2022-1학기 스터디 #7

나정휘

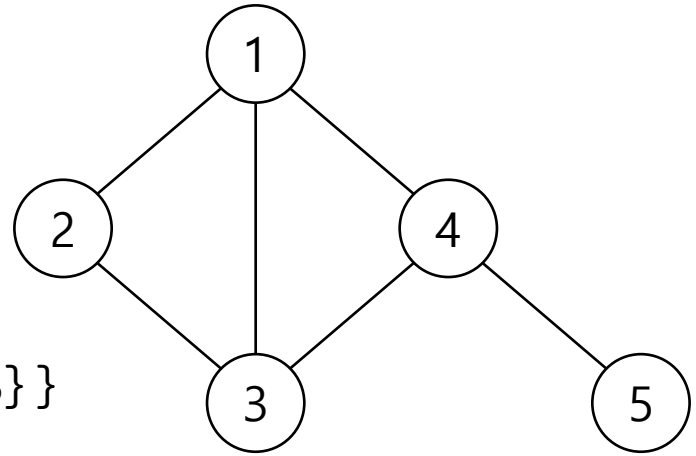
https://justiceHui.github.io/

목차

그래프

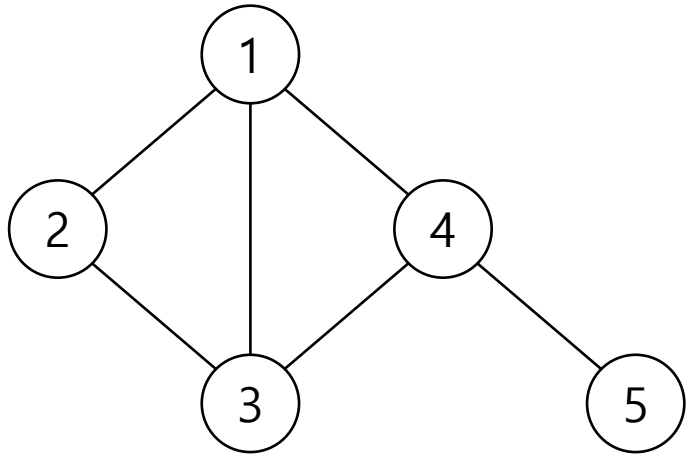
그래프

- 그래프
  - 정점 집합과 두 정점을 연결하는 간선 집합으로 구성된 구조
    - G = (V, E)
    - V: 정점들의 집합
    - $E \subseteq \{ \{x,y\} \mid x,y \in V \}$: 간선들의 집합
    - ex. V = {1, 2, 3, 4, 5}, E = { {1,2}, {2,3}, {3,4}, {4,1}, {4,5}, {1,3} }



  - 간선에 방향이 없으면 무향 그래프 (Undirected Graph)
  - 간선에 방향이 있으면 유향 그래프 (Directed Graph)
  - 간선에 가중치가 있으면 가중치 그래프 (Weighted Graph)
  - 모든 정점이 간선을 통해 직/간접적으로 연결되어 있으면 연결 그래프 (Connected Graph)

그래프



- 그래프 관련 용어
  - 차수(degree): 한 정점에 연결되어 있는 간선의 수
    - 입력 차수(in-degree): 한 정점으로 들어오는 간선의 수
    - 출력 차수(out-degree): 한 정점에서 나가는 간선의 수
  - 보행(walk): 정점과 변이 교대로 등장하는 나열, 각 간선의 앞/뒤는 간선이 연결하는 두 정점
    - ex. 1 – {1,2} – 2 – {2,3} – 3 – {3,1} – 1 – {1,2} - 2
  - 트레일(trail): 간선이 중복되지 않는 보행
    - ex. 1 – {1,2} – 2 – {2,3} – 3 – {3,1} – 1 – {1,4} - 4
  - 경로(path): 정점이 중복되지 않는 트레일
    - ex. 1 – {1,2} – 2 – {2,3} - 3
  - 사이클(cycle): 시작과 끝점이 동일하고, 나머지는 정점의 중복이 없는 트레일
    - ex. 1 – {1,2} – 2 – {2,3} – 3 – {3,1} – 1
  - 연결 요소(Connected Component): 경로를 통해 서로 연결된 정점들의 집합

그래프

- 특별한 그래프
  - 사이클이 없는 방향 그래프(Directed Acyclic Graph)
  - 완전 그래프(Complete Graph): 서로 다른 두 정점이 한 개의 간선으로 모두 연결된 그래프
  - 이분 그래프(Bipartite Graph): 정점을 서로 연결되지 않은 집합 2개로 나눌 수 있는 그래프
  - 완전 이분 그래프(Complete Bipartite Graph)
  - 숲(Forest): 사이클이 없는 무향 그래프
  - 트리(Tree): 연결 요소가 1개인 숲
  - 선인장(Cactus): 모든 간선이 최대 1개의 사이클에만 속한 그래프

그래프

- 특별한 그래프
  - 2 Connected Graph: 정점을 임의로 1개 제거해도 연결되어 있는 그래프
    - k Connected Graph
  - 2-edge Connected Graph: 간선을 임의로 1개 제거해도 연결되어 있는 그래프
    - k-edge Connected Graph
  - 현 그래프(Chordal Graph): 길이 4 이상의 모든 사이클이 chord를 갖는 그래프
    - chord: 사이클을 구성하는 간선은 아니지만, 사이클에 포함된 두 정점을 연결하는 간선
  - 평면 그래프(Planar Graph): 2차원 평면에 간선이 교차하지 않도록 그릴 수 있는 그래프
    - Outer Planar Graph: 모든 정점이 가장 바깥 평면과 인접하도록 그릴 수 있는 평면 그래프

질문?

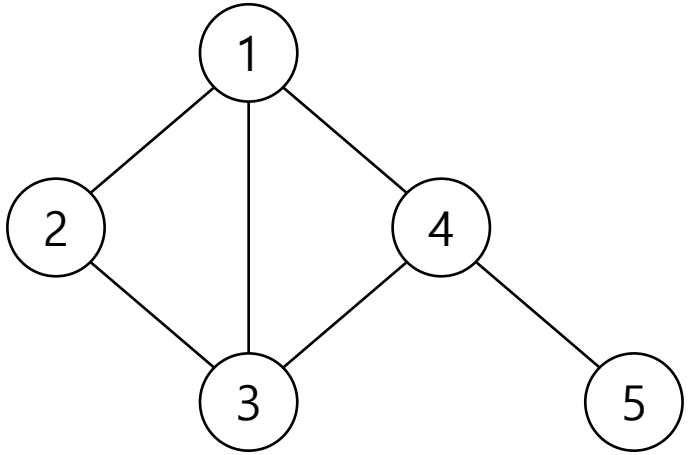
그래프의 표현 방식

그래프의 표현 방식

- 그래프의 표현 방식
  - 그래프를 이용한 작업을 수행하기 편하도록 그래프를 효율적으로 저장하는 방식
  - 인접 행렬
  - 인접 리스트
  - 간선 리스트

그래프의 표현 방식

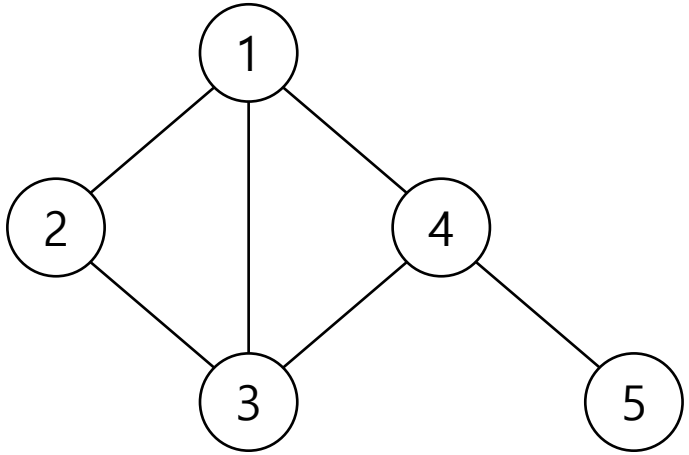
- 인접 행렬
  - |V| * |V| 크기의 2차원 배열 G
  - u에서 v로 가는 간선이 있으면 G[u][v] = 1
    - 무향 그래프면 G[u][v] = G[v][u]
    - 가중치 그래프면 G[u][v] = {가중치}
  - 공간 복잡도: $O(|V|^2)$

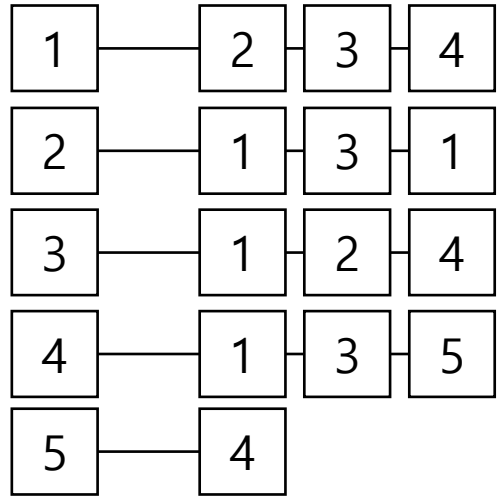


0	1	1	1	0
1	0	1	0	0
1	1	0	1	0
1	0	1	0	1
0	0	0	1	0

그래프의 표현 방식

- 인접 리스트
  - 연결 리스트 |V|개 관리
  - u번째 리스트는 u에서 나가는 간선을 관리함
  - 공간 복잡도: O(|V| + |E|)
    - O(|V| + sum deg(v))인데 sum deg(v) = 2E임
    - Handshaking Lemma





그래프의 표현 방식

- 인접 행렬 vs 인접 리스트
  - 공간 복잡도
    - 인접 행렬: $O(|V|^2)$
    - 인접 리스트: $O(|V| + |E|)$
  - 임의의 두 정점 u, v를 연결하는 간선이 있는지 확인
    - 인접 행렬: $O(1)$
    - 인접 리스트: $O(\deg(u))$
  - 모든 간선을 순회
    - 인접 행렬: $O(|V|^2)$
    - 인접 리스트: $O(|V| + |E|)$
  - 구현
    - 인접 행렬: 2차원 배열
    - 인접 리스트: 연결 리스트 또는 동적 배열 사용 (STL 사용)

그래프의 표현 방식

- 간선 리스트
  - 인접 리스트와 비슷하지만 인접 리스트의 단점을 상쇄
    - 연결 리스트 또는 동적 배열(std::vector) 기반이라 느림
    - 간선을 모두 순회하기 불편함
    - STL 없으면 구현하기 귀찮음
  - 뒤에서 DFS/BFS 할 때 코드와 함께 알아보자!

질문?

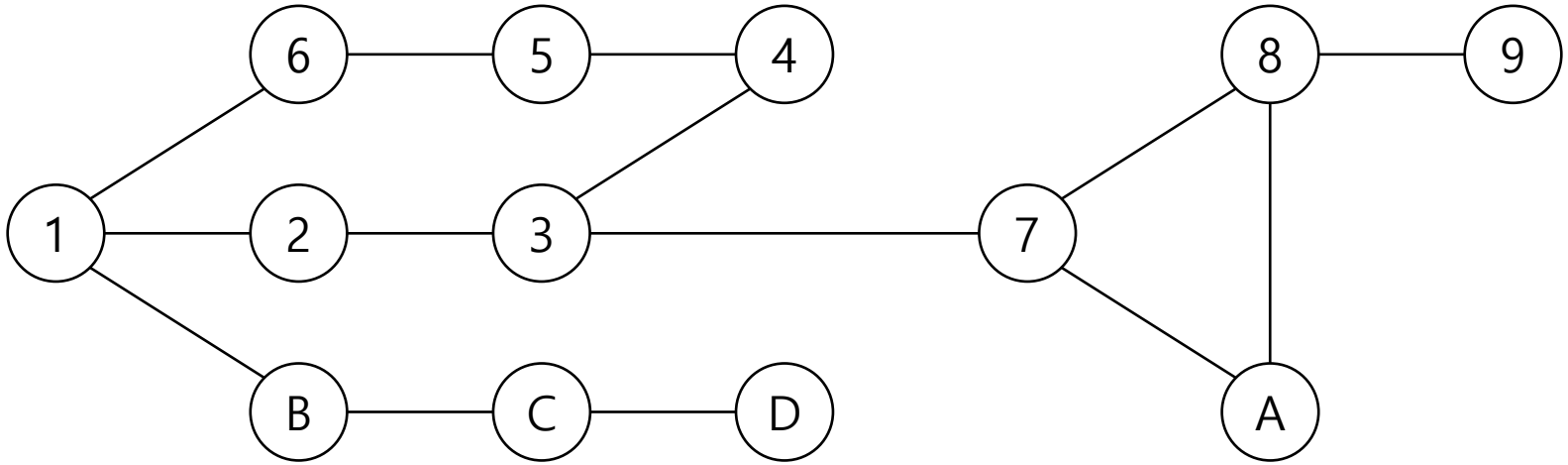
그래프의 탐색

그래프의 탐색

- 그래프의 탐색
  - 그래프의 정점을 적당한 순서로 한 번씩 모두 보는 방법이라고 생각하면 편함
  - 깊이 우선 탐색
  - 너비 우선 탐색

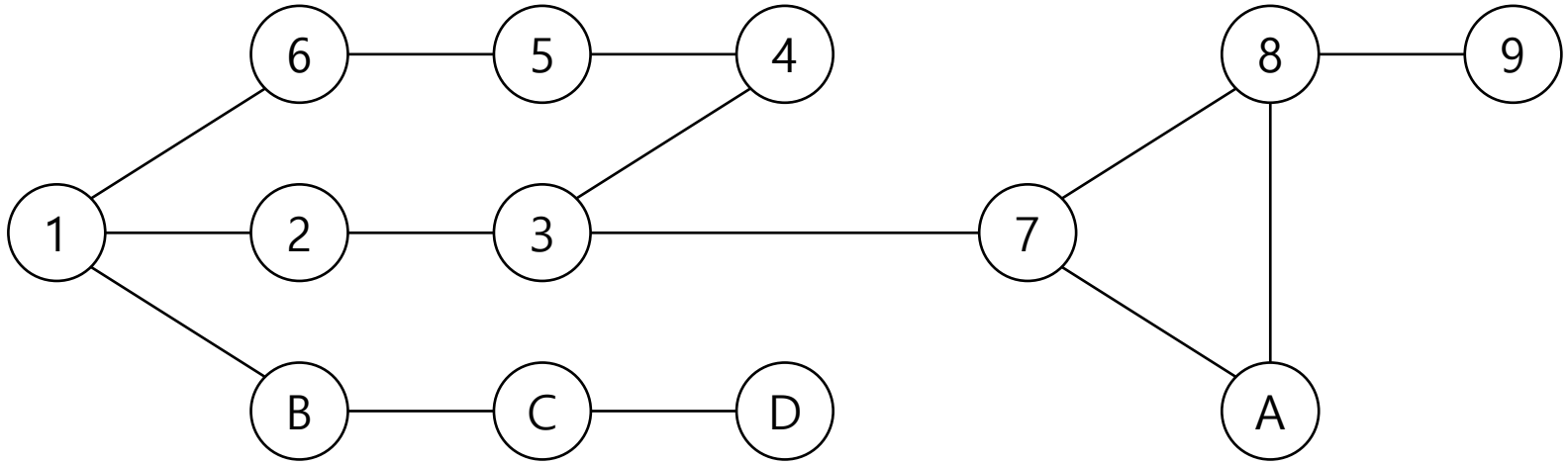
그래프의 탐색

- 깊이 우선 탐색 (Depth-First Search, DFS)
  - 한 정점에서 시작
  - 현재 정점과 인접한 정점 중 아직 방문하지 않은 정점을 하나 선택해 방문
  - 만약 인접한 정점을 모두 방문한 상태라면 이전 정점으로 돌아감



그래프의 탐색

- 깊이 우선 탐색 (Depth-First Search, DFS)
  - 만약 인접한 정점을 모두 방문한 상태라면 이전 정점으로 돌아감
    - 스택이나 재귀를 이용해서 구현할 수 있음
    - PS할 때는 보통 재귀를 사용해서 구현함
    - 스택을 사용하는 구현은 함수 호출 스택을 직접 구현하면 됨




```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M, S, G[1010][1010], C[1010];

void DFS(int v){
    cout << v << " ";
    C[v] = 1;
    for(int i=1; i<=N; i++){
        if(G[v][i] && !C[i]) DFS(i);
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> S;
    for(int i=0; i<M; i++){
        int u, v; cin >> u >> v;
        G[u][v] = G[v][u] = 1;
    }
    DFS(S);
}
```

그래프의 탐색

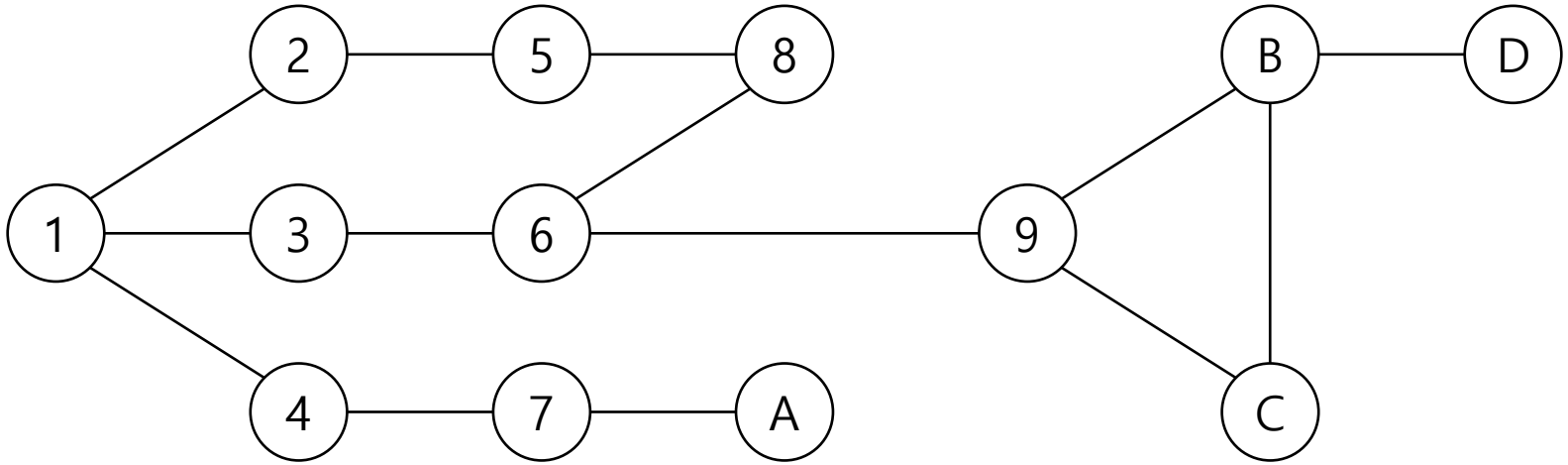
- 깊이 우선 탐색 (Depth-First Search, DFS)
  - 인접 리스트 / 간선 리스트를 사용한 구현

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M, S, C[1010];
vector<int> G[1010];

void DFS(int v){
    cout << v << " ";
    C[v] = 1;
    for(auto i : G[v]) if(!C[i]) DFS(i);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> S;
    for(int i=0; i<M; i++){
        int u, v; cin >> u >> v;
        G[u].push_back(v);
        G[v].push_back(u);
    }
    DFS(S);
}
```

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M, S, C[1010];
int head[1010], nxt[20202], to[20202], cnt;

void AddEdge(int s, int e){
    int id = ++cnt;
    to[id] = e;
    nxt[id] = head[s];
    head[s] = id;
}

void DFS(int v){
    cout << v << " ";
    C[v] = 1;
    for(int i=head[v]; i!=-1; i=nxt[i]) if(!C[to[i]]) DFS(to[i]);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> S;
    for(int i=1; i<=N; i++) head[i] = -1;
    for(int i=1; i<=M*2; i++) nxt[i] = -1;
    for(int i=0; i<M; i++){
        int u, v; cin >> u >> v;
        AddEdge(u, v);
        AddEdge(v, u);
    }
    DFS(S);
}
```

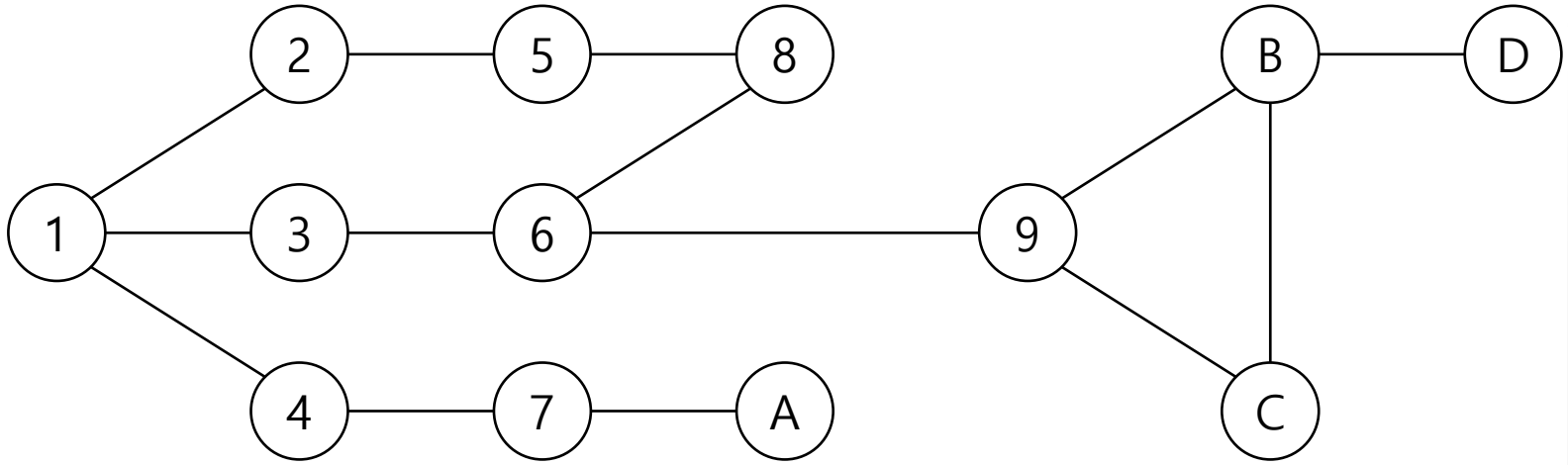
그래프의 탐색

- 너비 우선 탐색 (Breadth-First Search, BFS)
  - 한 정점에서 시작
  - 현재 정점과 인접한 정점 중 아직 방문하지 않은 정점을 모두 방문



그래프의 탐색

- 너비 우선 탐색 (Breadth-First Search, BFS)
  - 현재 정점과 인접한 정점 중 아직 방문하지 않은 정점을 모두 방문
    - 정점들의 방문 “대기열”을 만들어야 하므로 큐를 사용해서 구현
    - 시작 정점과 거리가 가까운 정점부터 방문하게 됨
    - 큐에 넣는 시점에 방문 체크를 해야 함
      - 큐에서 빼는 시점에 체크하면 안 됨




```
void BFS(int v){
    queue<int> Q;
    Q.push(v); C[v] = 1;
    while(!Q.empty()){
        v = Q.front(); Q.pop();
        cout << v << " ";
        for(int i=1; i<=N; i++){
            if(G[v][i] && !C[i]) Q.push(i), C[i] = 1;
        }
    }
}
```

그래프의 탐색

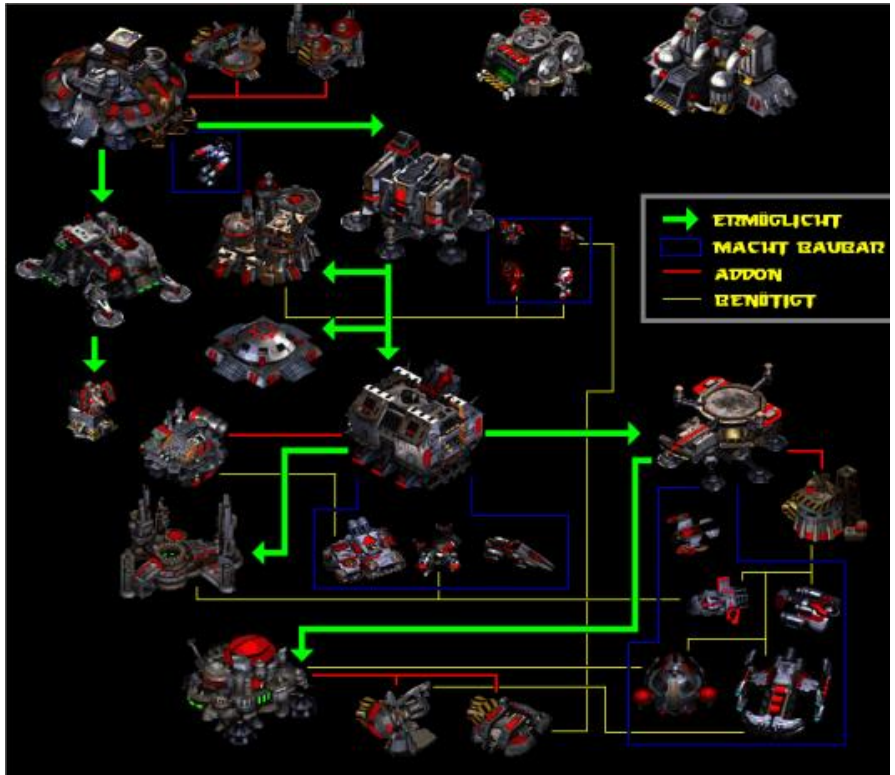
- 깊이 우선 탐색 vs 너비 우선 탐색
  - 구현 방법
    - DFS: 재귀 or 스택
    - BFS: 큐
  - 방문 순서
    - BFS: 거리가 가까운 정점부터 방문
  - 시간 복잡도
    - 인접 행렬 사용하면 $O(|V|^2)$
    - 인접 리스트 사용하면 $O(|V| + |E|)$

질문?

위상 정렬

위상 정렬

- 위상 정렬
  - 간선의 방향을 거스르지 않는 순서대로 정점을 나열하는 것
    - A → B 간선이 있다면 A는 B보다 먼저 나와야 함
    - 위상 정렬 순서대로 정점을 나열했을 때, 간선은 모두 오른쪽을 향하게 됨
    - DAG(사이클 없는 방향 그래프)에서만 가능
    - 스타크래프트에서 valid한 건물의 건설 순서
  - 위상 정렬을 구하는 방법
    - BFS를 응용하는 방법(Kahn's Algorithm)
    - DFS를 응용하는 방법



위상 정렬

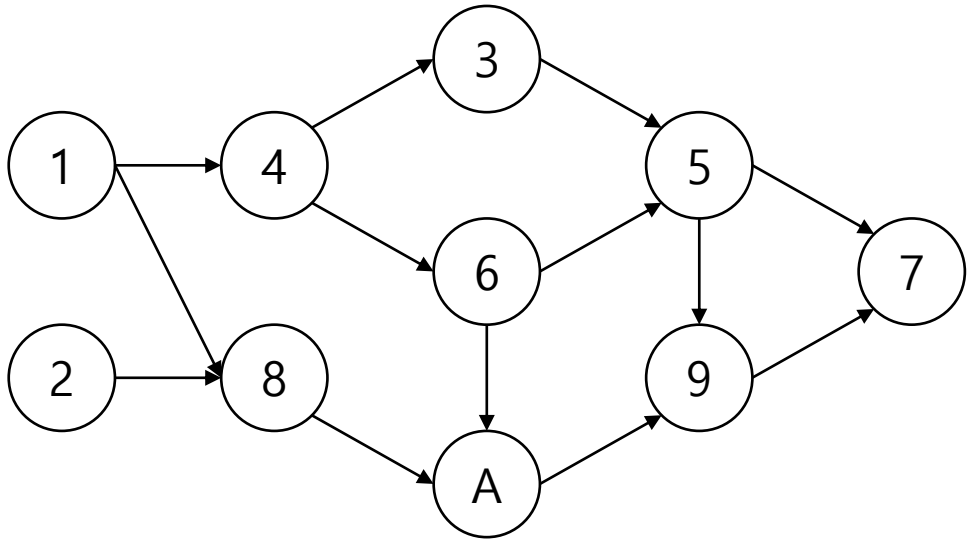
- 위상 정렬의 성질
  - 가장 앞에 오는 정점은 항상 in-degree가 0임
    - 만약 0이 아니라면 그 정점보다 다른 정점이 먼저 나와야 함
  - in-degree가 0이면서 맨 앞이 아닌 정점은 앞으로 옮겨도 됨

위상 정렬

- Kahn's Algorithm
  - in-degree가 0인 정점을 큐에 삽입
  - 큐가 비어 있지 않으면 아래 과정을 반복
    - 큐에서 정점 v를 제거
    - v를 정답에 추가
    - v에서 나가는 간선을 모두 제거, 해당 정점의 in-degree 1 감소
    - in-degree가 0이 된 정점을 큐에 삽입

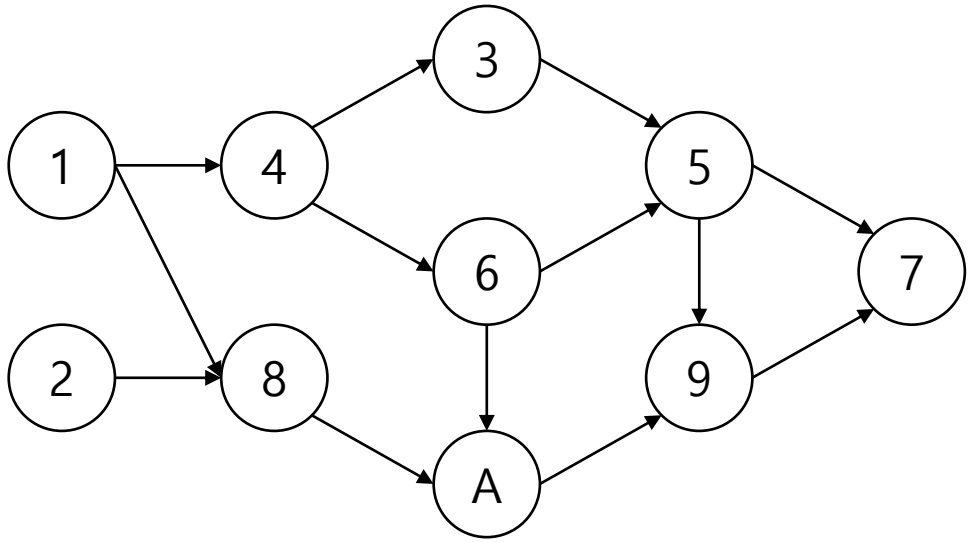
  - 사이클이 존재하면 모든 정점을 방문하기 전에 종료됨

위상 정렬



- Q = {1, 2}
- V = 1 Q = {2, 4}
- V = 2 Q = {4, 8}
- V = 4 Q = {8, 3, 6}
- V = 8 Q = {3, 6}
- V = 3 Q = {6}
- V = 6 Q = {A, 5}
- V = A Q = {5}
- V = 5 Q = {9}
- V = 9 Q = {7}
- V = 7 Q = {}
- 1 2 4 8 3 6 A 5 9 7

위상 정렬



- Q = {2, 1}
- V = 2 Q = {1}
- V = 1 Q = {8, 4}
- V = 8 Q = {4}
- V = 4 Q = {6, 3}
- V = 6 Q = {3, A}
- V = 3 Q = {A, 5}
- V = A Q = {5, 9}
- V = 5 Q = {9}
- V = 9 Q = {7}
- V = 7 Q = {}
- 2 1 8 4 6 3 A 5 9 7

위상 정렬

- Kahn's Algorithm
  - 시간 복잡도: O(|V| + |E|)

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M, In[1010];
vector<int> G[1010];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=M; i++){
        int s, e; cin >> s >> e;
        G[s].push_back(e);
        In[e] += 1;
    }

    queue<int> Q;
    for(int i=1; i<=N; i++) if(!In[i]) Q.push(i);
    while(!Q.empty()){
        int v = Q.front(); Q.pop();
        cout << v << " ";
        for(auto i : G[v]) if(!--In[i]) Q.push(i);
    }
}
```

질문?

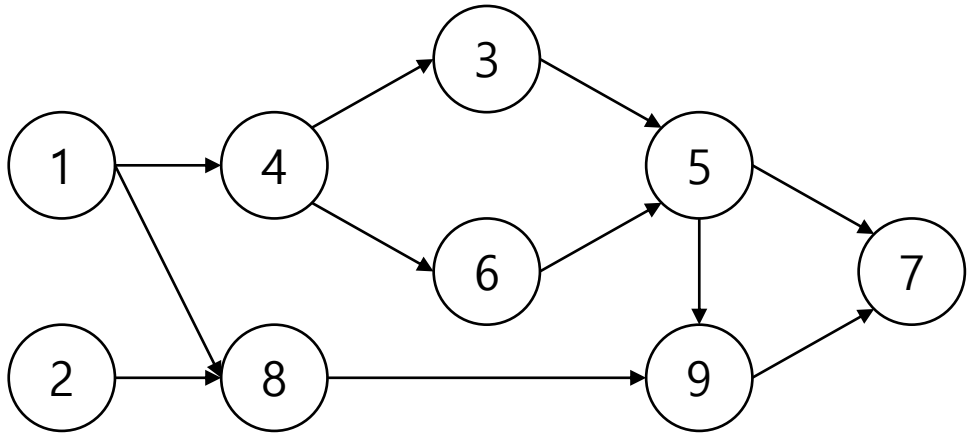
위상 정렬

- 깊이 우선 탐색
  - DFS를 하면서, 정점을 빠져나오는 순서대로 기록
  - 그 순서의 역순은 위상 정렬임

  - 간선 A → B가 있을 때
  - A를 B보다 먼저 방문하면 항상 B를 먼저 탈출함
    - B를 탈출한 다음에 A를 탈출해야 함
  - B를 A보다 먼저 방문하면 항상 B를 먼저 탈출함
    - B를 탈출한 다음에 A를 방문할 수 있음
    - B를 탈출하기 전에 A를 방문하면 사이클

  - 시간 복잡도: O(|V| + |E|)

위상 정렬



- 1 진입
- 4 진입
- 6 진입
- 5 진입
- 7 진입
- 7 탈출(5)
- 9 진입
- 9 탈출(5)
- 5 탈출(6)
- 7 9 5 6 3 4 8 1 2
- 2 1 8 4 3 6 5 9 7

6 탈출(4)
3 진입
3 탈출(4)
4 탈출(1)
8 진입
8 탈출(1)
1 탈출
2 진입
2 탈출

질문?

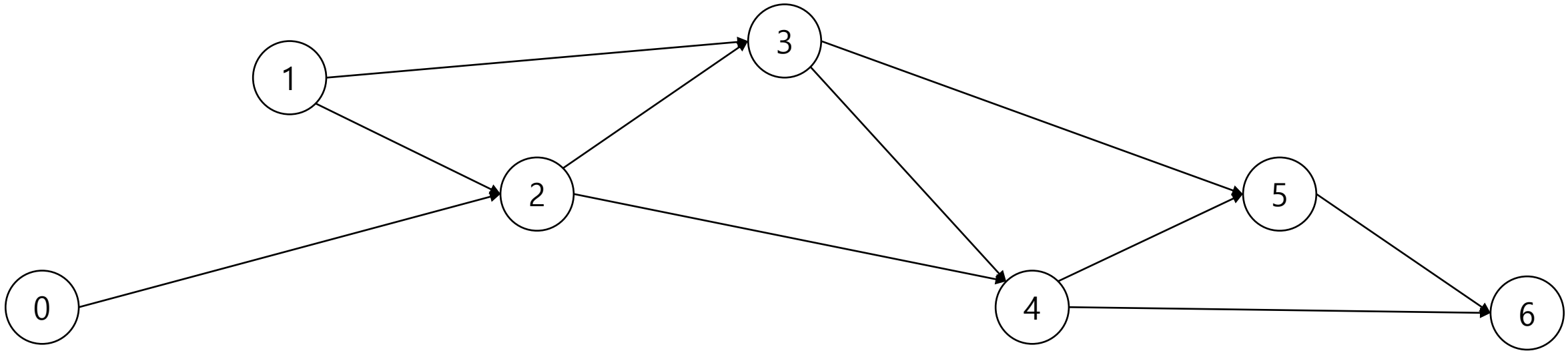
동적 계획법

동적 계획법

- DAG와 DP의 관계
  - DAG: 사이클이 없는 방향 그래프
    - 사이클이 없음
    - 위상 정렬을 할 수 있음
  - DP: 작은 문제의 답을 이용해 큰 문제의 답을 구하는 방법
    - 작은 문제와 큰 문제 간의 연결 관계 → DAG
    - 부분 문제를 위상 정렬 순서대로 해결
      - 재귀 함수 + 메모이제이션

동적 계획법

- DAG와 DP의 관계
  - ex. 피보나치 수
    - 위상 정렬 상에서 앞선 문제의 답을 구해야 뒤에 있는 문제의 답을 구할 수 있음
    - 사이클 없음



동적 계획법

- DAG와 DP의 관계
  - 일반적인 그래프에서 풀지 못하는 문제를 DAG에서는 DP를 이용해 빠르게 해결할 수 있음
    - 최장 경로, S-T 경로의 개수
      - 일반적으로 다항 시간에 해결할 수 없지만 DAG에서는 O(|V| + |E|)
  - 몇몇 DP 문제는 DAG로 생각해서 푸는 것이 편한 경우도 있음
    - 최장 공통 부분 문자열: DAG에서의 최장 경로
    - 동전 교환 경우의 수: DAG에서의 경로 개수
    - ...

질문?

과제

- 필수
  - 1260 DFS와 BFS
  - 7576 토마토
  - 2178 미로 탐색
  - 14567 선수과목
  - 14442 벽 부수고 이동하기 2
- 심화
  - 15971 두 로봇
  - 20530 양분
  - 18780 Timeline
  - 14289 본대산책 3