# 제 6 장 그래프(계속)

# 6.3 최소비용 신장 트리

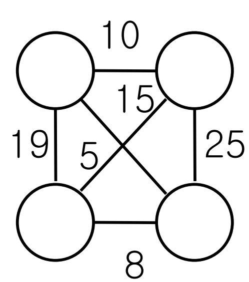
- 최소비용 신장 트리(minimum cost spanning tree):
  가중치 무방향 그래프에서 신장 트리의 비용은 신장 트리에 포함된 간선들의 비용이며, 최소 비용 신장 트리는 최소의 비용을 갖는 신장 트리이다.

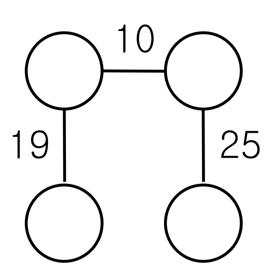
- Kruskal, Prim, Sollin 알고리즘이 있다.
- 최소 비용 신장 트리는 다음의 제한적인 조건을 갖는다.

(i) 그래프 내의 간선들만을 사용해야 한다.

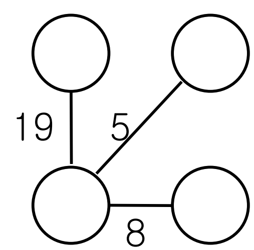
(ii) n-1 개의 간선만을 포함해야 한다.

(iii) 사이클을 포함하지 않는다.




비용: 54



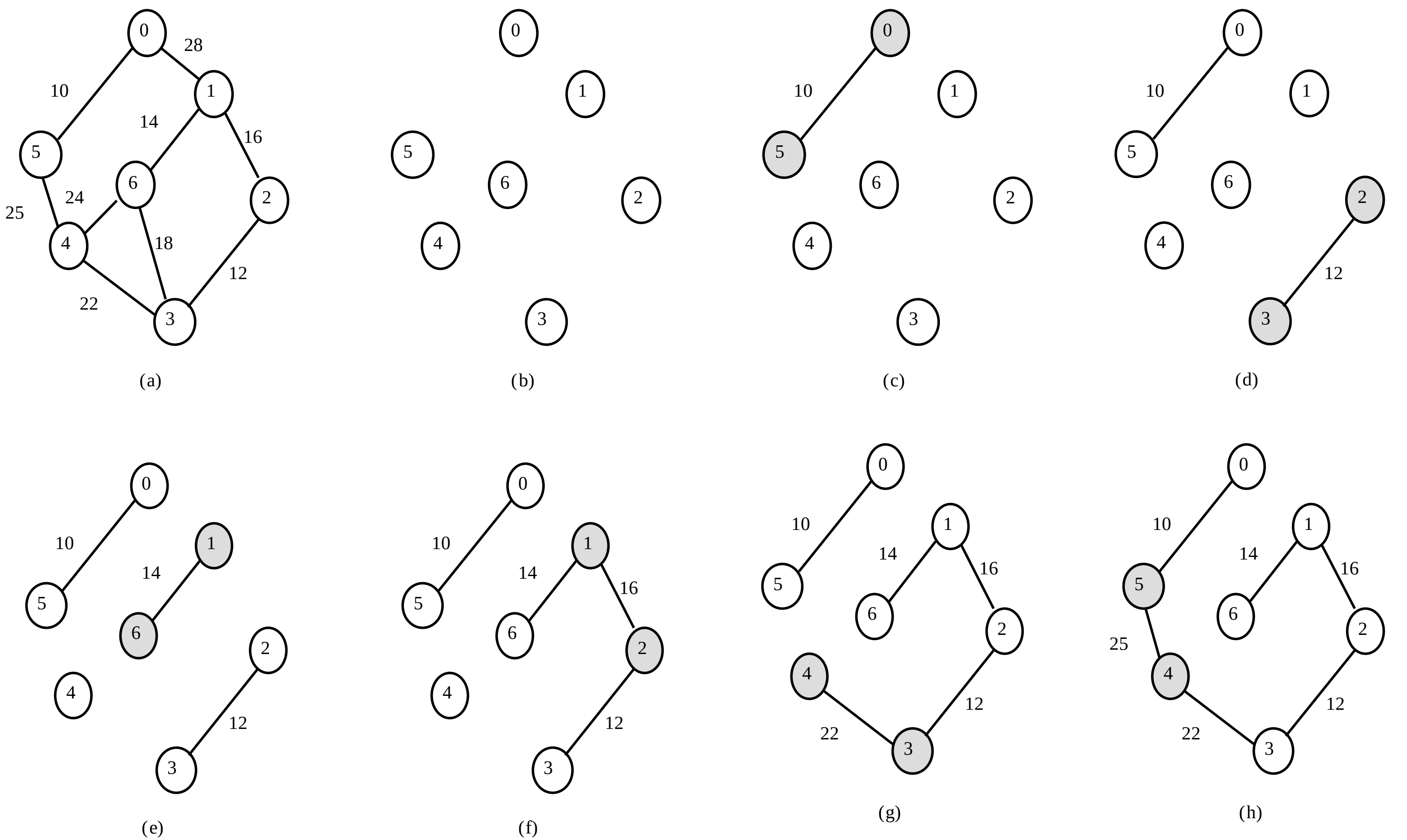

비용: 32

# (1) Kruskal 알고리즘

- 한 번에 하나씩 비용이 가장 작은 간선을 선택하여 T 에 이미 포함된 간선들과 사이클을 형성하지 않는 간선들만을 차례로 T에 추가한다. T 에 n-1개의 간선들이 존재하면 멈춘다.
- 알고리즘

```
T = Ø
while ((T가 n-1개 미만의 간선을 포함) && (E가 공백이 아님)) {
    E에서 최소 비용 간선 (v,w) 선택;
    E에서 (v,w)를 삭제;
    if ((v,w)가 T에서 사이클을 형성하지 않음) T에 (v,w)를 추가;
    else (v,w)를 거부;
}
if (T가 n-1개 미만의 간선을 포함) cout << "신장 트리 없음"<<endl;
```

# Kruskal 알고리즘의 예

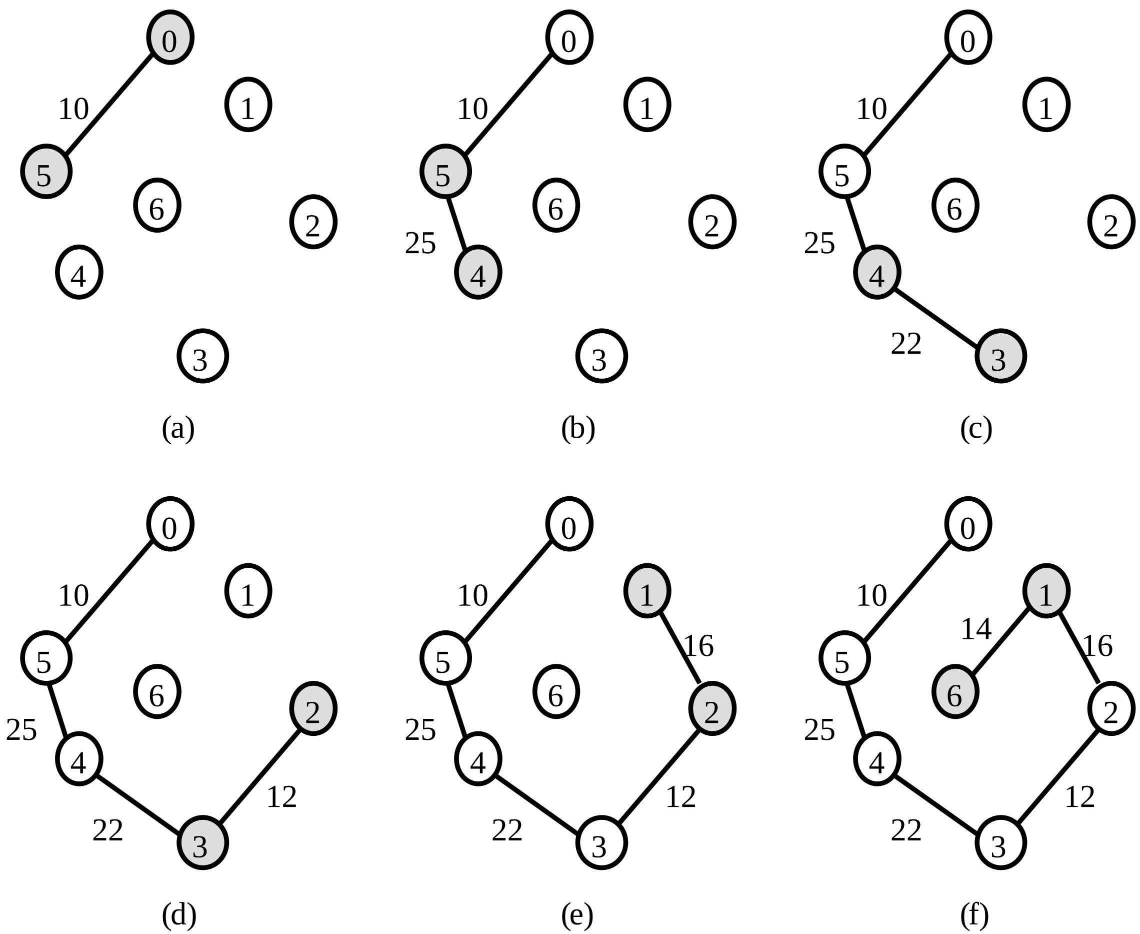


# (2) Prim 알고리즘

- 트리 T에 인접한 간선들 중 T와 사이클을 형성하지 않는 최소 비용 간선 (u, v)를 구해 T에 추가한다. T에 n-1 개의 간선이 포함될 때까지 이러한 추가 단계를 반복한다. (초기에 T의 정점 집합은 하나의 정점만을 포함한다.)
- 알고리즘

```
// G가 최소한 하나의 정점을 가진다고 가정.
TV = {0}; // 정점 0으로 시작. 간선은 비어있음.
for (T=Ø ; T의 간선수가 n-1보다 적음; (u,v)를 T에 추가)
{
  u ∈ TV이고 v ¬∈ TV인 최소 비용 간선을 (u,v)라 함;
  if (그런 간선이 없음) break;
  v를 TV에 추가;
}
if (T의 간선수가 n-1보다 적음) cout << "신장트리 없음" << endl;
```

# Prim 알고리즘의 예



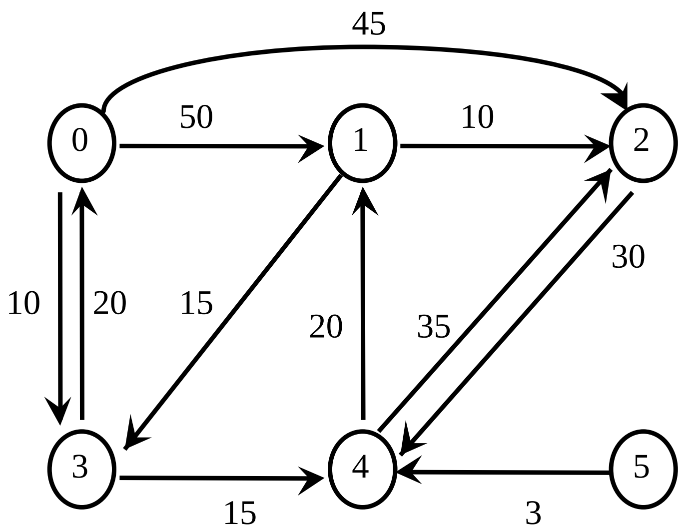
0
1
10
5
6
2
4
3
(a)
0
1
10
5
6
2
25
4
3
(b)
0
1
10
5
6
2
25
4
22
3
(c)
0
1
10
5
6
2
25
4
12
22
3
(d)
0
1
10
16
5
6
2
25
4
12
22
3
(e)
0
1
10
14
16
5
6
2
25
4
12
22
3
(f)

# 6.4 최단 경로

- 가중치 그래프, 출발점(source), 종점(destination), 방향 그래프
- 두 가지 문제:
  - 단일 출발점/모든 종점(간선의 길이가 양수인 경우)
  - 모든 쌍의 최단 경로

# (1)단일 출발점/모든 종점 (간선의 길이가 양수일 때)

- 방향그래프 G=(V,E)와 G의 간선에 대한 가중치 함수 length(i,j) ≥0 와 출발점 v가 주어진다.




(a) 그래프

| *Path* | *Length* |
|---|---|
| 1) 0, 3 | 10 |
| 2) 0, 3, 4 | 25 |
| 3) 0, 3, 4, 1 | 45 |
| 4) 0, 2 | 45 |

(b) 0에서부터 최단 경로

# 최단 경로 구하기의 예(비용)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 50 | 45 | 10 | ∞ | ∞ |
| 1 | ∞ | 0 | 10 | 15 | ∞ | ∞ |
| 2 | ∞ | ∞ | 0 | ∞ | 30 | ∞ |
| 3 | 20 | ∞ | ∞ | 0 | 15 | ∞ |
| 4 | ∞ | 20 | 35 | ∞ | 0 | ∞ |
| 5 | ∞ | ∞ | ∞ | ∞ | 3 | 0 |

min{ ∞, 10+15 }

| S | 선택 정점(u) | dist 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| | | 0 | 50 | 45 | 10 | ∞ | ∞ |
| 0 | 3 | 0 | 50 | 45 | 10 | 25 | ∞ |
| 0, 3 | 4 | 0 | 45 | 45 | 10 | 25 | ∞ |
| 0, 3, 4 | 1 | 0 | 45 | 45 | 10 | 25 | ∞ |
| 0, 1, 3, 4 | 2 | 0 | 45 | 45 | 10 | 25 | ∞ |
| 0, 1, 2, 3, 4 | | | | | | | |

min{ 50, 25+20 }

# Dijkstra 알고리즘

- 단계 1: 출발점 v를 포함하여 이미 최단 경로가 발견된 정점들의 집합을 S라 하자.
- 단계 2: S에 속하지 않는 정점들 중 dist의 값이 가장 작은 정점 u를 선택한다. 여기서 선택된 정점 u는 S의 원소가 된다.
- 단계 3: S에 속하지 않는 정점들에 대해 새로운 최단 경로가 존재하는 경우, 이 정점의 dist 값을 갱신한다.

  즉 dist[w] = min { dist[w], dist[u]+ length[u, w] },

  (단 w는 S에 포함 안된 정점)
- 단계 4: S에 n-1 개의 정점이 포함될 때까지 단계 2, 3을 반복한다.

  → n-2 번 반복한다(초기에 시작 정점이 S에 포함되기 때문)

# 클래스 정의

```
class Graph
{
private:
  int   length[nmax][nmax];   // 이차원 배열
  int   dist[nmax];           // 일차원 배열
  bool s[nmax];               // 일차원 배열
  int n ;                     // 정점의 수
public:
  Graph(const int vertices = 0): n(vertices) {} ;
  void ShortestPath(const int);
  int choose(const int);
};
```

- s[nmax]는 정점이 S에 포함된지를 boolean 값으로 표시

# 클래스 정의(동적생성)

```
class Graph
{
private:
  int   **length;              // 이차원 배열
  int   *dist;                 // 일차원 배열
  bool *s;                     //  일차원 배열
   int n;                      // 정점의 수
public:
   Graph(const int vertices = 0): n(vertices)
      { length = new int*[n];   // 이차원 배열의 동적 생성
         for(int i=0; i<n; i++) length[i] = new int [n];  ..... };
  void ShortestPath(const int);
  int choose(const int);
};
```

* s[n]는 정점이 S에 포함된지를 boolean 값으로 표시

# 최단 경로 함수

```
void Graph::ShortestPath(const int v)
// dist[j],0 ≤ j< n은 n개의 정점을 가진 방향 그래프 G에서 정점 v로부터 정점 j
// 까지의 최단 경로 길이로 설정됨. 간선의 길이는 length[j][j]로 주어짐.
{
  for (int i=0; i<n; i++) {s[i]=FALSE; dist[i]=length[v][i];}// 초기화
  s[v] = TRUE;
  dist[v] = 0;
  for (i=0; i<n-2; i++) { // 정점 v로부터 n-1개 경로를 결정
    int u = choose(n); // choose는 dist[u] = minimum dist[w]인 u를 반환
                       // (여기서 s[w]=FALSE)
     s[u] = TRUE;
     for (int w=0; w<n; w++)
       if(!s[w])
         if(dist[u] + length[u][w] < dist[w])
                   dist[w] = dist[u] + length[u][w];
} // for(i=0; ...)의 끝
}
```

# 최단 경로 구하기의 예(경로)

**min{ ∞, 10+15 }**

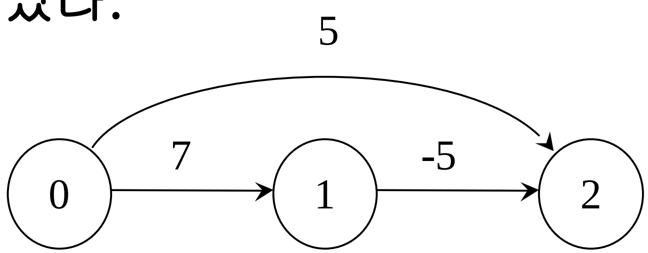
| | | dist | | | | | | path | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **S** | **선택 정점(u)** | **0** | **1** | **2** | **3** | **4** | **5** | **0** | **1** | **2** | **3** | **4** | **5** |
| | | **0** | **50** | **45** | **10** | **∞** | **∞** | **0** | **0** | **0** | **0** | **0** | **0** |
| **0** | **3** | **0** | **50** | **45** | **10** | **25** | **∞** | **0** | **0** | **0** | **0** | **3** | **0** |
| **0, 3** | **4** | **0** | **45** | **45** | **10** | **25** | **∞** | **0** | **4** | **0** | **0** | **3** | **0** |
| **0, 3, 4** | **1** | **0** | **45** | **45** | **10** | **25** | **∞** | **0** | **4** | **0** | **0** | **3** | **0** |
| **0, 1, 3, 4** | **2** | **0** | **45** | **45** | **10** | **25** | **∞** | **0** | **4** | **0** | **0** | **3** | **0** |
| **0, 1, 2, 3, 4** | | | | | | | | | | | | | |

**min{ 50, 25+20 }**      **0**은 시작 정점

- 일차원 배열 path를 역추적하여 최단경로를 구한다.
  정점 1: 1 – 4 – 3 – 0 → 0 – 3 – 4 – 1
  정점 2: 2 – 0 → 0 – 2
  …

# 최단 경로 알고리즘의 분석

- 분석: 8행의 for 루프는 n−2번 실행
  for 루프 내의 9행은 최대 n 번 실행, 그리고 12번행 역시
  최대 n 번 실행, 따라서 $O(n^2)$ 시간 걸린다.
- 간선이 음의 길이를 가질 때, Dijkstra 알고리즘이 정확히 작동하지 않을 수 있다.




- 정점 2까지의 길이는 dist[2]=5 이지만 실제 2가 최단 경로의 길이이다.

# (2) 모든 쌍들의 최단 경로

- G = (V, E)를 n 개의 정점들을 갖는 방향 그래프라 하고, length는 간선들 상의 가중치를 갖는 인접 행렬이다.
- 모든 쌍들의 최단 경로 문제는 서로 다른 정점의 쌍 u와 v 간의 최단 경로의 길이를 구하는 문제이다(단 음수 길이의 사이클이 존재하지 않을 때). 즉, A(i, j)가 정점 i 로부터 j 까지의 최단 경로의 길이인 행렬 A 를 결정하는 것이다.
- 방법 1: 각 정점을 출발점으로 하여 Dijkstra 알고리즘을 n번 적용
- 방법 2: 동적 프로그래밍 적용(Ford 알고리즘)
- 위의 두 방법들 모두 시간복잡도가 $O(n^3)$ 이나 방법 2가 훨씬 더 빠르다.

# Ford 알고리즘의 원리

- i 에서 j 로의 최단 경로에서 k가 이 최단 경로 상의 중간에 있는 정점이라면, i 에서 k로의 부분 경로와 k 에서 j로의 부분 경로가 각각 최단 경로가 되어야만 한다. 만약 그렇지 않다면 다른 경로가 최단 경로가 되기 때문이다.
- 원리:

  $A^k(i,j)$ = k 보다 큰 인덱스를 갖는 중간 정점을 통과하지 않고 i 에서 j 로 가는 최단 경로 길이

  $A^{-1}, A^0, \ldots, A^{n-1}$ 을 순서대로 구할 때 $A^{n-1}$ 이 모든 쌍의 최단 경로 길이가 된다
- 순환식:

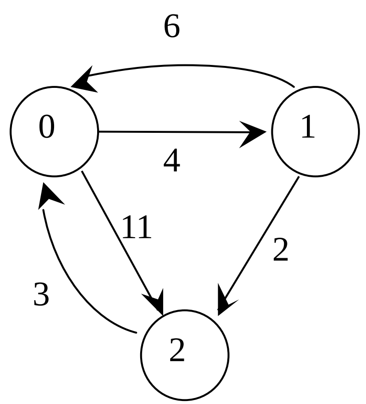
  $A^{-1}(i,j) = \text{length}(i, j),\ 0 \le i \le n-1$ 이다.

# Ford 알고리즘의 원리(계속)

- $A^{k-1}$ 에서 $A^k$ 를 만드는 방법:
  임의의 정점 쌍 i, j 에 대하여 다음의 두 규칙 중 한 가지를 적용한다.
  (i) 쌍 i, j의 최단 경로가 정점 k를 통과하지 않을 때
  $A^k(i,j) = A^{k-1}(i,j)$
  (ii) 쌍 i, j의 최단 경로가 정점 k를 통과할 때
  $A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j)$

  따라서 (i) (ii)를 한 식으로 나타내면,
  $A^k(i,j) = \min \{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}, 0 \le k \le n-1$

# 모든 쌍들의 최단 경로 예




(a) 예제 방향그래프

| $A^{-1}$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 4 | 11 |
| 1 | 6 | 0 | 2 |
| 2 | 3 | $\infty$ | 0 |

(b) $A^{-1}$

| $A^{0}$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 4 | 11 |
| 1 | 6 | 0 | 2 |
| 2 | 3 | 7 | 0 |

min {∞, 3+4}

(c) $A^{0}$

| $A^{1}$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 4 | 6 |
| 1 | 6 | 0 | 2 |
| 2 | 3 | 7 | 0 |

min {11, 4+2}

(d) $A^{1}$

| $A^{2}$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 4 | 6 |
| 1 | 5 | 0 | 2 |
| 2 | 3 | 7 | 0 |

min {6, 2+3}

(e) $A^{2}$

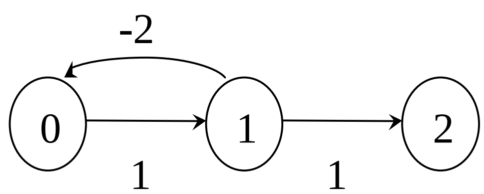
# 모든 쌍들의 최단 경로 알고리즘

```
void Graph::AllLengths(const int n)
// length[n][n]은 n개의 정점을 가진 그래프의 인접 행렬
// a[i][j]는 i와 j 사이의 최단 경로의 길이
{
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      a[i][j] = length[i][j]; // length를 a에 복사
  for(int k=0; k<n; k++) // 제일 큰 정점의 인덱스가 k인 경로에 대해
    for(i=0; i<n; i++) // 가능한 모든 정점의 쌍에 대해
      for(int j=0; j<n; j++)
        if((a[i][k]+a[k][j])<a[i][j]) a[i][j] = a[i][k] + a[k][j];
}
```

- 시간복잡도: $O(n^3)$

# 음인 길이의 순환 경로를 갖는 경우

- 이 경우는 AllLengths 알고리즘이 정확히 동작하지 않는다.
- 예:




- 이 그래프에서 $A^1(0,2) = \min\{ A^0(0,2), A^0(0,1)+A^0(1,2) \} = 2$ 로 계산된다.

  그러나 실제 최단 경로의 비용은 $A^1(0,2) = -\infty$ 이어야 한다.

  이것은 경로 0,1,0,1,0,1, …,0,1,2 의 길이를 임의로 작게 만들 수 있기 때문이다.

# 레포트 #4

- 가중치 그래프(인접 행렬을 사용)를 입력하고, 출발점을 입력하여 최단경로(Dijkstra 알고리즘)를 구하는 프로그램을 작성하라.
- 입력:

  정점 수와 간선 수 입력 > n m
  1번째 간선과 가중치 입력> 0 1 30
  2번째 간선과 가중치 입력> 0 3 25
  :
  m 번째 간선과 가중치 입력>3 6 22
  시작 정점 입력> 0
- 출력:

  – 가중치를 갖는 인접행렬
  – 최종 결과인 배열 dist 의 값
  – 각 정점까지의 최단 경로

** 3개의 그래프에 대해서 테스트할 것.