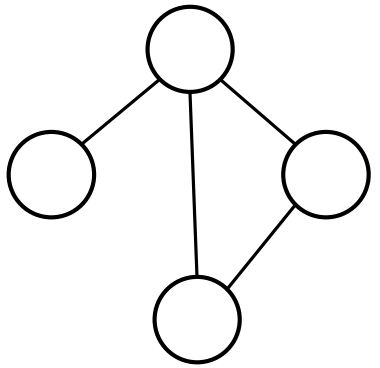
Graph (그래프)

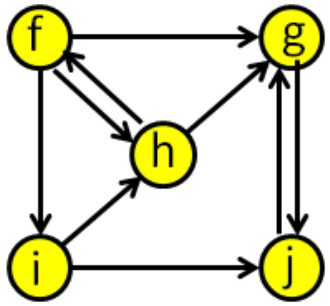
그래프

- 정점(vertex)과 간선(edge)으로 구성된 자료구조
  - 정점은 보통 객체, 데이터 아이템 등을 나타냄
  - 간선은 정점 간의 관계를 표현
  - $G = (V, E)$
- 예 (정점과 간선?)
  - SNS에서 사람들의 관계
  - 교통망 (도시를 연결하는 도로), 지하철 노선도
  - 신경망
  - 교과목 이수체계

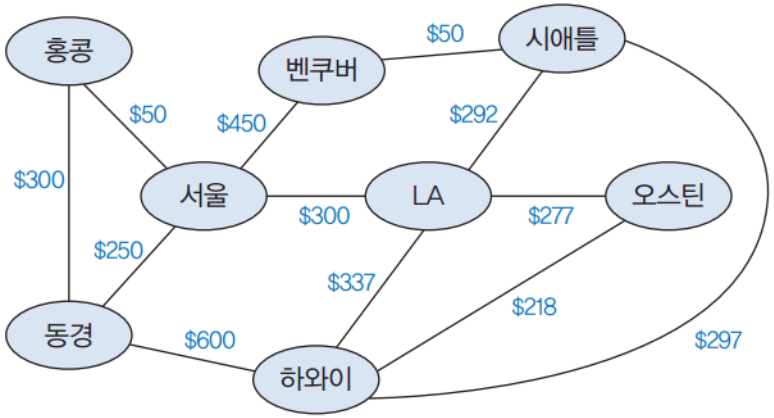


그래프

- 종류
  - 무방향(undirected) 그래프
    - 간선의 방향성이 없음
  - 방향(directed) 그래프
    - 간선의 방향성이 있음
    - 교과목이수체계, 신경망, …
  - 가중치(weighted) 그래프
    - 간선에 비용 또는 가중치가 할당된 그래프
    - 교통망, …



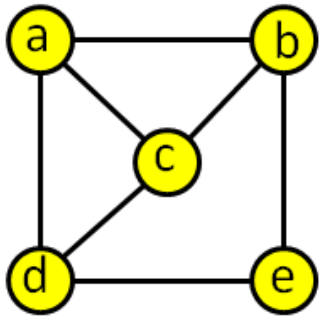
방향(directed) 그래프

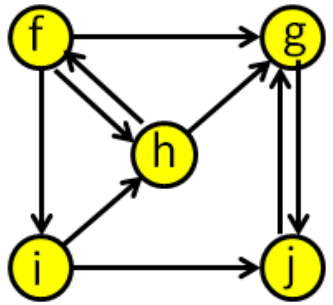


가중치 그래프

용어

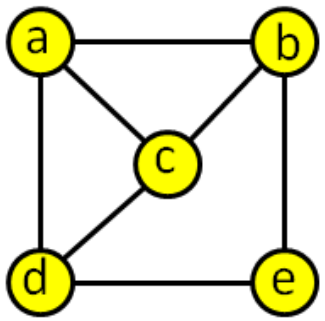
- 차수(Degree)
  - 타겟 정점과 간선으로 연결된 정점의 수
  - a의 차수는?
- 진입(In-degree) 차수
  - 방향 그래프에서 정점으로 들어오는 간선의 수
- 진출(Out-degree) 차수로 구분
  - 방향 그래프에서 정점에서 나가는 간선의 수





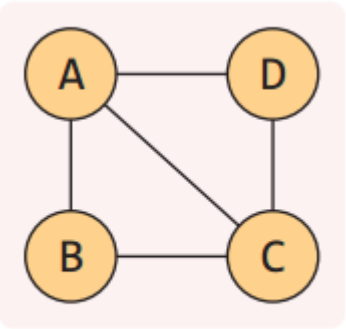
용어

- 경로(Path)
  - 시작 정점부터 도착 정점까지의 정점들을 나열하여 표현
  - [a, c, b, e]: 정점 a로부터 도착점 e까지의 여러 경로들 중 하나
- 단순 경로
  - 경로 상의 정점들이 모두 다른 경로
- 싸이클(Cycle)
  - 시작 정점과 도착 정점이 동일한 경로
  - [a, b, e, d, c, a]
- 트리(Tree): 싸이클이 없는 그래프
- 신장(spanning) 트리
  - 그래프의 모든 정점을 싸이클 없이 연결하는 부분 그래프



그래프의 저장 (표현)

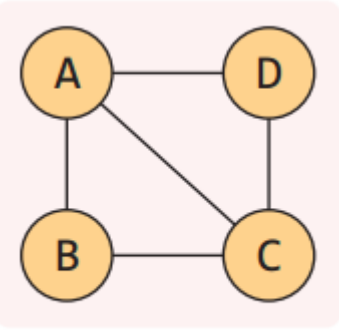
- 그래프의 표현
  - V(G1) = {A, B, C, D}
  - E(G1) = {(A, B), (A, C), (A, D), (B, C), (C, D)}
- 그래프의 저장
  - 인접 행렬 (adjacency matrix)
  - 인접 리스트 (adjacency list)



G1

그래프의 저장 (표현)

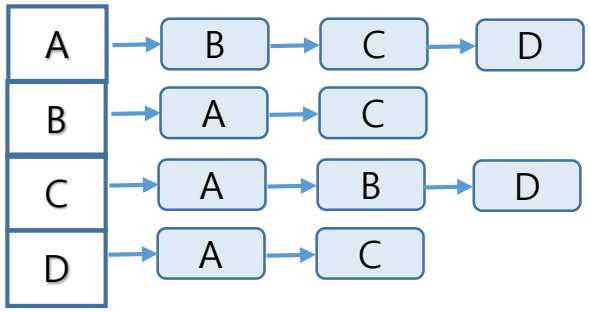
- 인접 행렬 (무방향 그래프 예)
  - 2차원 배열 활용
  - 행 정점과 열 정점을 연결하는 간선이 있으면 해당 배열 원소의 값이 1 또는 가중치, 아니면 0
  - vertex = ['A' , 'B' , 'C' , 'D']
  - adjMatx = [[0, 1, 1, 1],
    [1, 0, 1, 0],
    [1, 1, 0, 1],
    [1, 0, 1, 0]]
  - 정점의 수가 V라면 V*V 크기의 배열 필요함
  - 메모리 사용량 측면에서 간선의 수가 많은 조밀한 (dense) 그래프에 유리

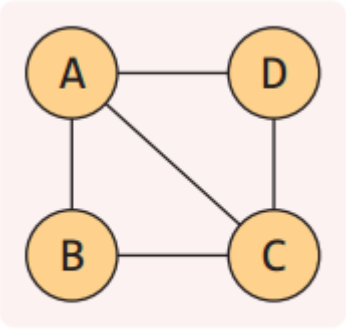


G1

그래프의 저장 (표현)

- 인접 리스트 (무방향 그래프 예)
  - 여러 연결 리스트 활용
  - 각 정점과 연결된 정점 정보를 리스트로 저장





G1

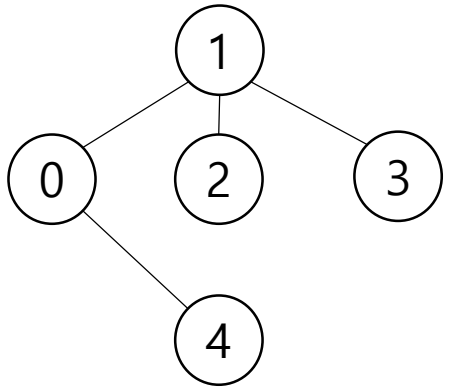
  - 정점의 개수가 V라면 V개의 연결 리스트가 필요하며, 전체 간선의 수가 E라면 2*E개의 노드가 필요함.
  - 메모리 사용량 측면에서 정점에 비해 간선의 개수가 적은 희소 (sparse) 그래프에 적합

그래프의 탐색

- 깊이우선탐색 (DFS)
- 너비우선탐색 (BFS)

깊이우선탐색 (DFS)

- 알고리즘
  1. 임의의 정점을 시작 정점으로 하여 방문 시작
  2. 현재 정점의 이웃 정점 중, 아직 방문하지 않은 정점 중 하나를 방문
  3. 새로운 정점을 출발점으로 하여 2번 반복하되, 이웃 정점이 없으면 직전 정점으로 돌아가 2번 반복
  4. 모든 정점을 반복하면 완료.

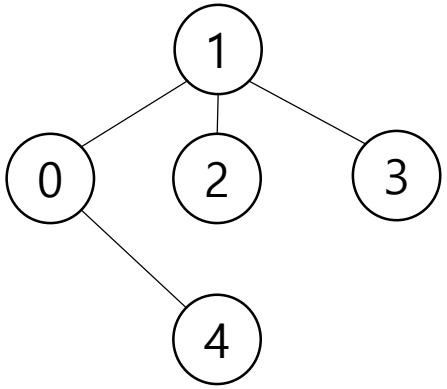


- 방문 가능한 정점들이 여러 개인 경우 낮은 숫자의 정점을 먼저 방문한다고 가정하면,
  - 0 -> 1 -> 2 -> 3 -> 4 순서로 방문

DFS 구현 – 1단계

- 그래프의 표현 (인접리스트)

```
graph_adjlist = [[1, 4], [0, 2, 3], [1], [1], [0]]

graph = Graph(graph_adjlist)
graph.dfs(0)
```



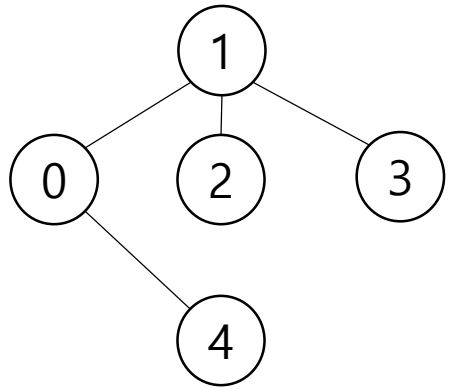
DFS 구현 – 2단계

- Graph 클래스 구현
  - 멤버변수: graph (인접 리스트), vertex_count, visited (정점 별 방문 여부)
  - 메서드: 생성자, dfs()

```
class Graph:
    def __init__(self, graph):
        self.graph = graph
        self.vertex_count = len(self.graph)
        self.visited = [False] * self.vertex_count

    def dfs(self) :
        …
```

DFS 구현 – 2단계

- dfs() 구현
  1. 임의의 정점을 시작 정점으로 하여 방문 시작
  2. 현재 정점의 이웃 정점 중, 아직 방문하지 않은 정점 중 하나를 방문
  3. 2번 반복하되, 이웃 정점이 없으면 직전 정점으로 돌아가 2번 반복
  4. 모든 정점을 방문하면 완료.



⇒ 방문한 정점들의 정보를 관리해야 함 (visited)

⇒ 2번과 3번 과정을 재귀 형태로 구현 (방문한 정점이 새로운 출발점이 됨). 방문하지 않은 이웃 정점이 없으면 재귀 함수 종료.

⇒ 4번 구현??

DFS 구현 – 2단계

- Graph 클래스 구현
  - 멤버변수: graph (인접 리스트), vertex_count, visited (정점 별 방문 여부)
  - 메서드: 생성자, dfs()

```
class Graph:
    …
    def dfs(self, start) :
        for vertex in range(self.vertex_count) :
            self.visited[vertex] = False

        self.dfs_recursive(start)
```

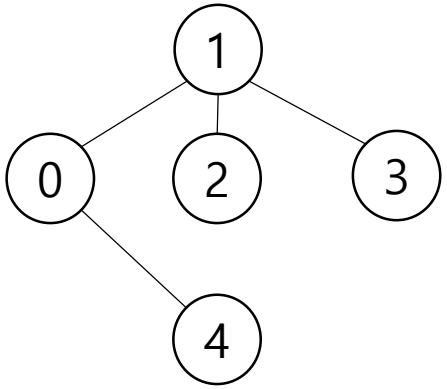
DFS 구현 – 2단계

- Graph 클래스 구현
  - 멤버변수: graph (인접 리스트), vertex_count, visited (정점 별 방문 여부)
  - 메서드: 생성자, dfs()

```
class Graph:
    …
    def dfs_recursive(self, vertex):
        self.visited[vertex] = True
        print(vertex, ' ', end='')
        for neighbor in self.graph[vertex] :
            if not self.visited[neighbor] :
                self.dfs_recursive(neighbor)
```

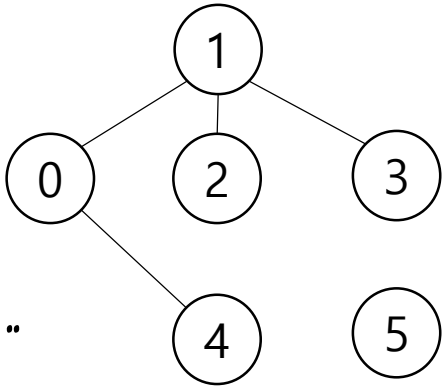
DFS 테스트 코드

```
graph_adjlist = [[1, 4], [0, 2, 3], [1], [1], [0]]
graph = Graph(graph_adjlist)
graph.dfs(0)
graph.dfs(1)
```



DFS 구현 – 문제점?

```
def dfs(self, start) :
    for vertex in range(self.vertex_count) :
        self.visited[vertex] = False

    self.dfs_recursive(start)
```



- 그래프가 연결이 단절된 부분 그래프로 구성된 경우..
  - 하나의 부분 그래프만 방문하게 됨..

DFS 구현 – 3단계 – 수정 구현

```
class Graph:
    …
    def dfs(self, start) :
        for vertex in range(self.vertex_count) :
            self.visited[vertex] = False

        self.dfs_recursive(start)

        for vertex in range(self.vertex_count) :
            if (self.visited[vertex] == False) :
                self.dfs_recursive(vertex)

        print()
```

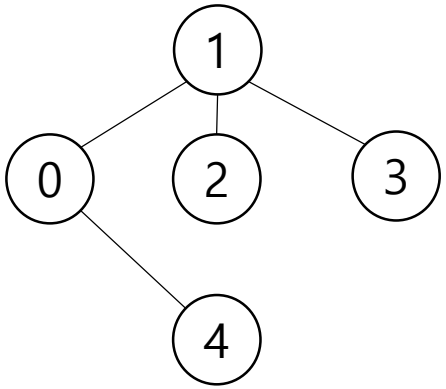
그래프의 탐색

- 너비우선탐색 (BFS)
  - 임의의 정점에서 시작
  - 타겟 정점의 모든 이웃 정점 방문 (처음 방문한 정점은 큐에 삽입)
  - 큐의 앞쪽 정점을 타겟 정점으로 수정하여 반복 (타겟 정점을 큐에서 제거)
  - 모든 정점을 방문하면 완료.

너비우선탐색

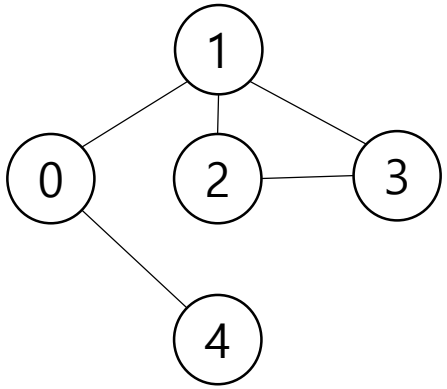
- 방문 가능한 정점들이 여러 개인 경우 낮은 숫자의 정점을 먼저 방문한다고 가정하면,

  - 0 -> 1 -> 4 -> 2 -> 3 순서로 방문



- 방문 가능한 정점들이 여러 개인 경우 낮은 숫자의 정점을 먼저 방문한다고 가정하면,

  - 0 -> 1 -> 4 -> 2 -> 3 순서로 방문



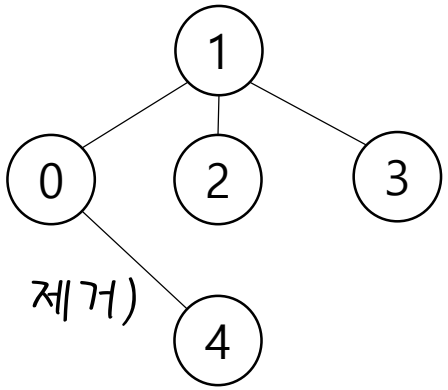
BFS 구현 – 1단계

- Graph 클래스 구현
  - 멤버변수: graph (인접 리스트), vertex_count, visited (정점 별 방문 여부), bfsQ (큐)
  - 메서드: 생성자, bfs()

```
import Queue
class Graph:
    def __init__(self, graph):
        self.graph = graph
        self.vertex_count = len(self.graph)
        self.visited = [False] * self.vertex_count

        self.bfsQ = Queue.Queue()

    def bfs(self) :
        …
```

BFS 구현 – 1단계

- bfs() 구현
  1. 임의의 정점에서 시작
  2. 타겟 정점의 모든 이웃 정점 방문 (처음 방문한 정점은 큐에 삽입)
  3. 큐의 앞쪽 정점을 타겟 정점으로 수정하여 반복 (타겟 정점을 큐에서 제거)
  4. 모든 정점을 방문하면 완료.



⇒ 방문한 정점들의 정보를 관리해야 함 (visited)

⇒ 2번과 3번 과정을 반복문으로 구현.

⇒ 4번 구현??

BFS 구현 – 1단계

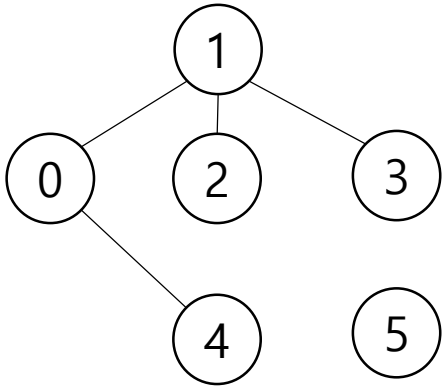
```
class Graph:
    …
    def bfs(self, start) :
        for vertex in range(self.vertex_count) :
            self.visited[vertex] = False

        self.bfsQ.enqueue(start)
        self.do_bfs()
```

BFS 구현 – 1단계

```
def do_bfs(self):
        while self.bfsQ.size() > 0 :
            vertex = self.bfsQ.dequeue()

            print(vertex, ' ', end='')
            self.visited[vertex] = True

            for neighbor in self.graph[vertex] :
                if not self.visited[neighbor] :
                    self.visited[neighbor] = True
                    print(neighbor, ' ', end='')
                    self.bfsQ.enqueue(neighbor)
```

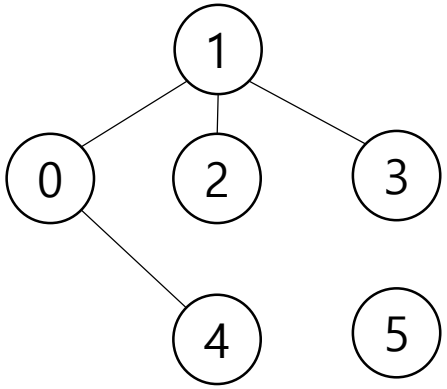
BFS 테스트 코드

```
graph_adjlist = [[1, 4], [0, 2, 3], [1], [1], [0], []]
graph = Graph(graph_adjlist)
graph.dfs(0)
graph.dfs(1)
graph.bfs(0)
graph.bfs(1)
```



BFS 구현 – 문제점?

```
def bfs(self, start) :
    for vertex in range(self.vertex_count) :
        self.visited[vertex] = False

    self.bfsQ.enqueue(start)
    self.do_bfs()
```



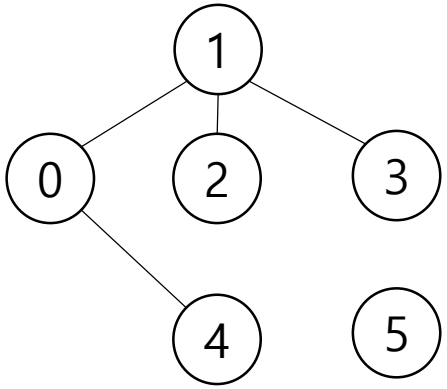
- 그래프가 연결이 단절된 부분 그래프로 구성된 경우..
  - 하나의 부분 그래프만 방문하게 됨..

BFS 구현 – 2단계 – 수정 구현

```
class Graph:
    …
    def bfs(self, start) :
        for vertex in range(self.vertex_count) :
            self.visited[vertex] = False

        self.bfsQ.enqueue(start)
        self.do_bfs()

        for vertex in range(self.vertex_count) :
            if (self.visited[vertex] == False) :
                self.bfsQ.enqueue(vertex)
                self.do_bfs()

        print()
```

BFS 테스트 코드

```
graph_adjlist = [[1, 4], [0, 2, 3], [1], [1], [0], []]
graph = Graph(graph_adjlist)
graph.dfs(0)
graph.dfs(1)
graph.bfs(0)
graph.bfs(1)
```



Summary

- 그래프
  - 정점과 간선으로 구성
- 그래프의 표현
  - 인접행렬
  - 인접리스트
- 그래프 탐색
  - DFS
  - BFS