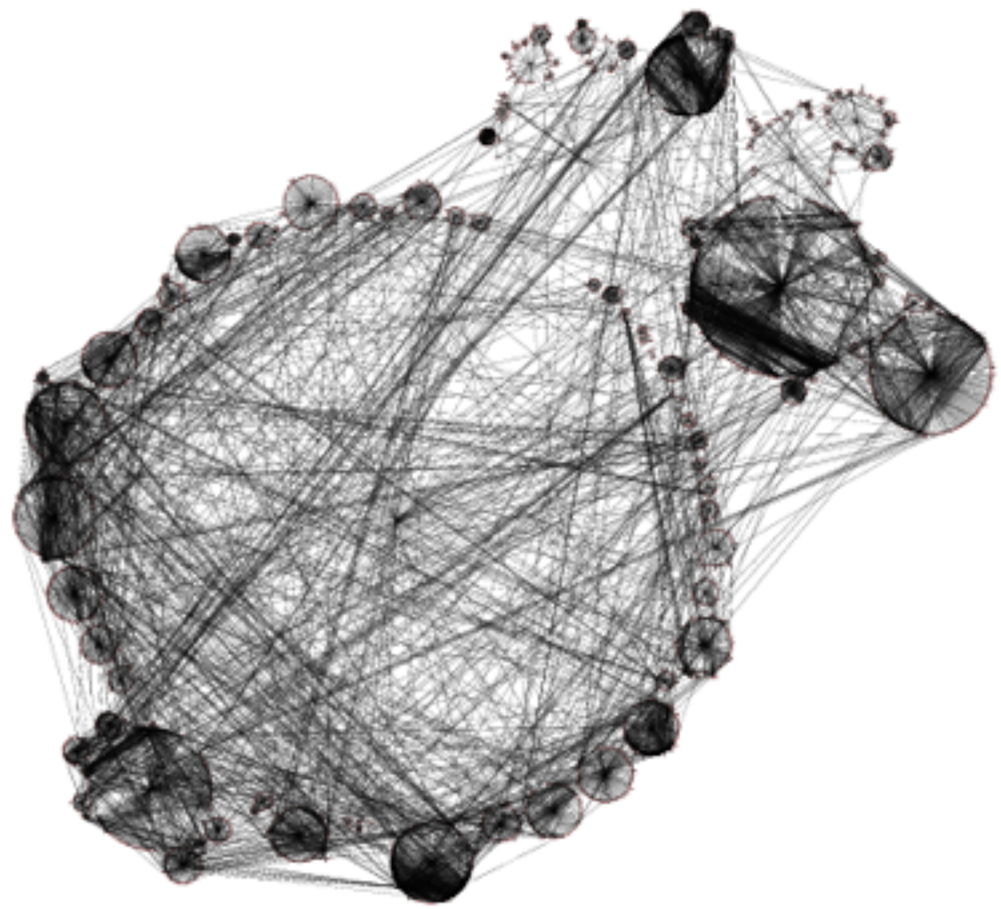


제6장

그래프 알고리즘 III

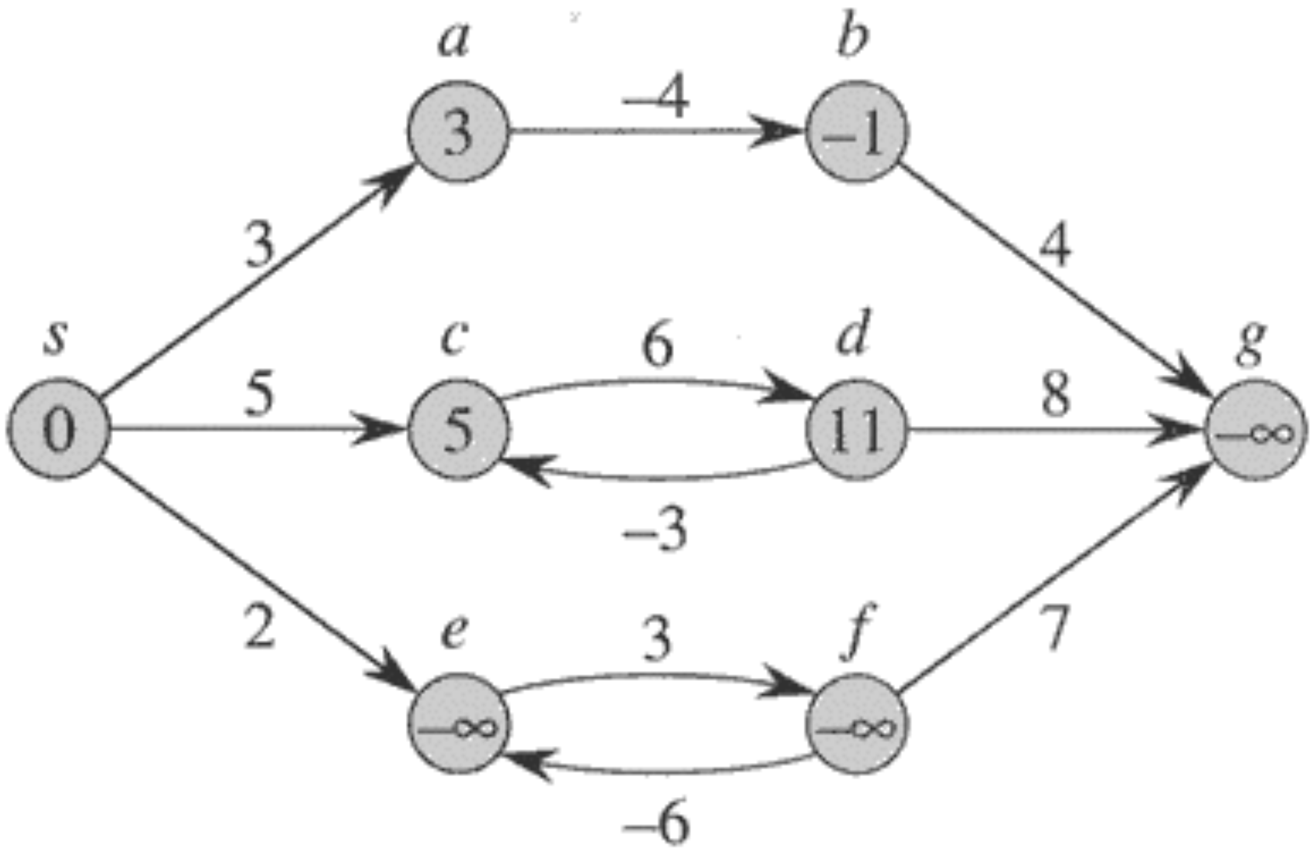
최단경로
Shortest Path

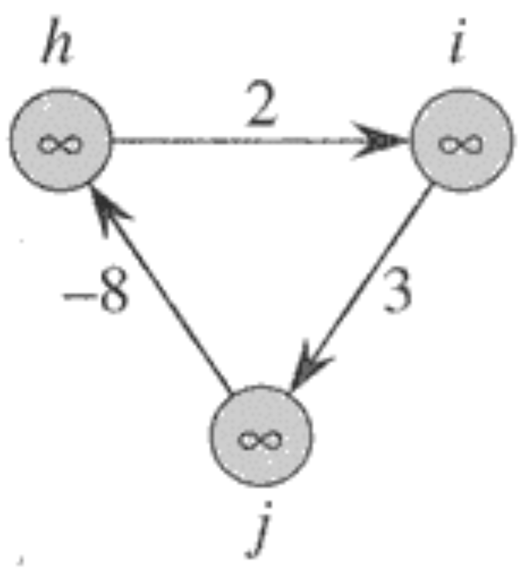
- 가중치 (방향) 그래프 G=(V,E), 즉 모든 에지에 가중치가 있음
- 경로 $p=(v_0, v_1, ..., v_k)$의 길이는 경로상의 모든 에지의 가중치의 합
- 노드 u에서 v까지의 최단경로의 길이를 $\delta(u,v)$라고 표시하자.

최단경로문제의 유형

- Single-source:
  - 하나의 출발 노드 s로부터 다른 모든 노드까지의 최단 경로를 찾아라.
  - 예: Dijkstra의 알고리즘
- Single-destination:
  - 모든 노드로부터 하나의 목적지 노드까지의 최단 경로를 찾아라.
  - Single-source 문제와 동일
- Single-pair:
  - 주어진 하나의 출발 노드 s로 부터 하나의 목적지 노드 t까지의 최단 경로를 찾아라
  - 최악의 경우 시간복잡도에서 Single-source 문제보다 나은 알고리즘이 없음
- All-pairs:
  - 모든 노드 쌍에 대해서 최단 경로를 찾아라.

최단경로와 음수 가중치

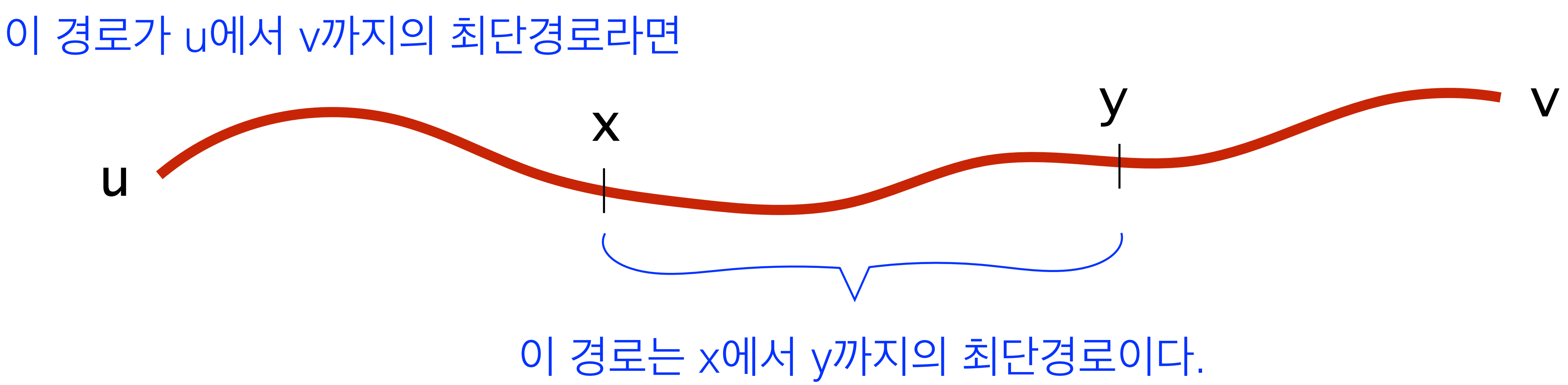




음수 사이클(negative cycle)이 있으면 최단 경로가 정의되지 않음

알고리즘에 따라 음수 가중치가 있어도 작동하는 경우도 있고 그렇지 않은 경우도 있음

- 최단 경로의 어떤 부분경로도 역시 최단 경로이다.

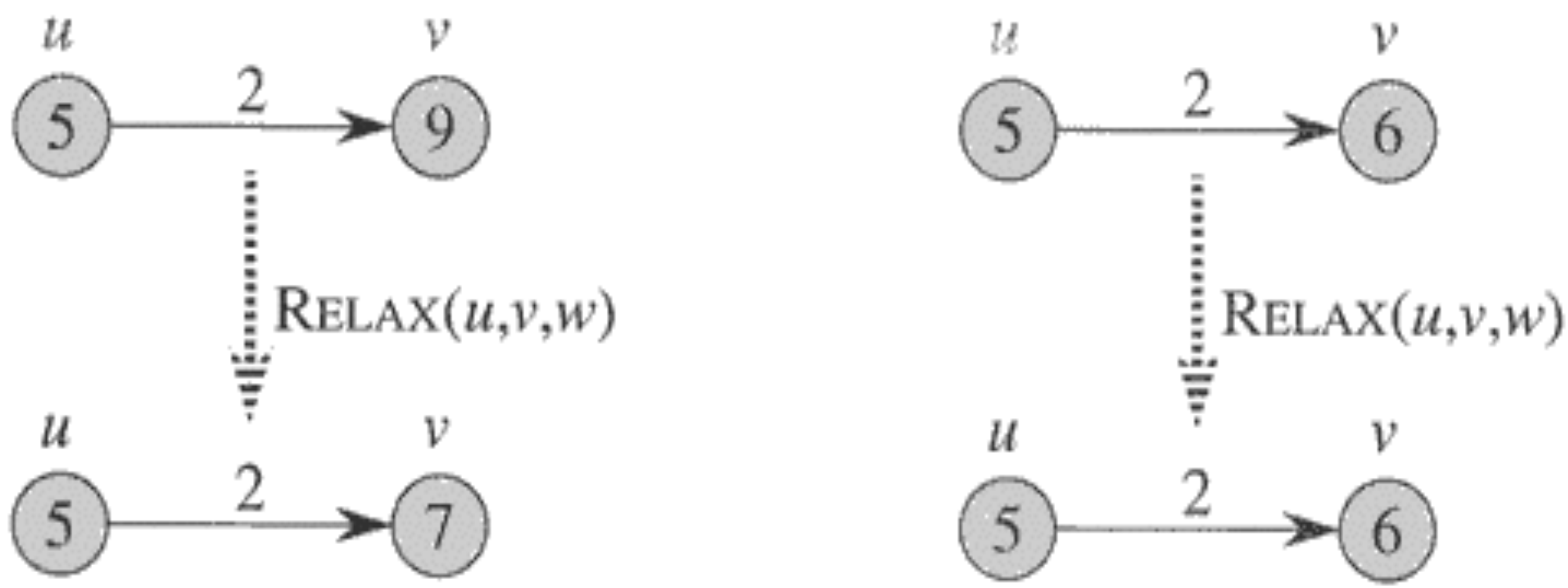


- 최단 경로는 사이클을 포함하지 않는다. (음수 사이클이 없다는 가정하에서)

Single-source 최단경로문제

- 입력: 음수 사이클이 없는 가중치 방향그래프 $G=(V,E)$와 출발 노드 $s \in V$
- 목적: 각 노드 $v \in V$에 대해서 다음을 계산한다.
  - d[v]
    - 처음에는 $d[s]=0$, $d[v]=\infty$로 시작한다.
    - 알고리즘이 진행됨에 따라서 감소해간다. 하지만 항상 $d[v] \geq \delta(s,v)$를 유지한다
    - 최종적으로는 $d[v]=\delta(s,v)$가 된다.
  - $\pi[v]$: s에서 v까지의 최단경로상에서 v의 직전 노드(predecessor)
    - 그런 노드가 없는 경우 $\pi[v]$=NIL.

기본 연산: Relaxation



(a) (b)

RELAX(u, v, w)

if $d[v] > d[u] + w(u, v)$
    then $d[v] \leftarrow d[u] + w(u, v)$
        $\pi[v] \leftarrow u$

Single-source 최단경로

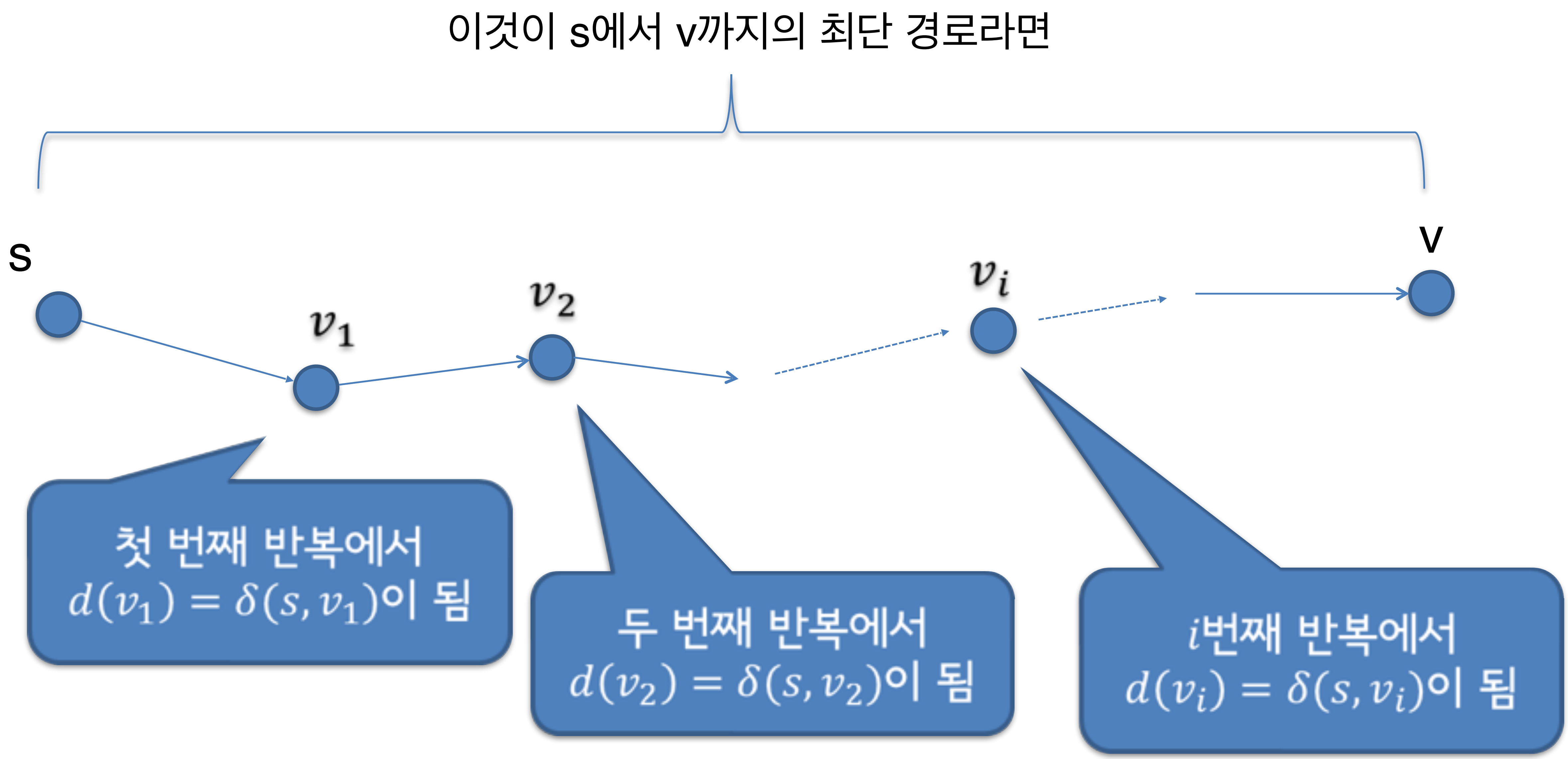
- 대부분의 single-source 최단경로 알고리즘의 기본 구조
  1. 초기화: d[s]=0, 노드 v≠s에 대해서 d[v]=∞,π[v]=NIL.
  2. 에지들에 대한 반복적인 relaxation
- 알고리즘들 간의 차이는 어떤 에지에 대해서, 어떤 순서로 relaxation을 하느냐에 있음

Generic-Single-Source($\boldsymbol{G}, \boldsymbol{w}, \boldsymbol{s}$)

1. INITIALISE-SINGLE-SOURCE(G, s)
2. repeat
3. for each edge $(u, v) \in E$
4. RELAX(u, v, w)
5. until there is no change.

질문 2: 몇 번 반복해야 ?

질문 1: 이렇게 계속 반복하면 최단 경로가 찾아지는가?



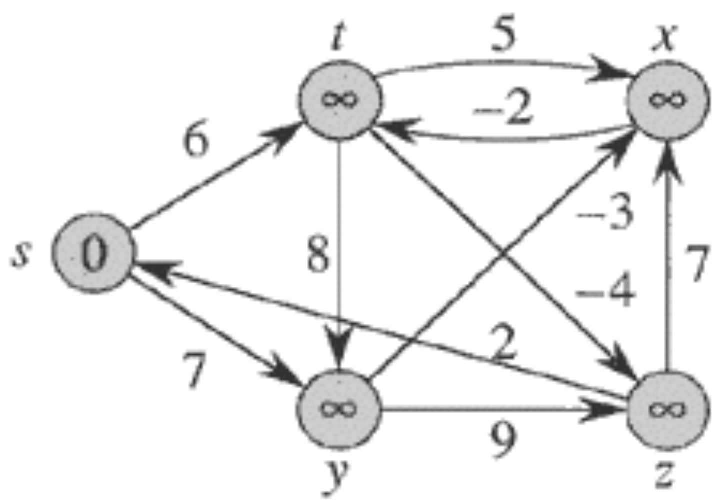
즉, n-1번의 반복으로 충분하다.

Bellman-Ford 알고리즘

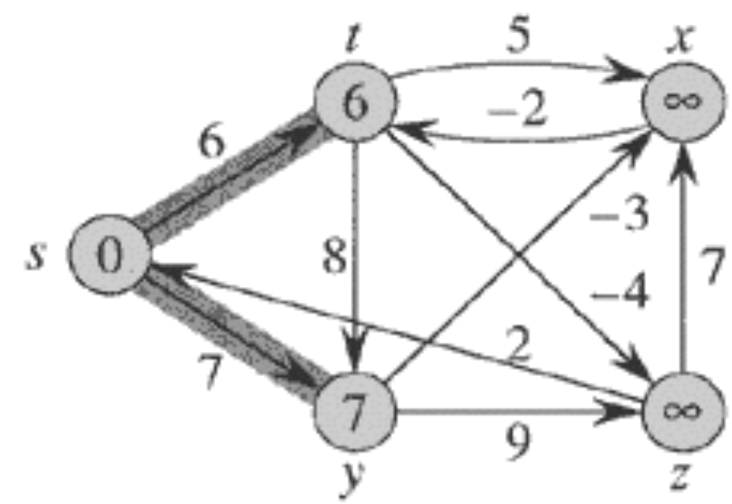
```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| − 1
     do for each edge (u, v) ∈ E[G]
          do RELAX(u, v, w)
for each edge (u, v) ∈ E[G]
     do if d[v] > d[u] + w(u, v)
          then return FALSE
return TRUE
```

음수 사이클이 존재한다는 의미

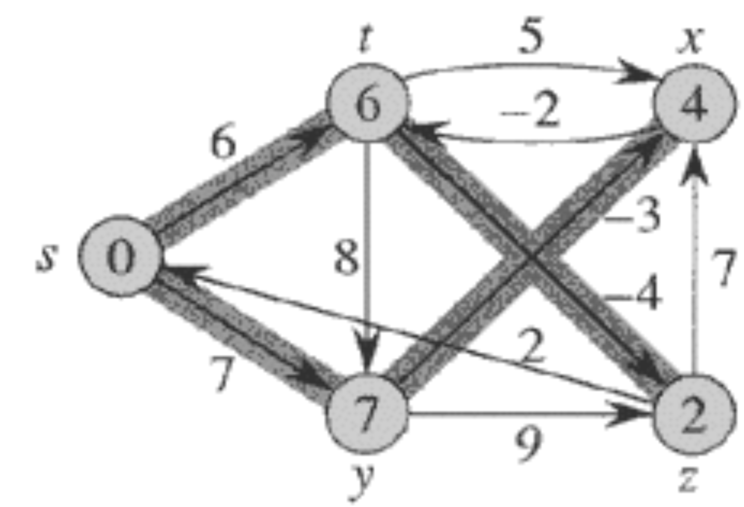
시간복잡도 O(nm)


t
x
s
y
z
5
−2
6
8
−3
−4
7
2
7
9
0
∞
∞
∞
∞

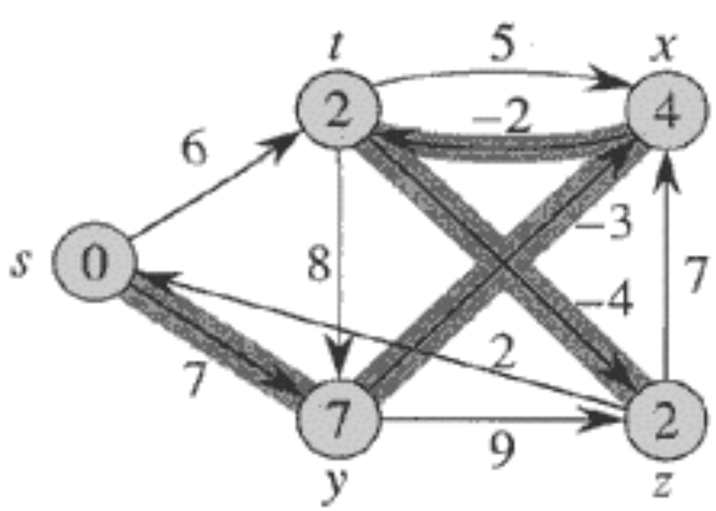
(a)


t
x
s
y
z
5
−2
6
8
−3
−4
7
2
7
9
0
6
∞
7
∞

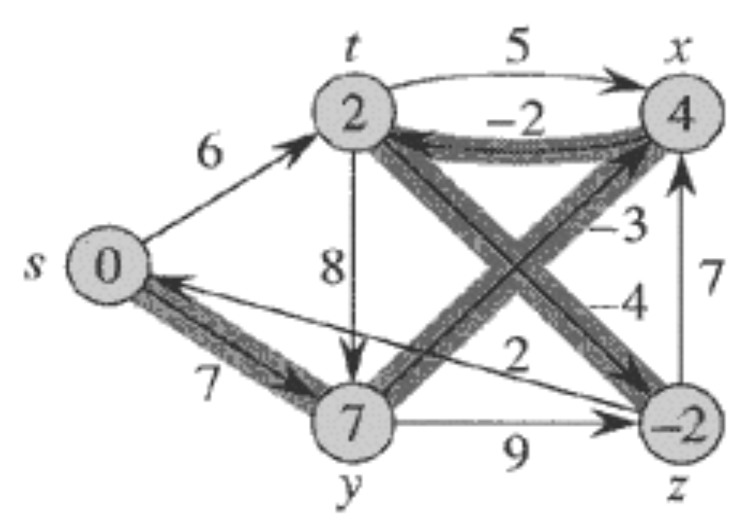
(b)


t
x
s
y
z
5
−2
6
8
−3
−4
7
2
7
9
0
6
4
7
2

(c)

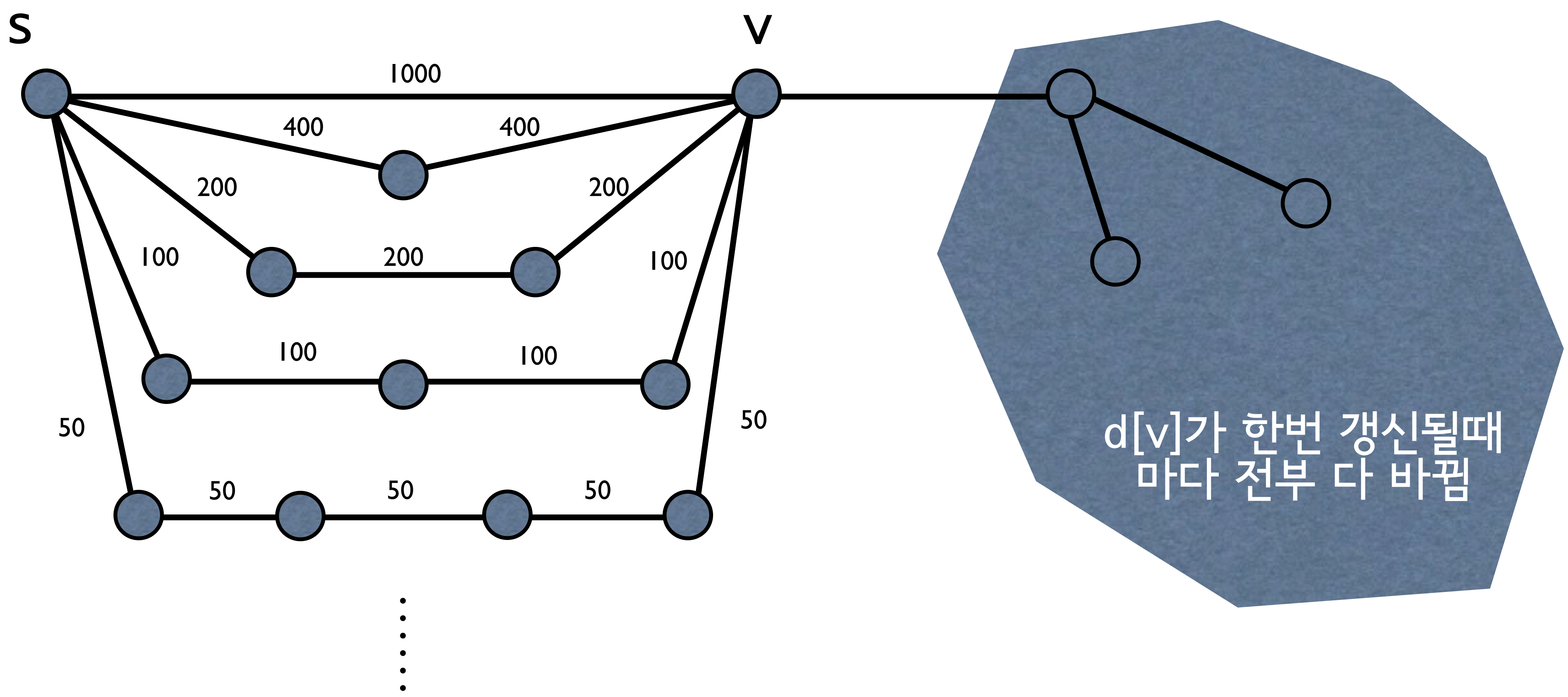

t
x
s
y
z
5
−2
6
8
−3
−4
7
2
7
9
0
2
4
7
2

(d)

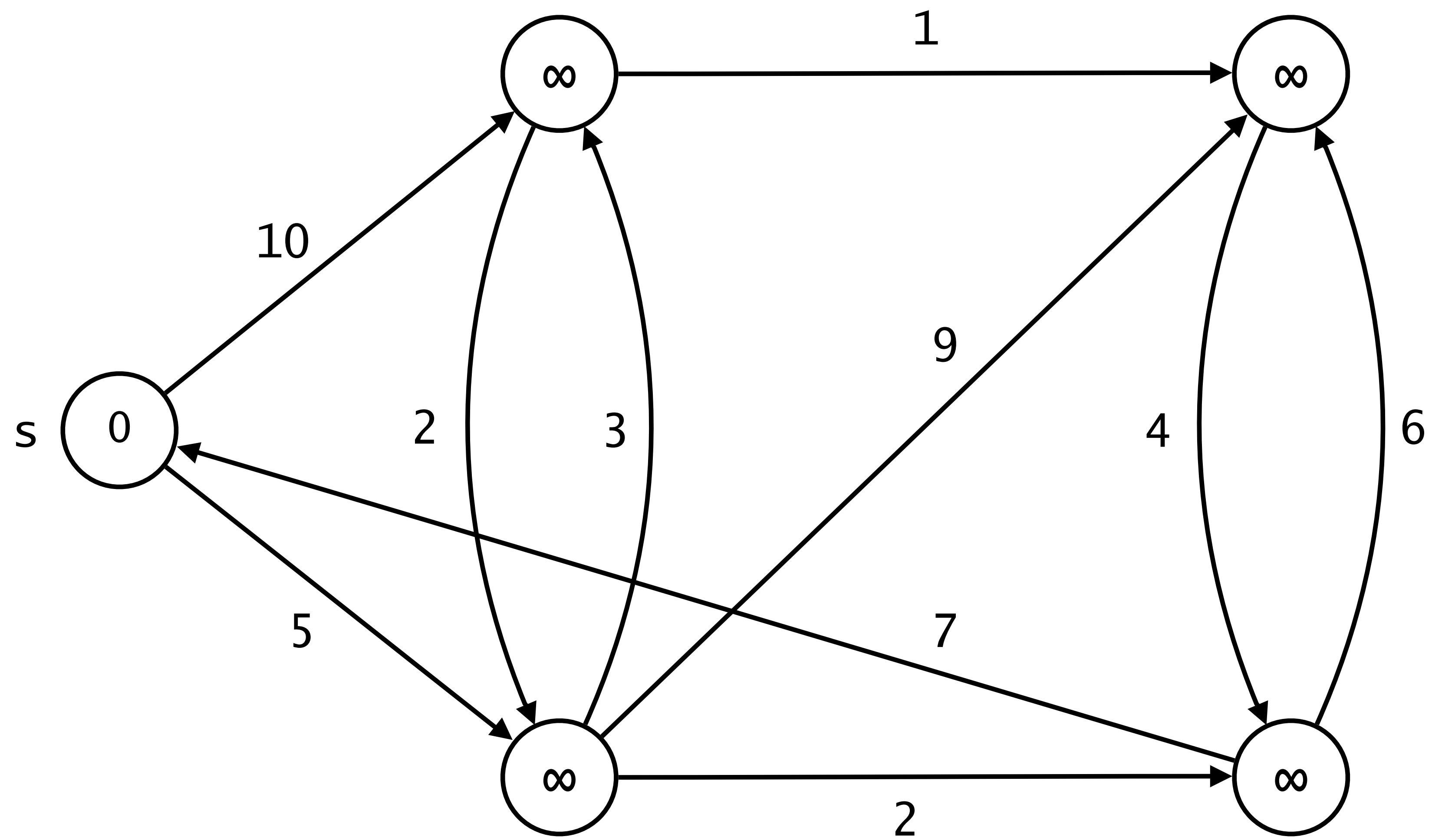

t
x
s
y
z
5
−2
6
8
−3
−4
7
2
7
9
0
2
4
7
−2

(e)

Worst Scenario

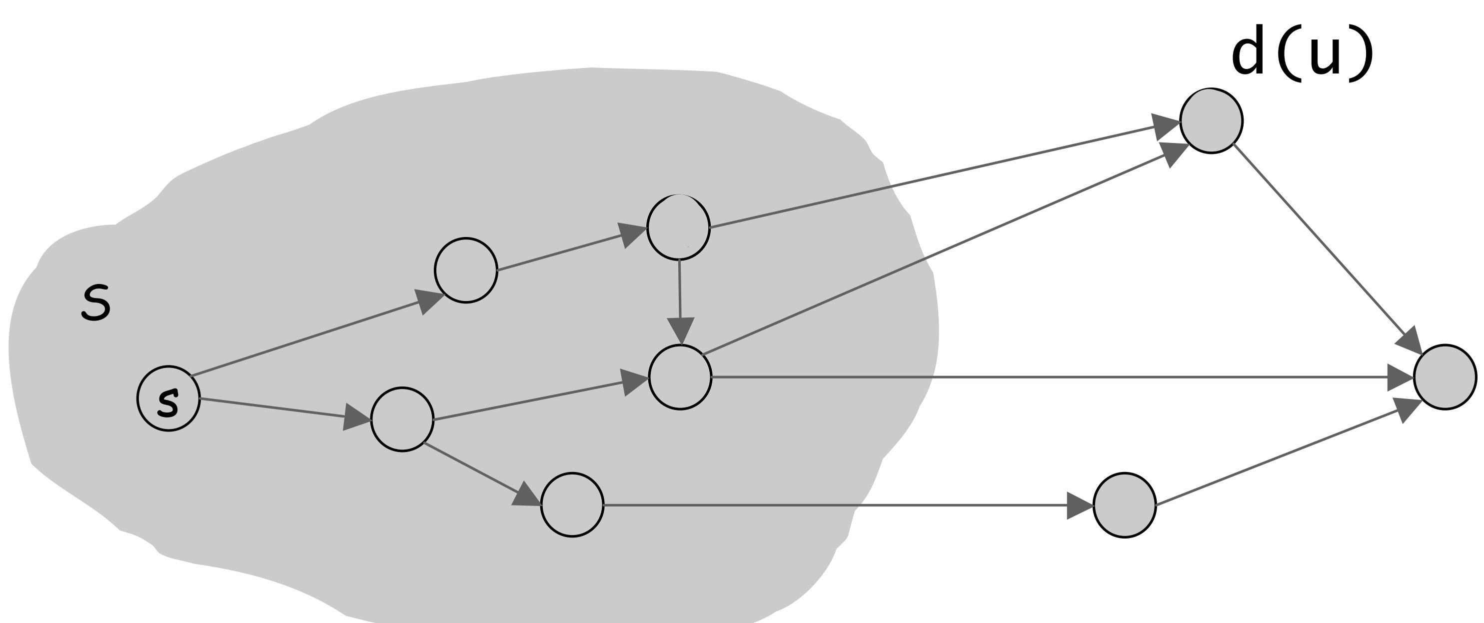


Dijkstra의 알고리즘



Dijkstra의 알고리즘

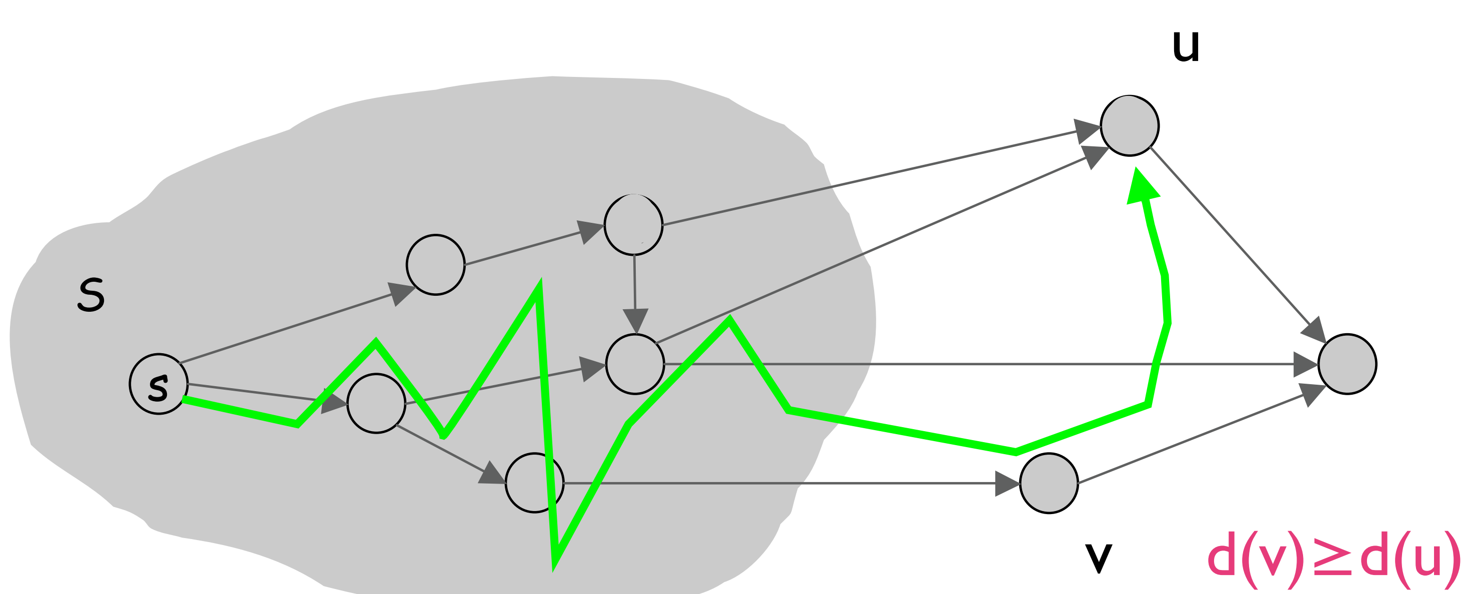
- 음수 가중치가 없다고 가정
- s로부터의 최단경로의 길이를 이미 알아낸 노드들의 집합 S를 유지. 맨 처음엔 S={s}.
- Loop invariant:
  - u∉S인 각 노드 u에 대해서 d(u)는 이미 S에 속한 노드들만 거쳐서 s로부터 u까지 가는 최단경로의 길이



정리: $d(u)=\min_{v\notin S} d(v)$인 노드 u에 대해서, d(u)는 s에서 u까지의 최단경로의 길이이다.

증명: (proof by contradiction)

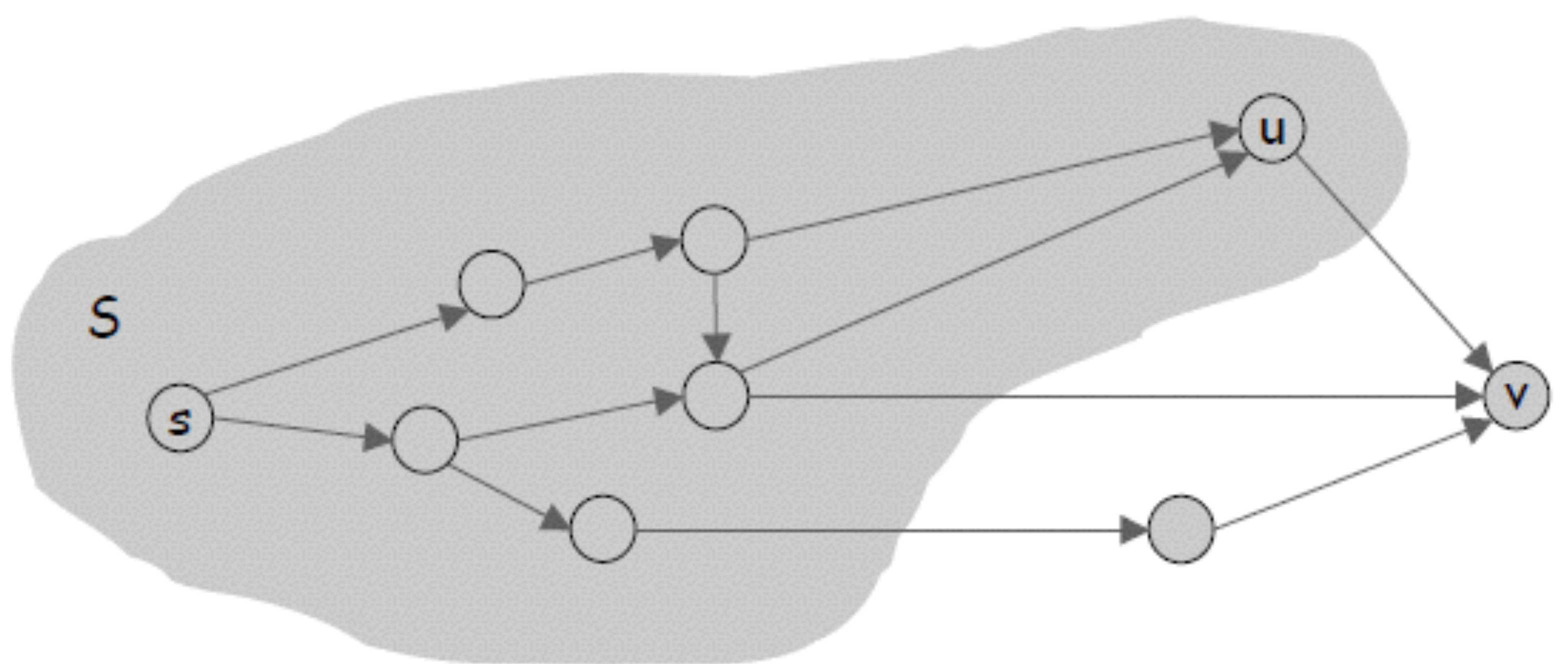
- 아니라고 하자. 그러면 s에서 u까지 다른 최단경로가 존재



- $d(v)\geq d(u)$이므로 모순

Dijkstra의 알고리즘

- d(u)가 최소인 노드 u∉S를 찾고, S에 u를 추가
- S가 변경되었으므로 다른 노드들의 d(v)값을 갱신



$$d(v) = \min\{d(v), d(u) + w(u,v)\}$$

즉, 에지 (u,v)에 대해서 relaxation하면 Loop Invariant가 계속 유지됨

Dijkstra의 알고리즘

```
Gijkstra(G, w, s)
for each u∈V do
    d[u] ← ∞
    π[u] ← NIL
end.
S ← {s}
d[s] ← 0
while |S|<n do
    find u∉S with the minimum d[u] value;
    S ← S ∪ {u}
    for each v∉S adjacent to u do
        if d[v] > d[u]+w(u,v) then
            d[v] ← d[u]+w(u,v)
            π[v] ← u
        end.
    end.
end.
```

while문은 n-1번 반복

최소값 찾기 O(n)

degree(u) = O(n)

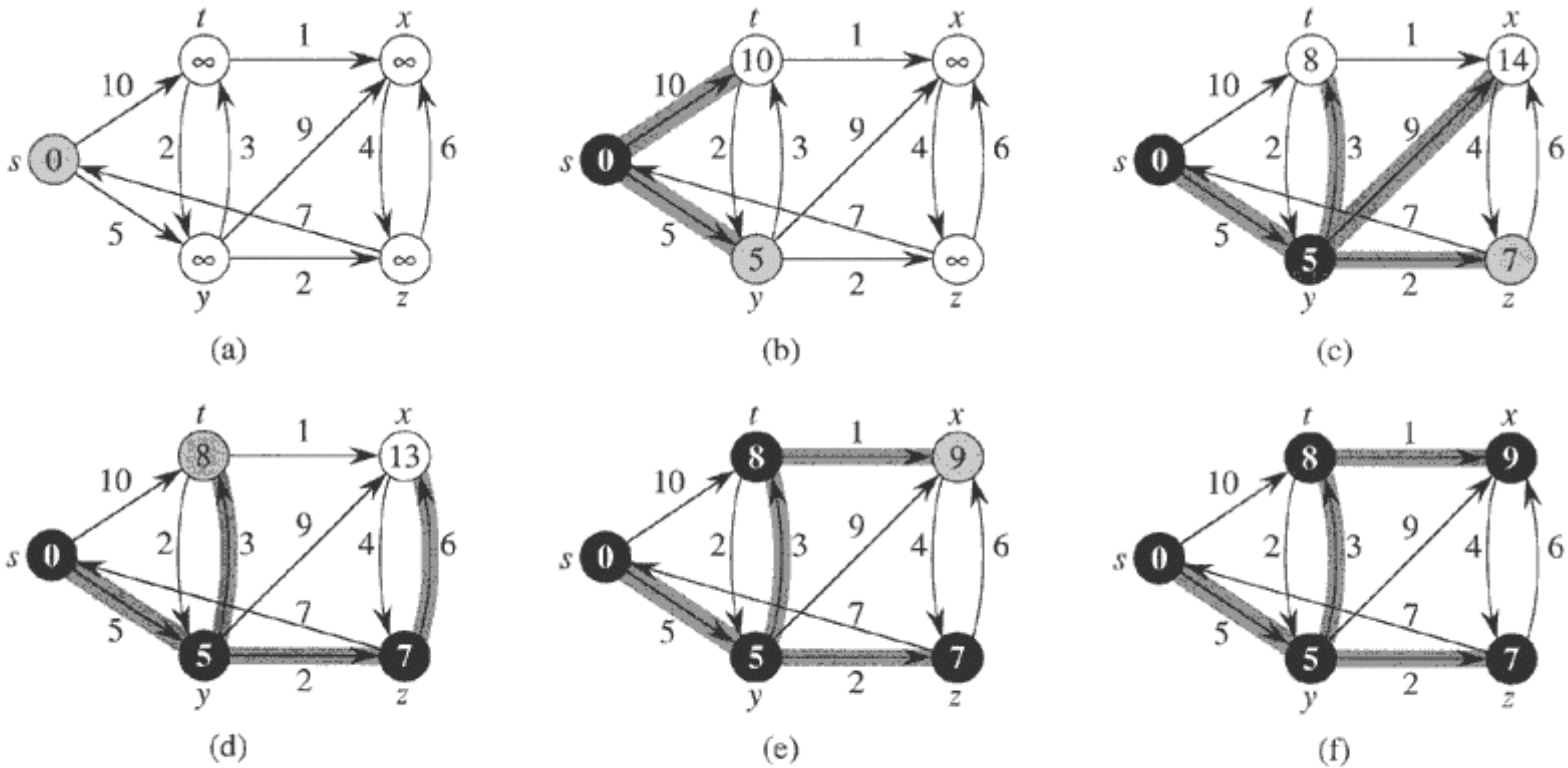
시간복잡도 $O(n^2)$

Dijkstra의 알고리즘

```
DIJKSTRA(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  S ← ∅
  Q ← V[G]
  while Q ≠ ∅
      do u ← EXTRACT-MIN(Q)
         S ← S ∪ {u}
         for each vertex v ∈ Adj[u]
             do RELAX(u, v, w)
```

Q는 최소우선순위큐

Dijkstra의 알고리즘



시간복잡도

- Prim의 알고리즘과 동일함
- 우선순위 큐를 사용하지 않고 단순하게 구현할 경우 $O(n^2)$
- 이진힙을 우선순위 큐로 사용할 경우 $O(n\log_2 n + m\log_2 n)$
- Fibonacci Heap을 사용하면 $O(n\log_2 n+m)$에 구현가능

Floyd-Warshall Algorithm

- 가중치 방향 그래프 G=(V,E), V={1,2,...,n}
- 모든 노드 쌍들간의 최단경로의 길이를 구함
- $d^k[i,j]$
  - 중간에 노드집합 {1,2,...,k}에 속한 노드들만 거쳐서 노드 i에서 j까지 가는 최단경로의 길이

Floyd-Warshall Algorithm

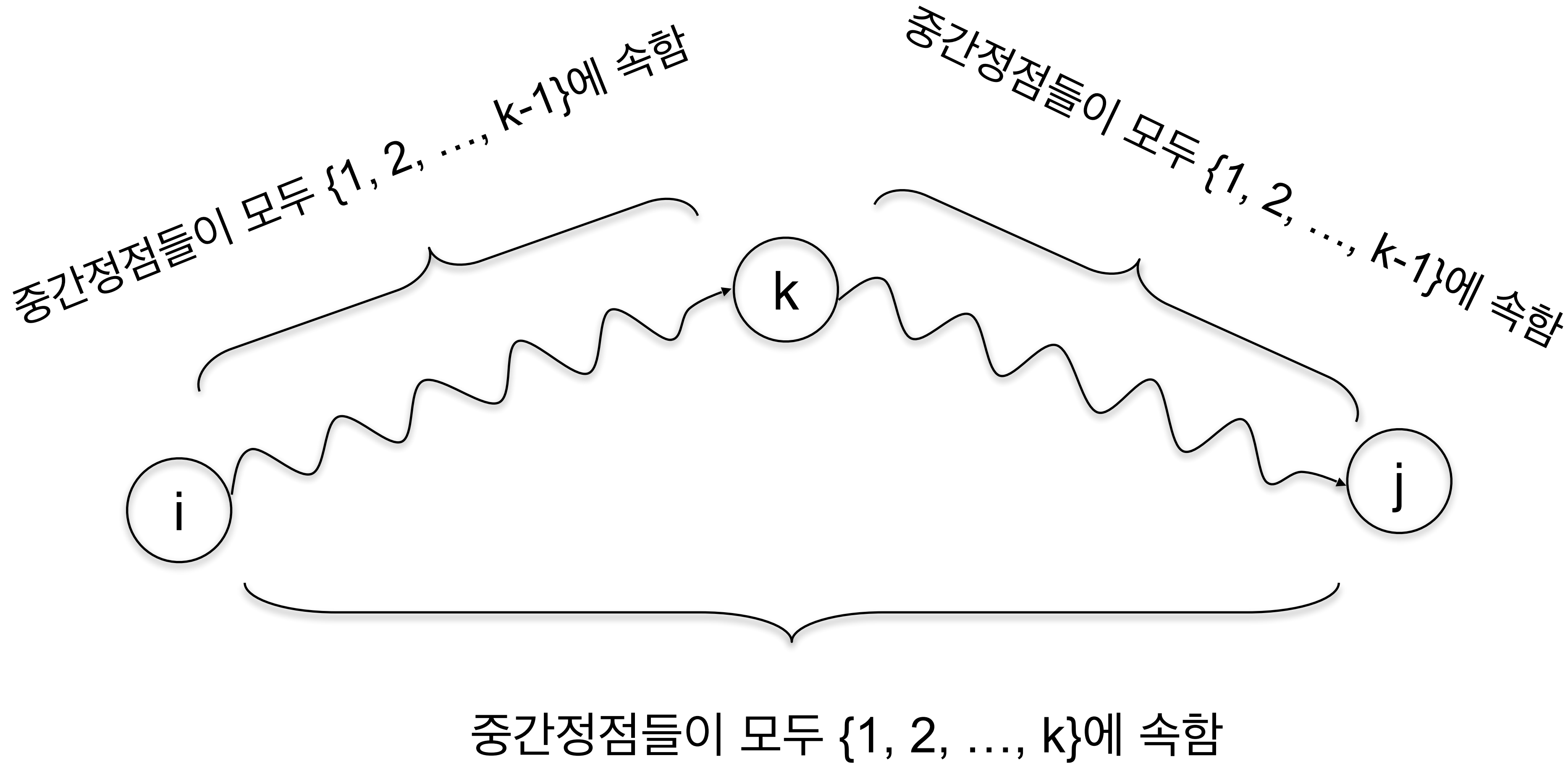
$$d^0[i,j] = \begin{cases} w_{ij}, & \text{if } (i,j) \in E \\ \infty, & \text{otherwise.} \end{cases}$$

$$d^k[i,j] = \min\{d^{k-1}[i,j], d^{k-1}[i,k] + d^{k-1}[k,j]\}$$

$$d^n[i,j] = \delta(i,j)$$

중간에 노드집합 {1,2,...,k}에 속한 노드들만 거쳐서 노드 i에서 j까지 가는 최단경로는 두가지 경우가 있음: 노드 k를 지나는 경우와 지나지 않는 경우

Floyd-Warshall Algorithm



Floyd-Warshall Algorithm

$$d^k[i,j] = \min\{d^{k-1}[i,j], d^{k-1}[i,k] + d^{k-1}[k,j]\}$$

```
FloydWarshall(G)
{
   for i ← 1 to n
      for j ← 1 to n
         d⁰[i,j] ← wij;
   for k ← 1 to n           ▷ 중간정점 집합 {1,2,…,k}
      for i ← 1 to n
         for j ← 1 to n
            dᵏ[i,j] ← min{dᵏ⁻¹[i,j], dᵏ⁻¹[i,k]+dᵏ⁻¹[k,j]};
}
```

시간복잡도: $O(n^3)$

Floyd-Warshall Algorithm

```
FloydWarshall(G)
{
    for i ← 1 to n
        for j ← 1 to n
            d[i,j] ← wij;
    for k ← 1 to n          ▷ 중간정점 집합 {1,2,…,k}
        for i ← 1 to n
            for j ← 1 to n
                d[i,j] ← min{d[i,j], d[i,k]+d[k,j]};
}
```

Why is it okay?

```
FloydWarshall(G)
{
    for i ← 1 to n
        for j ← 1 to n
            d[i,j] ← wij;
            π[i,j] ← NIL;
    for k ← 1 to n          ▷ 중간정점 집합 {1,2,…,k}
        for i ← 1 to n
            for j ← 1 to n
                if d[i,j] > d[i,k]+d[k,j] then
                    d[i,j] = d[i,k]+d[k,j];
                    π[i,j] = k;
}
```

경로 출력하기

s에서 t까지 가는 경로가 존재한다는 가정하에 최단경로상의 중간노드들(s와 t자신은 제외)을 출력함

```
Print-PATH(s, t, π)
{
    if π[s,t]=NIL then
        return;
    print-PATH(s, π[s,t]);
    print(π[s,t]);
    print-PATH(π[s,t], t);
}
```

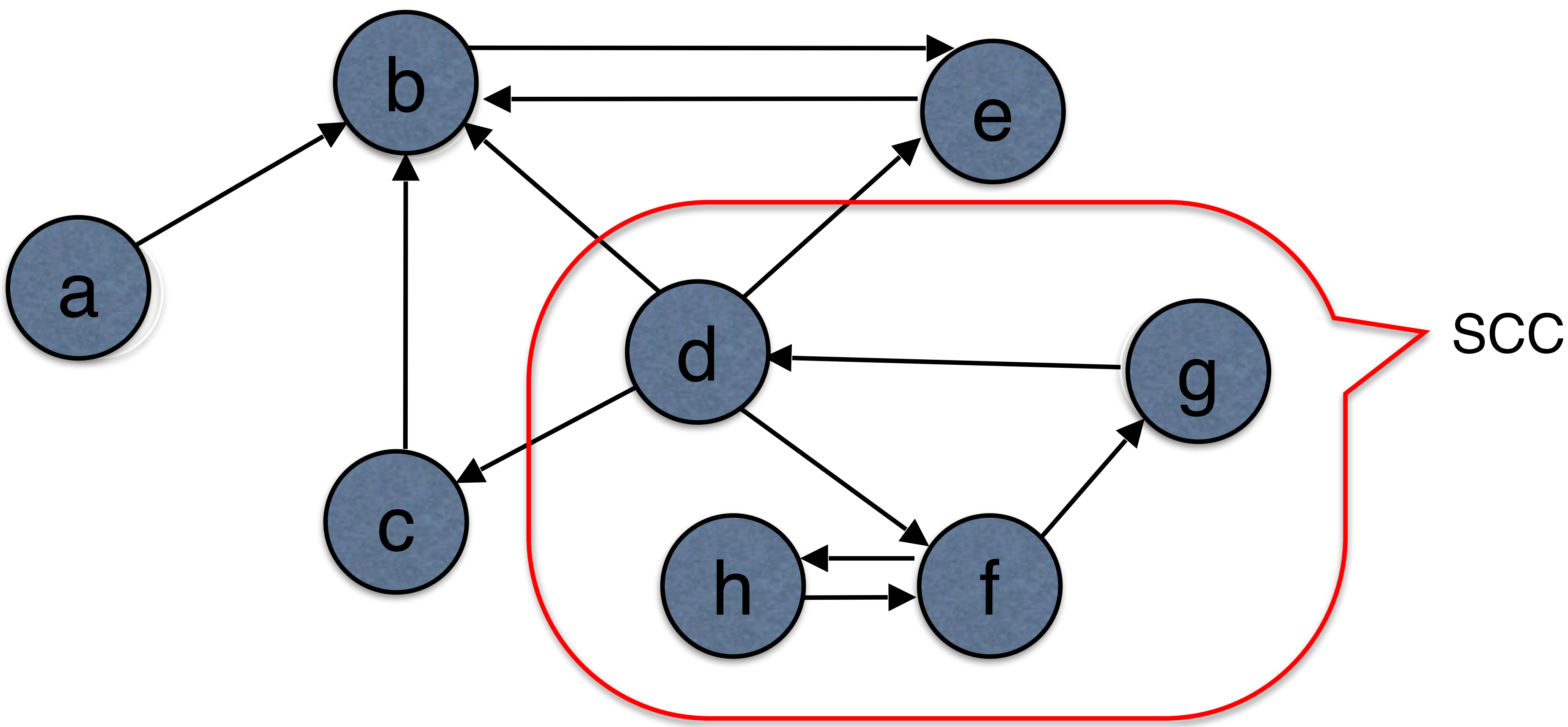
강연결성분
Strongly Connected Components

강연결성분
strongly-connected component

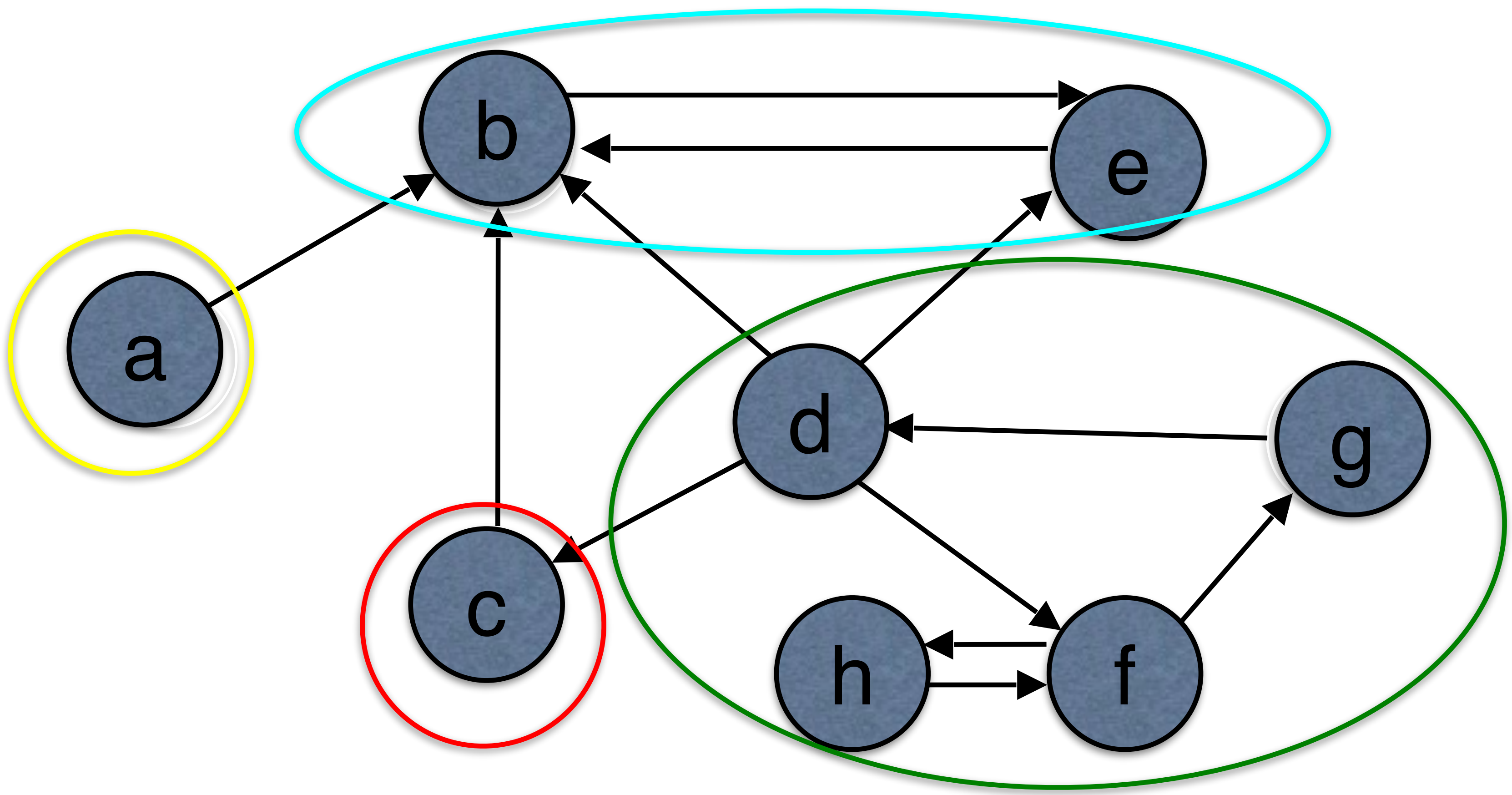
- 그래프 G의 강연결성분(SCC) C는 다음과 같은 조건을 만족하는 부그래프 (subgraph)이다.
- C에 속한 모든 노드들은 서로 reachable하다.
- C에 속하지 않은 어떤 노드도 C에 속한 어떤 노드와 서로 reachable하지 않다.

즉, maximal subset of mutually reachable nodes

강연결성분

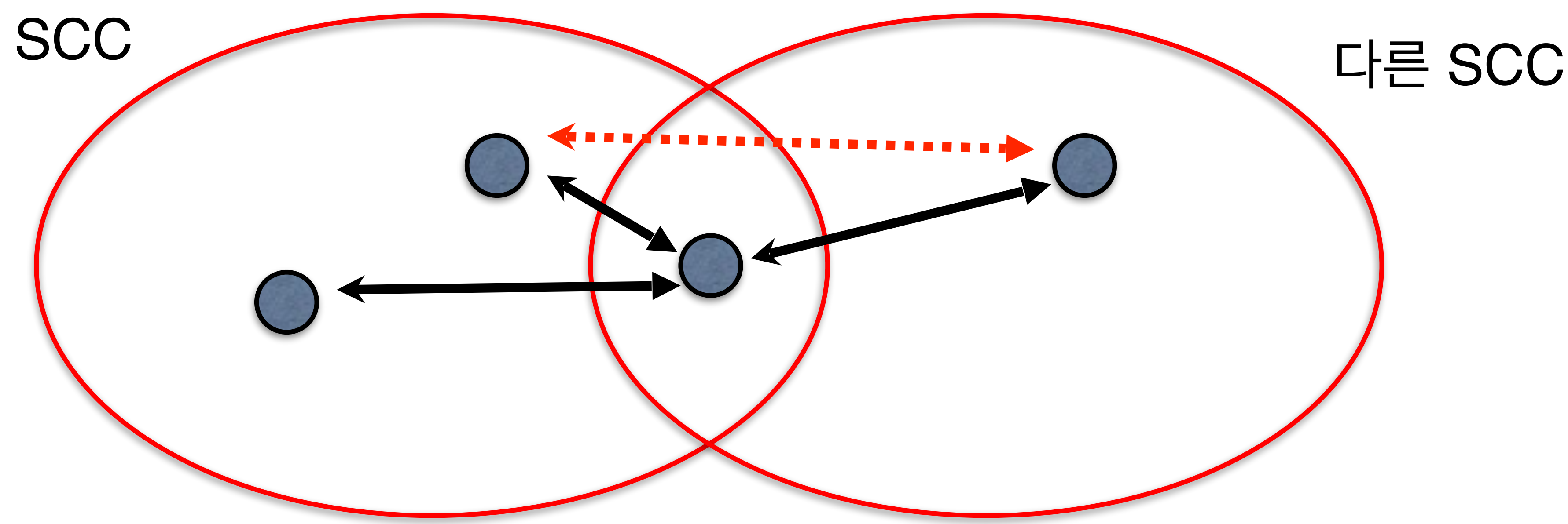


강연결성분



4개의 강연결 성분을 가짐

- 강연결성분들은 서로 disjoint한가?



YES !!

강연결성분들은 그래프의 정점들을 분할(partition)한다.

- 그래프 G^T
  - $G^T=(V,E^T)$, $E^T=\{(u,v) \mid (v,u) \in E\}$
  - $O(n+m)$시간에 G^T의 인접리스트 구성 가능
- 관찰: G와 G^T는 동일한 SCC를 가진다. 즉 임의의 두 노드 u와 v가 G에서 서로 reachable하면 G^T에서도 서로 reachable하다.

- 가장 간단한 알고리즘

```
while there remains a node in G do
    choose a node v in G;
    perform DFS(v) in G ;
    perform DFS(v) in GT;
    find nodes common to both;
    remove them from G;
end.
```

시간 복잡도 ?

DFS revisited

DFS-VISIT(u)

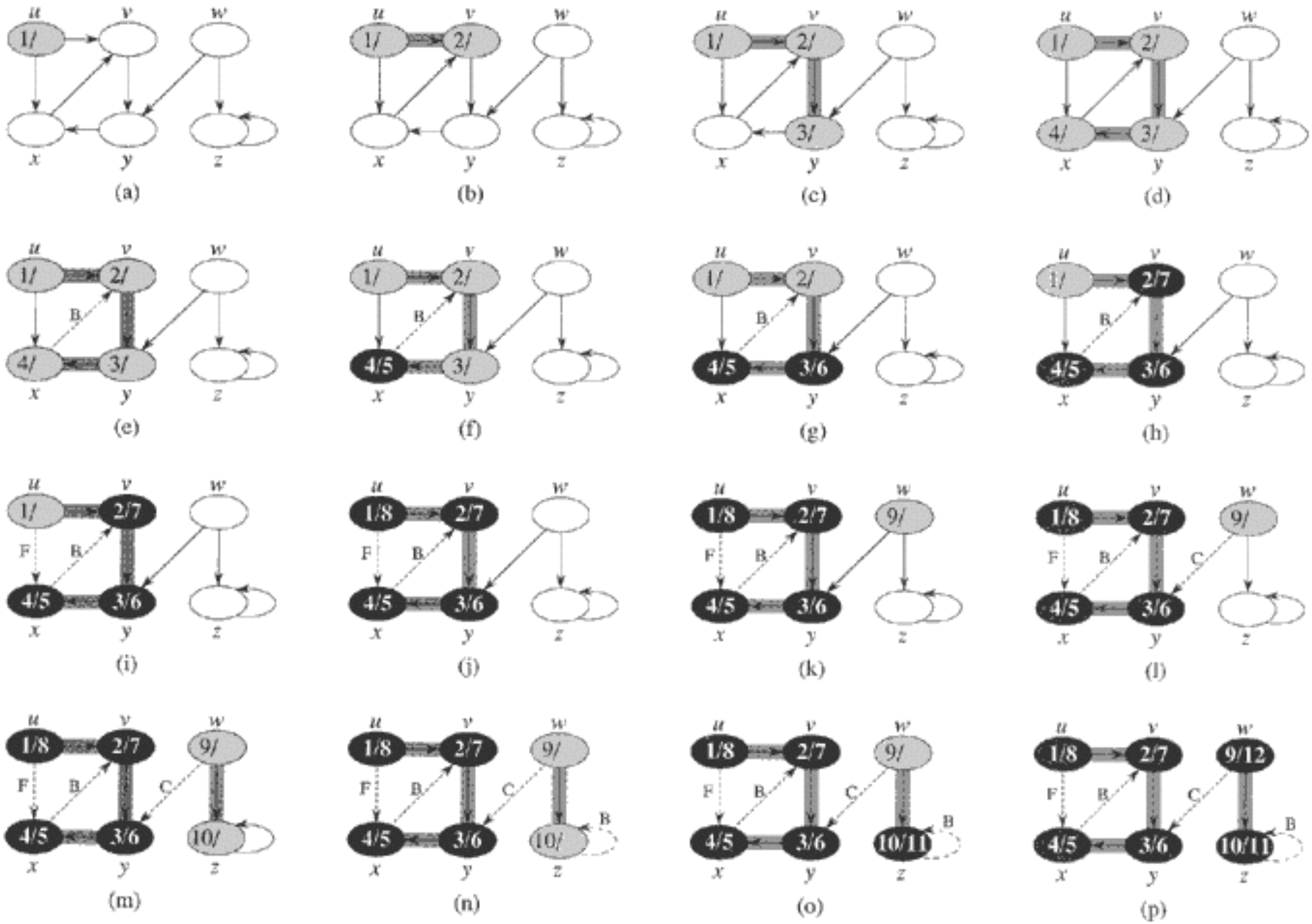
$color[u] \leftarrow$ GRAY ▷ White vertex u has just been discovered.
$time \leftarrow time + 1$
$d[u] \leftarrow time$
for each $v \in Adj[u]$ ▷ Explore edge (u, v).
    do if $color[v] =$ WHITE
        then $\pi[v] \leftarrow u$
            DFS-VISIT(v)
$color[u] \leftarrow$ BLACK ▷ Blacken u; it is finished.
$f[u] \leftarrow time \leftarrow time + 1$

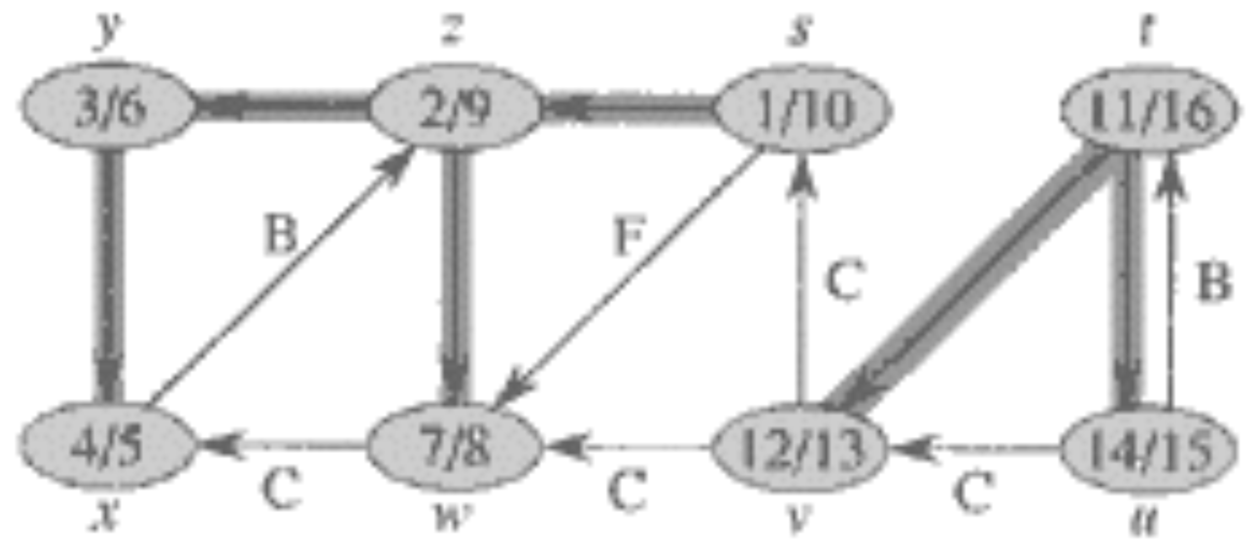
- white: not visited
- gray: visited, but not finished
- black: Finished

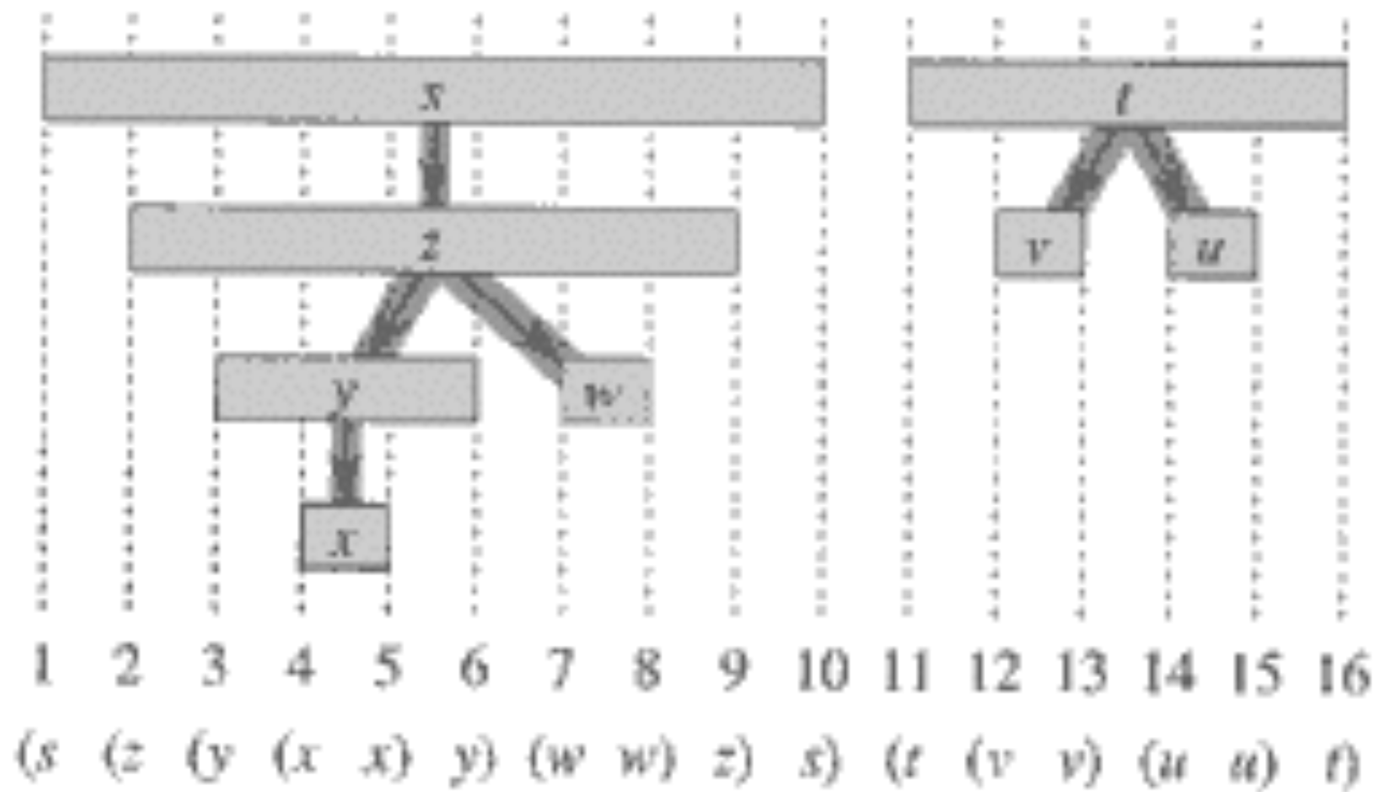
DFS revisited

DFS(G)

for each vertex $u \in V[G]$
    do $color[u] \leftarrow$ WHITE
        $\pi[u] \leftarrow$ NIL
$time \leftarrow 0$
for each vertex $u \in V[G]$
    do if $color[u] =$ WHITE
        then DFS-VISIT(u)

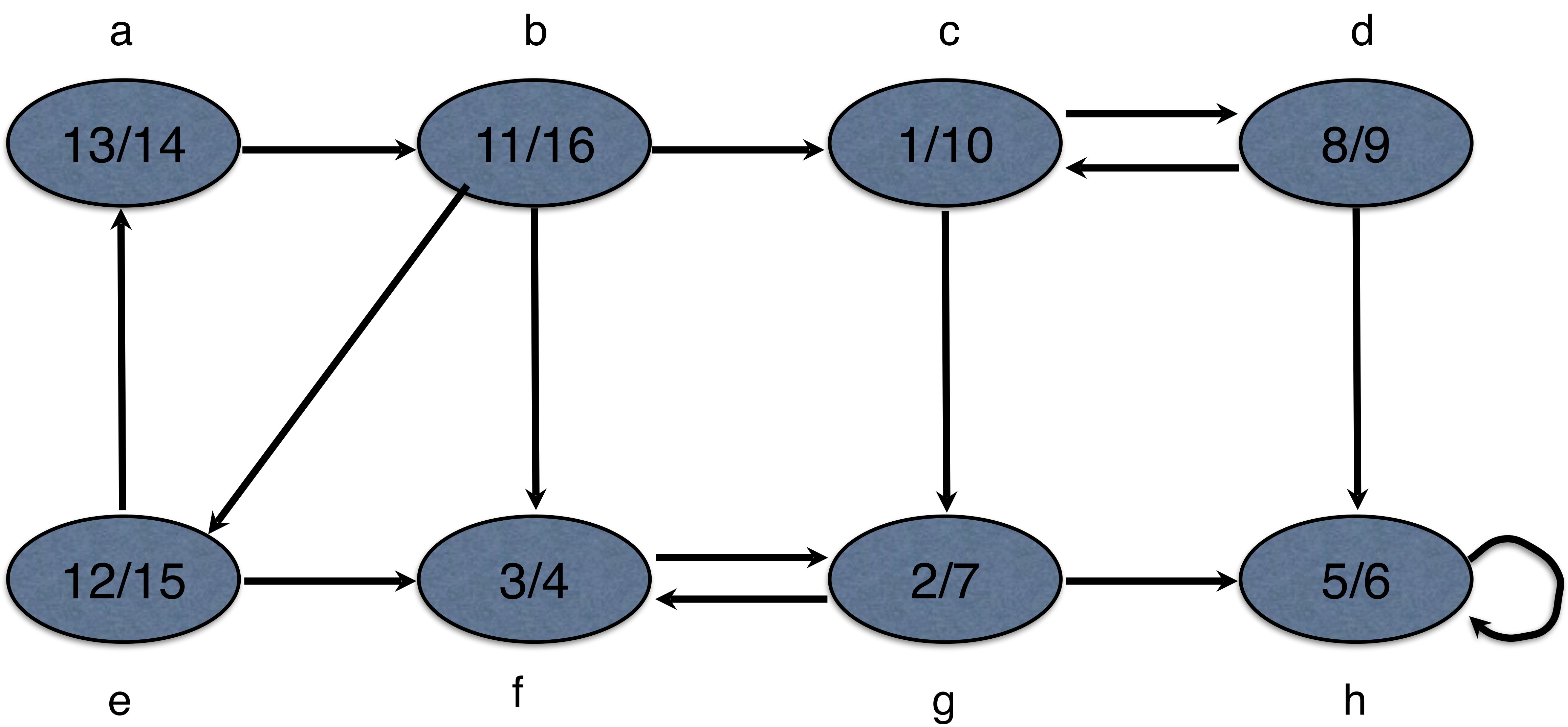

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)


y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u


s
t
z
v
u
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

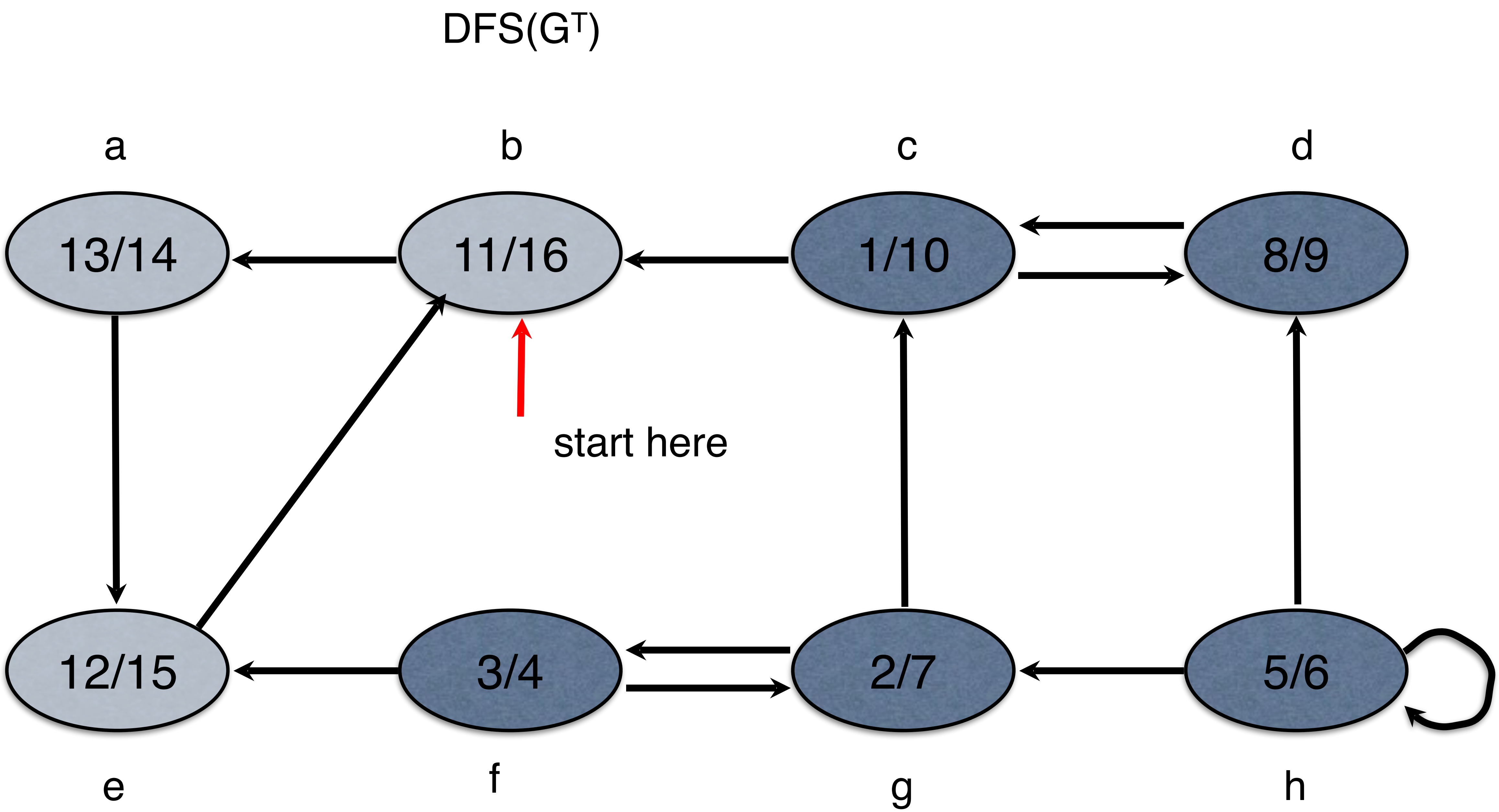
개선된 알고리즘

- DFS(G)를 호출하여 모든 노드들에 대해서 종료시간 f[u]를 계산한다.
- G^T를 만든다.
- DFS(G^T)를 호출한다. 단 노드들을 1단계에서 계산해둔 f[u]값이 감소하는 순서대로 고려한다.
- DFS(G^T)에 의해서 만들어지는 DFS forest의 각각의 트리가 하나의 SCC가 된다.



DFS(G)의 결과

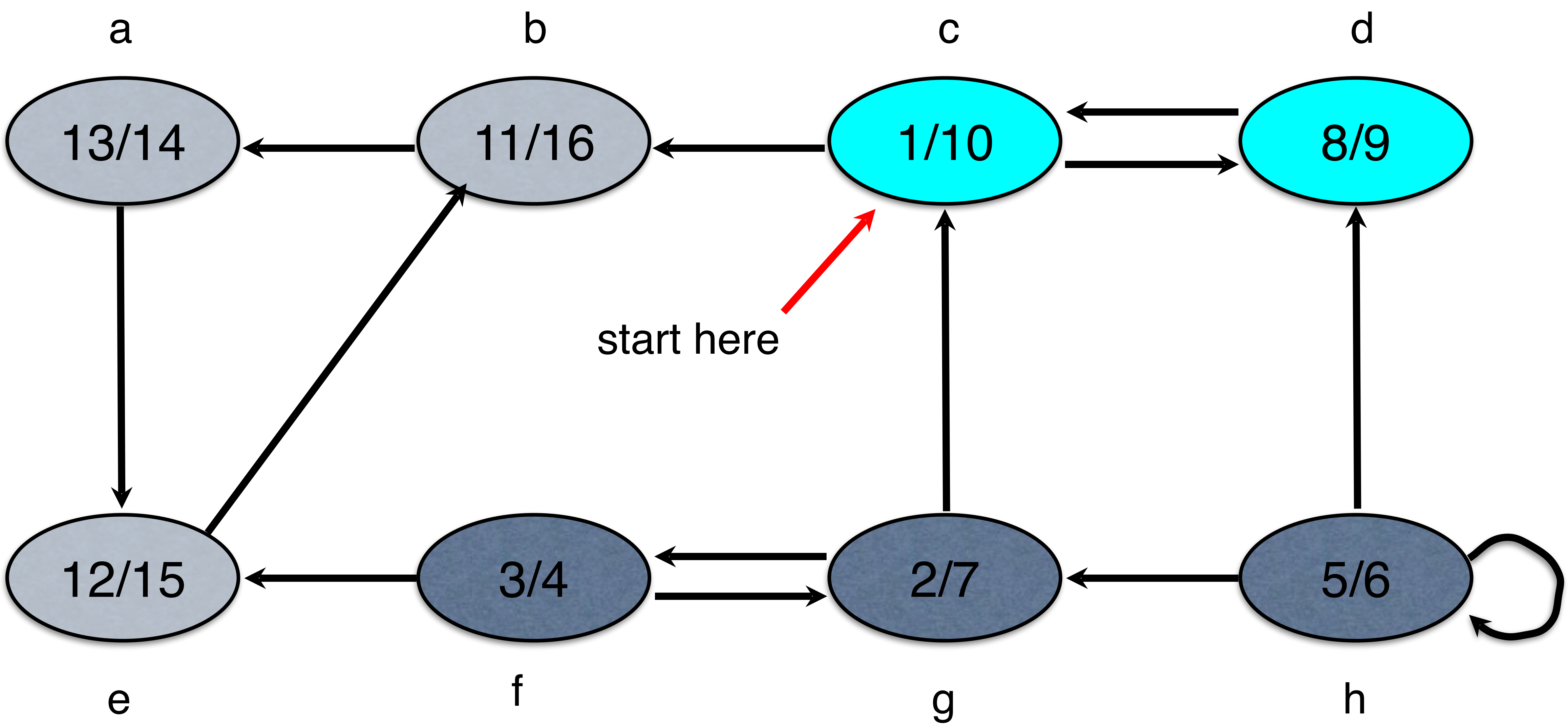
개선된 알고리즘



노드 {b,a,e} 가 하나의 SCC를 만듬

개선된 알고리즘

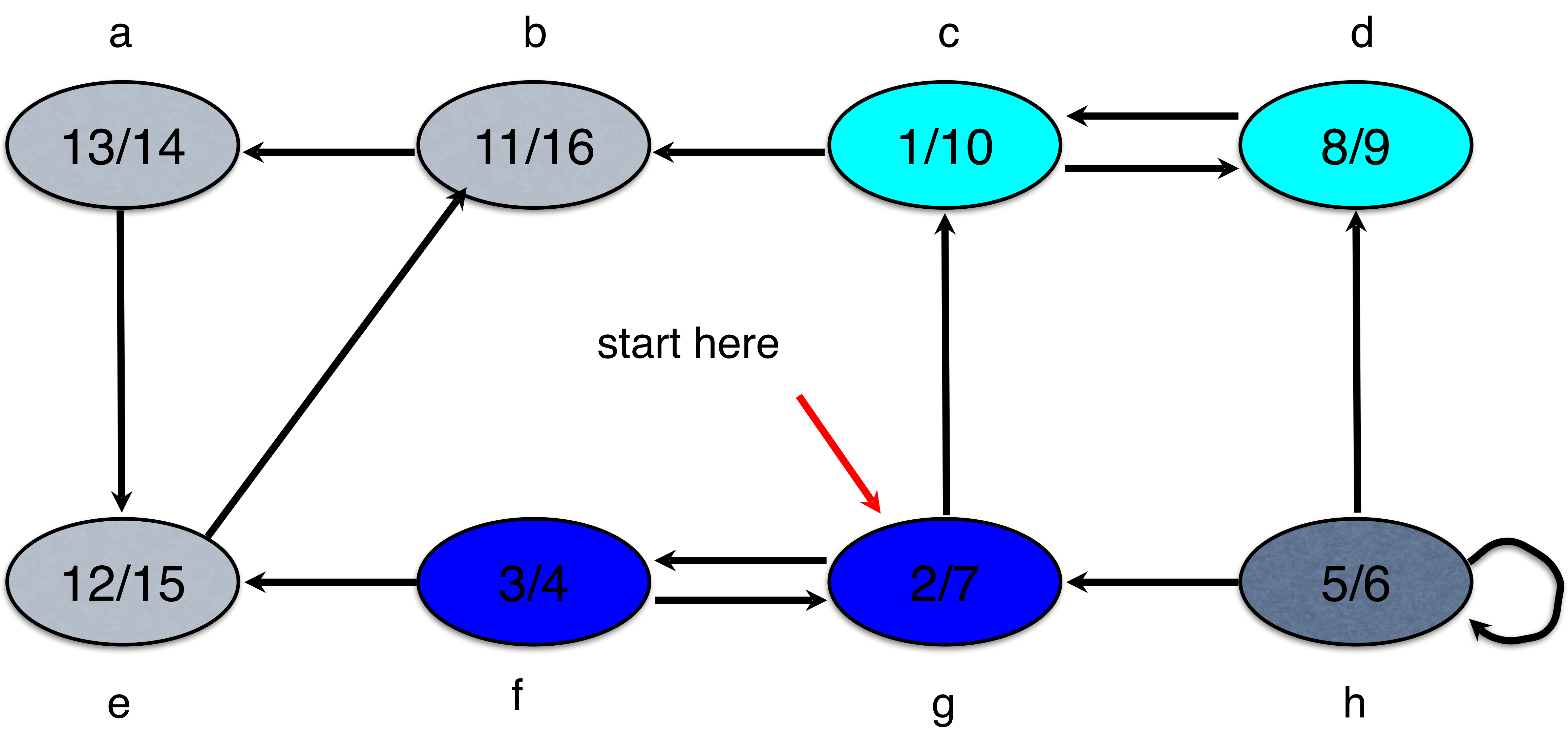
DFS(G^T)



노드 {c,d} 가 하나의 SCC를 만듬

개선된 알고리즘

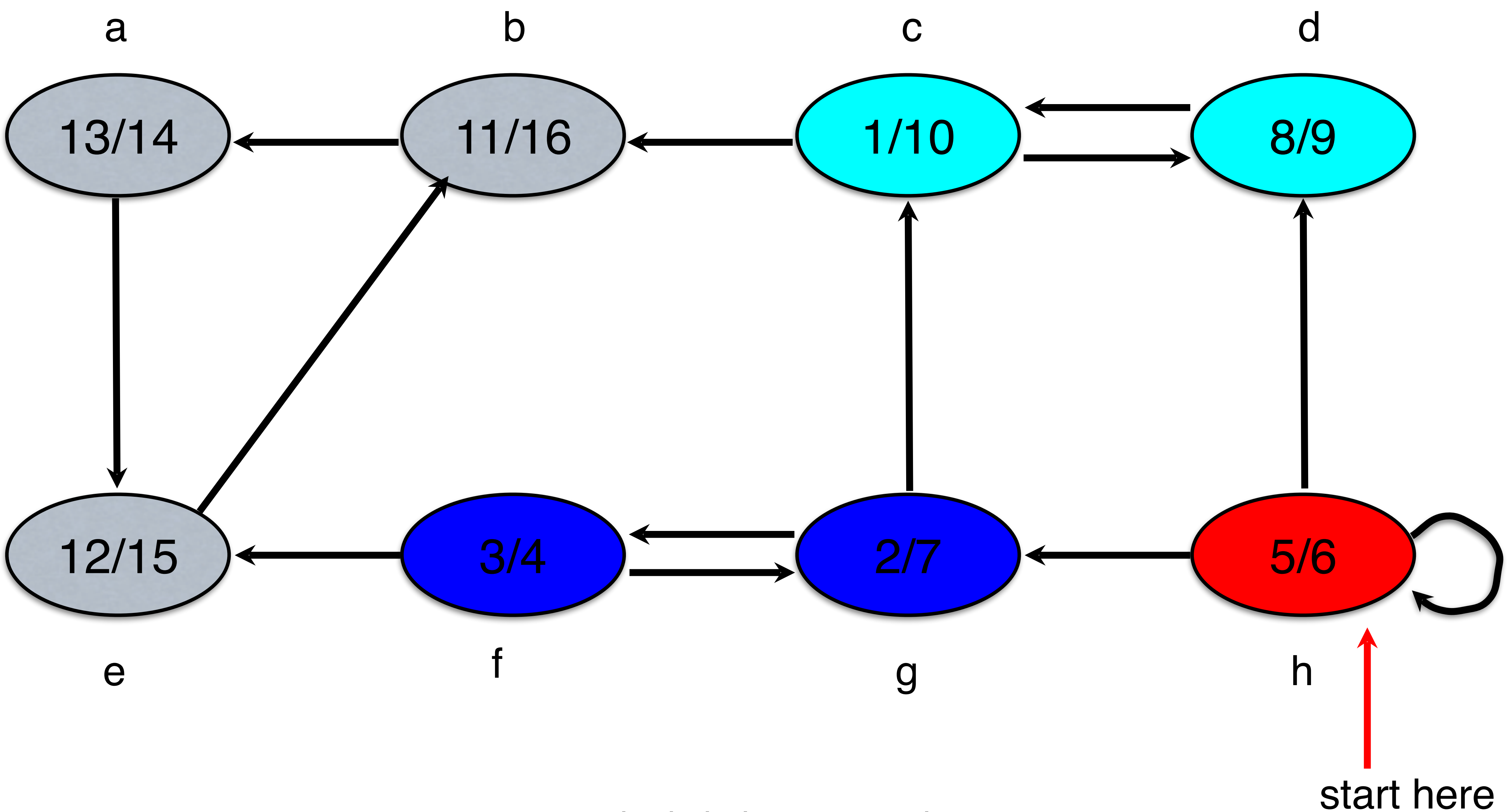
DFS(G^T)



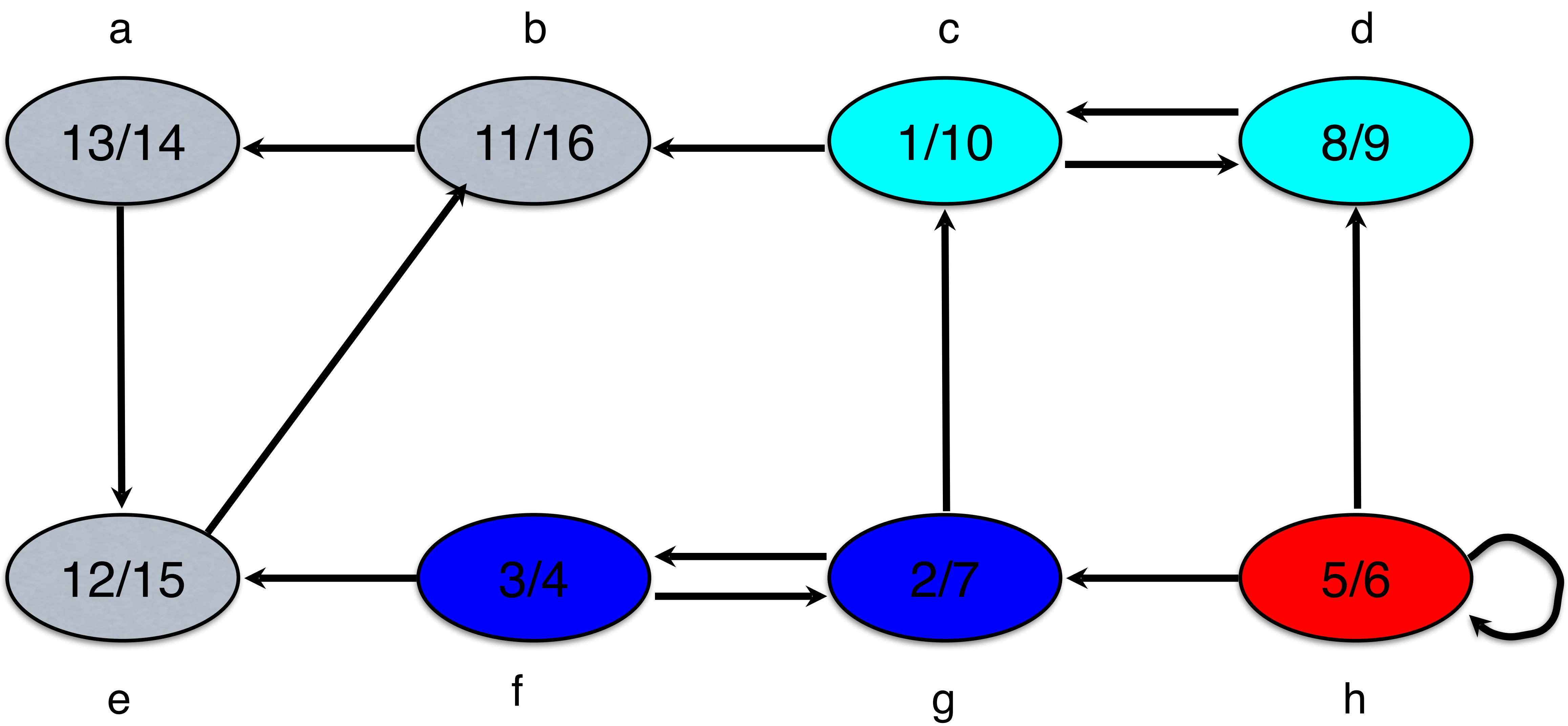
노드 {g,f} 가 하나의 SCC를 만듬

개선된 알고리즘

DFS(G^T)



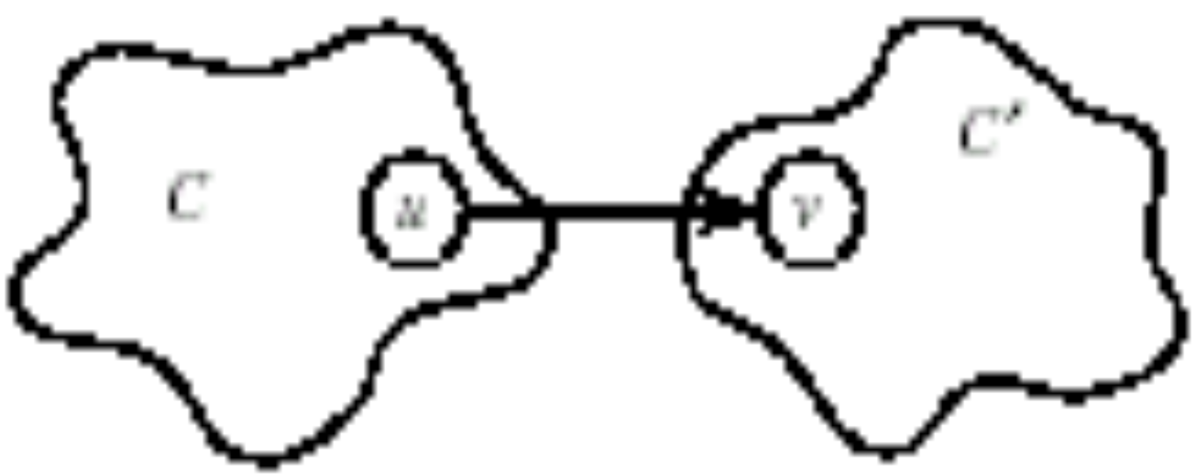
노드 {h} 가 하나의 SCC를 만듬



4개의 SCC가 존재

Correctness

- 임의의 SCC C에 대해서 f(C)=max{f(u) | u∈C}
- C와 C’를 G의 두 SCC라고 하자.
- 만약 u∈C이고 v∈C’인 두 노드 사이에 에지 (u,v)가 존재한다면 f(C)>f(C’)이다.



DFS를 C에 속한 노드에서 먼저 시작한 경우와, 반대의 경우로 나누어 생각해보면 간단히 증명가능

Correctness (계속)

- G의 두 SCC C와 C'에 대해 f(C)>f(C')라고 하자.
- 그러면 G^T에는 C로부터 C'으로 가는 에지가 존재하지 않는다.
- 따라서 f[u]가 최대인 노드에서 출발한 DFS는 그 노드가 속한 SCC를 벗어날 수 없다.

모든 강연결성분 구하기

<u>Strongly-Connected-Components(G)</u>

1. call DFS(G) to compute finishing times f[u] for each vertex u;
2. compute G^T;
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing f[u];
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component;

시간복잡도 O(n+m)