

# 자료구조



## 그래프 응용



한국기술교육대학교  
온라인평생교육원

## 학습내용

- 그래프의 탐색
- 연결 성분 검사
- 신장 트리
- 위상 정렬

## 학습목표

- 그래프의 탐색 방법을 설명할 수 있다.
- 연결 성분 검사에 대해 설명할 수 있다.
- 신장 트리에 대해 설명할 수 있다.
- 위상 정렬에 대해 설명할 수 있다.

# 그래프 탐색



그래프 응용

그래프의 탐색

## 1 그래프 탐색의 개념

### 그래프 탐색

- 시작 정점부터 차례대로 모든 정점들을 한 번씩 방문하는 작업
- 가장 기본적인 연산
- 많은 문제들이 단순히 탐색만으로 해결



그래프 응용

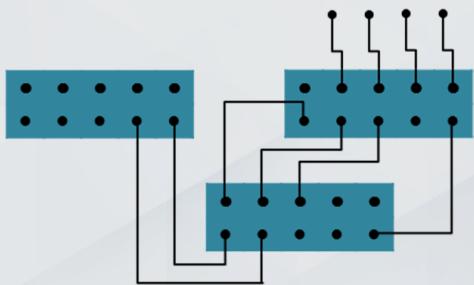
그래프의 탐색

## 1 그래프 탐색의 개념

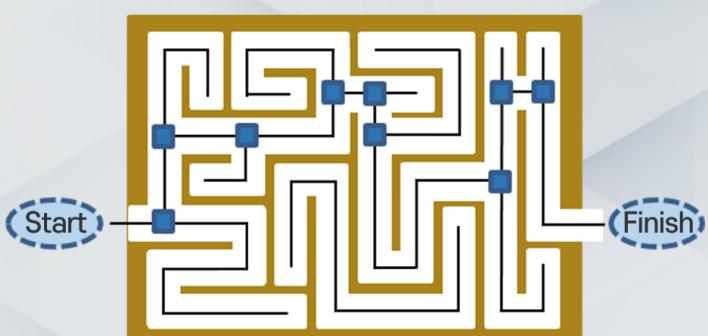
### ● 그래프 탐색의 예

▶ 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

# 그래프 탐색



## 그래프 응용

### 그래프의 탐색

#### 1 그래프 탐색의 개념

##### ● 그래프 탐색 전략

▶ 모든 정점을 체계적으로 방문할 수 있어야 함 → **완전 탐색**

방법 1	깊이우선탐색(Depth First Search)
방법 2	너비우선탐색(Breadth First Search)

## 그래프 응용

### 그래프의 탐색

#### 2 깊이우선탐색 알고리즘

##### ● 깊이우선탐색 알고리즘의 개념

DFS(Depth-First Search)
<ul style="list-style-type: none"> <li>▪ 한 방향으로 끝까지 가다가 더 이상 갈 수 없게 되면 가장 가까운 갈림길로 돌아와서 다른 방향으로 다시 탐색 진행</li> <li>▪ 되돌아가기 위한 방법?           <ul style="list-style-type: none"> <li>• 스택에 갈 수 있는 다른 길을 저장</li> <li>• 순환을 이용해 시스템 스택을 이용</li> </ul> </li> </ul>



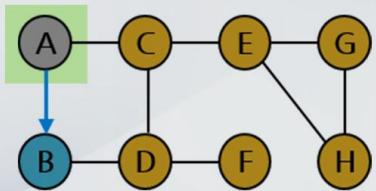
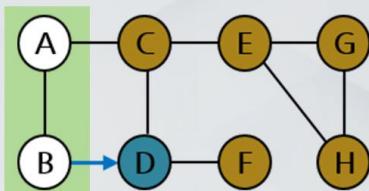
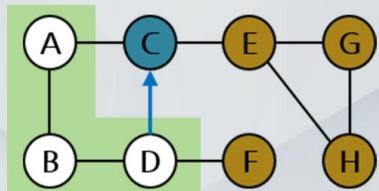
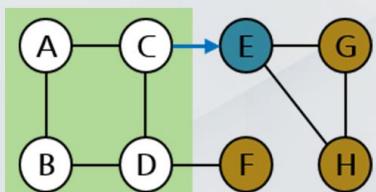
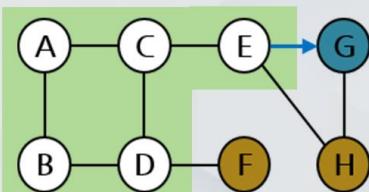
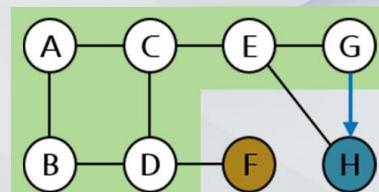
# 그래프 탐색

## 그래프 응용

### 그래프의 탐색

#### 2 깊이우선탐색 알고리즘

##### ● 깊이우선탐색 예

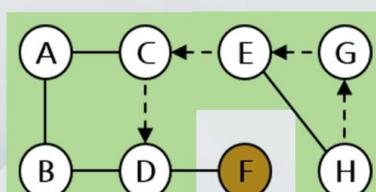
(a) A에서 시작:  $A \rightarrow B$ (b)  $B \rightarrow D$ (A는 방문했음)(c)  $D \rightarrow C$ (B는 방문했음)(d)  $C \rightarrow E$ (A, D는 방문했음)(e)  $E \rightarrow G$ (C는 방문했음)(f)  $G \rightarrow H$ (E는 방문했음)

## 그래프 응용

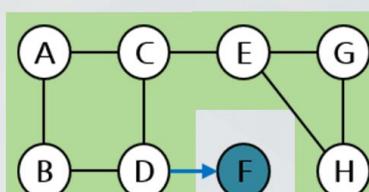
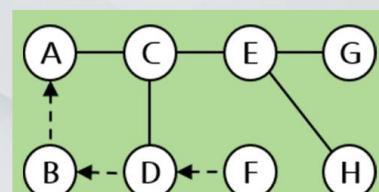
### 그래프의 탐색

#### 2 깊이우선탐색 알고리즘

##### ● 깊이우선탐색 예



(g) H에서는 모두 방문했음

(h)  $D \rightarrow F$ 

(i) F에서도 모두 방문했음

G, E, C, D순으로 되돌아감  
D에서는 가지 않은 F가 있음

D, B, A순으로 되돌아감

탐색 완료

방문 순서: ABDCEGHF

# 그래프 탐색



## 그래프 응용 그래프의 탐색

### 2 깊이우선탐색 알고리즘

#### ● 순환 구조의 알고리즘

```
dfs(v):
    v가 방문되지 않았으면
        v가 방문되었다고 표시
        v의 모든 인접 정점 u에 대해(u는 방문하지 않은 인접 정점)
            dfs(u)를 순환적으로 호출
```

```
def dfs(graph, v, visited):
    if v not in visited :          # v가 방문되지 않았으면
        visited.add(v)           # v를 방문했다고 표시
        print(v, end=' ')
        nbr = graph[v] - visited # v의 인접 정점 리스트

    for u in nbr:                 # v의 모든 인접 정점에 대해
        dfs(graph, u, visited)   # 순환 호출
```

## 그래프 응용 그래프의 탐색

### 3 너비우선탐색 알고리즘

#### ● 너비우선탐색 알고리즘의 개념

#### BFS(Breadth-First Search)

- 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법
- 큐를 사용하여 구현



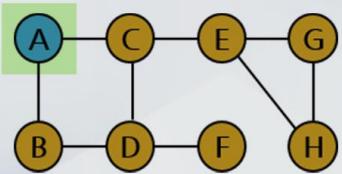
# 그래프 탐색

## 그래프 응용

### 그래프의 탐색

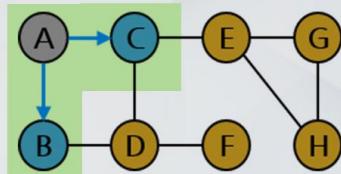
#### 3 너비우선탐색 알고리즘

##### ● 너비우선탐색 예



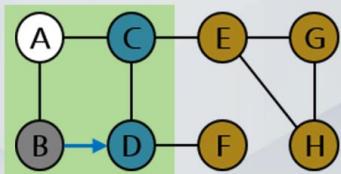
(a) A에서 시작

큐 내용: A



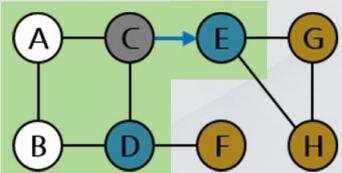
(b) A→B, C

큐 내용: BC



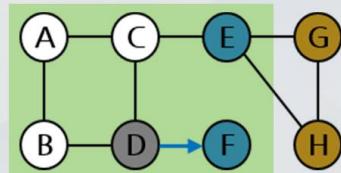
(c) B→D

큐 내용: CD



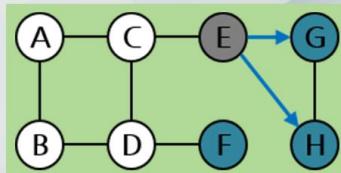
(d) C→E

큐 내용: DE



(e) D→F

큐 내용: EF



(f) E→G, H

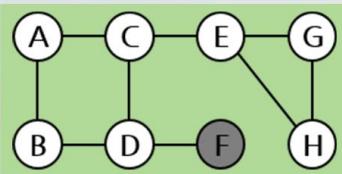
큐 내용: FGH

## 그래프 응용

### 그래프의 탐색

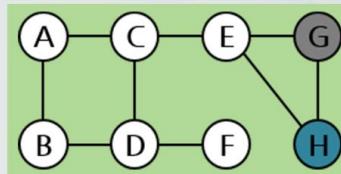
#### 3 너비우선탐색 알고리즘

##### ● 너비우선탐색 예



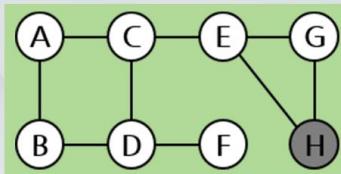
(g) F에서는 모두 방문했음

큐 내용: GH



(h) G에서는 모두 방문했음

큐 내용: H



(i) H에서도 모두 방문했음

큐 공백상태 → 탐색 완료  
방문 순서: ABCDEFGH

# 그래프 탐색



## 그래프 응용

### 그래프의 탐색

#### 3 너비우선탐색 알고리즘

##### ● 알고리즘(자연어)

bfs( $s$ ):

    공백 상태의 큐  $Q$ 에 시작 정점  $s$ 를 넣음.

$Q$ 가 공백이 아닐 때 까지:

        큐에서 하나의 정점  $v$ 의 꺼냄

$v$ 를 출력

$v$ 의 모든 인접 정점  $u$ 에 대해

$u$ 를 방문하지 않았으면

                방문했다고 표시

$u$ 를  $Q$ 에 삽입

## 그래프 응용

### 그래프의 탐색

#### 3 너비우선탐색 알고리즘

##### ● 알고리즘(파이썬)

```
import collections
```

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

# 맨 처음에는 start만 방문한 정점임

# 파이썬 컬렉션의 데 객체 생성(큐로 사용)

# 큐에 항목이 있을 때까지

# 큐에서 하나의 정점  $v$ 를 빼냄

#  $v$ 는 방문했음을 출력

#  $nbr = \{v\text{의 인접정점}\} - \{\text{방문정점}\}$

# 갈 수 있는 모든 인접 정점에 대해

# 이제  $u$ 는 방문했음

#  $u$ 를 큐에 삽입



# 그래프 탐색

## 그래프 응용

### 그래프의 탐색

#### 3 탐색 알고리즘 성능

##### ● 탐색 알고리즘 성능의 특징

- ▶ 깊이우선탐색 / 너비우선탐색
  - 인접 행렬 표현:  $O(n^2)$
  - 인접 리스트로 표현:  $O(n+e)$
- ▶ 완전 그래프와 같은 조밀 그래프 → 인접 행렬이 유리
- ▶ 희소 그래프 → 인접 리스트가 유리

# 그래프 탐색

```

graphSearchDicSet.py - Python 3.6 (64-bit)
graphSearchDicSet.py

1 def dfs(graph, v, visited):
2     if v not in visited:
3         visited.add(v)
4         print(v, end=' ')
5         nbr = graph[v] - visited
6
7         for u in nbr:
8             dfs(graph, u, visited)
9
10    import collections
11
12    def bfs(graph, start):
13        visited = set([start])
14        queue = collections.deque([start])
15
16        while queue:
17            v = queue.popleft()
18            print(v, end=' ')
19            nbr = graph[v] - visited
20            for u in nbr:
21                visited.add(u)
22                queue.append(u)

# v가 방문되지 않았으면
# v를 방문했다고 표시
# v를 출력
# v의 인접 정점 리스트
# v의 모든 인접 정점에 대해
# 순환 호출

# 맨 처음에는 start만 방문한 정점임
# 파이썬 컬렉션의 덱 객체 생성(큐로 사용)
# 큐에 항목이 있을 때 까지
# 큐에서 하나의 정점 v를 빼냄
# v는 방문했음을 출력
# nbr = {v의 인접정점} - {방문정점}
# 갈 수 있는 모든 인접 정점에 대해
# 이제 u는 방문했음
# u를 큐에 삽입

```

## 실습 단계

그래프 표현 방식 -> 전체는 dictionary 이용, 각각의 노드는 키값이 정점, value는 정점의 인접 정점 집합 이용

파이썬으로 그래프를 표현하면 compact하고 시각적으로 그래프의 형태를 쉽게 이해할 수 있음

Collections모듈을 사용해서 deque을 queue로 이용, 삽입 시 append 연산, 삭제 시 popleft 연산

nbr에서의 차 집합 연산을 한 부분은 추상적으로 개념화된 코드

깊이우선탐색에서는 공백집합(set())을 전달해야 함. -> 순환 호출 때문에

너비우선탐색에서는 반복구조로 구현했기 때문에 다시 호출할 일이 없어서 함수 내에서 visited 정의

앞의 예시와 실행결과가 다를 수 있음

-> 예시 설명에서는 알파벳 순으로 넣었지만, 실제적으로는 어떻게 될 지 알 수 없기 때문



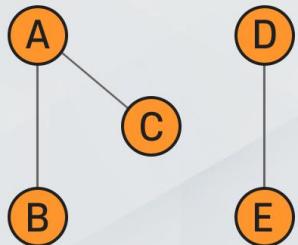
# 연결 성분 검사

그래프 응용 연결 성분 검사

## 1 연결 성분의 개념

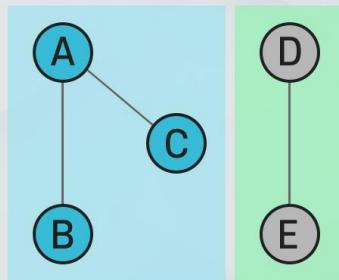
### 연결 성분(Connected Component)

원래 그래프의 최대로 연결된 부분 그래프들



원래의 그래프

연결 성분 검사



부분 그래프들

A	1
B	1
C	1
D	2
E	2

label

그래프 응용 연결 성분 검사

## 1 연결 성분의 개념

### ● 연결 성분 검사의 전략

DFS나 BFS를 반복적으로 적용

- 연결된 정점들 모두 출력

아직 연결되지 않은 정점을 다시 선택해 위의 과정 반복

# 연결 성분 검사



## 그래프 응용

### 연결 성분 검사

#### 2 연결 성분 검사 알고리즘

- 정점 v에 연결된 모든 정점 리스트 color를 구하는 함수

DFS 기반 구현	dfs() 함수 수정
dfs_cc()	color 추가: 같은 색을 칠하는 정점들의 집합

```
def dfs_cc(graph, color, v, visited):
    if v not in visited :
        visited.add(v)
        color.append(v)
        nbr = graph[v] - visited
        for u in nbr:
            dfs_cc(graph, color, u, visited)
```

# 아직 칠해지지 않은 정점에 대해  
# 이제 방문했음  
# 같은 색의 정점 리스트에 추가  
# nbr: 차집합 연산 이용  
# u ∈ {인접정점} - {방문정점}  
# 순환 호출

## 그래프 응용

### 연결 성분 검사

#### 2 연결 성분 검사 알고리즘

- 방문하지 않은 정점이 없도록 dfs\_cc() 반복

```
def find_connected_component(graph) :
    visited = set()
    colorList = []

    for vtx in graph :
        if vtx not in visited :
            color = []
            dfs_cc(graph, color, vtx, visited)
            colorList.append( color )

    print("그래프 연결성분 개수 = %d " % len(colorList))
    print(colorList)
```

# 이미 방문한 정점 집합  
# 부분 그래프 별 정점 리스트  
  
# 그래프의 모든 정점들에 대해  
# 방문하지 않은 정점이 있으면  
# vtx와 연결된 모든 정점 리스트  
# 새로운 컬러 리스트  
# 컬러 리스트 추가  
  
# 정점 리스트들을 출력

# 연결 성분 검사



그래프 응용

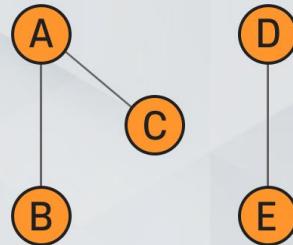
연결 성분 검사

## 2 연결 성분 검사 알고리즘

### ● 테스트 프로그램

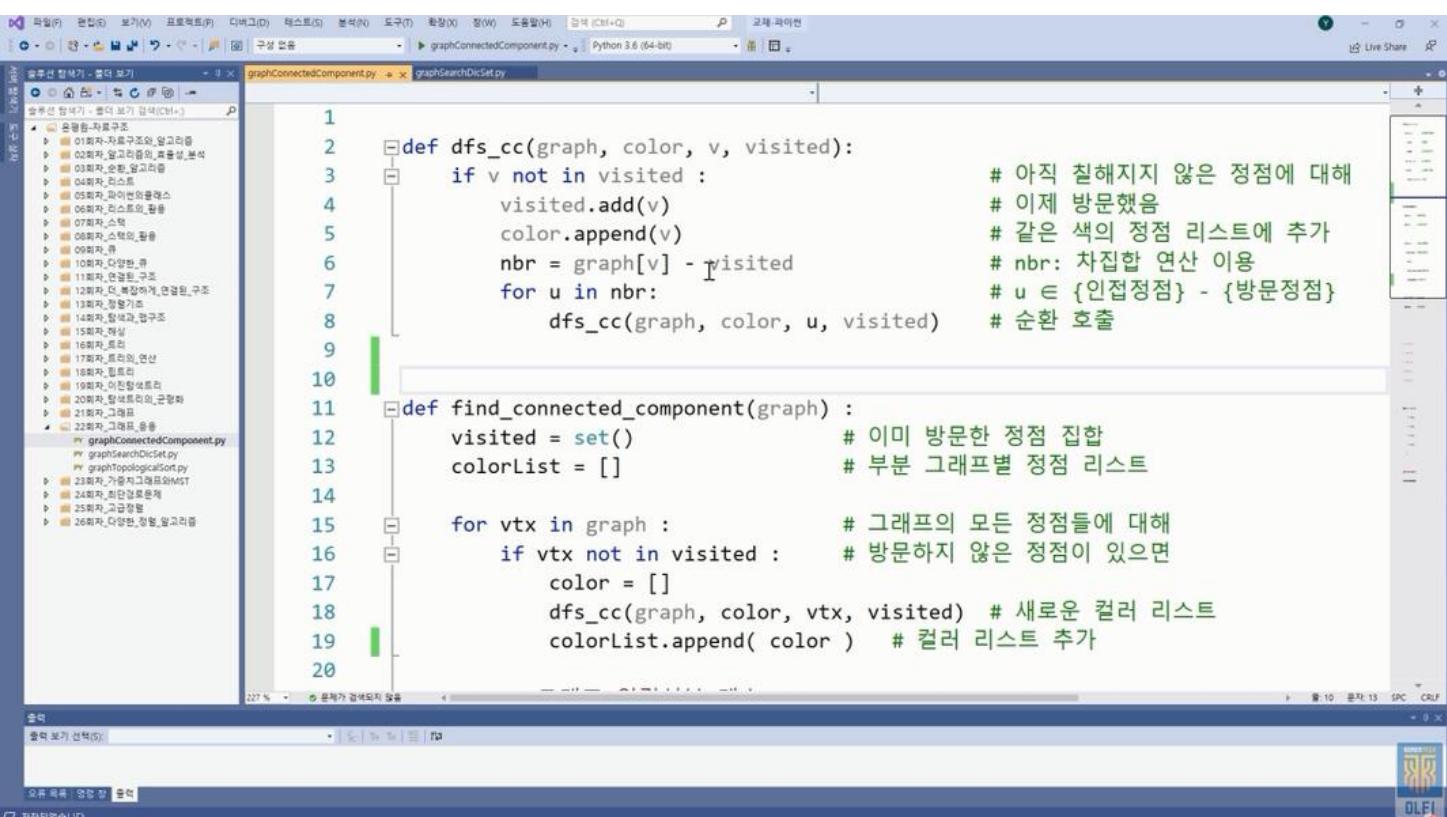
```
mygraph = { "A" : set(["B", "C"]),
            "B" : set(["A"]),
            "C" : set(["A"]),
            "D" : set(["E"]),
            "E" : set(["D"])
        }

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']]
```

# 연결 성분 검사



```

1  def dfs_cc(graph, color, v, visited):
2      if v not in visited :
3          visited.add(v)
4          color.append(v)
5          nbr = graph[v] - visited
6          for u in nbr:
7              dfs_cc(graph, color, u, visited)
8
9
10
11 def find_connected_component(graph) :
12     visited = set()           # 이미 방문한 정점 집합
13     colorList = []            # 부분 그래프별 정점 리스트
14
15     for vtx in graph :        # 그래프의 모든 정점들에 대해
16         if vtx not in visited : # 방문하지 않은 정점이 있으면
17             color = []
18             dfs_cc(graph, color, vtx, visited) # 새로운 컬러 리스트
19             colorList.append( color ) # 컬러 리스트 추가
20
21
22
23
24
25
26

```

실습 단계

dfs에 color가 추가, color는 정점리스트

실행 결과

# 신장 트리



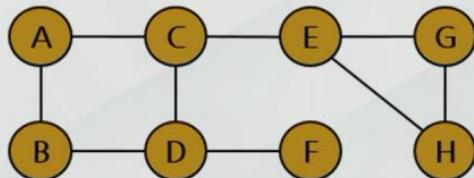
## 그래프 응용

### 신장 트리

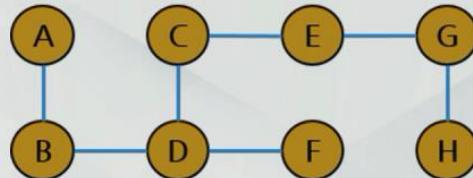
#### 1 신장 트리의 개념

##### 신장 트리(Spanning Tree)

- 연결 그래프 내의 모든 정점을 포함하는 트리
- 사이클을 포함하면 안됨
- 간선의 수 =  $n-1$
- 신장 트리의 예



연결 그래프 G



G의 신장 트리의 예



## 그래프 응용

### 신장 트리

#### 2 신장 트리 알고리즘

##### 전략

깊이우선이나 너비우선탐색 도중에 만나는 간선들을 모음

##### ▶ BFS를 이용한 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, {}) ".format(v, " ", u), end="")
            visited.add(u)
            queue.append(u)
```

# 맨 처음에는 start만 방문한 정점임  
# 파이썬 컬렉션의 덱 생성(큐로 사용)  
# 공백이 아닐 때까지  
# 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 , D ) ( C , E ) ( D , F ) ( E , H ) ( E , G )
```



# 위상 정렬

## 그래프 응용

### 위상 정렬

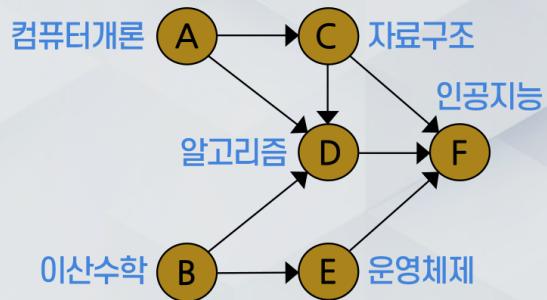
#### 1 위상 정렬의 개념

##### 위상 정렬(Topological Sort)

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

## 그래프 응용

### 위상 정렬

#### 2 위상 정렬 알고리즘

##### 위상 정렬의 조건

방향 그래프에 사이클이 존재하지 않아야 함

##### 위상 정렬 전략

- 축소 정복(Decrease-And-Conquer) 전략
- 진입 차수가 0인 정점을 반복적으로 그래프에서 삭제
  - 이때, 관련된 간선들도 삭제하고 이에 따른 나머지 정점들의 진입 차수를 갱신

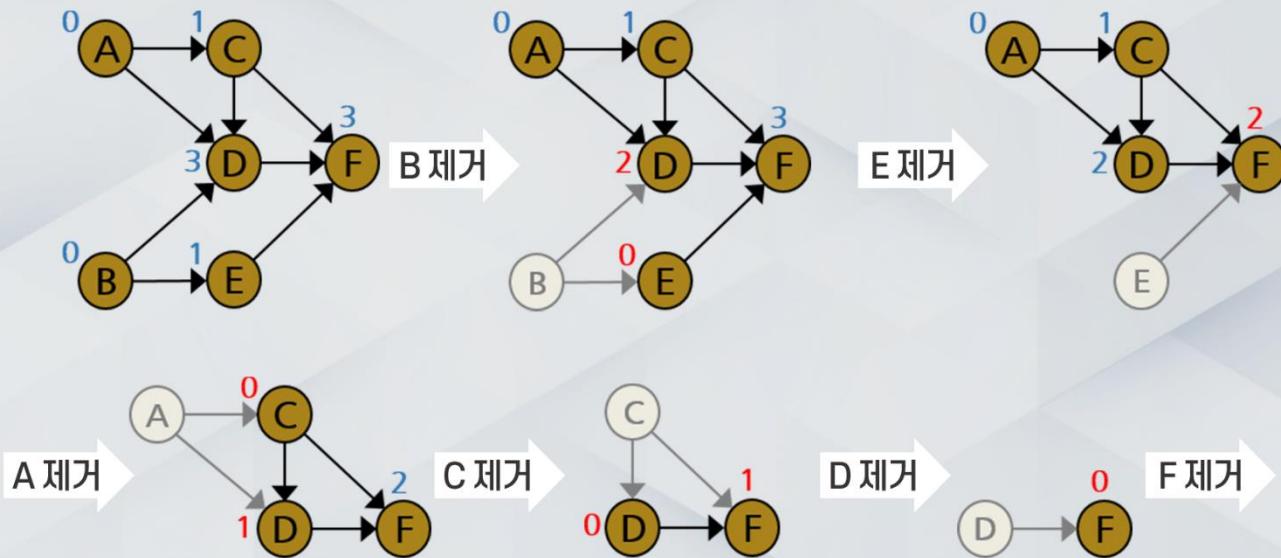
# 위상 정렬



## 그래프 응용

### 위상 정렬

#### 2 위상 정렬 알고리즘



## 그래프 응용

### 위상 정렬

#### 2 위상 정렬 알고리즘

- 그래프를 인접 행렬로 표현한 경우

▶ 초기화 과정

```
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 만듦
    for i in range(n) :
        if inDeg[i]==0 :
            vlist.append(i)
```



# 위상 정렬

## 그래프 응용

### 위상 정렬

#### 2 위상 정렬 알고리즘

##### ● 그래프를 인접 행렬로 표현한 경우

###### ▶ 위상 정렬 과정

```

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

    for u in range(n) :
        if v != u and graph[v][u] > 0 :
            inDeg[u] -= 1
            if inDeg[u] == 0 :
                vlist.append(u)
    # 연결된 정점의 진입차수 감소
    # 진입차수가 0이면
    # vlist에 추가
  
```

## 그래프 응용

### 위상 정렬

#### 2 위상 정렬 알고리즘

##### ● 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_AM: ')
topological_sort_AM(vertex, graphAM)
print()
  
```



```

C:\WINDOWS\system32\cmd.exe
topological_sort:
B E A C D F
  
```

# 위상 정렬



Code editor showing Python code for Topological Sort:

```

    67     for u in range(n) :
    68         if v != u and graph[v][u] > 0 :
    69             inDeg[u] -= 1           # 연결된 정점의 진입차수 감소
    70             if inDeg[u] == 0 :      # 진입차수가 0이면
    71                 vlist.append(u) # vlist에 추가
    72
    73     vertex = ['A', 'B', 'C', 'D', 'E', 'F' ]
    74     graphAM = [ [ 0, 0, 1, 1, 0, 0 ],
    75                  [ 0, 0, 0, 1, 1, 0 ],
    76                  [ 0, 0, 0, 1, 0, 1 ],
    77                  [ 0, 0, 0, 0, 0, 1 ],
    78                  [ 0, 0, 0, 0, 0, 1 ],
    79                  [ 0, 0, 0, 0, 0, 0 ] ]
    80
    81     print('topological_sort_AM: ')
    82     topological_sort_AM(vertex, graphAM)
    83     print()
    84

```

실습 단계
그래프를 인접 행렬로 표현한 것
<u>inDeg</u> -> 정점의 진입 차수를 위한 리스트
<u>vList</u> -> 진입 차수가 0인 정점 리스트
실행결과 -> 모든 정점을 방문할 수 있음