10차시

오릴러 문제(1800년대)
다리를 한번만 건너서 처음 출발했던 장소로 돌아오는 문제
위치 : 정점(노드), 다리 : 간선
모든 정점에 연결된 간선의 수가 짝수이면 오일러 경로 존재함
따라서 그래프(b)에는 오일러 경로가 존재하지 않음

그래프란
연결되어 있는 객체간의 관계를 표현하는 자료구조
가장 일반적인 자료구조 형태
그래프 G는 (V, E)로 표시
정점 또는 노드
간선 또는 링크 : 정점들 간의 관계 의미
시각 적으로 달라도, 모든 정점사이의 관계가 동일하면 같은 그래프로 판단

그래프의 용어
인접 정점 : 간선에 의해 직접 연결된 정점
차수 : 정점에 연결된 간선의 수
무방향 그래프의 차수의 합은 간선 수의 2배
방향 그래프에서 진입차수, 진출차수가 있고, 모든 진입(진출) 차수의 합은 간선의 수
그래프의 경로
무방향 그래프의 정점s로부터 정점e까지의 경로, 정점 : s,v1,v2,vk,e / 간선 (s, v1), (v1, v2) 등 존재
뱡향 그래프의 정점s로부터 정점e까지의 경로, 정점 : s,v1,v2,vk,e / 간선 <s, v1>, <v1, v2> 등 존재
경로의 길이 : 경로를 구성하는데 사용된 간선의 수
단순경로 : 경로중에 반복되는 간선이 없는 경로, 왔던 노드로 다시 돌아가지 않는 경로
사이클 : 시작 정점과 종료 정점이 동일한 경로
연결그래프 : 모든 정점들 사이에 경로가 존재하는 그래프
트리 : 사이클을 가지지 않는 연결 그래프
완전 그래프 : 모든 정점 간의 간선이 존재하는 그래프,
n개의 정점을 가진 무방향 완전그래프의 간선의 수 = n*(n-1)/2

간선의 종류에 따라 분류되는 그래프 종류
무방향 그래프
(A, B) = (B, A)
V(G1) = {A, B, C, D}
E(G1) = {(A, B), (A,C), (A,D), (B,C), (C,D)}

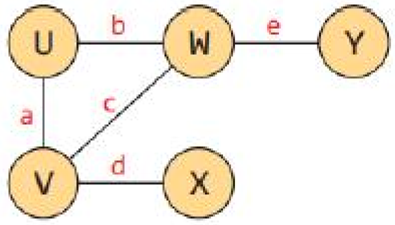
방향그래프
<A,B> != <B,A>
V(G3) = {A, B, C}
E(G3) = {<A, B, C>, <B,A>, <B,C>}

가중치 그래프, 네트워크
간선에 비용이나 가중치가 할당된 그래프

부분 그래프

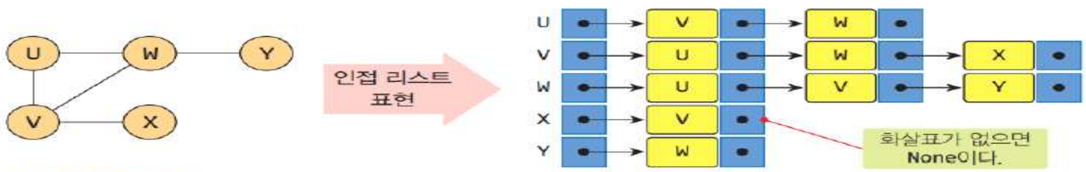
그래프의 추상자료형(ADT)
isEmpty() : 그래프가 공백 상태인지 확인한다.
countVertex() : 정점의 수를 반환한다.
countEdge() : 간선의 수를 반환한다.
getEdge(u,v) : 정점 u에서 정점 v로 연결된 간선을 반환한다.
degree(v) : 정점 v의 차수를 반환한다.
adjacent : 정점 v에 인접한 모든 정점의 집합을 반환한다.
insertVertex(v) : 그래프에 정점 v를 삽입한다.
insertEdge(u, v) : 그래프에 간선(u, v)를 삽입한다.
deleteVertex(v) : 그래프의 정점 v를 삭제한다.
deleteEdge(u, v) : 그래프의 간선 (u, v)를 삭제한다.

인접행렬을 이용한 그래프의 표현
2차 정사각배열을 이용하여 값이 1이면 연결, 0이면 연결되지 않음을 표현
무방향 그래프는 인접행렬이 대칭이다.

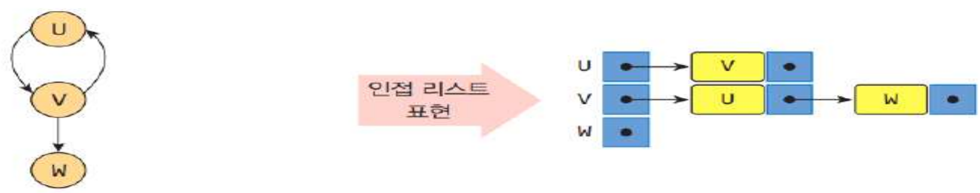


인접 리스트를 이용한 표현
무방향 그래프 : 각 노드에 연결리스트로 표현
방향 그래프 : 각 노드에 연결리스트로 표현, 반복되기 전까지 각 노드에 연결

❖ 무방향 그래프



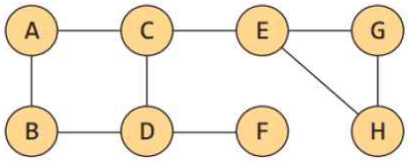
❖ 방향 그래프



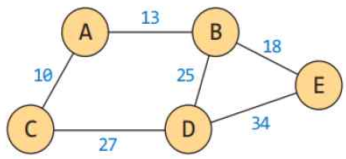
인접 행렬과 인접 리스트의 복잡도 비교

인접 행렬	인접 리스트
간선의 수에 무관하게 항상 n^2 **개의 메모리 공간**이 필요하다. 따라서 정점에 비해 간선의 수가 매우 많은 **조밀 그래프**(dense graph)에서 효과적이다.	n개의 연결 리스트가 필요하고, $2e$개의 노드가 필요하다. 즉 $n+2e$ **개의 메모리 공간**이 필요하다. 따라서 정점에 비해 간선의 개수가 매우 적은 **희소 그래프**(sparse graph)에서 효과적이다.
u와 v를 연결하는 간선의 유무는 `M[u][v]`를 조사하면 바로 알 수 있다. 따라서 `getEdge(u,v)`의 시간 복잡도는 $O(1)$이다.	`getEdge(u,v)`연산은 정점 u의 연결 리스트 전체를 조사해야 한다. 정점 u의 차수를 d_u라고 한다면 이 연산의 시간 복잡도는 $O(d_u)$이다.
정점의 차수를 구하는 `degree(v)`는 정점 v에 해당하는 행을 조사하면 되므로 $O(n)$이다. 즉, 정점 v에 대한 차수는 다음과 같이 계산된다. $$degree(v)=\sum_{k=0}^{n-1} M[v][k]$$	정점 v의 차수 `degree(v)`는 v의 연결 리스트의 길이를 반환하면 된다. 따라서 시간 복잡도는 $O(d_v)$이다.
정점 v의 인접 정점을 구하는 `adjacent(v)`연산은 해당 행의 모든 요소를 검사하면 되므로 $O(n)$의 시간이 요구된다.	정점 v에 간선으로 직접 연결된 모든 정점을 구하는 `adjacent(v)`연산도 해당 연결리스트의 모든 요소를 방문해야 되므로 $O(d_v)$이다.
그래프에 존재하는 모든 간선의 수를 알아내려면 인접 행렬 전체를 조사해야 하므로 n^2번의 조사가 필요하다. 따라서 $O(n^2)$의 시간이 요구된다.	전체 간선의 수를 알아내려면 헤더 노드를 포함하여 모든 인접 리스트를 조사해야 하므로 $O(n+e)$의 연산이 요구된다.

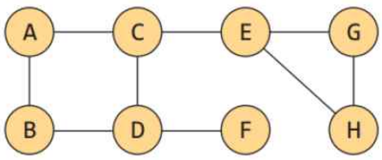
파이썬을 이용한 인접 행렬 표현
무방향 그래프

무방향 그래프	인접 행렬 표현
	vertex = ['A','B','C','D','E','F','G','H'] adjMat = [[0, 1, 1, 0, 0, 0, 0, 0], [1, 0, 0, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [0, 1, 1, 0, 0, 1, 0, 0], [0, 0, 1, 0, 0, 0, 1, 1], [0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0, 1], [0, 0, 0, 0, 1, 0, 1, 0]]

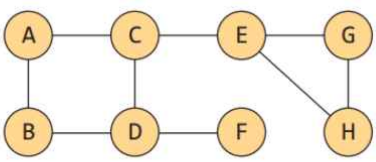
가중치 그래프

가중치 그래프	인접 행렬 표현
	vertex = ['A', 'B', 'C', 'D', 'E'] adjMat = [[0, 13, 10, None, None], [13, 0, None, 25, 18], [10, None, 0, 27, None], [None, 25, 27, 0, 34], [None, 18, None, 34, 0]]

인접 정점 인덱스의 리스트

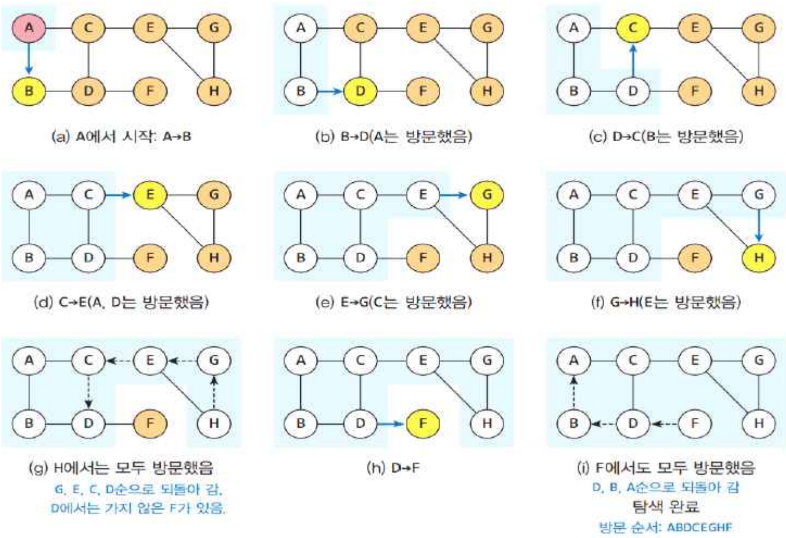
그래프	인접 정점 인덱스의 리스트
	vertex = ['A','B','C','D','E','F','G','H'] adjList = [[1, 2], # 'A'의 인접정점 인덱스 [0, 3], # 'B'의 인접정점 인덱스 [0, 3, 4], # 'C' [1, 2, 5], # 'D' [2, 6, 7], # 'E' [3], # 'F' [4, 7], # 'G' [4, 6]] # 'H'

파이썬의 딕셔너리와 인접 정점 집합이용

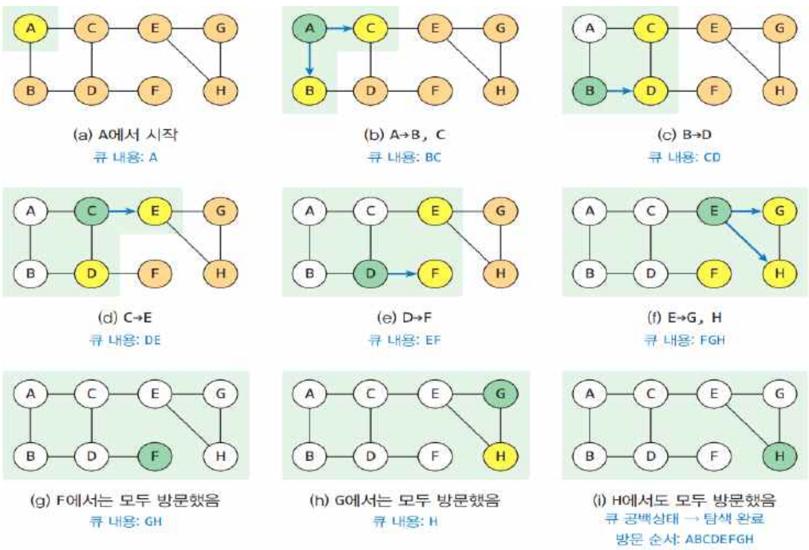
그래프	딕셔너리와 집합을 이용한 표현
	graph = { 'A': set(['B','C']), # 또는 'A': {'B', 'C'} 'B': set(['A','D']), 'C': set(['A','D','E']), 'D': set(['B','C','F']), 'E': set(['C','G','H']), 'F': set(['D']), 'G': set(['E','H']), 'H': set(['E','G']) }

그래프의 탐색
가장 기본적인 연산으로 시작 정점부터 차례대로 모든 정점들을 한 번씩 방문
많은 문제들이 단순히 탐색만으로 해결됨
방법으로 깊이 우선 탐색과 너비 우선 탐색이 있다.

깊이 우선 탐색
DFS (depth first search)
한 방향으로 끝까지 가다가 더 이상 갈 수 없게 되면 가장 가가운 갈림 길로 돌아와서 다른 방향으로 다시 탐색 진행
되돌아 가기 위해서 스택이 필요
순환함수 호출로 묵시적인 스택 이용



너비 우선 탐색
시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회방법
큐를 사용하여 구현됨

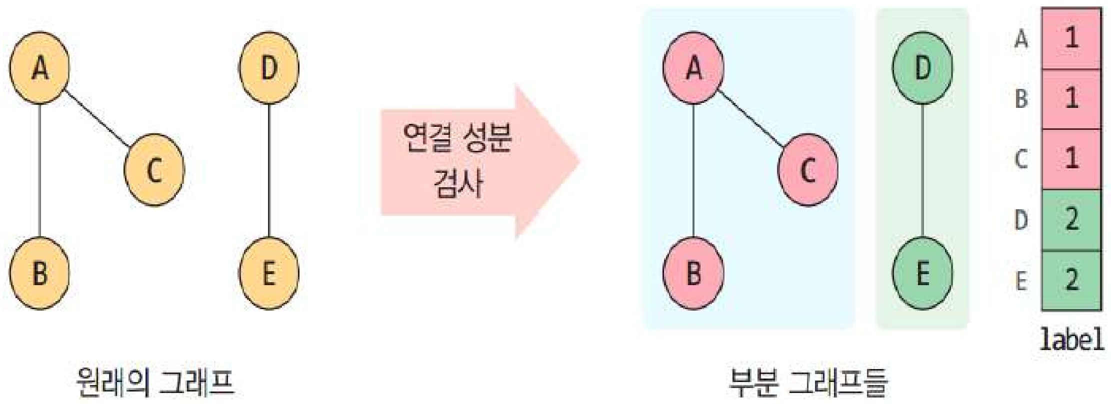


탐색 알고리즘 성능
깊이 우선 탐색 / 너비 우선 탐색
인접 행렬 표현 : O(n^2)
인접 리스트로 표현 : O(n + e)

완전그래프와 같은 조밀 그래프 -> 인접 행렬이 유리
희소 그래프 -> 인접 리스트가 유리

```python
import collections
# 깊이 우선 탐색
def dfs(graph, start, visited = set()): # 처음 호출할때 visited 공집합
    if start not in visited: # start가 방문하지 않은 정점이면
        visited.add(start) # start를 방문한 노드 집합에 추가
        print(start, end=' ') # start를 방문했다고 출력함
        nbr = graph[start] - visited # {인접정점 중에 가보지 않은 정점} = {인접정점} - {방문정점}
        for v in nbr:
            dfs(graph, v, visited) # v에 대해 dfs를 순환적으로 호출

# 너비 우선 탐색
def bfs(graph, start):
    visited = set([start]) # 맨 처음에는 start만 방문한 정점임
    queue = collections.deque([start]) # 컬렉션의 덱 객체 생성(큐로 사용)
    while queue: # 공백이 아닐 때 까지
        vertex = queue.popleft() # 큐에서 하나의 정점 vertex를 빼냄
        print(vertex, end=' ') # vertex는 방문했음을 출력
        nbr = graph[vertex] - visited # {인접정점 중에 가보지 않은 정점} = {인접정점} - {방문정점}
        for v in nbr:
            visited.add(v) # 이제 v는 방문했음
            queue.append(v) # v를 큐에 삽입

if __name__ == "__main__":

    graph = { 'A' : set(['B','C']),
              'B' : set(['A','D']),
              'C' : set(['A','D','E']),
              'D' : set(['B','C', 'F']),
              'E' : set(['C','G','H']),
              'F' : set(['D']),
              'G' : set(['E','H']),
              'H' : set(['E','G']) }

    dfs(graph, 'A')
    print("\n")
    bfs(graph, 'A')

# 출력
# A C E H G D B F

# A C B E D H G F
```

```
## 연결 성분이란
# 최대로 연결된 부분 그래프들을 구함
# DFS 또는 BFS를 반복적으로 이용
```




```
def find_connected_component(graph):
    visited = set() # 이미 방문한 정점 집합
    colorList = [] # 부분 그래프별 정점 리스트

    for vtx in graph: # 그래프의 모든 정덤들에 대해
        if vtx not in visited: # 방문하지 않은 정점이 있으면
            color = dfs_cc(graph, [], vtx, visited) # 새로운 컬러 리스트
            colorList.append(color) # 컬러리스트에 추가

    print("그래프 연결성분 개수 = %d" % len(colorList))
    print(colorList) # 부분 그래프별 정점 리스트 출력

def dfs_cc(graph, color, vertex, visited):
    if vertex not in visited : # 아직 가보지 않은 정점에 대해
        visited.add(vertex) # 전체기준 방문한 리스트에 추가(전역), set이므로 add로 요소를 추가
        color.append(vertex) # 같은 부분기준 방문한 리스트에 추가(지역)
        nbr = graph[vertex] - visited # 이어진 정점에서 이미 지나간 정점을 뺀 정점저장
        for v in nbr: # 저장된 정점에서 하나씩
            dfs_cc(graph, color, v, visited) # 순환 호출
    return color # 최종적으로 완성된 부분 정점 그래프 출

if __name__ == "__main__":
    mygraph = { "A" : set(["B", "C"]),
                "B" : set(["A"]),
                "C" : set(["A"]),
                "D" : set(["E"]),
                "E" : set(["D"])}

    print('find_connected_component:')
    find_connected_component(mygraph)

# 출력
# find_connected_component:
# 그래프 연결성분 개수 = 2
# [['A', 'C', 'B'], ['D', 'E']]
```

```
## 신장 트리란
# 인접 리스트로 구현을 하며 그래프 내의 모든 정점을 포함하는 트리이다.
# 인접한 두 정점을 이어주는 간선을 순서대로 출력하는 코드이다.
# 사이클을 포함하면 안됨, 간선의 수 = n -1

## DFS : 깊이 우선 탐색(depth - first search)
# 스택 사용

## BFS : 너비 우선 탐색(breadth-first search)
# 큐 사용

import collections

# 너비 우선 탐색을 기준으로 한 신장트리
def bfsST(graph, start):
    visited = set([start]) # 맨 처음에는 start만 방문한 정점임
    queue = collections.deque([start]) # 파이썬 컬렉션의 덱 생성(큐로 사용)
    while queue: # 공백이 아닐때까지
        v = queue.popleft() # 큐에서 하나의 정점 v를 빼냄
        nbr = graph[v] - visited # nbr = {v의 인접벙점} - {방문정점}
        for u in nbr: # 갈 수 있는 모든 인접 정점에 대해
            print("(", v, ",", u, ")", end= "") # (v, n) 간선 추가
            visited.add(u) # 이제 u는 방문 했음
            queue.append(u) # u를 큐에 삽입

if __name__ == "__main__":

    mygraph = { "A" : set(["B", "C"]),
                "B" : set(["A"]),
                "C" : set(["A"]),
                "D" : set(["E"]),
                "E" : set(["D"])}

    graph = { 'A' : set(['B','C']),
             'B' : set(['A','D']),
             'C' : set(['A','D','E']),
             'D' : set(['B','C', 'F']),
             'E' : set(['C','G','H']),
             'F' : set(['D']),
             'G' : set(['E','H']),
             'H' : set(['E','G']) }

    bfsST(mygraph, "A")
    print()
    bfsST(graph, "A")\

# 출력
# ( A , C )( A , B ) # A가 출발지점이므로 (D, E)는 출력되지 않는다.
# ( A , C )( A , B )( C , E )( C , D )( E , H )( E , G )( D , F )
```

위상 정렬
위상 정렬이란 방향 그래프에 대해 정점들의 선행 순서를 위해하지 않으면서 모든 정점을 나열하는 것

알고리즘(풀이 해석)

<table>
<tr>
<td>## 스택일 경우(FIFO)
[0] : 0,[1] : 0, [2] : 1, [3] : 3, [4] : 1, [5] : 3

값이 0인 A, B 출력
v = 0일때
(0), (1), 2, 3, (4), (5)
[2] = 0 -> 인덱스 2 vlist 추가 -> vertex[2] = C 출력
[3] = 2
v = 1일때
(0), (1), (2), 3, 4, (5)
[3] = 1
[4] = 0 -> 인덱스 4 vlist 추가 -> vertex[4] = E 출력
v = 2일때
(0), (1), (2), 3, (4), 5
[3] = 0 -> 인덱스 3 vlist 추가 -> vertex[3] = D 출력
[5] = 2
v = 3일때
(0), (1), (2), (3), (4), 5
[5] = 1
v = 4일때
(0), (1), (2), (3), (4), 5
[5] = 0 -> 인덱스 5 vlist 추가 -> vertex[5] = F 출력
v = 5일때
(0), (1), (2), (3), (4), (5)</td>
<td>## 큐일 경우(FILO)
[0] : 0,[1] : 0, [2] : 1, [3] : 3, [4] : 1, [5] : 3

vlist = 0, 1 이므로 1반환 -> B출력
v = 1일때
(0), (1), (2), 3, 4, (5)
[3] = 2
[4] = 0 -> 인덱스 4 vlist 추가
vlist = 0, 4 이므로 4반환 -> E출력
v = 4일때
(0), (1), (2), (3), (4), 5
[5] = 2
vlist = 0이므로 0반환 -> A출력
v = 0일때
(0), (1), 2, 3, (4), (5)
[2] = 0 -> 인덱스 2 vlist 추가
[3] = 1
vlist = 2 이므로 2반환 -> C출력
v = 2일때
(0), (1), (2), 3, (4), 5
[3] = 0 -> 인덱스 3 vlist 추가
[5] = 1
vlist = 3 이므로 3반환 -> D출력
v = 3일때
(0), (1), (2), (3), (4), 5
[5] = 0 -> 인덱스 5 vlist 추가
vlist = 5 이므로 5반환 -> F출력</td>
</tr>
</table>

```
def topological_sort_AM(vertex, graph):
    n = len(vertex)
    inDeg = [0]*n # 정점의 수를 저장 정점과 이어진 관계의 수

    for i in range(n):
        for j in range(n):
            if graph[i][j] > 0:
                inDeg[j] += 1 # 정점과 이어진 관계의 수를 1 증가시킴

    vlist = []  # 정점과 이어진 관계의 수가 0인 정점 리스트를 만듦
    for i in range(n):
        if inDeg[i] == 0:
            vlist.append(i)

    while len(vlist) > 0: # 리스트가 공백이 아닐 때까지
        v = vlist.pop() # 정점과 이어진 관계의 수가 0인 정점을 뒤에서 하나 꺼냄
        print(vertex[v], end=' ') # 화면 출력

        for u in range(n):
            if v != u and graph[v][u] > 0:
                inDeg[u] -= 1 # 해당 정점의 정점과 이어진 관계의 수를 감소
                if inDeg[u] == 0: # 정점과 이어진 관계의 수가 0이면
                    vlist.append(u) # vlist에 추가

if __name__ == "__main__":
    vertex = ['A', 'B', 'C', 'D', 'E', 'F']
    graphAM = [ [0, 0, 1, 1, 0, 0],
                [0, 0, 0, 1, 1, 0],
                [0, 0, 0, 1, 0, 1],
                [0, 0, 0, 0, 0, 1],
                [0, 0, 0, 0, 0, 1],
                [0, 0, 0, 0, 0, 0]]
    print('topological_sort: ')
    topological_sort_AM(vertex, graphAM) # 재귀 사용
    print()

# 출력
# topological_sort:
# B E A C D F
```