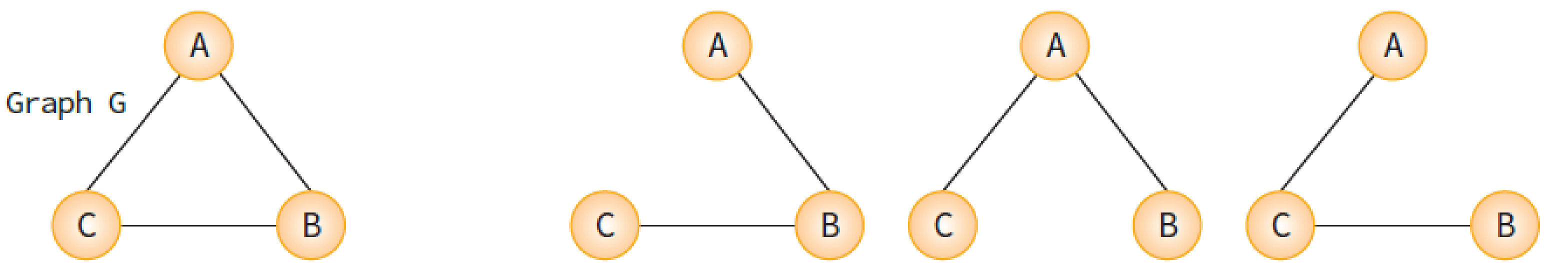
11장 그래프II

신장 트리(spanning tree)

- 그래프내의 모든 정점을 포함하는 트리
- n개의 정점을 가지는 그래프의 신장트리는 n-1개의 간선을 가짐



(a) 연결 그래프

(b) 신장 트리

© 생능출판사 2019

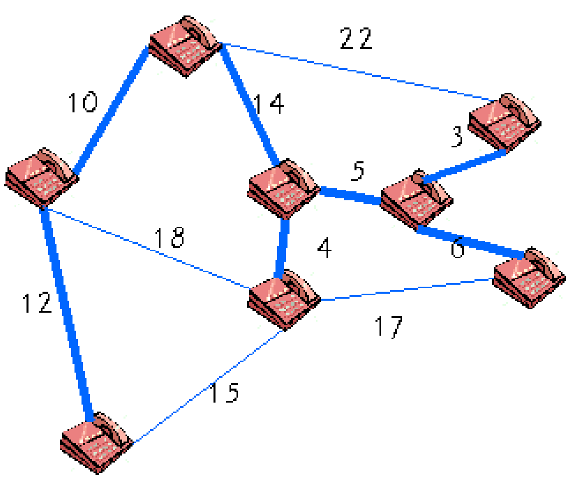
신장 트리 알고리즘

```
depth_first_search(v):
        v를 방문되었다고 표시;
        for all u ∈ (v에 인접한 정점) do
                if (u가 아직 방문되지 않았으면)
                        then    (v,u)를 신장 트리 간선이라고 표시;
                                depth_first_search(u)
```

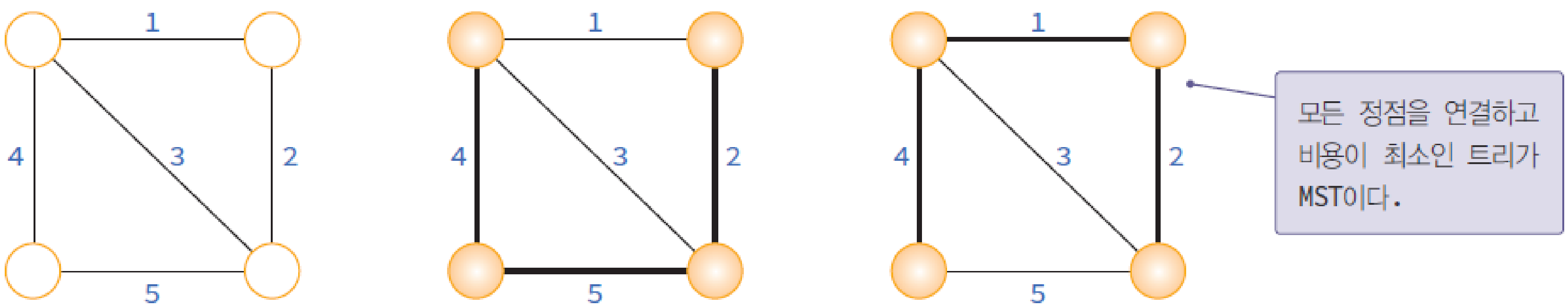
© 생능출판사 2019

최소비용 신장트리 (minimum spanning tree)

- 네트워크에 있는 모든 정점들을 가장 적은 수의 간선과 비용으로 연결
- MST의 응용
  - 도로 건설 - 도시들을 모두 연결하면서 도로의 길이를 최소가 되도록 하는 문제
  - 전기 회로 - 단자들을 모두 연결하면서 전선의 길이를 가장 최소로 하는 문제
  - 통신 - 전화선의 길이가 최소가 되도록 전화 케이블 망을 구성하는 문제
  - 배관 - 파이프를 모두 연결하면서 파이프의 총 길이를 최소로 하는 문제



MST의 예



그래프

신장 트리
전체 비용=4+5+2=11

최소 비용 신장 트리
전체 비용=4+1+2=7

© 생능출판사 2019

Kruskal의 MST 알고리즘

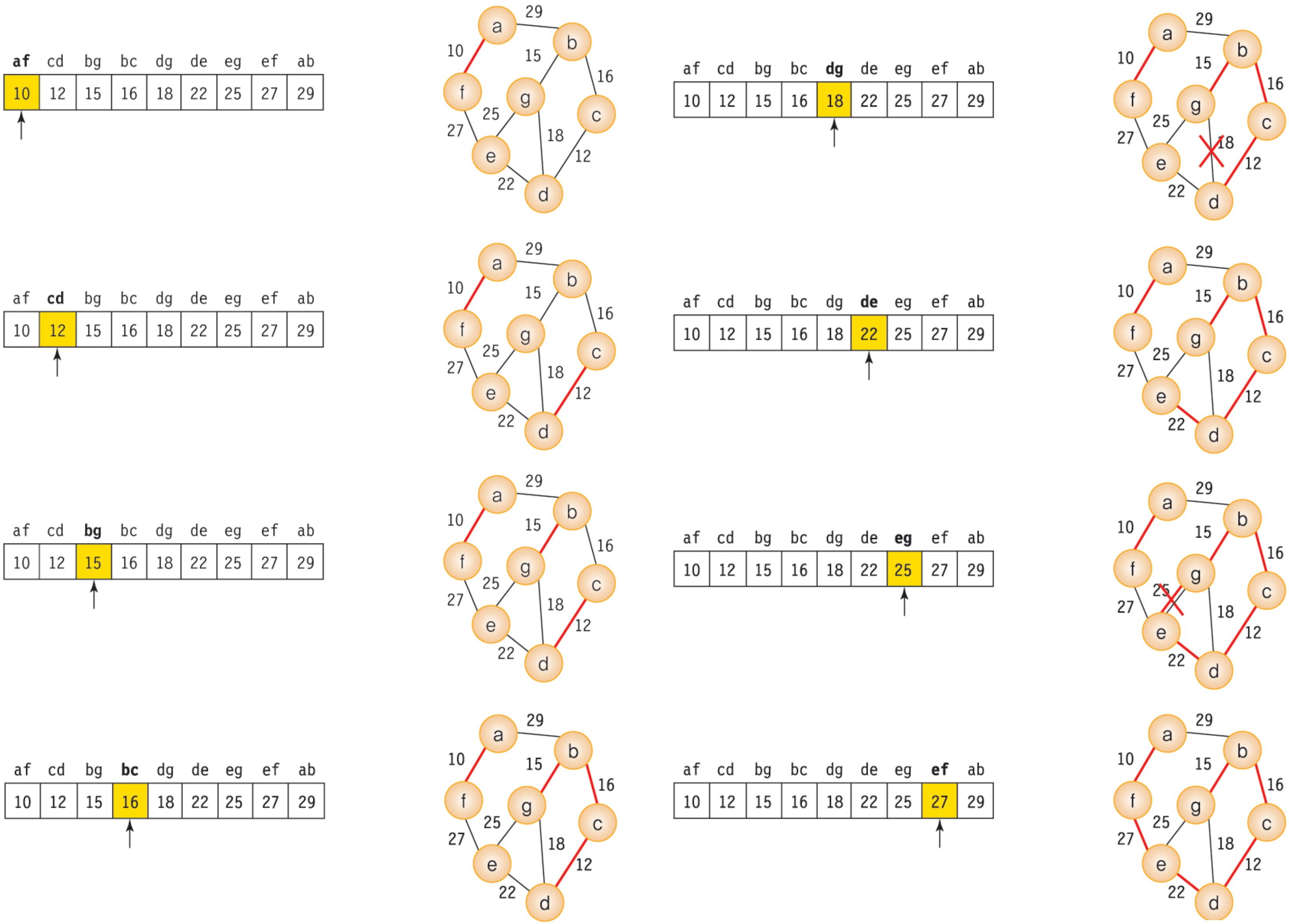
- 탐욕적인 방법(greedy method)
  - 주요 알고리즘 설계 기법
  - 각 단계에서 최선의 답을 선택하는 과정을 반복함으로써 최종적인 해답에 도달
  - 탐욕적인 방법은 항상 최적의 해답을 주는지 검증 필요
  - Kruskal MST 알고리즘은 최적의 해답 임이 증명됨

© 생능출판사 2019

Kruskal의 MST 알고리즘

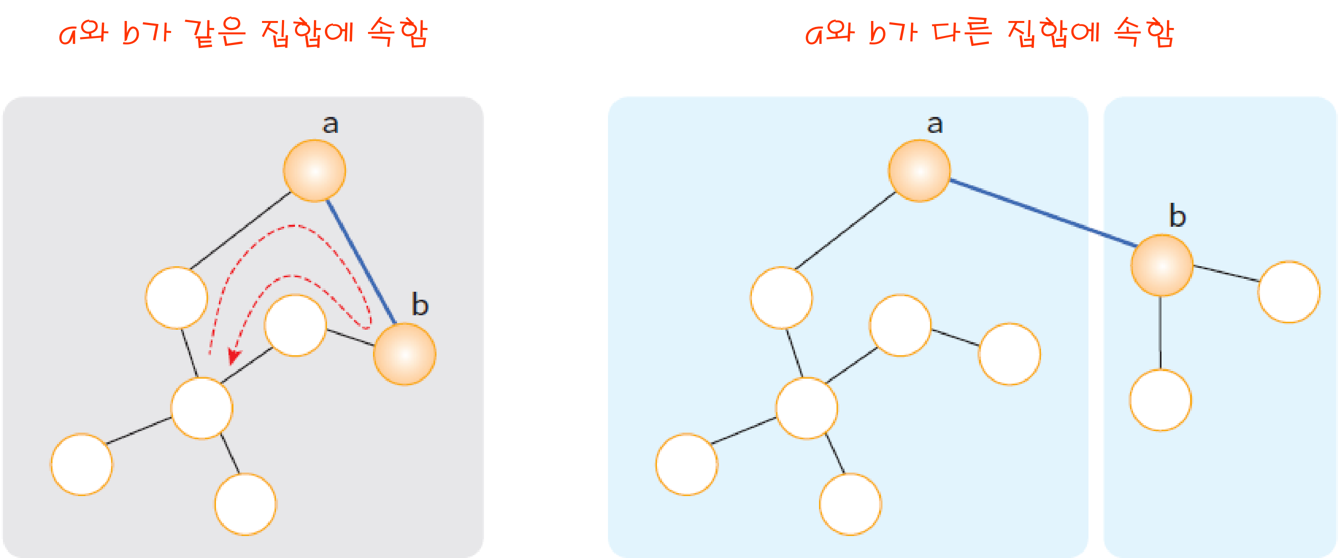
```
// 최소비용 스패닝트리를 구하는 Kruskal의 알고리즘
// 입력: 가중치 그래프 G=(V, E), n은 노드의 개수
// 출력: E_T, 최소비용 신장 트리를 이루는 간선들의 집합
kruskal(G)

    E를 w(e_1) ≤ ··· ≤ w(e_e) 가 되도록 정렬한다.
    E_T ← Φ; ecounter ← 0
    k ← 0
    while ecounter <(n − 1)  do
        k ← k + 1
        if E_T ∪ {e_k} 가 사이클을 포함하지 않으면
            then E_T ← E_T ∪ {e_k};  ecounter ← ecounter + 1
    return E_T
```

© 생능출판사 2019


af cd bg bc dg de eg ef ab
10 12 15 16 18 22 25 27 29
a b c d e f g
29 10 15 16 25 27 18 12 22

Kruskal의 MST 알고리즘

- union-find 알고리즘
  - 원소가 어떤 집합에 속하는지 알아냄
  - Kruskal의 MST 알고리즘에서 사이클 검사에 사용

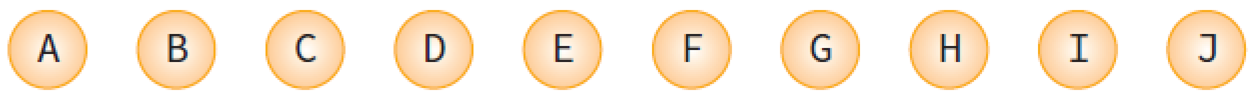


(a) 사이클 형성

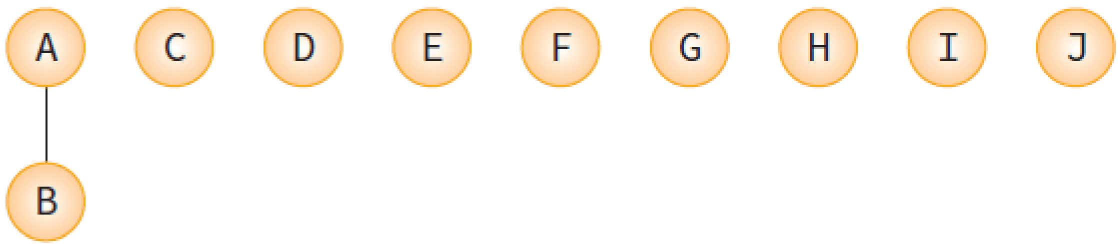
(b) 사이클 형성되지 않음

© 생능출판사 2019

Kruskal의 MST 알고리즘



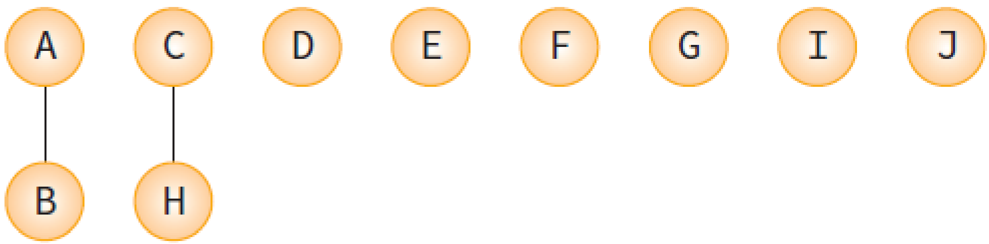
A	B	C	D	E	F	G	H	I	J
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1



A	B	C	D	E	F	G	H	I	J
-1	0	-1	-1	-1	-1	-1	-1	-1	-1

© 생능출판사 2019

Kruskal의 MST 알고리즘



A	B	C	D	E	F	G	H	I	J
-1	0	-1	-1	-1	-1	-1	2	-1	-1

© 생능출판사 2019

union-find 알고리즘

```
UNION(a, b):
   root1 = FIND(a);       // 노드 a의 루트를 찾는다.
   root2 = FIND(b);       // 노드 b의 루트를 찾는다.
   if root1 ≠ root2       // 합한다.
      parent[root1] = root2;

FIND(curr):               // curr의 루트를 찾는다.
   if (parent[curr] == -1)
      return curr;                          // 루트
   while (parent[curr] != -1) curr = parent[curr];
   return curr;
```

© 생능출판사 2019

union-find 프로그램

```
int parent[MAX_VERTICES];                    // 부모 노드
                                                                          // 초기화
void set_init(int n)
{
        for (int i = 0; i<n; i++)
                parent[i] = -1;
}
// curr가 속하는 집합을 반환한다.
int set_find(int curr)
{
        if (parent[curr] == -1)
                return curr;                                     // 루트
        while (parent[curr] != -1) curr = parent[curr];
        return curr;
}
```

© 생능출판사 2019

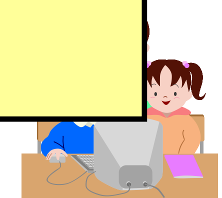
union-find 프로그램

```
// 두개의 원소가 속한 집합을 합친다.
void set_union(int a, int b)
{
		int root1 = set_find(a);			// 노드 a의 루트를 찾는다.
		int root2 = set_find(b);			// 노드 b의 루트를 찾는다.
		if (root1 != root2)		// 합한다.
				parent[root1] = root2;
}
```

© 생능출판사 2019

Kruskal의 MST 프로그램

```
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

#define MAX_VERTICES 100
#define INF 1000

int parent[MAX_VERTICES];                    // 부모 노드
                                                                        // 초기화
void set_init(int n)
{
        for (int i = 0; i<n; i++)
                parent[i] = -1;
}
// curr가 속하는 집합을 반환한다.
int set_find(int curr)
{
        if (parent[curr] == -1)
                return curr;                              // 루트
        while (parent[curr] != -1) curr = parent[curr];
        return curr;
}
```



© 생능출판사 2019

Kruskal의 MST 프로그램

```
// 두개의 원소가 속한 집합을 합친다.
void set_union(int a, int b)
{
        int root1 = set_find(a);        // 노드 a의 루트를 찾는다.
        int root2 = set_find(b);        // 노드 b의 루트를 찾는다.
        if (root1 != root2)     // 합한다.
                parent[root1] = root2;
}

struct Edge {                           // 간선을 나타내는 구조체
        int start, end, weight;
};

typedef struct GraphType {
        int n;      // 간선의 개수
        struct Edge edges[2 * MAX_VERTICES];
} GraphType;
```

© 생능출판사 2019

Kruskal의 MST 프로그램

```
// 그래프 초기화
void graph_init(GraphType* g)
{
		g->n = 0;
		for (int i = 0; i < 2 * MAX_VERTICES; i++) {
				g->edges[i].start = 0;
				g->edges[i].end = 0;
				g->edges[i].weight = INF;
		}
}
// 간선 삽입 연산
void insert_edge(GraphType* g, int start, int end, int w)
{
		g->edges[g->n].start = start;
		g->edges[g->n].end = end;
		g->edges[g->n].weight = w;
		g->n++;
}
// qsort()에 사용되는 함수
int compare(const void* a, const void* b)
{
		struct Edge* x = (struct Edge*)a;
		struct Edge* y = (struct Edge*)b;
		return (x->weight - y->weight);
}
```

Kruskal의 MST 프로그램

```
// kruskal의 최소 비용 신장 트리 프로그램
void kruskal(GraphType *g)
{
        int edge_accepted = 0;          // 현재까지 선택된 간선의 수
        int uset, vset;                         // 정점 u와 정점 v의 집합 번호
        struct Edge e;

        set_init(g->n);                                 // 집합 초기화
        qsort(g->edges, g->n, sizeof(struct Edge), compare);

        printf("크루스칼 최소 신장 트리 알고리즘 \n");
        int i = 0;
        while (edge_accepted < (g->n - 1))      // 간선의 수 < (n-1)
        {
                e = g->edges[i];
                uset = set_find(e.start);               // 정점 u의 집합 번호
                vset = set_find(e.end);         // 정점 v의 집합 번호
                if (uset != vset) {                     // 서로 속한 집합이 다르면
                        printf("간선 (%d,%d) %d 선택\n", e.start, e.end, e.weight);
                        edge_accepted++;
                        set_union(uset, vset);  // 두개의 집합을 합친다.
                }
                i++;
        }
}
```

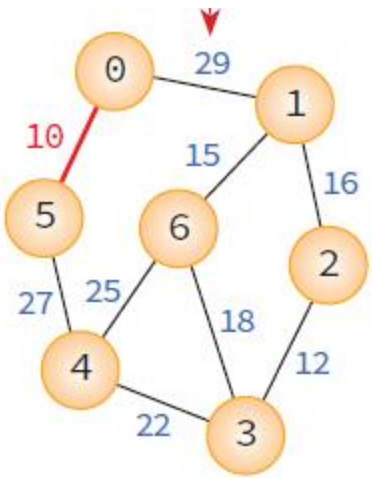
Kruskal의 MST 프로그램

```
int main(void)
{
		GraphType *g;
		g = (GraphType *)malloc(sizeof(GraphType));
		graph_init(g);

		insert_edge(g, 0, 1, 29);
		insert_edge(g, 1, 2, 16);
		insert_edge(g, 2, 3, 12);
		insert_edge(g, 3, 4, 22);
		insert_edge(g, 4, 5, 27);
		insert_edge(g, 5, 0, 10);
		insert_edge(g, 6, 1, 15);
		insert_edge(g, 6, 3, 18);
		insert_edge(g, 6, 4, 25);

		kruskal(g);
		free(g);
		return 0;
}
```

© 생능출판사 2019

실행 결과

```
크루스칼 최소 신장 트리 알고리즘
간선 (5,0) 10 선택
간선 (2,3) 12 선택
간선 (6,1) 15 선택
간선 (1,2) 16 선택
간선 (3,4) 22 선택
간선 (4,5) 27 선택
```



© 생능출판사 2019

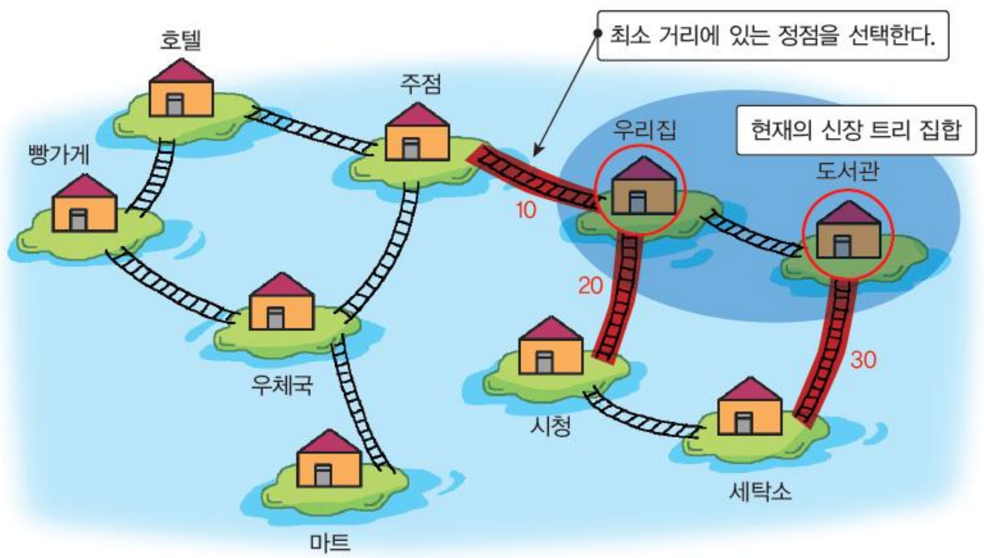
Kruskal의 MST 알고리즘 복잡도

- Kruskal 알고리즘은 대부분 간선들을 정렬하는 시간에 좌우됨
  - 사이클 테스트 등의 작업은 정렬에 비해 매우 신속하게 수행됨

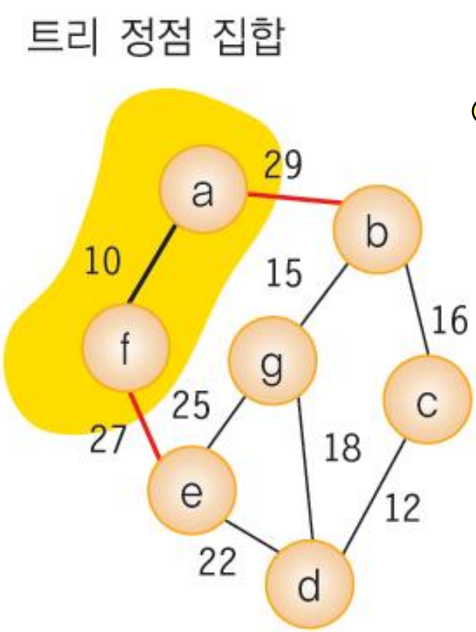
- 네트워크의 간선 *e개를 퀵정렬과 같은 효율적인 알고리즘으로 정렬* 한다면 Kruskal 알고리즘의 시간 복잡도는 *O(e*log(e))가 된다*

Prim의 MST 알고리즘

- 시작 정점에서부터 출발하여 신장 트리 집합을 단계적으로 확장해나감
- 신장 트리 집합에 인접한 정점 중에서 최저 간선으로 연결된 정점 선택하여 신장 트리 집합에 추가함



Prim의 MST 알고리즘

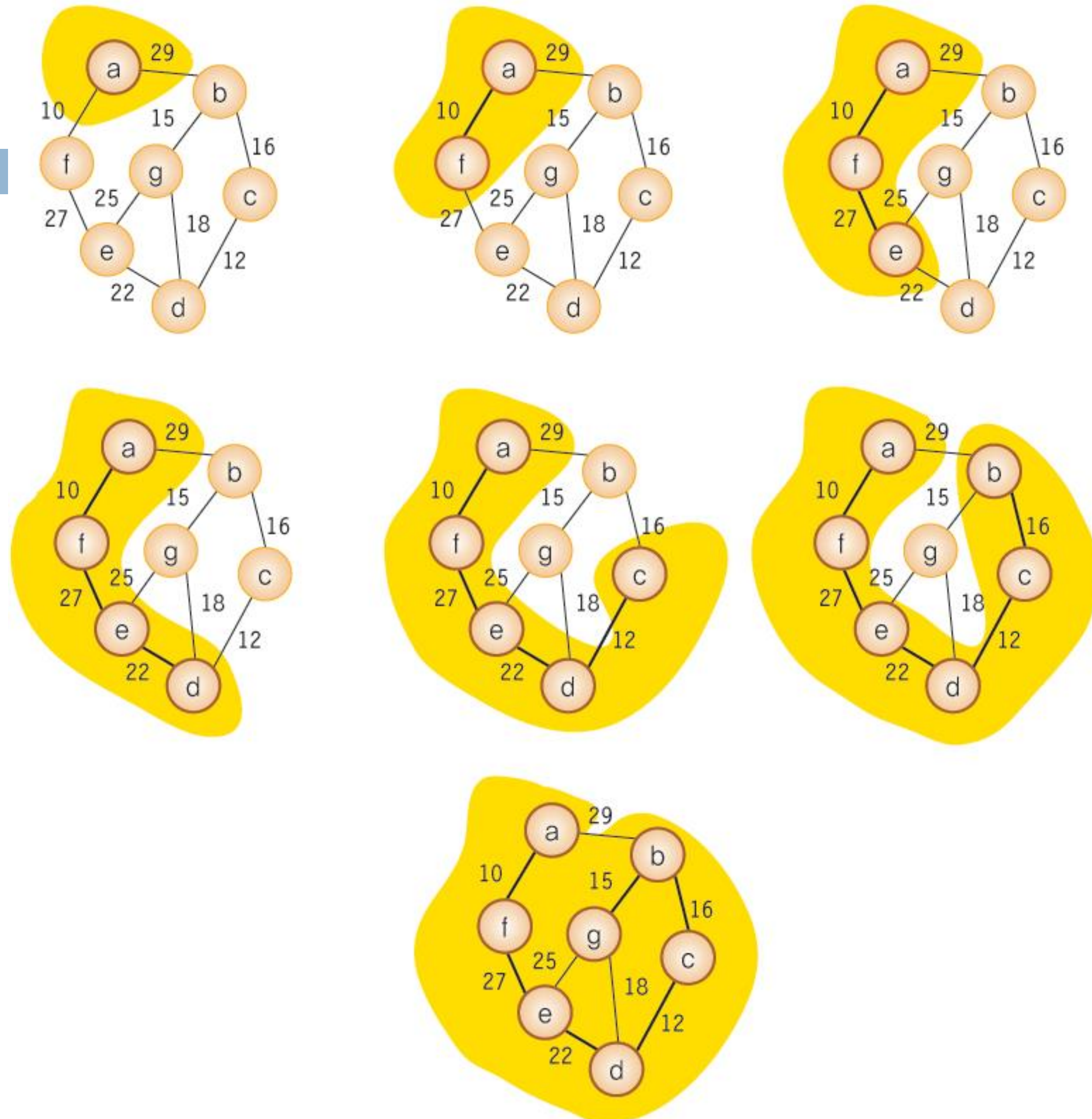


간선 (a, b)=29
간선 (f, e)=27

간선 (f, e) 선택

정점 e가 신장 트리 집합에 추가됨

© 생능출판사 2019


a
b
c
d
e
f
g
29
10
15
16
25
27
18
12
22

Prim의 MST 알고리즘

```
// 최소 비용 신장 트리를 구하는 Prim의 알고리즘
// 입력: 네트워크 G=(V, E), s는 시작 정점
// 출력: 최소 비용 신장 트리를 이루는 정점들의 집합
Prim(G, s)

    for each u∈V do
        dist[u] ← ∞
    dist[s] ← 0
    우선 순위큐 Q에 모든 정점을 삽입(우선순위는 dist[])
    for i ← 0 to n-1 do
        u ← delete_min(Q)
        화면에 u를 출력
        for each v∈(u의 인접 정점)
            if( v∈Q and weight[u][v]<dist[v] )
                then dist[v] ← weight[u][v]
```

© 생능출판사 2019

Prim의 MST 프로그램

```c
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0
#define MAX_VERTICES 100
#define INF 1000L

typedef struct GraphType {
		int n;		// 정점의 개수
		int weight[MAX_VERTICES][MAX_VERTICES];
} GraphType;

int selected[MAX_VERTICES];
int distance[MAX_VERTICES];
```


© 생능출판사 2019

Prim의 MST 프로그램

```
// 최소 dist[v] 값을 갖는 정점을 반환
int get_min_vertex(int n)
{
		int v, i;
		for (i = 0; i <n; i++)
			if (!selected[i]) {
				v = i;
				break;
			}
		for (i = 0; i < n; i++)
			if (!selected[i] && (distance[i] < distance[v])) v = i;
		return (v);
}
```

© 생능출판사 2019

Prim의 MST 프로그램

```
//
void prim(GraphType* g, int s)
{
		int i, u, v;

		for (u = 0; u<g->n; u++)
				distance[u] = INF;
		distance[s] = 0;
		for (i = 0; i<g->n; i++) {
				u = get_min_vertex(g->n);
				selected[u] = TRUE;
				if (distance[u] == INF) return;
				printf("정점 %d 추가\n", u);
				for (v = 0; v<g->n; v++)
						if (g->weight[u][v] != INF)
								if (!selected[v] && g->weight[u][v]< distance[v])
										distance[v] = g->weight[u][v];
		}
}
```

© 생능출판사 2019

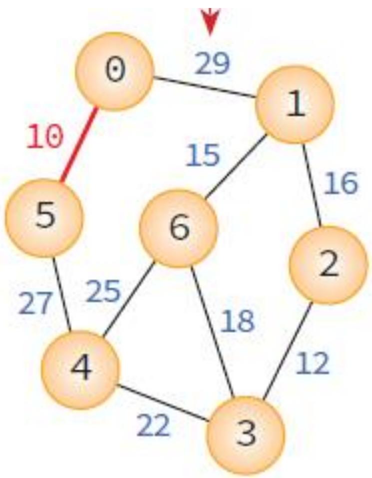
Prim의 MST 프로그램

```
int main(void)
{
        GraphType g = { 7,
        {{ 0, 29, INF, INF, INF, 10, INF },
        { 29,  0, 16, INF, INF, INF, 15 },
        { INF, 16, 0, 12, INF, INF, INF },
        { INF, INF, 12, 0, 22, INF, 18 },
        { INF, INF, INF, 22, 0, 27, 25 },
        { 10, INF, INF, INF, 27, 0, INF },
        { INF, 15, INF, 18, 25, INF, 0 } }
        };
        prim(&g, 0);
        return 0;
}
```


© 생능출판사 2019

실행 결과

```
정점 0 추가
정점 5 추가
정점 4 추가
정점 3 추가
정점 2 추가
정점 1 추가
정점 6 추가
```



© 생능출판사 2019

Prim의 MST 알고리즘 복잡도

- 주 반복문이 정점의 수 *n만큼 반복하고, 내부 반복문이 n번 반복하므*로 Prim의 알고리즘은 $O(n^2)$ *의 복잡도를 가진다.*
- 희박한 그래프
  - *O(e*log(e))* 인 Kruskal의 알고리즘이 유리
- 밀집한 그래프
  - $O(n^2)$ 인 Prim의 알고리즘이 유리

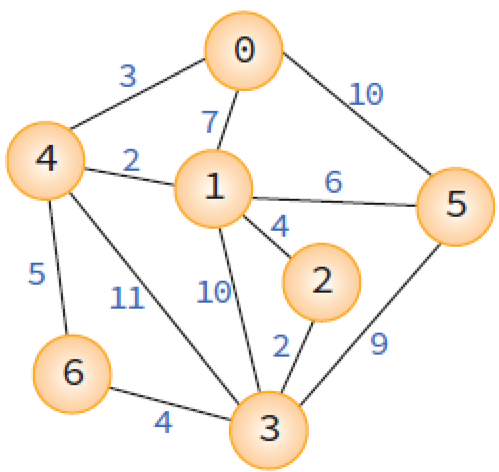
© 생능출판사 2019

최단 경로(shortest path)

- 네트워크에서 정점 u와 정점 v를 연결하는 경로 중에서 간선들의 가중치 합이 최소가 되는 경로
- 간선의 가중치는 비용, 거리, 시간 등

© 생능출판사 2019

가중치 인접행렬



	0	1	2	3	4	4	5
0	0	7	∞	∞	3	10	∞
1	7	0	4	10	2	6	∞
2	∞	4	0	2	∞	∞	∞
3	∞	10	2	0	11	9	4
4	3	2	∞	11	0	∞	5
5	10	6	∞	9	∞	0	∞
6	∞	∞	∞	4	5	∞	0

© 생능출판사 2019

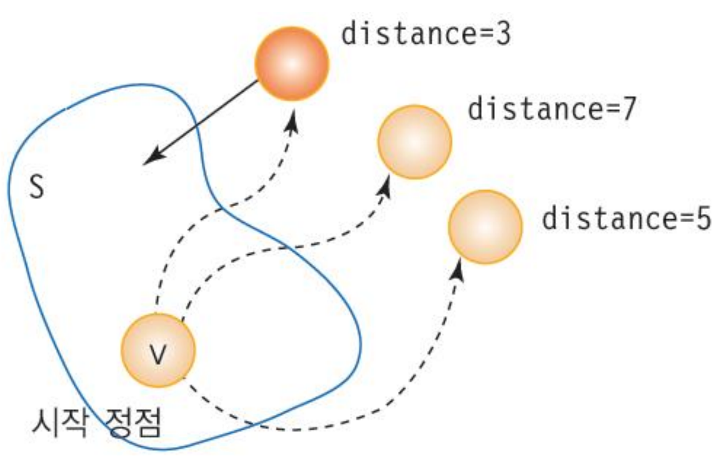
최단경로 알고리즘

- Dijkstra 알고리즘: 하나의 시작 정점에서 다른 정점까지의 최단경로 계산
- Floyd 알고리즘은 모든 정점에서 다른 모든 정점까지의 최단 경로를 계산

© 생능출판사 2019

Dijkstra의 최단경로 알고리즘

- 하나의 시작 정점으로부터 모든 다른 정점까지의 최단 경로 찾음
- 집합 S: 시작 정점 v로부터의 최단경로가 이미 발견된 정점들의 집합
- distance 배열: 최단경로가 알려진 정점들만을 이용한 다른 정점들까지의 최단경로 길이
- 매 단계에서 가장 distance 값이 작은 정점을 S에 추가

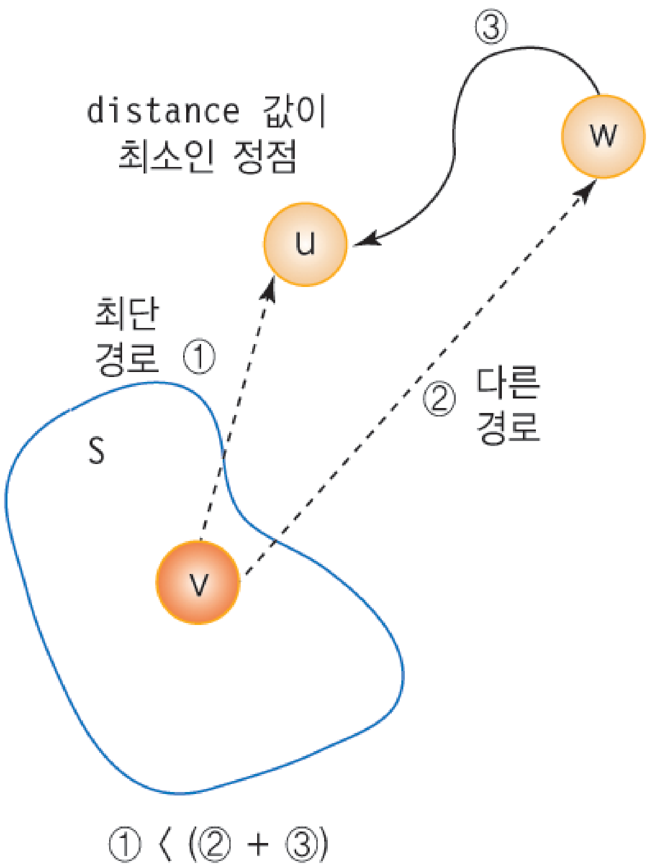


© 생능출판사 2019

Dijkstra의 최단경로 알고리즘

- 각 단계에서 S안에 있지 않은 정점 중에서 가장 distance값이 작은 정점을 S에 추가한다.
- 정점 w를 거쳐서 정점 u로 가는 가상적인 더 짧은 경로가 있다고 가정해보자,

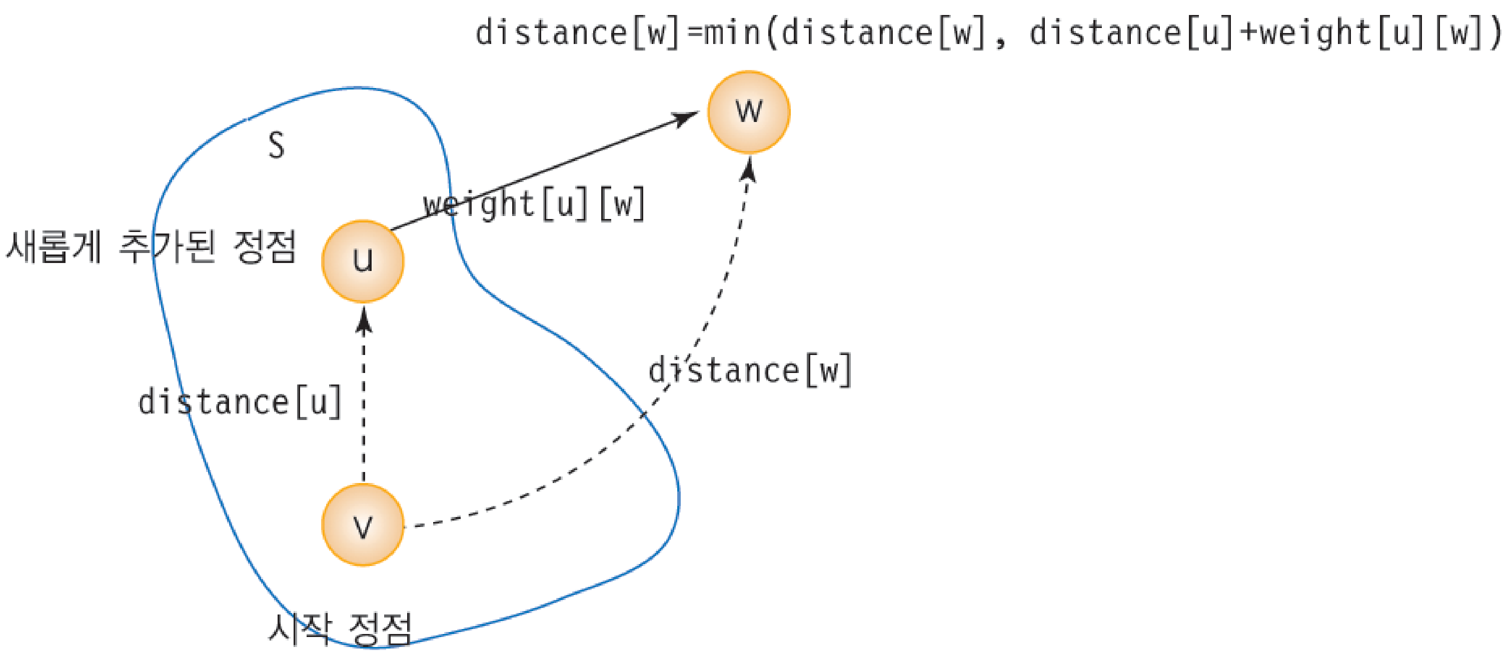
  그러면 정점 v에서 정점 u까지의 거리는 정점 v에서 정점 w까지의 거리 ②와 정점 w에서 정점 u로 가는 거리③을 합한 값이 된다.
- 그러나 경로 ②는 경로 ①보다 항상 길 수 밖에 없다. 왜냐하면 현재 distance 값이 가장 작은 정점은 u이기 때문이다.



© 생능출판사 2019

Dijkstra의 최단경로 알고리즘

- 새로운 정점이 S에 추가되면 distance값 갱신



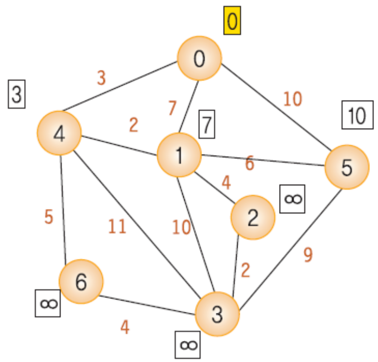
© 생능출판사 2019

Dijkstra의 최단경로 알고리즘

```
// 입력: 가중치 그래프 G, 가중치는 음수가 아님.
// 출력: distance 배열, distance[u]는 v에서 u까지의 최단 거리이다.
shortest_path(G, v)

S←{v}
for 각 정점 w∈G do
        distance[w]←weight[v][w];
while 모든 정점이 S에 포함되지 않으면 do
        u←집합 S에 속하지 않는 정점 중에서 최소 distance 정점;
        S←S∪{u}
        for u에 인접하고 S에 있는 각 정점 z do
                if distance[u]+weight[u][z] < distance[z]
                        then distance[z]←distance[u]+weight[u][z];
```

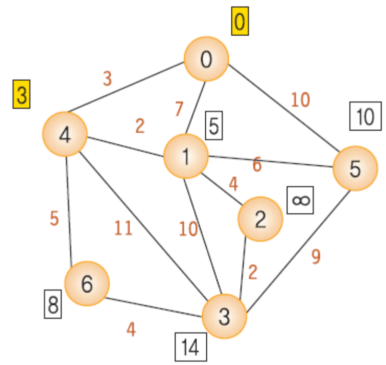
© 생능출판사 2019

Dijkstra의 최단경로 알고리즘



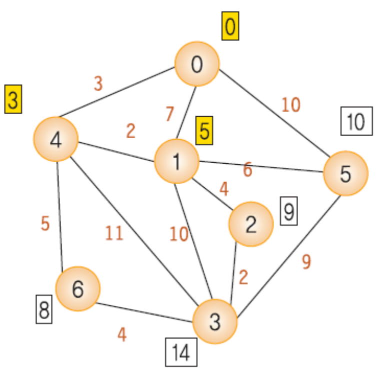
S={0}

	0	1	2	3	4	5	6
distance[]=	0	7	∞	∞	3	10	∞



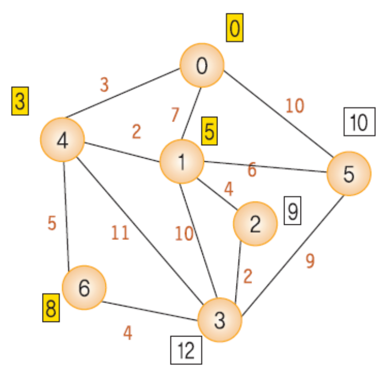
S={0, 4}

	0	1	2	3	4	5	6
distance[]=	0	5	∞	14	3	10	8



S={0, 4, 1}

	0	1	2	3	4	5	6
distance[]=	0	5	9	14	3	10	8

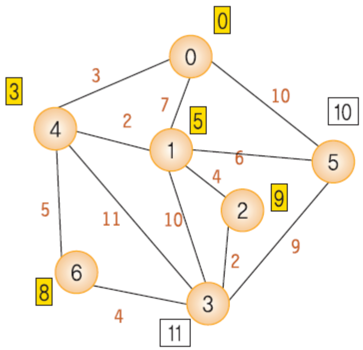


S={0, 4, 1, 6}

	0	1	2	3	4	5	6
distance[]=	0	5	9	12	3	10	8

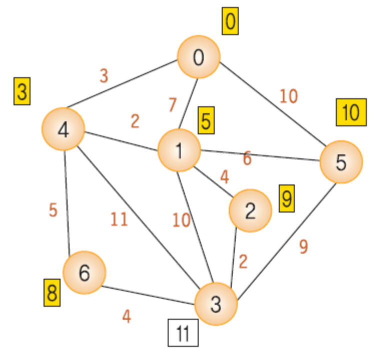
© 생능출판사 2019

Dijkstra의 최단경로 알고리즘



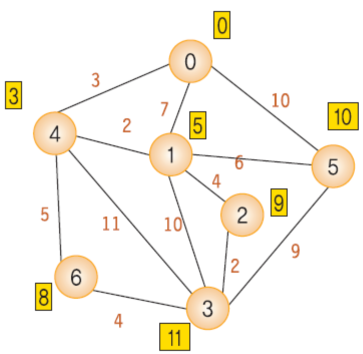
S={0, 4, 1, 6, 2}

	0	1	2	3	4	5	6
distance[]=	0	5	9	11	3	10	8



S={0, 4, 1, 6, 2, 5}

	0	1	2	3	4	5	6
distance[]=	0	5	9	11	3	10	8



S={0, 4, 1, 6, 2, 5, 3}

	0	1	2	3	4	5	6
distance[]=	0	5	9	11	3	10	8

© 생능출판사 2019

Dijkstra의 최단경로 프로그램

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define TRUE 1
#define FALSE 0
#define MAX_VERTICES 100
#define INF 1000000     /* 무한대 (연결이 없는 경우) */

typedef struct GraphType {
        int n;      // 정점의 개수
        int weight[MAX_VERTICES][MAX_VERTICES];
} GraphType;

int distance[MAX_VERTICES];/* 시작정점으로부터의 최단경로 거리 */
int found[MAX_VERTICES];              /* 방문한 정점 표시 */
```

© 생능출판사 2019

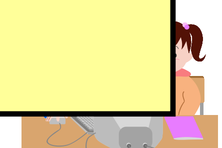
Dijkstra의 최단경로 프로그램

```
int choose(int distance[], int n, int found[])
{
		int i, min, minpos;
		min = INT_MAX;
		minpos = -1;
		for (i = 0; i<n; i++)
			if (distance[i]< min && !found[i]) {
				min = distance[i];
				minpos = i;
			}
		return minpos;
}
```

© 생능출판사 2019

Dijkstra의 최단경로 프로그램

```
//
void shortest_path(GraphType* g, int start)
{
		int i, u, w;
		for (i = 0; i<g->n; i++) /* 초기화 */
		{
				distance[i] = g->weight[start][i];
				found[i] = FALSE;
		}
		found[start] = TRUE;    /* 시작 정점 방문 표시 */
		distance[start] = 0;
		for (i = 0; i<g->n-1; i++) {
				print_status(g);
				u = choose(distance, g->n, found);
				found[u] = TRUE;
				for (w = 0; w<g->n; w++)
						if (!found[w])
								if (distance[u] + g->weight[u][w]<distance[w])
										distance[w] = distance[u] + g-
>weight[u][w];
		}
}
```



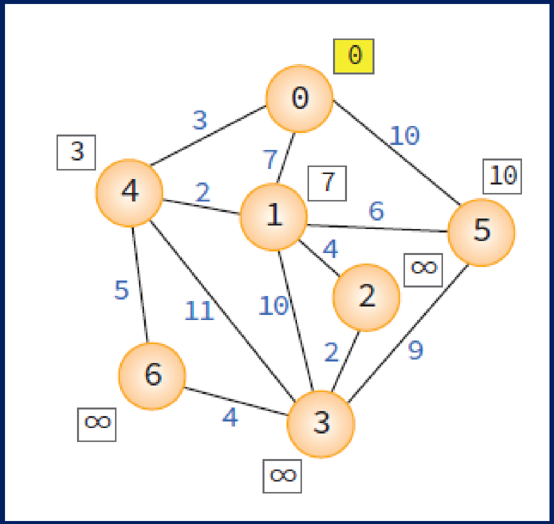
Dijkstra의 최단경로 프로그램

```
int main(void)
{
		GraphType g = { 7,
		{{ 0,  7,  INF, INF,   3,  10, INF },
		{ 7,  0,    4,  10,    2,    6, INF },
		{ INF,  4,     0,    2, INF, INF, INF },
		{ INF, 10,     2,     0,   11,    9,    4 },
		{ 3,  2,   INF,  11,    0, INF,    5 },
		{ 10,  6,   INF,    9, INF,    0, INF },
		{ INF, INF, INF,    4,    5, INF,    0 } }
		};
		shortest_path(&g, 0);
		return 0;
}
```


© 생능출판사 2019

실행 결과

```
STEP 1: distance:  0  7  *  *  3 10  *
        found:     1  0  0  0  0  0  0

STEP 2: distance:  0  5  * 14  3 10  8
        found:     1  0  0  0  1  0  0

STEP 3: distance:  0  5  9 14  3 10  8
        found:     1  1  0  0  1  0  0

STEP 4: distance:  0  5  9 12  3 10  8
        found:     1  1  0  0  1  0  1

STEP 5: distance:  0  5  9 11  3 10  8
        found:     1  1  1  0  1  0  1

STEP 6: distance:  0  5  9 11  3 10  8
        found:     1  1  1  0  1  1  1
```



Dijkstra의 최단경로 알고리즘 복잡도

- 네트워크에 n개의 정점이 있다면, Dijkstra의 최단경로 알고리즘은 주반복문을 n번 반복하고 내부 반복문을 2n번 반복하므로 $O(n^2)$의 복잡도를가진다.

© 생능출판사 2019

Floyd 알고리즘

```
floyd(G):

for k ← 0 to n - 1
    for i ← 0 to  n - 1
       for j ← 0 to n - 1
           A[i][j] = min(A[i][j], A[i][k] + A[k][j])
```

© 생능출판사 2019

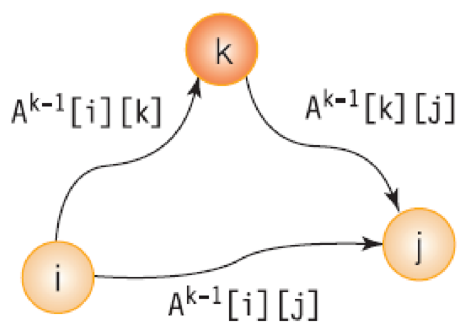
Floyd의 최단경로 알고리즘

- 모든 정점 사이의 최단경로를 찾음
- 2차원 배열 A를 이용하여 3중 반복을 하는 루프로 구성

© 생능출판사 2019

Floyd의 최단경로 알고리즘

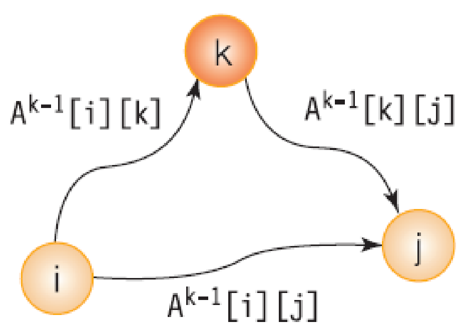
- A^k [i][j]
  - 0부터 k까지의 정점만을 이용한 정점 i에서 j까지의 최단 경로 길이
- $A^{-1} \rightarrow A^0 \rightarrow A^1 \rightarrow \ldots \rightarrow A^{n-1}$순으로 최단 경로 구해감
- A^{k-1}까지 구해진 상태에서 **k**번째 정점이 추가로 고려되는 상황을 생각하자



© 생능출판사 2019

Floyd의 최단경로 알고리즘

- 0부터 k까지의 정점만을 사용하여 정점 i에서 정점 j로 가는 최단 경로는 다음의 2가지의 경우로 나뉘어진다.
- 정점 k를 거치지 않는 경우:
  - $A^k[i][j]$ 는 k보다 큰 정점은 통과하지 않으므로 최단거리는 여전히 $A^{k-1}[i][j]$임
- 정점 k를 거치는 경우:
  - i에서 k까지의 최단거리 $A^{k-1}[i][k]$에 k에서 j까지의 최단거리 $A^{k-1}[k][j]$를 더한 값

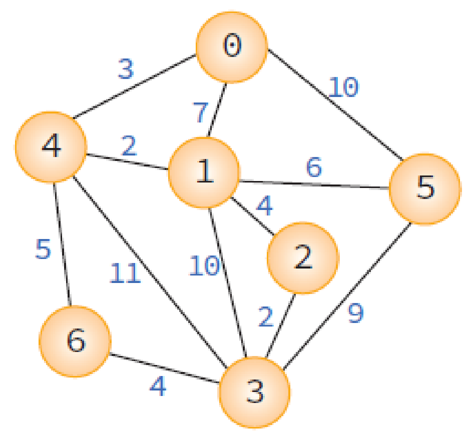


© 생능출판사 2019

Floyd의 최단경로 알고리즘

(1) 그래프의 가중치 행렬로 배열 A를 초기화한다.

```
===============================
    0    7    *    *    3   10    *
    7    0    4   10    2    6    *
    *    4    0    2    *    *    *
    *   10    2    0   11    9    4
    3    2    *   11    0    *    5
   10    6    *    9    *    0    *
    *    *    *    4    5    *    0
===============================
```



(2) 정점 0을 거쳐서 가는 경로와 비교하여 최단 경로를 수정한다.

```
===============================
    0    7    *    *    3   10    *
    7    0    4   10    2    6    *
    *    4    0    2    *    *    *
    *   10    2    0   11    9    4
    3    2    *   11    0   13    5
   10    6    *    9   13    0    *
    *    *    *    4    5    *    0
===============================
```

$$A^0[i][j] = \min(A^{-1}[i][j], A^{-1}[i][0]+A^{-1}[0][j])$$

© 생능출판사 2019

Floyd의 최단경로 알고리즘

(3) 정점 1을 거쳐서 가는 경로와 비교하여 최단 경로를 수정한다.

```
=================================
    0    7   11   17    3   10    *
    7    0    4   10    2    6    *
   11    4    0    2    6   10    *
   17   10    2    0   11    9    4
    3    2    6   11    0    8    5
   10    6   10    9    8    0    *
    *    *    *    4    5    *    0
=================================
```

$$A^1[i][j] = \min(A^0[i][j],\ A^0[i][1]+A^0[1][j])$$

© 생능출판사 2019

Floyd의 최단경로 프로그램

```
void floyd(GraphType* g)
{

        int i, j, k;
        for (i = 0; i<g->n; i++)
                for (j = 0; j<g->n; j++)
                        A[i][j] = g->weight[i][j];
        printA(g);

        for (k = 0; k<g->n; k++) {
                for (i = 0; i<g->n; i++)
                        for (j = 0; j<g->n; j++)
                                if (A[i][k] + A[k][j] < A[i][j])
                                        A[i][j] = A[i][k] + A[k][j];
                printA(g);
        }
}
```

© 생능출판사 2019

Floyd의 최단경로 프로그램

```
int main(void)
{
		GraphType g = { 7,
		{{ 0,   7,  INF, INF,   3,  10, INF },
		{ 7,  0,    4,  10,   2,   6, INF },
		{ INF,  4,    0,   2, INF, INF, INF },
		{ INF, 10,    2,   0,  11,   9,   4 },
		{ 3,  2,  INF,  11,   0, INF,   5 },
		{ 10,  6,  INF,   9, INF,   0, INF },
		{ INF, INF, INF,   4,   5, INF,   0 } }
		};
		floyd(&g);
		return 0;
}
```

Floyd의 최단경로

	0	1	2	3	4	5	6
0	0	7	INF	INF	3	10	INF
1	7	0	4	10	2	6	INF
2	INF	4	0	2	INF	INF	INF
3	INF	10	2	0	11	9	4
4	3	2	INF	11	0	13	5
5	10	6	INF	9	13	0	INF
6	INF	INF	INF	4	5	INF	0

	0	1	2	3	4	5	6
0	0	7	11	17	3	10	INF
1	7	0	4	10	2	6	INF
2	11	4	0	2	6	10	INF
3	17	10	2	0	11	9	4
4	3	2	6	11	0	8	5
5	10	6	10	9	8	0	INF
6	INF	INF	INF	4	5	INF	0

	0	1	2	3	4	5	6
0	0	7	11	13	3	10	INF
1	7	0	4	6	2	6	INF
2	11	4	0	2	6	10	INF
3	13	6	2	0	8	9	4
4	3	2	6	8	0	8	5
5	10	6	10	9	8	0	INF
6	INF	INF	INF	4	5	INF	0

	0	1	2	3	4	5	6
0	0	7	11	13	3	10	17
1	7	0	4	6	2	6	10
2	11	4	0	2	6	10	6
3	13	6	2	0	8	9	4
4	3	2	6	8	0	8	5
5	10	6	10	9	8	0	3
6	17	10	6	4	5	13	0

	0	1	2	3	4	5	6
0	0	5	9	11	3	10	8
1	5	0	4	6	2	6	7
2	9	4	0	2	6	10	6
3	11	6	2	0	8	9	4
4	3	2	6	8	0	8	5
5	10	6	10	9	8	0	13
6	8	7	6	4	5	13	0

	0	1	2	3	4	5	6
0	0	5	9	11	3	10	8
1	5	0	4	6	2	6	7
2	9	4	0	2	6	10	6
3	11	6	2	0	8	9	4
4	3	2	6	8	0	8	5
5	10	6	10	9	8	0	13
6	8	7	6	4	5	13	10

	0	1	2	3	4	5	6
0	0	5	9	11	3	10	8
1	5	0	4	6	2	6	7
2	9	4	0	2	6	10	6
3	11	6	2	0	8	9	4
4	3	2	6	8	0	8	5
5	10	6	10	9	8	0	13
6	8	7	6	4	5	13	0

Floyd의 최단경로 알고리즘 복잡도

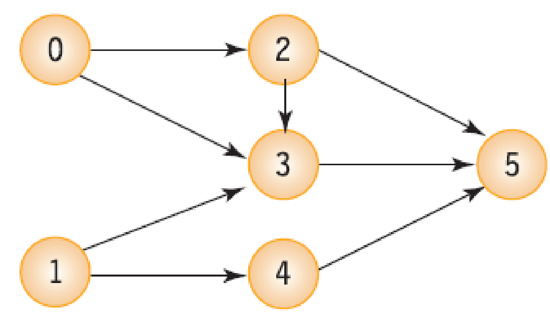
- 네트워크에 n개의 정점이 있다면, Floyd의 최단경로 알고리즘은 3중 반복문을 실행되므로 시간 복잡도는 $O(n^3)$ 이 된다
- 모든 정점상의 최단경로를 구하려면 Dijkstra의 알고리즘 $O(n2)$ 을 *n*번 반복해도 되며, 이 경우 전체 복잡도는 $O(n^3)$ 이 된다

© 생능출판사 2019

위상정렬(topological sort)

- 방향 그래프에서 간선 <u, v>가 있다면 정점 u는 정점 v를 선행함
- 방향 그래프 정점들의 선행 순서를 위배하지 않으면서 모든 정점을 나열
- 선수 과목은 과목들의 선행 관계 표현함

과목번호	과목명	선수과목
0	컴퓨터개론	없음
1	이산수학	없음
2	C언어	0
3	자료구조	0, 1, 2
4	확률	1
5	알고리즘	2, 3, 4

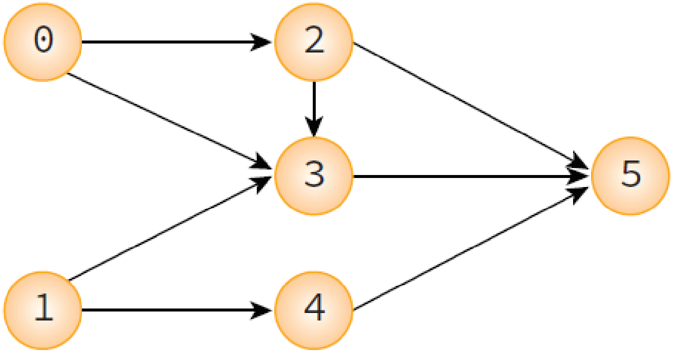


- 위상 순서(topological order)
  - (0,1,2,3,4,5) , (1,0,2,3,4,5)
- (2,0,1,3,4,5)는 위상 순서가 아님
  - 왜냐하면 2번 정점이 0번 정점 앞에 오기 때문

© 생능출판사 2019

위상정렬

과목번호	과목명	선수과목
0	컴퓨터 개론	없음
1	이산수학	없음
2	C언어	0
3	자료구조	0, 1, 2
4	확률	1
5	알고리즘	2, 3, 4



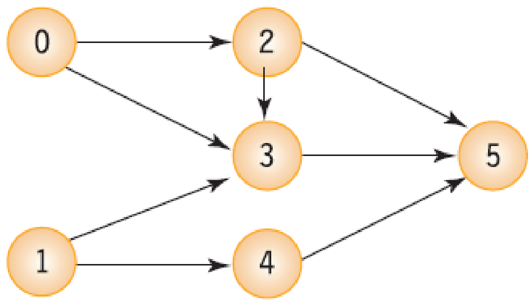
© 생능출판사 2019

위상정렬 알고리즘

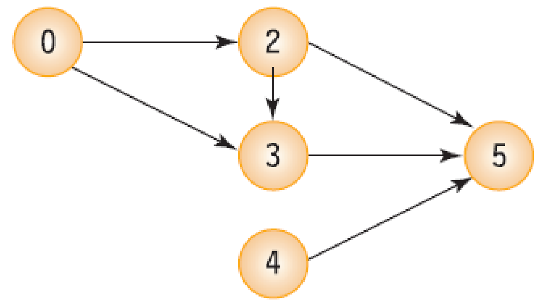
```
// Input: 그래프 G=(V,E)
// Output: 위상 정렬 순서

topo_sort(G)

for i←0 to n-1 do
    if( 모든 정점이 선행 정점을 가지면 )
        then 사이클이 존재하고 위상 정렬 불가;
    선행 정점을 가지지 않는 정점 v 선택;
    v를 출력;
    v와 v에서 나온 모든 간선들을 그래프에서 삭제;
```

© 생능출판사 2019

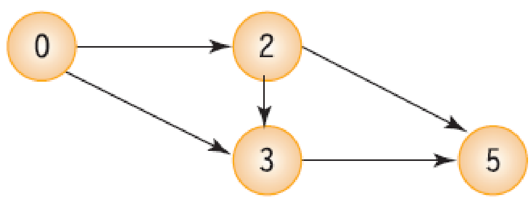
위상정렬의 예



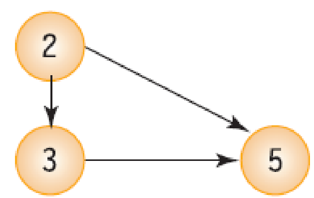
(a) 초기 상태



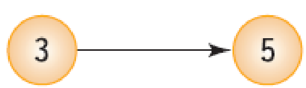
(b) 1 제거



(c) 4 제거



(d) 0 제거



(e) 2 제거

(f) 3 제거

© 생능출판사 2019

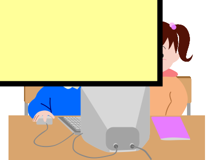
위상정렬 프로그램

```
// 위상정렬을 수행한다.
int topo_sort(GraphType *g)
{
		int i;
		StackType s;
		GraphNode *node;

		// 모든 정점의 진입 차수를 계산
		int *in_degree = (int *)malloc(g->n * sizeof(int));
		for (i = 0; i < g->n; i++)					// 초기화
			in_degree[i] = 0;
		for (i = 0; i < g->n; i++) {
			GraphNode *node = g->adj_list[i];//정점 i에서 나오는 간선들
			while (node != NULL) {
				in_degree[node->vertex]++;
				node = node->link;
			}
		}
```

© 생능출판사 2019

위상정렬 프로그램(cont.)

```
		// 진입 차수가 0인 정점을 스택에 삽입
		init(&s);
		for (i = 0; i < g->n; i++) {
			if (in_degree[i] == 0) push(&s, i);
		}
		// 위상 순서를 생성
		while (!is_empty(&s)) {
			int w;
			w = pop(&s);
			printf("정점 %d ->", w);				//정점 출력
			node = g->adj_list[w]; //각 정점의 진입 차수를 변경
			while (node != NULL) {
				int u = node->vertex;
				in_degree[u]--;					//진입 차수를 감소
				if (in_degree[u] == 0) push(&s, u);
				node = node->link;   // 다음 정점
			}
		}
		free(in_degree);
		printf("\n");
		return (i == g->n);	// 반환값이 1이면 성공, 0이면 실패
}
```



© 생능출판사 2019

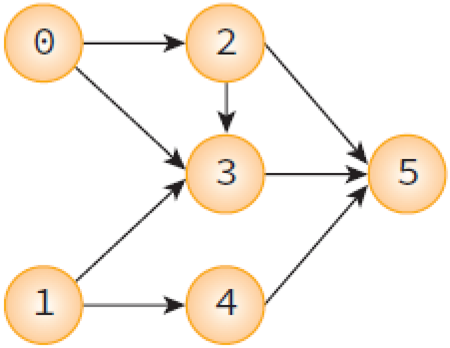
위상정렬 프로그램(cont.)

```
int main(void)
{
		GraphType g;

		graph_init(&g);
		insert_vertex(&g, 0);	insert_vertex(&g, 1);
		insert_vertex(&g, 2);	insert_vertex(&g, 3);
		insert_vertex(&g, 4);	insert_vertex(&g, 5);

		//정점 0의 인접 리스트 생성
		insert_edge(&g, 0, 2); insert_edge(&g, 0, 3);
		//정점 1의 인접 리스트 생성
		insert_edge(&g, 1, 3); insert_edge(&g, 1, 4);
		//정점 2의 인접 리스트 생성
		insert_edge(&g, 2, 3); insert_edge(&g, 2, 5);
		//정점 3의 인접 리스트 생성
		insert_edge(&g, 3, 5); //정점 4의 인접 리스트 생성
		insert_edge(&g, 4, 5);
		//위상 정렬
		topo_sort(&g);
		// 동적 메모리 반환 코드 생략
		return 0;
}
```


© 생능출판사 2019

실행 결과

```
정점 1 ->정점 4 ->정점 0 ->정점 2 ->정점 3 ->정점 5 ->
```



© 생능출판사 2019