Data Structure

Project 3

학번: 2020202054

이름: 이종혁

학과: 컴퓨터정보공학부

수강과목: 데이터구조설계(월6/수5)

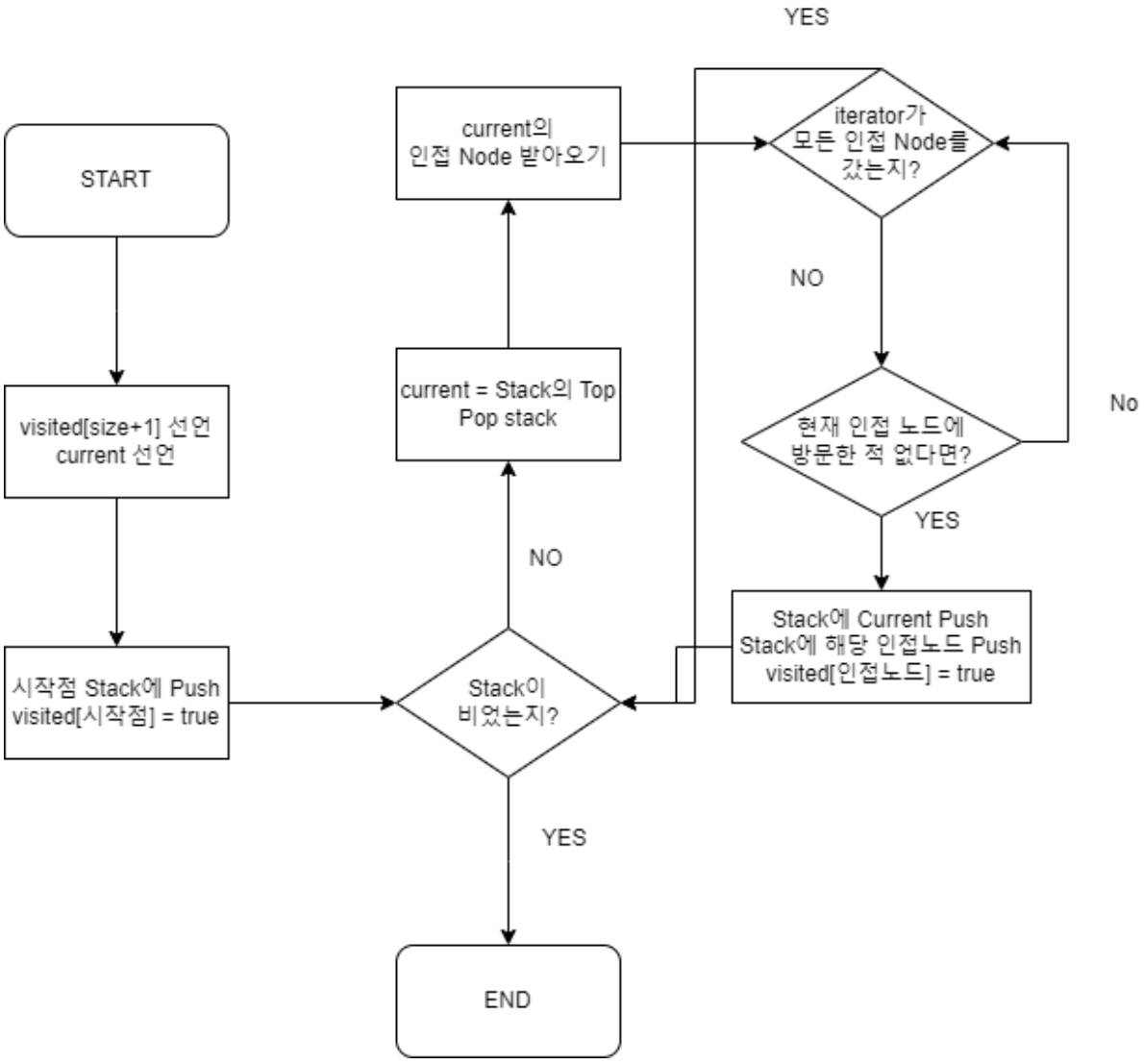
데이터구조실습(수7)

- Introduction

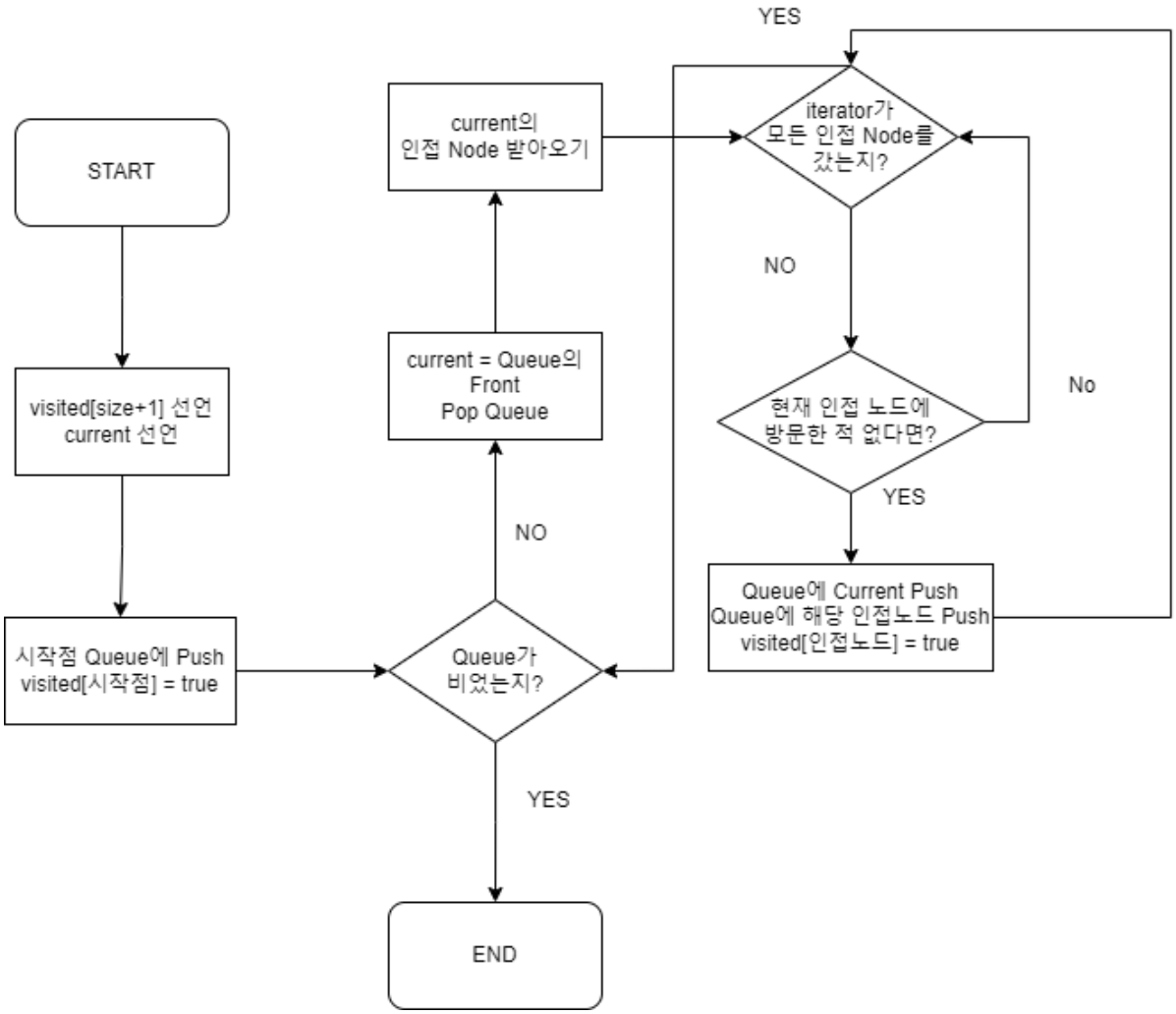
이번 프로젝트는 그래프 데이터를 Adjacency List, 혹은 Adjacency Matrix 형태로 입력 받아, 요구하는 7 가지 그래프 탐색 알고리즘을 수행하는 것이 주된 내용입니다. 구현해야 하는 그래프 탐색 알고리즘으로는 DFS, BFS, Kwangwoon(프로젝트 내에서 새롭게 주어진 탐색 방법), Kruskal, Dijkstra, Bellman-Ford, Floyd 총 7 가지입니다. 각 탐색 방법에 따라 방향성을 사용하거나 안 하는 경우도 있으며, 가중치를 사용하는 경우와 사용하지 않는 경우도 있습니다.

- Flowchart

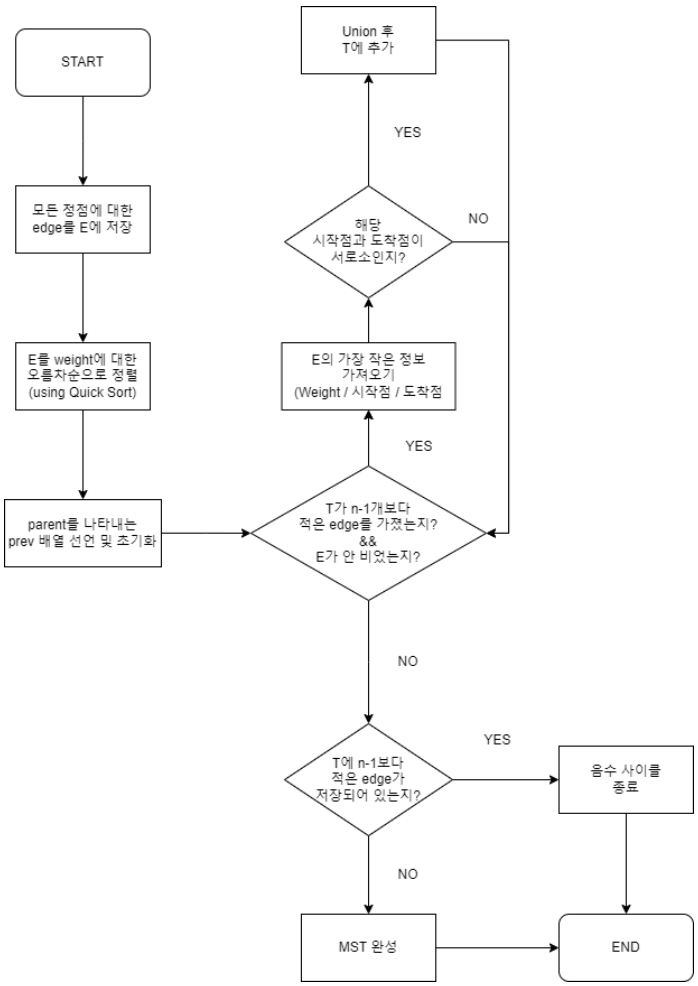
[DFS]



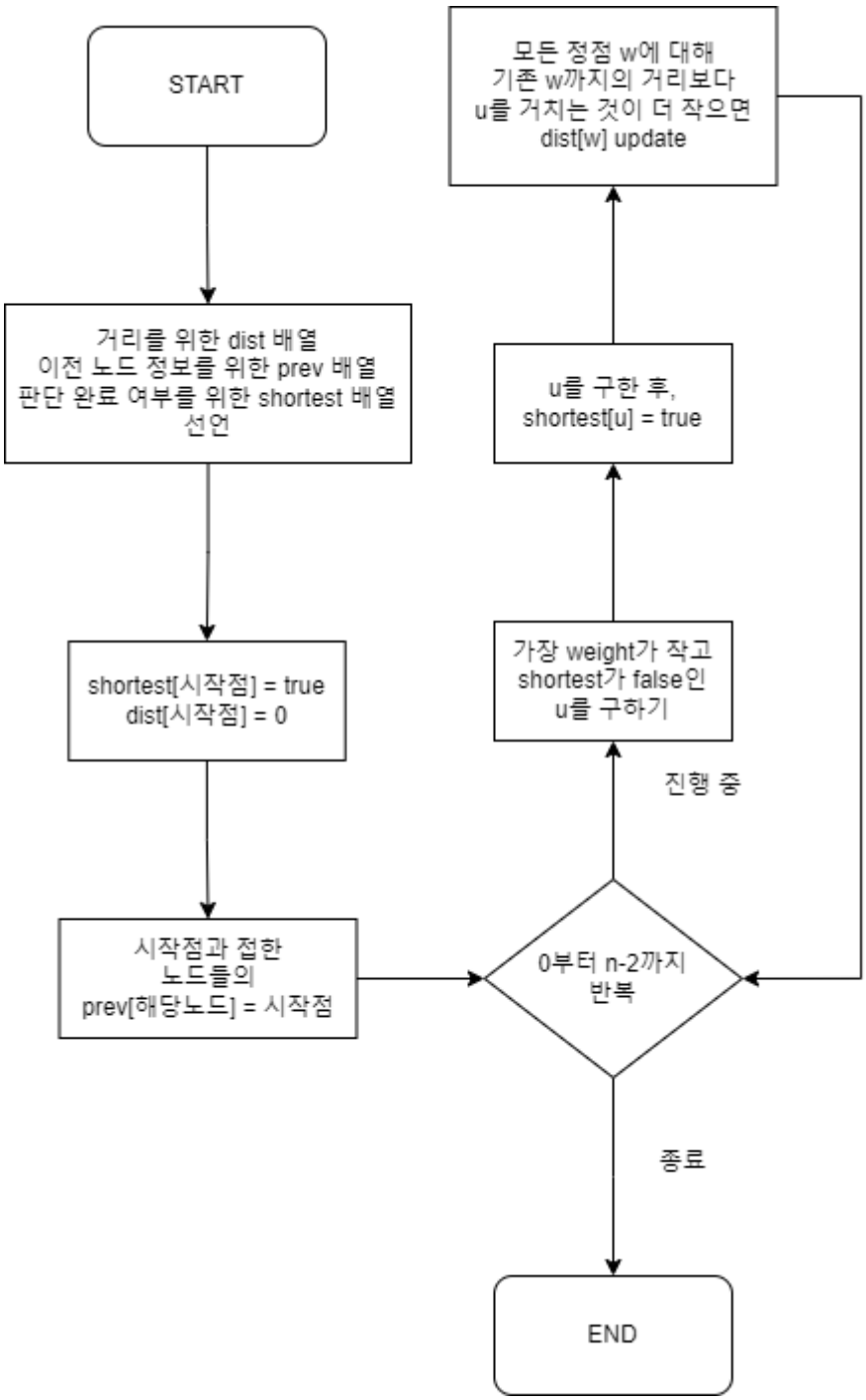
[BFS]


START
visited[size+1] 선언
current 선언
시작점 Queue에 Push
visited[시작점] = true
Queue가
비었는지?
YES
END
NO
current = Queue의
Front
Pop Queue
current의
인접 Node 받아오기
iterator가
모든 인접 Node를
갔는지?
YES
Queue에 Current Push
Queue에 해당 인접노드 Push
visited[인접노드] = true
NO
현재 인접 노드에
방문한 적 없다면?
No

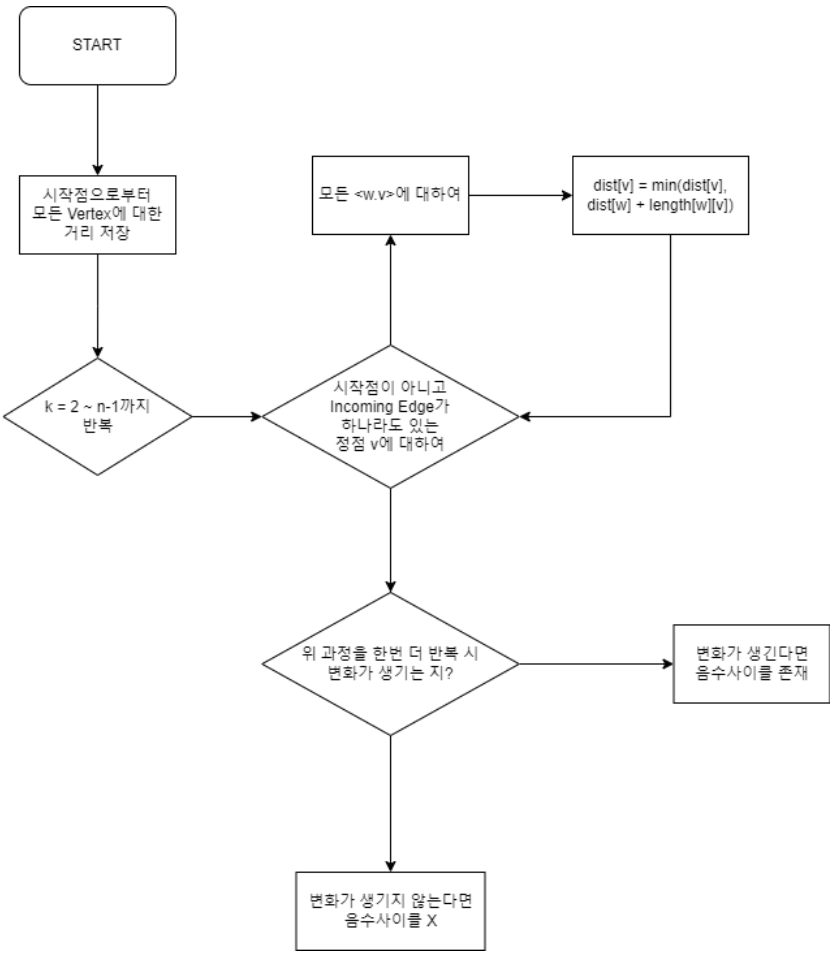
[KRUSKAL]


START
모든 정점에 대한
edge를 E에 저장
E를 weight에 대한
오름차순으로 정렬
(using Quick Sort)
parent를 나타내는
prev 배열 선언 및 초기화
T가 n-1개보다
적은 edge를 가졌는지?
&&
E가 안 비었는지?
YES
E의 가장 작은 정보
가져오기
(Weight / 시작점 / 도착점
해당
시작점과 도착점이
서로소인지?
NO
YES
Union 후
T에 추가
NO
T에 n-1보다
적은 edge가
저장되어 있는지?
YES
음수 사이클
종료
NO
MST 완성
END

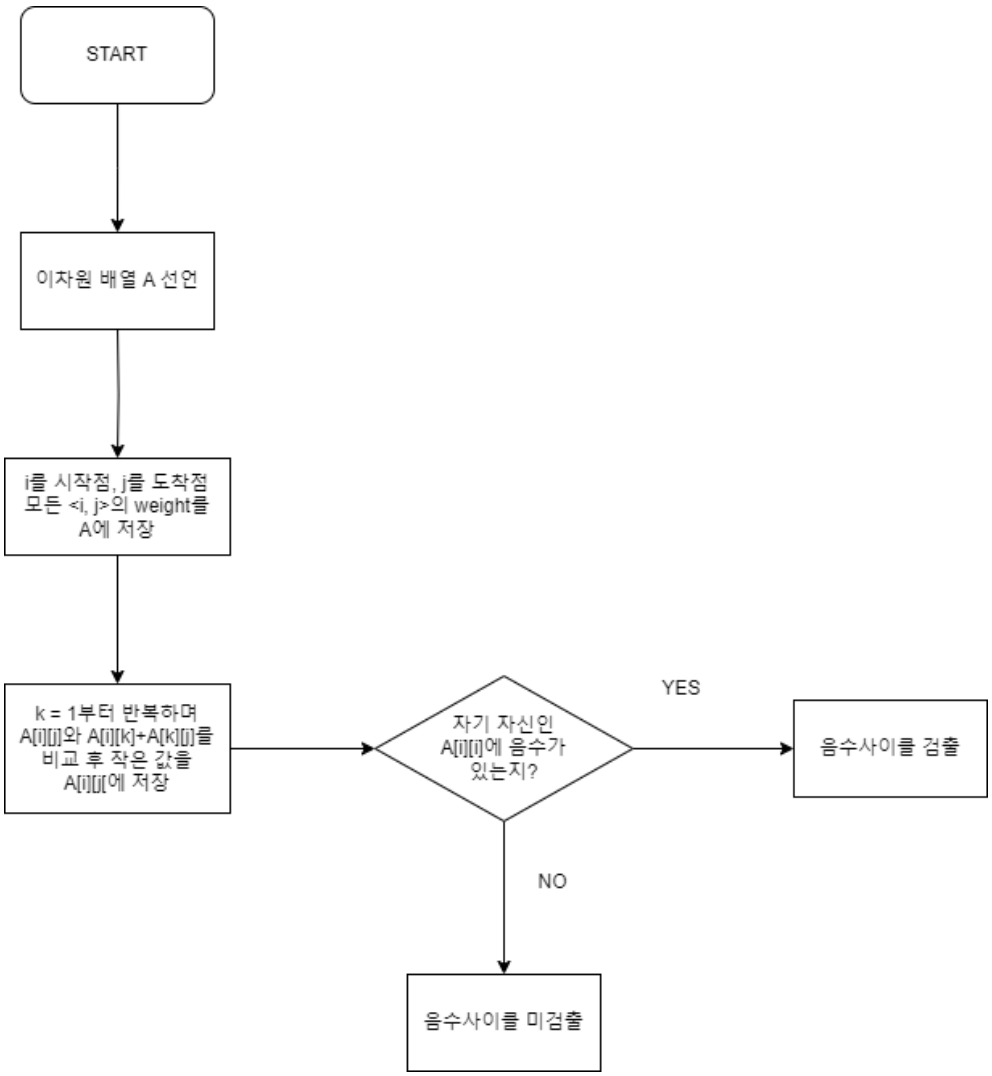
[DIJKSTRA]


START
거리를 위한 dist 배열
이전 노드 정보를 위한 prev 배열
판단 완료 여부를 위한 shortest 배열
선언
shortest[시작점] = true
dist[시작점] = 0
시작점과 접한
노드들의
prev[해당노드] = 시작점
0부터 n-2까지
반복
진행 중
가장 weight가 작고
shortest가 false인
u를 구하기
u를 구한 후,
shortest[u] = true
모든 정점 w에 대해
기존 w까지의 거리보다
u를 거치는 것이 더 작으면
dist[w] update
종료
END

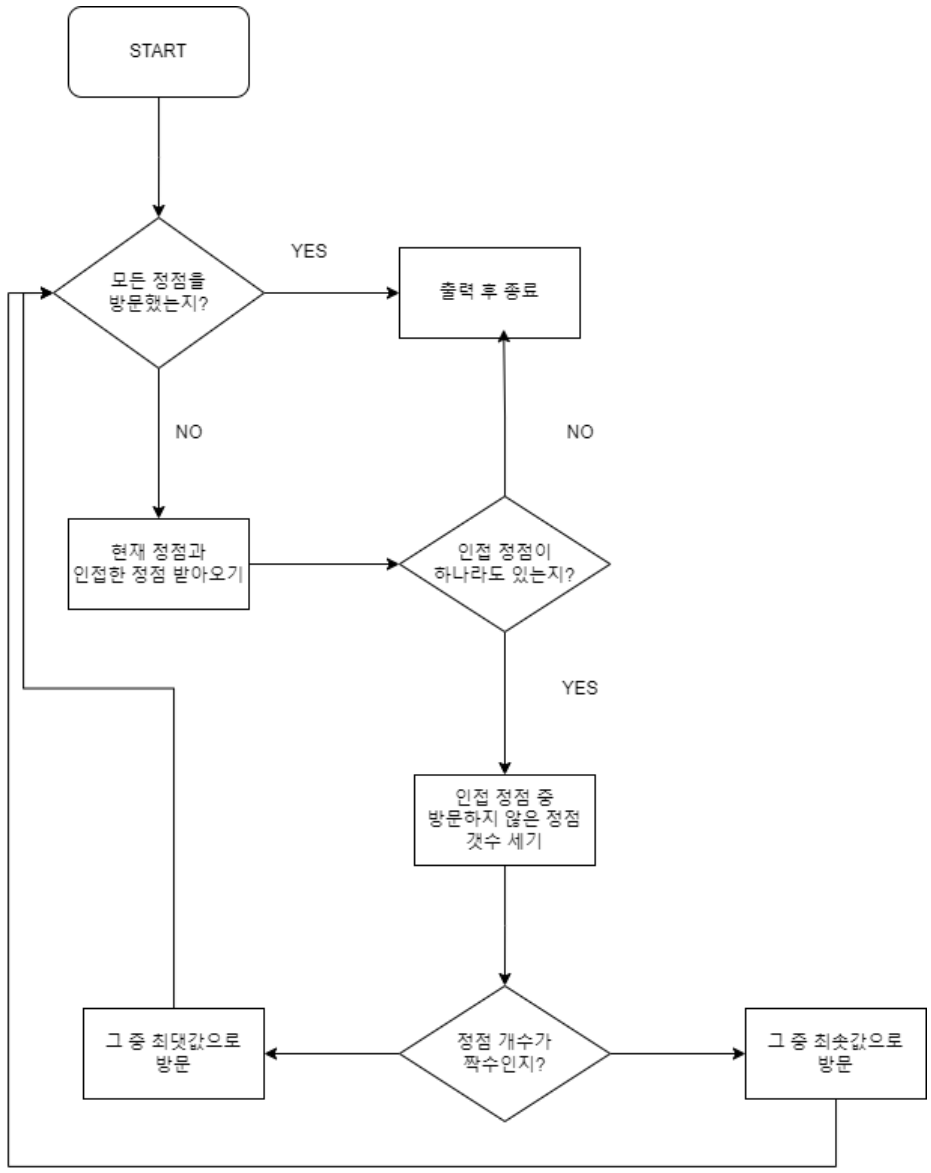
[BELLMANFORD]



[FLOYD]


START
이차원 배열 A 선언
i를 시작점, j를 도착점
모든 <i, j>의 weight를
A에 저장
k = 1부터 반복하며
A[i][j]와 A[i][k]+A[k][j]를
비교 후 작은 값을
A[i][j[에 저장
자기 자신인
A[i][i]에 음수가
있는지?
YES
음수사이클 검출
NO
음수사이클 미검출

[KwangWoon]



- Algorithm

relation: 인접 Node 를 받아오는 Map(vertex 번호 순으로 오름차순 정렬)

visited[Vertex]: 해당 Vertex 방문 여부를 나타내는 bool 형 배열

prev[현재 Vertex]: 현재 Vertex 의 최단 경로 update 를 하게 될 때, 최단 경로를 위해 거친 Vertex 정보를 저장하는 배열

dist[도착점]: 시작점부터 도착점까지의 거리를 저장한 배열

Algorithm 설명 간에 해당 변수명들을 섞어서 사용하겠습니다. 감사합니다.

[DFS]

DFS 는 깊이 우선 탐색이라는 이름을 가진 것처럼, BFS 처럼 넓게 퍼지는 형태가 아닌 깊숙히 들어가서 막다른 길이 나올 때까지 들어가다가, 막힌 길이 나오면 다른 경로가 나올 때까지 왔던 길을 다시 가면서 진행하는 방식입니다. **"막다른 길에 다다르면 직전에 왔던 경로로 돌아간다"** 라는 특성을 살리기 위해서 Stack 을 사용합니다.

프로젝트 기준대로 여러 개의 경로가 있을 때에는 Vertex 번호가 작은 것을 우선으로 방문하면 됩니다. 이러한 점은 map 으로 인접 노드의 정보를 받으면 자동으로 정렬이 되기 때문에, 정보를 가진 map(제 코드에서는 relation 으로 통일하겠습니다.)을 처음부터 읽게 됩니다.

그렇게 vertex 번호가 작은 순부터 읽으며 visit 했다는 정보가 없는 vertex 를 만나게 되면, 현재 current, 그 후 해당 vertex(귀환을 해야 하므로 current 를 먼저 Push 합니다.)를 Stack 에 Push 하고 visited 정보를 true 로 합니다. 이러한 과정을 Stack 이 empty 일 때까지 반복하면 Stack 을 이용한 DFS 알고리즘이 완성됩니다.

[BFS]

BFS 는 DFS 와는 다르게 너비우선 탐색입니다. 현재 Vertex 에서 갈 수 있는 모든 Vertex 를 Queue 에 Push 합니다(프로젝트 조건 상 가장 작은 번호의 Vertex 부터 넣습니다.). 그 후 인접 Vertex 를 넣을 때 가장 처음 Push 한 Vertex 로 이동해서 인접 Vertex 를 Push 하는 것을 반복합니다.

가장 처음에 들어간 Vertex 의 정보를 가져와서 이동을 해야 하기 때문에, Queue 가 쓰인 것입니다.

Queue 가 Empty 일 때까지 반복하게 되는데, 인접 Vertex 를 구한 후, 크기 순서대로 읽으며 해당 Vertex 의 visited 가 false 이면 Queue 에 Push 를 하고 마찬가지로 visited 정보를 true 로 합니다.

이렇게 Queue 를 사용해서 넓게 퍼지는 듯한 형태로 그래프를 탐색하는 BFS 의 알고리즘은 위와 같습니다.

[Djikstra]

Dijkstra 알고리즘은 현재 자신이 위치한 Vertex 에서 가장 작은 Weight 를 가진 Vertex 로 이동하며 최단 경로를 찾는 그래프 탐색 방법입니다. Greedy 알고리즘이라고 할 수 있는데, 왜냐하면 다른 최단 경로 탐색 방법처럼 전체적으로 보는 것이 아닌, 현재 자기 상태에서만 가장 작은 Weight 를 가진 edge 를 타고 가기 때문입니다. 마치 숲을 보지 않고 나무만 보는 것과 같습니다.

전반적인 흐름을 설명하자면, 우선 shortest 배열과 dist 배열이 있습니다. shortest 배열은 해당 Vertex 까지의 거리를 최단 거리로 결정했는지 여부를 알 수 있게 저장합니다.

입력 받은 시작점인 vertex 로부터 모든 정점까지의 거리를 dist 에 저장합니다. 그 후 0 부터 N-1 까지의 반복을 시작합니다. 처음에는 dist 배열을 돌면서 shortest 로 확정되지 않은 동시에 가장

거리가 짧은 Vertex 를 u 로 정합니다. u 가 정해지면 shortest[u] = true 로 하고 v 와 인접한 w 에 대해 반복을 합니다. shortest 로 확정되지 않은 Vertex w 에 대해서, 기존에 저장된 w 까지의 거리인 dist[w]보다 dist[u]+length[u][w]가 더 짧다면 u 를 거쳐서 w 로 가는 것이 더 짧다는 것이기 때문에, dist[w]를 dist[u]+length[u][w]로 업데이트를 하고, prev[w]는 u 로 저장합니다.

이러한 과정을 반복하면, Dijkstra 알고리즘에 따른 최종 결과를 dist 와 prev 배열에 저장하게 됩니다. 여기서 만약 Dijkstra 알고리즘에는 음수 가중치를 다룰 수 없기 때문에, Dijkstra 함수 초기에 시작하자마자 모든 vertex 간의 edge 를 가져와서 하나라도 음수가 있으면 false 를 반환하고 함수가 종료되도록 하였습니다.

[Kruskal]

Kruskal 알고리즘은 생각보다 굉장히 직관적인 알고리즘입니다. 처음에 시작하면 그래프 상에 존재하는 모든 Edge 를 E 라는 vector 에 저장합니다.

E 는 pair<int, pair<int, int>> 형 vector 로 선언하였는데, 사실 이 부분에 있어서 고민이 많았습니다. 굳이 시작점/도착점/가중치 3 개를 저장해야 하는 것인지? 에 대한 고민도 많았는데, Edge 에 관한 정보다 보니 무조건 3 개를 한 번에 저장하는 것이 맞다고 생각했습니다. 그러나 pair 는 한 쌍만 저장하는 것이기 때문에, tuple 이라는 STL 을 사용할까 했지만, 스켈레톤 코드에 추가된 헤더파일 이외에는 추가하는 것은 리스크가 있다고 생각하여 고민한 결과 이중 pair 선언이 가능하다는 것을 알게 되었고, 그 결과 이중 pair 을 사용하게 되었습니다.

vector<Weight, <Start Vertex, End Vertex>> E 라고 생각하시면 될 것 같습니다.

그렇게 모든 Edge 들을 E 에 받아오면, Quick Sort 를 통해 Weight 를 기준으로 오름차순으로 정렬합니다. 그 후 T 라는 <int, int>형 Map 을 추가로 선언합니다. T 는 Minimum Spanning Tree 내에 존재하는 Edge 들의 집합입니다.

T 내의 edge 가 n-1 보다 작거나 E 가 empty 일 때까지 다음과 같은 작업을 반복합니다.

- E 에서 가장 앞에 있는(가장 가중치가 작은) Edge 를 불러옵니다.

- v 는 해당 Edge 의 시작점/w 는 해당 Edge 의 도착점/cost 는 해당 Edge 의 Weight 로 저장합니다.

- 만약 v 와 w 가 Union 되어 있지 않다면, 두 Vertex 를 Union 하고 해당 Edge 를 T 에 insert 합니다.

Union 여부는 prev 배열을 통해 부모가 같다면 같은 Set 이라고 생각하고, 같지 않다면 서로 다른 Set 이라고 생각하게 하였습니다.

(Simple Union 과 Simple Find 알고리즘을 사용했습니다.) 하지만 최종적으로 T 에 n-1 보다 적은 edge 가 포함되어 있다면, MST 를 찾는 것을 실패한 것이기 때문에 fail 처리를 하도록 하였습니다.

[Bellman-Ford]

Bellman-Ford 는 시작점으로부터 각각의 Vertex 에 대한 거리를 **"Edge 를 늘려가며"** Update 해서 최종으로 n-1 번째 Edge 에 대해서 Update 를 완료하면 시작점으로부터 각 Vertex 에 대한 최단 거리를 얻을 수 있게 됩니다.

위와 비슷하게 prev 와 dist 배열을 시작점을 기준으로 모든 Vertex 에 대해서 초기 세팅을 합니다.

그 후 k = 2 부터 n-1 까지의 반복을 시작합니다. 이는 Edge 개수에 해당합니다.

시작점을 s, 도착점을 v 라고 할 때, w 는 v 에 도착하기 직전 Vertex 입니다. v 에 대해서 적어도 하나의 Incoming Edge 가 있다고 한다면, v 에 대한 거리가 바뀔 여지가 있는 것이기 때문에, 그 경우 모든 w 에 대해서 반복을 시작합니다.

만약 dist[v]에 대해서 이전 dist[v]보다 w 를 거친 dist[w]+length[w][v]가 더 작다면, dist[v]에 해당 값을 update 해주고, prev[v]에는 w 를 저장하면 됩니다.

이 반복을 끝까지 하고 나서 한 번 더 해당 로직을 수행합니다. 이 경우에 또 dist[v]가 업데이트가 되려고 한다면, 이는 음수 사이클로 인한 것이기 때문에, return false 를 해줍니다.

[Floyd]

Floyd 는 Bellman-Ford 와는 다르게 Edge 가 아닌 Vertex 에 대한 그래프 탐색 방법입니다.

i 부터 j 까지의 경로를 정할 때, 중간에 k 라는 Vertex 를 거쳐가는 것이 더 빠르지 않을까? 에 대한 알고리즘이라고 할 수 있습니다. 그래서 초기에 i 부터 j 까지의 경로를 초기화하고, 모든 k 에 대해서 k 를 거쳐가는 것이 더 짧다면, A[i][j](i 부터 j 까지의 거리)에 A[i][k] + A[k][j]의 값을 업데이트 해주게 됩니다.

이러한 과정을 1 부터 n 까지 반복하면, 모든 최단 경로를 구하게 됩니다.

마지막으로 자기 자신에 대해서 A[i][i]가 음수인 경우, 음수 사이클이 생겼다고 판정하고 false 를 반환합니다. 초기에 A[i][i]가 0 인데, 과정을 거치며 음수가 되었다는 것은 음수 사이클이 생겼다는 의미이기 때문입니다.

[KwangWoon]

KwangWoon 알고리즘은 사실 이번 프로젝트 기간 내에 완벽하게 구현을 하지 못했습니다.

그래서 최대한 같은 결과가 나올 수 있도록 구현을 해보았습니다. 인접 정점 중 방문하지 않은 곳의 개수를 체크하고, 그 개수가 홀수면 그 중 최대값을, 짝수면 그 중 최소값을 방문하게 했습니다.

출력에 있어서는 Queue 를 사용해서 가장 먼저 방문한, 가장 먼저 Push 된 곳부터 차례대로 출력하게 했습니다.

하지만 이것이 정석 KwangWoon 알고리즘은 아니기 때문에, 제가 구상만 해본 세그먼트 트리를 사용한 수도코드도 적어보겠습니다.

-init 으로 초기화

-1 번 부터 시작하니 맨처음에 모든 광운그래프 돌면서 1 을 없애기 위해 update 실행

-current 를 상황에 맞는 곳으로 이동 후 while 문 시작

while (현재 Node 에 대한 segement Tree 의 root 가 0 이 아닐때까지)

{

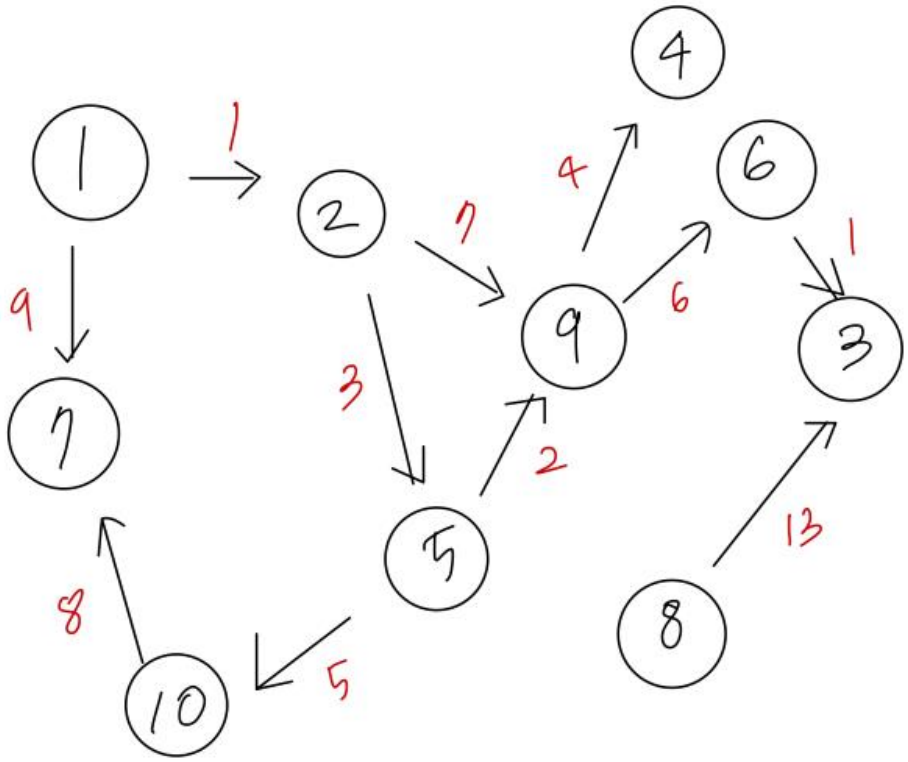
현재 노드에 대해서, 모든 kwgraph 노드에 대한 현재 노드 값 update

상황에 맞는 곳으로 이동

}

사실 직접 구현을 성공하지 못했기 때문에, 맞는지는 잘 모르겠지만, 다음에 기회가 된다면 제대로 구현해보고 싶습니다.

- Result Screen



제가 직접 만든 10개의 노드로 이루어진 음수 가중치가 없는 그래프입니다. 이 그래프로 Bellman-Ford와 Floyd를 제외한 그래프 탐색 방법에 대해서 설명해보겠습니다.

[LOAD / PRINT]

```
======== LOAD ========
Success
====================

======== PRINT ========
[1]-> (2,1)-> (7,9)
[2]-> (5,3)-> (9,7)
[3]
[4]
[5]-> (9,2)-> (10,5)
[6]-> (3,1)
[7]
[8]-> (3,13)
[9]-> (4,4)-> (6,6)
[10]-> (7,8)
====================
======== LOAD ========
Success
====================

======== PRINT ========
    [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1] 0   1   0   0   0   0   9   0   0   0
[2] 0   0   0   0   3   0   0   0   7   0
[3] 0   0   0   0   0   0   0   0   0   0
[4] 0   0   0   0   0   0   0   0   0   0
[5] 0   0   0   0   0   0   0   0   2   5
[6] 0   0   1   0   0   0   0   0   0   0
[7] 0   0   0   0   0   0   0   0   0   0
[8] 0   0   13  0   0   0   0   0   0   0
[9] 0   0   0   4   0   6   0   0   0   0
[10] 0   0   0   0   0   0   8   0   0   0
====================
```

각 형식에 맞는 그래프가 잘 LOAD가 되고 출력도 되는 것을 확인할 수 있습니다.

[DFS]

```
======== LOAD ========
Success
====================

======== PRINT ========
[1]-> (2,1)-> (7,9)
[2]-> (5,3)-> (9,7)
[3]
[4]
[5]-> (9,2)-> (10,5)
[6]-> (3,1)
[7]
[8]-> (3,13)
[9]-> (4,4)-> (6,6)
[10]-> (7,8)
====================

======== DFS ========
Directed Graph DFS result
startvertex: 1
1 -> 2 -> 5 -> 9 -> 4 -> 6 -> 3 -> 10 -> 7
====================

======== DFS ========
Undirected Graph DFS result
startvertex: 1
1 -> 2 -> 5 -> 9 -> 4 -> 6 -> 3 -> 8 -> 10 -> 7
====================
```

1번에서 출발해서 directed의 경우,

1 -> 2 -> 5 -> 9 -> 4 -> X / Pop 4 / 9 -> 6 -> 3 -> X / Pop 3 / Pop 6 / Pop 9 / Pop 5 / 5 -> 10 -> 7

따라서 1 -> 2 -> 5 -> 9 -> 4 -> 6 -> 3 -> 10 -> 7 이라는 결과가 나오게 됩니다.

undirected의 경우,

1 -> 2 -> 5 -> 9 -> 4 -> X / Pop 4 / 9 -> 6 -> 3 -> 8 -> X / Pop 8 / Pop 3 / Pop 6 / Pop 9 / Pop 5 / 5 -> 10 -> 7

따라서 1 -> 2 -> 5 -> 9 -> 4 -> 6 -> 3 -> 8 -> 10 -> 7 이라는 결과가 나오게 됩니다.

[BFS]

```
======== LOAD ========
Success
====================

======== PRINT ========
[1]-> (2,1)-> (7,9)
[2]-> (5,3)-> (9,7)
[3]
[4]
[5]-> (9,2)-> (10,5)
[6]-> (3,1)
[7]
[8]-> (3,13)
[9]-> (4,4)-> (6,6)
[10]-> (7,8)
====================

======== BFS ========
Directed Graph BFS result
startvertex: 1
1 -> 2 -> 7 -> 5 -> 9 -> 10 -> 4 -> 6 -> 3
====================

======== BFS ========
Undirected Graph BFS result
startvertex: 1
1 -> 2 -> 7 -> 5 -> 9 -> 10 -> 4 -> 6 -> 3 -> 8
====================
```

Directed의 경우,

1(Push 2&7) -> 2(Push 5&9) -> 7 -> 5(Push 10) -> 9(Push 4&6) -> 10 -> 4 -> 6(Push 3) -> 3

Undirected의 경우,

1(Push 2&7) -> 2(Push 5&9) -> 7(Push 10) -> 5 -> 9(Push 4&6) -> 10 -> 4 -> 6(Push 3) -> 3(Push 8) -> 8 이라는 순서가 나오므로, 정확한 값이 출력되는 것을 확인할 수 있습니다.

[Dijkstra]

```
======== Dijkstra ========
Directed Graph Dijkstra result
startvertex: 1
[2] 1 -> 2 (1)
[3] 1 -> 2 -> 5 -> 9 -> 6 -> 3 (13)
[4] 1 -> 2 -> 5 -> 9 -> 4 (10)
[5] 1 -> 2 -> 5 (4)
[6] 1 -> 2 -> 5 -> 9 -> 6 (12)
[7] 1 -> 7 (9)
[8] X
[9] 1 -> 2 -> 5 -> 9 (6)
[10] 1 -> 2 -> 5 -> 10 (9)
====================

======== Dijkstra ========
Undirected Graph Dijkstra result
startvertex: 1
[2] 1 -> 2 (1)
[3] 1 -> 2 -> 5 -> 9 -> 6 -> 3 (13)
[4] 1 -> 2 -> 5 -> 9 -> 4 (10)
[5] 1 -> 2 -> 5 (4)
[6] 1 -> 2 -> 5 -> 9 -> 6 (12)
[7] 1 -> 7 (9)
[8] 1 -> 2 -> 5 -> 9 -> 6 -> 3 -> 8 (26)
[9] 1 -> 2 -> 5 -> 9 (6)
[10] 1 -> 2 -> 5 -> 10 (9)
====================
```

Directed Dijkstra의 결과는 다음과 같습니다.

now	1	2	3	4	5	6	7	8	9	10
1	0	1	-	-	-	-	9	-	-	-
2	0	1	-	-	4	-	9	-	8	-
5	0	1	-	-	4	-	9	-	6	9
9	0	1	-	10	4	12	9	-	6	9
7	0	1	-	10	4	12	9	-	6	9
10	0	1	-	10	4	12	9	-	6	9
4	0	1	-	10	4	12	9	-	6	9
6	0	1	13	10	4	12	9	-	6	9
3	0	1	13	10	4	12	9	-	6	9
8	0	1	13	10	4	12	9	-	6	9

Undirected Dijkstra의 결과는 다음과 같습니다.

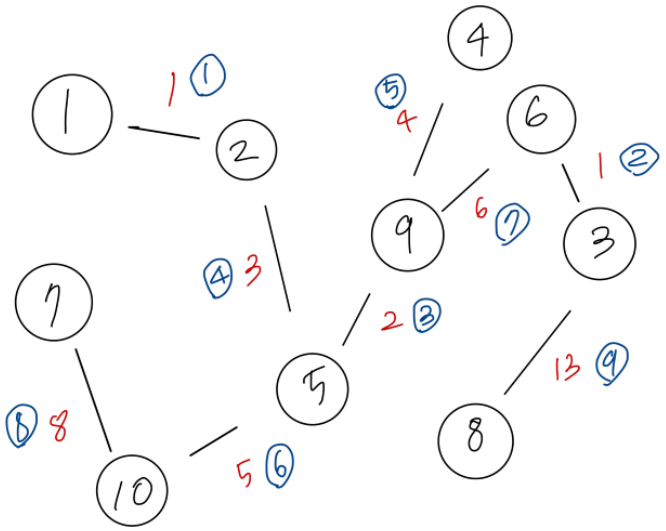
now	1	2	3	4	5	6	7	8	9	10
1	0	1	-	-	-	-	9	-	-	-
2	0	1	-	-	4	-	9	-	8	-
5	0	1	-	-	4	-	9	-	6	9
9	0	1	-	10	4	12	9	-	6	9
7	0	1	-	10	4	12	9	-	6	9
10	0	1	-	10	4	12	9	-	6	9

4	**0**	**1**	**-**	**10**	**4**	**12**	**9**	**-**	**6**	**9**
6	**0**	**1**	**13**	**10**	**4**	**12**	**9**	**-**	**6**	**9**
3	**0**	**1**	**13**	**10**	**4**	**12**	**9**	**26**	**6**	**9**
8	**0**	**1**	**13**	**10**	**4**	**12**	**9**	**26**	**6**	**9**

이로써 정확한 결과값이 출력되는 것을 확인할 수 있습니다.

[Kruskal]

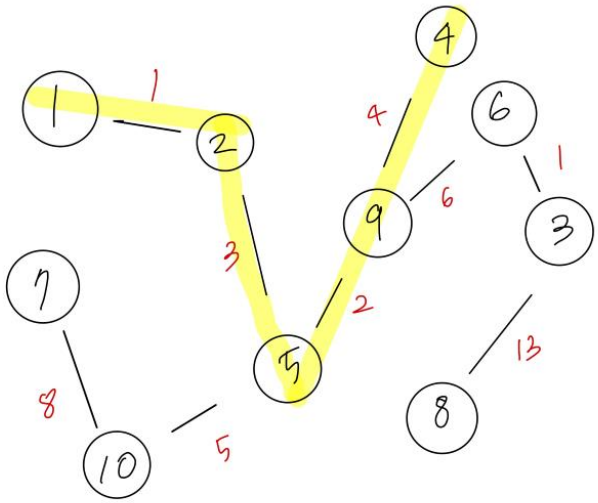
```
======== Kruskal ========
[1]       2(1)
[2]       1(1)5(3)
[3]       6(1)8(13)
[4]       9(4)
[5]       2(3)9(2)10(5)
[6]       3(1)9(6)
[7]       10(8)
[8]       3(13)
[9]       4(4)5(2)6(6)
[10]       5(5)7(8)
cost: 43
====================
```



두 결과가 동일하고 Total Cost도 알맞게 나오는 것을 보아, 가장 가중치가 적은 edge부터 Cycle을 형성하지 않고 MST를 잘 구해낸 것을 확인할 수 있습니다.

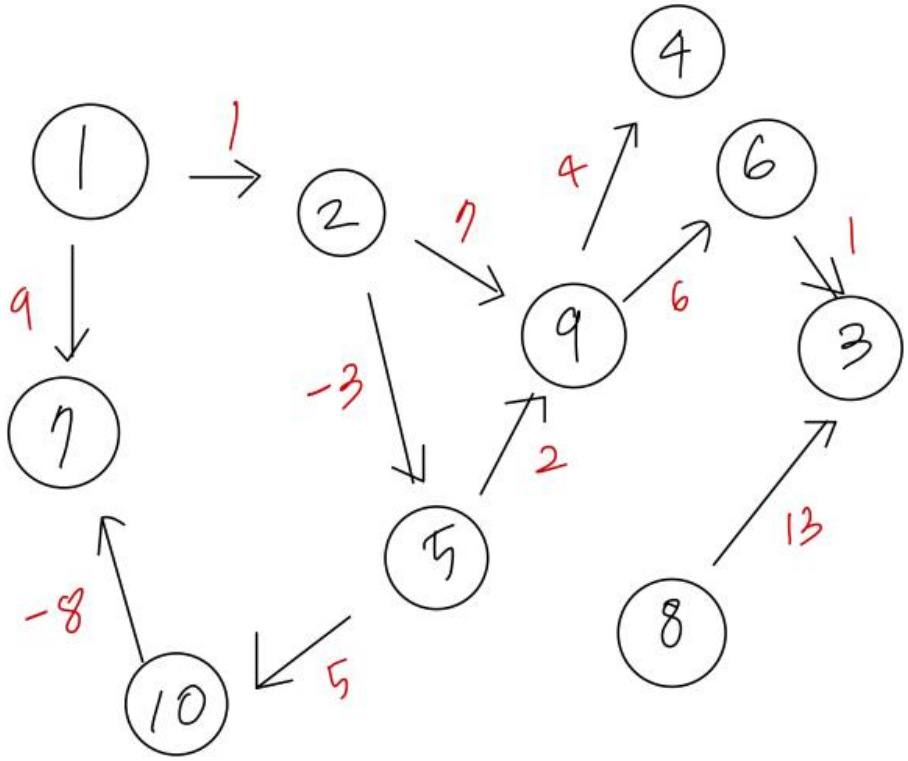
[KwangWoon]

```
======== LOAD ========
Success
====================

======== PRINT ========
[1]-> (2,1)-> (7,9)
[2]-> (5,3)-> (9,7)
[3]
[4]
[5]-> (9,2)-> (10,5)
[6]-> (3,1)
[7]
[8]-> (3,13)
[9]-> (4,4)-> (6,6)
[10]-> (7,8)
====================

======== KWANGWOON ========
startvertex: 1
1 -> 2 -> 5 -> 9 -> 4
====================
```



결과는 위와 같습니다. 1번부터 시작해서 막 다다른 길에 갈 때까지 탐색을 하였고, 방문하지 않

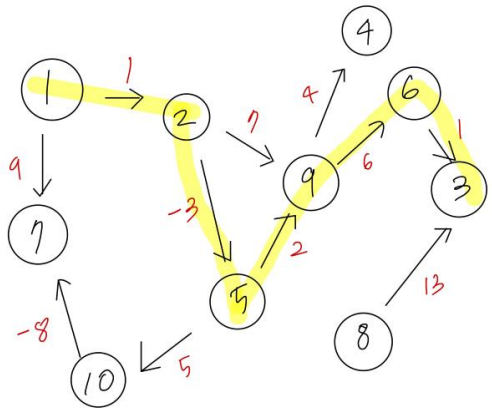
은 인접 노드가 짝수면 가장 작은 번호의 노드를, 홀수라면 가장 큰 번호의 노드를 방문하게 했습니다.



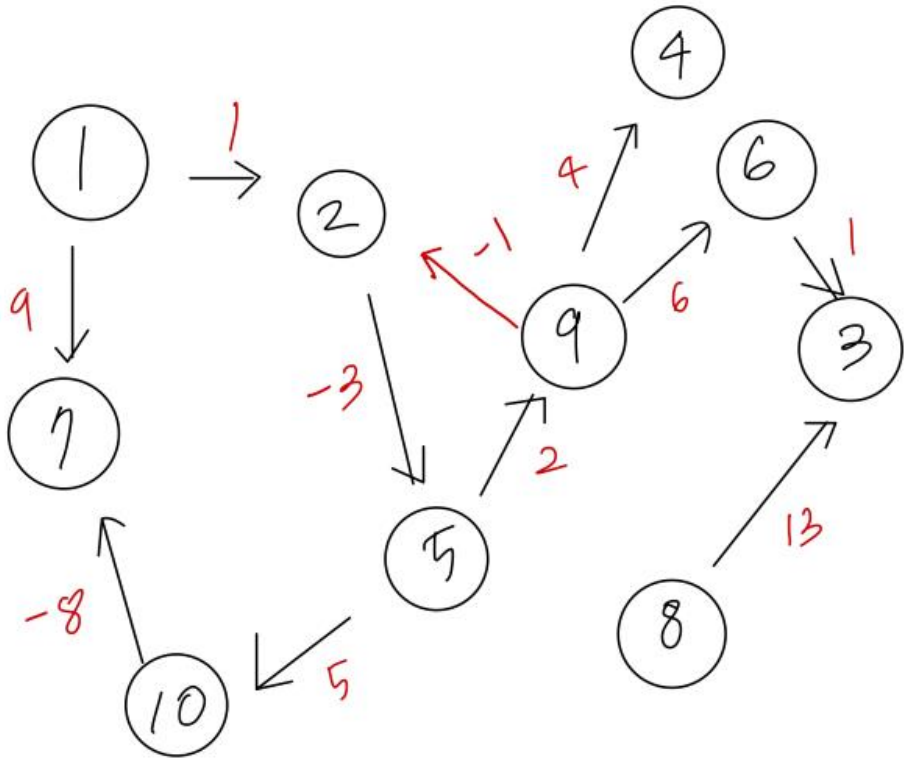
위는 원래 그래프와 매우 흡사하나, 음수 가중치를 실험하기 위해서 만든 그래프입니다.

[Bellman-Ford]

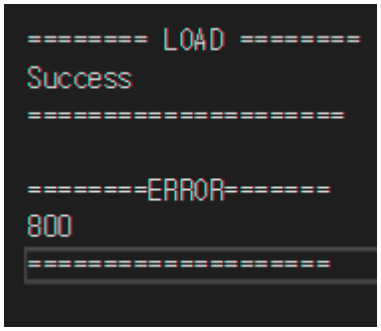
```
1   ======== LOAD ========
2   Success
3   ====================
4
5   ======== Bellman-Ford ========
6   Directed Graph Bellman-Ford result
7   1 -> 2 -> 5 -> 9 -> 6 -> 3
8   cost: 7
9   ====================
10
11  ========ERROR=======
12  800
13  ====================
14
15
```



처음에 나오는 결과는 Directed, 아래에 나오는 결과는 Undirected입니다. Directed인 경우는 1에서 3을 가기 위해, 가능한 최소 Cost를 얻기 위하여 음수 가중치가 있는 2 -> 5를 거쳐서 3에 도달하는 알맞은 결과가 나왔습니다. 하지만 Undirected의 경우에는 보다 적은 Total cost를 위해 2 – 5를 서로 반복해서 돌게 되는 음수사이클이 발생하여 False 결과가 나온 것입니다.

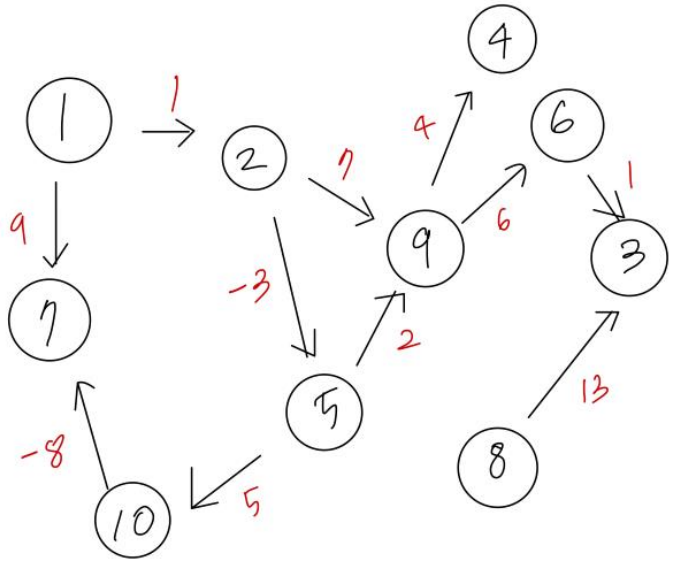


Directed에서도 음수 사이클을 잘 판독하는지 확인하기 위해, 해당 그래프를 넣고 Directed Bellman-Ford를 실행해보았습니다.



Directed의 경우에도 음수 사이클을 잘 판독하는 것을 확인할 수 있습니다.

[Floyd]

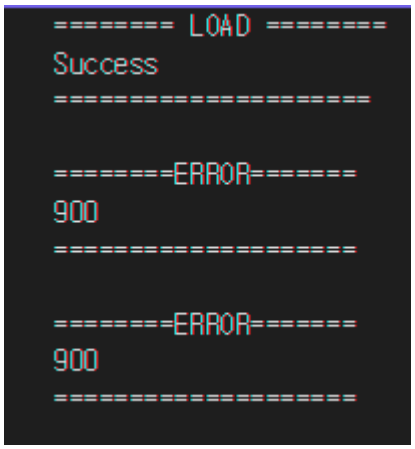


Bellman-Ford와 같이 해당 그래프를 넣어보겠습니다.

```
======== LOAD ========
Success
====================

======== FLOYD ========
Directed Graph FLOYD result
    [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1]  0   1   7   4  -2   6  -5   X   0   3
[2]  X   0   6   3  -3   5  -6   X  -1   2
[3]  X   X   0   X   X   X   X   X   X   X
[4]  X   X   X   0   X   X   X   X   X   X
[5]  X   X   9   6   0   8  -3   X   2   5
[6]  X   X   1   X   X   0   X   X   X   X
[7]  X   X   X   X   X   X   0   X   X   X
[8]  X   X  13   X   X   X   X   0   X   X
[9]  X   X   7   4   X   6   X   X   0   X
[10] X   X   X   X   X   X  -8   X   X   0
====================

========ERROR=======
900
====================
```

처음은 Directed, 다음은 Undirected입니다. Undirected에 경우 Bellman-Ford와 같은 맥락으로 음수 사이클이 나옵니다.

Floyd는 값이 잘 나오는 것을 확인할 수 있고, 특히 Bellman-Ford에서 테스트한 1->3 케이스를 확인해보면, 동일하게 7이라는 값이 출력됩니다.



Bellman-Ford의 두번째 테스트 케이스를 동일하게 넣으면, Directed 상황에서 음수 사이클이 감지되어 모두 False 결과가 나옵니다.

- Consideration

이번 과제에서는 두 가지 어려움이 있었습니다. 첫번째는, 다양한 그래프 탐색 방법에서 출발지부터 도착지까지 도달할 수 있었던 경로를 출력해야 하는 것이었습니다. 이 경우에는 고민 끝에 prev라는 배열을 만들어 각각의 직전 노드를 저장하였고, 그 prev 배열을 연쇄적으로 읽으며 출발지까지의 수를 확인할 수 있었습니다. 그런데 prev 배열의 특성 상 도착지부터 먼저 접하기 때문에, 문제에서 요구하는 것과는 역순으로 출력을 하게 됐습니다. 그래서 고심 끝에, prev 배열을 읽을 때마다, Vector에 Push back한 후, 출력할 때 Pop Back을 하는 방식으로 출력하게 되었습니다. 이렇게 하면 마치 Stack을 사용한 것처럼 역순으로 출력을 할 수 있습니다.

또한 이번 과제에서는 아쉽게도 문제에서 요구하는 바와 같이 KwangWoon 알고리즘을 세그먼트 트리를 사용해서는 구현하지 못했습니다. 그래도 포기하는 것보다는 낫다고 생각하여, 제가 생각한 간단한 방식으로 구현하였습니다. 조금의 시간이 더 있더라면, 세그먼트 트리를 사용해서 구현할 수 있지 않을까 하는 아쉬움이 있기 때문에 방학 때 꼭 다시 한번 세그먼트 트리를 사용해서 구현을 하도록 하겠습니다. 감사합니다.