Chapter 3. Decompositions of graphs

- graph

: vertex(node) + edge

: edge between x and y = {x, y} → undirected edge   = {y. x}

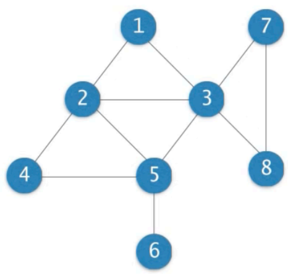
 * directed edge : from x to y = {x,y}, from y to x = {y, x}

3.1.1 How is a graph represented?

1) adjacency matrix(인접 행렬)로 표현

$n = |V|$ **vertices** $v_1, \ldots, v_n$, 이면 n x n 행렬로 표현할 수 있음

$$a_{ij} = \begin{cases} 1 & \textbf{if there is an edge from } v_i \textbf{ to } v_j \\ 0 & \textbf{otherwise.} \end{cases}$$



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1
8	0	0	1	0	0	0	1	0

대칭행렬

https://ict-nroo.tistory.com/78

→ undirected graph 의 경우 행렬은 대칭이다

→ 장점: 특정 edge 의 존재유무를 빠르게 확인할 수 있음

→ 단점: O(n^2)의 space 로, edge 의 수가 적으면 공간낭비가 크다

* 저장공간(공간복잡도): O(n^2)

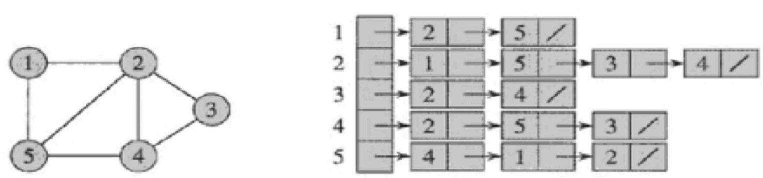
* node v 에 인접한 모든 node 찾기: O(n)

* edge(u,v)가 존재하는지 : O(1)

2) adjacency list 로 표현

 각 vertex 가 linked list 로 표현된

 Vertex u 의 linked list 는 u 에서 나가, 연결되는 vertex 와 연결되어 있다



https://ict-nroo.tistory.com/78

 → 저장공간(공간복잡도): O(E)

3.2 Depth-first search in undirected graph

3.2.1 Exploring mazes

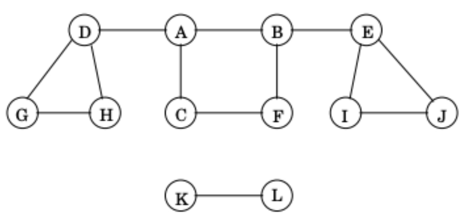
* DFS(깊이 우선 탐색)
 - exploration 동안 intermediate 정보를 저장
  1. chalk marks : 각 vertex 마다 방문 유무를 Boolean 으로 표기
  2. ball of string: stack 을 이용하여 이전 지점으로 돌아간다

Figure 3.3 Finding all nodes reachable from a particular node.

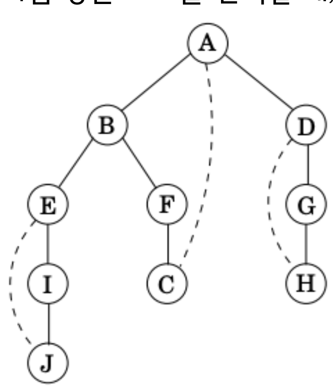
```
procedure explore(G,v)
Input:    G = (V,E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v,u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

→ previsit(v), postvisit(v)은 optional
  : vertex 를 처음 발견했을 때, 마지막으로 방문(떠날 때)를 표시하기 위한
→ explore(u)
  : 다음 노드로 이동하여 탐색하는 함수
  : 이웃 노드로만 가야한다!



→ 위 그래프에서 A 를 시작 노드로 하였을 때 explore 의 결과
  (다음 방문 노드를 선택할 때, 알파벳 순으로 연결을 끊음)



3.2.2 Depth-first search

위의 알고리즘은 시작 노드에서 그래프가 갈 수 있는 부분만 방문

→ 방문하지 않았던 부분을 다 방문하기 위해 알고리즘을 다시 작성해보면..

Figure 3.5 Depth-first search.

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v):  explore(v)
```

Figure 3.3 Finding all nodes reachable from a particular node.

```
procedure explore(G, v)
Input:    G = (V, E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

: 이렇게 하면 그래프의 모든 노드를 방문할 때까지 함수를 반복하게 된다.

* running time

1) fixed work: visited 표기, previsit(), postvisit()

→ O(|V|)

2) 인접한 edge 들을 보고, 그 edge 로 이동가능한지 확인

→ 각 edge 는 2 번씩 탐색된다.

이유 – e {x, y}가 있을 때, 이 edge 는 v 가 x 일 때, v 가 y 일 때 총 두번 탐색된다.

→ O(|E|)

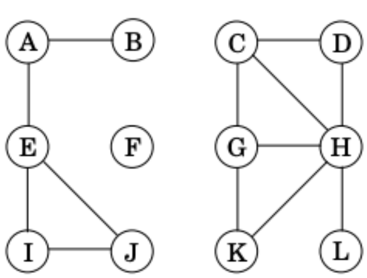
3) 최종적으로,,,

→ 입력 크기가 linear 할 때, DFS 최종 running time = O(|V|+|E|)

3.2.3 Connectivity in undirected graphs

- 아래 그래프는 not connected graph

(a)



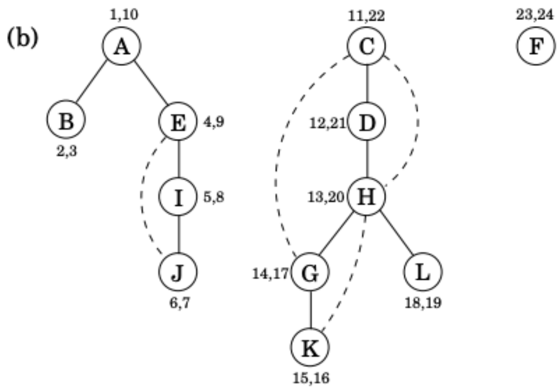
→ 시작 노드가 A, C, F 인 3 개의 트리로.. $\{A, B, E, I, J\}$ $\{C, D, G, H, K, L\}$ $\{F\}$

→ The outer loop of DFS calls explore three times

→ 분리된 3 개의 region 은 connected components

3.2.4 Previsit and postvisit orderings

이제 previsit 과 postvisit 에 대해서..



→ 총 24 번의 event

- counter 를 위한 변수 clock 을 1 로 초기화하여 pre 배열과 post 배열을 아래와 같이 update

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1
procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

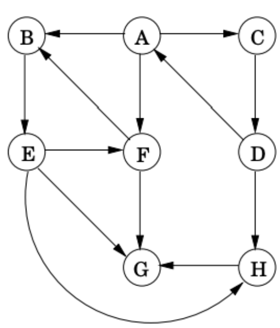
* Property

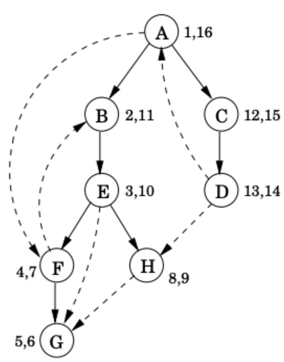
: node u, v 에 있어서, [pre(u), post(u)]와 [pre(v), post(v)] 두 간격은 하나가 다른 하나를 포함하거나 분리된다.
(분리되는 그래프형태면 후자에 속함)

: [pre(u), post(u)]은 노드 u 가 스택에 있는 시간과 같다

3.3 Depth-first search in directed graphs

3.3.1 Types of edges



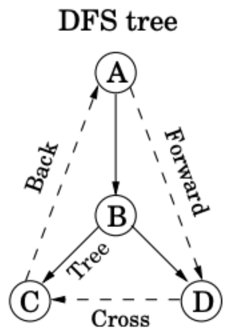


→ vertices 를 알파벳 순으로 생각했을 때의 search tree

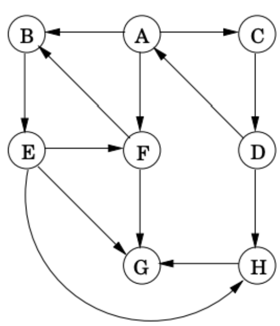
→ A: root node, 나머지 node: descendant node

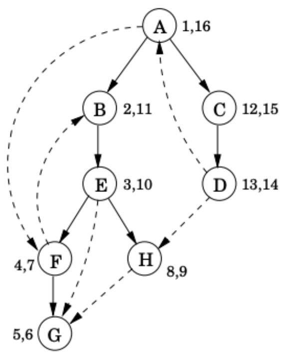
→ E 를 봤을 때, F,G,H – 자식 노드

- Undirected Graph 의 경우 tree edges 와 nontree edges 로 구분
- Directed Graph 의 경우 좀 더 elaborate taxonomy(정교한 분류법)로

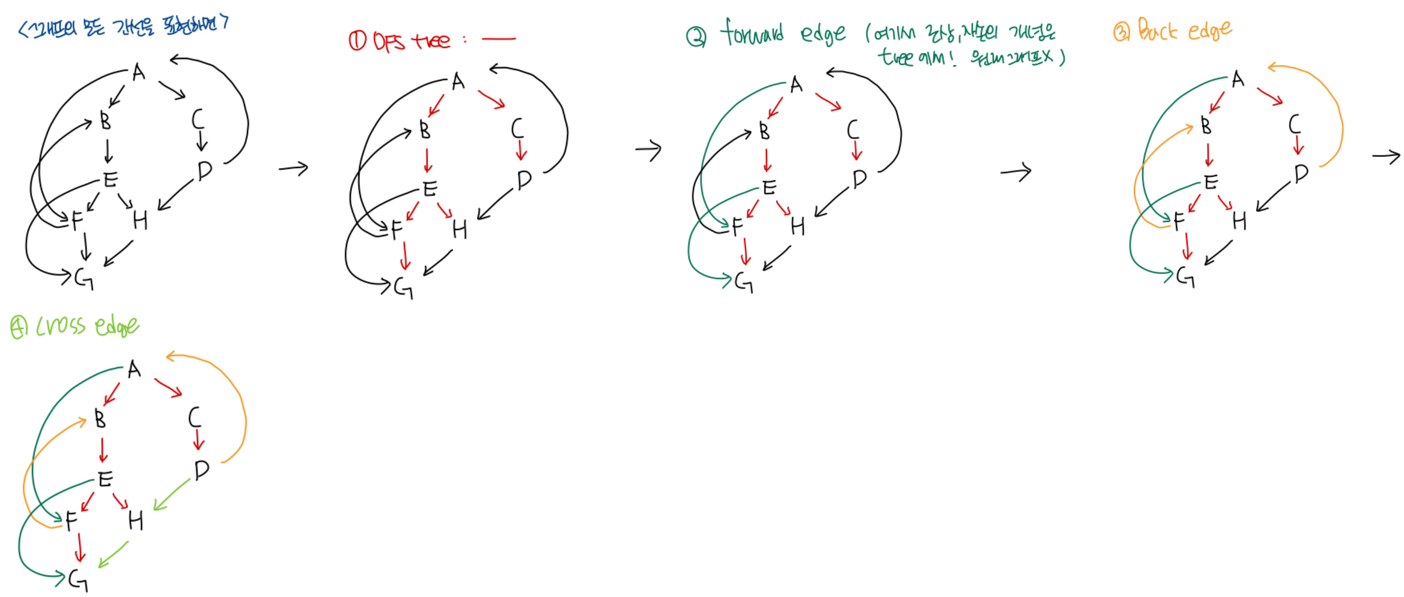


- Tree edges
  : DFS forest 의 한 부분 (DFS 의 결과로 완성된 트리를 이루고 있는 간선)
- Forward edges
  : 조상에서 자손으로 연결되지만 tree edge 는 아닌 edge
- Back edges
  : 자손에서 조상으로 연결되는 edge
- Cross edges: 이미 방문했던 노드로, tree edge, forward edge, back edge 를 제외한 edge





* 위 tree 에서는 2 개의 forward edge, 2 개의 back edge, 2 개의 cross edge



- pre, post 수로 자손/조상 관계를 알 수 있다
  : u 가 v 의 조상일 때, $\mathrm{pre}(u) < \mathrm{pre}(v) < \mathrm{post}(v) < \mathrm{post}(u)$

- pre, post 수로 edge type 을 알 수 있다

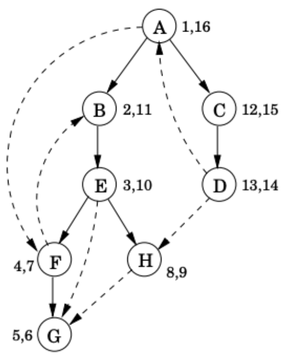
pre/post *ordering for* (u, v)	*Edge type*
$[_u \ [_v \ ]_v \ ]_u$	Tree/forward
$[_v \ [_u \ ]_u \ ]_v$	Back
$[_v \ ]_v \ [_u \ ]_u$	Cross

3.3.2 Directed acyclic graphs – 방향 비 순환 그래프

- directed graph 에서의 cycle = 아래와 같은 circular path 를 가지는

$$v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0$$

아래와 같은 그래프에서 B → E → F → B



* Property - A directed graph has a cycle if and only if its depth-first search reveals a back edge.
(DFS tree 에서 back edge 가 있는 경우에만 cycle 이 존재한다)

- Directed acyclic graphs(dags)은 항상 존재한다
: 선행작업을 directed graph 로 표현할 수 있다. 하지만 이때 cycle 이 생기면 안된다
(Cycle 이 있으면 순서를 생성할 수 없기 때문에!!)
Q. 어떤 타입의 dags 를 linearized 할 수 있는가?
A. 모두!

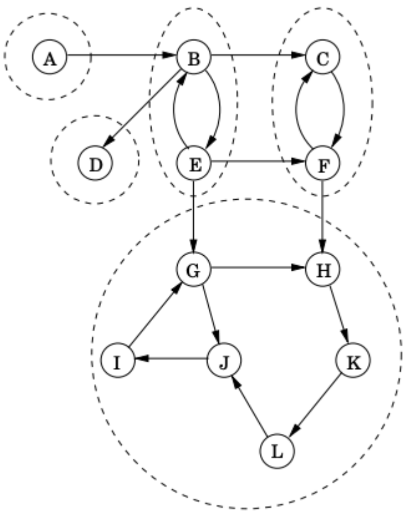
* Property - In a dag, every edge leads to a vertex with a lower post number
(dag 에서 모든 edge 는 가장 낮은 post 숫자를 가진 node 에 다다른다)
→ dag 의 노드들을 순서화하는 알고리즘은 linear-time 이 걸린다
→ 결국, acyclicity == linearizability == the absence of back edge
(비순환 == 선형화 == back edge 가 없다)
→ post 숫자가 점점 작아지도록 linearize(선형화)하기 때문에, 가장 작은 post 숫자를 가지는 노드는 linearization 의 맨 마지막에 위치하게 된다.
→ 이렇게 마지막에 위치하는 노드는 outgoing edge 가 없는 sink(끝점)이다

* Property - Every dag has at least one source and at least one sink
(모든 dag 은 적어도 하나의 source 와 하나의 sink 를 가진다)
1) source 를 찾고, 출력하고 source 를 그래프에서 삭제
2) 그래프가 빌 때까지 위 과정 반복

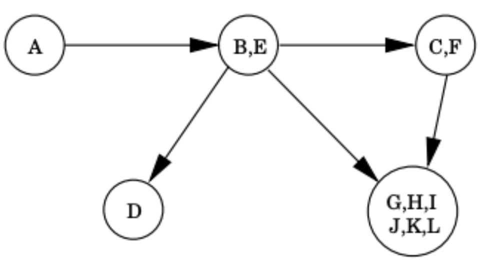
3.4 Strongly connected components

3.4.1 Defining connectivity for directed graphs

아래의 Directed graph 는 connected graph 로 보이고, edge 를 자를 수 없을 것 같아 보인다



→ 하지만, connected graph 가 아니다! – G to B, F to A 의 path 가 존재하지 않음
→ * Strongly Connected component
: 모든 노드들이 모든 다른 노드들에 대해 도달이 가능한 경우 (path 가 존재하는 경우)
→ 위와 같은 directed graph 를 하나 이상의 SCC 로 분리→점섬
→ 이제 각각의 SCC 를 single meta-node 로 (노드 하나가 하나의 SCC)



→ meta-graph

→ 그 결과는 dag 이다

* Property = Every directed graph is a dag of its strongly connected components.
(도느 directed graph 는 scc 들의 dag 이다)

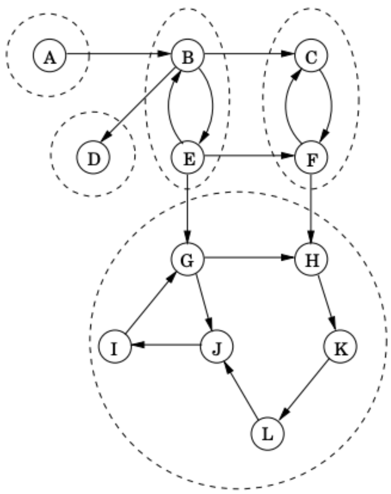
위 property 를 통해 Directed graph 는 두 계층으로 이루어져 있다는 것을 알 수 있음
1. 선형화 할 수 있는 dag
2. dag 노드안의 ssc

3.4.2 An efficient algorithm

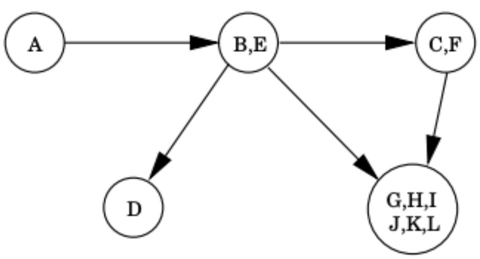
* **Property 1** - If the explore subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.

: explore subroutine 이 u 에서 시작되었다면, u 로부터 갈 수 있는 모든 노드가 방문되어야 subroutine 이 종료된다

: sink scc 인 노드에서 explore 를 시작하면 그 노드 값을 추출



→ 여기서는 두개의 sink SCC 를 가진다 왜?



→ 여기서 보면 sink 가 두개

: 이는 SCC 를 찾는 방법을 suggest 하지만, 여전히 2 개의 문제가 존재함

Problem 1. sink 인 SCC 를 어떻게 찾을 것인가?

Problem 2. first component 를 찾으면 어떻게 진행해 나갈건지?

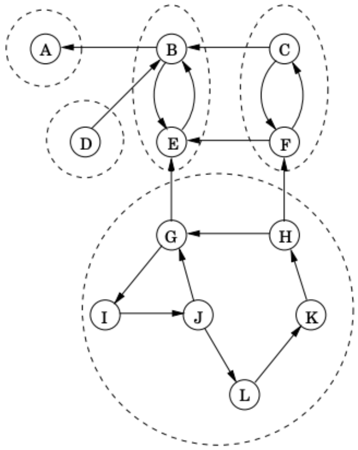
Problem1 에서 바로 sink 인 scc 를 찾기는 힘들지만, source 인 scc 는 찾을 수 있다

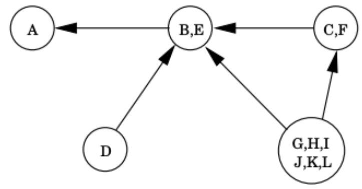
* **Property 2** - The node that receives the highest post number in a depth-first search must lie in a source strongly connected component.

(전체 DFS 에서 가장 큰 post number 를 갖는 노드는 반드시 source 인 scc 안에 있다)

: 우리가 이제까지 한 거는 source 에서 시작해서 sink 로 가는…, 근데 여기서는 sink 에서 시작해서 source 를 찾는

: 그래서 아래와 같이 그래프를 reverse 한다





* **Property 3** - If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', then the highest post number in C is bigger than the highest post number in C'
(C 와 C'가 SCC 이고 C 에서 C'로 가는 edge 가 존재하면 C 에서 가장 큰 post number 는 C'에서의 가장 큰 post number 보다 더 크다.)
위 Property 가 말하고 있는 것 = scc 의 가장 높은 post number 를 감소시키는 순서로 배열하면 scc 를 linearization 할 수 있다
→ 이를 통해 source scc 를 찾을 수 있다
→ reverse graph 에서 DFS 를 하고 가장 높은 post number 을 가진 노드는 source scc 에 있다는 걸 알 수 있었다
== reverse graph 에서 가장 높은 post number 을 가진 노드는 원래 graph 에서의 sink scc 에 속한다
→ Problem1 해결

Problem2 해결은
1. G'에 대해 깊이 우선 탐색을 실행한다.
2. G 에 대해 undirected 연결 성분 알고리즘을 실행하고, DFS 동안에 앞선 1 단계부터 post 숫자가 감소하는 순서로 node 를 처리한다.

참고: https://tttto-factory.tistory.com/3?category=740226