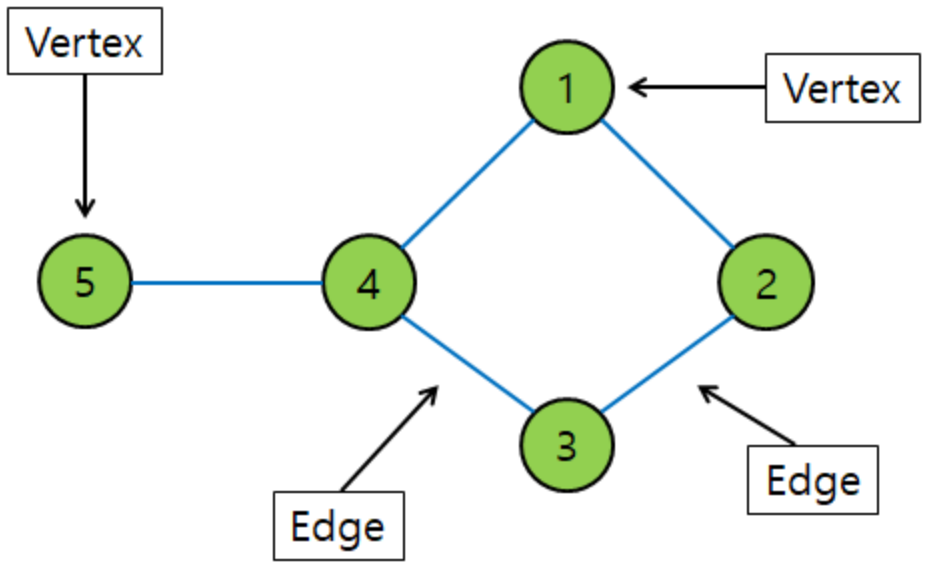
스터디 5주차 (엄태호)

그래프 (Graph)

- 그래프 자료구조는 정점과 간선으로 이루어진 비선형 자료구조로 원소간의 관계를 표현하고 분석하는데 주로 사용됩니다.
- 그래프는 트리와 다르게 부모와 자식이라는 개념이 없으며 간선은 있을수도, 없을수도 있습니다.
- 그래프는 자체 간선(Self-Loop)이 가능합니다.
  즉, 특정 노드가 자기 자신을 향하는 간선을 가질 수 있습니다.
- 지도, 지하철 노선도, 도로, 전기회로 소자, 소셜 네트워크 분석, 컴퓨터 네트워크 등 다양한 분야에서 사용되는 자료구조입니다.

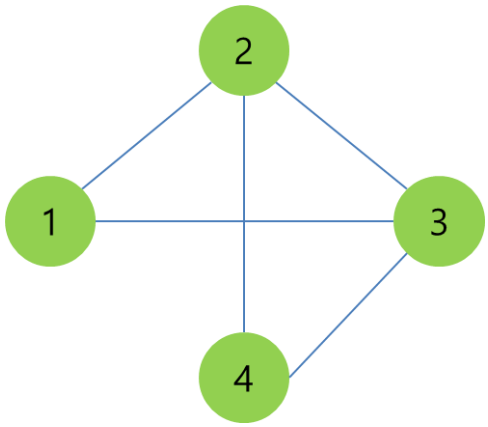
그래프에서 사용되는 용어



- **정점 (Vertex, Vertice)**
  정점은 노드라고도 하며 데이터가 저장되는 공간을 말합니다.
  - **인접 정점(Adjacent Vertex)**
    두 노드를 연결하는 간선이 있을 때 연결된 두 노드의 관계를 말하며 이처럼 두 정점이 연결되어있다면 서로 인접하다고 말합니다.
- **간선 (Edge)**
  링크라고도 부르며 정점과 정점을 잇는 선으로 정점간의 관계를 나타냅니다.
  그래서, 그래프는 G(V, E)라고 표현하기도 합니다.
  - **부속 (Incident)**
    인접한 두 정점 사이의 간선은 두 정점에 부속되었다고 표현합니다.
- **차수 (Degree)**
  정점에 부속된 간선의 수를 말하며 A라는 정점에 부속된 간선이 3개라면 A의 차수는 3이 됩니다.
  - **진입차수 (In-degree)**
    특정 정점을 머리로 하는 간선의 수를 말하며 다른 정점에서 특정 정점으로 들어오는 간선의 수를 말합니다.
  - **진출차수 (Out-degree)**
    진입차수와 반대로 특정 정점을 꼬리로 하는 간선의 수를 말하며 특정 정점에서 외부로 향하는 간선의 수를 말합니다.
    ✅ **진입차수와 진출차수는 방향 그래프에서만 분류합니다.**
- **경로 (Path)**
  특정 노드의 간선을 따라 갈 수 있는 길을 나열한 것으로
  Vi정점에서 Vj정점까지 간선으로 연결된 정점을 순서대로 나열한 리스트를 말합니다.
  - **단순 경로 (Simple Path)**
    모두 다른 정점으로 구성된 경로를 말합니다.
    즉, 경로에 반복해서 방문하는 정점이 없고 같은 간선을 다시 지나가지 않는 경로를 말합니다.
    - **순환 (Cycle)**
      단순 경로 중에서 경로의 시작 정점과 마지막 정점이 같은 경로를 말합니다.
      즉, 경로에서 시작점과 끝점을 제외하고는 중복되는 노드가 없어야 합니다.
  - **경로 길이 (Path Length)**
    경로를 구성하는 간선의 수를 말합니다.

그래프를 표현하는 방식

- 일반적으로 그래프는 트리 자료구조처럼 배열로 표현하는데 그 중에서도 인접 행렬과 인접 리스트라는 2가지 방식으로 나눠집니다.



- 인접 행렬 (Adjacency Matrix)

- 그래프의 관계를 2차원 배열로 구현해 표현한 방식으로 X, Y축의 인덱스는 각각의 노드를 의미하며 각 교차 지점의 값은 연결여부에 따라 0과 1로 표현합니다.
- 구현이 쉽고 특정 노드의 연결관계를 확인할 때 O(1)의 시간복잡도로 확인할 수 있다는 장점이 있지만, 간선의 개수가 정점의 개수보다 현저히 적은 상황에서도 모든 노드들의 관계를 표현해야하기 때문에 불필요하게 공간을 낭비할 수 있다는 단점이 있습니다.

	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	1
4	0	1	1	0

- 인접 리스트 (Adjacency List)

- 각 정점의 인접 정점들을 리스트 형태로 나열한 표현방식입니다.
- 실제 존재하는 간선들의 정보만 담고 있기 때문에 인접 행렬과는 달리 메모리를 덜 사용한다는 장점이 있지만, 반면에 노드간의 연결 관계를 파악하기 위해서는 정점의 리스트를 모두 파악해야 하기 때문에 정점의 차수만큼의 시간이 필요하며 구현이 비교적 어렵다는 단점이 있습니다.

- 인접 행렬과 인접 리스트의 차이점 정리

- 아래에서 간선은 e, 정점은 v로 표기하였습니다.

	인접 행렬	인접 리스트
간선 검색	O(1)	O(degree(v))
정점의 차수 계산	O(n)	O(degree(v))
전체 노드 탐색	O(n^2)	O(e)
메모리	v*v	v+e
구현	비교적 쉬움	비교적 어려움

- 두 방식은 이러한 차이가 있으므로 전체 간선의 개수나 탐색 빈도 등을 고려하여 적절한 구조를 선택해야 합니다.

그래프의 종류

- 방향성

- **방향 그래프 (Directed Graph)**
  모든 간선이 특정 방향을 가진 그래프로 해당 방향의 반대로 이동하고 싶다면 반대 방향을 가지는 간선 하나를 더 추가해주어야 합니다.
  즉, 방향 그래프에서 양방향 간선이 필요하다면 2개의 간선을 사용해야 합니다.
- **무방향 그래프 (Undirected Graph)**
  간선에 방향이 없는 그래프로 하나의 간선으로 양방향으로 움직일 수 있습니다.

> 💡 n개의 정점을 가진 무방향 그래프의 최대 간선 수는 `n(n-1)/2` 이며
> 방향 그래프의 최대 간선 수는 왕복 간선을 포함 할 경우 `2n(n-1)` 입니다.

- 연결 여부

- **연결 그래프 (Connected Graph)**
  하나의 노드로부터 모든 노드를 방문할 수 있는 그래프를 말합니다.
- **비연결 그래프 (= 단절 그래프, Disconnected Grpah)**
  최소 한 개의 노드라도 다른 노드에 닿지 못하면 비연결 그래프라고 합니다.

- 순환 여부

- **순환 그래프 (Cycle Graph)**
  Cyclic Graph라고도 하며 한 개 이상의 순환 경로(사이클)를 가진 그래프를 말합니다.
  즉, 시작 정점과 종료 정점이 같은 경로를 가지는 그래프를 말하며 상호 의존적인 관계를 나타내거나 순환적인 동작을 모델링할 때 사용합니다.
- **비순환 그래프 (Acyclic Graph)**
  그래프 내에서 어떠한 순환 경로도 포함되지 않는 그래프를 말하며 어떤 경로를 선택해도 시작 정점과 종료 정점은 항상 다른 그래프입니다.
  비순환 그래프는 일반적으로 계층적인 구조를 표현할 때 사용됩니다.

- 그 외

- **Null 그래프**
  간선이 전혀 없는 그래프를 말합니다.

- **Trivial 그래프**
  하나의 정점만 있는 그래프로 가장 작은 그래프입니다.
- **정규 그래프 (Regular Graph)**
  모든 정점의 차수가 같은 그래프를 말하며 만약 이 그래프에서 정점의 차수를 K라고 하면 K-레귤러 그래프라고 부릅니다.
  Ex) 차수가 2라면 2-Regular Graph, 차수가 4개라면 4-Regular Graph
- **완전 그래프 (Complete Graph)**
  모든 정점이 서로를 잇는 간선이 있는 그래프입니다.
  모든 정점은 서로의 인접 정점이며 가질 수 있는 최대의 간선을 가집니다.
- **이분 그래프 (Bipartite Graph)**
  그래프내의 정점들을 2개의 집합으로 나눌 수 있고 인접 정점은 서로 다른 그룹끼리만 이루어져야 하는 그래프입니다.
  자신의 그룹에 있는 정점과는 인접 정점이 될 수 없습니다.
- **가중치 그래프 (Weighted Graph)**
  간선에 가중치가 할당되어있는 그래프로 정점을 도시라고 했을 때 간선은 도로가 되며, 가중치는 비용 혹은 거리로 표현될 수 있습니다.
- **부분 그래프 (Sub Grah)**
  기존 그래프에서 일부 정점이나 간선을 제외해서 만든 그래프를 말합니다.
- **유향 비순환 그래프 (= 방향성 비순환 그래프, Directed Acyclic Graph)**
  간선에 방향성은 있지만 모두 한 방향으로 이루어져 있어 되돌아 오지 못하는 비순환 그래프를 말하며 블록체인에서 사용되는 구조 중 하나입니다.

그래프 탐색 알고리즘

- 그래프의 대표적인 탐색 알고리즘으로는 DFS와 BFS 알고리즘이 있습니다.

- DFS (Depth-First Search)

- DFS 알고리즘은 루트 노드부터 시작해 한 방향으로 리프 노드를 만날때까지 내려가 해당 분기에서 만날 수 있는 모든 노드를 접근한 뒤 다음 분기로 넘어갑니다.
- 따라서, DFS 알고리즘은 그래프의 구조 전체를 탐색해야 하거나 특정 분기의 노드들을 모두 조합하는 등의 목적일 때 유용합니다.
- DFS는 스택 혹은 재귀함수를 이용해 구현할 수 있습니다.

- 아래 코드는 재귀함수를 이용한 DFS입니다.

```
class Graph {
    constructor(v) {
        this.v = v; // 전체 노드의 개수

        // fill로 할 경우 얕은 복사로 인해 다른 인덱스의 배열이 영향을 받을 수 있음.
        this.adj = Array.from({ length: v }, () => []);
    }

    addEdge(v1, v2) {
        this.adj[v1].push(v2);
        this.adj[v2].push(v1);
    }

    dfsUtil(vertex, visited) {
        visited[vertex] = true;
        console.log(vertex);

        for (let i of this.adj[vertex].values()) {
            if (!visited[i]) {
                this.dfsUtil(i, visited);
            }
        }
    }

    dfs(startV) {
        let visited = new Array(this.v);

        for (let i = 0; i < this.v; i++) {
            visited[i] = false;
        }

        this.dfsUtil(startV, visited);
    }
}

const g = new Graph(4);

g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 3);

g.dfs(0); // 0 1 2 3
```

- BFS (Breadth-First Search)

- BFS 알고리즘도 일반적으로 루트 노드부터 시작하지만 DFS와는 다르게 현재 노드에서 방문할 수 있는 인접 노드를 모두 방문한 뒤 다음 깊이의 인접 노드를 방문해나가는 식으로 진행됩니다.
- 그래서, BFS 알고리즘은 특정 노드를 찾아야 하거나 최단 거리를 구할 때 사용됩니다.
- 일반적으로 큐 자료구조를 이용해 방문해야 할 노드를 관리합니다.

```
class Graph {
    constructor(v) {
        this.v = v; // 전체 노드의 개수

        // fill로 할 경우 얕은 복사로 인해 다른 인덱스의 배열이 영향을 받을 수 있음.
        this.adj = Array.from({ length: v }, () => []);
    }

    addEdge(v1, v2) {
        this.adj[v1].push(v2);
        this.adj[v2].push(v1);
    }

    bfs(s) {
        // 노드 방문 처리
        const visited = new Array(this.v).fill(false);

        // 순환이 필요한 노드의 목록
        const queue = [];

        visited[s] = true;
        queue.push(s);

        while(0 < queue.length) {
            s = queue.shift();
            console.log(s);

            this.adj[s].forEach((adjacent, i) => {
                if (!visited[adjacent]) {
                    visited[adjacent] = true;
                    queue.push(adjacent);
                }
            });
        }
    }
}

const g = new Graph(4);

g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 3);
```

```
g.bfs(0); // 0 1 2 3
```

- 이 외에도 경로를 탐색하는 알고리즘인 다익스트라(Dijkstra) 알고리즘, A* 알고리즘, 벨만-포드(Bellman-Ford)등이 있습니다.