**벨만 – 포드 알고리즘**

**:** 벨만 – 포드 알고리즘은 가중 방향 그래프에서 노드 사이의 최단 경로를 찾는 알고리즘

벨만 – 포드 알고리즘은 가중치와 방향을 갖는 엣지를 가지는 그래프에서 한 정점으로부터 다른 정점까지의 최단 거리를 찾는 알고리즘으로, 그 목표는 다익스트라 알고리즘과 같다.

하지만 이 알고리즘의 시간 복잡도는 O(VE)로, 다익스트라 알고리즘보다 일반적으로는 더 느리다.

같은 목표를 가지고 더 느린 시간 복잡도를 가진다면 왜 굳이 이 벨만 – 포드 알고리즘을 사용하는 것일까?

그 이유는 바로 다익스트라 알고리즘으로는 처리할 수 없는 ‘음수의 가중치’를 갖는 엣지를 처리할 수 있기 때문이다.

그래서 코드를 작성할 때 다익스트라 알고리즘과 다른 점은 바로 음수 사이클이 있는지 판단한다는 것이다.

음수 사이클이 있는지 코드 내에서 체크하고, 만약 있다면 사이클을 돌 때마다 간선을 완화한다. 그렇기 때문에 간선을 V-1번까지 확인했을 때까지도 완화할 수 있는 엣지가 있다면 음수 사이클이 있다고 판단할 수 있다.

BellmanFord(G,w,s):

//초기화 과정

for each u in G.V: //노드를 초기화 하기

distance[v] = inf //모든 노드의 최단거리를 무한으로 지정

parent[v] = null //모든 노드의 부모 노드를 널값으로 지정

distance[s] = 0 //출발점의 최단거리는 0으로 지정한다

//거리측정 과정

for i from 1 to len(G.V): //노드의 숫자만큼

for each (u,v) in G.E: //모든 변을 체크해 최단 거리를 찾아본다.

if distance[u] + w[(u,v)] < distance[v]:

//만약 u를 경유하여 v로 가는 거리가 현재 v의 최단 거리보다 짧으면

distance[v] = distance[u] + w[(u,v)] //그 거리를 v의 최단거리로 지정

parent[v] = u //u를 v의 부모 노드로 지정

//음수 사이클 체크 과정

for each (u,v) in G.E:

if distance[u] + w[(u,v)] < distance[v]:

return false //음수 사이클을 확인하고 알고리즘을 정지

return distance[], parent[]

출처: https://namu.wiki/w/%EB%B2%A8%EB%A8%BC-%ED%8F%AC%EB%93%9C%20%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98

**Leetcode 743 – 다익스트라 알고리즘**

class Solution:

def networkDelayTime(self, times, n: int, k: int) -> int:

graph = collections.defaultdict(list)

# 그래프 인접 리스트 구성

for u,v,w in times:

graph[u].append((v,w))

# 큐 변수: [(소요 시간, 정점)]

Q = [(0, k)]

dist = collections.defaultdict(int)

# 우선순위 큐 최솟값 기준으로 정점까지 최단 경로 삽입

while Q:

time, node = heapq.heappop(Q)

if node not in dist:

dist[node] = time

for v, w in graph[node]:

alt = time + w

heapq.heappush(Q, (alt,v))

# 모든 노드의 최단 경로 존재 여부 판별

if len(dist) == n:

return max(dist.values())

return -1

출처: https://cool-developer.tistory.com/73

**Leetcode 743 - 벨만 – 포드 알고리즘**

class Solution {

public:

int networkDelayTime(vector<vector<int>>& times, int N, int K) {

constexpr int MAX\_TIME = 101 \* 100;

vector<int> dist(N, MAX\_TIME);

dist[K - 1] = 0;

for (int i = 1; i < N; ++i)

for (const auto& time : times) {

int u = time[0] - 1, v = time[1] - 1, w = time[2];

dist[v] = min(dist[v], dist[u] + w);

}

int max\_dist = \*max\_element(dist.begin(), dist.end());

return max\_dist == MAX\_TIME ? -1 : max\_dist;

}

};

출처: https://zxi.mytechroad.com/blog/graph/leetcode-743-network-delay-time/

**백준 1916 – 다익스트라 알고리즘**

#include <iostream>

#include <vector>

#include <queue>

#define INF 1e9 // dist 배열에 저장한 무한대 정의

using namespace std;

int main() {

cin.tie(NULL);

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

int N,M; cin>>N>>M; // 도시의수, 버스의 수

vector<pair<int,int> > arr[N+1]; // 각 노드에 연결된 [정점, 비용] 저장할 배열

bool visited[N+1];

fill(visited,visited+N+1,false);

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

int from,to,val; // from에서 to까지 걸리는 비용 val

cin>>from>>to>>val; // 각 경로 정보 입력

arr[from].push\_back(pair<int,int>(to,val));

}

int dist[N+1]; // 시작점에서 각 노드까지의 최단거리를 저장해주는 배열

fill(dist,dist+N+1,INF); // 모든 거리의 비용을 최대로 설정

priority\_queue< pair<int,int> ,vector<pair<int,int> >,greater<pair<int,int> > > qu;

// 작은 비용을 가진 정점으로 자동 정렬되는! -> [from에서 to까지의 비용, to정점]

int start, end; cin>>start>>end; // 출발점, 도착점

qu.push(pair<int,int>(0,start)); // 출발점 저장

dist[start]=0; // 시작점의 최단거리 (자기 자신까지의 거리 = 0)

while(!qu.empty()){

int cost = qu.top().first; // 현재 탐색할 정점의 비용

int here = qu.top().second; // 현재 탐색할 정점의 번호

qu.pop();

if(!visited[here]){ // 이미 방문한 정점은 pass

visited[here]=true;

if(here==end) continue; // 도착점이라면 pass

for(int i=0; i<arr[here].size(); i++){

// 현재 정점에서 연결된 정점들까지의 비용 탐색

int next = arr[here][i].first;

int nextCost = arr[here][i].second;

if(dist[next] > dist[here] + nextCost){

// 여기까지 온 거리가 다른 경우보다 최단이라면, 갱신

dist[next] = dist[here] + nextCost;

qu.push(pair<int,int>(dist[next],next));

}

}

}

}

cout<<dist[end];

return 0;

}

출처: https://velog.io/@szun8/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EB%B0%B1%EC%A4%80-%EB%8B%A4%EC%9D%B5%EC%8A%A4%ED%8A%B8%EB%9D%BC%EB%A5%BC-%EC%9D%B4%EC%9A%A9%ED%95%9C-1916.-%EC%B5%9C%EC%86%8C%EB%B9%84%EC%9A%A9-%EA%B5%AC%ED%95%98%EA%B8%B0

**백준 1916 – 벨만 – 포드 알고리즘**

#include <iostream>

#include <vector>

#define INF 987654321

using namespace std;

/\*

벨만포트 알고리즘

마지막 V번에서도 완화가 되면 음수 사이클이 존재함을 알려준다.

\*/

int V,M;

vector<pair<int, int> > v[1001];

int upper[1001];

int bellmanFord(int src) {

//시작점 제외 모든 정점까지의 최단거리 INF로 초기화

fill\_n(upper, 1001, INF);

upper[src] = 0;

bool updated;

for(int i = 0; i < V; i++) {

updated = false;

for(int j = 1; j <= V; j++)

for(int k = 0; k < v[j].size(); k++) {

int there = v[j][k].first;

int cost = v[j][k].second;

if(upper[there] > upper[j] + cost) { // 완화 성공

upper[there] = upper[j] + cost;

updated = true;

}

}

if(!updated) break; //모든 간선에 대해 완화가 실패할경우 break;

}

if(updated) return 1; //음수 사이클이 존재

return 0;

}

int main(void) {

cin >> V >> M;

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

int a,b,c;

cin >> a >> b >> c;

v[a].push\_back(make\_pair(b, c));

}

int start, arrive; cin >> start >> arrive;

if(!bellmanFord(start))

cout<<upper[arrive];

return 0;

}

출처: https://huiung.tistory.com/106