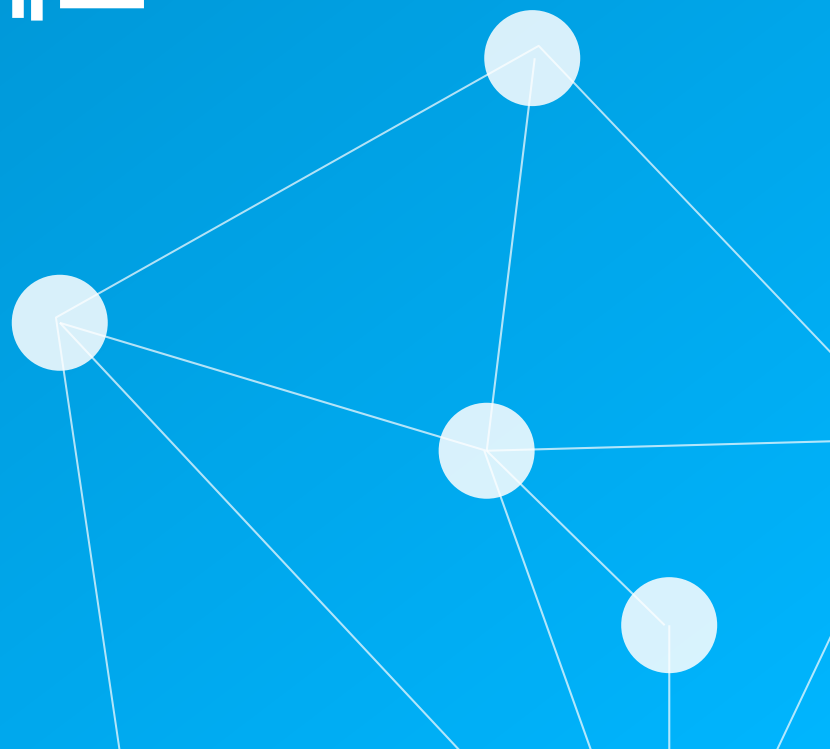

파이썬
자료구조

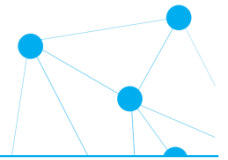
10

CHAPTER

그래프

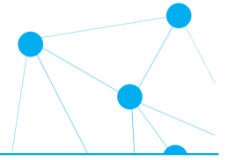


10장. 학습 목표



- 그래프의 개념을 이해한다.
- 그래프를 표현하는 방법을 이해한다.
- 파이썬의 내장 자료형을 이용해 그래프를 표현하는 방법들을 이해한다.
- 그래프의 탐색 방법을 이해한다.
- 그래프 탐색을 이용한 문제해결 능력을 배양한다.
- 다양한 문제에 그래프를 활용할 수 있는 능력을 기른다.

10.1 그래프란?



- 그래프의 역사
- 그래프의 정의,
- 그래프의 종류,
- 그래프 용어
- 그래프 ADT

그래프란?



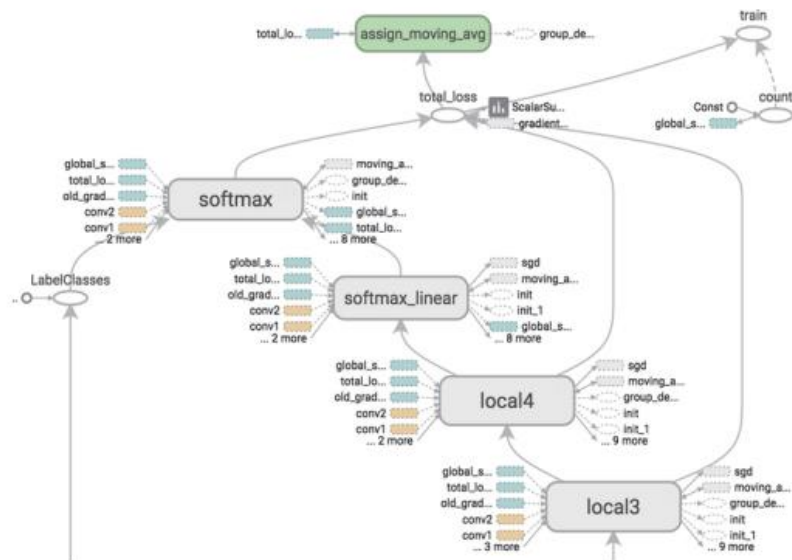
- 연결되어 있는 객체 간의 관계를 표현하는 자료구조
- 가장 일반적인 자료구조 형태



지하철 노선도

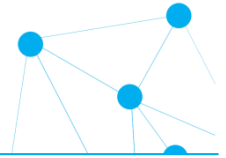


항공 노선도



텐서플로우 그래프

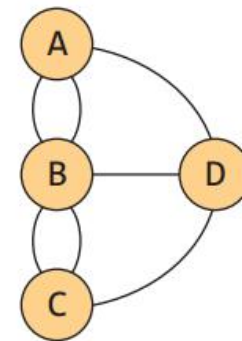
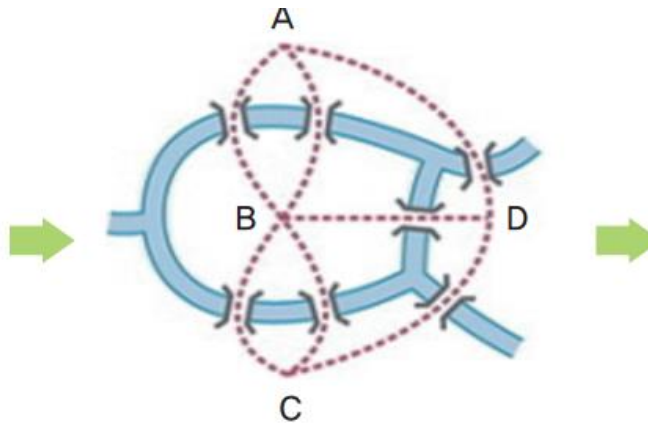
그래프 역사



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

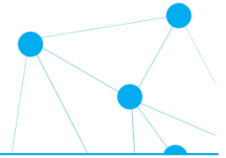


원래의 문제

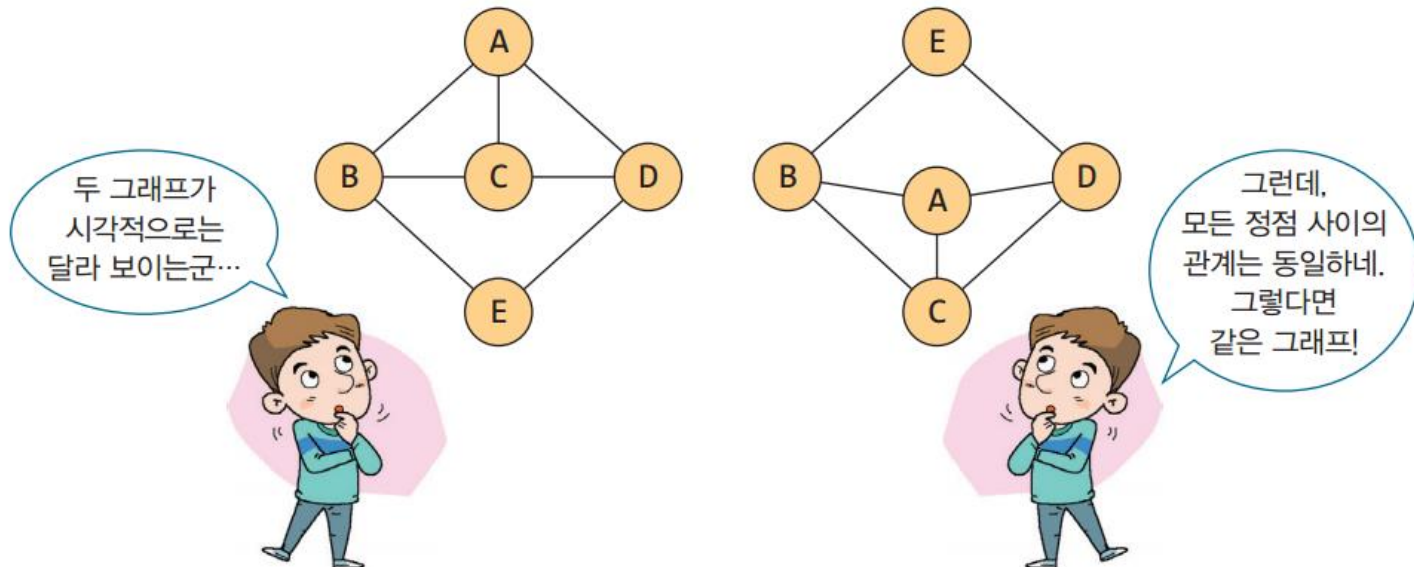


그래프 문제

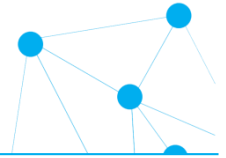
그래프 정의



- 그래프 G 는 (V, E) 로 표시
 - 정점(vertices) 또는 노드(node)
 - 간선(edge) 또는 링크(link): 정점들 간의 관계 의미
- 다른 그래프? 같은 그래프?



그래프의 종류

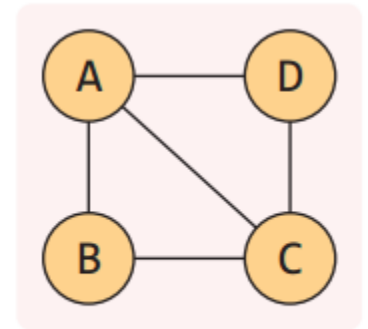


- 간선의 종류에 따라
 - 무방향 그래프(undirected graph)

- $(A, B) = (B, A)$

$$V(G1) = \{A, B, C, D\}$$

$$E(G1) = \{(A, B), (A, C), (A, D), (B, C), (C, D)\}$$



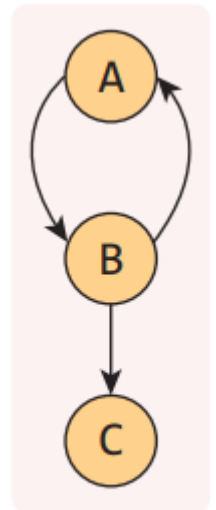
G1

- 방향 그래프(directed graph)

- $\langle A, B \rangle \neq \langle B, A \rangle$

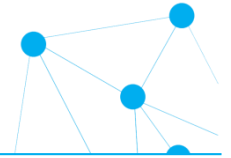
$$V(G3) = \{A, B, C\},$$

$$E(G3) = \{\langle A, B \rangle, \langle B, A \rangle, \langle B, C \rangle\}$$

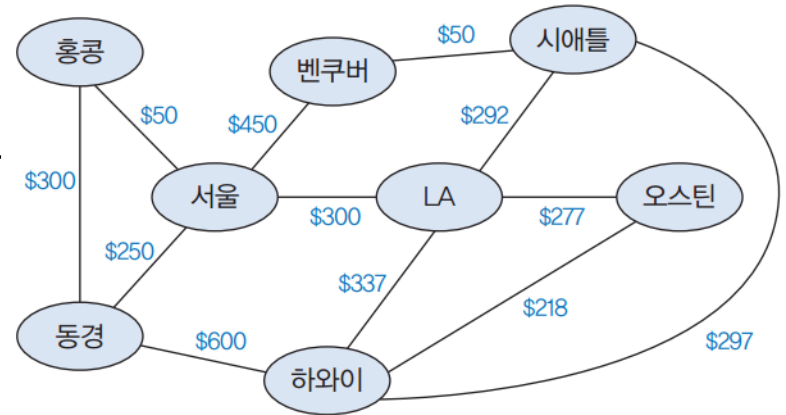


G3

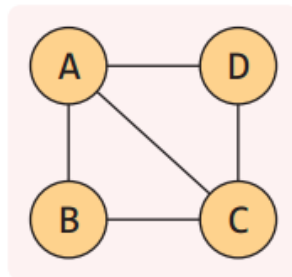
그래프의 종류



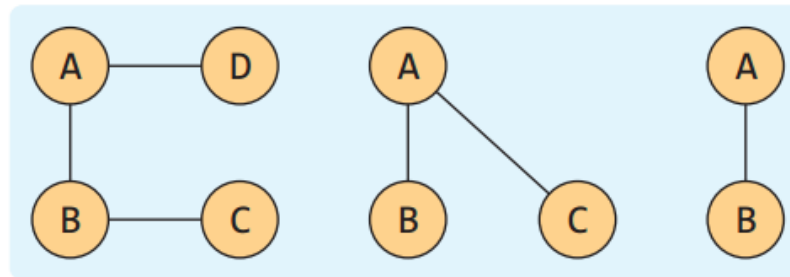
- 가중치 그래프, 네트워크
 - 간선에 비용이나 가중치가 할당된 그래프



- 부분 그래프

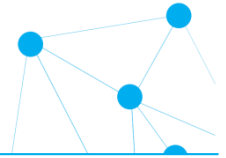


G1

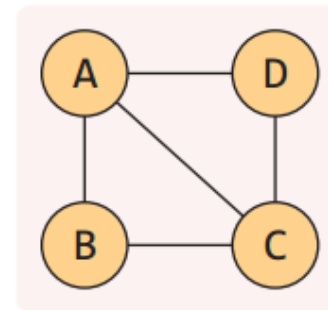


G1의 부분 그래프들

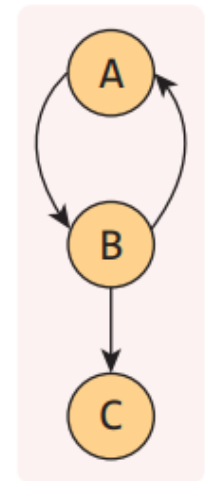
그래프의 용어



- 인접 정점
 - 간선에 의해 직접 연결된 정점
- 차수(degree)
 - 정점에 연결된 간선의 수
 - 무방향 그래프
 - 차수의 합은 간선 수의 2배
 - 방향 그래프
 - 진입차수, 진출차수
 - 모든 진입(진출) 차수의 합은 간선의 수

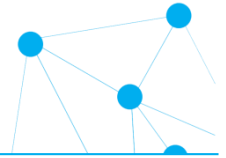


G1



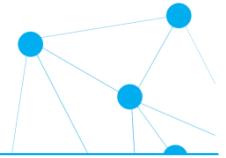
G3

그래프의 용어

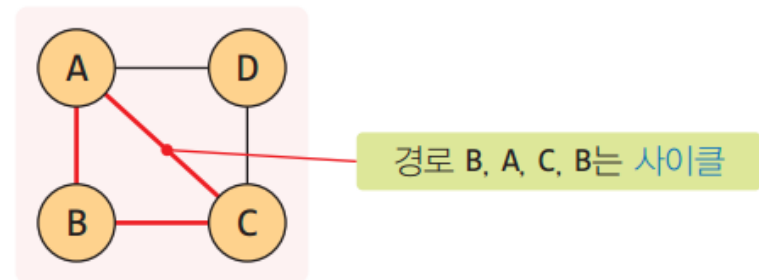
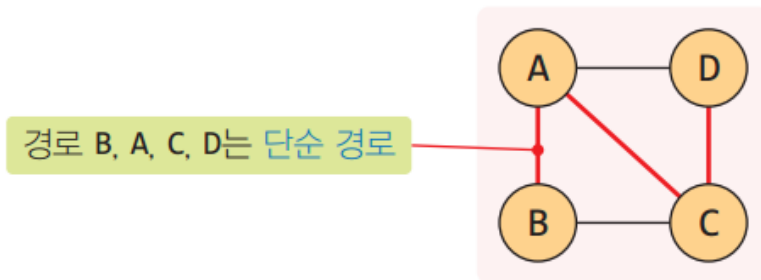


- 그래프의 경로(path)
 - 무방향 그래프의 정점 s 로부터 정점 e 까지의 경로
 - 정점의 나열 $s, v_1, v_2, \dots, v_k, e$
 - 반드시 간선 $(s, v_1), (v_1, v_2), \dots, (v_k, e)$ 존재해야 함
 - 방향 그래프의 정점 s 로부터 정점 e 까지의 경로
 - 정점의 나열 $s, v_1, v_2, \dots, v_k, e$
 - 반드시 간선 $\langle s, v_1 \rangle, \langle v_1, v_2 \rangle, \dots, \langle v_k, e \rangle$ 존재해야 함
- 경로의 길이(length)
 - 경로를 구성하는데 사용된 간선의 수

그래프의 용어

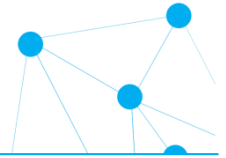


- 단순 경로(simple path)
 - 경로 중에서 반복되는 간선이 없는 경로
 - B,A,C,D는 단순 경로
 - B,A,C,A는 단순 경로 아님

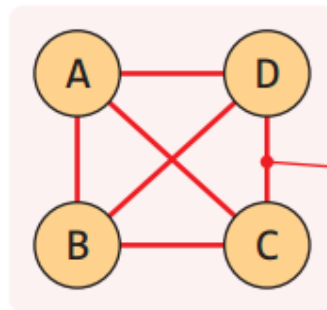


- 사이클(cycle)
 - 시작 정점과 종료 정점이 동일한 경로
 - B,A,C,B는 사이클

그래프의 용어

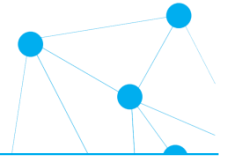


- 연결 그래프(connected graph)
 - 모든 정점들 사이에 경로가 존재하는 그래프
- 트리(tree)
 - 사이클을 가지지 않는 연결 그래프
- 완전 그래프(complete graph)
 - 모든 정점 간에 간선이 존재하는 그래프
 - n 개의 정점을 가진 무방향 완전그래프의 간선의 수 = $n \times (n-1) / 2$
 - $n=4$, 간선의 수 = $(4 \times 3) / 2 = 6$



모든 정점 간에 간선이 존재하니까 완전 그래프!

그래프 ADT



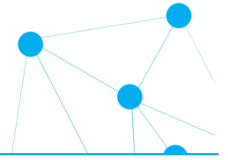
정의 10.1 Graph ADT

데이터: 정점과 간선의 집합

연산

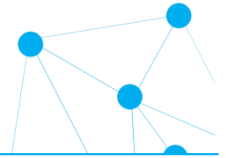
- isEmpty(): 그래프가 공백 상태인지 확인한다.
- countVertex(): 정점의 수를 반환한다.
- countEdge(): 간선의 수를 반환한다.
- getEdge(u,v): 정점 u 에서 정점 v 로 연결된 간선을 반환한다.
- degree(v): 정점 v 의 차수를 반환한다.
- adjacent(v): 정점 v 에 인접한 모든 정점의 집합을 반환한다.
- insertVertex(v): 그래프에 정점 v 를 삽입한다.
- insertEdge(u,v): 그래프에 간선 (u,v) 를 삽입한다.
- deleteVertex(v): 그래프의 정점 v 를 삭제한다.
- deleteEdge(u,v): 그래프의 간선 (u,v) 를 삭제한다.

10.2 그래프의 표현

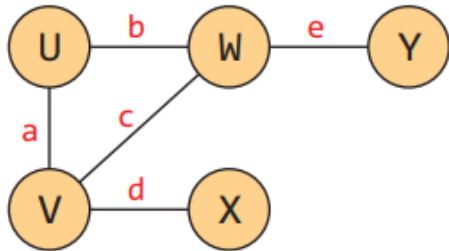


- 인접 행렬을 이용한 표현
- 인접 리스트를 이용한 표현
- 인접 행렬과 인접 리스트의 복잡도 비교
- 파이썬을 이용한 그래프의 인접 행렬 표현
- 파이썬을 이용한 그래프의 인접 리스트 표현

인접 행렬을 이용한 표현



- 인접 행렬 M 을 이용
- 간선 (i, j) 가 있으면: $M[i][j] = 1$, 또는 true
- 그렇지 않으면: $M[i][j] = 0$, 또는 false
- 무방향 그래프: 인접 행렬이 대칭



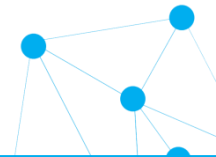
인접 행렬
표현

	0	1	2	3	4
U → 0		a	b		
V → 1	a		c	d	
W → 2	b	c			e
X → 3		d			
Y → 4			e		

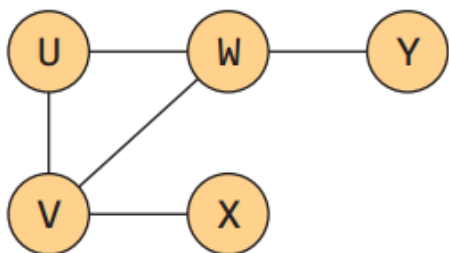
가중치 그래프가
아니면
a, b, ..., e는
모두 1이다.

빈 곳은 0 또는
None이다.

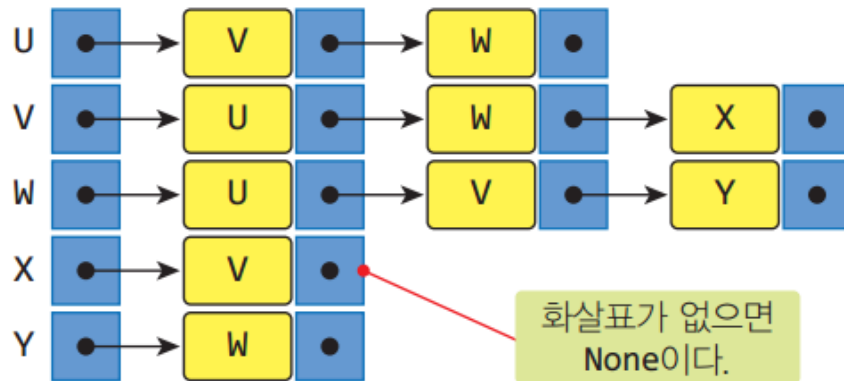
인접 리스트를 이용한 표현



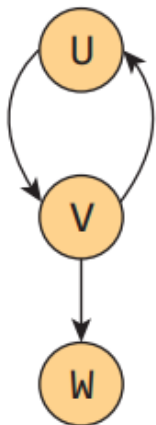
- 무방향 그래프



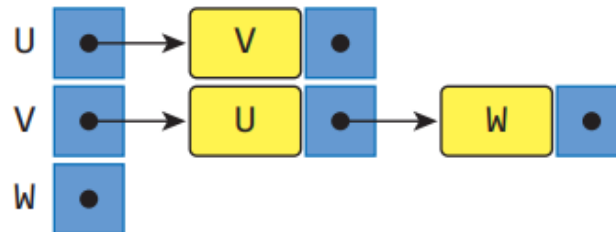
인접 리스트
표현



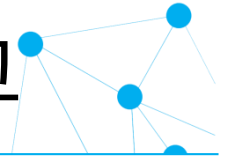
- 방향 그래프



인접 리스트
표현

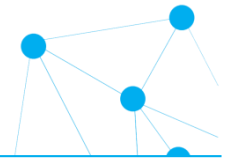


인접 행렬과 인접 리스트의 복잡도 비교



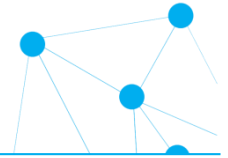
인접 행렬	인접 리스트
간선의 수에 무관하게 항상 n^2 개의 메모리 공간이 필요하다. 따라서 정점에 비해 간선의 수가 매우 많은 조밀 그래프(dense graph)에서 효과적이다.	n 개의 연결 리스트가 필요하고, $2e$ 개의 노드가 필요하다. 즉 $n+2e$ 개의 메모리 공간이 필요하다. 따라서 정점에 비해 간선의 개수가 매우 적은 희소 그래프(sparse graph)에서 효과적이다.
u 와 v 를 연결하는 간선의 유무는 $M[u][v]$ 를 조사하면 바로 알 수 있다. 따라서 $\text{getEdge}(u,v)$ 의 시간 복잡도는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

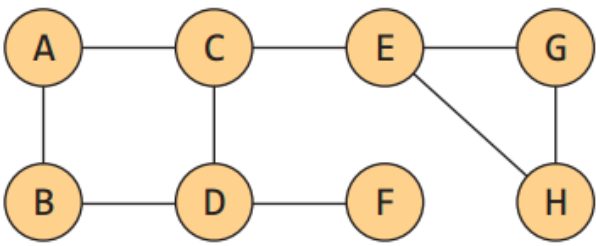


무방향 그래프	인접 행렬 표현
	<pre>vertex = ['A','B','C','D','E','F','G','H'] adjMat = [[0, 1, 1, 0, 0, 0, 0, 0], [1, 0, 0, 1, 0, 0, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0], [0, 1, 1, 0, 0, 0, 1, 0], [0, 0, 1, 0, 0, 0, 0, 1], [0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0, 1], [0, 0, 0, 0, 1, 0, 1, 0]]</pre>
가중치 그래프	인접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

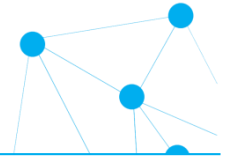
파이썬을 이용한 인접 리스트 표현



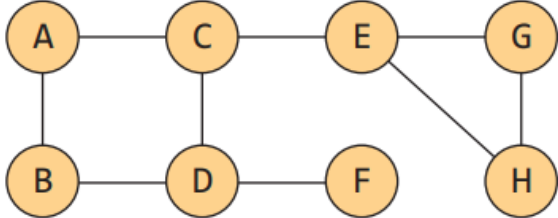
- 다양한 방법으로 인접 리스트를 표현할 수 있음
- 방법1: 인접 정점 **인덱스의 리스트**

그래프	인접 정점 인덱스의 리스트
	<pre>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'</pre>

파이썬을 이용한 인접 리스트 표현



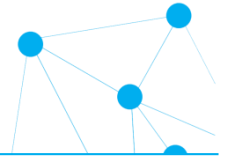
- 방법4: 파이썬의 **딕셔너리**와 **인접 정점 집합** 이용

그래프	딕셔너리와 집합을 이용한 표현
	<pre>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']) }</pre>

- graph['C'] : 정점 'C'의 인접 정점 집합 { 'A', 'D', 'E' }
- graph['C']의 모든 원소 출력 코드

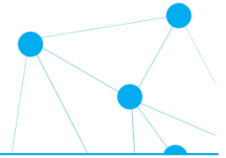
```
for v in graph['C'] :      # 정점 C의 인접 정점 집합의 모든 원소에 대해
    print(v)               # 그 원소를 화면에 출력
```

10.3 그래프의 탐색

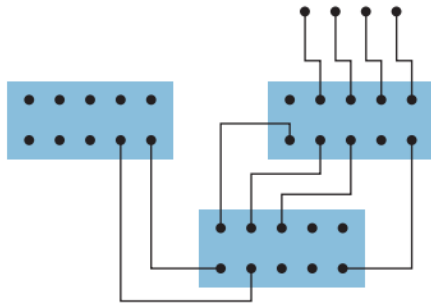


- 그래프의 탐색이란?
- 깊이 우선 탐색
 - 인접 리스트 구현
- 너비 우선 탐색
 - 인접 리스트 구현
- 탐색 알고리즘 성능

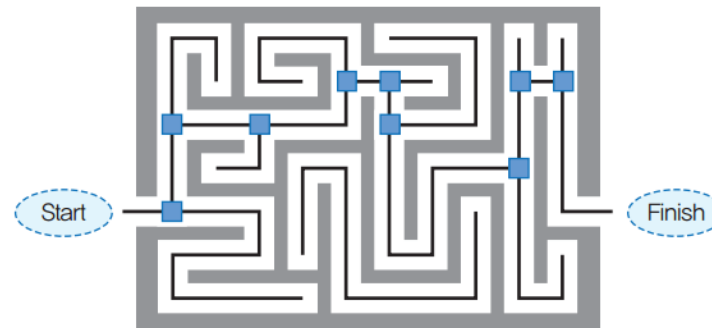
그래프의 탐색이란?



- 가장 기본적인 연산
 - 시작 정점부터 차례대로 모든 정점들을 한 번씩 방문
 - 많은 문제들이 단순히 탐색만으로 해결됨
 - 도로망 예: 특정 도시에서 다른 도시로 갈 수 있는지 여부
 - 전자회로 예: 특정 단자와 다른 단자의 연결 여부



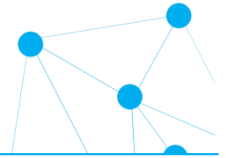
단자들 간의 연결성 검사



미로 탐색 문제

- 방법: 깊이 우선 탐색 (DFS) / 너비 우선 탐색 (BFS)

깊이 우선 탐색 알고리즘

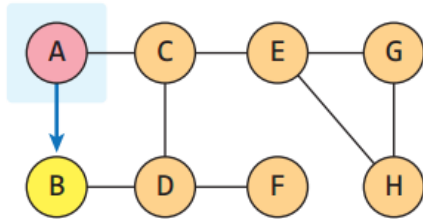
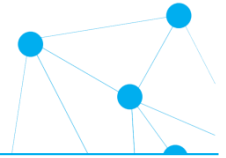


- DFS: depth-first search
 - 한 방향으로 끝까지 가다가 더 이상 갈 수 없게 되면 가장 가까운 갈림길로 돌아와서 다른 방향으로 다시 탐색 진행
 - 되돌아가기 위해서는 스택 필요
 - 순환함수 호출로 묵시적인 스택 이용

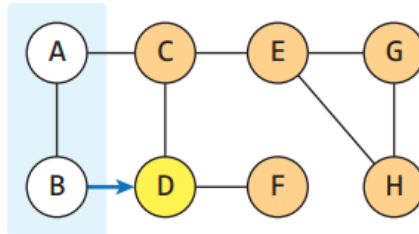
```
def dfs(graph, start, visited = set() ):
    if start not in visited :
        visited.add(start)
        print(start, end=' ')
        nbr = graph[start] - visited
        for v in nbr:
            dfs(graph, v, visited)
```

처음 호출할 때 visited 공집합
start가 방문하지 않은 정점이면
start를 방문한 노드 집합에 추가
start를 방문했다고 출력함
nbr: 차집합 연산 이용
$v \in \{\text{인접정점}\} - \{\text{방문정점}\}$
v에 대해 dfs를 순환적으로 호출

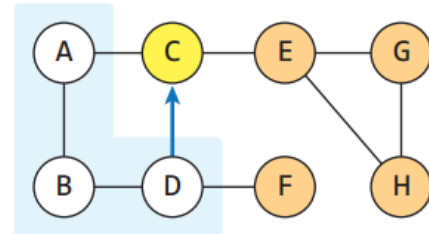
깊이우선탐색 예



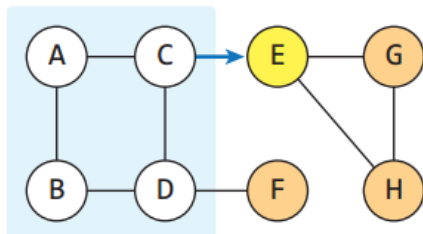
(a) A에서 시작: A→B



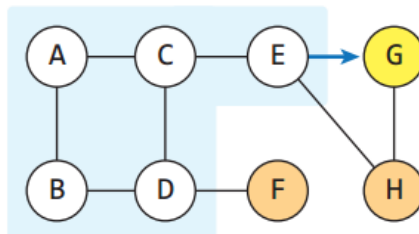
(b) B→D(A는 방문했음)



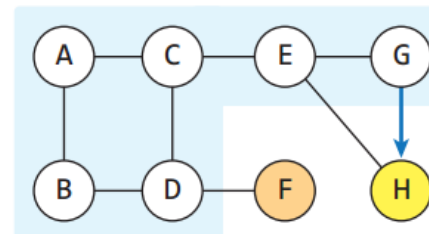
(c) D→C(B는 방문했음)



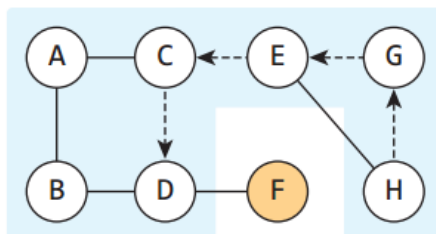
(d) C→E(A, D는 방문했음)



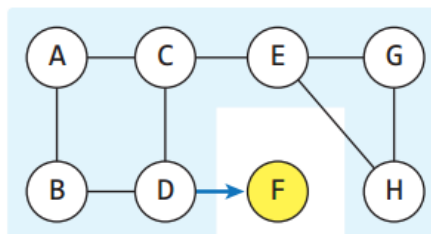
(e) E→G(C는 방문했음)



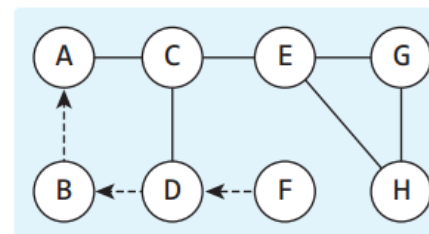
(f) G→H(E는 방문했음)



(g) H에서는 모두 방문했음
G, E, C, D순으로 되돌아 감.
D에서는 가지 않은 F가 있음.

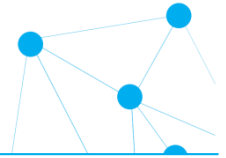


(h) D→F



(i) F에서도 모두 방문했음
D, B, A순으로 되돌아 감
탐색 완료
방문 순서: ABCDEGHF

너비 우선 탐색 알고리즘

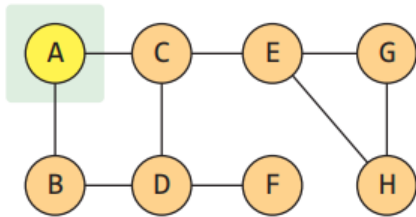
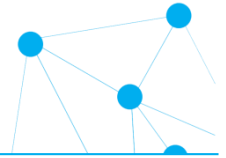


- BFS: breadth-first search
 - 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법
 - 큐를 사용하여 구현됨

```
def bfs(graph, start):  
    visited = set([start])  
    queue = collections.deque([start])  
    while queue:  
        vertex = queue.popleft()  
        print(vertex, end=' ')  
        nbr = graph[vertex] - visited  
        for v in nbr:  
            visited.add(v)  
            queue.append(v)
```

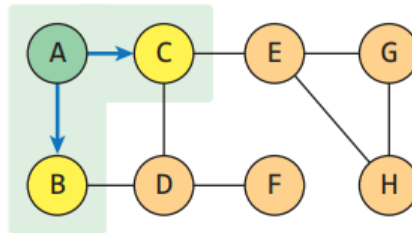
맨 처음에는 start만 방문한 정점임
컬렉션의 덱 객체 생성(큐로 사용)
공백이 아닐 때 까지
큐에서 하나의 정점 vertex를 빼냄
vertex는 방문했음을 출력
nbr: 차집합 연산 이용
$v \in \{\text{인접정점}\} - \{\text{방문정점}\}$
이제 v는 방문했음
v를 큐에 삽입

너비우선탐색 예



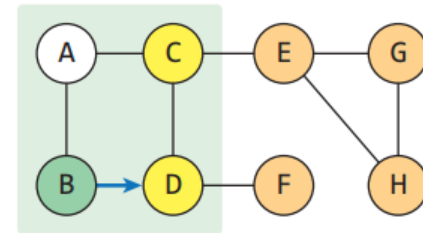
(a) A에서 시작

큐 내용: A



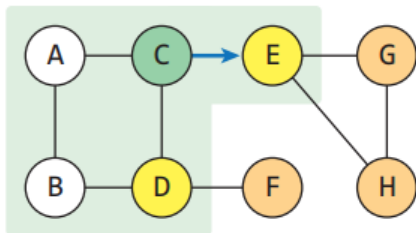
(b) A→B, C

큐 내용: BC



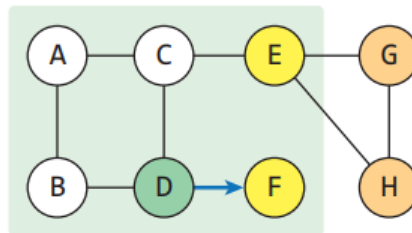
(c) B→D

큐 내용: CD



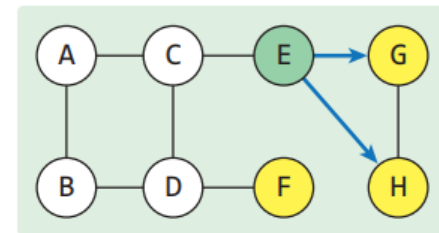
(d) C→E

큐 내용: DE



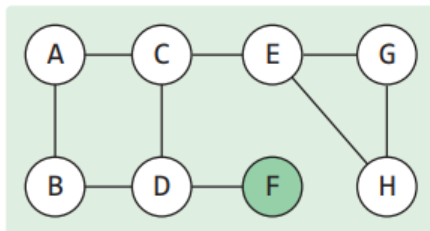
(e) D→F

큐 내용: EF



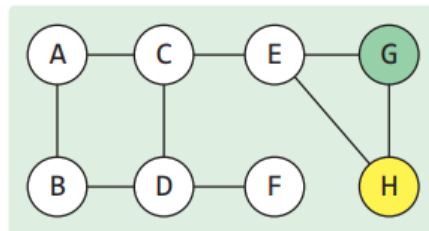
(f) E→G, H

큐 내용: FGH



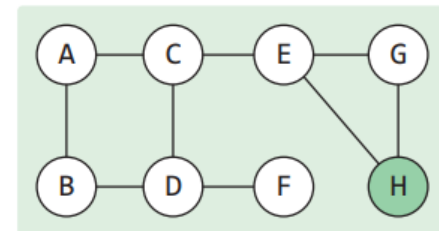
(g) F에서는 모두 방문했음

큐 내용: GH



(h) G에서는 모두 방문했음

큐 내용: H

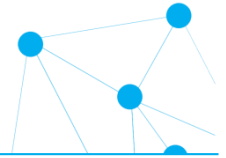


(i) H에서도 모두 방문했음

큐 공백상태 → 탐색 완료

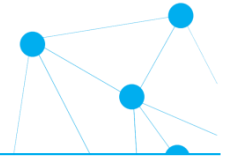
방문 순서: ABCDEFGH

탐색 알고리즘 성능



- 깊이 우선 탐색 / 너비 우선 탐색
 - 인접 행렬 표현: $O(n^2)$
 - 인접 리스트로 표현: $O(n + e)$
- 완전 그래프와 같은 조밀 그래프 → 인접 행렬이 유리
- 희소 그래프 → 인접리스트가 유리

10.4 연결 성분 검사

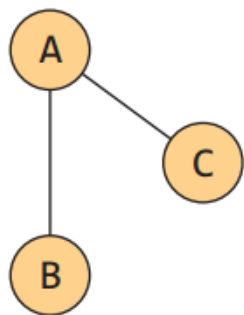


- 연결 성분이란?
- 연결 성분 검사 알고리즘
 - 인접 리스트 구현

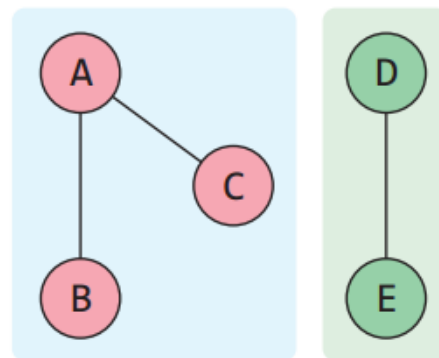
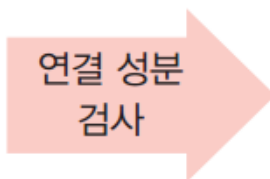
연결 성분이란?



- 최대로 연결된 부분 그래프들을 구함
 - DFS 또는 BFS를 반복적으로 이용



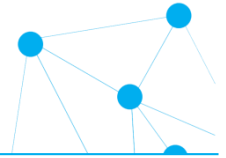
원래의 그래프



부분 그래프들

A	1
B	1
C	1
D	2
E	2
label	

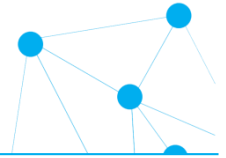
연결 성분 검사 알고리즘



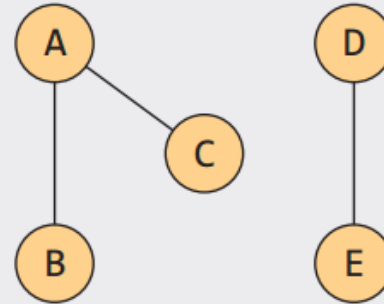
```
def find_connected_component(graph) :  
    visited = set()                                # 이미 방문한 정점 집합  
    colorList = []                                # 부분 그래프별 정점 리스트  
  
    for vtx in graph :                             # 그래프의 모든 정점들에 대해  
        if vtx not in visited :                   # 방문하지 않은 정점이 있으면  
            color = dfs_cc(graph, [], vtx, visited) # 새로운 컬러 리스트  
            colorList.append( color )              # 컬러 리스트 추가  
  
    print("그래프 연결성분 개수 = %d " % len(colorList))  
    print(colorList)                               # 정점 리스트들을 출력
```

```
def dfs_cc(graph, color, vertex, visited):  
    if vertex not in visited :                     # 아직 칠해지지 않은 정점에 대해  
        visited.add(vertex)                        # 이제 방문했음  
        color.append(vertex)                       # 같은 색의 정점 리스트에 추가  
        nbr = graph[vertex] - visited              # nbr: 차집합 연산 이용  
        for v in nbr:                              #  $v \in \{\text{인접정점}\} - \{\text{방문정점}\}$   
            dfs_cc(graph, color, v, visited)        # 순환 호출  
    return color                                    # 같은 색의 정점 리스트 반환
```

테스트 프로그램



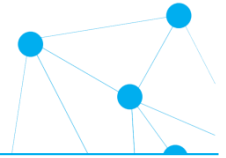
```
mygraph = { "A" : set(["B","C"]),  
            "B" : set(["A"]),  
            "C" : set(["A"]),  
            "D" : set(["E"]),  
            "E" : set(["D"])  
          }
```



```
print('find_connected_component: ')  
find_connected_component(mygraph)
```

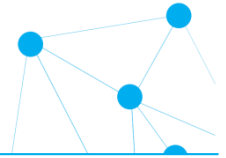
```
C:\WINDOWS\system32\cmd.exe  
find_connected_component:  
그래프 연결성분 개수 = 2  
[['A', 'B', 'C'], ['D', 'E']]
```

10.5 신장 트리

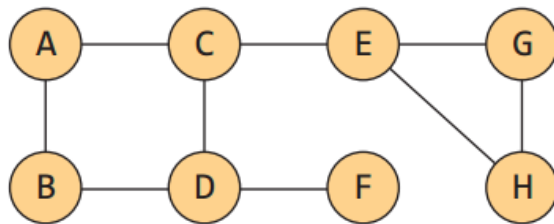


- 신장 트리란?
- 신장 트리 알고리즘
 - 인접 리스트 구현

신장 트리란?



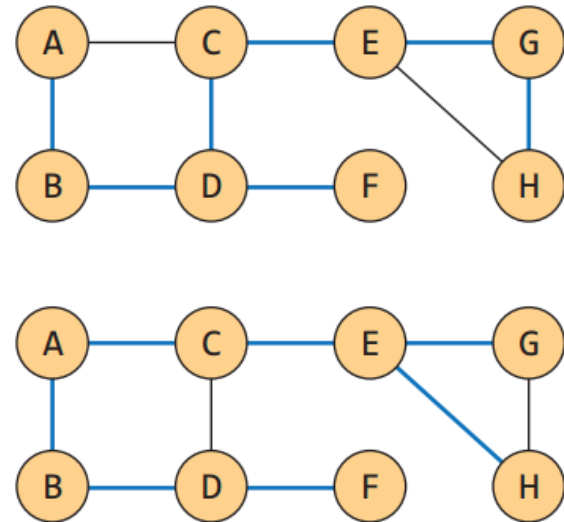
- 그래프 내의 모든 정점을 포함하는 트리
 - 사이클을 포함하면 안됨, 간선의 수 = $n-1$



연결 그래프

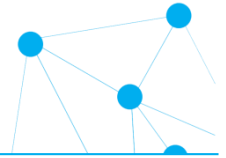
깊이우선
탐색의 간선

너비우선
탐색의 간선



신장 트리의 예

신장 트리 알고리즘

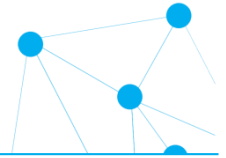


```
def bfsST(graph, start):  
    visited = set([start])  
    queue = collections.deque([start])  
    while queue:  
        v = queue.popleft()  
        nbr = graph[v] - visited  
        for u in nbr:  
            print("(" , v , "," , u , ") ", end="")  
            visited.add(u)  
            queue.append(u)
```

```
# 맨 처음에는 start만 방문한 정점임  
# 파이썬 컬렉션의 덱 생성(큐로 사용)  
# 공백이 아닐 때 까지  
# 큐에서 하나의 정점 v를 빼냄  
# nbr = {v의 인접정점} - {방문정점}  
# 갈 수 있는 모든 인접 정점에 대해  
# (v,u)간선 추가  
# 이제 u는 방문했음  
# u를 큐에 삽입
```

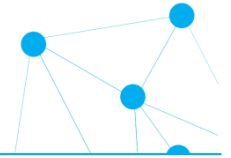
```
C:\WINDOWS\system32\cmd.exe  
( A , C ) ( A , B ) ( C , E ) ( C , D ) ( E , G ) ( E , H ) ( D , F )
```

10.6 위상 정렬



- 위상 정렬이란?
- 신장 트리 알고리즘
 - 인접 행렬 구현

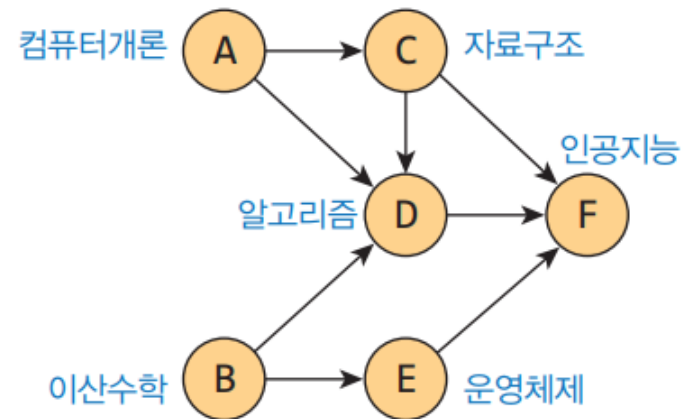
위상 정렬이란?



- 방향 그래프에 대해 정점들의 선행 순서를 위배하지 않으면서 모든 정점을 나열하는 것

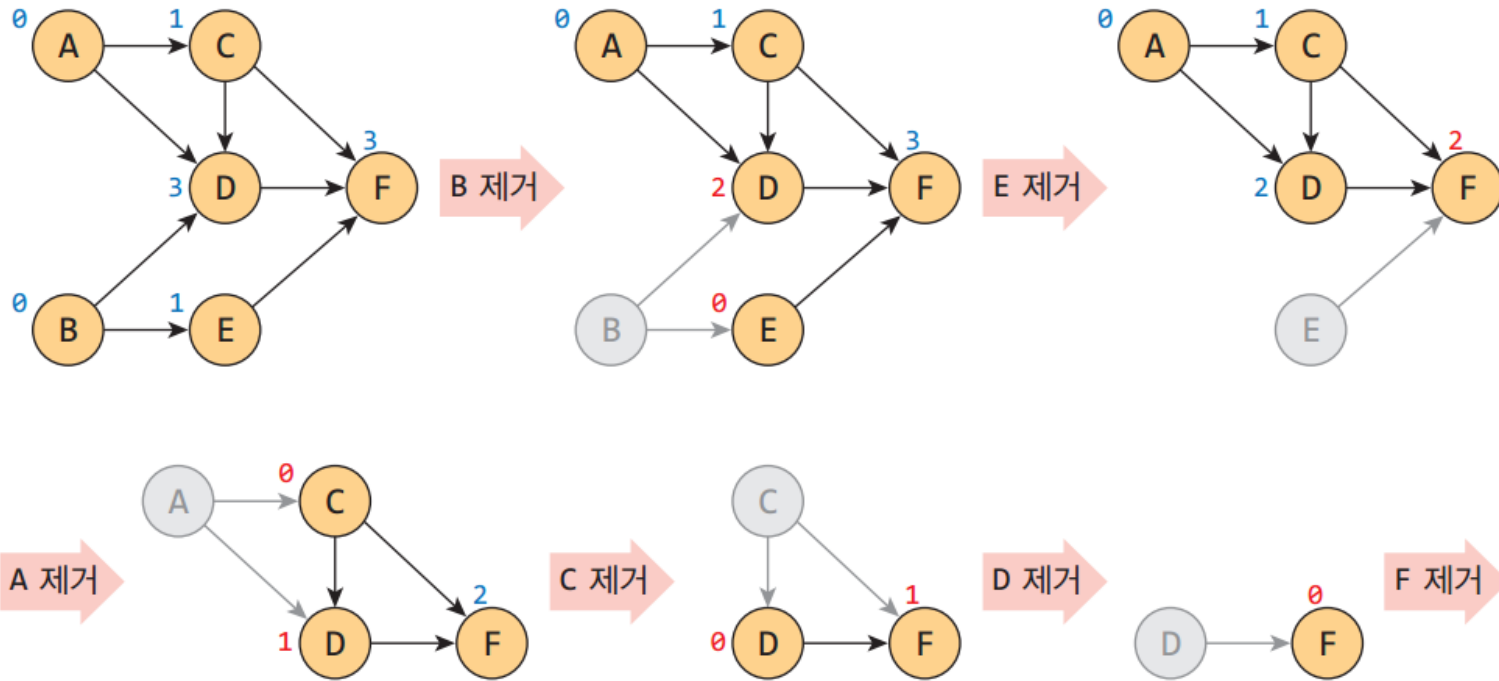
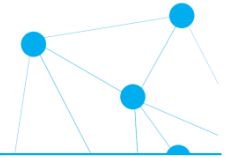
과목번호	과목명	선수과목
A	컴퓨터개론	없음
B	이산수학	없음
C	자료구조	A
D	알고리즘	A, B, C
E	운영체제	B
F	인공지능	C, D, E

교과목의 선후수 관계 표

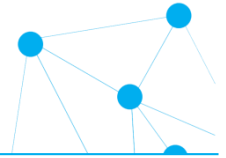


방향 그래프로 표시한 선후수 관계

위상 정렬 과정

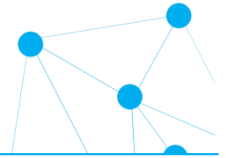


위상 정렬 알고리즘

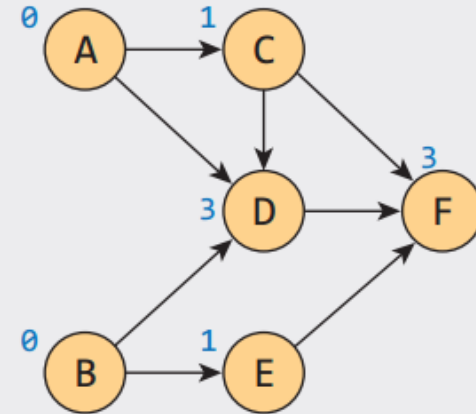


```
def topological_sort_AM(vertex, graph) :  
    n = len(vertex)  
    inDeg = [0] * n                # 정점의 진입차수 저장  
  
    for i in range(n) :  
        for j in range(n) :  
            if graph[i][j] > 0 :  
                inDeg[j] += 1      # 진입차수를 1 증가시킴  
  
    vlist = []                     # 진입차수가 0인 정점 리스트를 만들  
    for i in range(n) :  
        if inDeg[i]==0 :  
            vlist.append(i)  
  
    while len(vlist) > 0 :         # 리스트가 공백이 아닐 때 까지  
        v = vlist.pop()           # 진입차수가 0인 정점을 하나 꺼냄  
        print(vertex[v], end=' ') # 화면 출력  
  
        for u in range(n) :  
            if v != u and graph[v][u] > 0 :  
                inDeg[u] -= 1      # 연결된 정점의 진입차수 감소  
                if inDeg[u] == 0 : # 진입차수가 0이면  
                    vlist.append(u) # vlist에 추가
```

테스트 프로그램



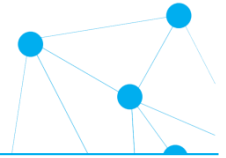
```
vertex = ['A', 'B', 'C', 'D', 'E', 'F']  
graphAM = [ [ 0, 0, 1, 1, 0, 0 ],  
             [ 0, 0, 0, 1, 1, 0 ],  
             [ 0, 0, 0, 1, 0, 1 ],  
             [ 0, 0, 0, 0, 0, 1 ],  
             [ 0, 0, 0, 0, 0, 1 ],  
             [ 0, 0, 0, 0, 0, 0 ] ]  
  
print('topological_sort: ')  
topological_sort_AM(vertex, graphAM)  
print()
```

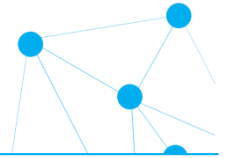


C:\WINDOWS\system32\cmd.exe

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

10장 연습문제, 실습문제





감사합니다!