



Data Structure & Algorithm

자료구조 및 알고리즘

25. 그래프 3 (Graph Part 3)



23강 보고서 돌아보기



- 그래프의 정점 개수를 V , 간선 개수를 E 라고 하자. 다음을 빅-오 표기법으로 나타내어라.
- 인접 행렬을 사용하여 그래프를 나타낼 때 공간복잡도 $O(V^2)$
- 인접 행렬을 사용했을 때, 두 노드 간 간선 존재 여부를 알아 낼 때 시간복잡도 $O(1)$
- 인접 리스트를 사용하여 그래프를 나타낼 때 공간복잡도 $O(V + E)$
- 인접 리스트를 사용했을 때, 두 노드 간 간선 존재 여부를 알아 낼 때 시간복잡도 $O(V)$

24강 보고서 돌아보기



- 1) DFS를 이용하여 연락을 했을 때 X가 연락을 받았다면 시작 사람으로부터 X까지 연락이 전해진 경로가 최단 연락 횟수를 가지는 것이 보장되는가? BFS로 했을 때는?
 - 1) DFS는 보장이 안된다.
 - 2) BFS는 보장이 된다. 단, 연락 예제에서는 간선의 가중치를 모두 동등하다고 봤지만 가중치가 다를 경우 합을 최소화하는 것은 아니다 -> 최단 거리 탐색 알고리즘의 필요성

24강 보고서 돌아보기



1) DFS 또는 BFS를 이용하여 오른쪽 그림과 같은 미로에서 S에서 F로 가는 길을 찾을 수 있을까?

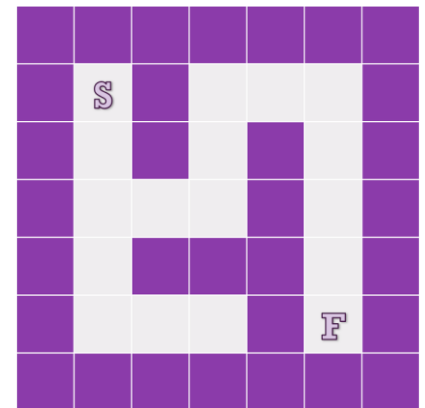
1) 미로를 어떻게 그래프로 변환해야 할까?

- 정점: 미로의 칸
- 간선: 각 칸에서 인접한 4칸으로 갈 수 있다면 연결

2) 미로의 크기가 $N * M$ 이라면 시간복잡도는 얼마일까?

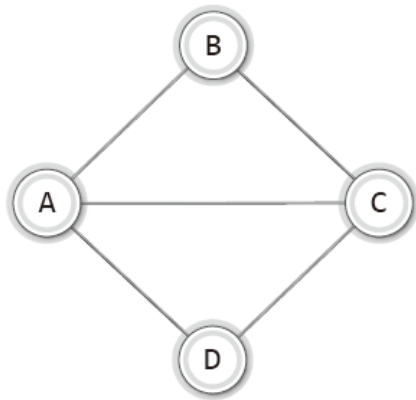
- 정점 수: $O(NM)$
- 간선 수: $O(4NM)$
- BFS 시간복잡도: $O(V+E) = O(NM)$

3) 최소 길이가 보장?



최소 비용 신장 트리

사이클의 이해



정점 B에서 점점 D에 이르는 단순 경로

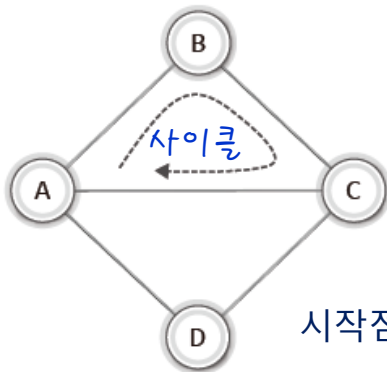
- B-A-D 단순 경로
- B-C-D 단순 경로
- B-A-C-D 조금 돌아가는 단순 경로
- B-C-A-D 조금 돌아가는 단순 경로

단순 경로는 간선을 중복 포함하지 않는다.

단순 경로가 아닌 정점 B에서 점점 D에 이르는 경로

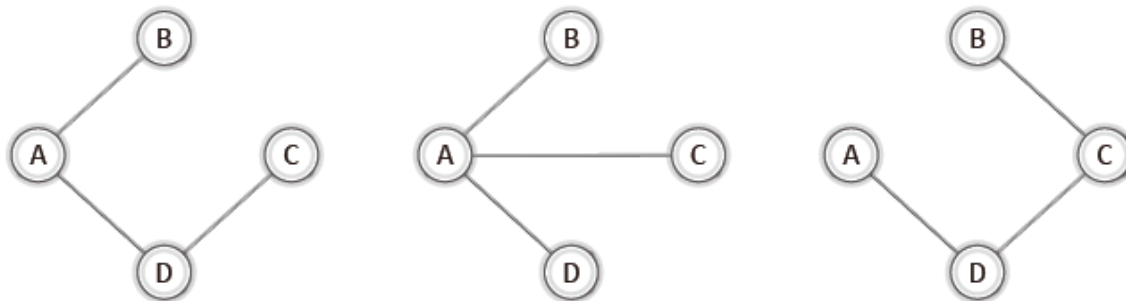
- B-A-C-B-A-D

B와 A를 잇는 간선이 두 번 포함됨!



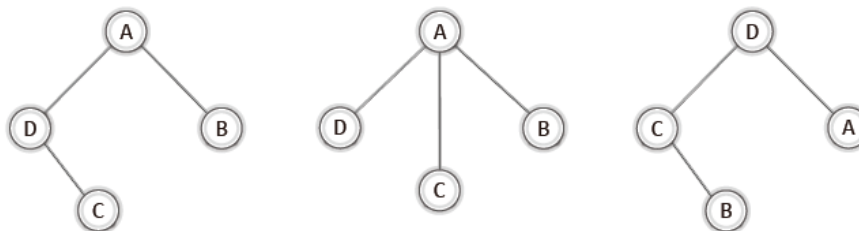
시작점과 끝점이 같은 단순 경로를 가리켜 '사이클' 이라 한다.

사이클을 형성하지 않는 그래프



사이클을 형성하지 않는 그래프들은 일종의 트리로 볼 수 있다.

위의 그래프를 회전시킨 결과

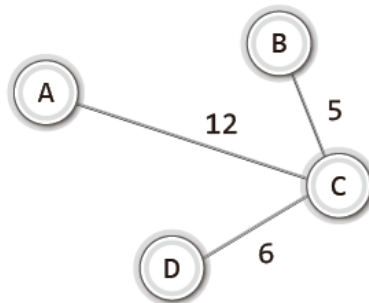
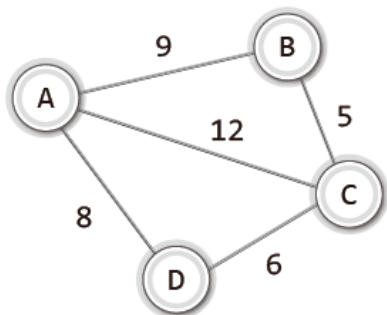


최소 비용 신장 트리의 이해와 적용

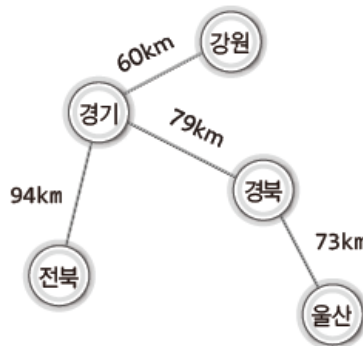
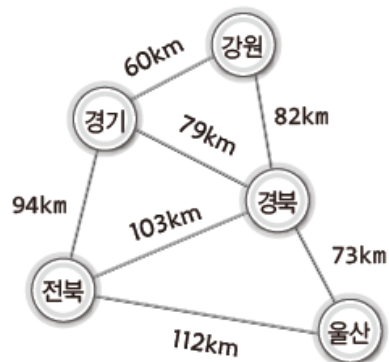


- 그래프의 모든 정점을 포함한다.
- 그래프의 모든 정점이 간선에 의해서 하나로 연결되어 있다.
- 그래프 내에서 사이클을 형성하지 않는다.

신장 트리의 특징



'최소 비용 신장 트리' 구성의 예

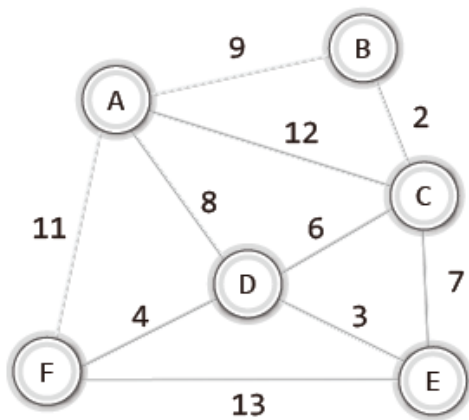


'최소 비용 신장 트리' 구성의 예

크루스칼 알고리즘 1: 과정 1~



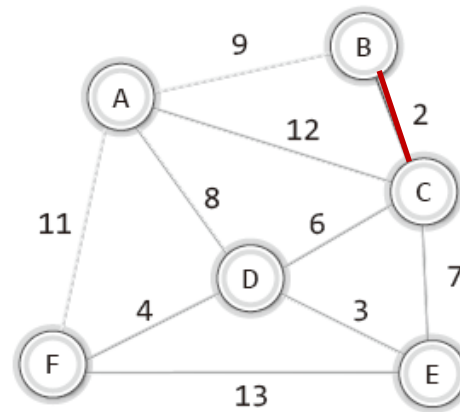
가중치를 기준으로 간선을 정렬한 후에 MST가 될 때까지 간선을 하나씩 선택 또는 삭제해 나가는 방식



▶ [그림 14-49: 크루스칼 알고리즘 1의 1/4]

2, 3, 4, 6, 7, 8, 9, 11, 12, 13

가중치의 오름차순 정렬



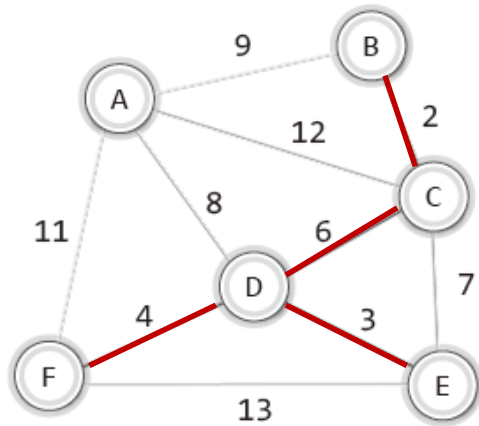
▶ [그림 14-50: 크루스칼 알고리즘 1의 2/4]



2, 3, 4, 6, 7, 8, 9, 11, 12, 13

가중치의 오름차순 정렬

크루스칼 알고리즘 1: 과정 3~



이동

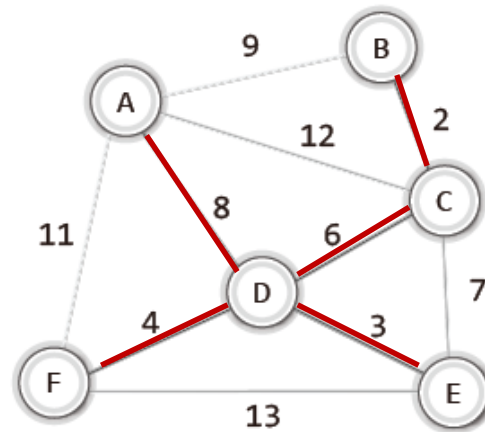
 2, 3, 4, 6, 7, 8, 9, 11, 12, 13
 가중치의 오름차순 정렬

▶ [그림 14-51: 크루스칼 알고리즘 1의 3/4]

최소 비용 신장 트리의 조건인

간선의 수 + 1 = 정점의 수를 만족하니

이것으로 최소 비용 신장 트리 형성 완료!



가중치가 7인 간선을 포함시키면
 사이클이 형성된다! 따라서 건너 뛴다!

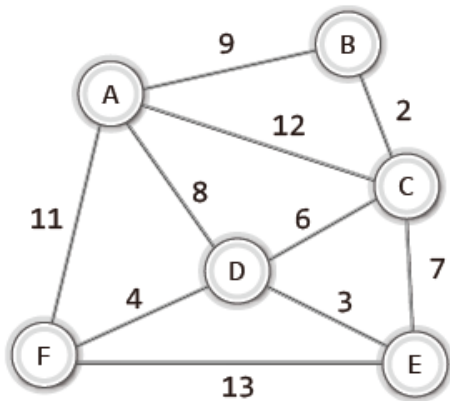
2, 3, 4, 6, 7, 8, 9, 11, 12, 13
 가중치의 오름차순 정렬

▶ [그림 14-52: 크루스칼 알고리즘 1의 4/4]

크루스칼 알고리즘 2: 과정 1~



높은 가중치의 간선을 하나씩 빼는 방식의 크루스칼 알고리즘

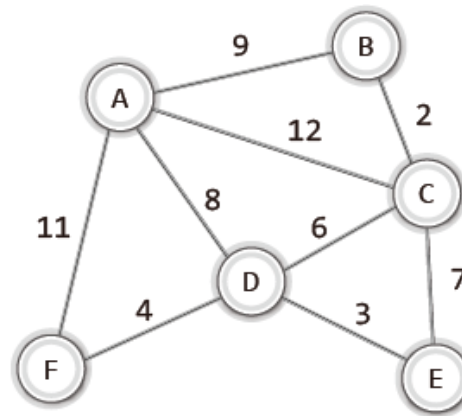


13, 12, 11, 9, 8, 7, 6, 4, 3, 2

가중치의 내림차순 정렬

▶ [그림 14-54: 크루스칼 알고리즘 2의 1/4]

가중치가 13인 간선이 없어도 모든 정점은
연결이 되므로 이를 삭제한다.



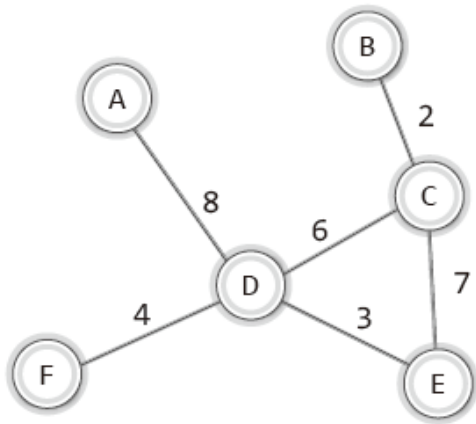
▶ [그림 14-55: 크루스칼 알고리즘 2의 2/4]



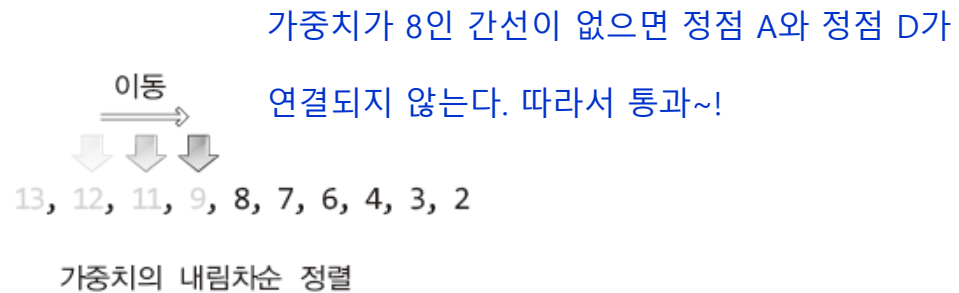
13, 12, 11, 9, 8, 7, 6, 4, 3, 2

가중치의 내림차순 정렬

크루스칼 알고리즘 2: 과정 3~



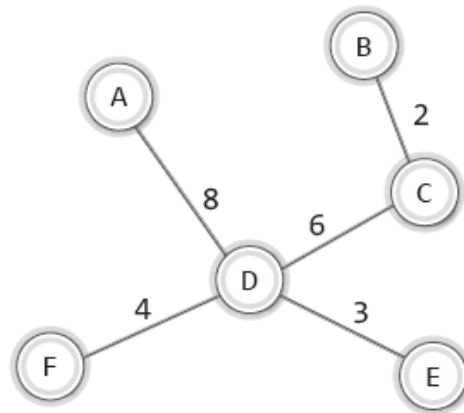
▶ [그림 14-56: 크루스칼 알고리즘 2의 3/4]



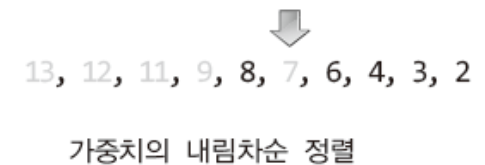
최소 비용 신장 트리의 조건인

간선의 수 + 1 = 정점의 수를 만족하니

이것으로 최소 비용 신장 트리 형성 완료!



▶ [그림 14-57: 크루스칼 알고리즘 2의 4/4]



크루스칼 알고리즘의 구현을 위한 계획1

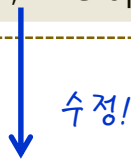


우리가 선택한 구현 방식

가중치를 기준으로 간선을 내림차순으로 정렬한 다음 높은 가중치의 간선부터 시작해서 하나씩 그래프에서 제거하는 방식

구현에 사용할 도구들

- DLinkedList.h, DLinkedList.c 연결 리스트
- ArrayBaseStack.h, ArrayBaseStack.c 배열 기반 스택
- ALGraphDFS.h, ALGraphDFS.c 깊이 우선 탐색을 포함하는 그래프



크루스칼 알고리즘이 담기는 파일들

- ALGraphKruskal.h, ALGraphKruskal.c 가중치 그래프의 구현 결과

그리고 가중치 그래프의 구현을 위해서는 가중치가 포함된 간선을 표현한 구조체가 정의되어야 한다.
헤더파일 ALEdge.h를 만들어서 해당 구조체를 정의한다.

크루스칼 알고리즘의 구현을 위한 계획2



다음 질문에 답을 하는 함수가 필요하다!

이 간선을 삭제한 후에도 이 간선에 의해 연결된 두 정점을 연결하는 경로가 있는가?

이를 위해 DFS 알고리즘을 활용! DFS의 구현결과인 DFShowGraphVertex 함수를 확장하여 이 질문에 답을 하도록 한다!

이는 크루스칼 알고리즘의 일부이다.

그래프를 구성하는 간선들을 가중치를 기준으로 정렬할 수 있어야 한다.

이를 위해서 앞서 구현한 우선순위 큐를 활용

- PriorityQueue.h, PriorityQueue.c 우선순위 큐
- UsefulHeap.h, UsefulHeap.c 우선순위 큐의 기반이 되는 힙

크루스칼 알고리즘의 구현을 위한 계획3



- DLinkedList.h, DLinkedList.c 연결 리스트
- ArrayBaseStack.h, ArrayBaseStack.c 배열 기반 스택
- ALGraphKruskal.h, ALGraphKruskal.c 크루스칼 알고리즘 기반의 그래프
- PriorityQueue.h, PriorityQueue.c 우선순위 큐
- UsefulHeap.h, UsefulHeap.c 우선순위 큐의 기반이 되는 힙
- ALEdge.h 가중치가 포함된 간선의 표현을 위한 구조체

최종적으로 크루스칼 알고리즘의 구현을 보이기 위한 프로젝트의 헤더파일과 소스파일의 구성

크루스칼 알고리즘의 구현: 헤더파일



```
enum {A, B, C, D, E, F, G, H, I, J};
```

```
typedef struct _ual
```

```
{
```

```
    int numV;
```

```
    int numE;
```

```
    List * adjList;
```

```
    int * visitInfo;
```

```
    PQueue pqueue;    // 간선의 가중치 정보 저장
```

```
} ALGraph;
```

```
void GraphInit(ALGraph * pg, int nv);
```

```
void GraphDestroy(ALGraph * pg);
```

```
void AddEdge(ALGraph * pg, int fromV, int toV, int weight);
```

```
void ShowGraphEdgeInfo(ALGraph * pg);
```

```
void DFSshowGraphVertex(ALGraph * pg, int startV);
```

```
void ConKruskalMST(ALGraph * pg);    // 최소 비용 신장 트리의 구성
```

```
void ShowGraphEdgeWeightInfo(ALGraph * pg);    // 가중치 정보 출력
```

```
typedef struct _edge
```

```
{
```

```
    int v1;    // 간선이 연결하는 첫 번째 정점
```

```
    int v2;    // 간선이 연결하는 두 번째 정점
```

```
    int weight; // 간선의 가중치
```

```
} Edge;
```

ALEdge.h

ALGraphKruskal.h

크루스칼 알고리즘을 구현한 함수: 수정된 함수들



```
void GraphInit(ALGraph * pg, int nv)
{
    . . . . 여기까지는 ALGraphDFS.c의 GraphInit 함수와 동일 . . .

    // 우선순위 큐의 초기화
    PQueueInit(&(pg->pqueue), PQWeightComp);    // 추가된 문장
}
```

```
int PQWeightComp(Edge d1, Edge d2)
{
    return d1.weight - d2.weight;
}
```

가중치 기준 내림차순으로
간선 정보 꺼내기 위한 정의!

```
void AddEdge(ALGraph * pg, int fromV, int toV, int weight)
{
    Edge edge = {fromV, toV, weight};    // 간선의 가중치 정보를 담음
    LInsert(&(pg->adjList[fromV]), toV);
    LInsert(&(pg->adjList[toV]), fromV);
    pg->numE += 1;

    // 간선의 가중치 정보를 우선순위 큐에 저장
    PEnqueue(&(pg->pqueue), edge);
}
```

크루스칼 알고리즘을 구현한 함수: ConKruskalMST



```
void ConKruskalMST(ALGraph * pg)    // 크루스칼 알고리즘 기반 MST의 구성
{
    Edge recvEdge[20];    // 복원할 간선의 정보 저장
    Edge edge;
    int eidx = 0;
    int i;

    // MST를 형성할 때까지 아래의 while문을 반복
    while(pg->numE+1 > pg->numV)    // MST 간선의 수 + 1 == 정점의 수
    {
        edge = PDequeue(&(pg->pqueue)); 가중치 순으로 간선 정보 획득!
        RemoveEdge(pg, edge.v1, edge.v2); 획득한 정보의 간선 실제 삭제!

        if(!IsConnVertex(pg, edge.v1, edge.v2)) 삭제 후 두 정점 연결 경로 있는지 확인!
        {
            RecoverEdge(pg, edge.v1, edge.v2, edge.weight); 연결 경로 없으면 간선 복원!
            recvEdge[eidx++] = edge;
        }
    }

    // 우선순위 큐에서 삭제된 간선의 정보를 회복
    for(i=0; i<eidx; i++)
        PEnqueue(&(pg->pqueue), recvEdge[i]);
}
```

- RemoveEdge 그래프에서 간선을 삭제한다.
- IsConnVertex 두 정점이 연결되어 있는지 확인한다.
- RecoverEdge 삭제된 간선을 다시 삽입한다.

크루스칼 알고리즘의 완성을 돕는 함수들 1



```
// 간선의 소멸
void RemoveEdge(ALGraph * pg, int fromV, int toV)
{
    RemoveWayEdge(pg, fromV, toV);
    RemoveWayEdge(pg, toV, fromV);
    (pg->numE)--;
}
```

인접 리스트 기반 무방향 그래프인 관계로 하나의 간선을 완전히 소멸하기 위해서는 두 개의 간선 정보를 소멸시켜야 한다.

```
void RecoverEdge(ALGraph * pg, int fromV, int toV, int weight)
{
    LInsert(&(pg->adjList[fromV]), toV);
    LInsert(&(pg->adjList[toV]), fromV);
    (pg->numE)++;
}
```

AddEdge 함수와 달리 간선의 가중치 정보를 별도로 저장하지 않는다. 이렇듯 가중치 정보를 별도로 저장하지 않는 이유는 크루스칼 알고리즘의 구현 내용을 통해 이해할 수 있다.

크루스칼 알고리즘의 완성을 돕는 함수들 2



```
// 한쪽 방향의 간선 소멸
void RemoveWayEdge(ALGraph * pg, int fromV, int toV)
{
    int edge;
    if(LFirst(&(pg->adjList[fromV]), &edge))
    {
        if(edge == toV) {
            LRemove(&(pg->adjList[fromV]));
            return;
        }
        while(LNext(&(pg->adjList[fromV]), &edge))
        {
            if(edge == toV) {
                LRemove(&(pg->adjList[fromV]));
                return;
            }
        }
    }
}
```

이렇듯 RemoveEdge 함수의 완성을 돕는 RemoveWayEdge 함수를 별도로 정의하면 방향 그래프의 구현을 위한 확장이 용이하다!

크루스칼 알고리즘의 완성을 돕는 함수들 3



// 인자로 전달된 두 정점이 연결되어 있다면 TRUE, 그렇지 않다면 FALSE 반환
int IsConnVertex(ALGraph * pg, int v1, int v2)

```
{
    Stack stack;
    int visitV = v1;
    int nextV;

    StackInit(&stack);
    VisitVertex(pg, visitV);
    SPush(&stack, visitV);

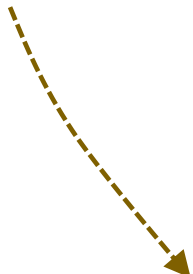
    while(LFirst(&(pg->adjList[visitV]), &nextV) == TRUE)
    {
        int visitFlag = FALSE;
        // 정점을 돌아다니는 도중에 목표를 찾는다면 TRUE를 반환한다.
        if(nextV == v2) {
            // 함수가 반환하기 전에 초기화를 진행한다.
            memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
            return TRUE;    // 목표를 찾았으니 TRUE를 반환!
        }
    }
```

```
        if(VisitVertex(pg, nextV) == TRUE)
        {
            SPush(&stack, visitV);
            visitV = nextV;
            visitFlag = TRUE;
        }
        else
        {
            while(LNext(&(pg->adjList[visitV]), &nextV) == TRUE)
            {
                // 정점을 돌아다니는 도중에 목표를 찾는다면 TRUE를 반환한다.
                if(nextV == v2) {
                    // 함수가 반환하기 전에 초기화를 진행한다.
                    memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
                    return TRUE;    // 목표를 찾았으니 TRUE를 반환!
                }
                if(VisitVertex(pg, nextV) == TRUE) {
                    SPush(&stack, visitV);
                    visitV = nextV;
                    visitFlag = TRUE;
                    break;
                }
            }
        }
    }
}
```

DFShowGraphVertex 함수와의 비교를 통해서 어떻게
수정되었고 또 그 결과 어떻게 두 정점의 연결을 확인하는지
이해하자!

크루스칼 알고리즘의 완성을 돕는 함수들 3





```
if(visitFlag == FALSE)
{
    if(SIsEmpty(&stack) == TRUE)
        break;
    else
        visitV = SPop(&stack);
}

memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
return FALSE;    // 여기까지 왔다는 것은 목표를 찾지 못했다는 것!
}
```

이로써 부분적으로 필요한 모든 설명이 완료 되었으니 전체 코드를 확인하고 교재에서 제공하는 main 함수의 실행 결과도 직접 확인해보자!

요약



- 최소 비용 신장 트리 (Minimum Spanning Tree)
 - 그래프 내의 모든 정점을 포함하는 트리 중 최소의 간선의 가중치 합을 가지는 트리
 - 통신망 연결 등
- 크루스칼 알고리즘
 - 1) 낮은 가중치를 가지는 간선들부터 선택하되, 선택된 간선들끼리 사이클이 생기지 않도록 한다.
 - 2) 높은 가중치를 가지는 간선부터 제거하되, 남아있는 그래프가 분리되지 않도록 한다.

25강 출석 인정을 위한 보고서



- 아래 문제에 대한 답을 포털에 제출
- 1) 크루스칼 알고리즘의 최악의 경우 시간복잡도는 얼마일까?
 - 2) 그래프 내의 모든 정점을 포함하는 트리 중 **최대**의 가중치 합을 가지는 트리를 **최대 비용** 신장 트리라고 하자. 어떻게 계산할 수 있을까?