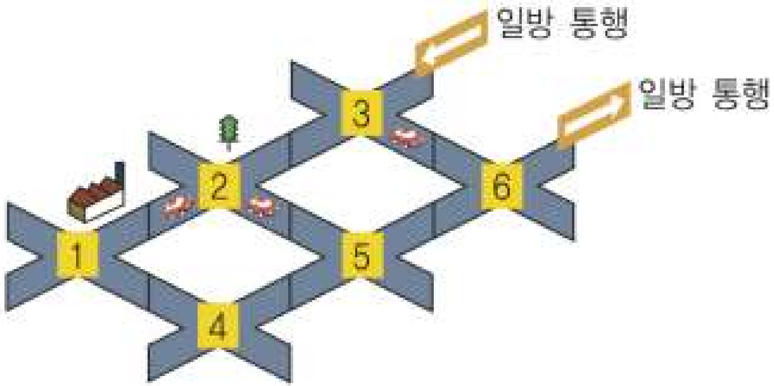
12주차 그 래 프

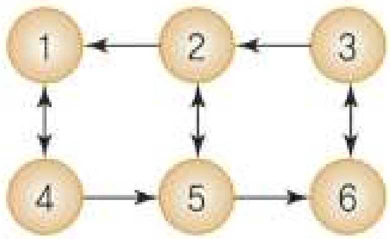
❖ 그래프(Graph)

- 연결되어 있는 객체 간의 관계를 표현하는 자료구조
  – (예) 우리가 배운 트리(tree)도 그래프의 특수한 경우임
  – (예) 전기회로의 소자 간 연결 상태
  – (예) 지도에서 도시들의 연결 상태.

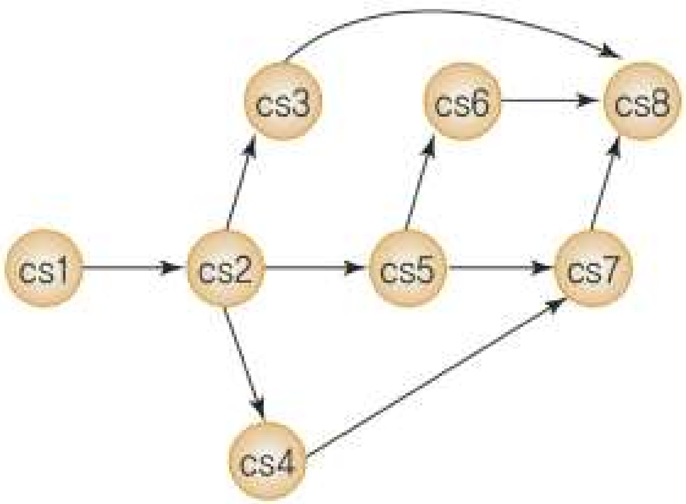
❖ 그래프로 표현하는 것들

- 도로망





- 선수과목 관계



내 용

- 그래프 용어
- 깊이우선탐색(DFS)
- 너비우선탐색(BFS)
- 연결성분(Connected Components)
- 이중연결성분(Doubly Connected Components)
- 강연결성분(Strongly Connected Components)
- 위상정렬(Topological Sort)
- 최소신장트리(Minimum Spanning Tree)
- 최단경로(Shortest Paths)

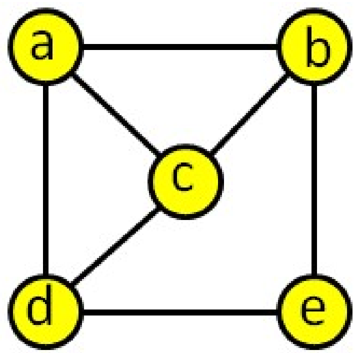
❖ 그래프

➢ 그래프 용어

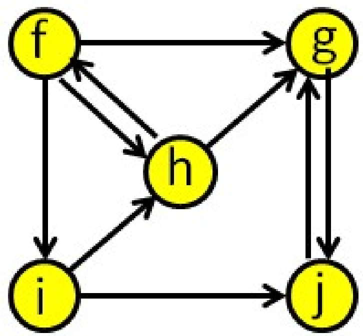
- 그래프는 정점(Vertex)과 간선(Edge)의 집합으로 하나의 간선은 두 개의 정점을 연결
- 그래프는 G=(V, E)로 표현, V=정점의 집합, E=간선의 집합
- 방향 그래프(Directed Graph): 간선에 방향이 있는 그래프
- 무방향 그래프(Undirected Graph): 간선에 방향이 없는 그래프

(a) 무방향그래프　　　(b) 방향 그래프

- 정점 a와 b를 연결하는 간선을 (a, b)로 표현
- 정점 a에서 b로 간선의 방향이 있는 경우 <a, b>로 표현
- 차수(Degree): 정점에 인접한 정점의 수
- 방향 그래프에서는 차수를 진입 차수(In-degree)와 진출 차수(Out-degree)로 구분
- 그림(a) 정점 a의 차수 = 3, 정점 e의 차수 = 2.
- 그림(b) 정점 g의 진입 차수 = 3, 진출 차수 = 1.

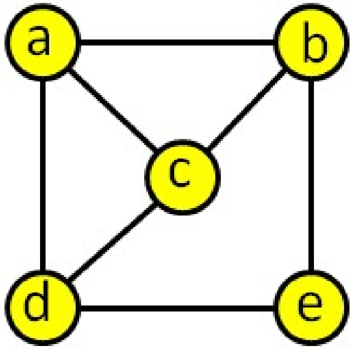


(a) 무방향그래프

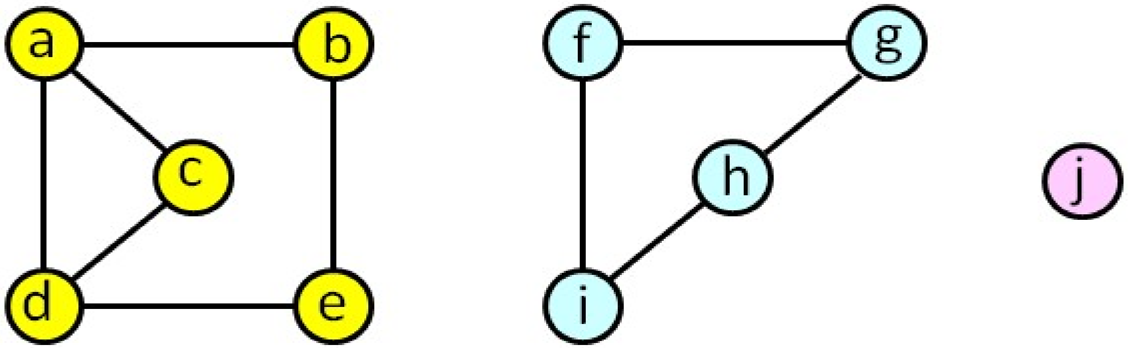


(b) 방향 그래프

- 경로(Path)는 시작 정점 u부터 도착점 e 까지의 정점들을 나열하여 표현
    - [a, c, b, e]: 정점 a로부터 도착점 e까지의 여러 경로들 중 하나
- 단순 경로(Simple Path): 경로 상의 정점들이 모두 다른 경로
- '일반적인' 경로: 동일한 정점을 중복하여 방문하는 경우를 포함
    - [a, b, c, b, e]: 정점 a로부터 도착점 e까지의 경로
- 싸이클(Cycle): 시작 정점과 도착점이 동일한 단순 경로
    - [a, b, e, d, c, a]



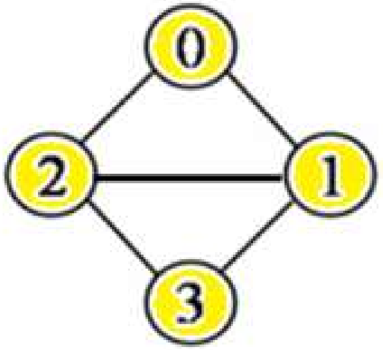
- 연결성분(Connected Component): 그래프에서 정점들이 서로 연결되어 있는 부분
- (예) 3개의 연결성분, [a, b, c, d, e], [f, g, h, i], [j]로 구성



- 가중치(Weighted) 그래프: 간선에 가중치가 부여된 그래프
  - 가중치는 두 정점 사이의 거리, 지나는 시간이 될 수도 있다. 또한 음수인 경우도 존재
- 부분그래프(Subgraph): 주어진 그래프의 정점과 간선의 일부분(집합)으로 이루어진 그래프
  - 부분그래프는 원래의 그래프에 없는 정점이나 간선을 포함하지 않음
- 트리(Tree): 싸이클이 없는 그래프
- 신장트리(Spanning Tree): 주어진 그래프가 하나의 연결성분으로 구성되어 있을 때, 그래프의 모든 정점들을 싸이클 없이 연결하는 부분그래프

➢ 그래프 자료구조

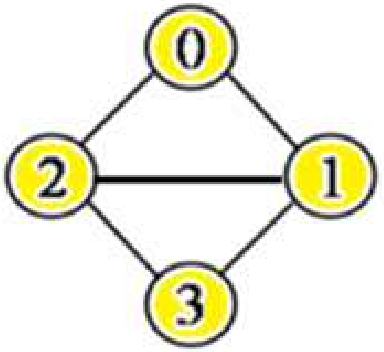
- 그래프를 자료구조로서 저장하는 방법
  - 인접 행렬(Adjacency Matrix)
  - 인접 리스트(Adjacency List)
- N개의 정점을 가진 그래프의 인접 행렬은 2차원 NxN 리스트에 저장
- 리스트가 a라면, 정점들을 0, 1, 2,…, N-1로 하여, 정점 i와 j 사이에 간선이 없으면 a[i][j] = 0, 간선이 있으면 a[i][j] = 1로 표현
- 가중치 그래프는 1 대신 가중치 저장



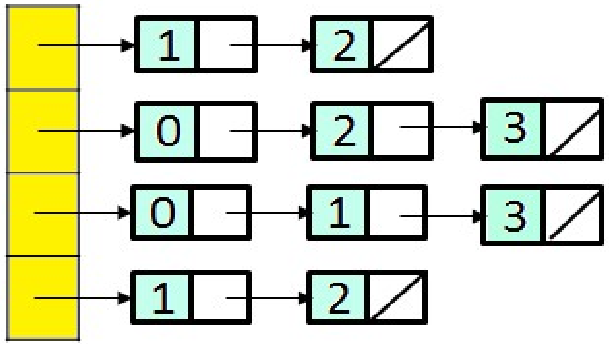
인접 행렬

$$
\begin{array}{c}
\\ 0 \\ 1 \\ 2 \\ 3
\end{array}
\begin{array}{c}
\begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\
\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}
\end{array}
$$

- 인접 리스트는 각 정점마다 1 개의 단순연결리스트를 이용하여 인접한 각 정점을 노드에 저장



인접 리스트



인접 리스트와 인접 행렬 혼합

```
adj = [[1, 2], [0, 2, 3], [0, 1 ,3], [1, 2]]
```

- 실세계의 그래프는 대부분 정점의 평균 차수가 작은 희소 그래프(Sparse Graph)이다.
- 희소그래프의 간선 수는 최대 간선 수인 N(N-1)/2보다 훨씬 작으므로 인접리스트에 저장하는 것이 매우 적절
  - 무방향 그래프를 인접리스트를 사용하여 저장할 경우 간선 1 개당 2개의 Edge 객체를 저장하고, 방향 그래프의 경우 간선 1 개당 1개의 Edge 객체만 저장하기 때문
- 조밀 그래프(Dense Graph): 간선의 수가 최대 간선 수에 근접한 그래프

▪ 그래프 탐색

- 그래프에서는 두 가지 방식으로 모든 정점을 방문
- 깊이우선탐색(DFS; Depth First Search)
- 너비우선탐색(BFS; Breadth First)

➢ 깊이우선탐색(DFS)

[핵심 아이디어] DFS는 실타래를 가지고 미로에서 출구를 찾는 것과 유사하다. 새로운 곳으로 갈 때는 실타래를 풀면서 진행하고, 길이 막혀 진행할 수 없을 때에는 실타래를 되감으며 왔던 길을 되돌아가 같은 방법으로 다른 경로를 탐색하여 출구를 찾는다.

- 그래프에서의 DFS는 임의의 정점에서 시작하여 이웃하는 하나의 정점을 방문하고,
- 방금 방문한 정점의 이웃 정점을 방문하며,
- 이웃하는 정점들을 모두 방문한 경우에는 이전 정점으로 되돌아 가서 탐색을 수행하는 방식으로 진행

```python
adj_list = [[2, 1], [3, 0], [3, 0], [9, 8, 2, 1],
            [5], [7, 6, 4], [7, 5], [6, 5], [3], [3]]
N = len(adj_list)
visited = [None] * N

def dfs(v):
    visited[v] = True
    print(v, ' ', end='')
    for w in adj_list[v]:
        if not visited[w]:
            dfs(w)

print('DFS 방문 순서:')
for i in range(N):
    if not visited[i]:
        dfs(i)
```

그래프 인접리스트

정점 방문 여부 확인 용

정점 v 방문

정점 v에 인접한 정점으로 dfs() 재귀호출

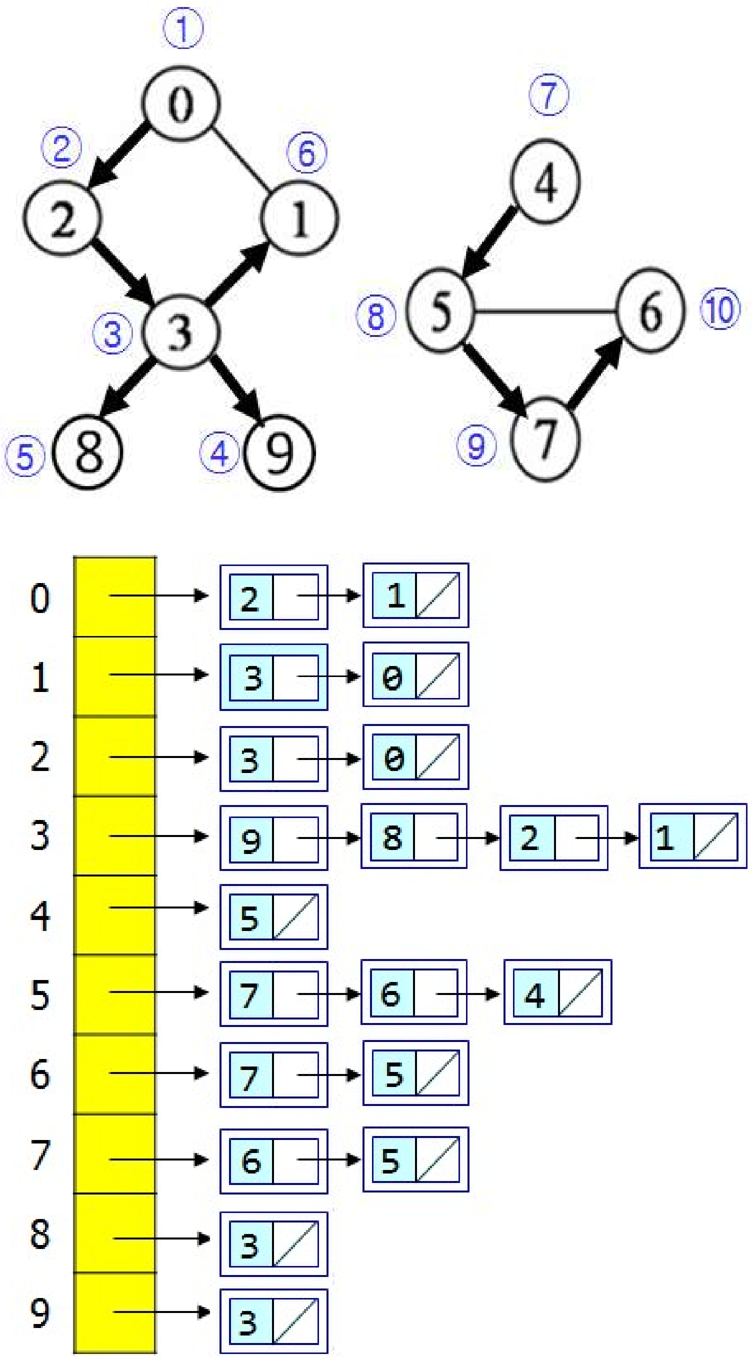
dfs() 호출

[프로그램 8-1] dfs.py

Console PyUnit

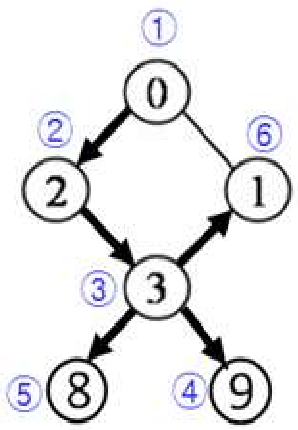
```
<terminated> dfs.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32
DFS 방문 순서:
0  2  3  9  8  1  4  5  7  6
```

DFS 수행 과정

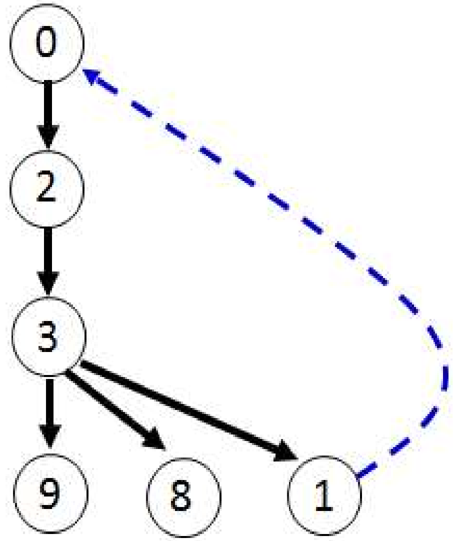


방문순서	dfs()호출	visited[]	출력
①	dfs(0)	visited[0] = True	0
②	dfs(2)	visited[2] = True	2
③	dfs(3)	visited[3] = True	3
④	dfs(9)	visited[9] = True	9
⑤	dfs(8)	visited[8] = True	8
⑥	dfs(1)	visited[1] = True	1
⑦	dfs(4)	visited[4] = True	4
⑧	dfs(5)	visited[5] = True	5
⑨	dfs(7)	visited[7] = True	7
⑩	dfs(6)	visited[6] = True	6

- (a)의 DFS 방문순서대로 정점 0부터 위에서 아래방향으로 정점들을 그리면 (b)와 같은 트리가 만들어진다.
- 1개의 연결성분인 그래프에서 DFS를 수행하며 만들어지는 트리를 깊이우선 신장 트리(Depth First Spanning Tree)라고 한다.



(a)



(b)

수행 시간

- DFS의 수행 시간은 탐색이 각 정점을 한번씩 방문하며, 각 간선을 한번씩만 사용하여 탐색하기 때문에 O(N+M)
- N은 그래프의 정점의 수이고, M은 간선의 수

➢ 너비우선탐색

[핵심 아이디어] BFS는 연못에 돌을 던져서 만들어지는 동심원의 물결이 퍼져나가는 것 같이 정점들을 방문한다.

- BFS는 임의의 정점 s에서 시작하여 s의 모든 이웃하는 정점들을 방문하고, 방문한 정점들의 이웃 정점들을 모두 방문하는 방식으로 그래프의 모든 정점을 방문
- BFS는 이진트리에서의 레벨순회와 유사

```python
adj_list = [[2, 1], [3, 0], [3, 0], [9, 8, 2, 1],
            [5], [7, 6, 4], [7, 5], [6, 5], [3], [3]]
N = len(adj_list)
visited = [None] * N

def bfs(i):
    queue = []
    visited[i] = True
    queue.append(i)
    while len(queue) != 0:
        v = queue.pop(0)
        print(v, ' ', end='')
        for w in adj_list[v]:
            if not visited[w]:
                visited[w] = True
                queue.append(w)

print('BFS 방문 순서:')
for i in range(N):
    if not visited[i]:
        bfs(i)
```

그래프 인접리스트

정점 방문 여부 확인 용

큐를 리스트로 구현

큐의 맨 앞에서 제거된 정점을 v가 참조하게 함

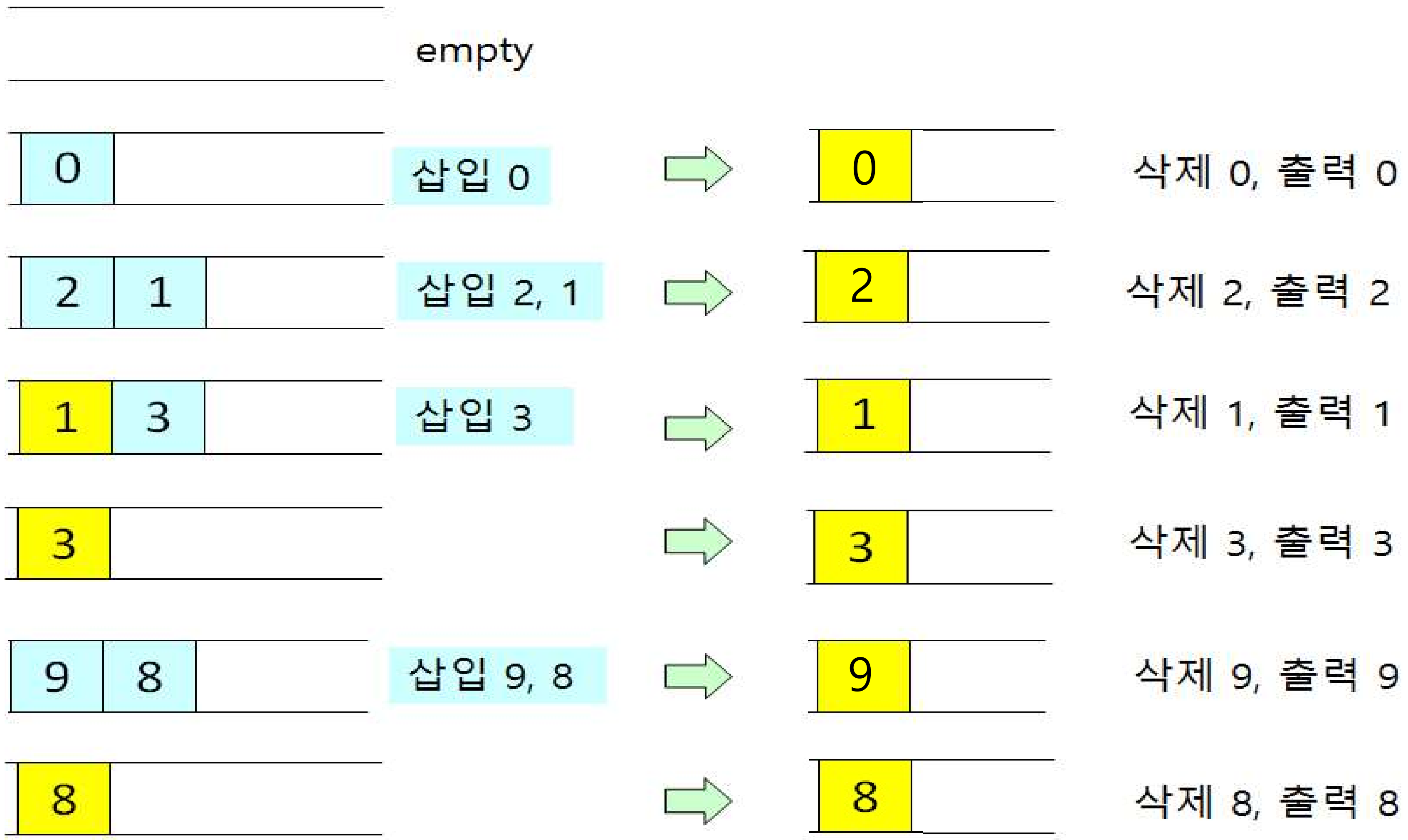
정점 v 방문

v에 인접하면서 방문 안된 정점 큐에 삽입

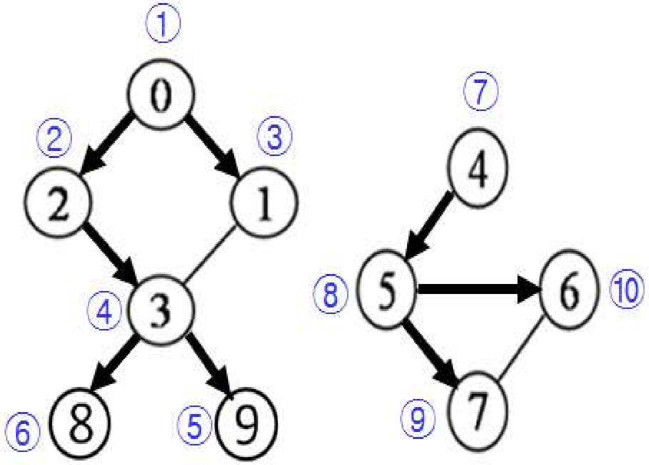
bfs() 호출

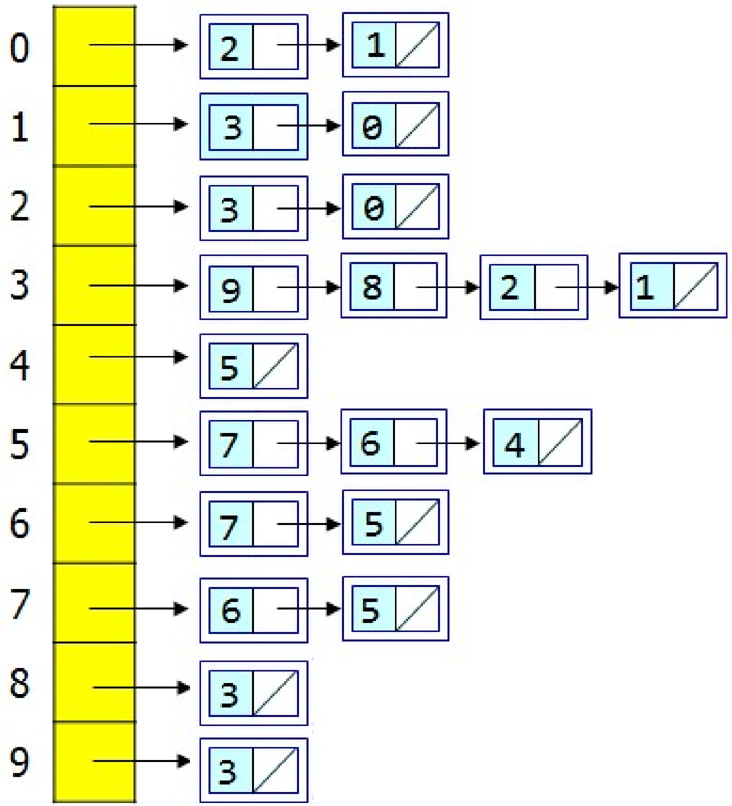
[프로그램 8-2] bfs.py

- bfs(0)부터 수행되며 첫 번째 연결성분의 정점들을 모두 방문할 때까지의 큐의 상태



BFS 수행 과정



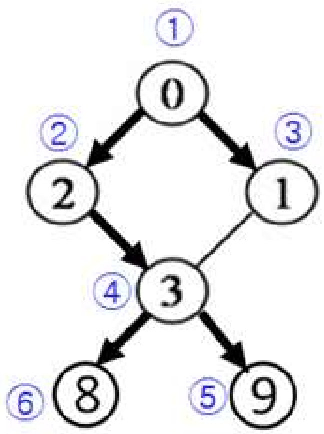


방문순서	visited[]	출력
①	visited[0] = True	0
②	visited[2] = True	2
③	visited[1] = True	1
④	visited[3] = True	3
⑤	visited[9] = True	9
⑥	visited[8] = True	8
⑦	visited[4] = True	4
⑧	visited[5] = True	5
⑨	visited[7] = True	7
⑩	visited[6] = True	6

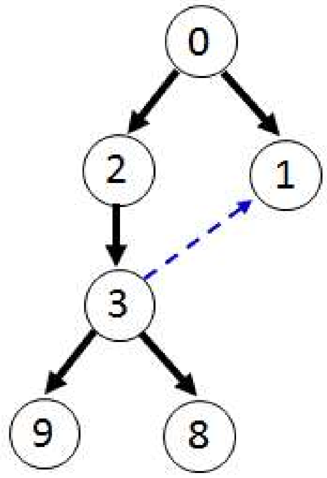
프로그램 수행 결과

```
Console  PyUnit
<terminated> bfs.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32
BFS 방문 순서:
0  2  1  3  9  8  4  5  7  6
```

- (a)의 그래프에서 BFS 방문순서대로 정점 0부터 위에서 아래방향으로 그려보면 (b)와 같은 트리가 만들어짐
- 그래프가 1개의 연결성분으로 되어 있을 때 BFS를 수행하며 만들어지는 트리: 너비우선 신장 트리 (Breadth First Spanning Tree)



(a)



(b)

수행 시간

- BFS는 각 정점을 한번씩 방문하며, 각 간선을 한 번씩만 사용하여 탐색하기 때문에 O(N+M)의 수행시간이 소요
- BFS와 DFS는 정점의 방문 순서나 간선을 사용하는 순서만 다를 뿐이다.

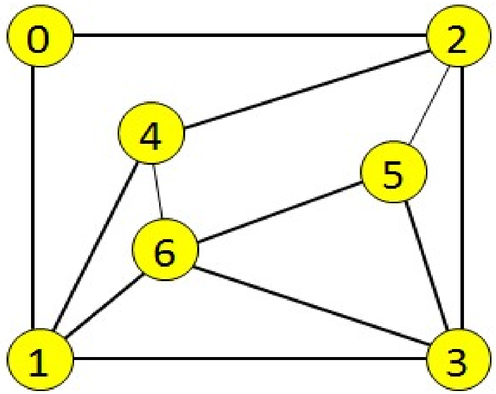
DFS와 BFS로 수행 가능한 그래프 응용

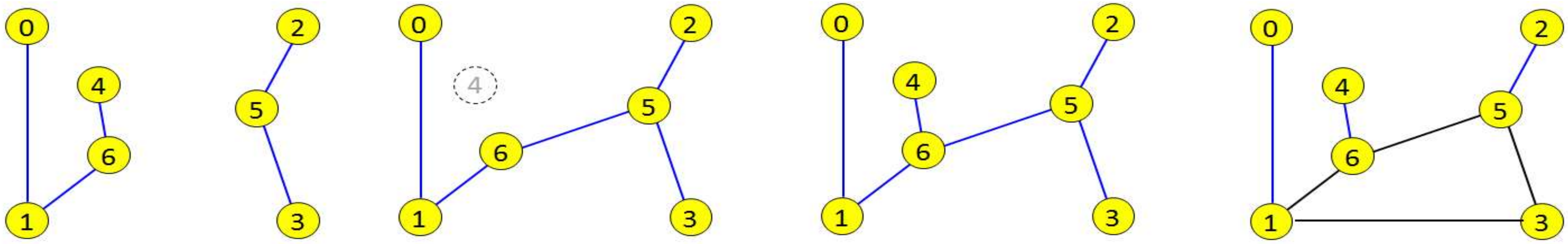
응용	DFS	BFS
신장트리, 연결성분, 경로, 싸이클	√	√
최소 선분을 사용하는 경로		√
위상 정렬, 이중 연결성분, 강연결성분	√	

최소신장트리

- 최소 신장 트리(Minimum Spanning Tree, MST): 하나의 연결성분으로 이루어진 무방향 가중치 그래프에서 간선의 가중치의 합이 최소인 신장 트리
- MST를 찾는 대표적인 알고리즘은 Kruskal, Prim, Sollin 알고리즘 - 모두 그리디 (Greedy) 알고리즘
- 그리디 알고리즘은 최적해(최솟값 또는 최댓값)를 찾는 문제를 해결하기 위한 알고리즘 방식들 중 하나로서, 알고리즘의 선택이 항상 '욕심내어' 지역적인 최솟값(또는 최댓값)을 선택하며, 이러한 부분적인 선택을 축적하여 최적해를 찾음

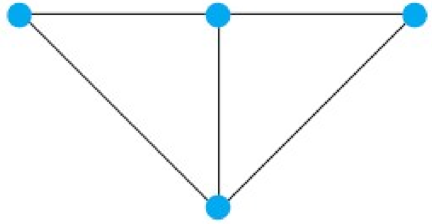
어느 그래프가 신장 트리일까?



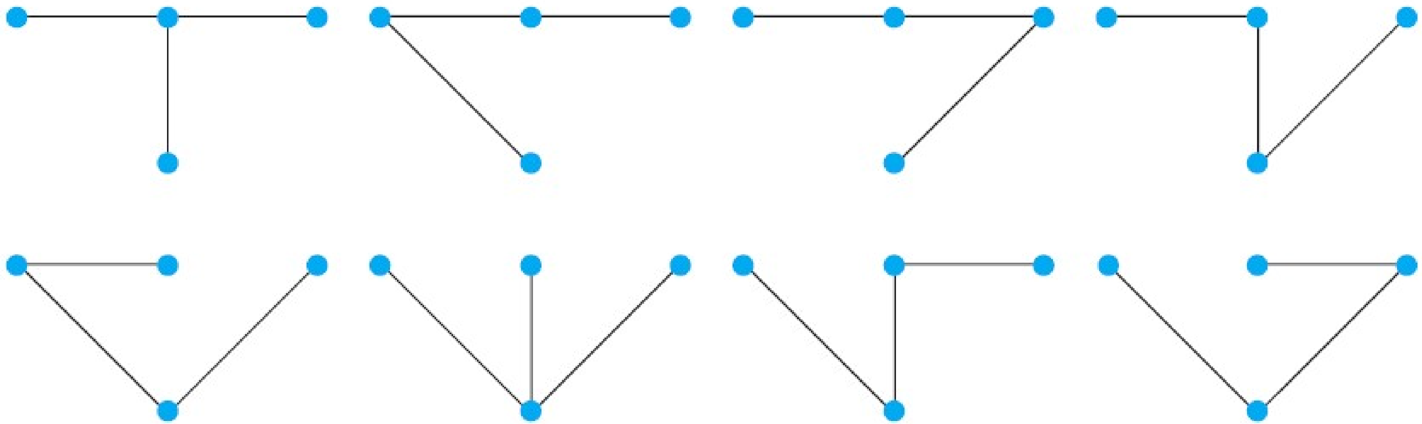


(a) (b) (c) (d)

다음과 같은 그래프 G의 생성 트리를 모두 구해보자.



풀이 G의 생성 트리는 모두 8가지인데 다음과 같다.



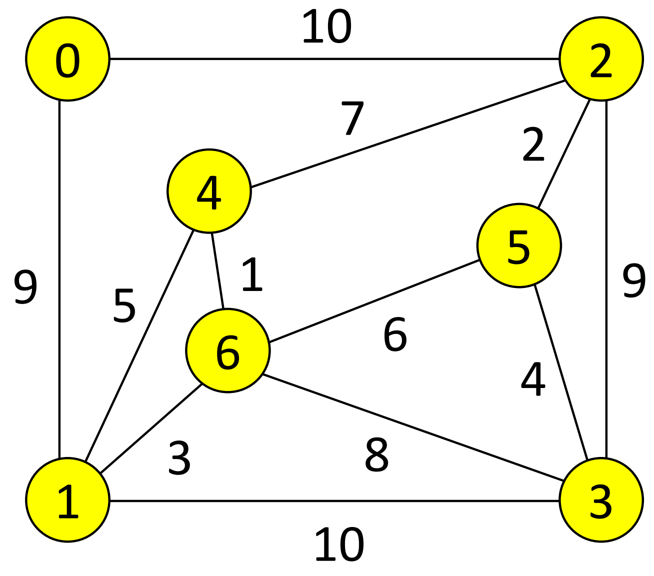
Kruskal 알고리즘

- 간선을 가중치가 감소하지 않는 순서로 정렬
- 가장 가중치가 작은 간선을 트리에 추가하여 사이클을 만들지 않으면 트리 간선으로 선택
- 사이클을 만들면 버리는 것을 반복
- n-1개의 간선이 선택되면 알고리즘 종료
- Kruskal 알고리즘이 그리디 알고리즘인 이유: 남아있는 (정렬된) 간선들 중에서 항상 '욕심 내어' 가중치가 가장 작은 간선 선택

Kruskal 알고리즘

```
가중치가 감소하지 않는 순서로 간선 리스트 L을 만든다.
while 트리의 간선 수 < N-1:
    L에서 가장 작은 가중치를 가진 간선 e를 가져오고, e를 L에서 제거
    if 간선 e가 T에 추가하여 싸이클을 만들지 않으면:
        간선 e를 T에 추가
```

[예제]

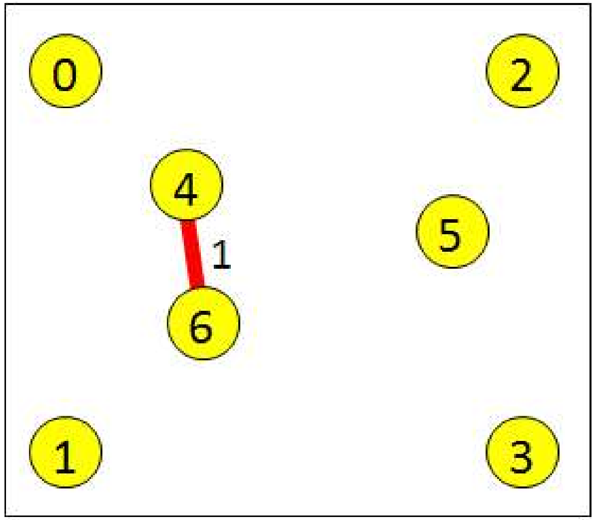


(0, 1)	9
(0, 2)	10
(1, 3)	10
(1, 4)	5
(1, 6)	3
(2, 3)	9
(2, 4)	7
(2, 5)	2
(3, 5)	4
(3, 6)	8
(4, 6)	1
(5, 6)	6

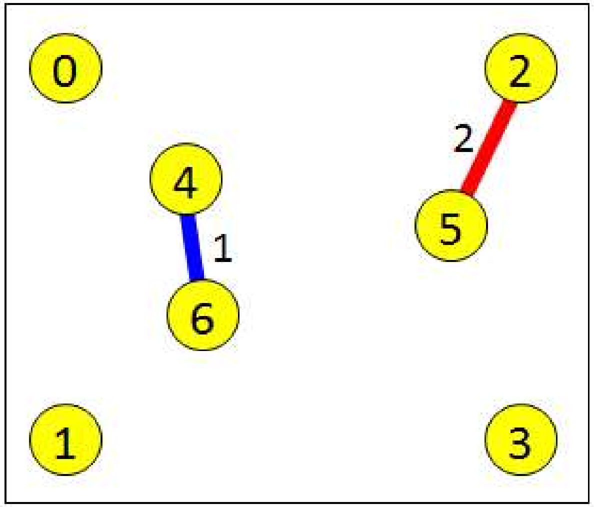
정렬된 L

(4, 6)	1
(2, 5)	2
(1, 6)	3
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10

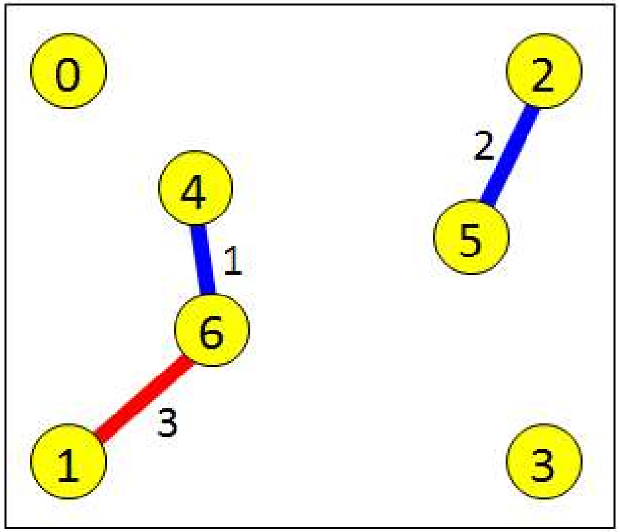
(4, 6)	1
(2, 5)	2
(1, 6)	3
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



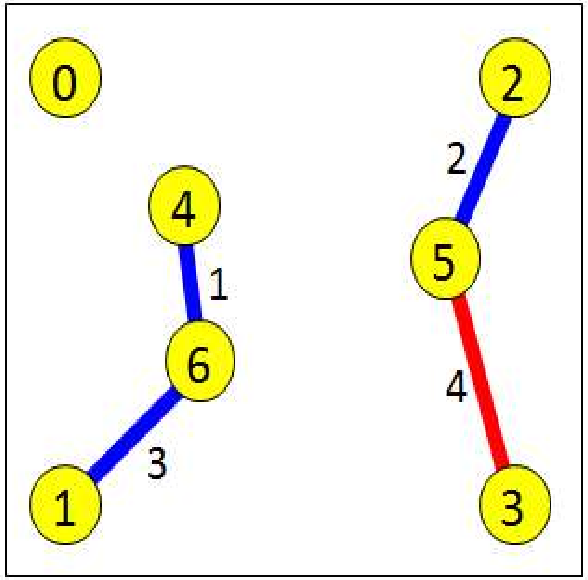
(2, 5)	2
(1, 6)	3
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



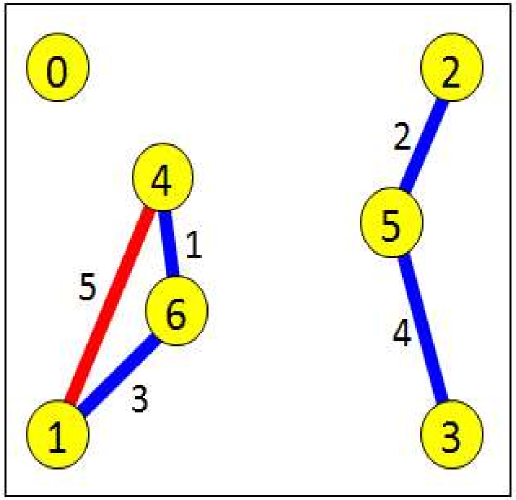
(1, 6)	3
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



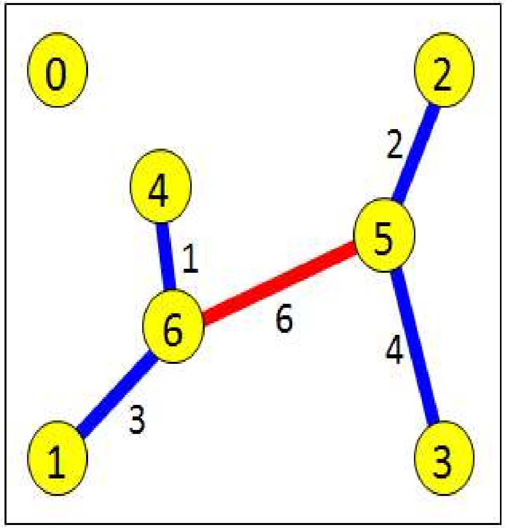
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



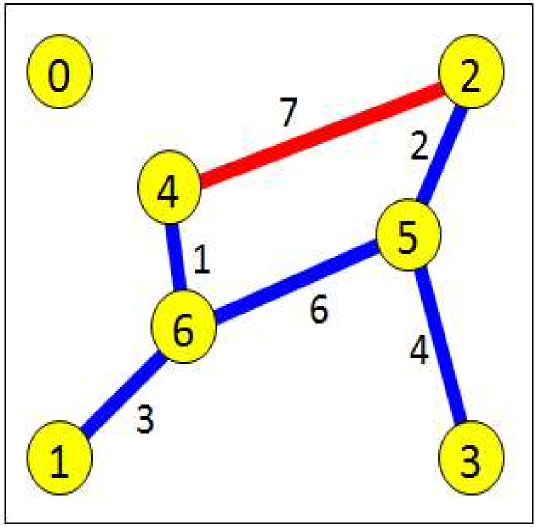
(1, 4) 5
(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



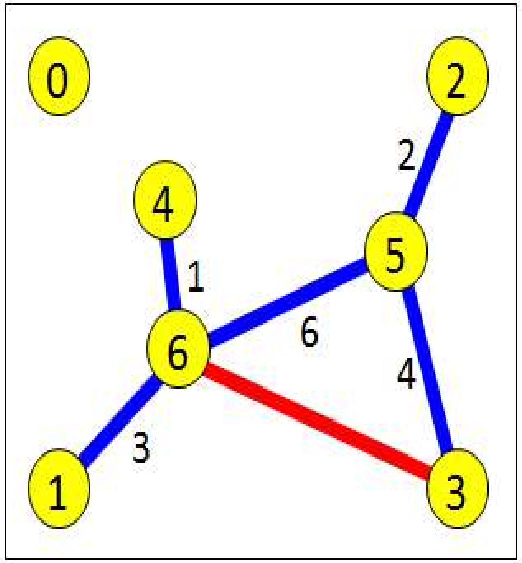
(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



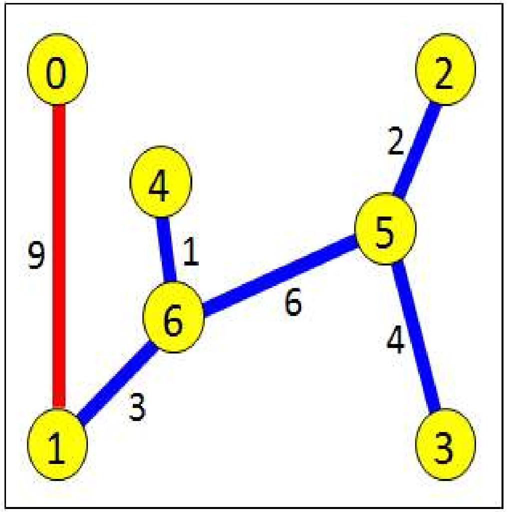
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10

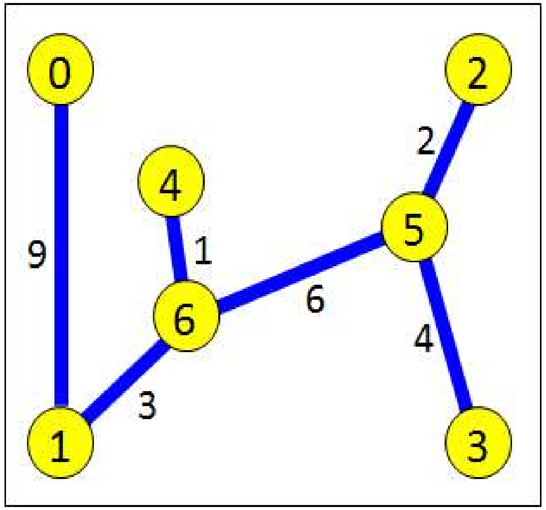


(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10





최소신장트리

최소신장트리의 간선의 가중치의 합 = 1 + 2 + 3 + 4 + 6 + 9 = 25

입력 그래프
(간선의 두 정점, 가중치)

```python
weights = [(0, 1, 9), (0, 2, 10), (1, 3, 10), (1, 4, 5),
           (1, 6, 3), (2, 3, 9), (2, 4, 7), (2, 5, 2),
           (3, 5, 4), (3, 6, 8), (4, 6, 1), (5, 6, 6)]
weights.sort(key = lambda t: t[2])
mst = []
N = 7
p = [] * N
for i in range(N):
    p.append(i)

def find(u):
    if u != p[u]:
        p[u] = find(p[u])
    return p[u]

def union(u, v):
    root1 = find(u)
    root2 = find(v)
    p[root2] = root1

```

가중치로 간선 정렬

서로소 집합

각 정점 자신이 집합의 대표(루트)

find 연산

경로압축

union 연산

임의로 root1가
root2의 부모가 됨

```python
tree_edges = 0
mst_cost = 0
while True:
    if tree_edges == N-1:
        break
    u, v, wt = weights.pop(0)
    if find(u) != find(v):
        union(u, v)
        mst.append((u, v))
        mst_cost += wt
        tree_edges += 1

print('최소 신장 트리: ', end='')
print(mst)
print('최소 신장 트리 가중치:', mst_cost)
```

다음 최소 가중치를 가진 간선 가져오기

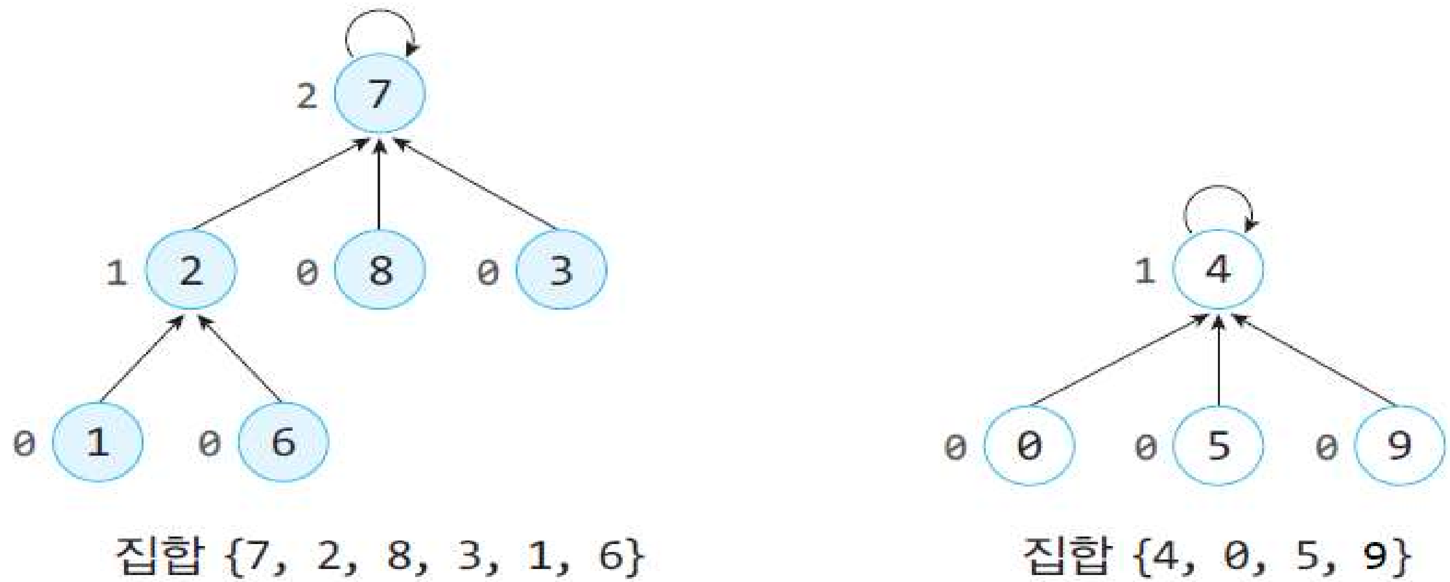
u와 v가 서로 다른 집합에 속해 있으면

트리에 (u, v) 추가

[프로그램 8-5] kruskal.py

- Kruskal 알고리즘에서 추가하려는 간선이 사이클을 만드는지의 여부는 집합과 관련된 연산인 union(합집합) 연산과 주어진 원소가 어느 집합에 속해 있는지를 찾는 find 연산을 사용
- 특히 어느 두 집합도 중복된 원소를 갖지 않는 경우, 이러한 집합들을 서로소 집합(Disjoint Set)이라고 한다.

- [그림 8-22]는 2개의 서로소 집합을 일반적인 트리 형태로 표현하여 리스트에 저장한 상태
- 여기서 각 집합은 루트가 대표하고, 루트의 리스트 원소에는 루트 자신을 저장하며, 루트가 아닌 노드의 원소에는 부모를 저장한다.



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

[그림 8-22]

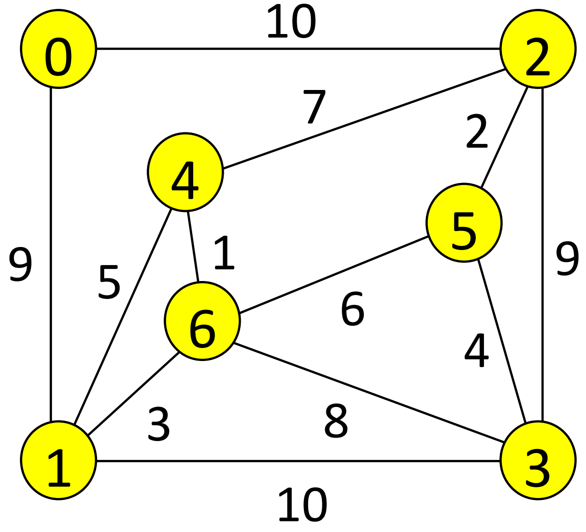
- union: 2개의 집합을 하나의 집합으로 만드는 연산

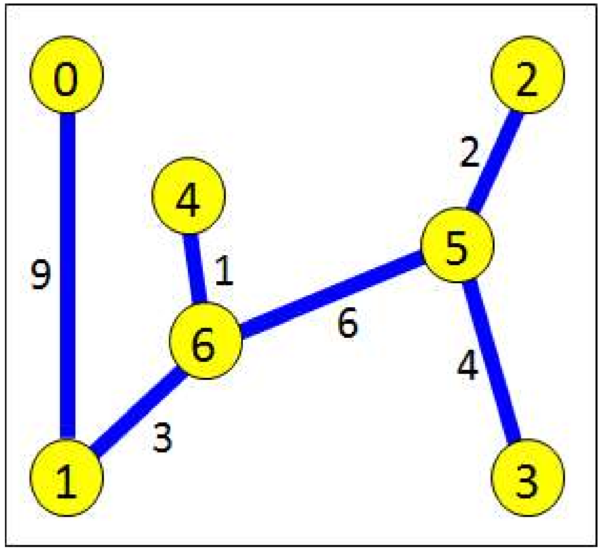
- find(x): x가 속한 집합의 대표 노드, 즉, 루트를 찾는 연산
- [예] find(6): p[6] = 2를 통해 6의 부모인 2를 찾고, p[2] = 7로 2의 부모를 찾으며, 마지막으로 p[7] = 7이기 때문에 7을 반환
  - 즉, "6은 7이 대표 노드인 집합에 속해 있다"

- find(3)도 7을 리턴하므로, 6과 3은 동일한 집합에 속함. 하지만 find(9) = 4이므로, 6과 9는 서로 다른 집합에 속함

프로그램 수행 결과

```
Console  PyUnit
<terminated> kruskal.py [C:\Users\sbyang\AppData\Local\Programs\Python\Python36-32
최소 신장 트리: [(4, 6), (2, 5), (1, 6), (3, 5), (5, 6), (0, 1)]
최소 신장 트리 가중치:  25
```





수행시간

- 간선을 정렬(또는 우선순위큐의 삽입과 삭제)하는데 소요되는 시간인 O(MlogM) = O(MlogN)과 트리에 간선을 추가하려 할 때 find와 union을 수행하는 시간인 O((M+N)log*N)의 합이다.
- 즉, O(MlogN) + O((M+N)log*M) = O(MlogN)이다.
- union 연산은 단순히 하나의 루트가 다른 루트의 자식이 되는 것이므로 O(1) 시간 소요

Prim알고리즘

- Prim 알고리즘은 임의의 시작 정점에서 가장 가까운 정점을 추가하여 간선이 하나의 트리를 만들고, 만들어진 트리에 인접한 가장 가까운 정점을 하나씩 추가하여 최소신장트리를 만든다.

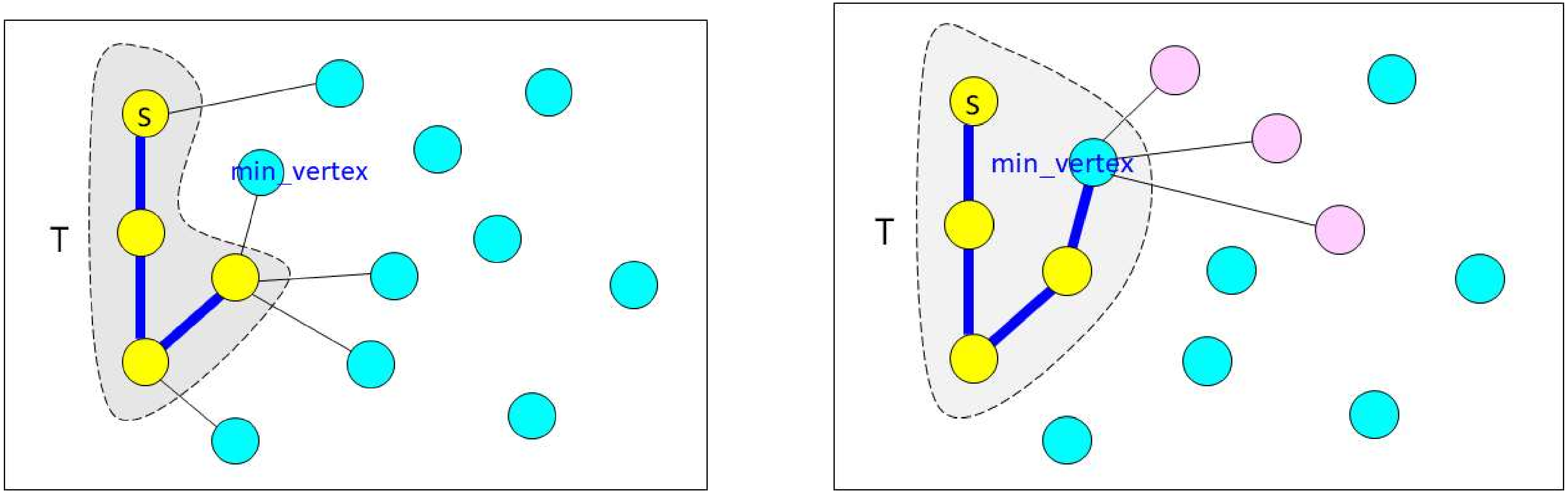
- Prim의 알고리즘에서는 초기에 트리 T는 임의의 정점 s만을 가지며, 트리에 속하지 않은 각 정점과 T의 정점(들)에 인접한 간선들 중에서 가장 작은 가중치를 가진 간선의 끝점을 찾기 위해 리스트 D를 사용

Prim 알고리즘

```
[1] D를 ∞로 초기화한다. 시작 정점 s의 D[s] = 0

[2] while T의 정점 수 < N:

[3]     T에 속하지 않은 각 정점 i에 대해 D[i]가 최소인 정점
          min_vertex를 찾아 T에 추가

[4]     for T에 속하지 않은 각 정점 w에 대해서:
[5]         if 간선 (min_vertex, w)의 가중치 < D[w]:
[6]             D[w] = 간선 (min_vertex, w)의 가중치
```

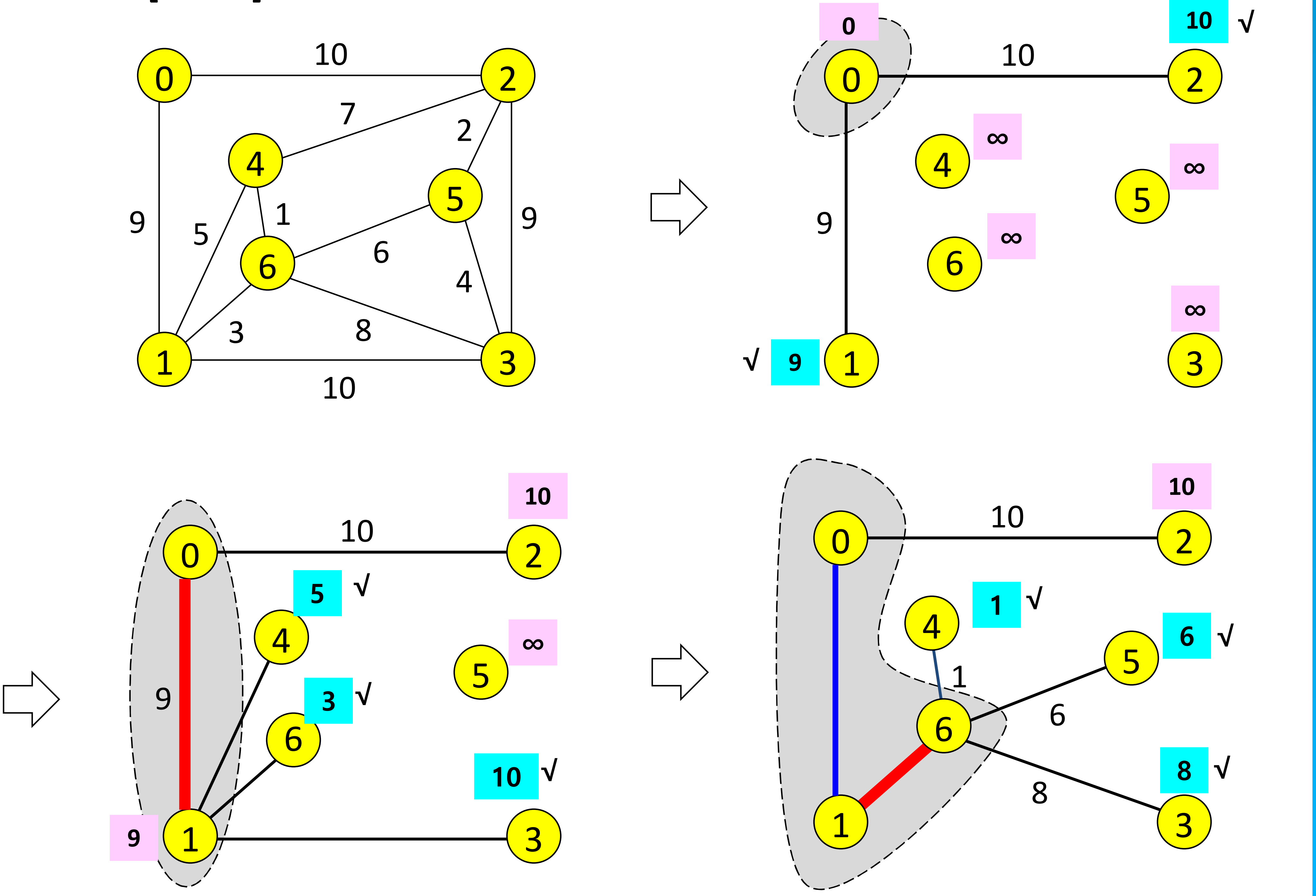
Prim 알고리즘의 step [3]~[6]

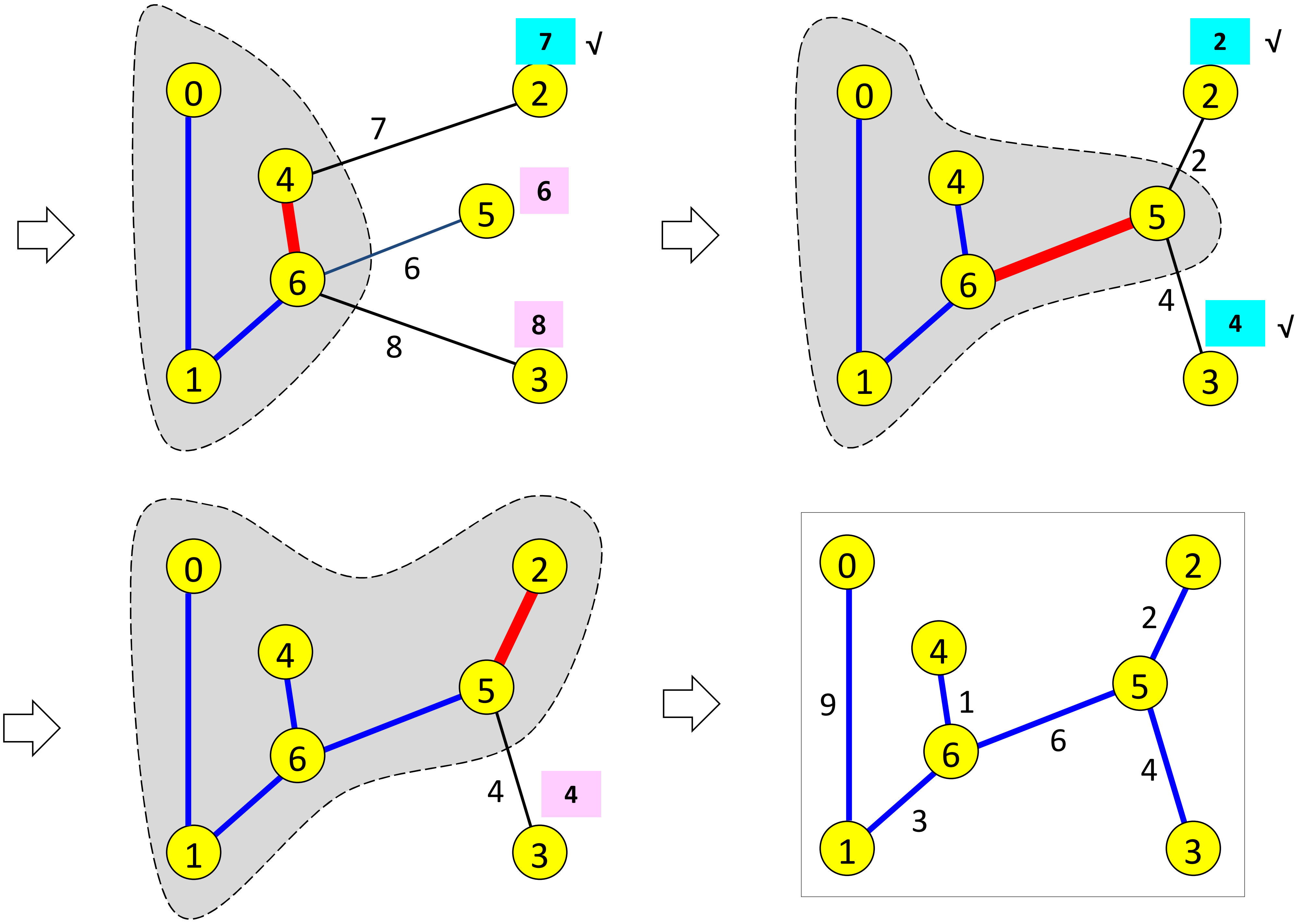
- (a) 트리에 가장 가까운 정점 min_vertex를 찾아(트리 밖에 있는 정점들의 D의 원소들 중에서 최솟값을 찾아)
- (b) 트리에 추가한 후, 정점 min_vertex에 인접하면서 트리에 속하지 않은 각 정점의 D 원소가 이전 값보다 작으면 갱신



(a) (b)

[예제]





7 √
6
8
2 √
4 √
4
9
1
6
3
2
4

```python
import sys
N = 7
s = 0
g = [None] * N
g[0] = [(1, 9), (2, 10)]
g[1] = [(0, 9), (3, 10), (4, 5), (6, 3)]
g[2] = [(0, 10), (3, 9), (4, 7), (5, 2)]
g[3] = [(1, 10), (2, 9), (5, 4), (6, 8)]
g[4] = [(1, 5),  (2, 7), (6, 1)]
g[5] = [(2, 2),  (3, 4), (6, 6)]
g[6] = [(1, 3),  (3, 8), (4, 1), (5, 6)]

visited = [False] * N
D = [sys.maxsize] * N
D[s] = 0
previous = [None] * N
previous[s] = s

```

sys.maxsize(최댓값) 사용 위해

입력 그래프의 인접리스트

각 원소를 최댓값으로

초기화

트리 간선 추출을 위해

```python
for k in range(N):
    m = -1
    min_value = sys.maxsize
    for j in range(N):
        if not visited[j] and D[j] < min_value:
            min_value = D[j]
            m = j
    visited[m] = True

    for w, wt in list(g[m]):
        if not visited[w]:
            if wt < D[w]:
                D[w] = wt
                previous[w] = m

print('최소신장트리: ', end='')
mst_cost = 0
for i in range(1,N):
    print('(%d, %d)'% (i, previous[i]), end='')
    mst_cost += D[i]
print('\n최소신장트리 가중치: ', mst_cost)
```

m = min_vertex

방문 안된 정점들의 D 원소들 중에서 최솟값을 가진 정점 m 찾기

정점 m에 인접한 정점 w와 간선 (m, w)의 가중치 wt에 대해

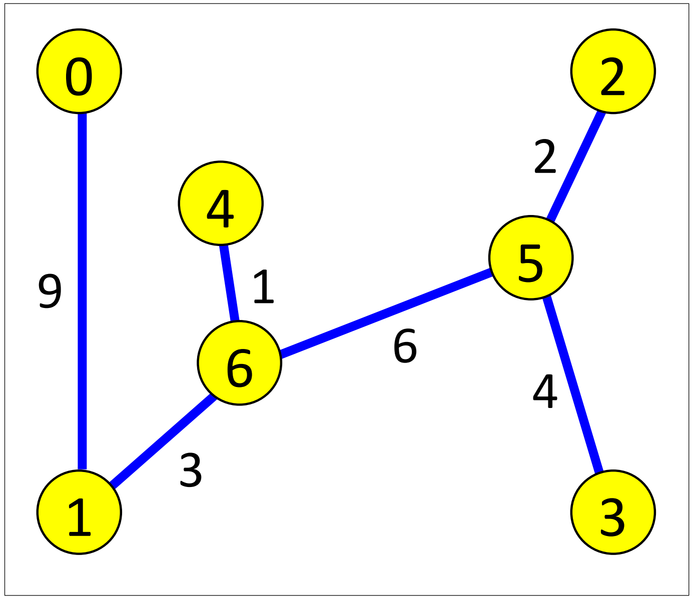
D[w] 갱신

D[w]가 정점 m 때문에 갱신되었음을 기록

[프로그램 8-6] prim.py

프로그램 수행 결과

```
Console ✕  PyUnit
<terminated> prim.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32
최소신장트리: (1, 0)(2, 5)(3, 5)(4, 6)(5, 6)(6, 1)
최소신장트리 가중치:  25
```



수행 시간(1)

- Prim 알고리즘은 N번의 반복을 통해 min_vertex를 찾고 min_vertex에 인접하면서 트리에 속하지 않은 정점에 해당하는 D의 원소 값을 갱신
- min_vertex를 배열 D에서 탐색하는 과정에서 O(N) 시간이 소요되고, min_vertex에 인접한 정점들을 검사하여 D의 해당 원소를 갱신하므로 O(N) 시간이 소요된다.
- 따라서 총 수행 시간은 Nx(O(N) +O(N)) = $O(N^2)$

수행 시간(2)

- min_vertex 찾기 위해 이진힙을 사용하면 각 간선에 대한 D의 원소를 갱신하며 힙 연산을 수행해야 하므로 총 O(MlogN) 시간이 필요, M은 그래프 간선의 수
- 이진힙은 각 정점에 대응되는 D원소를 저장하므로 힙의 최대 크기는 N
- 또한 가중치가 갱신되어 감소되었을 때의 힙 연산에는 O(logN) 시간이 소요
- 입력 그래프가 희소 그래프라면, 예를 들어, M = O(N)이라면, 수행시간이 O(MlogN) = O(NlogN)이 되어 이진힙을 사용하는 것이 매우 효율적

Sollin 알고리즘

- Sollin 알고리즘은 각 정점을 독립적인 트리로 간주하고, 각 트리에 연결된 간선들 중에서 가장 작은 가중치를 가진 간선을 선택한다. 이때 선택된 간선은 2 개의 트리를 1개의 트리로 만든다.
- 같은 방법으로 한 개의 트리가 남을 때까지 각 트리에서 최소 가중치 간선을 선택하여 연결
- Sollin 알고리즘은 병렬알고리즘(Parallel Algorithm)으로 구현이 쉽다는 장점을 가짐

Sollin 알고리즘

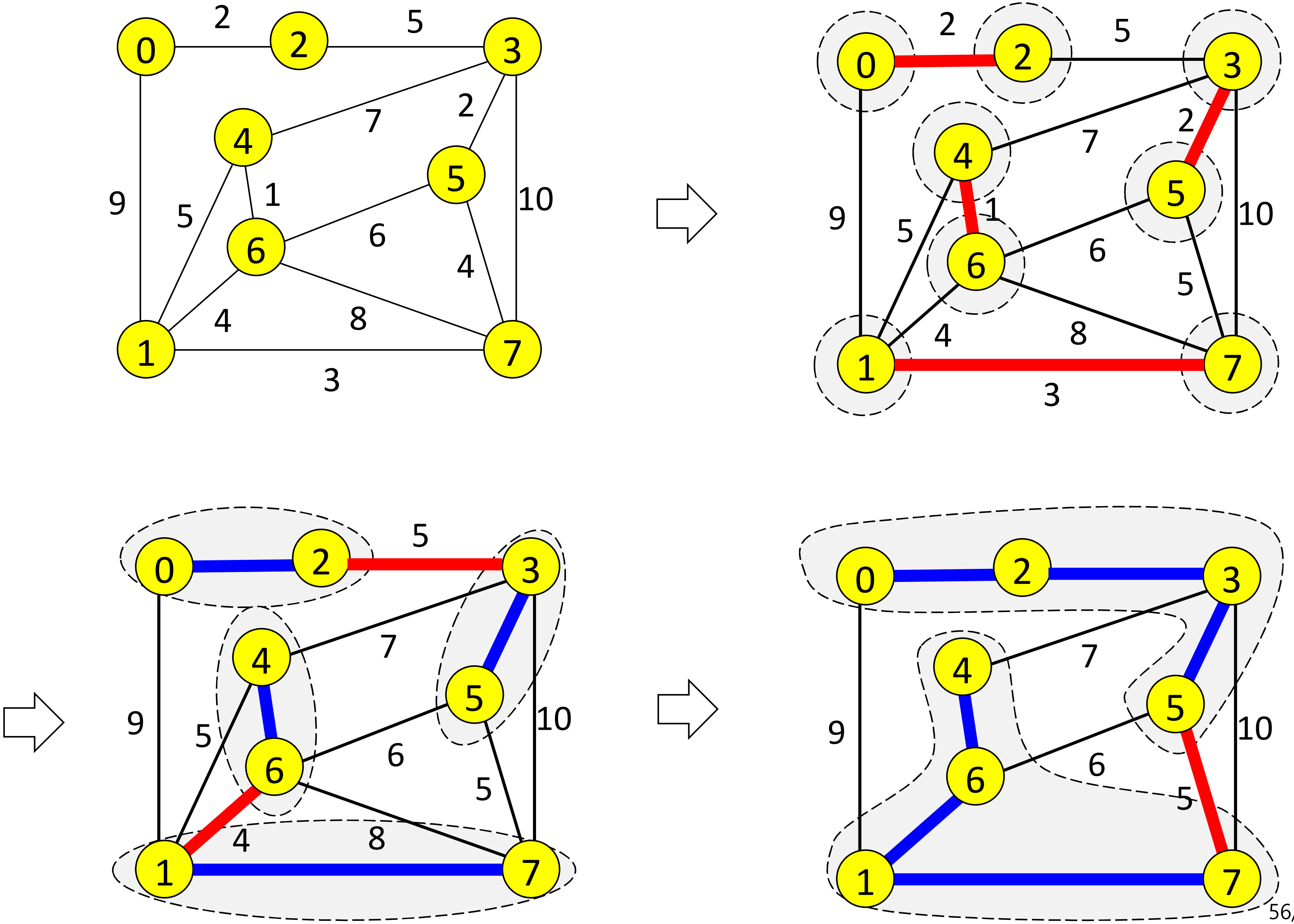
[1] 각 정점은 독립적인 트리이다.

[2] repeat

[3] 각 트리에 닿아 있는 간선들 중에서 가중치가 가장 작은 간선을 선택하여 트리를 합친다.

[4] until (1개의 트리만 남을 때까지)

[예제]


0 2 3 4 5 6 1 7
2 5 7 2 9 5 1 6 10 4 4 8 3
0 2 3 4 5 6 1 7
2 5 7 2 9 5 1 6 10 5 4 8 3
0 2 3 4 5 6 1 7
5 7 9 5 6 10 5 4 8
0 2 3 4 5 6 1 7
7 9 6 10 5

수행 시간

- Sollin 알고리즘에서 repeat-루프가 예제와 같이 각 쌍의 트리가 서로 연결된 간선을 선택하는 경우 최대 logN번 수행
- 루프 내에서는 각 트리가 자신에 닿아 있는 모든 간선들을 검사하여 최소 가중치를 가진 간선을 선택하므로 O(M) 시간이 소요
- 따라서 알고리즘의 수행 시간은 O(MlogN)

최단경로 알고리즘

- Dijkstra 알고리즘
- Floyd-Warshall 알고리즘

Dijkstra 알고리즘

- 최단 경로(Shortest Path) 찾기는 주어진 가중치그래프에서 출발점으로부터 도착점까지의 최단경로를 찾는 문제
- Dijkstra 알고리즘: 출발점으로부터 각 정점까지의 최단거리 및 경로를 계산
- Dijkstra 알고리즘은Prim의 MST 알고리즘과 매우 유사
- 차이점

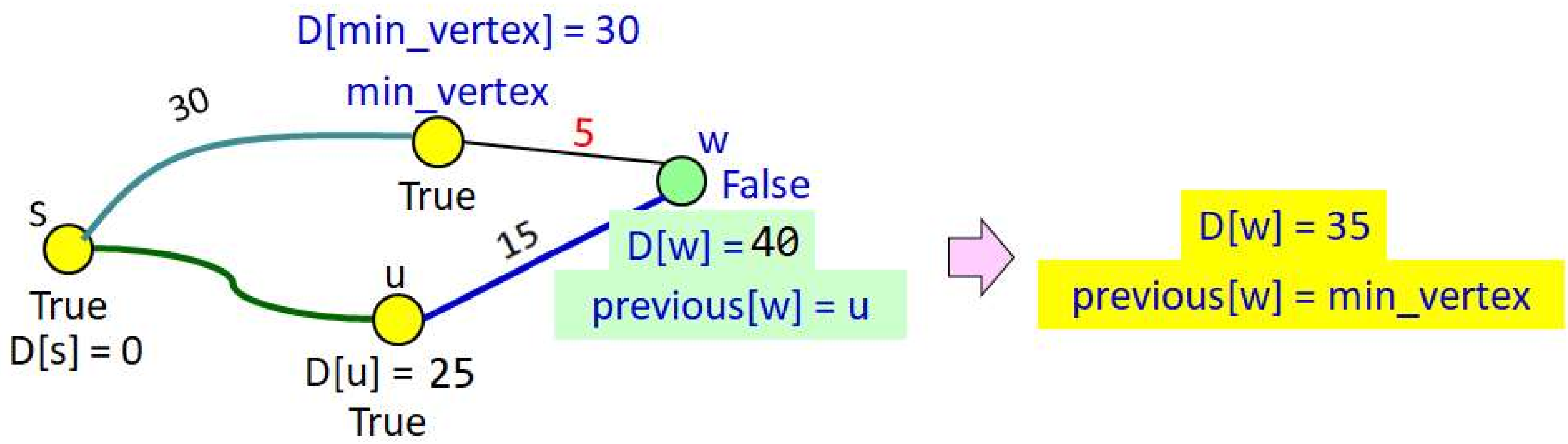
1. Dijkstra 알고리즘은 출발점이 주어지지지만 Prim알고리즘에서는 출발점이 주어지지 않는다는 것
2. Prim 알고리즘에서는 D의 원소에 간선의 가중치가 저장되지만, Dijkstra 알고리즘에서는 D의 원소에 출발점으로부터 각 정점까지의 경로의 길이가 저장됨

Dijkstra 알고리즘

```
[1] D를 ∞로 초기화한다. 단, D[s]=0으로 초기화한다.
[2] for k in range(N):
[3]     방문 안된 각 정점 i에 대해 D[i]가 최소인 정점 min_vertex를 찾고
        방문한다.
[4]     for min_vertex에 인접한 각 정점 w에 대해서:
[5]         if w 가 방문 안된 정점이면:
                wt = 간선 (min_vertex, w)의 가중치
[6]             if D[min_vertex] + wt < D[w]:
[7]                 D[w] = D[min_vertex] + wt
[8]                 previous[w] = min_vertex
```

- Step [7]의 간선 완화(Edge Relaxation)는 min_vertex가 step [3]에서 선택된 후에 s로부터 min_vertex를 경유하여 정점 w까지의 경로의 길이가 현재의 D[w]보다 더 짧아지면 짧은 길이로 D[w]를 갱신하는 것을 의미

- 그림은 D[w]가 min_vertex 덕분에 40에서 35로 완화된 것을 나타냄

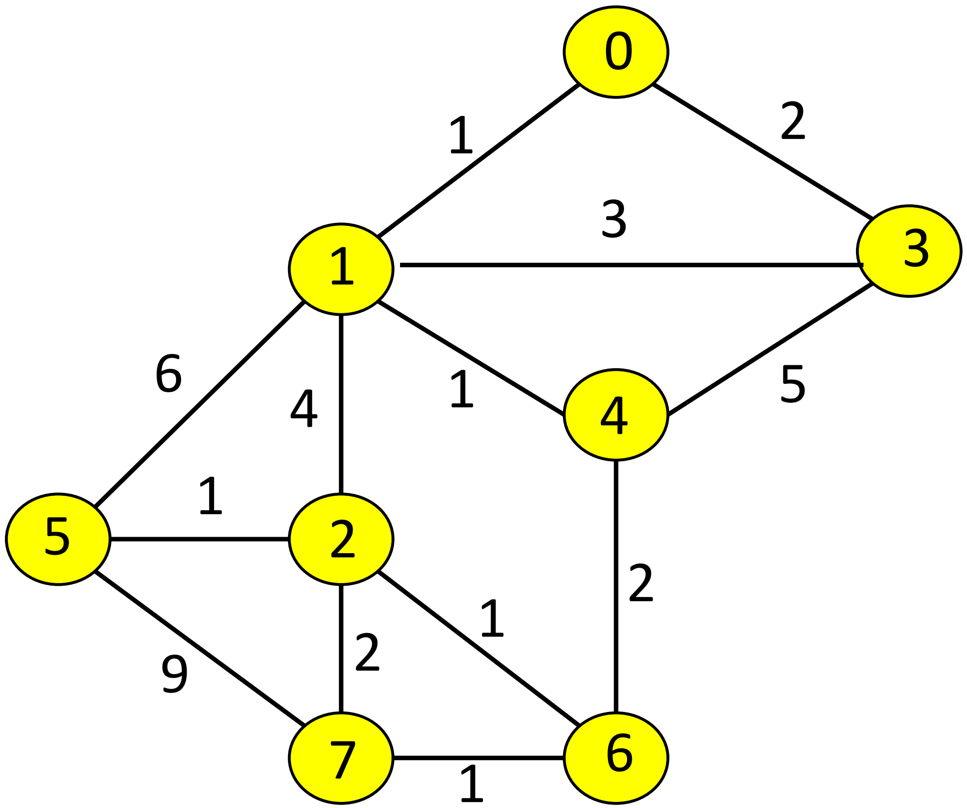


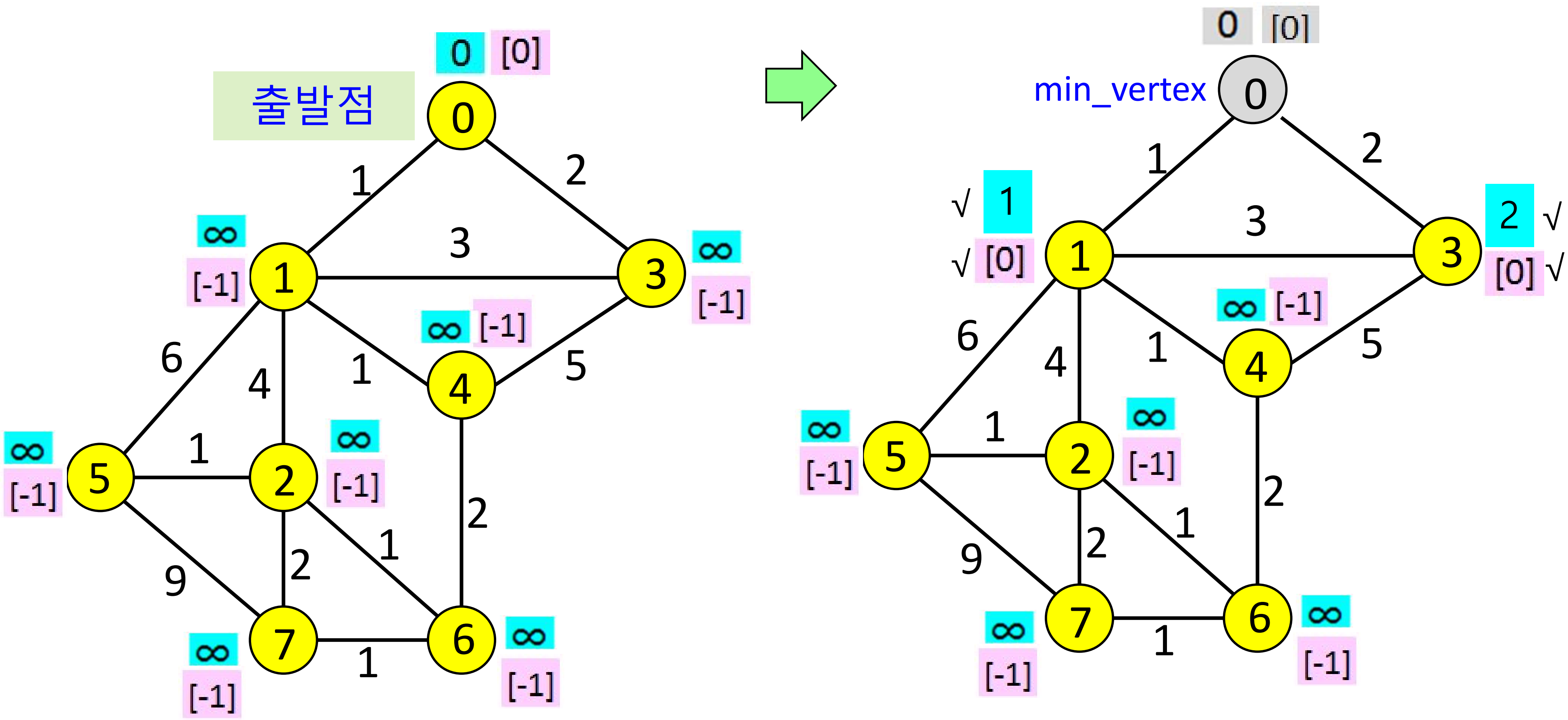
[핵심 아이디어]

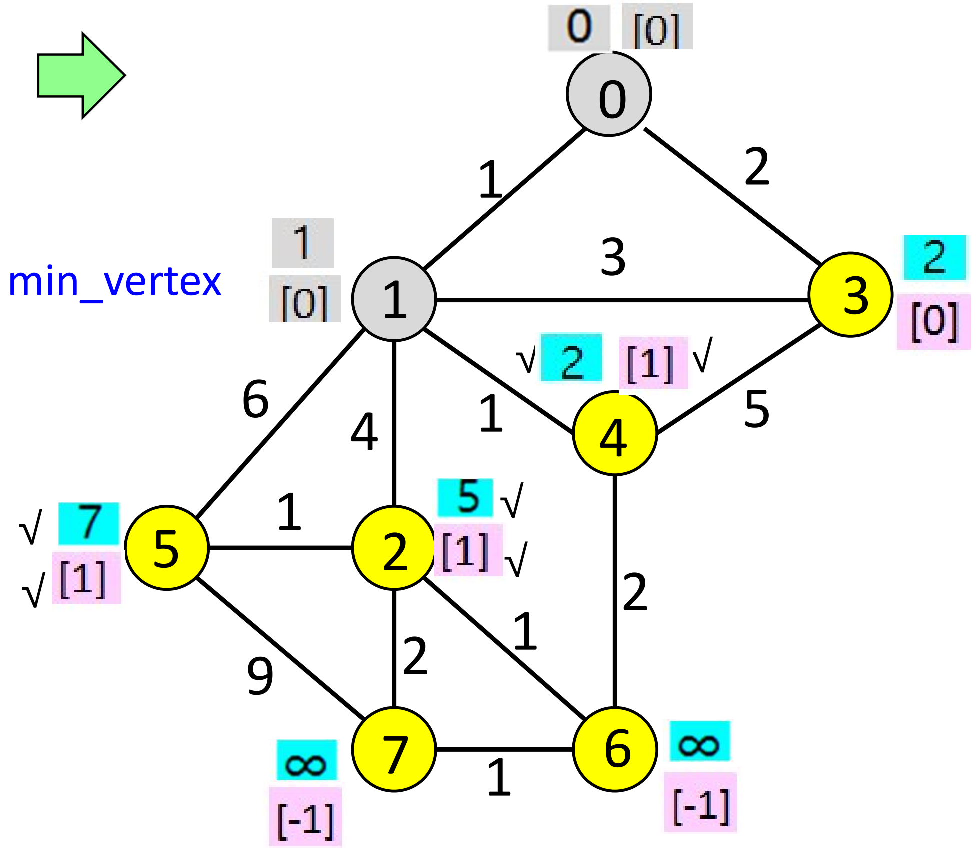
그리디하게 정점을 선택하여 방문하고, 선택한 정점의 방문 안된 인접한 정점들에 대한 간선 완화를 수행한다.

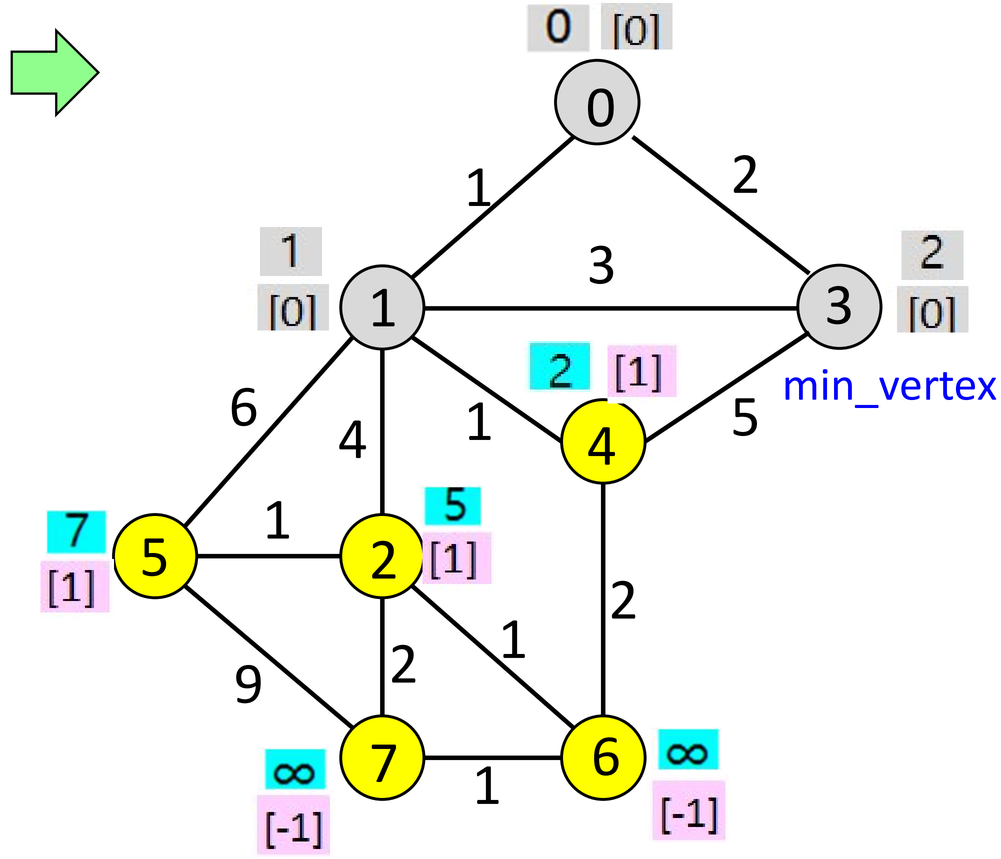
한번 방문된 정점의 D원소 값은 변하지 않는다.

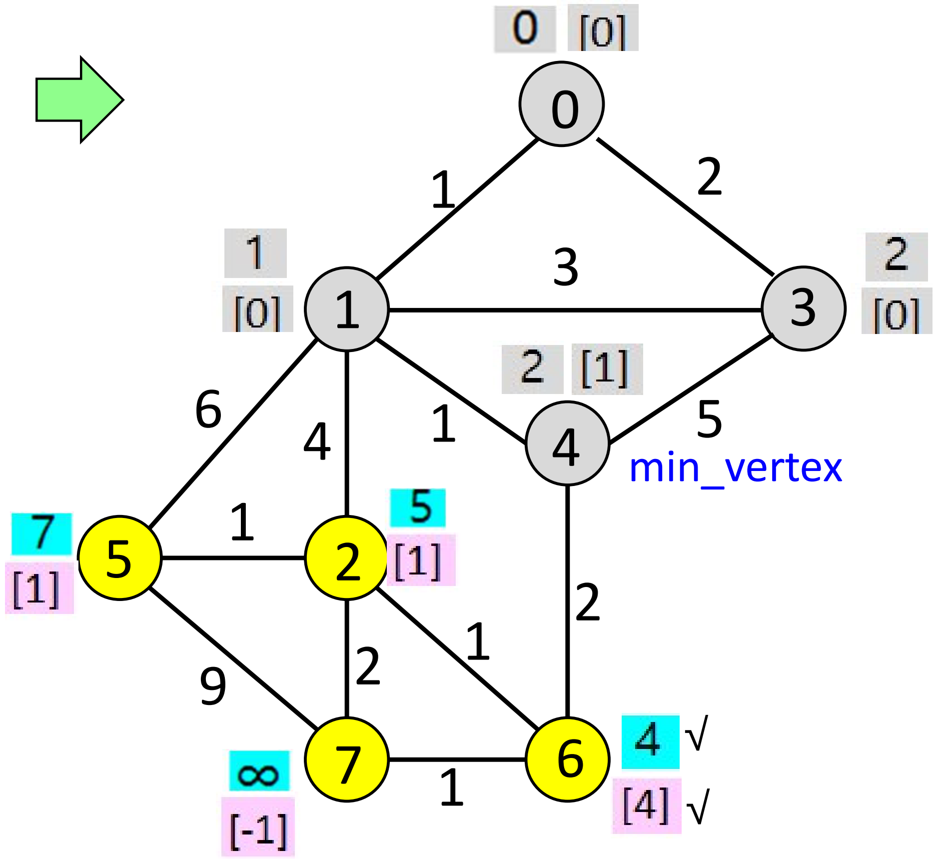
[예제]

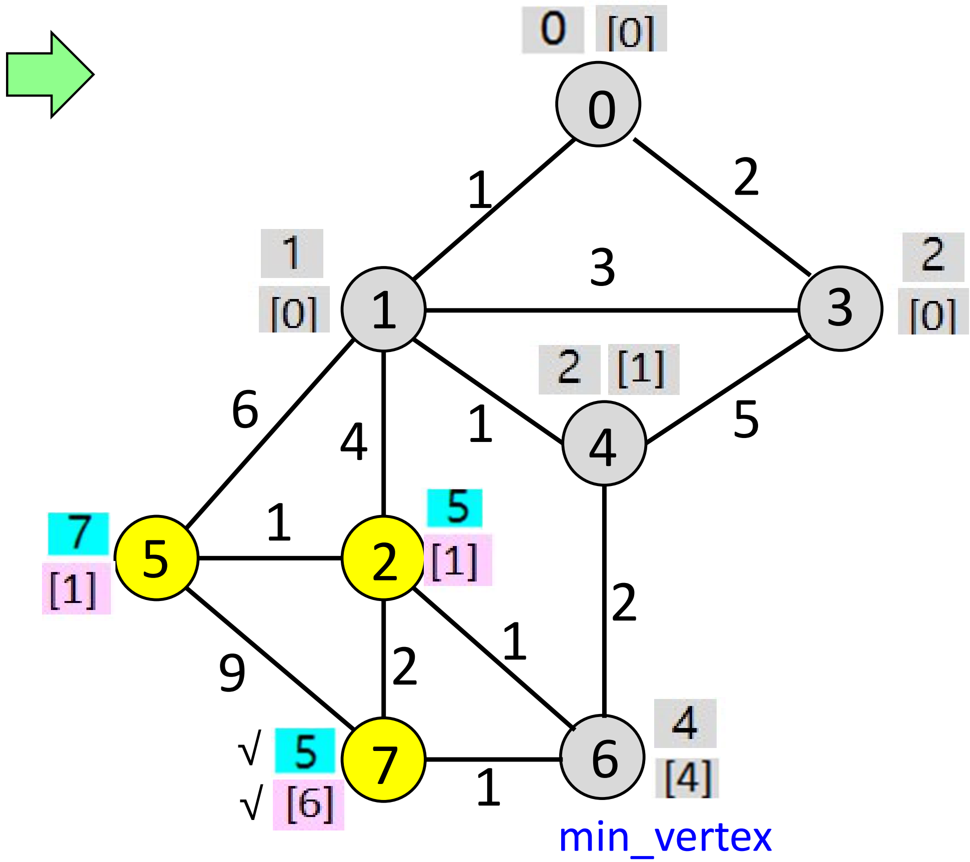


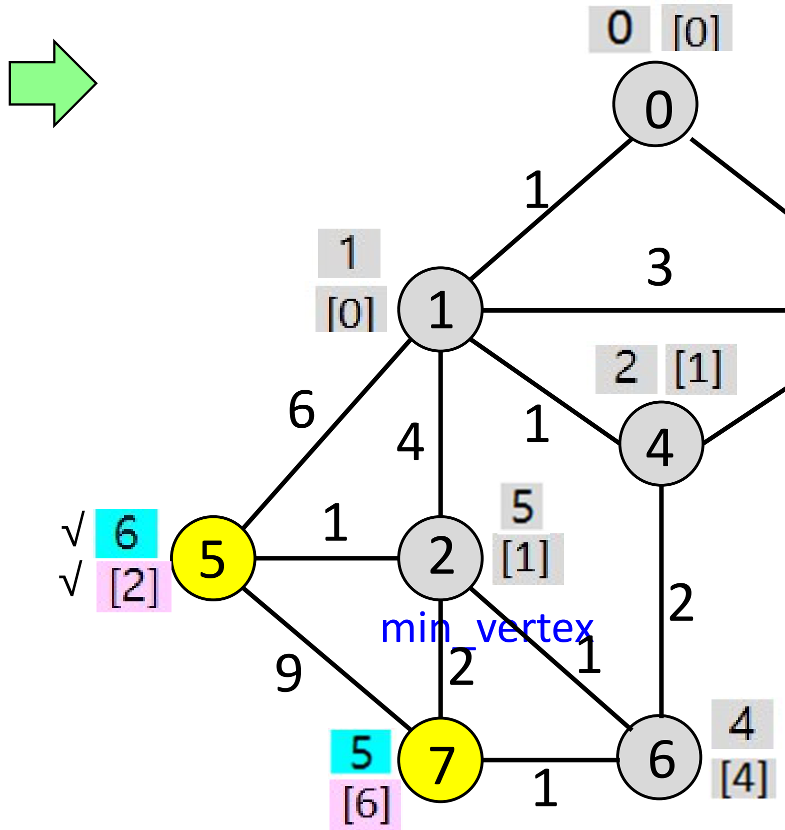

출발점
0 [0]
∞ [-1]
min_vertex
√ 1
√ [0]
2 √
[0] √


0 [0]
0
1
2
1
[0]
min_vertex
1
3
3
2
[0]
√ 2 [1] √
6
4
1
4
5
√ 7
√ [1]
5
1
2
5 √
[1] √
2
9
2
1
7
6
∞
[-1]
1
∞
[-1]

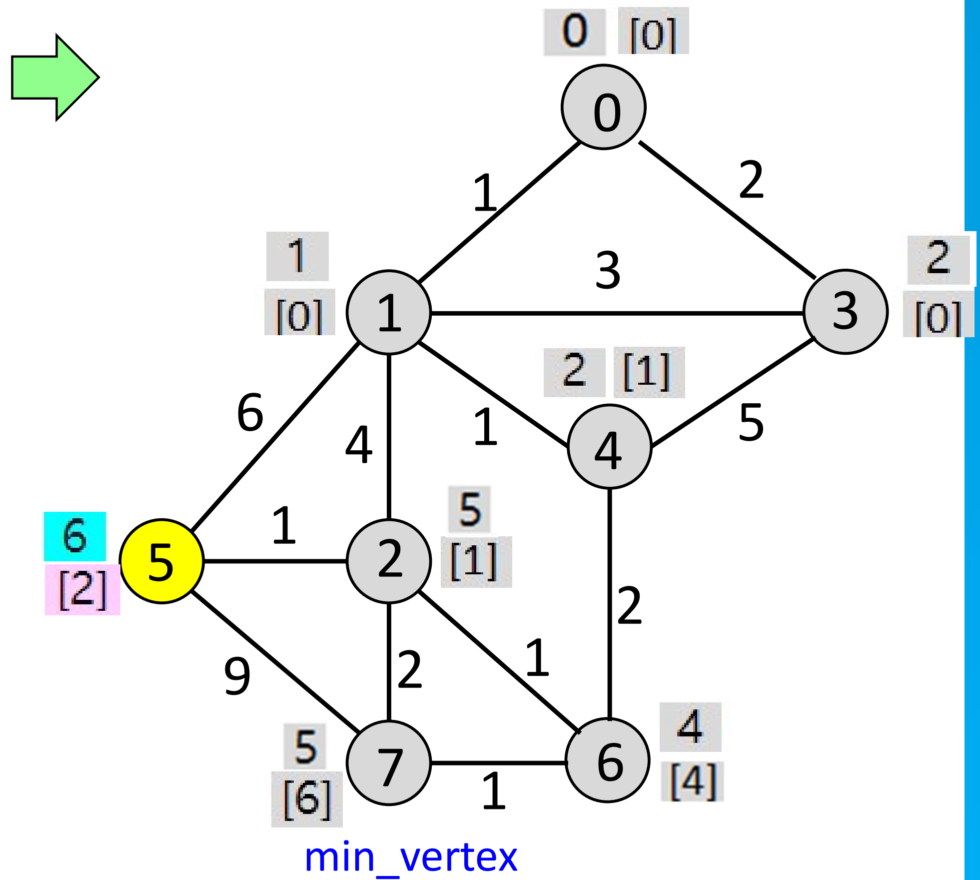

0 [0]
0
1
2
1
[0]
1
3
3
2
[0]
2 [1]
min_vertex
6
4
1
4
5
7
[1]
5
1
2
5
[1]
2
9
2
1
7
6
∞
[-1]
1
∞
[-1]


0 [0]
1 [0]
2 [0]
2 [1]
min_vertex
7 [1]
5 [1]
∞ [-1]
4 √
[4] √

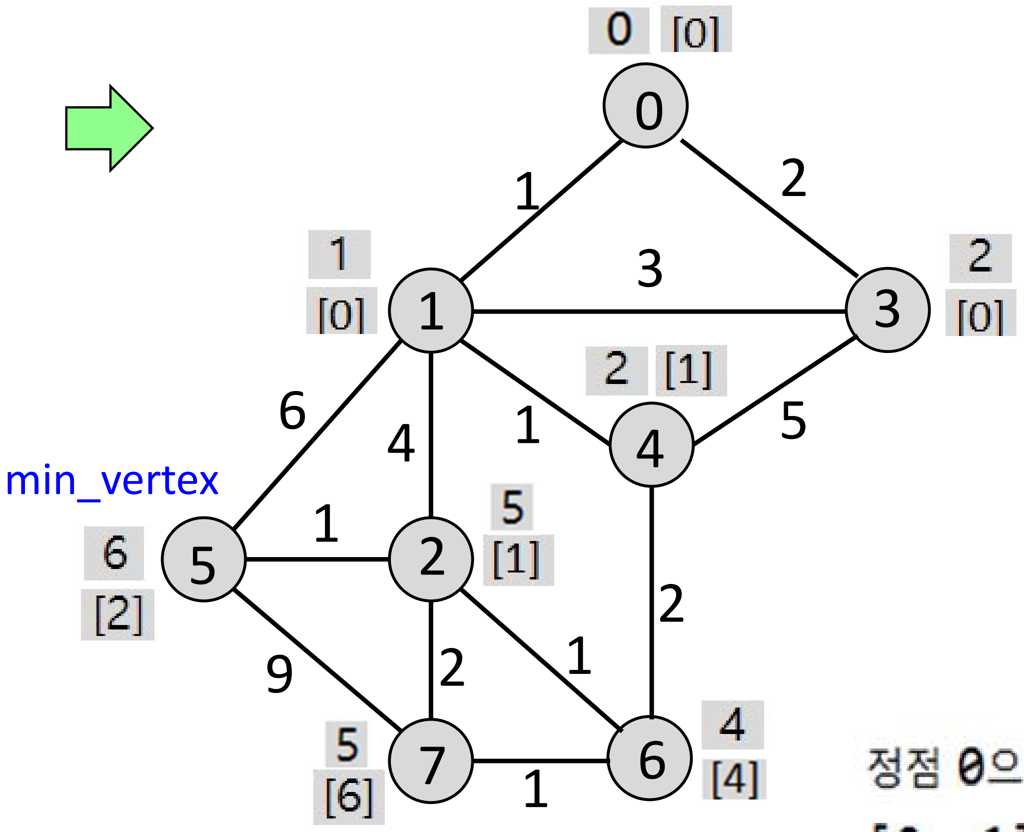

0 [0]
1 [0]
2 [0]
2 [1]
7 [1]
5 [1]
√ 5
√ [6]
4 [4]
min_vertex



0 [0]
1 [0]
2 [0]
2 [1]
5 [1]
√ 6
√ [2]
5 [6]
4 [4]
min_vertex



0 [0]
1 [0]
2 [0]
2 [1]
5 [1]
6 [2]
5 [6]
4 [4]
min_vertex



정점 0으로부터의 최단 거리	정점 0으로부터의 최단 경로
[0, 1] = 1	1<-0
[0, 2] = 5	2<-1<-0
[0, 3] = 2	3<-0
[0, 4] = 2	4<-1<-0
[0, 5] = 6	5<-2<-1<-0
[0, 6] = 4	6<-4<-1<-0
[0, 7] = 5	7<-6<-4<-1<-0

```python
import sys
N = 8
s = 0
g = [None] * N
g[0] = [(1, 1), (3, 2)]
g[1] = [(0, 1), (2, 4), (3, 3), (4, 1), (5, 6)]
g[2] = [(1, 4), (5, 1), (6, 1), (7, 2)]
g[3] = [(0, 2), (1, 3), (4, 5)]
g[4] = [(1, 1), (3, 5), (6, 2)]
g[5] = [(1, 6), (2, 1), (7, 9)]
g[6] = [(2, 1), (4, 2), (7, 1)]
g[7] = [(2, 2), (5, 9), (6, 1)]

visited = [False] * N
D = [sys.maxsize] * N
D[s] = 0
previous = [None] * N
previous[s] = s

```

sys.maxsize(최댓값) 사용 위해

입력 그래프의 인접리스트

각 원소를 최댓값으로

초기화

최단경로 추출을 위해

```python
for k in range(N):
    m = -1
    min_value = sys.maxsize
    for j in range(N):
        if not visited[j] and D[j] < min_value:
            min_value = D[j]
            m = j
    visited[m] = True
    for v, wt in list(g[m]):
        if not visited[v]:
            if D[m]+wt < D[v]:
                D[v] = D[m] + wt
                previous[v] = m
```

m = min_vertex

방문 안된 정점들의 D 원소들 중에서 최솟값을 가진 정점 m 찾기

정점 m에 인접한 v와 (m, v)의 가중치 wt에 대해

D[v] 갱신: 간선완화

D[v]가 정점 m 때문에 갱신되었음을 기록

```python
print('정점 ', s,'(으)로부터 최단거리:')
for i in range(N):
    if D[i] == sys.maxsize:
        print(s,'와(과) ', i ,' 사이에 경로 없음.')
    else:
        print('[%d, %d]'% (s, i), '=', D[i])

print('\n정점 ', s,'(으)로부터의 최단 경로')
for i in range(N):
    back = i
    print(back, end='')
    while back != s:
        print(' <-', previous[back], end='')
        back = previous[back]
    print()
```

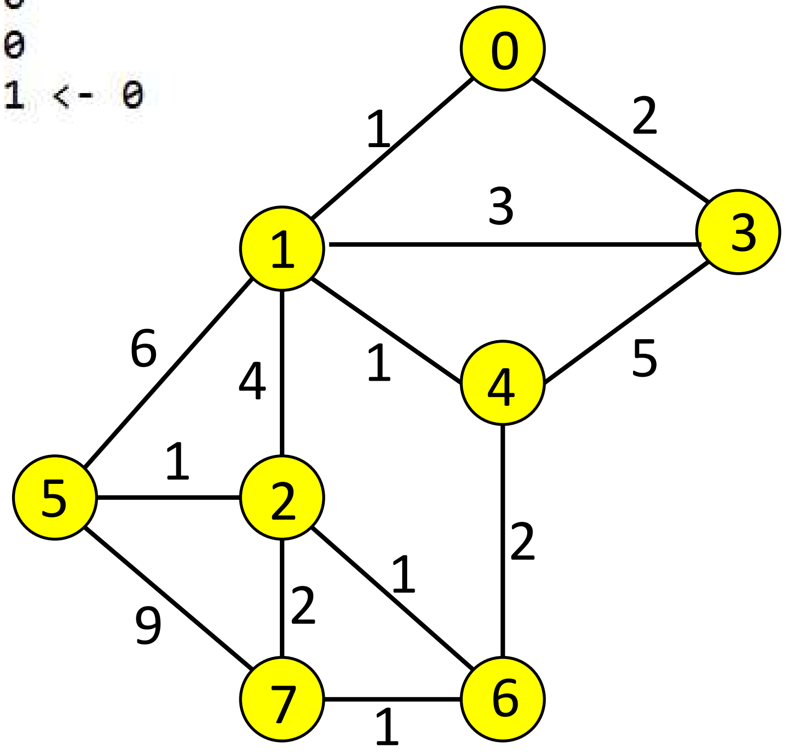
[프로그램 8-7] dijkstra.py

프로그램 수행 결과

Console ✕   PyUnit

<terminated> dijkstra.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32

```
정점 0 (으)로부터 최단거리:        정점 0 (으)로부터의 최단 경로
[0, 0] = 0                       0
[0, 1] = 1                       1 <- 0
[0, 2] = 5                       2 <- 1 <- 0
[0, 3] = 2                       3 <- 0
[0, 4] = 2                       4 <- 1 <- 0
[0, 5] = 6                       5 <- 2 <- 1 <- 0
[0, 6] = 4                       6 <- 4 <- 1 <- 0
[0, 7] = 5                       7 <- 6 <- 4 <- 1 <- 0
```

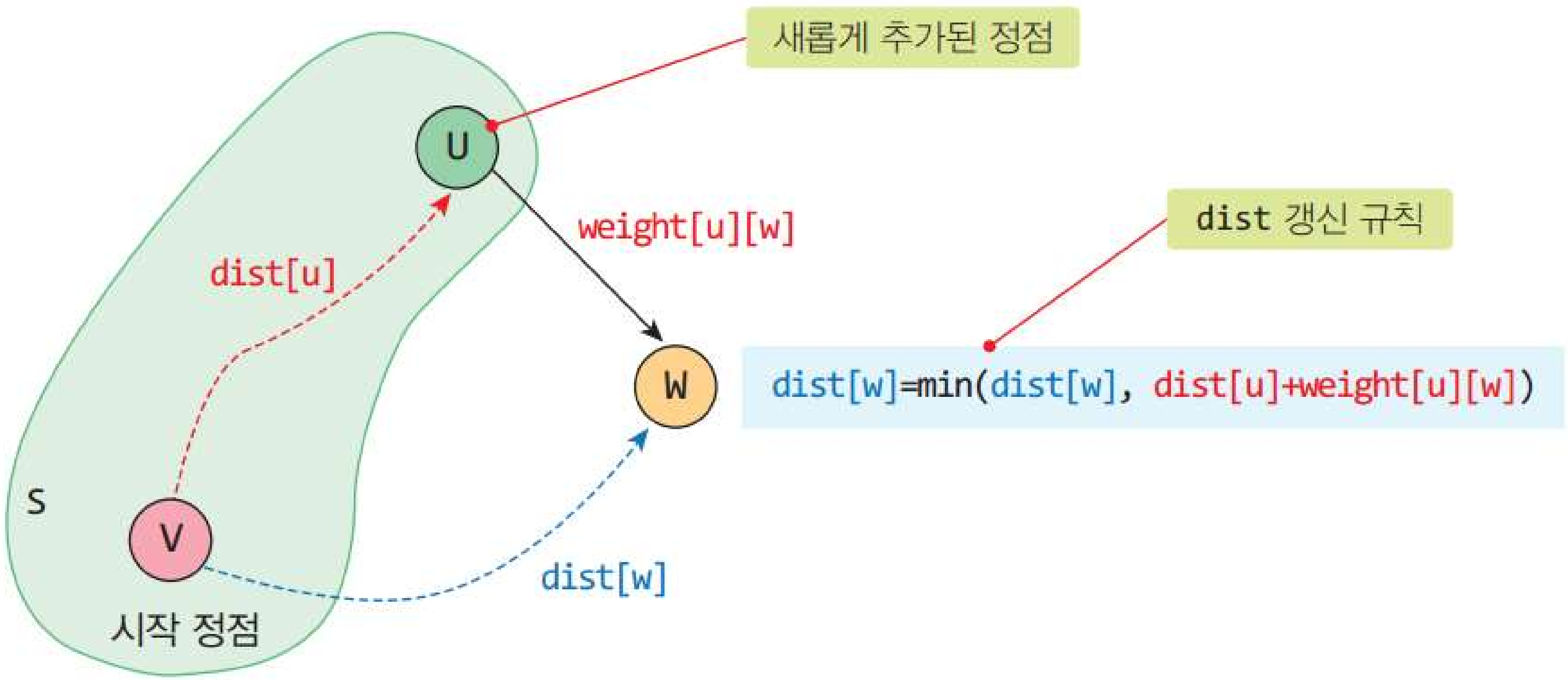


수행 시간(1)

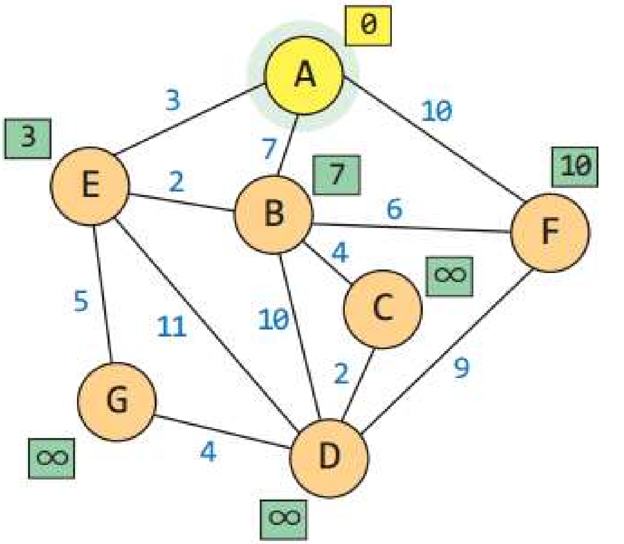
- Dijkstra 알고리즘은 N번의 반복을 거쳐 min_vertex를 찾고 min_vertex에 인접하면서 방문되지 않은 정점들에 대한 간선완화를 시도
- 이후 D에서 min_vertex를 탐색하는데 O(N) 시간이 소요되고, min_vertex에 인접한 정점들을 검사하여 D의 원소들을 갱신하므로 추가로 O(N) 시간이 소요
- 따라서 총 수행 시간은 Nx(O(N) +O(N)) = $O(N^2)$

dist 갱신

- 새로운 정점이 S에 추가되면 dist 갱신

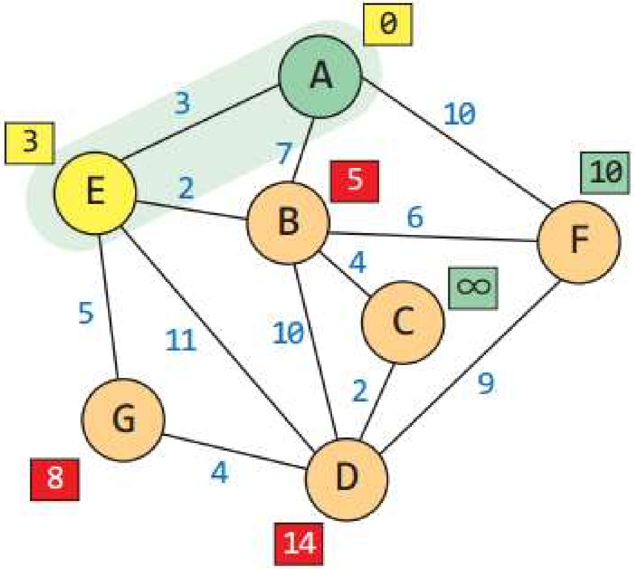


알고리즘 실행 과정: Step1-2



S = { A }

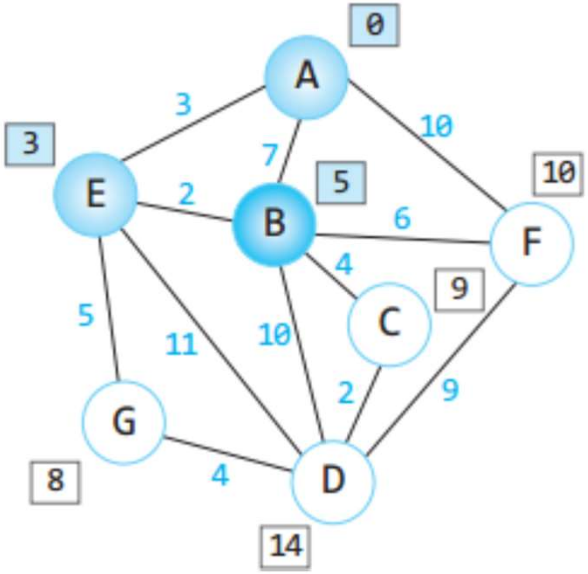
dist(A)=w(A,A)=0
dist(B)=w(A,B)=7
dist(C)=w(A,C)=∞
dist(D)=w(A,D)=∞
dist(E)=w(A,E)=3
dist(F)=w(A,F)=10
dist(G)=w(A,G)=∞



S = { A, E }

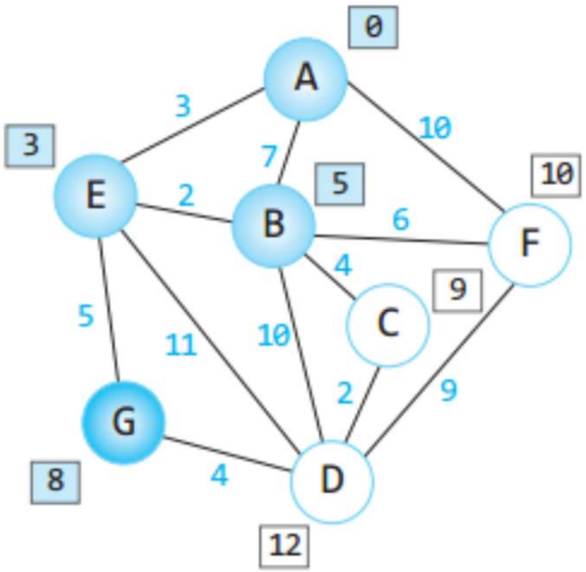
dist(A)=0
dist(E)=w(A,E)=3
dist(B)=min(dist(B),dist(E)+w(E,B))=min(7,3+2)=5
dist(C)=min(dist(C),dist(E)+w(E,C))=min(∞,3+∞)=∞
dist(D)=min(dist(D),dist(E)+w(E,D))=min(∞,3+11)=14
dist(F)=min(dist(F),dist(E)+w(E,F))=min(10,3+∞)=10
dist(G)=min(dist(G),dist(E)+w(E,G))=min(∞,3+5)=8

알고리즘 실행 과정: Step3 - 4



S={A, E, B}

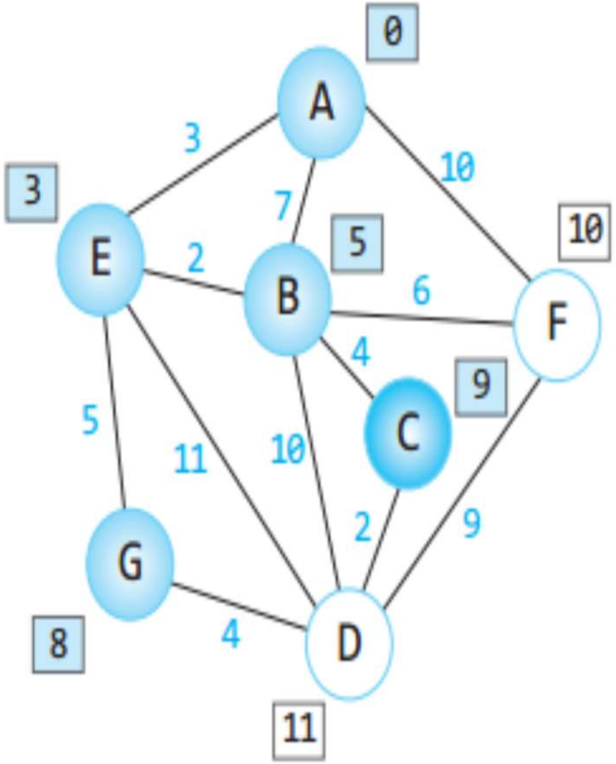
dist(A)=0
dist(B)=5
dist(C)=min(dist(C),dist(B)+w(B,C))=min(∞,5+4)=9
dist(D)=min(dist(D),dist(B)+w(B,D))=min(14,5+10)=14
dist(E)=3
dist(F)=min(dist(F),dist(B)+w(B,F))=min(10,5+6)=10
dist(G)=min(dist(G),dist(B)+w(B,G))=min(8,5+∞)= **8**



S={A, E, B, G}

dist(A)=0
dist(B)=5
dist(C)=min(dist(C),dist(G)+w(G,C))=min(9,8+∞)=9
dist(D)=min(dist(D),dist(G)+w(G,D))=min(14,8+4)=12
dist(E)=3
dist(F)=min(dist(F),dist(G)+w(G,F))=min(10,8+∞)=10
dist(G)=8

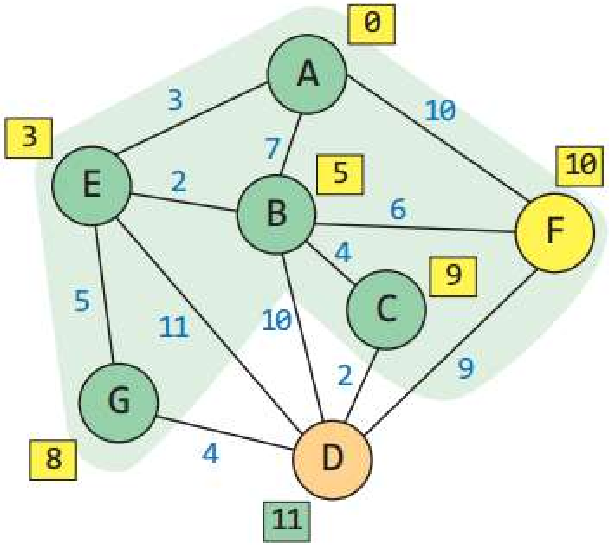
알고리즘 실행 과정: Step5 - 6



S={A, E, B, G, C}

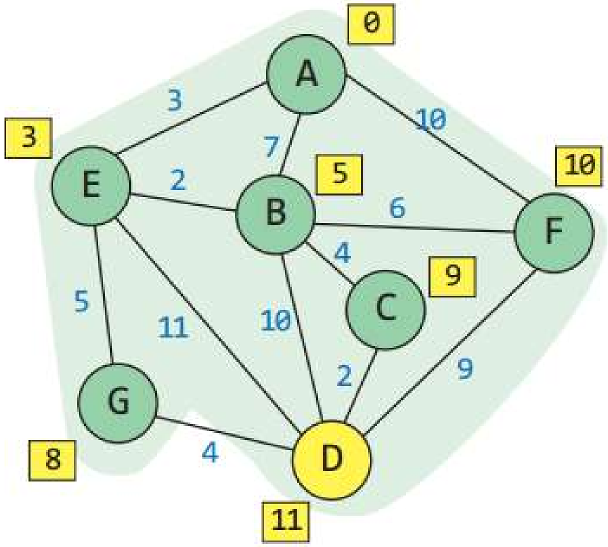
dist(A)=0
dist(B)=5
dist(C)=9
dist(D)=min(dist(D),dist(C)+w(C,D))=min(12,9+2)=11
dist(E)=3
dist(F)=min(dist(F),dist(C)+w(C,F))=min(10,9+∞)=10
dist(G)=8

알고리즘 실행 과정: Step6-최종



S = { A, E, B, G, C, F }

dist(A)=0
dist(B)=5
dist(C)=9
dist(E)=3
dist(F)=10
dist(G)=8
dist(D)=min(dist(D),dist(F)+w(F,D))=min(11,10+9)=11



S = { A, E, B, G, C, F, D }

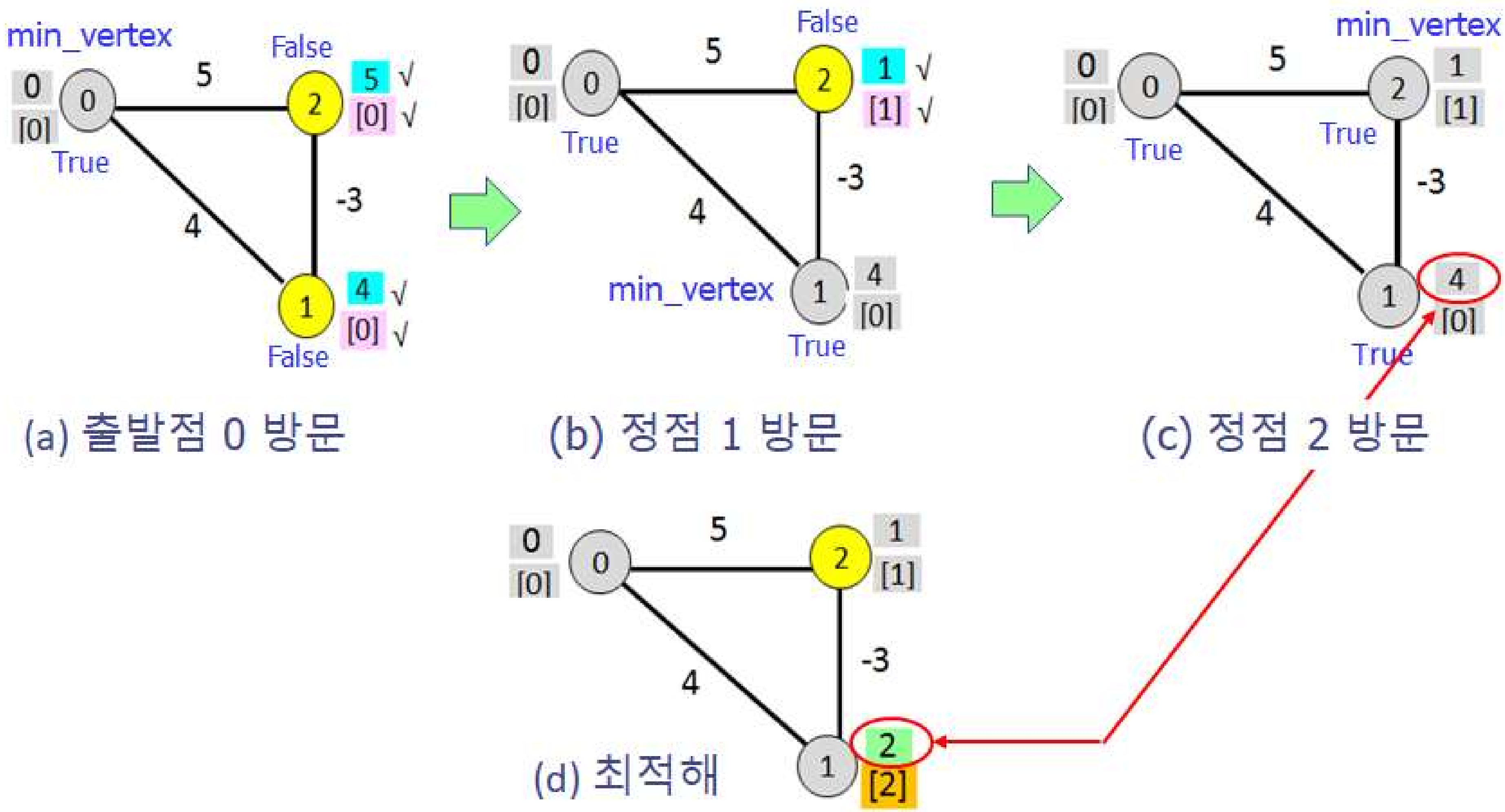
dist(A)=0
dist(B)=5
dist(C)=9
dist(D)=11
dist(E)=3
dist(F)=10
dist(G)=8

수행 시간(2)

- Dijkstra 최단경로 알고리즘은 Prim MST 알고리즘과 전체적으로 동일하므로 수행시간도 동일하다.
- 따라서 이진힙과 피보나치힙을 사용하는 경우의 수행시간도 각각 동일한 수행 시간을 갖는다.

- Dijkstra 알고리즘은 입력그래프에 음수 가중치가 있으면 최단 경로 찾기에 실패하는 경우가 발생

- Dijkstra 알고리즘이 최적해를 찾지 못하는 반례



- (a) 출발점이 방문되어 visited[0] = true
- 이후 D[1] = 4, previous[1] = 0 그리고 D[2] = 5, previous[2] = 0으로 각각 갱신
- (b) D[1]이 최솟값이므로 정점 1이 방문되고, D[2] = 1, previous[2] = 1로 갱신
- (c) 마지막으로 방문 안된 정점 2가 방문되고 알고리즘 종료
- 그러나 (d)를 보면 출발점 0에서 정점 1까지 최단 경로는 [0-2-1]이고, 거리는 2

- [이러한 문제점이 발생한 이유] Dijkstra 알고리즘이 <u>D의 원소 값의 증가순으로 min_vertex를 선택</u>하고, 한번 방문된 정점의 D 원소를 다시 갱신하지 않기 때문

수고하셨습니다