8 장. 그래프

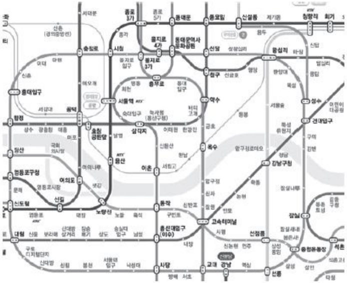
Jinseog Kim

학습 내용

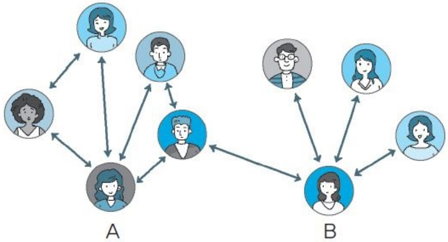
1. 그래프란?
2. 그래프의 표현
3. 그래프 탐색
4. 신장 트리와 최소비용 신장 트리
5. 최단 경로

그래프란?

- 연결되어 있는 객체 간의 관계를 표현하는 자료구조
  - 가장 일반적인 자료구조 형태



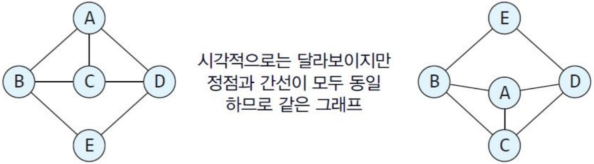
지하철 노선도



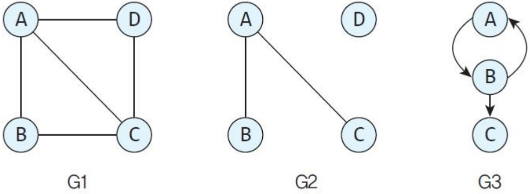
SNS 인맥 관계

- 그래프의 예
  - 지도, 지하철 노선도, 전기회로의 연결 상태
  - OS 의 프로세스와 자원 관계, 인맥지도 등

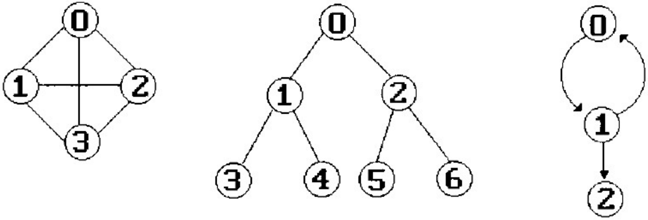
- 그래프 G 는 (V, E) 로 표시
  - 정점 (vertices) 또는 노드 (node): 여러 가지 특성을 가질 수 있는 객체 의미
  - 간선 (edge) 또는 링크 (link): 정점들 간의 관계 의미



  - 동일한 그래프? = 동형



- 간선의 종류에 따라
  - 무방향 그래프 (undirected graph): 간선을 통해서 양방향으로 갈수 있음
    (A, B) 로 표현: (A, B) = (B, A)
  - 방향 그래프 (directed graph): 간선을 통해서 한쪽 방향으로만 갈 수 있음
    <A, B> 로 표현: <A, B> ≠ <B, A>



G_1 G_2 G_3

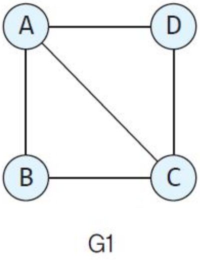
- V(G1) = {0,1,2,3}, E(G1) = {(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}
- V(G3} = {O, 1,2}, E(G3) = {<0, 1>, <l,0>, <1,2>}

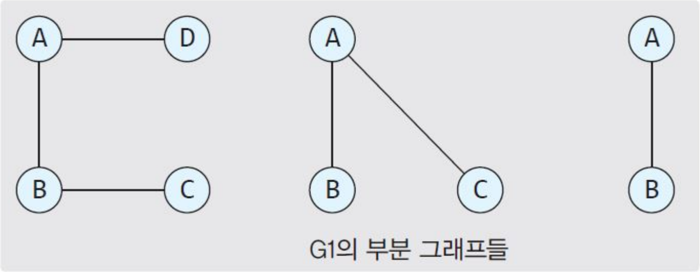
가중그래프 (weighted graph)

- 간선에 비용이나 가중치가 할당된 그래프
  - 네트워크 (network) 라고도 함
- 가중치 그래프 예
  - 정점: 각 도시를 의미
  - 간선: 도시를 연결하는 도로 의미
  - 가중치: 도로의 길이

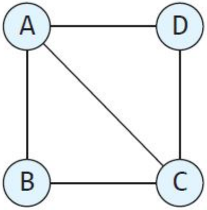
부분 그래프 (subgraph)

- 정점 집합 V(G) 와 간선 집합 E(G) 의 부분 집합으로 이루어진 그래프

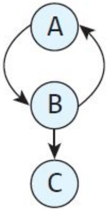




그래프의 용어



G1



G3

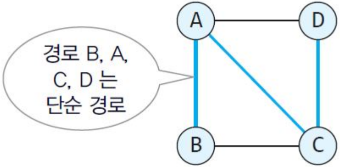
1. 인접 정점 (adjacent vertex): 하나의 정점에서 간선에 의해 직접 연결된 정점
   - G1 에서 정점 0 의 인접 정점: 정점 1, 정점 2, 정점 3
2. 차수 (degree): 하나의 정점에 연결된 다른 정점의 수
   1. 무방향 그래프
      G1 에서 정점 0 의 차수: 3
      차수의 합은 간선 수의 2 배
   2. 방향 그래프
      진입차수 (incoming degree), 진출차수 (outgoing degree)
      모든 진입 (진출) 차수의 합은 간선의 수

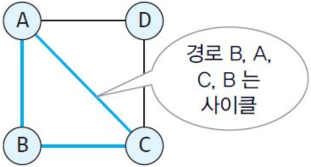
- 그래프의 경로 (path)
  - 무방향 그래프의 정점 s 로부터 정점 e 까지의 경로
  - 정점의 나열 $s, v_1, v_2, \ldots, v_k, e$
  - 반드시 간선 $(s, v_1), (v_1, v_2), \ldots, (v_k, e)$ 존재해야 함
  - 방향 그래프의 정점 s 로부터 정점 e 까지의 경로

    정점의 나열 $s, v_1, v_2, \ldots, v_k, e$

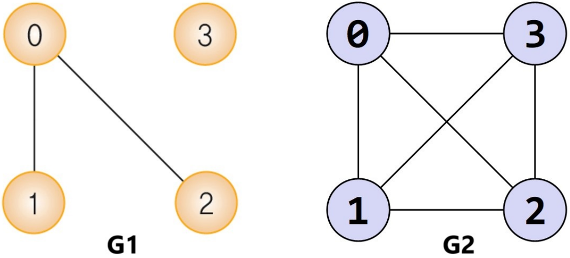
    반드시 간선 $< s, v_1 >, < v_1, v_2 >, \ldots, < v_k, e >$ 존재해야 함
- 경로의 길이 (length): 경로를 구성하는데 사용된 간선의 수

- 단순 경로 (simple path): 경로 중에서 반복되는 간선이 없는 경로
  - B, A, C, D: 단순경로
  - B, A, D, A: 단순경로 아님
- 사이클 (cycle): 시작 정점과 종료 정점이 동일한 경로
  - A, B, C, A: 사이클





- 연결 그래프 (connected graph): 모든 정점쌍에 대한 경로 존재
  - G1 는 비연결 그래프임
- 트리 (tree): 그래프의 특수한 형태로서 사이클을 가지지 않는 연결 그래프
- 완전 그래프 (complete graph): 모든 정점이 연결되어 있는 그래프 (G2)
  - n 개의 정점을 가진 무방향 완전그래프의 간선의 수: $n \times (n-1)/2$
  - $n = 4$, 간선의 수 $= (4 \times 3)/2 = 6$

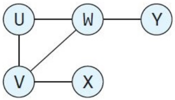


그래프의 추상 자료형 (ADT)

- **데이터 (객체):** 정점과 간선으로 이루어진 객체, G=(V, E),
  - 가중그래프는 가중치가 포함됨 G=(V, E, W)
- **연산 (함수)**
  - `is_empty()`: 그래프의 공백 상태 검사
  - `insert_vertex(v)`: 그래프에 정점 `v` 를 추가삽입
  - `insert_edge(u, v)`: 그래프에 간선 `u, v` 를 추가
  - `delete_vertex(v)`: 그래프의 정점 `v` 를 삭제
  - `delete_edge(u, v)`: 그래프의 간선 `u, v` 를 삭제
  - `adjcent(v)`: 그래프의 정점 `v` 의 인접 정점을 반환
  - `degree(v)`: 그래프의 정점 `v` 의 차수 (degree) 을 반환

그래프의 표현 - 인접 행렬을 이용한 표현

- 무방향 그래프



(a) 무방향 그래프

	0	1	2	3	4
U → 0	0	1	1	0	0
V → 1	1	0	1	1	0
W → 2	1	1	0	0	1
X → 3	0	1	0	0	0
Y → 4	0	0	1	0	0

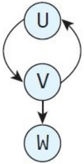
간선 (U,Y) 는 없음

간선 (W,Y) 가 있음

무방향 그래프는 항상 대칭 행렬

(b) 간선의 인접 행렬 표현

- 방향 그래프



(a) 방향 그래프

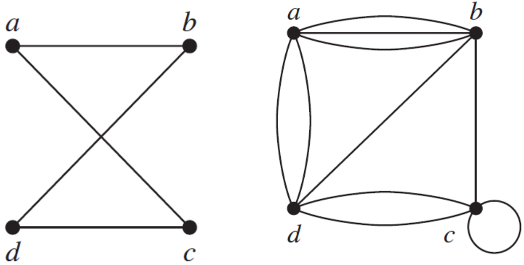
	0	1	2
U → 0		1	0
V → 1	1		1
W → 2	0	0	

방향 그래프에서는 대칭 행렬이 아님

(b) 간선의 인접 행렬 표현

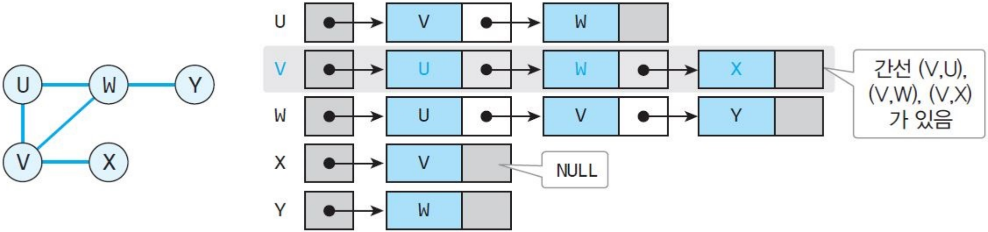
dongguk UNIVERSITY WISE

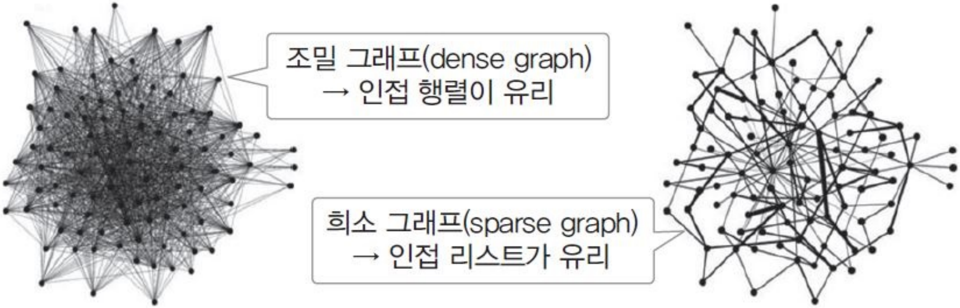
- 아래의 예는 위의 그래프를 4×4 아래의 인접행렬로 표현한 것임



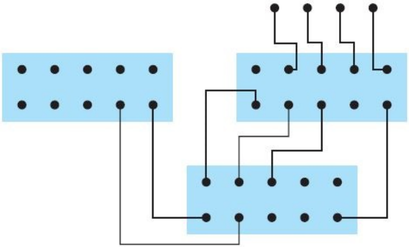
$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}$$

그래프의 표현 - 인접 리스트를 이용한 표현

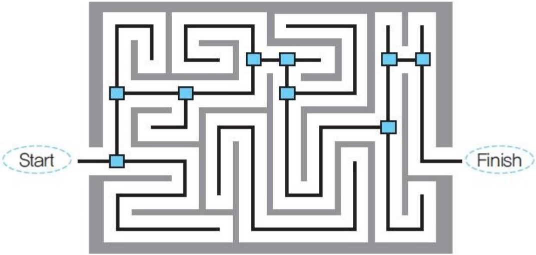



조밀 그래프(dense graph)
→ 인접 행렬이 유리
희소 그래프(sparse graph)
→ 인접 리스트가 유리

- 많은 문제가 단순히 그래프 탐색만으로 해결됨

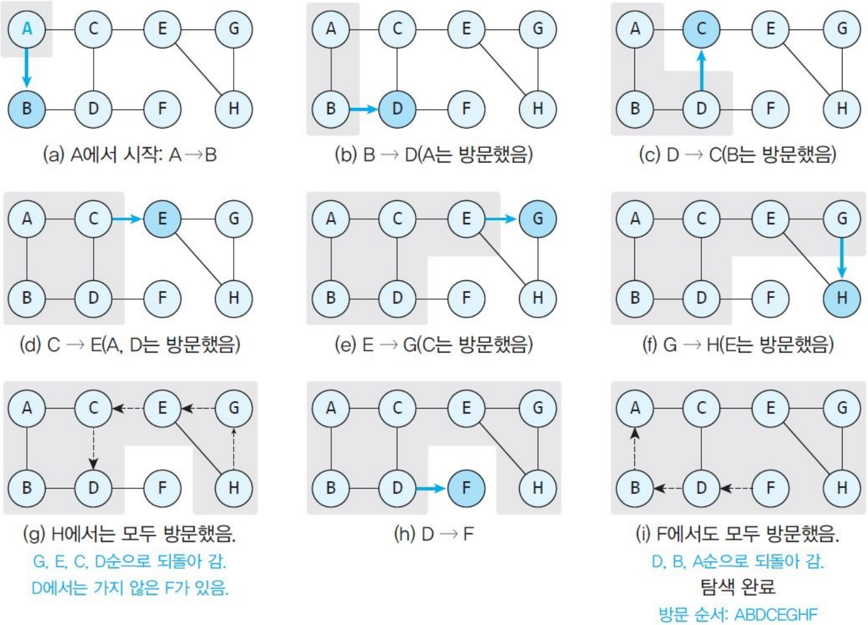


단자들 간의 연결성 검사



미로 탐색 문제

그래프 탐색 - 깊이 우선 탐색 (depth first search, DFS)



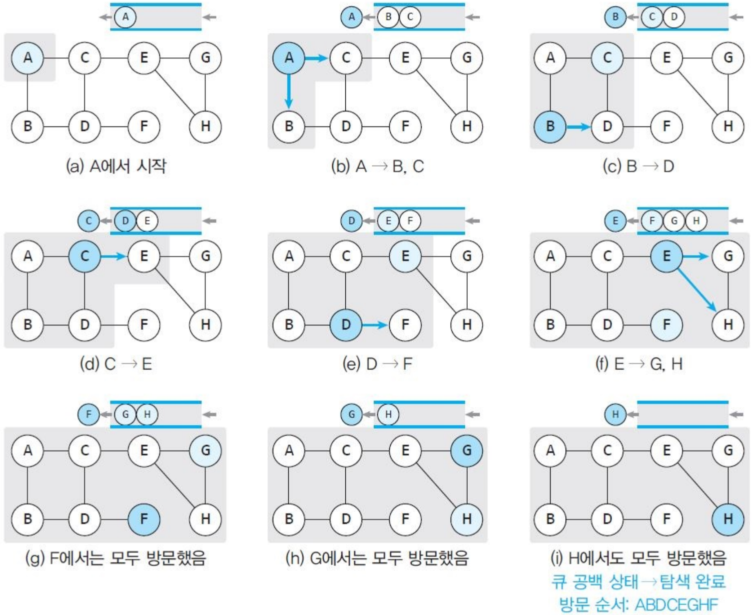
그래프 탐색 - 깊이 우선 탐색 알고리즘 구현

1. 깊이 우선 탐색은 시작점 v 부터 방문함
2. v 에 인접한 정점 중에서 방문하지 않은 정점 w 를 방문하고, 다시 w 로부터 탐색을 시작함
3. 어떤 정점 u 를 방문하고 u 에 인접한 모든 정점들을 이미 방문한 경우에는 그 전에 마지막으로 방문한 정점으로 되돌아가서 위의 과정들을 반복함
4. 모든 정점들을 방문한 후 탐색을 종료함

- 순차적인 프로그램보다는 DFS 알고리즘을 재귀 (recursive) 알고리즘으로 구현하는 것이 좋음
- 재귀 알고리즘으로 구현할 경우에는 스택 (stack) 을 사용함

procedure DFS(G, v) # 꼭지점 v 를 시작점
  for all $w \in N(v)$ 인접 꼭지점 **do**
    if w 를 방문하였는지 체크, 아직 방문하지 않았으면 **then**
      call DFS(G, w)

그래프 탐색 - 너비 우선 탐색 (breath first search: BFS)



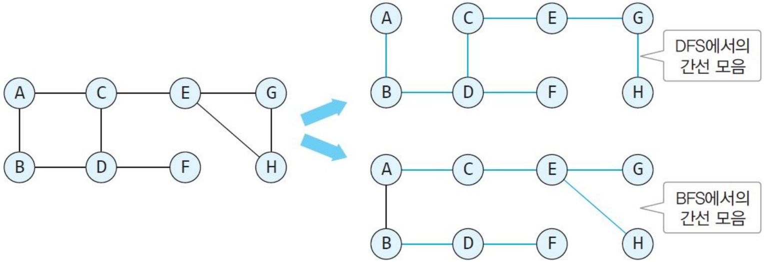
그래프 탐색 - 너비 우선 탐색 알고리즘 구현

1. 처음에 방문한 정점과 인접한 정점들을 차례로 방문한다는 점에서 깊이 우선 탐색과 차이가 있음
2. 먼저 시작점 v 를 방문한 후 v 에 인접한 모든 정점들을 차례로 방문함
3. 더 이상 방문할 정점이 없는 경우 다시 v 에 인접한 정점 가운데 맨 처음으로 방문한 정점과 인접한 정점들을 차례로 방문함
4. v 에 인접한 정점 중 두 번째로 방문한 정점과 인접한 정점들을 차례로 방문하는 과정을 반복함
5. 모든 정점들을 방문한 후 탐색을 종료함

- 깊이 우선 탐색이 스택 (stack) 을 사용하는데 비해 너비 우선 탐색은 큐 (queue) 를 사용함

그래프 탐색 - 너비 우선 탐색 알고리즘 구현

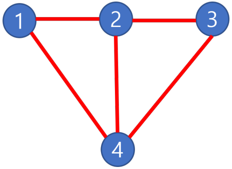
- **신장 트리 (spanning tree):** 그래프의 모든 정점을 포함하는 트리
- **생성트리, 스패닝트리**라고도 함



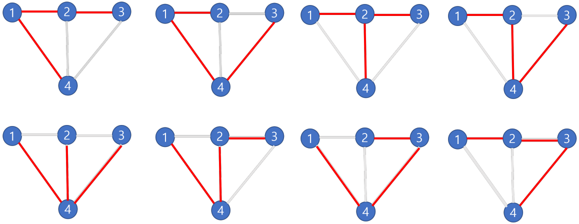
- **생성트리의 비용 (cost of spanning tree):** 생성트리에서 간선의 가중치를 모두 합친 값

신장 트리와 최소비용 신장 트리

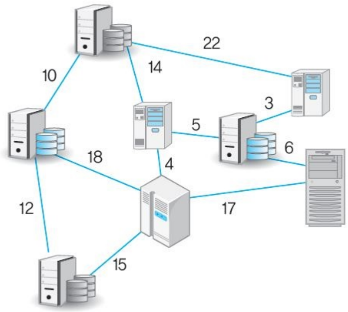
- 아래 그래프의 생성트리



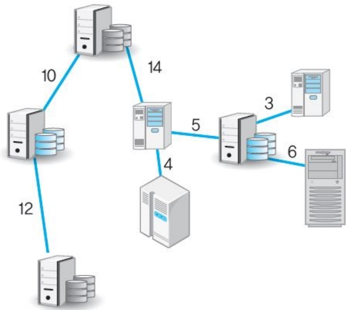
- 완전그래프의 가능한 생성트리의 수는 $n^{n-2} = 4^2 = 8$ 개



- **최소비용 신장트리 (Minimum cost Spanning Tree, MST)**: 그래프의 신장트리 중 최소의 **비용**을 가지는 신장트리



(a) 사이트 사이의 연결 비용

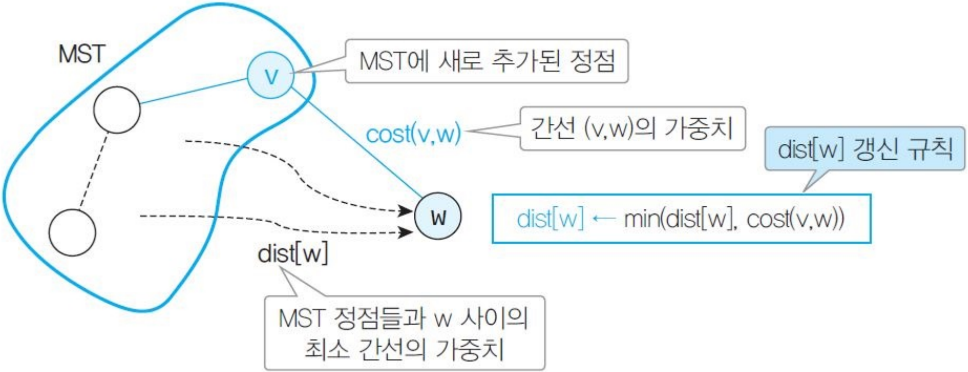


(b) 최소 비용의 통신망

최소비용 신장트리 (MST)

- 최소 비용 생성 트리의 대표적인 활용 사례
  1. 최적 경로 탐색: Dijkstra 의 알고리즘 또는 A* 알고리즘 등의 중간 단계에서 MST 를 이용
  2. 통신망, 전력망, 도로망 구축에서 네트워크 구축 비용을 최소화하기 위한 방법으로 활용
  3. 전기 회로 설계: 단자들을 모두 연결하면서 연결선의 길이 또는 전력소모가 최소가 되도록 하는 회로를 설계
- 알고리즘
  1. Prim 알고리즘: 임의의 꼭지점에서 시작, 최소 비용의 연결선을 탐색
  2. Kruskal 알고리즘: 모든 연결선 중에서 최소비용의 연결선을 탐색

- 하나의 정점에서부터 MST 를 단계적으로 확장하는 방법



MST 알고리즘: Prim

PRIM 알고리즘 노드의 수가 n 개인 가중그래프 ($G = (V, E)$) 에서

1. 노드의 집합: $U = \{1\}$
2. $u \in U, v \in V - U$ 일 때, U 와 $V - U$ 를 연결하는 가장 짧은 연결선 (u, v) 를 찾아 $U = U \cup \{u\}$, 이 때, (u, v) 는 사이클을 형성하지 않는 것이라야 함
3. $U = V$ 가 될 때까지 두번째 과정을 반복

MST 알고리즘: Kruskal

- 사이클을 만들지 않는 가장 가중치가 작은 간선 n-1 개를 순서대로 선택
- 사이클 검사: union-find 알고리즘
- 시간 복잡도: $O(e \log e)$
  - 희소 그래프에서는 Prim 보다 Kruskal 알고리즘이 유리

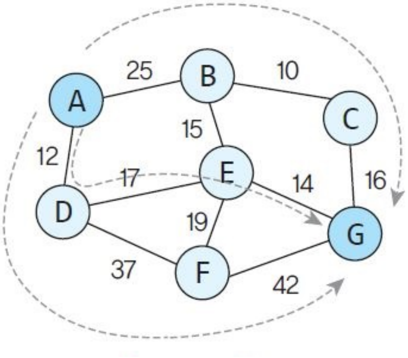
Kruskal's 알고리즘 노드의 수가 n 개인 가중그래프 ($G = (V, E)$) 에서

1. MST 에 포함할 연결선의 집합: $E' = \emptyset$.
2. 비용이 적은 순으로 그래프의 연결선 (E) 을 정렬
3. 비용이 최소인 연결선 $e_{\text{min}} \in E$ 를 찾아 E' 에 추가하여 업데이트 ($E' \cup e_{\text{min}}$), 단, e_{min} 를 추가하여 사이클이 만들어 지면 다음 최소 연결선을 찾음
4. E' 의 갯수가 $|V| - 1$ 가 될 때까지 세번째 과정을 반복

MST 알고리즘: Kruskal

최단 경로

- 정점 u 와 정점 v 를 연결하는 경로 중에서 가중치 합이 최소인 경로의 비용 구하는 문제
  - 간선의 가중치는 양수라 가정



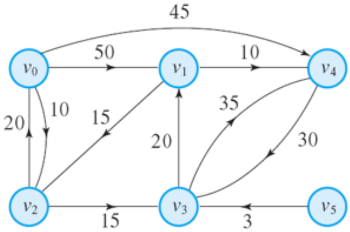
경로1: (A,B,C,G): 비용 = 25+10+16 = 51

경로2: (A,D,F,G): 비용 = 12+37+42 = 91

경로3: (A,D,E,G): 비용 = 12+17+14 = 43

- Dijkstra, Floyd 알고리즘

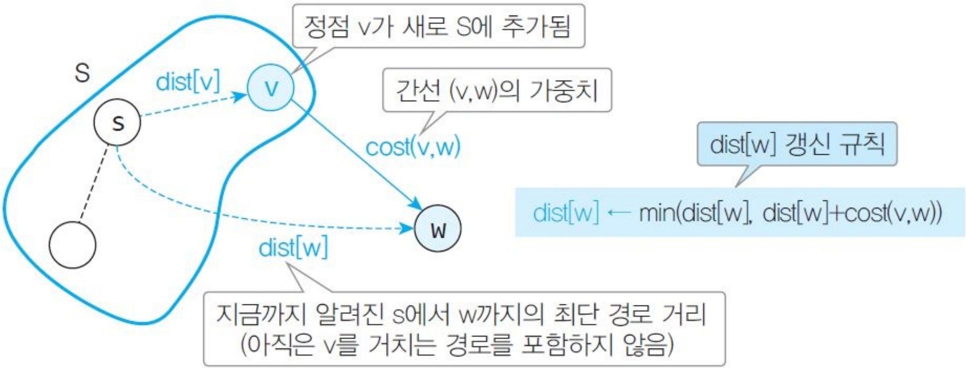
아래 방향그래프에서 v_0 에서 v_1 으로 가는 **최단 경로**를 찾아보자.



- 먼저 v_0 에서 v_1 으로 가는 가능한 **경로**는
  1. v_0, v_1
  2. v_0, v_2, v_3, v_1
  3. $v_0, v_2, v_3, v_4, v_3, v_1, \ldots$
- 각 경로별 거리를 계산하면
  1. v_0, v_1: 50
  2. v_0, v_2, v_3, v_1: 10 + 15 + 20 = 45
  3. $v_0, v_2, v_3, v_4, v_3, v_1, \ldots$
- 최단 거리는 45 이며 최단 경로는 v_0, v_2, v_3, v_1 임

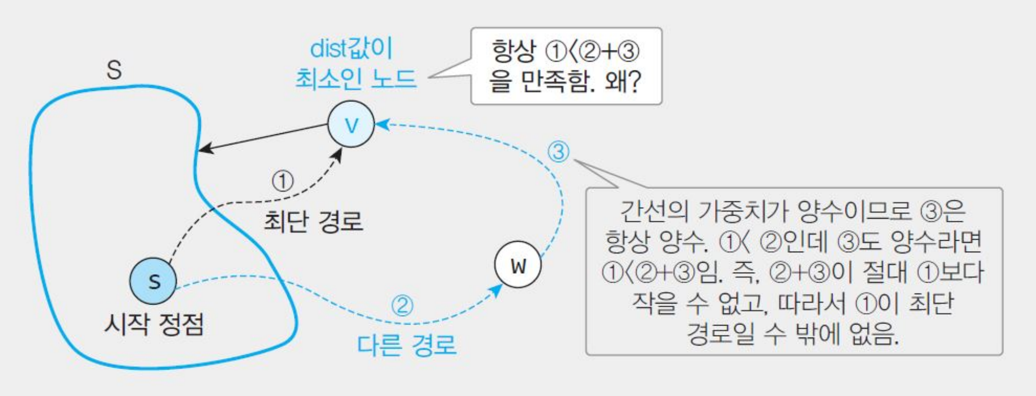
- 위의 예에서 처럼 모든 가능한 경우를 찾아 각 경우의 거리를 계산하여 최적을 탐색하는 방법을 **완전탐색 (Exhaustive Search)** 또는 **무대포탐색 (Brute-force search)** 알고리즘이라고 함

- 하나의 정점에서 다른 모든 정점까지의 최단 경로를 구하는 알고리즘
- 아이디어
  - S 에 포함되지 않은 정점 중에서 dist 가 최소인 정점 v 를 찾음
  - v 까지의 거리를 확정하고 S 에 추가
  - v 인접 정점의 거리 갱신



```
function Dijkstra(Graph, source):
    Q ← ∅
    visit = []
    for each v ∈ V:
        dist[v] ← ∞ # source(시작점) 에서 v 까지 최단 거리
        Q ← Q ∪ {v} # 방문할 노드들의 집합
    dist[source] ← 0
    while Q is not empty:
        u ← Q 의 노드 중 최단거리인 노드 (min dist(u))
        Q ← Q − {u}          # Q 에서 u 를 제거
        for each v ∈ N(u) ⊂ Q: # u 의 인접노드
            temp_dist(v) := dist(u) + w(u, v) # 시작노드에서 u 를경유한 v 까지의 거리
            if temp_dist(v) < dist(v):
                dist(v) ← temp_dist(v) # 최단거리 업데이트
        visit.append(u)
    return dist, visit
```

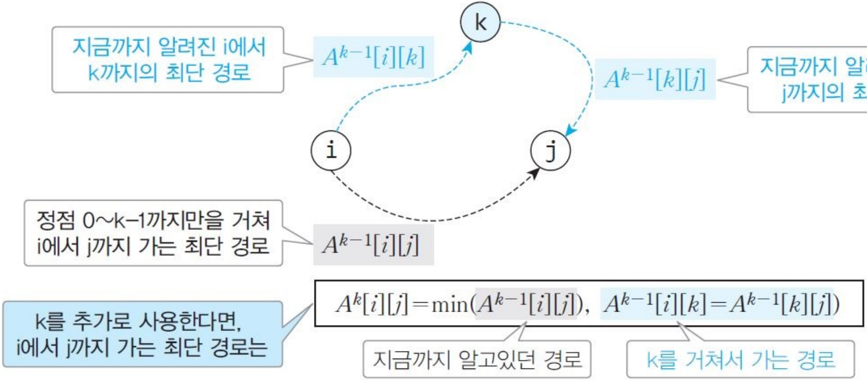
- Dijkstra 알고리즘은 타당할까?



최단 경로 - Floyd 알고리즘

- 모든 정점 사이의 최단 경로를 한꺼번에 찾음
  - 최단 경로 거리 행렬 A 이용
  - $A^k[i][j]$ 0 부터 k 까지의 정점만을 이용한 i~j 의 최단 경로 길이
  - $A^{-1} \rightarrow A^0 \rightarrow A^1 \rightarrow ... \rightarrow A^{n-1}$ 순으로 최단 경로 구함
  - A 의 초기값 (A^{-1}) 은 그래프의 인접 행렬
  - 삼중 루프로 구성: 시간복잡도 $= O(n^3)$

```
//Floyd 알고리즘
```



지금까지 알려진 i에서 k까지의 최단 경로
$A^{k-1}[i][k]$
k
$A^{k-1}[k][j]$
지금까지 알려진 k에서 j까지의 최단 경로
i
j
정점 0~k–1까지만을 거쳐 i에서 j까지 가는 최단 경로
$A^{k-1}[i][j]$
k를 추가로 사용한다면, i에서 j까지 가는 최단 경로는
$A^k[i][j] = \min(A^{k-1}[i][j],\ A^{k-1}[i][k] + A^{k-1}[k][j])$
지금까지 알고있던 경로
k를 거쳐서 가는 경로

연습문제