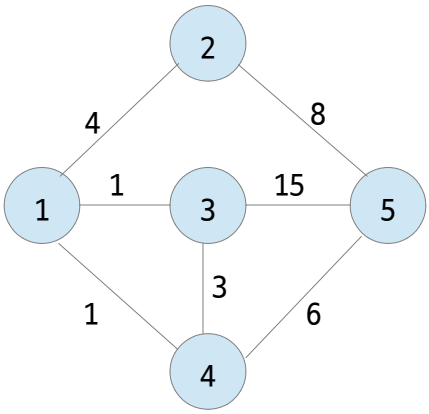
플로이드 알고리즘

sianke1991

1. 비용 테이블

가중치가 주어진 그래프에 대해서 모든 정점 쌍에 대해 최단 거리를 구하는 알고리즘으로, 유향·무향 그래프 모두 적용할 수 있고, 음의 가중치를 갖는 간선이 포함된 그래프에도 적용할 수 있다. 단, 그래프에 음의 사이클이 있으면 (폐곡선에 가중치의 합이 음이 되는 경우가 있으면) 이 알고리즘을 적용할 수 없다.



이 무방향 그래프에 대해 플로이드 알고리즘을 적용하면
s(1≤s≤5)에서 출발하여 t(1≤t≤5)에 도착하는데 필요한 최소 비용을 구할 수 있다.

Step1. 대각 성분은 0, 나머지 성분은 무한대의 값을 갖는 비용 테이블을 준비한다.
비용 테이블에서 s행 t열 성분은 s 노드에서 시작하여 t 노드에 도착하는데 필요한 비용을 의미한다.

출발 노드 \ 도착 노드	1	2	3	4	5
1	0	∞	∞	∞	∞
2	∞	0	∞	∞	∞
3	∞	∞	0	∞	∞
4	∞	∞	∞	0	∞
5	∞	∞	∞	∞	0

Step2. 그래프에 있는 간선의 가중치를 비용 테이블에 적는다.

출발 노드 \ 도착 노드	1	2	3	4	5
1	0	4	1	1	∞
2	4	0	∞	∞	8
3	1	∞	0	3	15
4	1	∞	3	0	6
5	∞	8	15	6	0

Step3. 1번 노드를 경유하여 도착 노드에 가는 것이 더 효율적인 경우 비용 데이터를 갱신한다.
예를 들어 3번 노드에서 시작하여 4번 노드에 도착하는 경우 직접 3 → 4로 가는 것 보다 3 → 1 → 4가 더 효율적이므로 이 값은 갱신한다.
1번 노드를 경유하여 도착 노드에 가는 비용은 비용 테이블[s][1] + 비용 테이블[1][t]로 구할 수 있다. 즉,

```
if (비용 테이블[s][t]>비용 테이블[s][1]+비용 테이블[1][t]) {
    비용 테이블[s][t] = 비용 테이블[s][1]+비용 테이블[1][t];
}
```

으로 적을 수 있다.

출발 노드 \ 도착 노드	1	2	3	4	5
1	0	4	1	1	∞
2	4	0	5	5	8
3	1	5	0	2	15
4	1	5	2	0	6
5	∞	8	15	6	0

Step4. 2번 노드를 경유하여 도착 노드에 가는 것이 더 효율적인 경우 비용 데이터를 갱신한다.

```
if (비용 테이블[s][t]>비용 테이블[s][2]+비용 테이블[2][t]) {
    비용 테이블[s][t] = 비용 테이블[s][2]+비용 테이블[2][t];
}
```

출발 노드 \ 도착 노드	1	2	3	4	5
1	0	4	1	1	12
2	4	0	5	5	8
3	1	5	0	2	13
4	1	5	2	0	6
5	12	8	13	6	0

Step5. 1번 노드, 2번 노드에 적용했던 step3, step4의 과정을 나머지 노드, 3·4·5 번 노드에 대해 적용한다. 즉, step3, step4, step5의 과정을 한 데 묶어서 3중 for loop로 작성할 수 있다.

```
for (int k=1; k<=5; k++) { //k 노드를 경유하는 경우를 고려- (여기서 5는 전체 노드의 개수)
    for (int s=1; s<=5; s++) {
        for (int t=1; t<=5; t++) {
            if (비용 테이블[s][t]>비용 테이블[s][k]+비용 테이블[k][t]) { //s 노드에서 시작해서 t 노드에 도착하는데 k 노드를 경유하는
                                                                        // 것이 더 효율적인 경우-
                비용 테이블[s][t] = 비용 테이블[s][k]+비용 테이블[k][t];
            }
        } //t loop (도착 노드 루프)
    } //s loop (시작 노드 루프)
} //k loop (경유 노드 루프)
```

출발 노드 \ 도착 노드	1	2	3	4	5
1	0	4	1	1	7
2	4	0	5	5	8
3	1	5	0	2	8
4	1	5	2	0	6
5	7	8	8	6	0

step5까지 적용했다면 모든 노드 쌍에 대해서 최적의 비용을 구한 것이다. 예를 들어 1 번 노드에서 시작하여 5 번 노드에 도착하는데 필요한 비용은 비용 테이블[1][5]의 값인 7이 된다.

Note.

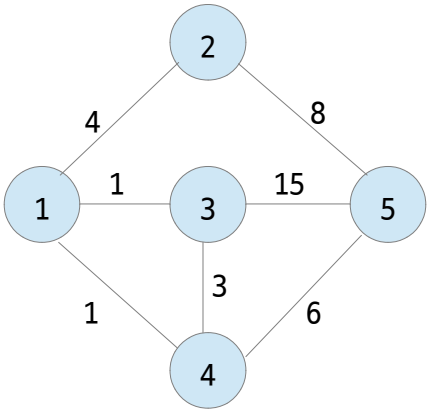
① 실제 코드 상 무한대의 값은 덧셈을 통해서 도달하지 못할 적당히 큰 값을 사용한다. 즉, 무한대가 아닌 비용을 더하는 과정으로는 도달하지 않고, 무한대의 값 두 개를 더해서 오버플로가 나지 않는 적당히 큰 값을 사용한다.

② 경유 노드 루프는 플로이드 알고리즘의 각 단계로써, 시작 노드 루프 및 도착 노드 루프를 감싸는 가장 바깥 루프가 되어야 한다.

③ 플로이드 알고리즘을 적용하여 모든 노드 쌍에 대해 최적의 비용을 계산했음에도 불구하고 무한대의 비용이 나온다면 해당 노드 쌍은 서로 끊어진 것이다. 비용 테이블[3][6]이 무한대인 상태로 알고리즘이 종료되었다면 3 번 노드에서 시작하여 6 번 노드에 도착할 방법은 없는 것이다.

2. 이웃 테이블

위의 방법을 사용하면 모든 노드 쌍에 대해서 최적의 비용을 구할 수 있으나 해당 비용이 나오는 그 경로는 구할 수 없게 된다. 이 경로를 구하기 위해 중간 단계로 이웃 테이블을 구성한다. 이웃 테이블[s][t]는 s 노드에서 시작하여 t 노드에 도착할 때 최적의 비용을 얻기 위해 s 노드에서 가야 할 다음 노드를 의미한다.



이 예시에서 3 번 노드에서 시작하여 5 번 노드에 최적의 비용으로 도착하기 위해서는 3 → 1 → 4 → 5의 경로를 사용해야 한다. 즉, 3 번 노드의 다음 노드는 1 번 노드가 되므로 이웃 테이블[3][5] = 1 이다.

Step1. 모든 성분이 -1인 이웃 테이블을 준비한다.

출발 노드 \ 도착 노드	1	2	3	4	5
1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1
5	-1	-1	-1	-1	-1

Step2. 그래프의 모든 간선 (s, t)에 대해 이웃 테이블[s][t] = t로 설정한다. (즉, 어떠한 노드도 경유하지 않는다면 최적의 비용은 직통 간선을 사용하는 것이다.)

출발 노드 \ 도착 노드	1	2	3	4	5
1	-1	2	3	4	-1
2	1	-1	-1	-1	5
3	1	-1	-1	4	5
4	1	-1	3	-1	5
5	-1	2	3	4	-1

Step3. 1번 노드를 경유하여 도착 노드에 가는 것이 더 효율적인 경우 이웃 데이터를 갱신한다.
예를 들어 3번 노드에서 시작하여 4번 노드에 도착하는 경우 직접 3 → 4로 가는 것 보다 3 → 1 → 4가 더 효율적이므로 이 값은 갱신한다.
이 때 이웃 테이블[3][4] = 1 로 갱신하는 것이 아닌, 이웃 테이블[3][4] = 이웃 테이블[3][1] 로 갱신한다.

```
if (비용 테이블[s][t]>비용 테이블[s][1]+비용 테이블[1][t]) {
    비용 테이블[s][t] = 비용 테이블[s][1]+비용 테이블[1][t];
    이웃 테이블[s][t] = 이웃 테이블[s][1]; //s 노드에서 t 노드로 갈 때 1번 노드를 거쳐 가는 것이 더 효율적이므로 [s 노드에서 t 노드로
                                           // 갈 때 이웃 노드]는 [s 노드에서 1번 노드로 갈 때 이웃 노드]로 갱신한다.
}
```

출발 노드 \ 도착 노드	1	2	3	4	5
1	-1	2	3	4	-1
2	1	-1	1	1	5
3	1	1	-1	1	5
4	1	1	1	-1	5
5	-1	2	3	4	-1

Step4. 위의 과정을 2번 노드, 3번 노드 … 5번 노드에 대해서도 똑같이 적용한다.
따라서 이웃 테이블을 구성하는 경우 플루이드 알고리즘의 3중 for loop는 다음과 같은 모양이 된다.

```
for (int k=1; k<=5; k++) { //k 노드를 경유하는 경우를 고려- (여기서 5는 전체 노드의 개수)
    for (int s=1; s<=5; s++) {
        for (int t=1; t<=5; t++) {
            if (비용 테이블[s][t]>비용 테이블[s][k]+비용 테이블[k][t]) { //s 노드에서 시작해서 t 노드에 도착하는데 k 노드를 경유하는
                                                                        // 것이 더 효율적인 경우-
                비용 테이블[s][t] = 비용 테이블[s][k]+비용 테이블[k][t];
                이웃 테이블[s][t] = 이웃 테이블[s][k];
            }
        } //t loop (도착 노드 루프)
    } //s loop (시작 노드 루프)
} //k loop (경유 노드 루프)
```

출발 노드 \ 도착 노드	1	2	3	4	5
1	-1	2	3	4	4
2	1	-1	1	1	5
3	1	1	-1	1	1
4	1	1	1	-1	5
5	4	2	4	4	-1

일단 이웃 테이블을 완성했다면 재귀적인 방법을 통해 s 노드에서 t 노드로 가는 최적의 경로를 구할 수 있다. 예를 들어 1번 노드에서 5번 노드로 이동하는 최적의 경로를 찾아본다.

이웃 테이블[1][5] = 4 이다. 즉, 1번 노드에서 5번 노드로 최적의 경로로 가기 위해서는 1번 노드에서 4번 노드로 이동해야 한다. 따라서 1번 노드에서 5번 노드로 가는 최적의 경로는 다음과 같은 그림이 나온다.

1번 → 4번 ---> 5번 (4번 노드에서 5번 노드로 갈 때 한 번에 갈 수 있다는 보장은 없다.)

4번 노드에서 5번 노드로 가는 최적의 경로를 찾으면 전체 경로인 1번 노드에서 5번 노드로 가는 최적의 경로를 찾을 수 있다.

이웃 테이블[4][5] = 5 이다. 즉, 4번 노드에서 5번 노드로 최적의 경로로 가기 위해서는 4번 노드에서 5번 노드로 이동한다. 본래 목표인 5번 노드에 도착했으므로 전체 경로 찾기를 종료한다.

1번 노드에서 시작하는 전체의 과정과 4번 노드에서 시작하는 부분의 과정이 유사하므로 재귀의 방법을 사용할 법 하다. 전체 경로를 공백 문자로 구분하는 문자열을 반환하는 함수를 구성해보면 다음과 같다.

```
string path(int from, int to) {
    if (from==to ¦¦ 비용 테이블[from][to]>=INFINITE_PRICE) { //from 노드에서 to 노드로 갈 수 있는 방법이 없는 경우에 재귀를 호출한다면
                                                            //재귀가 종료되지 않아 스택오버플로가 발생할 수 있다. 따라서 from 노드
                                                            //에서 to 노드로 이동할 수 없다면 재귀 호출하면 안된다. from 노드에서 to
                                                            //노드로 이동할 수 없는지 여부 판단은 비용 테이블의 값으로 내릴 수 있다.
        return “”;
    }
    int mid = 이웃 테이블[from][to]; //from에서 to로 가기 위해서 경유해야 하는 노드
    if (mid==to) { //from에서 to로 가는데 직통 간선을 사용하는 것이 효율적인 경우. (재귀 종료 조건)
        return to_string(from) + “ “ + to_string(to);
    } else {
        return to_string(from) + “ “ + path(mid, to); //mid에서 to로 가는 경로는 재귀 호출을 통해서 구한다.
    }
}
```

하지만 이 방법으로 모든 노드 쌍에 대해서 경로를 구하면 시간이 오래 걸릴 수 있다. 모든 노드 쌍에 대해서 경로를 구할 때 걸리는 시간을 줄이는 방법을 생각해 본다. 위에서는 1번 노드에서 5번 노드로 가는 경로를 구했는데 이번에는 3번 노드에서 5번 노드로 가는 방법을 구해본다.

이웃 테이블[3][5] = 1 이다. 즉, 3번 노드에서 5번 노드로 최적의 경로로 가기 위해서는 3번 노드에서 1번 노드로 이동해야 한다.

3번 → 1번 ---> 5번

그런데 1번 노드에서 5번 노드로 가는 경로는 한 번 계산했었다. 이 계산 결과를 저장해 두었다면 3번 노드에서 5번으로 가는 경로를 구하는데 사용할 수 있다.

한 번 계산한 최적 경로를 버리지 않고 저장해 둠으로써 다음 최적 경로 계산에 걸리는 시간을 줄일 수 있다. 즉, 여기서는 메모이제이션을 사용한다.

3. 경로 테이블, 경로 길이 테이블

모든 노드 쌍에 대해서 최적 경로를 계산하는데 한 번 계산한 경로는 저장해두고 다음 경로를 계산할 때 다시 참조할 수 있도록 한다. 이를 위해 경로 테이블을 둔다.

```
string 경로 테이블[5][5]; //경로 테이블[s][t]는 s노드에서 시작하여 t노드로 갈 때 거쳐할 노드들을 공백 문자로 구분하여 작성된 문자열
                          //전역변수로 선언한 이 2차원 배열의 각 원소는 빈 문자열(“”)로 초기화되고, 이 값을 초기값으로 그대로 사용한다.
int 경로 길이 테이블[5][5]; //경로 길이 테이블[s][t]는 s노드에서 시작하여 t노드로 갈 때 거쳐할 노드의 개수
                            //전역변수로 선언한 이 2차원 배열의 각 원소는 0으로 초기화되고, 이 값을 초기값으로 그대로 사용한다.
```

```
/**
 * from에서 to로 가기 위해 거쳐야 할 경로를 반환한다.<br/>
 * 계산 중 경로 테이블의 내용을 참조하고 갱신한다.<br/>
 * 이 함수 전 비용 테이블 및 이웃 테이블은 완성되어 있어야 한다.
 */
string path(int from, int to) {
    if (from==to ¦¦ 비용 테이블[from][to]>=INFINITE_PRICE) {
         return “”;
    }
    if (경로 테이블[from][to]!=””) { //from에서 to로 가는 경로를 이미 구한 경우 경로 테이블에 저장된 값을 그대로 반환한다.
        return 경로 테이블[from][to];
    } else {
        int mid = 이웃 테이블[from][to]; //경유 노드
        if (mid==to) { //경유 노드가 도착 노드와 같은 경우 from에서 to로 직접 가는 것이 가장 빠른 것이다. (재귀 호출 종료 조건)
            경로 테이블[from][to] = to_string(from) + “ “ + to_string(to);
            경로 길이 테이블[from][to] = 2; //시작 노드와 도착 노드, 이렇게 2 개의 노드를 거친다.
            return 경로 테이블[from][to];
        } else { //경유 노드가 도착 노드와 다른 경우 from에서 mid를 거쳐서 to로 가야 한다. mid에서 to로 가는 경로는 재귀 호출을
                 //통해서 구한다.
            경로 테이블[from][to] = to_string(from) + “ “ + path(mid, to); //mid는 path(mid, to)의 일부이므로 to_string(mid)는
                                                                          //더하면 안된다.
            경로 길이 테이블[from][to] = 1+경로 길이 테이블[mid][to]; //경로 테이블[from][to]는 경로 테이블[mid][to] 앞에 from이
                                                                     //붙은 것이다. 즉, 경로 길이는
                                                                     //pathLengthMatrix[from][to]는 pathLengthMatrix[mid][to] 보다
                                                                     //1 큰 값이다.
            return 경로 테이블[from][to];
        }
    }
}
```

비용 테이블 및 이웃 테이블이 완성한 다음 path 함수를 호출함으로써 경로 테이블 및 경로 길이 테이블을 완성시킬 수 있다.

```
for (int s=1; s<=numNodes; s++) {
    for (int t=1; t<=numNodes; t++) {
        path(s, t);
    } //t loop (도착 노드 루프)
} //s loop (출발 노드 루프)
```