# 그래프 탐색

한국기술교육대학교 컴퓨터공학부 김상진


```
while(!morning){
    alcohol++
    dance++
} #party
```

```
while(!sleep){
    think++
    solve++
} #cse-mode
```

## 교육목표

- 그래프
  - 그래프 표현: 인접 행렬, 인접 리스트
- 그래프 탐색: BFS, DFS
- BFS 응용
  - 무방향 그래프에서 최단 경로, 무방향 그래프의 연결 여부
- DFS 응용
  - 방향 그래프에서 위상 정렬, 방향 그래프의 강한 연결 요소
- 다익스트라 알고리즘
  - 가중치(≥0) 방향 그래프에서 최단 경로
  - 대표적인 탐욕적 알고리즘

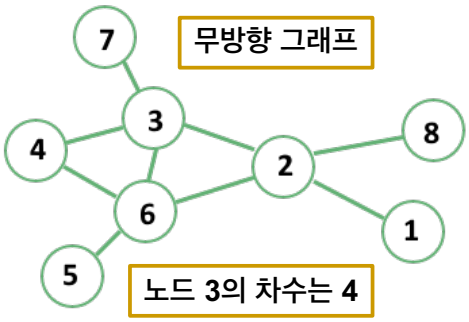
알고리즘 관련 많은 재미난 또 난이도가 있는 문제는 그래프 문제임

# 그래프 (1/3)

- 그래프(graph): 공집합이 아닌 노드(vertex, node)들의 유한 집합 $V$와 이 노드들을 연결하는 간선(edge)들의 집합 $E$로 구성
  - $G = (V, E), |V| = n, |E| = m$
  - 트리는 그래프의 한 종류임
- 무방향 그래프(undirected graph): 간선의 방향성이 없는 그래프
- 방향 그래프(directed graph, digraph): 간선의 방향성이 있는 그래프
  - 방향 그래프에서 간선을 아크(arc)라고도 함
- 가중치 그래프(weighted graph): 간선에 가중치가 있는 그래프
- 비가중치 그래프: 간선에 가중치가 없는 그래프
  - 모든 간선의 가중치가 같으면 가중치가 의미가 없음
- 다중 그래프(multigraph): 두 노드를 잇는 간선이 여러 개 존재할 수 있음
  - 이와 같은 간선을 병행 간선(parallel edge)이라 함
- 인접 노드(adjacent vertex): 간선에 의해 연결되어 있는 노드 (이웃 노드)

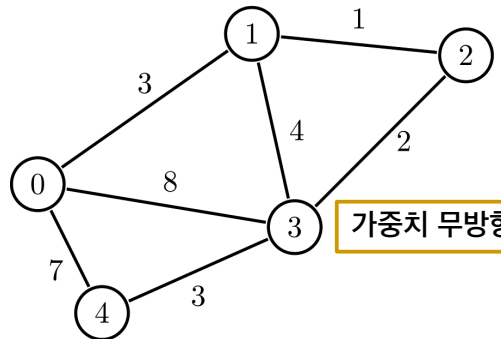


노드 0에서 2까지 경로
(0,1), (1,2): 길이 4
(0,3), (3,1), (1,2): 길이 13

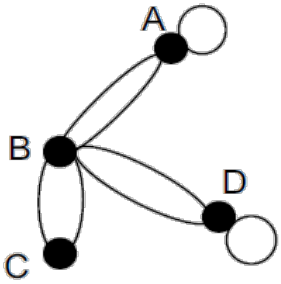



무방향 그래프

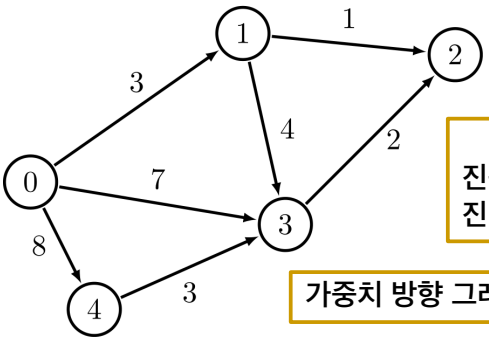
노드 3의 차수는 4




가중치 무방향 그래프




다중 그래프



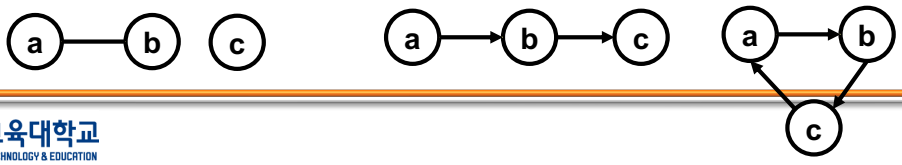

노드 3
진출 차수: 1
진입 차수: 3

가중치 방향 그래프

노드 1은 0의 인접 노드이지만
노드 0은 1의 인접 노드가 아님

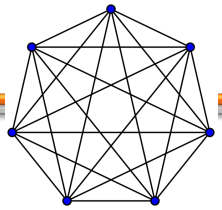
# 그래프 (2/3)

- 노드의 차수: 노드에 연결된 간선의 수를 말하며, 방향 그래프에서는 진입 차수(indegree)와 진출 차수(outdegree)로 나누어 고려함
- 경로(path): 두 개의 노드를 연결하는 일련의 노드들
  - 노드 $a$에서 $b$까지의 경로 $a, v_1, v_2, \dots, v_k, b$가 존재하기 위해서는 간선 $(a, v_1), (v_1, v_2), \dots, (v_{n-1}, v_k), (v_k, b)$가 존재해야 함
  - 단순 경로: 한 노드를 두 번 거치지 않는 경로
- 경로의 길이
  - 가중치 그래프: 경로를 구성하는 간선의 가중치 합
  - 비가중치 그래프: 경로를 구성하는 간선의 수
- 주기(cycle): 첫 번째 노드와 마지막 노드가 같은 경로
- 연결 그래프(connected graph): 그래프에서 서로 다른 모든 노드 쌍 사이에 경로가 존재하는 그래프
  - 방향 그래프: 약한(weakly) vs. 강한(strongly) 연결 그래프

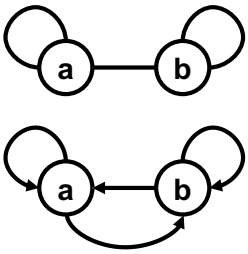





# 그래프 (3/3)



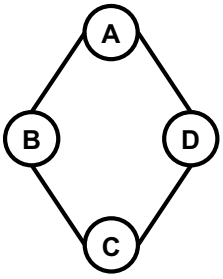
- 완전 그래프(complete graph): 모든 노드가 인접 노드가 되는 그래프
  - 완전 그래프이면 연결 그래프이지만 반대는 성립하지 않음
- 부분 그래프(subgraph): 다음이 성립하면 $G' = (V', E')$는 $G = (V, E)$ 의 부분 그래프라 함
  - $V' \subseteq V$, $E' \subseteq E$
- 유한 그래프: 노드의 수가 유한한 그래프
  - 반대 개념: 무한 그래프
- $n$개의 노드로 구성된 무방향 그래프의 최대 간선 수
  - 진출과 진입이 같은 간선의 존재 고려: $\frac{n(n+1)}{2} = \frac{n(n-1)}{2} + n$
  - 진출과 진입이 같은 간선을 고려하지 않으면: $n(n-1)/2$
    - 연결 무방향 그래프의 최소 간선 수: $n - 1$ (트리의 간선 수)
- 그래프의 간선 수가 최대 간선 수에 가까우면 밀집(dense) 그래프
  - 반대로 간선 수가 0에 가까우면 희소(sparse) 그래프라 함

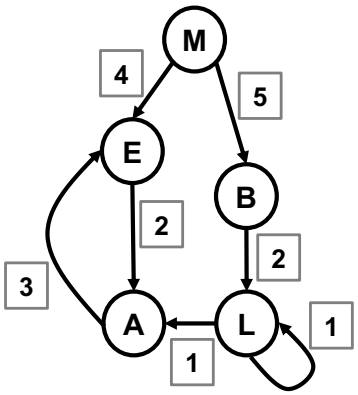


# 그래프의 표현 (1/2)

장점.
- 밀집 방향 그래프에 유리한 방식

## 방법 1. 인접 행렬




| | A | B | E | L | M |
|---|---|---|---|---|---|
| A | - | - | 3 | - | - |
| B | - | - | - | 2 | - |
| E | 2 | - | - | - | - |
| L | 1 | - | - | 1 | - |
| M | - | 5 | 4 | - | - |

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 0 | 1 |
| B | 1 | 0 | 1 | 0 |
| C | 0 | 1 | 0 | 1 |
| D | 1 | 0 | 1 | 0 |

합: 정점의 차수

단점. $n^2$ 공간이 필요
- 무방향 행렬은 이것의 반만으로 표현 가능
- 가중치 그래프에서 가중치 값의 범위에 따라 간선이 없는 경우를 표현하기 힘들 수 있음

- bool 2차원 배열

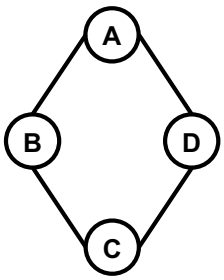
- 2개의 배열
- (bool, 값) 배열
- std::optional 배열

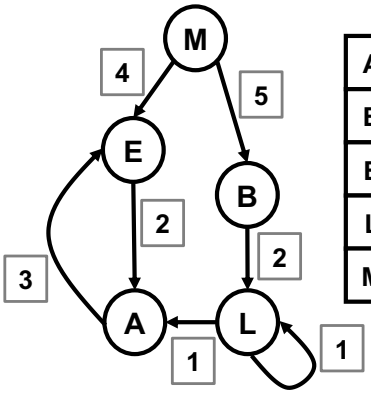


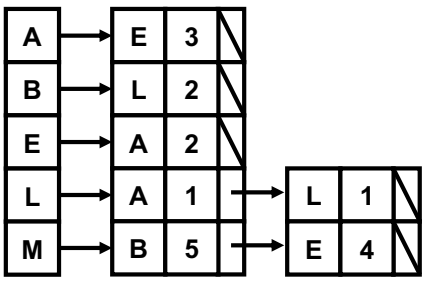
# 그래프의 표현 (2/2)

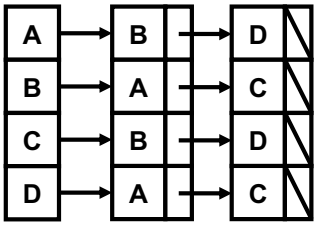
장점.
- 희소 무방향 그래프에 유리

## 방법 2. 인접 리스트



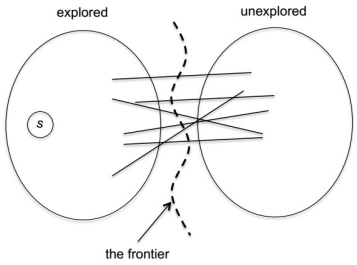

단점.
- 방향 그래프에서는 진입 차수를 계산하는 것이 힘듦
  - 역인접 리스트 유지 ⇒ 차라리 인접 행렬

- 실제 구현에서는 연결 리스트를 꼭 사용하지 않고 배열 리스트를 사용하는 경우도 많음

| | 희소 | 밀집 |
|---|---|---|
| 방향 | | 인접 행렬 |
| 무방향 | 인접 리스트 | |

# 일반 그래프 탐색 (1/2)

- 목표: 출발 노드부터 도달 가능한(경로가 있는) 모든 노드 찾기
- 입력. 그래프 $G$와 출발 노드 $s$
- 출력. $s$에서 출발하여 도달 가능한 노드의 집합
- 알고리즘




```
s := starting node
set s as explored and all other nodes as unexplored
while possible do
    choose an edge (u, v) with u explored and v unexplored
    mark v explored
    if none halt
```

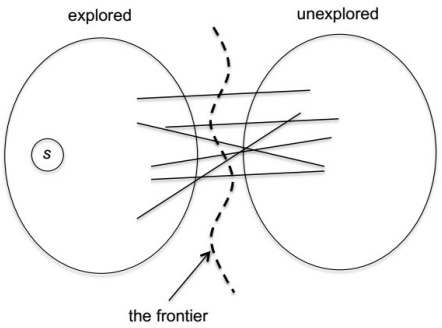
  - 그래프의 방향성 여부와 무관하게 올바르게 동작하는 알고리즘
  - 알고리즘의 정확성 증명 가능
  - 핵심. 여러 후보 간선이 있을 때 어떤 것을 선택
    - 어떤 기준? 방법?



# 일반 그래프 탐색의 정확성 증명

- 일반 그래프 탐색 알고리즘의 정확성 증명
  - 출발 노드가 $s$일 때, 알고리즘이 종료되어 $v$가 답에 포함되기 위한 필요충분조건은 $s$에서 $v$까지 경로가 있어야 함
- 증명)
  - $v$가 포함되면 $s$에서 $v$까지 경로가 있음
    - 당연. 알고리즘은 경로만 따라 이동함
  - $s$에서 $v$까지 경로가 있으면 $v$는 최종적으로 포함됨
    - 모순 증명) 경로가 있는데 $v$가 포함되지 않음
      - 최종적으로 포함된 집합이 $E$이고 포함되지 않은 집합이 $\bar{E}$이면 $v$는 $\bar{E}$에 속함
    - 하지만 이 가정은 모순임. $s$에서 $v$까지 경로가 존재하기 때문에 $E$에서 $\bar{E}$로 가는 간선이 존재하며, 이 때문에 반복문이 종료할 수 없음

# 일반 그래프 탐색 (2/2)



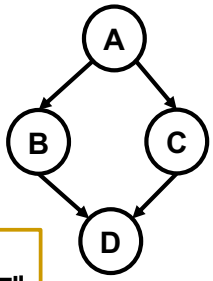

- 선택 기준 측면에서 생각하면
  - 출발 노드가 있으므로 첫 번째 선택 가능한 간선은 출발 노드에서 진출하는 간선 중 하나임
  - 특정 노드를 선택하였다면 그다음 반복에서는 두 노드에서 진출하는 간선 중에서 선택해야 함
- 목표를 생각하면
  - 출발 노드로부터 도달 가능한 모든 노드를 찾는 것이 목표임
  - 출발 노드에서 진출 간선에 의해 연결된 모든 노드는 최종 답에 포함되어야 함
  - 또 이 노드의 진출 간선에 의해 연결된 모든 노드도 최종 답에 포함되어야 함
    - 이 과정을 반복하면 목표를 달성할 수 있음
  - 출발 노드에서 진출 간선에 의해 연결된 모든 노드를 저장하여 처리해야 함
    - 어디에 저장? 리스트, 큐, 스택, 우선순위 큐 …
  - 먼저 꼭 처리해야 하는 노드는 없음. 처리한 노드는 다시 고려할 필요가 없음. 처리한 순서대로 자료구조에서 제거해야 함 ⟹ 큐, 스택



# BFS vs. DFS

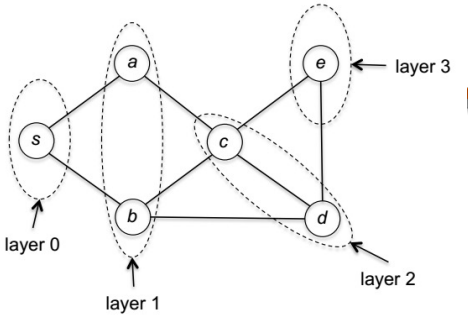
BFS vs. DFS
- 트리가 깊지 않으면?
- 트리의 폭이 크면?

- BFS (cautious approach)
  - 큐를 사용하여 구현
  - 선택한 노드에 인접한 모든 노드를 먼저 방문
  - 무방향 비가중치 그래프에서 최단 경로를 구할 수 있음
  - 무방향 그래프에서 연결 여부 확인에 사용 (방향도 가능)
- DFS (aggressive approach)
  - 스택을 사용하여 구현 (스택없이 재귀적으로 구현 가능)
  - 선택된 노드의 첫 번째 인접 노드를 방문하고, 그 노드의 첫 번째 인접 노드를 방문하는 형태
  - 방향 그래프에서 연결 여부 확인에 사용 (무방향도 가능)
- 보통 무방향 그래프 문제는 BFS, 방향 그래프는 DFS를 사용하지만 그래프의 모습과 해결하고자 하는 문제의 특성에 의해 탐색 방법을 결정해야 함
  - 보통 둘 다 가능하면 BFS를 많이 사용함




BFS vs. DFS
- 주기 찾기 문제

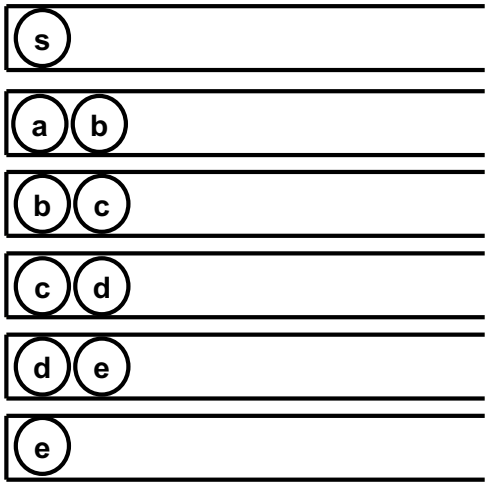
# 너비 우선 탐색(BFS)




- BFS
  - 같은 거리에 있는 노드들 순으로 방문
    - 옆 그림에서는 layer 순으로
  - 질문. 노드 개수가 $n$인 무방향 그래프에서 최대, 최소 layer 수는?
  - 알고리즘

```
s ≔ starting node
Q ≔ empty queue of nodes
visited[s] ≔ true
Q.push(s)
while Q is not empty do // O(n)
    v ≔ Q.pop()
    for each edge (v, w) do // O(m)
        if not visited[w] then
            visited[w] ≔ true
            Q.push(w)
```

- 한 번 큐에 삽입한 노드는 다시 삽입하지 않음



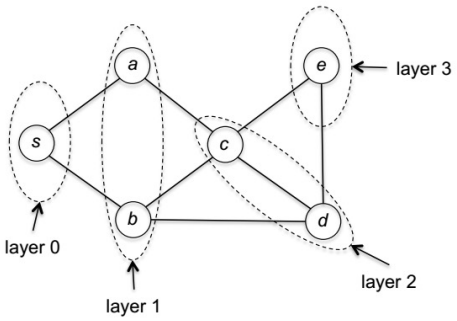



# 너비 우선의 시간복잡도

- while 루프 이전: $O(n)$
  - 큐 생성, visited 배열 초기화 등
- 바깥 while 루프: $n$번 반복
  - 한 번에 하나의 노드 처리
- 내부 for 루프
  - 인접 행렬: $n$번 반복 $\Longrightarrow O(n^2)$
  - 인접 리스트: 각 노드의 간선 수에 좌우
    - 전체적으로 모든 간선을 고려함 $\Longrightarrow O(m)$
  - 큐 연산: $O(n)$
- 전체 비용
  - 인접 행렬: $O(n^2 + n) = O(n^2)$
  - 인접 리스트: $O(m + n)$
- 무방향, 방향 차이가 없음

- $m$과 $n$의 관계는?
  - 무방향 그래프: $m \leq \frac{n(n-1)}{2}$
  - $O(m + n) = O(m) = O(n^2)$

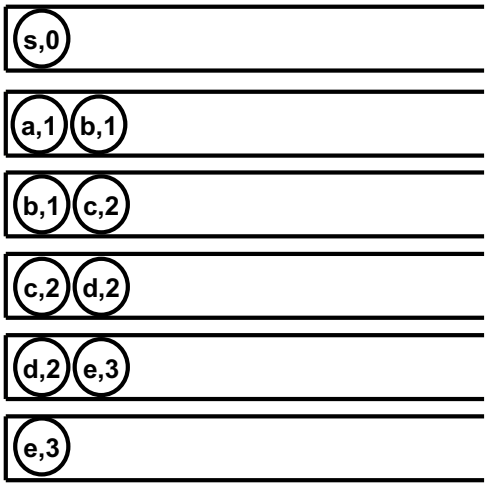
# BFS를 이용한 최단 경로




- 알고리즘

```
s := starting node
visited[s] := true and visited[x] := false for rest
distance[s] := 0 and distance[x] := ∞ for rest
Q := queue of nodes with distances
Q.push(s)
while Q is not empty do
    v := Q.pop()
    for each edge (v, w) do
        if not visited[w] then
            visited[w] := true
            distance[w] := distance[v] + 1
            Q.push(w)
```

- visited를 사용하지 않고 distances만 이용 가능
- 경로 자체
  1) distance를 이용하여 거꾸로 계산
  2) 진행하면서 이전 노드 정보 기록
     prevNode[w] := v

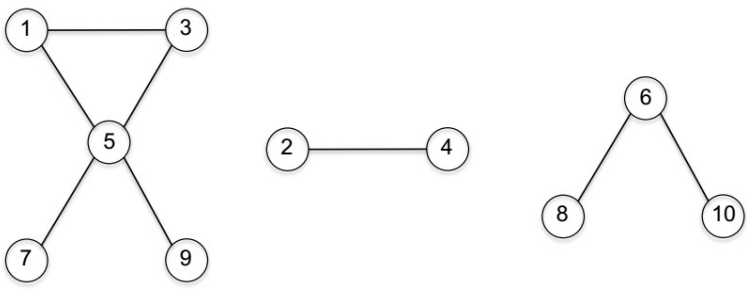




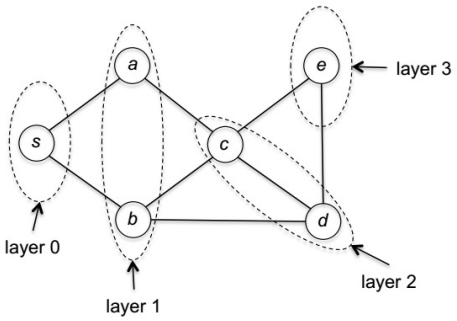

# BFS를 이용한 연결 여부 확인

- 모든 연결 component(부분 그래프)를 구하는 알고리즘

```
components := []
set visited[] to all false
for i := 1 to n do
    if not visited[i] then
        // i를 출발노드로 방문 가능한 모든 노드를
        // 리스트로 구축하여 반환하도록 BFS를 수정
        components.add(BFS(G, visited, i));
```



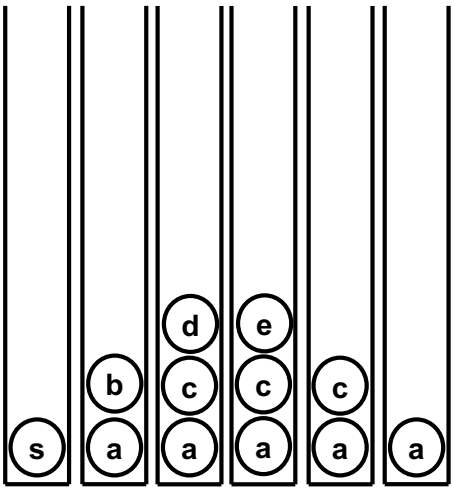
# 깊이 우선 탐색(DFS)




- DFS
  - 계속 아래로 탐색 진행
  - 알고리즘

```
s := starting node
S := empty stack of nodes
visited[s] := true and visited[x] := false for rest
S.push(s)
while S is not empty do // O(n)
    v := S.pop()
    for each edge (v, w) do // O(m)
        if not visited[w] do
            visited[w] := true
            S.push(w)
```

```
DFS(G, s) // recursive version
visited[s] := true
for each edge (s, w) do
    if not visited[w] then
        DFS(G, w)
```

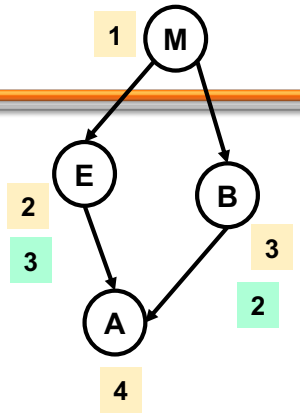





# DFS 공간복잡도

- 재귀 버전 vs. 비재귀 버전, $|V| = n$
  - 재귀 버전은 재귀 호출 깊이에 의존
    - 최악의 경우: 출발 노드부터 모든 노드가 선형적으로 연결되어 있다면 호출 깊이가 $n$에 비례함
  - 비재귀 버전은 스택 공간에 의존
    - 최악의 경우: 다른 모든 노드가 출발 노드의 이웃 노드이면 첫 반복에서 $n-1$개 노드가 스택에 push됨
      - 실제 사용된 공간은 스택 구현 방법에 따라 차이가 있을 수 있음

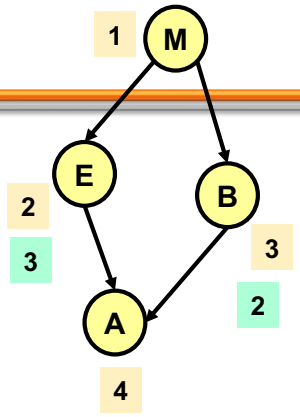
# 위상 정렬 (1/4)




- 방향 그래프에 대한 위상 정렬(topological sort)
  - 다음을 만족하도록 그래프 노드에 대한 명명하는 것
    - $f(v)$는 1, 2, …, $n$ 중 중복 없이 하나의 값을 가짐
    - $(u, v) \in G \rightarrow f(u) < f(v)$
- 위상 정렬은 여러 가지 응용에서 사용됨
  - 예) 대학 교과 선이수체계도 위상 정렬임
- 방향 그래프에서 주기가 없어야 위상 정렬을 할 수 있음
  - 이 외에 위상 정렬을 못하게 하는 요소는 없음
  - 주기가 없는 방향 그래프를 DAG(Directed Acyclic Graph)라 함
- DAG에는 항상 source 노드가 있음
  - source 노드: 진입 차수가 0인 노드. 예) M
  - sink 노드: 진출 차수가 0인 노드. 예) A



# 위상 정렬 (2/4)

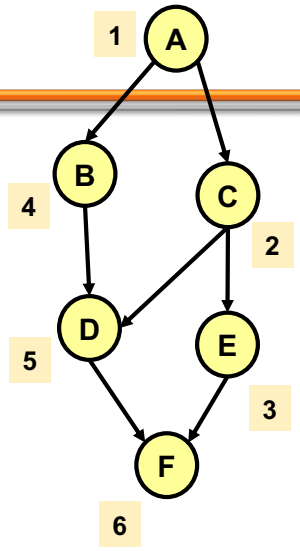



- 입력. DAG $G$, $|G| = n$
- 출력. 모든 노드를 1부터 $n$까지 수로 명명
- 알고리즘 1.
  - topologicalSort($G$, $o = 1$)
    - $G$가 빈 그래프이면 종료
    - 소스 노드에 $o$를 할당함
      - 여러 개가 있을 경우 그 중 하나를 임의로 선택
    - 소스 노드와 소스 노드에서 진출하는 간선들을 제거함.
      이 그래프를 $G'$라 하자.
      - $G'$도 DAG임 (노드와 간선의 제거는 주기를 만들 수 없음)
    - 재귀적으로 topologicalSort($G'$, $o + 1$)를 실행함

- 문제 1. 소스 노드를 어떻게 찾나?
  - 인접 행렬: 노드의 열 정보를 통해
  - 인접 리스트: 역 인접 리스트가 없으면 불편함
- 문제 2. 매번 그래프에 대한 수정 필요

# 위상 정렬 (3/4)

- 알고리즘 2. DFS 응용 알고리즘
  - 재귀적 방법

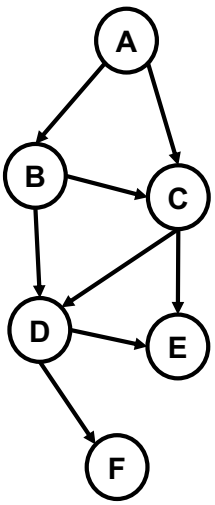



```
topologicalSort(G){
    order := empty list
    set all visited[i] to false
    for each node v in G do
        if not visited[v] then
            topologicalSort(G, order, visited, v)
    return order
}
```

```
topologicalSort(G, order, visited, currNode){
    visited[currNode] := true;
    for each node next in G do
        if currNode ≠ next and not visited[next]
            and graph.hasEdge(currNode, next) then
            topologicalSort(G, order, visited, next)
    add currNode to order
}
```

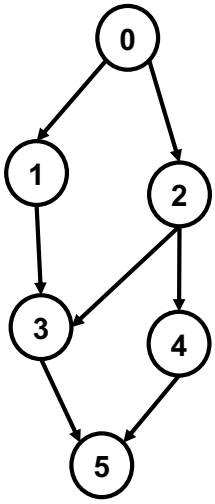
- DFS 방문하는 순을 조금 응용하면






| A | B | C | D | E | F |
|---|---|---|---|---|---|
| | | | | | |

```
topologicalSort(G, order, visited, currNode){
    visited[currNode] := true;
    for each node next in G do
        if currNode ≠ next and not visited[next]
            and graph.hasEdge(currNode, next) then
            topologicalSort(G, order, visited, next)
    add currNode to order
}
```




| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | 4 | 5 | | |
| | 2 | 3 | 3 | 3 | |
| 0 | 1 | 1 | 1 | 1 | 1 |

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| T | | | | | |
| T | T | T | | | |
| T | T | T | T | T | |
| T | T | T | T | T | T |

```
s := starting node
visited[s] := true
S := empty stack of nodes
S.push(s)
while S is not empty do // O(n)
    v := S.pop()
    for each edge (v, w) do // O(m)
        if not visited[w] do
            visited[w] := true
            S.push(w)
```

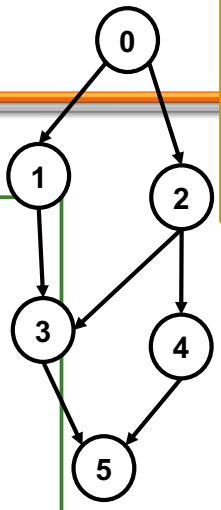
- 더 이상 진행할 수 없는 노드가 5인 것은 알 수 있음 (for문에서 새롭게 스택에 추가하는 노드가 없음)
- 그 다음은? 이미 처리한 노드는 스택에서 제거하였음
- 어떤 경로로 5를 방문했는지 거꾸로 거슬러 올라가면서 번호를 할당해야 하는데, 현재의 스택과 visited 배열을 이용해서는 처리하기 힘듦
- pop한 노드를 다시 스택에 push하자. 그러면 for 루프를 다시 반복해야 함
- 그것을 다시 반복하지 않고 두 번째 pop한 것인지 알 수 있으면 좋겠음



# 위상 정렬 (4/4)

- 비재귀적 방법

```
currentLabel := n
for each node v in G do
    if not visited[v] then do
        S := empty stack
        Q := empty queue
        S.push(v)
        visited[x] := true
        while S is not empty do
            x := S.pop()
            if x < 0 then Q.add(−x − 1)
            else
                S.push(−x − 1)
                for each edge (x, w) do
                    if not visited[w] then
                        visited[w] := true
                        S.push(w)
        while Q is not empty do
            x := Q.poll()
            label[x] := currentLabel
            currentLabel −= 1
```

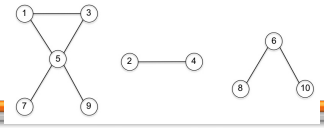



- DFS를 수행하면서 pop한 노드 값을 음수로 바꾸어 다시 push하여 번호 할당 순서를 파악함
- 다른 방법은? visited 배열을 정수 배열로

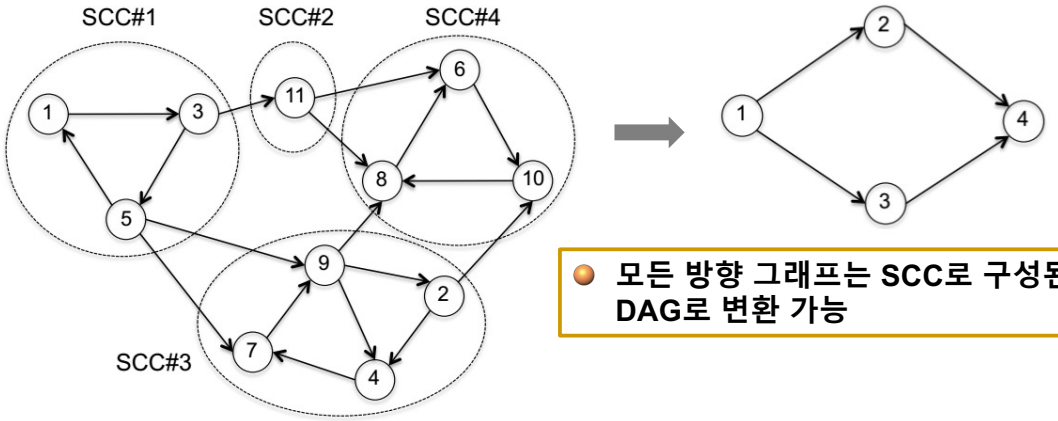
| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | 5 | -6 |
| | | | 4 | -5 | -5 |
| | | | 3 | 3 | 3 |
| | | 2 | -3 | -3 | -3 |
| | | 1 | 1 | 1 | 1 |
| 0 | -1 | -1 | -1 | -1 | -1 |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| -5 | | | | | |
| 3 | 3 | -4 | | | |
| -3 | -3 | -3 | -3 | | |
| 1 | 1 | 1 | 1 | 1 | -2 |
| -1 | -1 | -1 | -1 | -1 | -1 |

# 강한 연결 요소 (1/2)



- 강한 연결 요소(SCC, Strongly Connected Components)
  - 방향 그래프 $G = (V, E)$에서 SCC란 모든 노드 간의 상호 경로가 존재하는 $G$의 부분그래프를 말함
    - $S \subseteq V$가 SCC이기 위한 필요충분조건은 $u, v \in S$이면 $u$에서 $v$, $v$에서 $u$로 경로가 모두 존재해야 함
  - 참고. 무방향 그래프에서 연결 요소란?

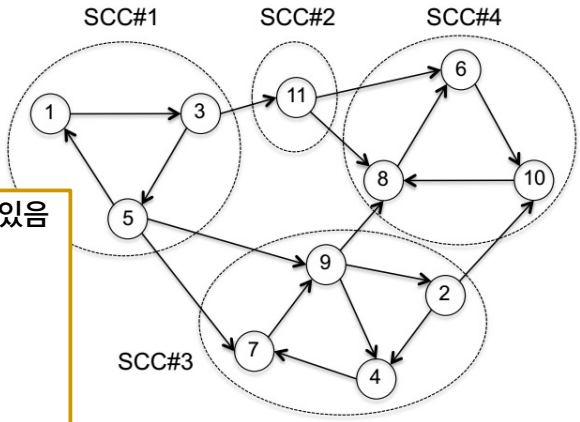



- 모든 방향 그래프는 SCC로 구성된 DAG로 변환 가능



# 강한 연결 요소 (2/2)

- DFS을 이용하면 강한 연결 요소를 구할 수 있을까?
- 아래 그래프에서 노드 10, 노드 4, 노드 1에서 DFS를 수행하면 얻게 되는 노드를 생각하여 보자
  - 노드 10: 10, 8, 6 ⟹ SCC
  - 노드 4: 4, 7, 9, 2, 8, 6, 10
  - 노드 1: 전체




- 적절한 곳(예: 노드 10)에서 시작하면 SCC를 구할 수 있음
- 하지만 SCC#1의 경우 어떤 노드에서 시작하더라도 해당 SCC를 구할 수는 없음
- SCC#4를 제외한 다른 SCC는 어떤 노드에서 DFS를 수행하더라도 SCC를 구할 수 없음
- 일반 방향 그래프를 SCC의 DAG로 변환하였을 때 sink 노드(SCC#4)에 해당하는 부분 그래프에서 시작하여 해당 부분 그래프를 제거하는 형태로 진행하면 SCC를 모두 구할 수 있음
- Sink SCC에 포함된 노드부터 DFS를 수행해야 하고, 그것을 통해 확보한 SCC 노드를 제외하고 그다음 Sink가 되는 노드를 이용하여 DFS를 수행해야 함.
  이 순서를 어떻게 확보? 위상정렬

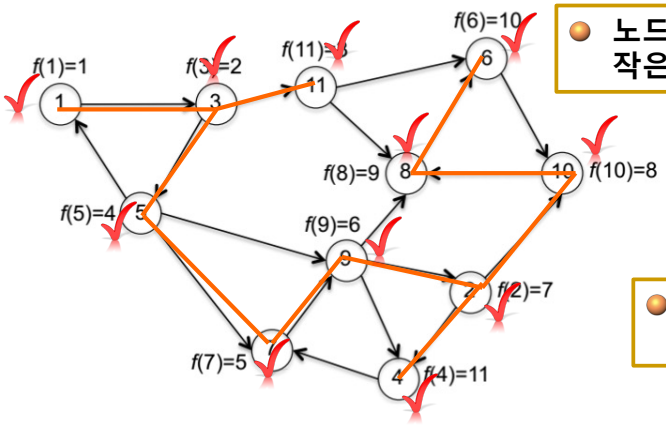
# Kosaraju의 Two-pass 알고리즘 (1/4)

- 알고리즘 (시간복잡도: $O(m+n)$)
  - 단계 1. 모든 그래프의 간선의 방향을 거꾸로 함
  - 단계 2. DFS 수행 (실제 위상 정렬을 수행)
    - 어떤 순서를 만들어 줌
  - 단계 3. 원래 그래프에 대한 DFS 수행
    - 단계 2에서 만든 순서로 수행하면 SCC를 발견할 수 있음
- 그런데 왜 역방향 그래프에서 위상 정렬을 수행?
- 간선의 방향을 거꾸로 하여 DFS를 수행하기 위해 실제 방향을 바꾸어야 하나?
  - 새 그래프를 만들지 않고 수행 가능
  - 인접 리스트는 역 인접 리스트 이용, 인접 행렬은 행과 열을 거꾸로 생각하여 구현

- 위상정렬은 소스부터 싱크 순으로 정렬
- 우리는 거꾸로 진행해야 함
- DFS 수행한 후 내림차순으로 DFS하면 되는 것 아닌가?
- 원 방향으로 위상정렬를 하면 싱크 SCC에 있는 노드가 항상 가장 큰 값을 받지 않을 수 있음
- 이를 위해 간선의 방향을 바꾸어 위상정렬을 하고 그 순서를 기반으로 DFS 진행 (다음 슬라이드)

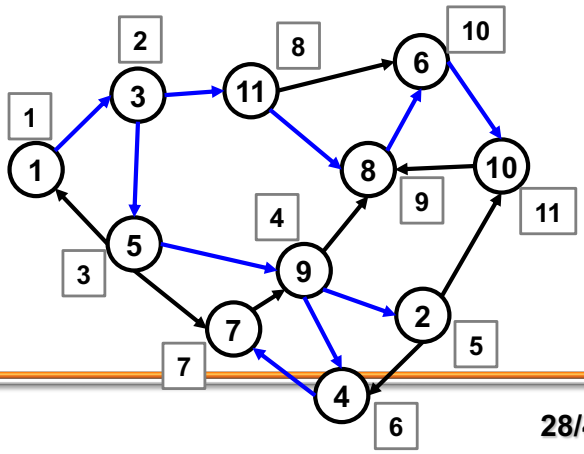


# Kosaraju의 Two-pass 알고리즘 (2/3)



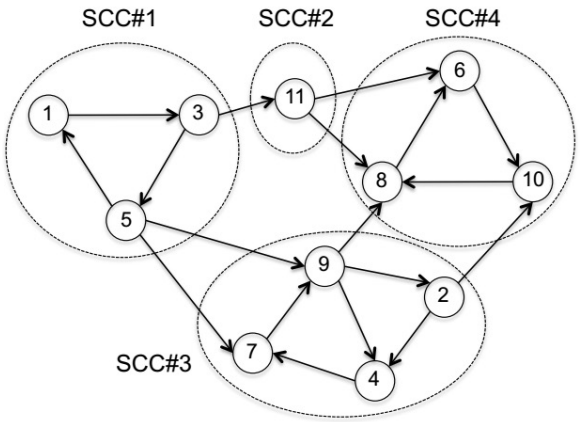

- 노드 1에서 시작하여 위상 정렬한 경우: 작은 번호 노드 먼저

- 노드 1에서 시작하여 위상 정렬한 경우: 큰 번호 노드 먼저

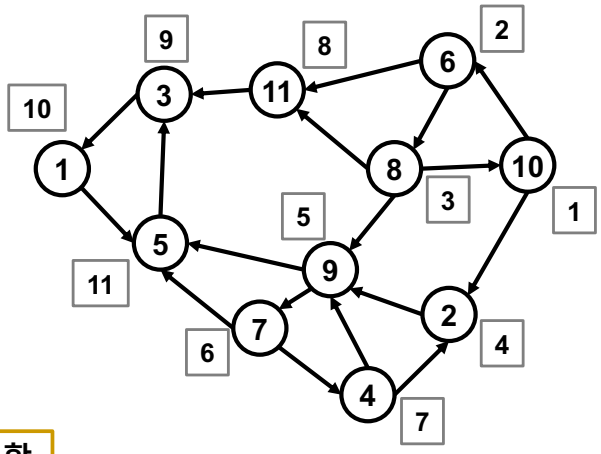


- 가장 큰 번호를 받은 노드가 항상 sink SCC에 소속된 노드는 아님
- 하지만 1번 노드는 항상 소스 SCC에 소속된 노드임
- 어떻게 하면 sink SCC에 소속된 노드가 가장 작은 또는 가장 큰 값이 되도록 할 수 있을까? 역방향 그래프

# Kosaraju의 Two-pass 알고리즘 (3/4)




- 간선의 방향을 모두 바꾼 후 위상 정렬을 함



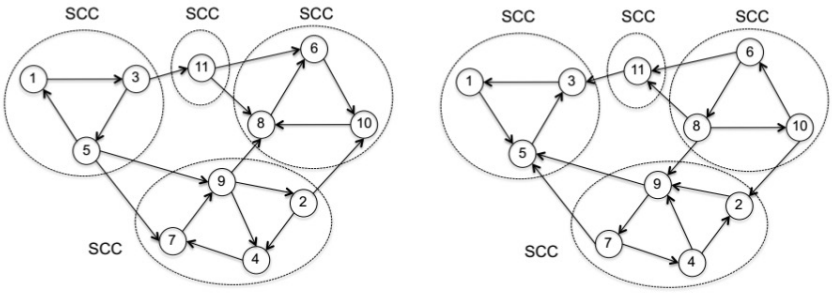

- 11에서 시작한 경우 위상정렬 결과

- 거꾸로 위상 정렬한 후에
  위상 정렬 순서로 DFS 수행하면
  SCC를 구할 수 있음



# Kosaraju의 Two-pass 알고리즘 (4/4)

- 방향성 그래프 $G$와 그것의 역방향 그래프 $G^R$이 주어졌을 때, 다음 중 성립하는 것은?
  - $G$의 SCC는 $G^R$의 SCC이고, 그것의 역도 성립함
  - $G$의 sink SCC는 $G^R$의 source SCC가 됨




(a) Original graph (b) Reversed graph

# 최단 경로 문제

- 최단 경로 문제의 분류
  - 분류 1. 한 노드에서 특정 노드까지 최단 경로 찾기
  - 분류 2. 한 노드에서 다른 모든 노느까지 최단 경로 찾기
    - SSSP(Single-Source Shortest Path) 문제
    - 비가중치 그래프: BFS 알고리즘
    - 양의 가중치 그래프: Dijkstra 알고리즘
    - 음의 가중치 존재 그래프: Bellman-Ford 알고리즘 (Note 11)
  - 분류 3. 모든 노드에서 다른 모든 노드까지 최단 경로 찾기
    - APSP(All-Pairs Shortest Path) 문제
    - 음의 가중치 존재 그래프: Floyd-Warshall 알고리즘 (Note 11)
- 분류 1 문제를 이용하여 분류 2를 해결할 수 있고,
  분류 2 문제를 이용하여 분류 3을 해결할 수 있음
  - 하지만 분류 2는 분류 2를 해결하는 전용 알고리즘을 이용하고, 분류 3은 분류 3을 해결하는 전용 알고리즘을 이용하는 것이 더 효과적임

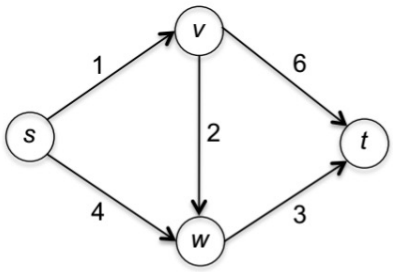


# 다익스트라 최단 경로 알고리즘 (1

ACM Turing Award 1972

- 문제
  - 입력. 방향 가중치 그래프 $G = (V, E)$, 출발노드 $s$
    - 각 가중치는 0과 같거나 큼
    - 무방향도 가능
  - 출력. 각 노드마다 $s$부터 해당 노드까지의 최단 경로의 길이
  - 가정. 모든 노드까지의 경로가 존재함
    - 경로가 존재하지 않아도 문제가 되지 않음
      - BFS를 통해 제거할 수 있음
- 모든 가중치가 같을 경우 BFS를 통해 최단 경로 길이를 구할 수 있음

- 예) 0,1,3,6



# 다익스트라 최단 경로 알고리즘 (2/3)

- 알고리즘

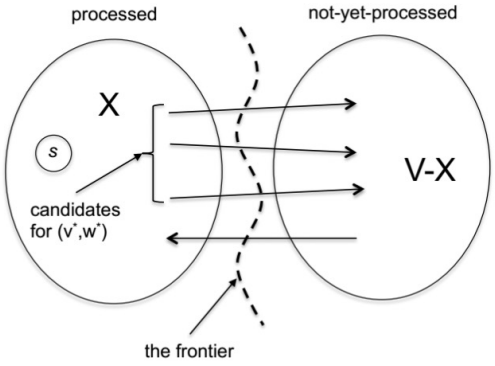
$X \coloneqq \{s\}$
$A[s] \coloneqq 0$
$B[s] \coloneqq$ *empty path*
***while*** $X \neq V$ ***do***
  $(v^*, w^*) \coloneqq$ ***the*** $(v, w) \in E$ ***with*** $v \in X$ ***and*** $w \notin X$ ***that minimizes*** $A[v] + D[v][w]$
  ***add*** $w^*$ ***to*** $X$
  $A[w^*] \coloneqq A[v^*] + D[v^*][w^*]$
  $B[w^*] \coloneqq B[v^*] + (v^*, w^*)$

$v$까지 최단 경로 길이

$w$까지 최단 경로 길이

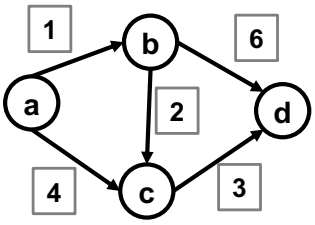
경로 구축

- 다음에 학습할 탐욕적 알고리즘의 대표적인 예






# 다익스트라 최단 경로 알고리즘 (3/3)

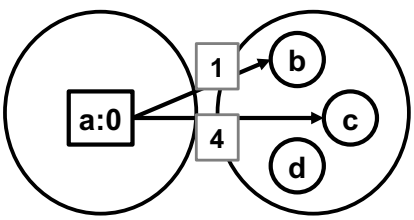


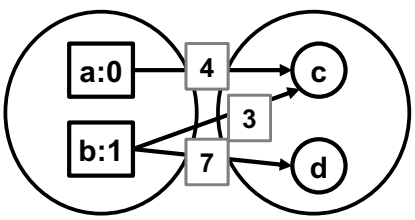

- $X = \{a\}$
- $A[a] = 0$
- $B[a] = \_$

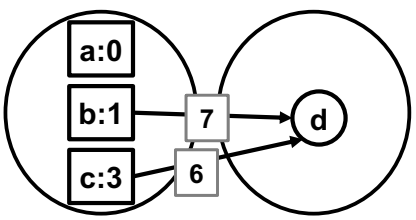
- $A[a] + (a, b) = 1$
- $A[a] + (a, c) = 4$
- $A[b] = 1$
- $B[b] = a$
- $X = \{a, b\}$

- $A[b] + (b, d) = 7$
- $A[b] + (b, c) = 3$
- $A[a] + (a, c) = 4$
- $A[c] = 3$
- $B[c] = a, b$
- $X = \{a, b, c\}$

- $A[b] + (b, d) = 7$
- $A[c] + (c, d) = 6$
- $A[d] = 6$
- $B[d] = a, b, c$
- $X = \{a, b, c, d\}$



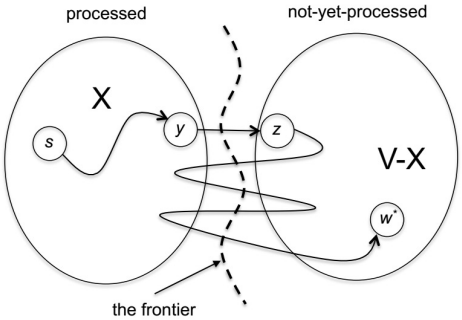
# 다익스트라 알고리즘 정확성 증명 (1/2)

- **정리.** 다익스트라 알고리즘은 가중치가 음수가 아닌 모든 가중치 방향 그래프에 대해 출발노드부터 모든 노드까지의 최단 경로 길이를 계산하여 줌
  - $L[v]$: $s$에 $v$까지의 실제 최단 경로의 길이
  - $A[v]$: 다익스트라 알고리즘을 통해 구한 $s$에 $v$까지의 최단 경로의 길이
  - 이 정리에 의하면 모든 $v$에 대해 $A[v] = L[v]$임
- 증명) 귀납법
  - 귀납법 출발: $A[s] = L[s] = 0$
  - 귀납가정: 지금까지 완료한 모든 $v$에 대해 $A[v] = L[v]$
  - 귀납절차
    - 이번 반복에서 $(v^*, w^*)$를 선택하여 $w^*$를 $X$에 추가
      - $A[w^*] = A[v^*] + d[v^*][w^*] = L[v^*] + d[v^*][w^*]$
    - 증명해야 하는 것? $L[v^*] + d[v^*][w^*] = L[w^*]$
      - $\forall dist(s, w^*) \geq A[w^*]$



# 다익스트라 알고리즘 정확성 증명 (2/2)

- $\forall \mathbf{dist}(s, w^*) \geq A[w^*]$
  - $s$에서 $w^*$까지 임의의 경로 P는 다음과 같이 표현 가능
    - 이 경로는 최소한 경계를 한 번 지나가야 함
    - 3가지 요소로 경로를 나눌 수 있음
    - $\mathbf{path}(s, y)$: $L[y] = A[y]$ (귀납가정)
    - $(y, z)$: $d[y][z]$
    - $\mathbf{path}(z, w^*) \geq 0$: 가중치는 0보다 크기 때문)
    - $\mathbf{dist}(\mathrm{P}) \geq A[y] + d[y][z]$




  - $A[w^*] = A[v^*] + d[v^*][w^*]$이므로 $\mathbf{dist}(\mathrm{P}) \geq A[y] + d[y][z] \geq A[w^*]$임
    - 알고리즘 규칙에 의해 $A[z]$와 같은 것들 중 가장 최소가 $A[w^*]$

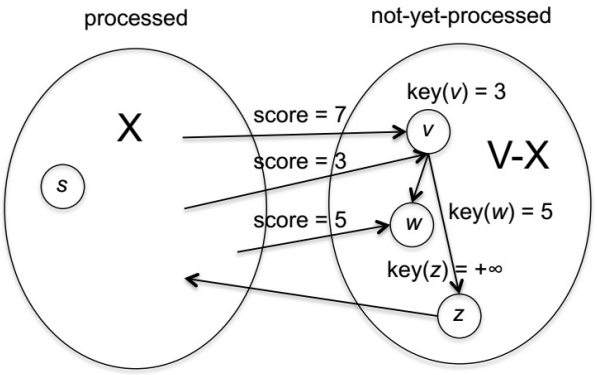
# 힙 기반 다익스트라 알고리즘 (1/5)

- 특별한 자료구조를 사용하지 않고 구현하면 시간복잡도는?
  - $O(mn)$
- 이 알고리즘에서 비용이 많이 소요되는 부분은?
  - 경계를 건너가는 간선 중 다익스트라 점수가 가장 낮은 것을 찾는 부분
  - 어떤 자료구조를 사용하면 좋을까? 힙(heap)
    - 힙에 무엇을 유지
      - 방법 1. 간선
      - 방법 2. 노드
    - 보통 간선이 노드보다 많음
    - 다익스트라 점수는 노드가 기준임



# 힙 기반 다익스트라 알고리즘 (2/5)

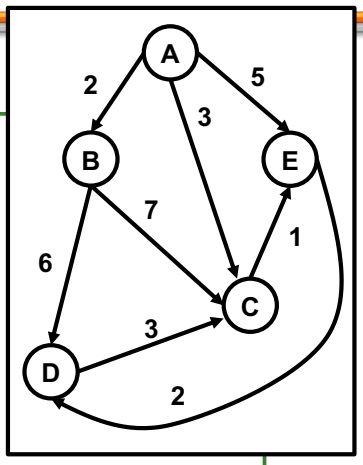
- 힙에 아직까지 처리하지 못한 노드 유지
- 키는 다익스트라 알고리즘에서 다익스트라 최소 점수
  - $w \in V - X$의 키는 $v \in X$에서 $w$가는 간선 중 다익스트라 점수가 최소인 점수를 키로 사용
  - How?



# 힙 기반 다익스트라 알고리즘 (3/5)

- 알고리즘

```
X := ∅
H := empty heap
for every node v do     // nlogn
    key(v) := 0 if v = s else ∞
    insert v into H
while H ≠ ∅ do
    w := H.extractMin()          // logn
    add w to X
    pathDistance[w] := key(w)
    // 이번에 추가된 노드만 검사
    for every edge (w, y) where y ∈ V − X do
        delete y from H      // logn
        key(y) := min(key(y), pathdistance[w] + d[w][y])
        insert y into H      // logn
return pathDistance
```




| A | B | C | D | E |
|---|---|---|---|---|
| 0 | 99 | 99 | 99 | 99 |

| B | C | E | D |
|---|---|---|---|
| 2 | 3 | 5 | 99 |

| C | E | D |
|---|---|---|
| 3 | 5 | 8 |

| E | D |
|---|---|
| 4 | 8 |

| D |
|---|
| 6 |

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | 2 | 3 | 6 | 4 |



# 힙 기반 다익스트라 알고리즘 (4/5)

- $X$와 $V - X$ 집합의 표현
  - 방법 1. visited boolean 배열
  - 방법 2. 집합 자료구조 활용
- 힙은 보통 삭제 연산을 제공하지 않음
  - 삭제 연산을 제공하는 힙을 만들어 다익스트라 알고리즘을 구현할 수 있음
    - 실제 삭제 연산보다 우선순위 갱신 연산을 만들어 사용함
- 다른 방법
  - 삭제 없이 계속 추가만 함 (Lazy Dijkstra Algorithm)
    - 초기에 모든 노드를 힙에 추가할 필요가 없음
    - 다 처리하여도 큐가 빈 상태가 아닐 수 있음
    - 큐에 유지하는 요소가 증가하기 때문에 연산 비용이 증가함

# 힙 기반 다익스트라 알고리즘 (5/5)

- 시간 복잡도
  - Outer for: $n \log n$
  - while: $n$번 반복
    - while 문 내부: $\log n$ (extract min)
    - for 문: 복잡. 하지만 간선 기준으로 보면 간선은 최대 한번만 고려됨
      - $2m\log n$
    - $n \log n + m\log n$
  - 전체: $n \log n + m\log n = (n + m)\log n$