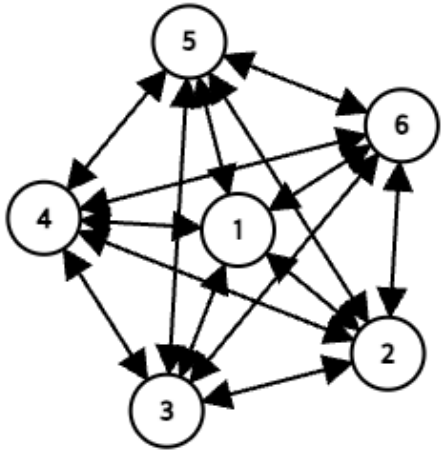
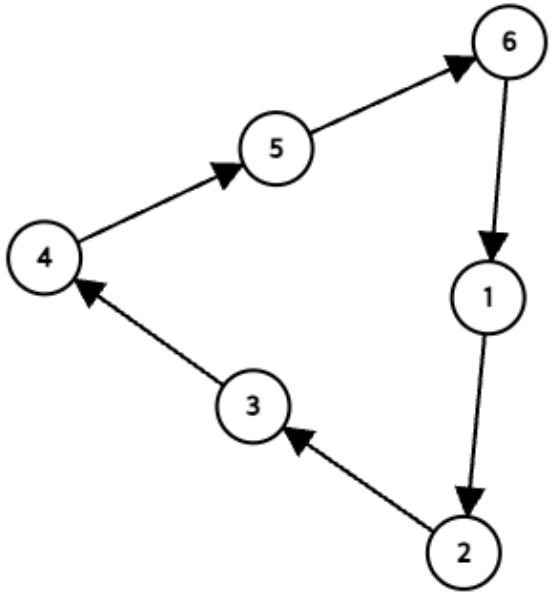
Strongly Connected Component

SCC

- 그래프가 모든 정점에 대하여 다른 모든 정점에 도달이 가능할 때, 강하게 연결되었다 (Strongly Connected)라고 한다





SCC

- 강한 연결 요소
- 부분 그래프의 모든 정점이 강하게 연결되어 있을 때, 부분 그래프의 포함된 모든 정점을 하나의 집합으로 묶는 알고리즘
- 전체 그래프 중에서 강하게 연결되어 있는 부분으로 분할하는 알고리즘

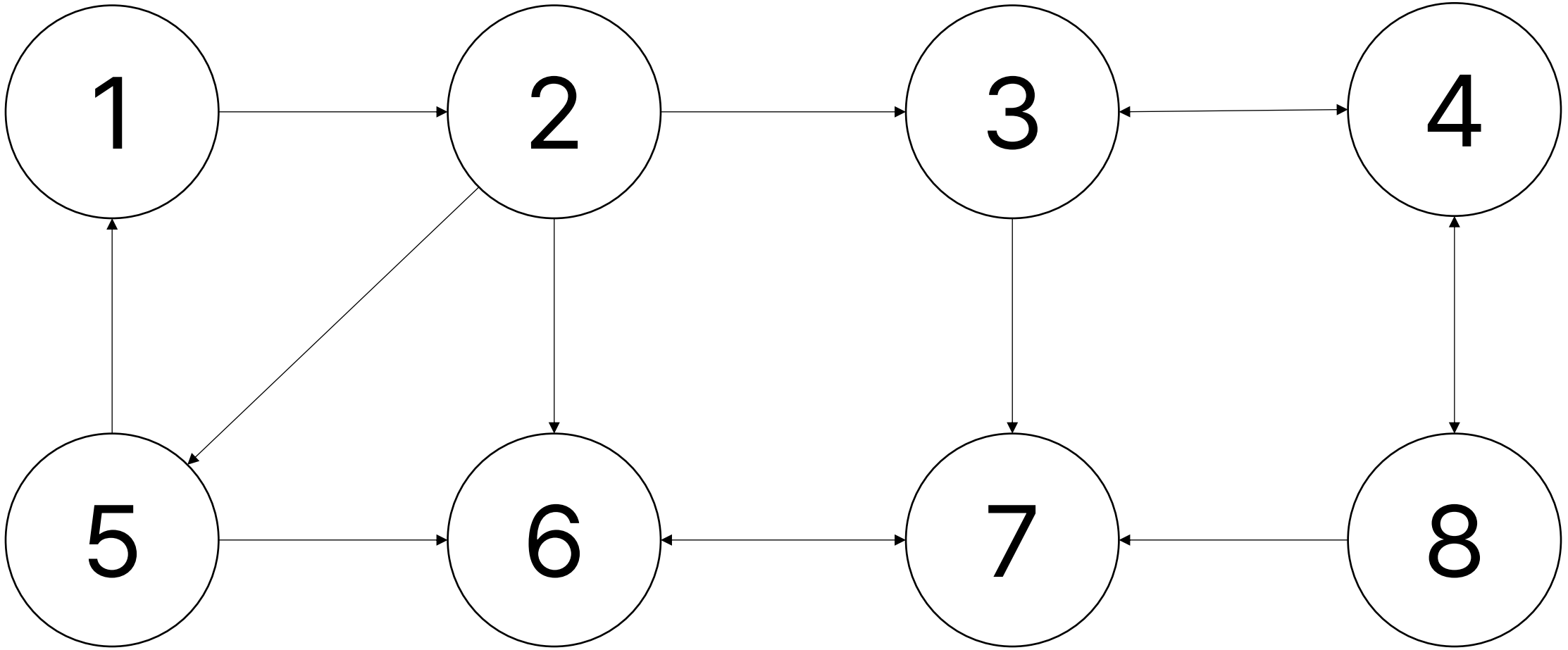
- Kosaraju
- Tarjan

SCC

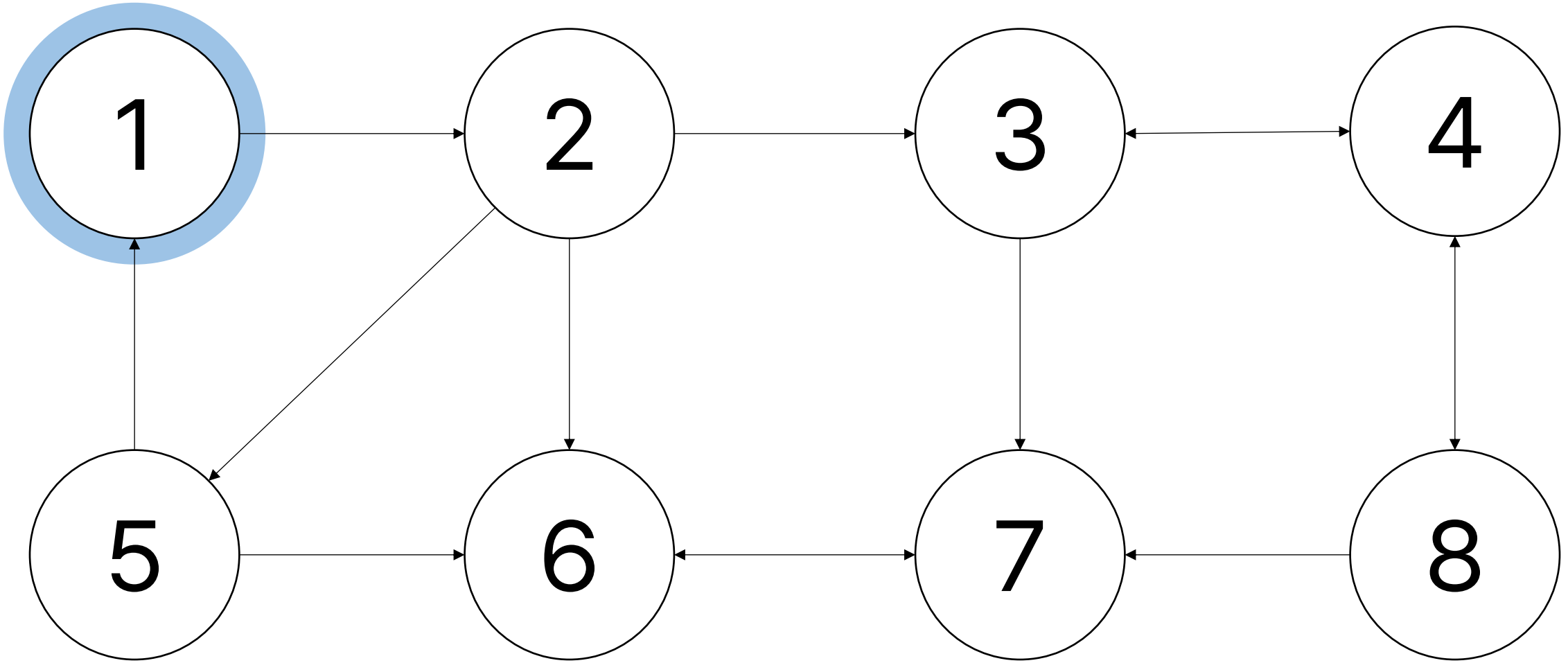
- 임의의 두 정점 u, v가 SCC인 경우, 정점 u와 v 사이에 u에서 v로 가는, v에서 u로 가는 경로가 존재한다

- 임의의 두 정점 u, v가 서로 다른 SCC에 존재할 때, 두 정점 u, v 사이에 u에서 v로 가는, v에서 u로 가는 경로가 동시에 존재하지 않는다

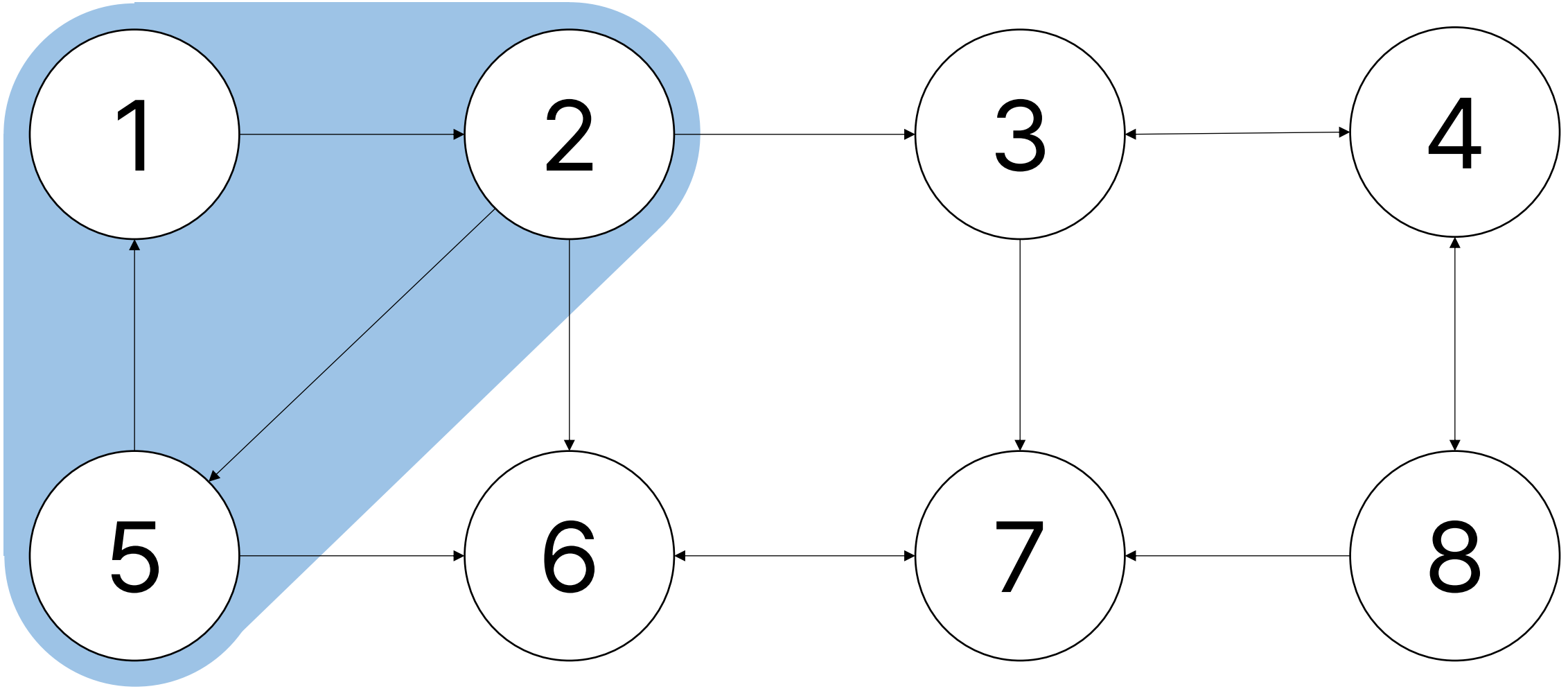
scc



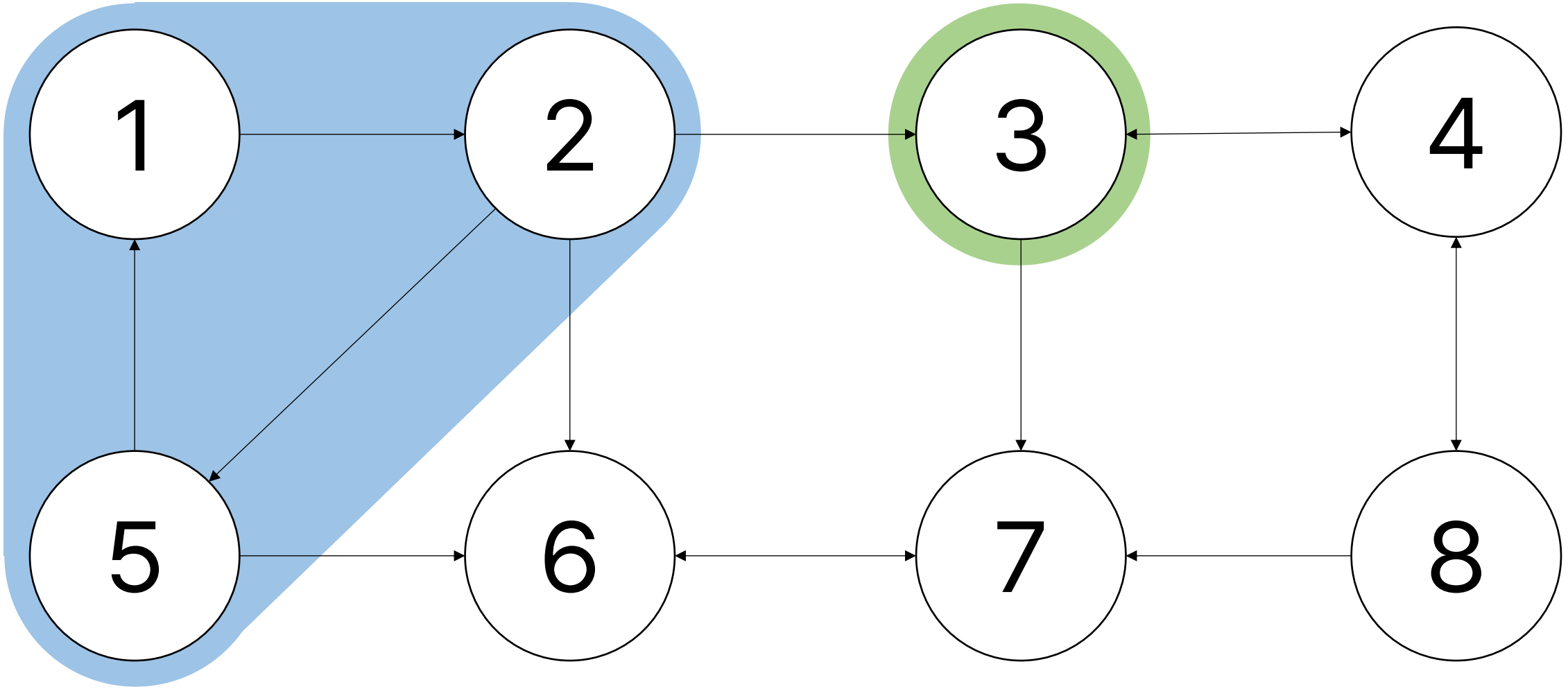
scc



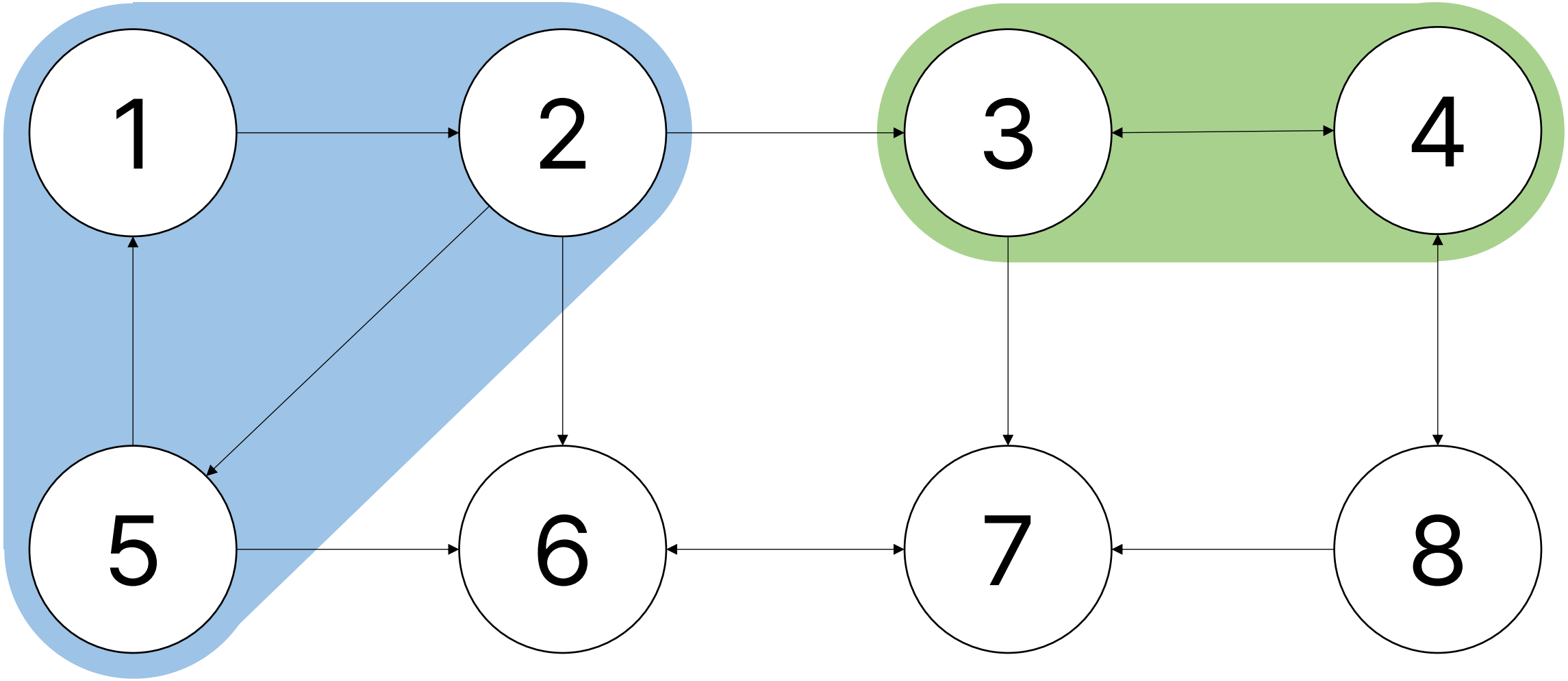
scc



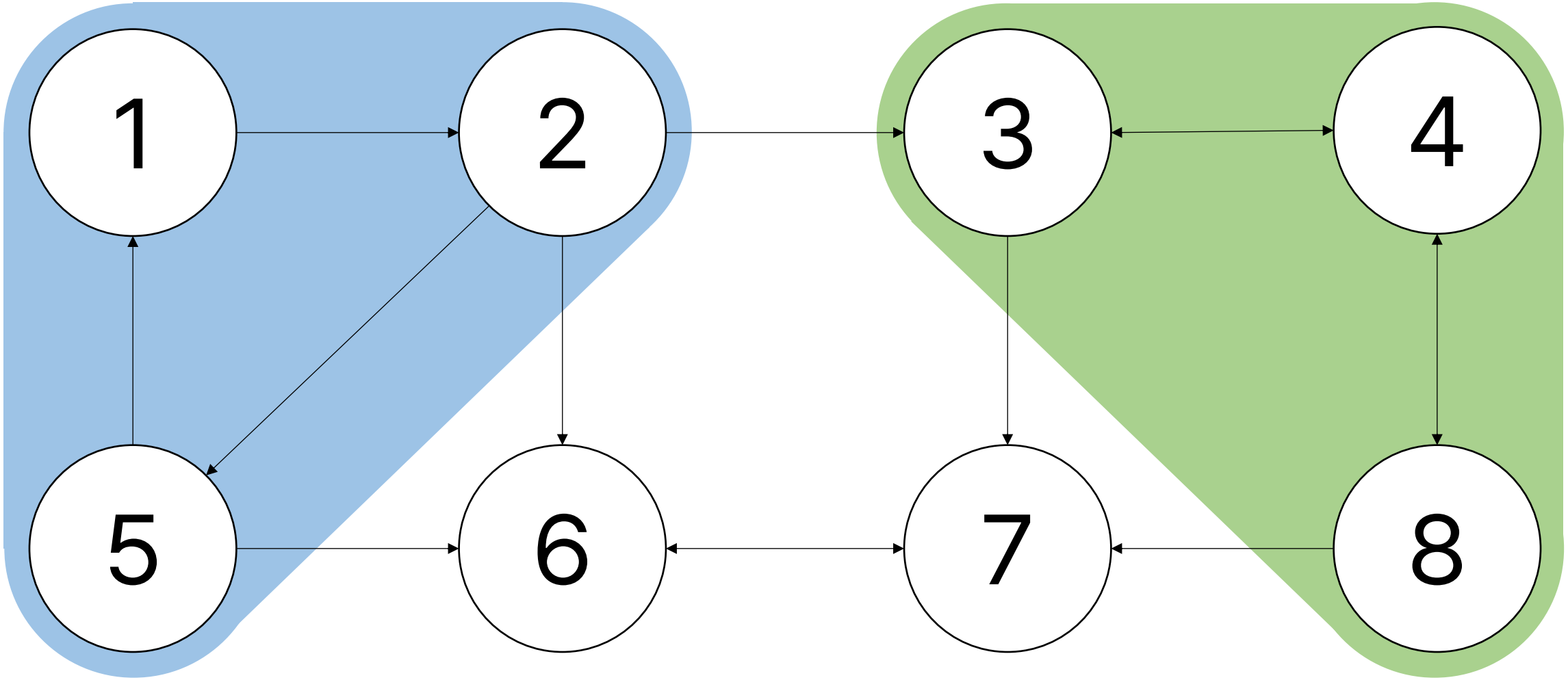
scc



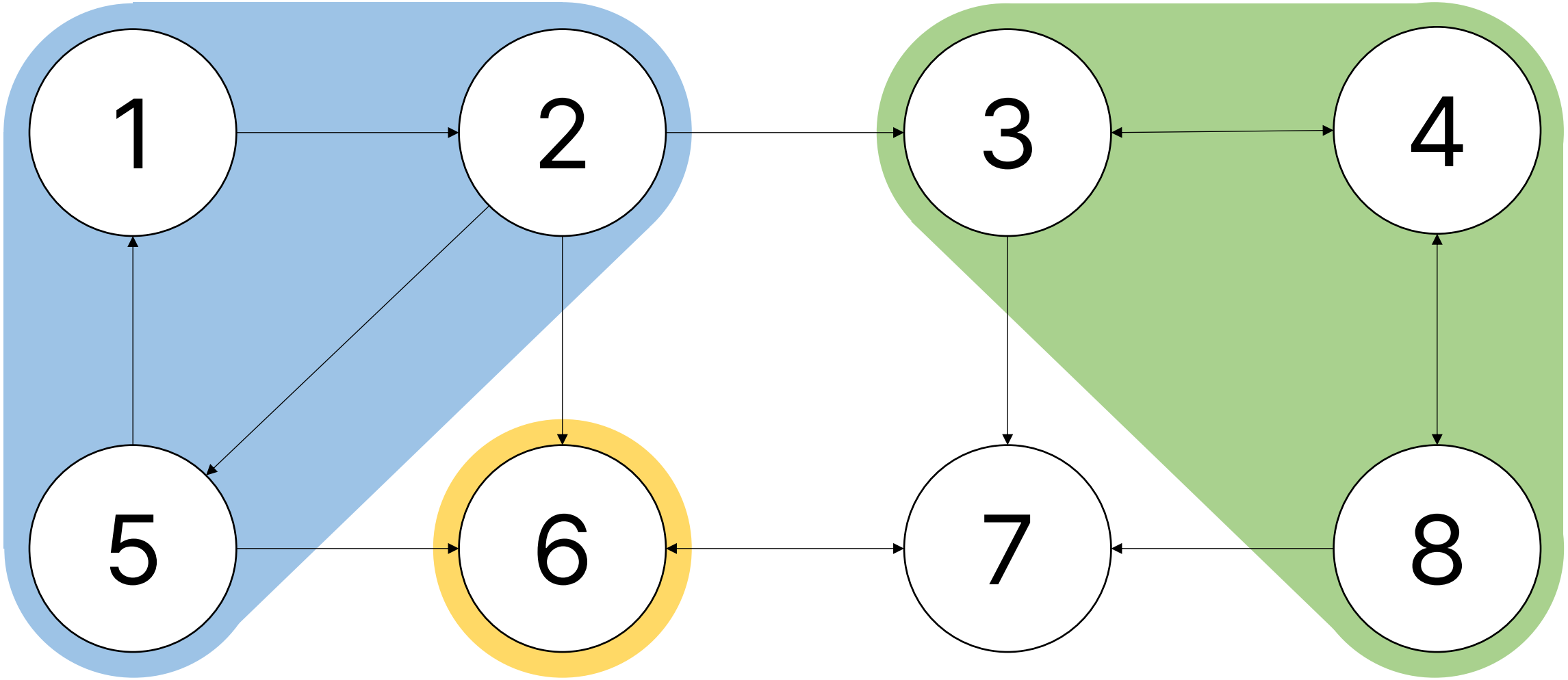
scc



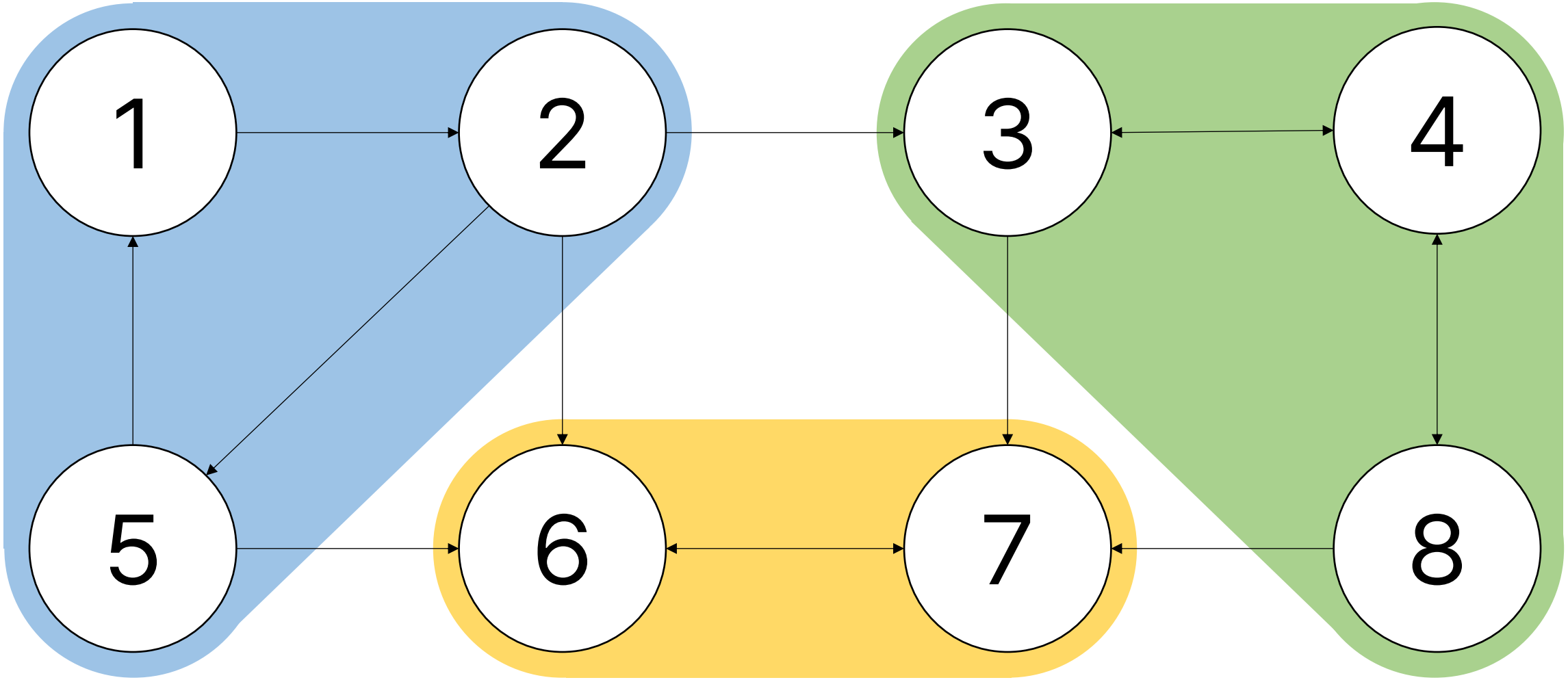
scc



scc



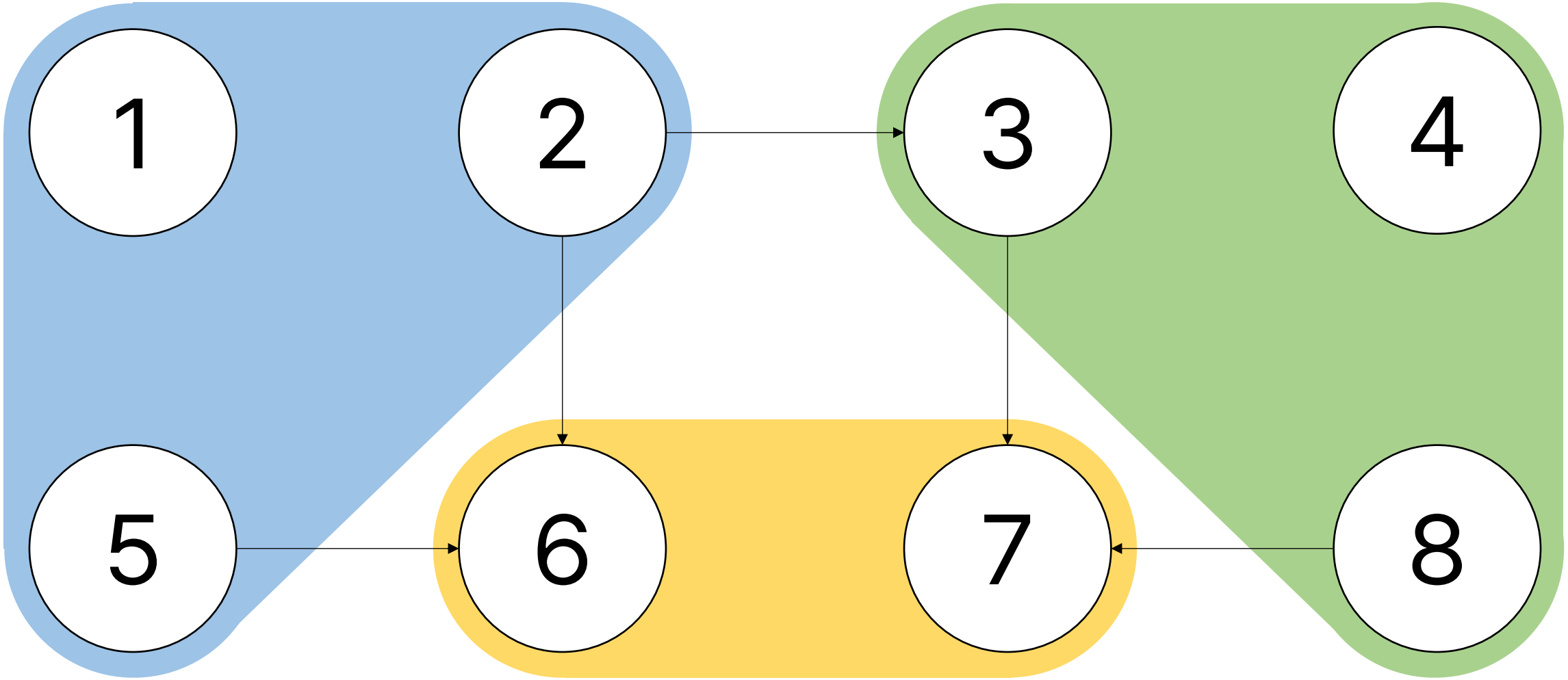
scc



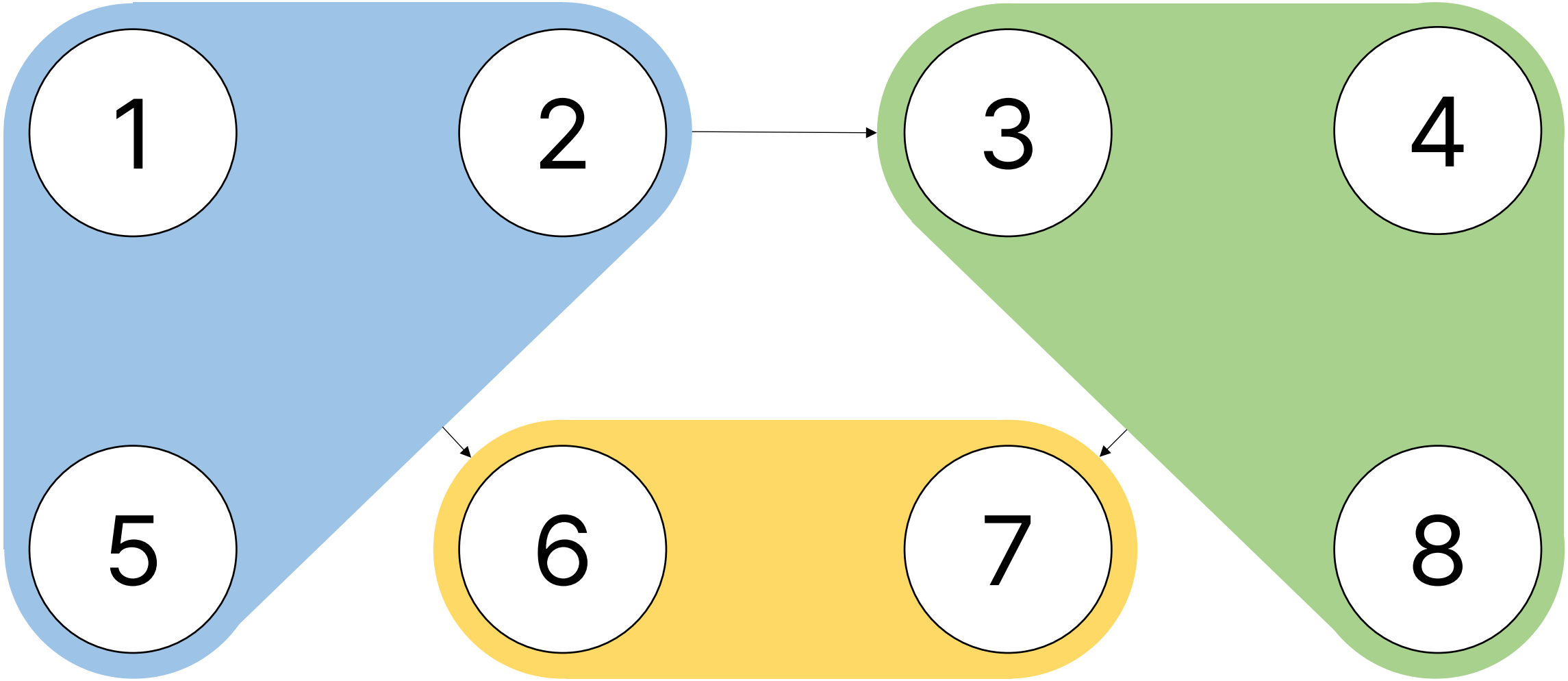
SCC

- 같은 SCC에 들어있는 노드들을 하나의 노드로 묶으면 사이클이 없는 방향 그래프(DAG)를 형성한다
- 기존에 위상정렬이 불가능한 그래프를 위상 정렬이 가능한 형태(DAG)로 변환이 가능하다

scc



scc



DFS?

- 우선 DFS를 돌려보자
- DFS를 돌리면 DFS tree를 형성하게 되는데 어느 노드에서 시작하느냐, 어느 간선으로 먼저 이동하냐에 따라서 DFS tree의 모양은 다르다
- 하나의 SCC 속하는 노드들이 여러 개의 DFS 트리에 있지는 않다
- DFS트리는 여러 개의 SCC로 구성되어 있다

DFS?

- 1번 트리와 2번 트리가 있을 때, 1번 트리가 먼저 만들어졌다고 가정하자
- 같은 SCC에 속하는 다른 노드 둘이 각각 1번 트리와 2번 트리에 속한다면 1번 트리와 2번 트리 사이에는 서로 오갈 수 있는 경로가 존재하게 된다
- 1번 트리에서 2번 트리로 가는 간선은 존재할 수 없으므로, DFS 트리 여러 개에 SCC가 뿌려질 수 없다.
- SCC는 하나의 DFS 트리에 속한다

DFS Tree

- 하나의 DFS 트리에서는 정방향 간선, 역방향 간선이 존재함

  정방향: DFS트리를 따라서 내려가는 간선

  역방향: 이미 방문한 노드로 올라오는 간선

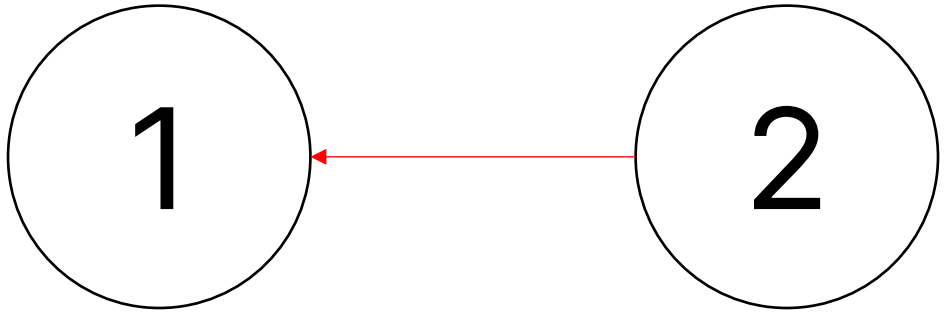
- 역방향 간선이 존재하는 경우, 사이클을 만들게 된다. SCC에 포함된다
- 정방향 간선은 SCC에 포함된다는 보장이 없다
- 다른 DFS 트리로 가는 간선은 절대 SCC에 포함되지 않는다

간선을 뒤집는다면?

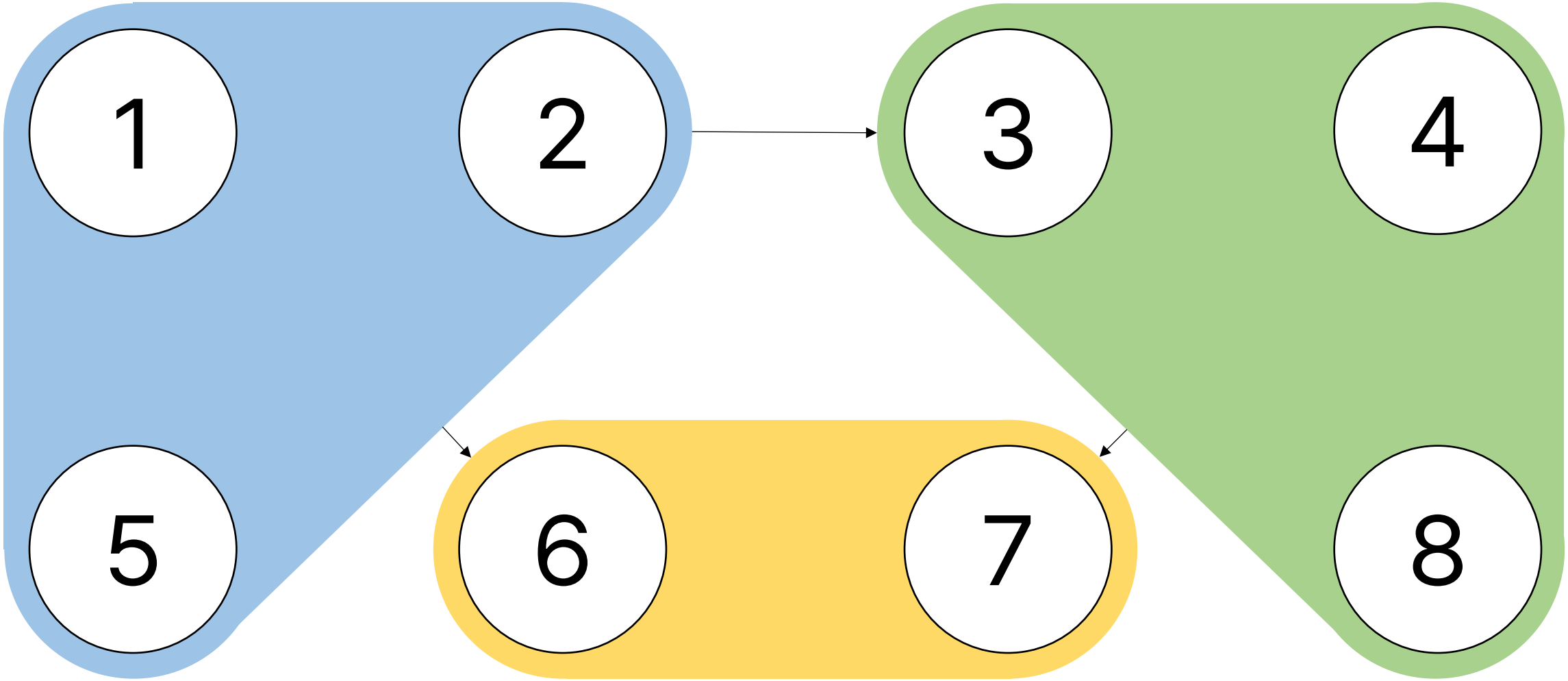
- 같은 SCC에 포함되어 있는 서로 다른 정점 u, v
- SCC 정의에 의해 u->v로 가는 경로와 v->u로 가는 경로가 존재하므로 간선을 모두 뒤집어도 하나의 SCC에 들어간다

간선을 뒤집는다면?

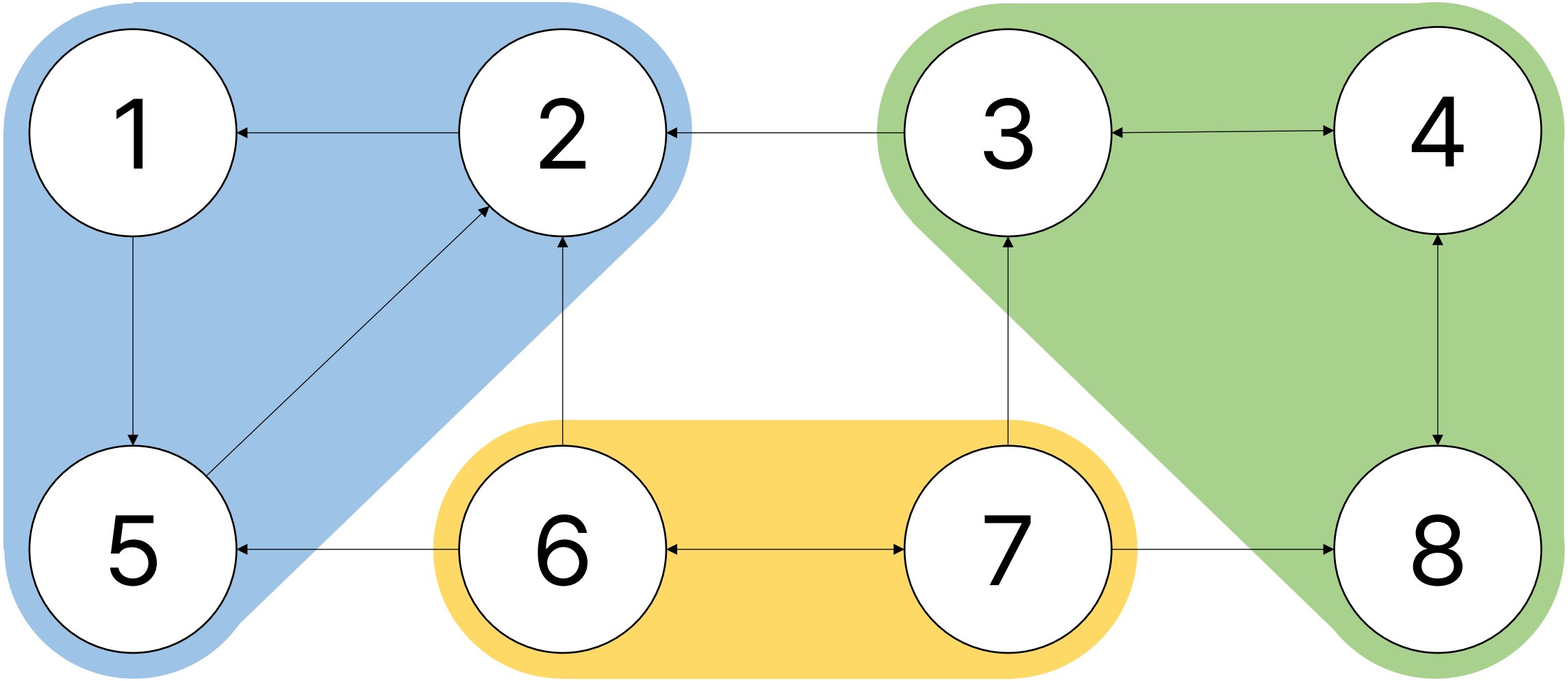
- SCC를 정점으로 생각해보자
- 1번 SCC에 포함되는 정점들은 반드시 2번 SCC에 포함되는 정점들보다 늦게 DFS가 끝나게 된다.
- 제일 마지막에 끝난 SCC에 포함된 노드에서 DFS를 시작하면 같은 SCC들만 방문하게 된다.
- 하나의 SCC로 묶는다, 다시 반복



정방향 그래프, SCC 묶음



역방향 그래프



Kosaraju

- 주어지는 간선을 통해 만드는 그래프, 기존 그래프의 모든 간선의 방향을 뒤집은 역방향 그래프를 이용한 SCC 알고리즘
- 간선을 뒤집어도 같은 SCC에 속하는 노드들은 간선을 뒤집어도 같은 SCC이다
- SCC에 속해 있는 서로 다른 노드 u, v에 대하여 경로가 존재한다

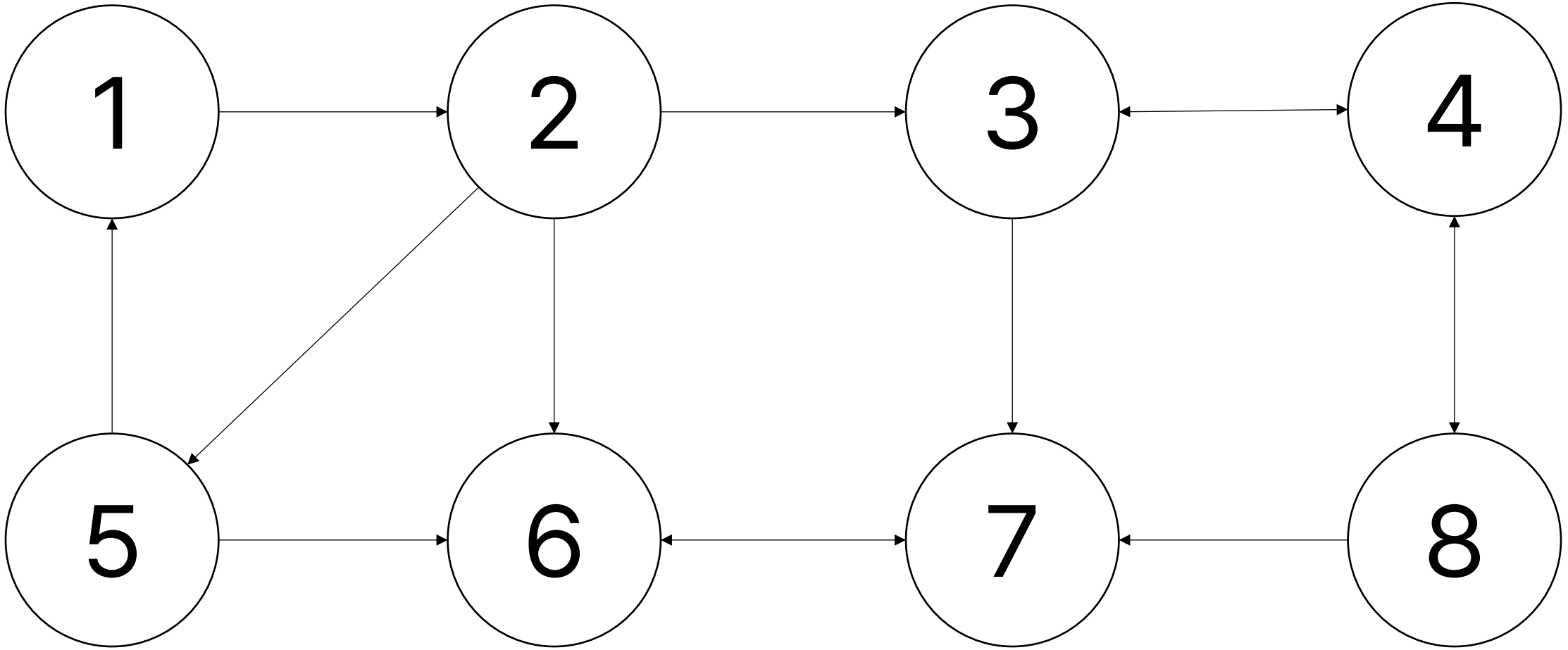
Kosaraju

- 정방향 그래프에서 DFS를 통해 모든 노드를 방문하며 **DFS가 끝난 순서**를 기록한다
- 역방향 그래프에서 늦게 끝난 노드부터 DFS를 한다
- 역방향 그래프에서 DFS를 할 때 한 번에 방문한 노드들끼리 하나의 SCC로 묶는다

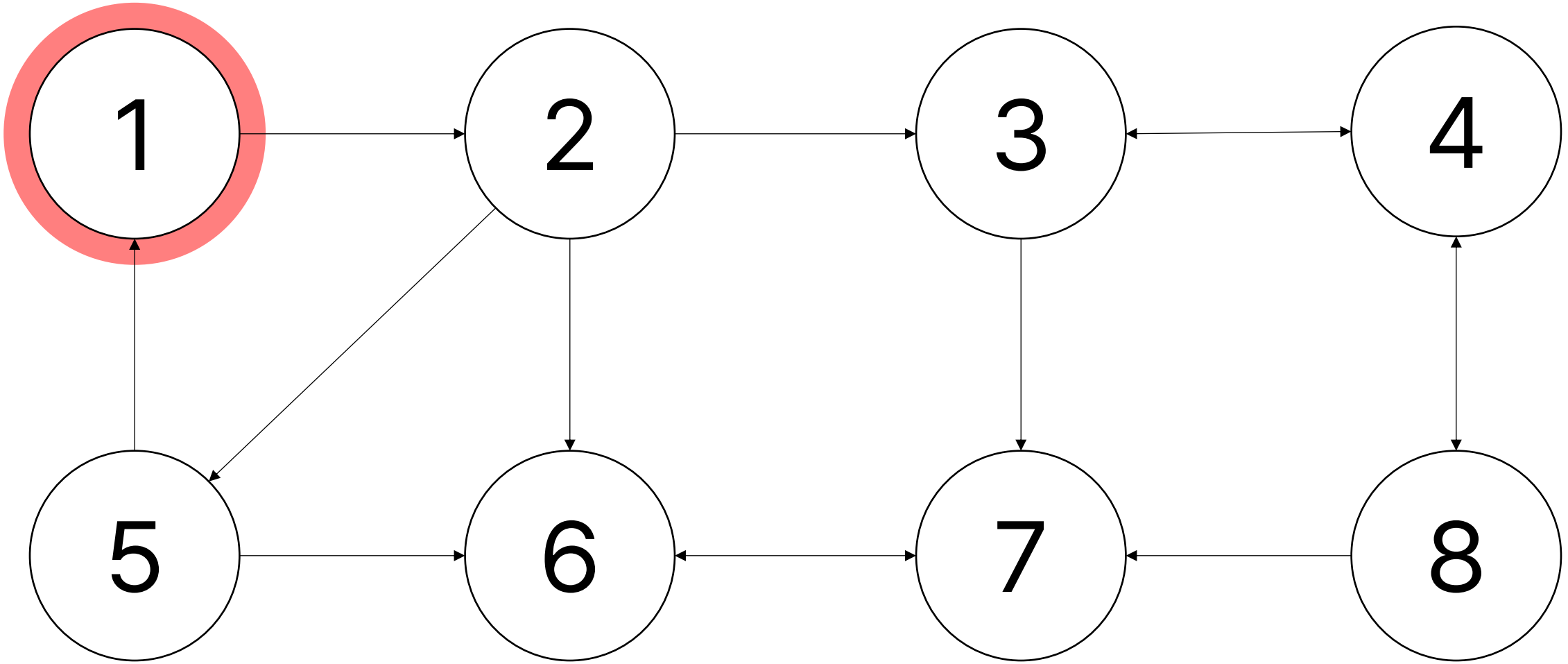
Kosaraju

- 정방향 그래프에서 DFS를 통해 모든 노드를 방문하며 **DFS가 끝난 순서**를 스택에 기록한다
- 스택에서 노드를 하나씩 꺼내며 역방향 그래프에서 DFS를 수행한다
- 한 번에 방문한 노드들끼리 하나의 SCC로 묶는다

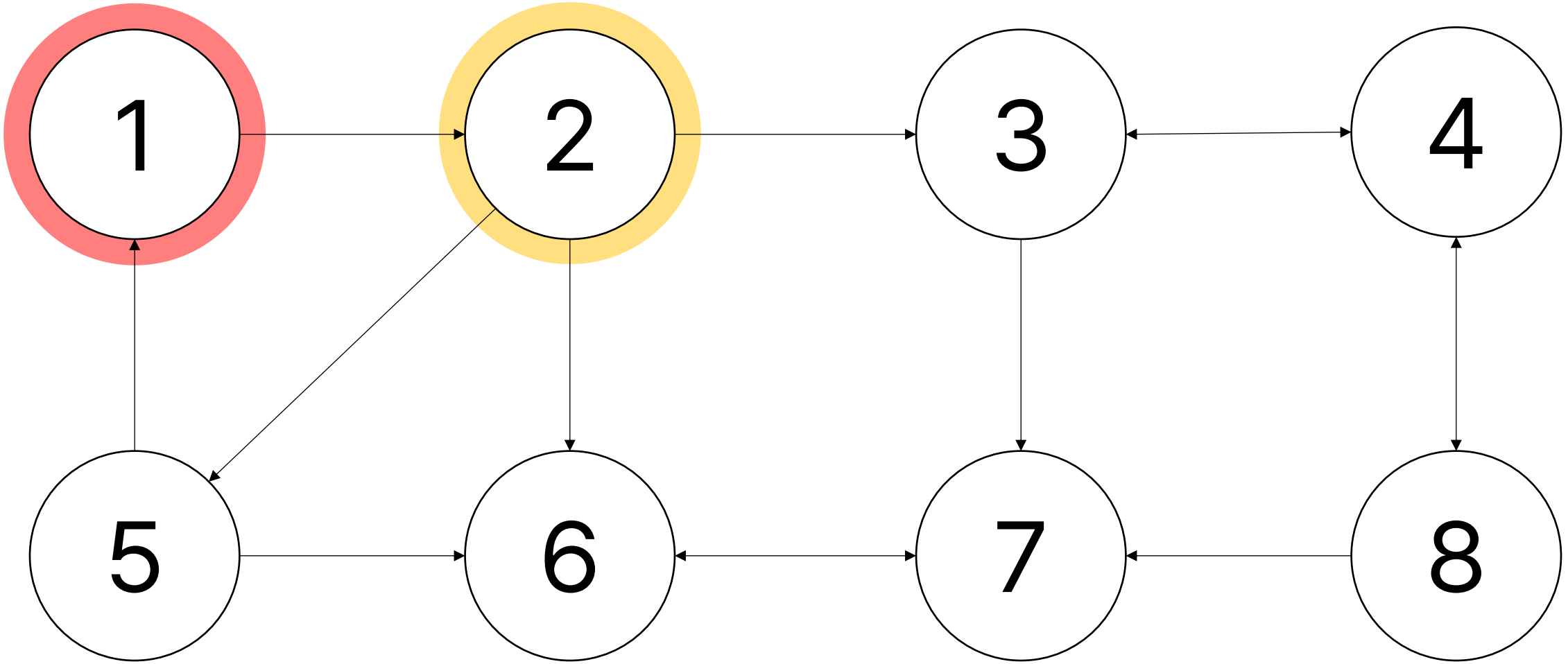
Kosaraju



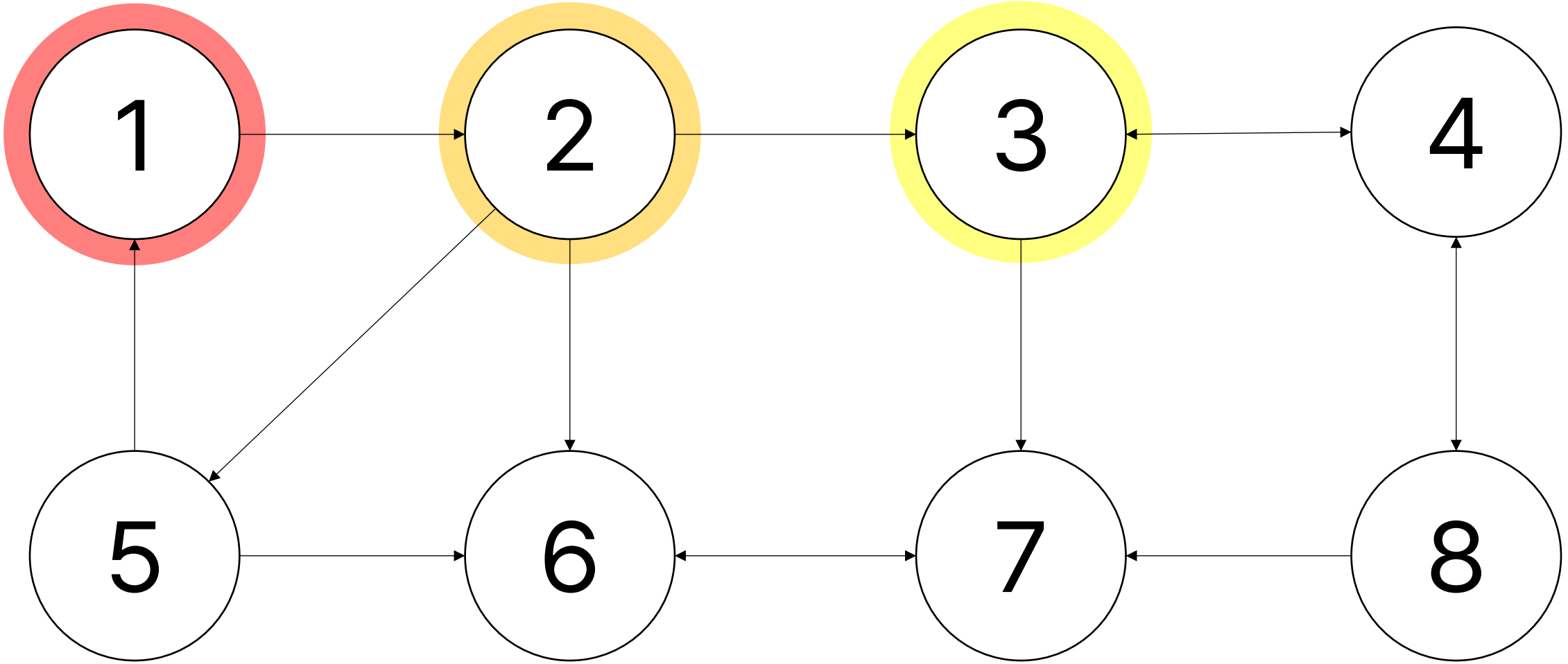
Kosaraju



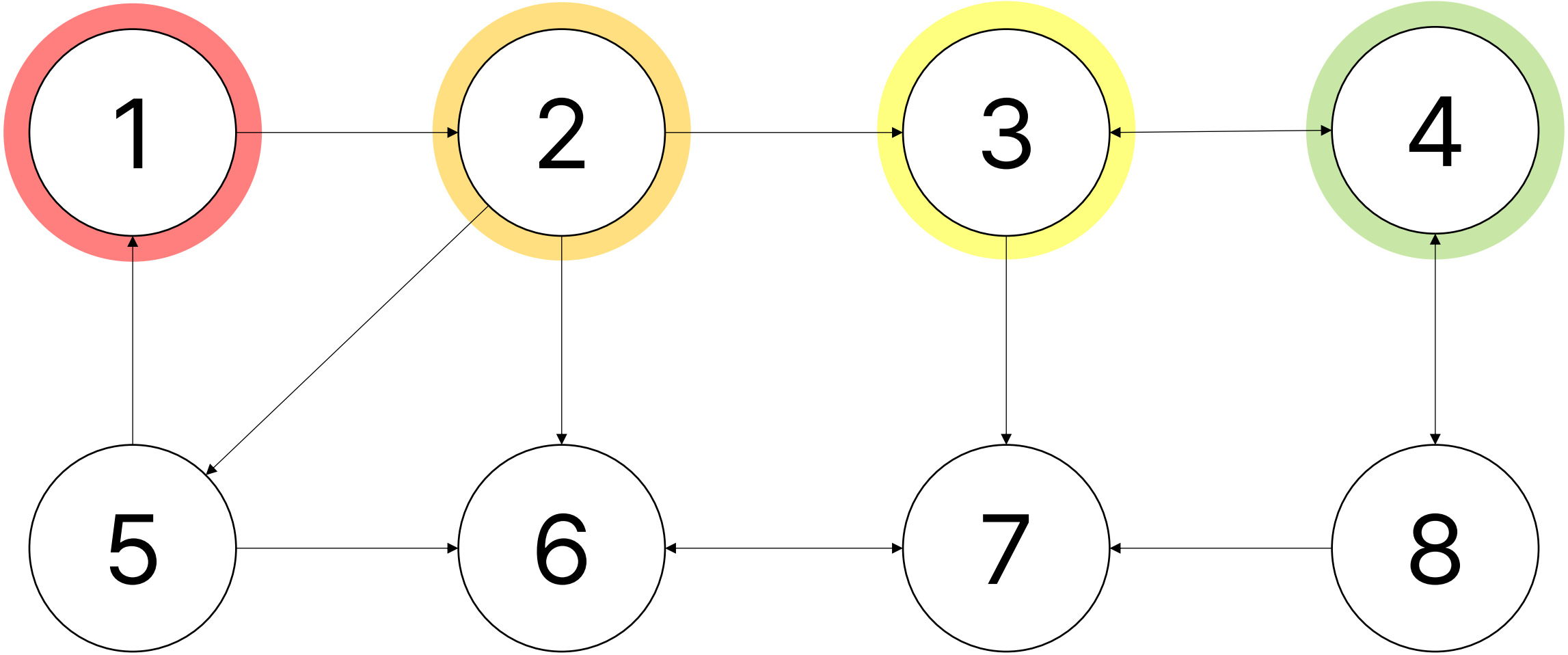
Kosaraju



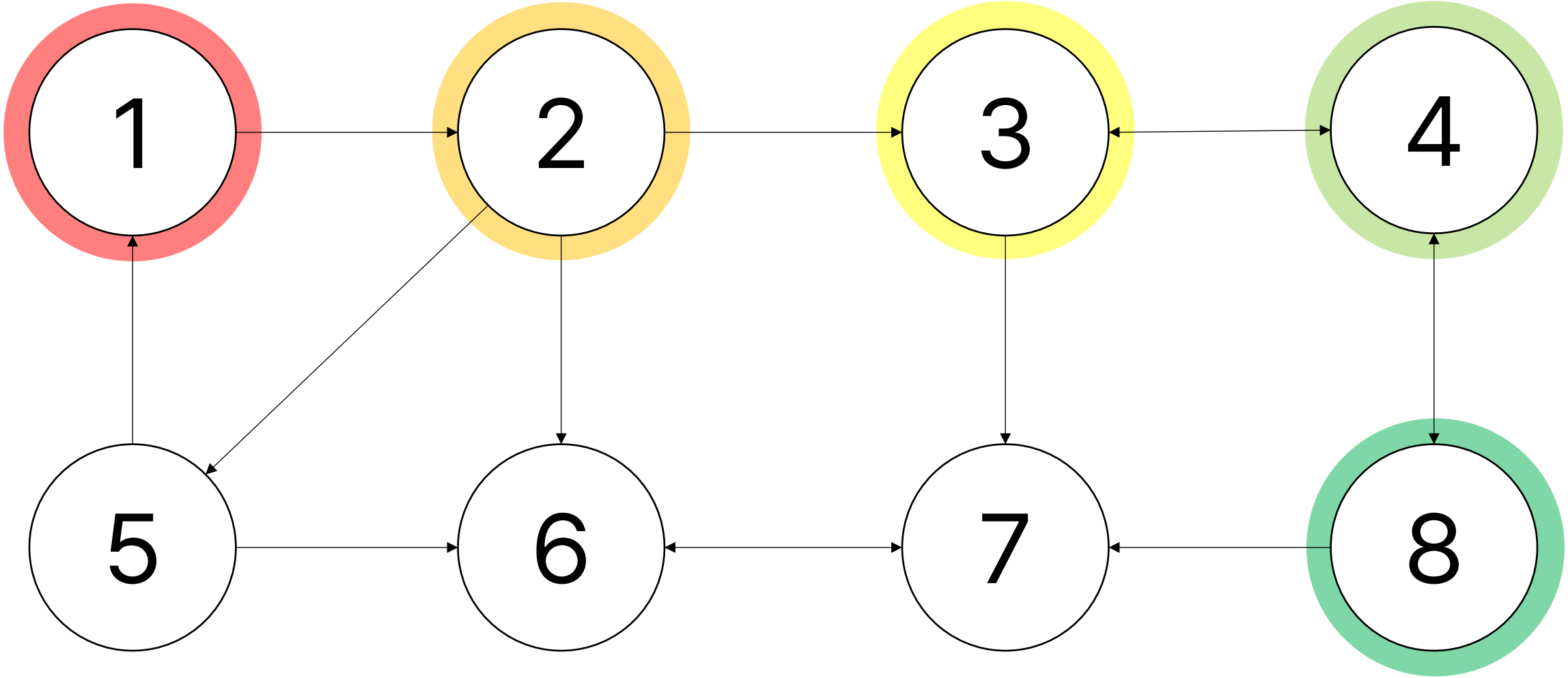
Kosaraju



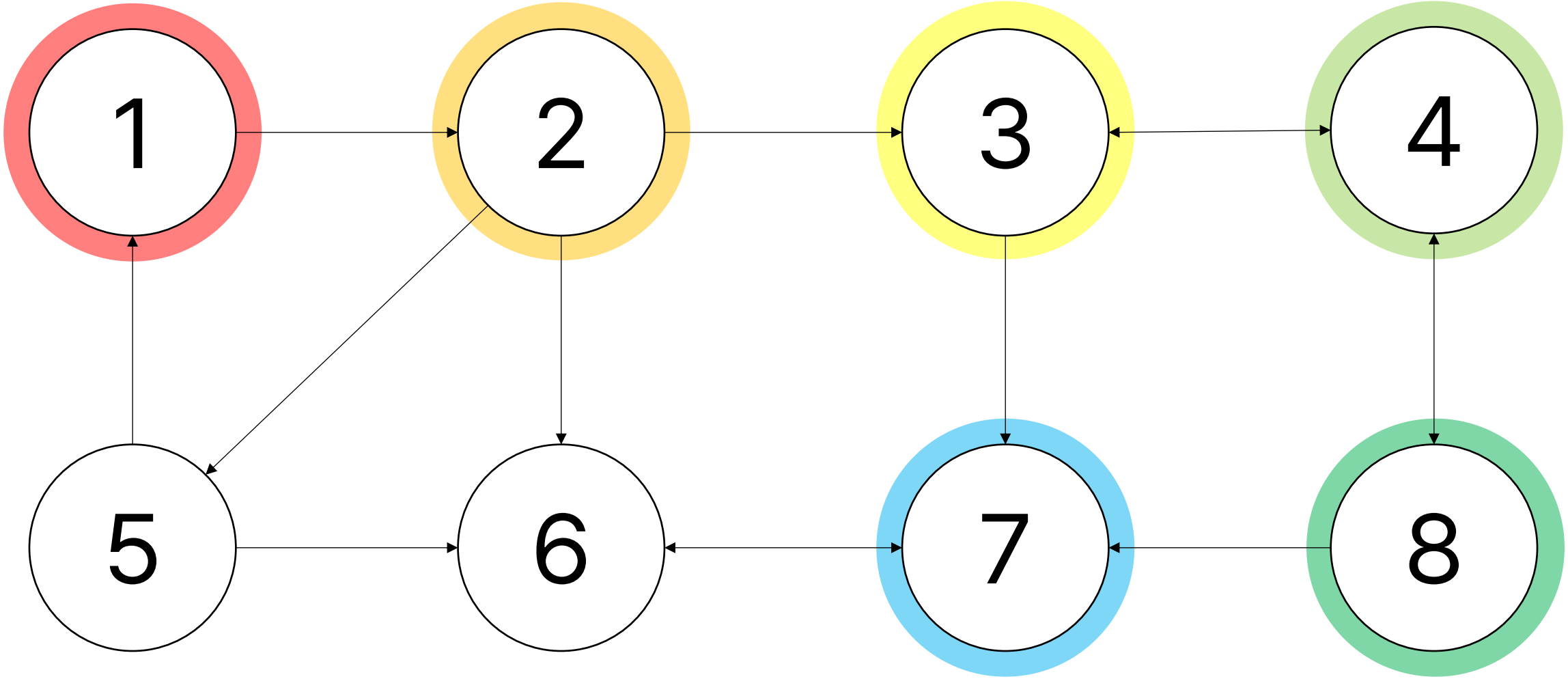
Kosaraju



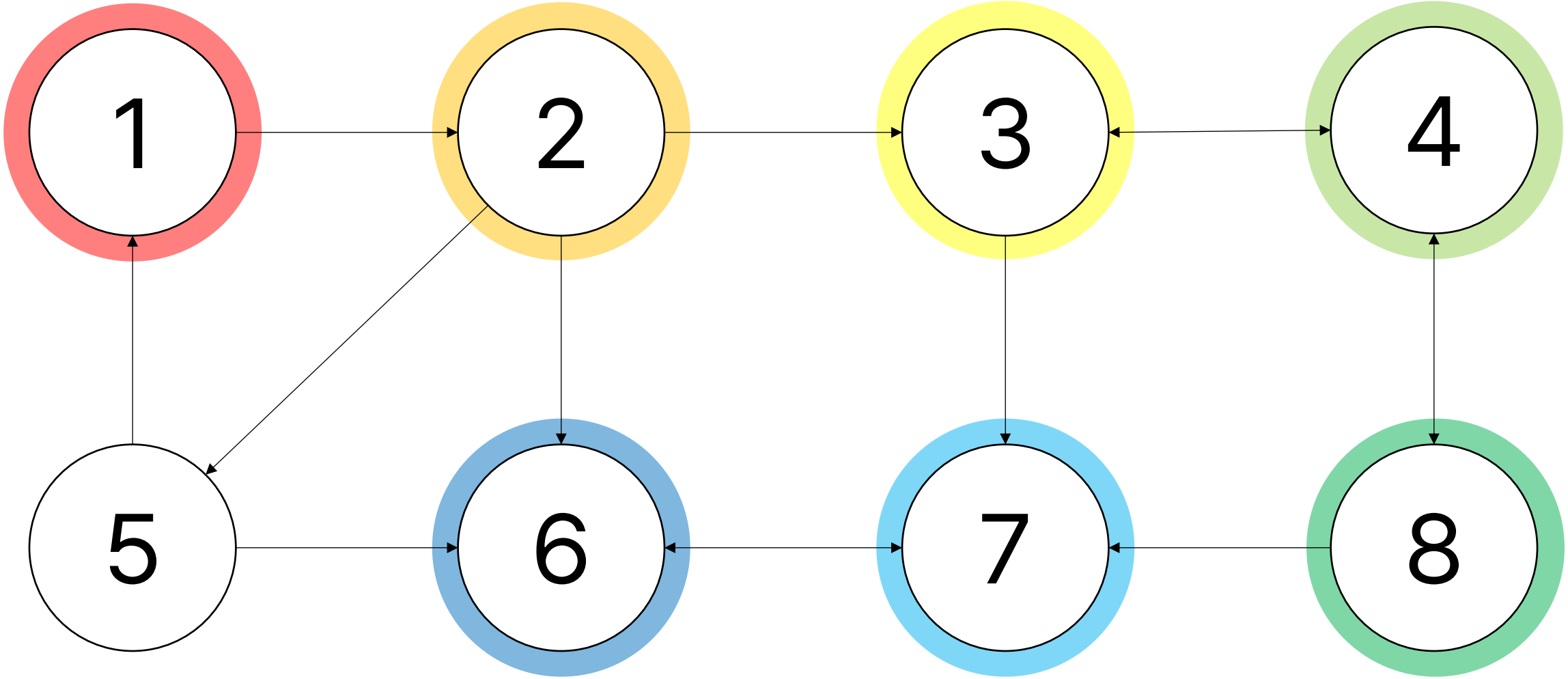
Kosaraju



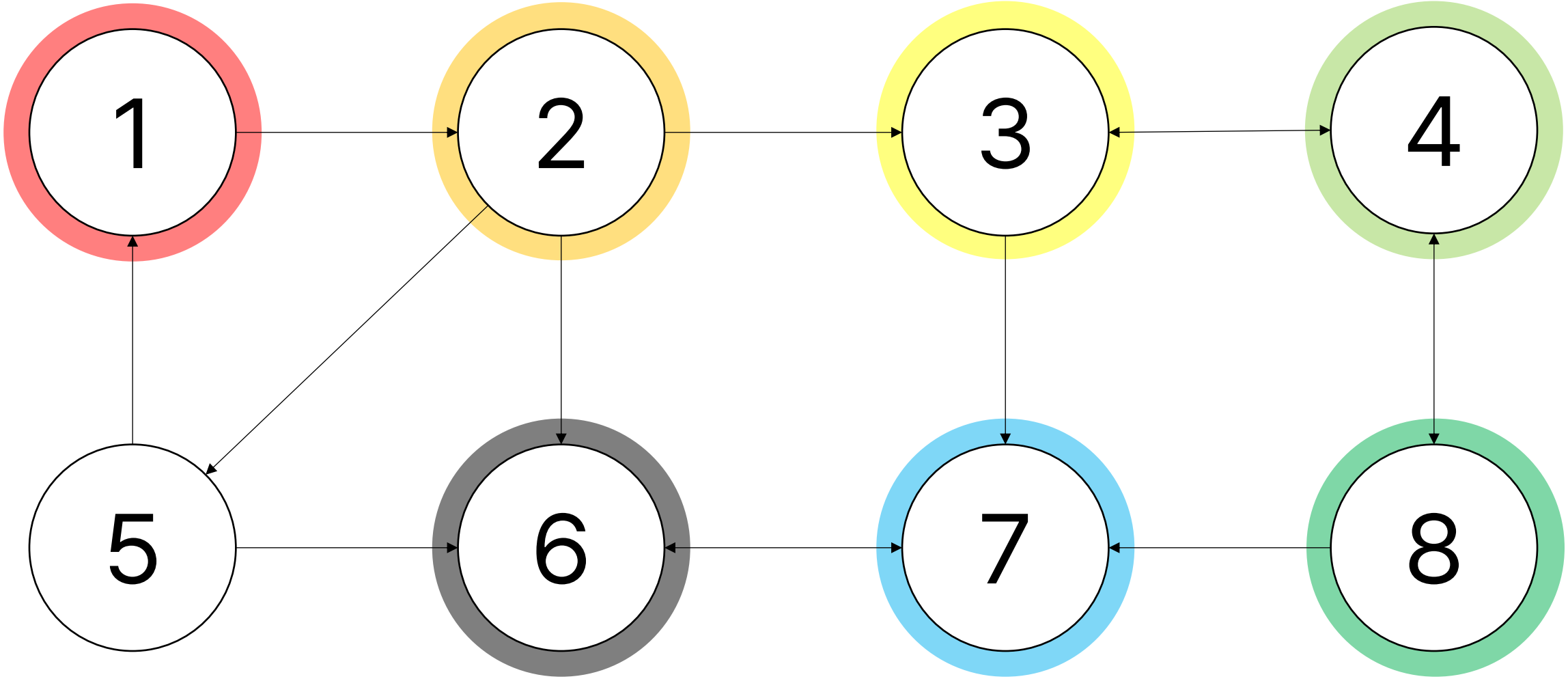
Kosaraju



Kosaraju

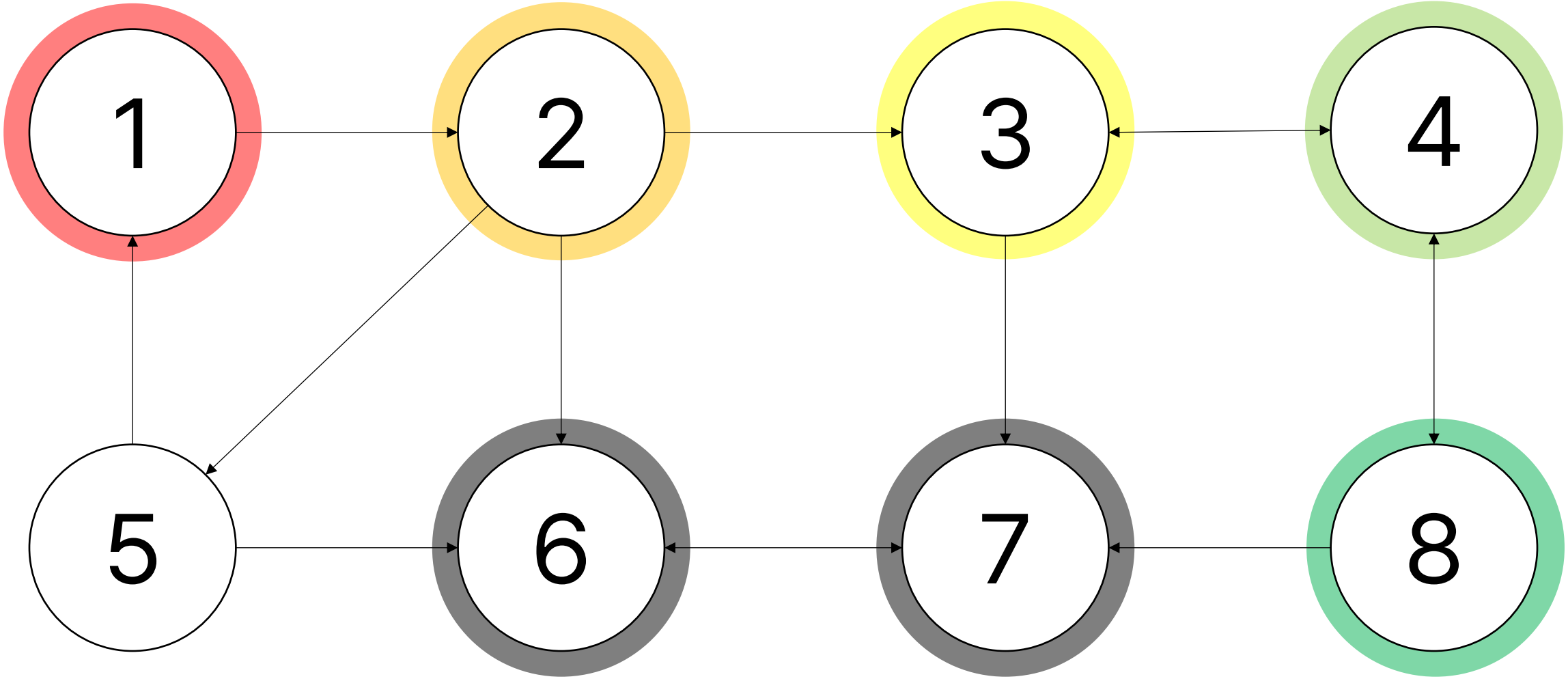


Kosaraju



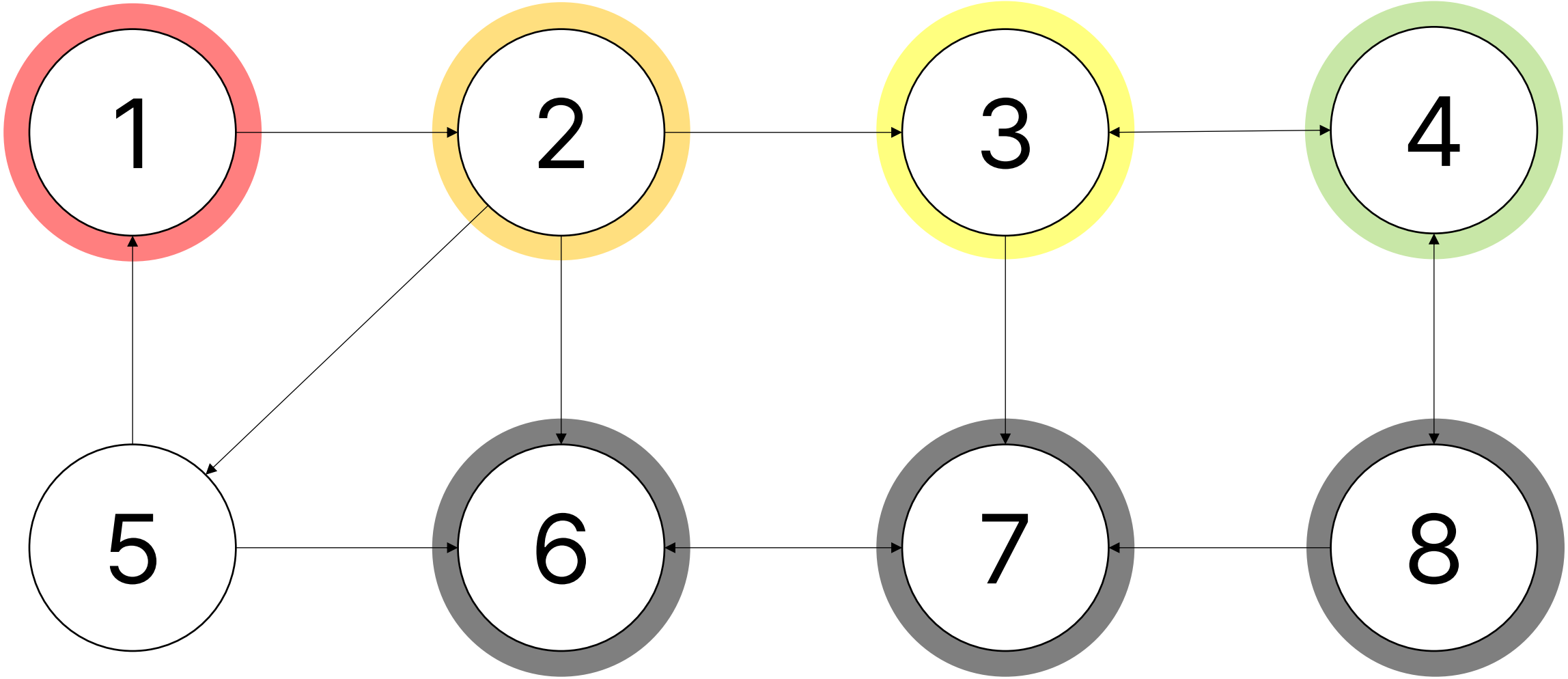
Kosaraju

6	7						



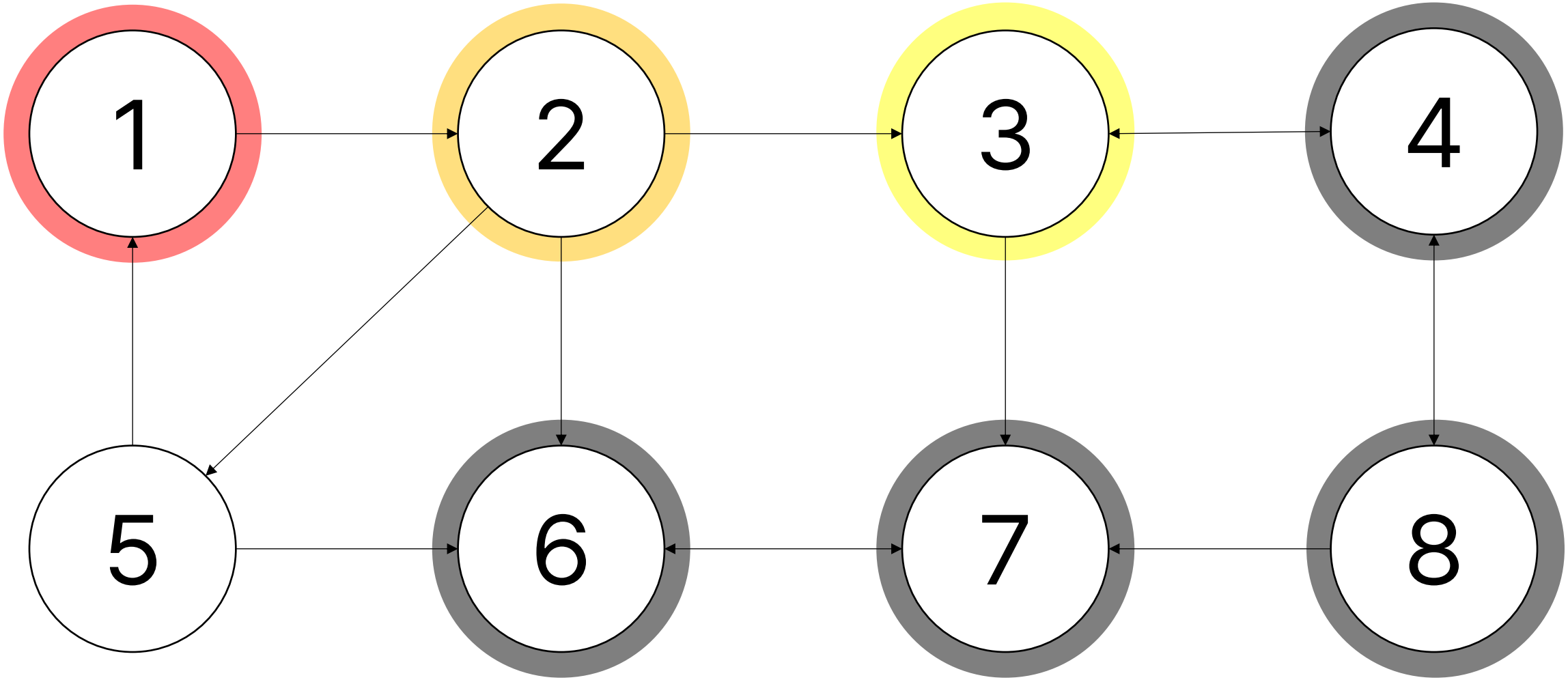
Kosaraju

6	7	8					



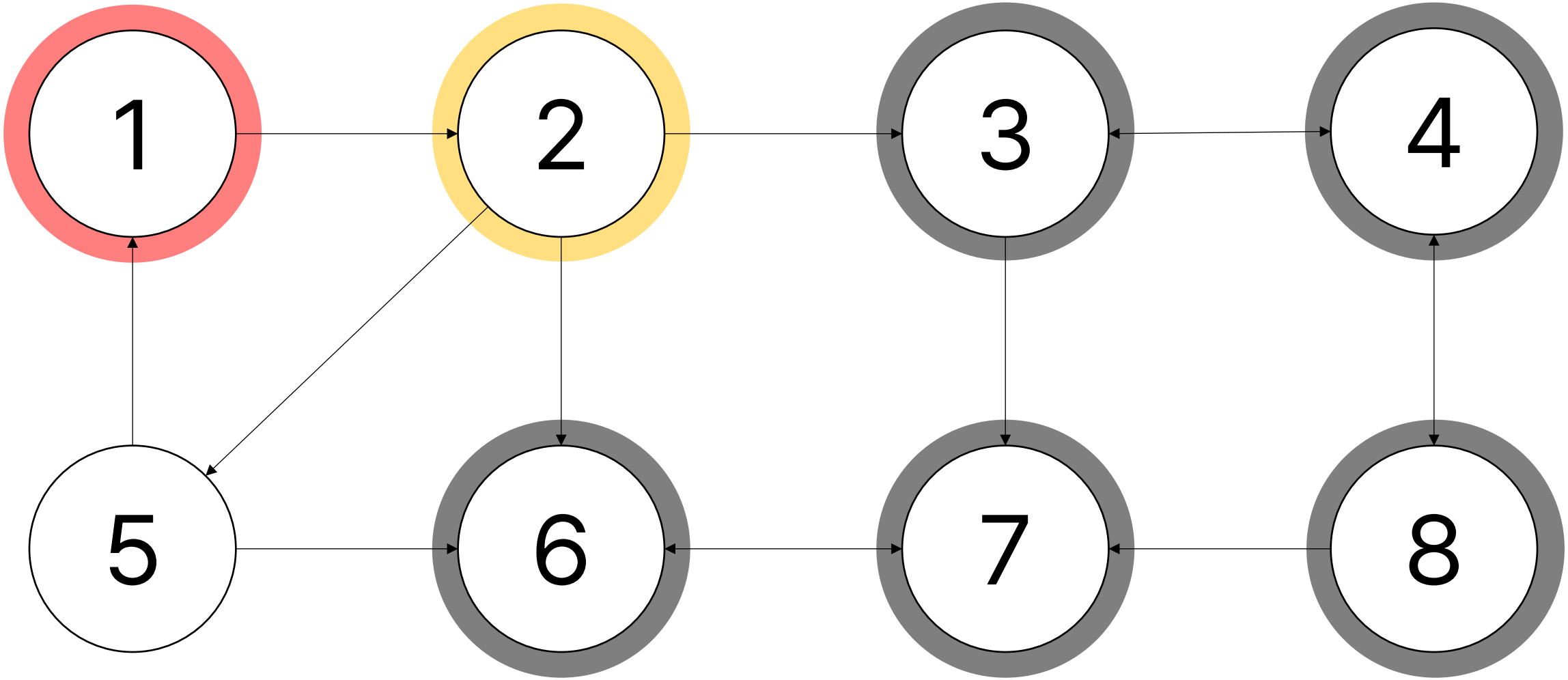
Kosaraju

6	7	8	4				



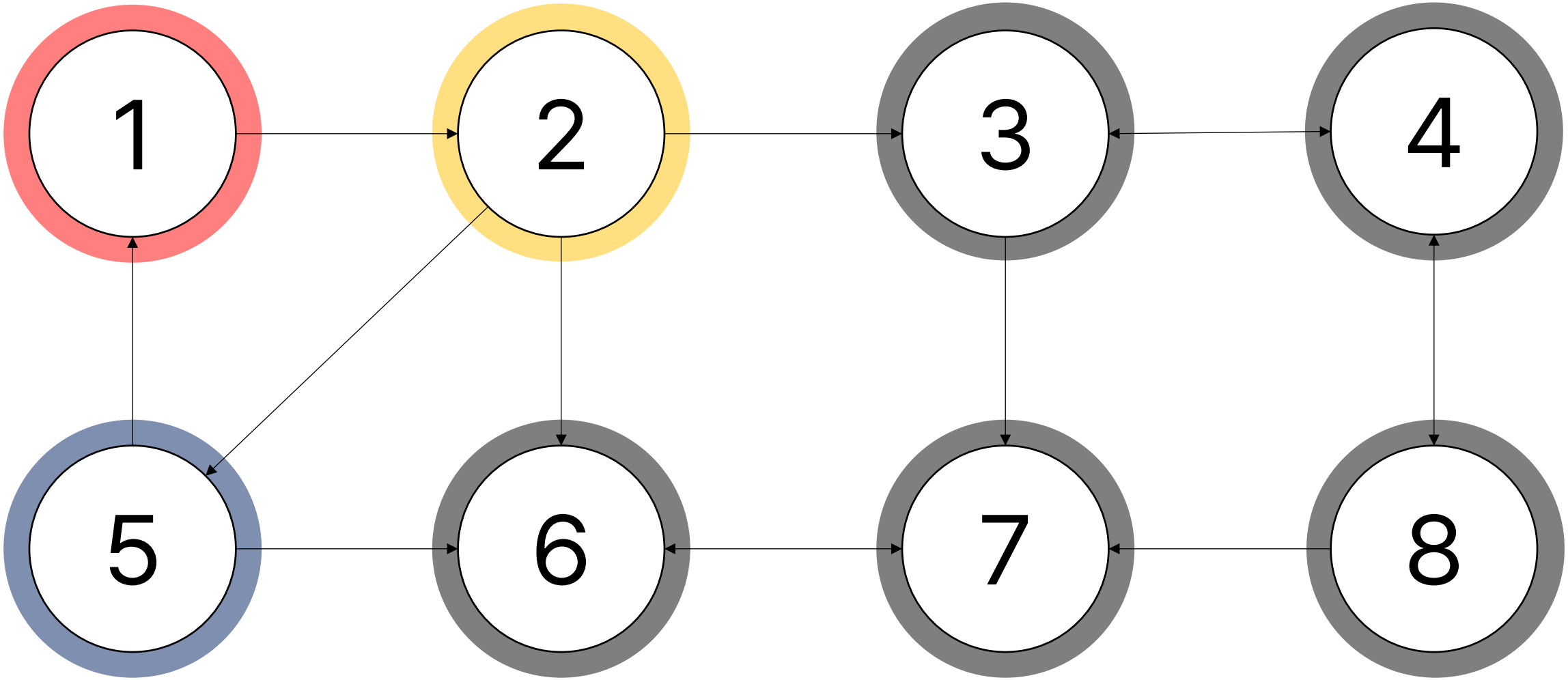
Kosaraju

6	7	8	4	3			



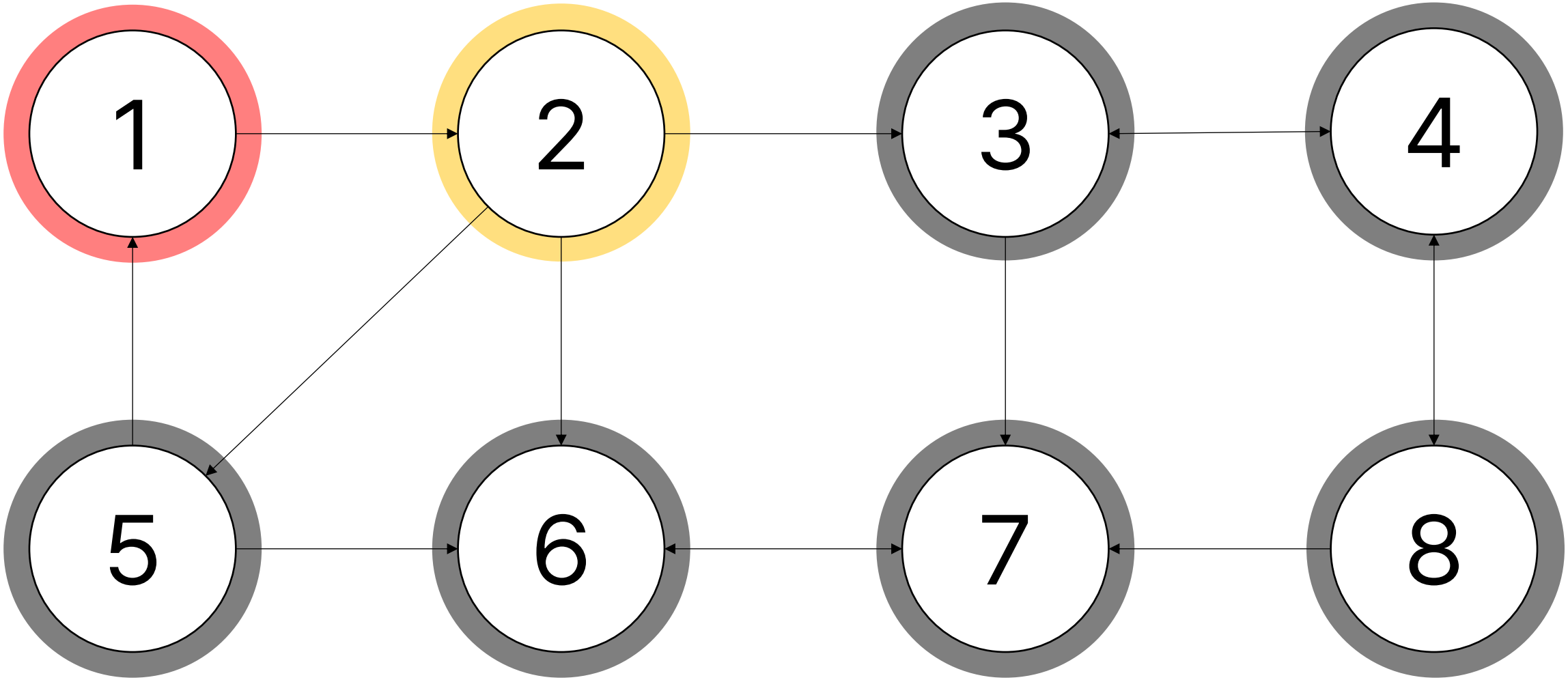
Kosaraju

6	7	8	4	3			



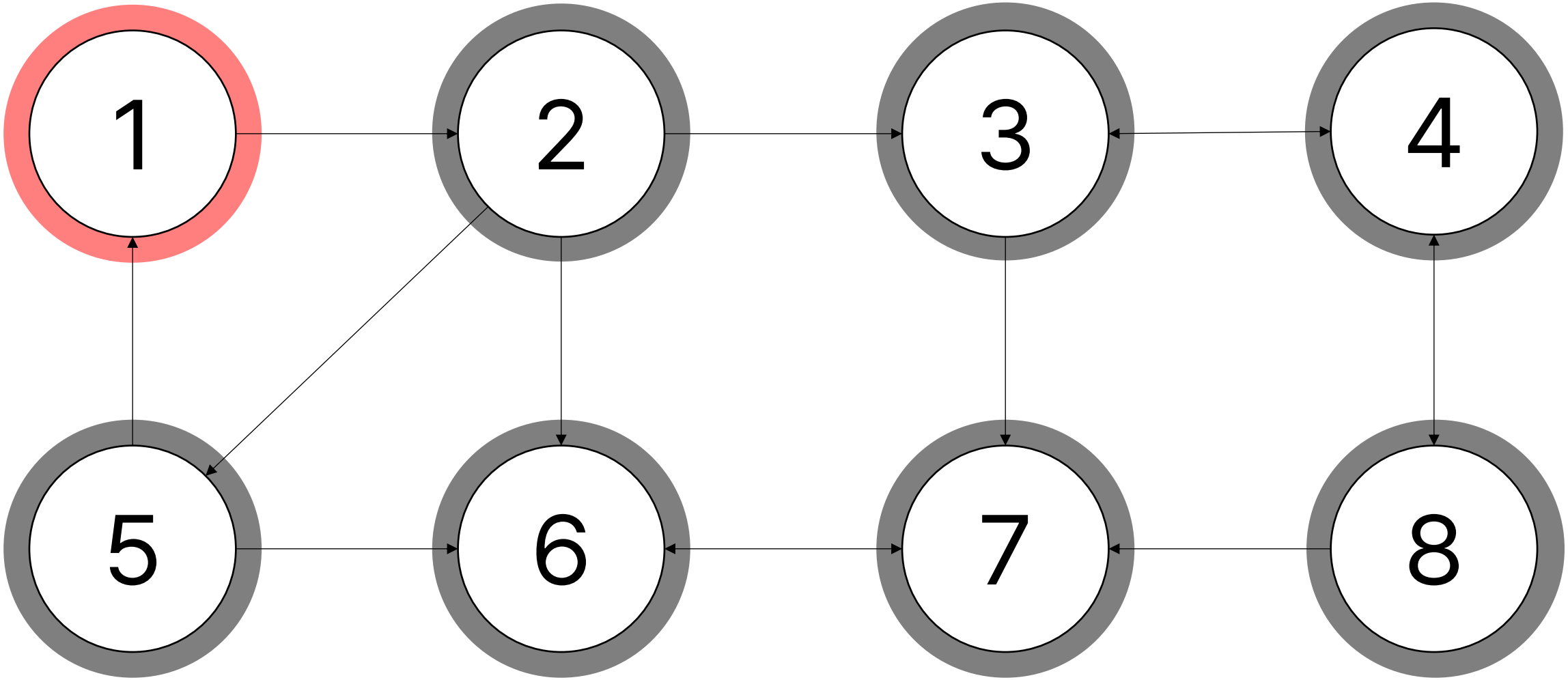
Kosaraju

6	7	8	4	3	5		



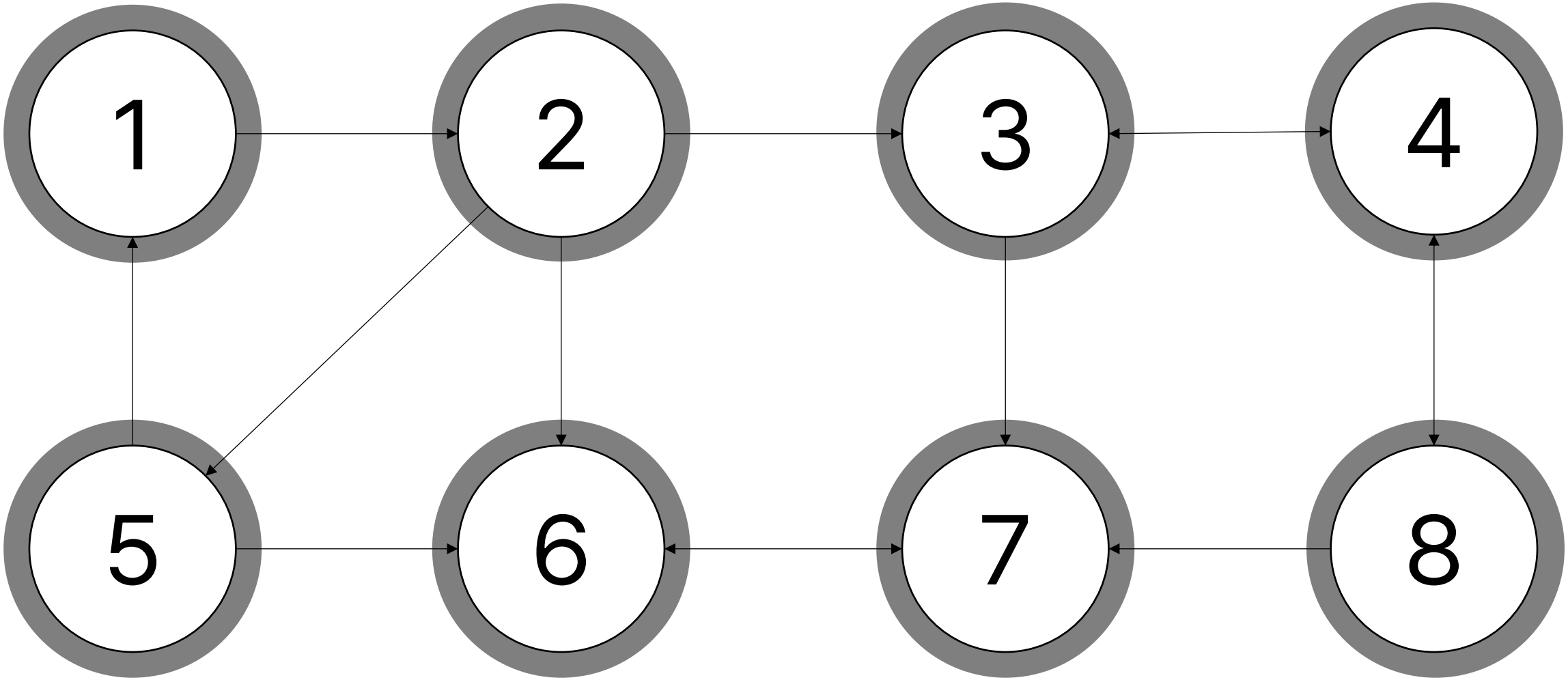
Kosaraju

6	7	8	4	3	5	2	



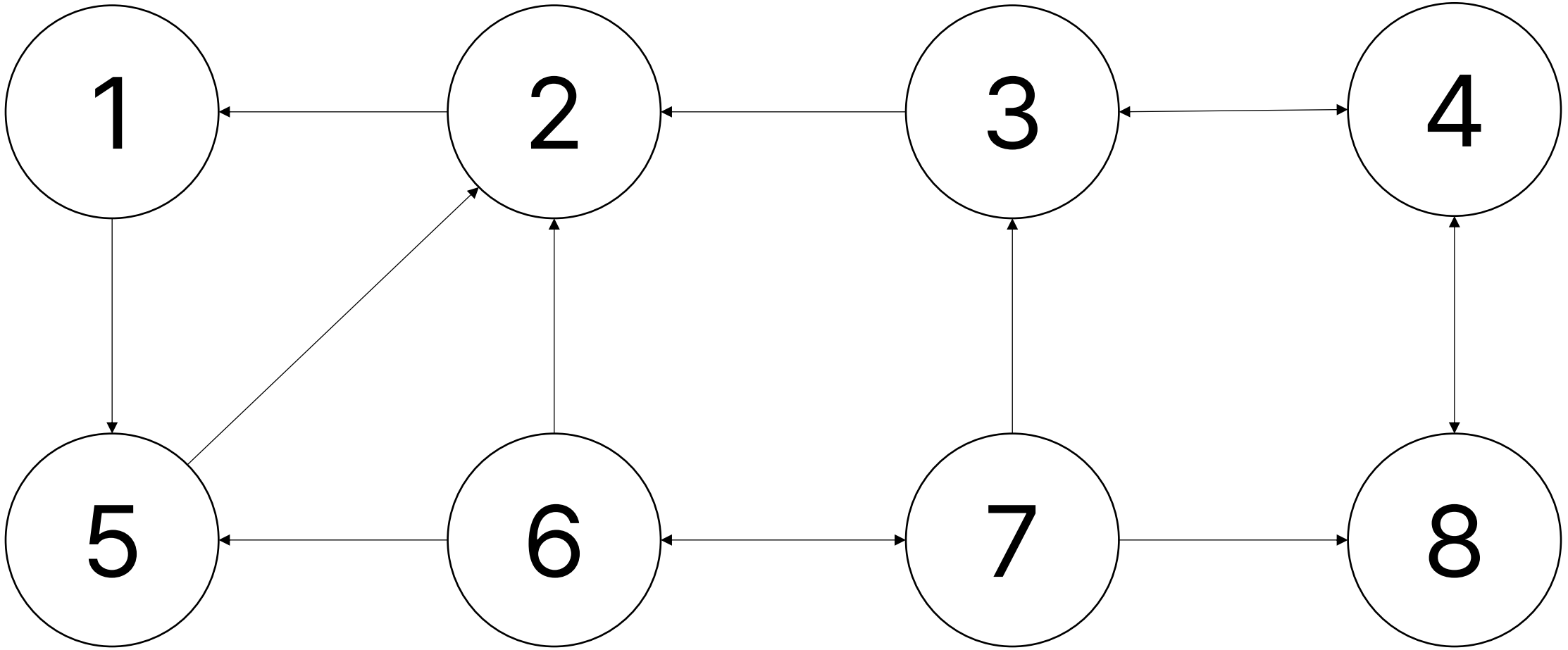
Kosaraju

6	7	8	4	3	5	2	1



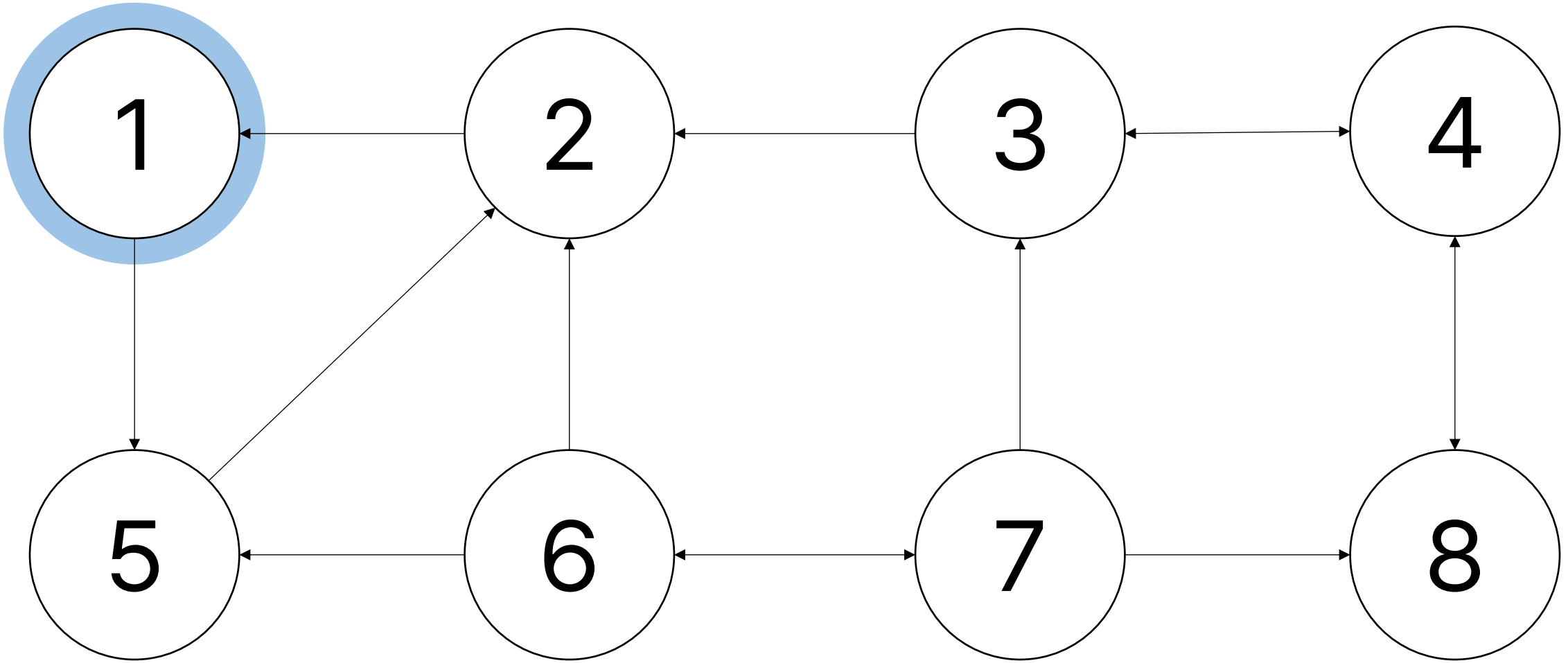
Kosaraju

6	7	8	4	3	5	2	1



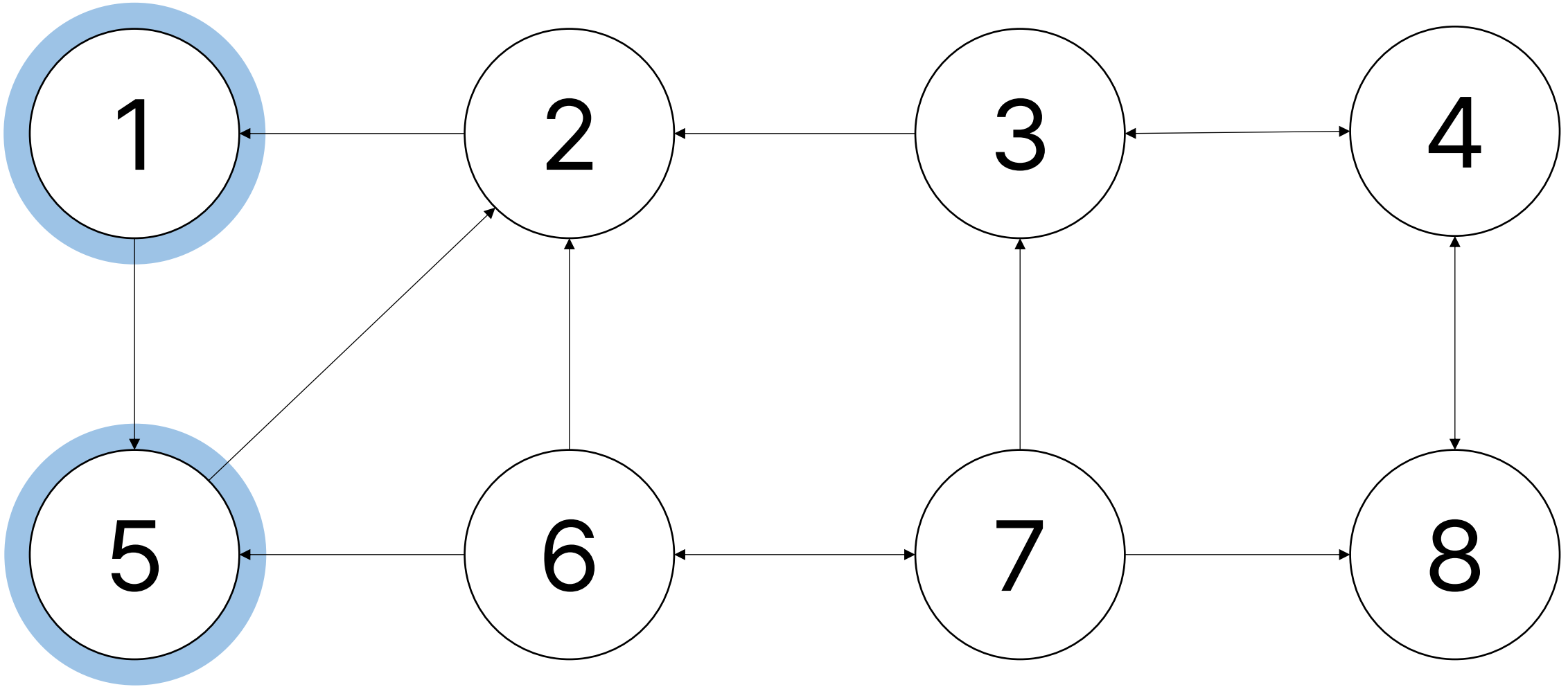
Kosaraju

6	7	8	4	3	5	2	



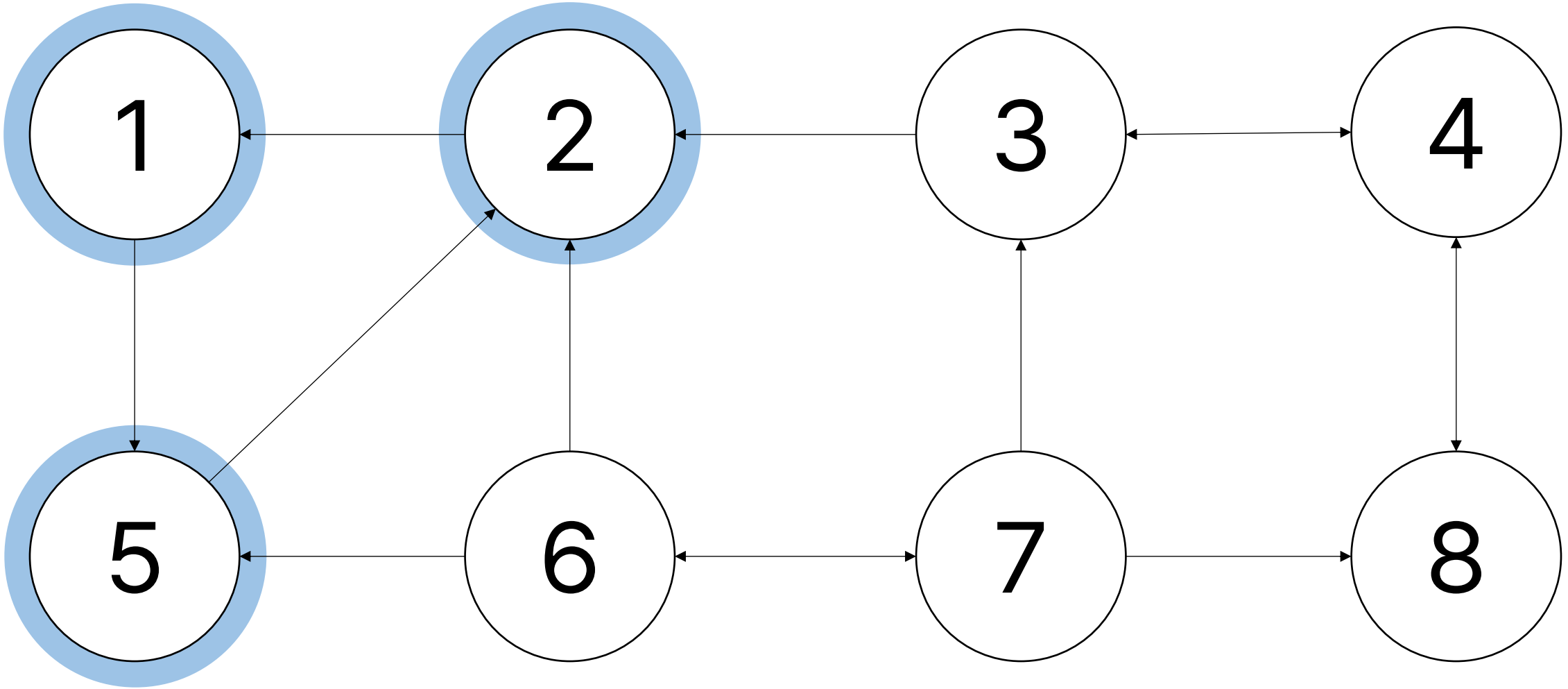
Kosaraju

6	7	8	4	3	5	2	



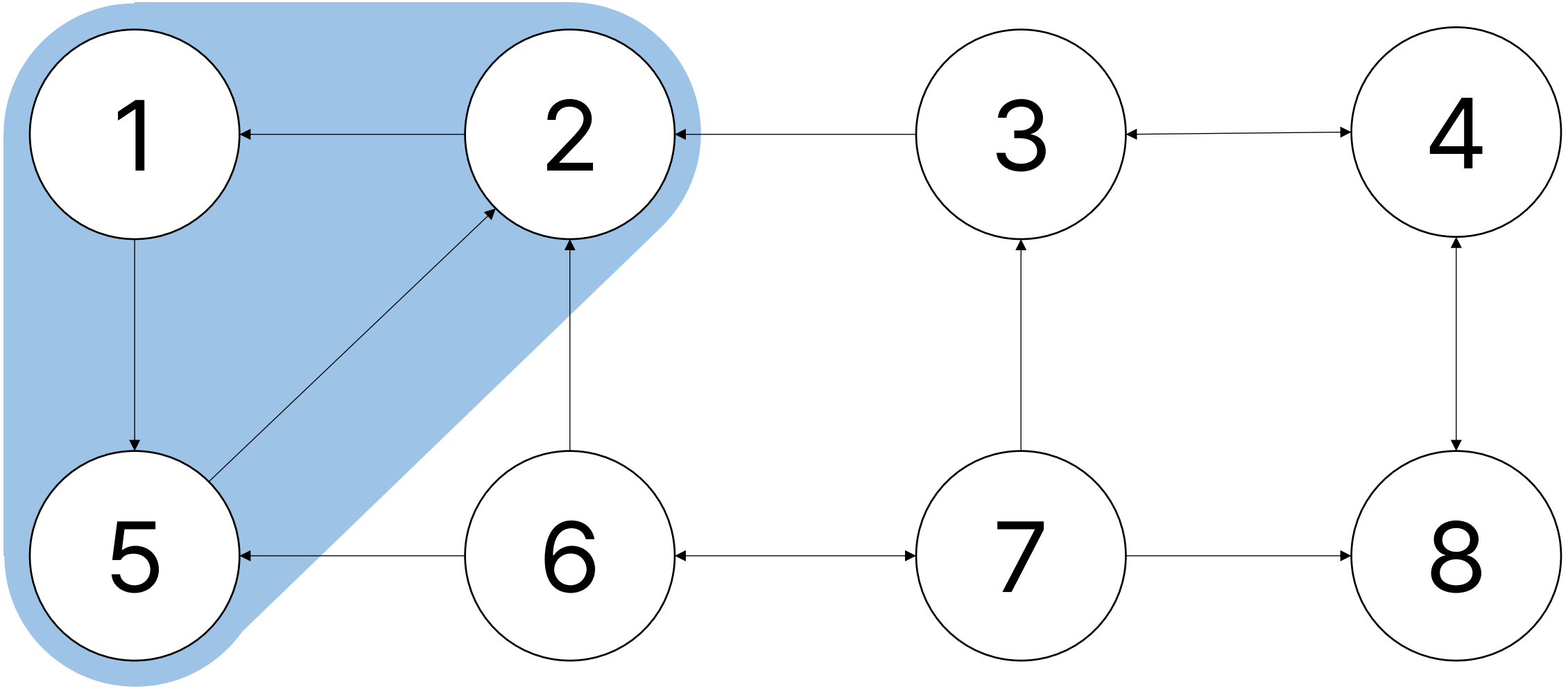
Kosaraju

6	7	8	4	3	5	2	



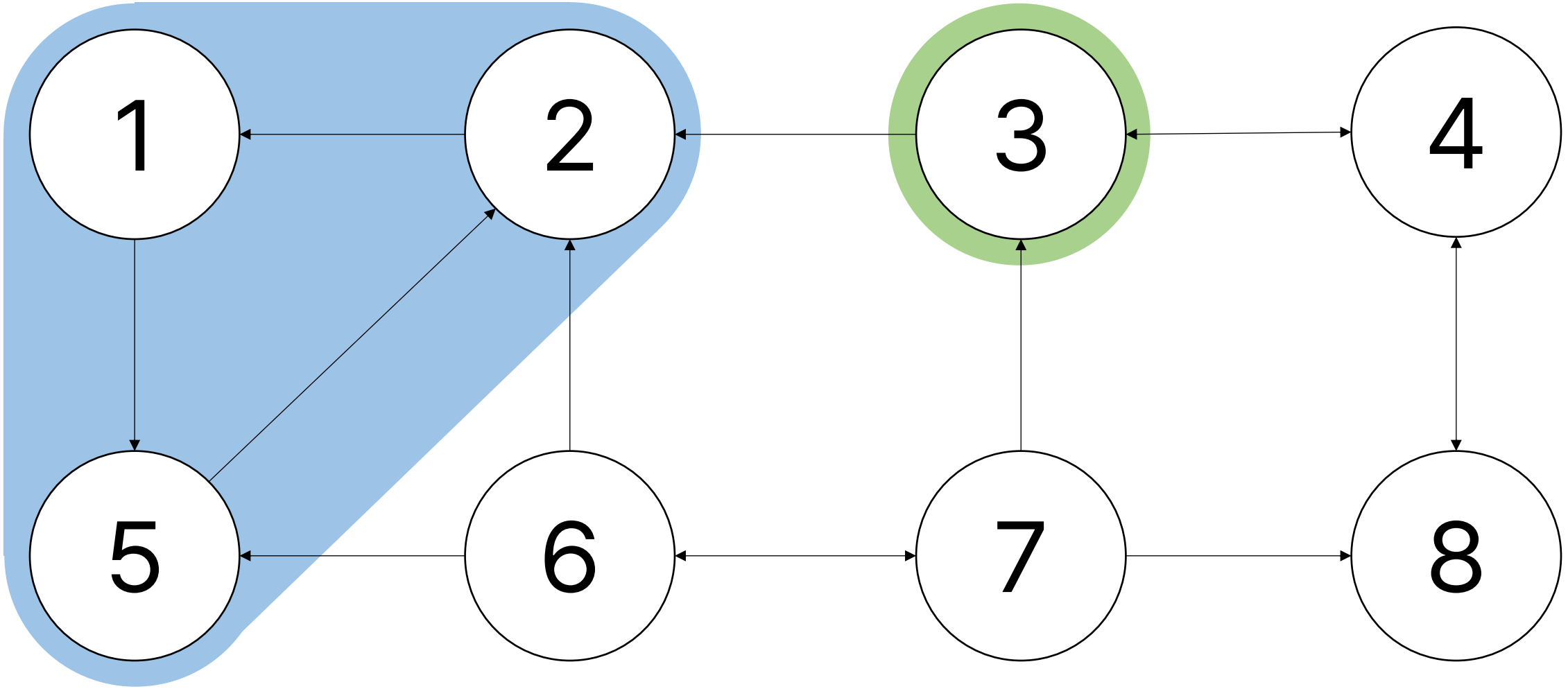
Kosaraju

6	7	8	4	3	~~5~~	~~2~~	



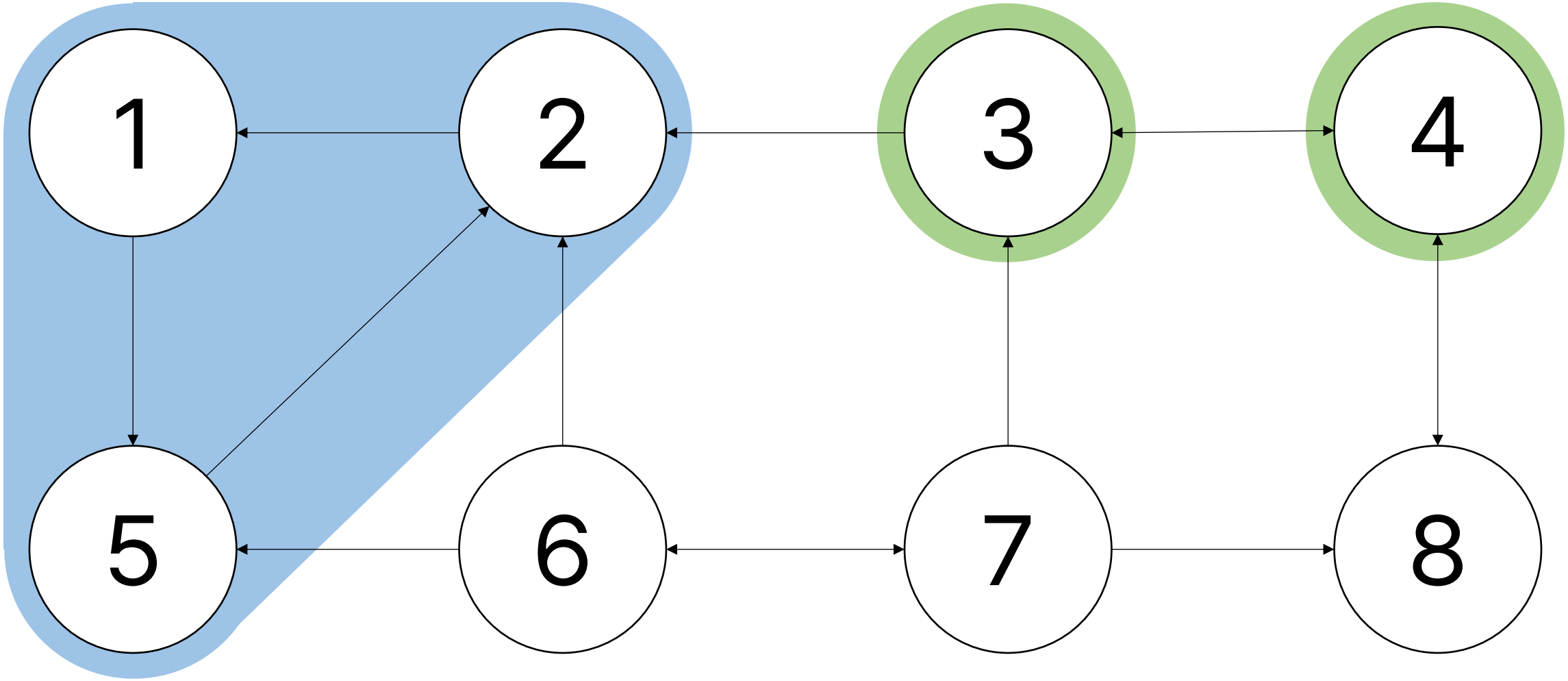
Kosaraju

6	7	8	4				



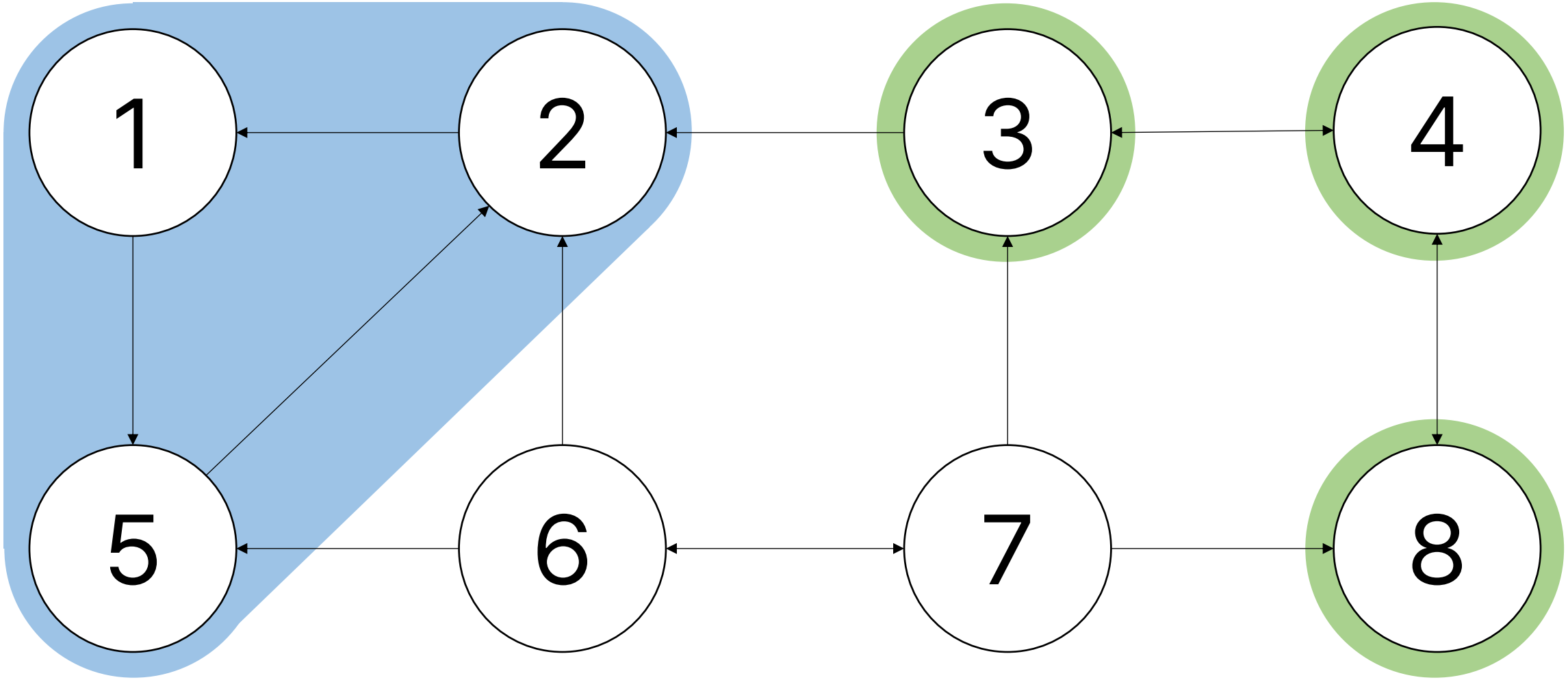
Kosaraju

6	7	8	4				



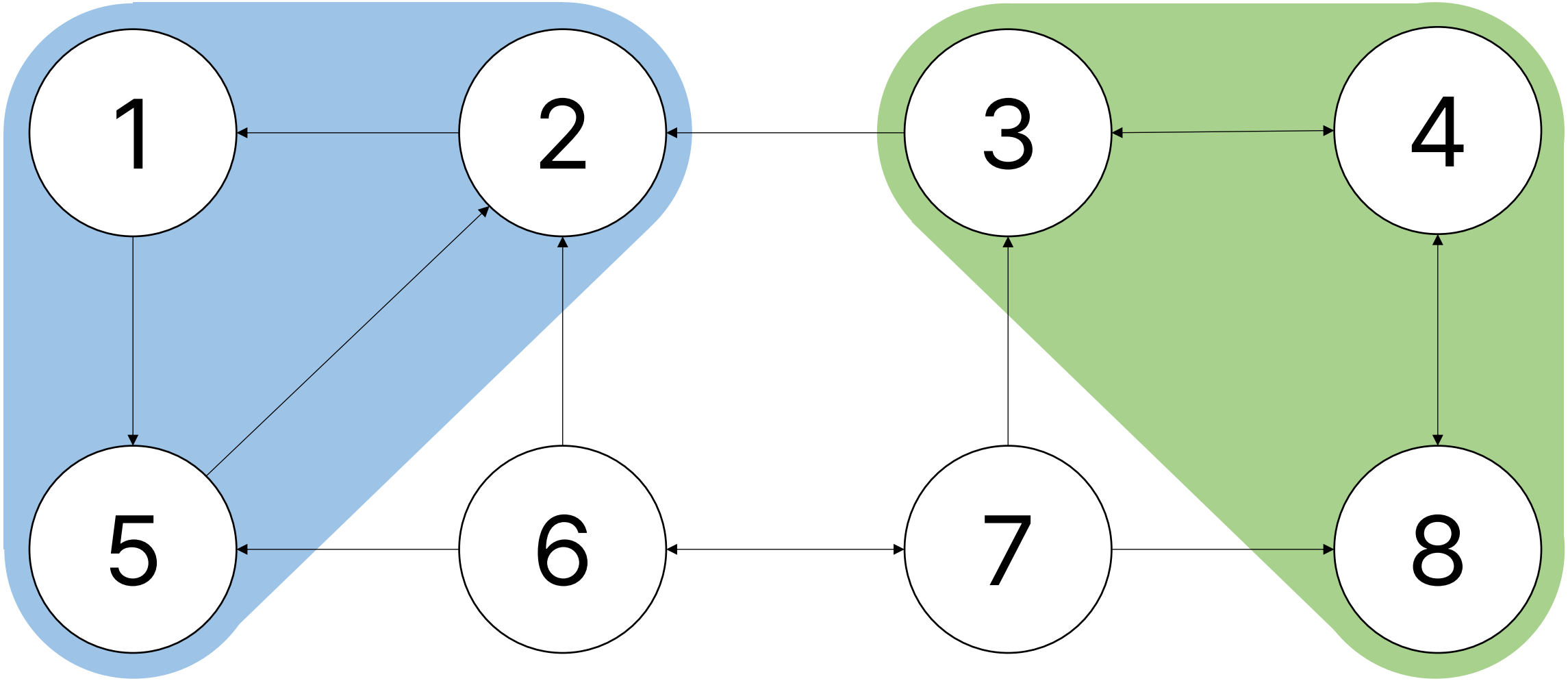
Kosaraju

6	7	8	4				

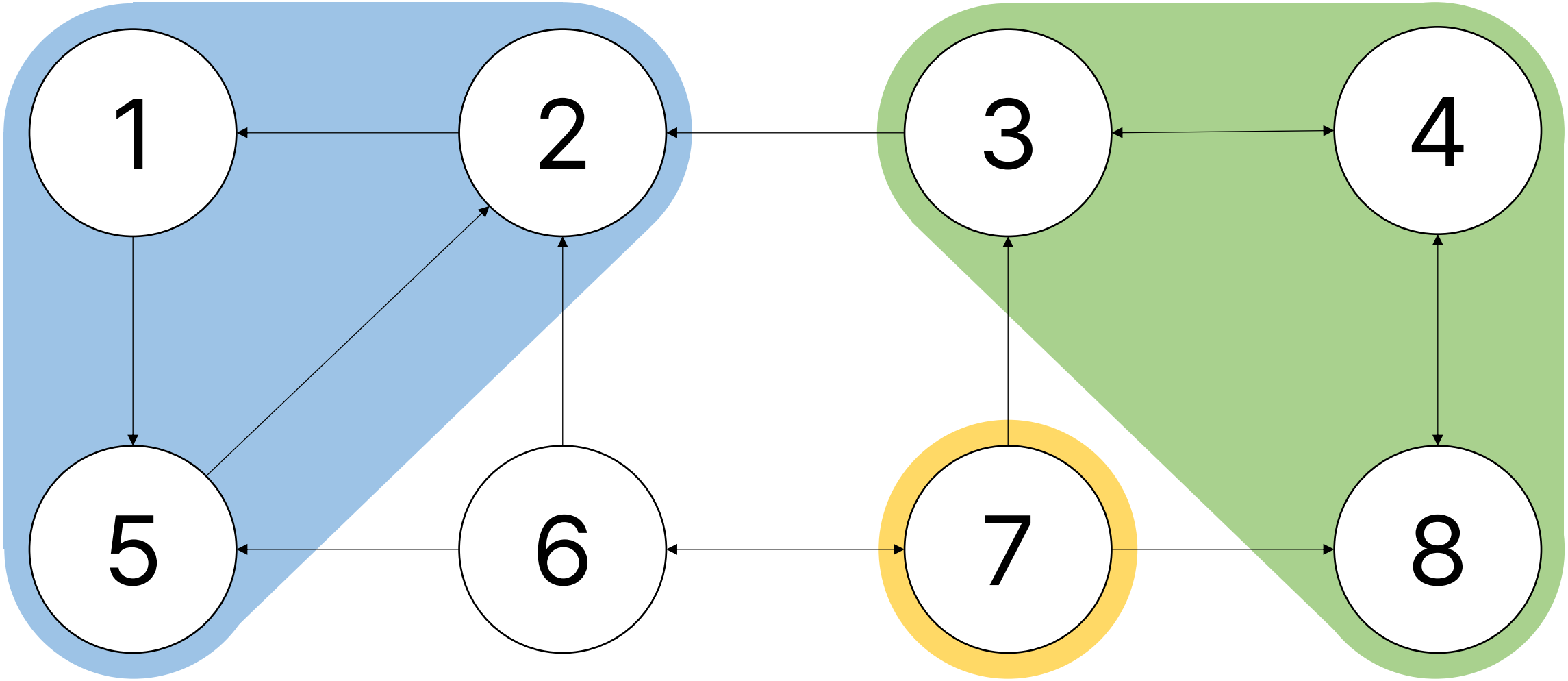


Kosaraju

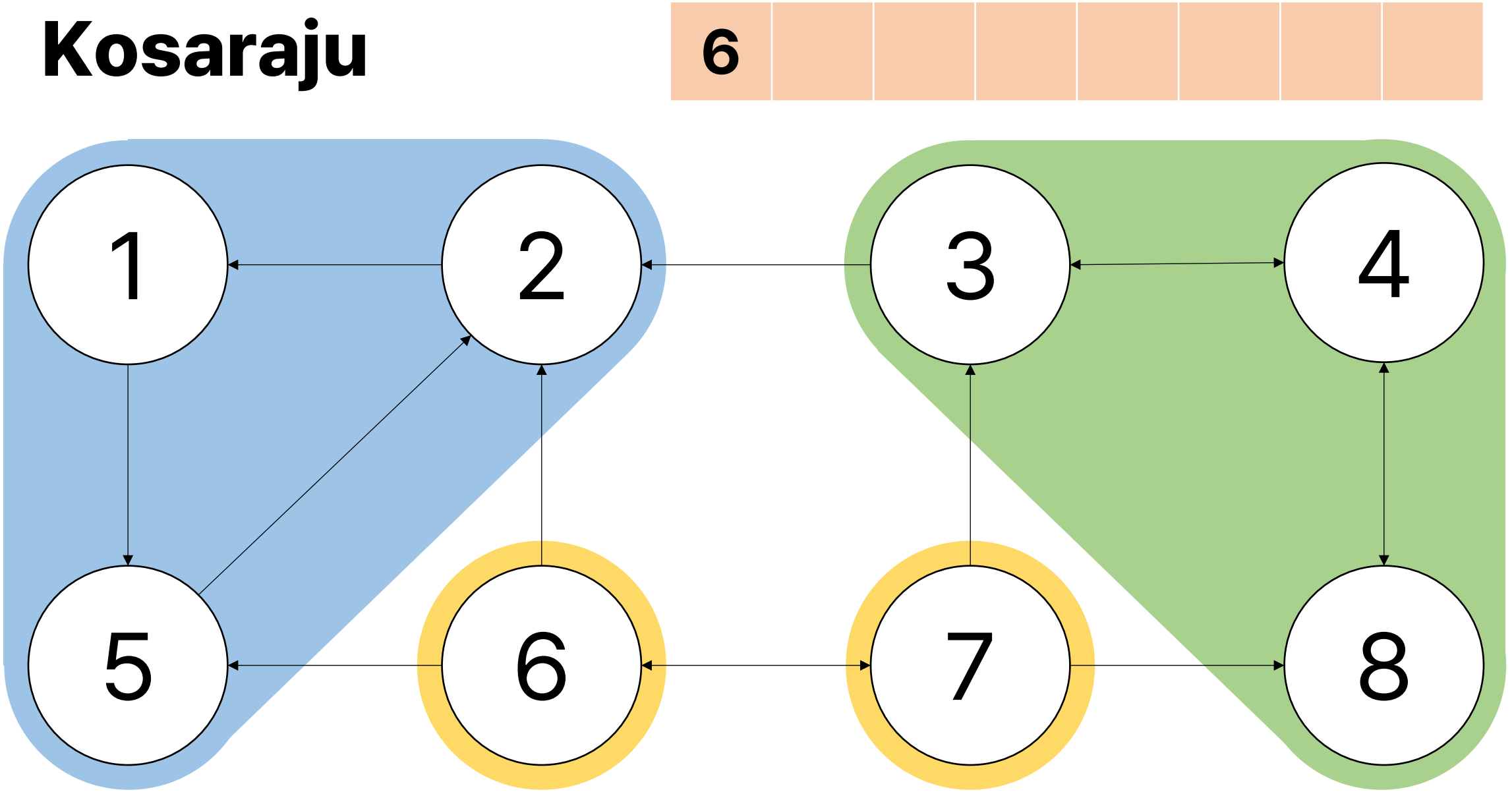
6	7	~~8~~	~~4~~				



Kosaraju

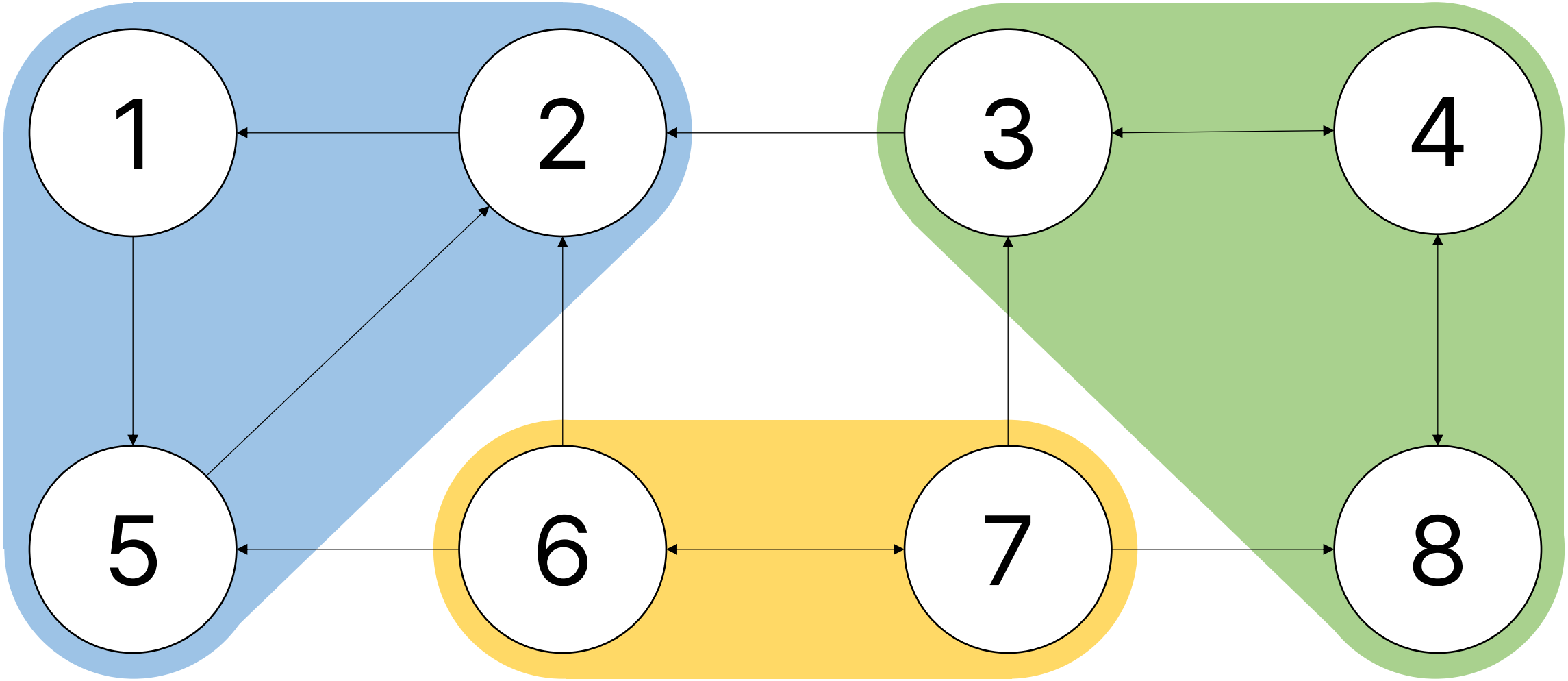


Kosaraju



Kosaraju

~~6~~							



Kosaraju

- 간선을 추가할 때 역방향 간선도 같이 추가

```
vector<int> edges[MAX_NODE];
vector<int> reverse_edges[MAX_NODE];

int s, e;

while(edges_cnt--){
    cin >> s >> e;
    edges[s].push_back(e);
    reverse_edges[e].push_back(s);
}
```

Kosaraju

- DFS가 끝나는 순서대로 스택에 기록

```
stack<int> st;

void dfs(int cur) {
    visited[cur] = true;
    for (auto next : edges[cur]) {
        if (visited[next])
            continue;
        dfs(next);
    }

    st.push(cur);
}
```

Kosaraju

- DFS가 끝난 순서대로 역방향 그래프에서 다시 DFS를 수행

```
int scc_num;
int scc[MAX_NODE];

void get_scc() {
    while (!st.empty()) {
        int cur = st.top();
        st.pop();
        if (scc[cur])
            continue;
        scc_num++;
        reverse_dfs(cur);
    }
}
```

Kosaraju

- 역방향 DFS를 수행하며 방문하는 노드들을 같은 SCC로 묶는다

```
int scc_num;
int scc[MAX_NODE];

void reverse_dfs(int cur) {
    scc[cur] = scc_num;
    for (auto next : reverse_edges[cur]) {
        if (scc[next])
            continue;
        reverse_dfs(next);
    }
}
```

Tarjan

- 2가지를 이용해 SCC를 구한다
- num: 몇 번째로 방문한 정점인지
- low: 해당 노드를 루트으로 하는 서브 트리에서 간선을 하나만 거쳐서 갈 수 있는 정점 중 num이 제일 작은 값

- **타잔 알고리즘을 통해 나온 SCC 번호는 위상정렬의 역순**
- SCC 알고리즘 이후 따로 위상정렬을 해줄 필요가 없다

Conjuctive Normal Form

- CNF
- $f = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$

  $\wedge$: AND

  $\vee$: OR

  $\neg$: NOT
- 각 변수(리터럴)들을 OR로 묶은 절, 그리고 각 절을 AND로 묶은 형태
- 절에 들어있는 변수의 최대 개수가 K개일 때, K-CNF식이라고 부른다(2-CNF)

Satisfiability Problem

- CNF 식이 주어질 때, 해당 CNF식이 참이 되는 변수 값이 존재하는지 찾는 문제

$$f = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$$

- $x_1 = 0, x_2 = 1, x_3 = 1$인 경우에 f가 참이 됨
- $x_1 = 1, x_2 = 0, x_3 = 1$인 경우도 f가 참이 됨

Satisfiability Problem

- K-CNF식에 대해 풀 때, K-SAT 문제라고 한다
- K가 3 이상인 K-SAT 문제에 대하여, 3-SAT 문제로 변환할 수 있다.
- 따라서 해결해야할 문제는 2-SAT, 3-SAT 두가지이다

- 2-SAT의 경우 다항시간에 해결이 가능하다
- 3-SAT의 경우 NP문제로 해결이 불가능하다

Satisfiability Problem

- f가 참이 되려면 CNF식의 모든 절이 참이어야 한다
- 각 절이 참이 되려면 적어도 하나의 변수가 참이면 된다

- $(x_1 \vee x_2)$에서 x_1 과 x_2 중 하나 이상이 참이면 된다

  -> x_1이 거짓인 경우 x_2는 반드시 참이여야 한다

  -> x_2이 거짓인 경우 x_1는 반드시 참이여야 한다

Satisfiability Problem

- 각 변수의 값을 나타낼 노드를 만든다.
- 각 변수가 true 또는 false를 가지므로, 각 변수별로 노드를 2개씩 만든다

  각 노드가 의미하는 것은 해당 변수가 어떤 값을 나타내는지 의미
- 각 절의 조건에 맞춰 간선을 추가

Satisfiability Problem

- $(x_1 \vee x_2)$에서 x_1 과 x_2 중 하나 이상이 참이면 된다

  -> x_1이 거짓인 경우 x_2는 반드시 참이여야 한다

  -> x_2이 거짓인 경우 x_1는 반드시 참이여야 한다

- $x_1(false) \rightarrow x_2(true)$ 간선을 추가
- $x_2(false) \rightarrow x_1(true)$ 간선을 추가

Satisfiability Problem

- 노드에서 나가는 간선이 존재하는 경우, 해당 노드가 나타내는 값이 다른 변수의 값에 영향을 미친다는 것을 의미한다
- 노드에서 들어오는 간선이 존재하는 경우, 해당 노드가 나타내는 값이 다른 변수의 특정한 값에 영향을 받는 것을 의미한다

Satisfiability Problem

- $x_k(true) \rightarrow x_k(false)$ 경로와 $x_k(false) \rightarrow x_k(true)$ 경로가 존재하는 경우

- x_k은 $true$ 일 때 $false$ 이어야 하며, $false$ 일 때 $true$ 이어야 한다
- 각 변수는 값을 하나만 지닐 수 있으므로, 모순이 발생한다

Satisfiability Problem

- $x_k(true) \rightarrow x_k(false)$ 경로와 $x_k(false) \rightarrow x_k(true)$ 경로가 존재하는 경우, $x_k(true)$ 노드와 $x_k(false)$ 노드가 같은 SCC인 경우, 해당 CNF식을 참으로 만드는 값이 존재하지 않는다

- 따라서 모든 변수가 나타내는 노드들에 대하여, 같은 변수를 지칭하는 노드들끼리 SCC를 형성하지 않는 경우, CNF식을 참으로 만드는 값이 존재한다

Satisfiability Problem

- $x_k(true) \rightarrow x_k(false)$ 경로만 존재하며, $x_k(false) \rightarrow x_k(true)$ 경로가 존재하지 않는 경우

- x_k이 $true$ 라면, x_k이 $false$ 이어야 한다
  모순이 발생하므로, x_k이 $true$ 값을 가질 수 없다
- x_k이 $false$라면, x_k이 $true$ 이지 않아도 된다
  모순이 발생하지 않으므로, x_k이 $false$값을 가질 수 있다

Satisfiability Problem

- $x_k(true) \rightarrow x_k(false)$, $x_k(false) \rightarrow x_k(true)$ 경로가 모두 존재하지 않는 경우 각 노드의 값 사이에 어떠한 의존성도 존재하지 않는다.
- x_k는 $true$와 $false$ 모두 가능하다

가능한 한가지 경우 구하기

- CNF식을 참으로 만드는 변수의 값이 존재하는 경우, 여러 경우 중 한가지 경우를 찾기 위해서는, 의존성이 있는 경우만 생각하면 된다
- 의존성이 있는 경우 불가능한 값에서 정답으로 향하는 경로가 존재한다
- 경로가 존재하고, 두 노드가 같은 SCC가 아니므로 위상 정렬 시 해답의 SCC가 항상 불가능한 값의 SCC보다 뒤에 나오게 된다

가능한 한가지 경우 구하기

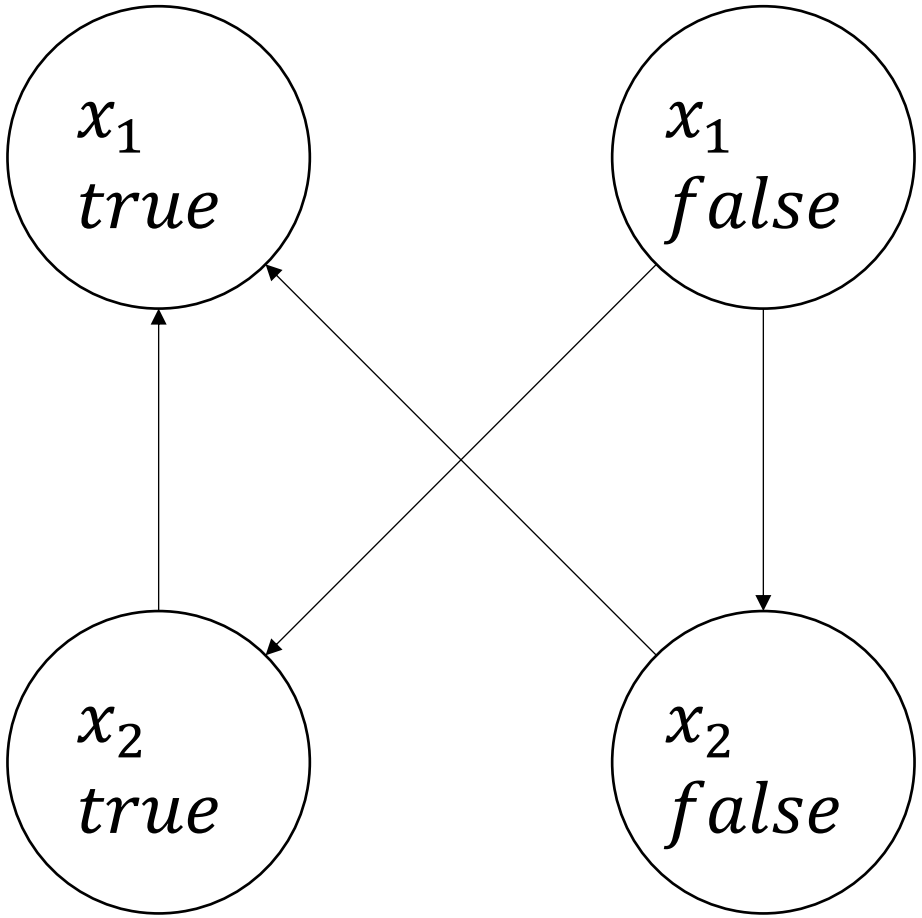
- 타잔 알고리즘을 통해 SCC를 결정한 경우, 위상 정렬의 역순으로 SCC를 부여하게 된다
- 위상 정렬의 역순이므로, 두 노드 중에 먼저 SCC로 묶인 노드가 나타내는 값이 정답이 되게 된다

Satisfiability Problem

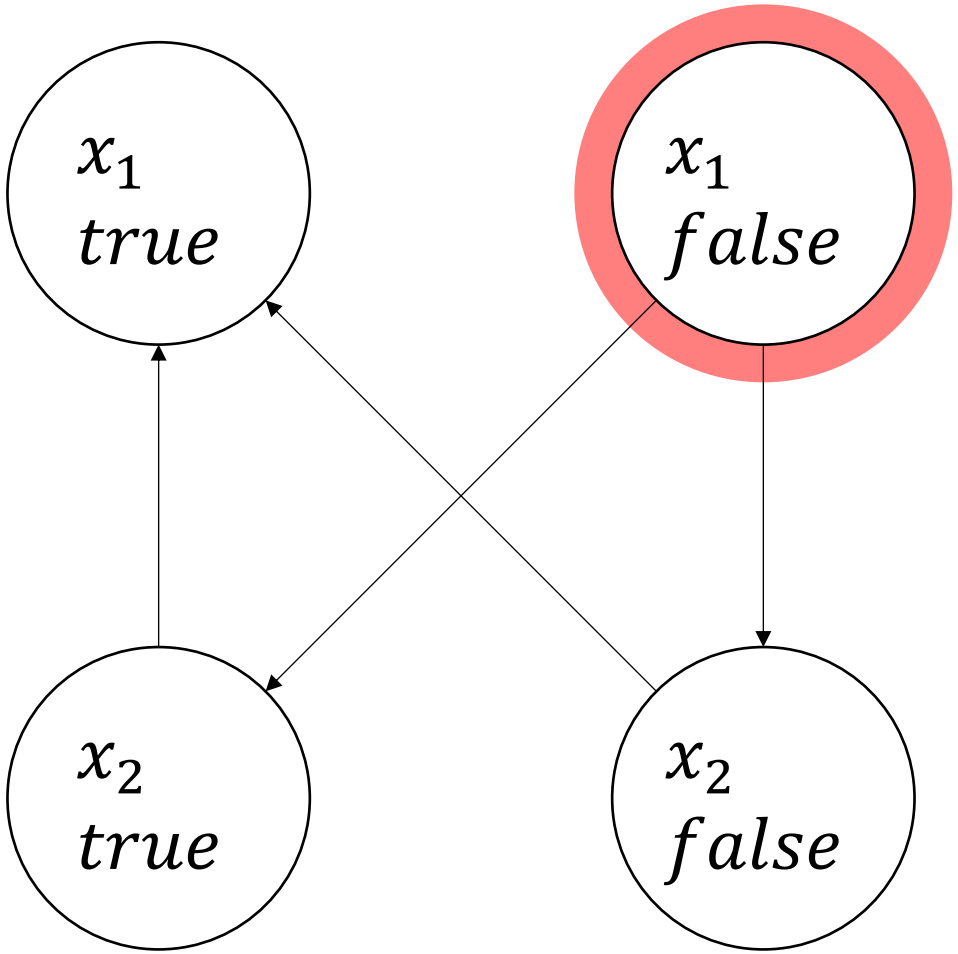
$$f = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_1)$$

- $x_1(false) \rightarrow x_2(true)$
- $x_2(false) \rightarrow x_1(true)$
- $x_2(true) \rightarrow x_1(true)$
- $x_1(false) \rightarrow x_2(false)$

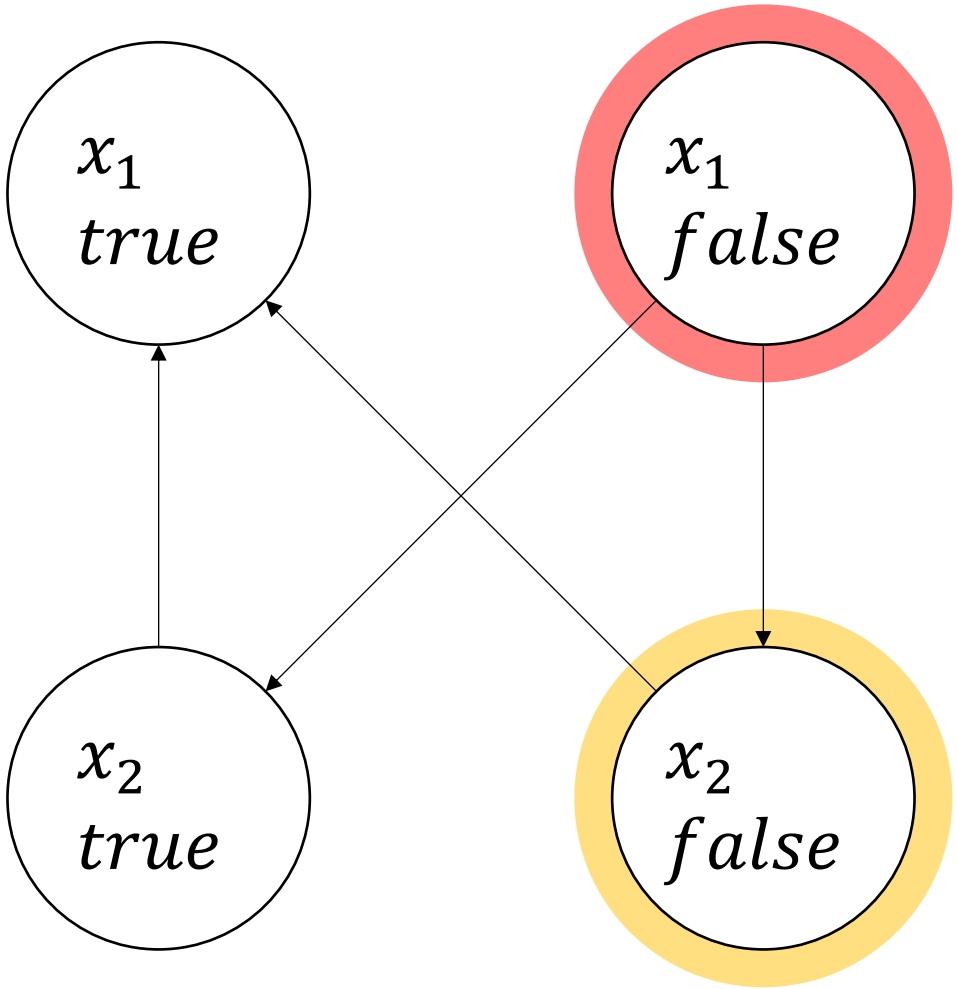
Satisfiability Problem



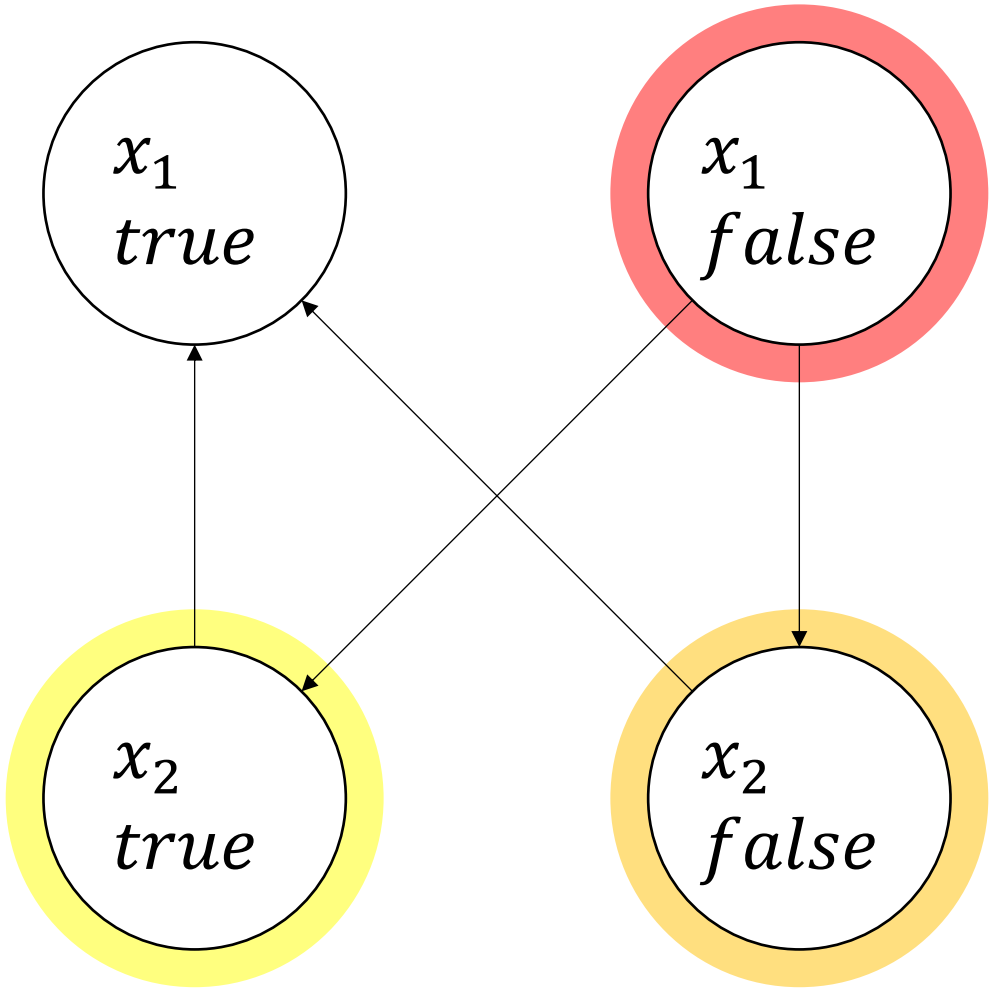
Satisfiability Problem



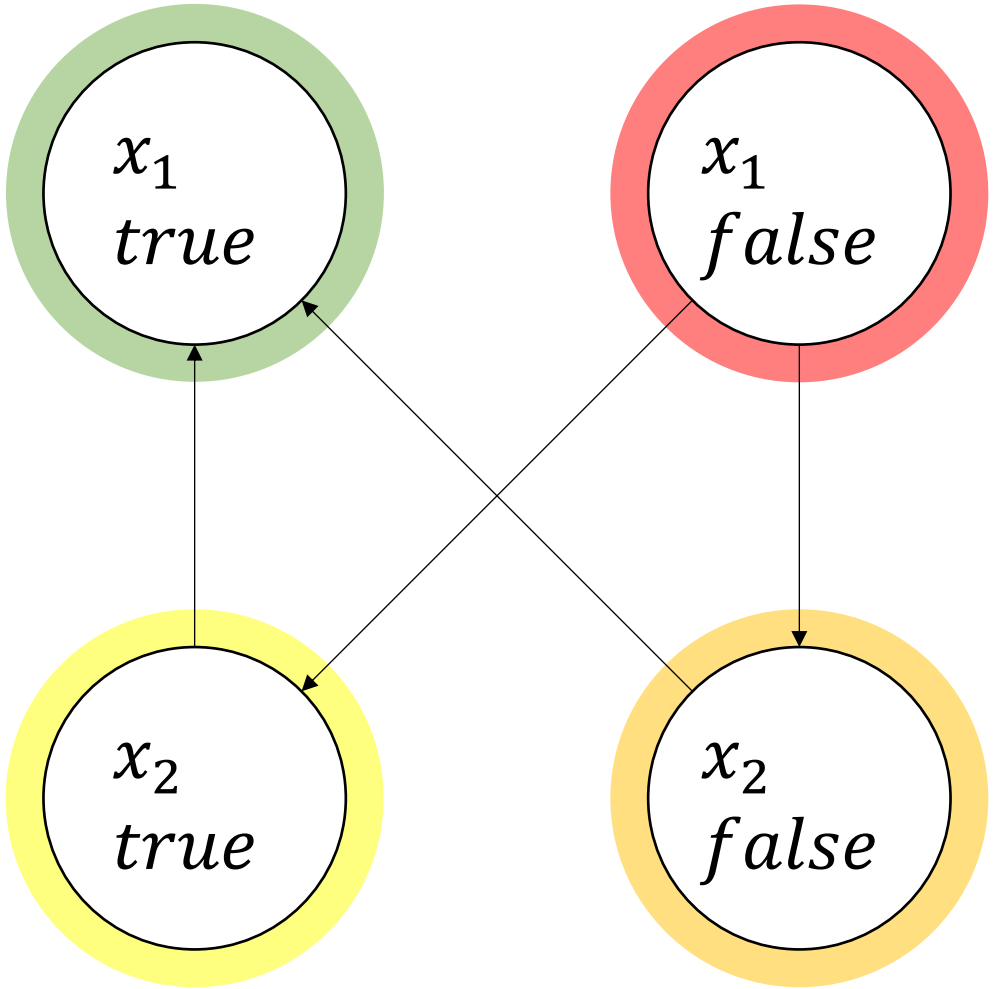
Satisfiability Problem



Satisfiability Problem



Satisfiability Problem



Satisfiability Problem

$$f = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_1)$$

- CNF식 f 를 참으로 만드는 변수의 값이 존재한다
- 가능한 변수의 값 중 한가지: $x_1(true), x_2(true)$

SCC BOJ 2150

- SCC를 구한 후 출력하는 문제
- 같은 SCC에 속한 정점들은 오름차순으로 출력
- 여러개의 SCC에 대해서는 그 안에 속해있는 가장 작은 정점의 정점 번호 순으로 출력
- SCC의 최대 개수는 정점의 개수와 동일, V개의 SCC 벡터 생성
- 같은 SCC끼리 정점을 벡터에 넣은 후 정렬, 정점 번호가 작은 정점의 SCC부터 출력
- $V \leq 10{,}000$

경찰서 BOJ 1506

- 도시 i에 세운 경찰서가 도시 j를 통제할 수 있으려면, i에서 j로 갔다가, 다시 돌아오는 경로가 존재해야 한다. (i와 j는 SCC가 성립함)
- 모든 도시를 통제하기 위해서는 하나의 SCC에 하나 이상의 경찰서 필요
- 최소 비용을 구하기 위해서는 하나의 SCC안에서 하나의 경찰서만 필요
- 모든 SCC의 최소 비용을 합
- $N \leq 100$

경찰서 BOJ 1506

- 도시 i에 세운 경찰서가 도시 j를 통제할 수 있으려면, i에서 j로 갔다가, 다시 돌아오는 경로가 존재해야 한다. (i와 j는 SCC가 성립함)
- 모든 도시를 통제하기 위해서는 하나의 SCC에 하나 이상의 경찰서 필요
- 최소 비용을 구하기 위해서는 하나의 SCC안에서 하나의 경찰서만 필요
- 모든 SCC의 최소 비용을 합
- $N \leq 100$

경찰서 BOJ 1506

```
memset(mn, INF, sizeof(mn));
get_scc();

for (int i = 1; i <= N; i++) {
    mn[scc[i]] = min(mn[scc[i]], cost[i]);
}

for(int i = 1; i <= scc_cnt; i++){
    ans += mn[i];
}
```

도미노 BOJ 4196

- 도미노 블록의 배치가 주어졌을 때, 모든 블록을 넘어뜨리기 위해 손으로 넘어뜨려야 하는 블록 개수의 최솟값
- 하나의 SCC는 도미노를 하나만 쓰러트려도 SCC에 포함된 도미노는 모두 쓰러진다.
- SCC당 하나의 도미노만 쓰러트리면 모두 쓰러지지만, SCC의 개수가 최소는 아니다.
- $N \leq 100{,}000$

도미노 BOJ 4196

- 하나의 도미노가 쓰러지면 연결되어 있는 SCC는 모두 쓰러진다
- 모든 도미노(노드)는 하나의 그래프로 연결되어 있다는 보장이 없다.
- SCC를 하나의 노드로 생각 할 때, 다른 도미노가 쓰러트리지 않는 SCC(in degree가 0인 SCC)(노드)만 쓰러트리면 된다.

도미노 BOJ 4196

```
get_scc();
for (int i = 1; i <= N; i++) {
    for (auto next : edges[i]) {
        if (scc[i] != scc[next]) {
            in_degree[scc[next]]++;
        }
    }
}
```

여행 계획 세우기 BOJ 2152

- 시작 도시에서 도착 도시로 가면서 최대한 많은 도시를 방문
- 같은 도시를 재방문 가능, 재방문시에는 세지 않는다
- 시작 도시에서 도착 도시로 가는 경로가 없는 경우 0을 출력
- $N \leq 100{,}000$

여행 계획 세우기 BOJ 2152

- 같은 도시를 여러 번 방문이 가능하므로 같은 SCC에 있는 모든 도시를 방문해도 영향을 미치지 않는다
- SCC를 기준으로 노드를 형성했을 때, 기준으로 위상정렬하여 각 SCC를 방문하기 전 최대로 방문할 수 있는 도시 수를 기록
- 각 SCC에 있는 도시가 최대로 방문 할 수 있는 도시의 수는 현재 SCC에 오기 전 가능한 최대의 도시의 수 + 현재 SCC에 포함되어있는 도시의 수

ATM BOJ 4013

- 여행 계획 세우기와 동일한 유형, 하지만 도착 가능 지역이 여러 개
- 각 지역의 있는 ATM의 금액은 SCC 단위로 묶어서 관리
- 여행 계획 세우기와 마찬가지로 출발 지역이 포함된 SCC에서 시작하여 방문할 수 있는 모든 SCC에 최대 금액을 기록함
- 도착 가능 지역이 포함된 모든 SCC를 관찰하여 최대값 출력
- $N \leq 500{,}000$

점프 점프 2 BOJ 14249

- 출발점이 주어질 때, 방문 가능한 최대 개수의 돌을 출력하는 것
- 여행 계획 세우기와 동일한 유형
- 각 돌을 하나의 노드로, 좌우로 점프하는 것을 간선으로 추가
- 도착지가 모든 돌이 가능한 형태의 ATM 문제로 변화
- $N \leq 100{,}000$

2-SAT - 3 BOJ 11280

- 2-CNF 식을 true로 만드는 값이 존재하는지 찾는 문제
- 입력으로 주어진 2-CNF식을 이용해 간선을 추가한다
- SCC 알고리즘을 이용해 각 변수의 true와 false 노드가 같은 SCC에 존재하는지 확인한다
- $N \leq 10\,000$

2-SAT - 3 BOJ 11280

- 각 노드 번호의 뜻을 정확히 결정
- 예시1

  $1 \leq k \leq N$: 변수 k가 true

  $\mathrm{N}+1 \leq k \leq 2N$: 변수 k가 false

- 예시2

  1001010: k 값(k << 1)

  1001010: 0일 경우 true, 1일 경우 false

2-SAT - 3 BOJ 11280

```
int normalize(int x) { return x > 0 ? x : -x + N; }

int NOT(int x) { return x <= N ? x + N : x - N; }

while (M--) {
    cin >> a >> b;
    a = normalize(a);
    b = normalize(b);
    edges[NOT(a)].push_back(b);
    edges[NOT(b)].push_back(a);
}
```

2-SAT - 3 BOJ 11280

```
int normalize(int x) { return (abs(x) << 1) | (x < 0 ? 1 : 0); }

int NOT(int x) { return x ^ 1; }

while (M--) {
    cin >> a >> b;
    a = normalize(a);
    b = normalize(b);
    edges[NOT(a)].push_back(b);
    edges[NOT(b)].push_back(a);
}
```

2-SAT - 3 BOJ 11280

```
get_scc();

for (int i = 1; i <= N; i++) {
    if (scc[i] == scc[NOT(i)]) {
        cout << 0;
        return 0;
    }
}
cout << 1;
```

2-SAT - 4 BOJ 11281

- 2-SAT – 3 문제 + 가능한 해 중 한가지를 구하는 것
- Tarjan 알고리즘의 위상정렬을 이용
- $N \leq 10\,000$

2-SAT - 4 BOJ 11281

```
get_scc();

for (int i = 1; i <= N; i++) {
    if (scc[i] == scc[NOT(i)]) {
        cout << 0;
        return 0;
    }
    ans[i] = scc[i] < scc[NOT(i)] ? 1 : 0;
}

cout << 1;
for (int i = 1; i <= N; i++) {
    cout << ans[i] << ' ';
}
```

아이돌 BOJ 3648

- 의심을 받지 않고 조작이 가능하려면, 각 심사위원의 투표가 1개 또는 2개 이루어져야 한다
- 각 심사위원들의 투표 결과 중 하나가 반대되는 결과라면, 다른 하나는 반드시 성립해야 한다(2-SAT)
- $N \leq 1\,000$

아이돌 BOJ 3648

- 어떠한 경우에도 의심을 피하지 못하는 경우(해가 없는 경우) -> no

- 상근이는 반드시 다음라운드에 진출해야 한다
- 따라서 해가 존재하는 경우, 상근이가 진출하는 경우가 가능한지 판단해야함

  -> yes? no?

아이돌 BOJ 3648

- 진출하는 경우와 탈락하는 경우 사이에 어떠한 경로도 없는 경우

  진출하는 것을 답으로 결정하면 됨 -> yes

- 탈락하는 경우에서 진출하는 경우로 가는 경로만 존재하는 경우

  진출하는 경우만 만들 수 있음 -> yes

- 진출하는 경우에서 탈락하는 경우로 가는 경로만 존재하는 경우

  탈락하는 경우만 만들 수 있음 -> no

아이돌 BOJ 3648

- 진출하는 경우에서 탈락하는 경우로 가는 경로가 존재하거나 같은 SCC에 존재해 모순이 생기는 경우 no, 그 외의 경우는 모두 yes
- 2-SAT의 해답이 존재하는 경우, 진출하는 경우에서 DFS(BFS)를 시작한다
  탈락하는 경우를 방문하면 no, 방문하지 않는 경우 yes

- 좀 더 쉽게 하는 법은?

아이돌 BOJ 3648

- 어떤 변수(K)의 값을 T로 결정하고 싶은 경우

  K = ~T에서 K = T로 향하는 간선을 추가

- 간선을 추가하기 전, K의 값이 T 또는 ~T인 경우, T인 경우

  항상 값이 T로 결정되게 된다(~T에서 T로 가는 경로 존재)

- 간선을 추가하기 전, K의 값이 ~T인 경우

  ~T에서 T로 가는 경로와 T에서 ~T로 가는 경로가 존재해 SCC를 형성하게 된다

아이돌 BOJ 3648

- 탈락하는 경우에서 진출하는 경우로 향하는 간선을 추가한다

  (1번이 false인 경우에서 1번이 true인 경우로 향하는 간선을 추가)

- 상근이가 탈락해야만 하는 경우, 2-SAT의 해답이 존재하지 않게 된다
- 반드시 상근이가 합격하는 경우 중에, 의심을 받지 않는 경우에만

  2-SAT의 해답이 존재하게 된다

신촌지역 초중고등학생 프로그... BOJ 20942

- 각 자리의 나이를 결정하는 문제
- 나이는 8 이상 19 이하
- 일부 자리는 값이 고정되어 있음
- 음양비트론: x_i자리와 y_i자리를 t_i비트 연산(AND, OR)을 한 값이 z_i이어야 한다
- $N \leq 50\,000$

신촌지역 초중고등학생 프로그... BOJ 20942

- 나이를 그대로 사용하는 경우 각 자리가 가능한 경우는 8~19, 12가지가 가능하다
- 각 자리마다 12개의 변수를 만들고 간선을 추가하는 경우 너무 많은 자원이 필요

  각 자리마다 24(12×2)개의 노드가 필요하며, 132(11×12)개의 간선 필요
- 만약 나이가 2가지만 가능했다면 쉽게 풀 수 있다
- 여러 변수를 조합해서 각 자리의 나이를 나타내보자

신촌지역 초중고등학생 프로그... BOJ 20942

- 각 자리를 비트 단위로 쪼개보자
- 8이상 19이하이므로 5개의 비트로 모두 표현 가능 -> 각 자리 별 10개의 변수

  첫번째 자리의 비트가 1이다 0이다, 두번째... (총 10개)

신촌지역 초중고등학생 프로그... BOJ 20942

- 특정 자리의 나이가 고정되어 있는 경우 간선을 추가해 값을 고정
- $15 = 01111_{(2)}$

  1번째 비트가 1 -> 1번째 비트가 0

  2번째 비트가 0 -> 2번째 비트가 1

  3번째 비트가 0 -> 3번째 비트가 1

  4번째 비트가 0 -> 4번째 비트가 1

  5번째 비트가 0 -> 5번째 비트가 1

신촌지역 초중고등학생 프로그... BOJ 20942

- 음양비트론에 따른 비트 연산
- AND

  z_i가 1인 경우, 두 나이 모두 해당 자리의 비트가 1

  z_i가 0인 경우, 둘 중 하나의 변수는 해당 자리의 비트가 0
- OR

  z_i가 1인 경우, 둘 중 하나의 변수는 해당 자리의 비트가 1

  z_i가 0인 경우, 두 나이 모두 해당 자리의 비트가 0

신촌지역 초중고등학생 프로그... BOJ 20942

- 문제를 2-SAT에 맞게 변환, 남은 문제는 나이를 8~19로 고정하는 것

$01000_{(2)}$:8 $01100_{(2)}$: 12 $10000_{(2)}$: 16

$01001_{(2)}$: 9 $01101_{(2)}$: 13 $10001_{(2)}$: 17

$01010_{(2)}$: 10 $01110_{(2)}$: 14 $10010_{(2)}$: 18

$01011_{(2)}$: 11 $01111_{(2)}$: 15 $10011_{(2)}$: 19

신촌지역 초중고등학생 프로그... BOJ 20942

- 첫번째 자리의 비트가 0인 경우, 반드시 두번째 자리의 비트가 1
- 첫번째 자리의 비트가 1인 경우, 반드시 두번째, 세번째 자리의 비트가 0
- 두번째 자리의 비트가 1인 경우, 반드시 첫번째 자리의 비트가 1
- ...
- 세번째 자리의 비트가 0인 경우, 첫번쨔 자리가 0, 1 둘 다 가능

  (반드시 역이 성립하지는 않음)

신촌지역 초중고등학생 프로그... BOJ 20942

- 모든 간선을 추가한 뒤에는 2-SAT – 4와 동일