Im4u 봄방학 캠프
DAY 5; Elementary Graph Theory

구종만
jongman@gmail.com

그래프 (Graphs)

- 가장 중요한 (?) 자료형인 그래프의 소개
- 그래프 문제의 큰 두 가지 벽
  - 모델링: 현실 세계의 개념을 그래프로 나타낸다
  - 알고리즘: 그래프로 나타낸 문제를 해결한다

- 대개의 문제에서는 한 가지 벽만 있다
  - 모델링: 문제를 풀면서 공부
  - 알고리즘: 비교적 배우기 쉽다 (^^)

자주 등장하는 주제들

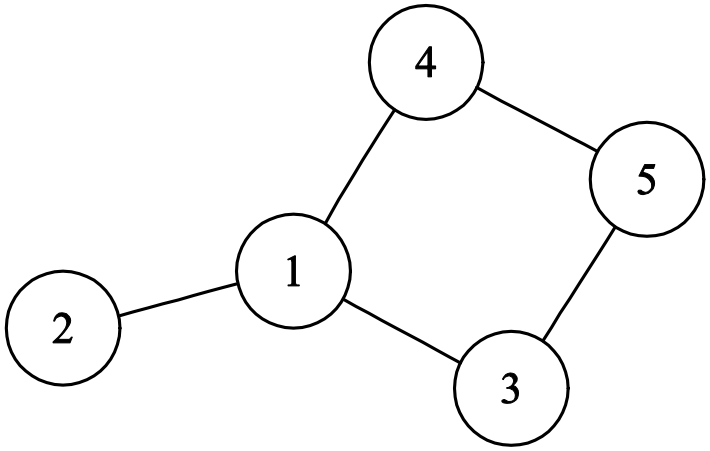
- 셀 수 없이 많지만..
- 그래프의 탐색
  - 그래프의 구조 파악
  - 컴포넌트들로 분해
  - 한붓그리기 (Eulerian Paths)
- 그래프를 대상으로 하는 최적화 문제들
  - 최소 신장 트리 (MST)
  - 최단 거리 문제 (Shortest Paths)
  - 네트워크 유량 (Network Flow)

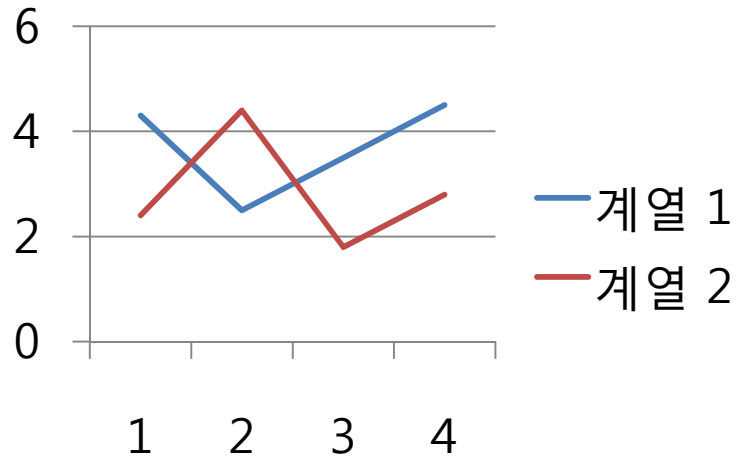
오늘 할 이야기

- (오늘은 알고리즘에 집중해 보죠)
- 그래프의 소개, 명시적/암시적 그래프
- 그래프의 구현
- 깊이 우선 탐색 및 너비 우선 탐색
- Dijkstra 의 최단거리 알고리즘
- 한붓그리기
- 현실 세계의 문제를 그래프로 바꾸기
- 네트워크 유량 (안할지도..)

다음주

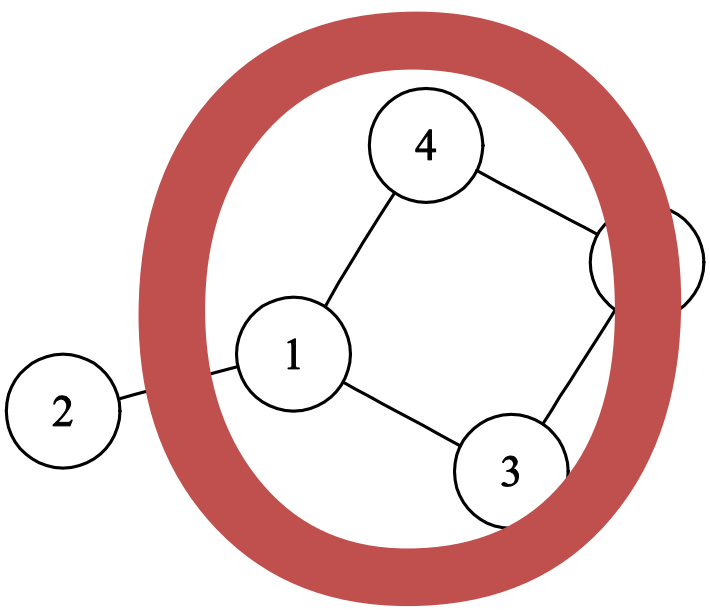
그래프

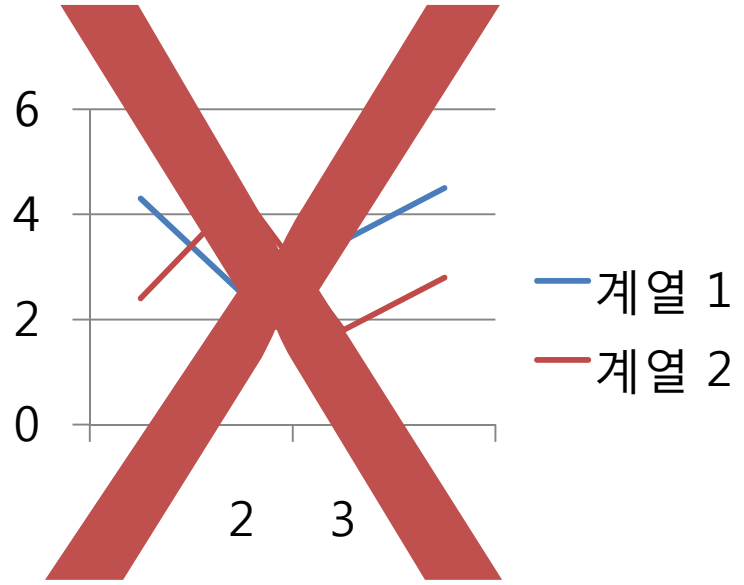




- 우리가 여기서 다루고 싶은 것은?

그래프





- 네 함수의 그래프 (graph of a function) 아니죠 그래프 맞습니다~

느슨한 정의

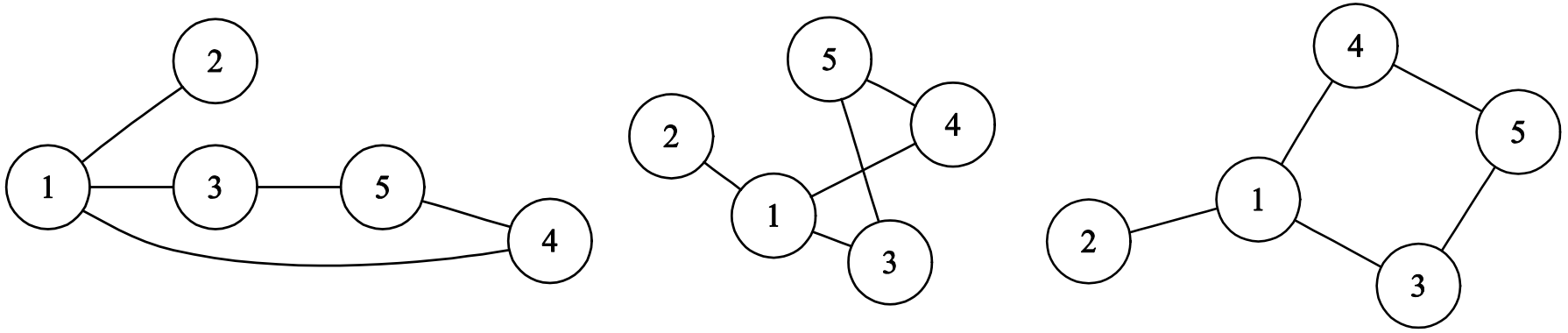
- 객체들과 그들의 연결 관계를 표현하는 자료구조
  - 기차역들과 그들을 연결하는 기차 선로
  - 라우터들과 그들을 연결하는 네트워크
  - 논문들과 그들의 상호 참조 관계
  - 사람들과 그들 사이의 친분 관계
  - 소년탐정 김전일: 누가 누구를 죽였는가?
- 현실 세계의 문제를 해결하기 위한 수학적인 모델
  - 모델: 현실 세계의 근사치, 우리 머릿속에서 둥둥 떠다닙니다

엄격한 정의

- 정점 (vertex) 들의 집합과 그들의 연결 관계인 간선 (edge) 들의 집합으로 구성되는 자료 구조

$$G(V, E)$$

- 다른 정보는 포함하지 않는다
  - 아래의 세 그래프는 모두 같은 그래프

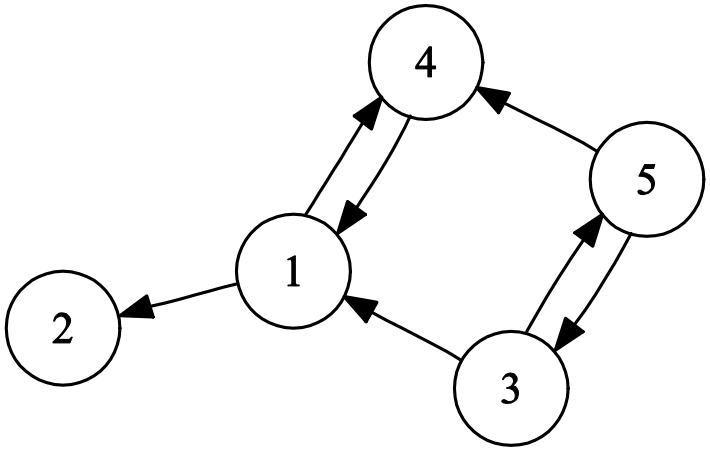


그래프는 어디서 튀어나오는가

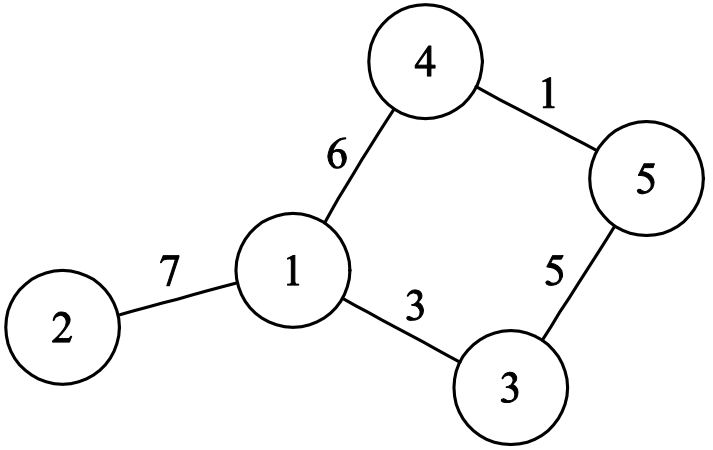
- 명시적 그래프 (Explicit Graph)
  - 문제에 대 놓고 주어지는 그래프
  - 라우터와 그들을 잇는 네트워크 선
  - 여러 남녀들과 그들의 짝사랑 관계
- 암시적 그래프 (Implicit Graph)
  - 15-Puzzle
  - 브라운 운동 트래킹하기
  - 이것들은 다음 주에 … (사실 이게 진짜 재밌는 부분)

그래프의 종류들

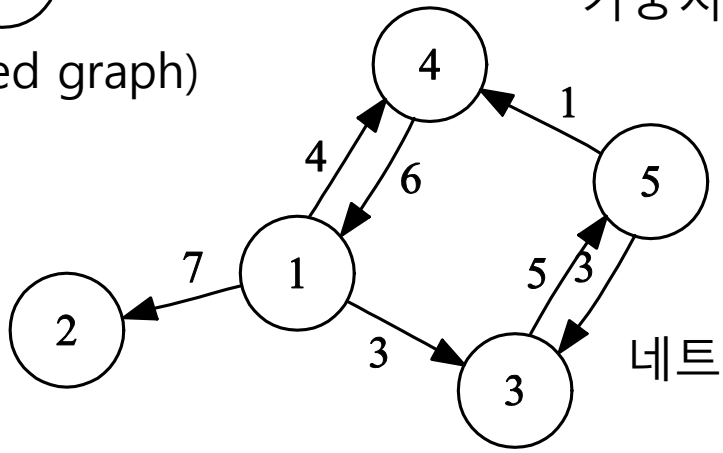
- 표현하려는 문제에 따라 그래프의 간선들은 다른 속성을 가질 수 있다



방향 그래프 (directed graph)



가중치 그래프 (weighted graph)



네트워크 (network)

그래프 구현하기

- The OOP Way

```
class Vertex {
   public:
      void connectTo(const Vertex& other, int weight);
      bool isConnectedTo(const Vertex& other);
      VertexIterator getAdjacent() const;
};

class Edge {
   public:
      Vertex* getFirst();
      Vertex* getSecond();
      int getWeight();
};
```

사실 대개는

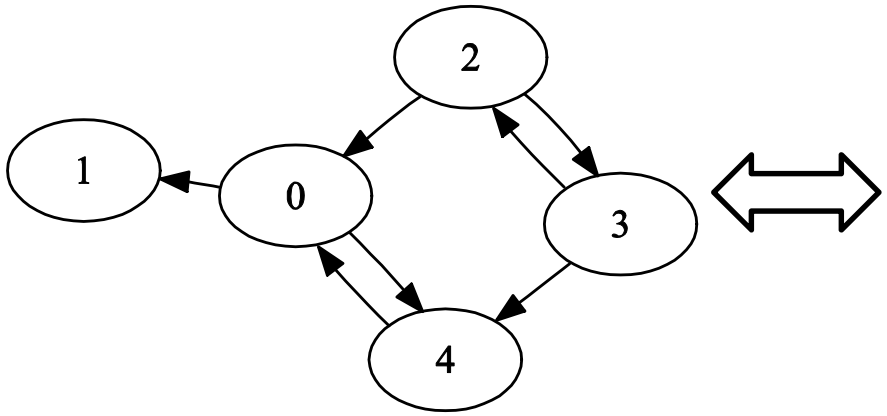
- 각 정점에 0 부터 V-1 까지의 번호를 부여
  - 배열을 통해 간단하게 구현하게 된다
- 정점이 문자열이라도 다음과 같이 변환 가능

```
int getVertexNo(const string& name) {

   static map<string,int> m;

   if(m.count(name) > 0) return m[name];
   int ret = m.size();
   return m[name] = ret;
}
```

가장 간단한 방법

- 인접 행렬 (adjacency matrix)
  - VxV 크기의 배열에 간선 정보를 담는다

```
int V;
bool graph[MAX_V][MAX_V]; // 간선 (u,v) 가 있으면 graph[u][v] == true
bool directed[MAX_V][MAX_V]; // graph[u][v]!= graph[v][u] 일 수도 있음
int weighted[MAX_V][MAX_V]; // graph[u][v] 는 (u,v) 의 가중치, 없으면 -1
```

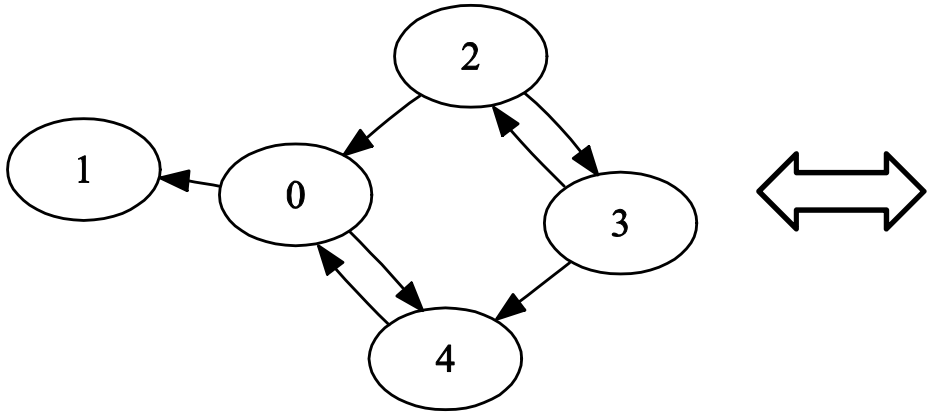


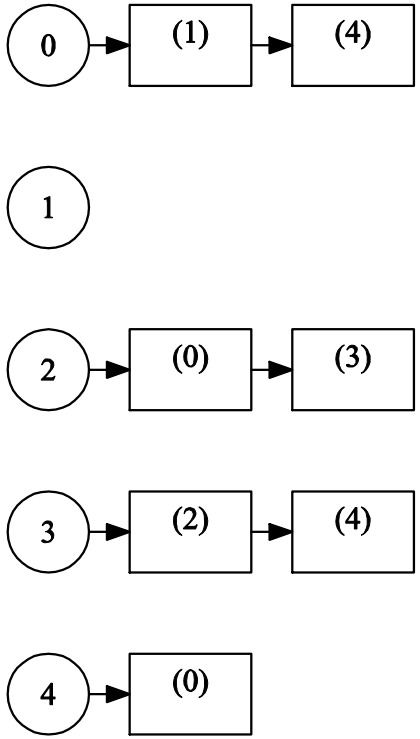
	0	1	2	3	4
0	0	1	0	0	1
1	0	0	0	0	0
2	1	0	0	1	0
3	0	0	1	0	1
4	1	0	0	0	0

가장 간단한 방법 2

- 인접 리스트 (adjacency list)
  - V개의 연결 리스트를 만들어 간선 정보를 담는다

```
int V;
std::list<int> graph[MAX_V];
std::list<pair<int,int> > weighed[MAX_V];
```

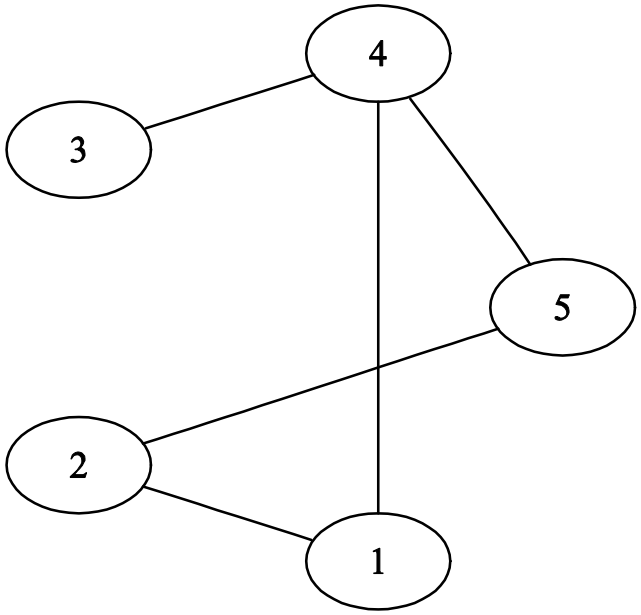


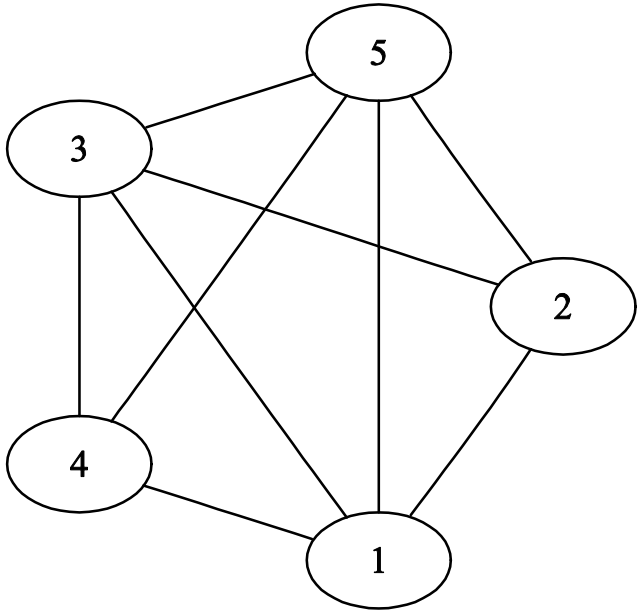


Matrix vs. List

- 인접 행렬
  - 두 정점이 주어졌을 때 간선의 정보를 O(1) 에 얻을 수 있다
  - 인접한 모든 정점을 효율적으로 찾을 수 없다
  - 메모리 사용량이 많다 O(V*V)
  - 밀집 그래프 (dense graph) 에 적합
- 인접 리스트
  - 두 정점이 주어졌을 때 연결 리스트를 처음부터 읽어가야 한다
  - 인접한 모든 정점을 효율적으로 찾을 수 있다
  - 메모리 사용량이 적다 O(E)
  - 희소 그래프 (sparse graph) 에 적합

희소 그래프 vs. 밀집 그래프



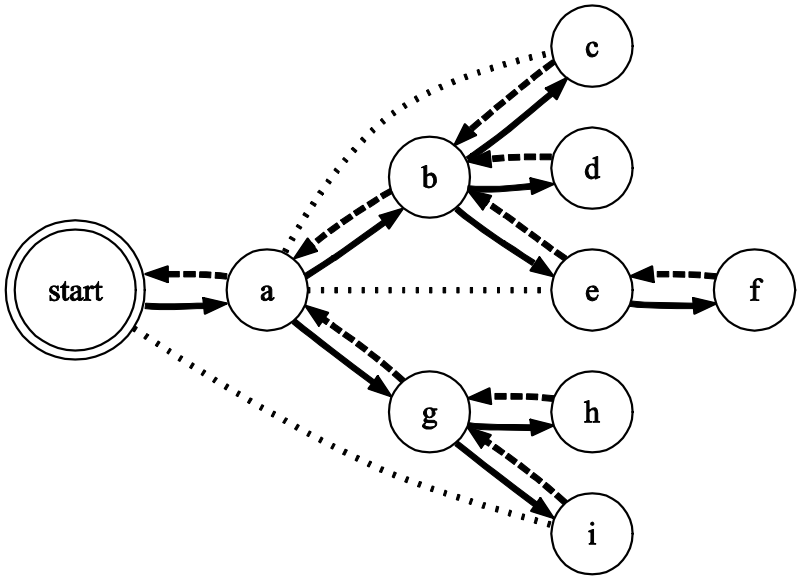


- 백문이 불여일견

그래프의 탐색 (search)

- YAM (Yet Another Misnamed)
  - 역시 또 다른 잘못 지어진 이름 중 하나 (라고 주장中)
- 탐색 보다 관광 에 가까움
- 특정한 순서에 따라 순서대로 '발견' 하는 전략
- 가장 유명한 두 개의 방법이 있음
  - 깊이 우선 탐색
  - 너비 우선 탐색
- 이들은 많은 그래프 알고리즘의 골격을 형성함

깊이 우선 탐색



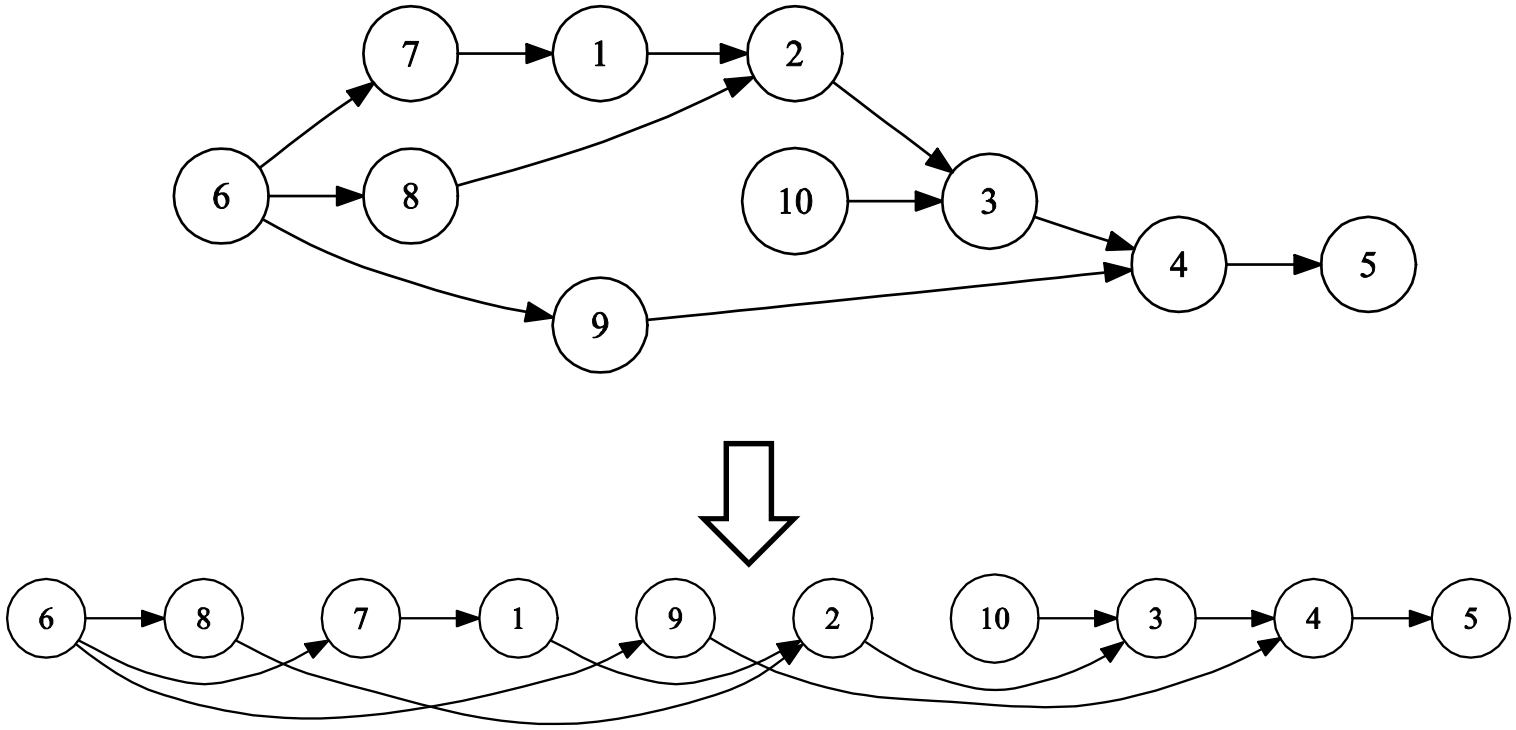
- 인접한 정점 중 아직 방문하지 않은 것을 하나 골라서 그 간선을 따라 간다
- 더 따라갈 간선이 없으면 마지막에 따라온 간선으로 돌아간다
- 계속 반복
- 간선의 종류에 주목

깊이 우선 탐색의 구현

```
int V;
std::vector<int> graph[MAX_V];
bool visited[MAX_V];

void dfs(int here) {
   visited[here] = true;
   for(int i = 0; i< graph[here].size(); ++i) {
      int there = graph[here][i];
      if(!visited[there])
         dfs(there);
   }
}

for(int i = 0; i < V; ++i) visited[i] = false;
for(int i = 0; i < V; ++i) if(!visited[i]) dfs(i);
```

- 대개 재귀호출로 구현
- 모든 정점에서 시작하는 것에 유의: 왜?

위상 정렬



- DAG (Directed Acyclic Graph) 를 재배열
- 모든 간선이 왼쪽에서 오른쪽으로 가도록

위상 정렬 by DFS

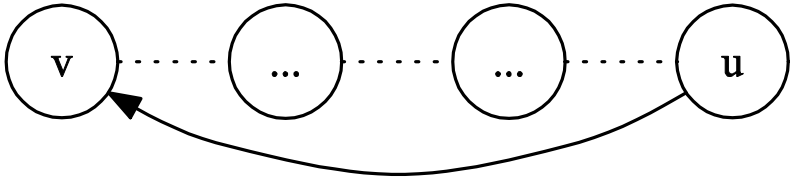
- 앞 절의 코드에서 dfs() 함수가 종료할 때마다 해당 정점을 목록의 맨 *앞*에 추가한다
- 이것만으로도 위상 정렬 완료!

- 증명?

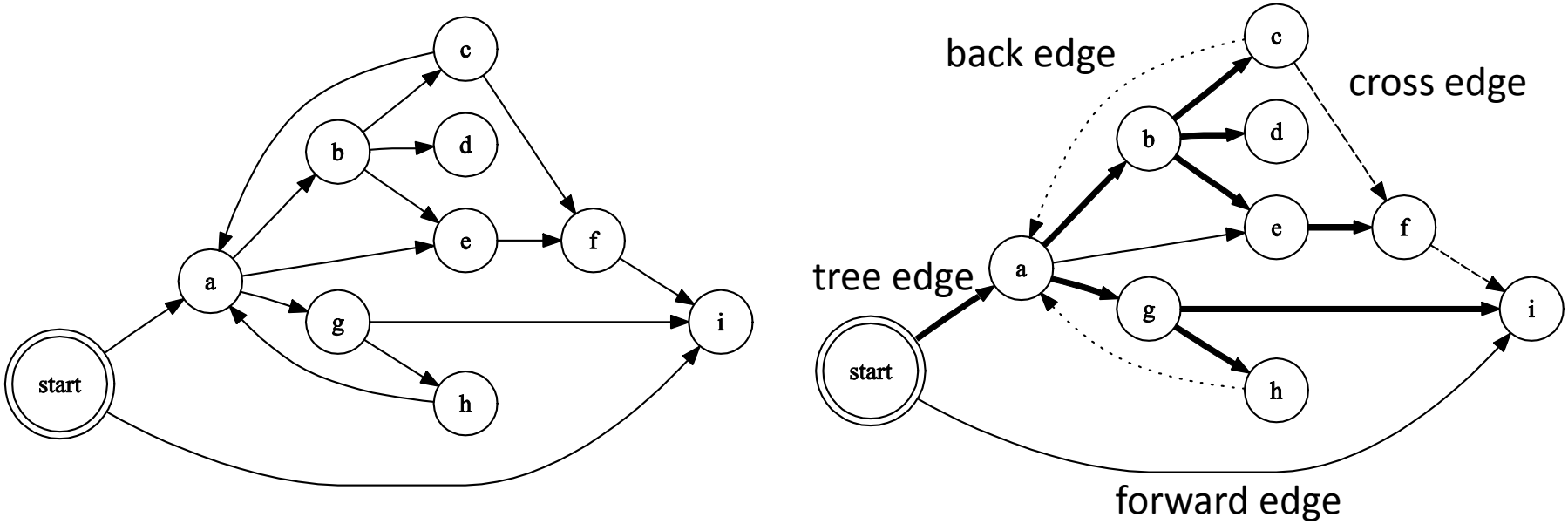
위상 정렬 by DFS

- 앞 절의 코드에서 dfs() 함수가 종료할 때마다 해당 정점을 목록의 맨 *앞*에 추가한다
- 이것만으로도 위상 정렬 완료!

- 증명 by 귀류법: 반대로 가는 간선 (u,v) 가 있다고 하자. 어느 쪽이 먼저 dfs() 함수가 종료했을까?



깊이 우선 탐색 신장 트리



- 깊이 우선 탐색에 사용된 간선 (tree edge) 만 남긴 트리

간선의 구분

- DFS as a coloring process

```
enum COLOR { WHITE, GRAY, BLACK };

int V;
std::vector<int> graph[MAX_V];
COLOR color[MAX_V];
int rank[here], time = 0;
void dfs(int here) {
   color[here] = GRAY;
   rank[here] = time++;
   for(int i = 0; i < graph[here].size(); ++i) {
      int there = graph[here][i];
      if(color[there] == WHITE) dfs(there);
      if(color[there] == GRAY) _______;
      if(color[there] == BLACK) _______;
   }
   color[here] = BLACK;
}

for(int i = 0; i < V; ++i) color[i] = WHITE;
for(int i = 0; i < V; ++i) if(color[i] == WHITE) dfs(i);
```

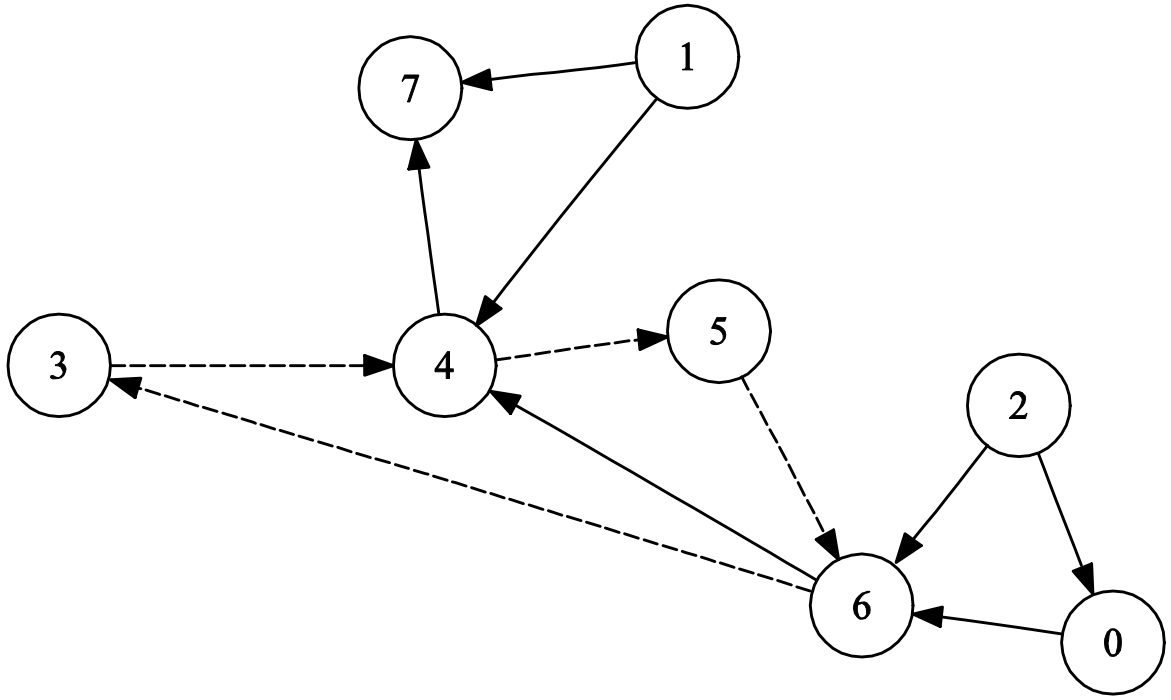
간선의 구분

- Tree Edge (u,v)
  - _____
- Forward Edge (u,v)
  - _____
- Back Edge (u,v)
  - _____
- Cross Edge (u,v)
  - _____

간선의 구분

- Tree Edge (u,v)
  - Color[there] == WHITE
- Forward Edge (u,v)
  - color[there] == BLACK, rank[there] > rank[here]
- Back Edge (u,v)
  - color[there] == GRAY
- Cross Edge (u,v)
  - Color[there] == BLACK, rank[there] < rank[here]

사이클 찾기



- 주어진 그래프에 사이클이 존재하는가?

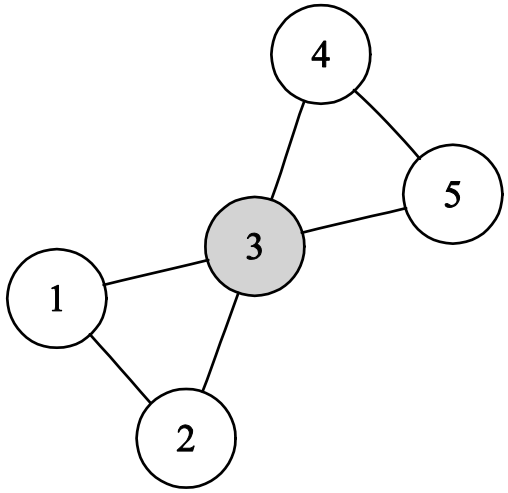
사이클 찾기

- 그래프에서 사이클의 존재 여부는 Back Edge 의 존재 여부와 동치

- 사이클 $(u_1, u_2, \cdots, u_{n-1}, u_n, u_1)$ 에서, WLOG, u_1 가 제일 먼저 발견되었다고 하자
  - dfs(u_1) 은 u_n 이 발견되기 전까지 끝나지 않는다
  - 간선 (u_n, u_1) 을 발견했을 때, u_1 의 색깔은?

절단점 찾기

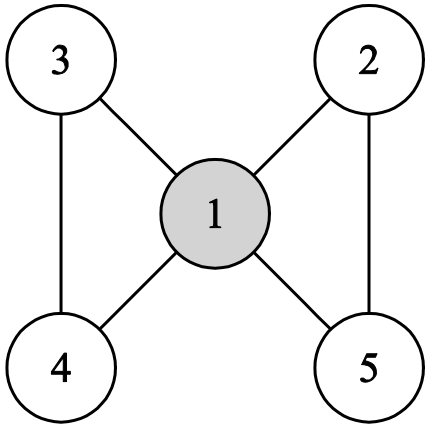
- 해당 정점과 인접 간선을 없애면 그래프가 두 개 이상으로 찢어지는 정점들 (cut vertex)
- 이 문제에서는 연결된 무방향 그래프를 가정



절단점 찾기 by DFS

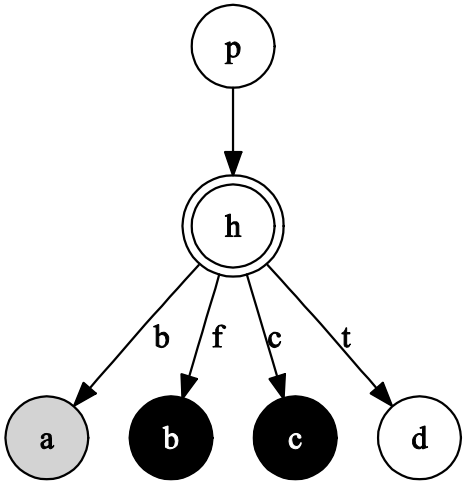
- DFS 를 하면서, dfs(u) 가 종료할 때 u 가 절단점인지 아닌지의 여부를 판정할 수 있다
  - 연결된 정점들의 rank, color, …

현재 정점이 절단점일까?



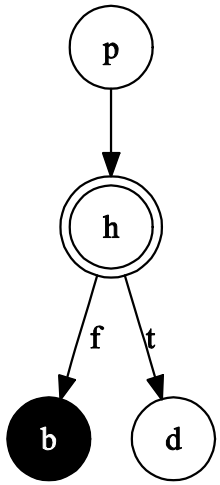
- 자신이 DFS 신장 트리의 루트인가/아닌가로 나뉨
- 루트인 경우: 비교적 판단하기 쉽다
  - 한 개의 간선만 따라가서 탐색해도 모든 정점을 만날 수 있어야 함

루트가 아닌 경우



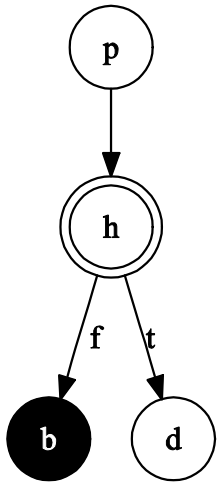
- Forward edge 를 만났다면?
  - 글쎄다
- Cross edge 를 만났다면?
  - _____
- Tree edge 를 만났다면?
  - 글쎄다
- Back edge 를 만났다면?
  - _____

루트가 아닌 경우



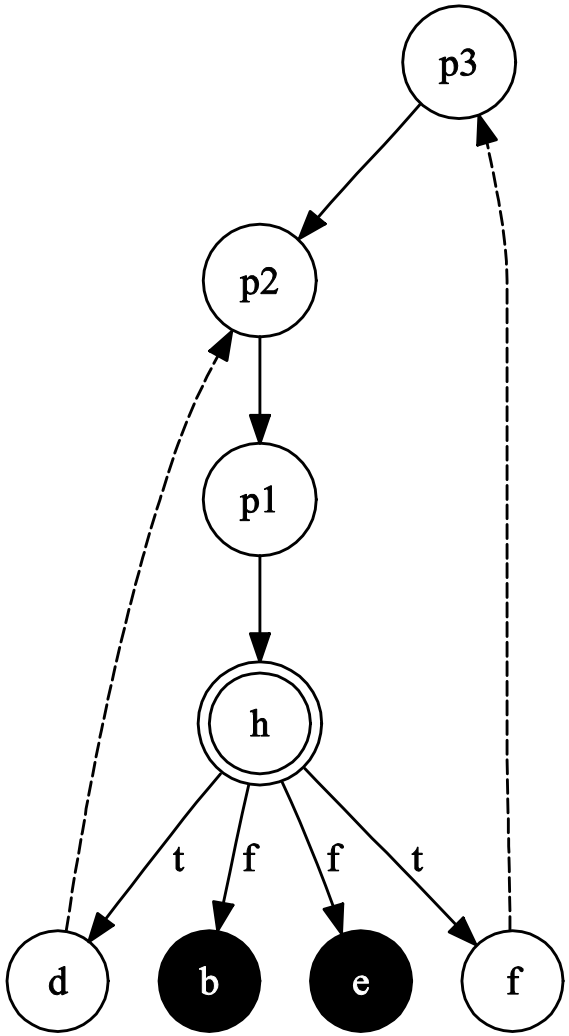
- Forward edge 를 만났다면?
  - 글쎄다
- Cross edge 를 만났다면?
  - Cross edge 는 없다
- Tree edge 를 만났다면?
  - 글쎄다
- Back edge 를 만났다면?
  - h 는 사이클 위에 있다 (절단점 X)

Forward 와 Tree



- 이 정점이 절단점이 아니라면
  - p 와 d 는 (h 빼고) 어떻게든 연결되어 있어야 한다
- 이것을 어떻게 찾을까?
  - dfs(u) = DFS 신장트리에서 u 와 그 후손들에 대해, 직접 연결된 정점 중 최소의 rank 를 반환

내가 절단점이 아니라면…



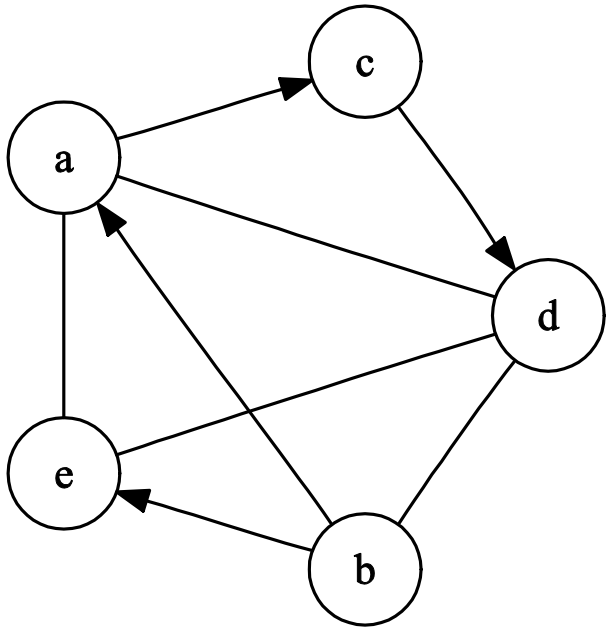
- 모든 내 후손들은 내 선조 (rank 가 나보다 작은) 들로 가는 간선을 갖고 있어야 한다
  - 만약 (f,p3) 이 없다면 h 는 절단점일 수밖에 없다

따라서..

```
int time = 0;
std::vector<int> graph[MAX_V];
int rank[MAX_V]; // rank[i] == -1 이면 아직 방문 안 했음
int low[MAX_V]; // low[u] = u 를 루트로 하는 서브트리에서 연결된 가장 작은 rank
bool isCutVertex[MAX_V];

int dfs(int here) {
   rank[here] = low[here] = time++;
   isCutVertex[here] = false;
   for(int i = 0; i < graph[here].size(); ++i) {
      int there = graph[here][i];
      if(rank[there] == -1) // there 는 here 의 후손
         low[here] = min(low[here], dfs(there));
      else // forward edge 혹은 back edge
         low[here] = min(low[here], rank[there]);
      if(low[there] >= rank[here])
         isCutVertex[here] = true;
   }
}
dfs(0);
```

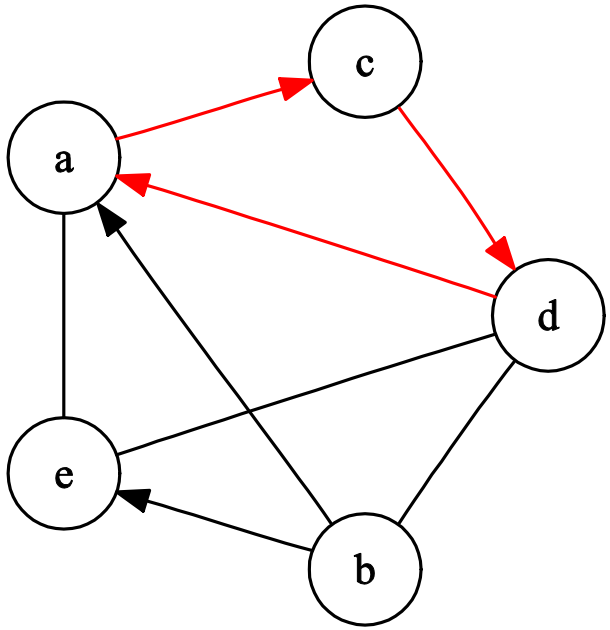
- 유일한 예외: isCutVertex[0]

OneWayStreets



- 양방향 간선을 한 방향으로 고정해서 사이클이 없도록 하고 싶다: 가능할까?
- 찍어봅시다!

OneWayStreets

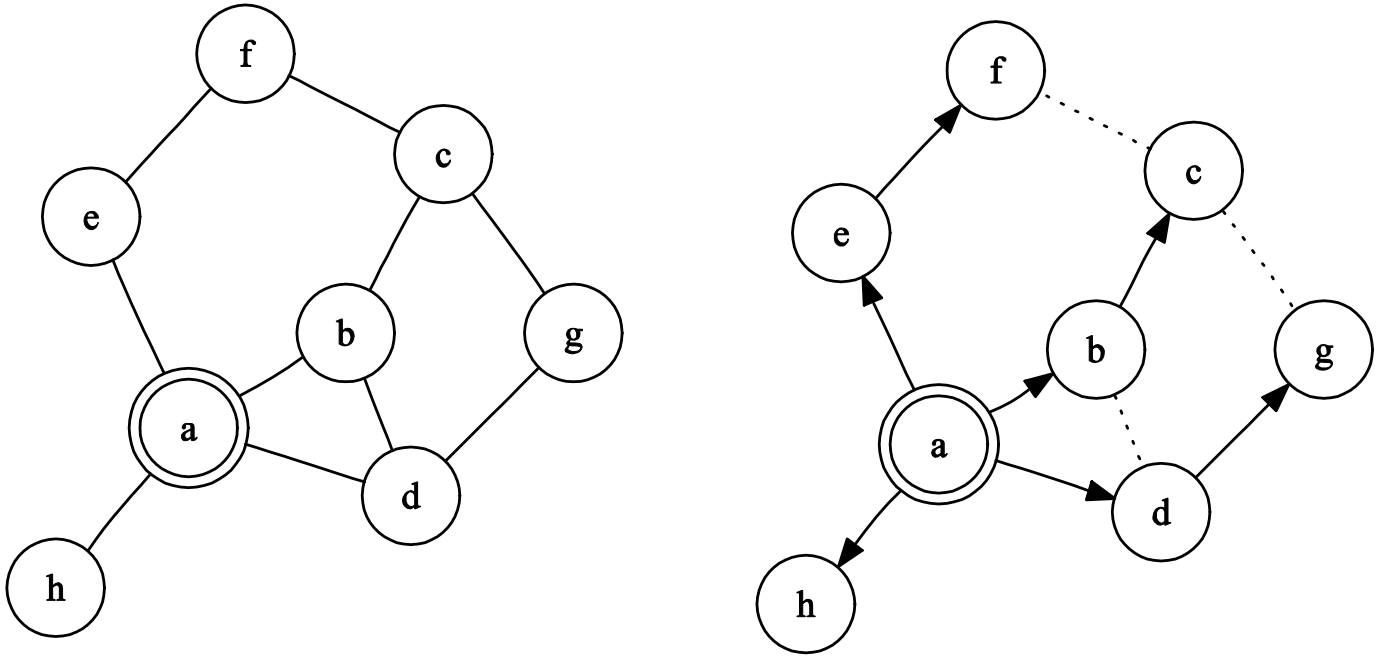


- 주어진 그래프에 이미 방향 있는 간선으로 구성된 사이클이 있다면, 당연히 불가능하다
- 만약 그런 사이클이 없다면 가능할까?

OneWayStreets

- 가능하다!
- 구성적 증명
  - 방향 간선만 남긴 그래프 G’ 를 얻는다
  - G’ 를 위상 정렬한다
  - 모든 무방향 간선을 왼쪽에서 오른쪽으로 가도록 방향을 고정한다

너비 우선 탐색



- 시작점에서의 최단거리 순서대로 방문해 나간다

Formally Explained

- 방문할 정점들의 목록 q 를 유지한다
  - q 는 반드시 FIFO (First-In-First-Out) 큐 여야 한다
- q 에서 정점을 꺼내 방문하고
  - 인접 정점 중에 아직 발견하지 못한 정점을 찾아 q 에 추가
- 더 q 에 정점이 남지 않을 때까지 반복한다

BFS 의 구현

```
int time = 0, V;
std::vector<int> graph[MAX_V];
bool seen[MAX_V];
void bfs(int start) {
   for(int i = 0; i < V; ++i) seen[i] = false;
   std::queue<int> q;
   q.push(start); seen[start] = true;
   while(!q.empty()) {
      int here = q.front(); q.pop();
      for(int i = 0; i < graph[here].size(); ++i) {
         int there = graph[here][i];
         if(!seen[there]) {
            q.push(there);
            seen[there] = true;
         }
      }
   }
}
```

- std::queue 를 사용하는 것이 간편하다
  - 이 세미나에서 배우는 모든 알고리즘 중 가장 유용--;;

어떻게 가까운 순서대로 방문할 수 있지?

- 모든 정점들을 최단거리별로 분류한다고 하자

$$V = S_0 \cup S_1 \cup \cdots \cup S_x$$

- 맨 처음 방문하는 정점은 $\{start\} = S_0$
- S_i 를 모두 방문하고 나면 큐에는 S_{i+1} 만 들었다.
- 왜냐면 $u \in S_i$ 를 방문 중에 v 를 큐에 추가했다면, $v \in S_{i+1}$ 일 테니까요
  - 이것은 귀류법으로 증명 가능

최단거리를 기록하는 너비 우선 탐색

```
int time = 0, V;
std::vector<int> graph[MAX_V];
int dist[MAX_V];
void bfs(int start) {
   for(int i = 0; i < V; ++i) dist[i] = -1;
   std::queue<int> q;
   q.push(start); dist[start] = 0;
   while(!q.empty()) {
      int here = q.front(); q.pop();
      for(int i = 0; i < graph[here].size(); ++i) {
         int there = graph[here][i];
         if(dist[there] == -1) {
            q.push(there);
            dist[there] = dist[here] + 1;
         }
      }
   }
}
```

- dist[x] = start ~ x 의 최단거리

SortingGame

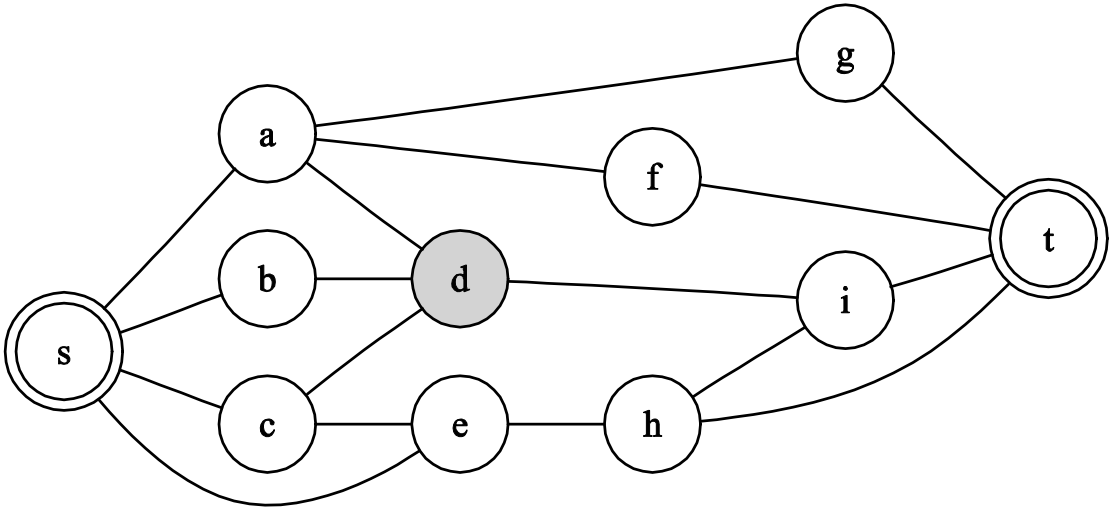
- 길이 n (<= 8) 인 정수 수열에 대해, 임의의 부분 구간을 골라 이것을 뒤집을 수 있다.
- 이 때, 주어진 수열을 정렬하기 위해 이 뒤집기를 최소 몇 번이나 해야 하는가?

SRM397 Div1 250

SortingGame

- 8! = 40320 개의 가능한 상태가 있다
- 게임판의 상태를 정점으로, 뒤집는 연산을 간선으로 하면 무방향 그래프를 얻을 수 있다
- BFS 로 풀면 되죠?

Avoiding Your Boss



- Boss 는 s 에서 t 까지 항상 최단거리로 움직인다
  - 2개 이상 있으면, 모든 경로에 대해 동일한 확률
- Boss 가 d 를 지날 확률은?

3-Tier Problem

- s 에서 t 까지 최단거리를 잰다
- s 에서 t 로 가는 최단 경로의 개수를 구한다
- d 를 지나는 최단 경로의 개수를 구한다

- Graph & DP combined!
- 이번 주의 숙제입니다 ☺