그래프

작성일시	@June 21, 2023 8:30 AM
자료	https://www.geeksforgeeks.org/implementation-graph-javascript/ https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
# 주차	5

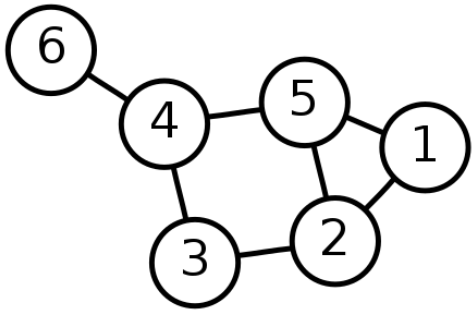
그래프(Graph)란?

https://www.codesmith.io/blog/introduction-to-graphs

- 네트워크 구조를 추상화한 모델로
- 간선(edge)으로 연결된 노드(node/ 정점 vertex)의 집합니다.
- 즉, 연결되어 있는 원소간의 관계를 표현한 자료 구조
- 비선형 자료구조

- 예) Google maps, 지하철 노선도

수학의 그래프



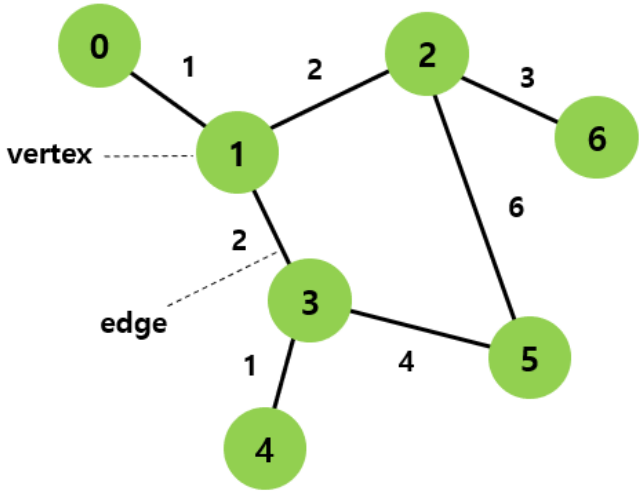
A graph with six vertices and seven edges(https://en.wikipedia.org/wiki/Graph_(discrete_mathematics))

그래프는 두 가지로 구성된다.

G = (V, E)

- V(Vertices): 정점의 집합
- E(Edges): 정점들을 연결한 간선의 집합

그래프 용어



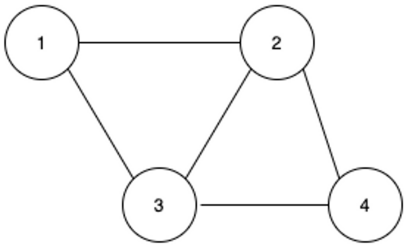
https://velog.io/@kasterra/핵심-자료구조-그래프

정점(vertex)	노드(node)라고도 하며 데이터를 저장하는 위치
간선(edge)	정점(노드)를 연결하는 선
인접 정점(adjacent vertex)	0-1 인접 정점이다. 0-2는 인접 정점이 아니다.
차수(degree)	인접 정점의 개수 • 1의 차수: 1 • 2의 차수: 3
단순 경로(simple path)	반복된 정점을 포함하지 않는 경로, 마치 한 붓그리기와 같이 같은 간선을 지나가지 않는 경로
경로 길이(path length)	경로를 구성하는데 사용된 간선의 수
사이클(cycle)	처음과 마지막 정점이 같은 단순 경로.
비사이클 그래프(acyclic graph)	사이클이 없는 그래프
연결되었다(connected)	모든 정점 간에 경로가 존재할 때 그래프가 연결되었다라고 한다.
강경합되었다(strongly connected)	두 정점이 양방향으로 경로를 갖고 있을 때 강결합되었다라고 한다.

그래프 종류

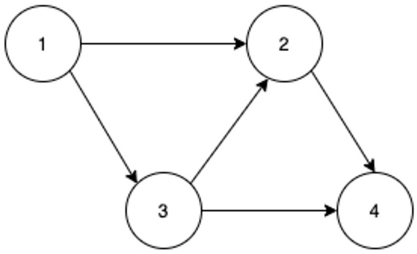
무바향 그래프(undirected graph)

두 정점을 연결하는 간선에 방향이 없는 그래프



https://80000coding.oopy.io/125156cf-79bb-48da-82ae-1f2ee7896bb8

방향 그래프(directed graph)

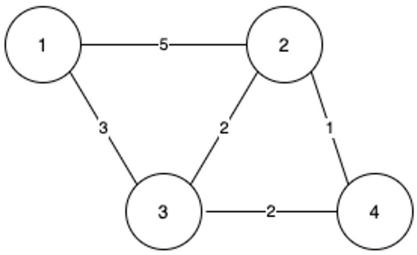


https://80000coding.oopy.io/125156cf-79bb-48da-82ae-1f2ee7896bb8

두 정점을 연결하는 간선에 방향이 존재하는 그래프

간선이 가리키는 방향으로만 이동할 수 있다.

가중치 그래프(weighted graph)

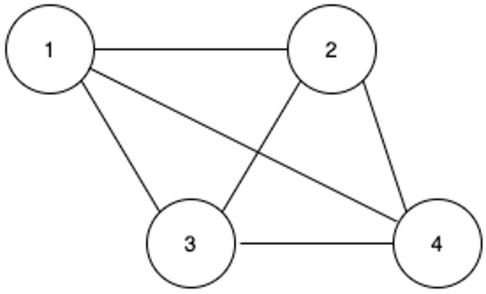


https://80000coding.oopy.io/125156cf-79bb-48da-82ae-1f2ee7896bb8

지금까지 살펴본 그래프는 가중치가 없는 그래프였는데, 위 그림처럼 가중치가 부여된 그래프도 있다.

예를 들어 ‘a 도시에서 b 도시는 5km, c에서 d도시는 4km 거리이다’를 나타내려면 간선에 가중치를 부여하면 된다.

완전 그래프(Complete graph)



https://80000coding.oopy.io/125156cf-79bb-48da-82ae-1f2ee7896bb8

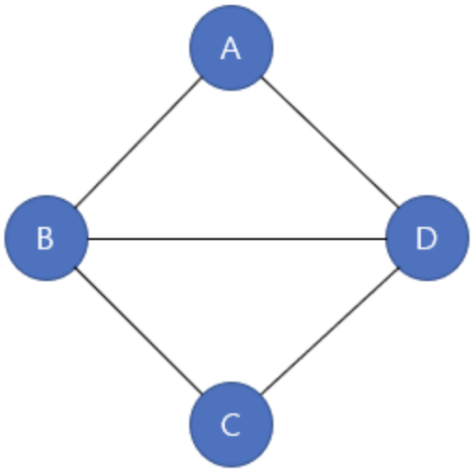
각 정점에서 다른 모든 정점들이 연결된, 최대한 많은 간선 수를 가진 그래프

- 정점이 N개인 무방향 그래프에서 최대 간선 수: `n*(n-1)/2`
- 정점이 N개인 방향 그래프에서 최대 간선 수: `n*(n-1)`

컴퓨터 과학에서는 그래프 이론을 응용해서 풀 수 있는 문제들이 많다. 그래프에서 특정한 정점, 간선, 경로 검색, 두 정점 간 최단 경로 찾기, 사이클 체크 등이 있다.

그래프 나타내기

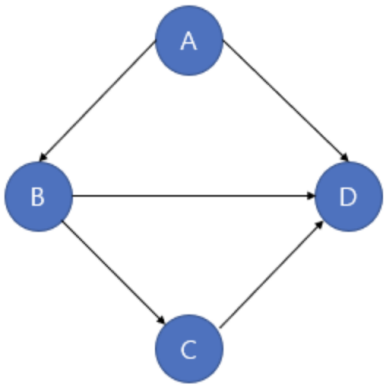
인접 행렬(adjacency matrix)



	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	1
D	1	1	1	0

무방향 그래프 인접행렬로 구현

https://code-lab1.tistory.com/13



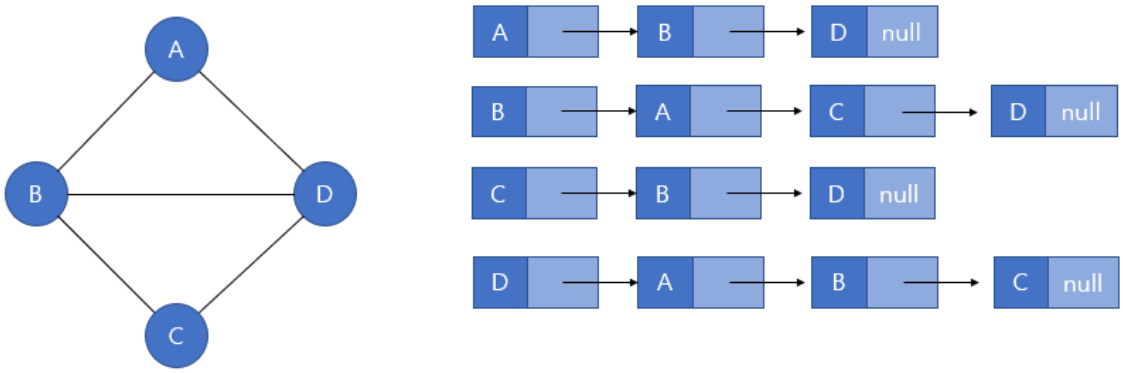
	A	B	C	D
A	0	1	0	1
B	0	0	1	1
C	0	0	0	1
D	0	0	0	0

방향 그래프 인접행렬로 구현

- 각 노드에 정수형의 배열 인덱스를 세팅한다.
- 정점 간 연결 상태는 2차원 배열의 값으로 표시하는데,
- 배열[i][j] === 1 은 인덱스 i인 노드와 인덱스 j인 노드 사이에 간선이 존재함을 의미한다.

인접 리스트(adjacency list)

인접 리스트는 각 정점별로 인접 정점들의 리스트를 저장하는데, 이를 자료 구조로 표현하는 방법은 리스트(배열), 연결 리스트, 해시 맵, 딕셔너리 중 어느 것을 채택하느냐에 따라 달라진다.



그래프 구현

이 예시는 Map을 활용하였다.

```
class Graph {
  constructor(numOfVertices) {
    this.numOfVertices = numOfVertices;
    this.AdjList = new Map();
  }
}
```

- numOfVerices: 그래프의 정점 개수 저장
- AdjList: to store an adjacency list of a particular vertex 특정 정점의 인접리스트 저장

> new Map() : 맵 생성

addVertex(v)

키로 정점을 추가하고 이것의 값을 배열로 초기화 한다.

```
// add vertex to the graph
addVertex(v) {
  // initialize the adjacent list with a
  // null array
  this.AdjList.set(v, []);
}
```

> map.set(key, value) : key를 이용해 value 저장

addEdge(src, dest)

src와 dest사이의 간선(edge)을 추가한다.

간선을 추가하기위해 먼저 src 정점에 해당하는 인접리스트를 얻어야한다. 그리고 dest를 인접 리스트에 추가한다.

```
addEdge(v, w) {
  // get the list for vertex v and put
  // the vertex w denoting edge between v and w
  this.AdjList.get(v).push(w);

  // since graph is undirected
  // add an edge from w to v also
  this.AdjList.get(w).push(v);
}
```

> map.get(key): key에 해당하는 value 반환

printGraph()

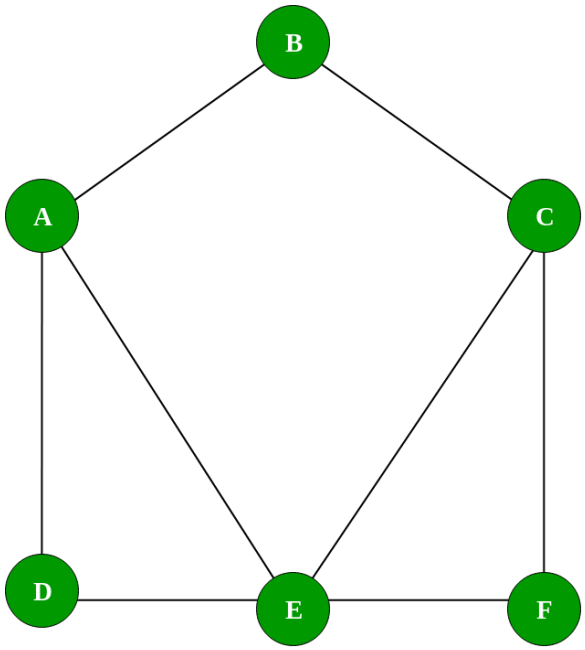
정점들과 그것의 인접리스트를 프린트한다.

```
printGraph() {
  const get_keys = this.AdjList.keys();

  for(const key of get_keys) {
    const get_values = this.AdjList.get(key);
    let cont = '';

    for(const value of get_values) {
      cont += value + ' ';
    }
```

```
    console.log(`${key} -> ${cont}`);
  }
}
```

> map.keys(): 각 요소의 키를 모은 반복 가능한(이터러블) 객체를 반환

예시




```
const graph = new Graph(6);
const vertices = ['A', 'B', 'C', 'D', 'E', 'F'];

// add vertices(맵에 키 추가)
for(let i=0; i<vertices.length; i++) {
  graph.addVertex(vertices[i]);
}

// adding edges
graph.addEdge('A', 'B');
graph.addEdge('A', 'D');
graph.addEdge('A', 'E');
graph.addEdge('B', 'C');
graph.addEdge('D', 'E');
graph.addEdge('E', 'F');
graph.addEdge('E', 'C');
graph.addEdge('C', 'F');
```

```
graph.printGraph();
// prints all vertex and
// its adjacency list
// A -> B D E
// B -> A C
// C -> B E F
// D -> A E
// E -> A D F C
// F -> E C
```

그래프 순회(graph traversal)

- Breadth First Traversal for a Graph
- Depth First Traversal for a Graph

bfs(startingNode)

startingNode로 시작해 Breadth First Search 를 수행한다.

아래 예시는 인접 행렬을 가지고 구현하였다.

```
class Graph {
  constructor(v) {
    this.V = v;
    this.adj = new Array(v).fill(0).map(el => new Array(v));
  }

  addEdge(v,w) {
    // add w to v's list
    this.adj[v].push(w)
  }
  bfs(s) {
    // mark all the vertices as not visited
    // by default set as false
    const visited = new Array(this.V);
    for(let i=0; i<this.V; i++) {
      visited[i] = false;
    }

    // create a queue for BFS
    const queue = [];

    // mark the current node as visited and enqueue it
    visited[s] = true;
```

```
    queue.push(s);

    while(queue.length >0) {
      s = queue[0];
      console.log(s+' ');
      queue.shift();

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