제9장 그래프

9.5 최단경로 알고리즘

- Dijkstra 알고리즘
- Bellman-Ford
- Floyd-Warshall 알고리즘

9.5.2 Bellman-Ford 알고리즘

- Dijkstra 알고리즘은 음수가중치를 가진 그래프에서 최단경로를 찾지 못함
- Bellman-Ford 알고리즘은 음수가중치 그래프에서도 문제 없이 최단경로를 찾을 수 있음
- 단, 입력그래프에 싸이클 상의 간선들의 가중치 합이 0보다 작은 음수싸이클(Negative Cycle)이 없어야
- 만약 어떤 경로에 음수싸이클이 존재한다면, 음수싸이클을 반복할수록 경로의 길이가 더 짧아지는 모순이 발생하기 때문

[핵심 아이디어] 입력그래프에 음수싸이클이 없으므로 출발점에서 각 정점까지 최단경로 상에 있는 간선의 수는 최대 N-1개 이다.

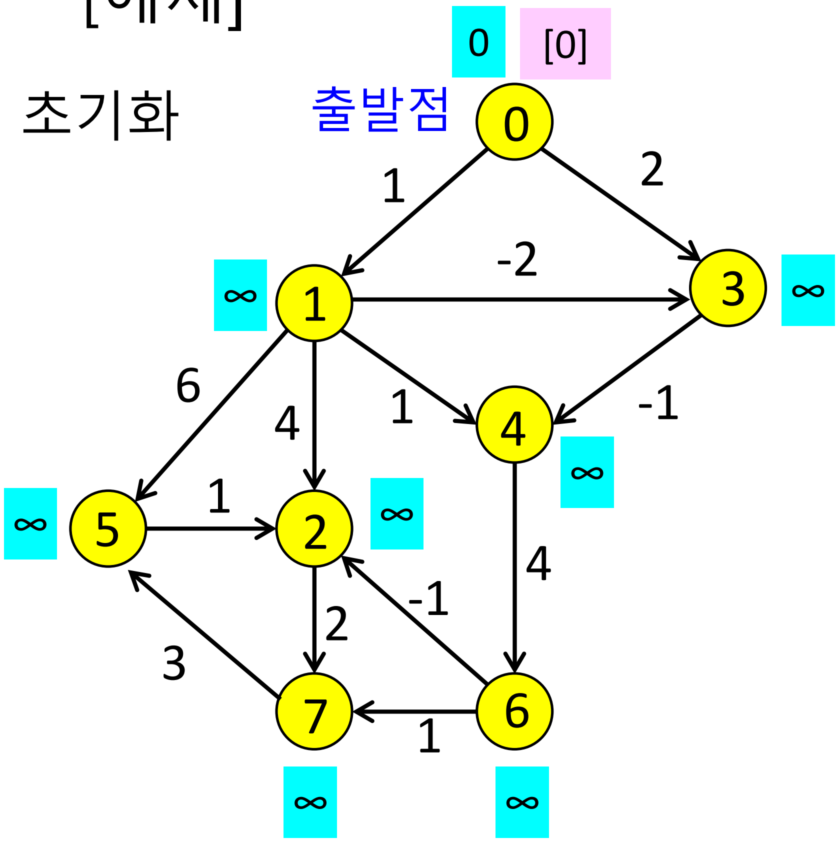
따라서 각 정점에 대해 간선완화를 N-1번 수행하면 더 이상 간선완화로 인한 갱신이 있을 수 없다.

```
[1] 배열 D를 ∞로 초기화한다. 단, D[s] = 0, s는 출발점
[2] for (k = 0; k < N-1; k++)
[3]     각 (i, j)에 대하여
[4]         if (D[j] > (D[i] + (i, j)의 가중치))
[5]             D[j] = D[i] + (i, j)의 가중치  // 간선완화
[6]             previous[j] =  i; ㄱ
```

- 배열 D를 ∞로 초기화, D[s] = 0
- Step [2]의 for-루프는 N-1번 수행되는데, 루프 내에서 각 간선의 양끝 정점에 대한 간선완화를 수행
- previous[j] = i는 출발점 s로부터 정점 j까지의 경로상에서 정점 i가 j의 직전 정점이라는 뜻

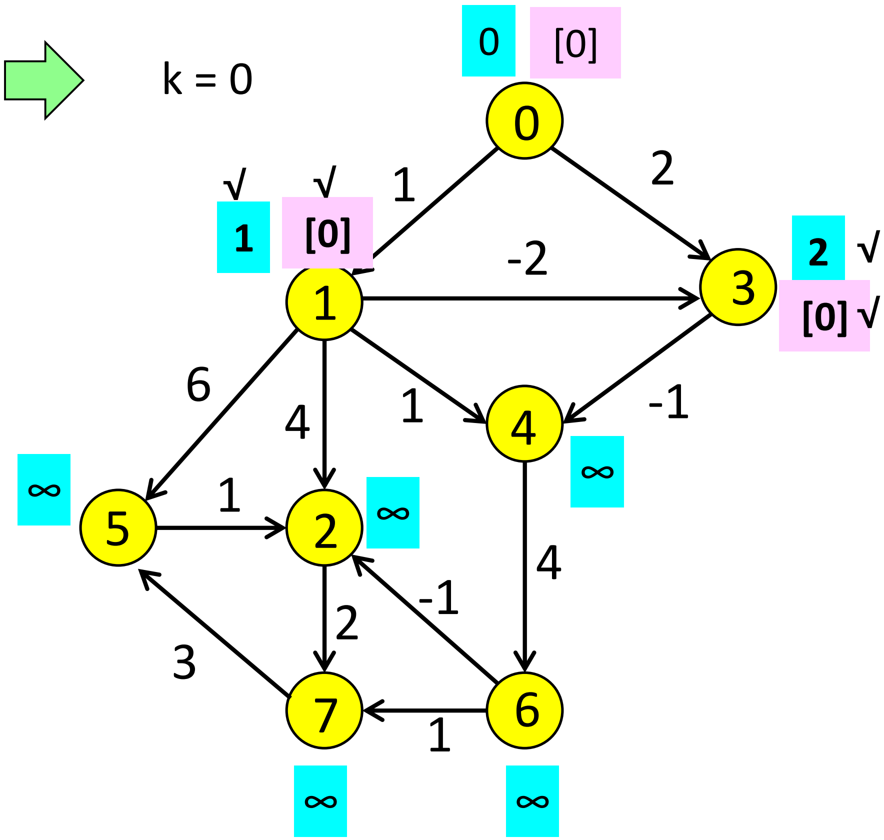
[예제]

초기화



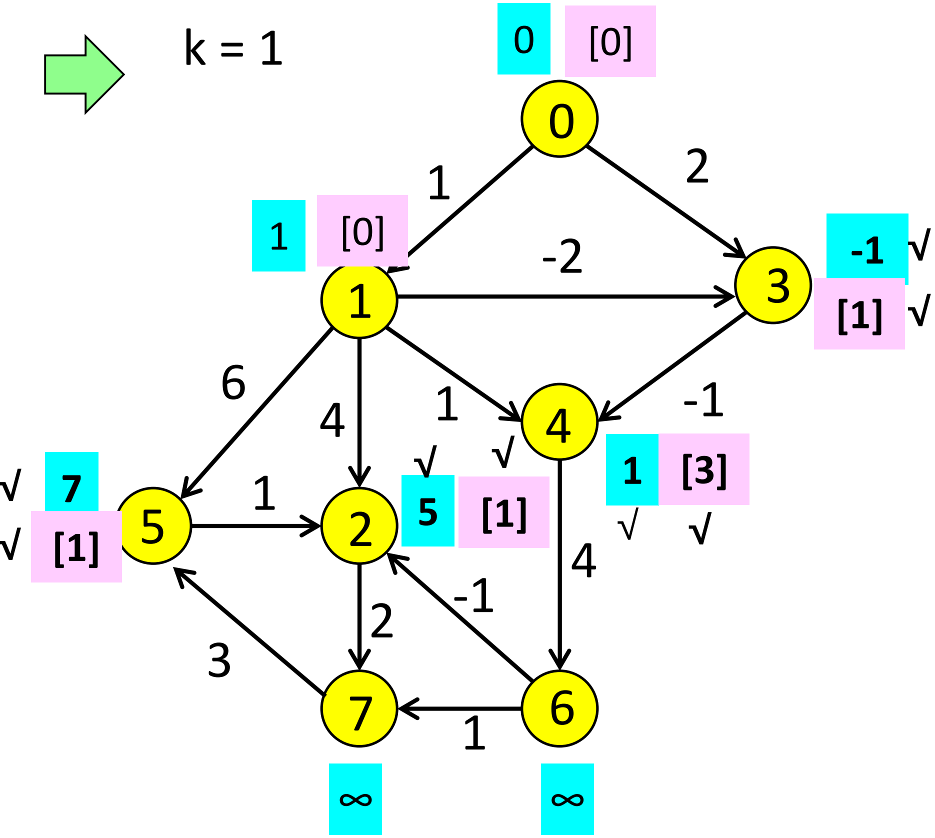
K	0	1	2	3	4	5	6	7
0		∞	∞	∞	∞	∞	∞	∞

k = 0



K	0	1	2	3	4	5	6	7
0		1	∞	2	∞	∞	∞	∞

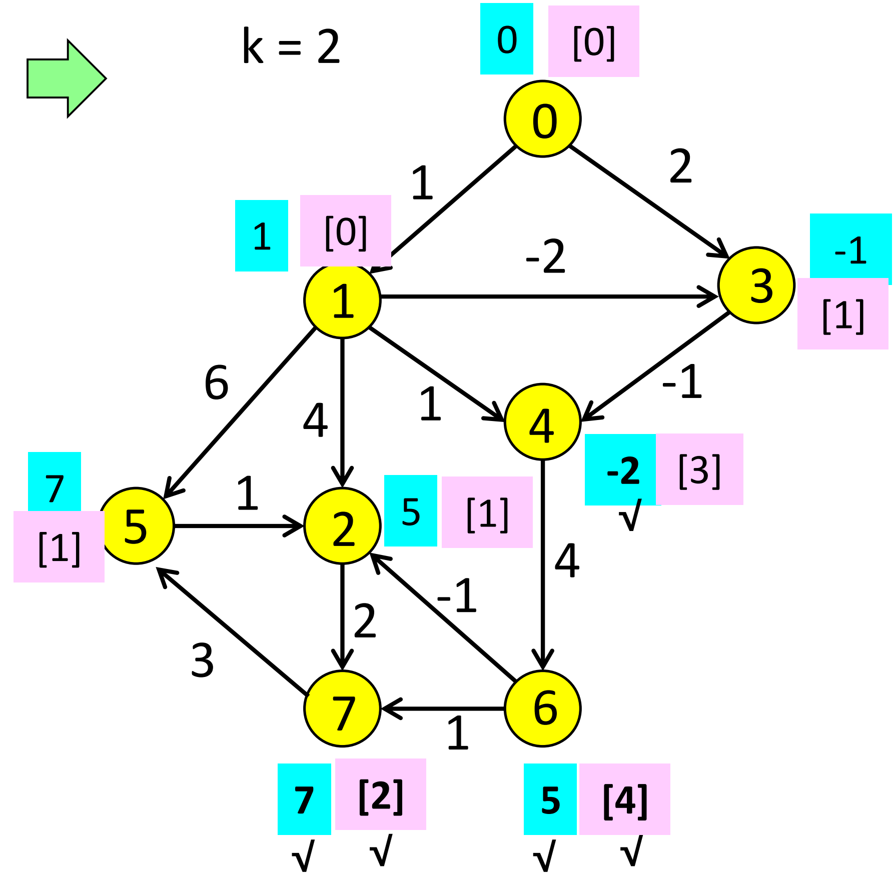
k = 1



K	0	1	2	3	4	5	6	7
0		1	∞	2	∞	∞	∞	∞
1		1	5	-1	2	7	∞	∞

K	0	1	2	3	4	5	6	7
0		1	∞	2	∞	∞	∞	∞
1		1	5	-1	1	7	∞	∞

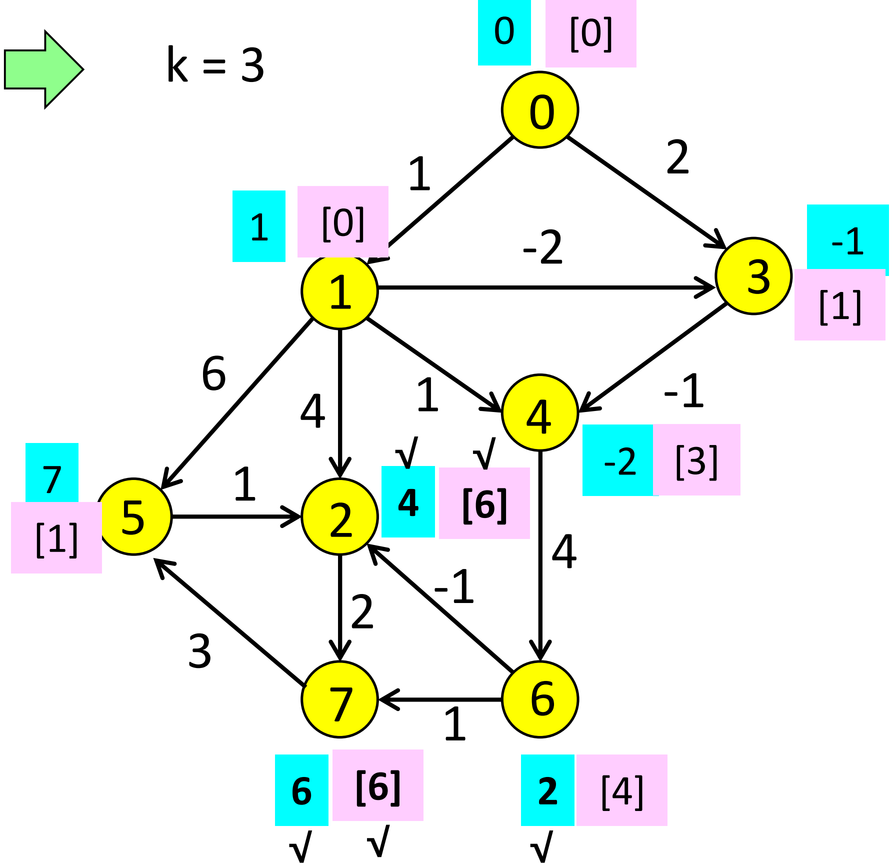
k = 2



K	0	1	2	3	4	5	6	7
1		1	5	-1	1	7	∞	∞
2		1	5	-1	1	7		7

K	0	1	2	3	4	5	6	7
1		1	5	-1	1	7	∞	∞
2		1	5	-1	-2	7		7

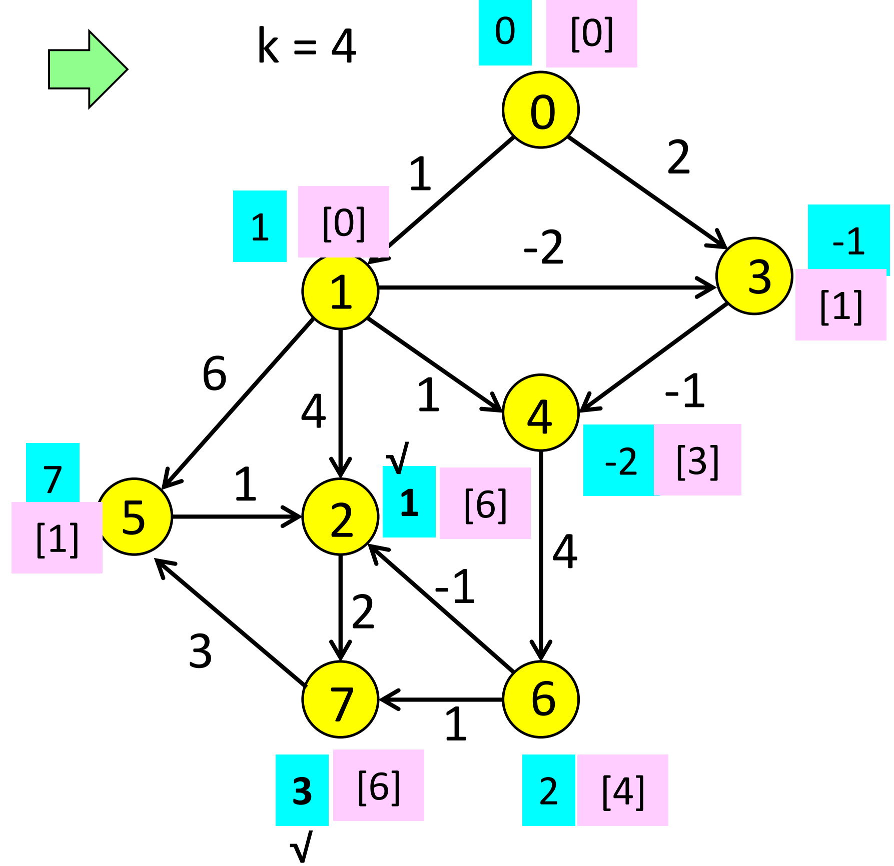
K	0	1	2	3	4	5	6	7
1		1	5	-1	1	7	∞	∞
2		1	5	-1	-2	7	5	7



K	0	1	2	3	4	5	6	7
2		1	5	-1	**-2**	7	5	7
3		1	5	-1	-2	7	**2**	7

K	0	1	2	3	4	5	6	7
2		1	5	-1	-2	7	**5**	7
3		1	**4**	-1	-2	7	2	**6**

K	0	1	2	3	4	5	6	7
2		1	5	-1	-2	7	5	**7**
3		1	4	-1	-2	7	2	6

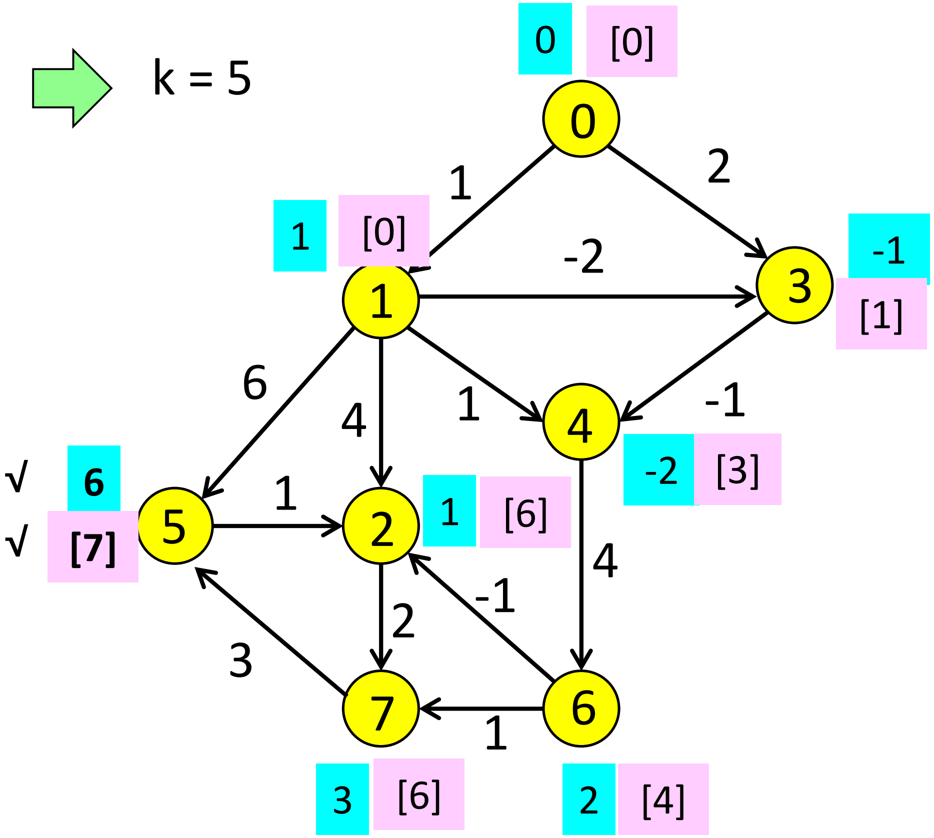


K	0	1	2	3	4	5	6	7
3		1	**4**	-1	-2	7	2	6
4		1	4	-1	-2	7	2	6

K	0	1	2	3	4	5	6	7
3		1	4	-1	-2	7	**2**	6
4		1	**1**	-1	-2	7	2	**3**

K	0	1	2	3	4	5	6	7
3		1	4	-1	-2	7	2	**6**
4		1	1	-1	-2	7	2	3

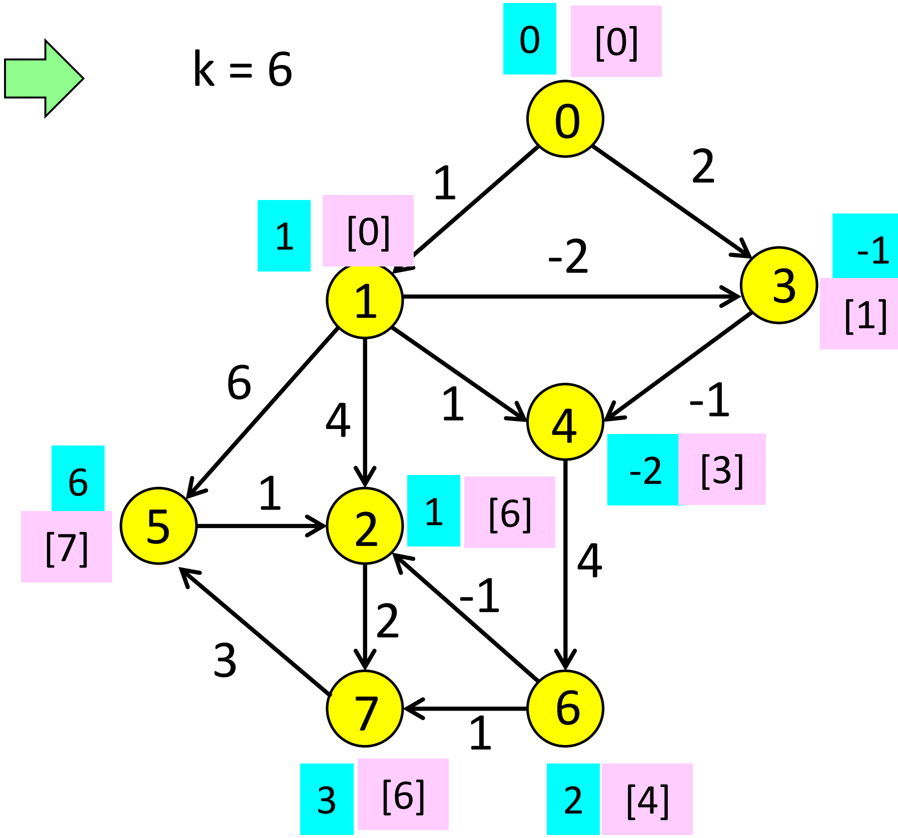
k = 5



K	0	1	2	3	4	5	6	7
4		1	1	-1	-2	7	2	3
5		1	1	-1	-2	7	2	3

K	0	1	2	3	4	5	6	7
4		1	1	-1	-2	7	2	3
5		1	1	-1	-2	6	2	3

k = 6



K	0	1	2	3	4	5	6	7
5		1	1	-1	-2	6	2	3
6		1	1	-1	-2	6	2	3

```java
public class BellmanFord {
    public static final int INF = Integer.MAX_VALUE;
    private int D[];
    private int previous[];  // 경로 추출을 위해
    private int N;

    public BellmanFord(int numOfVertices) { // 생성자
        N = numOfVertices;
        D = new int[N];             // 최단거리 저장
        previous = new int[N];      // 최단경로 추출하기 위해
    }

    public void shortestPath(int s, int adjMatrix[][]) {
        for (int i = 0; i < N; i++)
            D[i] = INF;  //초기화
        D[s] = 0; previous[s] = 0;
        for (int k = 0; k < N-1; k++) {  // 총 N-1번 반복
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    if (adjMatrix[i][j] != INF) {
                        if (D[j] > D[i] + adjMatrix[i][j]){
                            D[j] = D[i] + adjMatrix[i][j];  // 간선 완화
                            previous[j] = i;  // i 덕분에 j까지 거리가 단축됨
                        }
                    }
                }
            }
        }
    }
    public void printPaths(int s){   // 결과 출력
      // 생략
    }
}
```

- Bellman-Ford 클래스에서 line 17의 for-루프는 N-1회 반복 수행
- Line 21~22: 각 정점에 대해 if-조건이 만족되면 간선완화 수행
- Line 23: 갱신될 때 정점 i를 previous[j]에 저장
- Line 30 이후는 결과 출력을 위한 메소드 생략

```java
public class main {
    public static final int INF = Integer.MAX_VALUE;
    public static void main(String[] args) {
        int[][] weight = {
                { INF,   1, INF,   2, INF, INF, INF, INF},
                { INF, INF,   4,  -2, INF,   6, INF, INF},
                { INF, INF, INF, INF, INF, INF, INF,   2},
                { INF, INF, INF, INF,  -1, INF, INF, INF},
                { INF, INF, INF, INF, INF, INF,   4, INF},
                { INF, INF,   1, INF, INF, INF, INF, INF},
                { INF, INF,  -1, INF, INF, INF, INF,   1},
                { INF, INF, INF, INF, INF,   3, INF, INF}
        };
        int N = weight.length; // 그래프 정점의 수

        int s = 0; // 출발점
        BellmanFord bf = new BellmanFord(N);    // 객체 생성
        bf.shortestPath(s, weight);             // 최단경로 찾기
        bf.printPaths(s);                       // 결과 출력
    }
}
```

Console

```
<terminated> main (69) [Java Application] C:₩Program Files₩Jav
정점 0으로부터의 최단거리        정점 0으로부터의 최단경로
[0,1] =   1                  1<-0
[0,2] =   1                  2<-6<-4<-3<-1<-0
[0,3] =  -1                  3<-1<-0
[0,4] =  -2                  4<-3<-1<-0
[0,5] =   6                  5<-7<-6<-4<-3<-1<-0
[0,6] =   2                  6<-4<-3<-1<-0
[0,7] =   3                  7<-6<-4<-3<-1<-0
```

수행시간

- Bellman-Ford알고리즘은 그래프의 인접행렬을 사용하여 N-1번의 반복을 통해 각 간선 <i, j>에 대해 D[j]를 계산하므로 총 수행시간은 $(N-1) \times N \times O(N) = O(N^3)$
- 인접리스트를 사용하면 $(N-1) \times O(M) = O(NM)$의 수행시간이 소요

9.5.3 Floyd-Warshall 알고리즘

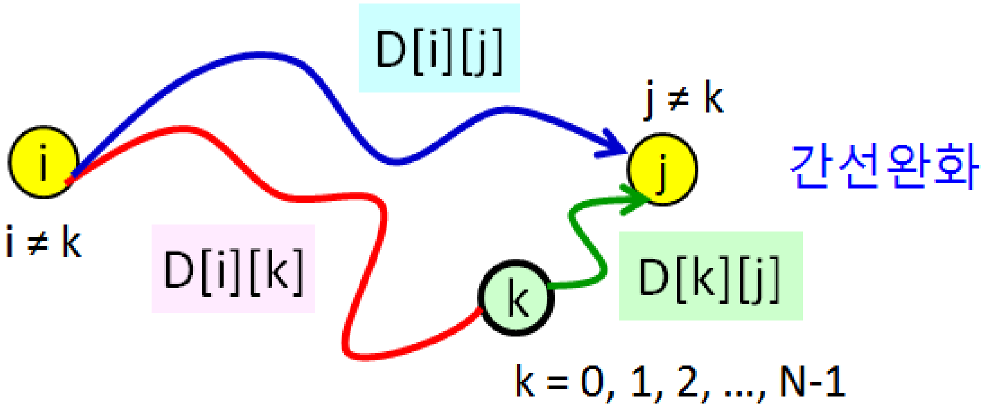
- Floyd-Warshall 알고리즘은 모든 정점 쌍 사이의 최단경로 계산
- 모든 쌍 최단경로(All Pairs Shortest Paths) 알고리즘
- 지도에서 도시간 거리를 계산한 표를 볼 수 있는데, Floyd-Warshall 알고리즘을 사용하면 얻을 수 있음

	서울 Seoul	인천 Incheon	수원 Suwon	대전 Daejeon	전주 Jeonju	광주 Gwangju	대구 Daegu	울산 Ulsan	부산 Busan
서울		40	40	155	230	320	300	410	430
인천			55	175	250	350	320	450	450
수원				130	190	300	270	355	390
대전					95	185	150	260	280
전주						105	220	330	320
광주							220	330	270
대구								110	135
울산									50
부산									

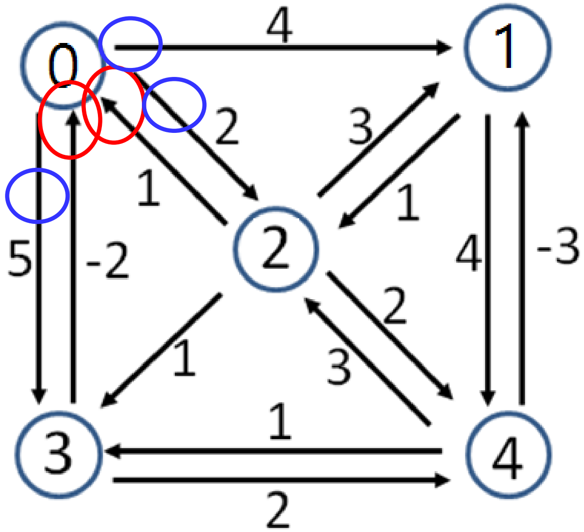
- **모든 쌍 최단경로 찾기는 출발점을 0에서 N-1까지 바꿔가며 Dijkstra 알고리즘을 각각 수행하는 것으로 모든 쌍에 대한 최단경로를 찾을 수 있음**
  - 이때 인접행렬을 사용하면 수행시간은 $O(N^3)$
- **Floyd-Warshall 알고리즘의 수행시간도 $O(N^3)$**
  - 하지만 Dijkstra 알고리즘에 비해 훨씬 알고리즘이 간단
  - 음수가중치 그래프에서도 최단경로를 찾을 수 있다는 장점을 가짐

[핵심 아이디어] 입력그래프의 정점들에 0, 1, 2, ⋯, N-1로 ID를 부여하고, 정점 ID를 증가시키며 간선완화를 수행

정점 0을 경유하는 경로에 존재하는 정점들에 대해 간선완화를 수행하고, 갱신된 결과를 바탕으로 정점 1을 경유하는 경로에 존재하는 정점들에 대해 간선완화를 수행, ⋯, 정점 N-1을 경유하는 경로에 존재하는 정점들에 대해 간선완화 수행

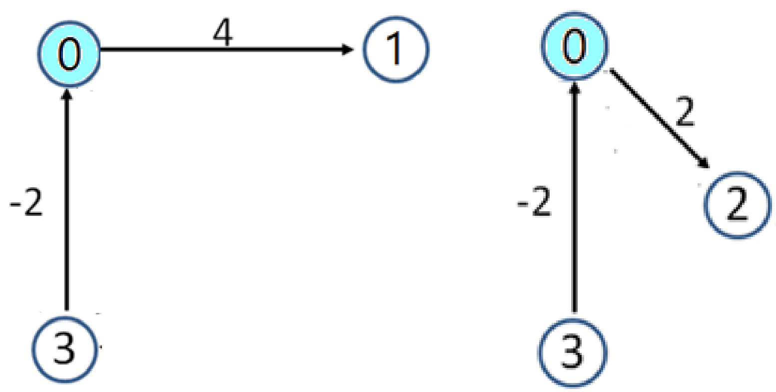


[예제]

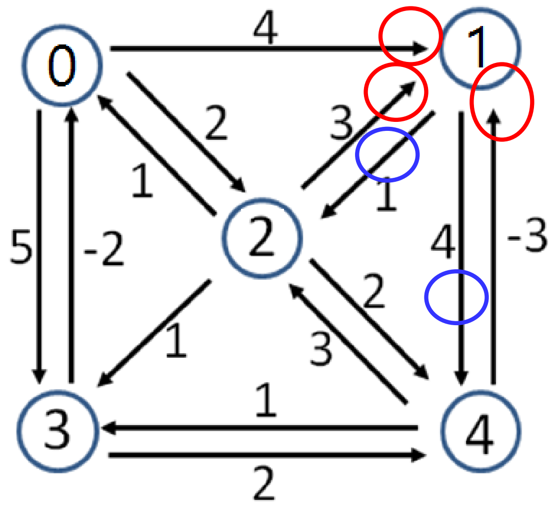


D	0	1	2	3	4
0	0	4	2	5	∞
1	∞	0	1	∞	4
2	1	3	0	1	2
3	-2	∞	∞	0	2
4	∞	-3	3	1	0

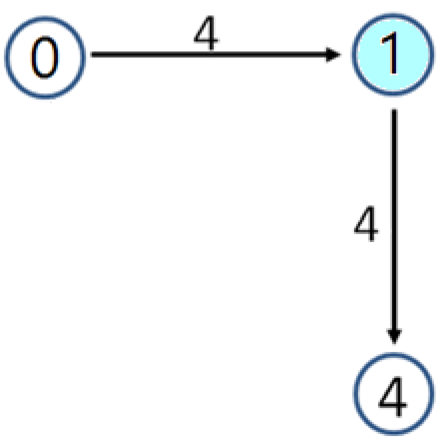
k=0

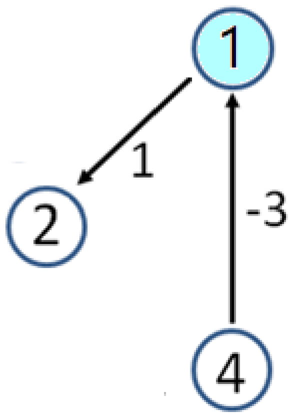


D	0	1	2	3	4
0	0	4	2	5	∞
1	∞	0	1	∞	4
2	1	3	0	1	2
3	-2	**2**	**0**	0	2
4	∞	-3	3	1	0



- **k=1**
  - D[0,4]= 8, due to 0 → 1 → 4
  - D[4,2]=-2, due to 4 → 1 → 2





D	0	1	2	3	4
0	0	4	2	5	**8**
1	∞	0	1	∞	4
2	1	3	0	1	2
3	-2	2	0	0	2
4	∞	-3	**-2**	1	0

- k=2

D	0	1	2	3	4
0	0	4	2	**3**	**4**
1	**2**	0	1	**2**	**3**
2	1	3	0	1	2
3	-2	2	0	0	2
4	**-1**	-3	-2	**-1**	0

- k=3

D	0	1	2	3	4
0	0	4	2	3	4
1	**0**	0	1	2	3
2	**-1**	3	0	1	2
3	-2	2	0	0	2
4	**-3**	-3	-2	-1	0

- k=4

D	0	1	2	3	4
0	0	**1**	2	3	4
1	0	0	1	2	3
2	-1	**-1**	0	1	2
3	-2	**-1**	0	0	2
4	-3	-3	-2	-1	0

Floyd-Warshall 알고리즘

```
[1] for (i = 0; i < N; i++)
[2]     for (j = 0; j < N; j++)              초기화
[3]         D[i][j] = adjMatrix[i][j];
[4] for (k = 0; k < N; k++)
[5]     for (i = 0; i < N; i++)
[6]         for (j = 0; j < N; j++)
[7]             if (D[i][j] > D[i][k] + D[k][j])
[8]                 D[i][j] = D[i][k] + D[k][j];   // 간선완화
```

- [1]~[3]에서 입력그래프의 인접행렬 adjMatrix를 모든 쌍 최단거리를 저장할 배열 D에 복사
- [4]의 for-루프는 경유하는 정점 ID를 0부터 N-1까지 수행
- 모든 쌍 i와 j에 대하여 [5]~[6]의 이중 for-루프가 i와 j를 각각 0부터 N-1까지 증가시키며, [7]~[8]에서 간선완화를 수행

수행시간

- **Floyd-Warshall 알고리즘의 수행시간은 $O(N^2) + O(N^3) = O(N^3)$**
  - [1] ~ [3]의 배열 복사에 $O(N^2)$이 소요되고, 이후 for-루프가 3개가 중첩되므로 $O(N^3)$이 소요
- **Bellman-Ford의 최단경로 알고리즘과 Floyd-Warshall의 최단경로 알고리즘: 동적계획(Dynamic Programming) 알고리즘**
  - 동적계획 알고리즘은 작은 부분문제(Subproblem)들의 해를 먼저 계산하고 그 해들을 바탕으로 그 다음으로 큰 부분문제들을 해결하면서 주어진 문제의 해를 계산
  - 반면에 Dijkstra의 최단경로 알고리즘은 그리디 알고리즘으로 입력 전체를 고려하지 않고 지역적인 입력에 대해 그리디하게 선택하며 이를 축적하여 해를 얻음

9.6 소셜네트워크 분석 (Social Network Analysis)

- **소셜네트워크 분석은 그래프이론을 통해 소셜 객체들의 관계를 연구**
  - 정점은 소셜 객체이고 간선은 객체의 관계를 나타냄
  - 관계는 친밀도, 유사도 등으로 표현

소셜네트워크의 예

- 소셜네트워크에는 페이스북(Facebook)이나 트위터(Twitter) 사용자들의 관계를 나타내는 사용자 네트워크
- 국제무역 관계 네트워크, 전염병 확산 네트워크, 전력공급 네트워크, 먹이사슬 관계 네트워크, 유전자 관계 네트워크 등

9.6 소셜네트워크 분석 (Social Network Analysis)

- **그래프이론은 이러한 네트워크들의 특징을 분석하는데 필수적인 도구**
  - 9.2 절에서 배운 너비우선탐색을 이용하여 두 사용자 간의 거리를 측정
  - 9.3절의 강연결성분 찾기에 기반하여 웹(www)의 구성을 분석, 소셜네트워크에서 커뮤니티(Community)를 분석
  - 전염병의 확산이나 정보확산(Information Diffusion)에 대해 분석
  - 전력 네트워크의 강건성(Robustness)에 관한 연구 등

9.6 소셜네트워크 분석 (Social Network Analysis)

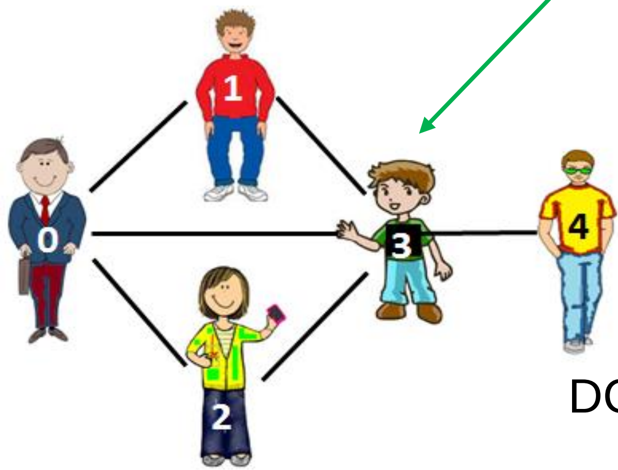
- **중심성(Centraility): 사용자가 네트워크에서 다른 사용자들에게 주는 영향력**
  - 사용자가 얼마나 중요한 지를 나타내는 척도
- **중심성(Centraility) 종류**
  - 차수중심성, 중개중심성, 근접중심성, 고유벡터중심성 등
- **사용자 중심성 분석:**
  - 정보가 얼마나 빠르게 확산되는지
  - 전염병의 급속한 확산을 막기 위한 대책
  - 전체 네트워크의 다운 방지대책 수립에 매우 중요한 정보를 제공

차수중심성(Degree Centrality, DC)

- **차수중심성: 정점(사용자)의 차수(Degree)를 N-1로 나눈 값**
- **차수를 N-1로 나누는 것을 정규화(Normalization)라고 함**
  - 정점 수가 수천만에서 수십억인 경우 정점들의 차수의 차이가 너무 크므로, 정점의 차수를 최대 차수인 N-1로 나누어 그 결과 값이 0.0 ∽ 1.0 사이 값이 되도록 하기 위함
- **차수중심성의 문제점**
  - 단순히 정점의 친구가 몇 명인가를 보는 것이기 때문에 네트워크 전체적인 관점에서의 사용자의 중요성을 대표하는 값으로는 부적절

[예제] 네트워크에서 정점 3의 차수가 4로서 가장 크고 총 5개의 정점이 있으므로 정점 3의 DC는 4/(5-1) = 1.0이다.

DC(0) = 3/4 = 0.75

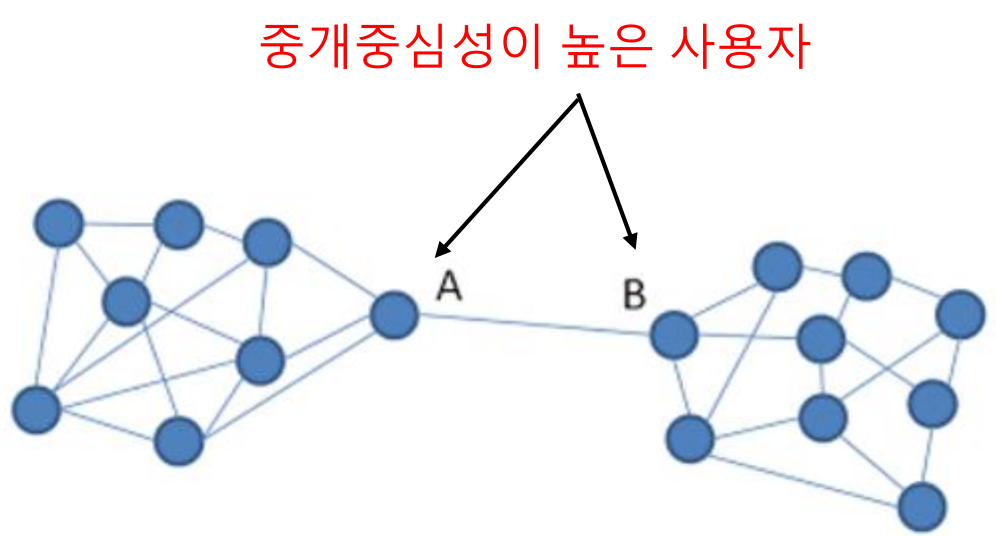


DC(4) = 1/4 = 0.25

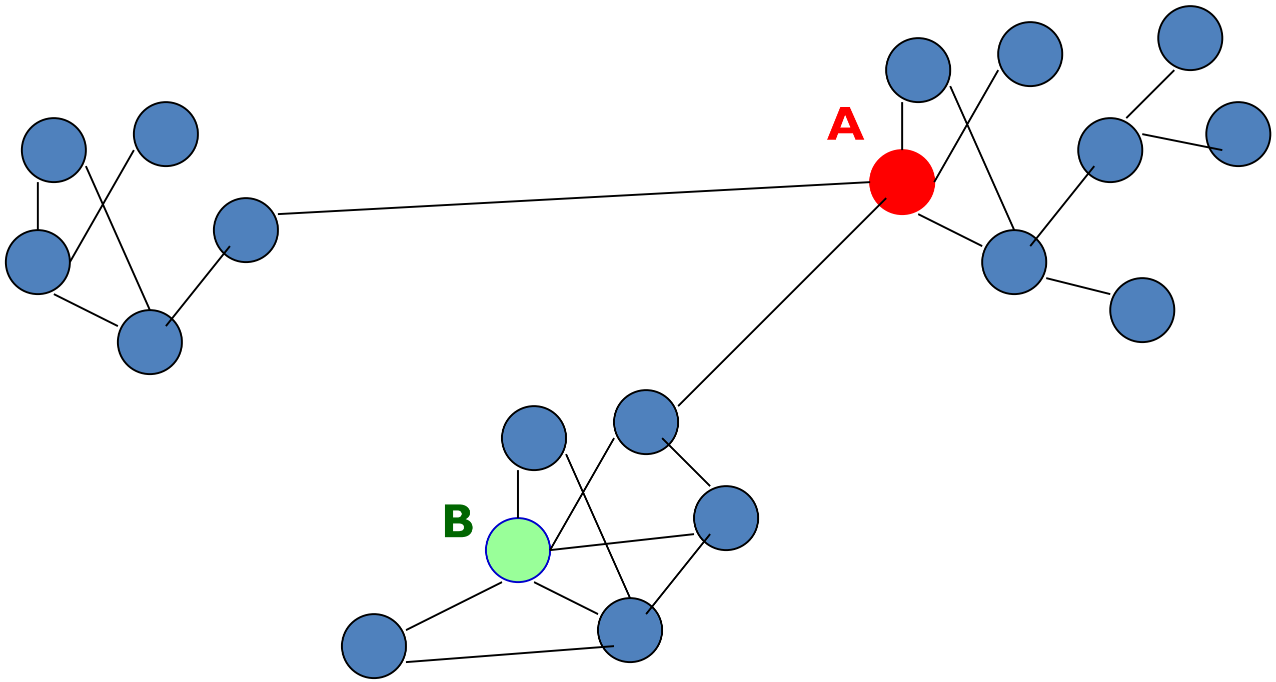
DC(2) = 2/4 = 0.5

중개중심성(Betweenness Centrality, BC)

- 중개중심성(Betweenness Centrality, BC): 얼마나 많은 쌍의 정점들 사이의 최단경로가 하나의 정점을 지나가는 지를 나타내는 척도
- 정점의 중개인(broker) 역할을 하는 정도를 의미



- A와 B의 차수중심성은 동일하나
- A는 네트워크에서 B보다 중요한 역할



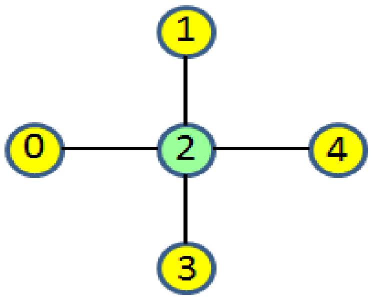
[예제]

(a) 정점 2를 통해 총 6개의 최단경로가 지나간다.

정점 0→ 1, 0→ 3, 0→ 4, 1→3, 1→4, 3→ 4

(b) 총 4개의 최단경로가 정점 2를 지난다.

정점0→ 3, 0→ 4 , 1→ 3, 1→ 4



(a)

(b)

(a) 정점 2의 차수중심성이 4/(5 – 1) = 1.0으로 가장 높고, 매개중심성도 높으나,

(b) 정점 1, 2, 3의 차수중심성이 2/(5 - 1) = 0.5로 동일하여 차수중심성은 정보를 전달하는데 어느 정점이 더 중요한 역할을 하는지를 보여주지 못함

$$BC(i) = \sum_{\forall j,k \neq i} \left(\frac{j\text{에서 } i\text{를 경유하여 } k\text{로 가는 최단경로의 수}}{j\text{에서 } k\text{로 가는 총 최단경로의 수}} \right)$$

Floyd-Warshall 최단경로 알고리즘 이용

$i \neq j,\ i \neq k,\ j < k,$

j가 k보다 작아야 하는 이유: 경로의 중복 계산 방지

예를 들어, 3에서부터 6까지의 최단경로를 계산했으면,
6에서부터 3까지의 경로는 계산하지 않는다

- BC 값을 정규화할 때에는BC(i)를 (N-1)(N-2)/2로 나눈다.
  - ✓ (N-1)(N-2)/2 = 정점 i를 제외한 N-1개의 정점에 대해, 2개씩 쌍을 만드는 최대 조합의 수

[예제]

아래의 네트워크에서 (N-1)(N-2)/2 = (5-1)(5-2)/2 = 12/2 = 6

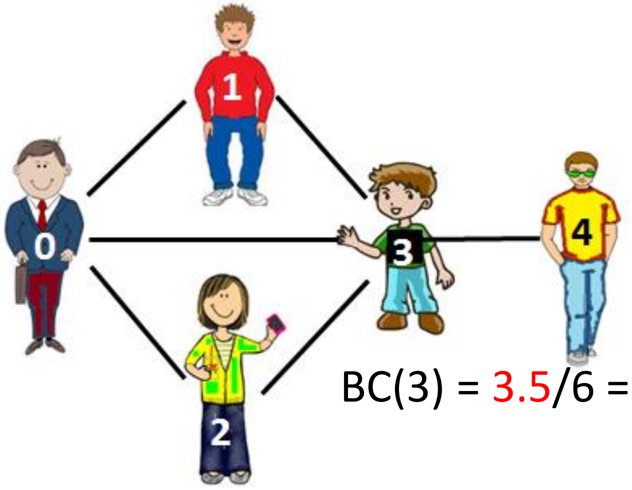
- BC(0) = 0.5/6 = 0.08

  정점 1에서 2로 가는 최단경로의 수는 2 개이고 0을 경유하는 경우가 1 개이므로 1/2 = 0.5

- BC(3) = 3.5/6 = 0.58

  {0, 1, 2} 각각에서 3을 거쳐야 정점 4를 갈 수 있고, 정점 1에서 2로 가는 최단경로의 수는 2 개이고 3을 경유하는 경우가 1 개이므로 3 + 1/2 = 3.5

BC(0) = 0.5/6 = 0.08

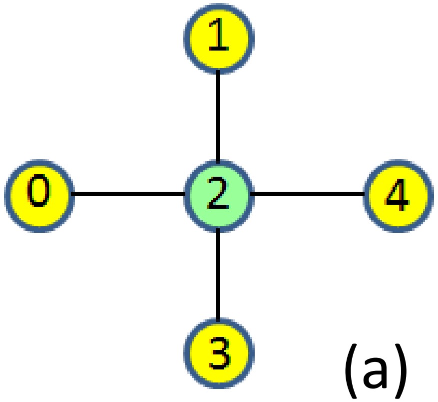


BC(3) = 3.5/6 = 0.58

BC(1) = BC(2) = BC(4) = 0/6 = 0

[예제]

- (a) BC(2) = 6/6 = 1.0
  BC(0) = BC(1) = BC(3) = 0/6 = 0.0



(a)

- (b) BC(0) = BC(4) = 0/6 = 0.0
  BC(1) = BC(3) = 3/6 = 0.50, BC(2) = 4/6 = 0.67

(b)

근접중심성(Closeness Centrality, CC)

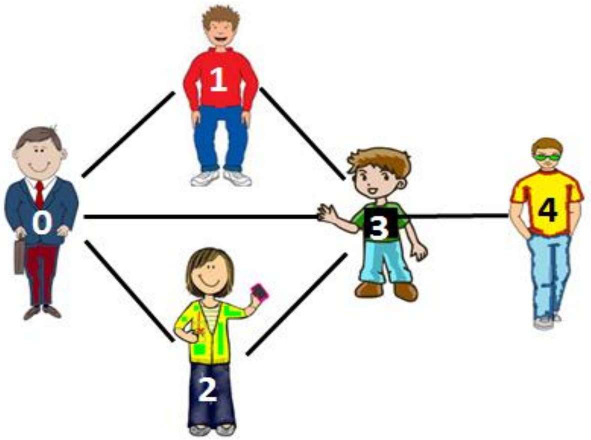
- **근접중심성: 정점 i에서 N-1개의 정점까지 각각의 최단거리의 합**
  - 정점 i가 네트워크에서 얼마나 중앙에 위치하는 지를 나타내는 척도 [정보 확산의 척도]
- **CC를 계산하는 정규화된 식은 다음과 같다.**
  - CC 값이 클수록 네트워크의 중앙에 위치

$$\mathrm{CC(i)} = \sum_{\mathrm{j=0}}^{\mathrm{N-1}} \frac{\mathrm{N-1}}{\mathrm{d_{ij}}}, \mathrm{i} \neq \mathrm{j}$$

$\mathrm{d_{ij}}$는 i에서 j까지의 최단거리

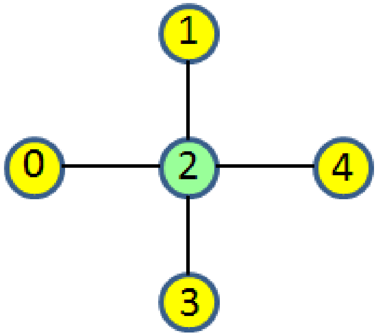
[예제]

- CC(0) = (5-1) / (1 + 1 + 1 + 2) = 4 / 5 = 0.80
- CC(1) = CC(2) = (5-1) / (1 + 1 + 2 + 2) = 4 / 6 = 0.67
- CC(3) = (5-1) / (1 + 1 + 1 + 1) = 4 / 4 = 1.00
- CC(4) = (5-1) / (1 + 2 + 2 + 2) = 4 / 7 = 0.57
- 정점 3이 가장 근접중심성이 높고, 정점 4가 가장 낮다.

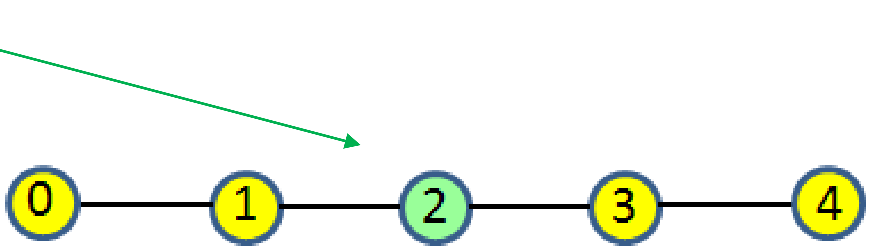


[예제]

- (a) CC(2) = (5-1)/(1 + 1 + 1 + 1) = 1.00
  각 주변 정점의 CC 값은 (5-1) / (1 + 2 + 2 + 2) = 4 / 7= 0.57



- (b) CC(2) = (5-1) / (1 + 1 + 2 + 2) = 4 / 6 = 0.67



고유벡터중심성(Eigenvector Centrality, EC)

- **정점 i 가 얼마나 중요한(Important 또는 Popular) 정점에 인접해있는지를 나타내는 척도**
  - 많은 사람들이 높은 관심을 가지는 인물(Influential)의 친구도 비교적 높은 인지도를 갖는다는 것을 반영하는 것
  - EC(i)는 정점 i에 인접한 정점(친구)들의 중심성에 의존

- **고유벡터중심성 알고리즘**

[1] 각 정점 i에 1.0을 초기 중심성 값으로 배정한다. 즉, $\mathrm{EC(i)} = 1.0.$

[2] 각 정점 i에 대하여 $\mathrm{EC(i)} = \sum_{j=0}^{N-1} \mathrm{A[i][j]EC(j)}$를 계산한다. A는 인접행렬

[3] 각 정점 i의 $\mathrm{EC(i)}$를 가장 큰 EC() 값으로 나누어 정규화한다.

[4] [2]와 [3]을 $\mathrm{EC(i)}$ 값이 변하지 않을 때까지 반복 수행

- **원래의 고유벡터중심성 계산방법은 인접행렬의 고유벡터를 계산하여 고유벡터중심성 계산**

$$Ax = \lambda x$$

- **A=인접행렬, x=고유벡터(Eigenvector), λ=고유값(Eigenvalue)**
  - 위의 식을 계산하면 N개의 고유값을 얻는데, 이때 가장 큰 고유값을 선택하여 고유벡터 x를 계산하면 각 정점의 고유벡터중심성을 얻음

[예제] 그래프의 인접행렬에 대한 5개의 고유값은

{-1.74, -1.27, 0.00, 0.33, 2.68}

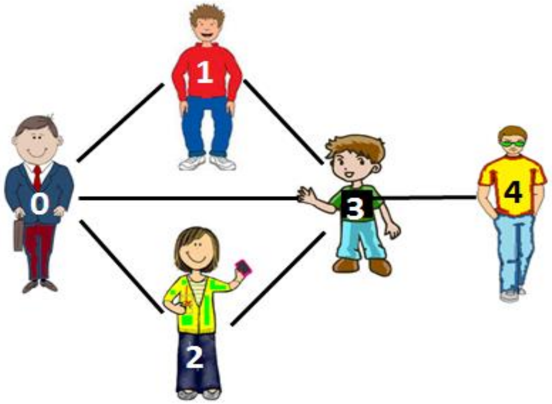
이 중에서 가장 큰 고유값인 2.68을 택하고,

2.68에 대응되는 고유벡터를 계산하면

[0.524, 0.412, 0.412, 0.583, 0.217]

차례로 EC(0) = 0.524, EC(1) = 0.412, EC(2) = 0.412, EC(3) = 0.583, EC(4) = 0.217이다.

정점 3의 고유벡터중심성이 가장 크다.

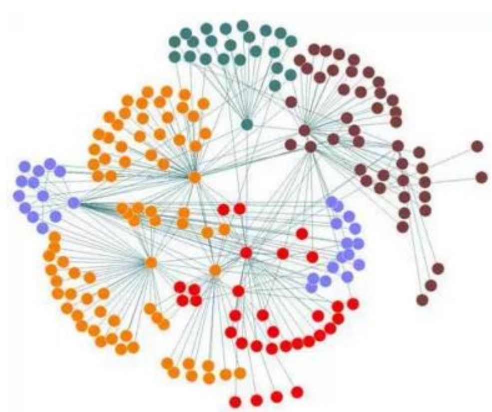


커뮤니티 찾기(Community Detection)

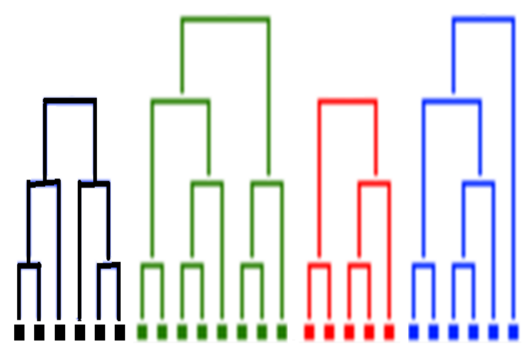
- **커뮤니티 찾기(Community Detection)는 소셜네트워크에서**
  - 유사도(Similarity)에 따라 사용자들을 그룹화하고,
  - 추천시스템(Recommendation System)에서는 취향이나 관심분야가 유사한 고객들을 집단으로 분류하며,
  - 웹 탐색엔진(Search Engine)의 탐색성능을 높이기 위해 유사한 웹 페이지들을 그룹으로 나누고,
  - 복잡한 네트워크를 간단하게 시각화(Visualization)하는데 도움을 주는 등
- **수많은 분야에서 유용하게 쓰이는 분석 도구이다.**

커뮤니티를 찾는 방법

- **강연결성분 찾기는 방향성을 가진 트위터 네트워크나 이메일/전화 등의 통신네트워크에서 커뮤니티를 분석하는데 상당히 중요한 역할.**
- **가장 대표적인 방법으로 계층적 클러스터링(Hierarchical Clustering)**
  - 가장 유사도가 높은 두 명의 사용자 또는 그룹을 하나의 그룹으로 연속적으로 합병하며, 남은 그룹 수가 원하는 그룹 수가 되었을 때 합병을 종료



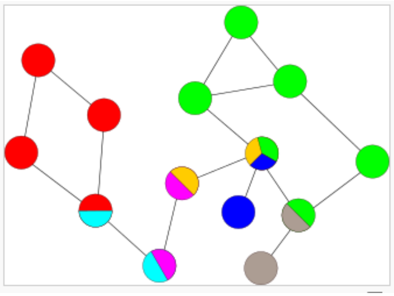
(a) 강연결성분을 이용한 커뮤니티



(b) 계층적 클러스터링으로 만든 4개의 커뮤니티

9-3-2 이중연결성분(Biconnected Component)

- **무방향그래프의 연결성분에서 임의의 두 정점들 사이에 적어도 두 개의 단순경로가 존재하는 연결성분**
  - 따라서 하나의 단순경로 상의 어느 정점 하나가 삭제되더라도 삭제된 정점을 거치지 않는 또 다른 경로가 존재하므로 연결성분내에서 정점들 사이의 연결이 유지
- **이중연결성분은 통신 네트워크 보안, 전력 공급 네트워크 등에서 네트워크의 견고성(Robustness)을 분석하는 주된 방법**

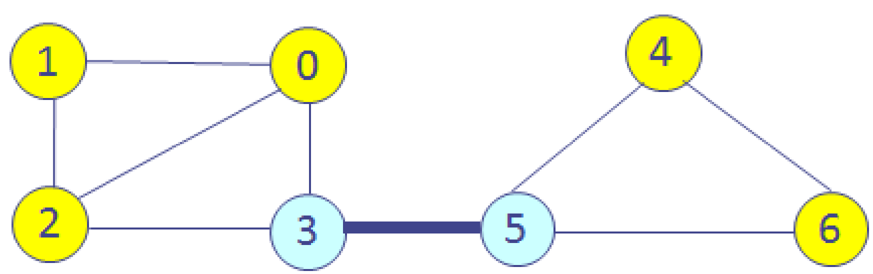


9.3 기본적인 그래프 알고리즘

- **9.3.1 위상정렬(Topological Sort)**
  - 싸이클이 없는 방향그래프(Directed Acyclic Graph, DAG)에서 정점을 선형순서(즉, 정점들을 일렬)로 나열하는 것
- **9.3.2 이중연결성분(Biconnected Component)**
- **9.3.3 강연결성분(Strongly Connected Component)**

9-3-2 이중연결성분(Biconnected Component)

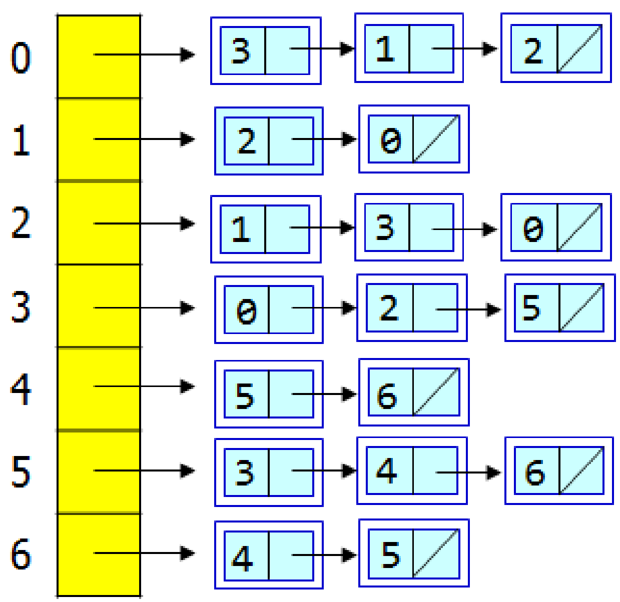
- **단절정점(Articulation Point 또는 Cut Point)**
  - 연결성분의 정점들 중 하나의 정점을 삭제했을 때, 두 개 이상의 연결성분들로 분리될 때 삭제된 정점
- **다리간선(Bridge)**
  - 간선을 제거했을 때 두 개 이상의 연결성분들로 분리될 때 삭제된 간선



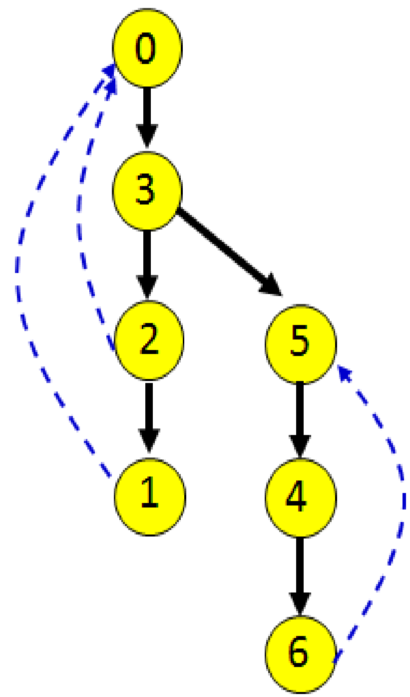
- 정점 3과 5는 각각 단절정점
- 간선 (3, 5)는 다리간선
- 위 그래프는 3 개의 이중연결성분,
  [0, 1, 2, 3], [3, 5], [4, 5, 6]으로 구성
- 단절정점은 이웃한 이중연결성분들에 동시에 속하고,
  다리간선은 그 자체로 하나의 이중연결성분

9-3-2 이중연결성분(Biconnected Component)

- 이중연결성분을 찾는 알고리즘을 알아보기 전에 DFS를 수행하며 만들어지는 DFS 신장트리와 이중연결성분과의 관계를 살펴보자.
- 그래프에 대한 인접리스트가 (a)와 같다면, 정점 0에서부터 DFS를 수행하면 (b)와 같은 신장트리를 얻음



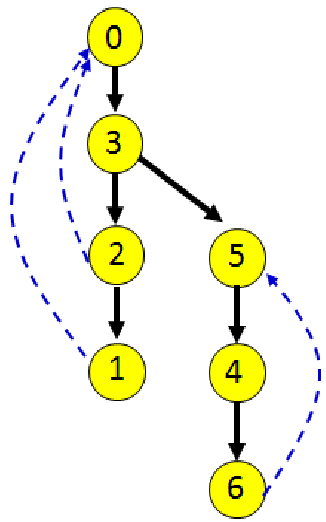
(a)



(b)

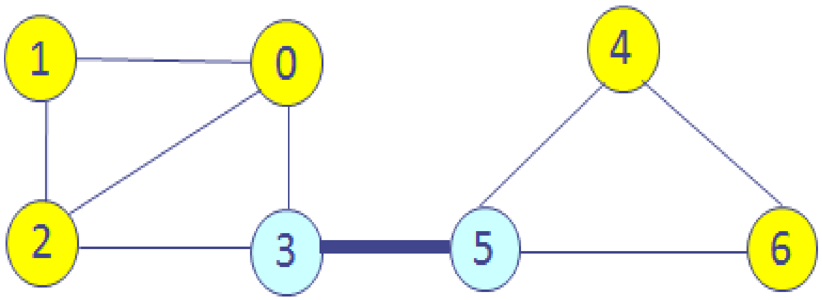
9-3-2 이중연결성분(Biconnected Component)

- **(b)의 트리에서 점선으로 표시된 각각의 뒷간선은 싸이클을 만듬**
  - 뒷간선 (2, 0)을 신장트리에 추가하면 [0-3-2-0]의 싸이클을 만들고,
  - 뒷간선 (1, 0)은 [0-3-2-1-0]의 싸이클을 만들며,
  - 뒷간선 (6, 5)는 [5-4-6-5]의 싸이클 형성
- **이중연결성분은 성분 내의 정점들 사이에 적어도 2개의 단순경로가 있어야 하므로, 뒷간선으로 만들어지는 싸이클 상의 정점들은 하나의 이중연결성분에 속함**



9-3-2 이중연결성분(Biconnected Component)

- 다음그래프는 [(0, 3), (3, 2), (2, 1), (1, 0), (2, 0)], [(3,5)], [(5, 4), (4, 6), (6, 5)]의 3개의 이중연결성분으로 구성되고 정점 3과 5가 단절정점이다.

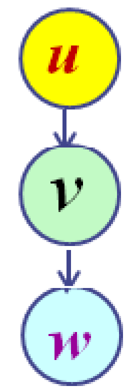


[핵심아이디어] DFS를 수행하면서 사용하는 간선을 스택에 저장하고 뒷간선의 적절한 활용을 통해 단절정점을 찾으며, 단절정점을 찾은 직후에 스택에서 이중연결성분에 속한 간선들을 모두 꺼내어 출력한다.

이중연결성분 찾기 알고리즘

- DFS를 수행하면서 각 정점에 방문번호(dfsNum)를 부여하고, DFS수행 과정에서 만들어지는 신장트리에서 뒷간선을 활용하여 가장 작은 dfsNum을 가진 정점에 도달 가능 여부를 표시하기 위해 lowNum 배열을 사용

```
sequence = 1;
dfsNum[]을 0으로 초기화;
biconnected(0, -1);    // 시작 정점 0으로 호출
biconnected(v, u) {
      dfsNum[v] = lowNum[v] = sequence++;
      for (each w adjacent to v)
            if (w≠u and dfsNum[w] < dfsNum[v])
                  간선 (v, w)를 스택에 push;
            if (dfsNum[w] == 0)  // w 가 방문 안되었으면
                  biconnected(w, v);
                  lowNum[v] = min{lowNum[v], lowNum[w]};
                  if (lowNum[w] ≥ dfsNum[v])  // v 는 단절정점
                        간선(v, w)가 나올 때까지 pop하여 출력;
            else if (w≠u)   // (v, w)가 뒷간선이면
                  lowNum[v] = min(lowNum[v], dfsNum[w]);
}
```

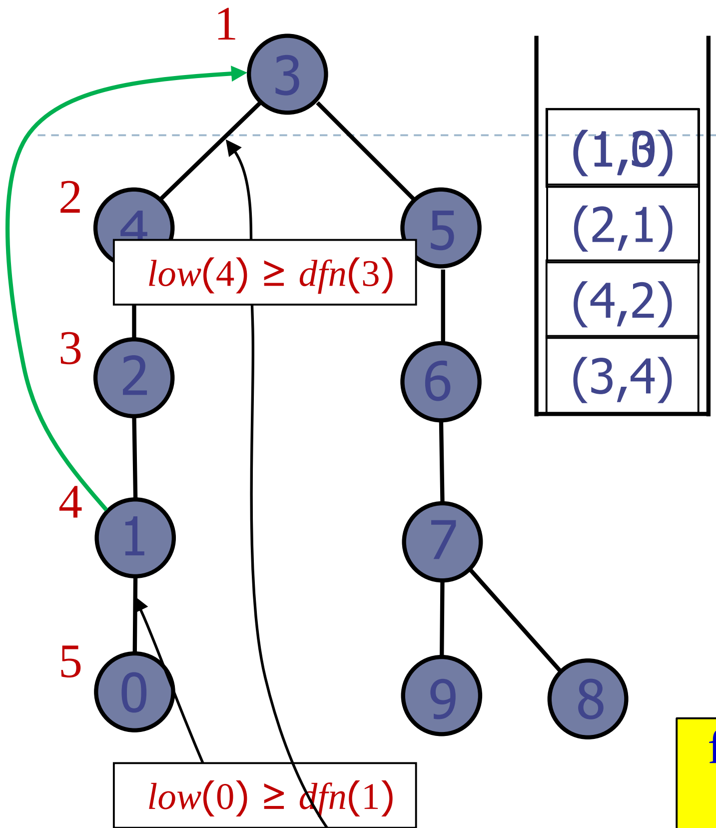


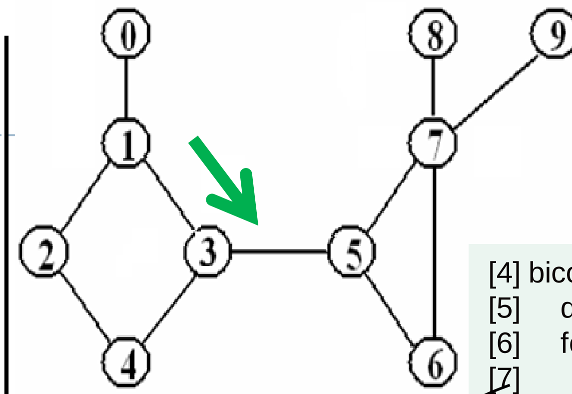
9-3-2 이중연결성분(Biconnected Component)

- Step [1]의 sequence = 1은 DFS 수행하면서 방문순서에 따라 정점에 1부터 (N까지) 방문번호(dfsNum)를 부여하기 위한 것
  - 참고로 한번 부여된 방문번호는 알고리즘이 종료될 때까지 변하지 않음
- Step [2]에서dfsNum 배열을0으로 초기화하며, dfsNum[i] = 0은 정점 i가 아직 방문되지 않았음을 의미
- Step [3]에서 biconnected(0, -1)을 호출하는데, 첫 번째 매개변수는 정점 0부터 DFS를 시작함을 의미하고, 두 번째 매개변수는 신장트리에서 첫 번째 매개변수의 부모노드이다.
  - 정점 0은 신장트리의 루트노드라서 부모노드가 없으므로 '-1'로 표현한 것
- Step [5]에서는dfsNum[v] = lowNum[v] = sequence++를 수행하여 1차적으로 정점 v에대해 dfsNum[v]와 lowNum[v]에 동일한 값을 부여
- Step [6]의 for-루프에서는 정점 v에 인접한 각 정점 w에 대해서 하나씩 step [7], [9], [14]에 있는 if-조건에 따라 해당 부분을 수행
- Step [7]의 if-문은 정점 w가 신장트리에서 정점 v의 부모노드가 아니고 dfsNum[w] < dfsNum[v]인 경우 (참고로 아직 방문되지 않은 정점의 dfsNum은 0이다), 간선 <v, w>를 스택에 push하기 위한 것

9-3-2 이중연결성분(Biconnected Component)

- Step [9]의 if-문에서는 dfsNum[w] = 0이면, 즉, w가 아직 방문되지 않은 정점이면, biconnected(w, v)를 호출
- Step [11]에서 호출이 끝나고 리턴되면 lowNum[v]값을 둘 중에서 작은 값으로 갱신
  - 만약 lowNum[w]로 갱신되었다면 v의 자식노드인 w의 도움으로 v로부터 w를 거쳐서 트리의 더 '높은' 정점에 도달할 수 있다는 것을 의미
- Step [12]는 정점 v가 단절정점인지를 검사하며, 그 조건은 lowNum[w] $\geq$ dfsNum[v]이다.
  - 즉, 정점 w에서 트리 위로 아무리 높이 올라가려고 해도 정점 v보다 위에 있는 정점에 도달 불가능하다면, 정점 v는 단절정점이다. 이는 정점 v와 w가 속한 이중연결성분을 발견한 것
- Step [13]에서 스택에 있는 간선들을 간선 (v, w)가 나올 때까지 꺼내어 출력한다. 이때, (v, w)도 포함하여 출력
- Step [14]의 if-문은 정점 w가 정점 u와 다를 때, 즉, 간선 (v, w)가 뒷간선인 경우, lowNum[v]를 둘 중에 작은 값으로 갱신
  - 이때 정점 w가 탐색 중 먼저 방문되어 v보다 작은 방문번호를 부여 받았다면, lowNum[v]는 dfsNum[w]로 갱신





Horowitz 책 예제

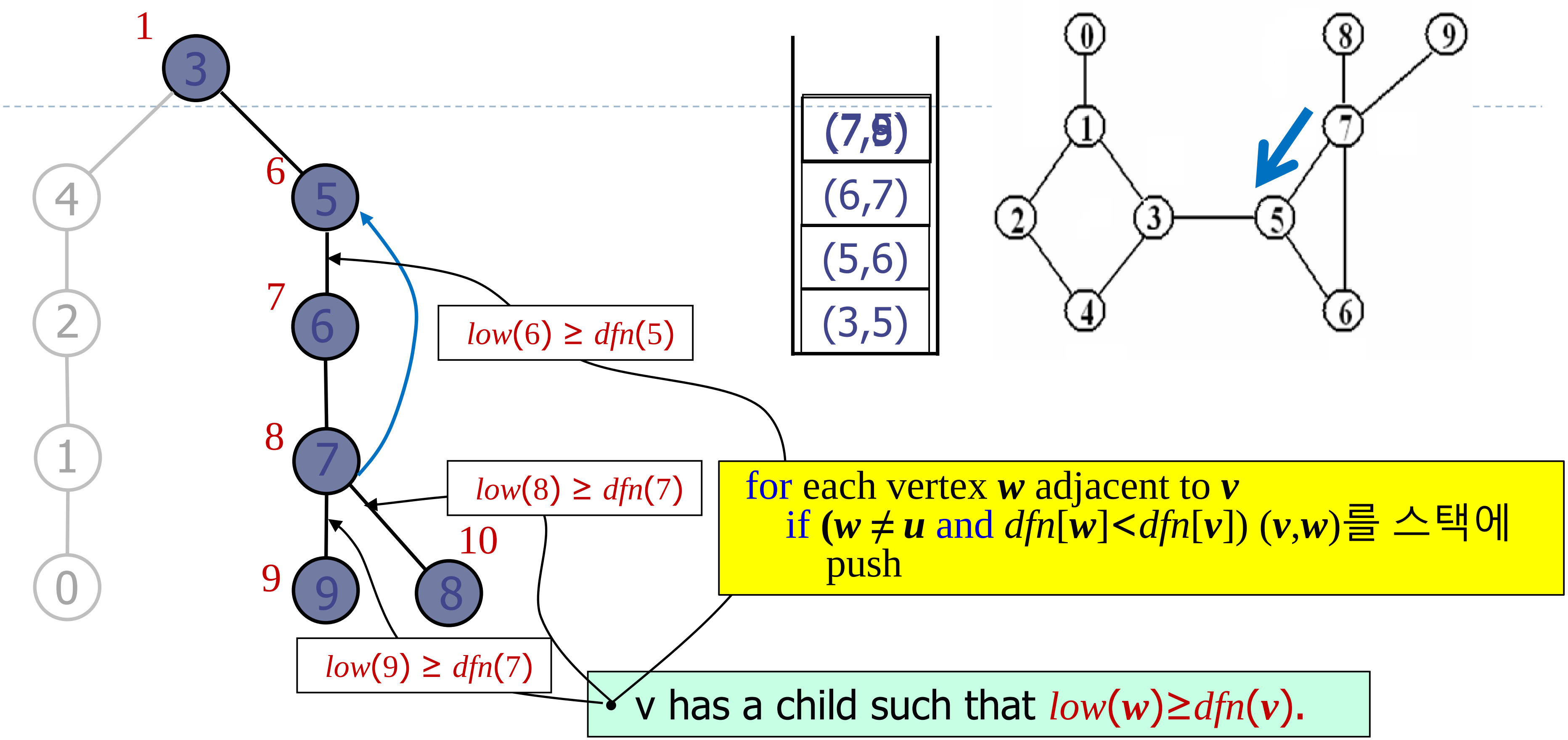
```
biconnected(v, u) {
    dfsNum[v] = lowNum[v] = sequence++;
    for (each w adjacent to v)
        if (w≠u and dfsNum[w] < dfsNum[v])
            간선 (v, w)를 스택에 push;
        if (dfsNum[w] == 0)  // w 가 방문 안되었으면
            biconnected(w, v);
            lowNum[v] = min{lowNum[v], lowNum[w]};
            if (lowNum[w] ≥ dfsNum[v])  // v 는 단절정
              간선(v, w)가 나올 때까지 pop하여 출력;
      else if (w≠u)   // (v, w)가 뒷간선이면
            lowNum[v] = min(lowNum[v], dfsNum[w]);
```

for each vertex ***w*** adjacent to ***v***
if (***w*** ≠ ***u*** and *dfn*[***w***]<*dfn*[***v***]) (***v***,***w***)를 스택에 push

- ***v*** has a child ***w*** such that *low* (***w***) ≥ *dfn*(***v***).

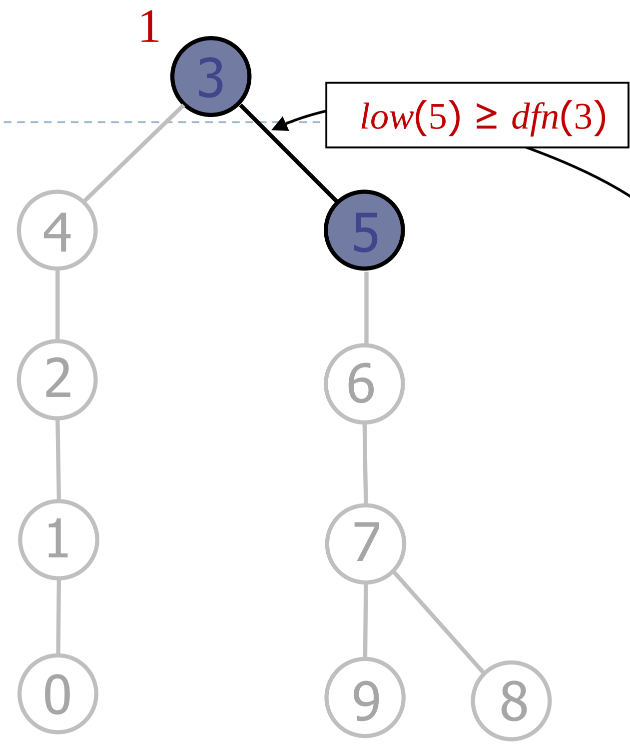
단절 정점 검사

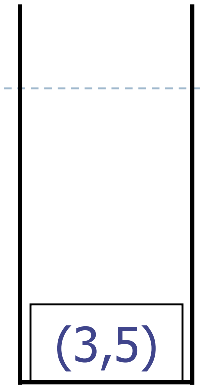
vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2					
low	5	1	1	1	1					

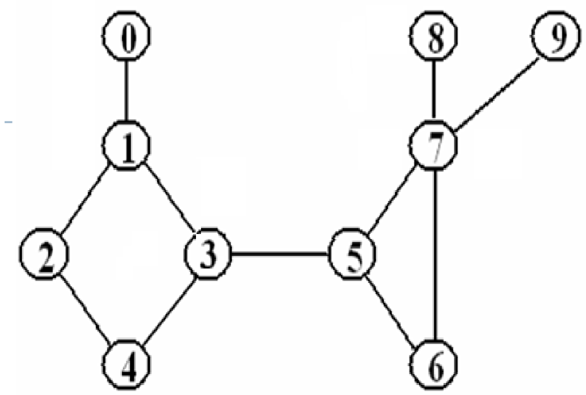


단절 정점 검사

vertex	0	1	2	3	4	5	6	7	8	9
dfn				1		6	7	8	10	9
low				1		6	6	6	10	9



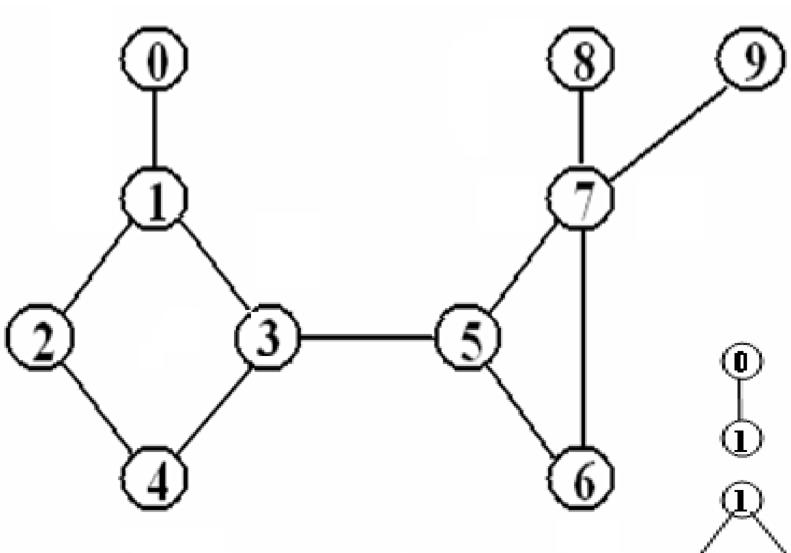


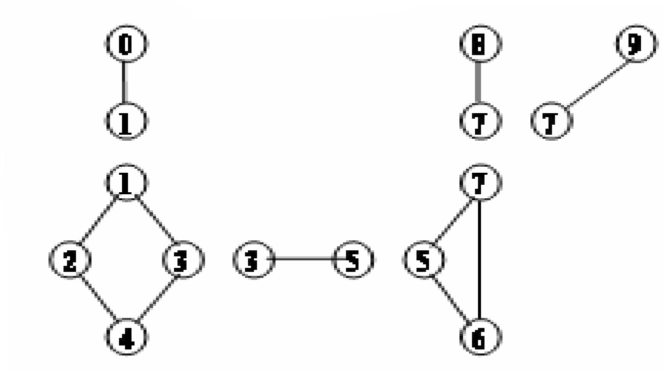


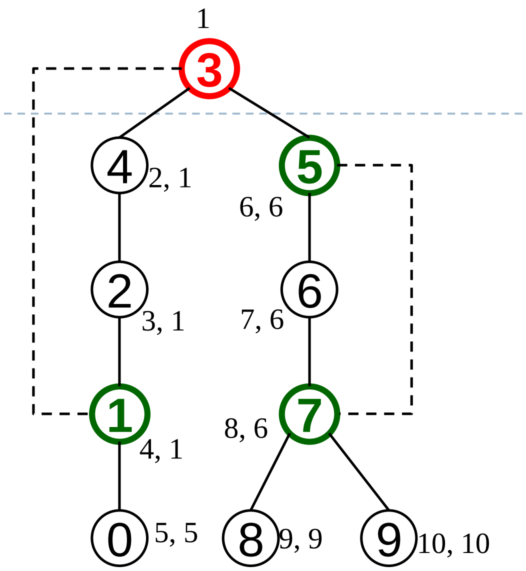
단절 정점 검사

- v has a child w of u such that $low(w) \geq dfn(v)$.

vertex	0	1	2	3	4	5	6	7	8	9
dfn				1		6				
low				1		6				







vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

깊이우선신장트리에서의 단절점 찾기

① 정점 u가 두개 이상의 자식을 가진 신장트리의 루트

② **정점 u**가 루트가 아니면서 **자식 w**가 **dfs(u)** ≤ **low(w)** 를 만족

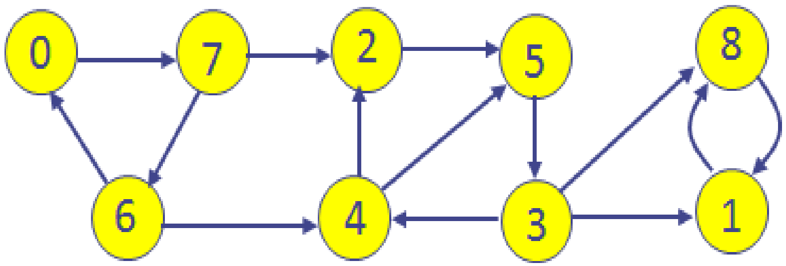
수행시간

- 이중연결성분 알고리즘의 수행시간은 DFS의 수행시간과 동일한 O(N+M)
- 기본적으로 깊이우선탐색을 수행하며 이와 별도로 소요되는 시간은 스택에 각 간선이 한번 push되고 한번 pop되는 시간으로 이는 O(M)
- 따라서 이중연결성분 알고리즘의 수행시간은 O(N+M) + O(M) = O(N+M)

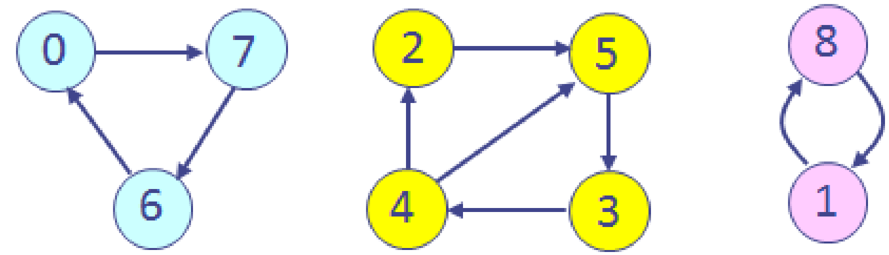
9.3.3 강연결성분(Strongly Connected Component)

- **방향그래프에서 연결성분 내의 임의의 두 정점 u와 v에 대해 정점 u에서 v로가는 경로가 존재하고 동시에 v에서 u로 돌아오는 경로가 존재하는 연결성분**
  - 강연결성분은 단절정점이나 다리간선을 포함하지 않음.
  - 강연결성분은 소셜네트워크에서 커뮤니티(Community)를 분석하는데 활용되며, 인터넷의 웹 페이지 분석에도 사용.

강연결성분을 찾는 알고리즘



그래프



3개의 강연결성분

- **스택을 사용하는 Tarjan의 알고리즘**
- **역방향그래프를 활용하는 Kosaraju의 알고리즘**

[핵심 아이디어] **입력그래프의 각각의 강연결성분은 역방향그래프에서도 각각 동일한 강연결성분이다. 입력그래프의 위상정렬순서로 역방향그래프에서 DFS를 수행하면서 하나의 강연결성분에서 다른 강연결성분으로 진행할 수 없다.**

Kosaraju 알고리즘

[1] DFS를 이용하여 위상정렬순서 S를 찾는다.

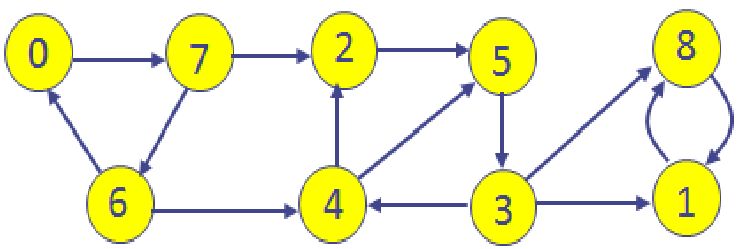
[2] 각 간선이 역방향으로 된 역방향그래프 G^R을 만든다.

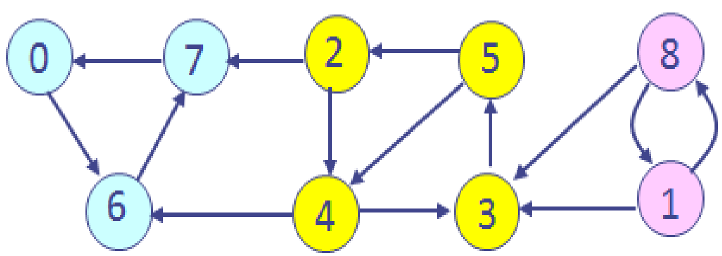
[3] S를 이용하여 G^R에서 DFS를 수행하면서 강연결성분들을 추출한다.

- Step [1]에서는 입력 그래프에서 DFS를 수행하면서 위상정렬순서 S를 찾음
- Step [2]에서는 역방향그래프 G^R을 만들고,
- Step [3]에서는 S를 이용하여 G^R에서 DFS를 수행하면서 강연결성분들을 추출
- G^R에서 각각의 강연결성분을 추출할 수 있는 이유는 S의 순서에 따라 G^R에서 DFS를 수행하면서 하나의 강연결성분의 정점들을 모두 방문한 후에 다른 강연결성분에 있는 정점을 방문할 수 없기 때문

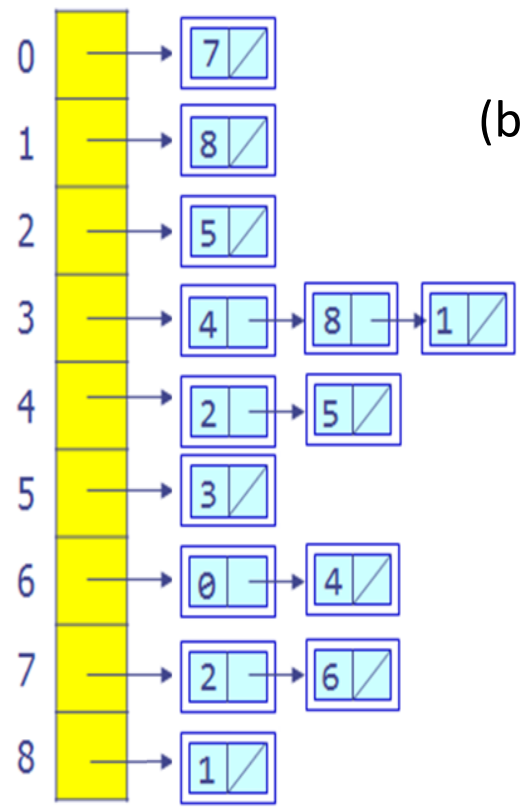
[예제] (a)의 그래프에 대해 step [1]에서 (b)의 인접리스트가 주어졌다고 가정하면, DFS를 정점 0으로부터 수행한 결과로 얻은 방문된 정점의 순서는 [4, 1, 8, 3, 5, 2, 6, 7, 0]이고, 이 순서의 역순인 [0, 7, 6, 2, 5, 3, 8, 1, 4] = 위상정렬순서 S

(a)





역방향 그래프 G^R

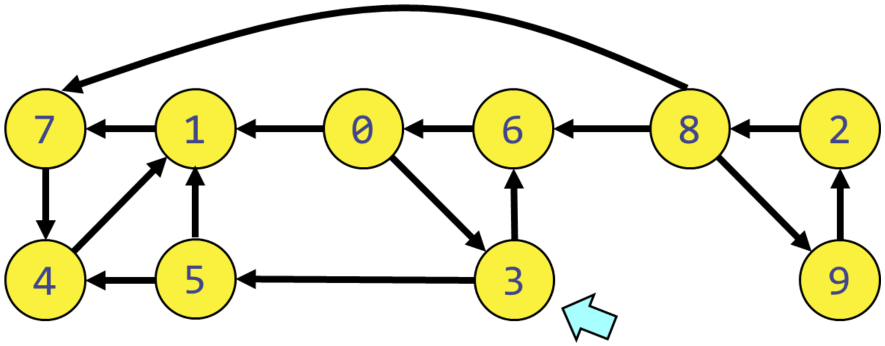


(b)

- Step [3]에서는 S = [0, 7, 6, 2, 5, 3, 8, 1, 4]의 순서에 따라서 GR에서 각각의 강연결성분을 추출
- 먼저 S의 첫 정점 0으로부터 DFS를 GR에서 수행하면, 정점 0, 6, 7을 방문한 후에 더 이상 다른 강연결성분에 있는 정점으로 이동할 수 없으므로 첫 번째 강연결성분인 [7, 6, 0]을 성공적으로 추출
- 그 다음엔 위상정렬순서 [0, 7, 6, 2, 5, 3, 8, 1, 4] 중 아직 방문 안된 첫 정점인 2부터 DFS를 수행하여, [5, 3, 4, 2]를 두 번째 강연결성분으로 추출하며,
- 마지막으로 위상정렬순서 [0, 7, 6, 2, 5, 3, 8, 1, 4] 중에서 방문 안된 첫 정점인 8부터 DFS를 수행하여 [1, 8]을 추출
- 단, DFS는 임의의 정점에서 시작해도 무방
- 예를 들어, 정점 5부터 DFS를 시작하면 [2, 4, 1, 8, 3, 5, 0, 7, 6]을 찾고, 이 순서의 역순: [6, 7, 0, 5, 3, 8, 1, 4, 2]

[예제]

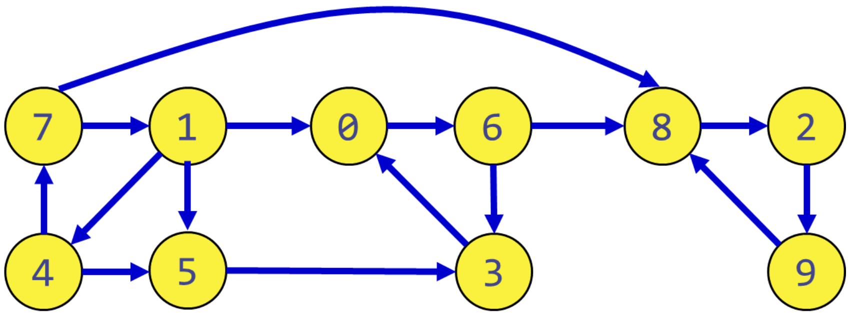
G



dfs(3) ⇨ [4 7 1 0 6 5 3 2 9 8]

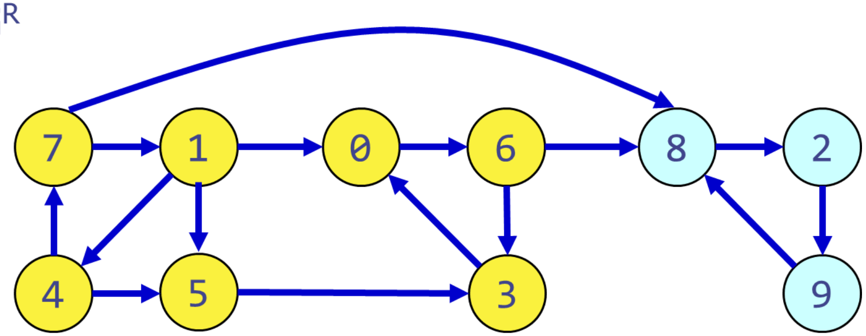
역순 S = [8 9 2 3 5 6 0 1 7 4]

G^R

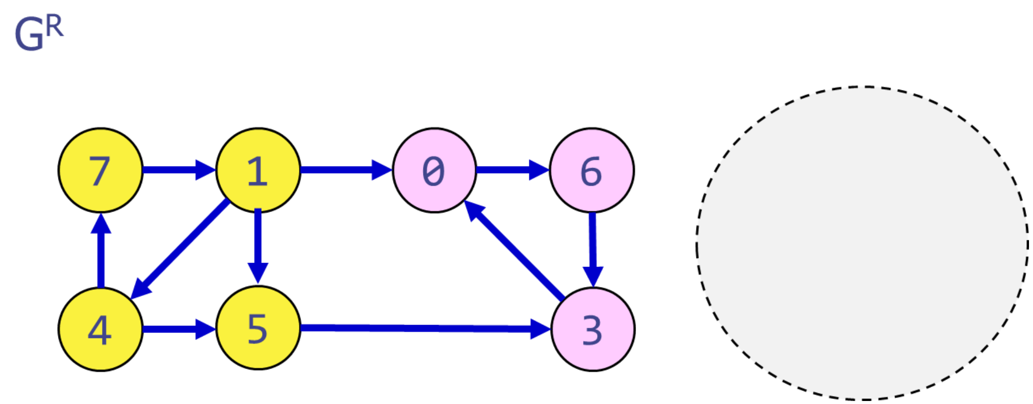


S = [8 9 2 3 5 6 0 1 7 4]

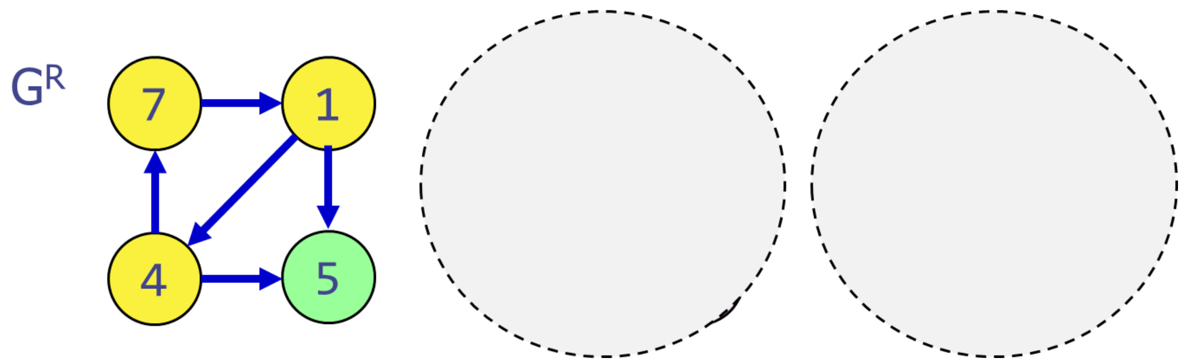
G^R



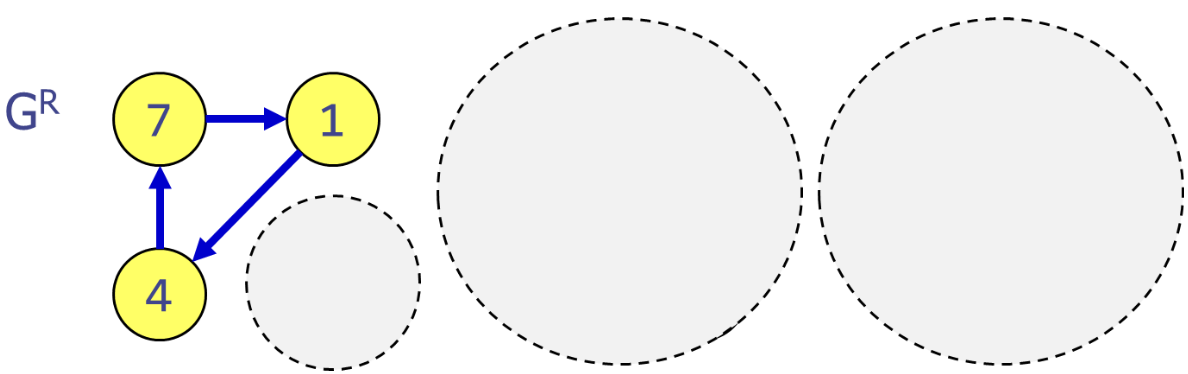
S = [~~8~~ ~~9~~ ~~2~~ 3 5 6 0 1 7 4]



S = [~~3~~ 5 ~~6 0~~ 1 7 4]

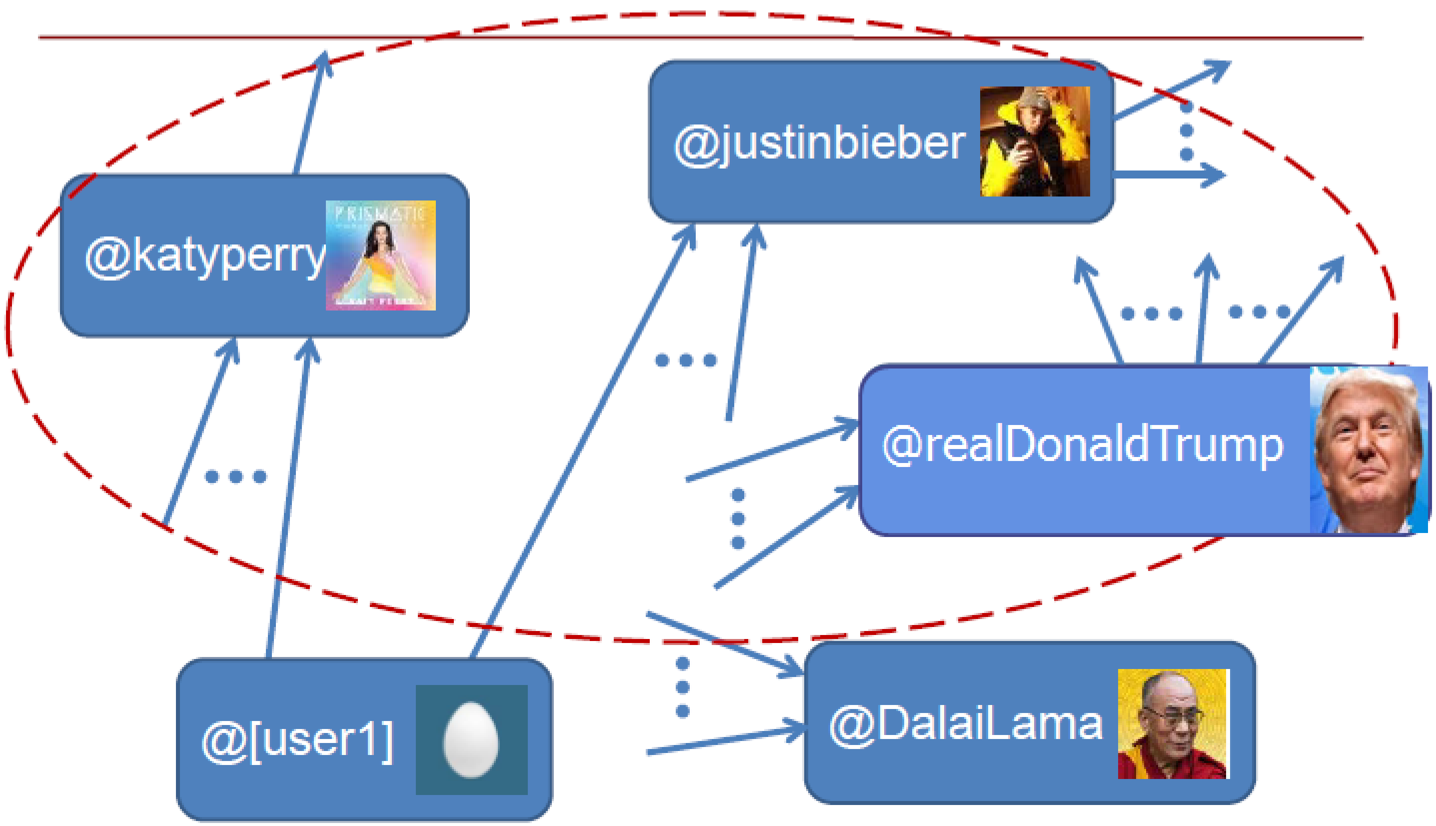


S = [~~5~~ 1 7 4]



S = ~~[1 7 4~~]

응용: 트위터 커뮤니티

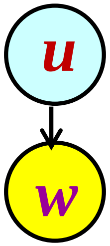


@justinbieber
@katyperry
@realDonaldTrump
@[user1]
@DalaiLama

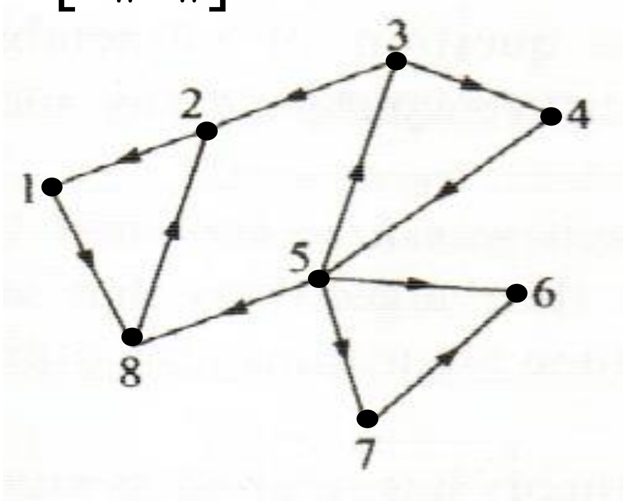
Tarjan 알고리즘

```
sequence = 1; 스택 초기화;
dfn[] = 0 // 초기화
while ( dfn[u]=0인 u가 있으면 )
    stronlyConnected(u);

stronglyConnected(u)
    dfn[u] = low[u] = sequence++;
    u 를 스택에 push
    for (each vertex w adjacent to u) {
        if (dfn[w]=0)    // w가 방문 안된 정점이면
            stronglyConnected(w)
            low[u] = min(low[u], low[w]) // w가 u의 자식이면
        else if (dfn(w) < dfn(u) && w가 스택에 있으면)
            low[u] = min(low[u], dfn[w]) }
    if (low[u]=dfn[u]) // 강연결성분 출력
        정점 u가 나올때까지 pop하고 출력
```



[예제]



strongly<u>Connected</u>(***u***)

dfn[***u***] = *low*[***u***] = sequence++;

u 를 스택에 push

for (each vertex *w* adjacent to ***u***) {

if (*dfn*[***w***]=0) // ***w***가 방문 안된 정점이면

strongly<u>Connected</u>(***w***)

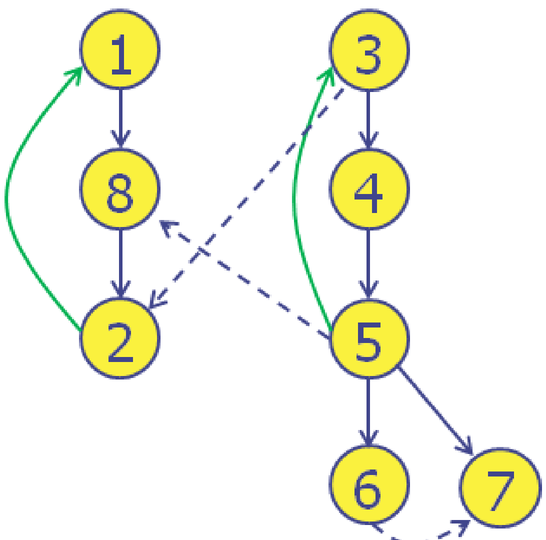
low[***u***] = min(*low*[***u***], *low*[***w***]) // ***w***가 ***u***의 자식이면

else if (*dfn*(***w***) < *dfn*(***u***) && ***w***가 스택에 있으면)

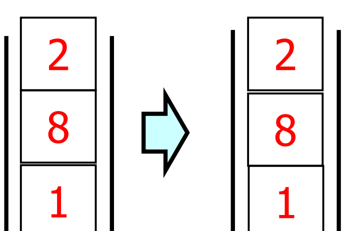
low[***u***] = min(*low*[***u***], *dfn*[***w***]) }

if (*low*[***u***]=*dfn*[***u***]) // 강연결성분 출력

정점 ***u***가 나올때까지 pop하고 출력

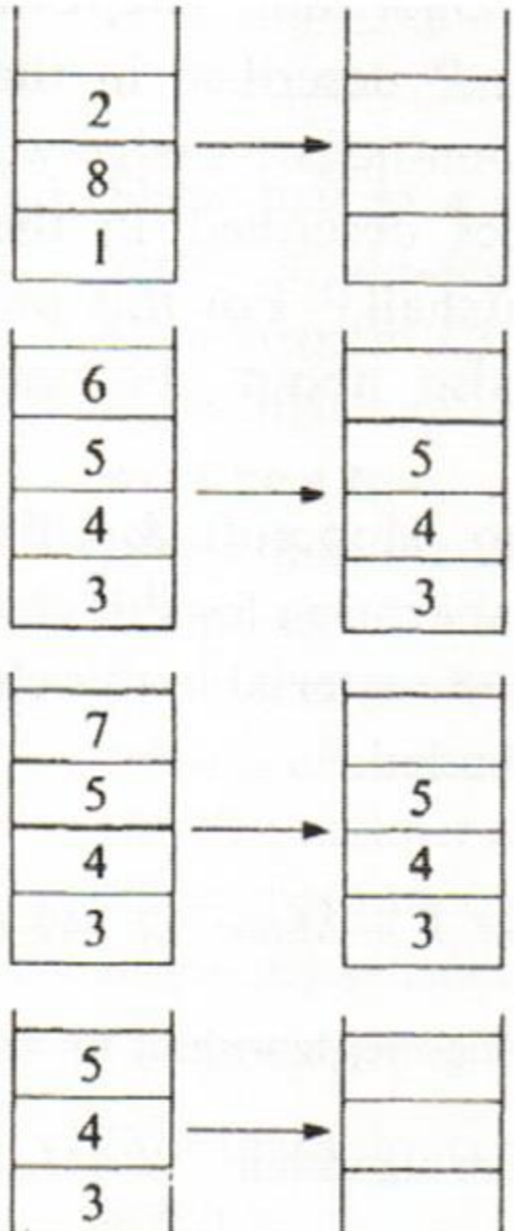


vertex	1	2	3	4	5	6	7	8
dfn	1	3	4	5	6	7	8	2
low	1	1	4	4	4	7	8	1



스택

강연결성분

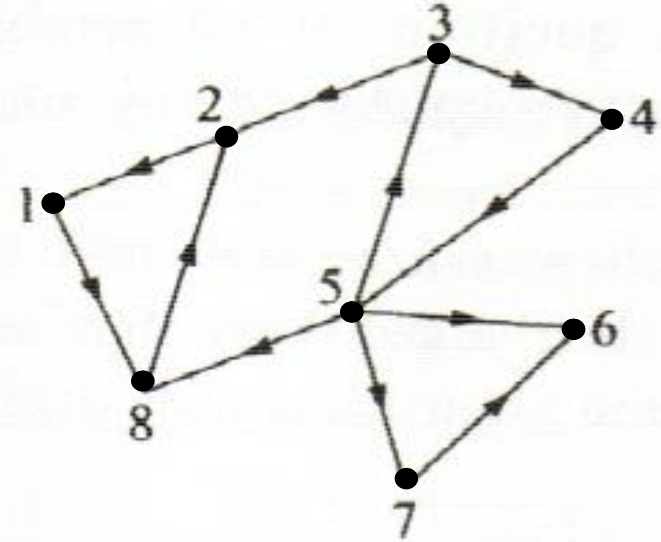


[1, 2, 8]

[6]

[7]

[3, 4, 5]



수행시간

- 깊이우선탐색을 2회 수행하고, 이와 별도로 소요되는 시간은 역방향그래프를 만드는데 소요되는 시간인 O(M)
- 따라서 강연결성분 알고리즘의 수행시간은 O(N+M) + O(M) = O(N+M)
- [참고] Knuth의 강연결성분 알고리즘의 수행시간도 O(N+M), 깊이우선탐색을 1회 수행하나 알고리즘이 Kosaraju 알고리즘에 비해 복잡

요약

- 이중연결성분 알고리즘은DFS를 수행하면서 사용하는 간선을 스택에 저장하고 뒷간선을 적절히 사용하여 단절정점을 찾으며, 단절정점을 찾은 직후에 스택에서 이중연결성분에 속한 간선들을 모두 꺼내어 출력하여 이중연결성분을 찾음
- Kosaraju의 강연결성분 알고리즘은 입력그래프에서 위상정렬순서를 찾고 역방향그래프에서 위상정렬순서에 따라 DFS를 수행하며 강연결성분들을 추출

- Bellman-Ford 알고리즘은 음수가중치 그래프에서도 최단경로를 찾을 수 있다. 각 정점에 대한 간선완화를 N-1번 수행
- Floyd-Warshall 알고리즘은 모든 정점 쌍 사이의 최단경로를 찾는 알고리즘이다. 입력그래프의 정점들을 0, 1, 2, …, N-1로 ID를 부여하고, 정점 ID를 증가시키며 간선완화를 수행
- 소셜네트워크분석에서 정점(사용자)의 중요도를 분석하기 위한 대표적인 방법에는 차수중심성, 중개중심성, 근접중심성, 고유벡터중심성이 있고, 커뮤니티를 분석하는 대표적인 방법에는 강연결성분과 계층적 클러스터링 방법이 있다.