7.1 소개

현실의 많은 문제들이 그래프와 연관

- 그래프 "운행" (traversal) 만으로 풀 수 있는 문제 ⇒ 쉬운 문제
- 그래프 운행 + 몇 가지 작업을 통해 다항 시간 내 해결 가능 ⇒ 중간 정도 어려운 문제
- 다항 시간 내의 풀이가 알려지지 않은 문제 ⇒ 어려운 문제

7.2 정의와 표현

Graph : 점 (정점이나 유한 노드)의 유한 집합

↳ 어떤 점은 선이나 화살표 (에지)로 연결

↳ 무방향 (무향) : 무향 그래프 (undirected)
유방향 (유향) : 유향 그래프 (directed)

• 기본적인 그래프 정의

〈정의 7.1〉 유향 그래프

directed graph $G = (V, E)$

V: 정점의 집합

E: V에 속해 있는 원소의 순서쌍의 집합 ⇒ edge의 정의

E의 원소 : 유향 에지 / 아크 (arc) $e \subseteq E$ $e = (v, w)$ $v \rightarrow w$

v: 꼬리 (tail)   w: 머리 (head)   간단하게 vw

〈정의 7.2〉 무향 그래프 (undirected graph)

기본적인 정의는 비슷   무향 그래프에서의 에지 : V에 속한 서로 다른 원소들의 순서없는 쌍의 집합

↳ 각 에지는 원소의 수가 2인 V의 부분집합

무향 그래프 에지의 표현 : $v - w$ $\{v, w\}$, vw $(vw = wv)$

〈정의 7.3〉 부그래프, 유향 대칭 그래프, 완전 그래프

• 부그래프 : 그래프 $G = (V, E)$의 부그래프

일종의 부분집합

$V' \subseteq V$, $E' \subseteq E$인 $G' = (V', E')$

↳ $E' = V' \times V'$

· 유향 대칭 그래프 (directed symmetric) : 임의의 에지 vw에 대해 반대 방향의 에지 wv를 가지는 그래프

$v \longrightarrow w \Rightarrow w \longrightarrow v$ 를 항상 가짐 → 무향 그래프는 그 정의에 의해 항상 대칭 그래프를 가짐

· 완전 그래프 (Complete Graph) : 모든 정점쌍에 대해 에지를 가지는 그래프 (일반적으로 무향)

∘ 에지 vw는 정점 v, w와 인접 (incident) 하다.

〈정의 7.4〉 인접 관계 (adjacency relation)

정점 집합에 대한 관계 인접 관계 A → 새로운 관계

$v, w \in V$, $vw \in E$ 이면 vAw → v에서 시작하여 edge 1개를 지나 w에 도달가능

"G가 무향이면 A는 대칭적"

〈정의 7.5〉 그래프에서의 경로 (path)

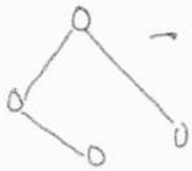
$G = (V, E)$ v에서 w까지의 경로 $v = v_0, \cdots v_k = w$인 에지의 열

$v_0v_1, v_1v_2, \cdots, v_{k-1}v_k$ ⇒ 경로의 길이가 k인 경로 (v에서 w까지) → $v_0 \sim v_k$ 까지 모두 다르면 단순 경로 (simple path)

v에서 w까지 경로가 존재하면 도달가능 (reachable)

〈정의 7.6〉 연결된 그래프, 강력 연결 그래프

↳ 무향/유향에 대한 연결성 (Connectivity)의 정의가 다름!



→ 완전 그래프는 아니지만 연결 그래프는 맞다

· 임의의 정점쌍 v, w에 대해 v에서 w까지의 경로가 존재 (무향 그래프) ⇒ 연결 (connected) 그래프

↳ 유향 그래프일 경우 강연결 그래프 (strongly connected graph)

↳ 무향의 경우 반대의 경로가 항상 존재, 유향은 X

〈정의 7.7〉 그래프에서의 사이클 (cycle) → 유/무향에서의 정의가 다름

· 유향 그래프에서의 사이클 : 처음 정점과 마지막 정점이 같은 경로

↳ 단순 사이클 (simple cycle) : 첫 째와 마지막이 같은 것을 제외하고 어떤 정점도 중복이 없는 것

· 무향 그래프에서의 사이클 : 같은 에지가 두 번 이상 나타나면 항상 같은 방향!

$v_i = x$, $v_{i+1} = y$ 인 에지 $(0 \le j < k)$ $v_j = y$, $v_{j+1} = x$인 에지 존재 X

· 무향 포레스트 (Undirected forest) : 사이클이 없는 무향 그래프 → 무향 포레스트가 연결성이 있으면 무향 트리! (자유 트리)

사이클이 없는 유향 그래프 (DAG ; Directed Acyclic Graph)

트리 무향 트리 (사이클이 없는 무향 그래프 + 연결성) → 여기에 별도의 루트를 지정해주면 부모-자식 관계 유도 (rooted tree)

사이클의 중요도에서의 유·무향 차이
- 사이클이 중요하지 않다면 유·무향은 같다 (크게 보면)
- 사이클이 중요하다면 매우 다름   o ⇄ o (사이클 o)   o — o (사이클 x)

〈정의 7.8〉 연결 성분 (Connected component)

무향 그래프 G의 연결 성분은 G의 최대 연결 부그래프

"최대"의 의미

→ 굳이 최대 정점에서로 가질 필요는 없음.

한 그래프 집합에서의 최대 ⇒ 그것이 집합에 속한 어떤 다른 부그래프가 되지 않았다는 것 (같은 것 제외)

↳ G의 모든 연결된 부그래프의 집합

성분 Component (그래프 추상적 구조)에서 '최대'라는 의미 내포

무향 그래프 연결 X → 서로 다른 연결 성분으로 분할

가중치 에지에 붙은 숫자

〈정의 7.9〉 가중치 그래프 (weighted graph)

가중치 그래프 : V, E, W 의 쌍 (V, E, W)의 쌍

(V, E)는 그래프   W: E에서 실수 집합 R로 매핑되는 함수 → 에지 e에 대한 $w(e)$를 e의 가중치로.

◦ 그래프 표현과 자료구조

$G=(V,E)$ $n=|V|$, $m=|E|$, $V=\{v_0, v_1, \cdots, v_n\}$

· 인접 행렬의 표현

G ⇒ 인접행렬 (adjacency matrix) $n \times n$ 행렬 $A=a_{ij}$로 표현 가능

$$A=a_{ij} \begin{cases} 1 & v_{ij} \in E \text{ 인 경우} \\ 0 & \text{그렇지 않은 경우} \end{cases}$$

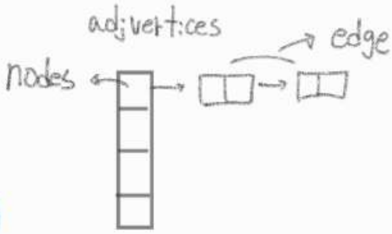
무향 그래프에 대한 인접 행렬은 대칭적

· 가중치 그래프의 경우

$$a_{ij} \begin{cases} w(v_i v_j) \\ c \to 0 \text{ 혹은 } \infty \text{ (혹은 임의의 수)} \end{cases}$$

가능한 에지의 수 : 유향 : n^2, 무향 : $n(n-1)/2$ $\Omega(n^2)$

• 인접 리스트의 배열 표현 (adjacency list)



정점 번호가 인덱스가 되는 배열 → 연결 리스트를 가짐.

↳ i번째 배열의 원소는 그 정점에서 나가는 모든 에지 → v_i가 꼬리 (유향) / 에지가 v_i에 인접(무향)

• 인접 리스트 구조의 장점 ; G에 존재하지 않는 에지는 표현에서도 존재 X.

• 무향 그래프의 경우 각 에지는 두 번 표현.   n개의 인접 리스트에 2m개의 에지

↳ uw wu 둘 다 표현

7.3 그래프 운행

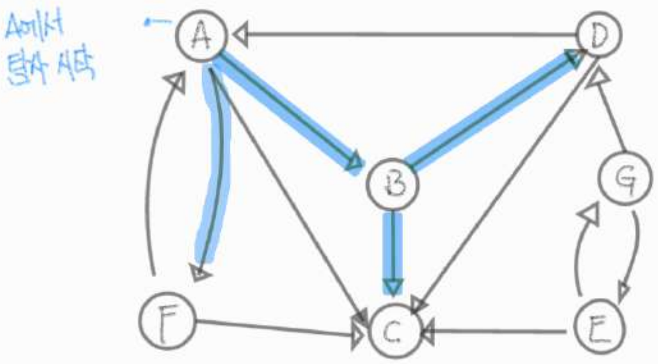
그래프를 푸는 많은 알고리즘 → 정점 / 에지에 대한 탐색 필요
- 너비우선탐색 (breadth first search)
- 깊이우선탐색 (depth first search)

· 깊이 우선 탐색

트리탐색을 일반화한 것   중요한 개념
- 탐사 (exploring)
- 퇴각 (backtracking)

노드의 발견 (discovered)   에지의 체크 (checked)

원래 그래프



깊이 방향으로 쭉 가는게 깊이 우선

① AB → BC (A, B, C 발견)

② C 종점 → B로 퇴각

③ BD (D발견)   두 에지의 차이 중요!

④ DA 체크 (A 이미 발견)
   DC 체크 (C 이미 발견)

⑤ B → A로 퇴각

⑥ AC 체크, AF 탐사 (F 발견)   지름길

⑦ FA, FC 체크, A로 퇴각

⑧ 종료 (E, G 미탐사)

DFS 탐색 시 미발견된 정점으로 가는 에지들로 정의된 트리를 DFS 트리라고 함

항상 온 길로 되돌아감 (A에서 B이면 언젠가 B→A)

퇴각 후 남은 탐사: 갔던 곳은 X. ⇒ 재귀적 분할 가능

dfs outline (G, u)

u가 "발견" 되었다고 표시

uw가 G의 에지인 각 정점 w에 대해

만약 w가 미방문 되었으면,

dfs(G, w) → uw를 탐사, w 방문, 거기서 최대한 탐색, w에서 u로 퇴각

그렇지 않으면,

w를 방문하지 않고 uw를 체크

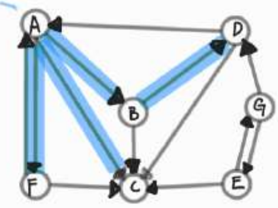
u가 "종료 되었다" 라고 표시

· 너비우선탐색의 아웃라인

정점들이 발견되는 순서 ← 깊이 우선과는 상당히 다름

⇒ 한 곳에서 동시에 독립적으로 펼쳐 나가는 탐사

A에서 시작



① A에서 갈 수 있는 모든 길 탐사 (B, C, F 발견)

② 발견된 B, C, F 에서 동일한 과정 반복

↳ 이미 발견된 정점으로 가는 에지는 체크!

↳ 깊이 우선 탐색과는 다른 트리 형성

같은 정점으로 가는 여러 경로가 존재할 때 최단 경로 선택 (최단 경로가 여러 개라면 어떻게든 승자 결정)

출발점 S에서

거리가 증가되는 순서대로 방문

↳ 최단 경로에서의 에지의 수

너비우선거리

d=0 에서 시작 → 새로운 정점이 발견되지 않을 때 까지 반복

↳ S로 부터 거리 d인 각 정점 u로부터 나가는 모든 에지를 고려

너비우선 탐색의 응용 : 출발점 S가 루트가 되는 너비 우선 스패닝 트리 찾기

스패닝 → S로 부터 도달 가능한 모든 그래프 정점 포함

트리에서 u의 깊이는 S로부터의 최소거리

너비 우선 탐색 알고리즘

Input : $G = (V, E)$, $V = \{1, 2, \ldots, n\}$ $S \in V$.

Output : 너비 우선 스패닝 트리.

breadth first search (G, n, s, parent) ← 트리용 배열

int[] color = new int[n+1] → 미발견 정점은 흰색 / 처리되지 않은 것은 회색. 처리 끝난 것은 검정

Queue Pending = create(n) ← white로 초기화

parent[s] = -1

color[s] = gray

enqueue(pending, s)

while (pending)

u = front(pending)

dequeue(pending)

u의 인접 리스트 정점 w들에 대해 ← "반복문"

if (color[w] = white)

color[w] = gray

enqueue(pending, w)

parent[w] = u.

color(u) = black

그래프 G는 n개의 정점, m개의 에지 ⇒ 비용은 $\Theta(m)$ → 모든 에지가 while에서 한번씩 처리

큐 연산 비용 $\Theta(n)$

- 깊이 우선 탐색과 너비 우선 탐색의 비교

둘 다 어느 것을 먼저 방문할 것인가 하는 모호함이 있음 → 구현에 따라 다름

↳ 효율적인 구현위해 발견한 정점에 인접한 정점들 중에 미발견 정점 유지 관리 필요

깊이 우선 탐색 : 퇴각 → 그 전에 발견된 정점으로 ⇒ 후입선출(LIFO)특성 → 스택

너비 우선 탐색 : u와 가까운 정점부터 방문 ⇒ 선입선출(FIFO) 특성 → 큐

↳ 두 알고리즘은 어디에 사용되느냐에 따라 여러 변형 확장 가능

- 깊이 우선 탐색 : 선처리 (최초 발견) / 후처리 (퇴각 시)

↳ 후처리 과정에서 들고 있는 정보가 많기 때문에 많이 활용됨 (응용 많다)

7.4 유향 그래프의 깊이 우선 탐색

깊이 우선 탐색 : 유향보다 무향에서 더 복잡

↳ dfs에서는 에지를 정확히 한 번 탐색 → 무향 그래프는 에지가 두 번 표현되어 있음

⇒ 에지를 탐사 방향으로 화살표를 두어 유향으로 변경

〈정의 7.10〉 전치 그래프 (Transpose graph)

유향 그래프 G의 전치 그래프는 G의 모든 에지 방향을 반대로 하여 생기는 그래프 G^T로 표기

DFS 기본 : "전진" 방향 탐색 ← 필요에 따라 "후진" 할수도 있음

↳ G로 부터 G^T 생성 후 G^T에서 탐색 (인접리스트 구조)

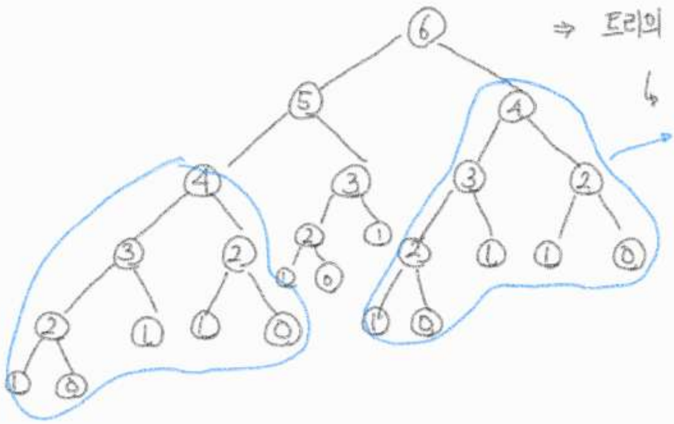
· 깊이 우선 탐색과 재귀

재귀(recursion)와 DFS 사이에는 깊은 관계가 있음 ← 각 프로시저에 대한 재귀호출 과정이 dfs와 유사

재귀 알고리즘으로 풀리는 많은 문제들의 해의 논리 구조는 트리의 깊이 우선 탐색!

ex) 피보나치 수 F(6)

재귀 구조



⇒ 트리의 postorder 순회

↳ 0에서 6까지 7개의 정점을 가진 그래프에서 깊이 우선 탐색

→ 겹치는 비효율을 막을 수 있음

ex) 체스판의 8 여왕 (서로 안겹치게)

backtrack search

↳ 이후 단계에서 겹치면 (즉, 종점을 만나면) 이전 단계로 퇴각 (퇴각 검색)

· 깊이 우선 탐색으로 연결성분 찾기

연결 성분 ← 최대 연결 부그래프

↳ DFS 응용 가능 : 한 정점에서 연결된 모든 정점, 에지찾고 남은 정점 있으면 거기서 또 탐색

인접 행렬은 없는 에지까지도 탐색 ⇒ 인접 리스트 구현이 더 효율적

연결성분 찾기 ← 두 단계로 진행
- 상위레벨 : 미발견된 정점 탐색 ⟩ 넘어갈 때 DFS 초기화
- 하위레벨 : 여기서 다시 DFS 실시

"미발견 → 발견" 상태 변경 기록 중요!

<정의 7.11> 정점의 탐색을 나타내는 세 가지 색 코드

흰 색 : 미발견　　회색 : 발견되었으나 아직 처리 종료되지 않음　　검정 : 모든 처리 종료

한 그래프에 연결성분 여러 개 ⇒ 그래프 분할 가능 (같은 성분끼리 번호 마킹 등으로 관리)

<DFS를 활용한 연결성분 탐색 알고리즘>

Input: 유향 대칭 그래프 $G=(V,E)$ ← 인접 리스트 표현, $n=|V|$

Output: 각 정점이 어느 성분에 속하는 지를 나타내는 배열 (cc)

```
Connected Components ( G (adj list) , n, cc )
    int[] Color = new int [n+1]
    int v
    모든 정점을 white로 배열 Color 초기화
    for (v=1, v≤n, v++)
        if ( Color [v] = white )
            ccDFS ( G, color, v, v, cc )
    return

ccDFS ( G, int [] Color, int v, int ccNum, int [] cc )
    int w
    IntList remAdj
    Color [v] = gray
    cc [v] = ccNum
    remAdj = ( v에서의 부그래프 (트리)
    while ( remAdj ≠ null )
        w = first (remAdj)
```

```
    if (color[w] = white)
        ccDFS(G, color, w, ccNum, cc)
    remAdj = rest(remAdj)
Color[v] = black
return
```

• 깊이 우선 트리, 깊이 우선 포레스트

<정의 7.12> 깊이 우선 탐색 트리, 깊이 우선 탐색 포레스트

유향 그래프 G에서 깊이 우선 탐색 → 미발견된 정점(흰색) ⇒ 루트있는 트리 → 깊이 우선 탐색 트리 (깊이 우선 스패닝 트리) DFS 트리

↳ 그래프가 여러 개로 분할된다면 깊이 우선 포레스트

<정의 7.13>

정점 v가 루트 → w까지의 경로 상에 존재 ⇒ v는 w의 조상(ancestor)

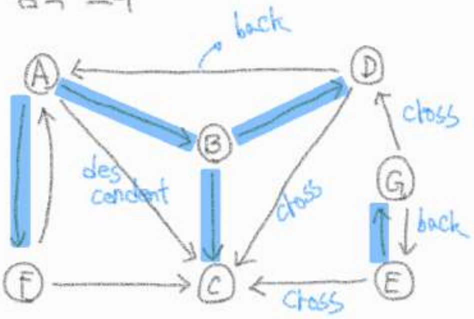
$v \neq w$ 이면 v는 w의 진조상 (proper ancestor)   v에 가장 가까운 v의 진조상 ⇒ 부모(parent)

<정의 7.14>

유향 그래프 G → 간선 방향(탐사 방향)에 따른 분류

1. vw가 탐사될 때 w가 미발견 정점 ⇒ vw는 트리 에지 v는 w의 부모
2. w가 v의 조상 ⇒ vw는 후진 에지
3. w가 v의 후손, vw가 탐사되기 전에 w가 발견 ⇒ vw는 (전진 에지) 후손 에지(descendent edge)
4. w는 v와 조상/후손 관계가 없으면 교차 에지

• 깊이 우선 탐색 트리



cross : 조상/후손이 아닐 때

↳ 전혀 다른 트리거나 트리에서 형제

· 깊이 우선 탐색 골격의 일반화

DFS → 간결·효율적인 알고리즘 구조

↳ 각 정점을 여러 번 지남 (발견 / 퇴각 ← 퇴각 후에는 다시 지나지 않음)

↳ 이 때마다 다른 계산

에지에 대한 계산 [에지에 대해 모두 / DFS 트리에 대해서만] 종류에 따라 다르게도 가능

(알고리즘 7.3) 유향 깊이 우선 탐색 골격 (DFS 골격) → 이 골격으로 다 처리 가능

Input : $G = (V, E)$ 인접 리스트 배열, $n = |V|$

```
dfsSweep( adjVertices, n, ... )      ← 이건 자유
    int ans
    배열 color 생성, white로 초기화
    G의 정점 v에 대해 특정 순서로,      ← 이건 구현에 따라 다름
     if ( color(v) = white )
        vAns = dfs ( G, color, v, ... )
        ↳ 이거 계속 처리
     return ans

dfs ( G, color, v )      ← 배열
    int w
    IntList remAdj
    int ans
    color(v) = gray
     v의 전순위 처리      ← 처음 발견 시 처리 (응용에 따라 다름)
     remAdj = G(v)
     while ( remAdj ≠ null )
         w = first ( remAdj )
          if ( color(w) = white )      ← 이 사이에 에지 vw 예비처리
            int  wAns = dfs( G, color, w, ... )
```

그 후 inorder 처럼 wAns를 이용해 uw의 되감 처리

else

트리에지가 아닌 uw를 체크 (처리)

remAdj = rest (remAdj)

ans의 최종 계산 포함하여 정점 v의 후순위 처리 → 어떤 의미에서 postorder 처리

Color(v) = black

return ans

↳ 응용에 따라 부분적인 탐색에서 문제가 풀릴수도 있음 ⇒ while 안에 break로 제어

· 깊이 우선 탐색의 구조 ※ DFS → Preorder, inorder, postorder를 동시에 수행 가능

어떤 정점 ← 루트 ↓ v ⇒ 회색 정점 프레임 스택에서 호출의 v-파라미터

경로 상에 있는 지 알 필요가 있음.

정점이 처음에 만나는 순서 + 그들간의 관계

↳ 관계의 관리 : discovertime, finishtime → 배열 두 개로 관리 (정점 색 바뀔 때 마다 증가)

<정의 7.15> 깊이 우선 탐색 용어

color(V) 흰색 ⇒ 미발견

color(v) = 회색 ⇒ 현재 time이 discovertime에 기록 (v의 전순위 처리) → v는 활동중 (active)

color(v) → 검정 ⇒ 현재 time이 finishtime에 기록 (v의 후순위 처리) → v 종료

v의 활동구간 (active interval) : active(v)

active(v) = discovertime(v), ..., finishtime(v)

↳ 이 사이 구간에서 v는 회색

time은 그래프 전체가 탐색되었다면 마지막에 2n → (발견, 되감)

깊이 우선 탐색 자취 (DFS 자취)

dfs sweep → time 0으로 초기화

dfs 호출 전에 parent(V) = -1 ← 새로운 cc의 root

dfs에서 dfs 호출 전에 parent(w) = v

그 다음 time 증가

〈정리 7.1〉

임의의 v, w에 대하여

① $active(w) \subseteq active(v)$일 때만 w는 v의 후손이다. $w \neq v$라면 포함에서 =이 빠진다

② v와 w가 DFS 포리스트에서 조상/후손 관계를 가지지 않으면 활동구간 겹치지 않음

③ $vw \in E$가 에지라면

a. $active(w)$가 $active(v)$를 완전히 앞설 때 vw는 교차 에지

b. 어떤 제 3의 정점 x가 존재하여 $active(w) \subset active(x) \subset active(v)$를 만족할때만 vw는 후손에지

c. $active(w) \subset active(v)$이고, $active(w) \subset active(x) \subset active(v)$인 제 3의 정점 x가 존재하지 않을 때만 vw는 트리에지

d. $active(v) \subset active(w)$일 때만 vw는 후진에지

v가 활동 중일 때 발견된 정점들은 모두 v의 후손

〈정리 7.3〉 흰색 경로 정리 (white path theorem)

그래프 G의 임의의 깊이 우선 탐색에서, 정점 w가 깊이 우선 탐색 트리에서 정점 v의 후손일 필요충분조건은 v가 발견될 때 (회색으로 칠하기 전에) v에서 w까지 흰색 정점으로만 이루어진 경로가 존재

- 사이클이 없는 유향 그래프 (DAG)
  - ↳ 유향 그래프의 특별한 경우 → 아주 중요 → 스케줄링 같은 문제가 자연스럽게 DAG로 표현 / 일반적인 유향 그래프에서보다 많은 문제들이 쉽고 효율적으로 풀림

모든 일반적인 유향 그래프는 압축 그래프 (condensation graph)라고 부르는 DAG와 연관

수학적으로 부분 순서 (partial order)   DAG (위상순서 / 임계경로

- 위상 순서 (topological order)

  모든 에지를 왼쪽에서 오른쪽으로 향하도록 재배열 가능?

  ↳ 사이클이 있으면 불가능 ⇒ DAG라면 가능 → 이런 배열을 찾는 문제를 위상 정렬 문제 (topological ordering)

〈정의 7.16〉 위상 순서

$G=(V,E)$, $n=|V|$인 유향 그래프 G의 위상 순서는 위상 번호라고 하는 서로 다른 정수 n을 V의 정점들에, 모든 에지 $vw \in E$에 대해 v의 위상 번호가 w의 위상 번호보다 작도록 할당하는 것. (역위상 순서는 v가 크게)

DAG 찾기 → 무턱대고 하기엔 쉽지 않음 ← 깊이 우선 탐색 골격 이용 필요

위상 순서 → 정점의 어떤 순열과 같다. 위상 정렬은 DAG에서의 기본 문제 → 모든 DAG는 하나 이상의 위상 순서를 갖는다

<예제: 종속 태스크 스케쥴링>

종속적인 태스크로 이루어진 프로젝트 수행 → 종속되어 있는 태스크가 끝나기 전에는 시작 불가

↳ 각 태스크 별로 종속된 태스크의 리스트를 가진 배열 이용 ⇒ 그래프 표현과 동일 (종속 그래프 (dependency) / 우선순위 그래프 (precedence))

보통 "시간이 흐르는 순서"의 반대인 전치 그래프 주로 이용

<알고리즘 7.5> 역위상순서 → DFS 골격 이용
역위상 순서가 더 자주 쓰임

Input: DFS 골격의 입력, 전역 배열 topo, 전역 카운터 topoNum

output: topo를 역위상 번호로 채운다

caution! 위상 순서를 계산하기 위해 toponum을 n으로 초기화하고 하나씩 뺀다

strategy:

dfs sweep 에서 toponum을 0으로 초기화

후순위 처리: toponum +1, topo(v) = toponum.

· 임계경로 분석 (critical path)

임계경로 분석 → 위상 순서와 연관 ⇒ DAG에서 최장 경로 찾는 문제 ⇒ 최적화 문제

<정의 7.17> 조기 시작 시간, 조기 종료 시간, 임계 경로

프로젝트: 1~n으로 번호가 붙은 태스크의 집합 → 각 태스크는 종속된 태스크의 리스트 가지고 있음

태스크 ← 수행시간 (duration)

태스크의 조기 시작 시간 (earliest start time; est)

1. 만약 v가 종속되어 있는 태스크가 없다면 0

2. v가 종속되어 있는 태스크가 있다면 est는 그것이 종속되어 있는 태스크의 조기 종료 시간의 최댓값

태스크의 조기 종료 시간 (earliest finish time; eft) : 조기 시작 시간과 수행 시간의 합

임계 경로 $(v_0, v_1, \cdots, v_k)$

1. v_0가 종속되어있는 task가 없다

2. 뒤따르는 태스크 v_i에 대해 $(1 \le i \le k)$ v_i는 v_{i-1}에 종속, v_{i-1}의 eft = v_i의 est

3. 프로젝트에 있는 태스크 중 v_k의 eft가 최대

임계 경로에는 "헐거운" 부분이 없다 ⇒ 태스크와 태스크 사이에 쉬는 시간이 없음.

↳ 하나를 연기하면 다음 것도 반드시 연기됨

〈알고리즘 7.6〉 임계경로

input: DFS 골격의 입력, 전역 배열 duration, critDep, eft (G는 DAG)

output: critDep와 eft 채우기

Caution! 에지가 시간이 흐르는 방향 ⇒ 조금만 수정하면 됨.

strategy: 1. 전순위 처리 : est = 0, critDep(v) = -1

2. 트리에지 / 트리가 아닌 에지에서 퇴각

if (eft(w) ≥ est)

est = eft(w)

critDep(v) = w

3. 후순위 처리 : eft(v) = est + duration(v)

- 사이클이 없는 유향 그래프 요약

주로 스케줄링 → DFS 골격에 조금 추가 DAG 매우 중요!!

7.5 유향 그래프의 강연결 성분

유향 그래프에서의 연결 ⇒ 에지의 방향대로 이어져야 함.

〈정의 7.18〉 강연결 성분

G의 강연결 성분 → G의 최대 연결 부그래프 (강성분)

〈정의 7.19〉 압축 그래프

$S_1, S_2, \cdots S_p$: G의 강성분. G의 압축 그래프는 $G\downarrow$로 표기하는 $G\downarrow = (V', E')$

V' = p개의 원소 $(s_1, S_2, \cdots, S_p)$, $i \neq j$ 이면서 S_i의 어떤 정점에서 S_j의 어떤 정점으로

가는 에지가 E에 있을 때만 $S_i S_j$가 E에 속한다. S_i의 모든 정점이 S_i로 압축

리더

유향 그래프 문제 ⇒ 강성분 + 압축 그래프의 문제 → 간결

강연결 사이클이 없음

- 강연결 성분의 성질

G^T의 강성분 ⇒ G의 강성분과 정점에 대해서는 같다

⇒ $(G^T)\downarrow = (G\downarrow)^T$ → 압축 그래프에서도 같다.

· 강성분 알고리즘 → 2단계 구성

1. G에서 표준 깊이 우선 탐색 시행 → 정점은 종료시점에 스택에 삽입

2. 전치 그래프 G^T에서 깊이 우선 탐색 시행 → 단계 1에서 만든 스택에서 삭제

↳ 각 정점 v가 속한 강성분의 리더를 scc(v)에 저장 이걸 이용해서 리더 찾는 게 가능 ⇒ 압축

7.6 무향 그래프의 깊이 우선 탐색

무향 그래프 → 대부분이 유향과 유사 ⇒ 자료구조에는 2번 포함되어 있지만 한 방향으로만 탐사 필요.
↳ 이게 복잡

깊이 우선 탐색 → 에지에 방향을 부여 (사이클이 있는 문제에서 중요)

↳ 에지가 두 번 처리되어도 상관없는 문제는 유향 대칭 그래프로 취급

분석.

1. 유향 대칭 그래프에서 교차 에지는 발생하지 않는다

p → v
v → p (후진) ⇒ 무시

2. DFS 트리에서 v로부터 부모 p까지의 후진 에지는 이 두 정점 사이의 무향 에지를 두 번째 만나는 것.

이외의 다른 후진에지는 처음 만나는 것.

3. 유향 대칭 그래프에서 전진 에지 ⇒ 항상 무향에지를 두 번째 만나는 것.

v → w 까지의 전진 에지. w는 전에 발견, wv는 후진에지로서 처리

교차 에지가 없으므로 검은 색 정점으로 가는 에지는 반드시 전진에지 ⇒ 무향 그래프에서는 무시.

<알고리즘 7.8> 무향 깊이 우선 탐색 골격

```
Input: G의 인접리스트, n=|V|
    dfs(G, color, v, p, ...)
        color(v) = gray
            // v의 전순위 처리
            remAdj = G(v)
        while (remAdj ≠ null)
            w = first(remAdj)
            if (color(w) = white)
                // vw의 예비 처리
                wAns = dfs(G, color, w, v)   ← 이걸 이용한 vw의 퇴각 처리
        else if (color(w) = gray and w≠p)
                // 후진 에지 vw 체크 (처리)
                remAdj = rest(remAdj)
        color(v) = black
```

· 무향 너비 우선 탐색

↳ 여기서도 무향 에지 재처리 문제 발생 → 무향 그래프를 유향 대칭 그래프로 취급.

7.7 무향그래프의 이중 연결 성분

연결된 그래프에서 정점 하나가 (인접한 에지와 같이) 제거되더라도 남은 부그래프는 연결일까?

↳ 통신, 교통 네트워크에서 중요 → 제거했을 때 분리되는 정점을 찾는 것도 중요

<정의 7.21> 이중 연결 성분

연결된 무향 그래프 G, 임의의 한 정점과 연결된 모든 에지 제거해도 연결 ⇒ 이중 연결 그래프

무향 그래프의 이중연결성분 (이중성분) ⇒ 최대 이중연결 부그래프

(articulation point / cut)

<정의 7.22> 절점

무향 그래프 G, 정점 v 가 $v \neq w, x$ 인 서로 다른 정점 w와 x를 잇는 모든 경로에 포함 ⇒ v는 절점

절점 제거 ⇒ 그래프 분리. 이중연결 그래프 → 절점을 가지고 있지 않다 (필요충분조건)

이중 연결 성분 → 에지에 대한 동치 관계 $e_1, e_2 \in E$일 때 $e_1 = e_2$ 이거나 e_1, e_2를 모두 포함하는 단순 사이클이 존재

⇒ 하나의 동치류에 속한 에지들과 인접한 정점으로 이루어진 부그래프는 이중연결성분

· 이중 성분 알고리듬

깊이 우선 탐색에서 정점 처리 ⇒ 발견 시에 (전순위 처리) / 탐색이 그 정점으로 퇴각할 때 (중순위 시간) / 종료 직전 (후순위 시간) 총 3번 가능

이중 성분 알고리듬은 퇴각 시에 그 정점이 절점인가를 검사

무향 그래프 DFS 트리 ⇒ 에지는 트리 에지 / 후진 에지

탐색이 w에서 v로 퇴각 ⇒ w가 루트인 부트리에서 v의 진조상으로 가는 후진에지가 없다면 v는 절점

↳ v는 DFS 트리의 루트에서 w로 가는 모든 경로에 포함

트리의 각 정점 (루트에서 멀어지는 방향으로) 트리 에지와 후진 에지를 따라 얼마나 위로 갈 수 있는 지 관리

↳ 지역 변수 back에 관리

<정리 7.13>

DFS 트리에서 루트가 아닌 정점 v가 절점인 필요충분조건은 v가 리프가 아니고 v의 어떤 부트리가 v의 진조상과 인접한 후진 에지를 갖지 않는 것.