디지털 교재

파이썬으로 시작하는

자료구조와 알고리즘

<4주차>

NAVER CONNECT

4 - 그래프 기본

수업 참고 자료
그래프 (Graph)
그래프의 기본 요소
그래프의 유형
그래프의 표현 방법
그래프의 정의
그래프의 활용
그래프의 종류
1. 무향 그래프(undirected graph) vs 방향 그래프(directed graph)
2. 단순 그래프 vs 다중 그래프
3. 가중치 그래프
그래프의 구현
1. 인접 행렬
2. 인접 리스트
3. 암시적 그래프
그래프의 순회
Breadth First Search (BFS)
Depth First Search (DFS)

수업 참고 자료

📌 **실전 예시**

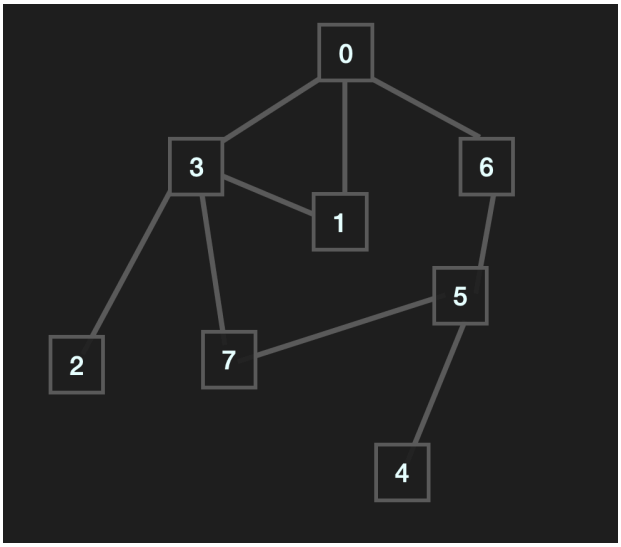
인접 행렬

인접 리스트 1

인접 리스트 2

암시적 그래프

- 그래프 직접 구현해보기




```
graph = [
    [0, 1, 0, 1, 0, 0, 1, 0],
    [1, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 1, 1],
    [1, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0, 1, 0, 0]
]
graph = {
 0: [1, 3, 6],
 1: [0, 3],
 2: [3],
 3: [0, 1, 2, 7],
 4: [5],
 5: [4, 6, 7],
 6: [0, 5],
 7: [3, 5]
}

graph = [
 0: [1, 3, 6],
 1: [0, 3],
 2: [3],
 3: [0, 1, 2, 7],
 4: [5],
 5: [4, 6, 7],
 6: [0, 5],
```

```
 7: [3, 5]
]
```

- BFS

  1. 기본형 그래프

```
def bfs(graph, start_v):
    # 초기 설정
    q = deque()
    visited = [False]*len(graph)
    # 시작점 예약
    q.append(start_v)
    visited[start_v] = True

    while q:
        # 방문
        cur_v = q.popleft()
            # *---------------------------------------*
                # 그래프를 방문하며 처리해야할 일을 여기서 한다
                # (예시)
              # if cur_v == value:
                #     << 현재 노드가 특정 값을 만족할 때 해야할 일 >>
                #      return cur_v
            # *---------------------------------------*

        # 다음 노드 예약
        for next_v in graph[cur_v]:
            if not visited[next_v]:
                q.append(next_v)
                visited[next_v] = True
```

  2. 그리드형 그래프

```
from collections import deque

def bfs(grid):
    def isValid(r, c):
        return (
            (r >= 0 and r < row_len)
            and (c >= 0 and c < col_len)
            and grid[next_r][next_c] == 1
```

```
        )
    row_len, col_len = len(grid), len(grid[0])
    visited = [[False] * col_len for _ in range(row_len)]
    dr = [0, 1, 0, -1]
    dc = [1, 0, -1, 0]

    queue = deque()
    queue.append(0, 0)
    visited[0][0] = True
    while queue:
        cur_r, cur_c = queue.popleft()
        # *---------------------------------------*
        # 그래프를 방문하며 처리해야할 일을 여기서 한다
        # (예시)
      # if cur_v == value:
        #     << 현재 노드가 특정 값을 만족할 때 해야할 일 >>
        #     return cur_v
        # *---------------------------------------*
        for i in range(4):
            next_r = cur_r + dr[i]
            next_c = cur_c + dc[i]
            if isValid(next_r, next_c):
                if not visited[next_r][next_c]:
                    queue.append((next_r, next_c))
                    visited[next_r][next_c] = True
```

- DFS

  1. 기본형 그래프

```
visited = {}
def dfs(cur_v):
    # 방문
    visited[cur_v] = True
    print(cur_v)
    for next_v in graph[cur_v]:
        if next_v not in visited:
            dfs(next_v)
dfs(3)
```

```
visited = {}
depth = 0
def dfs(cur_v):
```

```
    global depth
    # 방문
    visited[cur_v] = True
    print('   ' * depth, f"ㄴ{cur_v}")
    depth += 1
    for next_v in graph[cur_v]:
        if next_v not in visited:
            dfs(next_v)
    depth -= 1

dfs(3)
```

2. 그리드형 그래프

```
# DFS
grid = [[0,0,1],[1,0,1],[1,0,0],[0,1,0]]
row_len, col_len = len(grid), len(grid[0])
visited = [[False] * col_len for _ in range(row_len)]

def dfs(grid, r, c):
    # *---------------------------------------*
    # 그래프를 방문하며 처리해야할 일을 여기서 한다
    # (예시)
    # << 현재 노드가 특정 위치에 도달했을 때 해야할 일 >>
    # if r == row_len - 1 and c == col_len - 1:
    #     print('도착')
    # *---------------------------------------*
    dr = [0, 1, 0, -1]
    dc = [1, 0, -1, 0]
    visited[r][c] = True
    for i in range(4):
        next_r = r + dr[i]
        next_c = c + dc[i]
        if 0 <= next_r < row_len and 0 <= next_c < col_len:
            if grid[next_r][next_c] == 0 and not visited[next_r][next_c]:
                dfs(grid, next_r, next_c)

dfs(grid, 0,0)
```

그래프 (Graph)

그래프(Graph)는 객체 간의 연결 관계를 표현하는 자료구조입니다. 그래프는 노드(Node) 또는 정점(Vertex)과 이를 연결하는 간선(Edge)으로 구성됩니다. 그래프는 연결된 객체 간의 관계를 모델링하기 위해 널리 사용되며, 다양한 형태와 유형이 있습니다. 아래는 그래프의 주요 특징과 유형에 대한 설명입니다.

그래프의 기본 요소

- **노드(Node) / 정점(Vertex):** 그래프를 구성하는 개별 요소입니다. 객체나 위치 등을 나타낼 수 있습니다.
- **간선(Edge):** 노드를 연결하는 선입니다. 두 노드 간의 관계를 나타냅니다. 간선은 방향성이 있거나 없을 수 있습니다.

그래프의 유형

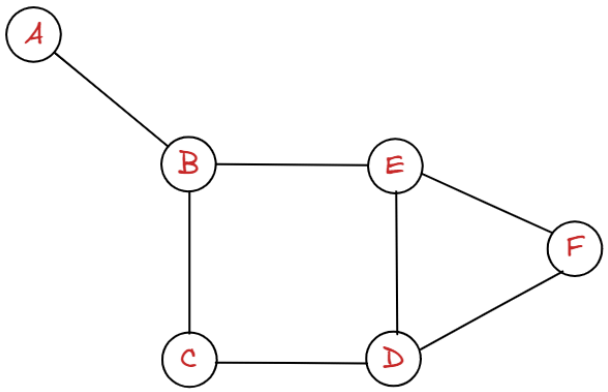
1. **무방향 그래프(Undirected Graph):** 간선에 방향성이 없는 그래프입니다. 두 노드는 양방향으로 연결됩니다.
2. **방향 그래프(Directed Graph):** 간선에 방향성이 있는 그래프입니다. 간선이 A에서 B로 향하면, A에서 B로만 이동할 수 있습니다.
3. **가중치 그래프(Weighted Graph):** 간선마다 가중치가 할당된 그래프입니다. 가중치는 거리, 비용 등 다양한 메트릭을 나타낼 수 있습니다.
4. **비가중치 그래프(Unweighted Graph):** 모든 간선이 동일한 가중치(또는 가중치 없음)를 가진 그래프입니다.
5. **연결 그래프(Connected Graph):** 무방향 그래프에서 모든 노드 쌍 사이에 경로가 존재하는 그래프입니다.
6. **비연결 그래프(Disconnected Graph):** 하나 이상의 노드가 다른 노드와 연결되어 있지 않은 그래프입니다.
7. **순환 그래프(Cyclic Graph):** 적어도 하나의 노드 사이클(노드를 시작점으로 하여 한 번 이상 방문하고 시작점으로 돌아오는 경로)이 존재하는 그래프입니다.
8. **비순환 그래프(Acyclic Graph):** 순환(cycle)이 없는 그래프입니다. 방향 비순환 그래프(DAG, Directed Acyclic Graph)는 특히 중요하며, 여러 알고리즘과 자료구조에서 사용됩니다.

그래프의 표현 방법

1. **인접 행렬(Adjacency Matrix):** 2차원 배열을 사용해 노드 간의 연결 관계를 표현합니다. 배열의 각 요소는 노드 쌍의 연결 상태(간선의 유무 또는 가중치)를 나타냅니다.
2. **인접 리스트(Adjacency List):** 각 노드에 연결된 노드 목록을 저장하는 방법입니다. 동적 배열, 연결 리스트 등으로 구현할 수 있습니다.

그래프의 정의

그래프(G)는 정점(vertex)들의 집합 V와 이들을 연결하는 간선(edge)들의 집합 E로 구성된 자료구조입니다.



위와 같은 그래프가 있다고 할 때, 정점은 A, B, C, D, E, F 입니다.

또 정점들을 연결하는 간선들은 A-B, B-C, B-E, C-D, E-D, E-F, E-D 입니다.

즉, 위의 그래프는 정점들의 집합 $V = \{A, B, C, D, E, F\}$와 이들을 연결하는 간선들의 집합 $E = \{(A, B), (B, C), (B, E), (C, D), (E, D), (E, F), (E, D)\}$로 이루어져 있습니다.

그래프의 활용

그래프는 이렇듯 연결 관계를 표현하기에 현실 세계의 사물이나 추상적인 개념들을 잘 표현할 수 있습니다.

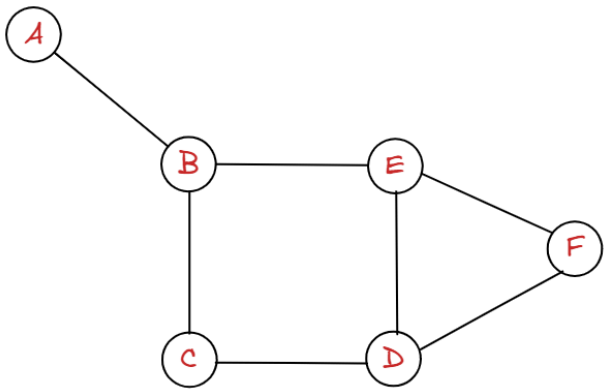
- 도시들을 연결하는 도로망: 도시(vertex), 도로망(edge)
- 지하철 연결 노선도: 정거장(vertex), 정거장을 연결한 선(edge)
- 컴퓨터 네트워크: 각 컴퓨터와 라우터(vertex), 라우터 간의 연결 관계(edge)
- 소셜 네트워크 분석: 페이스북의 계정(vertex), follow 관계(edge)

그래프의 종류

위와 같은 기본적인 그래프에서 시작하여, 다양한 그래프를 살펴볼 수 있습니다.

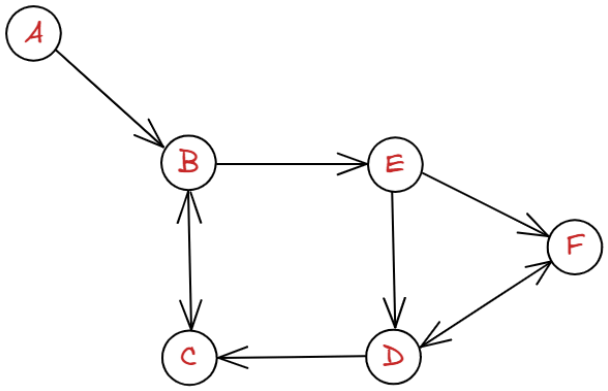
1. 무향 그래프(undirected graph) vs 방향 그래프(directed graph)

앞서서 다룬 그래프를 다시 살펴보도록 합시다.



위와 같은 그래프는 A→B로 갈 수 있으며, B→A로도 갈 수 있기에 **무향 그래프(undirected graph)**라고 합니다. 따로 방향이 안 정해져 있기 때문입니다.[(1)]

한편 다음과 같이 간선의 방향이 정해져 있는 그래프도 있습니다.

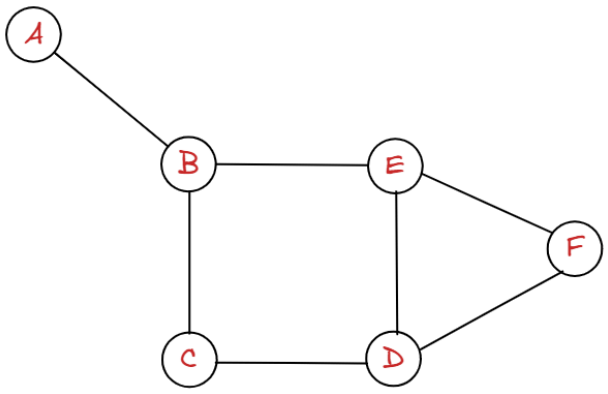


위와 같이 방향이 정해져 있는 그래프를 방향 그래프(directed graph)라 합니다.

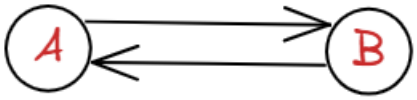
그래프를 잘 살펴보면 들어오는 간선과 나가는 간선으로 구분할 수 있습니다. 이때 들어오는 간선을 **indegree**라 하며, 나가는 간선을 **outdegree**라 합니다. 정점 B를 살펴보면 A와 C로부터 들어오는 **indegree** 2개, E로 나가는 **outdegree** 1개가 있는 것을 알 수 있습니다.

2. 단순 그래프 vs 다중 그래프

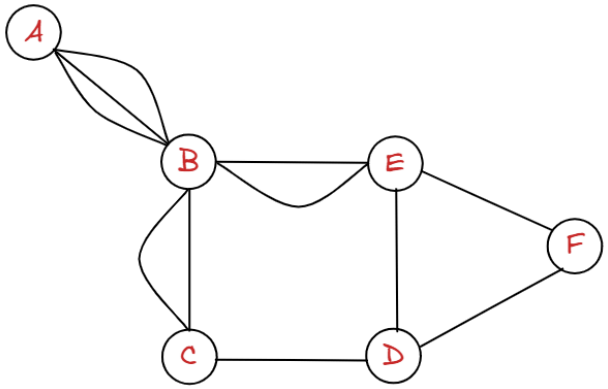
다시 그래프를 보도록 하겠습니다.



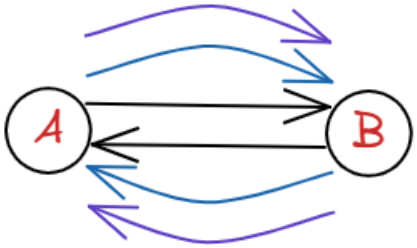
인접해 있는 두 정점과의 관계를 살펴보도록 합시다. $(A, B), (B, C), (B, E), (E, D) \cdots$ 등이 있을 것입니다. 이때 두 정점 사이에 indegree가 1개, outdegree가 1개인 그래프를 **단순 그래프(simple graph)** 라고 합니다.[2]



하지만 다음과 같이 여러 길이 존재 할 수 도 있습니다.



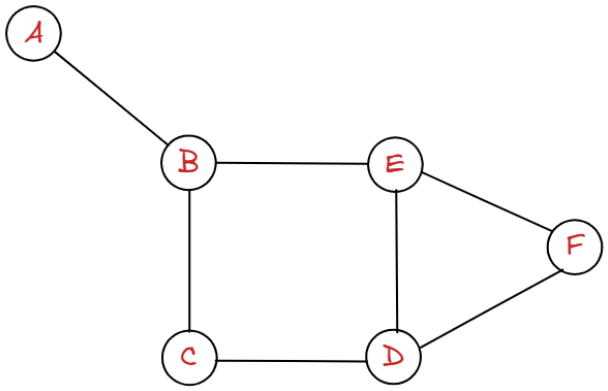
A에서 B까지 가는 방법이 총 3개 있습니다. 다시 말해, 인접해 있는 두 정점 A,B 사이에 A를 기준으로 ingree가 3개 outdegree가 3개가 있습니다.



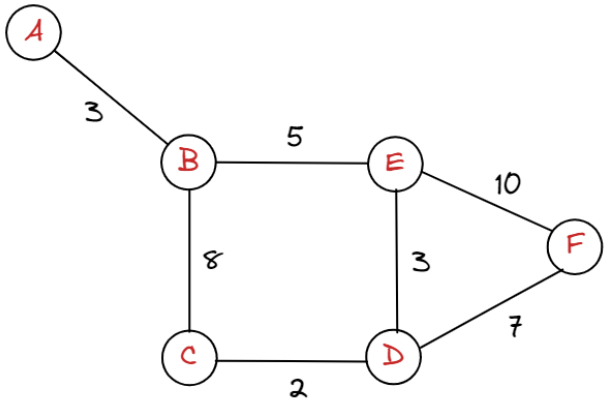
이렇게 인접해 있는 두 정점 A,B의 관계에서 outdegree와 indegree가 2개 이상인 그래프를 **다중그래프 (Multigraph)** 라고 합니다.

3. 가중치 그래프

다시 그래프를 보도록 하겠습니다.



현재 이 그래프에서는 A에서 B로 , B에서 E로 등 **모든 경로의 비용(시간)이 동일하다**고 가정했습니다. 하지만, 아래와 같이 경로마다 비용을 다르게 설정할 수 있습니다. 각 경로 마다 가중치가 다르다 하여 아래와 같은 그래프를 **가중치 그래프(Weighted Graph)** 라고 정의합니다.[(3)]



그래프의 구현

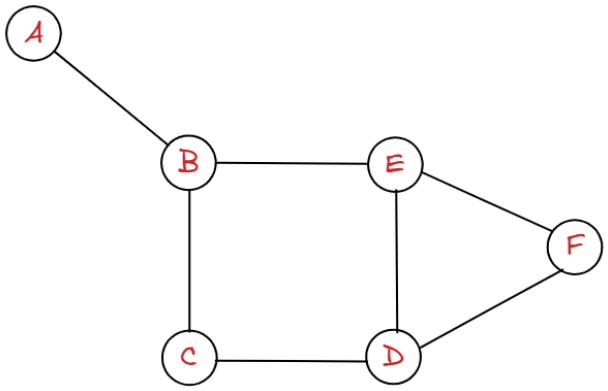
여러 가지 그래프를 보았으니, 이제 그래프를 구현하도록 하는 시간을 갖겠습니다. 그래프를 구현하는 방법으로는 총 3가지가 있습니다.

1. **인접 행렬**
2. **인접 리스트**
3. **암시적 그래프**

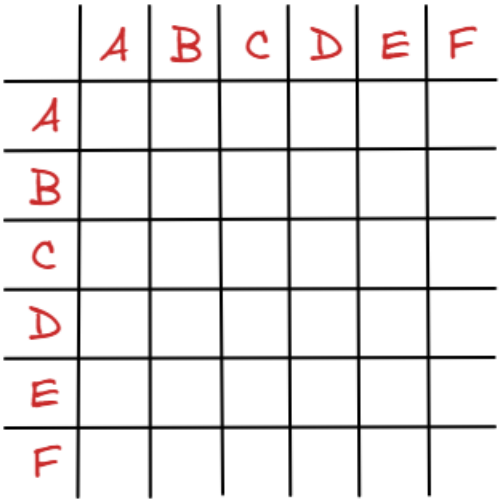
각 그래프를 구현 하는 방법을 알아보도록 하겠습니다.

1. 인접 행렬

행렬은 행(row)과 열(column)에 따라, 정보들을 직사각형 모양으로 배열한 것입니다.



다음 그래프가 있을 때, 각 정점들을 하나의 행(가로)과 열(세로)이라 생각할 수 있습니다.



와 같이 말이죠. 그 후 그래프의 연결 관계를 생각해 볼 수 있습니다. 정점끼리 연결되어 있으면 `1` ,연결되어 있지 않으면 `0`을 대입해 보도록 합시다.

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	0	0
D	0	0	1	0	1	1
E	0	1	0	1	0	1
F	0	0	0	1	1	0

흔히 행렬을 만들기 위해서는 이중리스트를 사용합니다.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

위와 같은 matrix가 있을 때 코드로 표현하면 다음과 같습니다.

```
matrix = [[1, 2, 3], [4, 5, 6]]
```

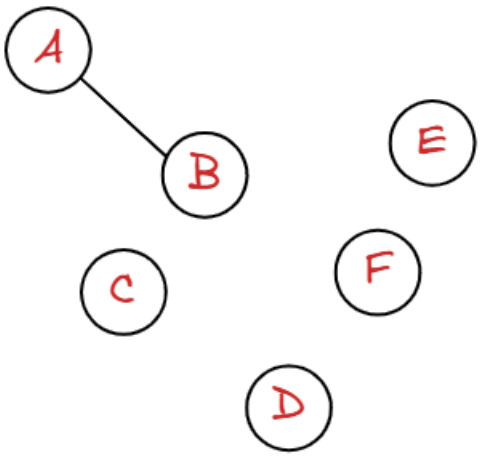
인접행렬 역시 마찬가지입니다. 다만, 리스트에서 `A,B ...` index는 존재 하지 않기에 `A`를 `0`, `B`를 `1`과 같이 생각해 줍니다.

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	0	0
D	0	0	1	0	1	1
E	0	1	0	1	0	1
F	0	0	0	1	1	0

```
matrix = [
    [0, 1, 0, 0, 0, 0],
    [1, 0, 1, 0, 1, 0],
    [0, 1, 0, 1, 0, 0],
    [0, 0, 1, 0, 1, 1],
    [0, 1, 0, 1, 0, 1],
    [0, 0, 0, 1, 1, 0],
]
```

위 그래프에서 특징적인 점은 **자기 자신(ex A-A, B-B)을 연결하는 간선이 없기** 때문에, 대각선의 값이 모두 `0`이라는 점과 **무향 그래프**이기 때문에 대각선을 기준으로 대칭입니다.

이제 저희는 모든 그래프에 대해서 인접 행렬을 그릴 수 있습니다. 하지만 인접 행렬이 만사는 아닙니다. 다음과 같이 정점은 엄청 많은데, 간선의 개수가 적을 때는 비효율적입니다.



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	0	0	0	0
C	0	0	0	0	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

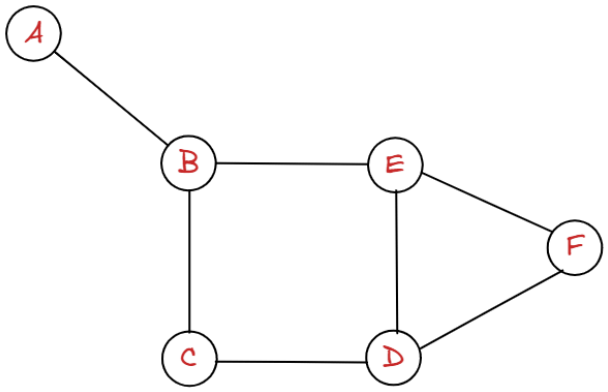
다음과 같이 A-B의 관계를 제외하고는 모두 `0`으로 나타내기 때문에, 메모리 사용 측면에서 비효율적이라고 할 수 있습니다. 그래프는 연결 관계를 나타내는 것이 중요한데, `0`은 정보로서 가치가 떨어집니다.

인접 리스트를 사용하면, 위와 같은 문제를 해결할 수 있습니다.

2. 인접 리스트

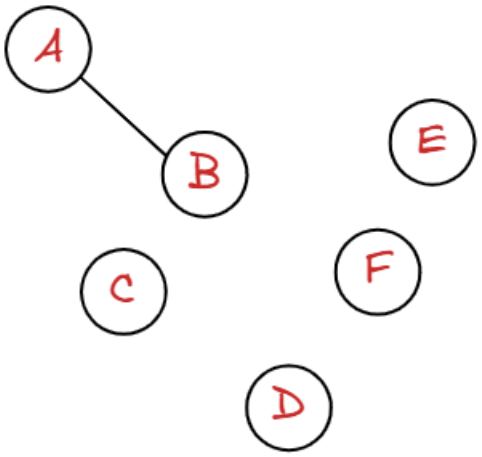
인접 리스트는 딕셔너리 `{}`의 형태로 정의됩니다. 딕셔너리는 `key`와 `value`로 정의되는 자료구조입니다. 인접 리스트 같은 경우, `key`에는 정점들이 들어가며, `value`에는 list 형태로 연결 관계를 표시해 줍니다.

실제 예시를 봐보도록 합시다.




```
graph = {
   "A": ["B"],
   "B": ["A", "C", "E"],
   "C": ["B", "D"],
   "D": ["C", "E", "F"],
   "E": ["B", "D", "F"],
   "F": ["D", "E"],
}
```

위에서 인접 행렬 같은 경우, 정점들이 많고, 간선이 적은 경우 대부분의 정보가 `0`으로 채워지기 때문에 비효율적이라고 하였습니다.



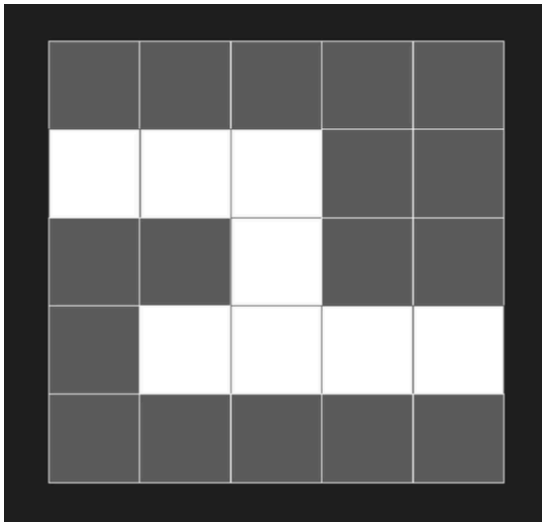
	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	0	0	0	0
C	0	0	0	0	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

하지만 인접 리스트를 사용하게 되면, 이를 간편하게 표현할 수 있습니다.[4]

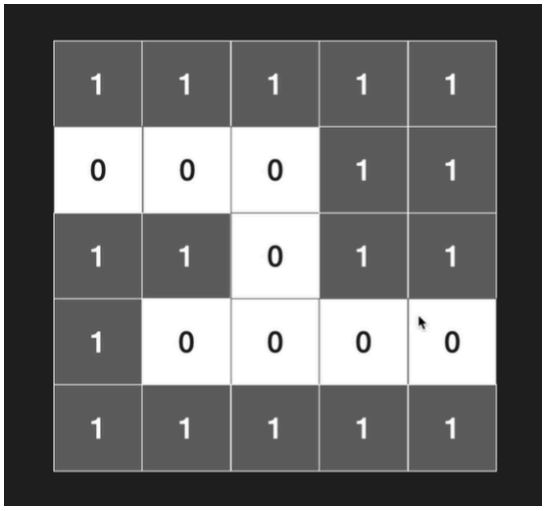
```
graph = {
   "A": ["B"],
   "B": ["A"],
   "C": [],
   "D": [],
   "E": [],
   "F": [],
}
```

3. 암시적 그래프

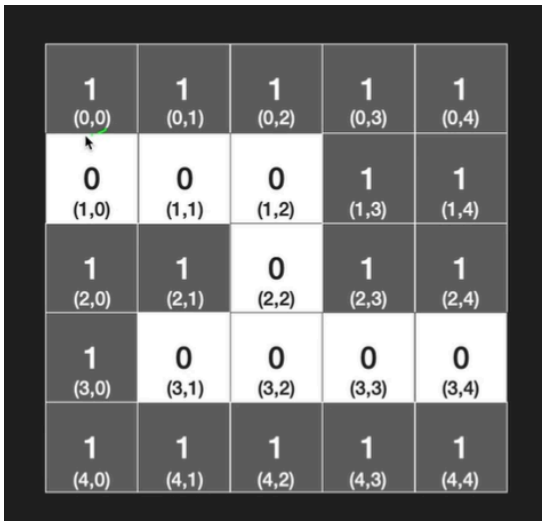
코딩 테스트에서 제일 많이 활용되는 **암시적 그래프**를 다루어 보도록 하겠습니다. 그래프 문제를 풀다 보면 다음과 같은 문제들을 자주 접해 볼 수 있습니다.



다음과 같이 흰색이 길이고, 검은색이 벽인 미로가 있다고 합시다. 전혀 그래프 같지 않지만, 암시적으로 그래프처럼 표현할 수 있습니다.



벽에는 `1`의 값을, 길에는 `0`의 값을 넣음으로써, 구분해 보도록 합시다. 그리고 각 영역을 좌표의 개념을 도입하여 표현할 수 있습니다.




```
graph = [
    [1, 1, 1, 1, 1],
    [0, 0, 0, 1, 1],
    [1, 1, 0, 1, 1],
    [1, 0, 0, 0, 0],
    [1, 1, 1, 1, 1],
]
```

위의 표현 방식 같은 경우, 연결 관계를 직접적으로 나타내지는 않았습니다.

하지만 상하좌우가 연결되어 있다는 것을 암시적으로 알 수 있습니다.

예를 들어 (1, 2)는 상(0, 2), 하(2, 2), 좌(1, 1), 우(1, 3)가 연결 되어있다는 것을 말이죠.

저희는 그래프의 값들을 `row`와 `col`이라는 변수명을 통해 값들에 접근할 것입니다.

	col0	col1	col2	col3	col4
row 0	(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)
row 1	(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)
row 2	(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)
row 3	(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)
row 4	(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)

가로를 `row`로 세로를 `col`로 생각하는 것입니다. `graph[row][col]`를 통해 원하는 값에 접근할 수 있게 됩니다. 즉, (2, 3)에 있는 값을 알고 싶으면, `graph[2][3]`을 통해 알 수 있습니다.

이제 그래프를 구현할 수 있으니, 순회하는 방법을 알아보도록 합시다.

그래프의 순회

그래프의 순회는 트리의 순회와 마찬가지로, 모든 정점을 지나야 합니다. 대표적으로 2가지 방법이 있는데 BFS와 DFS입니다.

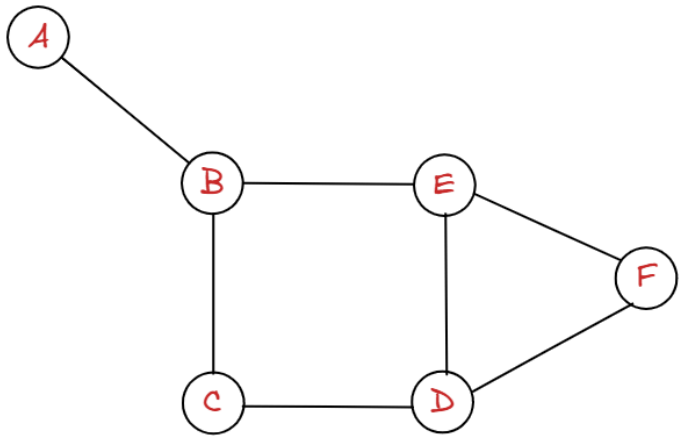
Breadth First Search (BFS)

6강에서 배운 트리의 순회인 level order traversal과 매우 유사합니다. 그래프 순회를 할 때, 시작점이 주어질 텐데, 이를 루트 노드라 생각하여, level별로 탐색하는 것이 BFS입니다.

```
from collections import deque

def bfs(graph, start_v):
    visited = [start_v]
    queue = deque(start_v)
    while queue:
        cur_v = queue.popleft()

            #해야할일을 여기서 한다
            #if cur_v == value:
            #  return cur_v

        for v in graph[cur_v]:
```

```
            if v not in visited:
                visited.append(v)
                queue.append(v)
    return visited
```

다음과 같은 그래프를 A에서 시작하여 탐색해 보도록 하겠습니다.




```
graph = {
    "A": ["B"],
    "B": ["A", "C", "E"],
    "C": ["B", "D"],
    "D": ["C", "E", "F"],
    "E": ["B", "D",  "F"],
    "F": ["D", "E"],
}
```

시작점은 A라고 가정하겠습니다.

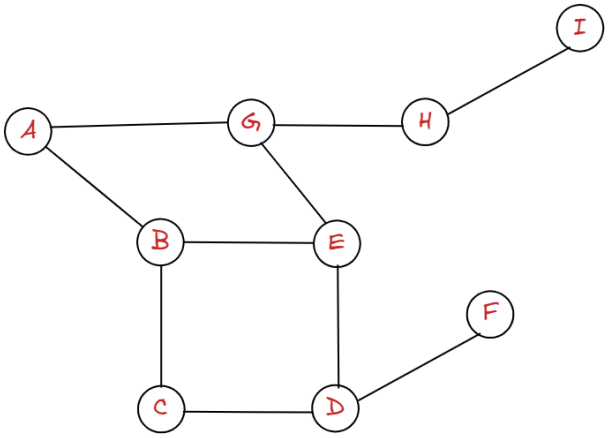
https://prod-files-secure.s3.us-west-2.amazonaws.com/4ff76b2c-f77b-4aec-b3a7-852800e015b4/168e9337-efdf-4899-8e55-9766463d4078/Graph_2._BFS.m4v

BFS의 코드 구현과 작동 원리에 대해서 알아보았습니다. tree의 level order traversal과 상당히 비슷함으로, 이 부분을 잘 공부했다면 큰 어려움 없이 따라서 오셨을 겁니다. BFS 코드는 아주 기본적인 템플릿 코드로서, 머릿속에서 바로

바로 구현될 수 있어야 합니다!

Depth First Search (DFS)

DFS는 출발점에 시작해서, 막다른 지점에 도착할 때까지 깊게 이동합니다. 만약 가다가 막히면 다시 그 전 노드로 돌아가고, 또 길이 있으면 깊게 이동하는 식의 과정을 통해 그래프를 순회할 수 있습니다.

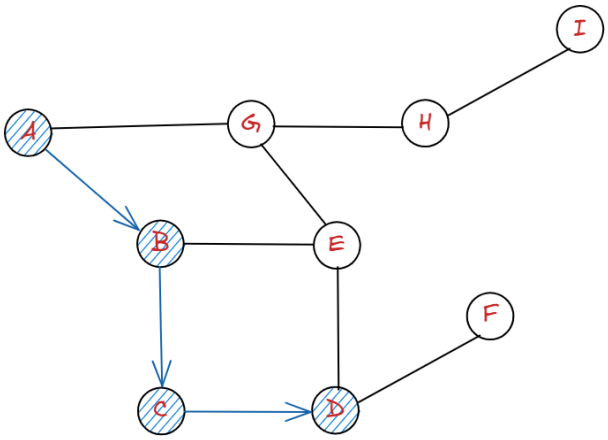


예를 들어 다음과 같은 그래프가 있다고 할 때 A부터 순회를 해보도록 해봅시다. 여러분들도 한 번 해보세요!

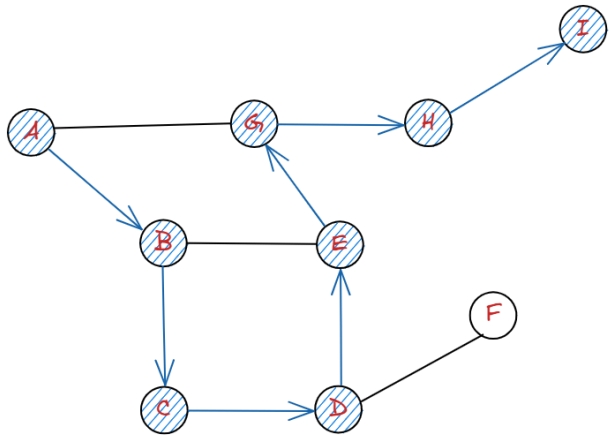
▼ 답은 !!?

A-B-C-D-E-G-H-I-F 입니다.

왜 그런 지 한 번 봐 볼까요?



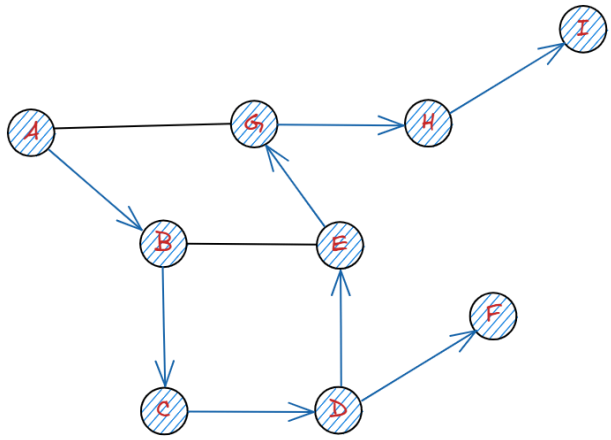
우선 깊게 깊게 파고들어 D까지 이동합니다. 그 이후로 알파벳 순서를 기준으로 E를 먼저 가도록 합시다.



I까지 간 후 이제 막혀 버렸습니다. 이제 왔던 길을 되찾는 과정을 거칩니다.

H, G, E, D 순으로 올라갑니다.

그 후 아직 탐색하지 않는 F를 방문해 줍니다.



F 이후 마찬가지로 갈 길이 없습니다.

왔던 길인 D, C, B, A 순으로 올라감으로써, 탐색을 마무리합니다.

이제 탐색 방법을 알았으니, DFS 구현을 알아볼 차례입니다. DFS는 스택과 재귀로 구현할 수 있습니다. 저희는 구현이 쉬운 재귀로 구현한 방법을 알아보겠습니다.

```
visited = []

def dfs(cur_v):
    visited.append(cur_v)
    for v in graph[cur_v]:
```

```
        if v not in visited:
            dfs(v)
```

tree의 preorder, inorder, postorder의 순회 방식과 매우 유사합니다.

https://prod-files-secure.s3.us-west-2.amazonaws.com/4ff76b2c-f77b-4aec-b3a7-852800e015b4/b966c50c-5235-40a5-a3e3-218101b914af/Graph_3._DFS.m4v

이런 식으로 DFS의 순회 과정을 알아보았습니다. 재귀에 익숙하지 않으면 이해하기 힘든 과정일 수 있습니다. 하지만 코드 작성 자체는 어렵지 않으므로, 위의 과정을 반복하여, 최종적으로 머릿속에서 그려지는 것이 익숙해지면 좋을 것 같습니다.

---

1. 코딩테스트에서는 주로 무향 그래프(undirected graph)를 다룹니다.

2. 코딩 테스트에서는 주로 단순 그래프(simple graph)를 다룹니다.

3. 가중치 그래프는 추후 **다익스트라** 알고리즘을 배울 때 더 자세히 알아보도록 하겠습니다.

4. 코딩 테스트에서는 주로, 인접 행렬보다는 **인접리스트**를 사용합니다.