

데이터 구조

3차

Project

학번: 2024402041

실습 분반: 수

이름: 심수정

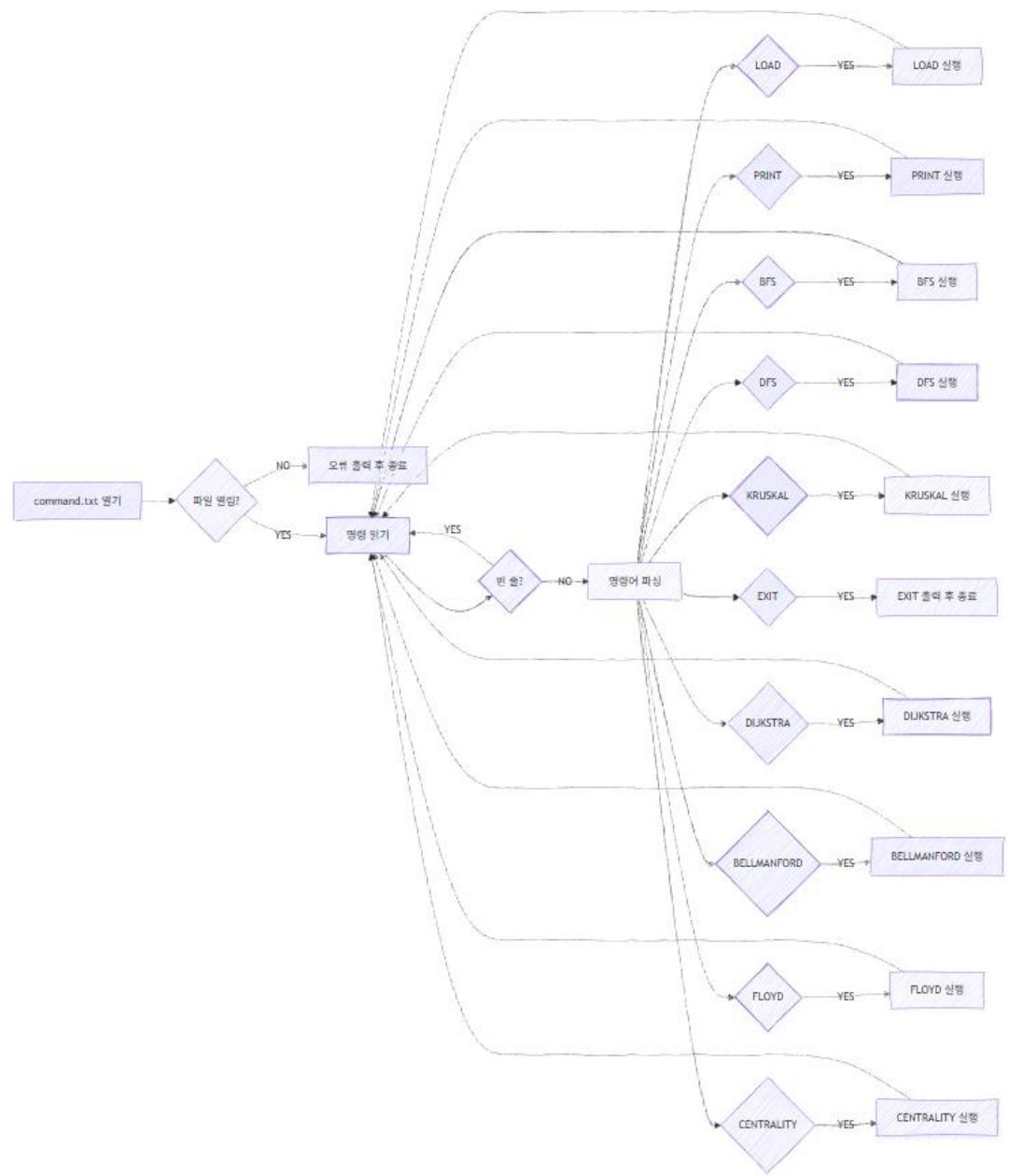
## 1. Introduction

본 프로젝트는 그래프 자료구조를 구축하고 이에 대한 핵심 알고리즘을 수행하는 프로그램을 구현한다. 텍스트 파일로 주어지는 그래프 정보를 파싱하여 그래프를 생성하고, 명령어 파일을 통해 탐색(BFS, DFS) 및 최단 경로 (Dijkstra, Bellman-Ford, Floyd), MST(Kruskal), 근접 중심성(Closeness Centrality) 분석을 수행한다. 이를 통해 방향성(Directed) 및 가중치 (Weighted) 유무에 따른 알고리즘의 동작 차이를 이해하고, 이를 효율적인 자료구조로 구현하는 것을 목표로 한다.

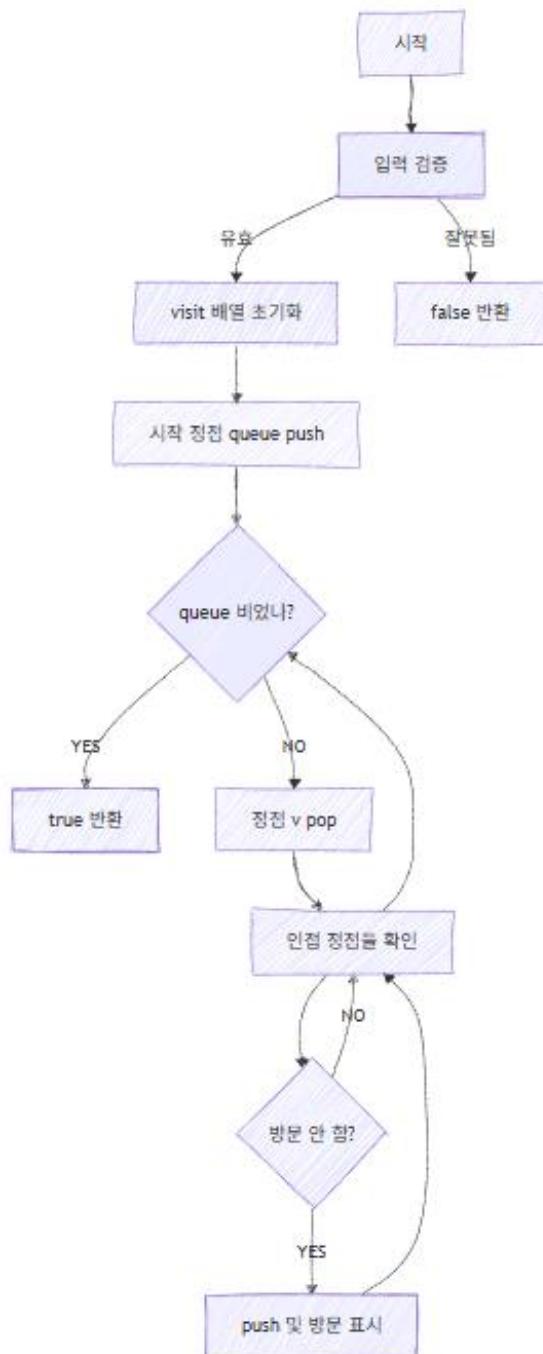
## 2. Flow Chart

각 알고리즘에 대한 설명은 3. Algorithm에서 다루었다.

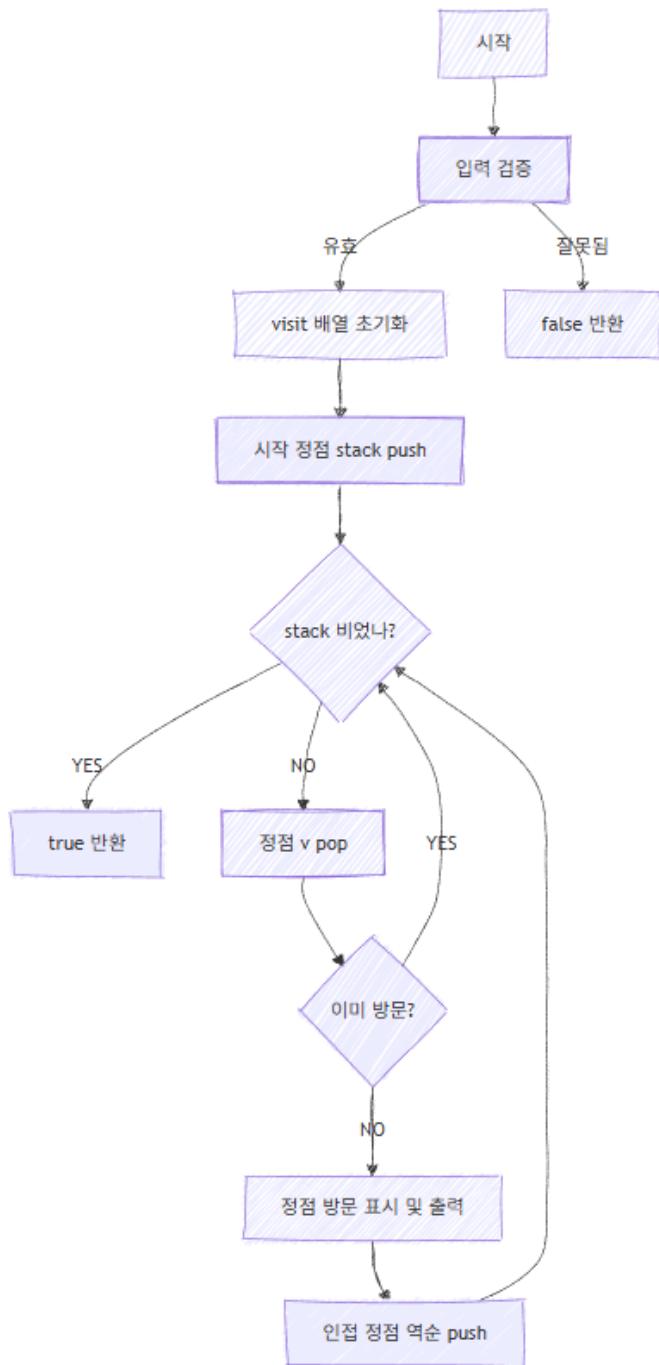
### A. Manager.cpp



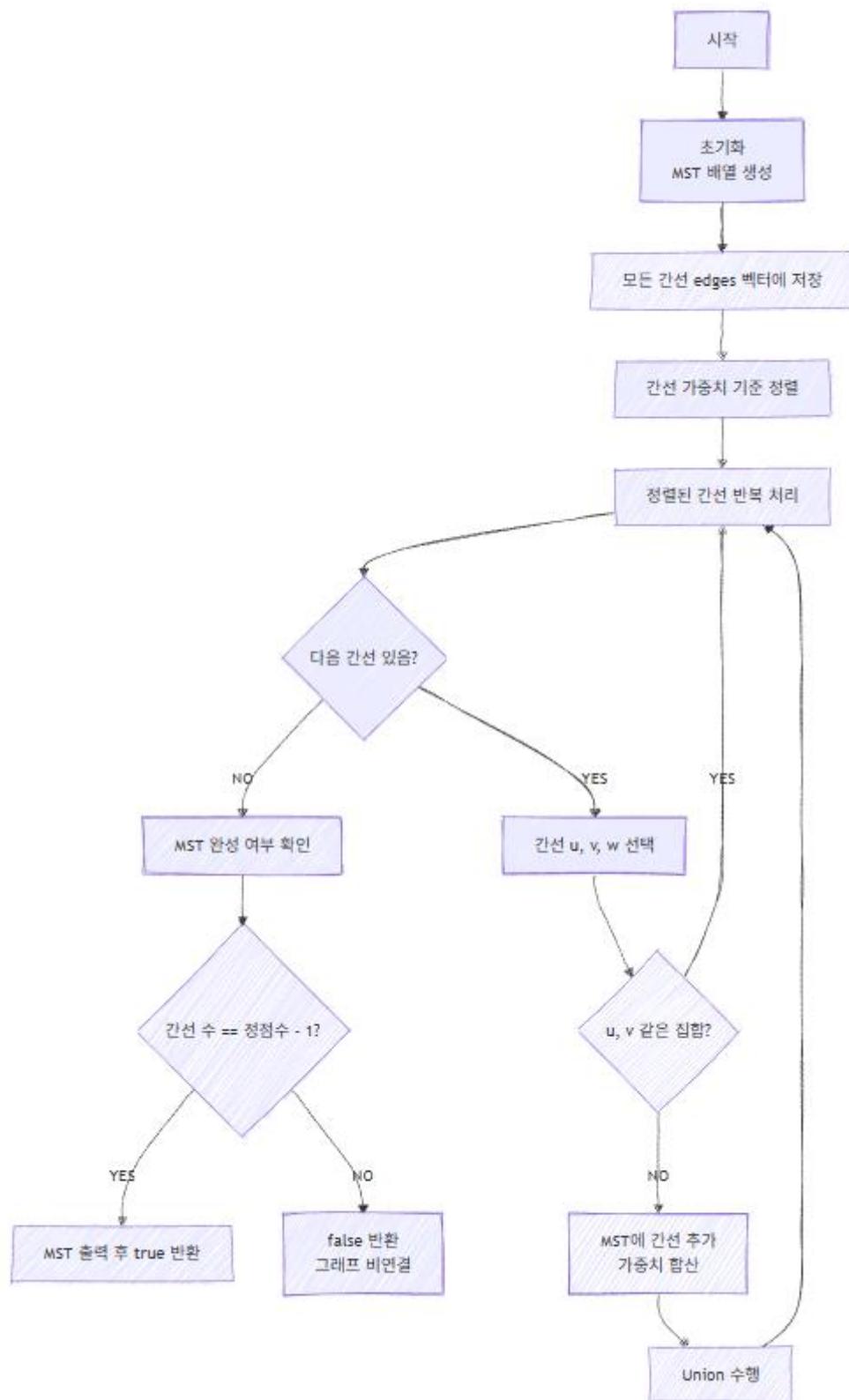
## B. BFS



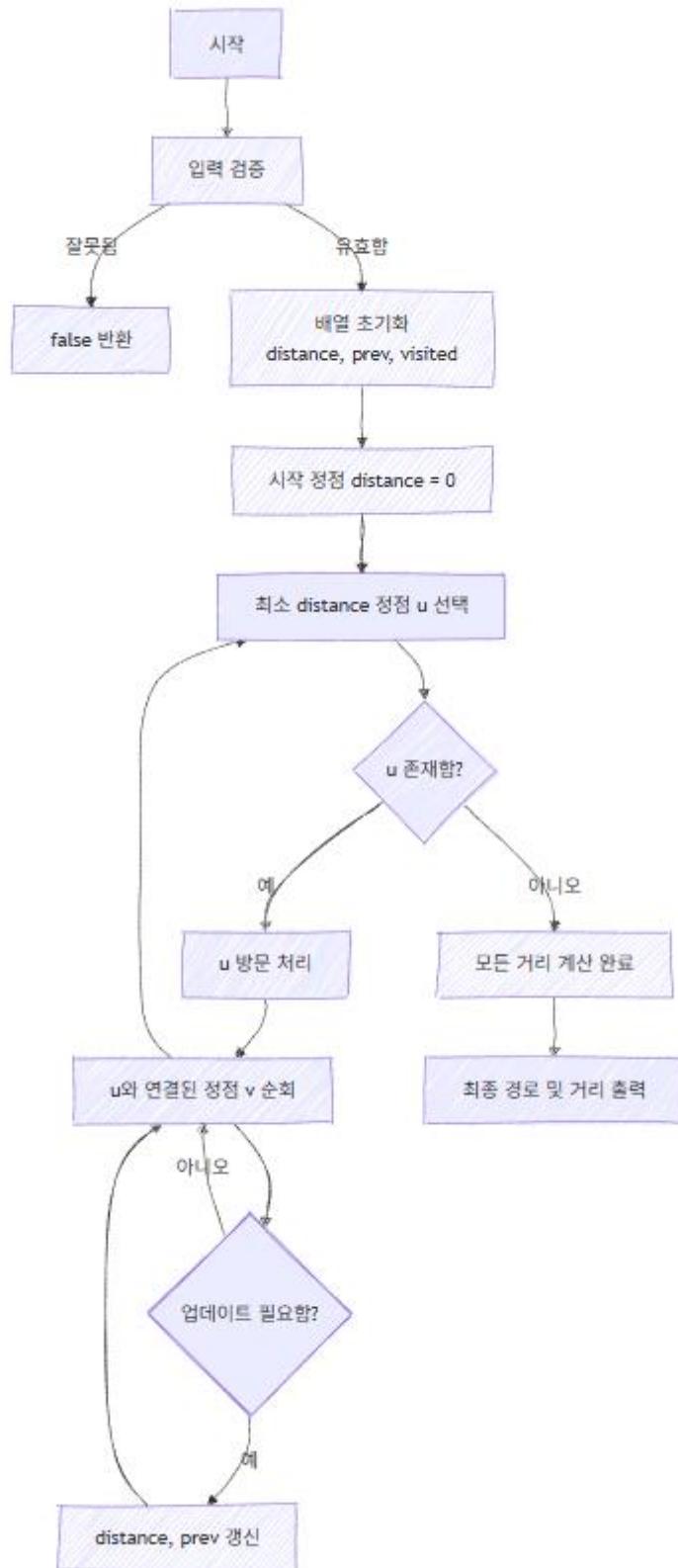
### C. DFS



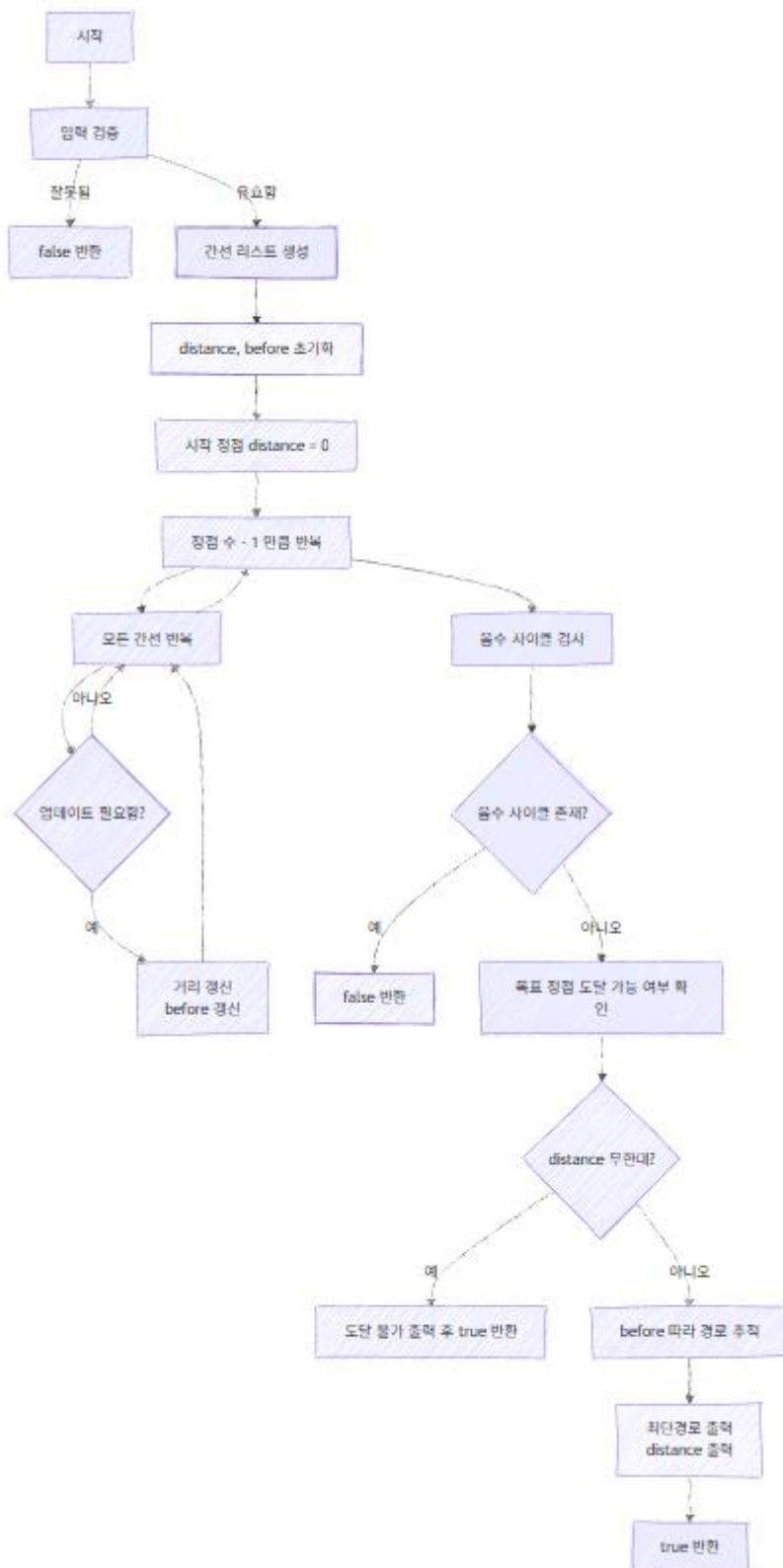
#### D. KRUSKAL



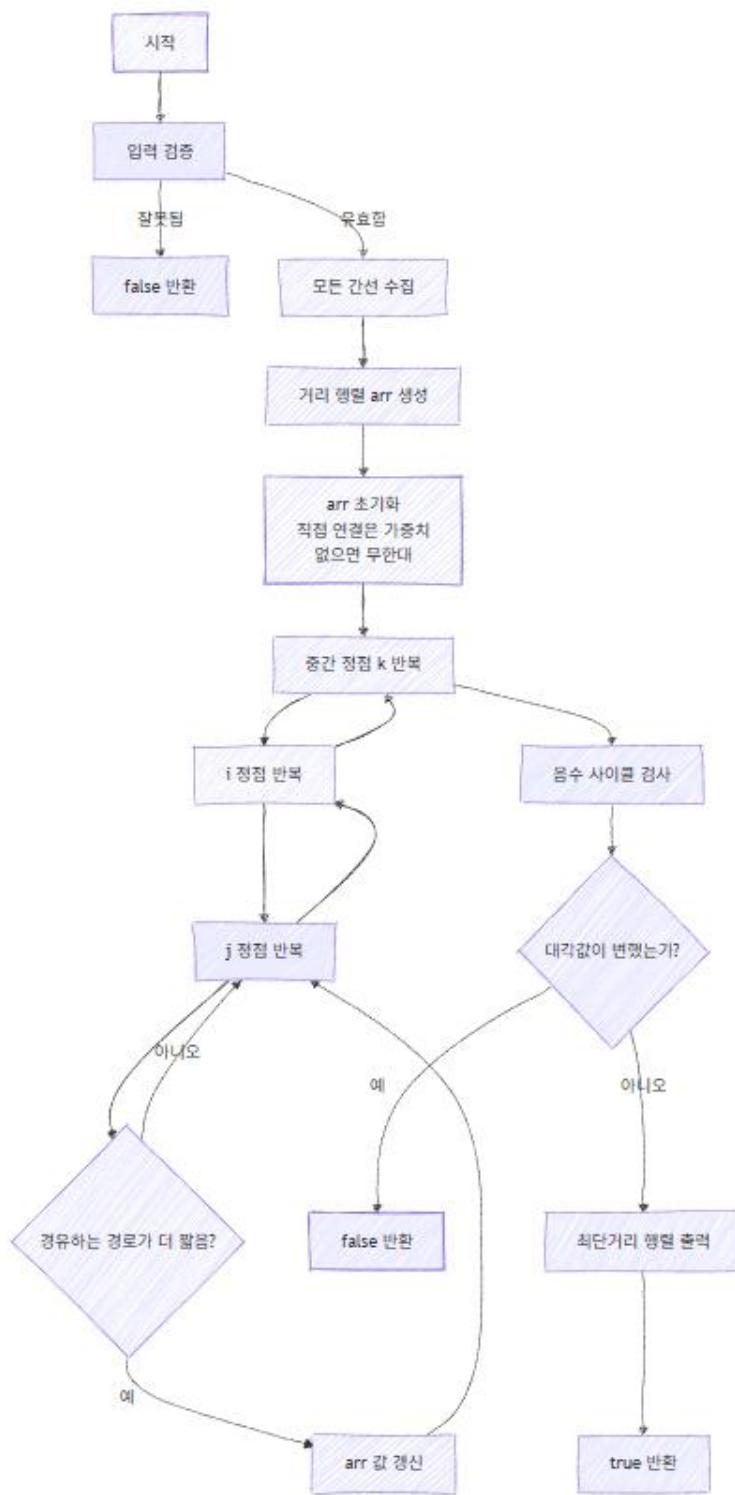
## E. Dijkstra



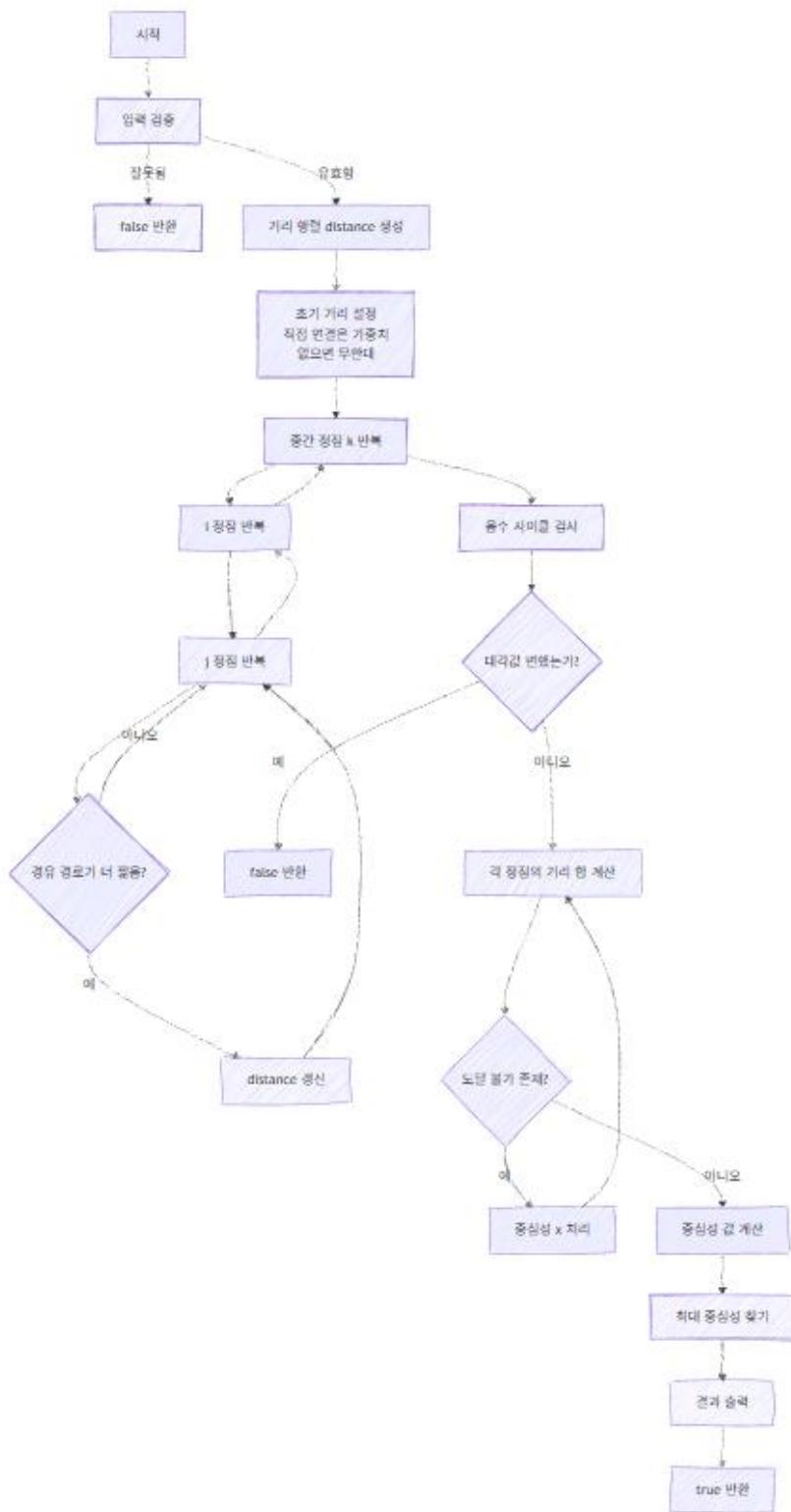
F. Bellman-Ford



G. FLOYD



#### H. 근접 중심성



### 3. Algorithm

#### A. BFS

BFS(Breadth-First Search)는 그래프에서 시작 정점으로부터 가까운 정점부터 차례대로 방문하는 탐색 알고리즘이다.

이 알고리즘은 너비 우선으로 탐색하기 위해 큐(queue) 자료구조를 사용하여, 먼저 발견된 정점을 먼저 처리하는 방식으로 탐색이 진행된다. 본 프로그램의 BFS는 우선 방문 여부를 저장할 배열을 초기화하고, 모든 정점을 아직 방문하지 않은 상태로 설정한다. 그리고 시작 정점을 큐에 삽입하고 해당 정점을 방문 처리한다. 큐가 비지 않는 동안 큐의 맨 앞 정점을 꺼내고 출력하고, 현재 정점과 인접한 모든 정점들을 가져와 아직 방문하지 않은 정점이면 큐에 추가하고 방문 표시한다. 이 과정은 반복적으로 처리해, 큐가 비게 되면 시작 정점에서 도달 가능한 모든 정점을 탐색한 것이다.

## B. DFS

DFS(Depth-First Search)는 그래프에서 한 방향으로 가능한 깊숙한 곳까지 먼저 탐색한 뒤, 더 이상 진행할 수 없으면 이전 정점으로 되돌아와 다른 경로를 탐색하는 방식의 깊이 우선 탐색 알고리즘이다. 이 알고리즘은 깊이 방향으로 탐색하기 위해 스택(stack) 자료구조를 사용한다. 본 프로그램의 DFS는 우선 BFS와 동일하게 방문 여부를 저장할 배열을 초기화하고, 모든 정점을 아직 방문하지 않은 상태로 설정하는 것으로 시작한다. 시작 정점을 스택에 넣어 탐색을 시작한다. 스택이 비어있지 않을 동안, 다음의 과정은 반복한다. 맨 위에 있는 정점을 꺼내 현재 정점으로 정한다. 이때 만약 이미 방문한 정점이라면 건너뛴다. 처음 방문하는 정점이라면 해당 정점을 출력하고 방문 표시를 한다. 현재 정점과 인접한 정점들을 가져와 이미 방문하지 않은 정점들은 역순으로 스택에 넣는다. 반복 과정을 통해 스택이 모두 비게 되면 시작 정점에서 도달 가능한 모든 정점을 탐색한 것이다.

## C. Kruskal

Kruskal 알고리즘은 가중치가 있는 무방향 그래프에서 최소 비용 신장 트리(MST)를 구성하기 위한 대표적인 알고리즘으로, 그래프의 모든 간선 중 가중치가 가장 작은 간선부터 선택해 나가며, 사이클(cycle)이 발생하지 않도록 조합하여 MST를 완성한다. 우선, 그래프의 정점 개수만큼 MST 배열을 생성하고, 모든 정점이 자기 자신을 부모로 갖는 독립 집합 형태로 초기화한다. 이후 그래프에서 모든 간선을 읽어 edges 벡터에 (weight, (start, end)) 형태로 저장한다. Edges 벡터에 저장된 모든 간선을 가중치 기준 오름차순으로 정렬한다. 정렬은 코드에서 직접 구현한 partition, Kruskal\_Sort 함수를 통해 수행된다. 이를 통해 가장 비용이 낮은 간선부터 순서대로 확인할 수 있다. 정렬된 간선을 앞에서부터 하나씩 확인하는 과정을 반복한다. 우선 간선의 양 끝 정점에 대해 Union-Find 연산을 사용해 두 정점이 같은 집합에 속해있는지 확인한다. 만약 같은 집합에 속해있다면 사이클이 발생하므로 무시한다. 두 정점이 다른

집합이라면 그 간선을 MST에 추가하고 Union함수로 두 정점을 같은 집합으로 합친다. 이렇게 추가된 모든 간선의 비용을 더해 총 비용을 계산한다. MST는 정점이 m개 일 때 m-1개의 간선을 가져야 한다. 모아진 간선의 수가 size-1개가 아니라면, 이는 그래프가 하나로 연결되지 않은 비연결 그래프이고, MST를 만들 수 없으므로 false를 반환한다.

#### D. Dijkstra

Dijkstra 알고리즘은 가중치가 모두 양수인 그래프에서 한 정점으로부터 모든 정점까지의 최단 경로를 구하는 알고리즘이다. 본 프로그램의 구현은 배열을 직접 탐색하여 최소 거리 정점을 선택하는 방식으로 구성되었다. 우선, 그래프의 정점 수만큼 distance 배열을 생성하여 시작 정점에는 0, 나머지 정점에는 무한대(int의 최댓값을 사용함)를 설정한다. Prev 배열은 해당 정점까지의 최단 경로에서의 이전 정점을 저장하기 위한 배열이며, 초기값은 모두 -1이다. Visited 배열은 최단 거리가 확정된 정점을 표시하기 위해 사용되며 처음에는 모두 false이다. 매 반복마다 visited==false인 정점 중 distance 값이 가장 작은 정점을 직접 탐색하여 선택한다. 선택된 정점 u는 최단 거리가 확정되었으므로 visited[u]=true로 표시된다. 정점 u의 모든 인접 정점을 확인하여 distance[v]가 distance[u] + weight(u->v)보다 클 때 거리 값을 갱신한다. 새로운 경로가 더 짧으면 distance[v]를 해당 값으로 갱신하고, 경로 추적을 위해 prev[v]=u로 저장한다. 전체 정점 수만큼 반복해 모든 정점의 최단 거리가 확정될 때까지 계속 수행한다. 모두 방문 되었을 때 종료하고 출력한다. 만약 distance[i]=INT\_MAX(무한대)라면 해당 정점으로는 도달할 수 없으므로 출력 시 x로 표시한다.

#### E. Bellman-Ford

Bellman-Ford 알고리즘은 시작 정점에서 모든 정점까지의 최단 경로를 계산하는 알고리즘으로, Dijkstra와 달리 음수 가중치가 존재해도 동작한다. 먼저 distance 배열을 생성하여 모든 정점의 거리를 INT\_MAX(무한대)로 초기화하고, 시작 정점만 0으로 설정한다. Before 배열은 각 정점까지의 최단 경로에서 직전 정점을 저장하며, 초기 값은 모두 -1이다. 그래프의 모든 간선은 (weight, (start, end)) 형태로 edges 벡터에 저장한다. 현재 알고 있는 최단 경로보다 더 짧은 경로가 발견되면 해당 값을 갱신하고 end 정점의 직전 정점을 start로 기록한다. 이 반복을 통해 각 정점의 최단 거리 후보 값이 점차 실제 최단 거리로 수렴한다. 모든 간선에 대해 한 번 더 검사를 할 때도 더 짧은 경로가 발견된다면 이는 음수 사이클이 존재한다는 의미이다. 이 경우에는 false를 반

환한다.  $\text{Distance}[e\_vertex] = \text{INT\_MAX}$ 라면 해당 정점까지 갈 수 없으므로  $x$ 를 출력한다. 도달이 가능한 경우에는 before 거꾸로 따라가며 스택에 기록하고, 이를 뒤집어 최단 경로를 출력한다.

## F. FLOYD

Floyd 알고리즘은 그래프의 모든 정점 쌍 사이의 최단 거리를 한 번에 계산하는 알고리즘으로, 동적 계획법(Dynamic Programming)을 기반으로 한다. 이 알고리즘은 음수 가중치가 있어도 동작하지만, 음수 사이클이 존재하면 올바른 결과를 낼 수 없다. 우선 그래프의 정점 수만큼  $\text{arr}[i][j]$ 의 2차원 배열을 생성하고, 각 위치에  $i \rightarrow j$ 로 가는 초기 거리를 저장한다.  $i$ 와  $j$ 가 같은 경우 거리는 0으로 설정한다.  $i$  와  $j$  사이에 간선이 존재하면 가중치를 저장하고 없다면 무한대로 설정한다. 이를 위해 adjacency list를 전부 읽어 모든 간선을 (weight, (start, end)) 형태로 edges 벡터에 저장한 뒤, 이를 기반으로 초기 거리 테이블을 만든다. 이 알고리즘의 핵심은  $k$ 를 경우 했을 때 더 짧은 경로가 된다면 갱신하는 것이다. 이 알고리즘에서는  $\text{arr}[i][i]$  값이 0이 아닌 경우 음수 사이클 존재로 판단할 수 있다. 코드에서도 마지막에 모든  $i$ 에 대한 검사를 진행하고 음수 사이클이 발견되면 `false`를 반환한다. 안전하게 모든 계산이 끝났다면  $i$ 에서  $j$ 로 가능 최단 거리를 표 형태로 출력한다. 무한대 값은 도달할 수 없는 것으로 판단되어  $x$ 를 출력한다.

## G. 근접 중심성

이 함수는 그래프의 각 정점이 전체 그래프 구조에서 얼마나 중심적인 위치에 있는지를 계산하는 알고리즘이다. 근접 중심성은 해당 정점에서 다른 모든 정점까지의 최단 거리 합을 이용해 계산한다.

먼저, 그래프의 정점 수만큼 2차원 배열(distance)을 생성한다. 모든 간선을 수집한 뒤  $i \rightarrow j$ 로 가는 간선이 존재하면 해당 가중치를, 존재하지 않으면 무한대로,  $i == j$ 는 0으로 설정한다. 여기까지는 Floyd의 초기화 단계와 동일하다. 중심성 계산을 위해서는 각 정점에서 다른 정점까지의 최단 거리가 필요하므로 floyd와 동일하게 최단 거리를 갱신한다. 모든 검사와 계산이 끝난 후,  $i$ 에서 시작해 도달할 수 있는 모든 정점까지의 거리 합을 계산한다. 만약 하나라도 도달할 수 없는 정점이 있다면 중심성을 계산할 수 없으므로  $x$ 로 표시하고 패스한다. 모든 정점에 도달 가능하면  $\text{centrality}[i] = (\text{size}-1)/\text{거리합}$  으로 계산한다. 즉, 거리합이 작을수록 중심성이 커진다. 이 모든 중심성 중 가장 큰 값을 찾아 해당 정점에 대해 가장 가깝다는 표시를 해준다.

## 4. Result Screen

### A. 입력 graph 파일 , command 파일

list, matrix graph는 다음과 같은 내용의 파일을 사용했다.

L  
10  
0  
1 4  
2 1  
3 7  
1  
3 2  
4 5  
2  
5 3  
6 8  
3  
7 6  
4  
1 5  
8 9  
5  
2 3|  
9 4  
6  
3 8  
7  
0 2  
8  
4 9  
5 1  
9  
6 7

M  
10  
0 5 2 0 9 0 0 0 0 0  
0 0 0 7 0 3 0 0 0 0  
0 0 0 4 1 0 8 0 0 0  
6 0 0 0 0 0 3 5 0 0  
0 0 0 0 0 2 0 0 7 0  
0 0 0 0 0 0 6 0 4  
0 0 0 0 0 0 0 3 8  
0 0 0 0 0 0 0 0 2  
0 0 0 0 0 0 0 0 1  
0 0 0 0 0 0 0 0 0  
|

```
LOAD graph_L.txt
PRINT
PRINT hi
BFS O 0
BFS O
BFS O 11
BFS O 2 3
BFS X 0
DFS O 0
DFS X 0
KRUSKAL
KRUSKAL 2
DIJKSTRA O 0
DIJKSTRA X 0
DIJKSTRA O 30
DIJKSTRA O 1 3
BELLMANFORD O 4 1
BELLMANFORD O 4
BELLMANFORD X|
BELLMANFORD X 1 3 2
FLOYD
FLOYD X 1
FLOYD O
CENTRALITY
CENTRALITY 2
DS
EXIT
```

Command 파일은 여러가지 예외처리가 정상적으로 되었는지 확인하기 위해 인자가 부족하거나 많은 경우, 명령어가 잘못된 경우를 포함하였다.

B. PRINT

```
PRINT
PRINT hi
```

```
=====PRINT=====
[0] -> (1,4) -> (2,1) -> (3,7)
[1] -> (3,2) -> (4,5)
[2] -> (5,3) -> (6,8)
[3] -> (7,6)
[4] -> (1,5) -> (8,9)
[5] -> (2,3) -> (9,4)
[6] -> (3,8)
[7] -> (0,2)
[8] -> (4,9) -> (5,1)
[9] -> (6,7)
=====

=====ERROR=====
200
=====
```

Commnad의 print의 첫 번째 print는 정상 입력, 두 번째는 인자가 추가된 경우이다. 각각에 알맞게, list 구조와 에러코드를 출력했다.

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

Matrix의 경우도 정상적으로 출력한다.

### C. BFS

```
BFS O 0
BFS O
BFS O 11
BFS O 2 3
BFS X 0
```

```
=====BFS=====
Directed Graph BFS
Start: 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9
=====

=====ERROR=====
300
=====

=====ERROR=====
300
=====

=====ERROR=====
300
=====

=====BFS=====
Undirected Graph BFS
Start: 0
0 -> 1 -> 2 -> 3 -> 7 -> 4 -> 5 -> 6 -> 8 -> 9
=====
```

Command에서 BFS는 순서대로, 정상, 예러, 예러, 정상의 결과를 유도하여 작성되었고, 실제 결과도 일치한다.

#### D. DFS

```
DFS O 0
DFS X 0
=====DFS=====
Directed Graph DFS
Start :0
0 -> 1 -> 3 -> 7 -> 4 -> 8 -> 5 -> 2 -> 6 -> 9
=====

=====DFS=====
Undirected Graph DFS
Start :0
0 -> 1 -> 3 -> 6 -> 2 -> 5 -> 8 -> 4 -> 9 -> 7
=====
```

BFS와 DFS는 입력 받는 구조가 동일하여 예외처리도 동일하게 해, 예외처리에 대한 확인은 생략하였고, Direct, Undirect에 대해 정상적으로 출력되는지만 확인하였다.

E. KRUSKAL

```
KRUSKAL
KRUSKAL 2
=====
[0] 1(4) 2(1) 7(2)
[1] 0(4) 3(2) 4(5)
[2] 0(1) 5(3)
[3] 1(2)
[4] 1(5)
[5] 2(3) 8(1) 9(4)
[6] 9(7)
[7] 0(2)
[8] 5(1)
[9] 5(4) 6(7)
Cost: 29
=====

=====
ERROR=====
500
=====
```

Kruskal도 정상, 예상 출력을 목적으로 command를 작성하였고 예상대로 출력됨을 확인할 수 있다.

F. DIJKSTRA

```
DIJKSTRA O 0
DIJKSTRA X 0
DIJKSTRA O 30
DIJKSTRA O 1 3
```

```
=====DIJKSTRA=====
Directed Graph Dijkstra
Start: 0
[0] 0 (0)
[1] 0 -> 1 (4)
[2] 0 -> 2 (1)
[3] 0 -> 1 -> 3 (6)
[4] 0 -> 1 -> 4 (9)
[5] 0 -> 2 -> 5 (4)
[6] 0 -> 2 -> 6 (9)
[7] 0 -> 1 -> 3 -> 7 (12)
[8] 0 -> 1 -> 4 -> 8 (18)
[9] 0 -> 2 -> 5 -> 9 (8)
=====

=====DIJKSTRA=====
Undirected Graph Dijkstra
Start: 0
[0] 0 (0)
[1] 0 -> 1 (4)
[2] 0 -> 2 (1)
[3] 0 -> 1 -> 3 (6)
[4] 0 -> 1 -> 4 (9)
[5] 0 -> 2 -> 5 (4)
[6] 0 -> 2 -> 6 (9)
[7] 0 -> 7 (2)
[8] 0 -> 2 -> 5 -> 8 (5)
[9] 0 -> 2 -> 5 -> 9 (8)
=====

=====ERROR=====
600
=====

=====ERROR=====
600
=====
```

Dijkstra도 정상, 정상, 에러, 에러 출력을 목적으로 command를 작성했고, 예상대로 출력되었다.

#### G. BELLMANFORD

```
BELLMANFORD O 4 1  
BELLMANFORD O 4  
BELLMANFORD X  
BELLMANFORD X 1 3 2
```

```
=====BELLMANFORD=====
```

```
Directed Graph Bellman-Ford
```

```
4 -> 1
```

```
Cost: 5
```

```
=====
```

```
=====ERROR=====
```

```
700
```

```
=====
```

```
=====ERROR=====
```

```
700
```

```
=====
```

```
=====ERROR=====
```

```
700
```

```
=====
```

Bellman-Ford도 정상, 예러, 예러, 예러를 목적으로 작성하였고, 예상 결과와 동일하게 출력되었음을 확인할 수 있다.

#### H. FLOYD

```
FLOYD  
FLOYD X 1  
FLOYD O
```

```

=====ERROR=====
800
=====

=====ERROR=====
800
=====

=====FLOYD=====
Directed Graph Floyd
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[0] 0 4 1 6 9 4 9 12 18 8
[1] 10 0 11 2 5 14 19 8 14 18
[2] 24 28 0 16 33 3 8 22 42 7
[3] 8 12 9 0 17 12 17 6 26 16
[4] 15 5 13 7 0 10 21 13 9 14
[5] 27 31 3 19 36 0 11 25 45 4
[6] 16 20 17 8 25 20 0 14 34 24
[7] 2 6 3 8 11 6 11 0 20 10
[8] 24 14 4 16 9 1 12 22 0 5
[9] 23 27 24 15 32 27 7 21 41 0
=====
```

Floyd도 에러, 에러, 정상을 목적으로 하였고, 예상 결과와 일치하는 출력을 확인할 수 있다.

본 그래프가 서로 모두 연결되어 있는 그래프이기 때문에 x인 곳이 존재하지 않는다.

#### I. CENTRALITY

```
CENTRALITY
CENTRALITY 2
```

```
=====CENTRALITY=====
[0] 9/48 <- Most Central
[1] 9/61
[2] 9/48 <- Most Central
[3] 9/71
[4] 9/90
[5] 9/57
[6] 9/91
[7] 9/62
[8] 9/63
[9] 9/81
=====
=====ERROR=====
900
=====
```

정상, 에러를 목적으로 했고, 동일하게 출력되었음을 확인할 수 있다.

#### J. EXIT

```
=====EXIT=====
Success
=====
```

종료까지 정상적으로 출력된다.

### 5. Consideration

Kruskal 알고리즘을 구현할 때 MST 비용이 기대값과 다르게 나오는 문제가 있었다. 원인은 동일한 정점 쌍 사이에 가중치가 다른 복수의 간선이 존재 할 때, 정렬과정에서 가중치가 작은 간선이 우선 선택되지 않거나 사이클 판별 로직이 불안정했기 때문이다. 그래서 직접 구현한 불안정한 함수를 사용하지 않고 <algorithm>의 sort함수를 사용하여 안정성을 확보하였다.