10장 가중치 그래프

순서

10.1 최소 비용 신장 트리

10.2 최단 경로

10.3 위상 순서

10.4 임계 경로

최소 비용 신장 트리

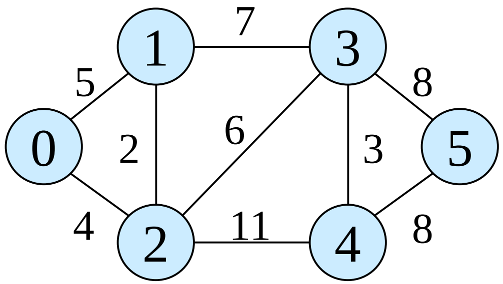
- **최소 비용 신장 트리(minimum cost spanning tree)**
  - 트리를 구성하는 간선들의 가중치를 합한 것이 최소가 되는 신장 트리
- **Kruskal, Prim, Sollin 알고리즘**
- **갈망 기법(greedy method)**
  - 최적의 해를 단계별로 구함
  - 각 단계에서 생성되는 중간 해법이 그 단계까지의 최적
- **신장 트리의 제한조건**
  - 전제 : 가중치가 부여된 무방향 그래프
  - n – 1 (n= |V|)개의 간선만 사용
  - 사이클을 생성하는 간선 사용 금지

Kruskal 알고리즘(1)

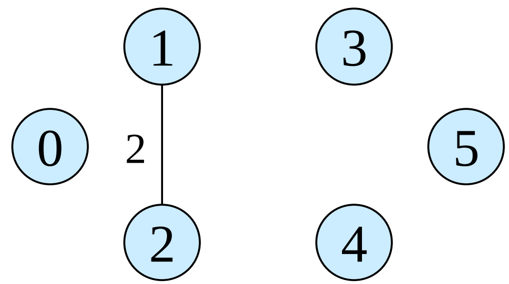
- 방법
  - 한번에 하나의 간선을 선택하여, 최소 비용 신장 트리 T에 추가
  - 비용이 가장 작은 간선을 선정하되, 이미 T에 포함된 간선들과 사이클을 형성하지 않는 간선만을 추가
  - 비용이 같은 간선들은 임의의 순서로 하나씩 추가

- 핵심 구현
  - 최소 비용 간선 선택
    - 가중치에 따라 오름차순으로 정렬한 간선의 순차 리스트 유지
  - 사이클 방지 검사
    - T에 추가로 포함될 정점들을 연결요소별로 정점 그룹을 만들어 유지
    - 간선 (i, j)가 T에 포함되기 위해서는 정점 i와 j가 각각 상이한 정점 그룹에 속해 있어야 사이클이 형성되지 않음

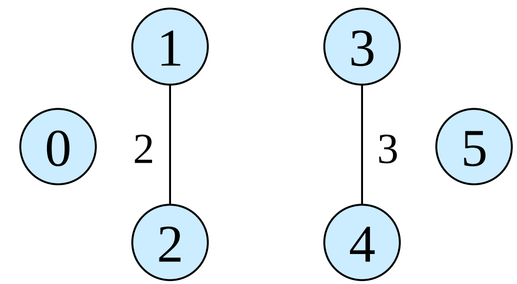
Kruskal 알고리즘(2)



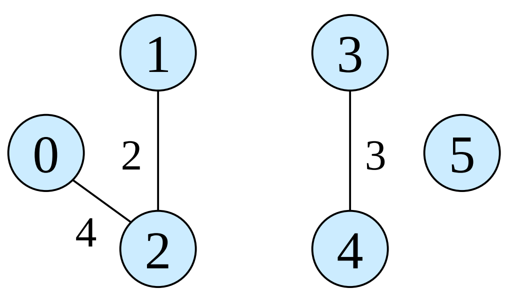
G=(V, E)
T={ }
S={{0}, {1}, {2}, {3}, {4}, {5}}
(a)



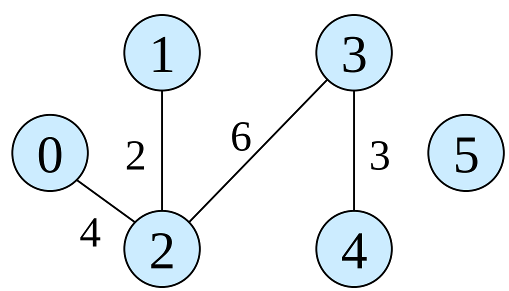
T={(1,2) }
S={{0}, {1,2}, {3}, {4}, {5}}
(b)



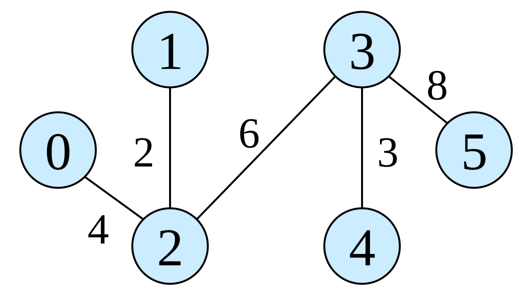
T={(1,2), (3,4) }
S={{0}, {1,2}, {3,4}, {5}}
(c)



T={(1,2), (3,4), (0,2) }
S={{0,1,2}, {3,4}, {5}}
(d)



T={(1,2), (3,4), (0,2), (2,3) }
S={{0,1,2,3,4}, {5}}
간선 (0,1)은 첨가 거절
(e)



T={(1,2), (3,4), (0,2), (2,3), (3,5) }
S={{0,1,2,3,4,5}}
간선 (1,3)은 첨가 거절
(f)

Kruskal 알고리즘 수행 단계

Kruskal 알고리즘(3)

```
Kruskal(G,n)
    //G=(E,V)이고 n=|V|, |V|는 정점 수
  T ← ∅;
  edgelist ← E(G); // 그래프 G의 간선 리스트
  S_0 ← {0}, S_1 ← {1} , ... ,S_{n-1} ←{n-1};
  while (|E(T)|<n-1 and |edgeList|>0) do  {
     // |E(T)|는 T에 포함된 간선 수, |edgeList|는 검사할 간선 수
    select least-cost (i, j) from edgeList;
    edgeList ← edgeList - {(i, j)};   // 간선 (i, j)를 edgeList에서 삭제
    if ({i, j} !⊆ S_k for any k) then  {
      T ← T ∪ {(i, j)};   // 간선 (i, j)를 T에 첨가
      S_i ← S_i ∪ S_j;   // 간선이 부속된 두 정점 그룹을 합병
    }
  }
  if (|E(T)|<n-1) then  {
    print ('no spanning tree');
  }
  return T;
end Kruskal()
```

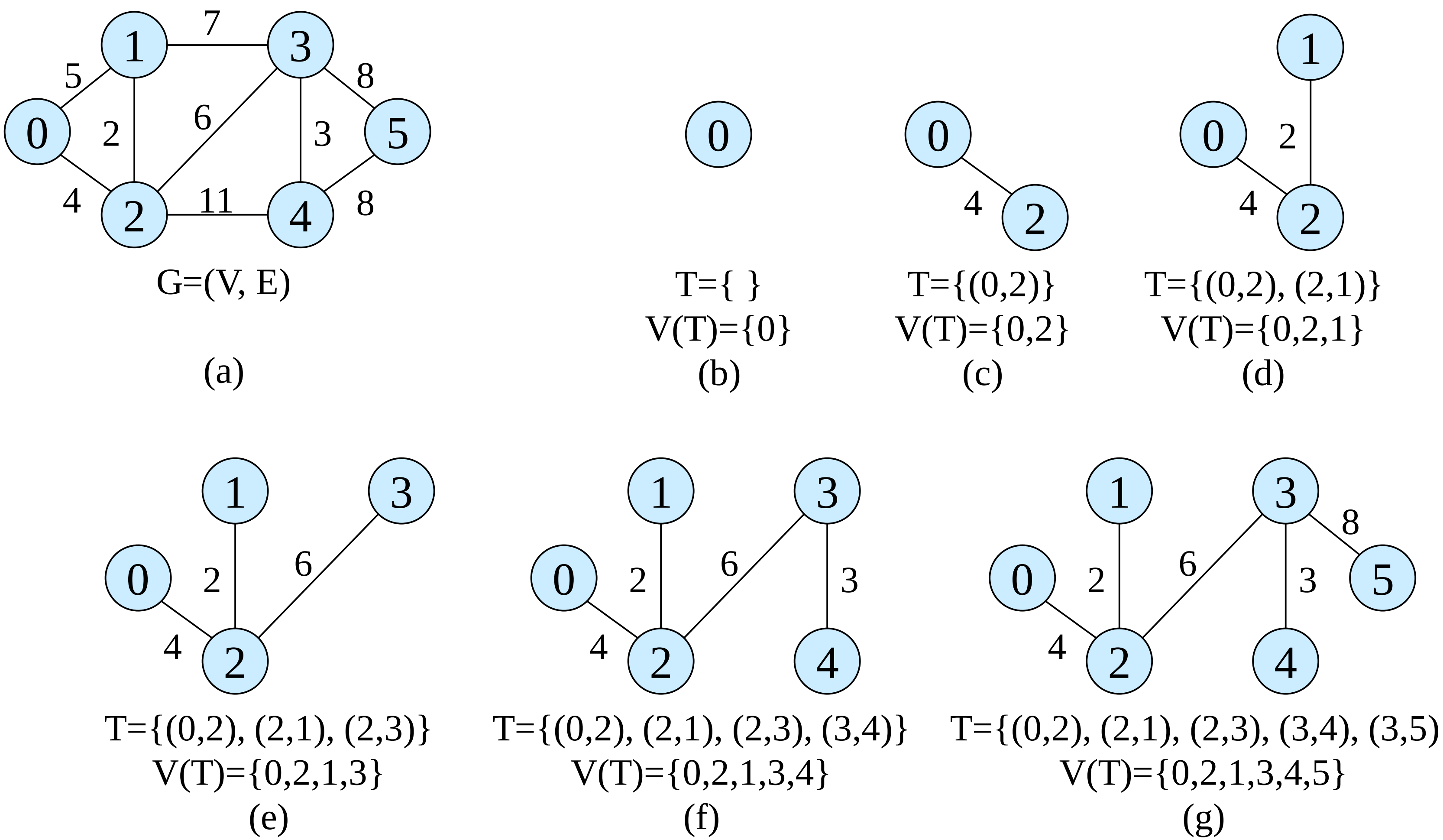
데이타베이스 Since 1982

Prim 알고리즘(1)

- **방법**
  - 한번에 하나의 간선을 선택하여, 최소 비용 신장 트리 T에 추가
  - Kruskal 알고리즘과는 달리 구축 전 과정을 통해 하나의 트리만을 계속 확장

- **구축 단계**
  - 하나의 정점 u를 트리의 정점 집합 V(T)에 추가
  - V(T)의 정점들과 인접한 정점들 중 최소 비용 간선 (u, v)를 선택하여 T에 추가, 정점은 V(T)에 추가
  - T가 n-1개의 간선을 포함할 때까지, 즉 모든 정점이 V(T)에 포함될 때까지 반복
  - 사이클이 형성되지 않도록 간선 (u, v) 중에서 u 또는 v 하나만 T에 속하는 간선을 선택

Prim 알고리즘(2)



Prim 알고리즘의 수행 단계

Prim 알고리즘(3)

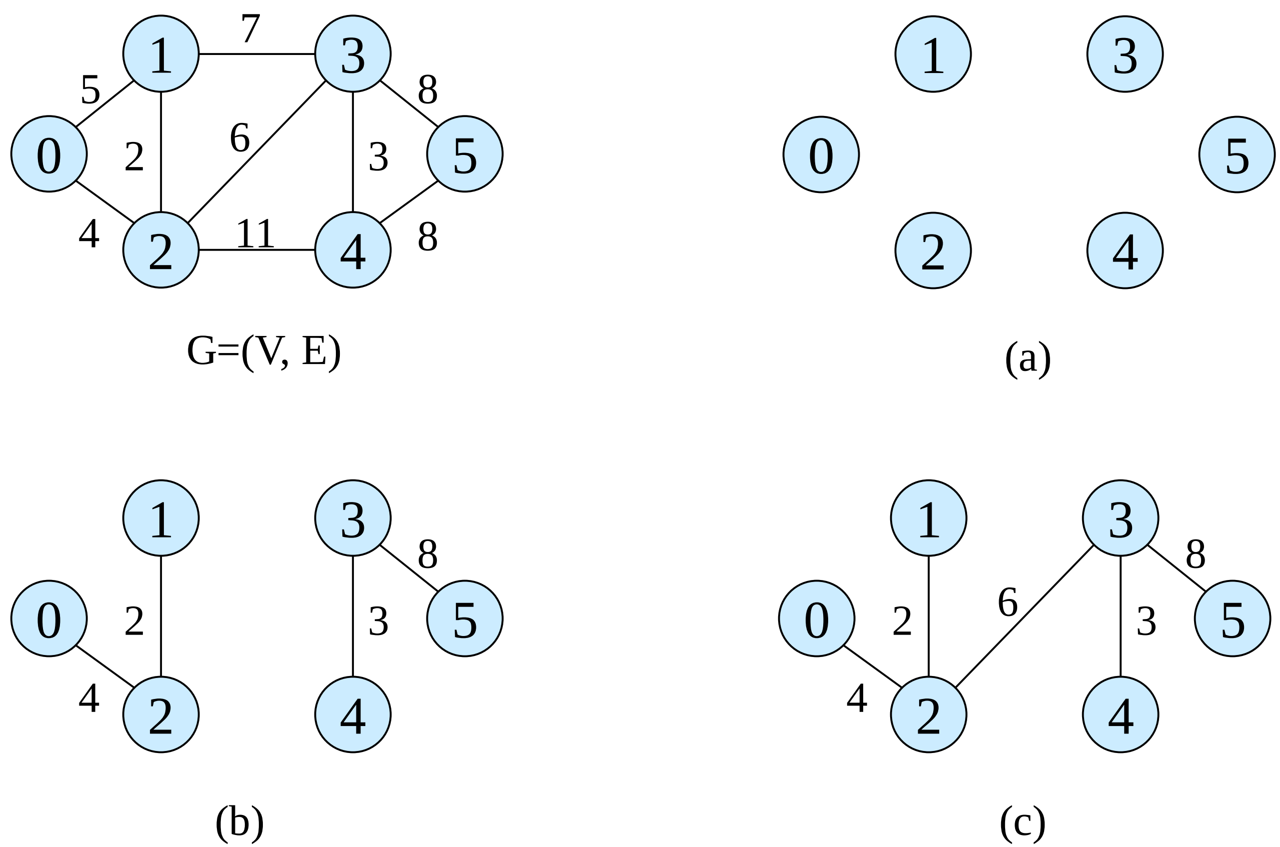
Prim(G, i)   // i는 시작 정점
  $T \leftarrow \emptyset$;   // 최소 비용 신장 트리
  $V(T) = \{ i \}$;   // 신장 트리의 정점
  while (|T| < n-1) **do** {
    if (select least-cost (u, v) such that $u \in V(T)$ and $v \notin V(T)$) **then** {
      $T \leftarrow T \cup \{(u, v)\}$;
      $V(T) \leftarrow V(T) \cup \{v\}$;
    }
    else {
      print("no spanning tree");
      return T;
    }
  }
  return T;
end Prim()

Sollin 알고리즘(1)

- 방법
  - 각 단계에서 여러 개의 간선을 선택하여, 최소 비용 신장 트리를 구축
  - 구축 과정 중에 두 개의 트리가 하나의 동일한 간선을 중복으로 선정할 경우, 하나의 간선만 사용

- 구축 단계
  - 그래프의 각 정점 하나만을 포함하는 n개의 트리로 구성된 신장 포리스트에서부터 시작
  - 매번 포리스트에 있는 각 트리마다 하나의 간선을 선택
  - 선정된 간선들은 각각 두 개의 트리를 하나로 결합시켜 신장 트리로 확장
  - n-1개의 간선으로 된 하나의 트리가 만들어지거나, 더 이상 선정할 간선이 없을 때 종료

Sollin 알고리즘(2)



Sollin 알고리즘의 수행 단계

Sollin 알고리즘(3)

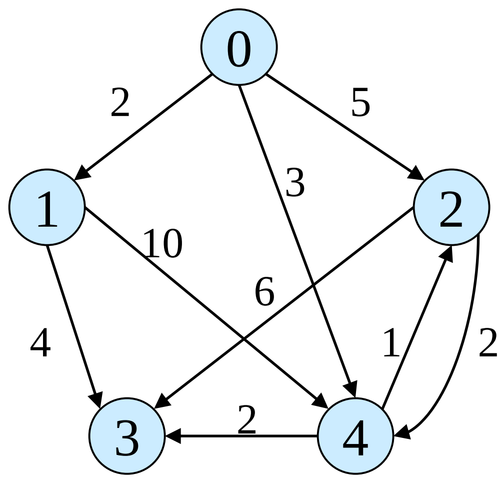
```
Sollin(G, n)
   // G = (V, E), n = |V|
   S_0 ← {0}; S_1 ← {1}; , ... , S_n-1 ← {n-1};    // n개의 노드 그룹으로 초기화
   T ← ∅;  // 최소 비용 신장 트리
   List ← ∅;                  // 연산 단계에서 선정된 간선
   while (|T| < n-1 and Edges ≠ ∅ ) do {
         for (each S_i) do {
               select least-cost (u, v) from Edges
               such that u∈S_i and v∉S_i;
               if ((u, v)∉List) then List ← List ∪ {(u, v)};     // 중복 간선은 제거
         }  //for
        while (List ≠ ∅) do {     // List가 공백이 될 때까지
            remove (u, v) from List;
            if ({u, v} !⊆ S_u and {u, v} !⊆ S_v) then {
                  // S_u와 S_v는 각각 정점 u와 v가 포함된 트리
                  T ← T ∪ {(u, v)};
                  S_u ← S_u ∪ S_v;
                  Edges ← Edges – {(u, v)};
   }}} //if, while, while
   if (|T| < n-1) then {
         print(“no spaning tree”); }
   return T;
end Sollin()
```

최단 경로

- 하나의 정점에서 다른 모든 정점까지의 최단 경로
  - 시작점 v에서 목표점 t까지의 경로 중 최단 경로
  - 0 이상의 가중치, 방향 그래프
- 음의 가중치가 허용된 최단 경로
  - 시작점 v에서 목표점 t까지의 경로 중 최단 경로
  - 음의 가중치 허용, 방향 그래프
- 모든 정점 쌍의 최단 경로
  - 모든 정점을 시작점으로 하는 최단 경로
  - 음의 가중치 허용, 방향 그래프
- 이행적 폐쇄 행렬 (transitive closure matrix)
  - 경로의 존재 유무
  - 가중치 없음, 방향 그래프

하나의 정점에서 다른 모든 정점까지의 최단경로(1)

- 시작점 v에서 G의 나머지 모든 정점까지의 최단 경로
- 시작점 v와 목표점 t까지의 경로 중, 경로를 구성하는 간선들의 가중치 합이 최소가 되는 경로
- 방향 그래프 **G=(V, E), weight[i, j] ≥ 0**



(a) G = (V, E)

경로	거리
0, 1	2
0, 4	3
0, 4, 2	4
0, 4, 3	5

(b) 최단 경로

방향 그래프와 최단 경로

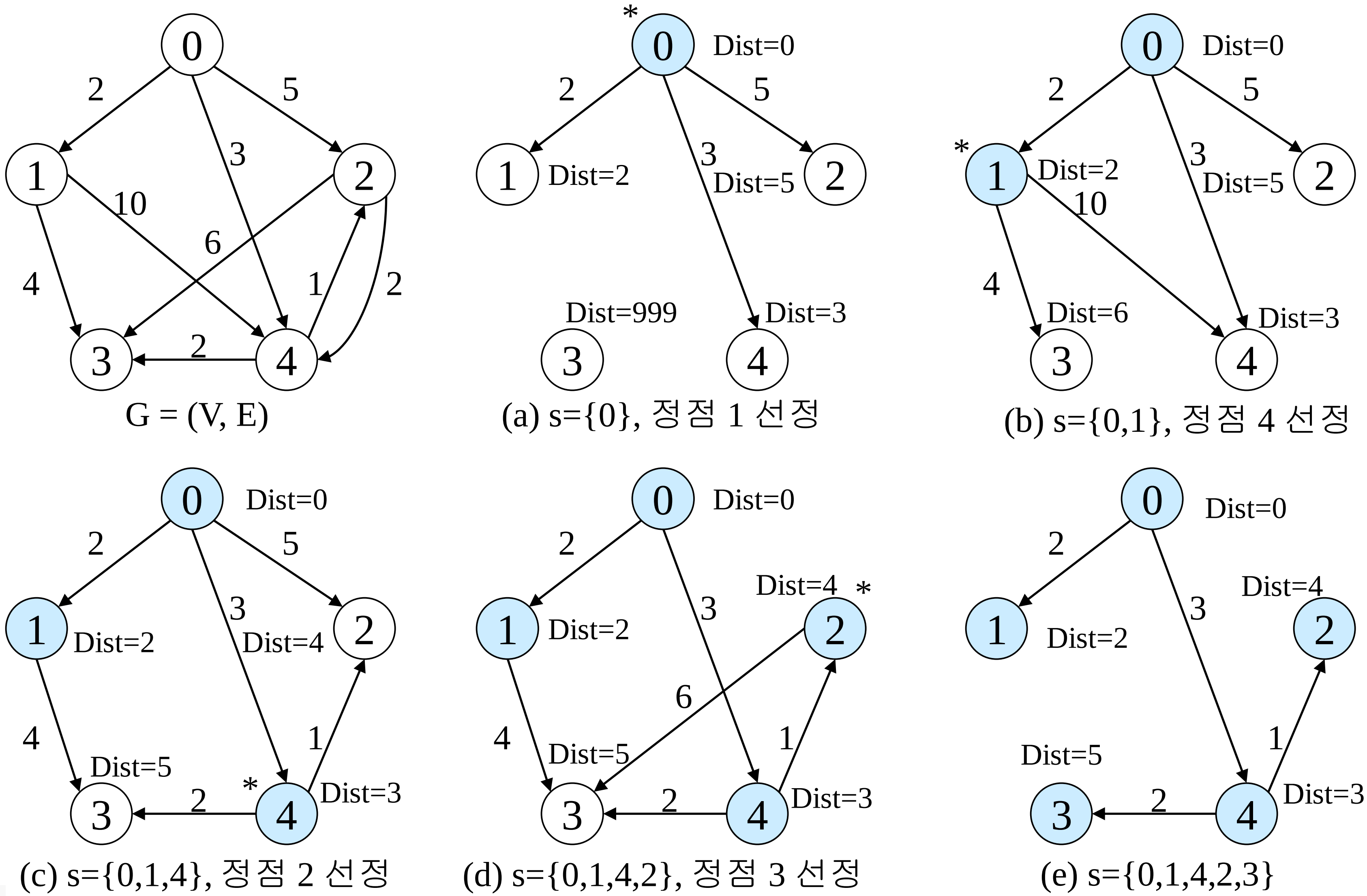
하나의 정점에서 다른 모든 정점까지의 최단경로(2)

◆ Dijkstra 최단 경로 알고리즘의 원리

- S : 최단 경로가 발견된 정점들의 집합
- weight[i, j] : 아크 <i, j>의 가중치.
- Dist[i] : S에 속하지 않은 i에 대해서, v에서 시작하여 S에 있는 정점만을 거쳐 정점 i에 이르는 최단 경로의 길이

1. 처음 S에는 시작점 v만 포함, Dist[v]=0
2. 가장 최근에 S에 첨가한 정점을 u로 설정
3. u의 모든 인접 정점 중에서 S에 포함 되지 않은 w에 대해 Dist[w]를 다시 계산
   - Dist[w]=min{Dist[w], Dist[u] + weight[u, w]}
4. S에 포함되지 않은 모든 정점 w중에서 dist[w]가 가장 작은 정점 w를 S에 첨가
5. 모든 정점에 대한 최단 경로가 결정될 때까지 단계 2~4 를 반복

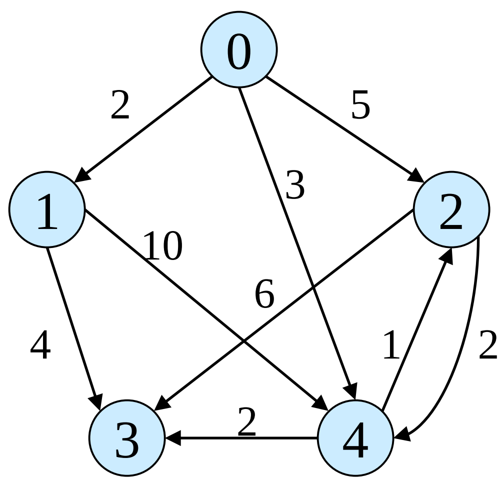
하나의 정점에서 다른 모든 정점까지의 최단경로(3)



최단 경로 계산의 예

하나의 정점에서 다른 모든 정점까지의 최단경로(4)

- **Dijkstra의 최단 경로 알고리즘**
  - G의 n개의 정점을 0에서 n-1까지 번호를 붙임
  - S[] : 정점 i 가 S에 포함되어 있으면 S[i] = true, 아니면 S[i]=false로 표현하는 불리언 배열
  - weight[n, n] : 가중치 인접행렬
    - weight[i, j] : 아크 < i, j>의 가중치.
      아크 <i, j>가 그래프에 포함되어 있지 않은 경우에는 아주 큰 값으로 표현



G = (V, E)

	[0]	[1]	[2]	[3]	[4]
[0]	0	2	5	999	3
[1]	999	0	999	4	10
[2]	999	999	0	6	2
[3]	999	999	999	0	999
[4]	999	999	1	2	0

weight[5, 5]

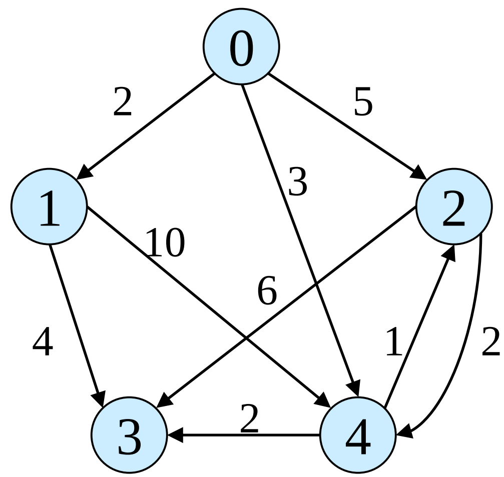
그래프 G와 가중치 인접 행렬

하나의 정점에서 다른 모든 정점까지의 최단경로(5)

◆ **Dijkstra의 최단 경로 알고리즘**

```
shortestPath(v, weight, n)
   // v는 시작점, weight는 가중치 인접 행렬, n은 정점수
   // create S[n], Dist[n]
  for (i←0; i<n; i←i+1) do {
      S[i] ← false;                  // S를 초기화
      Dist[i] ← weight[v, i];        // Dist를 초기화
  }
  S[v] ← true;
  Dist[v] ← 0;
  for (i←0; i<n-2; i←i+1) do {     // n-2번 반복
      select u such that                        // 새로운 최단 경로를 선정
          Dist[u] = min{Dist[j] | S[j] = false and 0≤j<n};
      S[u] ← true;
      for (w←0; w<n; w←w+1) do {                // 확정이 안된 경로들에 대해 다시 계산
          if (S[w] = false) then {
              if (Dist[w] > (Dist[u] + weight[u, w])
                   then Dist[w] ← Dist[u] + weight[u, w];
  }}}
end shortestPath
```

하나의 정점에서 다른 모든 정점까지의 최단경로(6)



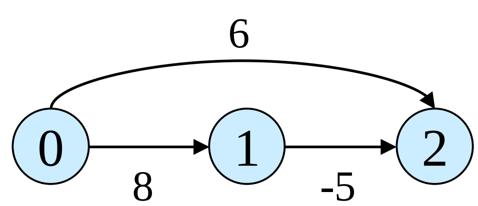
G = (V, E)

for 루프	선택된 정점	S=true인 정점	Dist[i]				
			[0]	[1]	[2]	[3]	[4]
초기화		[0]	0	2	5	999	3
1	[1]	[0],[1]	0	2	5	6	3
2	[4]	[0],[1],[4]	0	2	4	5	3
3	[2]	[0],[1],[4],[2]	0	2	4	5	3

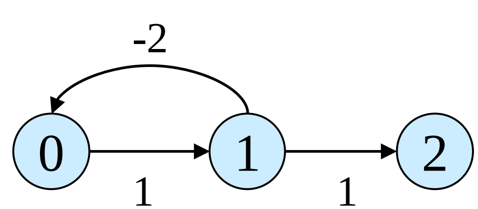
그래프 G에 대한 shortestPath 수행 내용

음의 가중치가 허용된 최단 경로(1)

- ◆ 음의 가중치를 가진 방향 그래프
  - Dijkstra 알고리즘으로 최단 경로를 구할 수 없음
  - 음의 길이값을 갖는 사이클은 허용하지 않음
  - 사이클이 없는 최단 경로가 가질 수 있는 최대 간선 수 (n-1)를 이용하여 알고리즘 작성



음의 가중치를 가진 방향 그래프
(최단경로 알고리즘에 적합치 않음)



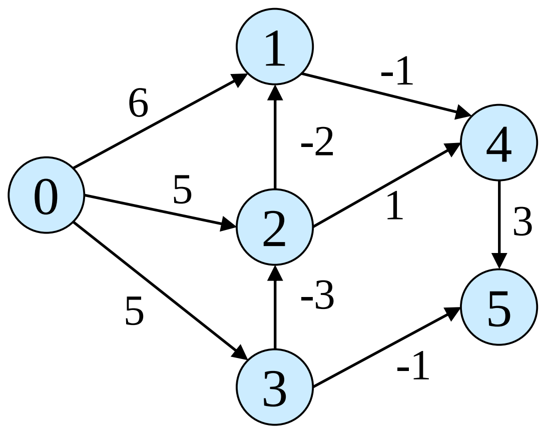
길이가 음인 사이클을 가진 방향 그래프

음의 가중치가 허용된 최단 경로(2)

- **Bellman and Ford 알고리즘의 원리**
  - $Dist^k[u]$: 시작점 v에서 정점 u까지 최대 k개의 아크를 갖는 최단 경로의 길이
  - $Dist^1[u] = weight[v, u]$
  - $Dist^{n-1}[u]$: 시작점 v에서 정점 u까지의 최단 경로의 길이
  - 만일 시작점 v에서 어떤 정점 u까지의 최단 경로가 최대 k개 (k>1)의 간선을 포함할 수 있는 경우에서
    - k-1개 이하의 간선만 포함 : $Dist^k[u] = Dist^{k-1}[u]$
    - k개 간선을 포함 : 시작점 v에서 정점 u에 인접한 어떤 정점 i까지의 최단 경로를 포함하므로, $Dist^k[u] = \min\{Dist^{k-1}[i]+weight[i, u]\}$
  - $Dist^k[u] \leftarrow \min\{Dist^{k-1}[u], \min\{Dist^{k-1}[i] + weight[i, u]\}\}$

    $(k = 2, 3, \ldots, n-1)$

음의 가중치가 허용된 최단 경로(3)



(a) 방향 그래프(시작점 0)

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	6	5	5	∞	∞
[1]	∞	0	∞	∞	−1	∞
[2]	∞	−2	0	∞	1	∞
[3]	∞	∞	−3	0	∞	−1
[4]	∞	∞	∞	∞	0	3
[5]	∞	∞	∞	∞	∞	0

(b) weight[6, 6]

$Dist^k$	Dist[6]					
	[0]	[1]	[2]	[3]	[4]	[5]
$Dist^1$	0	6	5	5	∞	∞
$Dist^2$	0	3	2	5	5	4
$Dist^3$	0	0	2	5	2	4
$Dist^4$	0	0	2	5	−1	4
$Dist^5$	0	0	2	5	−1	2

(c) $Dist^5$ 계산 단계

음의 가중치가 허용된 최단 경로

음의 가중치가 허용된 최단 경로(4)

◆ **Bellman and Ford 알고리즘**

```
generalShortestpath(v, n)
// 음의 가중치가 허용된 방향 그래프 G=(V, E)에서 단일 시작점 v로부터
// 모든 종점까지의 최단 경로 탐색
// 음의 길이를 갖는 사이클은 허용되지 않음
// n은 정점의 수 (0, 1, …, n-1)
   for (i←0; i<n; i ← i+1) do
      Dist[i] ← weight[v, i];   // Dist를 초기화
   for (k ← 2; k< n-1; k ← k+1) do {
         for (each u such that u≠v and indegree(u)>0) do {
               for (each <i, u> ∈ E) do {
                           // 진입차수가 0보다 큰 모든 정점에 대해
                     if (Dist[u] > Dist[i] + weight[i,u]) then
                           Dist[u] ← Dist[i] + weight[i, u];
               }
         }
   }
end generalShortestPath()
```

모든 정점 쌍의 최단 경로(1)

◆ 모든 정점 쌍의 최단 경로

- 하나가 아닌 모든 정점을 시작점으로 하는 최단 경로
- 각 정점을 시작점으로 n번 shortestpath 알고리즘 사용
  - 음이 아닌 가중치 : $O(n^3)$, 음의 가중치 : $O(n^4)$
  - 인접리스트 : $O(n^2 \cdot e)$
- 음의 가중치를 가진 그래프의 모든 쌍에 대한 최단 경로를 $O(n^3)$에 찾을 수 있는 알고리즘
  - 그래프 G를 가중치 인접 행렬 D로 표현
  - $D^k[i, j]$: 정점 i에서 j까지의 최단 경로로, 정점 인덱스가 0에서 k까지인 정점만 중간 정점으로 이용할 수 있음
  - $D^{n-1}[i, j]$: 최단 경로(∵ n-1보다 큰 인덱스를 가진 정점이 없음)
  - $D^{-1}[i, j]$: 중간 정점 없는 행렬(= weight[i, j])
  - 행렬 D^{-1}에서부터 시작하여, 계속 최단 거리 행렬 D^{n-1}까지 생성
- $D^k[i, j] \leftarrow \min\{D^{k-1}[i, j], D^{k-1}[i, k]+ D^{k-1}[k, j]\}, k \geq 0$

모든 정점 쌍의 최단 경로(1) (2?)

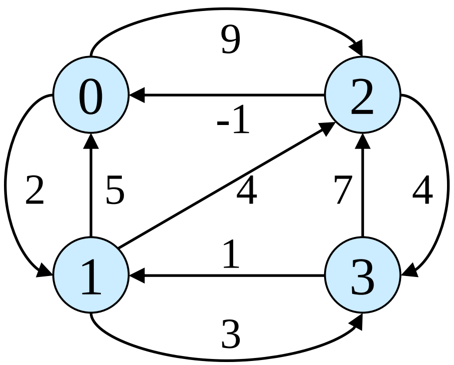
- $D^k[i, j] \leftarrow \min\{D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j]\}, k \geq 0$
  1. 정점 i에서 j까지의 최단경로 탐색에서 인덱스가 k인 정점까지 이용할 수 있는 환경에서 정점k가 최단경로에 포함되지 않는다면 그 최단 경로 $D^k[i, j]$는 $D^{k-1}[i, j]$와 같다.
  2. 정점 i에서 j까지의 최단경로 탐색의 인덱스가 k인 정점까지 이용할 수 있는 환경에서 정점 k가 최단경로에 포함되어야 한다면 (i,k)와 (k,j)모두 최단 거리이어야 하고 그 경로상에 있는 정점의 인덱스는 모두 k-1이하이다. 따라서 이때 $D^k[i, j]$는 $D^{k-1}[i, k] + D^{k-1}[k, j]$가 된다.

모든 정점 쌍의 최단 경로(2)

◆ **allShortestPath 알고리즘**

```
allShortestPath(G, n)
  // G=(V, E), |V|=n
   for (i←0; i<n; i←i+1) do {
      for (j←0; j<n; j←j+1) do {
            D[i, j] ← weight[i, j];      // 가중치 인접 행렬을 복사
      }
   }
   for (k←0; k<n; k←k+1) do {      // 중간 정점으로 0에서 k까지 사용하는 경로
      for (i←0; i<n; i←i+1) do {      // 모든 가능한 시작점
            for (j←0; j<n; j←j+1) do {      // 모든 가능한 종점
                  if (D[i, j] > (D[i, k]+D[k, j])) then
                        // 보다 짧은 경로가 발견되었는지를 검사
                        D[i, j] ←  D[i, k]+D[k, j];
            }
       }
   }
end allShortestPath()
```

모든 정점 쌍의 최단 경로(3)



(a) G=(V, E)

D^{-1}	[0]	[1]	[2]	[3]
[0]	0	2	9	∞
[1]	5	0	4	3
[2]	−1	∞	0	4
[3]	∞	1	7	0

(b) D^{-1}(weight[4,4])

D^{0}	[0]	[1]	[2]	[3]
[0]	0	2	9	∞
[1]	5	0	4	3
[2]	−1	1	0	4
[3]	∞	1	7	0

(c) D^0

D^{1}	[0]	[1]	[2]	[3]
[0]	0	2	6	5
[1]	5	0	4	3
[2]	−1	1	0	4
[3]	6	1	5	0

(d) D^1

D^{2}	[0]	[1]	[2]	[3]
[0]	0	2	6	5
[1]	3	0	4	3
[2]	−1	1	0	4
[3]	4	1	5	0

(e) D^2

D^{3}	[0]	[1]	[2]	[3]
[0]	0	2	6	5
[1]	3	0	4	3
[2]	−1	1	0	4
[3]	4	1	5	0

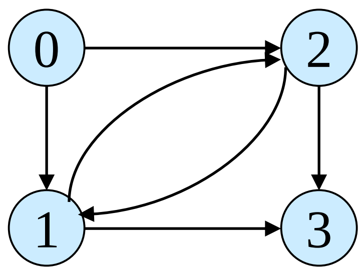
(f) D^3

그래프 G에 대한 allShortestPath 알고리즘의 수행 내용

이행적 폐쇄(1)

- **이행적 폐쇄(transitive closure)**
  - 가중치가 없는 방향 그래프 G에서 임의의 두 정점 i에서 j까지의 경로가 존재하는지 표현
- **이행적 폐쇄 행렬(D^+)**
  - $D^+[i, j] = 1$: 정점 i에서 j까지 길이가 0보다 큰 경로 존재
- **반사 이행적 폐쇄 행렬(D^*)**
  - $D^*[i, j] = 1$: 정점 i에서 j까지 길이가 0 이상인 경로 존재

이행적 폐쇄(2)



(a) 방향 그래프 G=(V, E)

A	[0]	[1]	[2]	[3]
[0]	0	1	1	0
[1]	0	0	1	1
[2]	0	1	0	1
[3]	0	0	0	0

(b) 인접 행렬(A)

D^+	[0]	[1]	[2]	[3]
[0]	0	1	1	1
[1]	0	1	1	1
[2]	0	1	1	1
[3]	0	0	0	0

(c) 이행적 폐쇄 행렬(D^+)

D^*	[0]	[1]	[2]	[3]
[0]	1	1	1	1
[1]	0	1	1	1
[2]	0	1	1	1
[3]	0	0	0	1

(d) 반사 이행적 폐쇄 행렬(D^{1*})

방향 그래프 G와 행렬 A, D^+, D^*

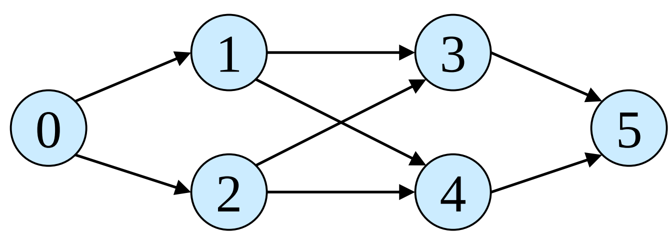
이행적 폐쇄(3)

- **D^+ : allShortestPath 알고리즘 이용**
  - 그래프 G에 간선 <i, j>가 있으면 $D^{-1}[i, j]=1$, 없으면 $D^{-1}[i, j] = \infty$로 초기화
  - 실행 끝내면서 $D^{n-1}[i, j] < \infty$인 것은 $D^+[i, j] = 1$으로 만들고, $D^{n-1}[i, j] = \infty$인 것은 $D^+[i, j] = 0$로 만듦
- **D^* : D^+에서 $D^+[i, i]$ 값을 1로 만듦**

- **불리언 행렬 사용**
  - 인접 행렬과 D^+를 true, false 값을 갖는 행렬로 만듦
  - $D^k[i, j] \leftarrow D^{k-1}[i, j]$ OR $(D^{k-1}[i, k]$ AND $D^{k-1}[k, j])$, $k \geq 0$

위상 순서(1)

- **AOV(activity on vertex) 네트워크**
  - 작업간의 선후 관계를 나타내는 방향 그래프
  - 정점 : 작업, 간선 : 작업들 간의 선후 관계
- **선행 관계(precedence relation)**
  - 정점들 간의 선행자와 후속자 관계
  - 선행자(predecessor)
    - 정점 i에서 j까지 방향 경로가 있을 때, i 는 j의 선행자
    - 직속 선행자(immediate predecessor)
  - 후행자(successor)
    - 정점 i 에서 j까지 방향 경로가 있을 때, j는 i 의 후행자
    - 직속 후행자(immediate successor)



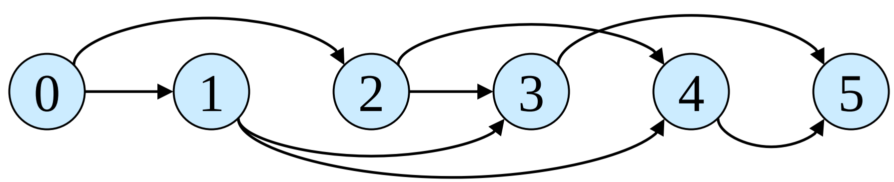
AOV 네트워크 G

위상 순서(2)

- **부분 순서(partial order)**
  - 이행적이고, 비반사적인 선행 관계일 때
  - 집합 S와 S에 대한 관계 R에서 S의 원소 i, j, k에 대하여,
    - R은 S에서 이행적(transivive) : $_iR_j$ & $_jR_k$이면 항상 $_iR_k$가 성립
    - R은 S에서 비반사적(irreflexive) : 모든 i에 대해 $_iR_i$ 성립하지 않음
  - 비대칭(asymmetric) : $_iR_j$이면, $_jR_i$는 성립하지 않음
  - DAG(directed acyclic graph)
- **위상 순서(topological order)**
  - 방향 그래프에서 두 정점 i와 j에 대해, i 가 j의 선행자이면 반드시 i 가 j보다 먼저 나오는 정점의 순차 리스트

위상 순서 : 0, 1, 2, 3, 4, 5

선행 관계 :



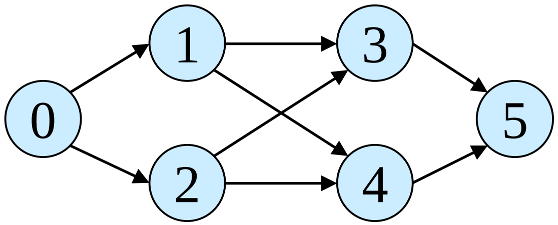
위상 순서 예

위상 순서(3)

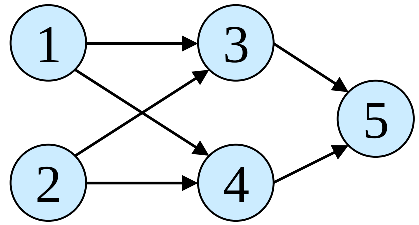
- 위상 정렬 알고리즘

```
topologicalSort(AOVnetwork, n)
   // G=(V, E), n=|V|
   for (i←0; i<n; i←i+1) do {
         select u with no predecessor;      // u∈V, indegree=0
         if (there is no such u) then return;
         print(u);
         remove u and all arcs incident from u;
   }
end topologicalSort
```

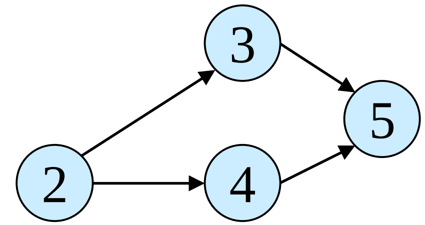
위상 순서(4)



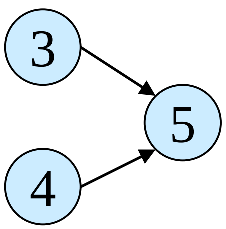
정점 0 삭제
위상 순서 : 0
(a)



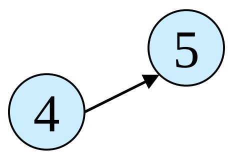
정점 1 삭제
위상 순서 : 0, 1
(b)



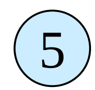
정점 2 삭제
위상 순서 : 0, 1, 2
(c)



정점 3 삭제
위상 순서 :
0, 1, 2, 3
(d)



정점 4 삭제
위상 순서 :
0, 1, 2, 3, 4
(e)



정점 5 삭제
위상 순서 :
0, 1, 2, 3, 4, 5
(f)

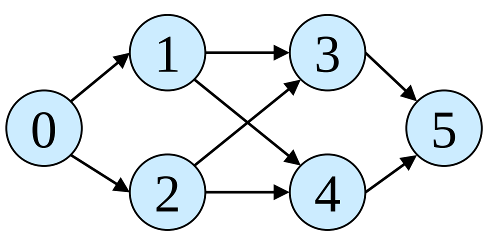
생성된
최종 위상 순서 :
0, 1, 2, 3, 4, 5
(g)

위상 정렬 알고리즘의 수행 과정

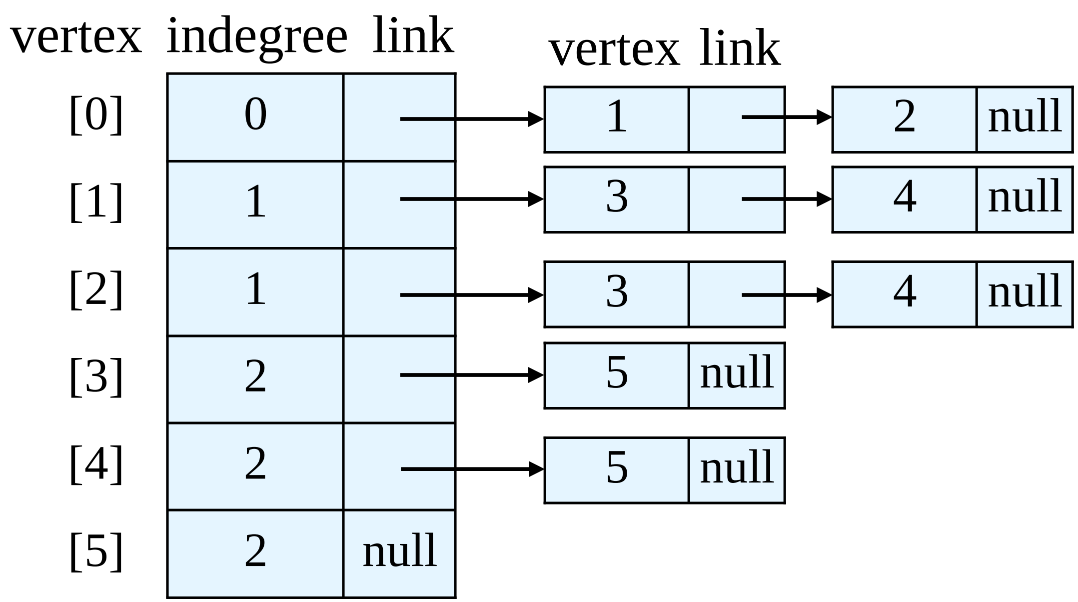
위상 순서(5)

◆ 위상 순서를 위한 인접 리스트 구조

- 기존 인접 리스트에 indegree 필드 추가
  - 정점의 진입 차수를 유지
  - 필드의 값은 초기에 인접 리스트를 구축하면서 결정
  - 필드값이 0인 정점들은 큐나 스택을 이용하여 관리



AOV 네트워크 G



위상 순서를 위한 AOV 네트워크 G에 대한 인접 리스트 구조

위상 순서(6)

◆ 위상 정렬 프로그램 (Java)

```
class Graph{
    Queue[] Q;      // 정점 i의 직속후속자를 저장하는 큐
    Queue ZeroPredQ;      //선행자가 없는 정점들을 저장하는 큐
    List sortedList;      // 위상 정렬 결과를 보관하는 리스트
    int[] indegree;      // 정점 i의 진입차수
    int n;      //정점 수
    public Graph(int vertices) {      // 생성자
        n = vertices;
        Q = new Queue[n];      // 큐 배열
        zeroPredQ = new Queue();
        sortedList = new List();
        for (int i=0; i<n; i++) {
            Q[i] = new Queue();      // 각 Q[i]에 대해 초기화
        }
        indegree = new int[n];
    }
```

위상 순서(7)

```
public void topologicalSort() {
    int i, v, successor;
    for (i=0; i<n; i++) {
        if (indegree[i]==0)     // 선행자 없음
            zeroPredQ.enqueue(i);
        }
    }
    if (zeroPredQ.isEmpty()) {
        System.out.println("network has a cycle");
        return;
    }
    while (!zeroPredQ.isEmpty()) {
            // indegree가 0인 정점들을 큐에서 하나씩 삭제해 처리
        v = zeroPredQ.dequeue();
            // ingree가 0인 정점들을 결과 리스트에 삽입
        sortedList.insert(v);
        if (Q[v].isEmpty()) continue;
            // 정점 v의 후속자가 없으면 밖의 while 루프로
        else successor = Q[v].dequeue();
            // 후속자가 있으면, 그 후속자를 successor로 설정
```

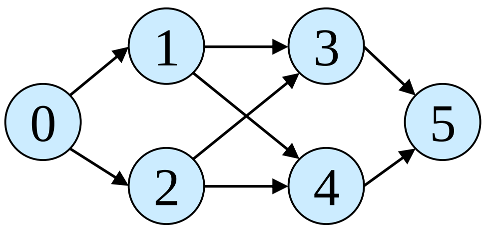
위상 순서(8)

```
                while (true) {      // v의 후속자 정점의 진입차수를 감소시킴
                    indegree[successor]--;
                    if(indegree[successor]==0)      // 0이 되면 zeroPredQ에 삽입
                        zeroPredQ.enqueue(successor);
                    if (!Q[v].isEmpty()) successor = Q[v].dequeue();
                    else break;
                }
            }     // end while
            System.out.println("Topological Order is : ");
            while (!sortedList.isEmpty())
                System.out.print(sortedList.remove() + " ");
            System.out.println ();
            System.out.println("End.");
        }     // end topologicalSort()

        public void insertEdge(int i, int j) {      // 인접 리스트를 큐 배열로 표현
            Q[i].enqueue(j);
            indegree[j]++;
        }     // end insertEdge()
}     // end Graph class
```

위상 순서(9)

```
class AOV_Topological_Sort {
   public static void main(String args[]) {
      Graph AOV = new Graph(6);

      AOV.insertEdge(0,1); // 정점 0의 간선들을 삽입
      AOV.insertEdge(0,2);

      AOV.insertEdge(1,3); // 정점 1의 간선들을 삽입
      AOV.insertEdge(1,4);

      AOV.insertEdge(2,3); // 정점 2의 간선들을 삽입
      AOV.insertEdge(2,4);

      AOV.insertEdge(3,5); // 정점 3의 간선들을 삽입
      AOV.insertEdge(4,5); // 정점 4의 간선들을 삽입

      AOV.topologicalSort(); //위상 정렬 함수 호출
   }   // end main()
}   // end AOV_Topological_Sort
```

위상 정렬의 main() 함수



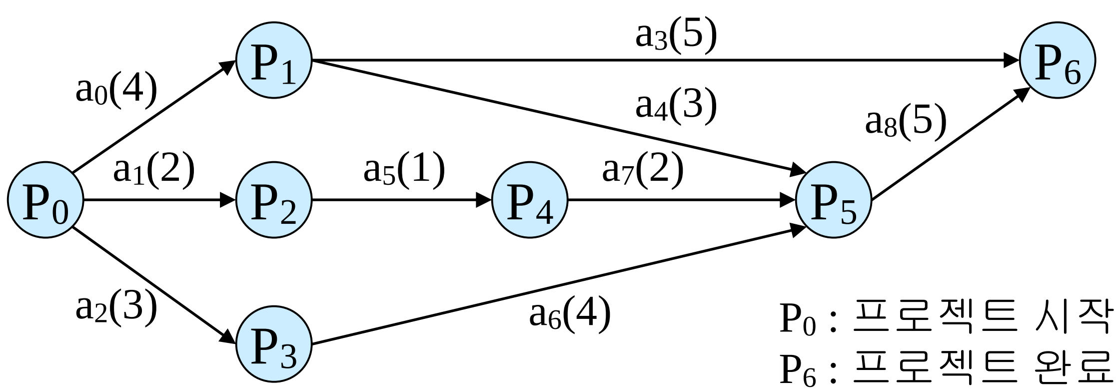
AOV 네트워크 G

```
% java AOV_Topological_Sort
Topological Order is :
0 1 2 3 4 5
End.
```

AOV 네트워크 G에 대한
위상 정렬의 실행 화면

임계 경로(1)

- **AOE(activity on edge) 네트워크**
  - 프로젝트의 스케줄을 표현하는 DAG
  - 정점 : 프로젝트 수행을 위한 공정 단계
  - 간선 : 공정들의 선후 관계와 각 공정의 작업 소요 시간
  - CPM, PERT 등 프로젝트 관리 기법에 사용됨
  - 7개의 공정(P_0, P_1, P_2, P_3, P_4, P_5, P_6)과 9개의 작업(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8)로 구성된 프로젝트 스케쥴



P_0 : 프로젝트 시작
P_6 : 프로젝트 완료
( ) : 소요시간

AOE 네트워크

임계 경로(2)

- **임계 경로(critical path)**
  - 시작점에서 완료점까지 시간이 가장 많이 걸리는 경로
  - 하나 이상의 임계 경로가 존재
- **공정 P_i의 조기 완료 시간(earliest completion time ; EC(i))**
  - 시작점에서부터 공정 P_i까지의 최장 경로 길이
  - EC(4)=3
EC(5)=7
EC(6)=12
- **공정 P_i의 완료 마감 시간(latest completion time ; LC(i))**
  - 전체 프로젝트의 최단 완료 시간에 영향을 주지 않고 공정 P_i가 여유를 가지고 지연하여 완료할 수 있는 시간
  - (전체 프로젝트 완료시간)-(공정P_i에서 최종공정까지 최장 경로 길이)
  - LC(4)=5
LC(5)=7

임계 경로(2)

- **공정 P_i의 임계도(criticality)**
  - EC(i)와 LC(i)의 시간 차이

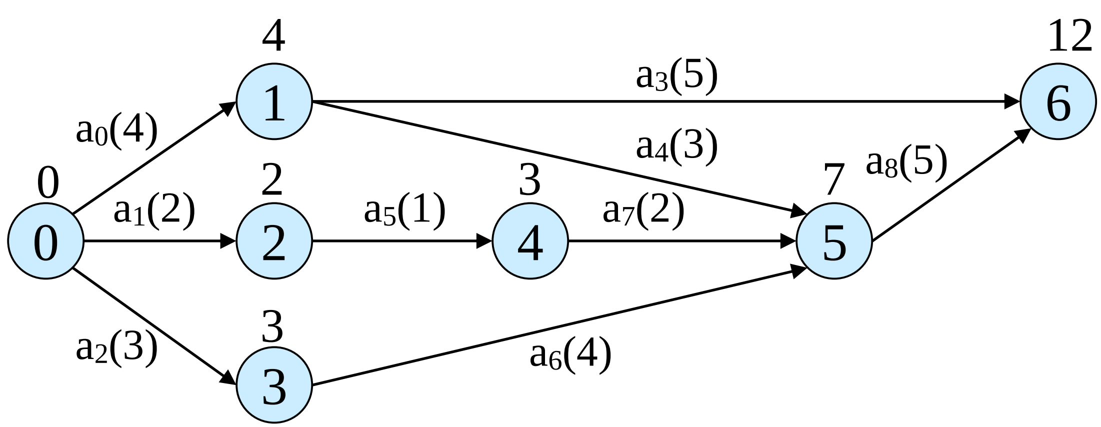
- **임계 작업(critical activity)**
  - 임계 경로 상에 있는 작업들
  - 작업 a(<i, j>) : EC(i)=LC(i)이고, EC(j)=LC(j)인 작업

- **임계 경로 분석(critical path analysis)의 목적**
  - 임계 작업을 식별해서 이들에 자원을 집중시킴으로 프로젝트 완료 시간을 단축

임계 경로(3)

◆ **공정 조기 완료 시간(EC(j)) 계산**

- weight(i, j) : 작업 <i, j>에 소요되는 작업 시간
- EC(0) ← 0
- EC(j) ← max {EC(i) + weight(i, j), j로 진입하는 모든 i}
- AOE 네트워크의 위상 순서에 따라 계산

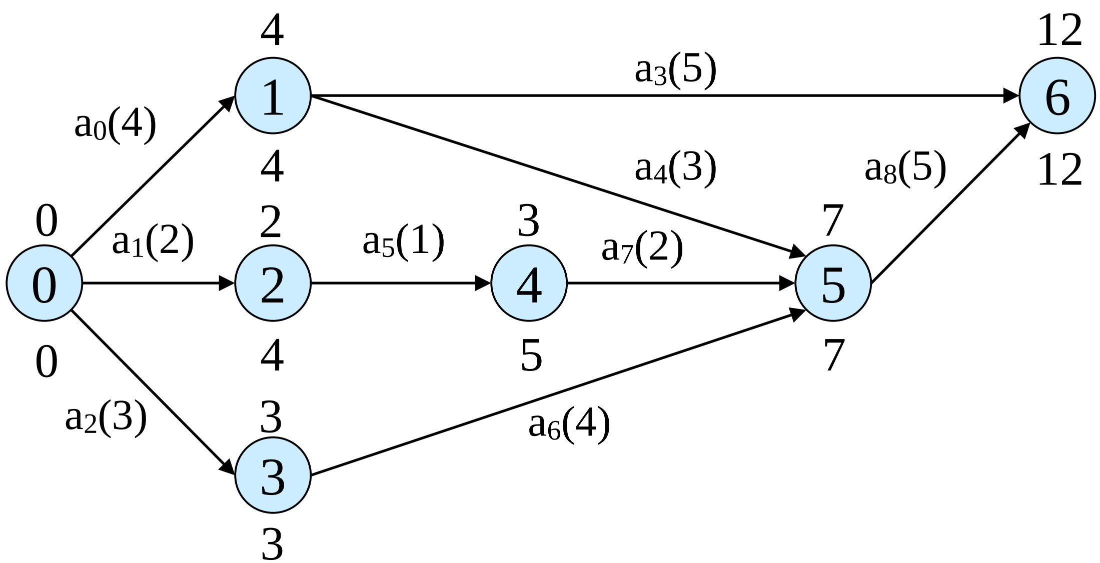


각 공정의 조기 완료 시간 E(i) (정점 위의 숫자)

- EC(6)=12는 프로젝트를 완료하는데 걸리는 최소한의 소요시간

임계 경로(4)

- **공정 완료 마감 시간(LC(i)) 계산**
  - LC(n-1) ← EC(n-1)
  - LC(i) ← min {LC(j) – weight(i, j), i에서 진출하는 모든 j}
  - AOE 네트워크 위상 순서의 역순으로 계산

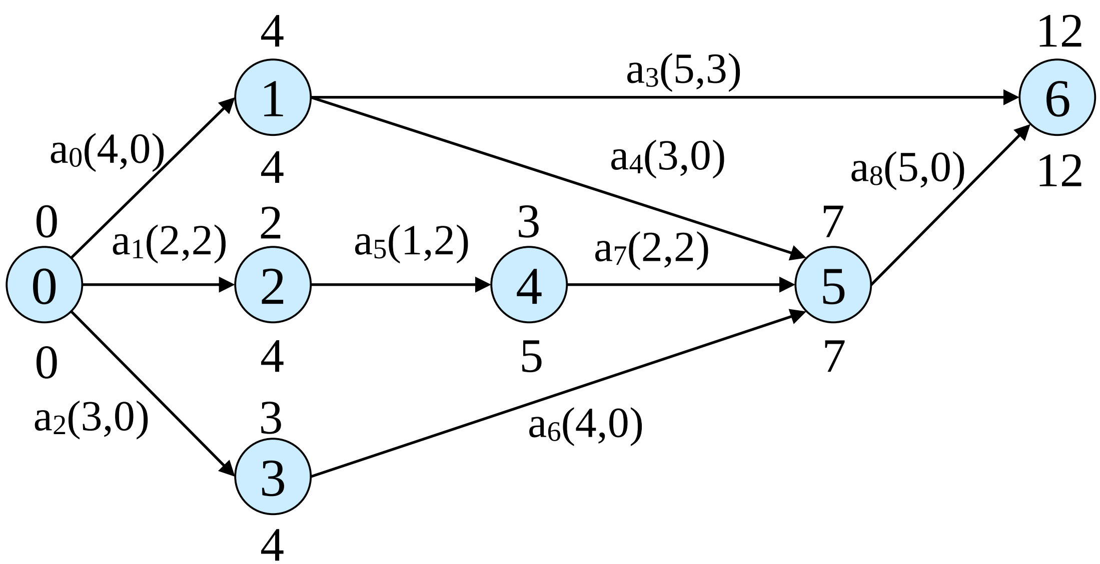


각 공정의 완료 마감 시간 LC(i) (정점 아래 숫자)

임계 경로(5)

◆ 작업 임계도(CR(i, j)) 계산

- $CR(\langle i, j \rangle) \leftarrow LC(j) - (EC(i) + weight(i, j))$

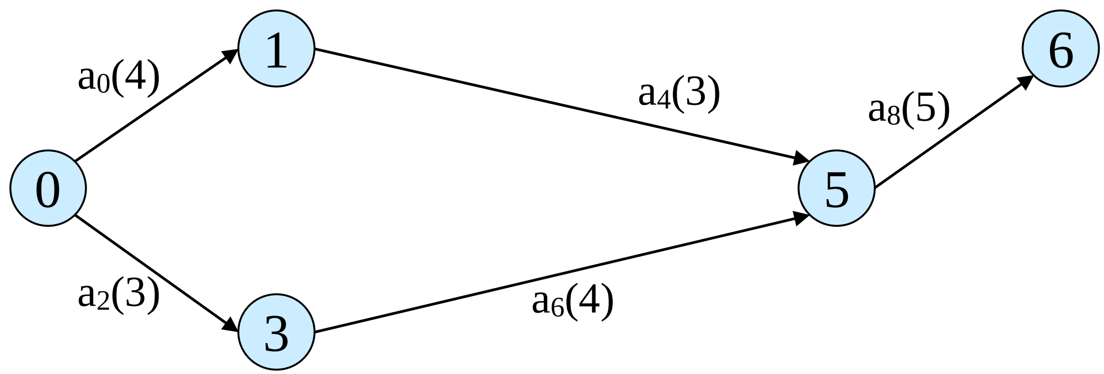


작업의 임계도(괄호 안의 두 번째 숫자)

임계 경로(6)

◆ **임계 작업으로 구성된 임계 경로**

- 네트워크에서 임계도가 0인 임계 작업만 남기고 제거



임계 작업으로 구성된 임계 경로

- 임계 경로
  0, 1, 5, 6
  0, 3, 5, 6