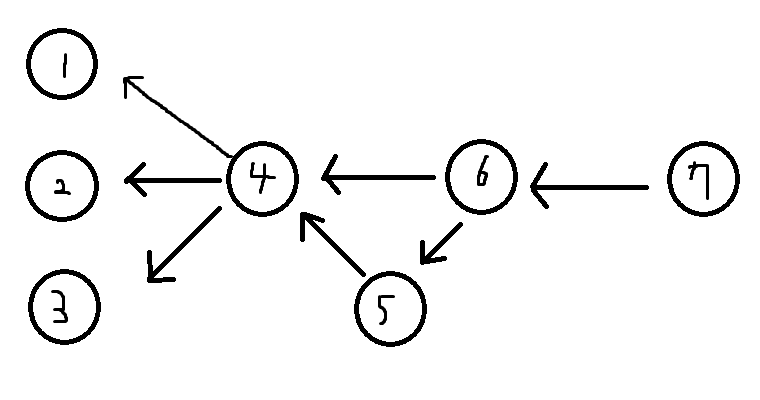
목차

[위상정렬(Topological sort) 2](#_Toc126695766)

[DP(다이나믹 프로그래밍) 5](#_Toc126695767)

위상정렬(Topological sort)

* 위상정렬이란?
  + 사이클이 없는 방향 그래프(DAG: Directed Acyclic Graph)에서 정점 사이의 선후 관계를 위배하지 않고 정렬하는 알고리즘.
  + Ex) 수강순서를 정하는 문제, 키 순서대로 서는 문제
* 위상정렬의 시간복잡도
  + O(V+E) V는 정점의 수, E는 간선의 수
* 알고리즘
  + 내가 이해한 위상정렬의 핵심은. 일을 마치기 전까지는 자신의 순번이 오지 않는다는 것이다. 나는 이를 Workout이라 표하기로 했다. 그래프들은 각각의 간선을 가지고 있고 사이클은 존재하지 않는다.



이러한 그래프를 기준으로 설명한다.

* 설명

1. 각 노드들의 workout은 자신이 가리키고 있는 간선의 개수를 뜻한다.

1,2,3 – 0개 4 – 3개 7 – 1개

5 – 1개 6 – 2개

1. Queue를 준비한다. (재정렬 할 컨테이너)
2. 먼저 전체를 탐색하며 Workout이 0인 노드들을 Queue에 넣는다.
3. Queue에서 하나를 뺀 값을 보관한다 (정렬된 데이터가 들어갈 곳임)
4. Queue에서 뺀 값의 Child들(현재 노드를 가리키는 노드들)의 workout을 1개 감소시킨다. 이는 child노드가 할 일이 1개 줄었다는 것을 의미한다.
5. 만약 4번에서 Child노드의 Workout이 줄어서 0이 됐다면, Queue에 넣는다.
6. 4번으로 돌아가고 Queue에 모든 요소가 없어질때까지 반복한다.

예외 : 만약 이 루프가 끝났는데 정렬되지 않은 노드들이 있다면 그것은 사이클 형태를 띄고있는 것이다.

* 소스코드 (백준 2252번 줄 세우기)

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

class Line {

public:

vector<Line\*> childs;

int workout;

int value;

Line():

workout(0)

,value(0)

{

}

void Push(Line\* child)

{

childs.push\_back(child);

child->workout++;

}

};

int main()

{

std::ios\_base::sync\_with\_stdio(false);

std::cin.tie(NULL);

int N, M;

cin >> N >> M;

Line\* arr;

arr = new Line[N] {};

for (int i = 0; i < M; ++i)

{

int a, b;

cin >> a >> b;

arr[a - 1].Push(&arr[b - 1]);

}

queue<Line\*> que;

for (int i = 0; i < N; ++i)

{

arr[i].value = i + 1;

if (0 == arr[i].workout)

que.push(&arr[i]);

}

while (!que.empty())

{

auto p = que.front();

que.pop();

cout << p->value << " ";

for (auto& child : p->childs)

{

child->workout--;

if (0 == child->workout)

que.push(child);

}

}

delete[] arr;

return 0;

}

DP(다이나믹 프로그래밍)

* 동적 계획법
  + 유래: Richard E. Bellman이 고안한 알고리즘으로 펀딩을 받기 위해 멋있어 보이는 단어인 Dynamic을 사용.
  + 이 알고리즘을 한국어로 비슷하게 해석하면 기억하며 풀기 정도가 된다.
  + 분할정복 알고리즘과 비슷하다. 중복된 하위 문제들을 계산하고, 이것들의 최적 부분구조를 구하며 진행한다.
  + 대표적인 방법론으로는

1. 메모이제이션(Memoization)
2. 타뷸레이션(Tabulation)

이 있다.

여러가지를 보고 느낀 결과 메모이제이션은 재귀형태로 푸는 것, 타뷸레이션은 포문으로 푸는 것 인 것 같다. 본인은 옛날엔 타뷸레이션으로 풀다 메모이제이션이 있다는 것을 알고 메모이제이션만 이용중.

* + 어찌됐건 예는 점화식을 세우는 것이 가장 중요하다. 부분해를 구하는 방식을 생각해내지 못하면 다음으로 시작조차 하지 못한다.
  + 보통 문제에 가치가 매겨져 있을 때, 제일 높은 가치를 받고 싶으면 DP 문제일 확률이 크다.
  + 내가 사용하는 Memoization방법.

1. 원본데이터가 담겨있는 배열, DP에 넣을 배열 두개를 만듬. 보통 백준 같은 코테용 문제를 풀때는 배열의 크기가 들쭉날쭉 하기 때문에 N값을 보고 최대값으로 크게 잡음 ex) int형 값이 최대 100,000까지 들어 갈수 있으면 int Original Data[100’001] 이런 식
2. Dp를 초기화 할 값은, 계산 도중 나올 수 없는 값으로 설정(Invalid라 표현)
3. OriginalData를 초기화 할 값 또한 계산 도중 나올 수 없는 값으로 설정
4. 재귀함수를 만듬.
5. 재귀함수 내부는 먼저 DP에 값이 있는지 확인하고, Invalid가 아니면 그 값을 리턴.
6. 그 다음 원본데이터가 Valid한지 보고 Invalid라면 0을 리턴
7. 이후 DP값을 0으로 Clear해줌. 이제 계산을 시작할 것이기 때문.
8. 그 다음 Step으로 갈 수 있는 경우의 재귀함수를 모두 실행해줌. (이 부분이 가장 생각하기 어려움) 그리고 그 값들 중 가장 높은 것(정답이 높은 걸 원하면 높은 것, 낮은 걸 원하면 낮은 걸 선택)을 구하여 원본 데이터 + 최적해를 dp에 새로 갱신해 주고 그 값을 리턴한다.
9. 그럼 최종적으로 보통 [0]번째 배열에 최적해가 들어있음.
   * 푼 문제
     + 2023-02-07 백준 9465 스티커