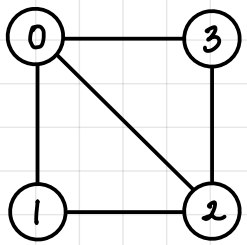
<그래프 1>

그래프란?

그래프는 정점과 간선으로 이루어진 자료구조이다. 정확히는 정점(vertex)간의 관계를 표현하는 조직도라고 볼 수도 있다. 이런면에서 트리는 그래프의 일종인 셈이다.
다만 트리와는 달리 그래프는 정점마다 간선이 없을 수도 있고 있을 수도 있으며 루트 노드, 부모와 자식이라는 개념이 존재하지 않는다.
또한 그래프는 네트워크 모델 즉, 객체와 이에 대한 관계를 나타내는 유연한 방식으로 이해할 수 있다.

특히 그래프를 순회하는 방식인 DFS와 BFS를 잘 알아두어야 한다.

그래프에서 사용하는 용어



정점(Vertice) : 노드(node)라고도 하며 정점에는 데이터가 저장된다. (0, 1, 2, 3)

간선(edge) : 링크(arcs)라고도 하며 노드간의 관계를 나타낸다.

인접 정점(adjacent vertex) : 간선에 의해 연결된 정점이다. (정점 0과 정점 1은 인접 정점)

단순 경로(Simple-path) : 경로 중 반복되는 정점이 없는 것으로, 같은 간선을 지나가지 않는 경로이다.

차수(degree) : 무방향 그래프에서 하나의 정점에 인접한 정점의 수를 말한다. (정점 0의 차수는 3)

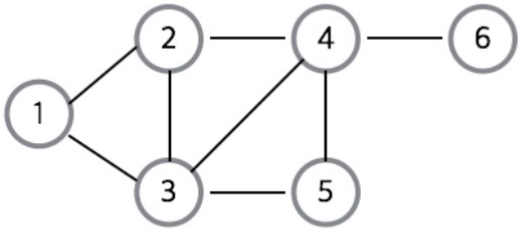
진출 차수(out-degree) : 방향 그래프에서 사용되는 용어로 한 노드에서 외부로 향하는 간선의 수를 뜻한다.

진입 차수(in-degree) : 방향 그래프에서 사용되는 용어로 외부 노드에서 들어오는 간선의 수를 뜻한다.

그래프 구현 방법

그래프를 구현하는 방법에는 인접행렬(Adjacency Matrix)와 인접리스트(Adjacency List) 방식이 있다.
두 개의 구현 방식은 각각의 상반된 장단점을 가지고 있는데 대부분 인접리스트 형식을 많이 사용한다.

인접 행렬 방식



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	0	0
3	1	1	0	1	1	0
4	0	1	1	0	1	1
5	0	0	1	1	0	0
6	0	0	0	1	0	0

인접행렬은 그래프의 노드를 2차원 배열로 만든 것이다.
완성된 배열의 모양은 1, 2, 3, 4, 5, 6의 정점을 연결하는 노드에 다른 노드들이 인접 정점이라면 1, 아니면 0을 넣어준다.

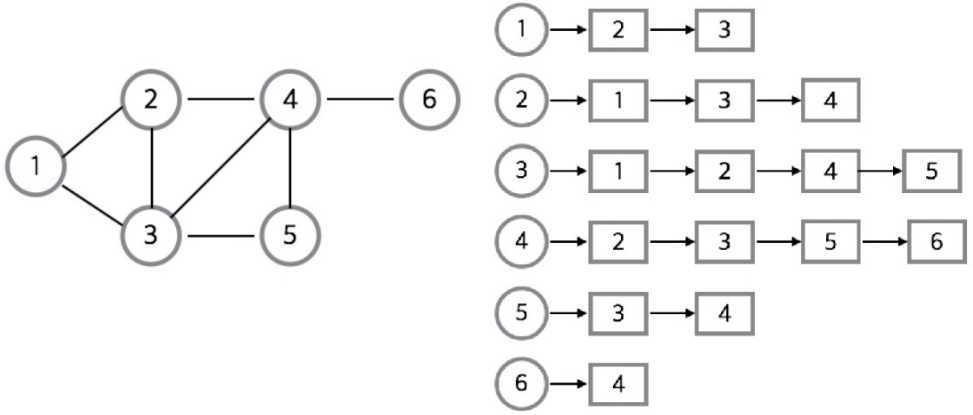
인접행렬 장점

1. 2차원 배열 안에 모든 정점들의 간선 정보를 담기 때문에 배열의 위치를 확인하면 두 점에 대한 연결 정보를 조회할 때 O(1)의 시간복잡도면 가능하다.
2. 구현이 비교적 간편하다.

인접행렬 단점

1. 모든 정점에 대해 간선 정보를 대입해야 하므로 $O(n^2)$의 시간 복잡도가 소요된다.
2. 무조건 2차원 배열이 필요하기에 필요 이상의 공간이 낭비된다.

인접 리스트 방식



인접리스트란 그래프의 노드들을 리스트로 표현한 것이다.
주로 정점의 리스트 배열을 만들어 관계를 설정해줌으로써 구현한다.

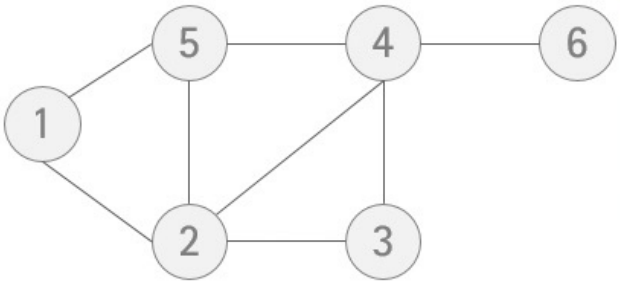
인접리스트 장점

1. 정점들의 연결 정보를 탐색할 때 O(n)의 시간이면 가능하다. (n : 간선의 갯수)
2. 필요한 만큼의 공간만 사용하기 때문에 공간의 낭비가 적다.

인접리스트 단점

1. 특정 두 점이 연결되었는지 확인하려면 인접행렬에 비해 시간이 오래 걸린다.
   (배열보다 Search 속도가 느림)
2. 구현이 비교적 어렵다.

간선리스트 방식

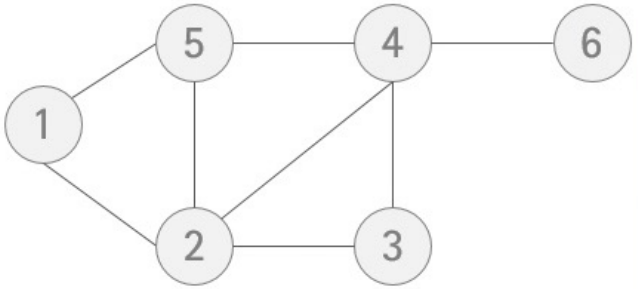


i	0	1	2	3	4	5	6
cnt[i]	0	2	4	2	4	3	1

E[0]	1 2	E[8]	4 2
E[1]	1 5	E[9]	4 3
E[2]	2 1	E[10]	4 5
E[3]	2 3	E[11]	4 6
E[4]	2 4	E[12]	5 1
E[5]	2 5	E13]	5 2
E[6]	3 2	E[14]	5 4
E[7]	3 4	E[15]	6 4

간선 리스트는 간선을 다른 배열(E[])에 모두 저장하는 방식이다.
그림에서 처럼 E라는 배열에 간선을 모두 저장한다.
각 간선의 앞 정점을 기준으로 개수를 센다.

(간선리스트는 인접 행렬과 인접 리스트를 사용하지 못할 때 사용한다.)



i	0	1	2	3	4	5	6
cnt[i]	0	2	6	8	12	15	16

E[0]	1 2	E[8]	4 2
E[1]	1 5	E[9]	4 3
E[2]	2 1	E[10]	4 5
E[3]	2 3	E[11]	4 6
E[4]	2 4	E[12]	5 1
E[5]	2 5	E13]	5 2
E[6]	3 2	E[14]	5 4
E[7]	3 4	E[15]	6 4

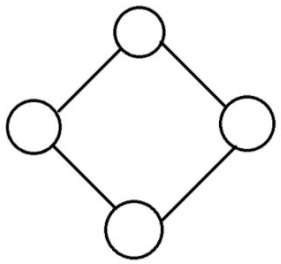
E 배열에서 i 정점과 연결된 간선을 찾기 위해서 i-1 정점의 개수와 i 정점의 개수를 더한다.

i 번 정점과 연결된 간선은 E 배열에서 Cnt[i-1] ~ Cnt[i]-1 까지이다.

예를 들어 3번 간선은 Cnt[2]부터 Cnt[3]-1, 즉 E[6]~E[7]이 3번 간선이다.

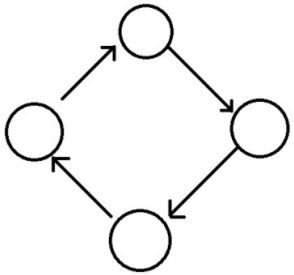
다양한 그래프의 종류

그래프는 구현되어진 특성에 따라 여러 가지 종류로 나누어진다. 대표적인 그래프 유형은 아래와 같다.



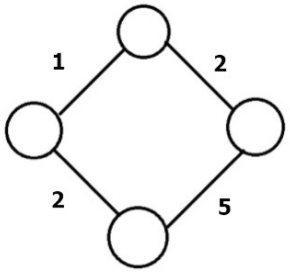
무방향 그래프

무방향 그래프는 두 정점을 연결하는 간선에 방향이 없는 그래프이다.



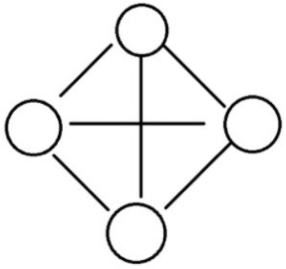
방향 그래프

방향그래프는 두 정점을 연결하는 간선에 방향이 존재하는 그래프이다.
간선의 방향으로만 이동할 수 있다.



가중치 그래프

가중치 그래프는 두 정점을 이동할 때 비용이 드는 그래프이다.

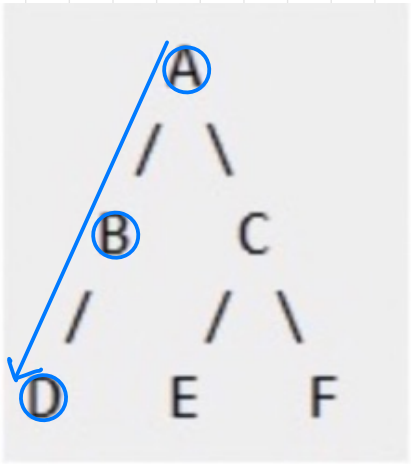


완전 그래프

완전 그래프는 모든 정점이 간선으로 연결되어 있는 그래프이다.

그래프의 탐색 : DFS (Depth First Search)

A, B, D, C, E, F



A, B, D, C, E, F

깊이 우선 탐색 (DFS)이란?

DFS는 그래프 전체를 탐색하는 방법 중 하나로써 시작점 부터 다음 분기로 넘어가기 전에 해당 분기를 완벽하게 탐색하고 넘어가는 방법이다.
스택이나 재귀함수를 통해서 구현할 수 있는데, 재귀함수가 구현이 간편하기에 대부분 재귀함수로 구현한다.
구현 시 주의할 점은 노드를 방문 시 방문 여부를 반드시 검사해야한다. 그렇지 않으면 무한루프에 빠질 수 있다.

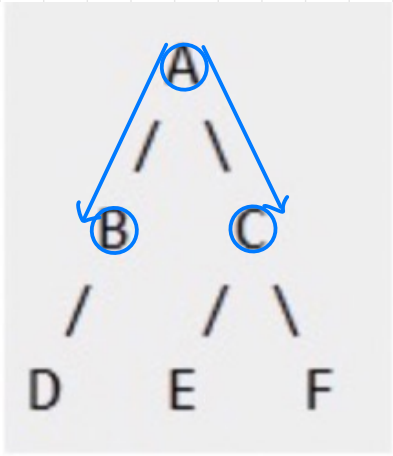
DFS의 장점

1. 현재 경로상의 노드들만 기억하면 되므로, 저장 공간의 수요가 비교적 적다.
2. 목표 노드가 깊은 단계에 있는 경우 해를 빨리 구할 수 있다.
3. 구현이 너비 우선 탐색(BFS) 보다 간단하다.

DFS의 단점

1. 단순 검색 속도는 너비 우선 탐색(BFS) 보다 느리다.
2. 해가 없는 경우에 빠질 가능성이 있다.
   (사전에 임의의 깊이를 지정한 후 탐색하고, 목표 노드를 발견하지 못할 경우 다음 경로를 탐색하도록 한다.)
3. 깊이 우선 탐색은 해를 구하면 탐색이 종료되므로, 구한 해가 최단 경로가 된다는 보장이 없다.
   (목표에 이르는 경로가 다수인 경우 구한 해가 최적이 아닐 수 있다.)

그래프의 탐색 : BFS (Breadth First Search)



A, B, C, D, E, F

A, B, C, D, E, F

너비 우선탐색 (BFS) 이란 ?

BFS는 그래프 전체를 탐색하는 방법 중 하나로, 루트 노드(혹은 다른 임의의 노드) 에서 시작해서 인접한 노드를 먼저 탐색하는 방법이다.
시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하여 순회함으로써 노드를 넓게(wide)탐색한다.
주로 두 노드 사이의 최단 경로 혹은 임의의 경로를 찾고 싶을 때 이 방법을 사용한다.
구현은 큐라는 자료에 이웃하는 정점을 다 담아놓고 차례대로 POP을 하는 방식으로 구현한다.

BFS의 장점

1. 노드의 수가 적고 깊이가 얕은 경우에 빠르게 동작할 수 있다.
2. 단순 검색 속도가 깊이 우선 탐색(DFS)보다 빠르다.
3. 너비를 우선 탐색하기에 답이 되는 경로가 여러 개인 경우에도 최단경로임을 보장한다.
4. 최단 경로가 존재한다면 어느 한 경로가 무한히 깊어 진다해도 최단 경로를 반드시 찾을 수 있다.

BFS의 단점

1. 재귀호출의 DFS와는 달리 큐에 다음에 탐색할 정점들을 저장해야 하므로 저장공간이 많이 필요하다.
2. 노드의 수가 늘어나면 탐색해야하는 노드 또한 많아지기에 비현실적이다.

DFS , BFS 간단한 구현

```c
#include <stdio.h>

int Graph[1001][1001]={0};
int DFSvisit[1001]={0};
int BFSvisit[1001]={0};
int queue[1001];


void DFS(int v,int N){
    int i;

    DFSvisit[v]=1;
    printf("%d ",v);
    for(i=1;i<=N;i++){
        if(Graph[v][i]==1 && DFSvisit[i]==0){
            DFS(i,N);
        }
    }

    return;
}

void BFS(int v,int N){
    int front=0,rear=0,Pop,i;

    printf("%d ",v);
    queue[0]=v;
    rear++;
    BFSvisit[v]=1;

    while(front<rear){
        Pop=queue[front];
        front++;

        for(i=1;i<=N;i++){
            if(Graph[Pop][i]==1 && BFSvisit[i]==0){
                printf("%d ",i);
                queue[rear]=i;
                rear++;
                BFSvisit[i]=1;
            }
        }
    }

    return;
}
```