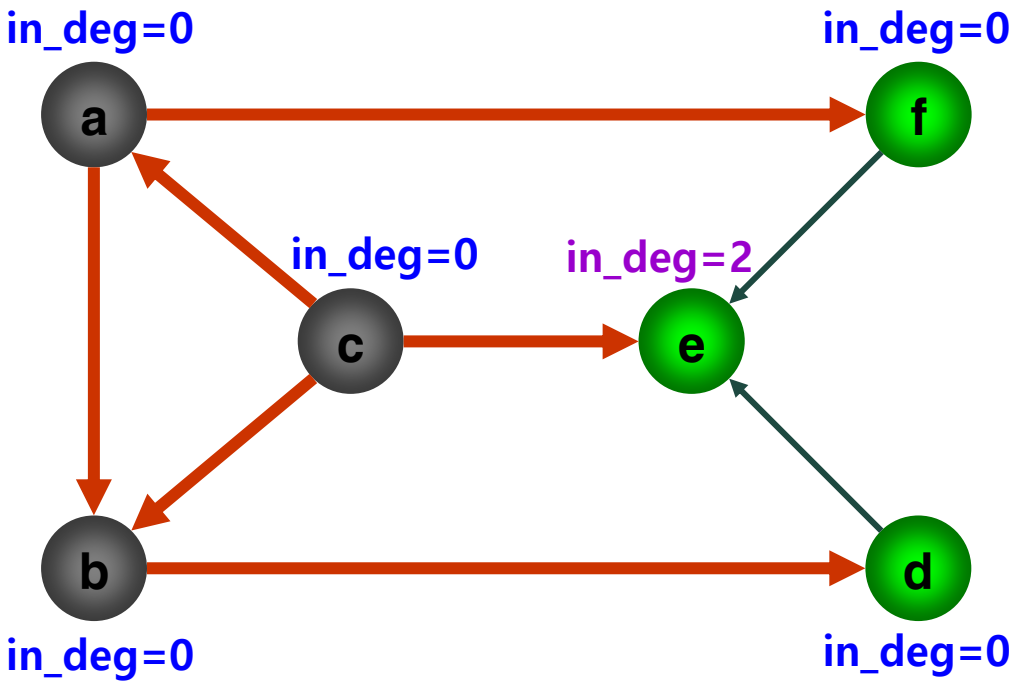
- While spanning from the node a, every indegree is decremented by 1
  - Since b's and f's indegree is 0, the nodes 'b' and 'f' are stored into Queue

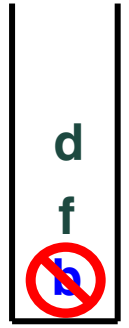


Sorted: c a

Topological Sorting: Method <4>

- While spanning from the ndoe b, every indegree is decremented by 1
  - Since the node d's indegree is 0, 'd' is stored into Queue



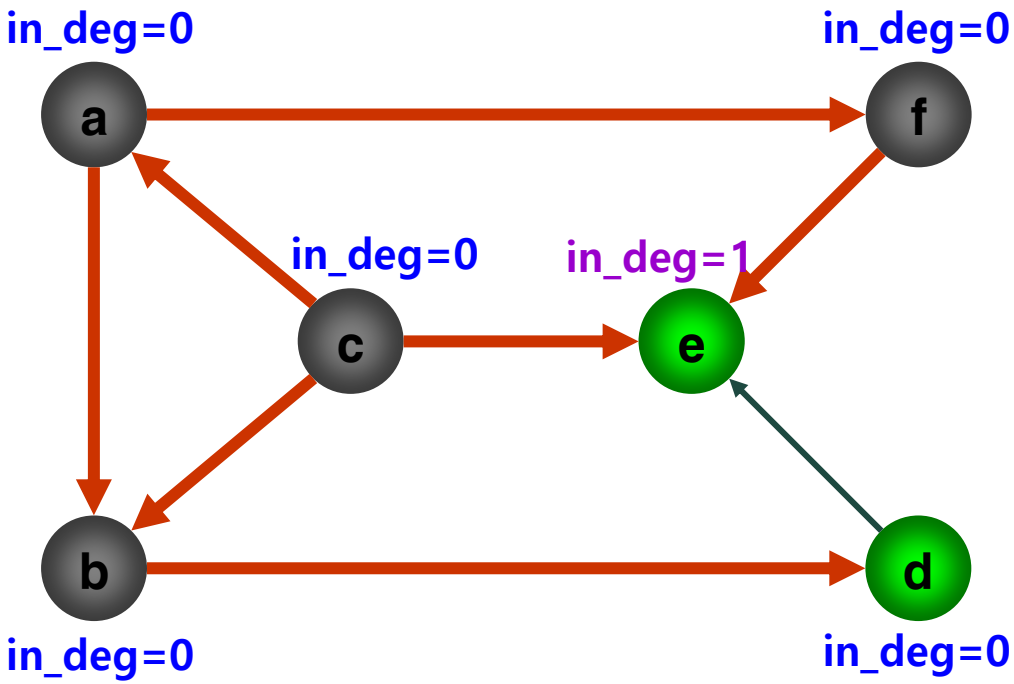


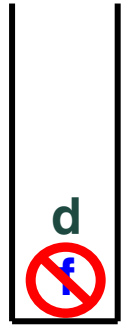
Queue

Sorted: c a b

Topological Sorting: Method <5>

- While spanning from the node f, every indegree is decremented by 1
  - There no node with 'zero' indegree



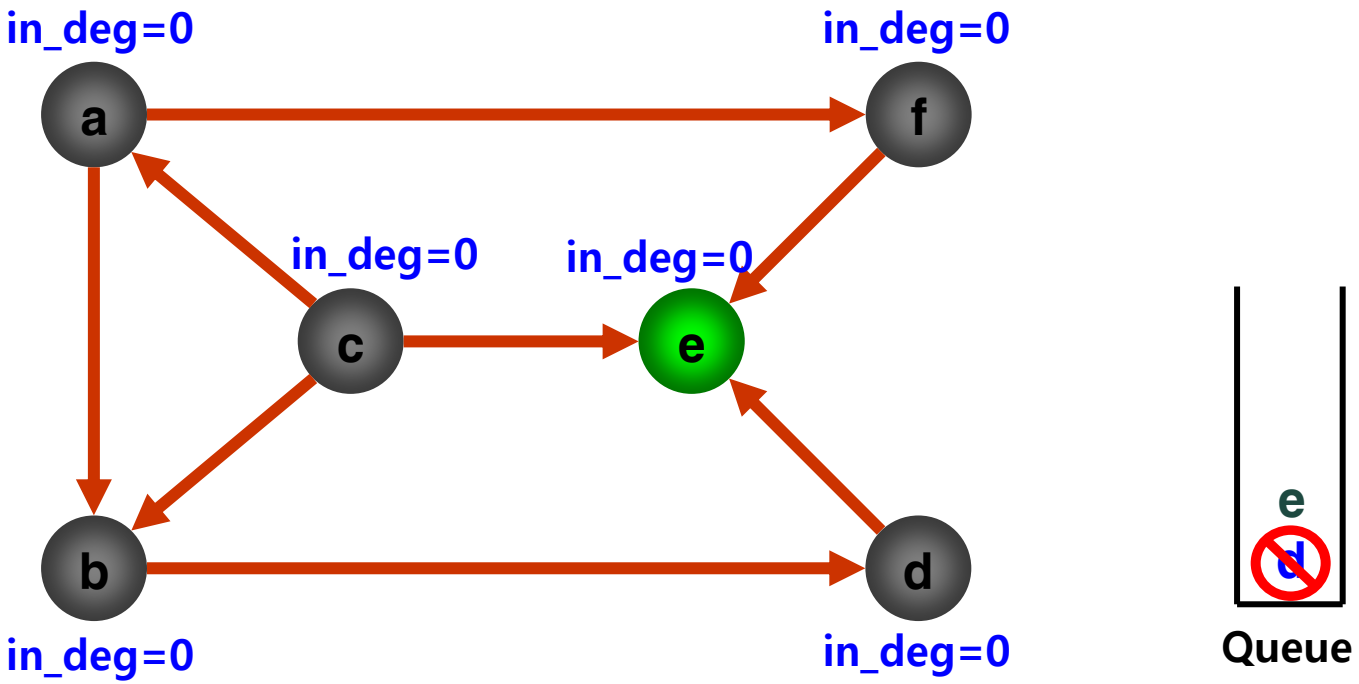


Queue

Sorted: c a b f

Topological Sorting: Method <6>

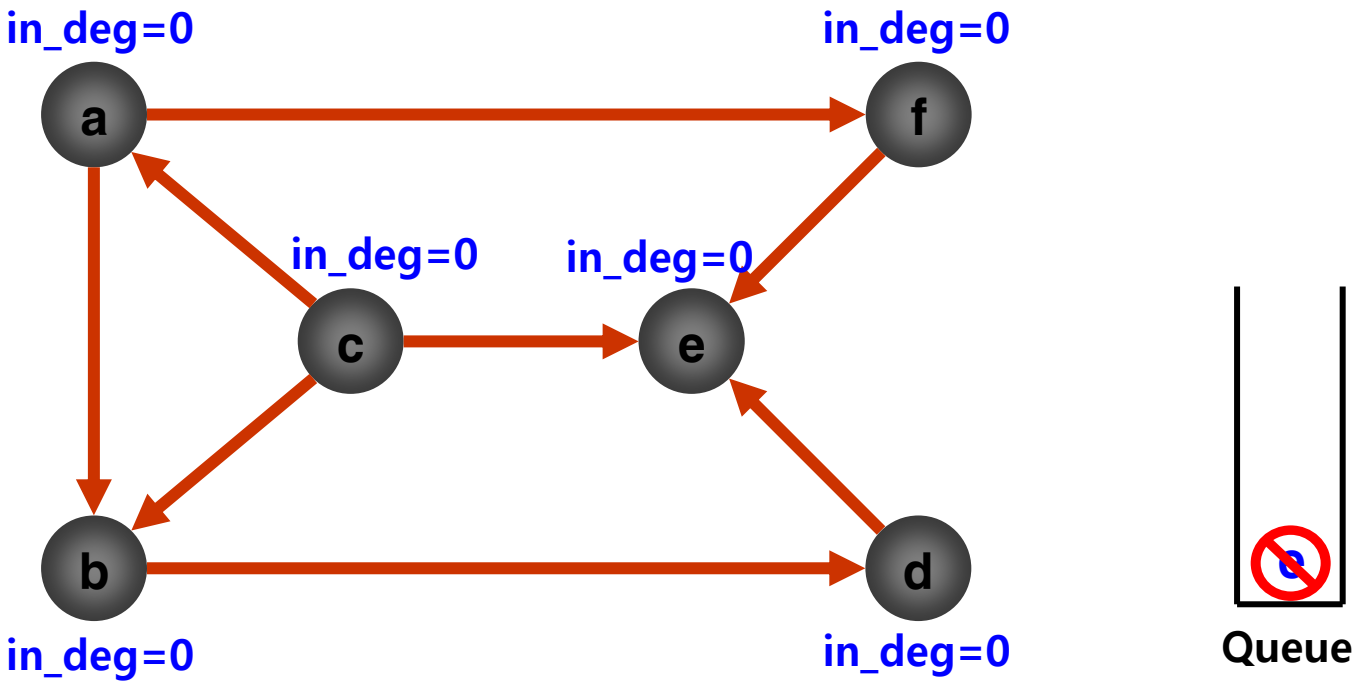
- While spanning from the node d, every indegree is decremented by 1
  - Since the node e's indegree is 0, 'e' is stored into Queue



Sorted: c a b f d

Topological Sorting: Method <7>

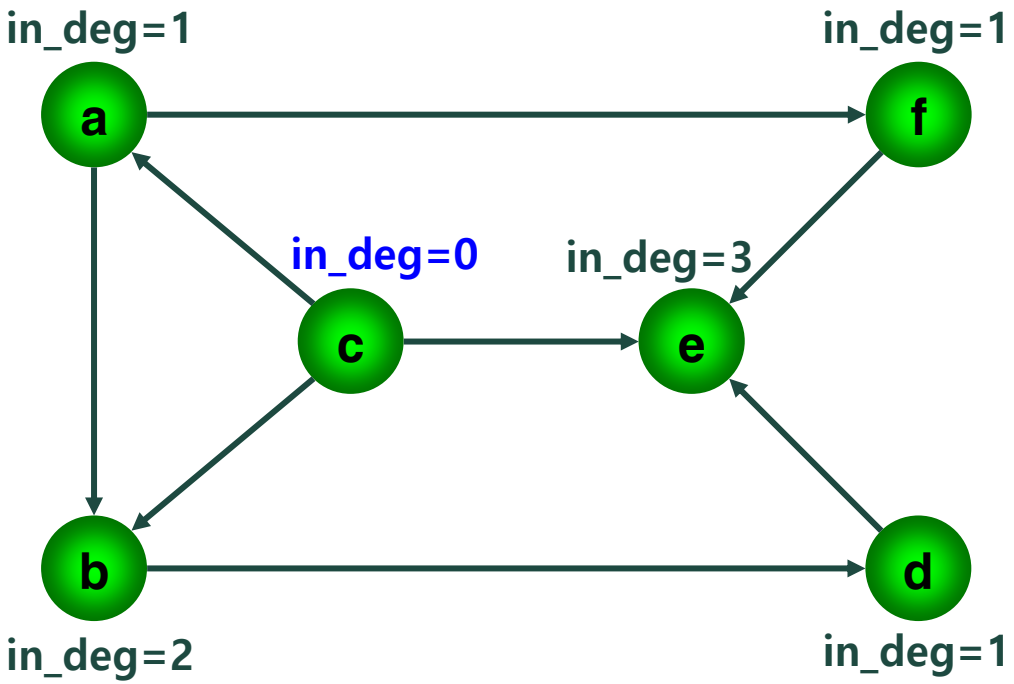
- While spanning from the node e, every indegree is decremented by 1
  - Since Queue is empty, the topological sorting is finished



Sorted: c a b f d e

Topological Sorting: Method <8>

- Compute each node's indegree
  - Starting after storing the node 'c' into Queue since c's indegree is '0'



Sorted: c a b f d e

Sorted:

Topological Sorting: Coding <1>

- **Fundamental operation**
  - **A straightforward algorithm is based on an analysis of the indegrees of each vertex in a DAG**
    - If a vertex has no incoming edges (has in-degree '0'), we may safely placed first in topological order
    - Deleting its outgoing edges may create new in-degree '0' vertices
    - This process continues until all vertices are placed in the ordering

    ➔ If not, the graph contained a cycle (i.e., not a DAG)!

Topological Sorting: Coding <2>

❖ Coding Tips!

- First, compute the in-degrees of each vertex of the DAG
- Second, maintain the in-degree '0' vertices using a queue
- Third, take the first node from the queue and decrement the in-degrees of the nodes connected to the taken node
- The sorting result is stored in the indegree array
- If the queue is empty, stop!

Topological Sorting: Coding <4>

```
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] ] ++;
}
```

```
#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;
```

```
read_graph(graph *g, bool directed)
{
        int i;                          /* counter */
        int m;                          /* number of edges */
        int x, y;                       /* vertices in edge (x,y) */

        initialize_graph(g);

        scanf("%d %d",&(g->nvertices),&m);

        for (i=1; i<=m; i++) {
                scanf("%d %d",&x,&y);
                insert_edge(g,x,y,directed);
        }
}
```

```
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 ++;
```

Topological Sorting: Coding <3>

```
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++) i
                        in[ g->edges[i][j] ] ++;
}
```

Starting from the indegree '0' node!

Until queue is empty, iterate the loop!

Topological Sorting: Coding <3>

```
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] ] ++;
}
```

Starting from the indegree '0' node!

Until queue is empty, iterate the loop!

Decrement the indegree of nodes connected with y!

When creating the indegree '0' node, store it into the queue!