Samsung Software Certification

Lecture 1: Introduction to Graph Algorithms

SPEL Technologies, Inc

Overview

- Fact or Fiction?
- Class Logistics
- Graph representations
- BFS/DFS
- Strongly connected components
- Kosaraju-Sharir algorithm
- Parallel Algorithms
- Applications

Fact or Fiction?

By Ehedaya at English Wikipedia (Own work (Original caption: “self-made”)) [Public domain], via Wikimedia Commons

An algorithm to solve the burnt pancake problem:

Flip each stack of pancakes to arrange them from smallest to largest with burned sides facing down quickly.

Fact

A sorting algorithm that must also flip the data in the smallest number of flips.

https://en.wikipedia.org/wiki/Pancake_sorting

Fact or Fiction?

By Adrian Pingstone (Own work) [Public domain], via Wikimedia Commons

An algorithm that uses information from mass surveillance to predict who the next perpetrator or victim will be.

Fiction

Science fiction drama

PERSON OF
INTEREST

By Vilnisr (Own work) [Public domain], via Wikimedia Commons

Fact or Fiction?

By Myke2020 [Public domain], via Wikimedia Commons

An algorithm that predicts where burglaries and car break-ins will occur next.

Fact

- Software built using computer science and anthropological research at Santa Clara University and UCLA is being used in LA to predict crime.

  https://www.technologyreview.com/s/428354/la-cops-embrace-crime-predicting-algorithm/

Fact or Fiction?

By Ab5602 (Own work) [Public domain], via Wikimedia Commons

An algorithm that plays a perfect game of poker and uses bluffing in its game.

Fact

True - the algorithm can play a perfect game in which it “always wins in the long run”.

http://www.nature.com/news/game-theorists-crack-poker-1.16683

Fact or Fiction?

Q. How many programmers does it take to change a light bulb?

A. None. It's a hardware problem.

An algorithm that recommends jokes for your sense of humor.

Fact

No joke - Jester is UC Berkeley’s official jester.

http://eigentaste.berkeley.edu

Fact or Fiction?

An algorithm that writes a novel.

Fact

NaNoGenMo or National Novel Generation Month uses computer programs to generate novels.

https://www.theverge.com/2014/11/25/7276157/nanogenmo-robot-author-novel

Outline of topics

- C/C++ implementation and analysis of algorithms.
- Applications from various fields including logic synthesis and image matching.
- Research into new, improved versions of older algorithms.

Analysis
- Asymptotic notation
- Recurrences
- Time complexity and execution time

Data structures
- Elementary data structures (arrays, linked lists, stacks, queues)
- Disjoint-set linked-lists and forests
- Binomial and Fibonacci heaps

Graph algorithms
- Graph representations

Outline of topics

•BFS of directed and undirected graphs
•DFS of directed and undirected graphs (with and without recursion)

•Strongly connected components
  - Kosaraju-Sharir algorithm

•Articulation points, bridges, biconnected components
  - Tarjan's algorithm to find articulation points

•Minimum Spanning tree
  - Prim's algorithm
  - Kruskal's algorithm

•Shortest paths
  - Properties
  - Dijkstra's algorithm
  - Bellman Ford algorithm

Outline of topics

• Directed acyclic graphs

•All pairs shortest paths
- Floyd-Warshall algorithm
- Johnson's algorithm

•Flow networks

Linear programming

NP-Completeness

•Polynomial time verification

•Solving various problems (using DFS, branch and bound, approximation algorithms, etc.):
- Knapsack
- activity selection
- traveling salesman
- set-covering
- priority-first search
- clustering

Assessments

Assessments: Tests will take place on the following dates during tutorial sessions.

(08/15/2017 11 am) Test 1 to identify areas of focus
(09/26/2017 11 am to 1 pm) Test 2 (2 hours)
(10/31/2017 10 am to 1 pm) Test 3 (3 hours)

Grading: Assignments: 20%
Test 2 : 40%
Test 3: 40%

Internet access is not allowed for tests 2 and 3.

Textbooks

- Cormen, Thomas H., Charles Eric. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Cambridge, MA: MIT, 2014. Print

- Sedgewick, Robert, and Kevin Wayne. Algorithms. Boston: Addison-Wesley, 2016. Print.

- A list of research papers will be provided.

- The content for the training will be posted on SPEL Technologies' Lemma Learning Management System.

Policies

Policies: Submissions should be turned in by the due date and work turned in late is not graded. Attendance is required for all tests and makeup tests are only given with proof of emergency.

Integrity: Trainees are allowed to discuss assignments but must complete all graded work privately and not share their solutions. References must be cited in all submissions.

Attendance: Please sign your name in each class on the attendance sheet provided by Samsung.

Graph representations

1. Adjacency List

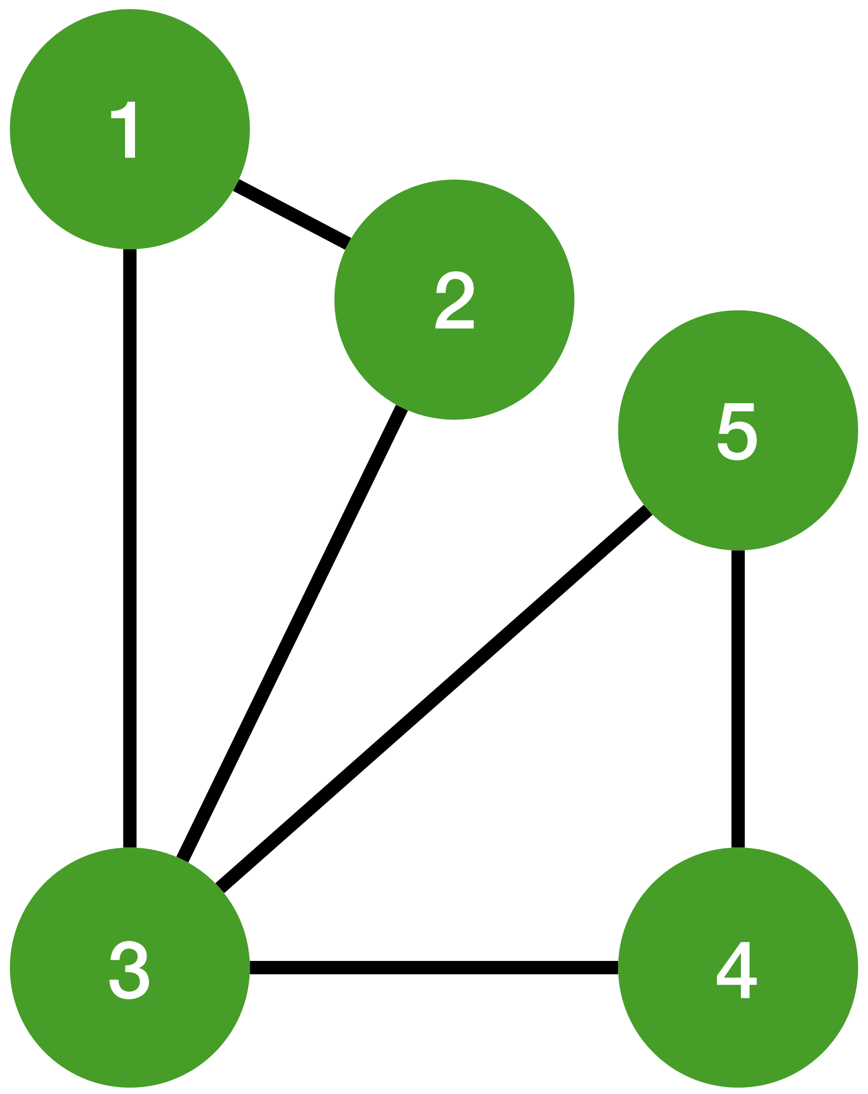
2. Adjacency Matrix

- Which representation is preferred?

    A. Dense Graph

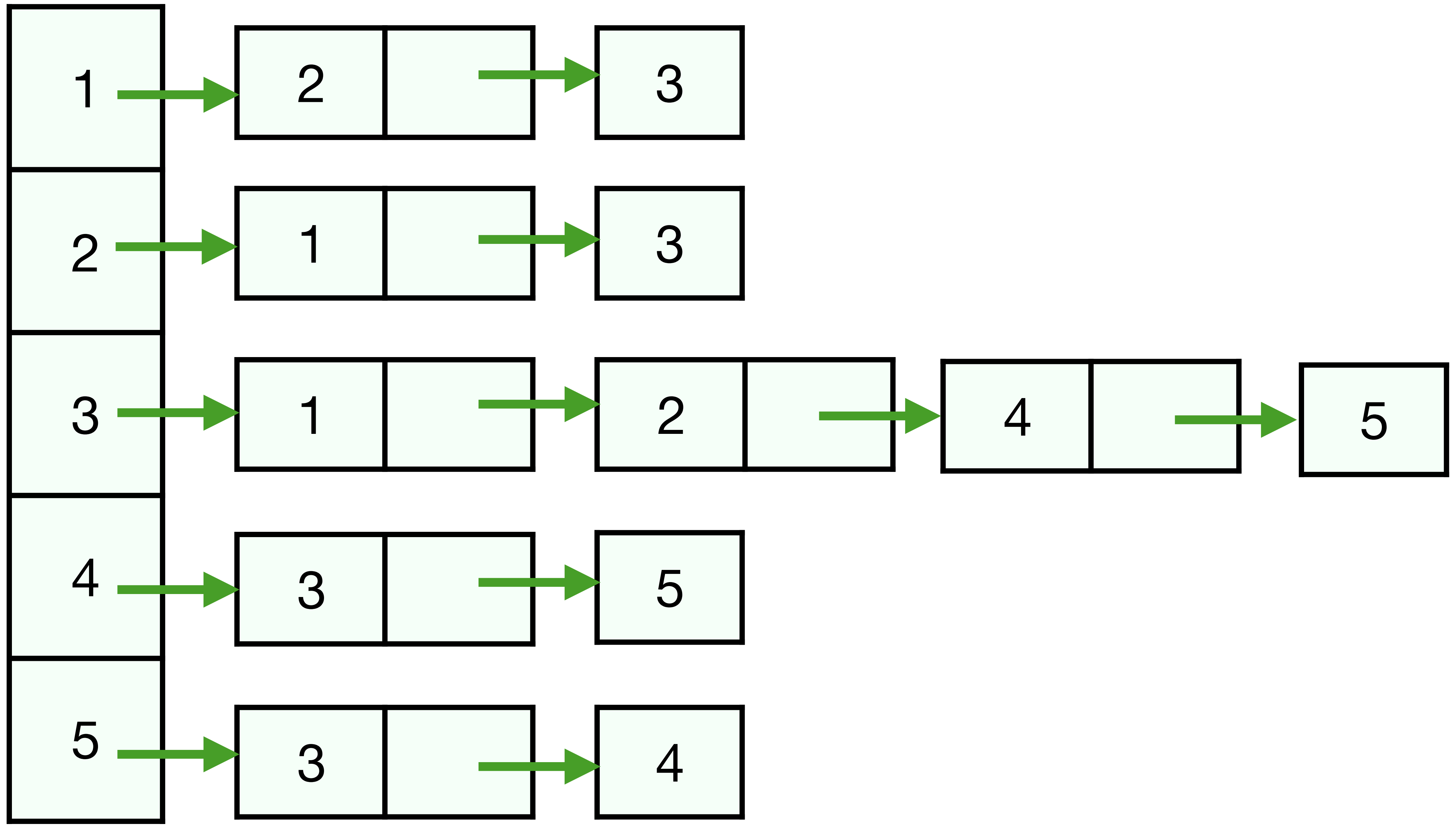
    B. Sparse Graph

Undirected graph



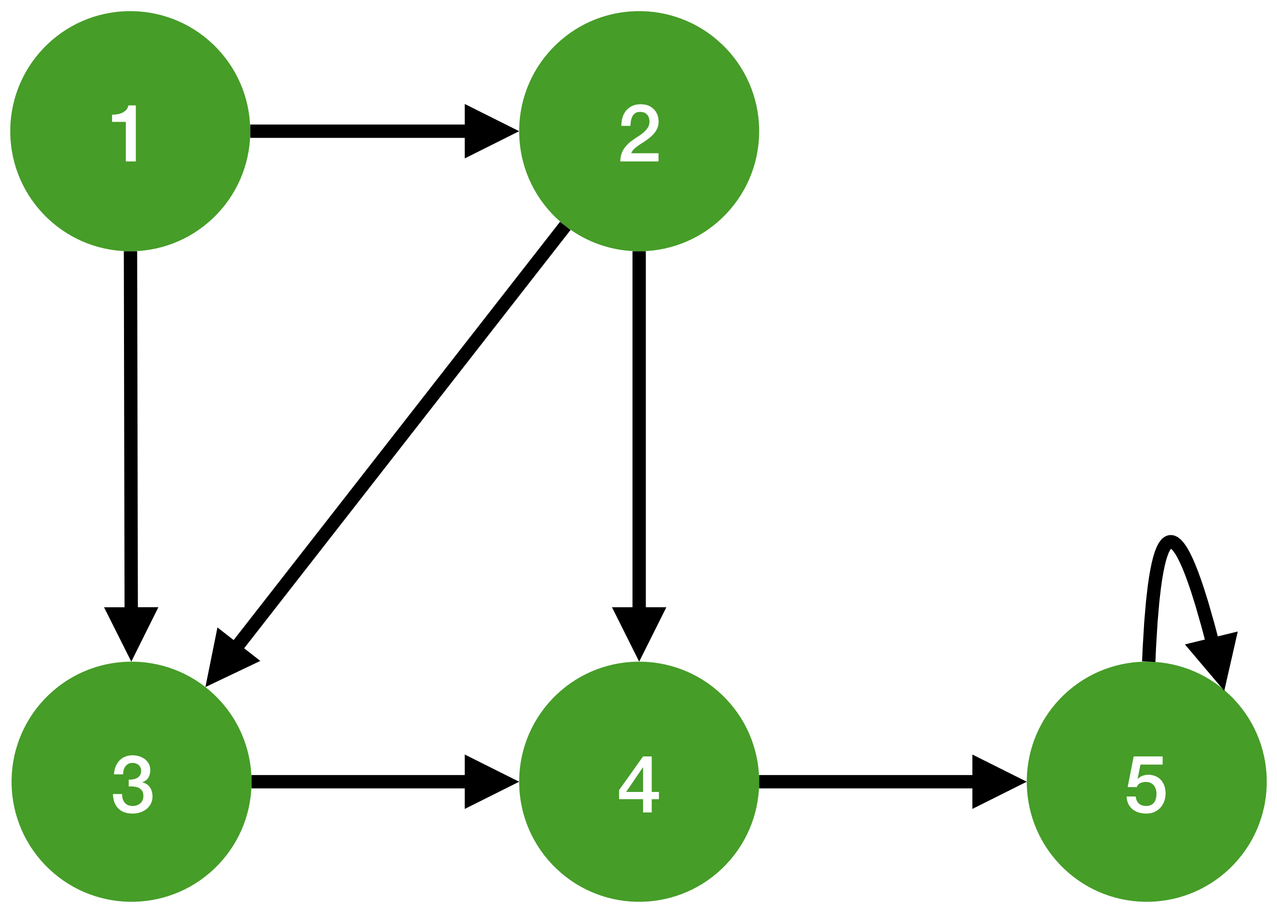
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	0	0
3	1	1	0	1	1
4	0	0	1	0	1
5	0	0	1	1	0

Adjacency Matrix



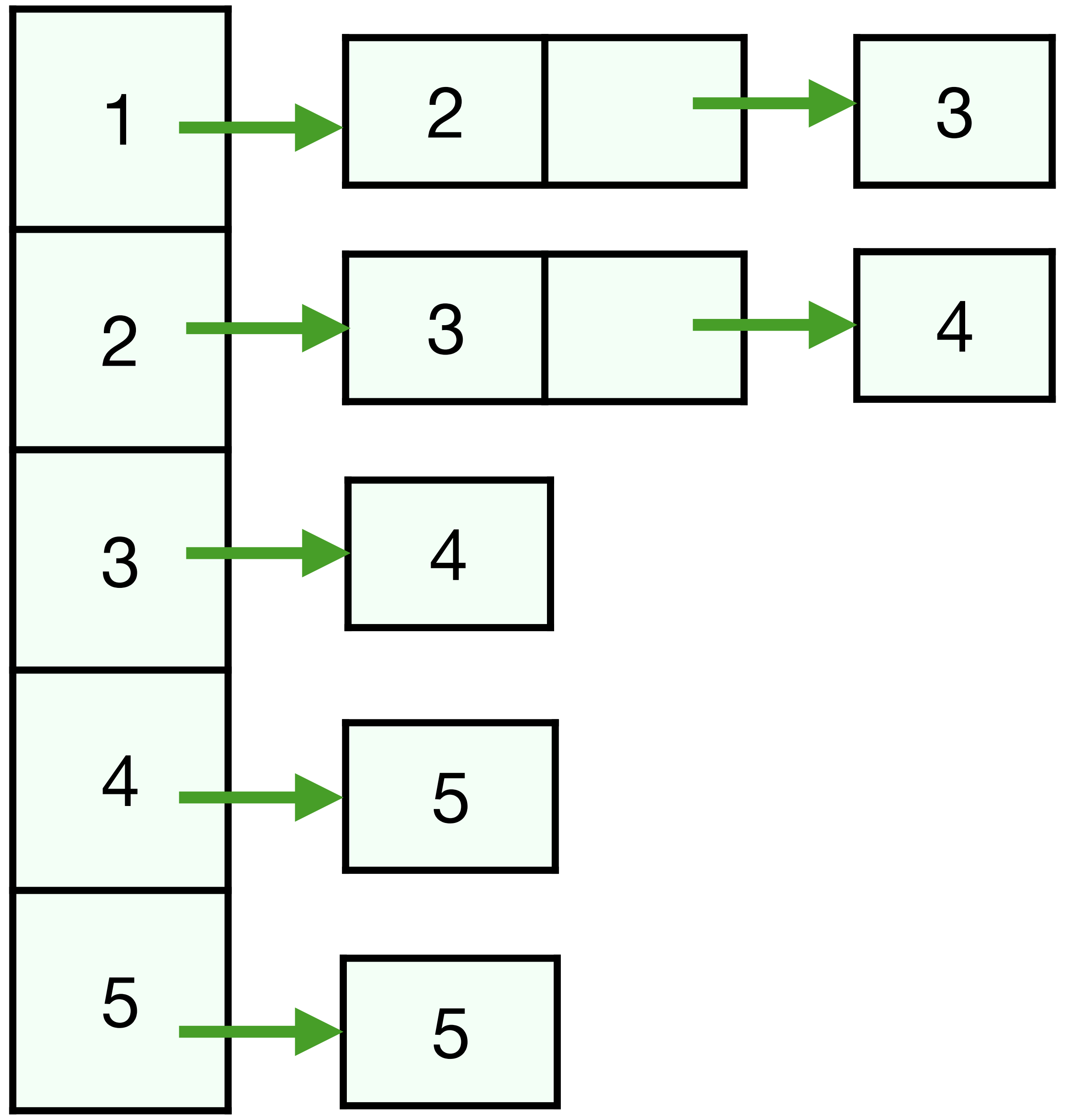
Adjacency List

Directed graph



	1	2	3	4	5
1	0	1	1	0	0
2	0	0	1	1	0
3	0	O	0	1	0
4	0	0	0	0	1
5	0	0	0	0	1

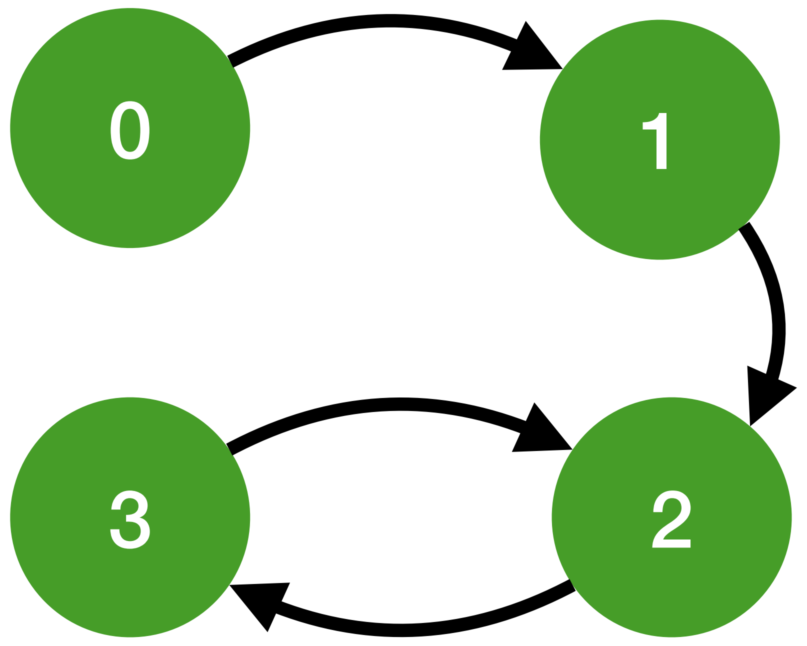
Adjacency Matrix



Adjacency List

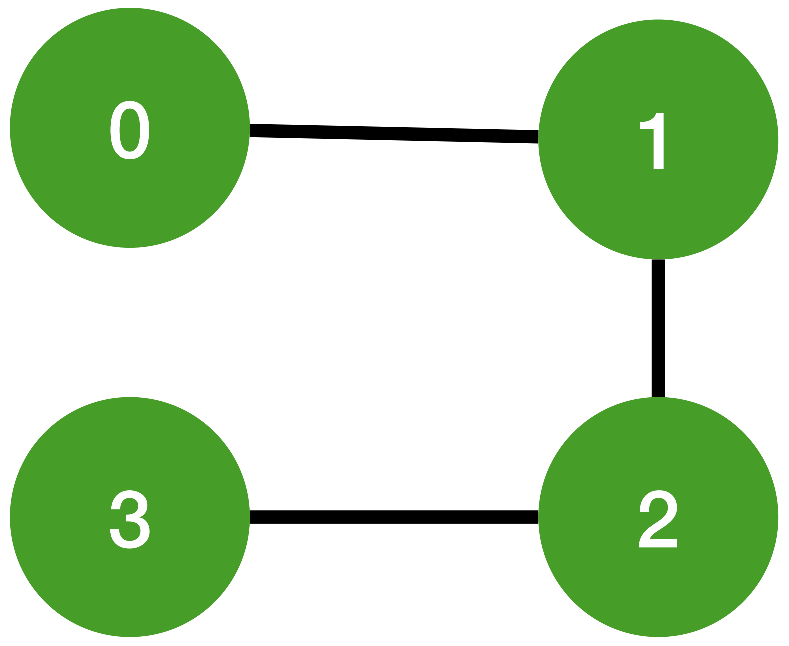
Directed graph

- The adjacency list and adjacency matrix can represent directed graph
- DFS and BFS can be used unchanged for both directed and undirected graph
- In a directed graph, we can only go from one vertex to another vertex using forward link



0 → 1 → 2 → 3

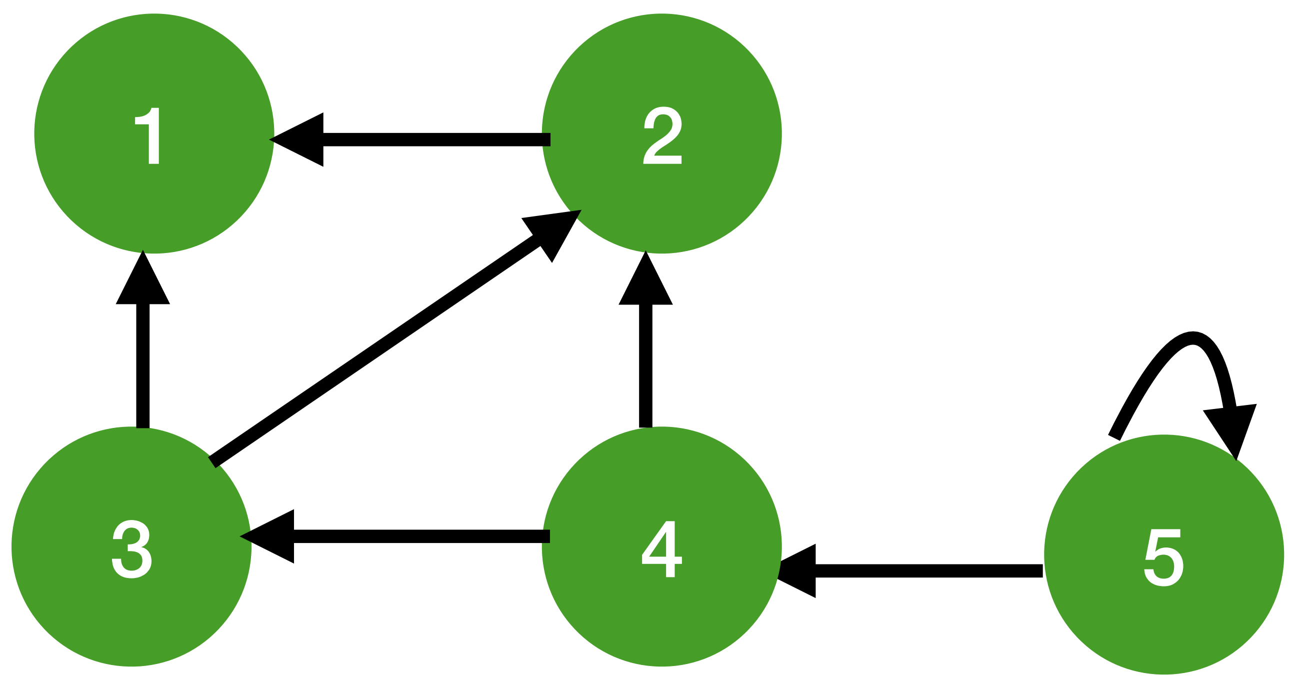
no path from 3 to 0

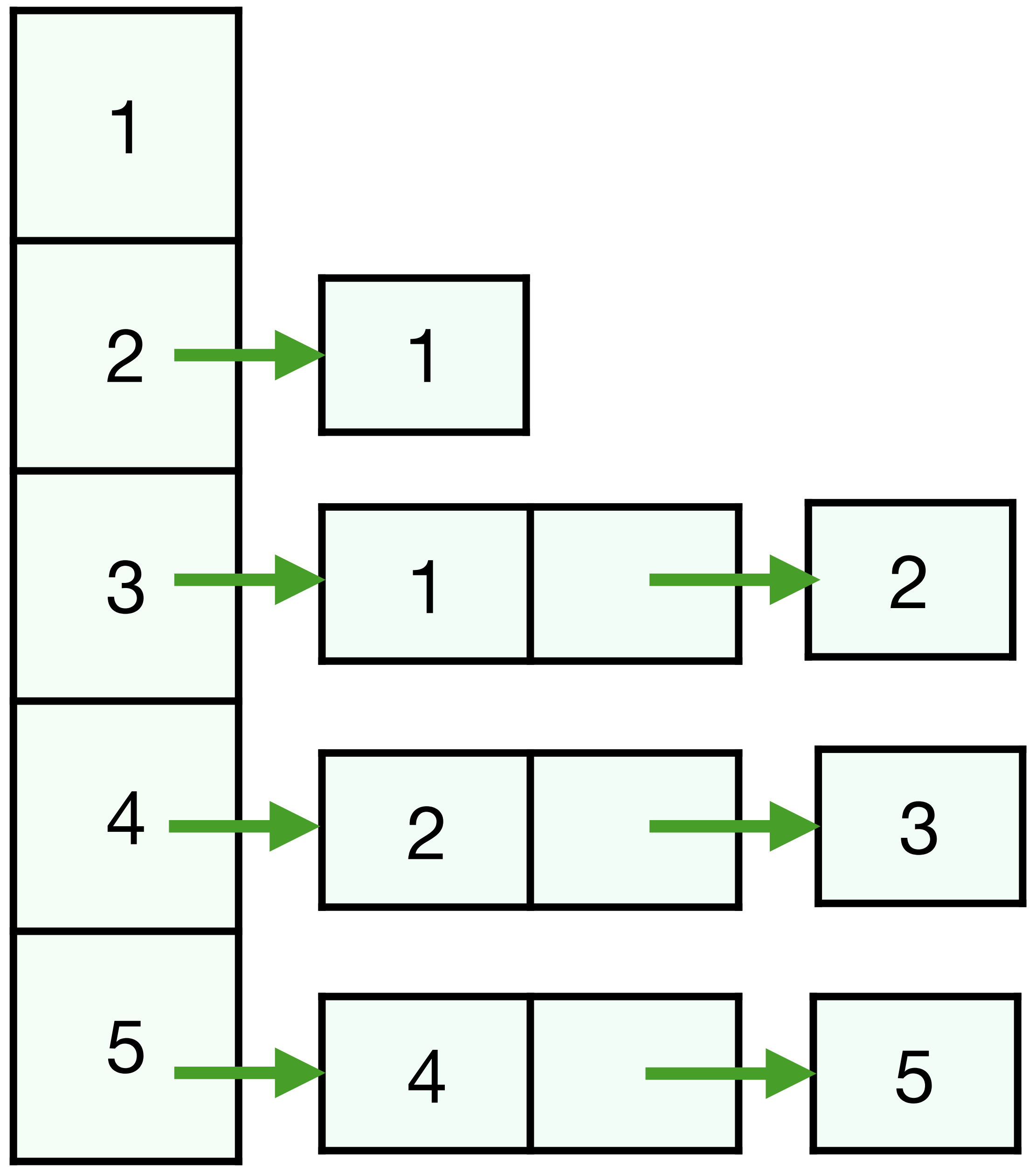


path from 0 to 3 and 3 to 0

Exercise

Create the adjacency list and matrix for the following graph:





Adjacency List

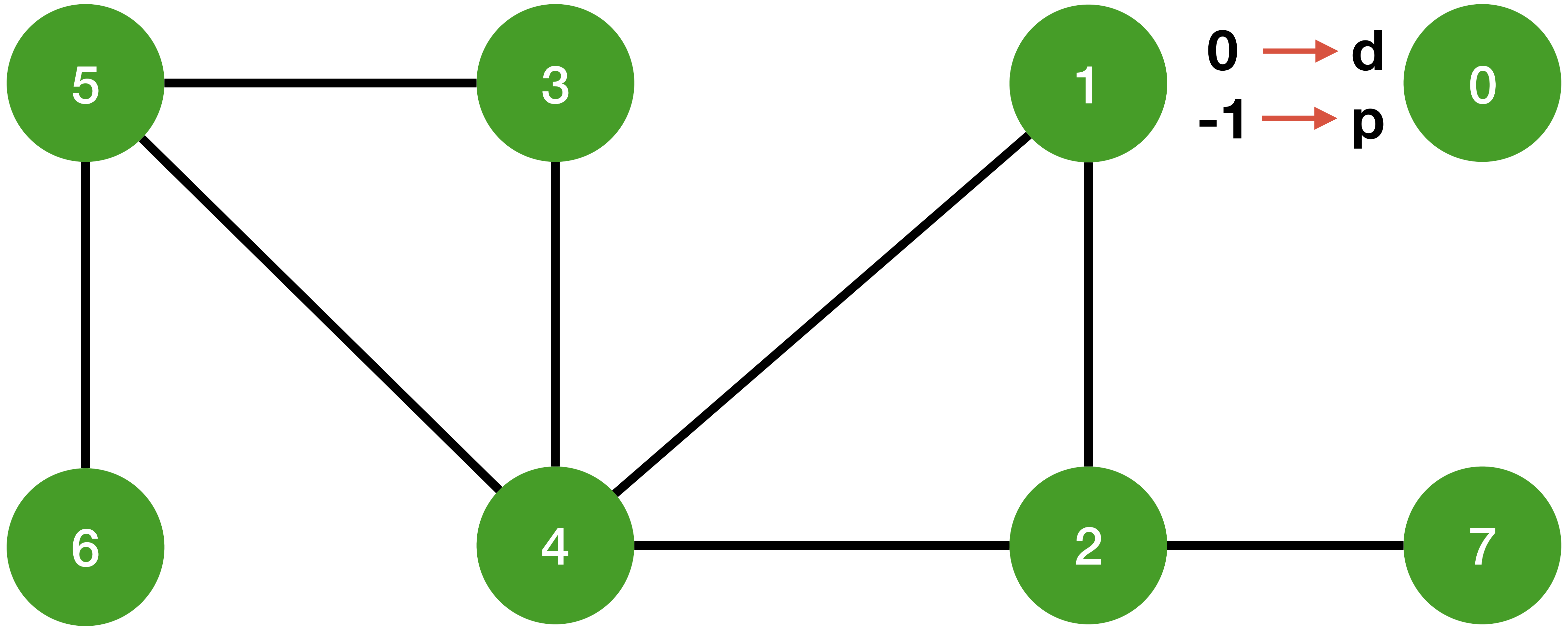
Breadth First Search (BFS)

Determines the shortest path from a source vertex to all other vertices

1. Start with a source vertex and find vertices v adjacent to s, for each vertex v, update :
    A. Distance d : minimum number of edge in any path from source vertex to vertex v
    B. Predecessor (Parent) p : closed vertex to v that is on the path from the source

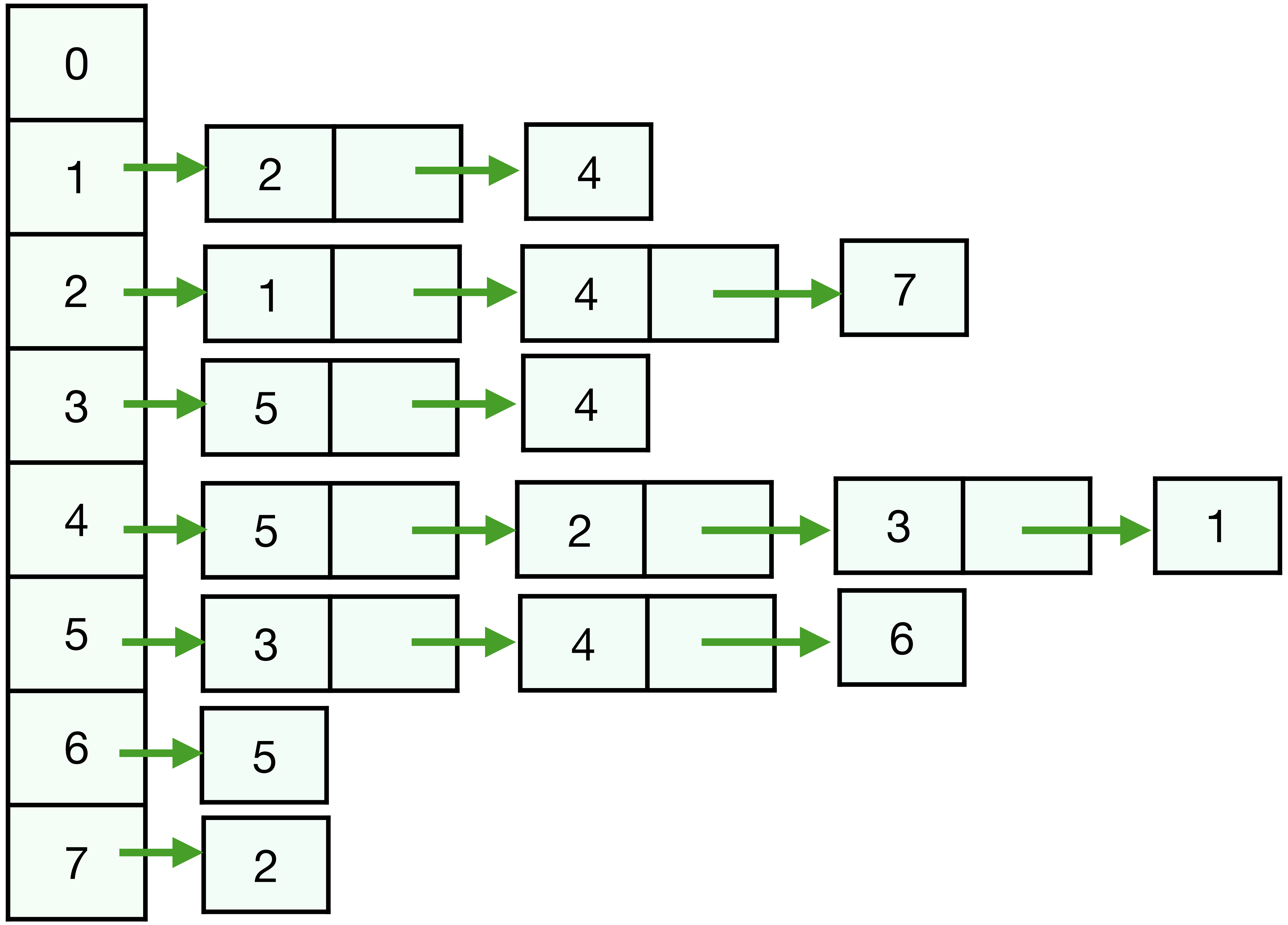
2. Treat each vertex v as a source vertex and repeat step one

BFS of undirected graph

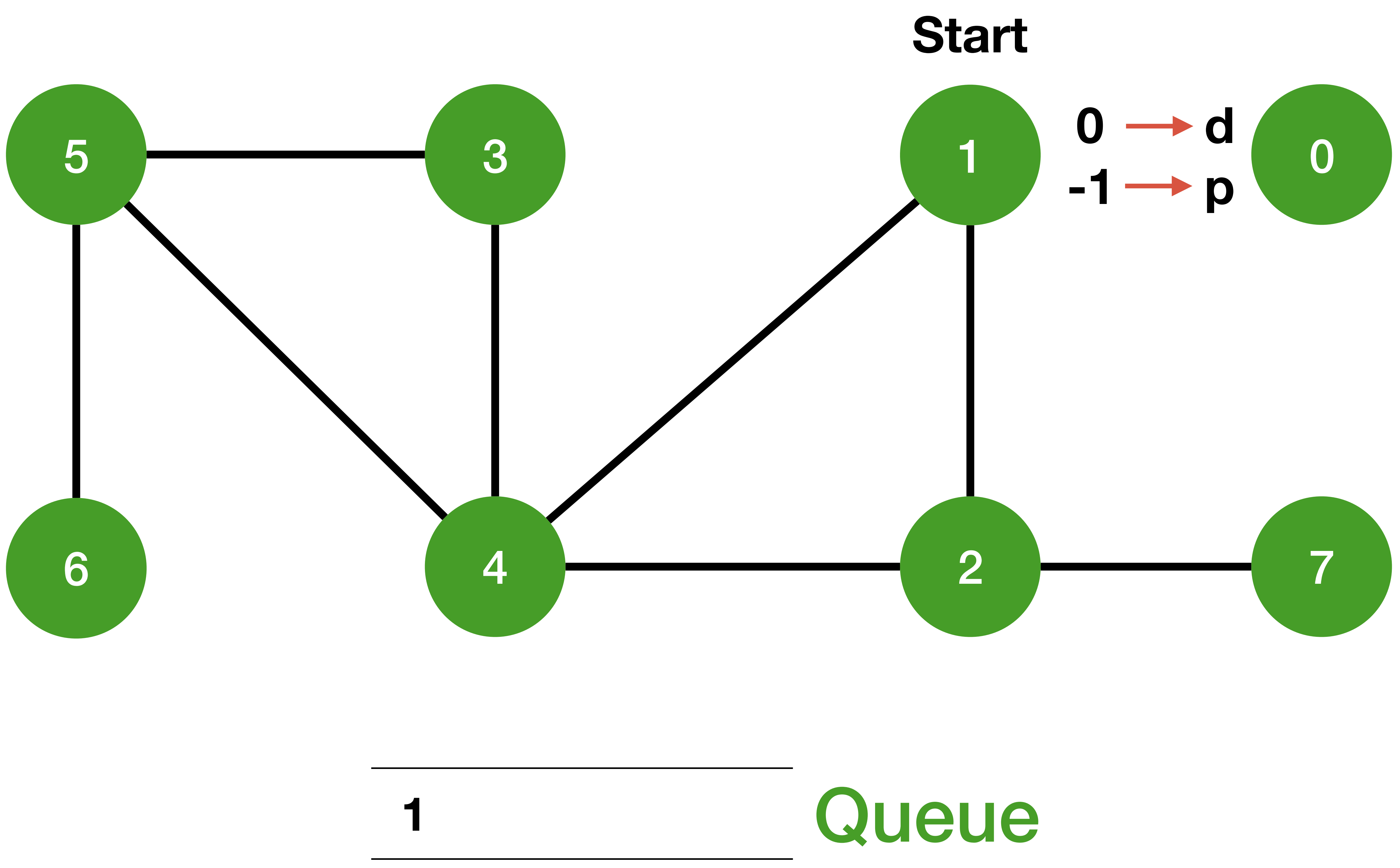


BFS implementation

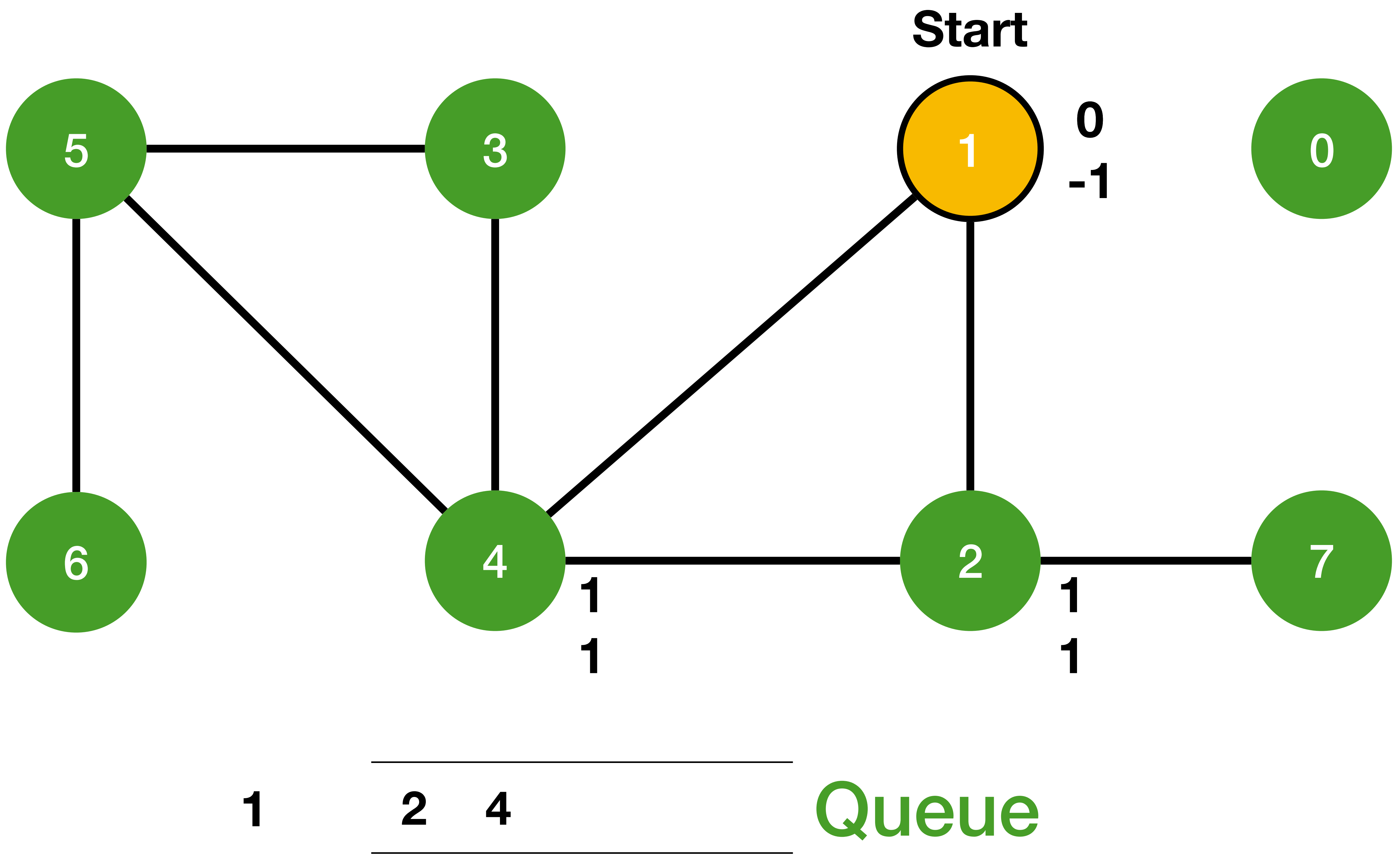
Using adjacency list to store graph



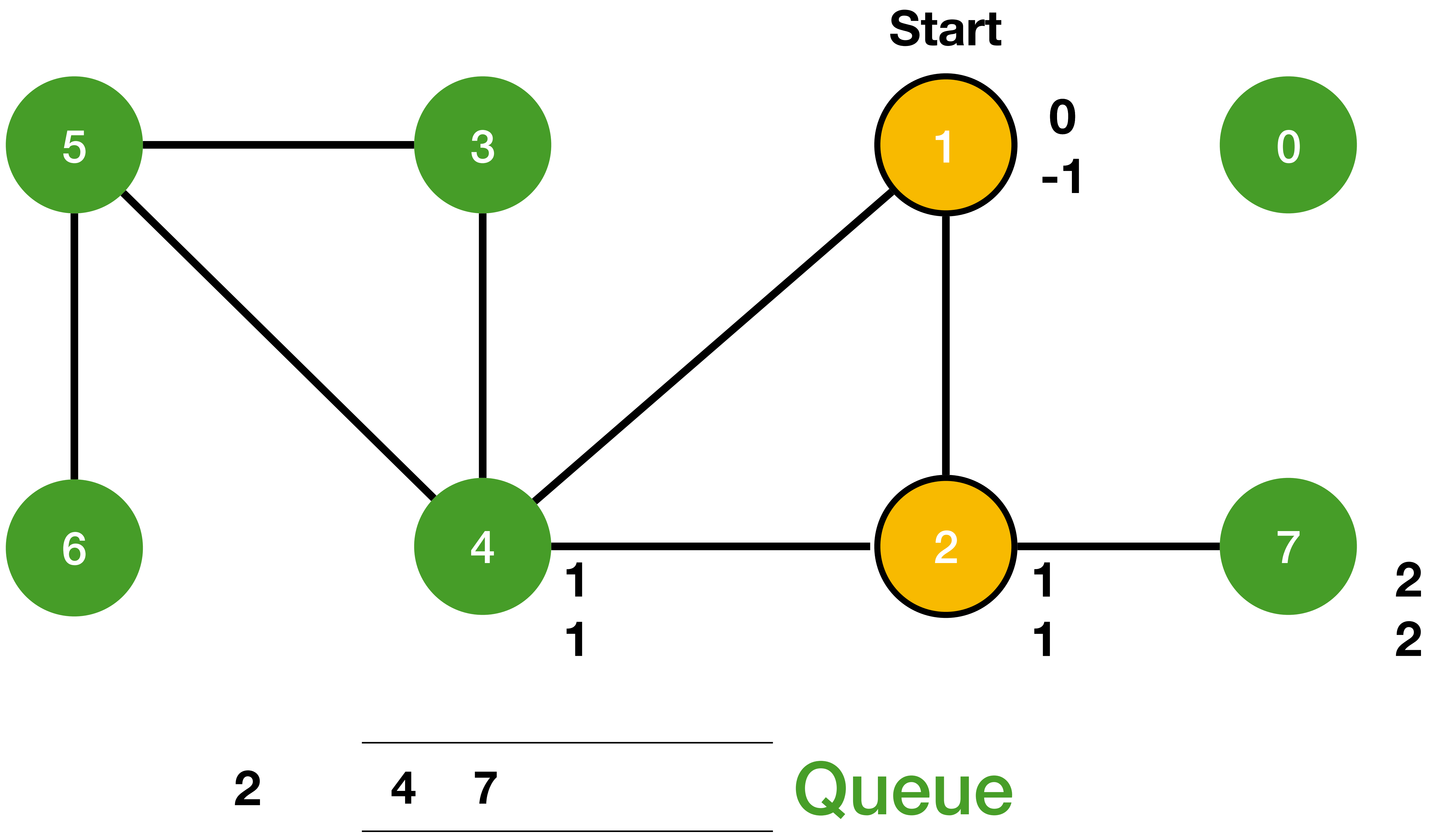
BFS of undirected graph



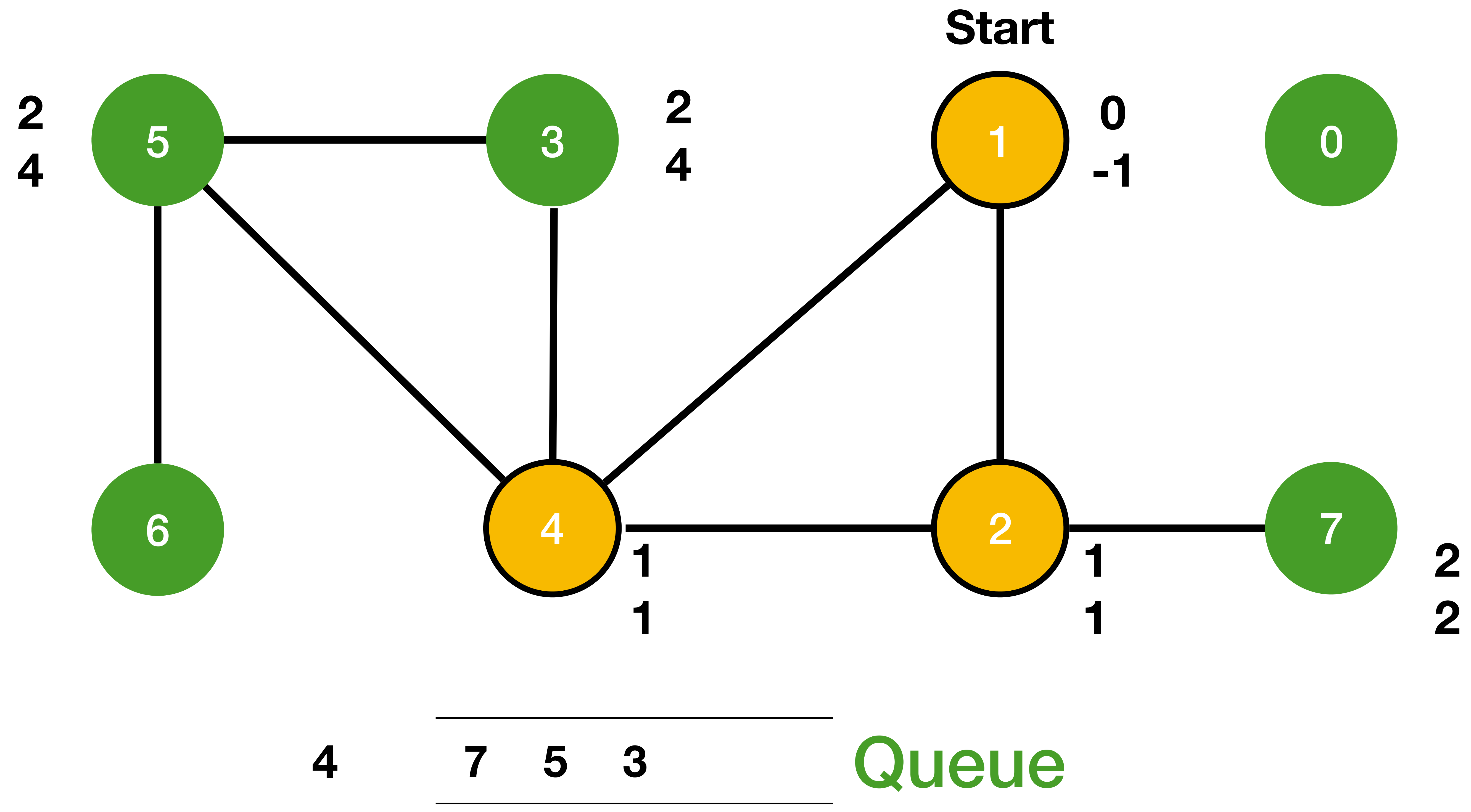
BFS of undirected graph



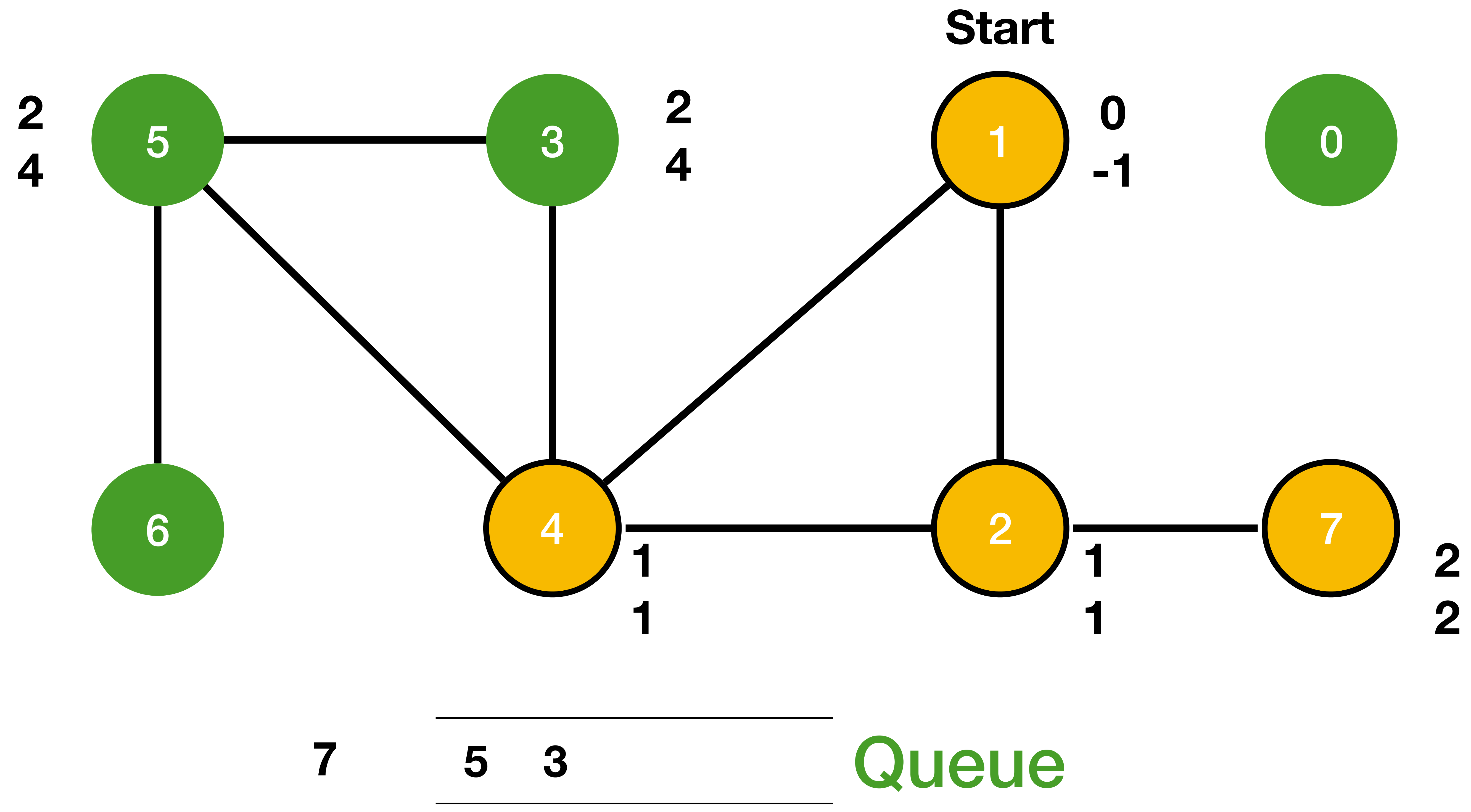
BFS of undirected graph



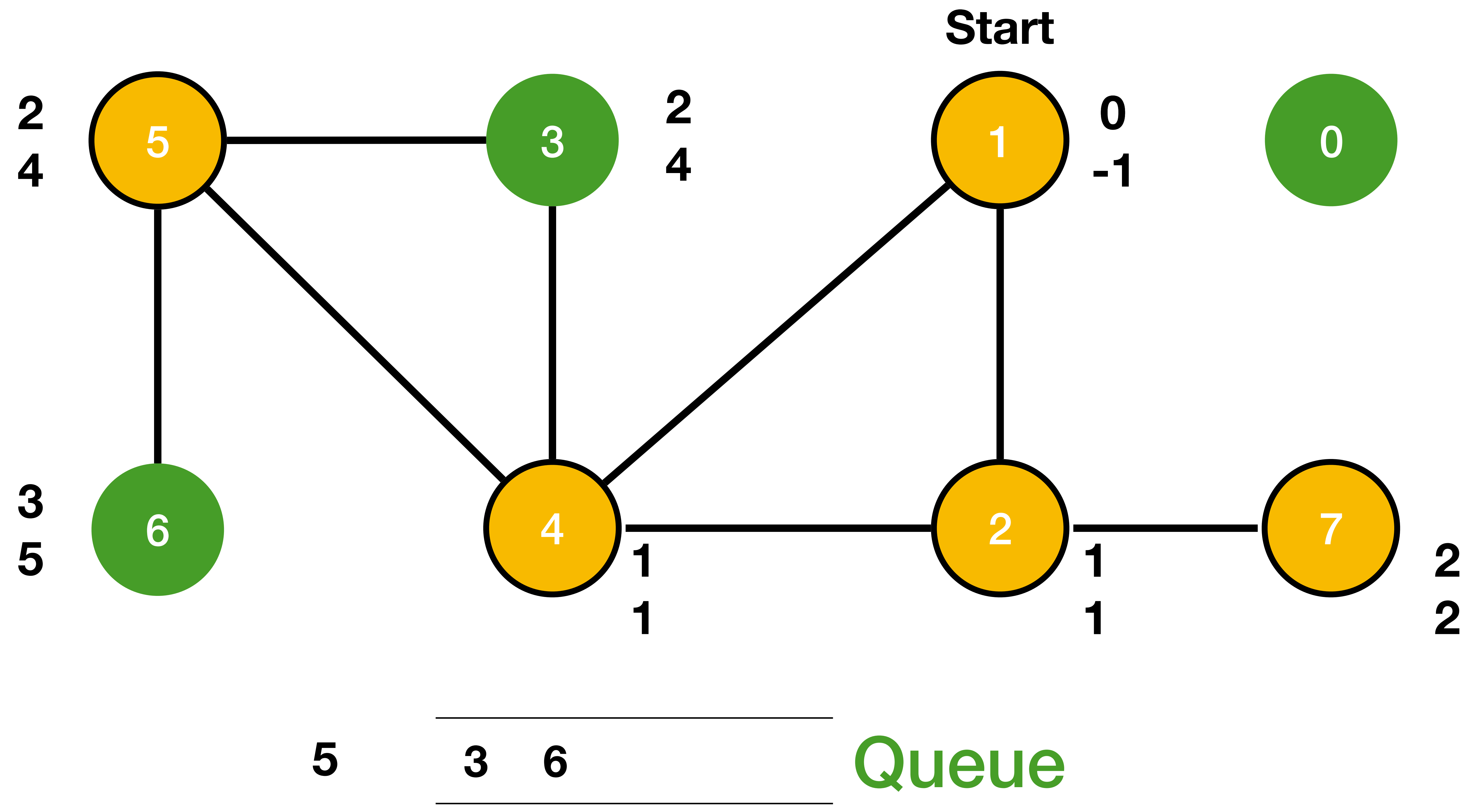
BFS of undirected graph



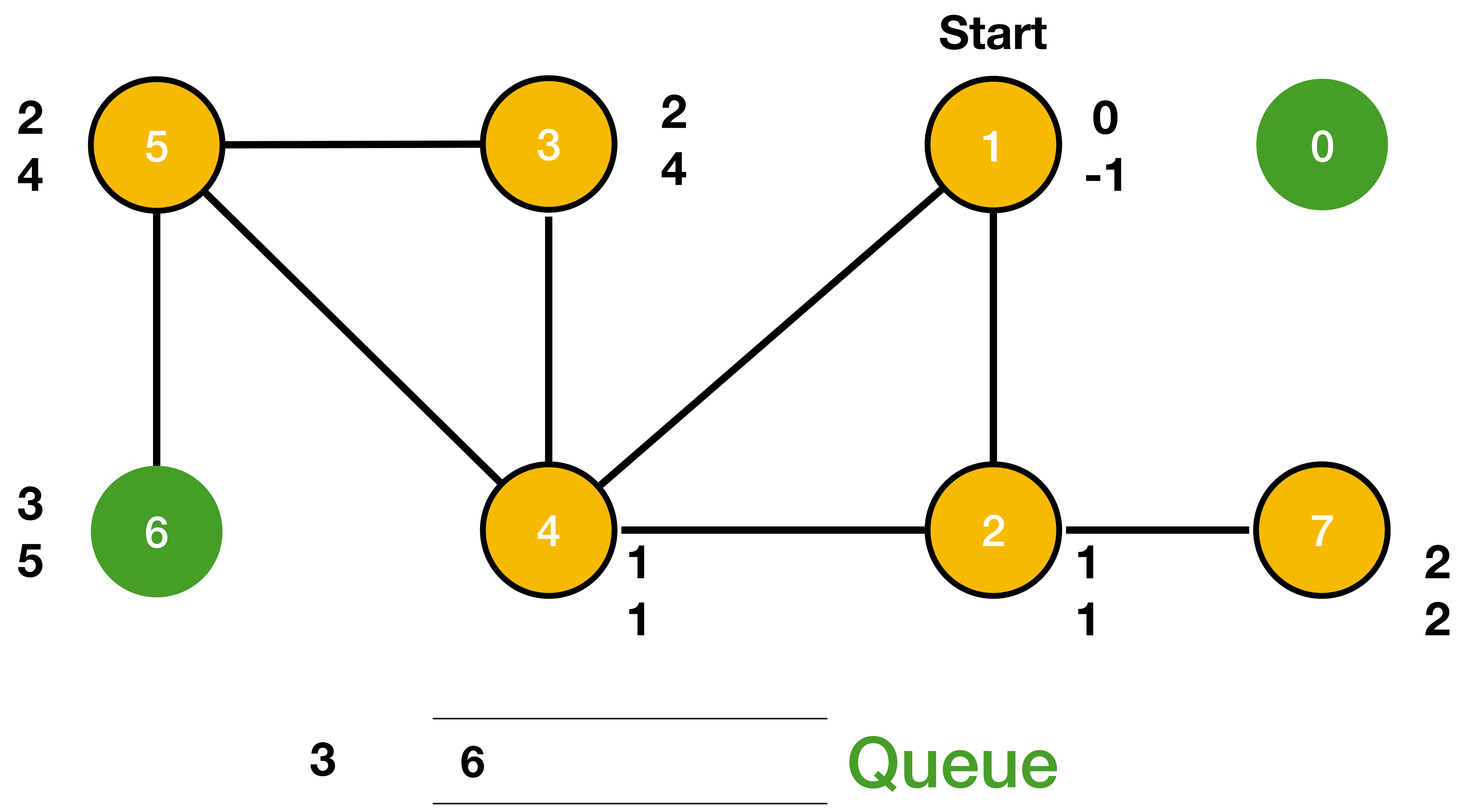
BFS of undirected graph



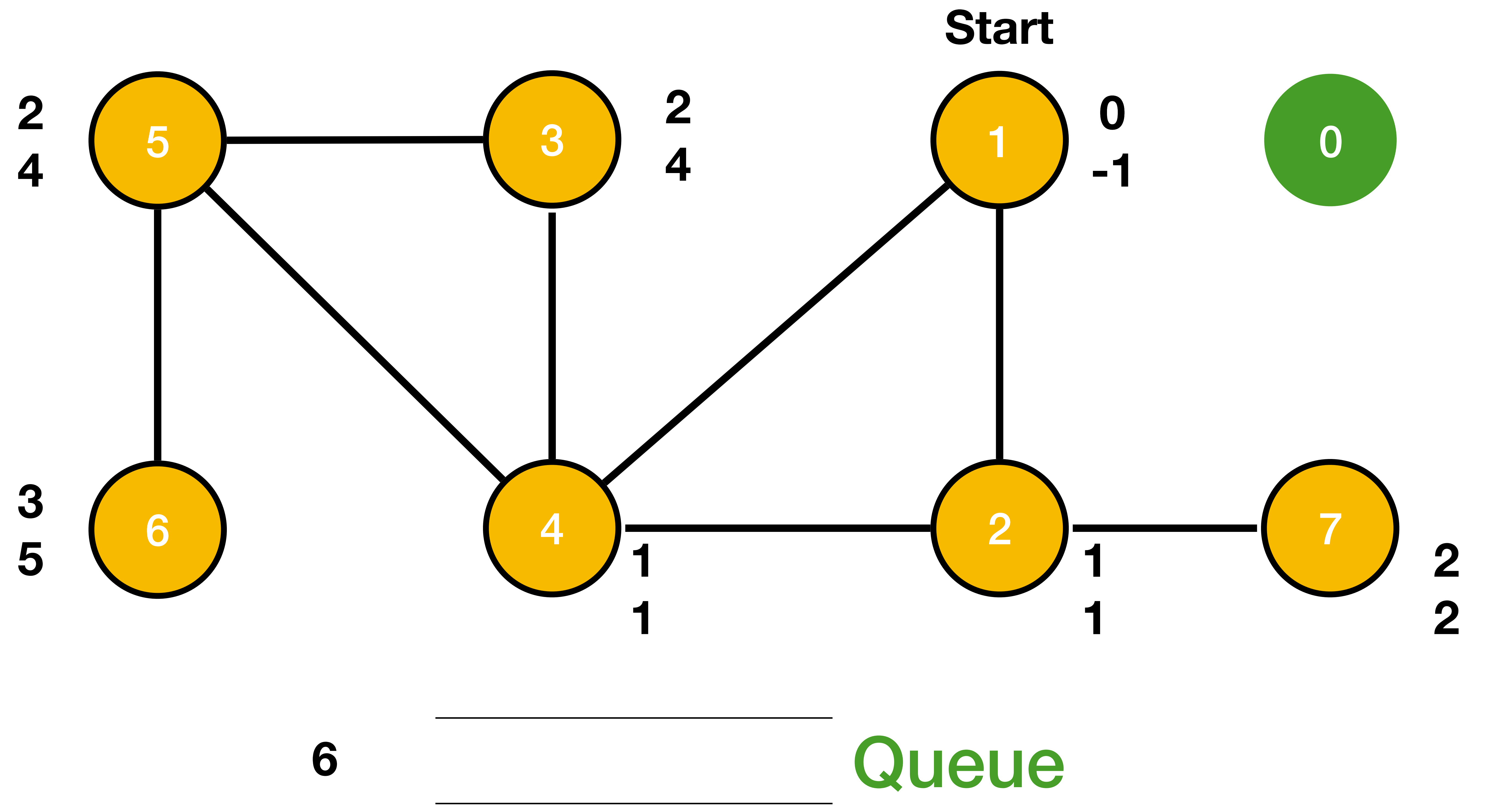
BFS of undirected graph



BFS of undirected graph



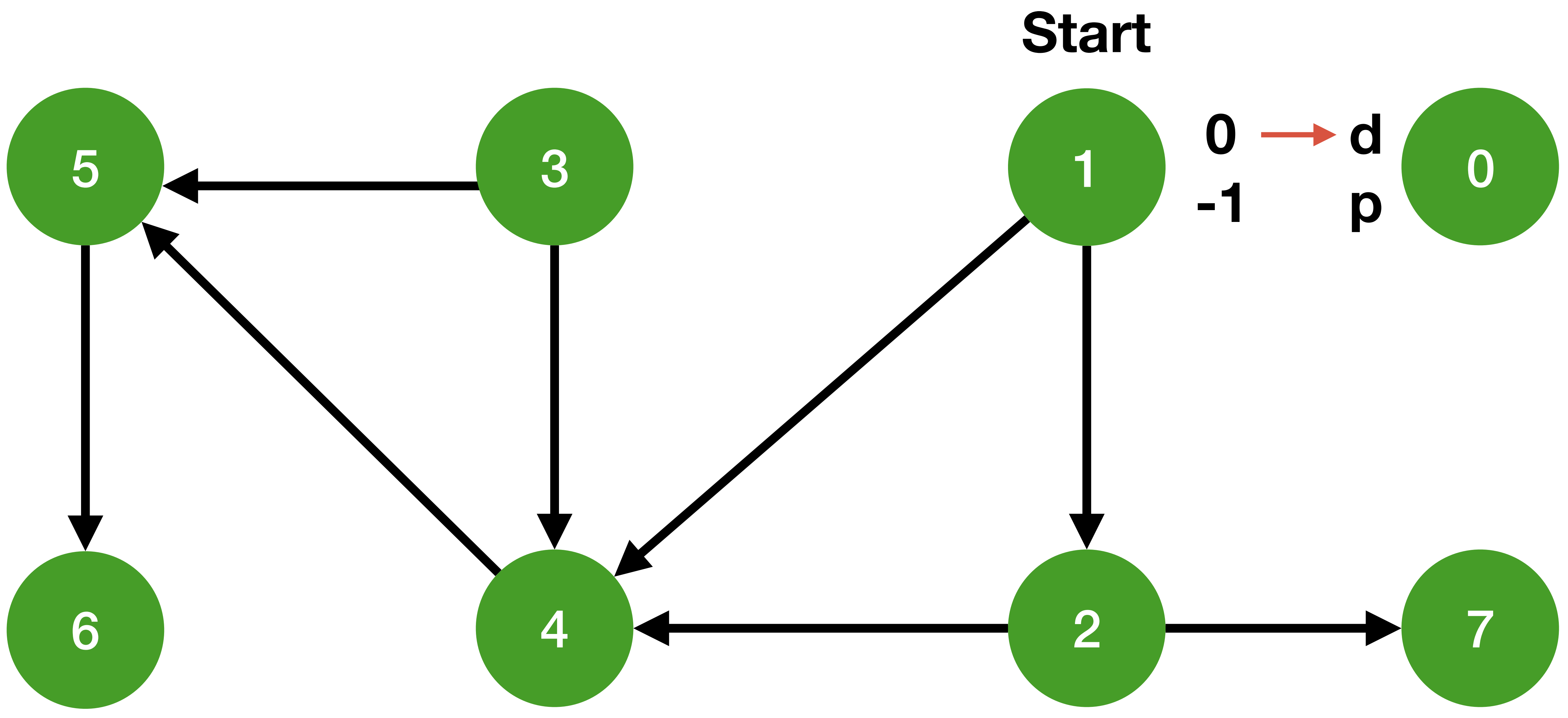
BFS of undirected graph



BFS algorithm

```
//s is the starting vertex
BFS(s){
      initialize array visited to zero;
      while (queue is not empty) {

        dequeue vertex s from queue ;

        for each vertex v adjacent to s that has not been visited {
             mark vertex v as visited ;
             distance(v) = distance(s) + 1;
             predecessor(v) = s;
             enqueue v;
        }
    }
}
```

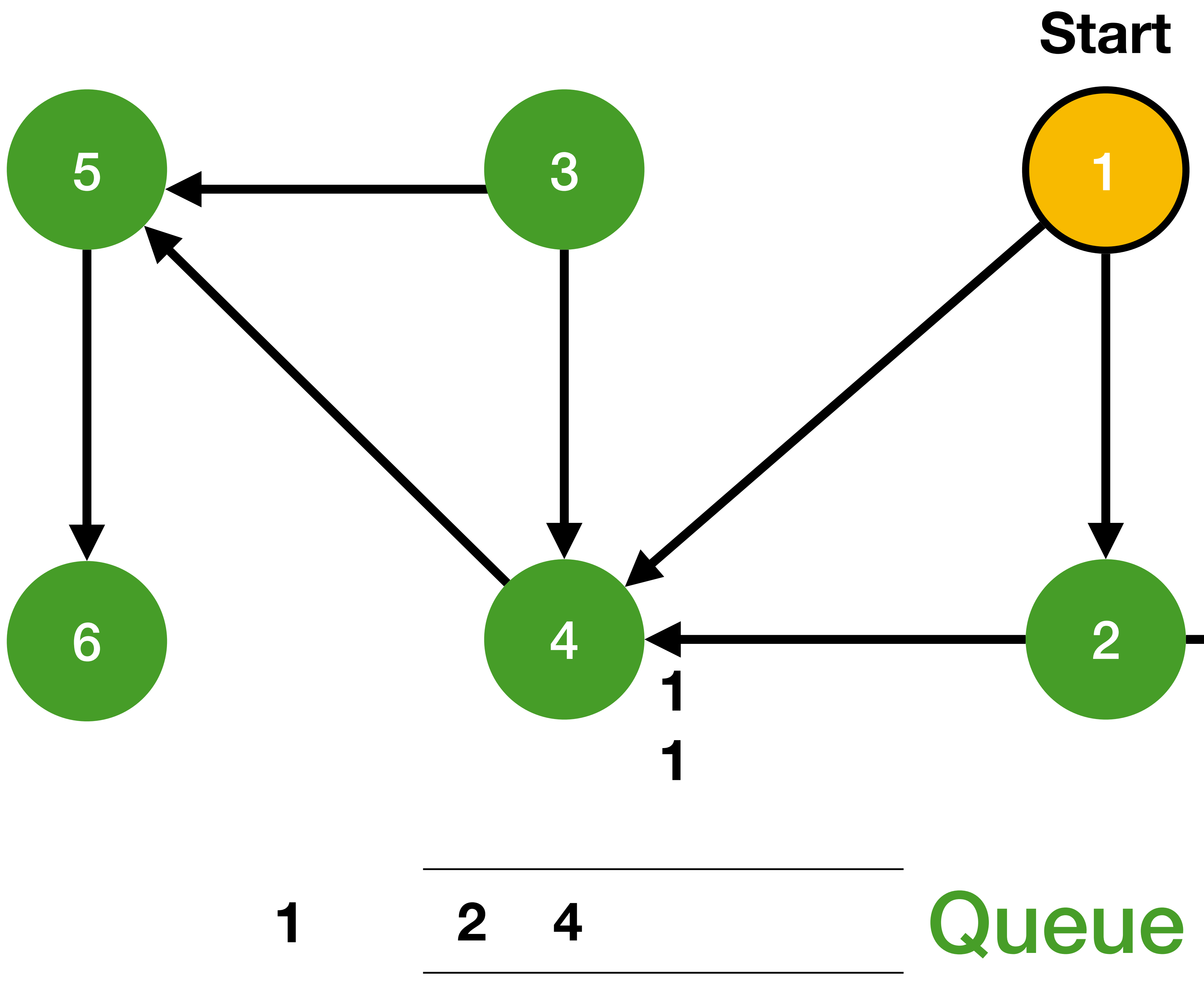
BFS of directed graph



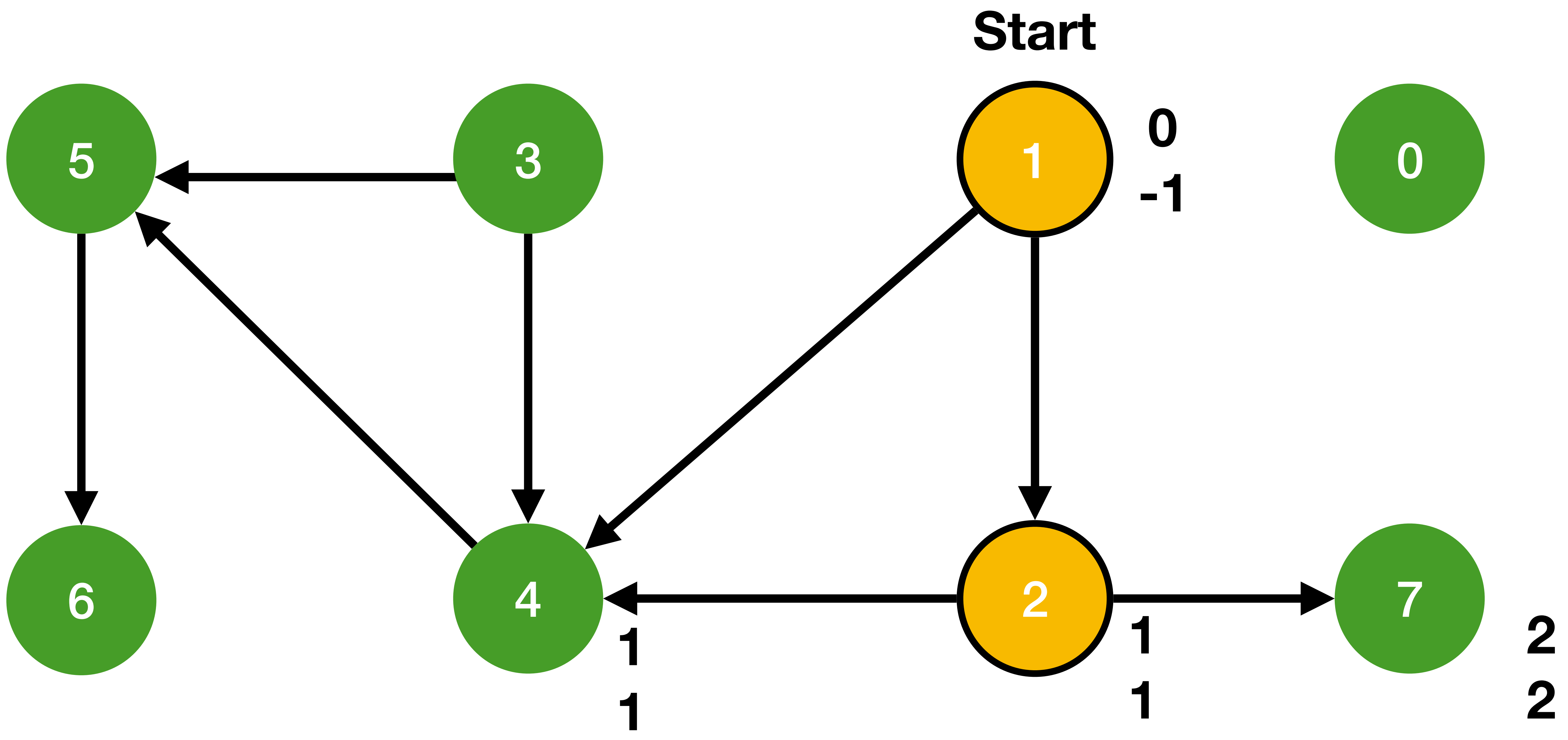
1

Queue

BFS of directed graph

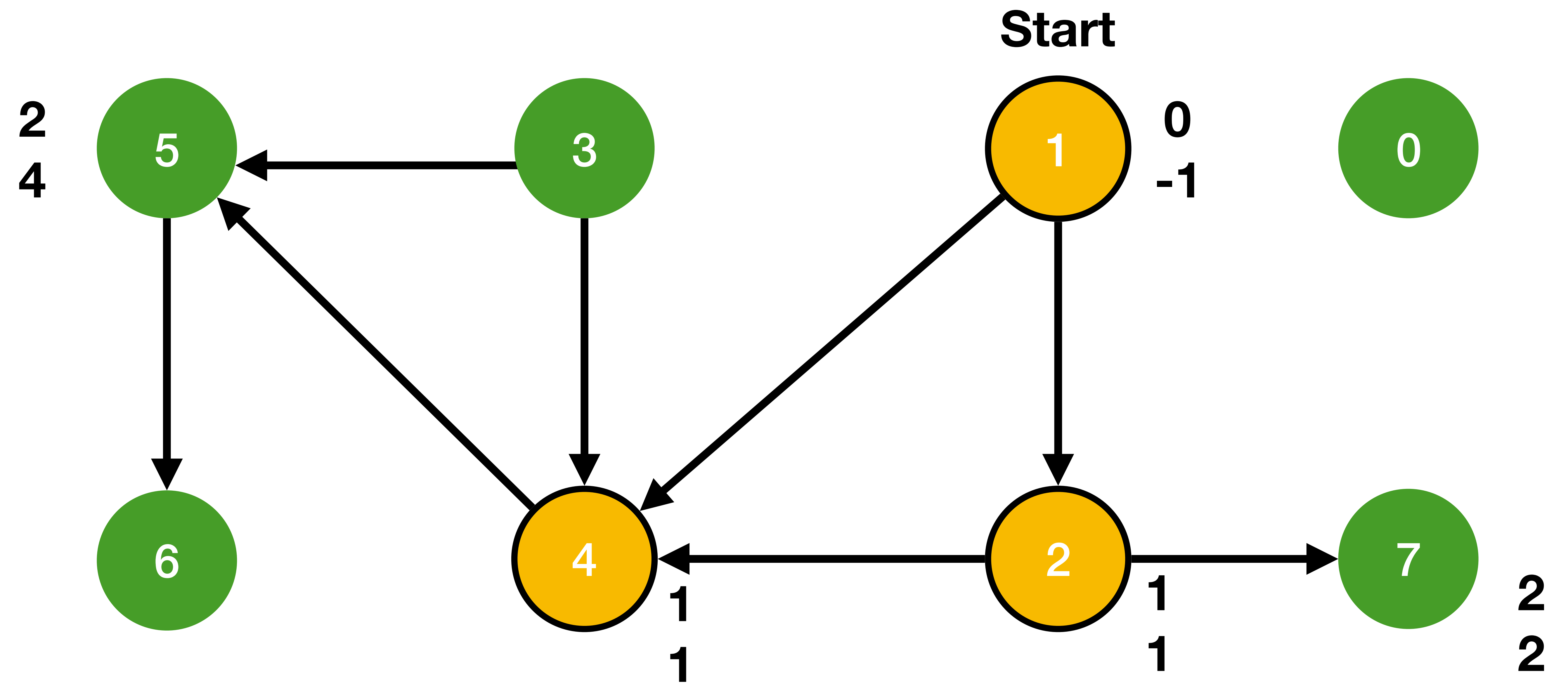


BFS of directed graph



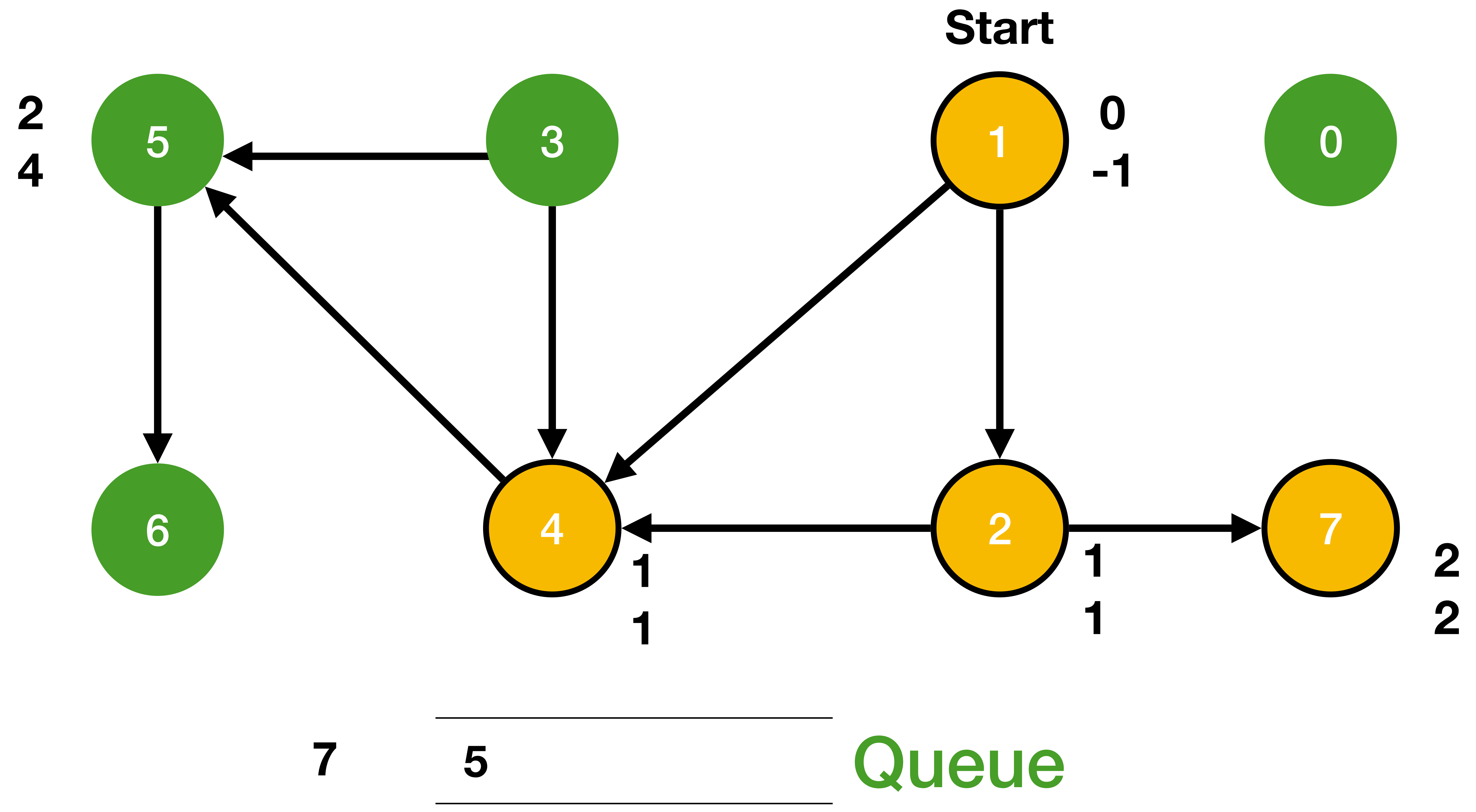
2 | 4 7 | Queue

BFS of directed graph

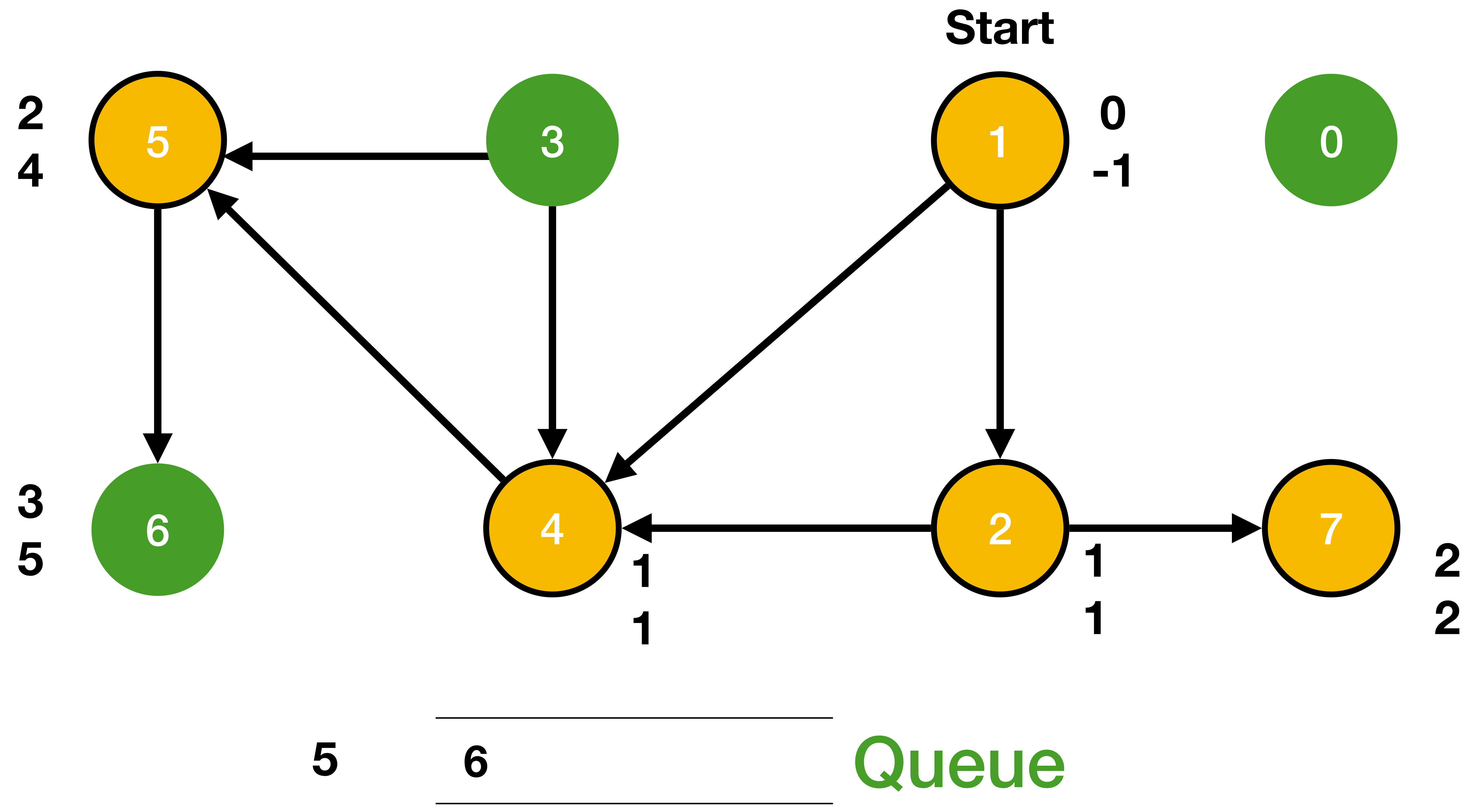


4 | 7 5 | Queue

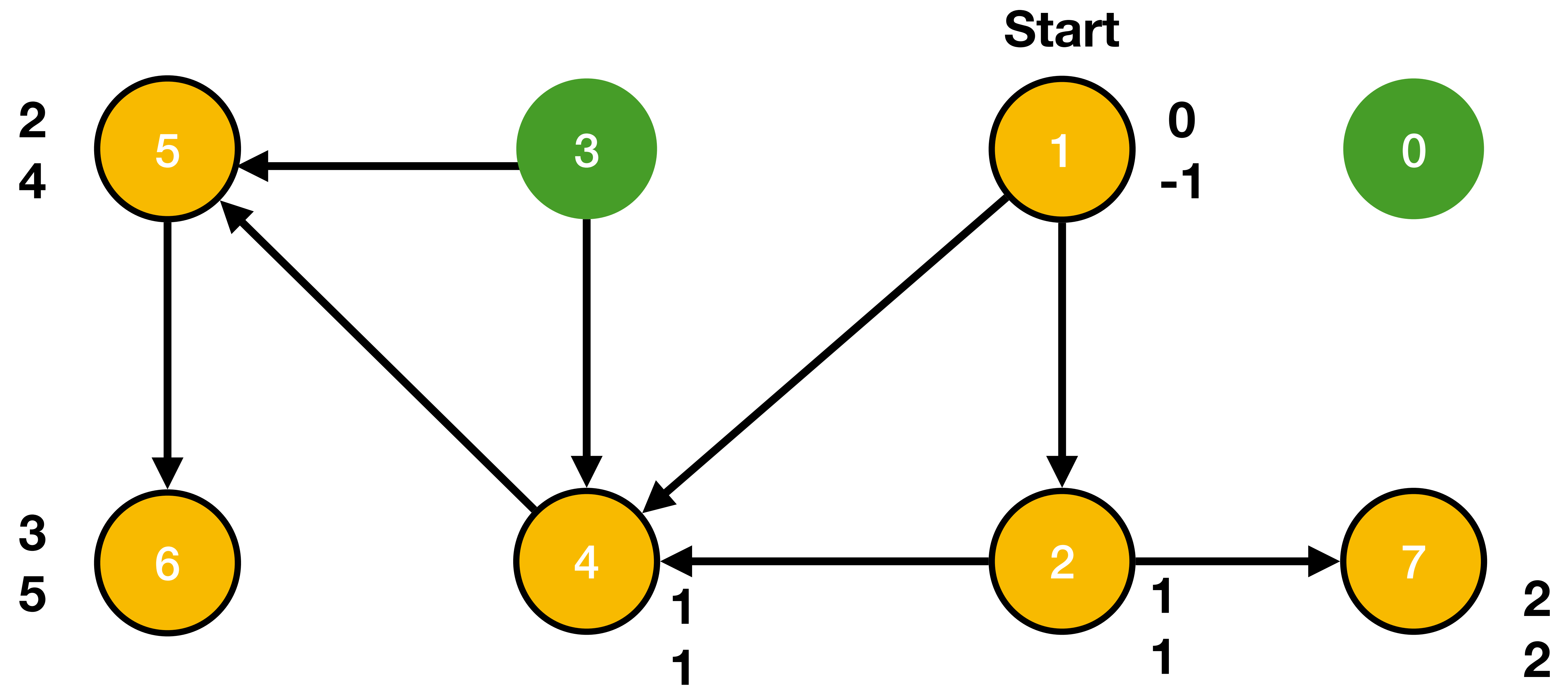
BFS of directed graph



BFS of directed graph



BFS of directed graph



6

Queue

BFS analysis

- Enqueue and Dequeue time O(1)
- Each adjacency list is scanned at most once, sum of length of all adjacency list is Θ(E)
- Total time scanning adjacency list = O(E)
- Initialization Overhead O(V)

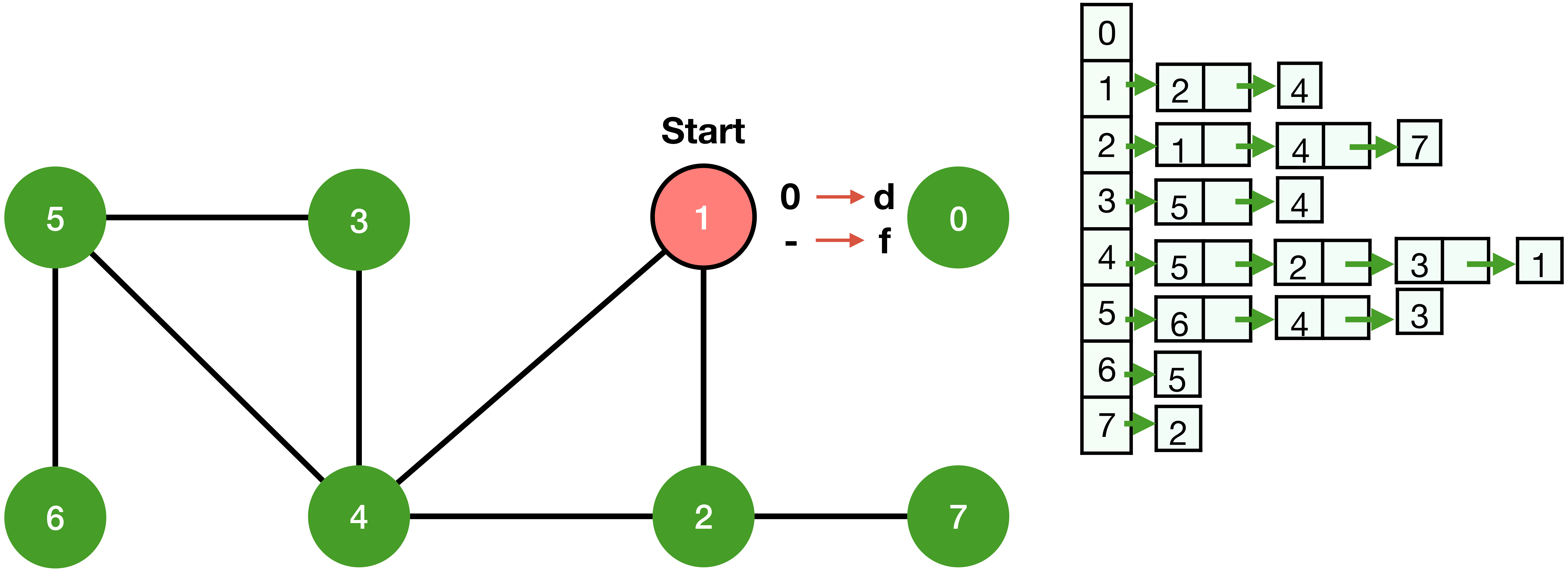
    Total time = O(E + V)

- What is the complexity if adjacency matrix is used?

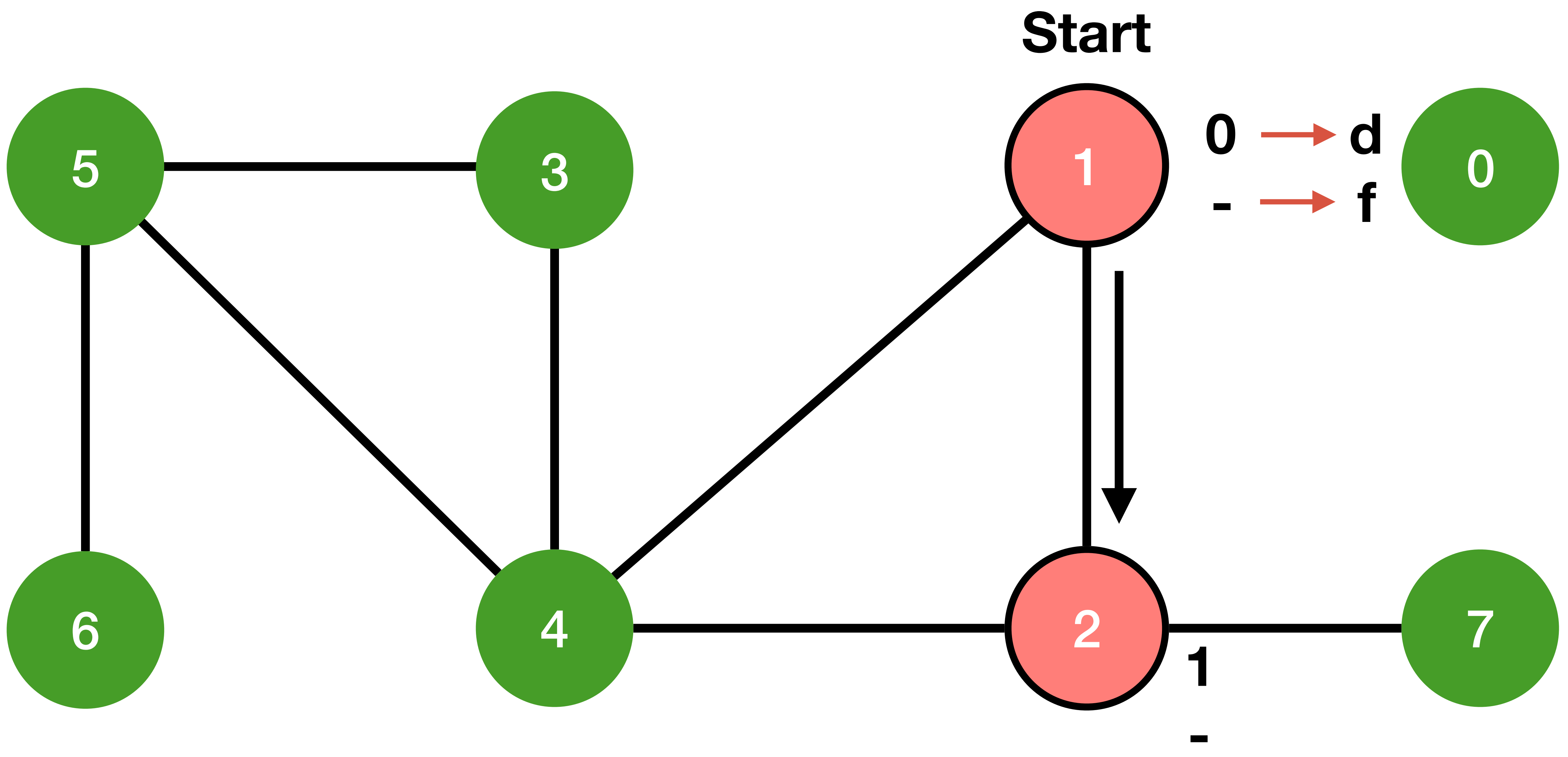
Depth First Search (DFS)

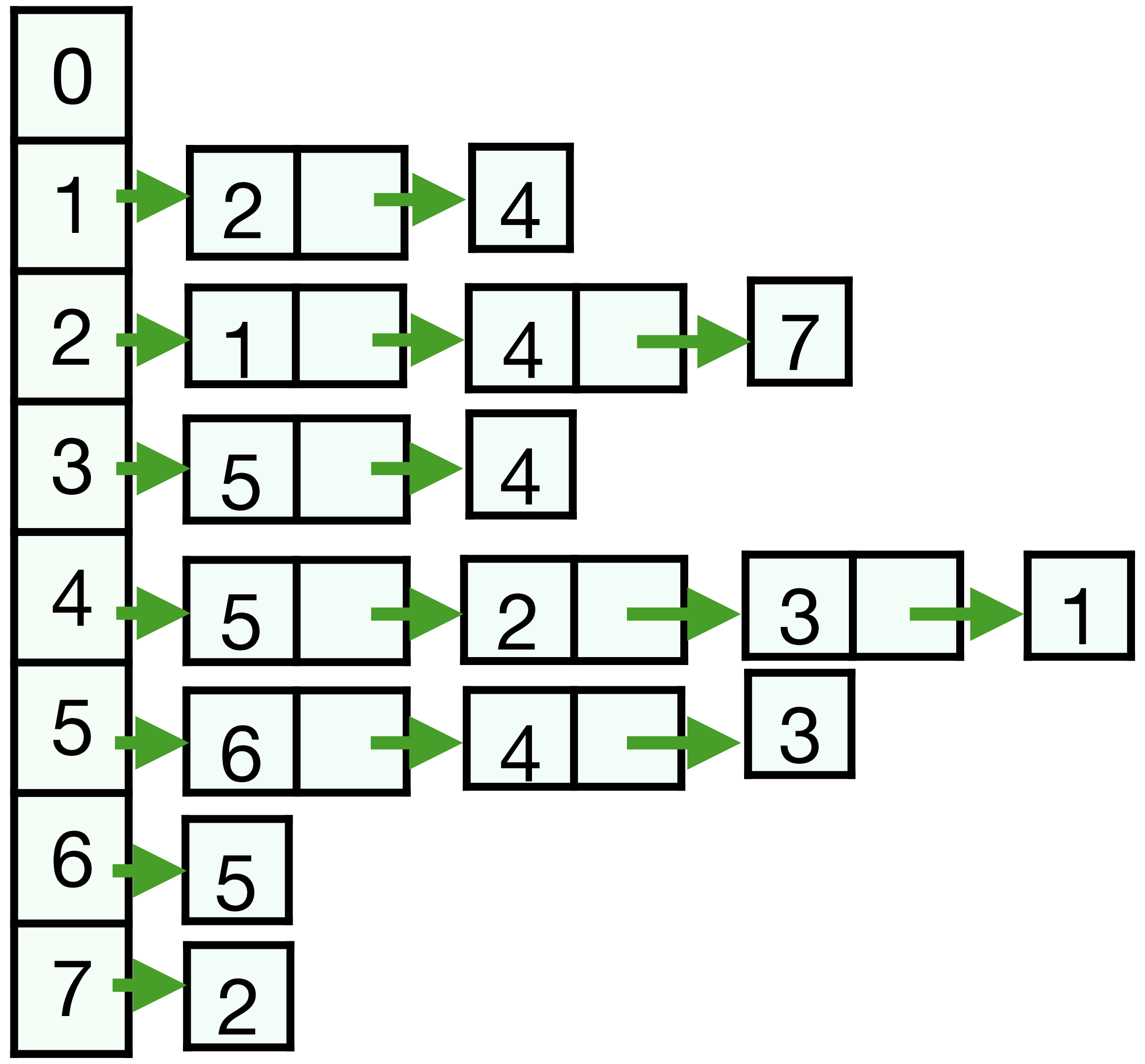
- Used as part of other algorithms such as cut vertices, strongly connected component
- Visit node adjacent to the node that was last visited
- BFS searches from a single source, but DFS can search from multiple sources
- **Stores (d, f)** where d = discovery time, f = finishing time

DFS of undirected graph

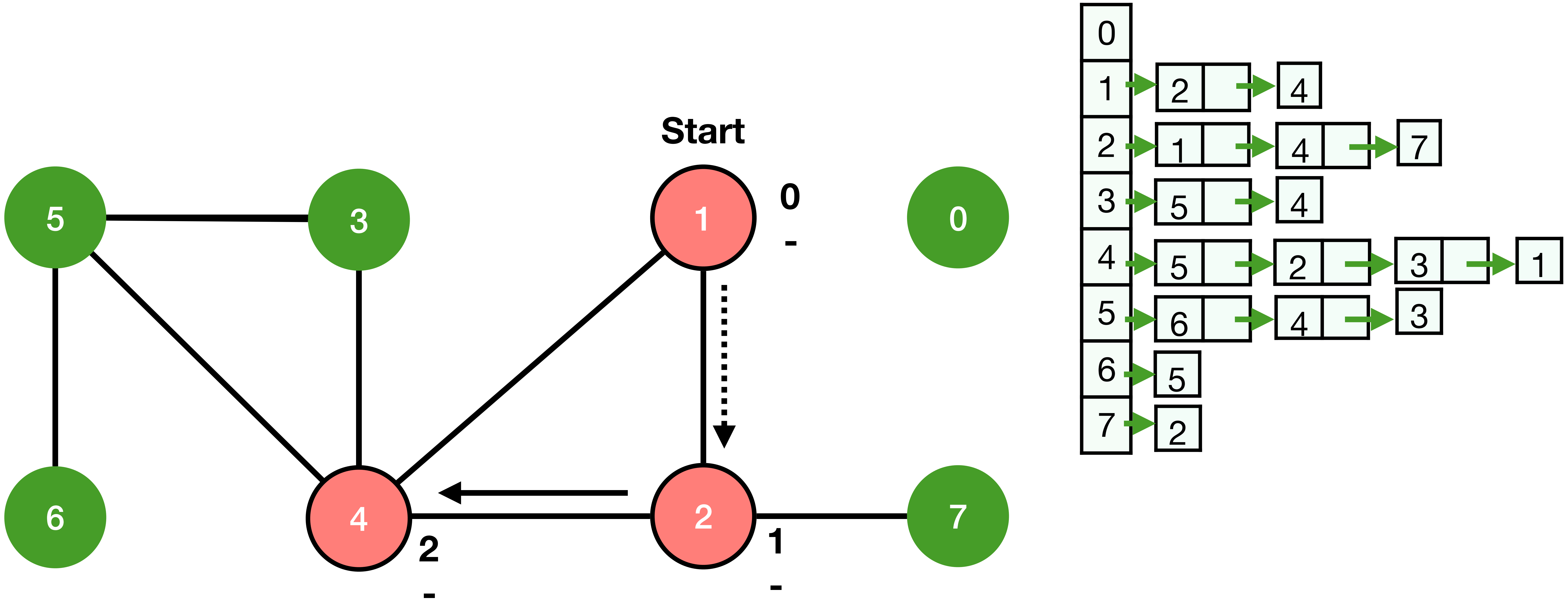


DFS of undirected graph

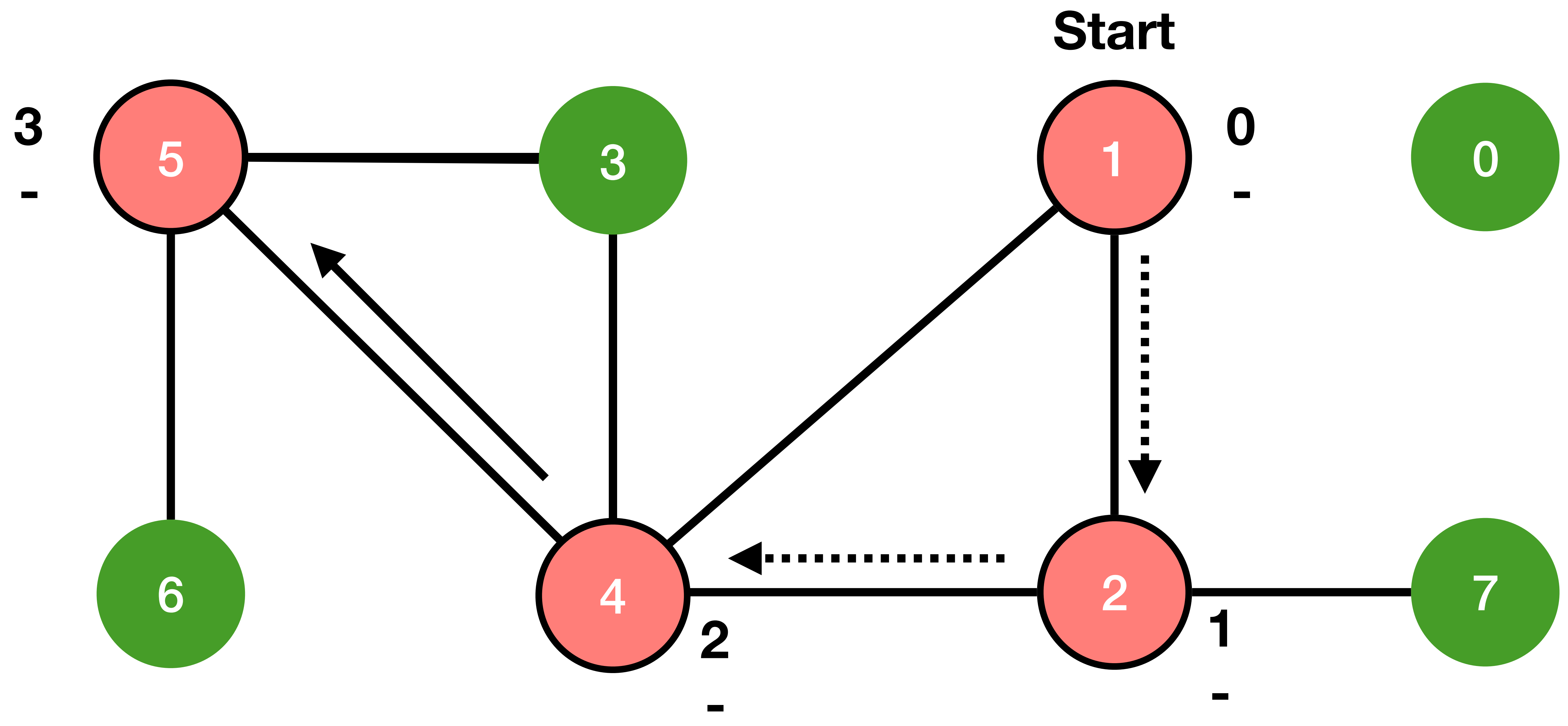


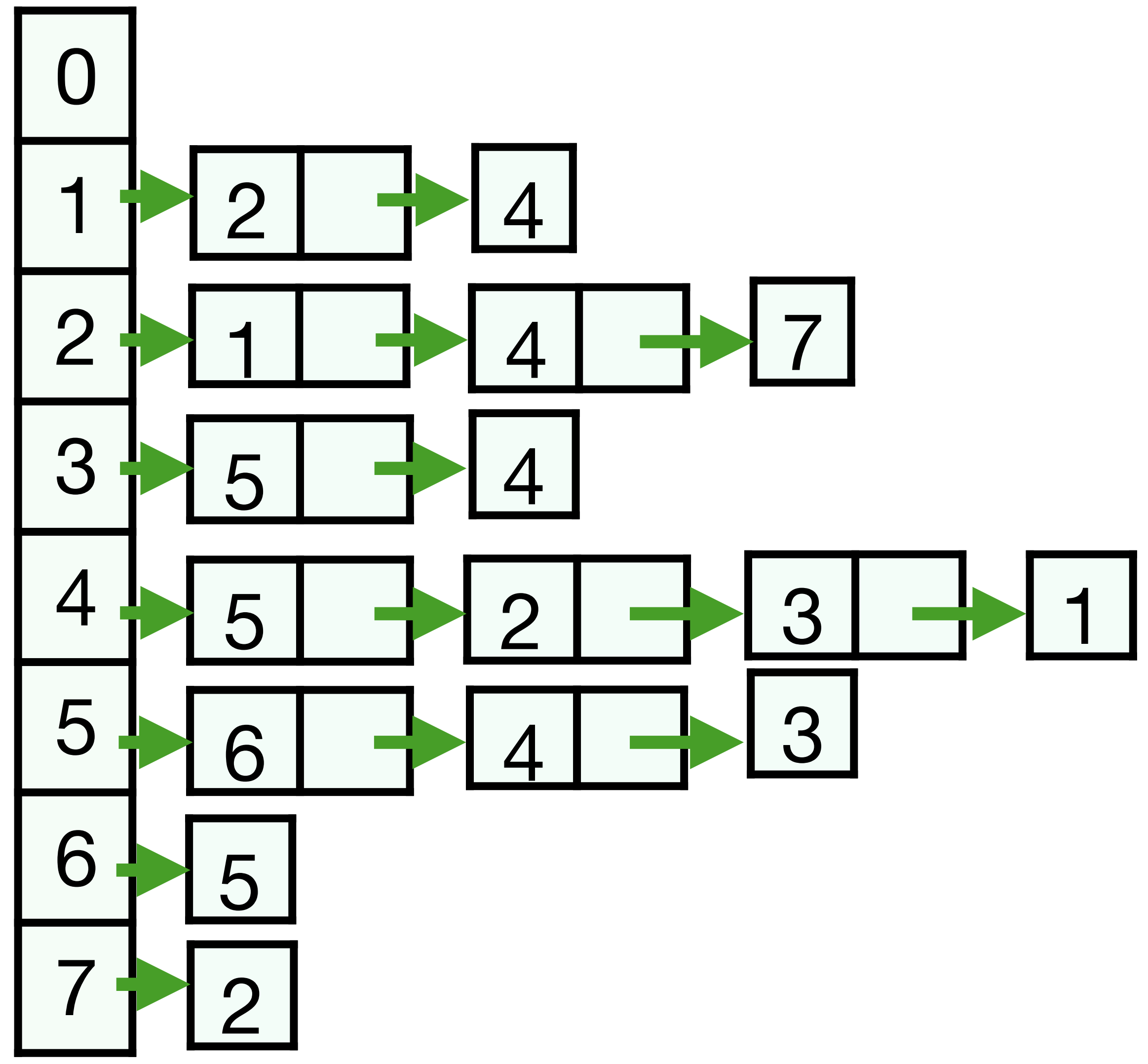


DFS of undirected graph



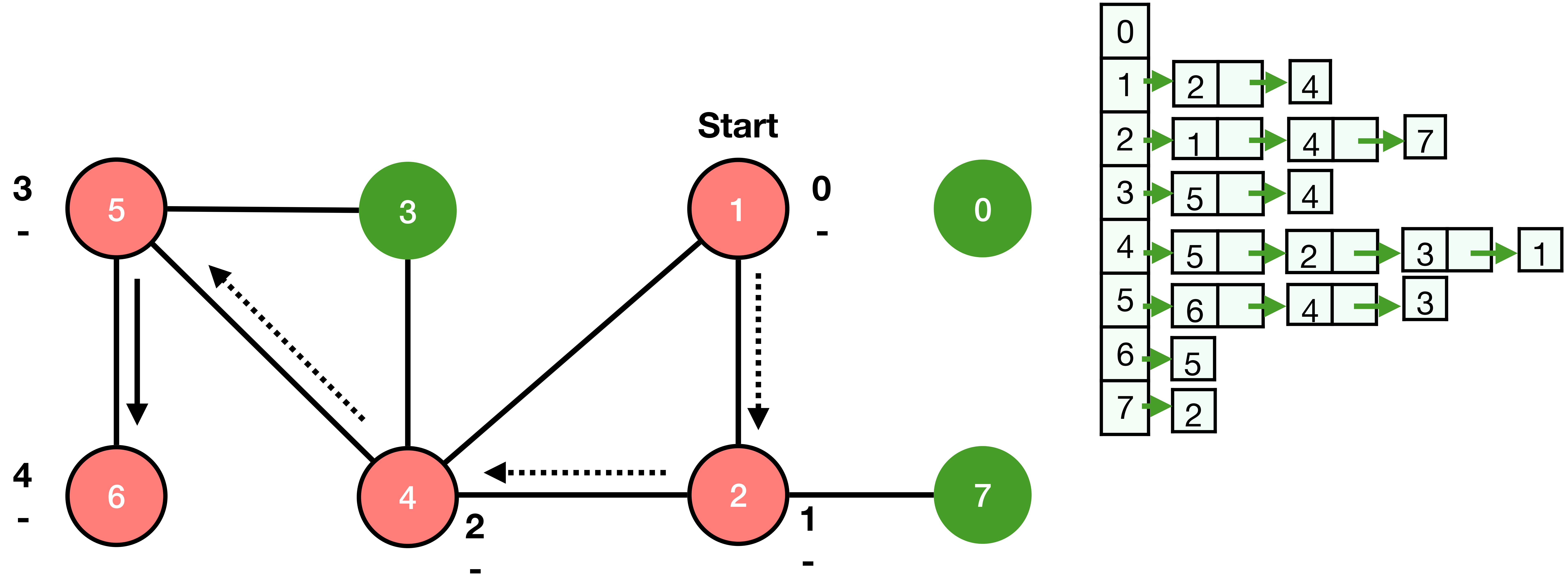
DFS of undirected graph



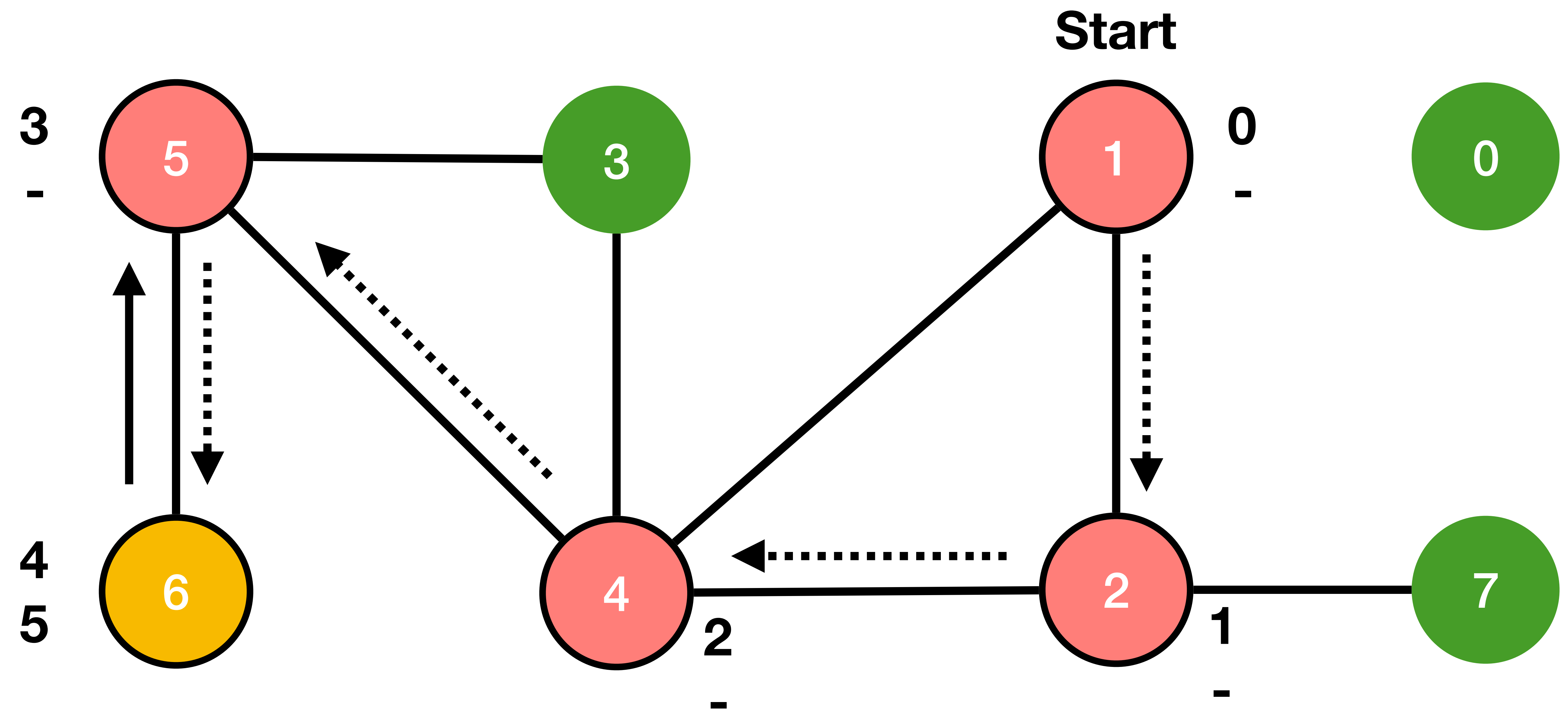


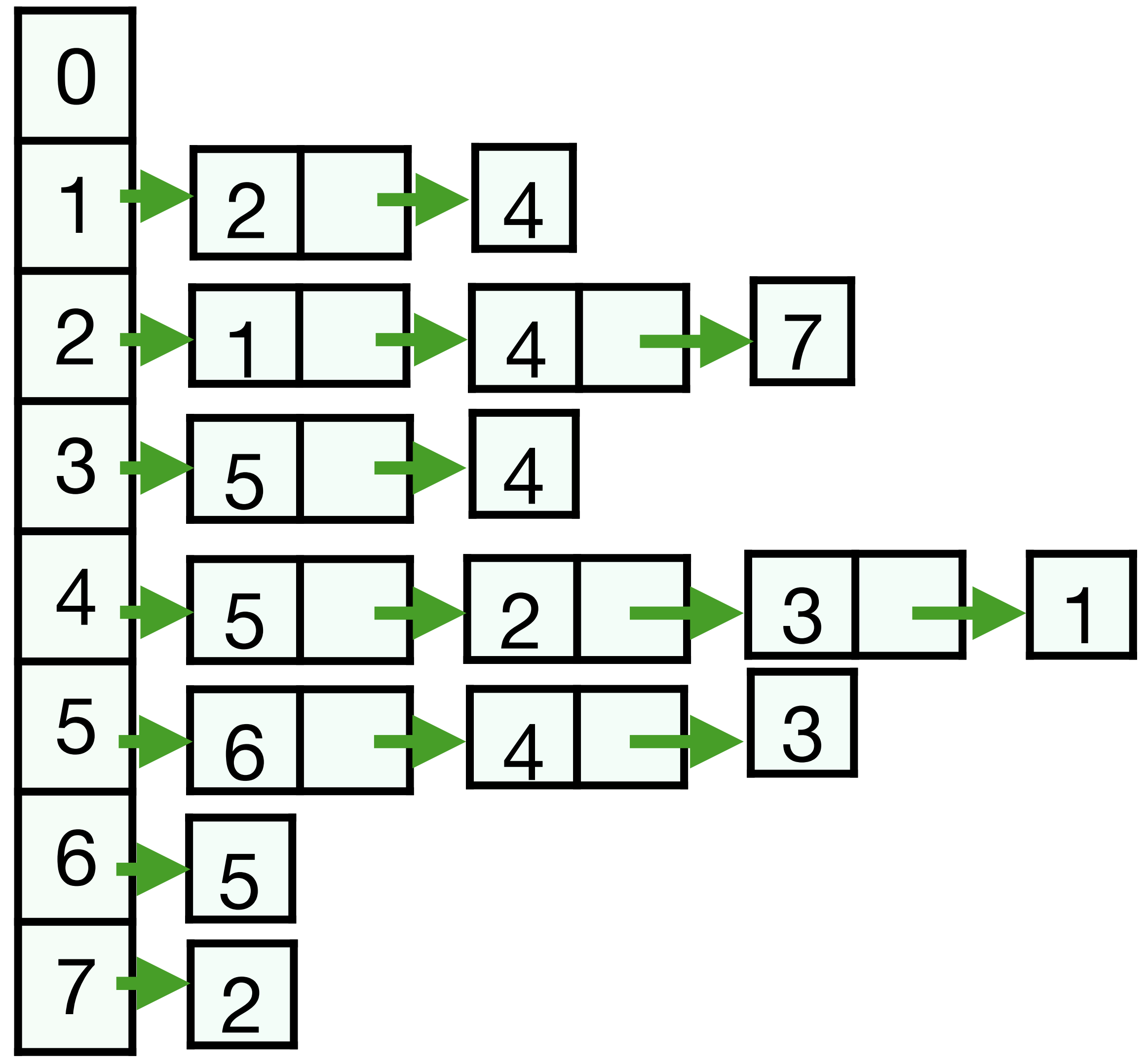
SPEL Technologies, Inc

DFS of undirected graph

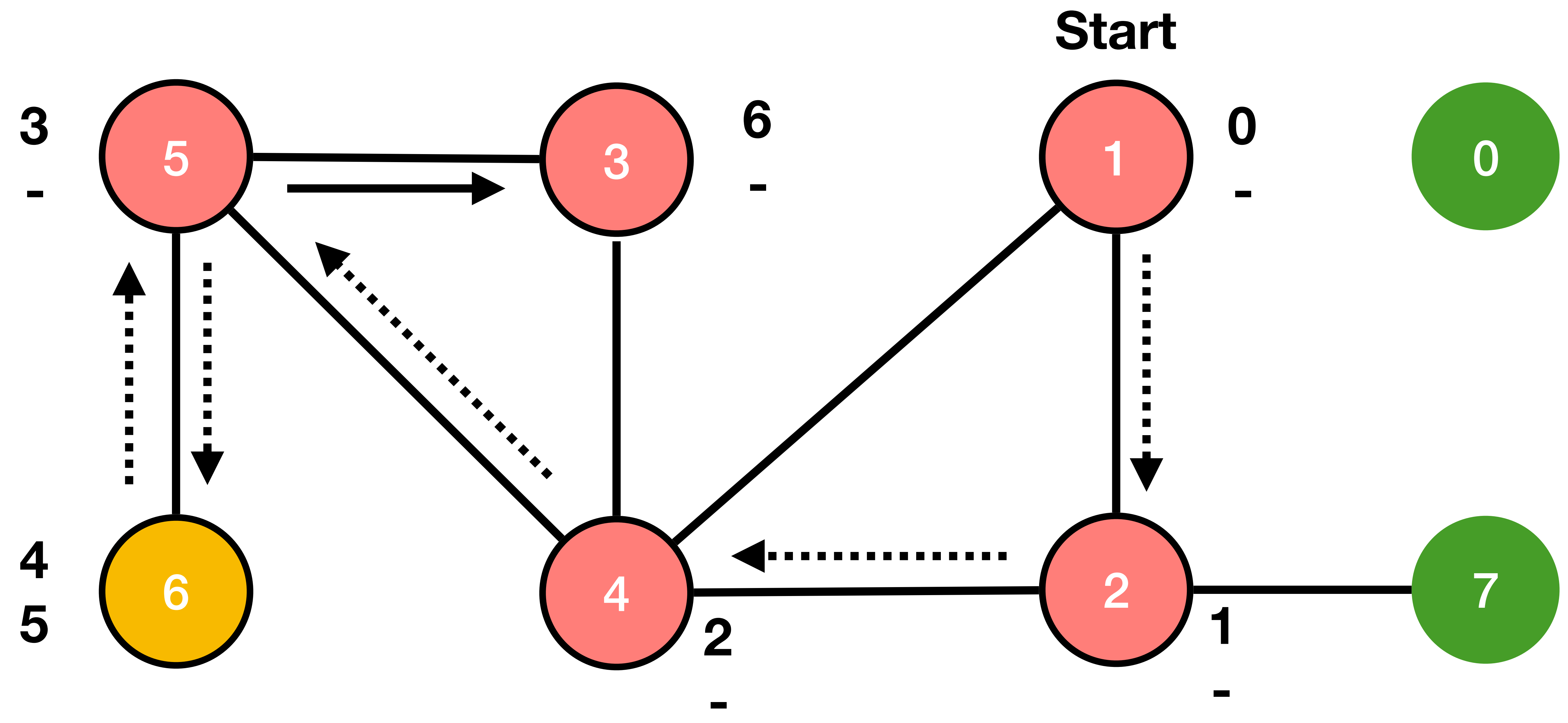


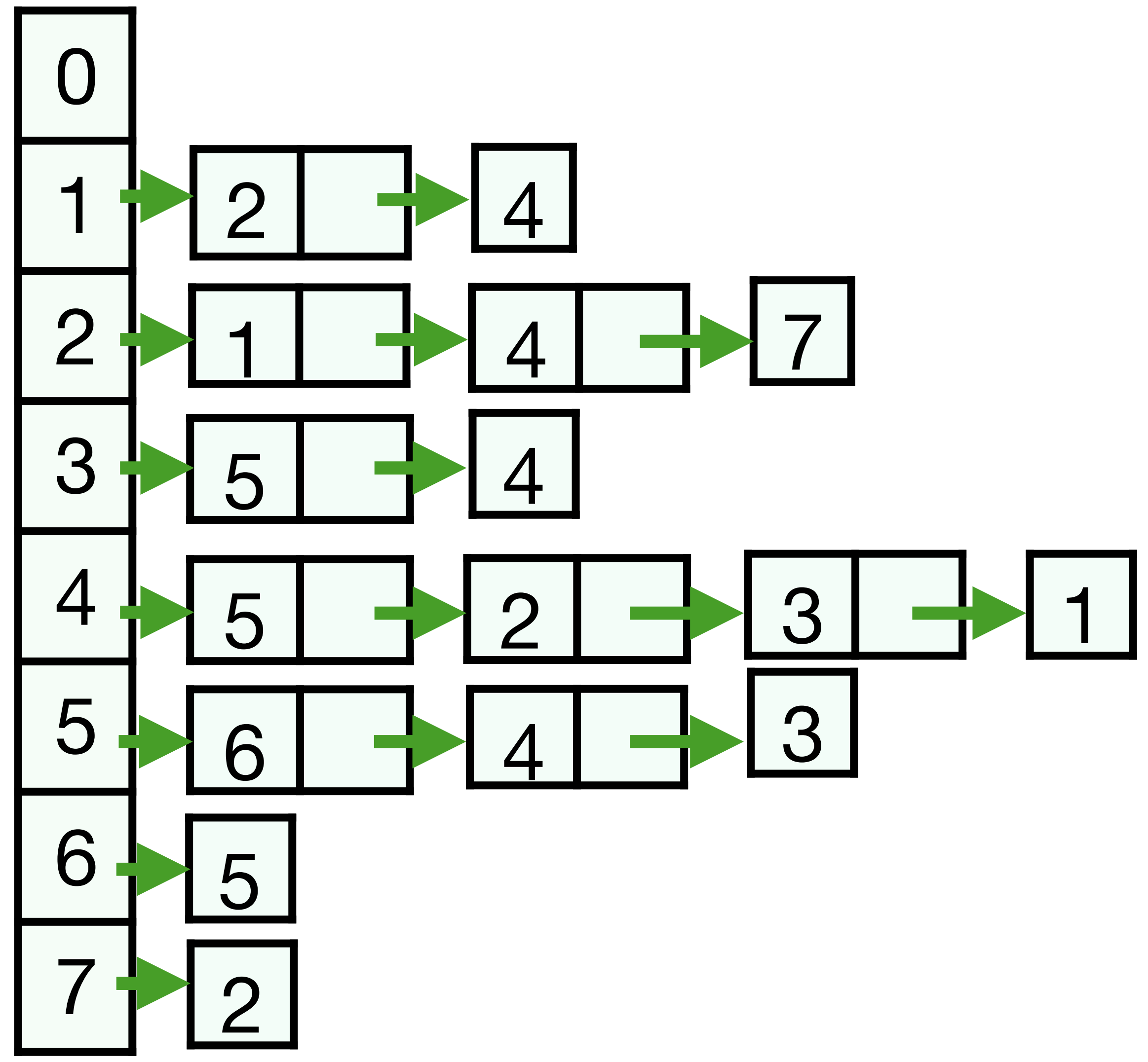
DFS of undirected graph





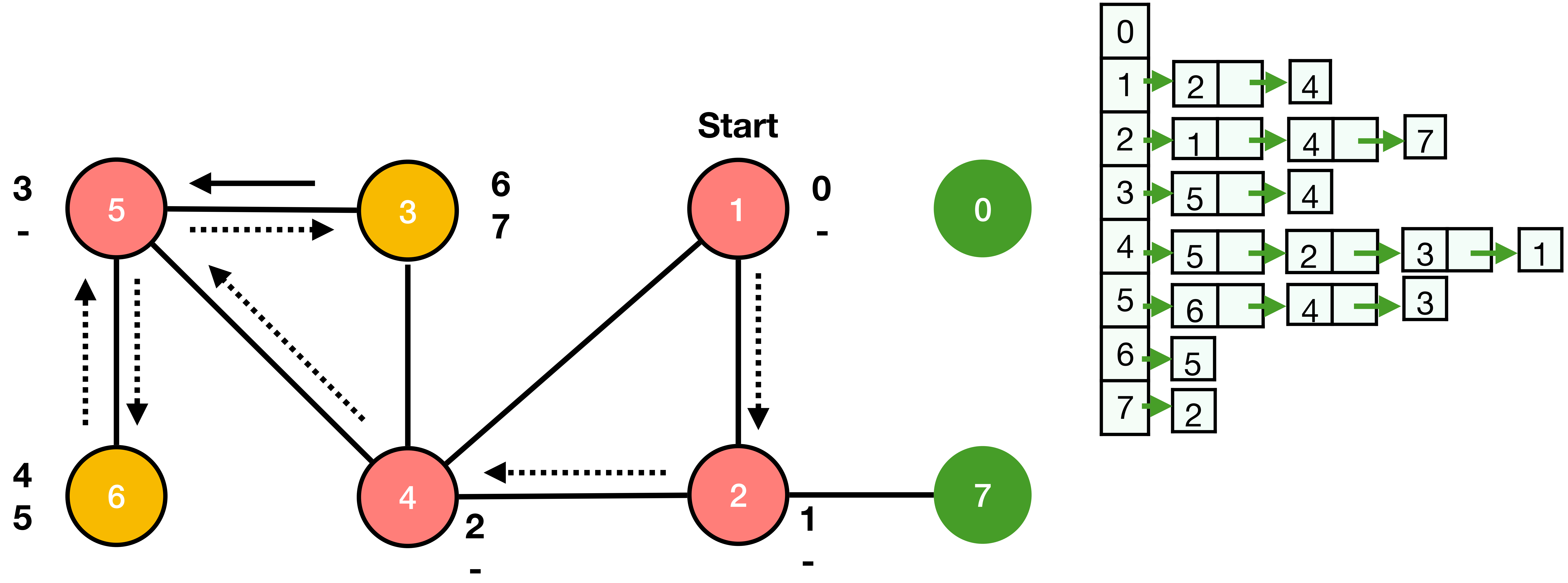
DFS of undirected graph



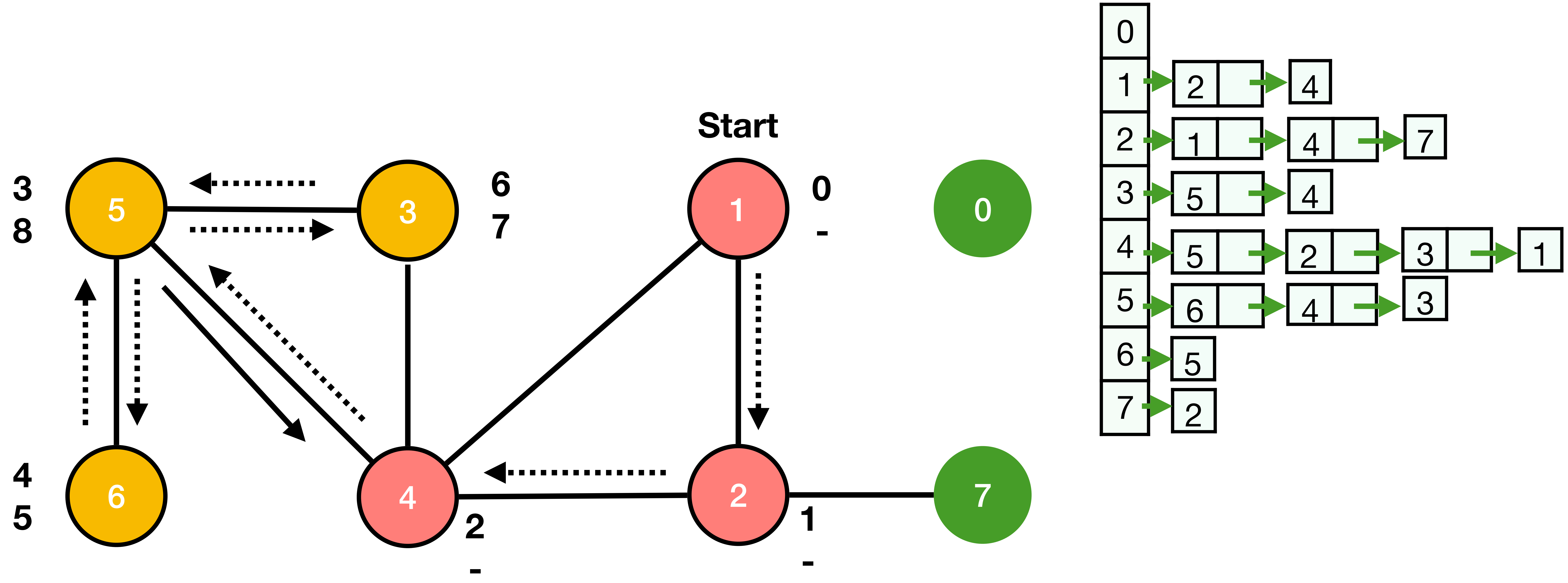


SPEL Technologies, Inc

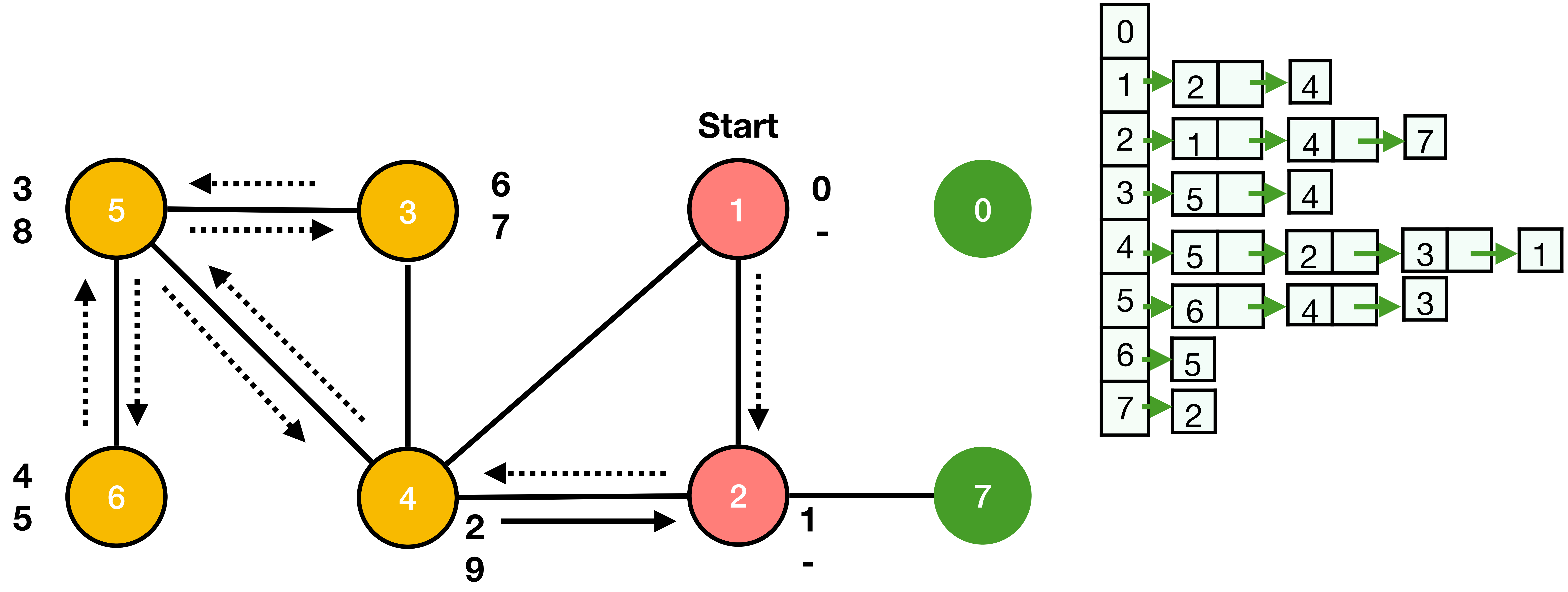
DFS of undirected graph



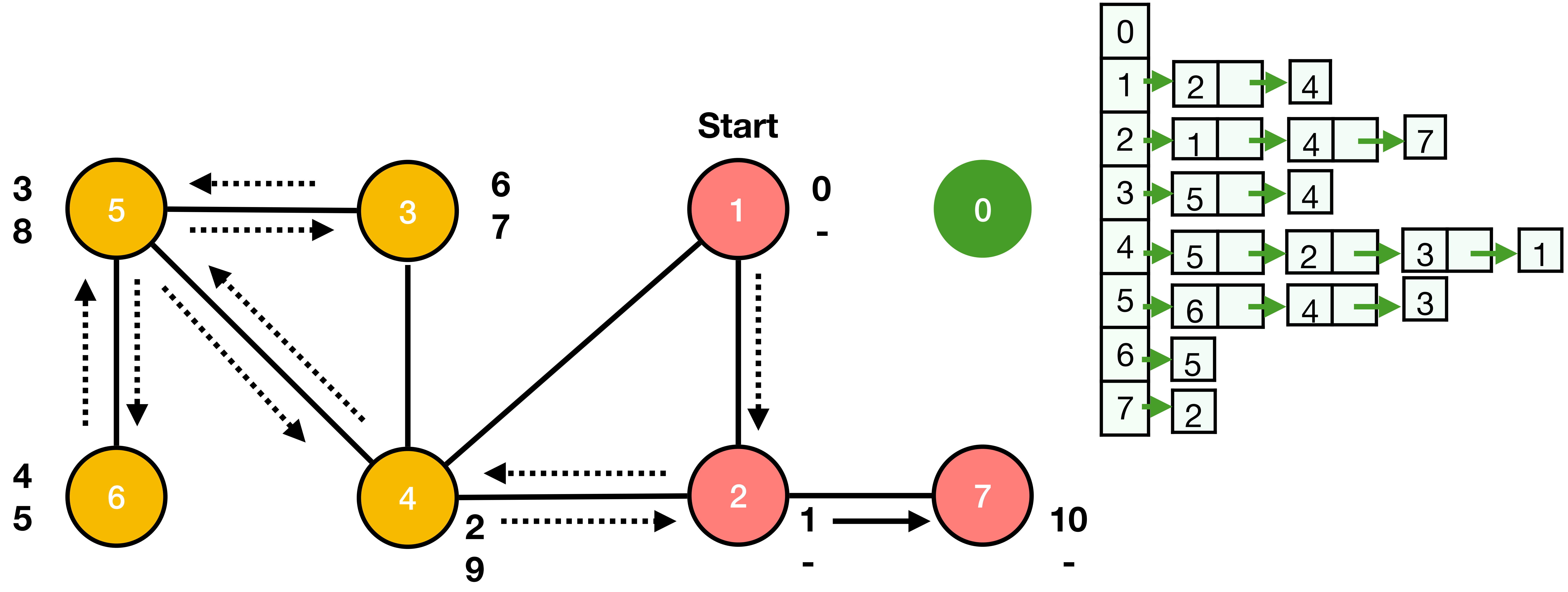
DFS of undirected graph



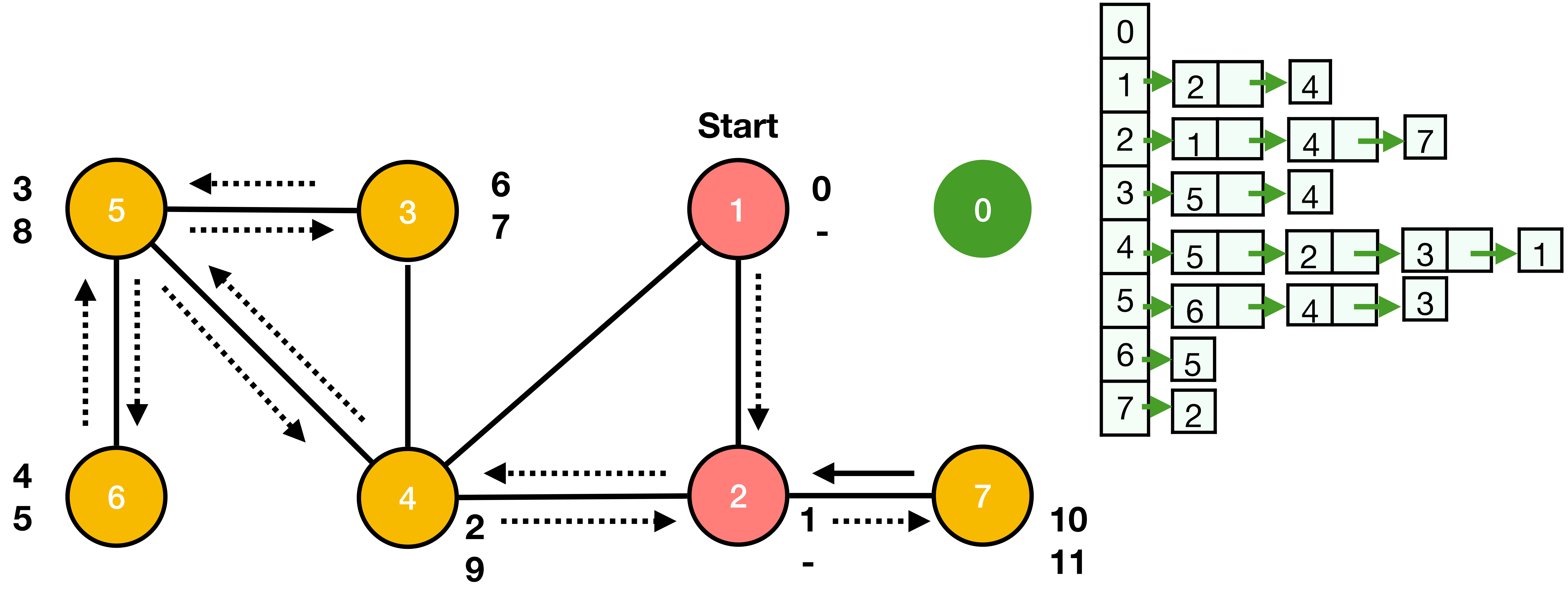
DFS of undirected graph



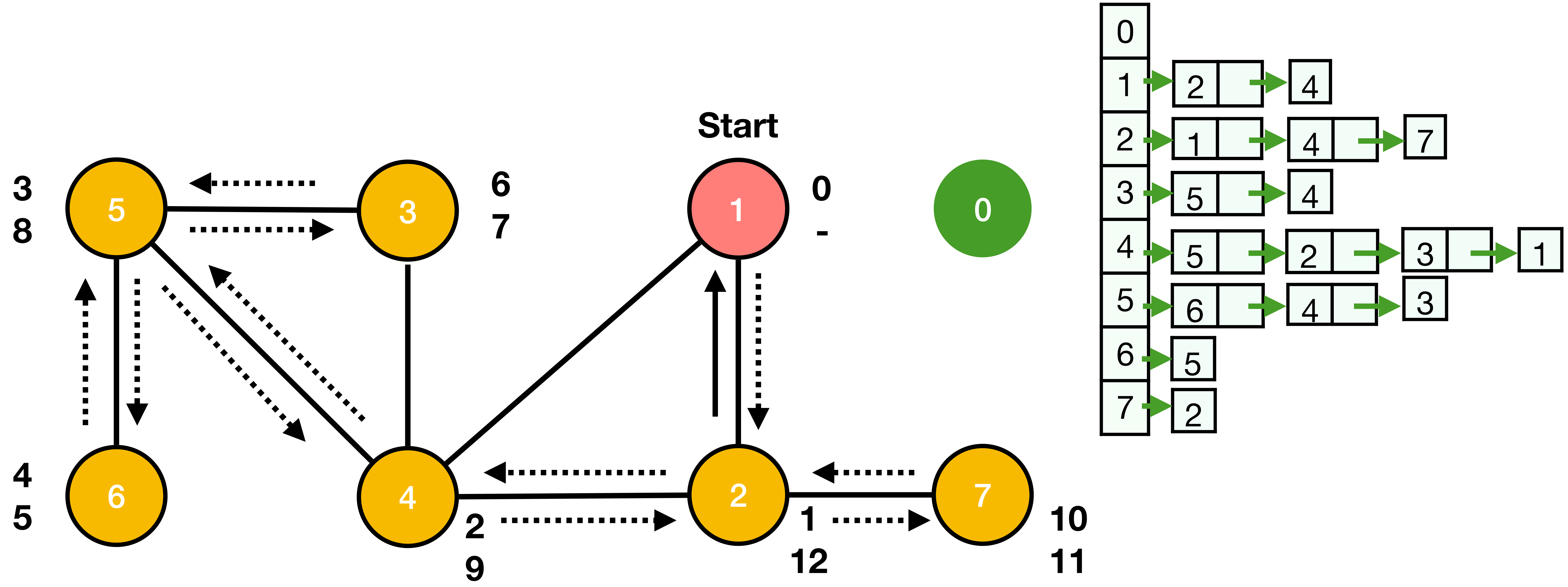
DFS of undirected graph



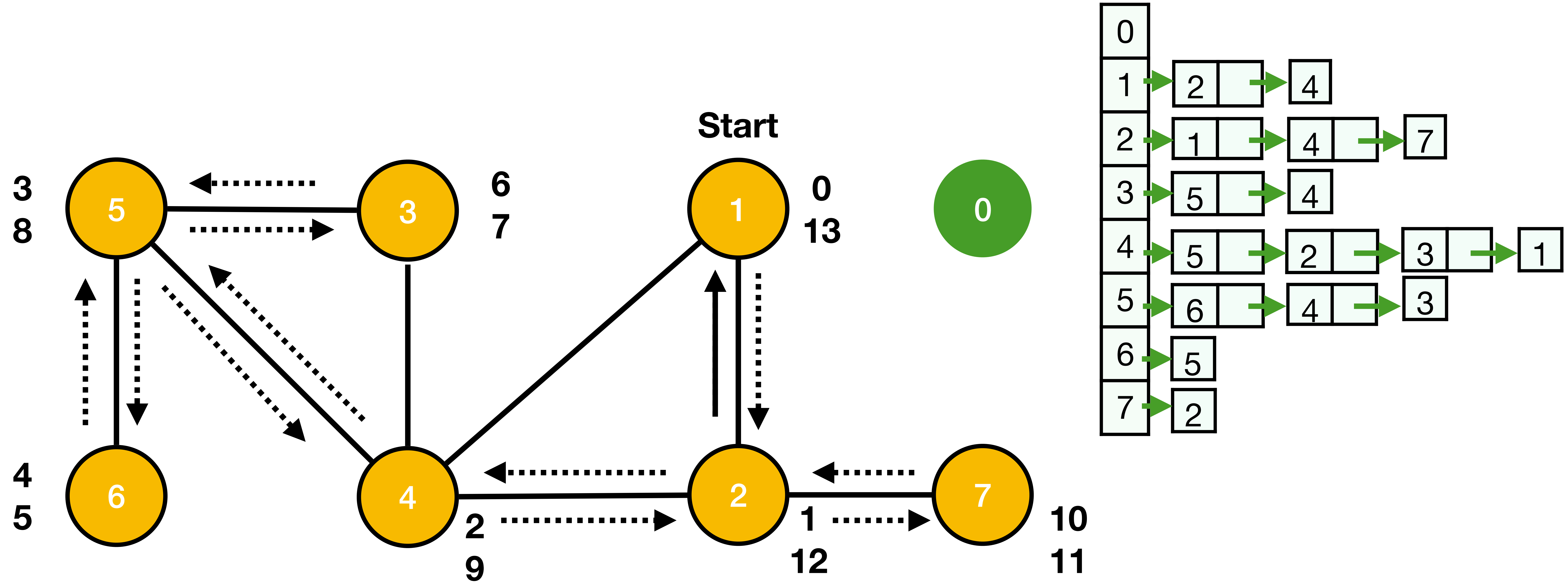
DFS of undirected graph



DFS of undirected graph



DFS of undirected graph



DFS implementation

Examine program o/p of DFS recursive method

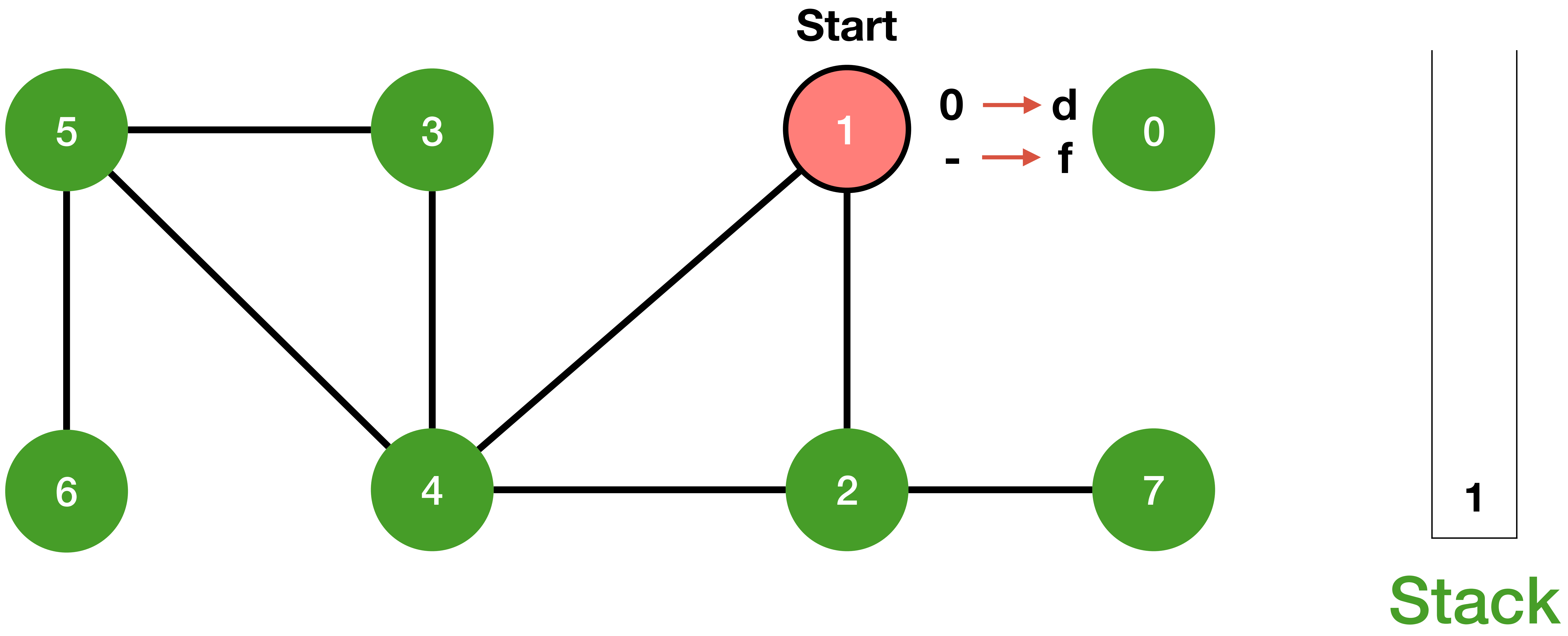
```
//s is starting vertex, myGraph stores adjacency list of graph
void depthFirstSearch(s, myGraph, visited){

  mark vertex s as visited

  for each vertex v adjacent to s {
    if (v has not been visited)
         depthFirstSearch(v, myGraph, visited);
  }
}
```

Time complexity: Θ(V+E)

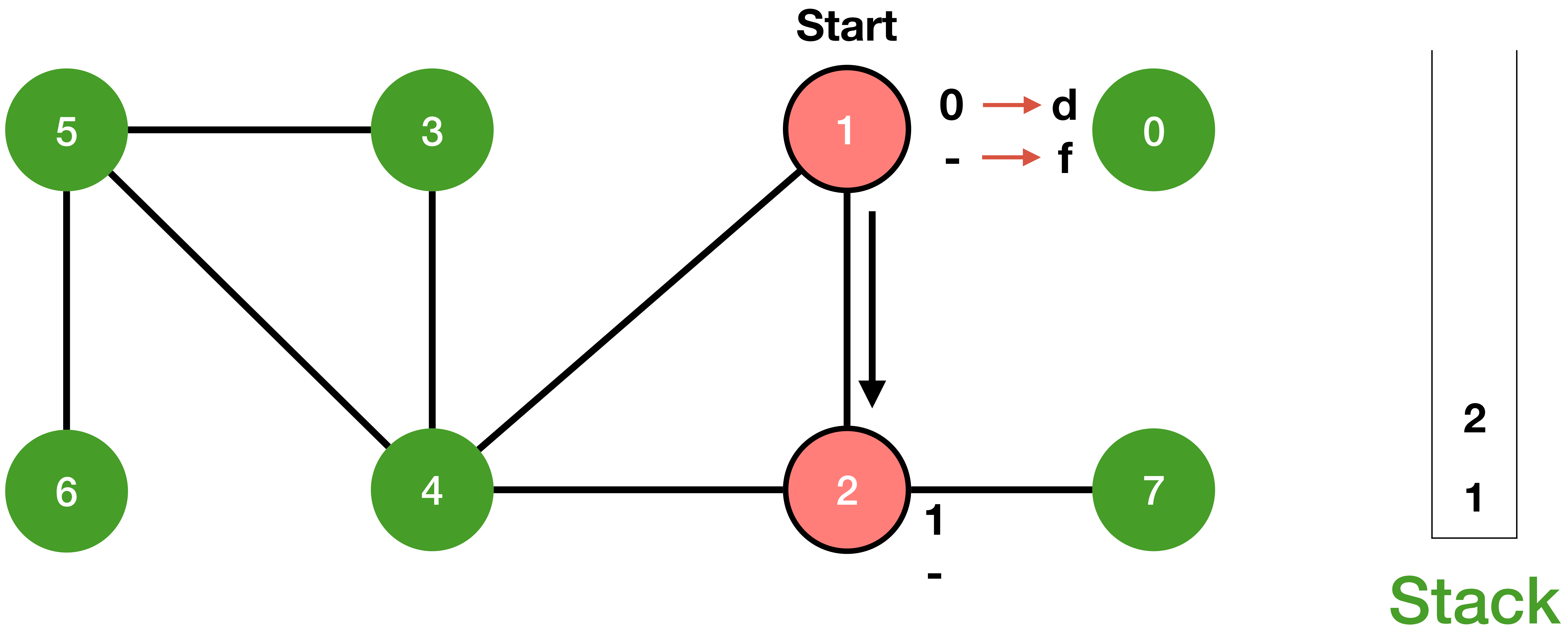
DFS implementation With Stack

- Advantages and disadvantages of recursion?
- What information must be stored with recursion?
- Avoid recursion by using a stack in the program.
  - Add vertices to the stack in DFS order
  - Pop vertex from stack when all its neighbors have been explored,

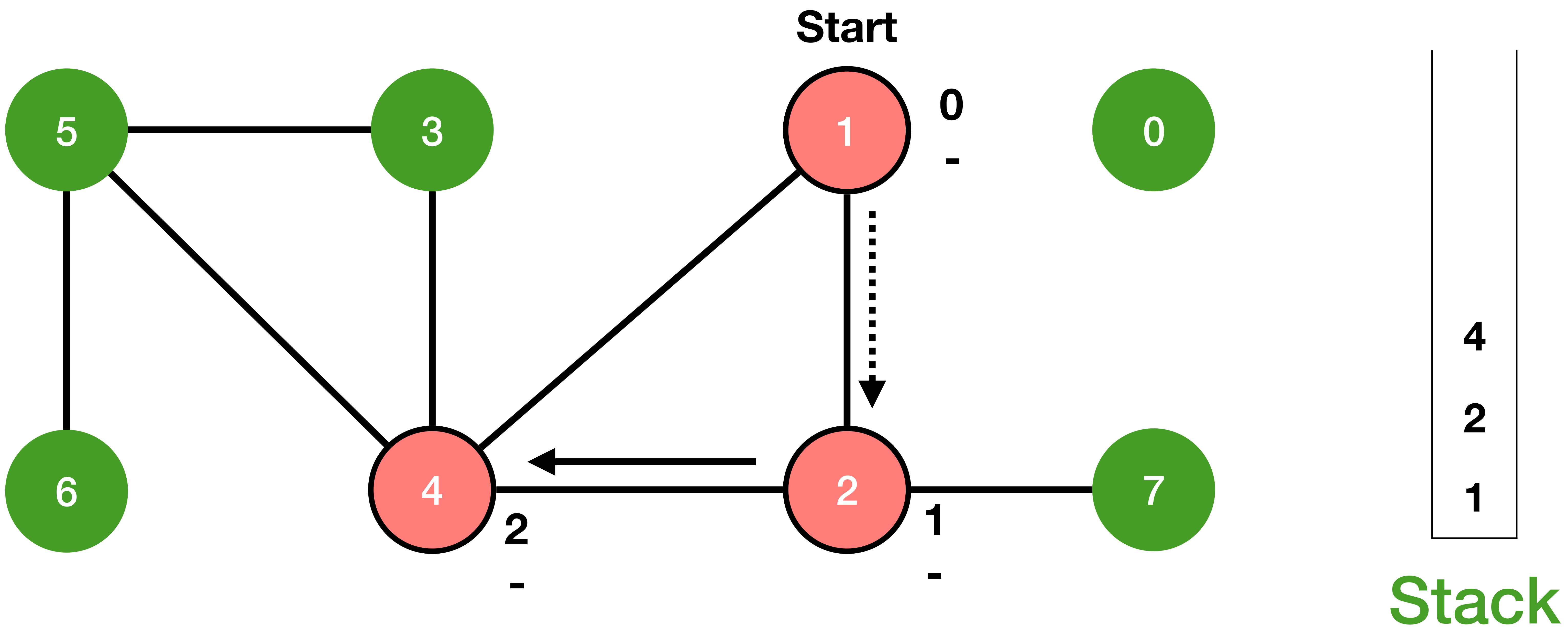
DFS using stack



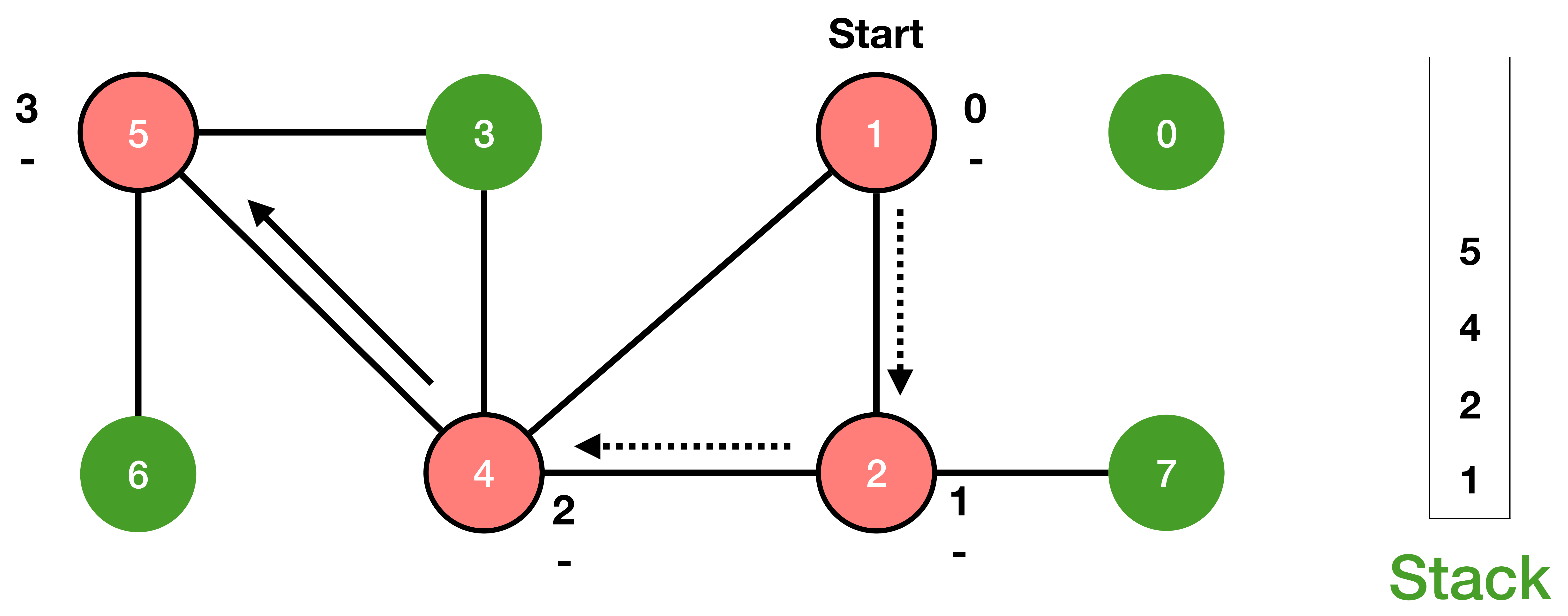
DFS using stack



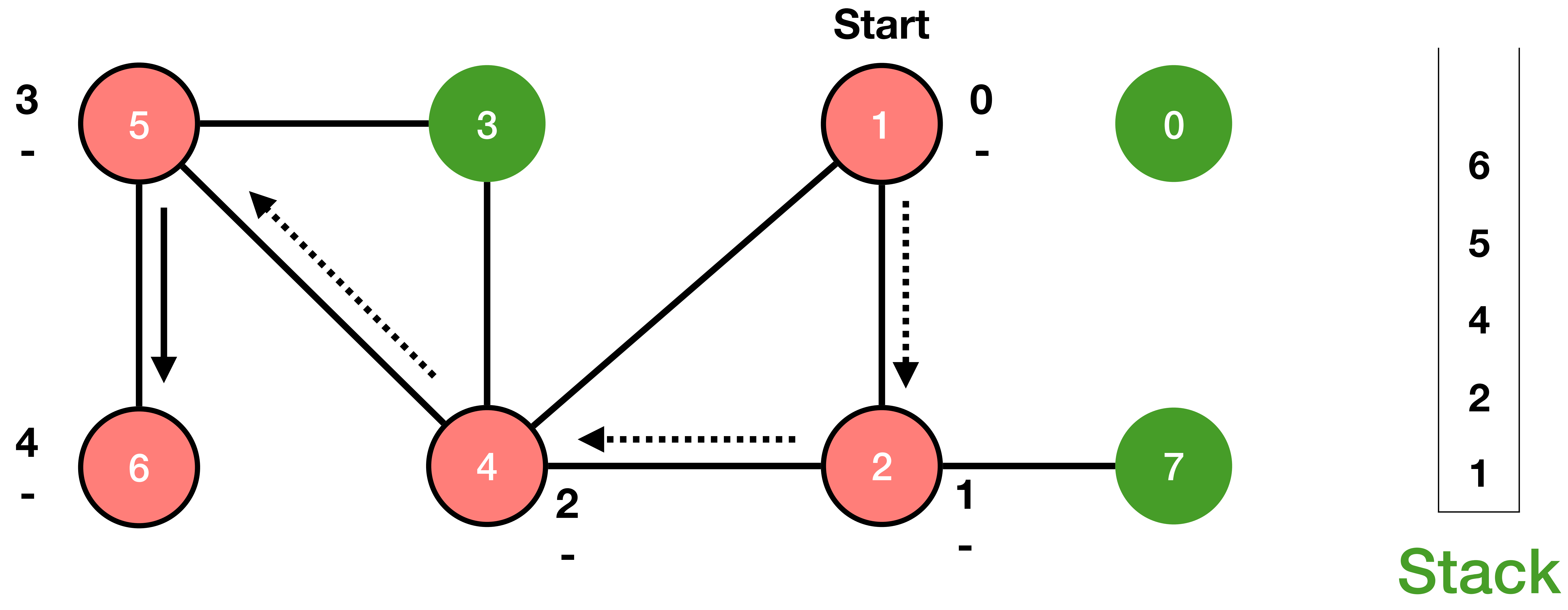
DFS using stack



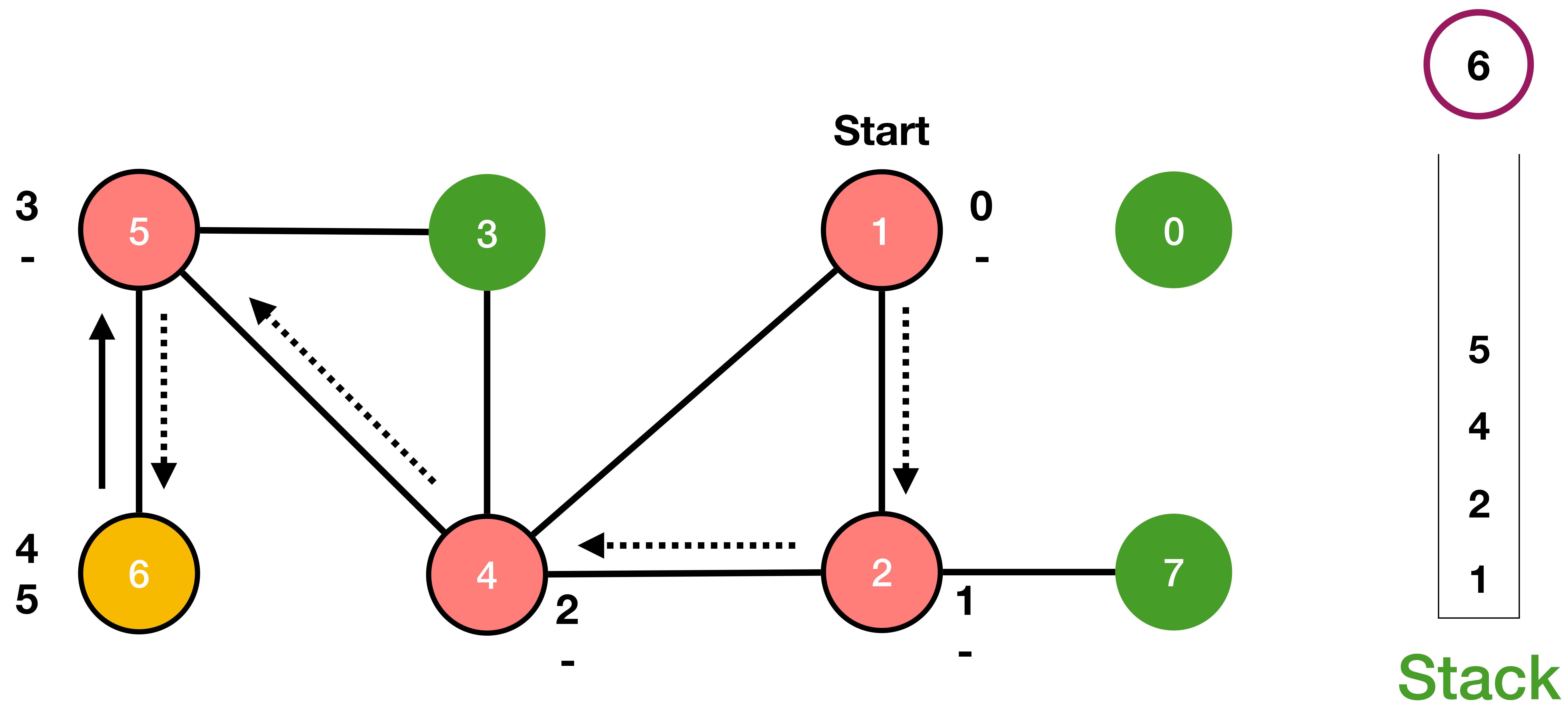
DFS using stack



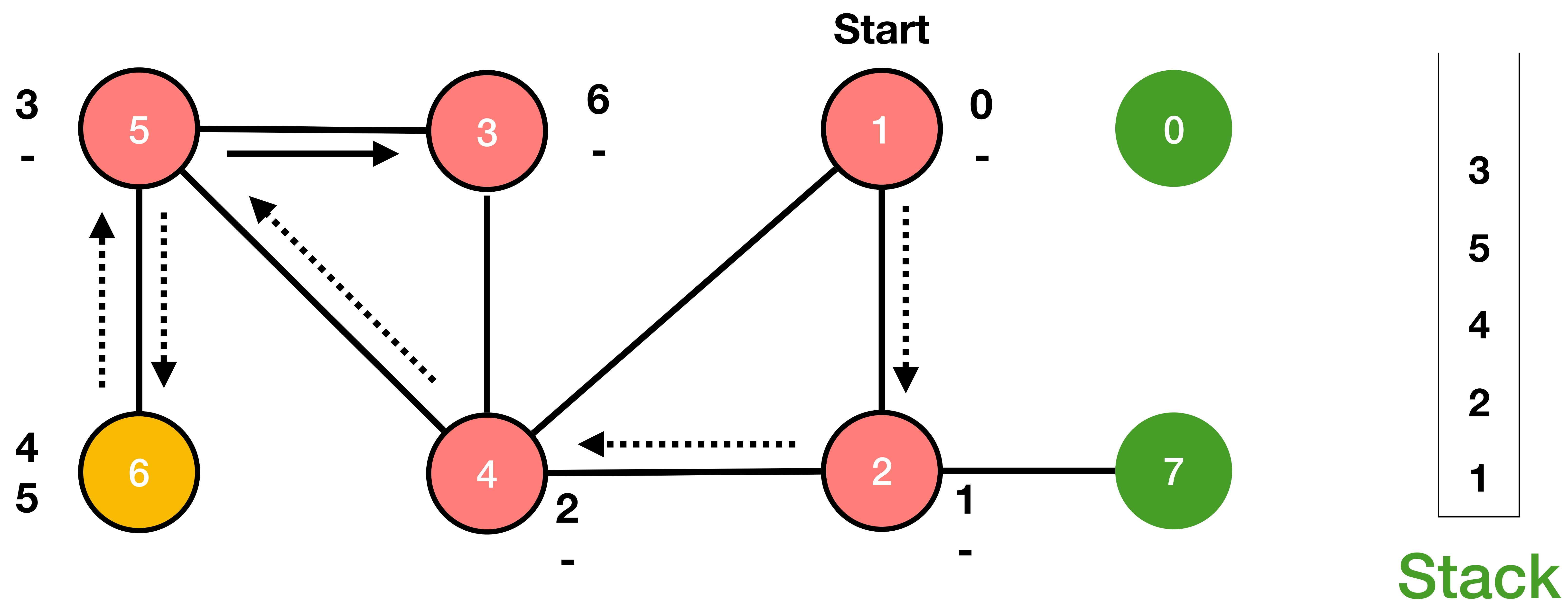
DFS using stack



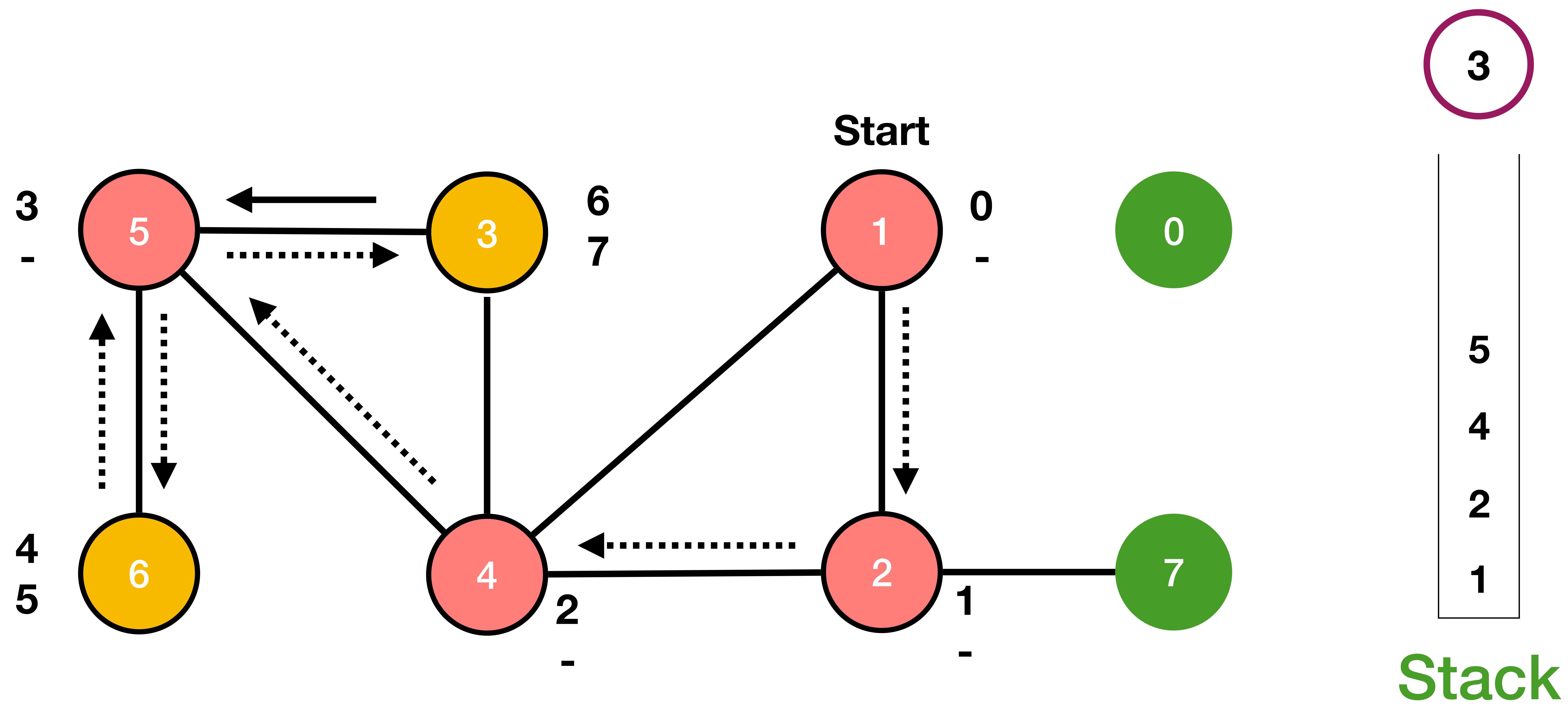
DFS using stack



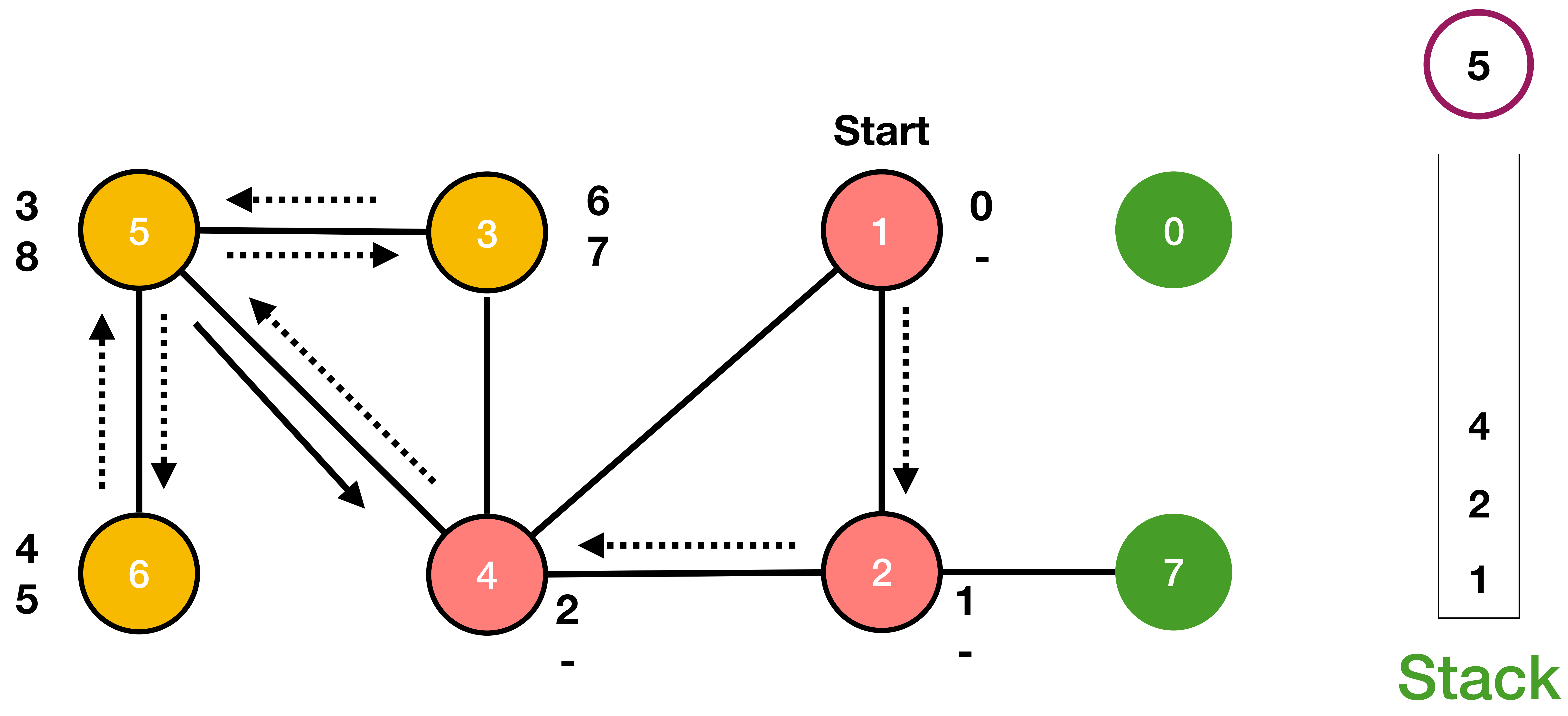
DFS using stack



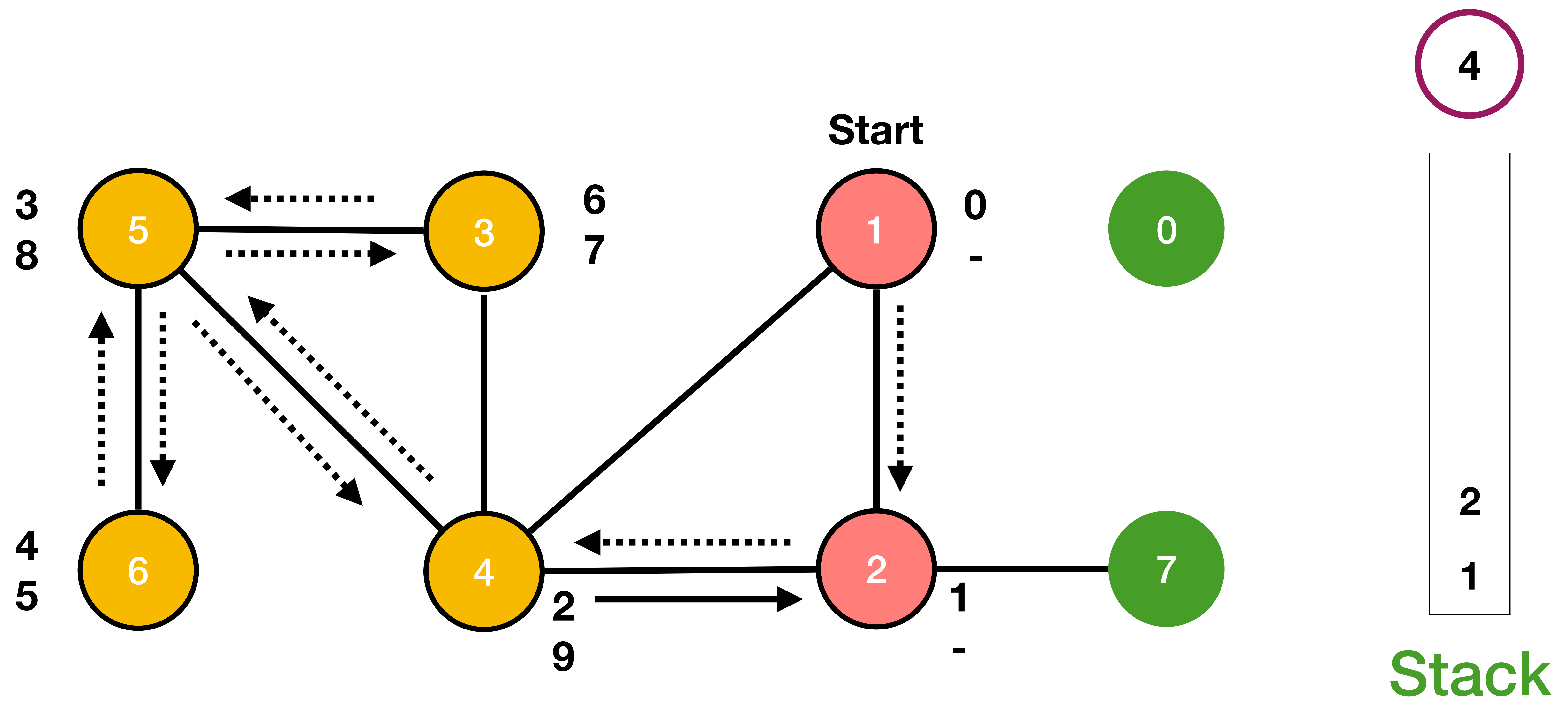
DFS using stack



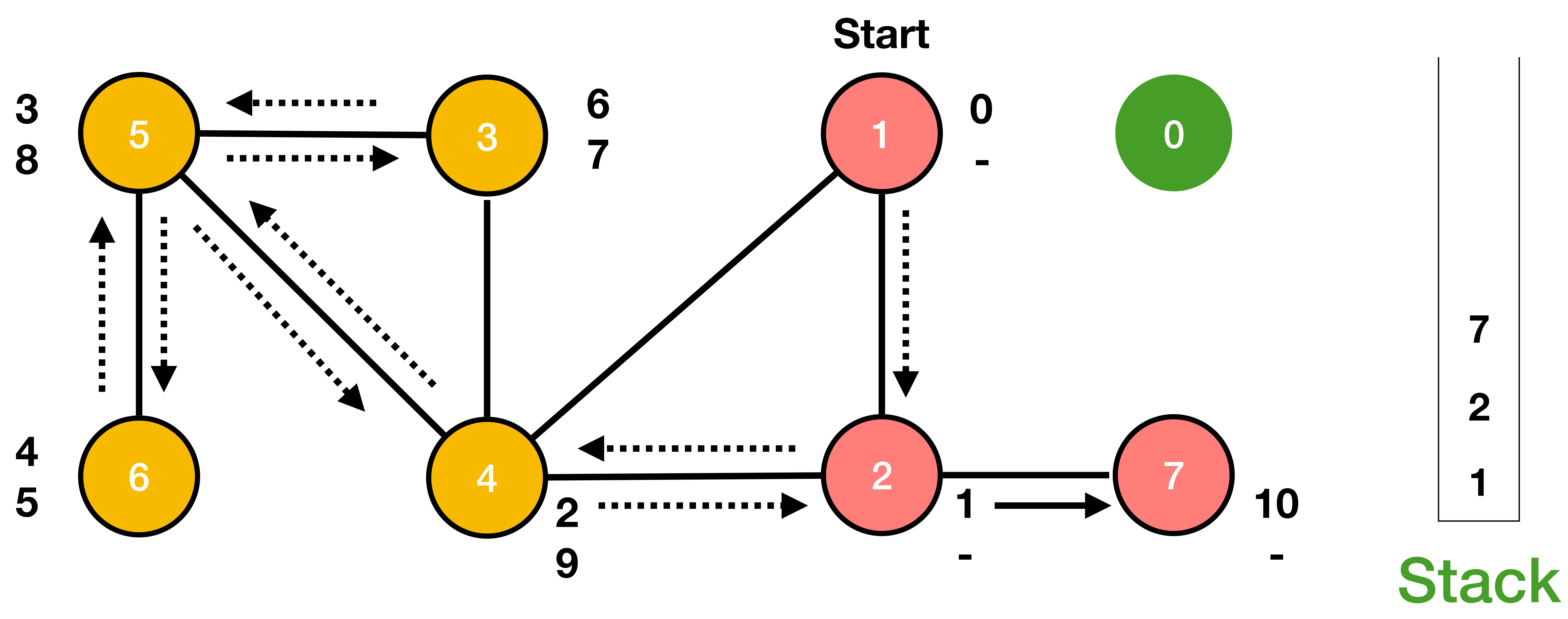
DFS using stack



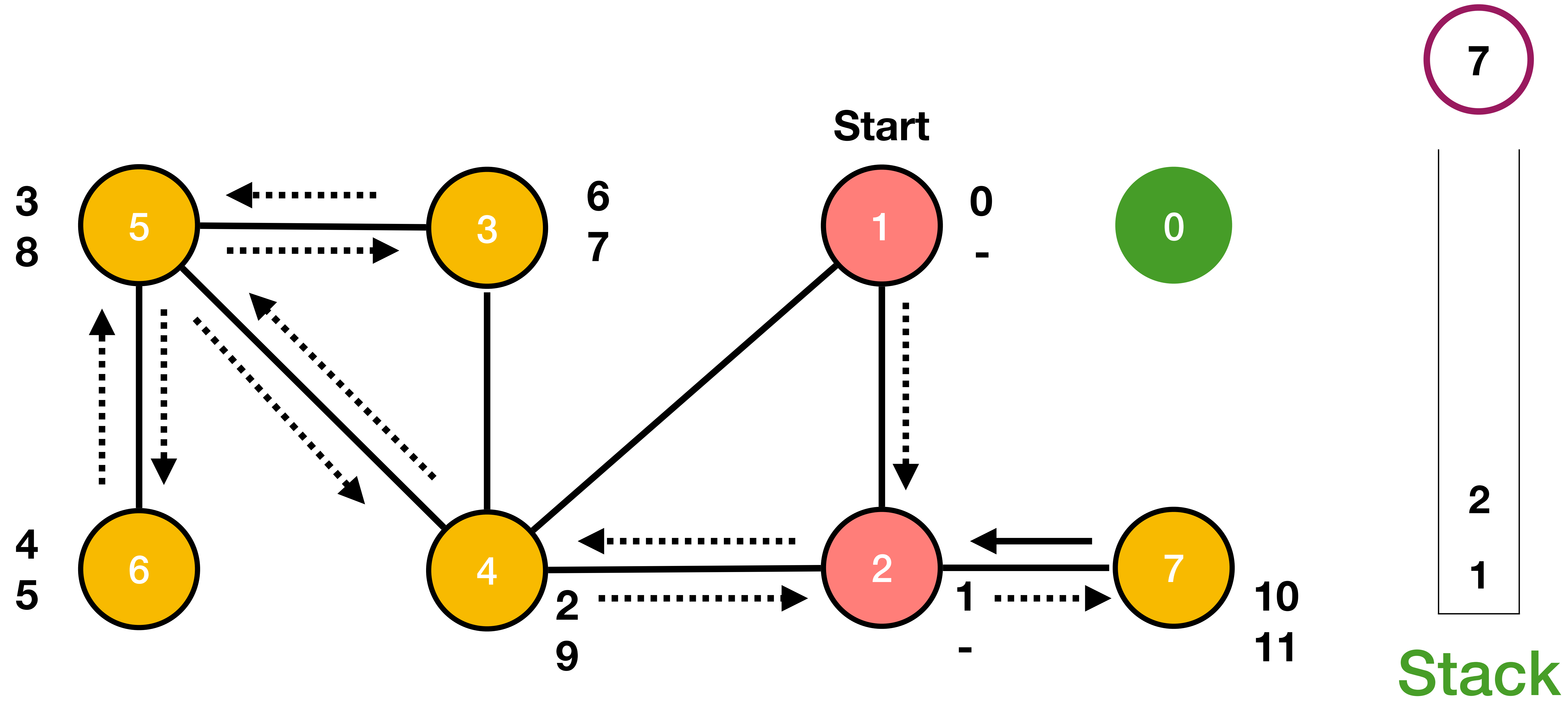
DFS using stack



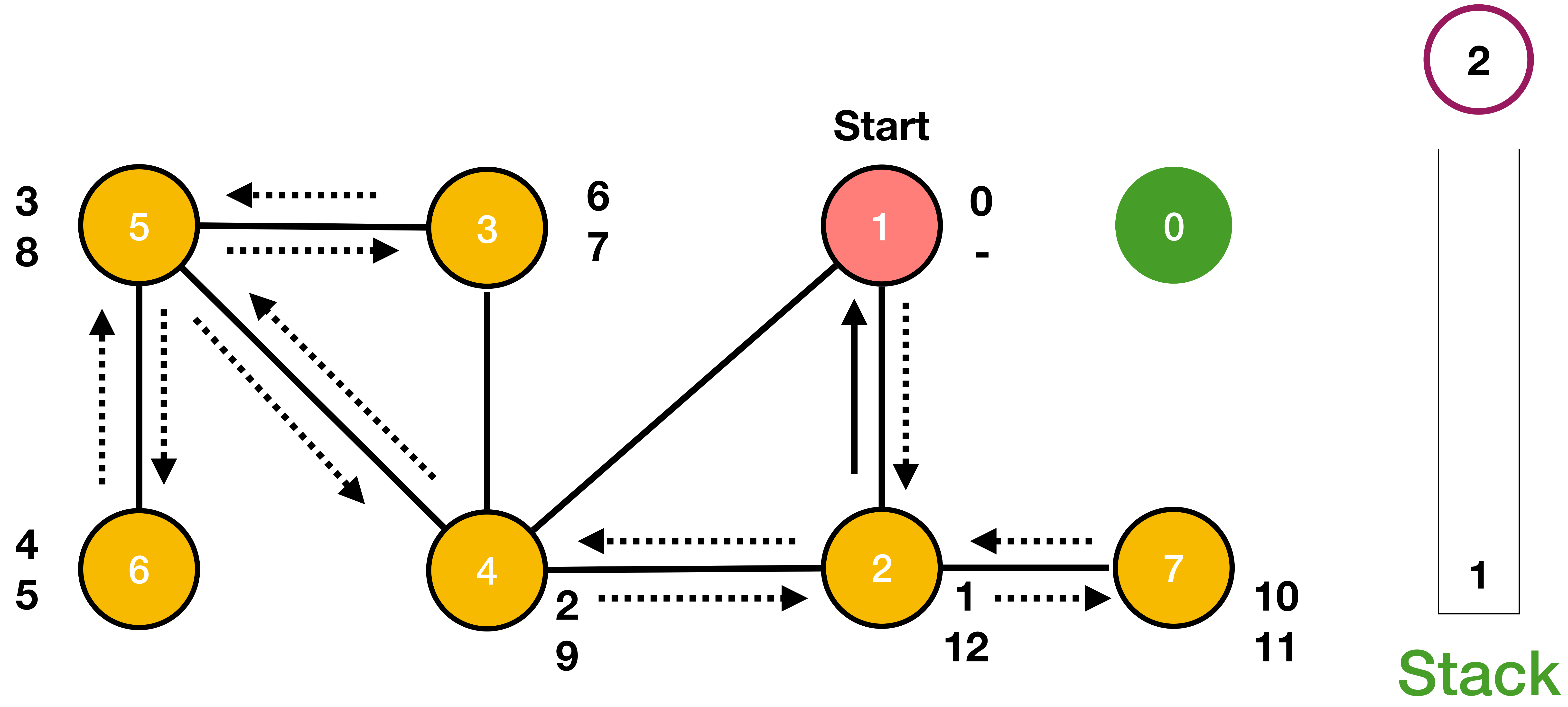
DFS using stack



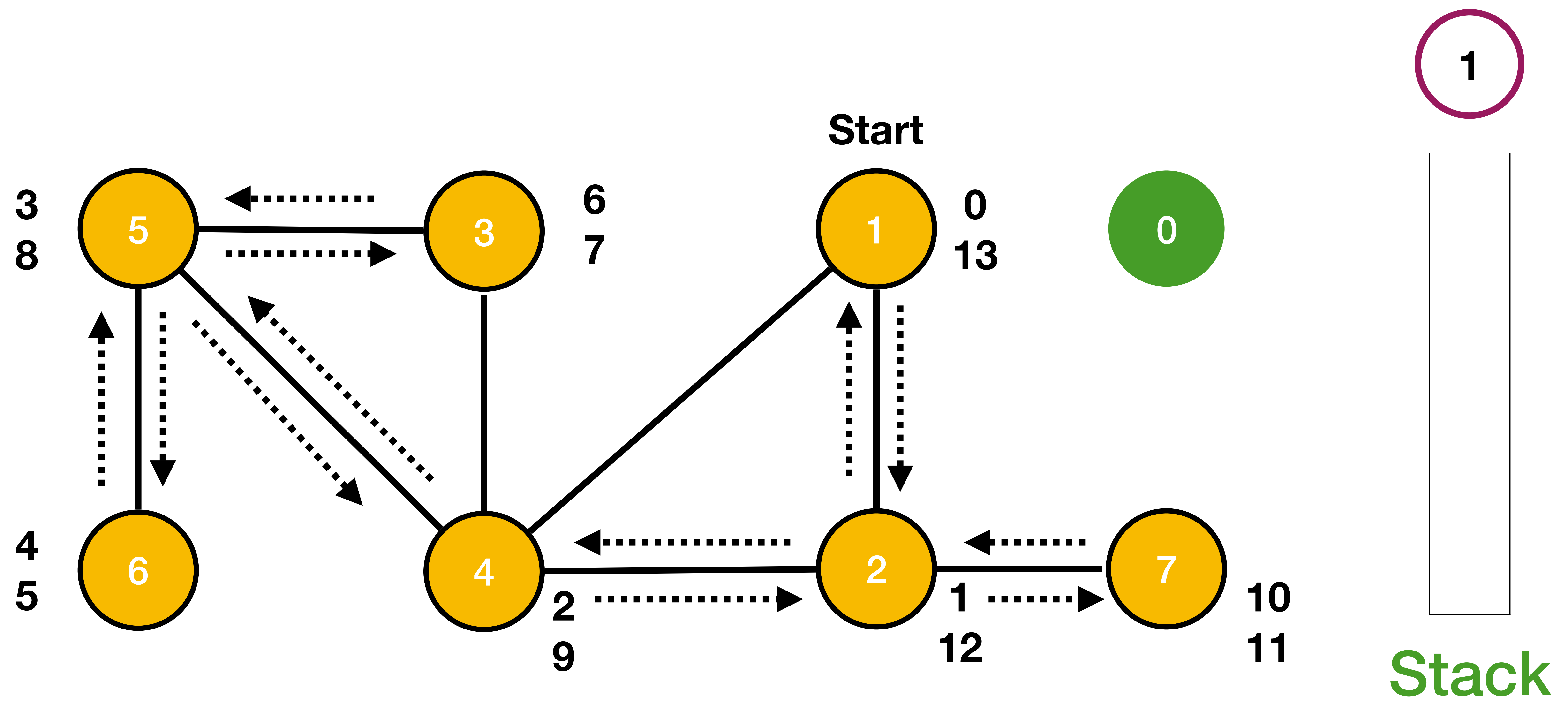
DFS using stack



DFS using stack

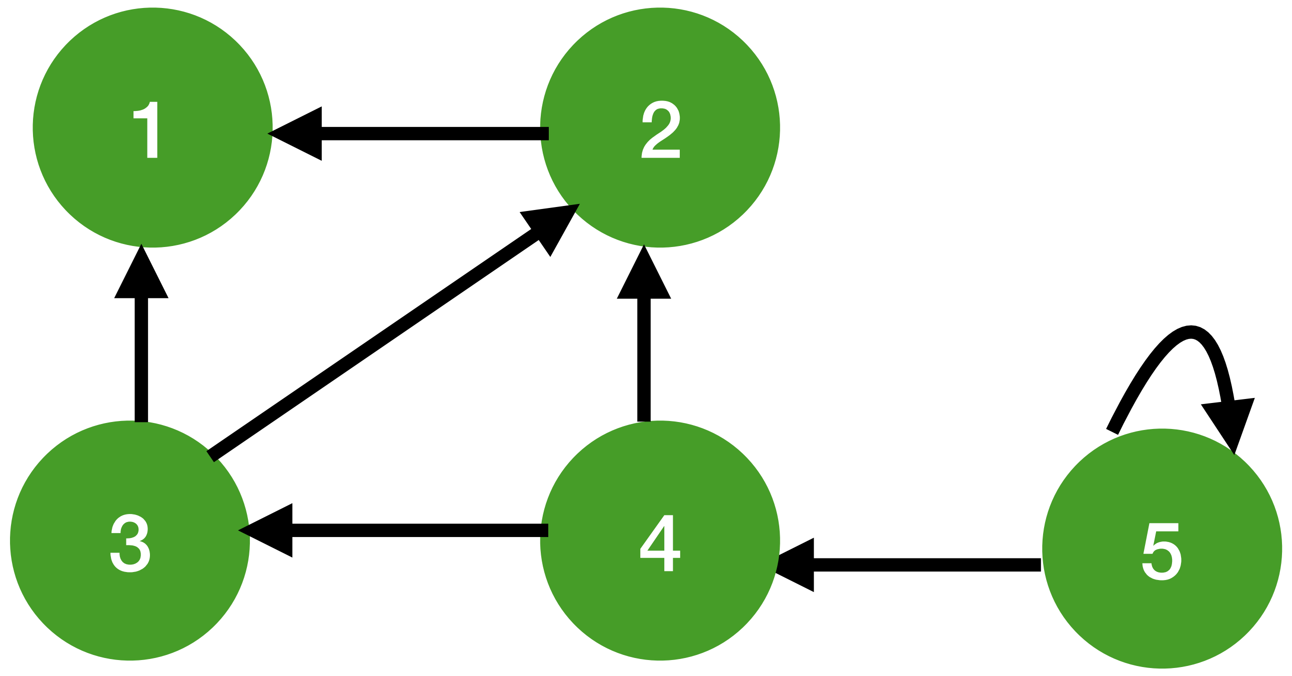


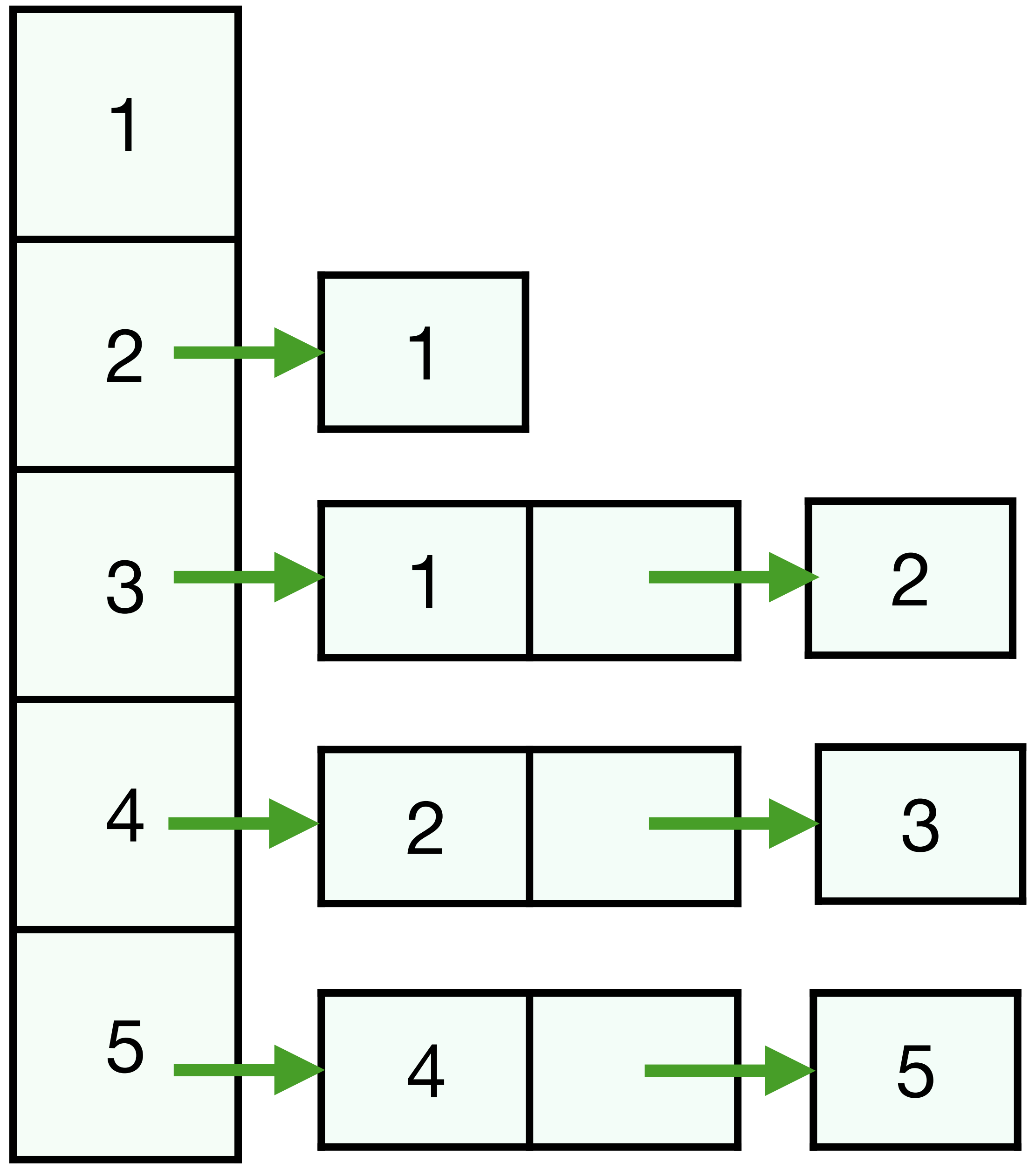
DFS using stack



Exercise

Find the discovery and finishing times for DFS with stack starting from vertex 4.





Homework assignment: Given an algorithm to compute G^T from G using adjacency list. For each vertex i -> j in adjacency list for G Create entry j -> i in adjacency list for G^T

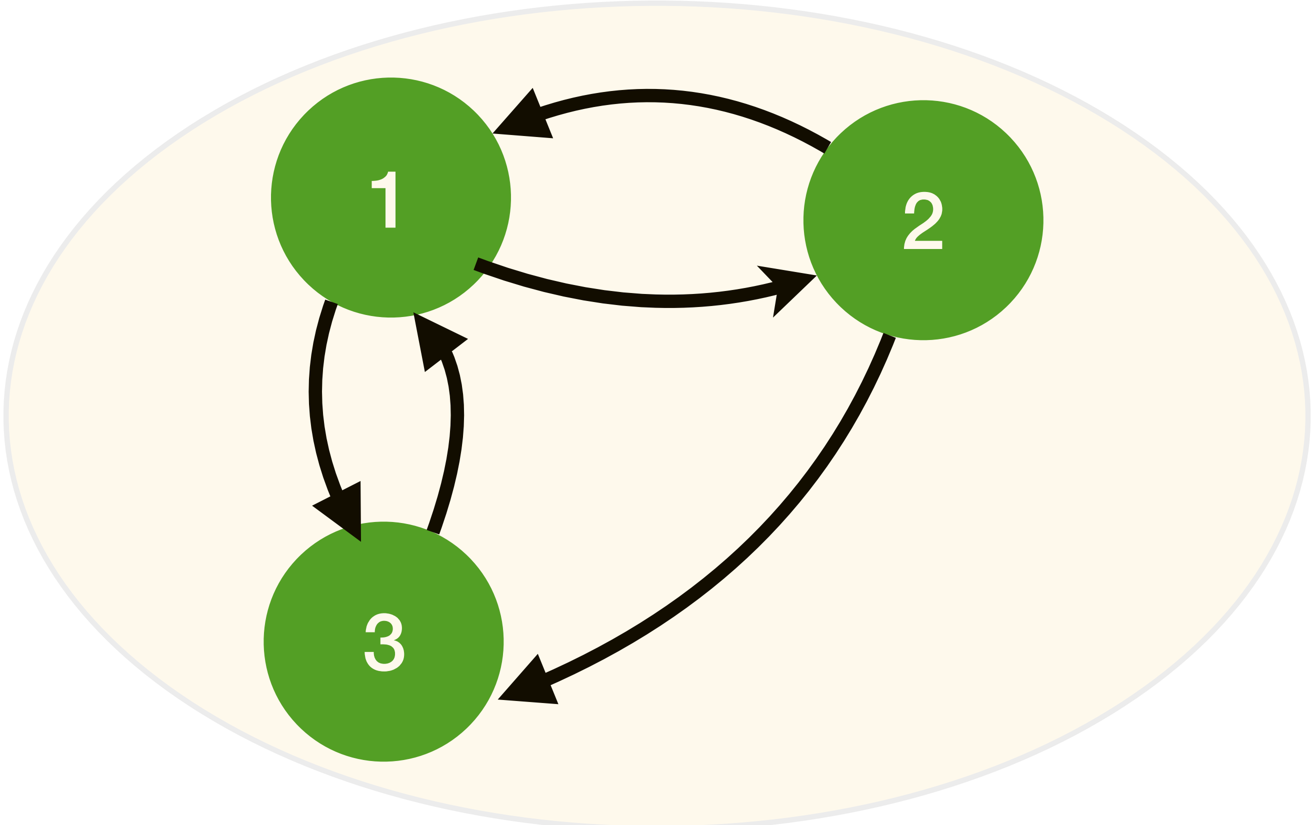
DFS With Stack Pseudocode

```
// s is the starting vertex, myGraph contains the graph, visited is 0
DFS(s, myGraph, visited) {
 stack s1;
 visited[s] = 1;
 s1.push(s);
 while (!s1.empty()) {
     s = s1.top(); // peek at vertex on top of stack
     v = get unvisited neighbor of s from myGraph;
     if (v exists) {
        visited[v] = 1;
        s1.push(v);   // push neighbor v on stack to explore
     }
     if (v does not exist)
       s1.pop();  // finished exploring s, remove from stack
 }
```

Strongly connected component (SCC)

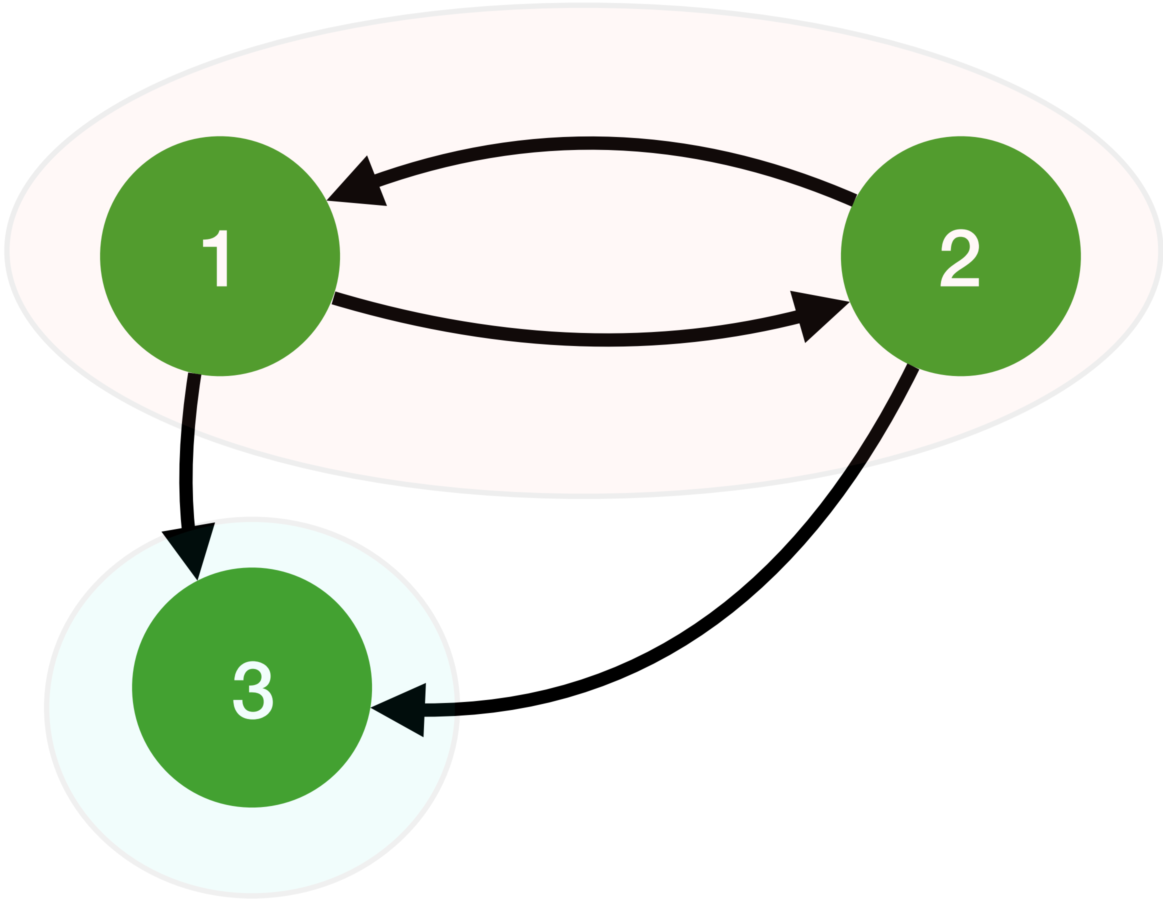
- SCC is a maximal set of vertices in G such that for every pair of vertices (u, v) it contains, there is a path from u to v and v to u.

- SCC is a cycle or individual vertex.



1 → 2 , 2 → 1
1 → 3 , 3 → 1
3 → 2 , 2 → 3

[1, 2, 3] is strongly connected

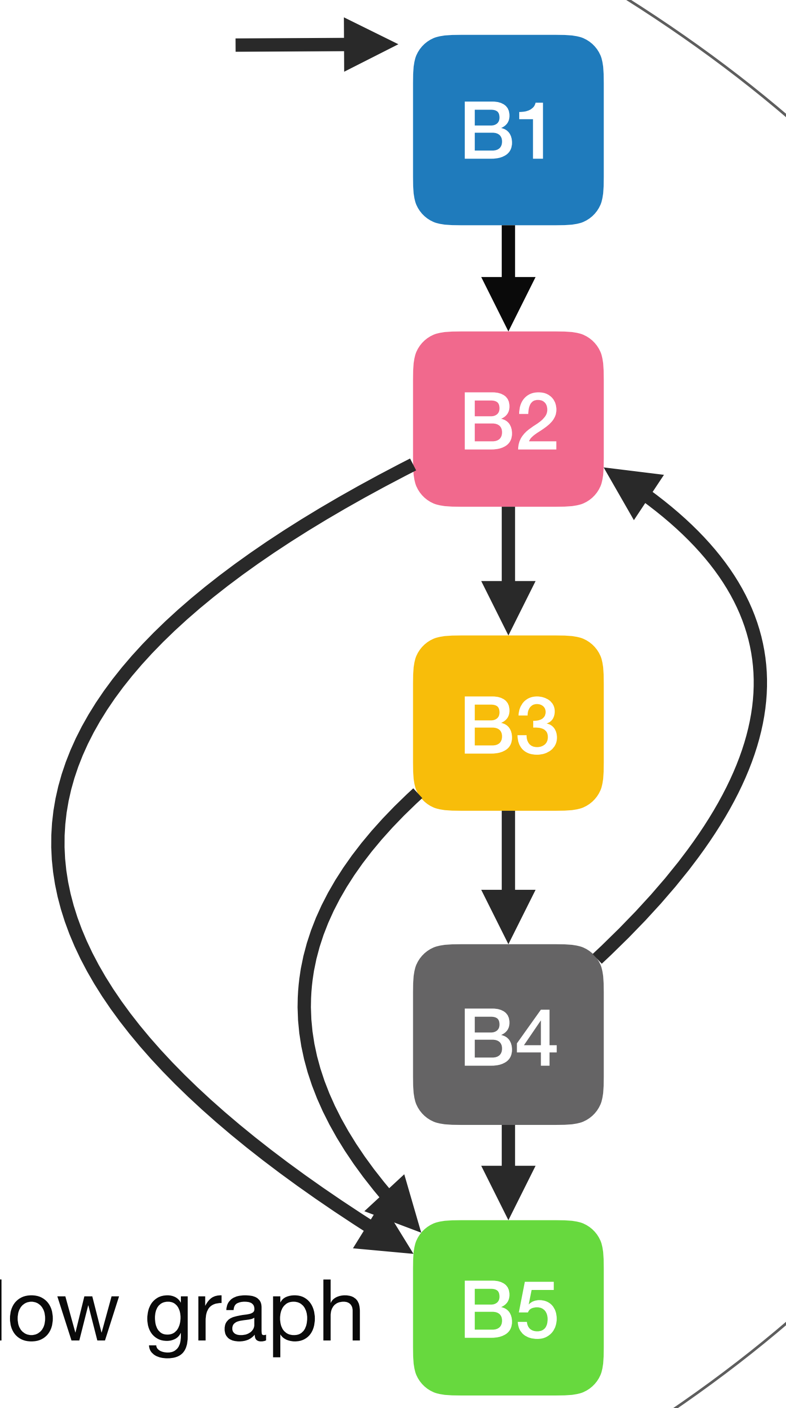


No path from 3 → 1 , 3 → 2

[1, 2, 3] is not strongly connected

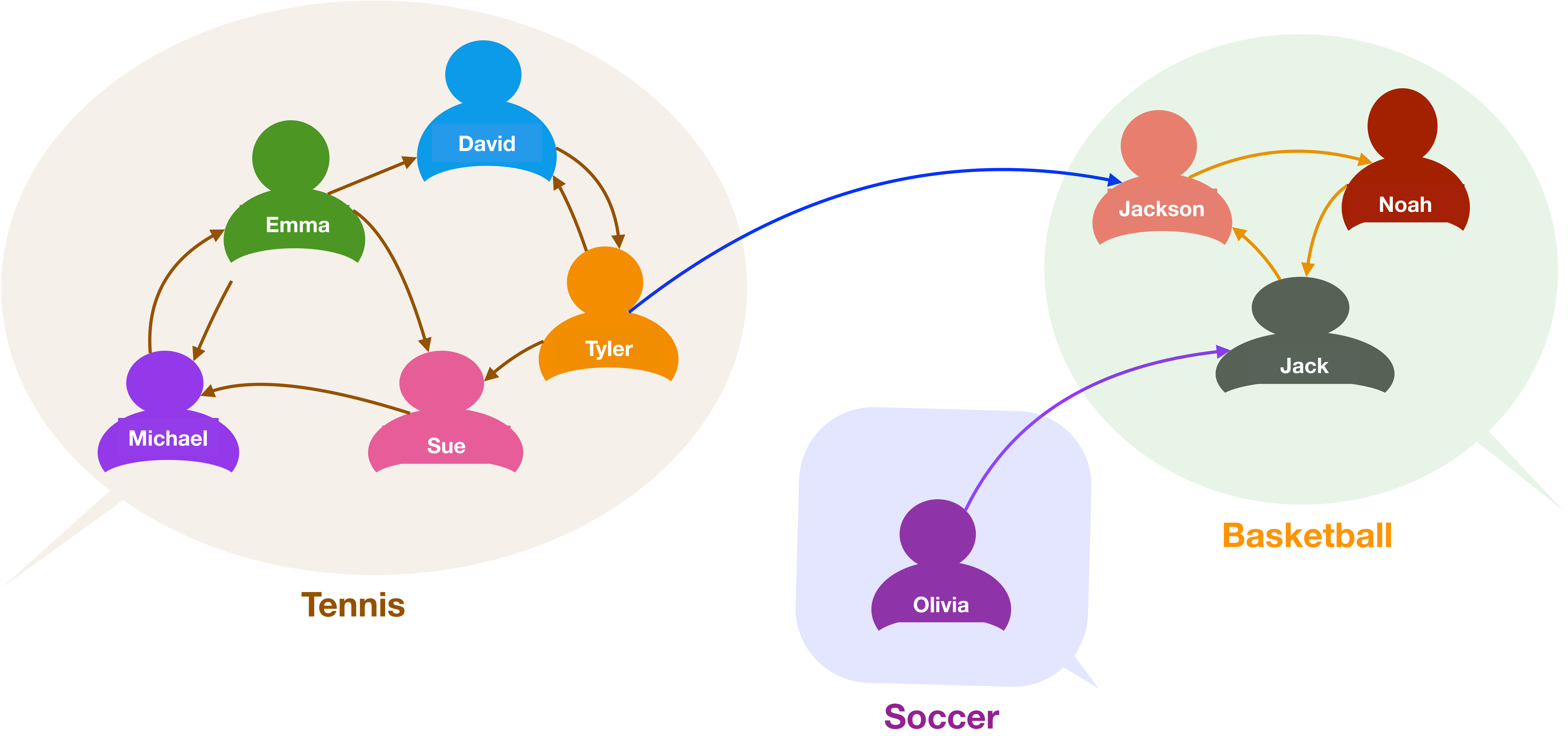
Application : Compiler

B1	a = 1; b = 0;
B2	L1 c = a + b ; if( ) go to L2
B3	d = a - b ; if( ) go to L2
B4	b = a + b ; if( ) go to L1
B5	L2



Flow graph

Application: Social network



Application : 2 - SAT

$$\varnothing = (x \lor y) \land (x \lor y') \land (x' \lor y) \land (x' \lor z')$$

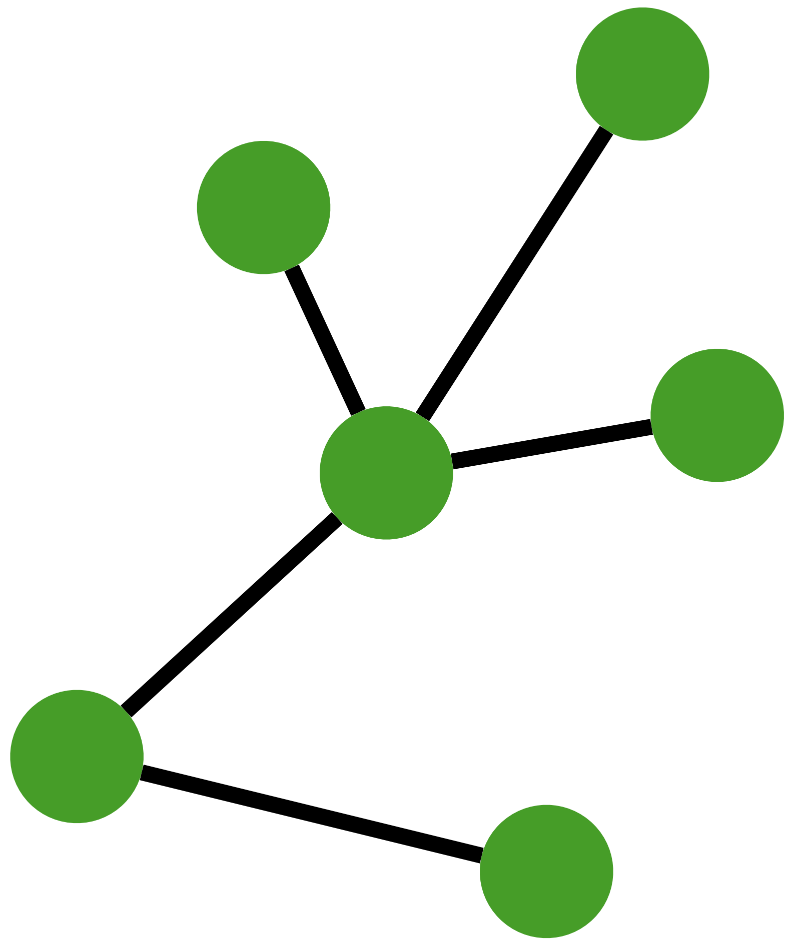
x, y and z have 2 possible values

Is there any assignment to x and y that makes ∅ satisfiable ?

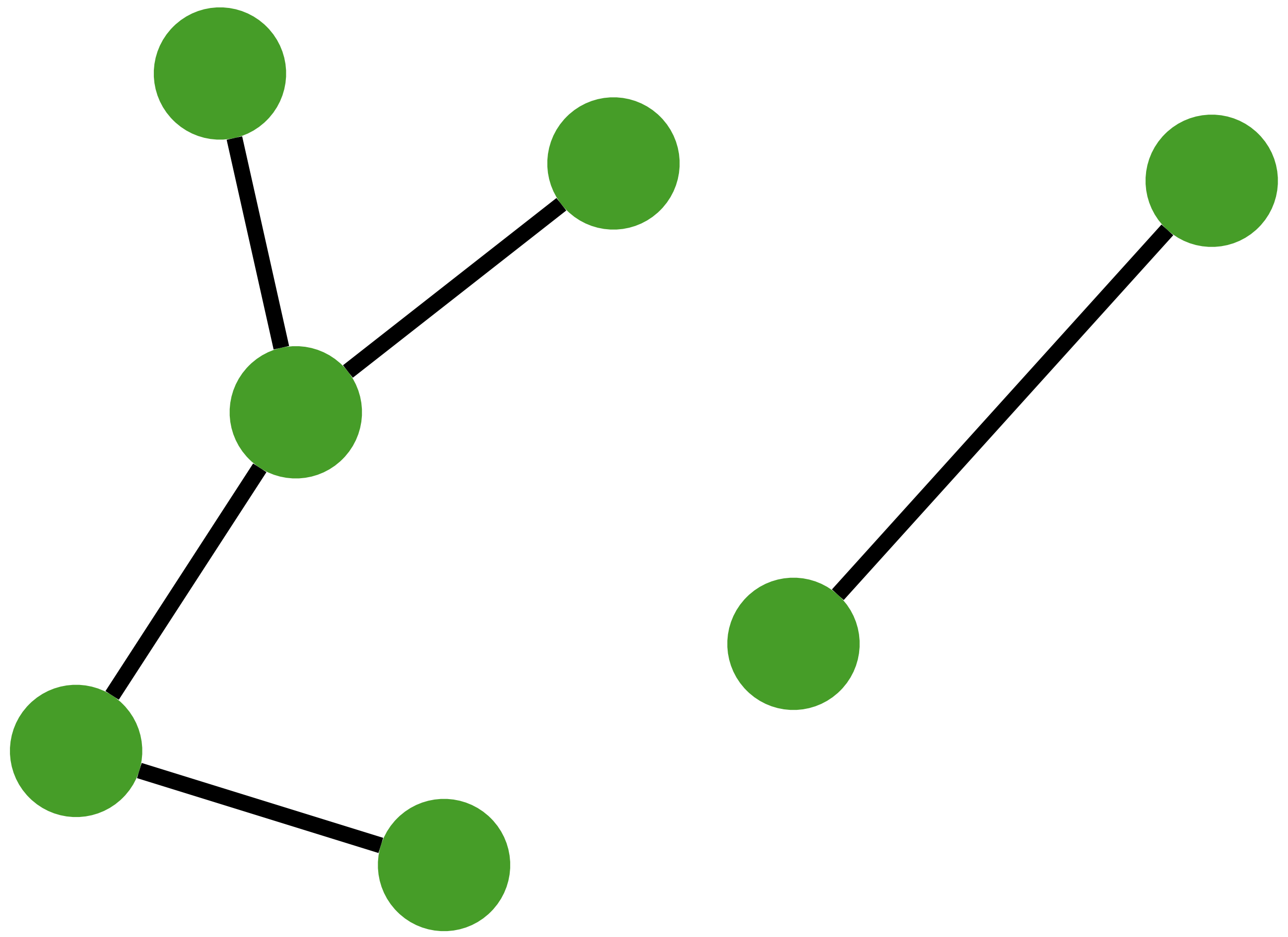
Definitions

Tree: an undirected graph with no cycles

Forest: disjoint union of trees



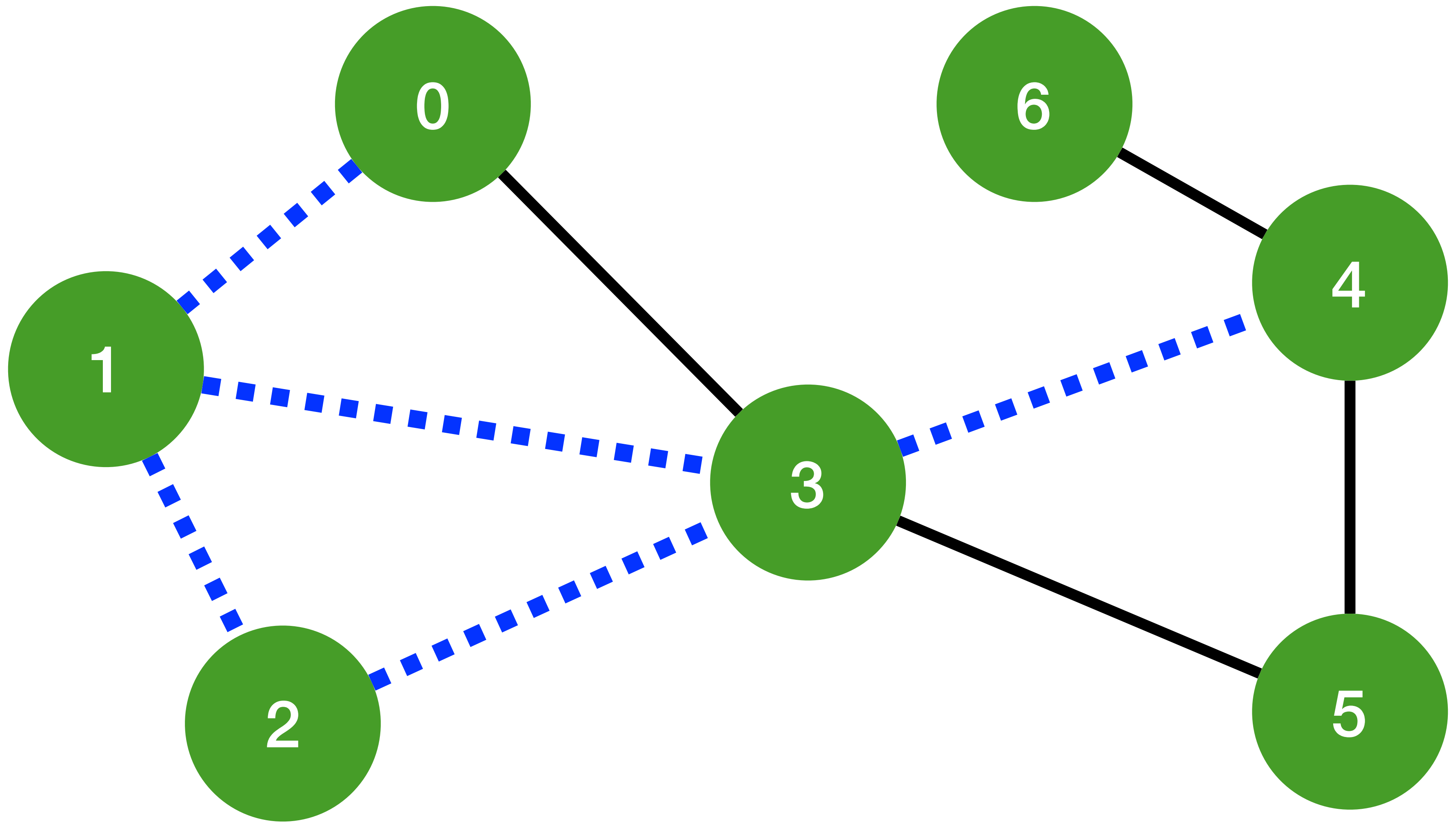
Tree



Forest

Walks

Any route from vertex to vertex

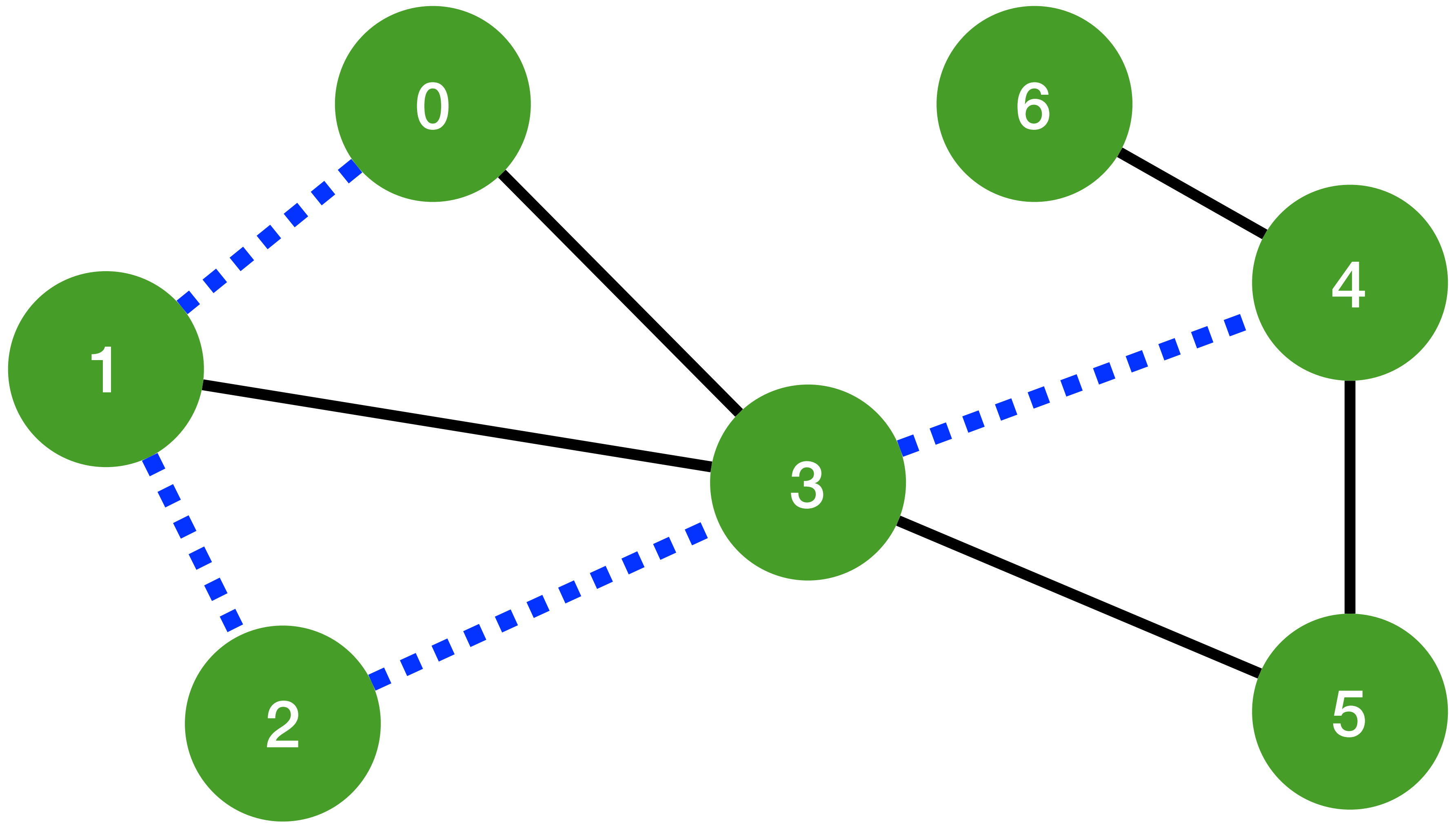


0 → 1 → 2 → 3 → 1 → 2 → 3 → 4

SPEL Technologies, Inc

Paths

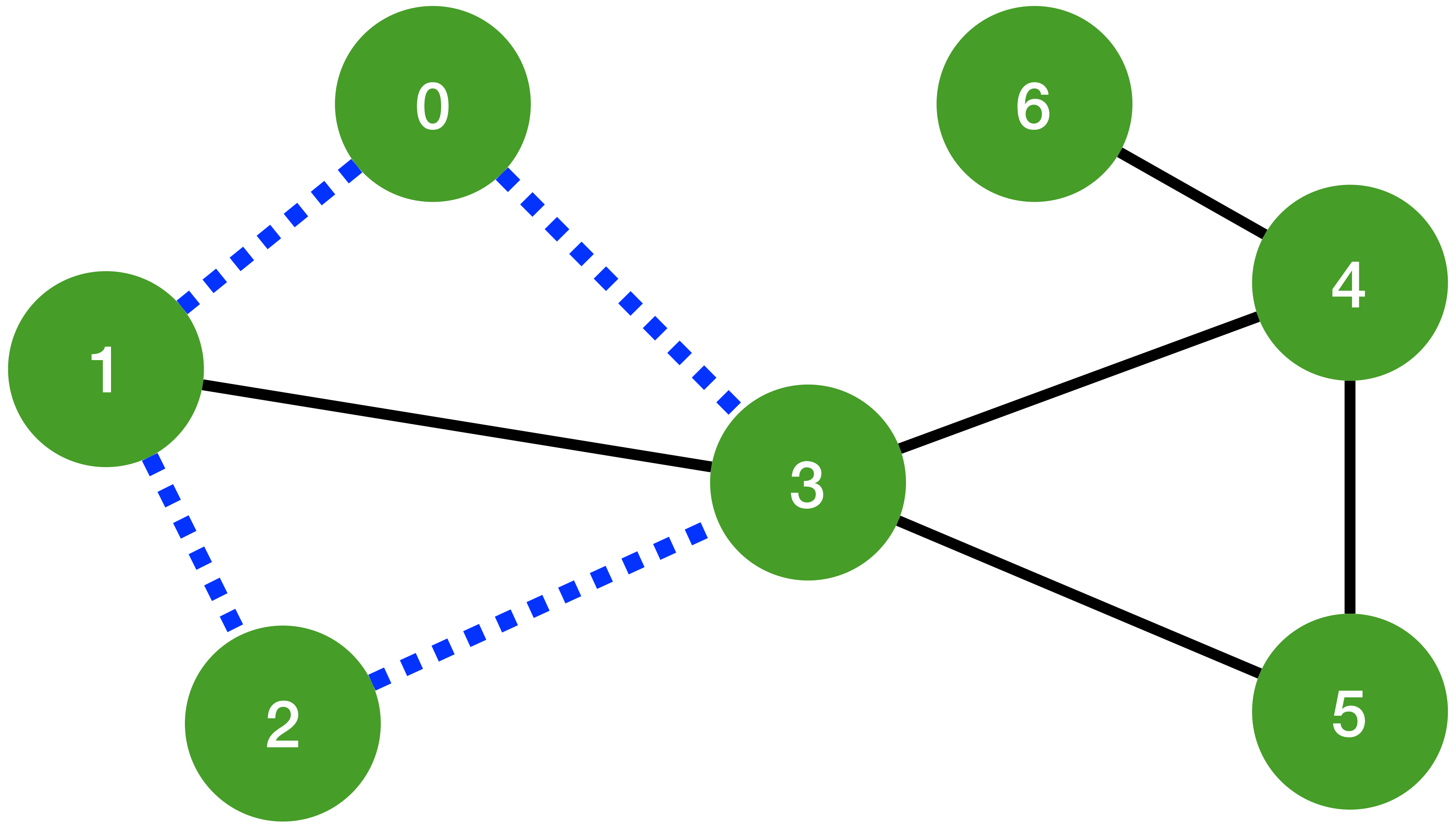
A walk that does not include any vertex twice (but starting vertex may be same as ending vertex)



0 → 1 → 2 → 3 → 4

Cycle

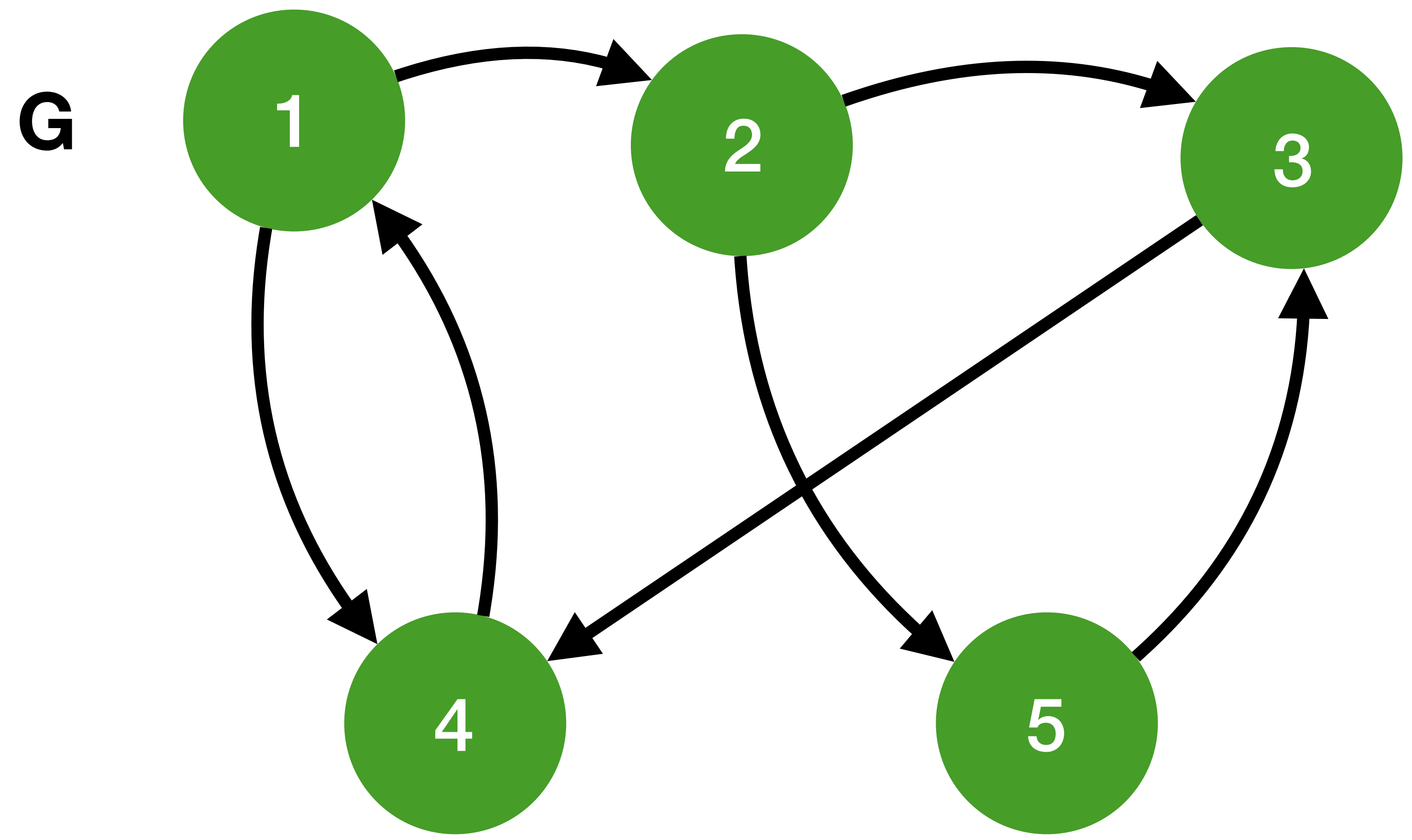
A path that begins and ends on the same vertex

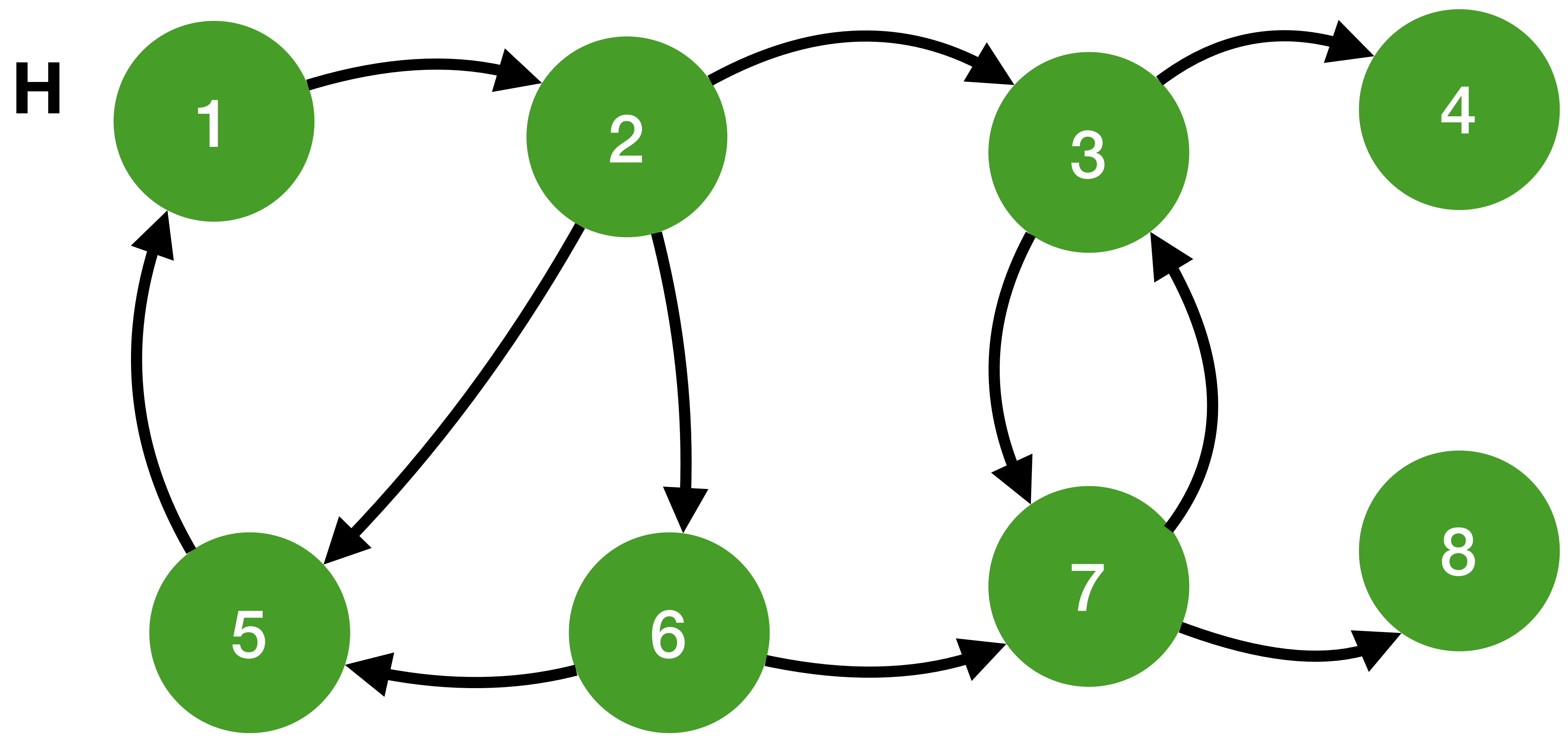


0 → 1 → 2 → 3

Exercise

Separate these graphs into strongly connected component

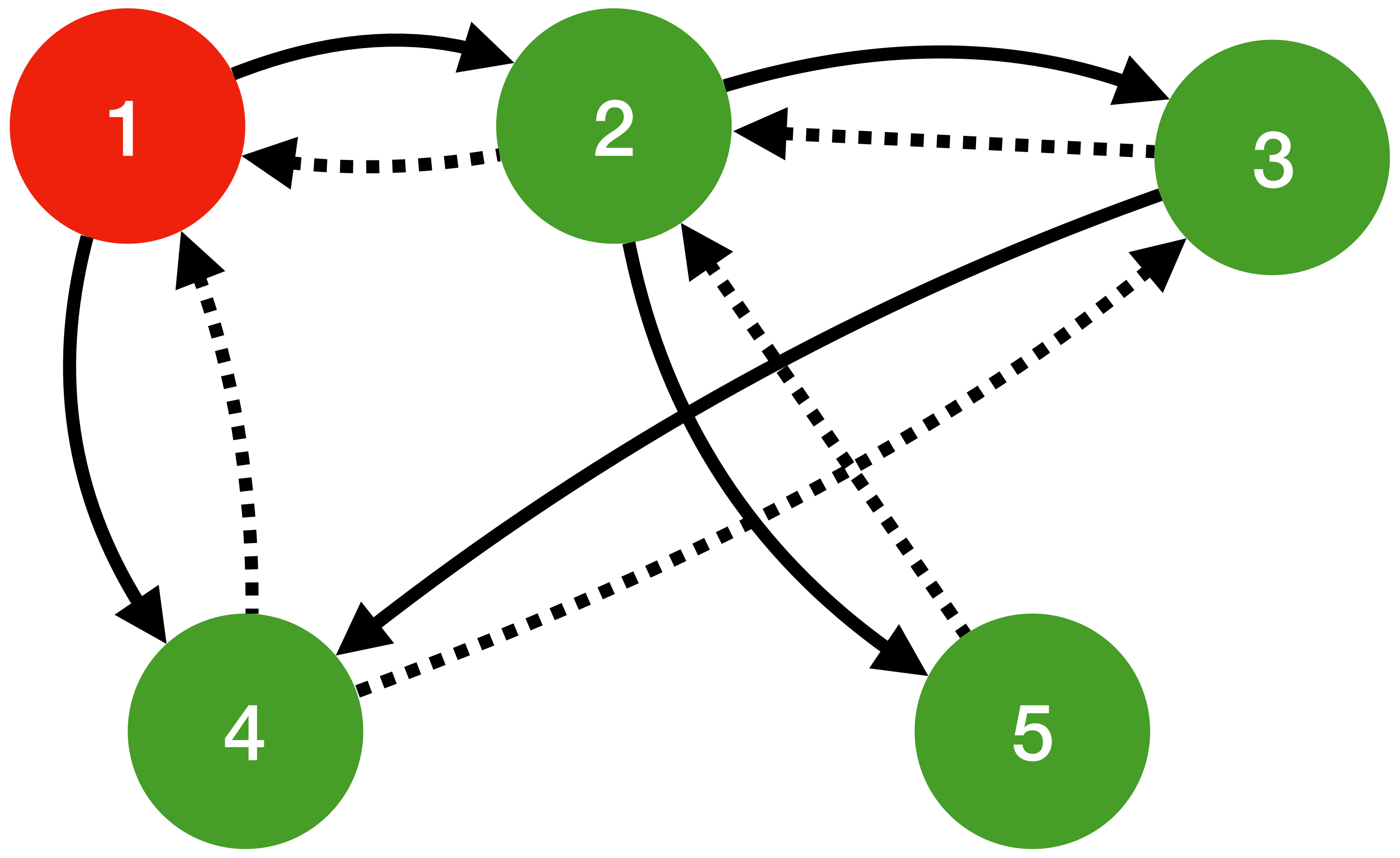




Finding an SCC

Simple DFS on G or G’ may not find the SCC.

DFS(G)

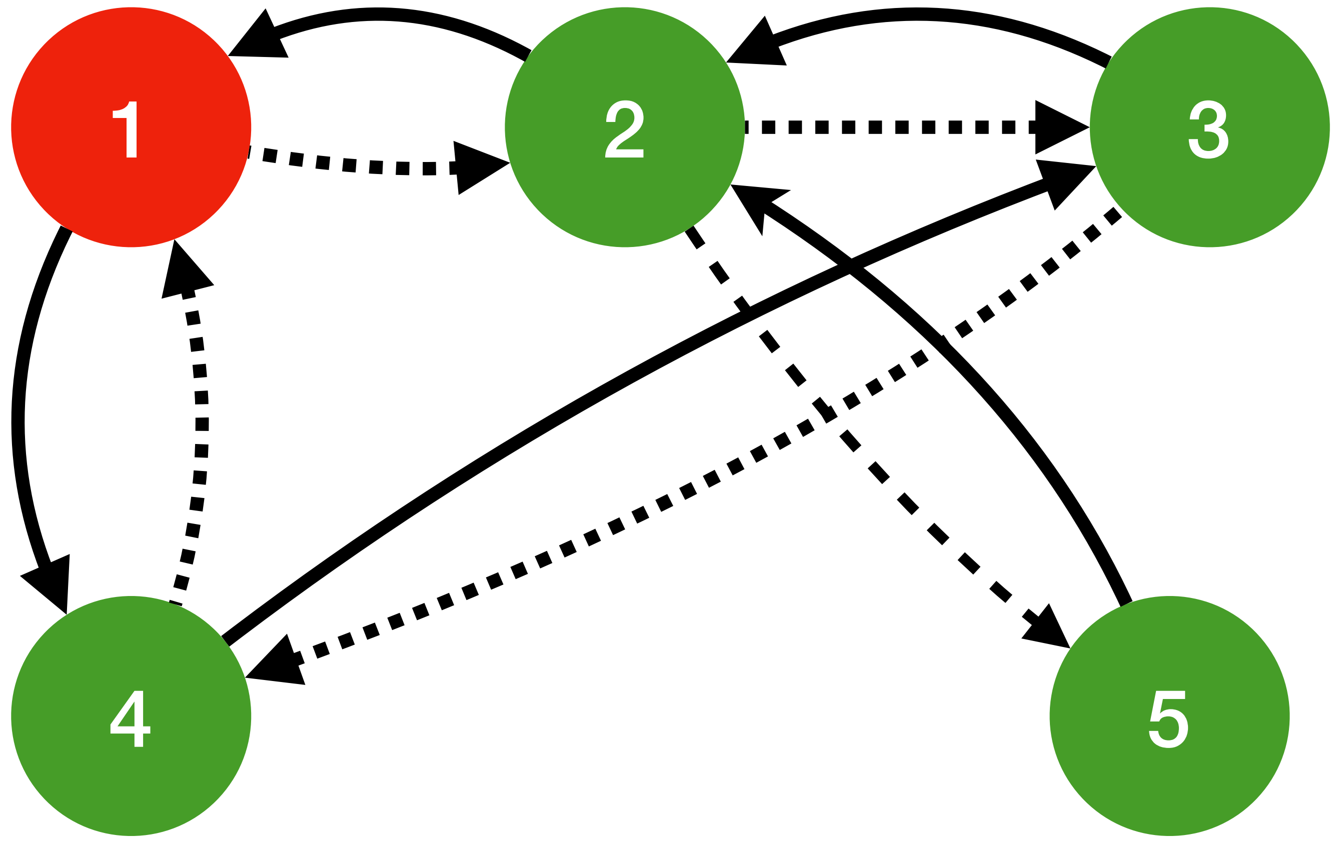


1 → 2 → 3 → 4 → 3 → 2 → 5 → 2 → 1

Does not discover the SCC :

1 → 2 → 5 → 3 → 4

DFS(G^T)



1 → 4 → 3 → 2 → 1 → 3 → 5 → 3

Does not discover the SCC :

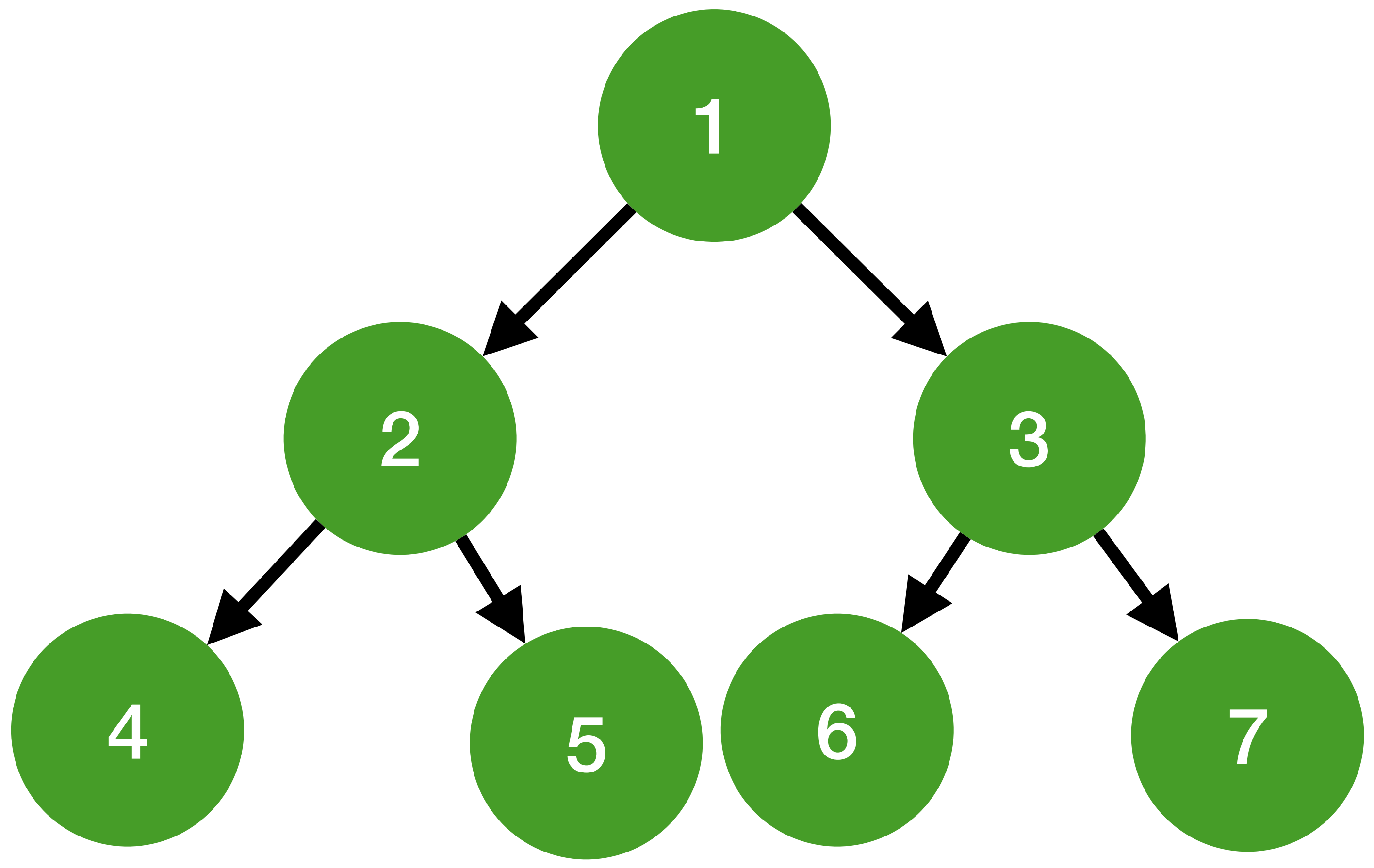
1 → 4 → 3 → 5 → 2

Kosaraju-Sharir Algorithm

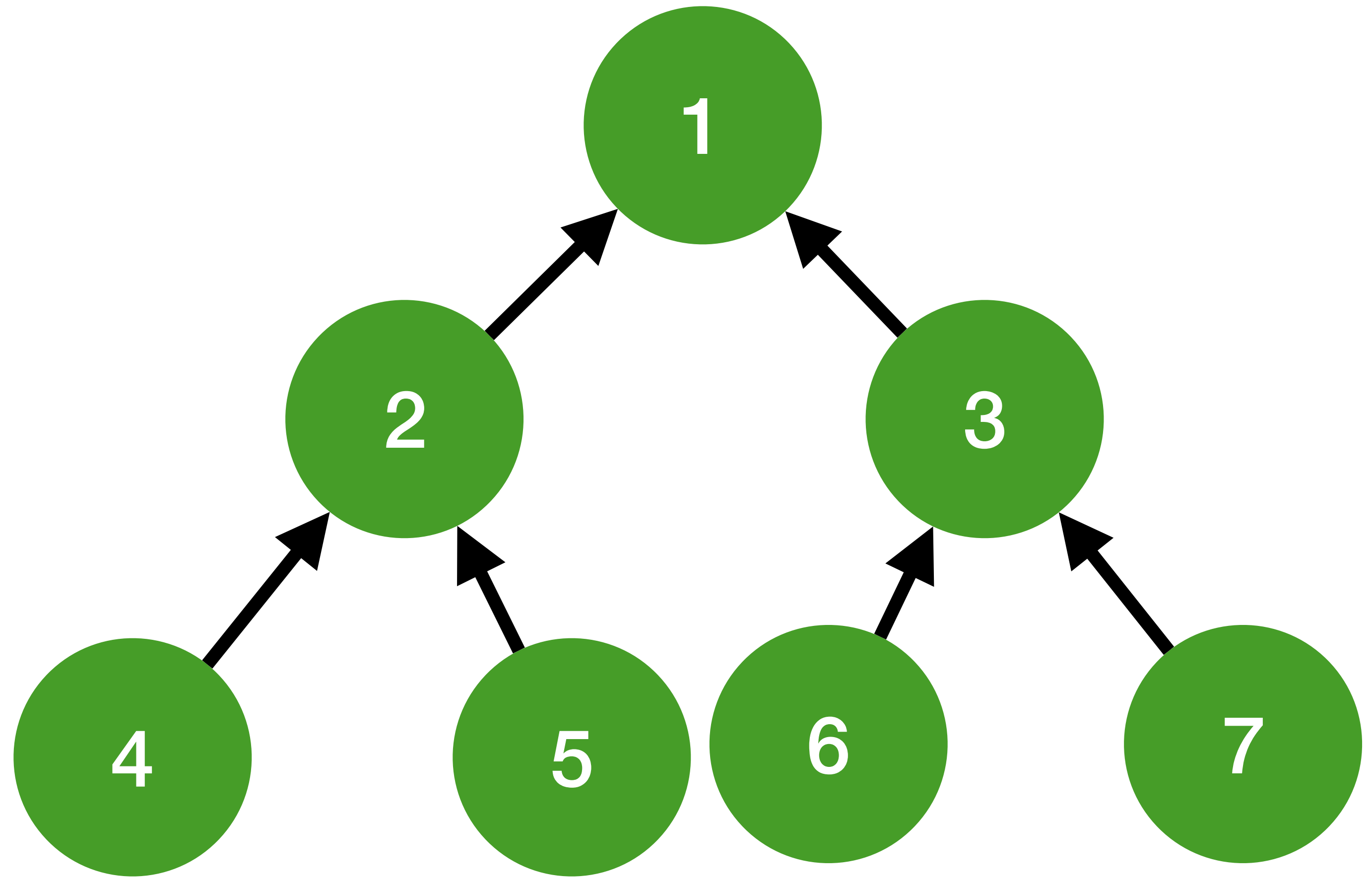
- Finds the SCCs in a graph.
- Based on the fact that the graphs of G and G^T have the same SCCs.
- Discovered independently by Kosaraju and Sharir.

Kosaraju-Sharir: Intuition 1

Tree T1

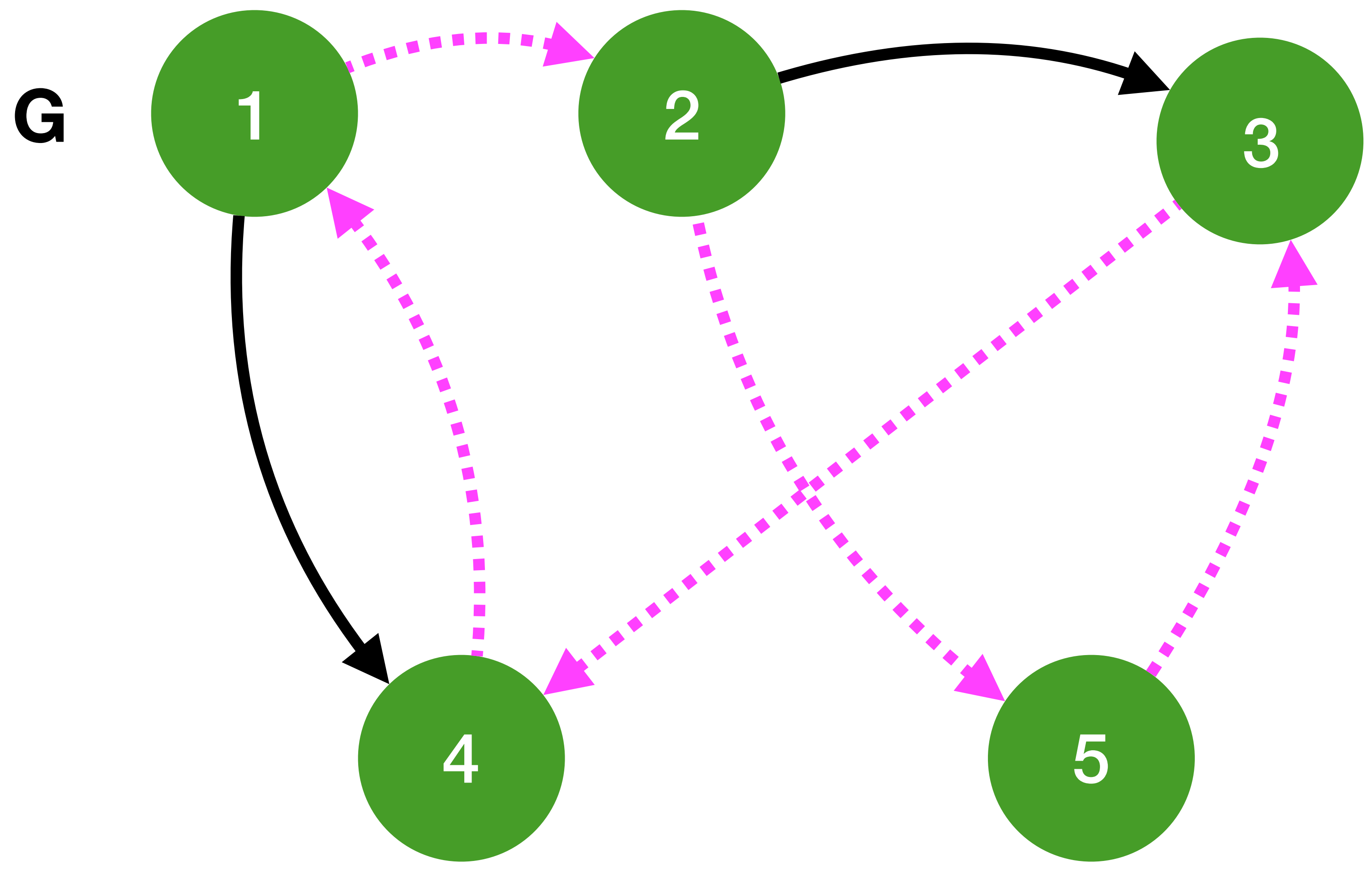


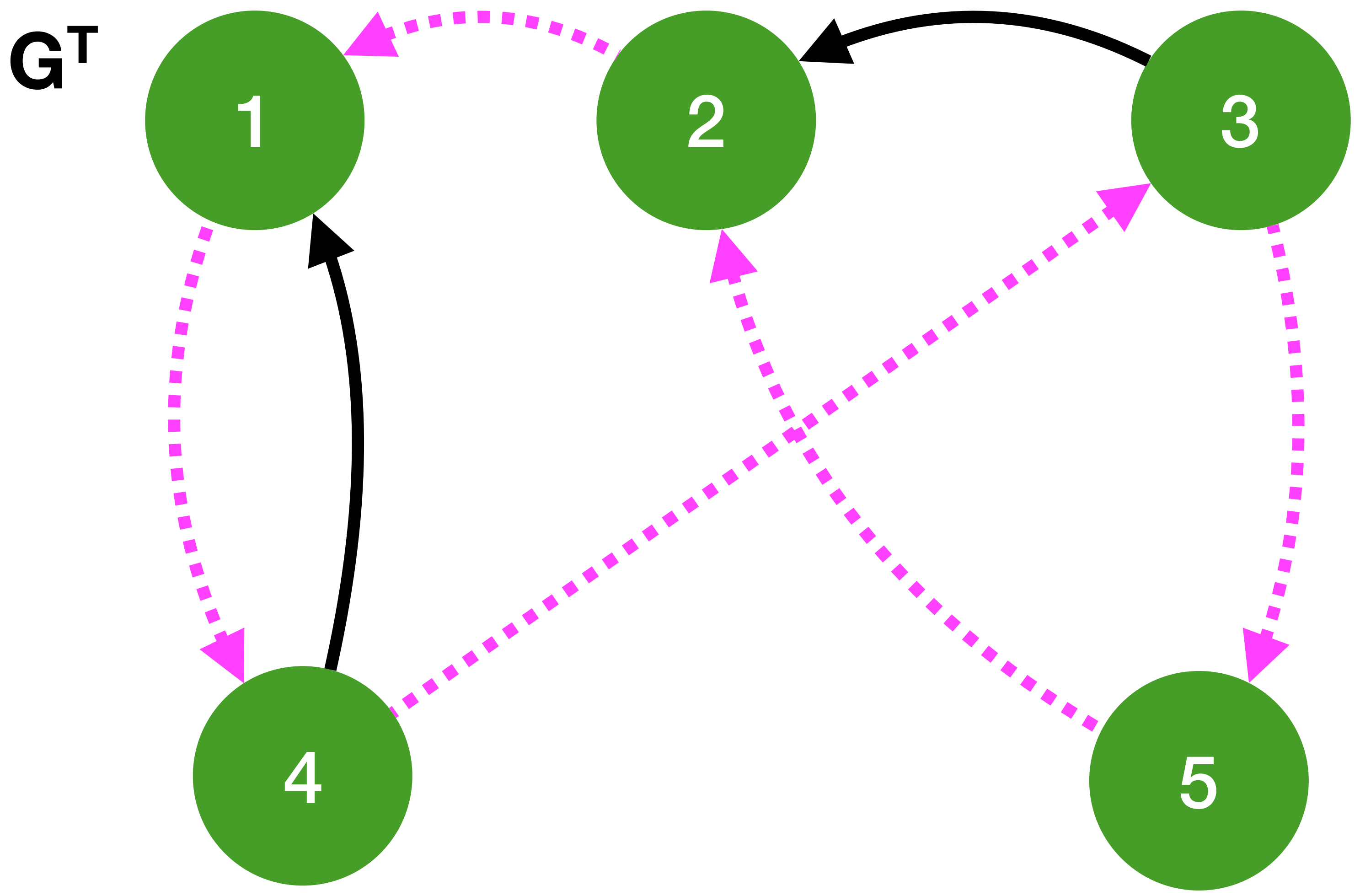
Transpose of Tree T1



1 → 2 → 4 → 2 → 5 → 2 → 1 → 3 → 6 → 3 → 7

Graph transpose



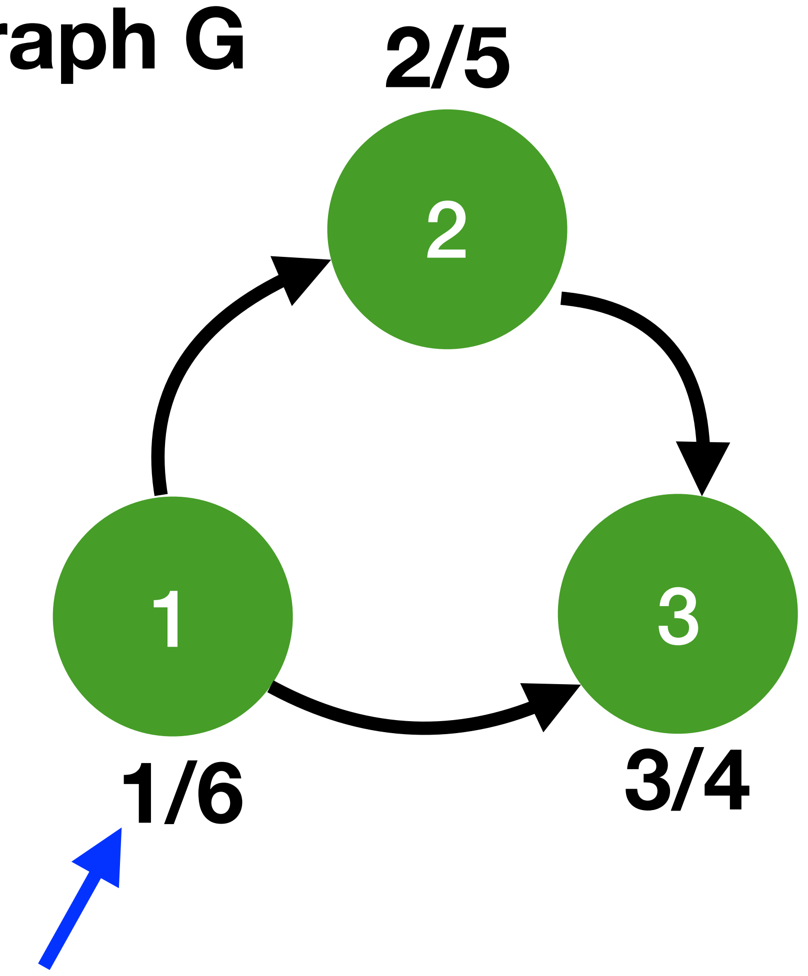


1 → 2 → 5 → 3 → 4 → 3 → 5 → 2 → 1

1 → 4 → 3 → 5 → 2 → 5 → 3 → 4 → 1

Kosaraju - Sharir algorithm
Intuition 2

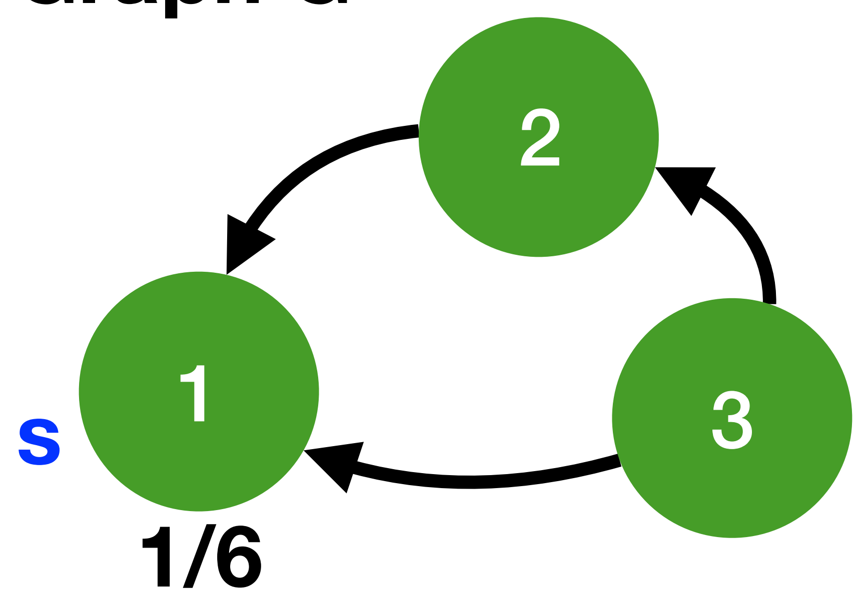
Graph G



discovery time / finishing time

Start DFS from 1 - visits [1, 2, 3].

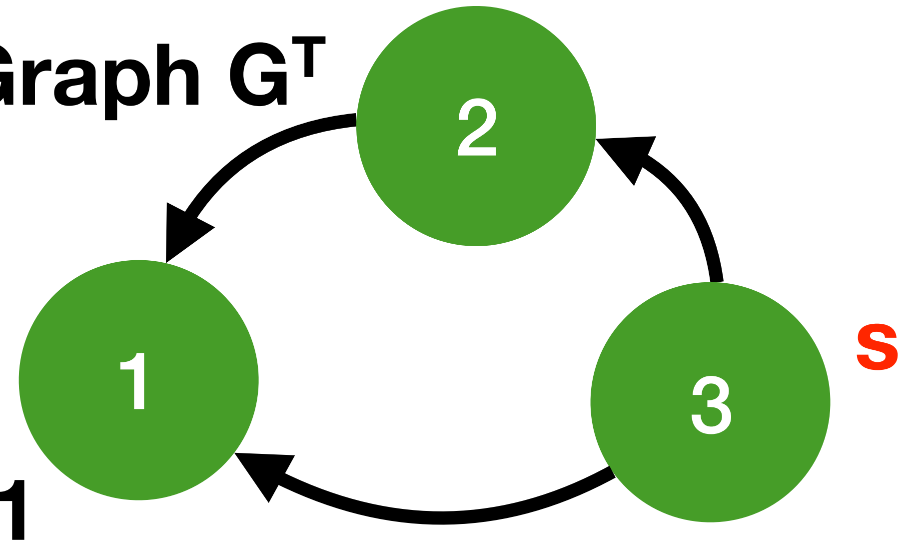
Graph G^T



Do DFS for G^T from ***same*** starting vertex as G

No way to visit all vertices => [1,2,3] not SCC

Graph G^T



Incorrect result if we do DFS on G^T with a different starting vertex (finishing time is not highest)

Choose 3 to start DFS

3 → 2 → 1 → 2 → 3 → 1

Able to visit all vertices => [1, 2, 3] is SCC ERROR !

SPEL Technologies, Inc

Idea for Kosaraju - Sharir algorithm

1. Perform DFS on G to find finishing time for all vertices
2. Transpose G (G^T) by selecting start vertex
3. Perform DFS on G^T in order of decreasing finishing times

Kosaraju - Sharir algorithm

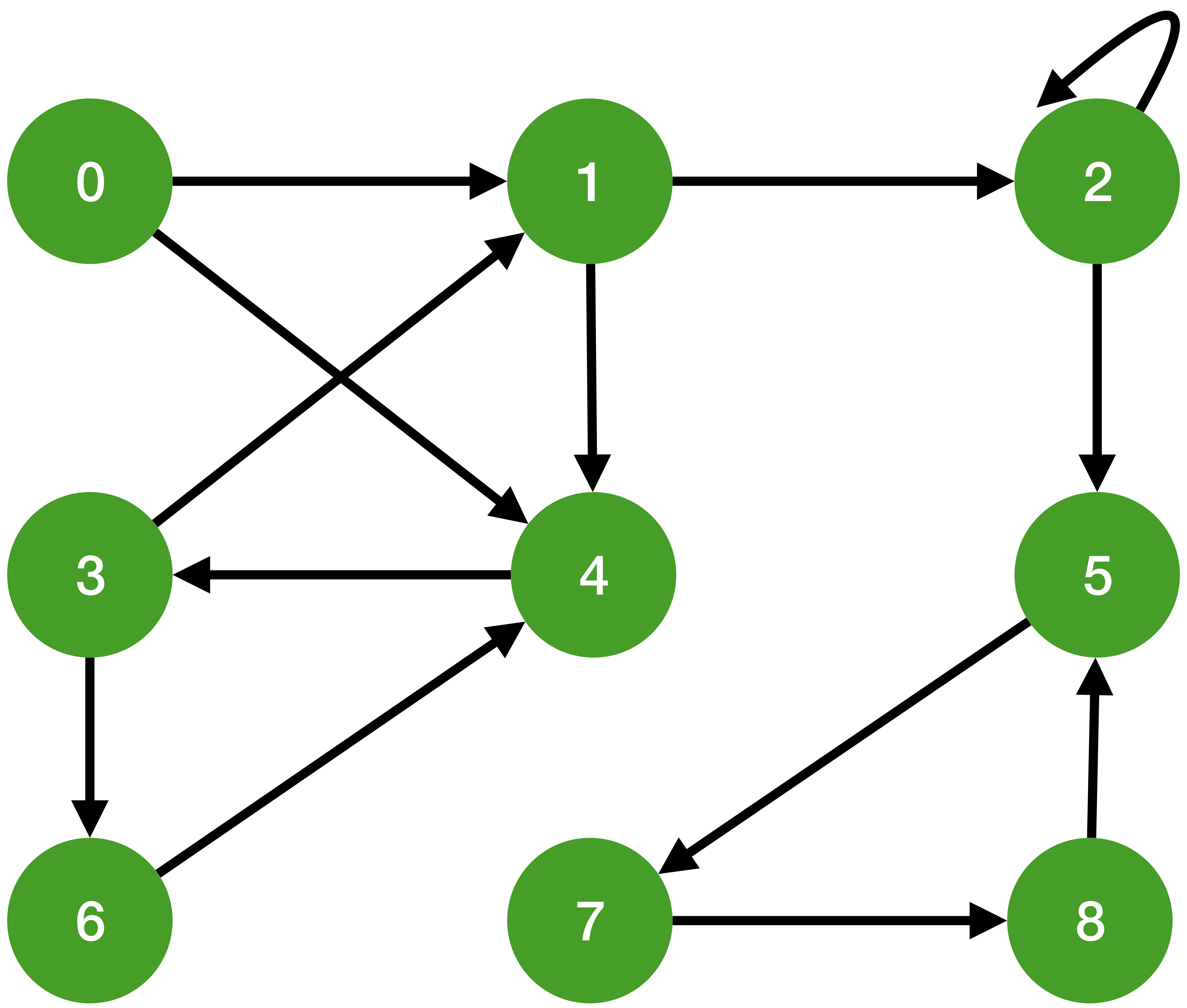
Find the SCC of directed graph G

1. Call depthFirstSearch(G), find finishing time to f(v) for each vertex v
2. Calculate transpose of G: G^T
3. Call depthFirstSearch(G^T) on vertices ordered by decreasing f(v) found in step 1
4. Print out each SCC

Each vertex v is pushed into a stack when f(v) is completed , so the stack holds all vertices in decreasing of f(v)

Kosaraju - Sharir algorithm

G

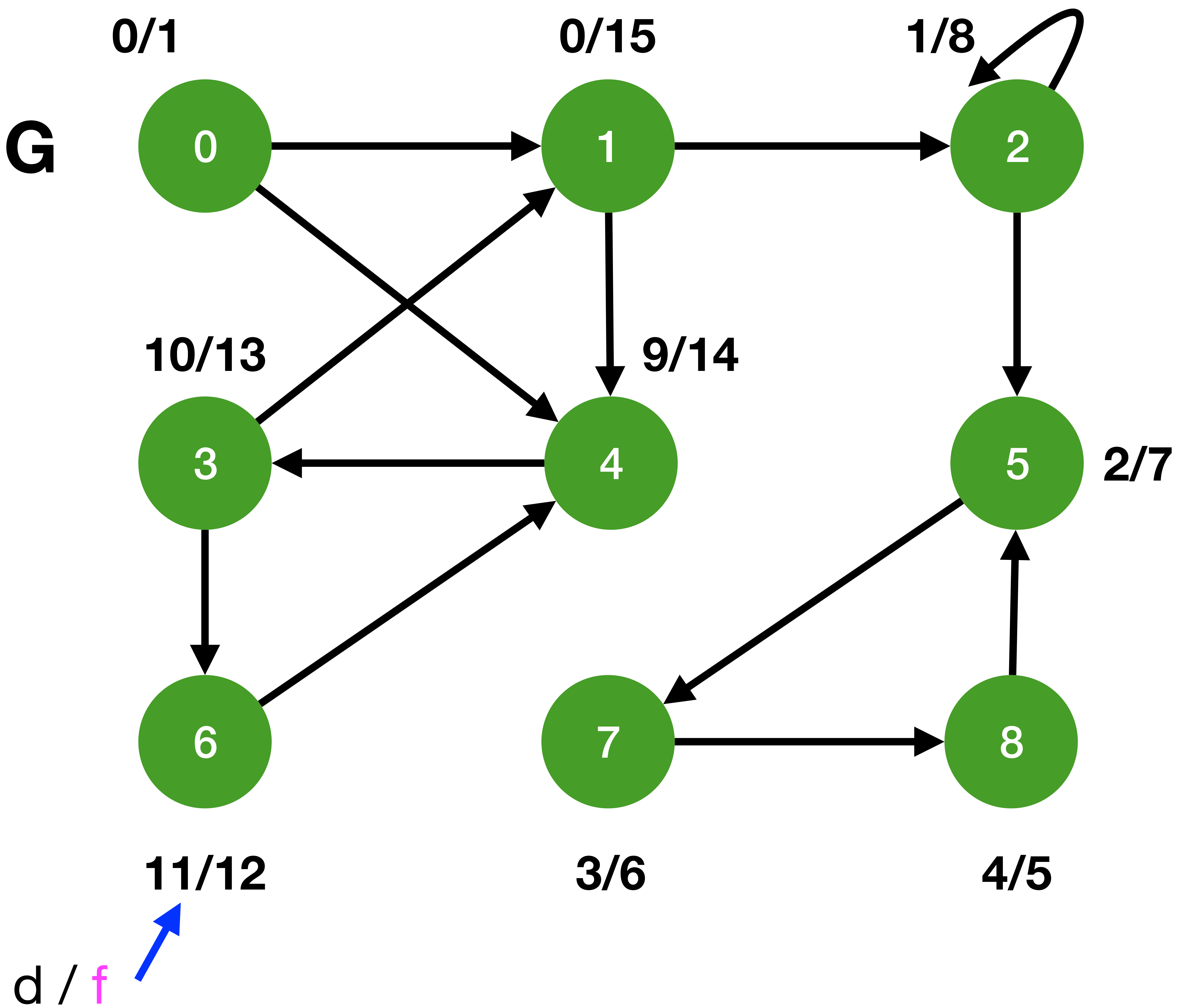


0	4 ,1
1	2, 4
2	5, 2
3	6, 1
4	3
5	7
6	4
7	8
8	5

Stack

adjacency list

Kosaraju - Sharir algorithm



Stack (f ↓)
1
4
3
6
2
5
7
8
0

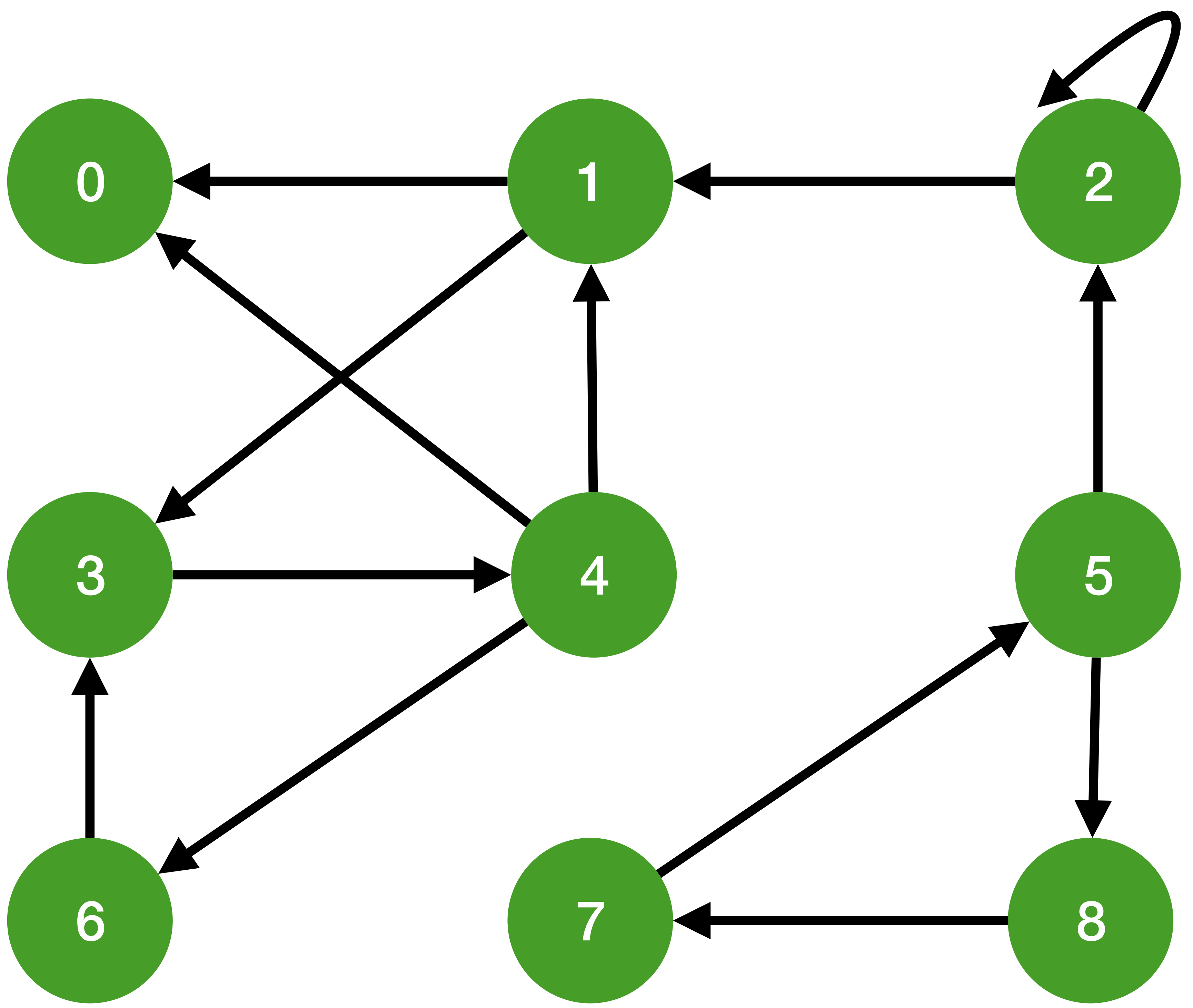
Stack

0	4 ,1
1	2, 4
2	5, 2
3	6, 1
4	3
5	7
6	4
7	8
8	5

adjacency list

Kosaraju - Sharir algorithm

G^T



G^T - Transpose of G

f

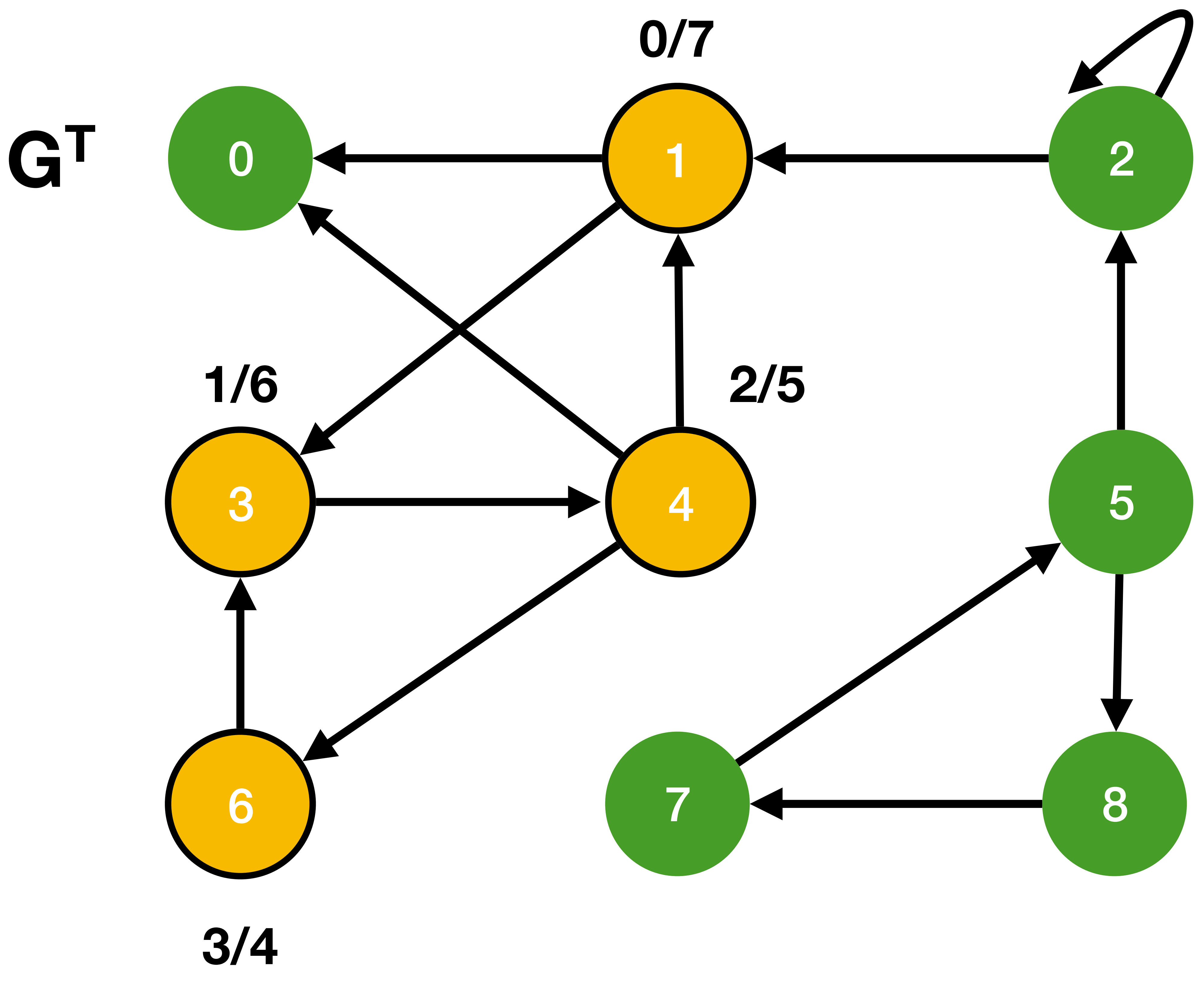
Stack
1
4
3
6
2
5
7
8
0

Stack

0	
1	3, 0
2	2, 1
3	4
4	6, 1, 0
5	8, 2
6	3
7	5
8	7

adjacency list

Kosaraju - Sharir algorithm



Visited 1, 3, 4, 6 with DFS

1

f

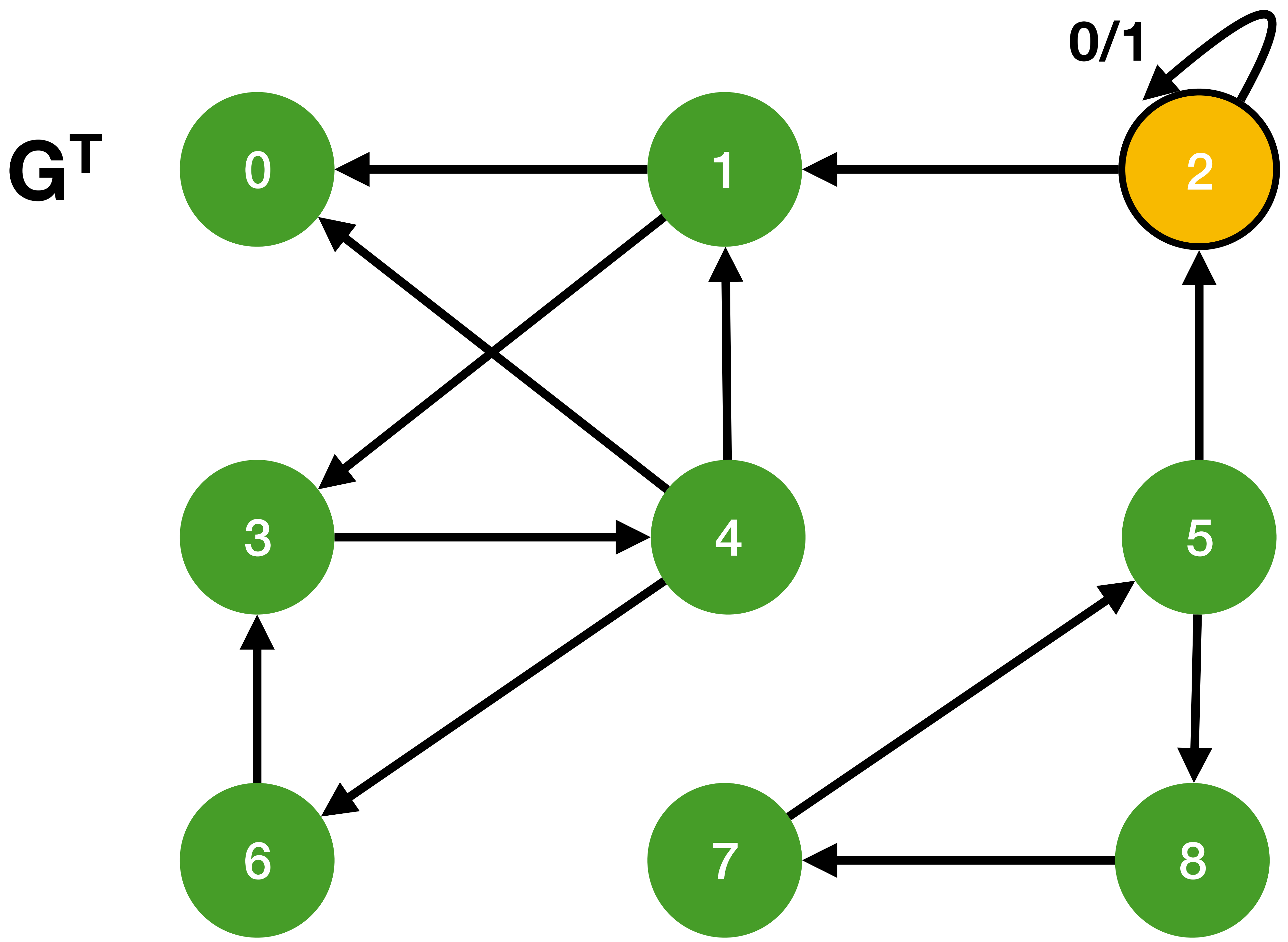
Stack
4
3
6
2
5
7
8
0

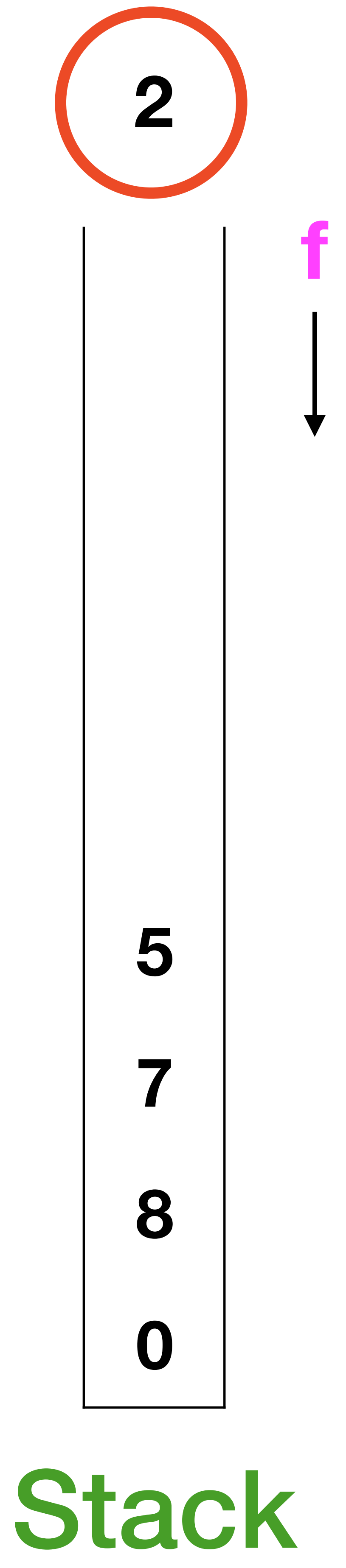
Stack

0	
1	3, 0
2	2, 1
3	4
4	6, 1, 0
5	8, 2
6	3
7	5
8	7

adjacency list

Kosaraju - Sharir algorithm



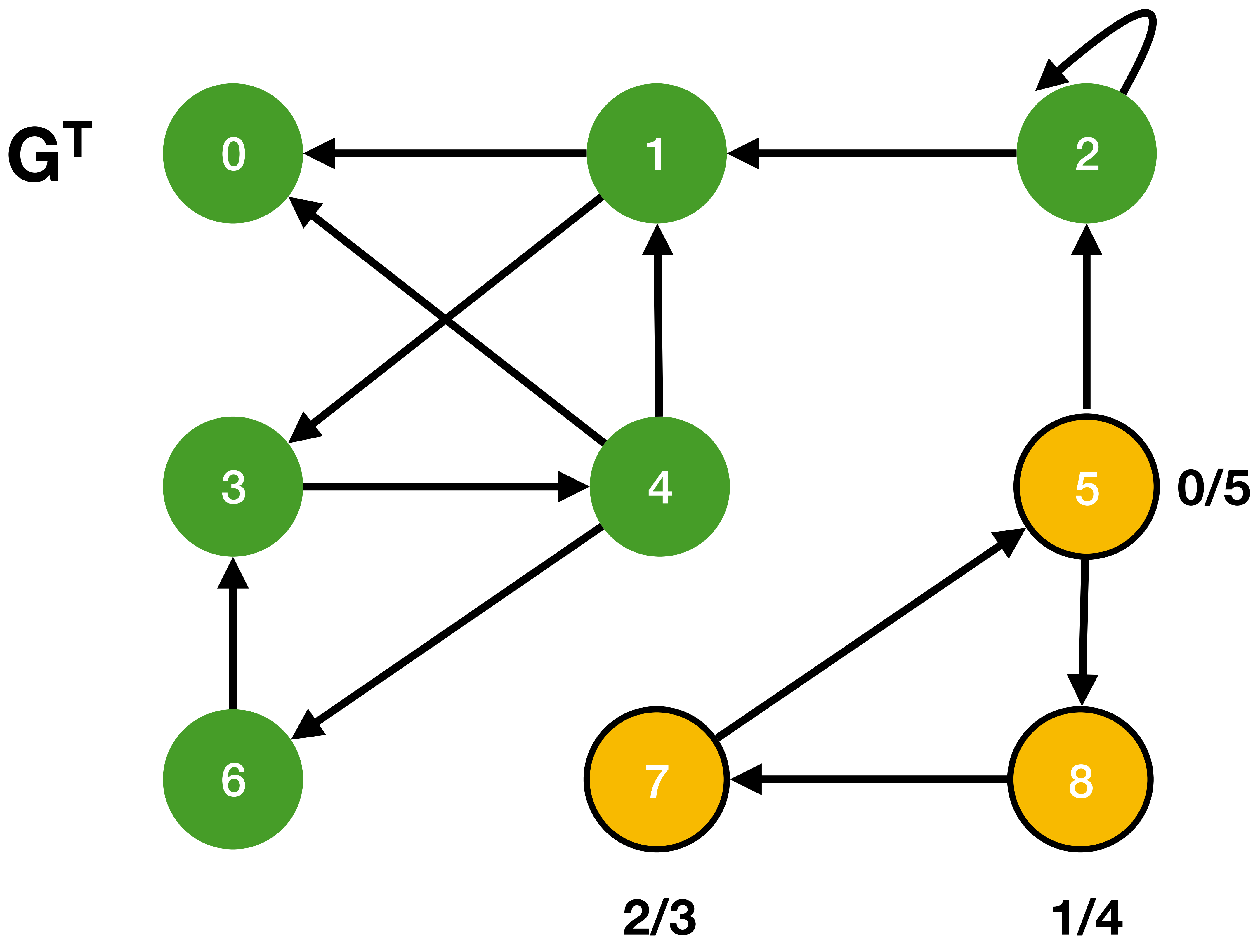


0	
1	3, 0
2	2, 1
3	4
4	6, 1, 0
5	8, 2
6	3
7	5
8	7

adjacency list

Visited 1, 3, 4, 6, 2 with DFS

Kosaraju - Sharir algorithm



5

f

7
8
0

Stack

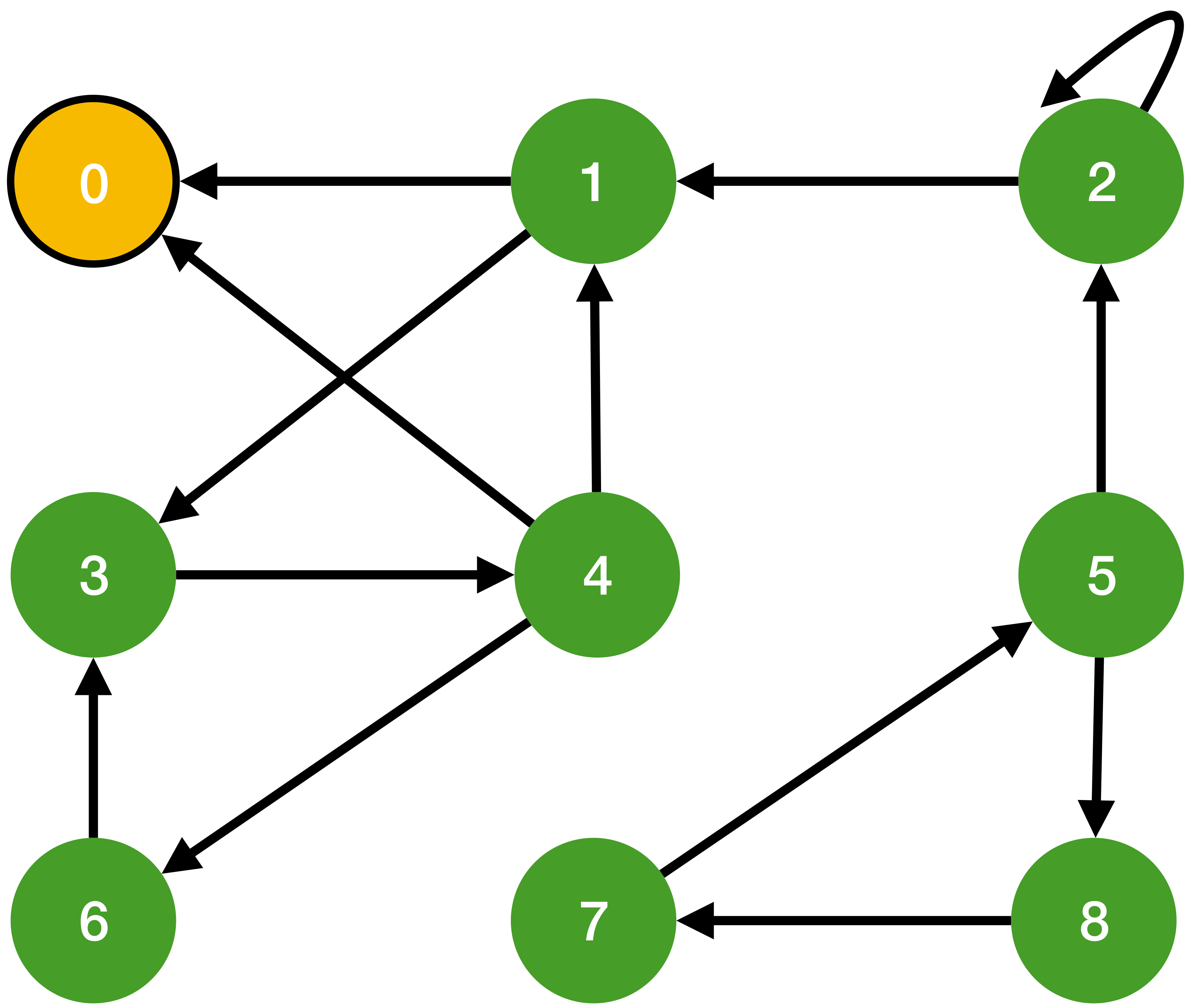
0	
1	3, 0
2	2, 1
3	4
4	6, 1, 0
5	8, 2
6	3
7	5
8	7

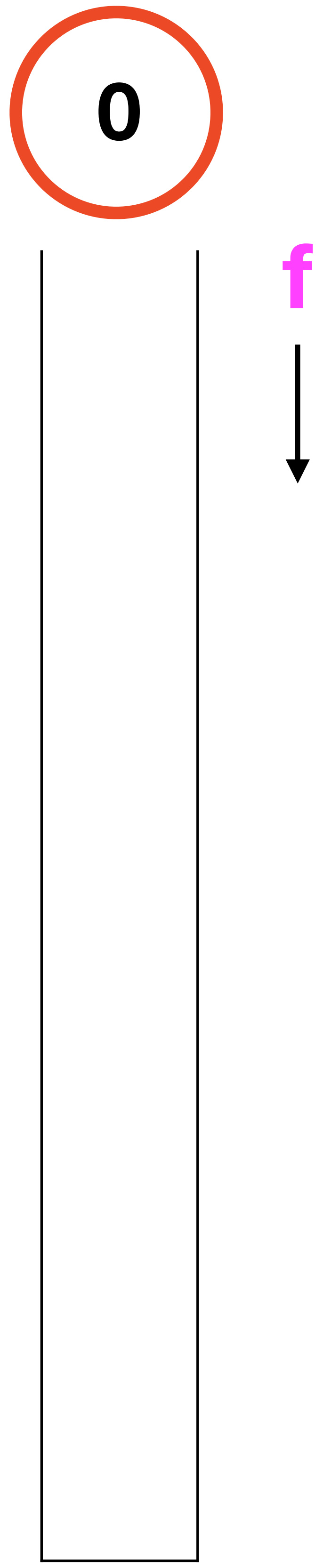
adjacency list

Visited 1, 3, 4, 6, 2, 5, 7, 8 with DFS

Kosaraju - Sharir algorithm

G^T





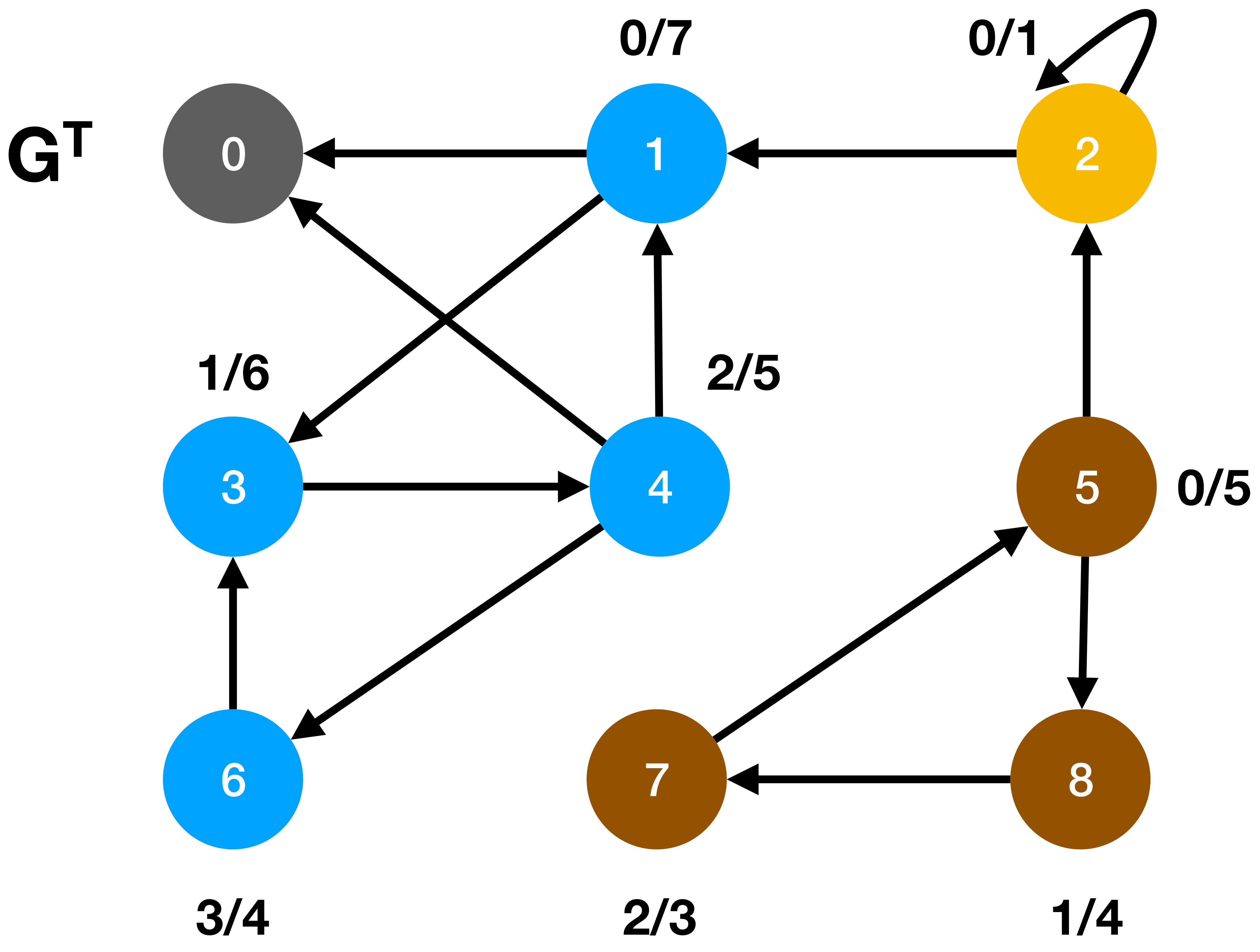
Stack

0	
1	3, 0
2	2, 1
3	4
4	6, 1, 0
5	8, 2
6	3
7	5
8	7

adjacency list

Visited 1, 3, 4, 6, 2, 5, 7, 8, 0 with DFS

Kosaraju - Sharir algorithm Solution



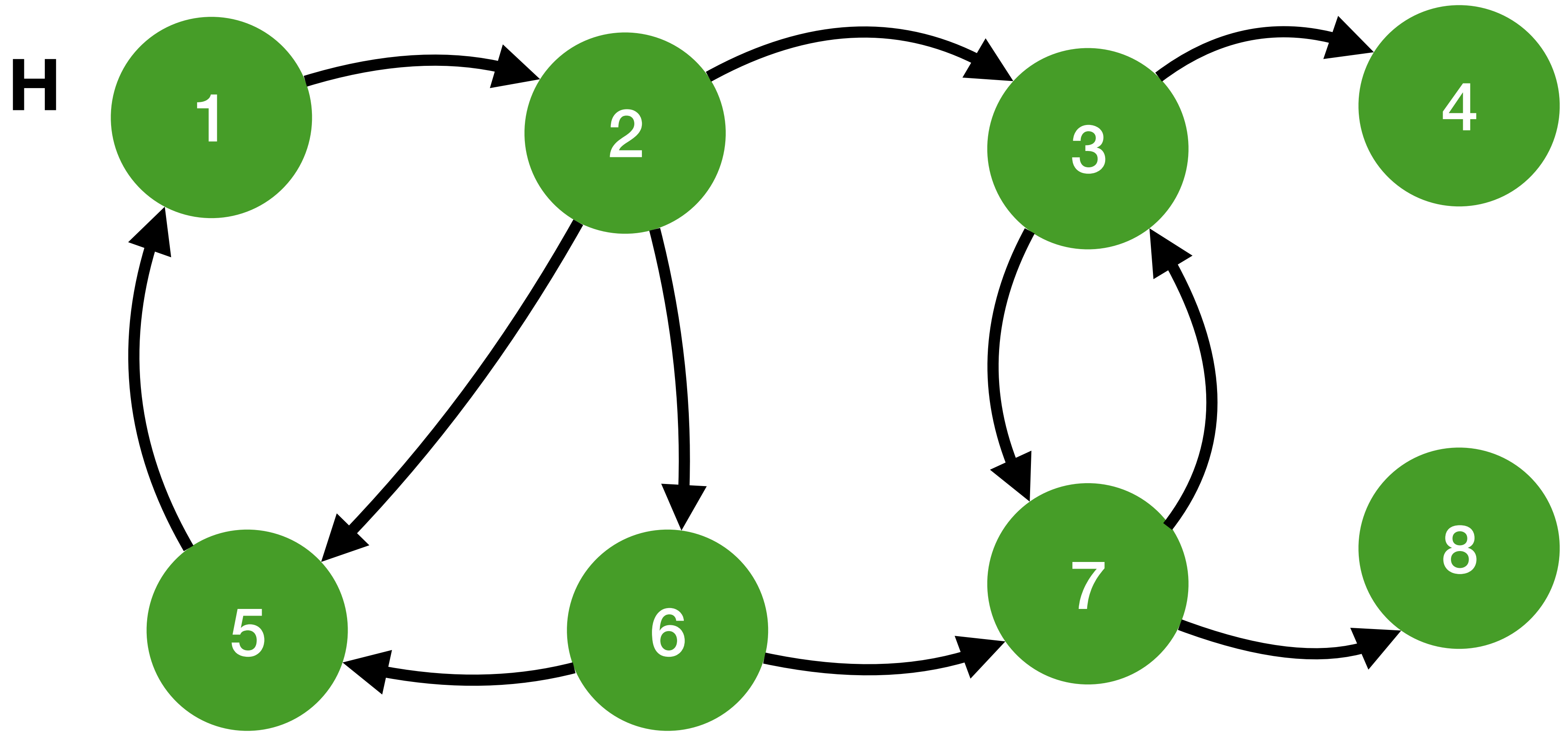
The SCC are-

0
5,7,8
2
1,3,4,6

Visited 1, 3, 4, 6, 2, 5, 7, 8, 0 with DFS

Exercise

Use the Kosaraju-Sharir algorithm to find the strongly connected components in this graph.



Modified DFS for Kosaraju - Sharir (using stack)

```
stack s2; //stores vertices in order of decreasing finishing times

depthFirstSearch (id, myGraph, visited, discoveryTime){
        stack s1;
        // set start vertex s with given id to visited and push it on stack
        visited[id] = 1;
        s1.push(id);
        while (!s1.empty()){
            s = s1.top(); // peek at vertex on top of stack
            find first neighbor v_id of s that is unvisited ;
            s1.push(v_id);
            visited [v_id] = 1;

            record discovery time of v_id;

           if there is no unvisited neighbor for s {
              s1.pop(); // pop s from s1
              record finishing time for s;
              Push s on s2;
           }
        }
 }
```

Pseudocode graph transpose

Using adjacency list

```
//GT is transpose of graph G
transpose (G){
    GT = new Graph();
    for each vertex v in G {
            for each edge (v, u) in adjacency list{
                GT -> addNewEdge(u, v);
        }}
   return GT;
}
```

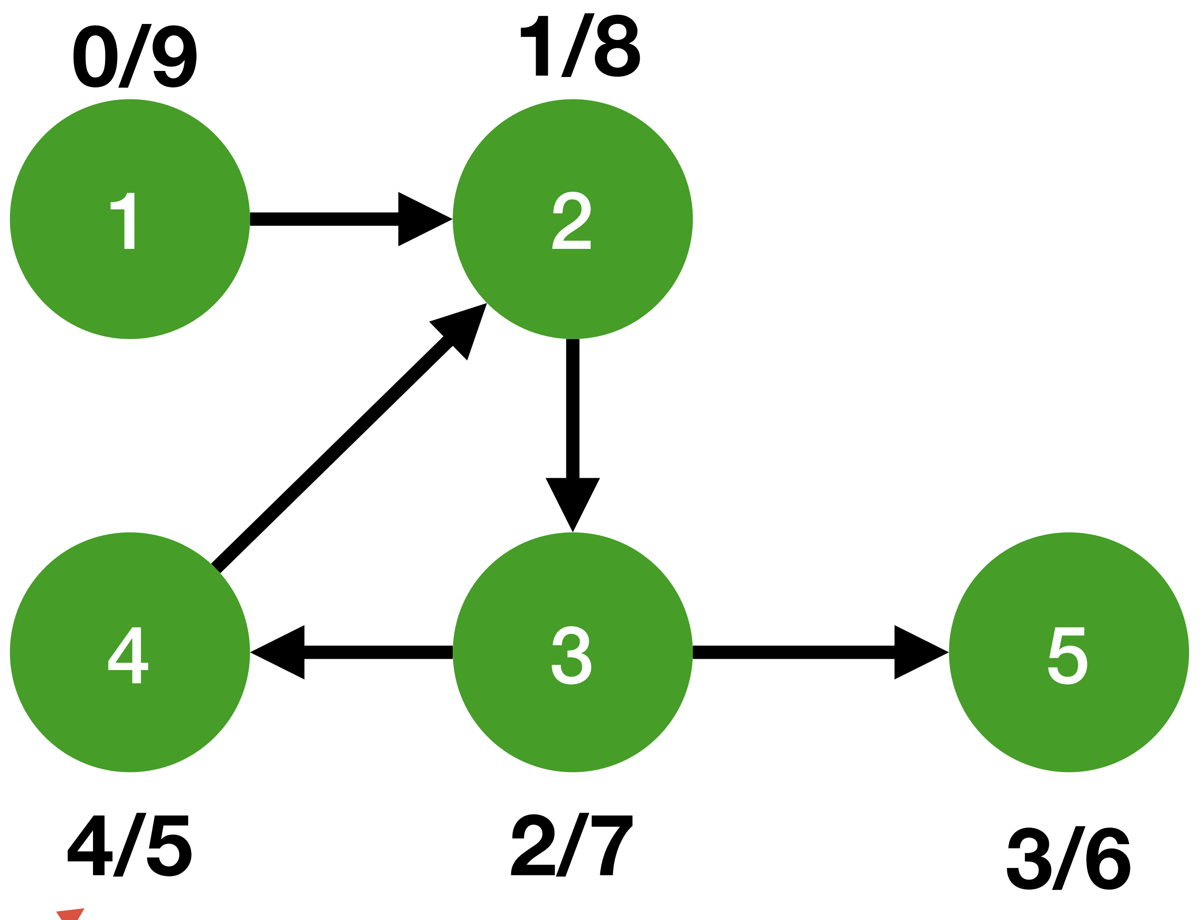
Pseudocode for Kosaraju - Sharir

```
kosarajuSharir() {
   initialize array visited to 0 with id v_id
   for all vertex v in G {
       if(visited[v_id] == 0)
        depthFirstSearch(v_id, G, visited, 0);
   }
   GT = transpose(G);
   re_initialize array visited to 0;

   while(! s2.empty()) {
         s_id = s1.top(); // peek at vertex on top
         pop s_id from s2;
         if(visited[s_id] == 0){
            // vertex is not part of another SCC
            depthFirstSearch(s_id, GT, visited, 0);
         }
   }
}
```

Exercise

Suppose that we did not transpose the graph, can we modify this algorithm to return SCC by traversing the original graph in the order of increasing finishing times of vertices ?

A. No



discovery time / finishing Time

Starting from DFS from 4

4 → **2** → **3** → **5** = discovers 5

= > 4, 2, 3, 5 is SCC which is incorrect

DFS-based algorithms for SCC

- Kosaraju-Sharir algorithm
- Tarjan’s algorithm
- Lowe’s algorithm

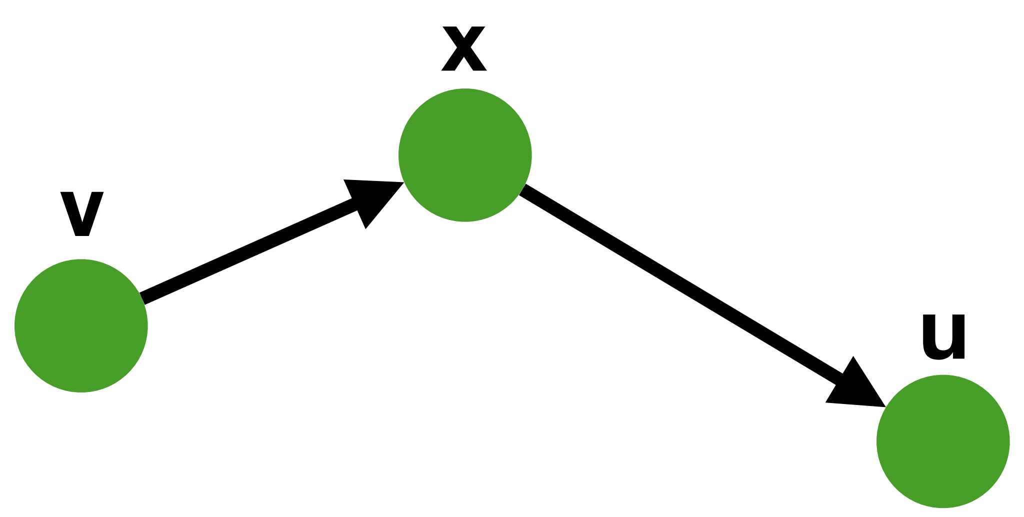
Parallel algorithms for SCC

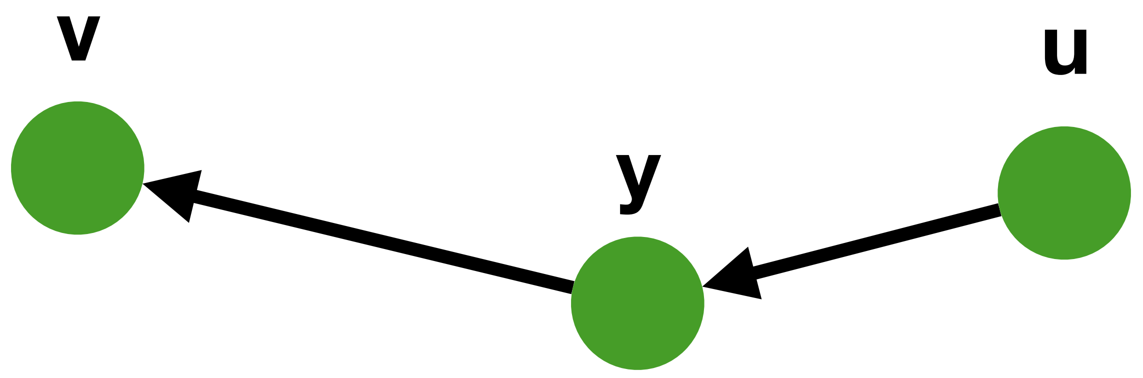
- Concurrent algorithm that can run on multi-core processor
- Need to adjust graph algorithm to use parallelism
- Divide and conquer algorithm
    - Forward - backward algorithm
    - Schudy's algorithm
    - Hong's algorithm

Forward - Backward algorithm

- Forward reachable in vertex u is forward reachable from vertex v if there is a path fro v to u

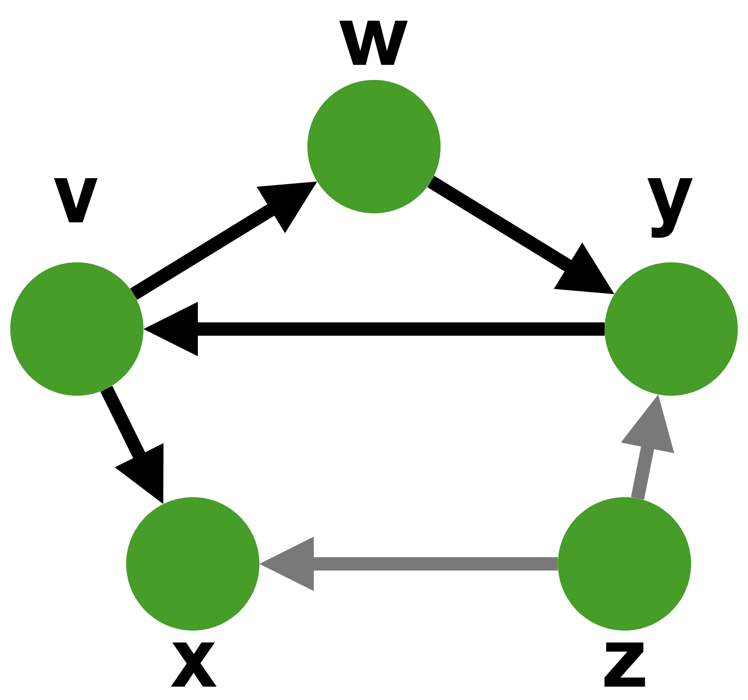
- Backward reachable vertex u is backward reachable from vertex v if there is a path from u to v

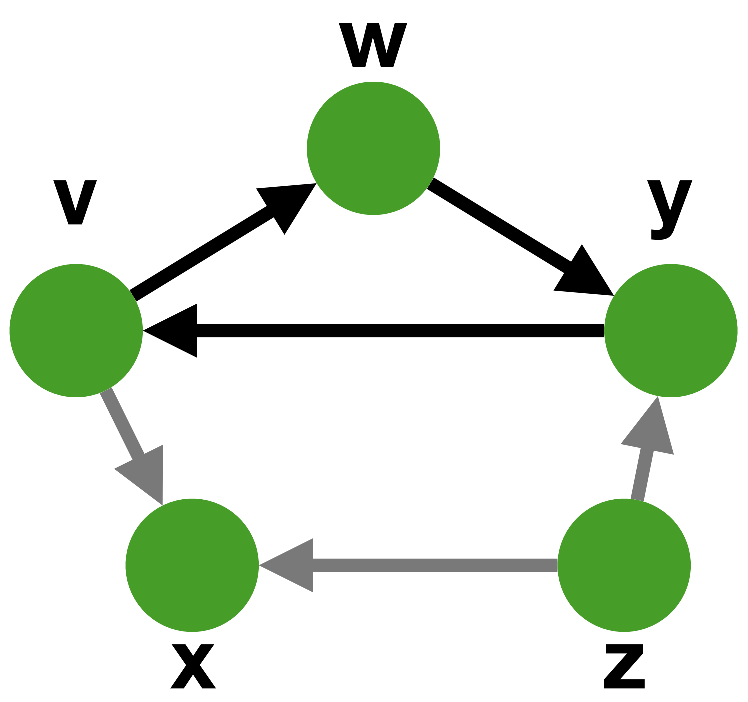




Forward - Backward algorithm

- Forward reachability set : set of all vertex that are forward reachable from v
- Backward reachability set, set of all vertices that are backward reachable from v





Forward reachability set of v = {w, y, x, v}   Backward reachability set of v = {w, y, v}

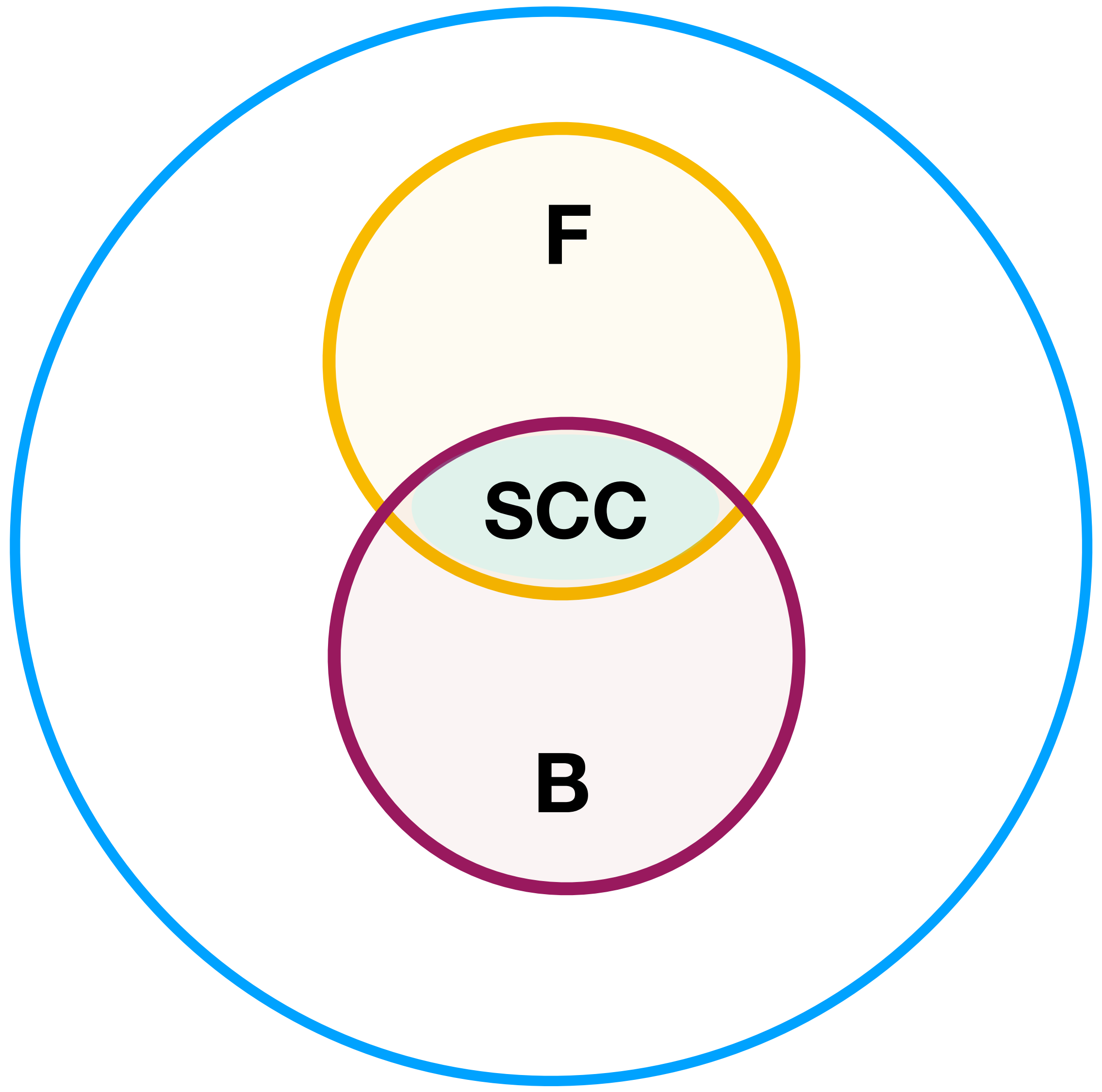
Note SCC = {w, y, x, v} ∩ {w, y, v} = {w, y, v}

Forward - Backward algorithm

Lemma : In a graph G(V, E) let F and B be the forward and backward reachability sets for vertex v. The intersection of F and B is a unique SCC.

Forward - Backward algorithm

The intersection of F and B (SCC) consists of vertices that are forward and backward reachable from v. If any vertex is backward and forward reachable from v, it must be in SCC



Forward - Backward algorithm

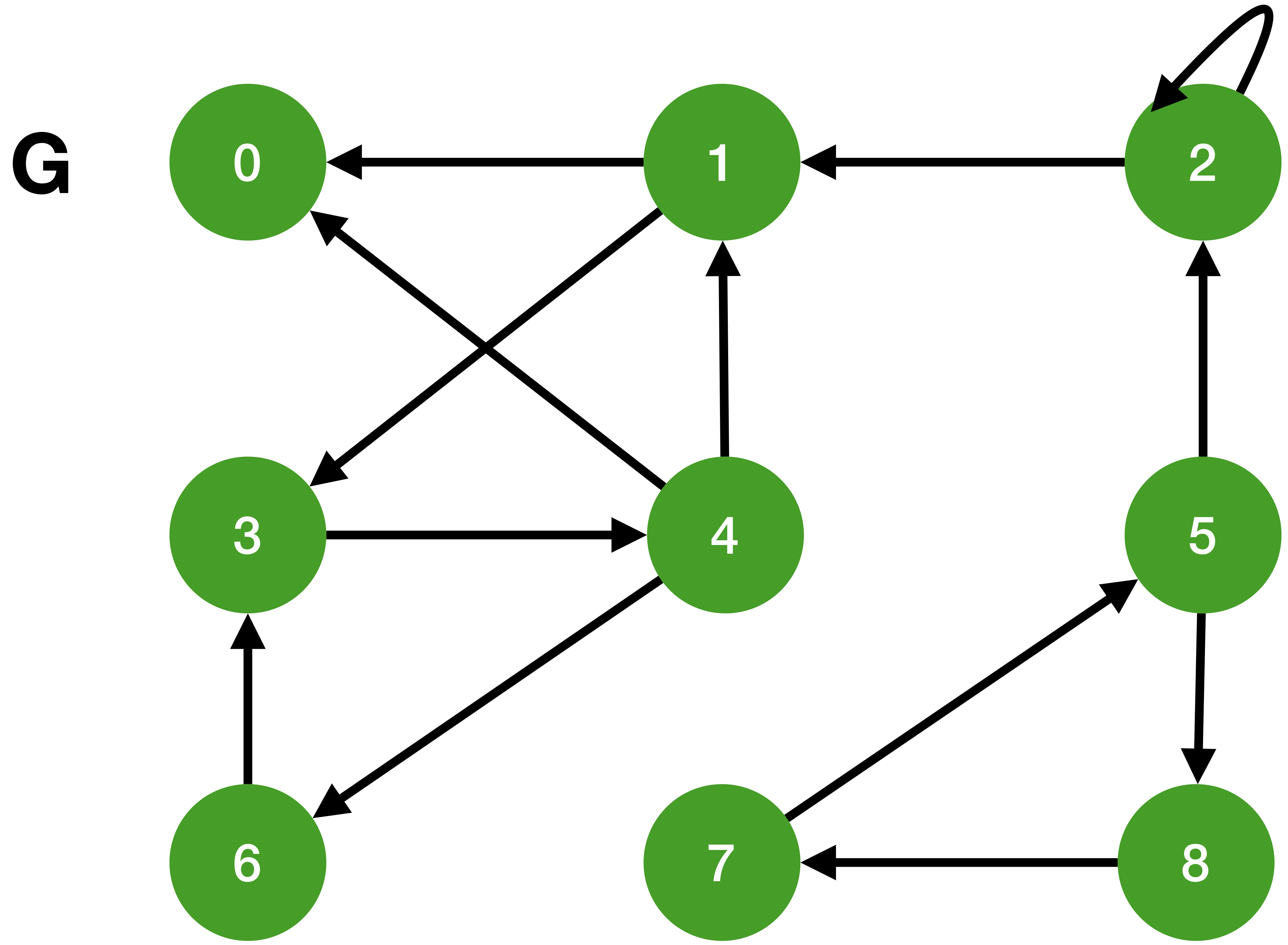
Lemma : Given graph $G = \{V, E\}$ all other SCCs $S \subseteq G$ are in the following subgraph

1. $S \subseteq F \setminus B$
2. $S \subseteq B \setminus F$
3. $S \subseteq V \setminus (F \cup B)$

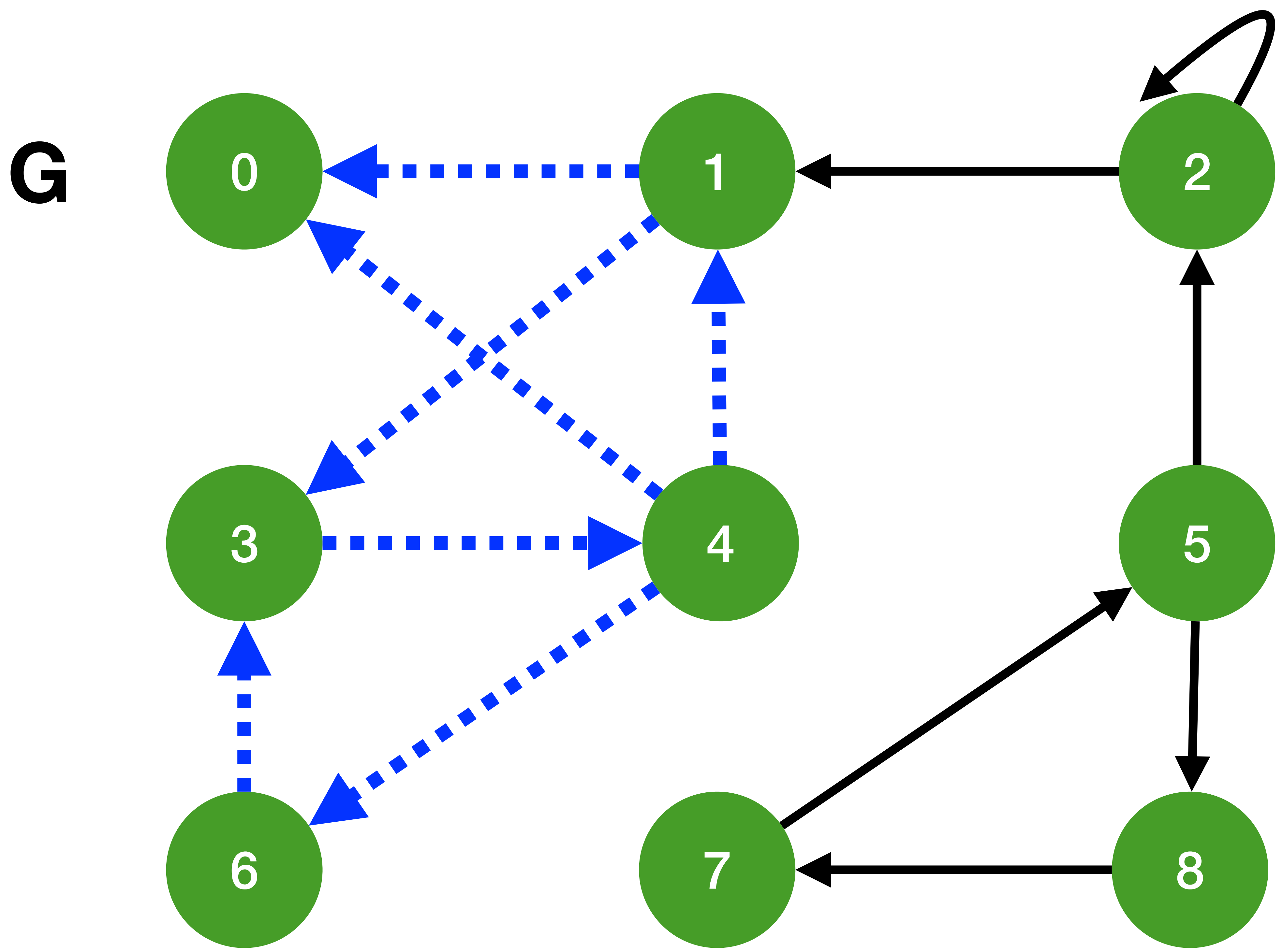
1, 2 and 3 are three independent instances

Process them recursively in parallel

Forward - Backward algorithm example

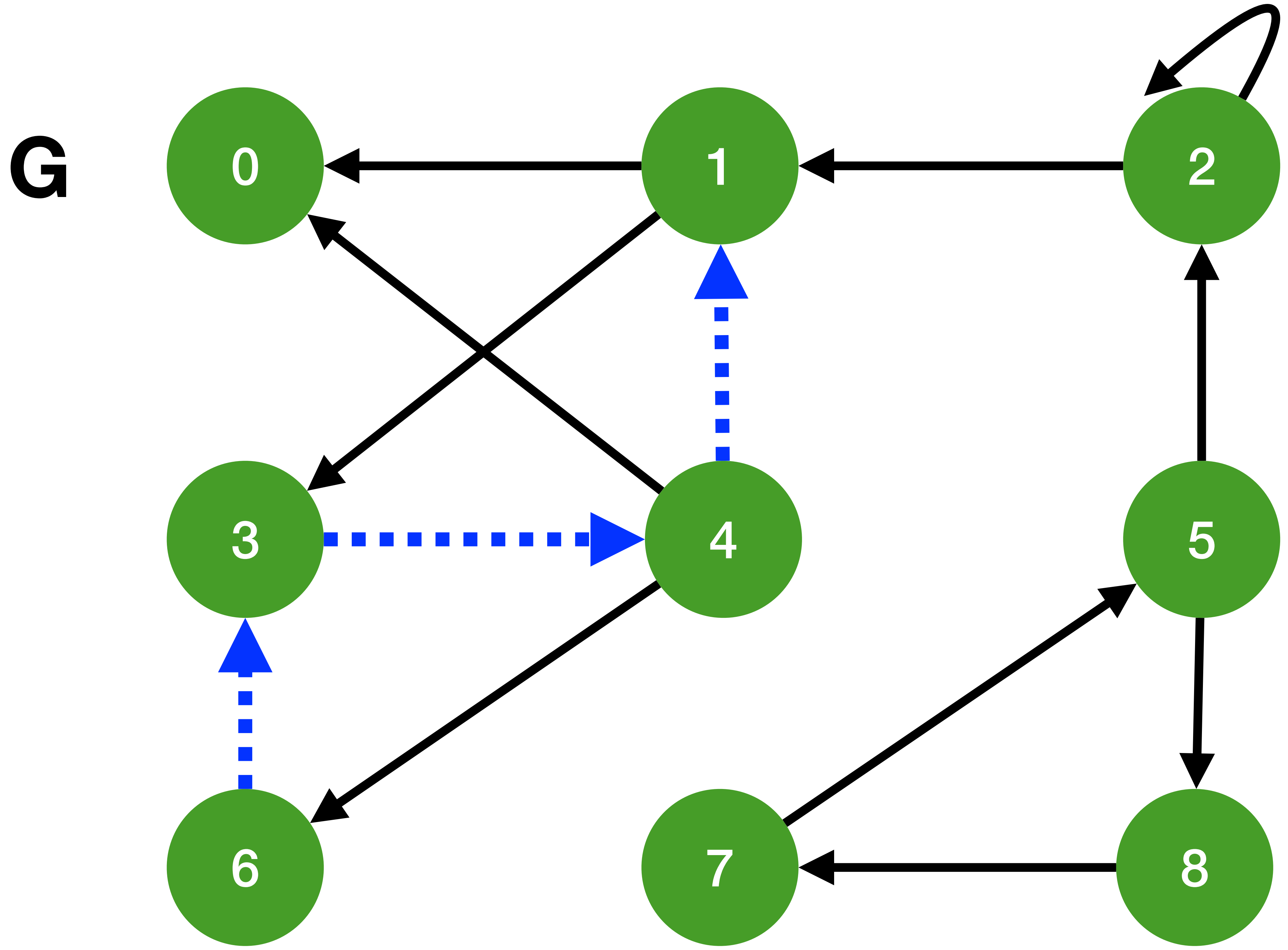


Forward - Backward algorithm example



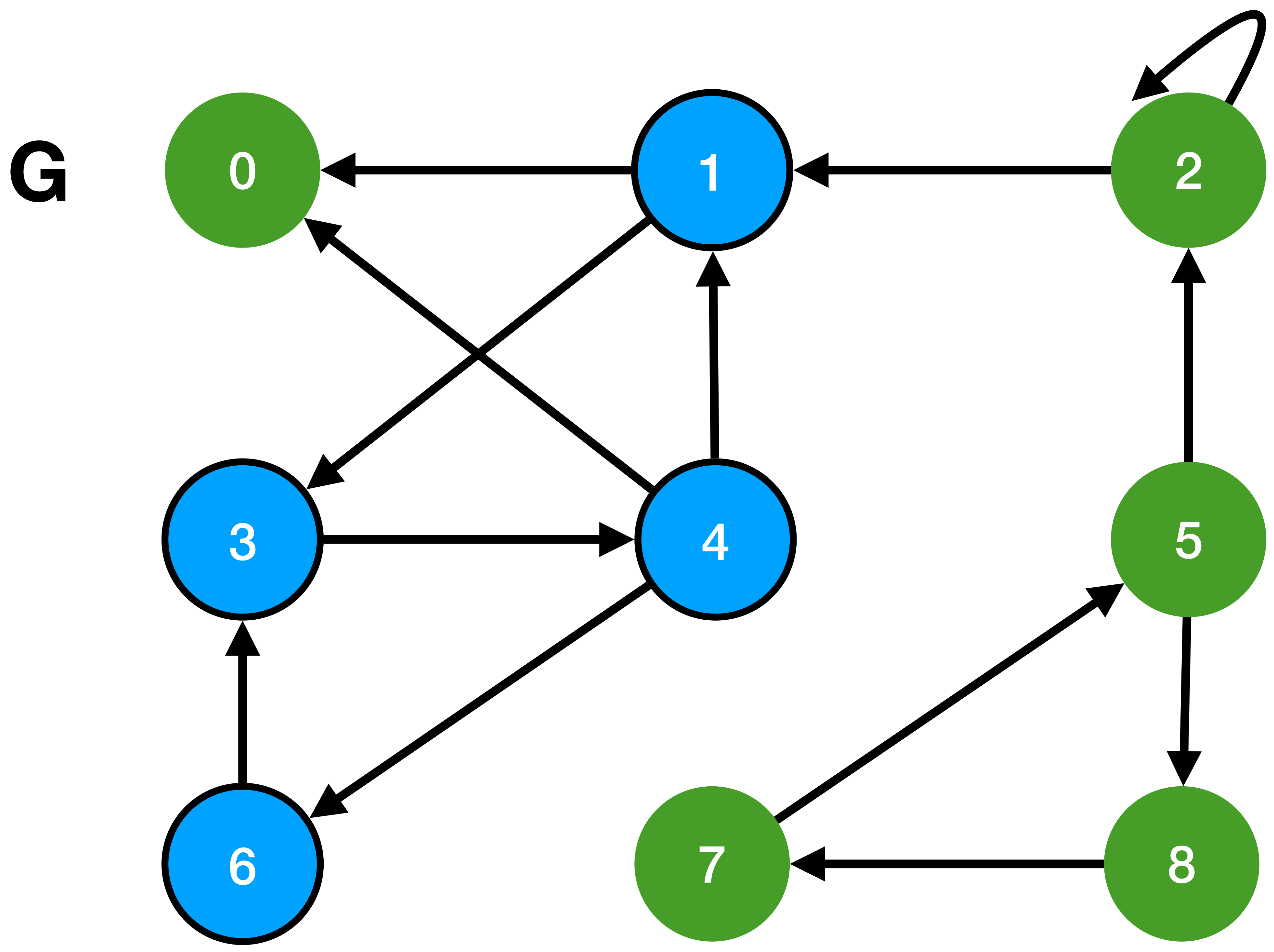
v = 1   F = { 0, 1, 3, 4, 6 }

Forward - Backward algorithm example



v = 1 B = { 1, 3, 4, 6 }

Forward - Backward algorithm example



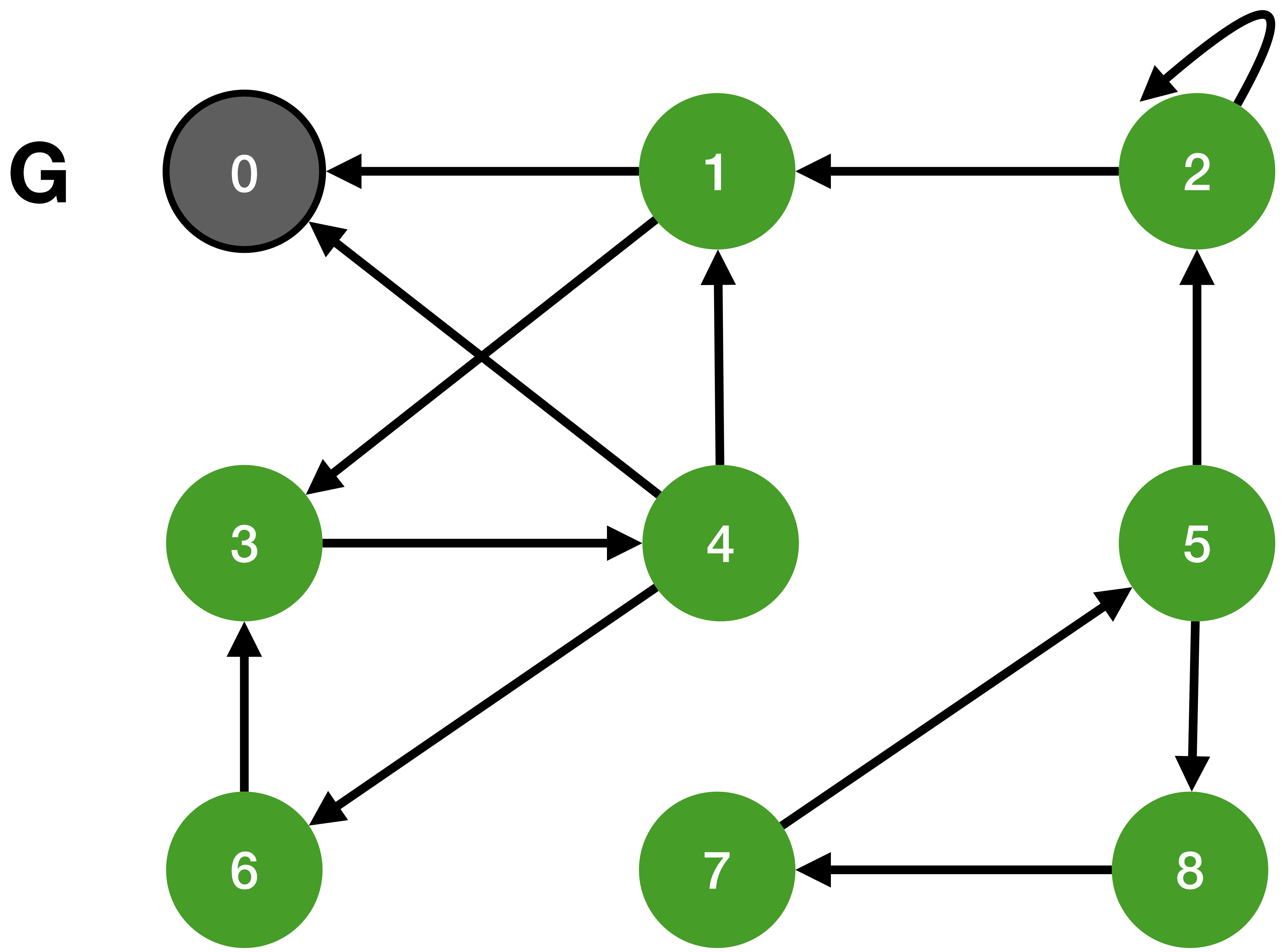
v = 1 F = { 0, 1, 3, 4, 6 }
B = { 1, 3, 4, 6 }

F ∩ B = { 1, 3, 4, 6 }

F \ B = {0}
B \ F = { }
V \ FUB = {2, 5, 7, 8}

Process (F \ B), (B \ F), (V \ FUB) in praallel

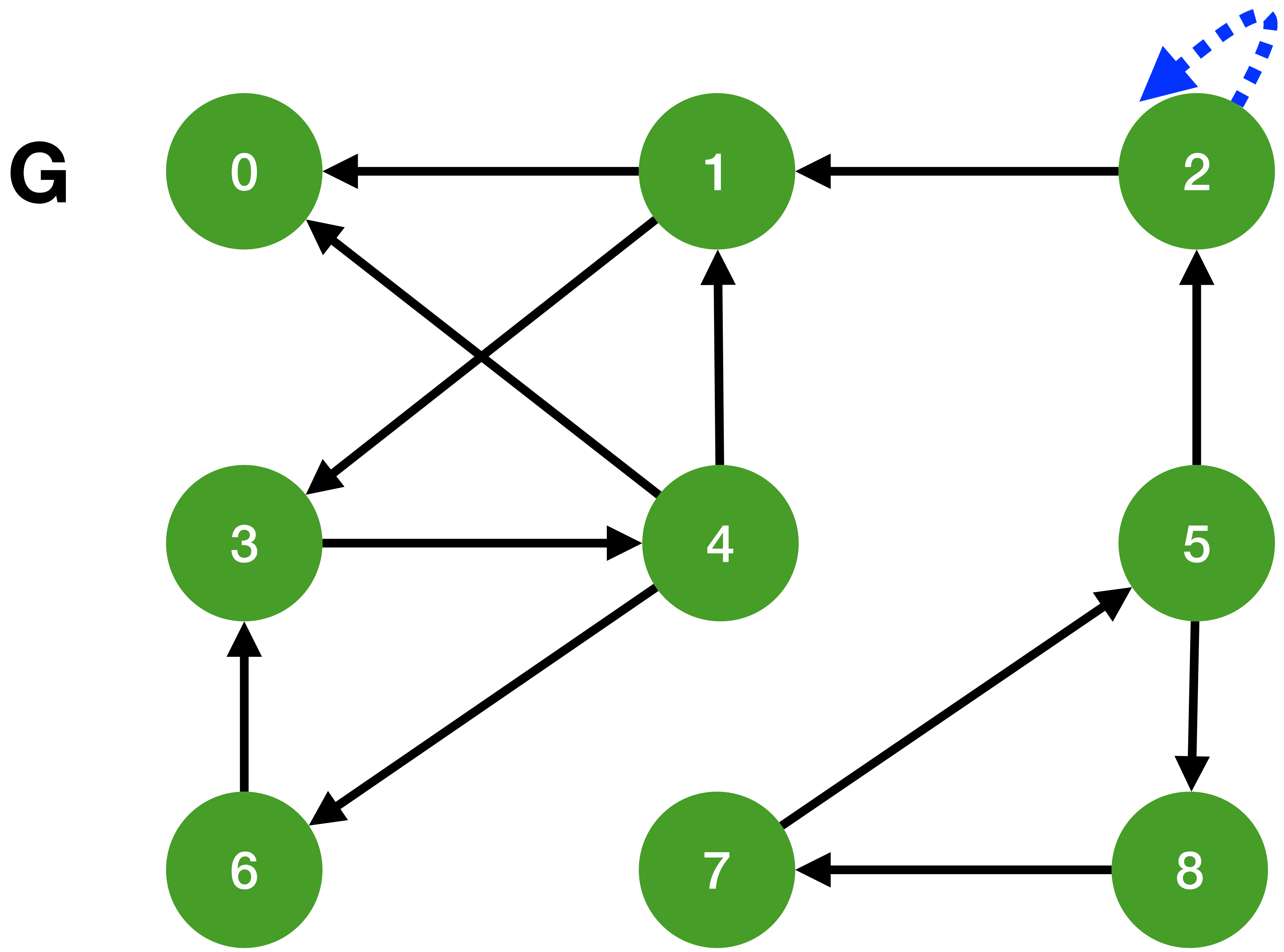
Forward - Backward algorithm example



Process F \ B = {0}

SCC = {0}

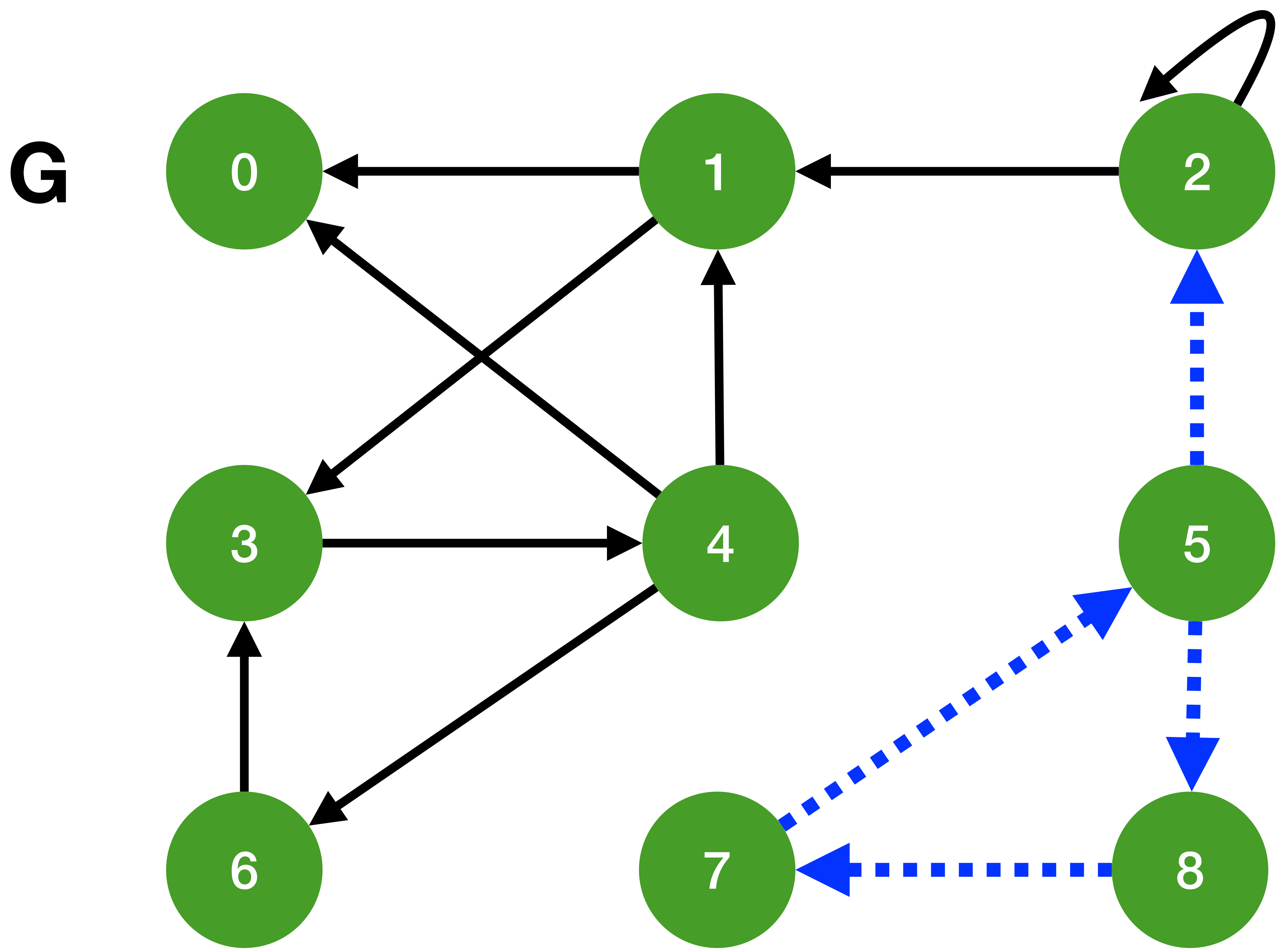
Forward - Backward algorithm example



Process V \ FUB = {2, 5, 7, 8}

v = 2 F = { 2}

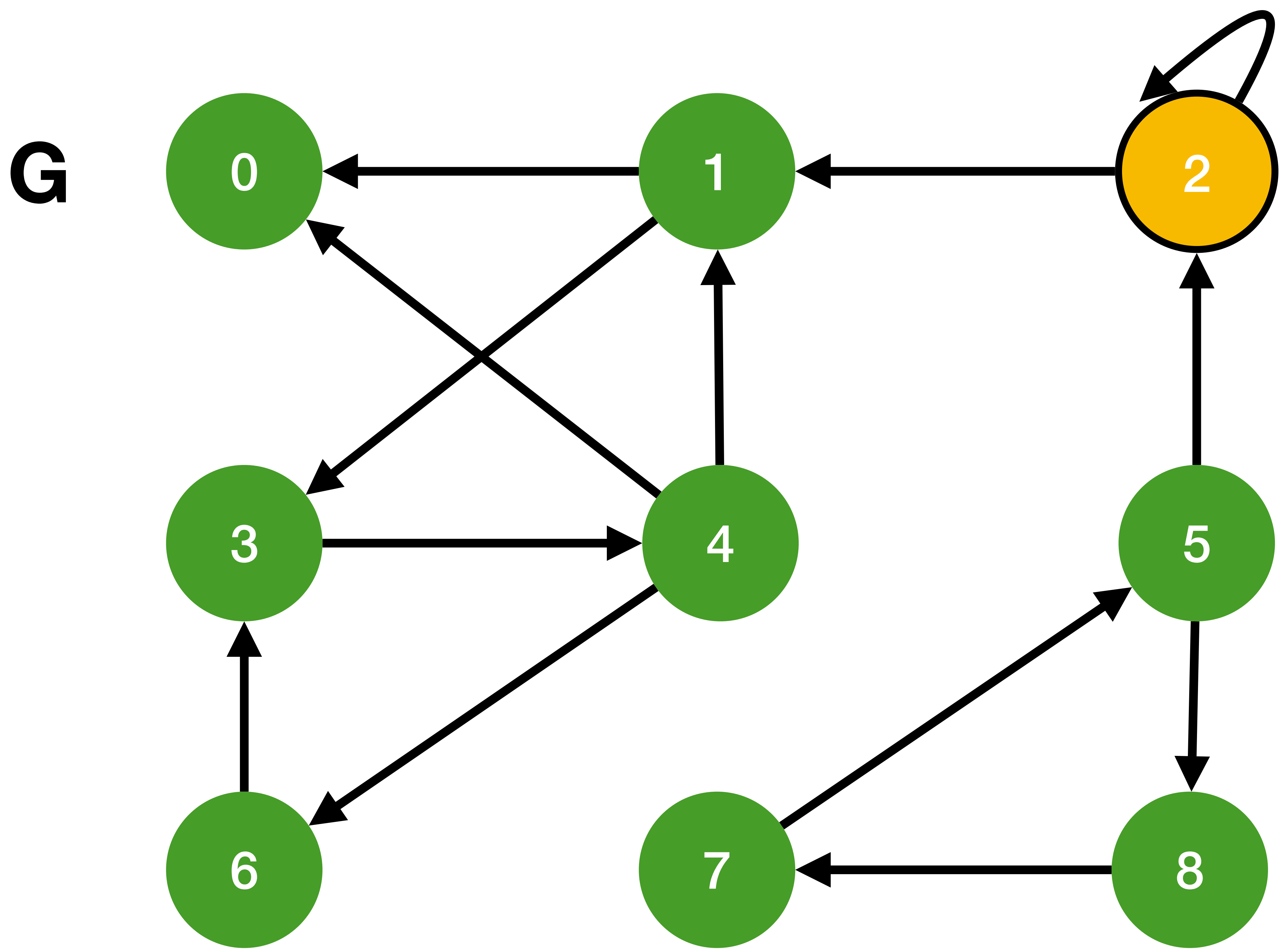
Forward - Backward algorithm example



Process V \ FUB = {2, 5, 7, 8}

v = 2 B = { 2, 5,7,8}

Forward - Backward algorithm example



Process V \ FUB = {2, 5, 7, 8}

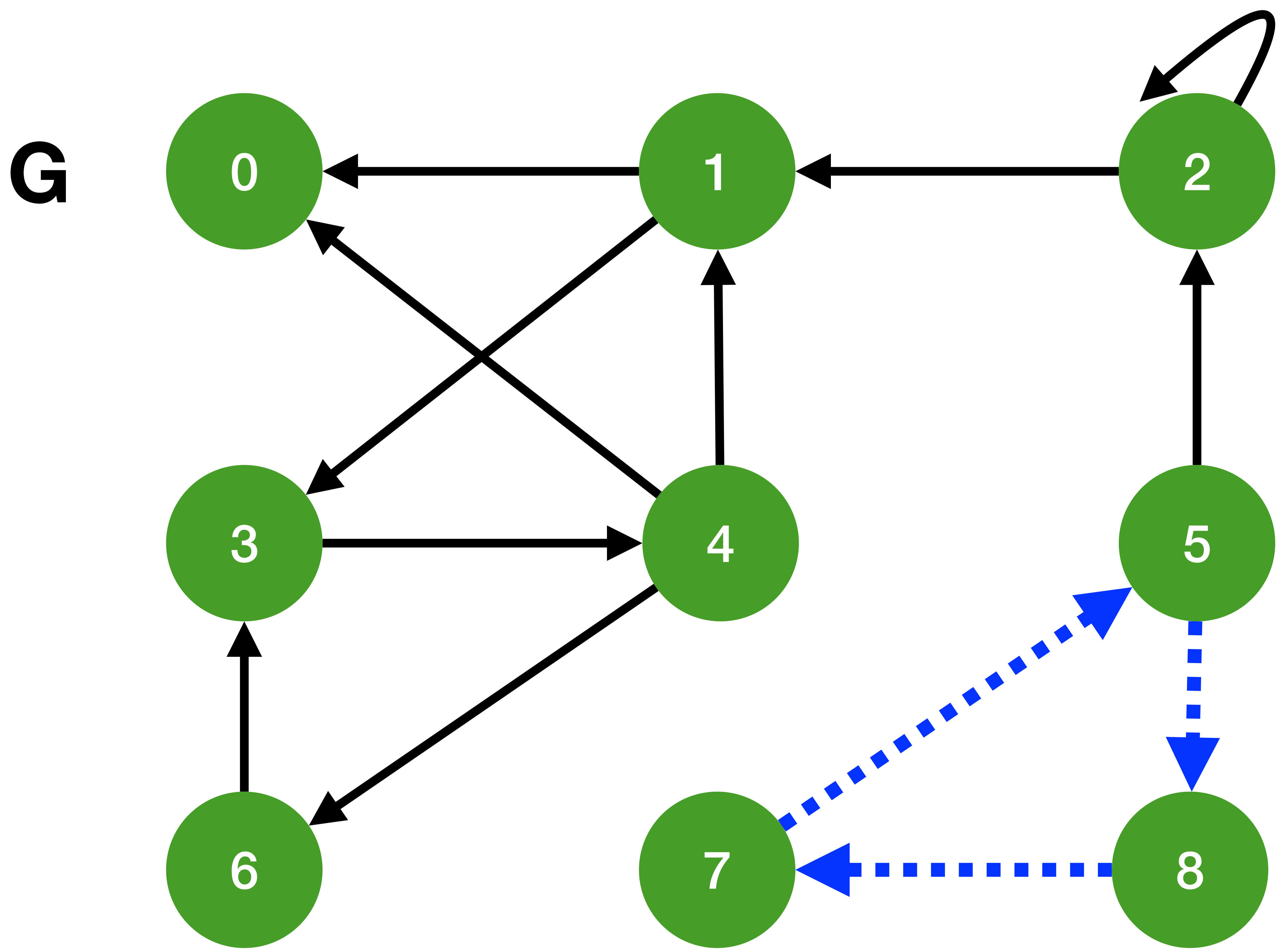
v = 2 F = {2}

B = { 2, 5,7,8}

F ∩ B = {2}

F \ B = { }
B \ F = { 5,7,8}
V \ FUB = { }

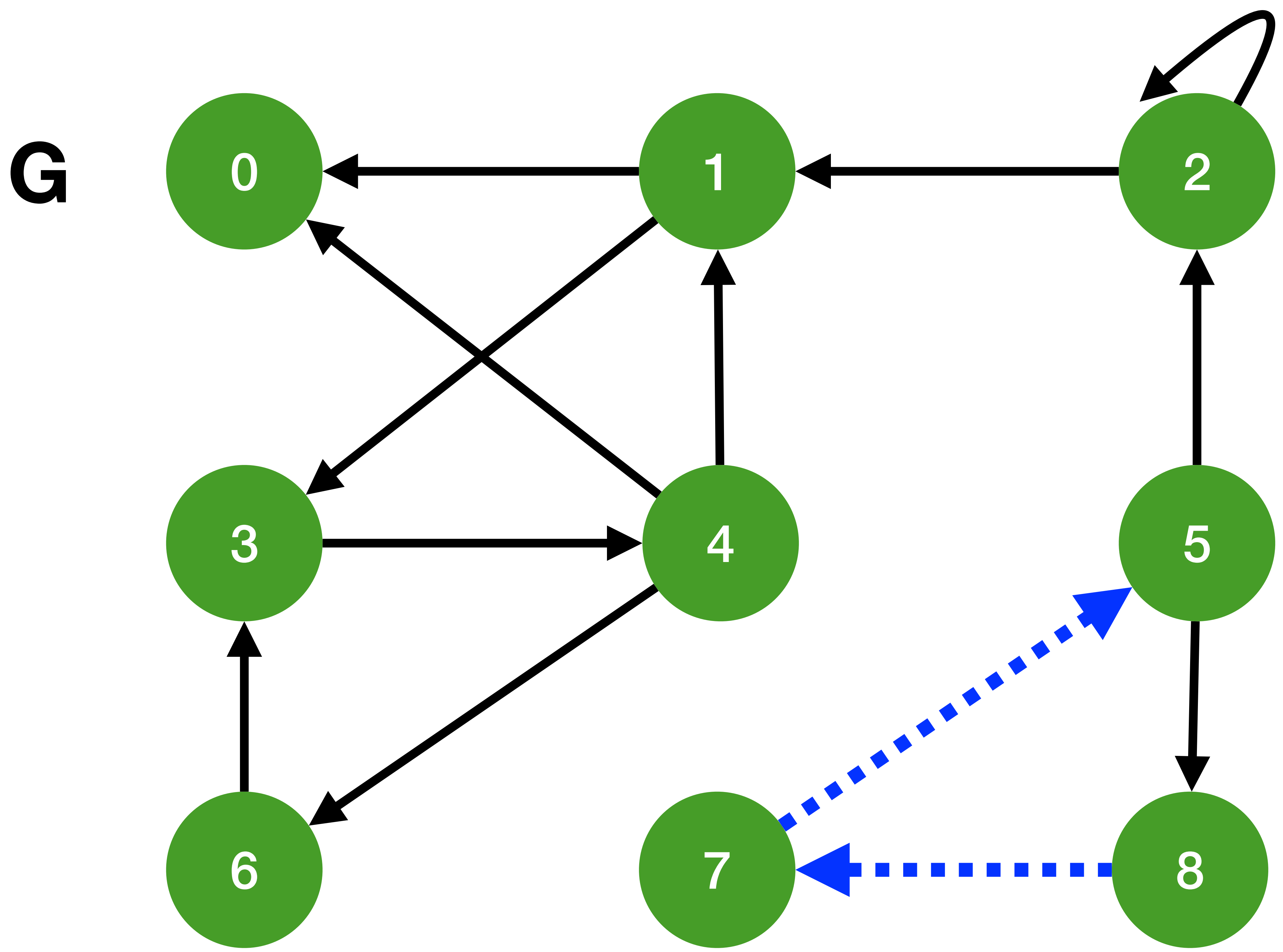
Forward - Backward algorithm example



Process B \ F = { 5, 7, 8}

v = 5 F = {5, 7, 8}

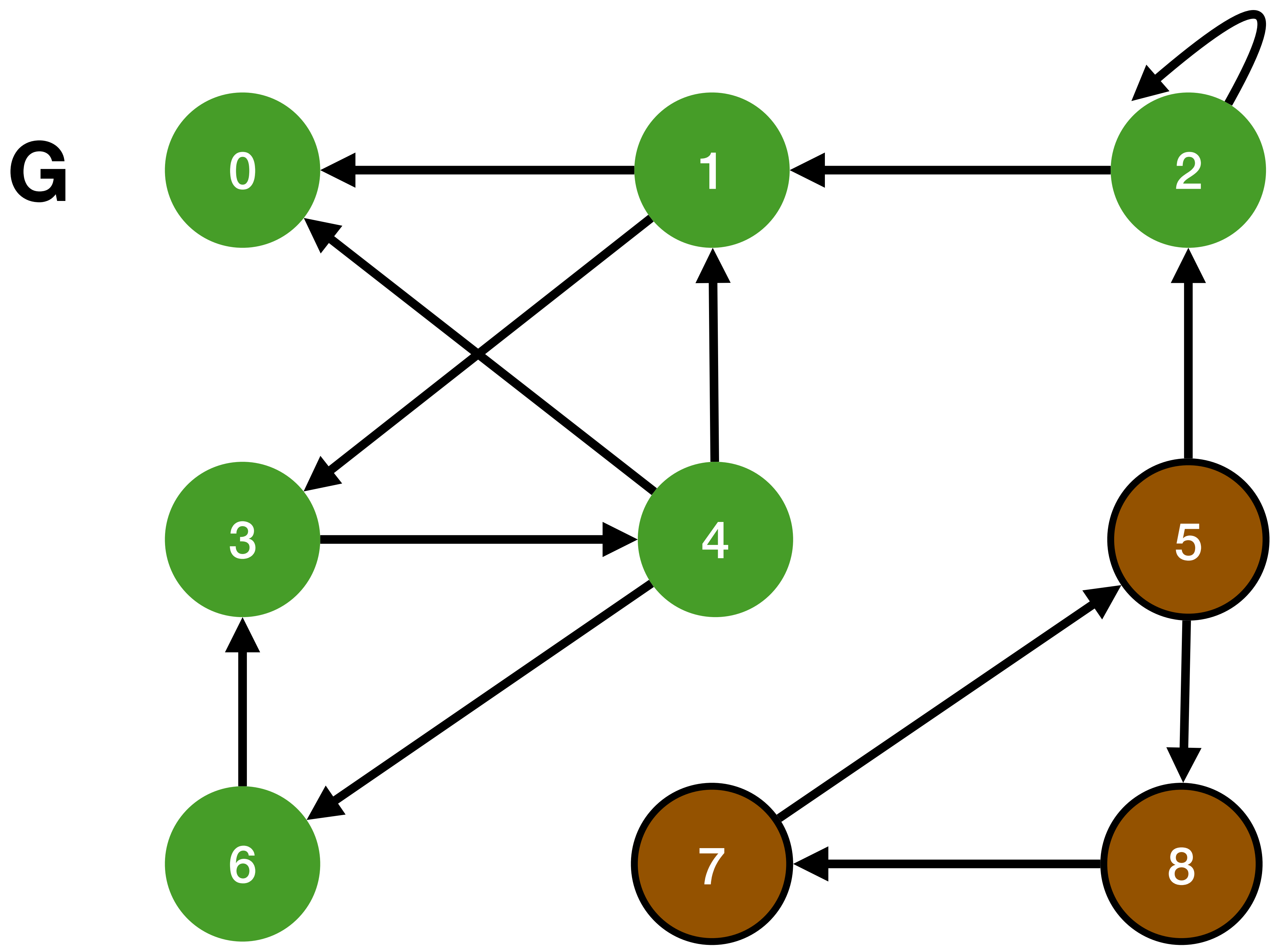
Forward - Backward algorithm example



Process B \ F = { 5, 7, 8}

v = 5 F = {5, 7, 8}

Forward - Backward algorithm example



Process B \ F = { 5, 7, 8}

v = 5 F = {5, 7, 8}

B = {5, 7, 8}

F ∩ B = { 5, 7, 8}

F \ B = { }
B \ F = { }
V \ FUB = { }

Forward - Backward algorithm pseudocode

```
// G = {V, E}
FB(V){
   if (V ≠ ∅){
       v = PIVOT(V);
       F = FWD(v, V); // find forward reachability sets
       B = BWD(v, V); // find backward reachability sets
       SCC = F∩B ;
       in parallel do{
        FB(F\B);
        FB(B\F);
        FB(V\(FUB));
       } end in parallel;
    }
 }
```

Forward - Backward algorithm trimming step

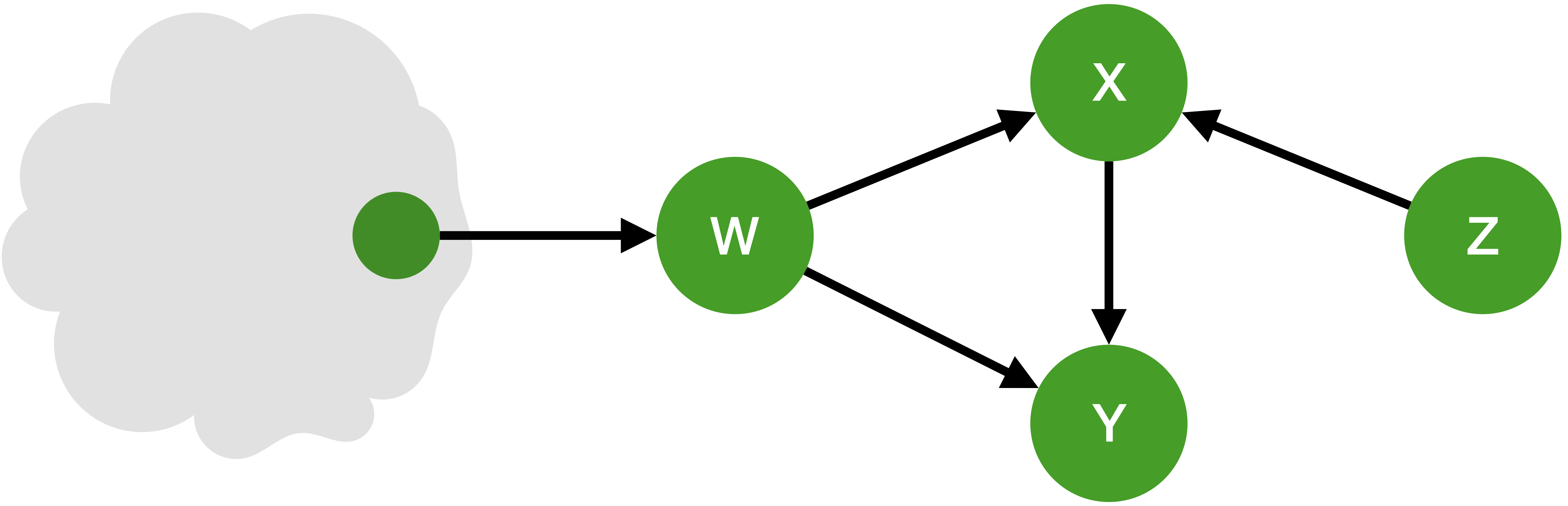
- Improvement to forward and backward algorithm
- Trimming step : Remove vertices with only zero incoming or zero outgoing edges (called trivial SCC)

Forward - Backward algorithm trimming step

1. Remove Y and Z (trivial SCC)

2. Remove X (new trivial SCC)

3. Remove W (new trivial SCC)



SCC Application: 2 - SAT

- SAT: Determine if a Boolean formula is satisfiable.
- Each variable is either TRUE or FALSE
- 2-SAT has 2 literals in each clause.

$\varnothing_1 = A \wedge \neg B$ is satisfiable for A = TRUE and B = FALSE

$\varnothing_2 = A \wedge \neg A$ is unsatisfiable

$\varnothing_3 = (\neg X \vee Y) \wedge (\neg Y \vee X) \wedge (X \vee \neg Y)$

SAT is NP - complete but 2-SAT can be solved in polynomial time

2 - SAT

Decision Problem : ∅ is satisfiable iff no literal and its negative are in the name SCC of its implication graph

Is ∅ satisfiable ?

An implication graph can be used to solve 2-SAT

1. Create a vertex for every literal of ∅
2. Replace disjunctions with implications
   $(X \vee Y) \equiv (\neg X \rightarrow Y) \wedge (\neg Y \rightarrow X)$
3. Create an edge for each implication
   $\varnothing = (X_1 \vee Y_1) \wedge \ldots\ldots\ldots \wedge (X_n \vee Y_n)$

X	**Y**	**X → Y**	***¬X V Y***	***¬X V ¬Y***
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	1	1	1

Example 1

Q. Is ∅ = (X ∨ Y) ∧ (Y ∨ Z) ∧ (X ∨ Z) satisfiable?

A. (X ∨ Y) ≡ (¬X → Y) ∧ (¬Y → X)

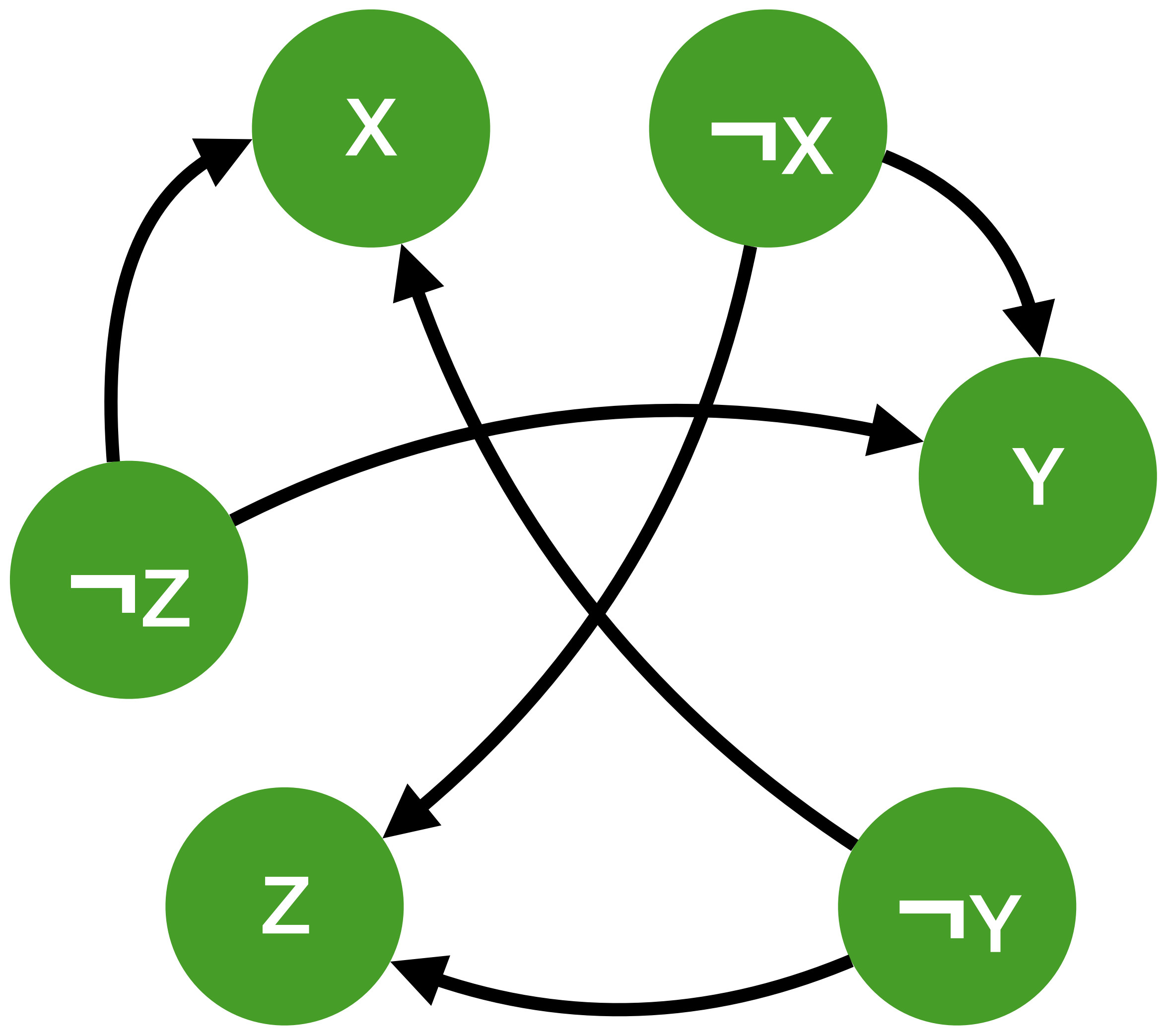
Draw edges (¬X to Y) , (¬Y to X)

Y ∨ Z ≡ (¬Y → Z) ∧ (¬Z → Y)

X ∨ Z ≡ (¬X → Z) ∧ (¬Z → X)

Draw edges (¬Y to Z) , (¬Z to Y), etc.

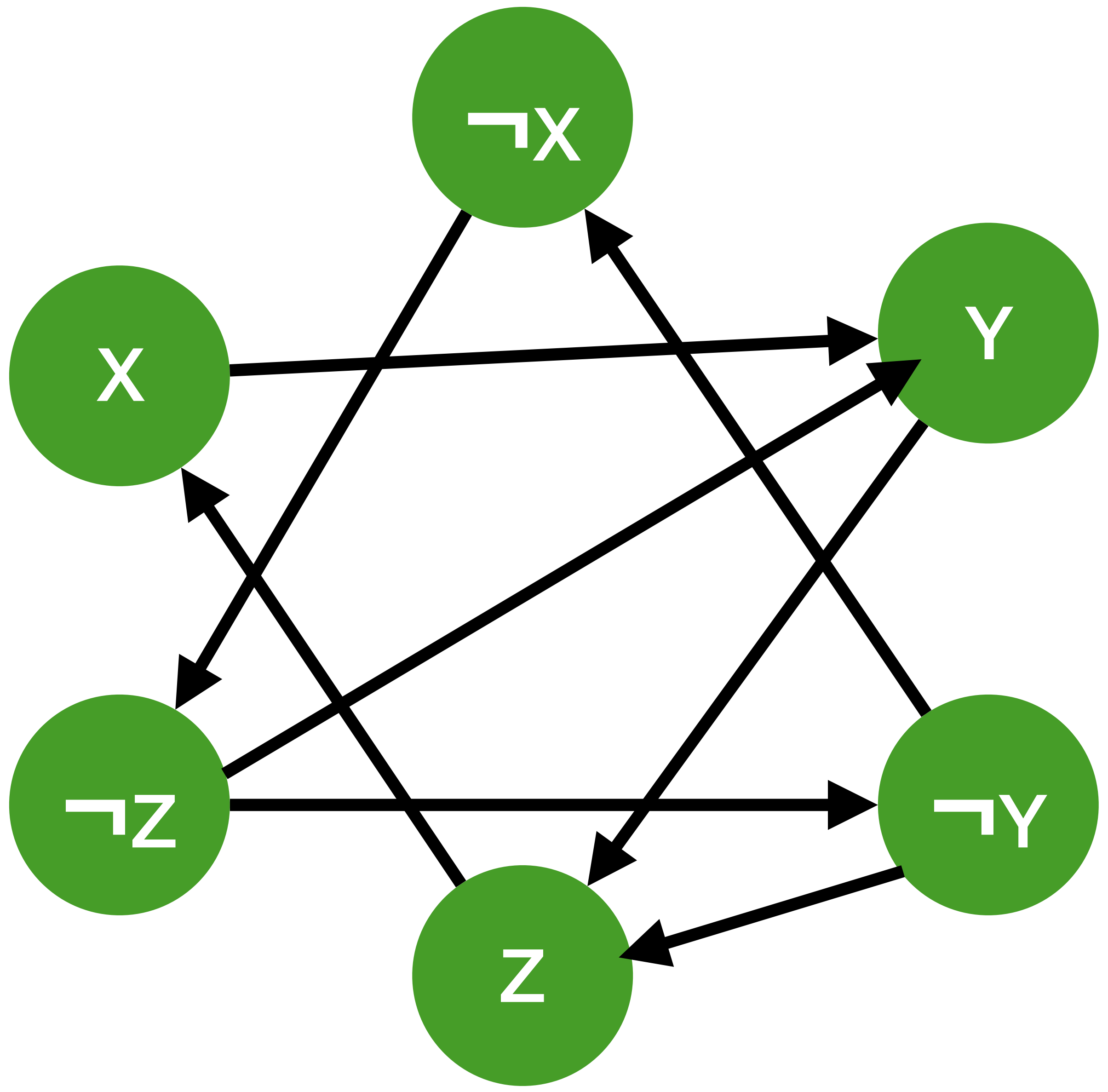
If there is a literal and its complement in the same SCC, ∅ is not satisfiable.



Implication Graph

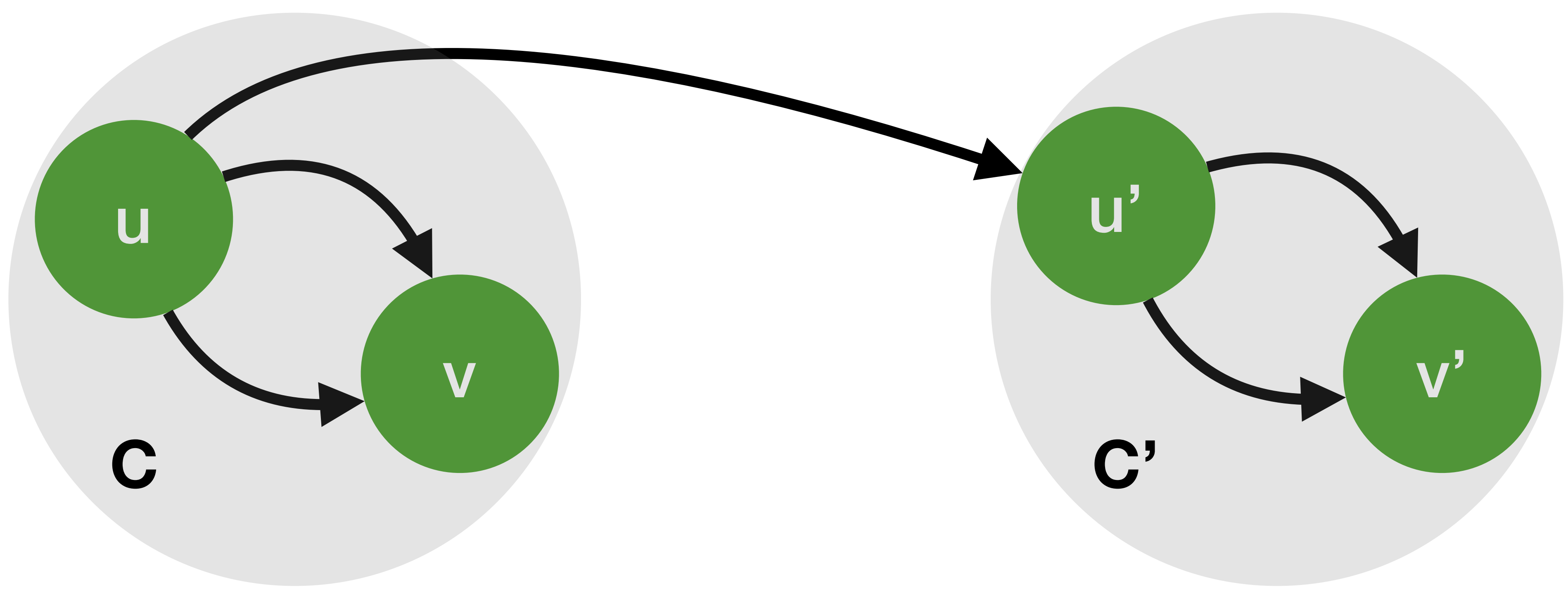
Exercise

Draw the implication graph for (¬X ∨ Y) ∧ (¬Y ∨ Z) ∧ (X ∨ ¬Z) ∧ (Z ∨ Y).



Kosaraju - Sharir algorithm

Lemma : Let C and C' be distinct SCCs in directed graph G = (V, E) suppose these is a path u ~> u'



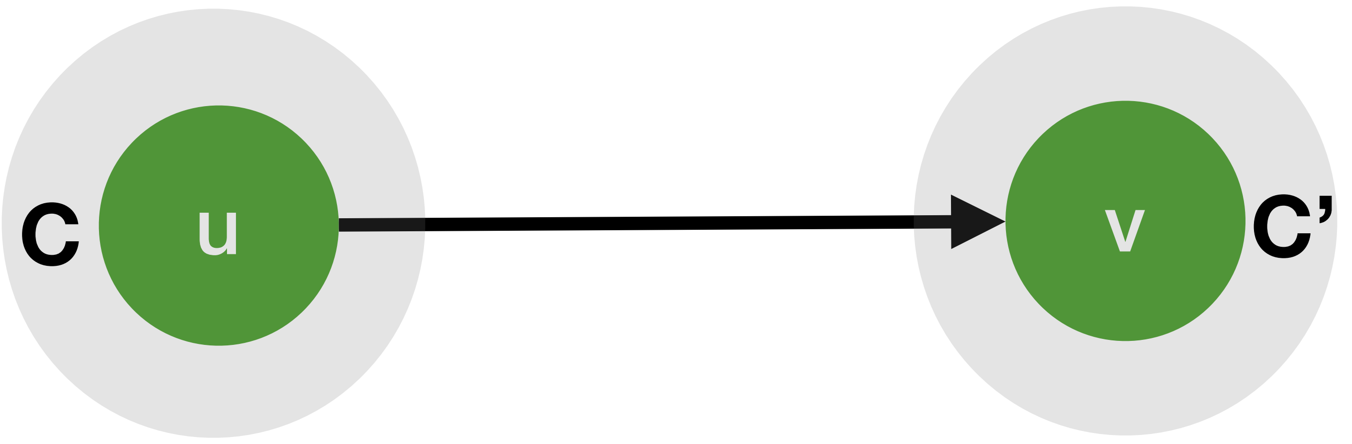
To prove : there cannot be a path v' ~> v

Proof by contradiction

Suppose that there is a path v~>v'. Then there is a path u~>u'~>v' and v'~>v~>u in G. Then u and v' are reachable from each other => C and C' cannot be distinct, which is a contradiction.

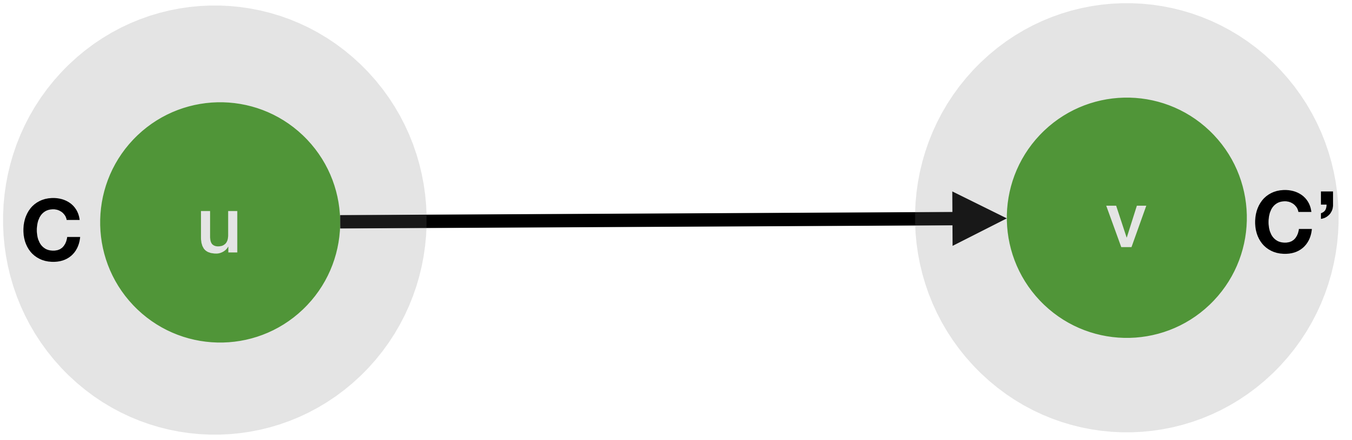
Kosaraju - Sharir algorithm

Lemma : Let C and C' be distinct SCCs in G = (V, E). Suppose there is a edge (u, v) as shown :



Then f(C) > f(C')

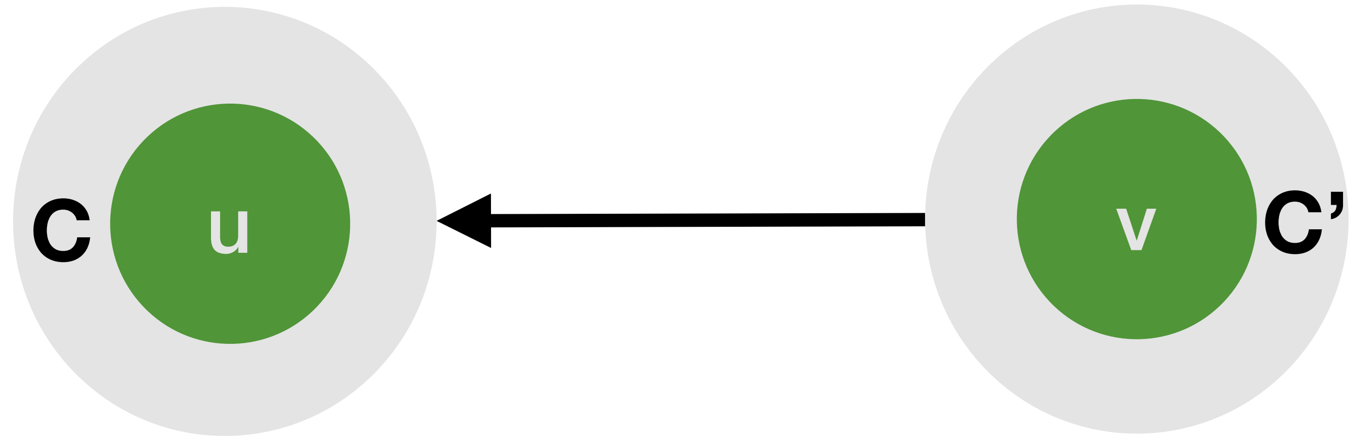
Corollary : let C and C' be distinct SCCs in G = (V, E). Suppose that there is an edge (u, v) in E^T so that



Then f(C) < f(C')

Kosaraju - Sharir algorithm

Corollary : let C and C’ be distinct SCCs in G and f(C) > f(C’)



Then there cannot be edge C to C’ in G^{T}.

References

- Cormen, Thomas H., et al. *Introduction to Algorithms*. The MIT Press, 2014.
- Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In International Parallel and Distributed Processing Symposium, pages 505–511. Springer, 2000.
- William McLendon III, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. Journal of Parallel and Distributed Computing, 65(8):901– 910, 2005.
- Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, page 92. ACM, 2013.
- https://en.wikipedia.org/wiki/2-satisfiability