DS_Project_3

보고서

제출일자: 2025년 12월 07일

학과: 컴퓨터정보공학부

담당교수: 최상호 교수님

학번: 2022202053

성명: 정윤철

1. Introduction

본 프로젝트는 다양한 그래프 알고리즘을 직접 구현함으로써, 그래프 자료구조에 대한 이해도를 높이고 실제 그래프 자료구조의 적용 능력을 함양하는 것을 목표로 둡니다.

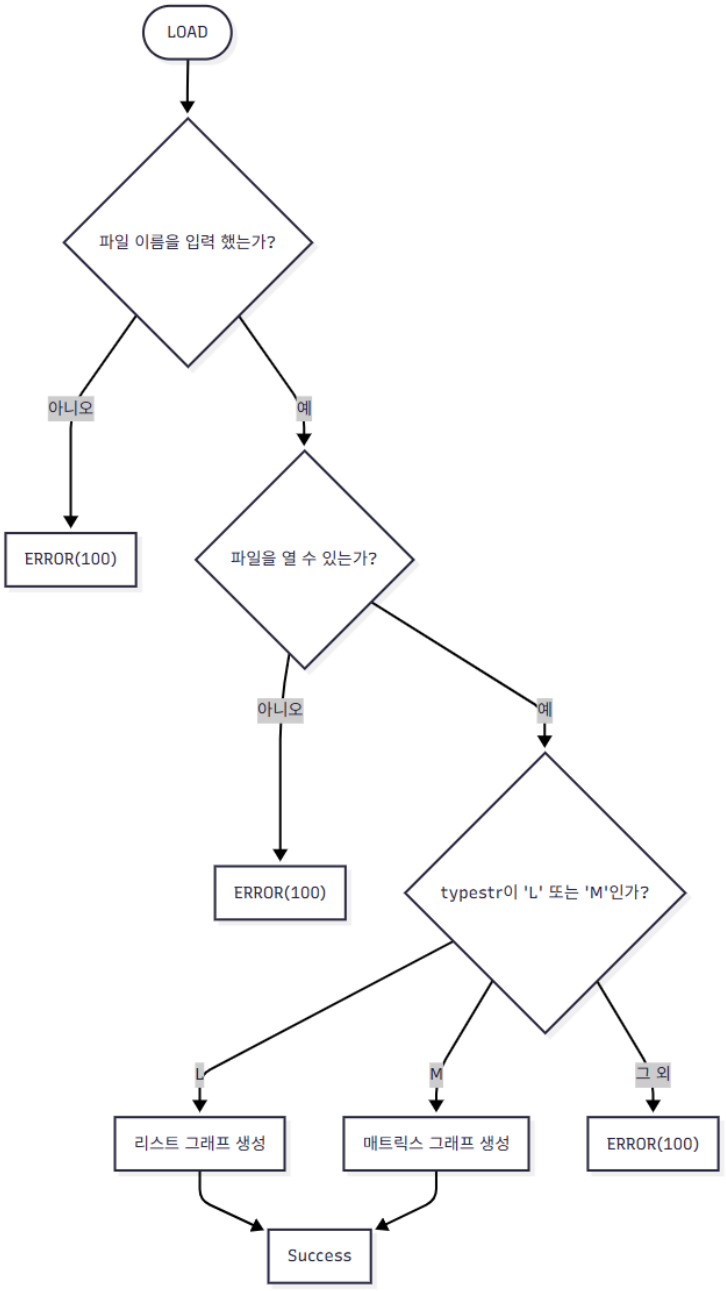
본 프로그램은 인접리스트, 인접 행렬 두가지 형태를 모두 지원합니다. 인접 리스트는 graph_L.txt 인접 행렬은 graph_M.txt 파일에 저장되어 있어서, LOAD 명령어를 통해서 .txt파일에 저장되어 있는 그래프를 인접행렬인지, 인접리스트 인지 판단해서, ListGragh 혹은 MatrixGraph 객체를 생성해서 데이터를 메모리에 저장합니다.

BFS와 DFS를 이용해서 그래프의 구조를 탐색하며, 최소 신장 트리를 구하는 Kruskal, 최단 경로 탐색을위한 Dijkstra, Bellman-Ford, FLoyd, 정점의 중요도를 분석하는 Centrality 알고리즘을 구현했습니다. 이를 통해서 그래프의 방향성의 유무, 가중치의 유무, 음의 가중치 포함 여부 등 다양한 조건에 맞춰서 최소 비용 경로를 도출할 수 있도록 구현했습니다.

2. Flowchart

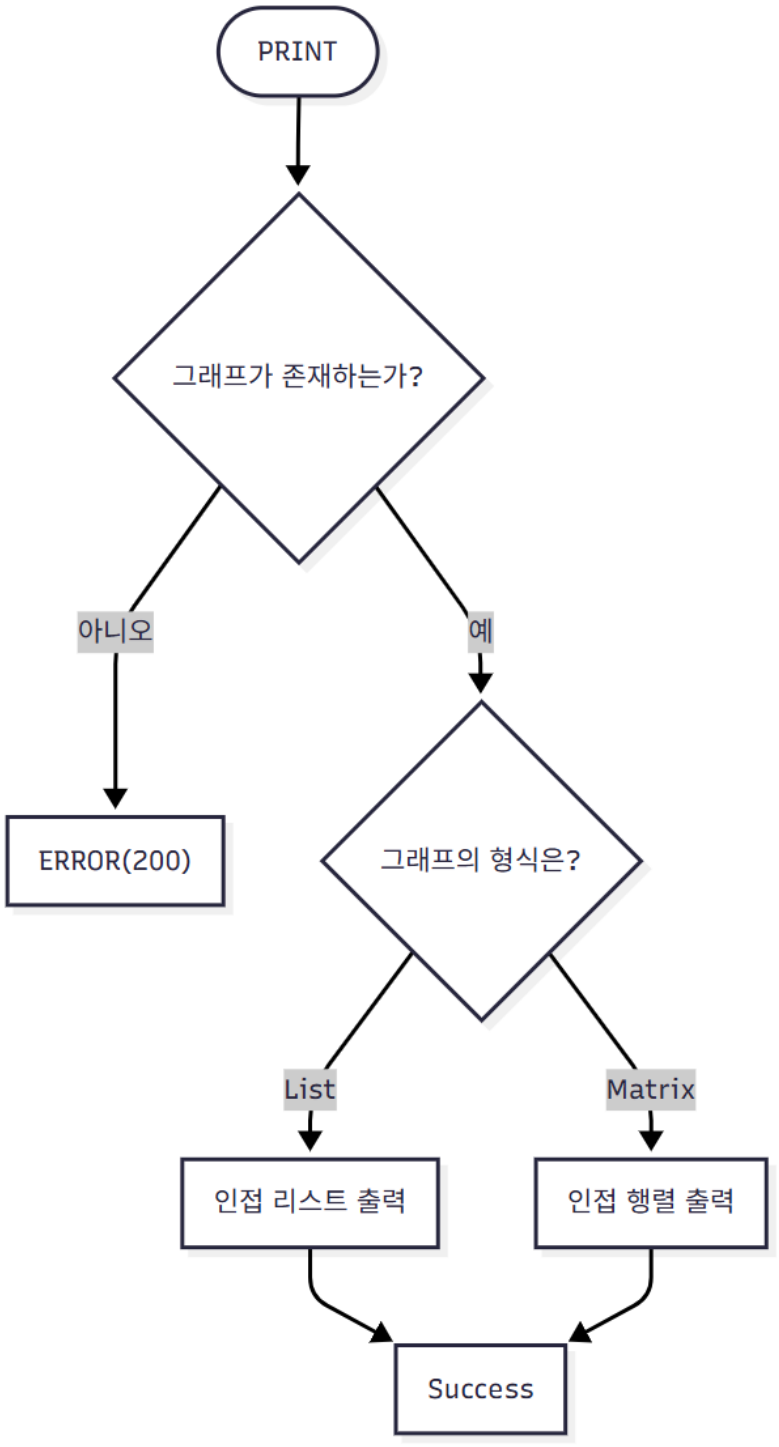
플로우차트는 명령어 별로 명령어가 어떻게 작동하는지에 대해서 작성했습니다.

LOAD



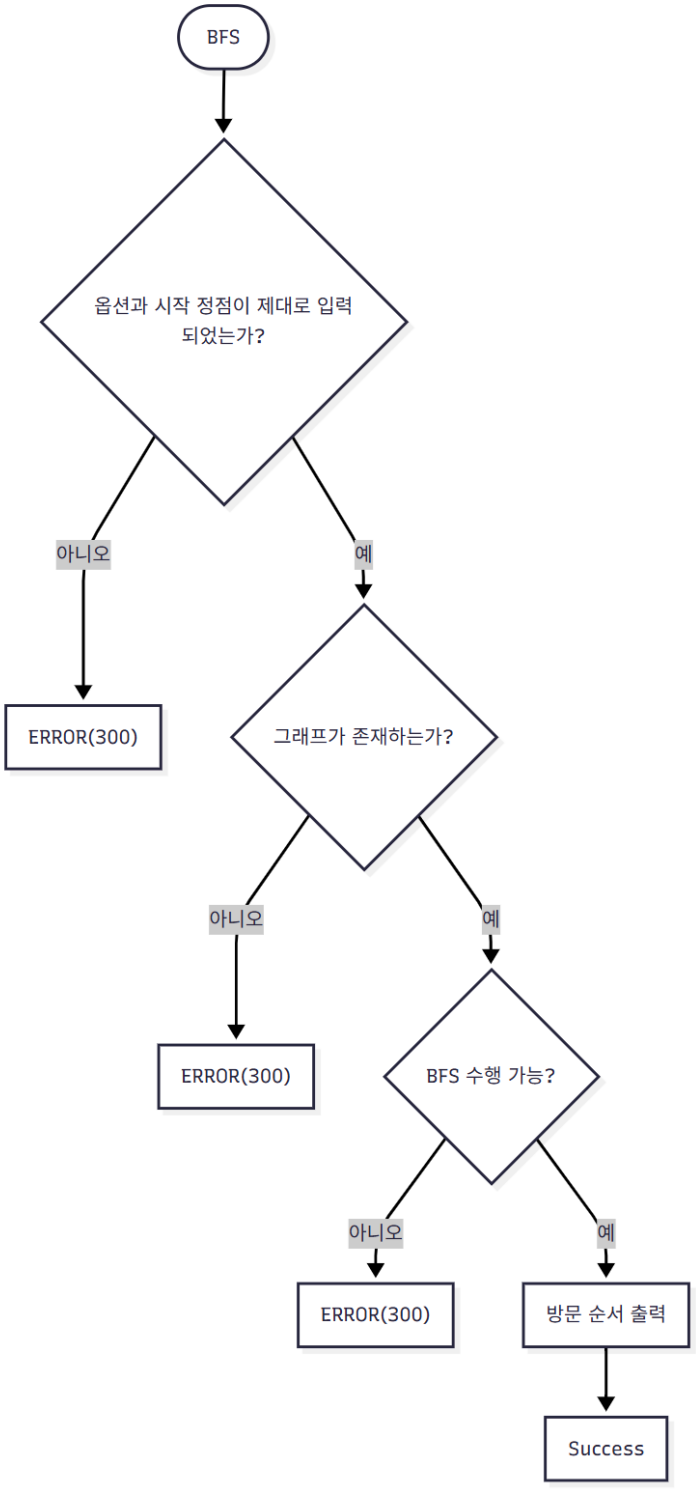
LOAD 명령어는 인자로 filename을 인자로 받고, graph_L, graph_M에는 첫번째 줄에 L또는 M이 입력 되어 있는데, 이를 typestr로 설정하고, typestr이 L이면 인접리스트 그래프를 생성하고, M이면 인접행렬 그래프를 생성한다.

PRINT



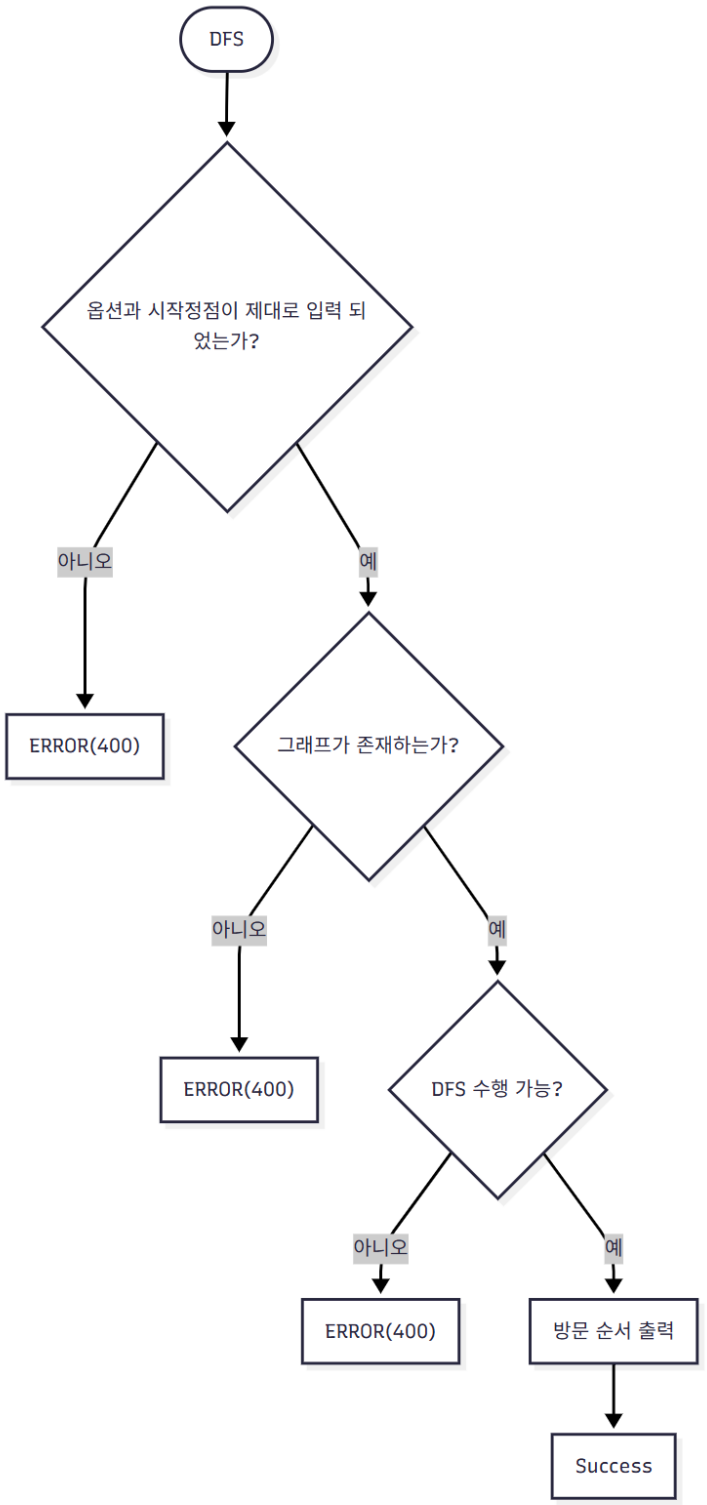
PRINT 명령어를 실행하면 현재 저장되어 있는 그래프가 인접행렬 그래프인지, 인접 리스트 그래프인지 파악 후에, 해당하는 그래프를 알맞은 형식으로 출력합니다.

BFS



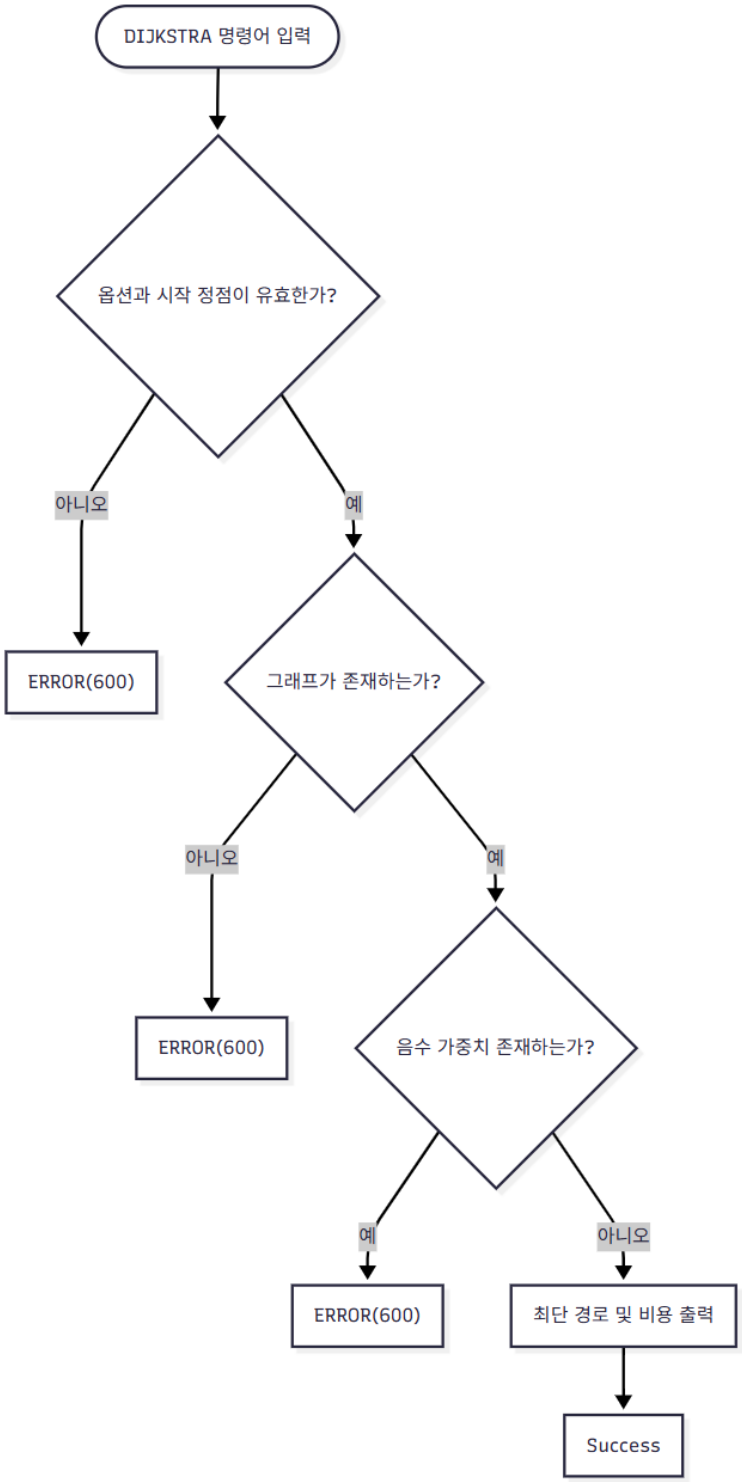
BFS는 인자로 방향성과, 시작 정점 번호를 입력받으며, 방향성은 'O' 이면 방향성이 존재하고, 'X'의 경우에는 방향성이 존재하지 않습니다. 그래프가 존재하지 않거나 입력한 시작 정점이 존재 하지 않아서 BFS를 진행하지 못하거나, 입력 인자 두 가지를 제대로 입력하지 않는 경우에 ERROR코드를 출력합니다.

DFS



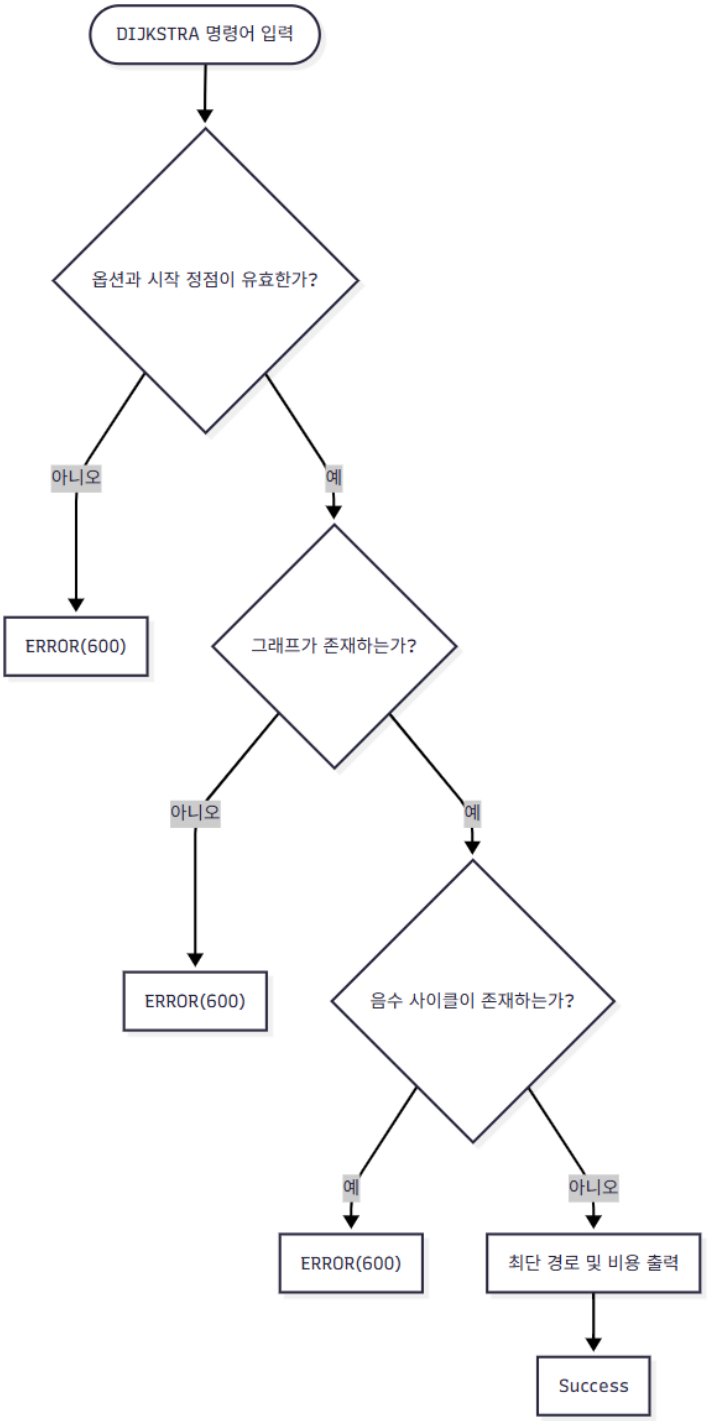
DFS는 인자로 방향성과, 시작 정점 번호를 입력받으며, 방향성은 'O' 이면 방향성이 존재하고, 'X'의 경우에는 방향성이 존재하지 않습니다. 그래프가 존재하지 않거나 입력한 시작 정점이 존재 하지 않아서 DFS를 진행하지 못하거나, 입력 인자 두 가지를 제대로 입력하지 않는 경우에 ERROR코드를 출력합니다.

DIJKSTRA



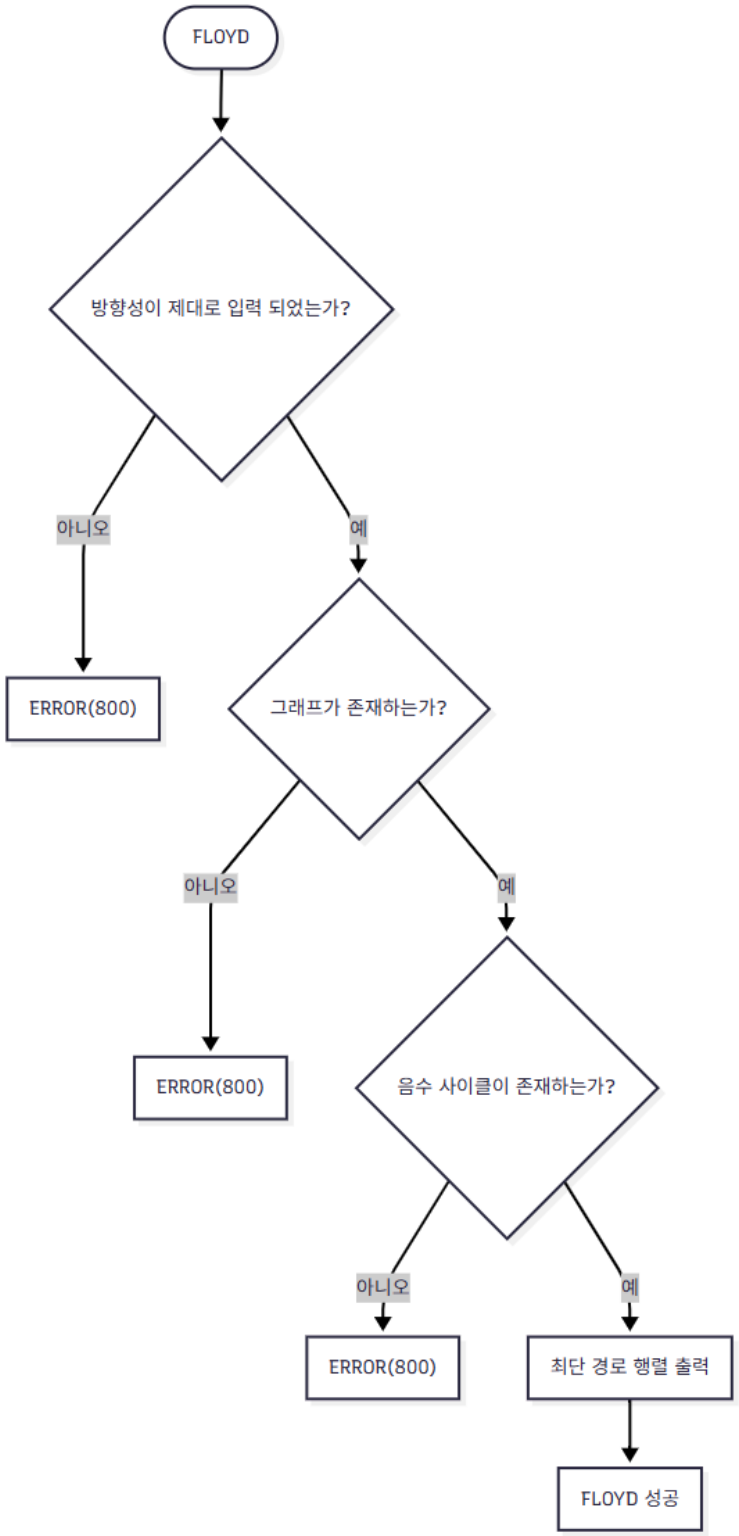
DIJKSTRA 명령어는 인자로 방향성과 시작 정점을 입력 받습니다. 시작 정점부터 알고리즘을 수행해서, 다른 모든 정점까지의 최단 경로 와 그 비용을 출력합니다. 시작 정점에서 도착할 수 없는 정점의 경우 'x'를 출력합니다. 인자 두 가지를 제대로 입력 받지 못하거나, 음수가중치가 존재하는 경우에 ERROR코드를 출력합니다. ERROR코드를 출력하지 않는 경우에는 최단 경로와 최소 비용을 출력합니다.

BELLMANFORD



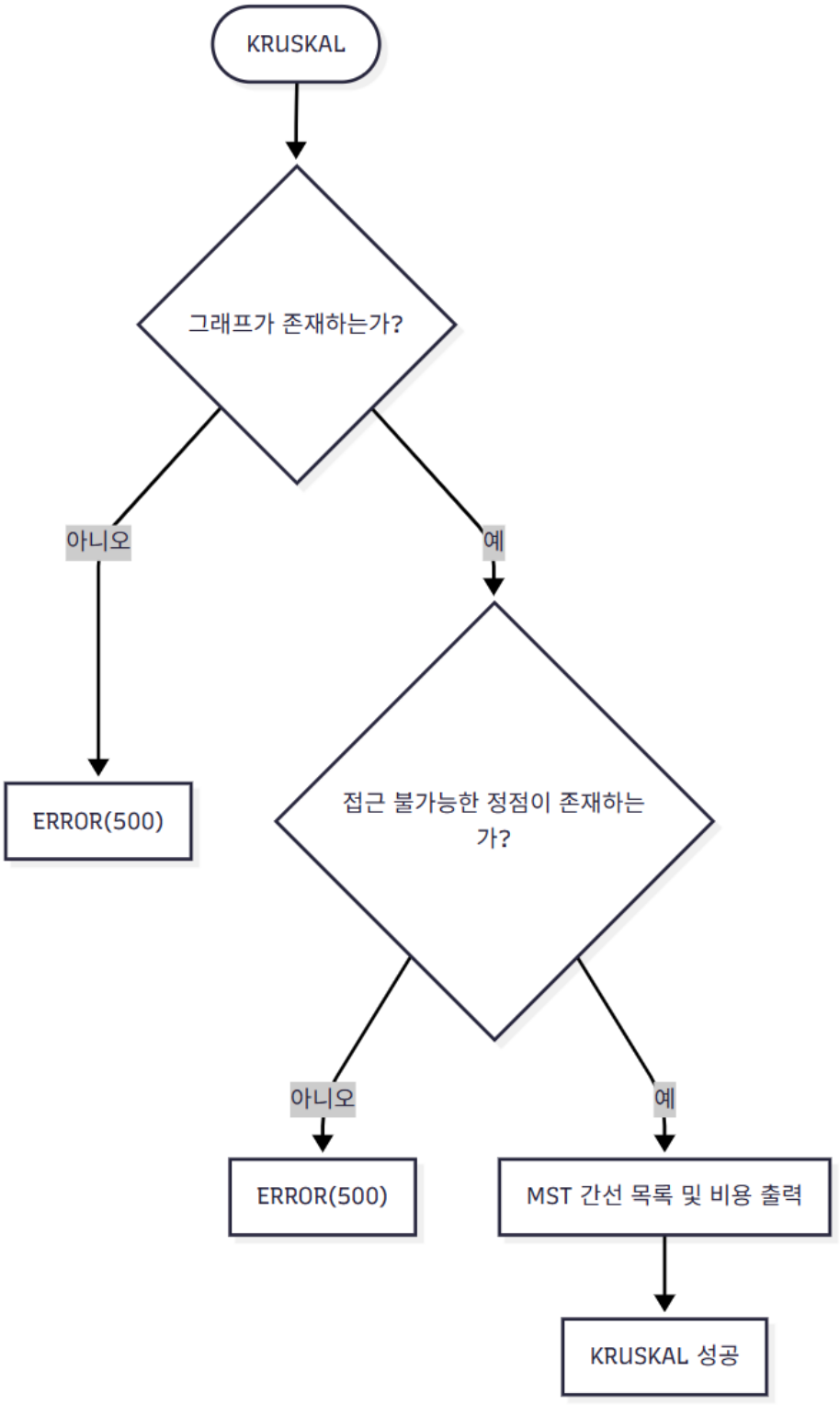
BELLMANFORD 명령어는 옵션과 시작 정점을 인자로 받고, 시작 정점에서 도착 정점까지의 최단경로와 비용을 출력합니다. 인자가 제대로 입력되지 않은 경우, 그래프가 존재하지 않는 경우, 음수 사이클이 존재하는 경우 ERROR코드를 출력한다. 시작 정점에서 도착 정점으로 도달할 수 없는 경우에는 'x'를 출력하도록 구성했습니다.

FLOYD



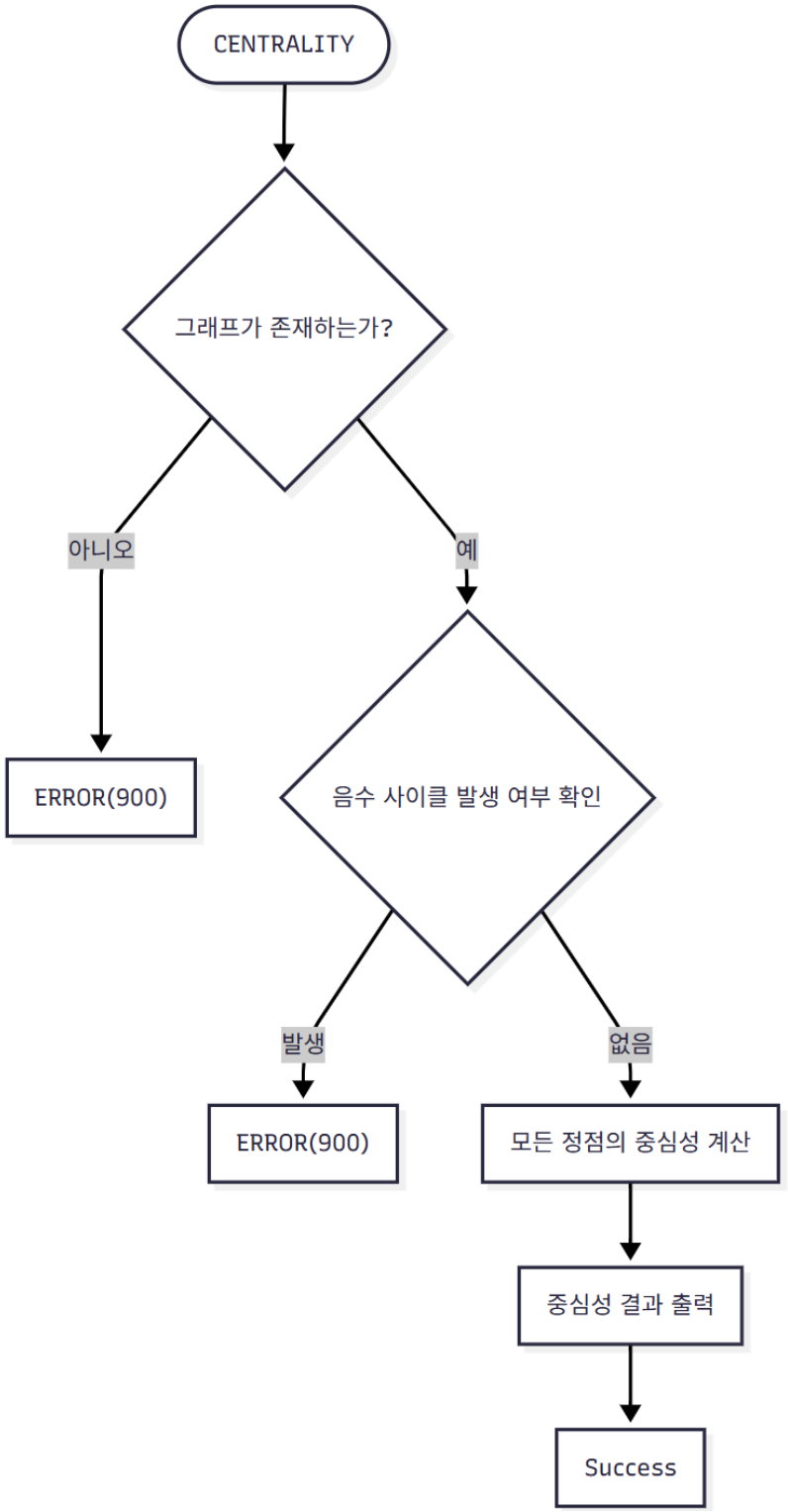
FLOYD 명령어는 방향성을 인자로 받으며, 행렬 형태로 행은 시작 정점, 열은 도착 정점을 나타내도록 출력하며 행렬의 요소에는 시작 정점에서 도착 정점으로 향하는 최단 경로를 출력하고, 도달할 수 없는 경우 'x'를 출력하도록 구성했습니다. 그래프가 없거나, 음수 가중치가 존재하는 경우에 ERROR 코드를 출력하도록 구성했습니다.

KRUSKAL



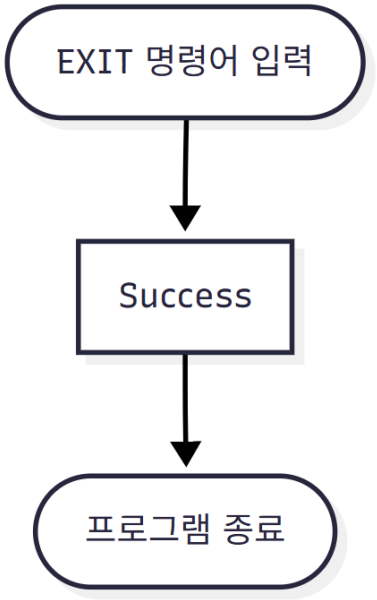
KRUSKAL 명령어는 어떠한 인자도 입력받지 않으며, 그래프가 없거나, 접근 불가능한 정점이 있는 경우 ERROR코드를 출력합니다. ERROR코드가 출력되는 경우가 아니라면, 저장된 그래프의 MST를 구한 후, MST와 MST의 가중치를 출력합니다.

CENTRALITY



CENTRALITY 명령어는 그래프가 없거나, 음수 사이클이 발생한 경우 ERROr코드를 출력하며, ERROR코드를 출력하지 않는 경우에는 모든 정점의 중심성을 계산하고, 정점 번호 오름차순 순으로 중심성을 분수 형태로 출력하며, 중심성이 가장 높은 정점에 "<- Most Central" 이라고 출력해서 표시해준다.

EXIT



EXIT 명령어는
프로그램에 할당된 모든 메모리를 해제하며 프로그램을 종료합니다.

3. Algorithm

알고리즘은 구현한 .cpp파일에 대한 설명과, 주요 함수들에 대해서 설명하겠습니다.

GraphMethod.cpp

해당 .cpp파일에서는 그래프 탐색, 최소 신장 트리 생성, 최단 경로 탐색, 중심성 분석 알고리즘을 구현했습니다.

Struct Edge

- 간선의 정보를 struct로 선언해서 객체 안에 u, v를 출발 정점과 도착 정점으로 관리하고, weight 변수로 가중치를 관리하며, operator< 연산자 오버로딩을 이용해서 가중치 오름차순 정렬을 구현했습니다.

Struct DisjointSet

- Kruskal 알고리즘 수행 중 사이클 형성을 효율적으로 감지하기 위한 구조체이다.
- vector<int>parent 각 정점이 속한 집합의 부모 노드를 저장해서 두 정점이 같은 집합에 속해 있는지를 판단한다.

bool BFS(Graph *graph, char option, int vertex)

- 큐 자료구조를 이용해서 그래프를 BFS, 너비 우선 탐색을 진행하는 함수입니다. q에서 정점을 꺼내 방문 처리하고, 해당 정점의 인접 정점 중에서 방문하지 않았던 정점들을 q에 삽입하는 로직입니다.
- vector<bool> visited – 벡터 선언을 이용해서 정점의 방문 여부를 저장해서 중복 방문을 방지했습니다.
- queue<int> q – 방문할 정점들을 순서대로 저장하기 위해서 queue 자료구조를 사용했습니다.
- map<int, int> adj - 현재 정점과 연결된 인접 정점 및 가중치를 파악하기 위해서 map을 이용했습니다.

bool DFS(Graph *graph, char option, int vertex)

- 스택 자료구조를 이용해서 그래프를 DFS, 깊이 우선 탐색을 진행하는 함수입니다. 정점 번호가 낮은 순서대로 방문하기 위해서 인접 정점들을 역순으로 스택에 삽입해서 스택에서 pop을 진행할 때, 오름차순이 되도록 구현했습니다.
- vector<bool> visited - 벡터 선언을 이용해서 정점의 방문 여부를 저장해서 중복 방문을 방지했습니다.
- stack<int> s – 탐색 경로를 파악하기 위해서 스택 자료구조를 이용했습니다.
- map<int, int> adj - 현재 정점과 연결된 인접 정점 및 가중치를 파악하기 위해서 map을 이용했습니다.

bool Kruskal(Graph *graph)

- 가중치가 가장 작은 간선부터 선택해서 사이클 없이 모든 정점을 연결하는 최소 비용의 트리(MST)를 구성하는 함수입니다.
- vector<Edge> edges – 그래프에 존재하는 모든 간선을 관리하기 위해서 vector를 이용해서 관리합니다.
- DisjointSet ds – 정점 간의 연결성을 확인하기 위해서 구조체로 선언했던 DisjointSet을 이용합니다.

- Logic
    1. 모든 간선을 가중치 오름차순으로 정렬한다.
    2. 정렬된 간선을 순회하며 ds.find()를 통해 두 정점이 서로 다른 집합에 속해 있는지 확인합니다.
    3. 다른 집합일 경우 ds.merge()로 연결 후에 최소 신장 트리에 포함시킵니다.

4. 그래프가 단절 되어 있어서 최소 신장 트리를 생성할 수 없는 경우에 ERROR를 출력합니다.

bool Dijkstra(Graph *graph, char option, int vertex)

- 음수 가중치를 허용하지 않는 그래프에서 최단 경로를 구하고, 음수 사이클이 존재하는지 판단합니다.
- priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq 현재 탐색 가능한 정점 중 최소 비용을 가진 정점을 시간 복잡도로 추출하기 위한 최소 힙
- vector<int> dist 시작 정점으로부터 각 정점까지 발견된 최단 거리를 저장 및 갱신하는 배열
- Logic
    1. 시작점의 거리를 0으로 설정하고, 큐에 넣은 뒤, 가장 거리가 짧은 정점을 꺼내서 인접 정점들의 거리를 갱신한다.
    2. 최종 계산된 dist배열을 바탕으로 경로를 역순으로 출력한다.

bool Bellmanford(Graph *graph, char option, int s_vertex, int e_vertex)

- 음수 가중치르 허용하는 그래프에서 최단 경로를 구하며, 음수 사이클의 존재 여부를 판단합니다.
- vector<int> dist – 정점의 최단거리를 저장하는 배열이며, 반복적으로 최단 거리를 찾는 과정에서 더 짧은 거리가 발견되면 갱신하는 vector 이다.
- Logic
    1. 시작 정점에서 시작 정점 제외 모든 정점들까지의 최단 거리를 갱신한다.
    2. 모든 정점까지의 거리 갱신을 완료하고, 한 번더 시행했을때, 거리가 줄어드는 정점이 있다면, 음수 사이클이 존재한다고 판단하고 에러를 출력한다.

bool FLOYD(Graph *graph, char option)

- 모든 정점 쌍 간의 최단 경로를 계산하는 함수이다.
- vector< vector<int>>> dist - 2차원 인접 행렬 형태로, dist[i][j]는 i정점에서 j정점으로 가는 최단 경로를 나타낸다.
- 음수 사이클이 존재하는 경우 ERROR를 출력한다.
- Logic
    1. dist[i][i], 즉 대각선의 값들은 모두 0으로 설정하고, dist[i][j]에서 간선이 존재하는 경우에는 간선의 가중치를 입력하고, 없는 경우에는 INF로 초기화한다.

2. 삼중반복문을 이용해서 dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]) 해당 식을 이용해서 i에서 j로 직접 가는것보다 k를 경유하는 경우가 더 빠를 경우 dist[i][j]를 갱힌한다.
3. 모든 연산이 종료된 후 대각 성분 dist[i][i]가 0보다 작은 값이 있다면, 음수 사이클이 존재한다고 판단하고 ERROR코드를 출력한다.

bool Centrality(Graph *graph)

- 각 정점이 다른 모든 정점과 얼마나 가깝게 연결되어 있는지를 숫자로 표현해서, 그래프 내의 정점들의 중요도를 파악하는 함수이다.
- vector<vector<int>>dist – Floyd 알고리즘을 통해서 계산된 모든 정점 간 최단 거리를 저장하고 있는 배열이다.
- vector<pair<double, int>> closeness – 각 정점별로 계산된 중심성 수치와 정점 번호를 쌍으로 저장하는 배열이다. 가장 높은 중심성을 가진 정점을 찾는데 사용된다.
- Logic
    1. Floyd 알고리즘의 로직을 재사용해서 모든 정점 쌍 간 최단 거리를 계산한다.
    2. 만약 dist[i][j]가 INF라면, 그래프가 단절되어 있다고 판단하고, 해당 정점으로 갈 수 없으므로, 중심성을 계산하지 않는다.
    3. 중심성을 계산할 수 있는 경로의 중심성을 계산한다.
    4. 계산된 중심성들 중에서 가장 최댓값을 가진 정점에 "Most Central"로 표시한다.

ListGraph.cpp, MatrixGraph.cpp

- LOAD 명령어를 진행한 .txt 파일의 첫번째 라인이 'L'인 경우에는 ListGraph 객체를 생성하고 , 'M'인 경우에는 MatrixGraph 객체를 생성한다.
- getAdjacentEdges(int vertex, map<int, int>* m) – 인접한 간선의 정보를 반환하는 함수로 방향성이 없는 그래프의 경우 사용한다.
- getAdjacentEdgesDirect(int vertex, map<int, int>* m) – 인접한 간선의 정보를 반환하는 함수로 방향성이 있는 함수에서 사용한다.
- insertEdge(int from, int to, int weight) – 간선을 삽입하는 함수로, from은 시작정점, to는 도착정점, weight는 가중치를 나타낸다
- printGraph(ofstream *fout) – 그래프를 포맷에 알맞게 출력하는 함수이다.

4. Result Screen

LOAD, PRINT, EXIT

command.txt	graph_L.txt	graph_M.txt

command.txt

```
LOAD graph_L.txt
PRINT
LOAD graph_M.txt
PRINT
EXIT
```

graph_L.txt

```
L
10
0
1 2
2 5
1
3 3
4 10
2
4 2
5 7
3
6 1
4
3 1
6 8
7 4
5
7 2
6
8 5
7
8 1
9 6
8
9 3
9
0 15
9
```

graph_M.txt

```
M
10
0 2 5 0 0 0 0 0 0 0
0 0 0 3 10 0 0 0 0 0
0 0 0 0 2 7 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 8 4 0 0
0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 0 0 5 0
0 0 0 0 0 0 0 0 1 6
0 0 0 0 0 0 0 0 0 3
15 0 0 0 0 0 0 0 0 0
```

위와 같이 파일들을 설정하고 실행 시켜서, LOAD를 두 번 실행 하면 기존의 그래프가 제대로 삭제가 되는지, 그래프를 제대로 출력하는지 확인했습니다.

log.txt

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

=======PRINT=======
[0]->(1,2)->(2,5)
[1]->(3,3)->(4,10)
[2]->(4,2)->(5,7)
[3]->(6,1)
[4]->(3,1)->(6,8)->(7,4)
[5]->(7,2)
[6]->(8,5)
[7]->(8,1)->(9,6)
[8]->(9,3)
[9]->(0,15)
====================

=======LOAD=======
Success
====================

=======PRINT=======
    [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[0] 0   2   5   0   0   0   0   0   0   0
[1] 0   0   0   3   10  0   0   0   0   0
[2] 0   0   0   0   2   7   0   0   0   0
[3] 0   0   0   0   0   0   1   0   0   0
[4] 0   0   0   1   0   0   8   4   0   0
[5] 0   0   0   0   0   0   0   2   0   0
[6] 0   0   0   0   0   0   0   0   5   0
[7] 0   0   0   0   0   0   0   0   1   6
[8] 0   0   0   0   0   0   0   0   0   3
[9] 15  0   0   0   0   0   0   0   0   0
====================

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

위의 인접 리스트 그래프와 인접 행렬 그래프 입력과 비교해보면, 값이 제대로 입력되었고, 두번째 LOAD 할 때 이전 그래프 정보는 사라진것을 확인했습니다. 또한 PRINT와 EXIT이 제대로 실행됨을 확인했습니다.

BFS

command.txt

```
LOAD graph_L.txt
BFS O 1
BFS X 0
LOAD graph_M.txt
BFS O 1
BFS X 0
EXIT
```

graph_L.txt

```
L
10
0
1 2
2 5
1
3 3
4 10
2
4 2
5 7
3
6 1
4
3 1
6 8
7 4
5
7 2
6
8 5
7
8 1
9 6
8
9 3
9
0 15
9
```

graph_M.txt

```
M
10
0 2 5 0 0 0 0 0 0 0
0 0 0 3 10 0 0 0 0 0
0 0 0 0 2 7 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 8 4 0 0
0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 0 0 5 0
0 0 0 0 0 0 0 0 1 6
0 0 0 0 0 0 0 0 0 3
15 0 0 0 0 0 0 0 0 0
```

각 파일들을 위처럼 설정하고 실행했을때의 log.txt 사진입니다.

log.txt

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

========PRINT========
[0]->(1,2)->(2,5)
[1]->(3,3)->(4,10)
[2]->(4,2)->(5,7)
[3]->(6,1)
[4]->(3,1)->(6,8)->(7,4)
[5]->(7,2)
[6]->(8,5)
[7]->(8,1)->(9,6)
[8]->(9,3)
[9]->(0,15)
====================

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

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

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

========PRINT========
    [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[0] 0   2   5   0   0   0   0   0   0   0
[1] 0   0   0   3   10  0   0   0   0   0
[2] 0   0   0   0   2   7   0   0   0   0
[3] 0   0   0   0   0   0   1   0   0   0
[4] 0   0   0   1   0   0   8   4   0   0
[5] 0   0   0   0   0   0   0   2   0   0
[6] 0   0   0   0   0   0   0   0   5   0
[7] 0   0   0   0   0   0   0   0   1   6
[8] 0   0   0   0   0   0   0   0   0   3
[9] 15  0   0   0   0   0   0   0   0   0
====================

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

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

BFS의 출력 결과가 제대로 출력 됨을 확인했으며, 방향성 설정에 따른 Directed, Undirected 구분이 제대로 되고 있으며, Start node의 셋팅도 제대로 되 있음을 확인했습니다.

DFS

command.txt	graph_L.txt	graph_M.txt

command.txt

```
LOAD graph_L.txt
DFS O 1
DFS X 0
LOAD graph_M.txt
DFS O 1
DFS X 0
EXIT
```

graph_L.txt

```
L
10
0
1 2
2 5
1
3 3
4 10
2
4 2
5 7
3
6 1
4
3 1
6 8
7 4
5
7 2
6
8 5
7
8 1
9 6
8
9 3
9
0 15
9
```

graph_M.txt

```
M
10
0 2 5 0 0 0 0 0 0 0
0 0 0 3 10 0 0 0 0 0
0 0 0 0 2 7 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 8 4 0 0
0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 0 0 5 0
0 0 0 0 0 0 0 0 1 6
0 0 0 0 0 0 0 0 0 3
15 0 0 0 0 0 0 0 0 0
```

각 파일들을 위처럼 설정하고 실행했을때의 log.txt 사진입니다.

log.txt

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

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

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

========LOAD========
Success
====================

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

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

DFS의 결과와 방향성 출력, 시작 노드 세팅이 제대로 됨을 확인했습니다.

KRUSKAL

command.txt	graph_L.txt	graph_M.txt

command.txt

```
LOAD graph_L.txt
KRUSKAL
LOAD graph_M.txt
KRUSKAL
EXIT
```

graph_L.txt

```
L
10
0
1 2
2 5
1
3 3
4 10
2
4 2
5 7
3
6 1
4
3 1
6 8
7 4
5
7 2
6
8 5
7
8 1
9 6
8
9 3
9
0 15
9
```

graph_M.txt

```
M
10
0 2 5 0 0 0 0 0 0 0
0 0 0 3 10 0 0 0 0 0
0 0 0 0 2 7 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 8 4 0 0
0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 0 0 5 0
0 0 0 0 0 0 0 0 1 6
0 0 0 0 0 0 0 0 0 3
15 0 0 0 0 0 0 0 0 0
```

각 파일들을 위처럼 설정하고 실행했을때의 log.txt 사진입니다

log.txt

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

========KRUSKAL========
[0] 1(2)
[1] 0(2) 3(3)
[2] 4(2)
[3] 1(3) 4(1) 6(1)
[4] 2(2) 3(1) 7(4)
[5] 7(2)
[6] 3(1)
[7] 4(4) 5(2) 8(1)
[8] 7(1) 9(3)
[9] 8(3)
Cost: 19
======================
```

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

========KRUSKAL========
[0] 1(2)
[1] 0(2) 3(3)
[2] 4(2)
[3] 1(3) 4(1) 6(1)
[4] 2(2) 3(1) 7(4)
[5] 7(2)
[6] 3(1)
[7] 4(4) 5(2) 8(1)
[8] 7(1) 9(3)
[9] 8(3)
Cost: 19
======================

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

값이 제대로 출력됨을 확인했으며, 두 그래프가 형태만 다르고 같은 그래프임을 고려했을때, 두 값이 똑같게 나오는 점에서 출력값이 안정된 값임을 확인했습니다.

DIJKSTRA

command.txt

```
LOAD graph_L.txt
DIJKSTRA O 1
DIJKSTRA X 1
LOAD graph_M.txt
DIJKSTRA O 1
DIJKSTRA X 1
EXIT
```

graph_L.txt

```
L
10
0
1 2
2 5
1
3 3
4 10
2
4 2
5 7
3
6 1
4
3 1
6 8
7 4
5
7 2
6
8 5
7
8 1
9 6
8
9 3
9
0 15
9
```

graph_M.txt

```
M
10
0 2 5 0 0 0 0 0 0 0
0 0 0 3 10 0 0 0 0 0
0 0 0 0 2 7 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 8 4 0 0
0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 0 0 5 0
0 0 0 0 0 0 0 0 1 6
0 0 0 0 0 0 0 0 0 3
15 0 0 0 0 0 0 0 0 0
```

각 파일들을 위처럼 설정하고 실행했을때의 log.txt 사진입니다

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

========DIJKSTRA========
Directed Graph Dijkstra
Start: 1
[0] 1 -> 3 -> 6 -> 8 -> 9 -> 0 (27)
[1] 1 (0)
[2] 1 -> 3 -> 6 -> 8 -> 9 -> 0 -> 2 (32)
[3] 1 -> 3 (3)
[4] 1 -> 4 (10)
[5] 1 -> 3 -> 6 -> 8 -> 9 -> 0 -> 2 -> 5 (39)
[6] 1 -> 3 -> 6 (4)
[7] 1 -> 4 -> 7 (14)
[8] 1 -> 3 -> 6 -> 8 (9)
[9] 1 -> 3 -> 6 -> 8 -> 9 (12)
========================

========DIJKSTRA========
Undirected Graph Dijkstra
Start: 0
[0] 0 (0)
[1] 0 -> 1 (2)
[2] 0 -> 2 (5)
[3] 0 -> 1 -> 3 (5)
[4] 0 -> 1 -> 3 -> 4 (6)
[5] 0 -> 2 -> 5 (12)
[6] 0 -> 1 -> 3 -> 6 (6)
[7] 0 -> 1 -> 3 -> 4 -> 7 (10)
[8] 0 -> 1 -> 3 -> 6 -> 8 (11)
[9] 0 -> 1 -> 3 -> 6 -> 8 -> 9 (14)
========================
```

```
========DIJKSTRA========
Directed Graph Dijkstra
Start: 1
[0] 1 -> 3 -> 6 -> 8 -> 9 -> 0 (27)
[1] 1 (0)
[2] 1 -> 3 -> 6 -> 8 -> 9 -> 0 -> 2 (32)
[3] 1 -> 3 (3)
[4] 1 -> 4 (10)
[5] 1 -> 3 -> 6 -> 8 -> 9 -> 0 -> 2 -> 5 (39)
[6] 1 -> 3 -> 6 (4)
[7] 1 -> 4 -> 7 (14)
[8] 1 -> 3 -> 6 -> 8 (9)
[9] 1 -> 3 -> 6 -> 8 -> 9 (12)
========================

========DIJKSTRA========
Undirected Graph Dijkstra
Start: 0
[0] 0 (0)
[1] 0 -> 1 (2)
[2] 0 -> 2 (5)
[3] 0 -> 1 -> 3 (5)
[4] 0 -> 1 -> 3 -> 4 (6)
[5] 0 -> 2 -> 5 (12)
[6] 0 -> 1 -> 3 -> 6 (6)
[7] 0 -> 1 -> 3 -> 4 -> 7 (10)
[8] 0 -> 1 -> 3 -> 6 -> 8 (11)
[9] 0 -> 1 -> 3 -> 6 -> 8 -> 9 (14)
========================
```

Directed와 Undirected 방향성 설정과, 시작 노드 설정이 제대로 됨을 확인했으며, 오른쪽 사진은 두번째 LOAD 이후 사진인데, 좌측과 우측 사진을 비교해보면 값이 같음을 통해서 값이 같음을 통해서 제대로 된 출력을 하고 있음을 확인 했습니다.

```
========DIJKSTRA========
Directed Graph Dijkstra
Start: 1
[0] 1 -> 3 -> 6 -> 8 -> 9 -> 0 (27)
[1] 1 (0)
[2] 1 -> 3 -> 6 -> 8 -> 9 -> 0 -> 2 (32)
[3] 1 -> 3 (3)
[4] 1 -> 4 (10)
[5] 1 -> 3 -> 6 -> 8 -> 9 -> 0 -> 2 -> 5 (39)
[6] 1 -> 3 -> 6 (4)
[7] 1 -> 4 -> 7 (14)
[8] 1 -> 3 -> 6 -> 8 (9)
[9] 1 -> 3 -> 6 -> 8 -> 9 (12)
[10] x
========================
```

위 사진은 아무런 연결도 없는 10번 노드를 만들어서 x의 값이 제대로 출력되는지 확인했습니다.

BELLMANFORD

command.txt	graph_L.txt	graph_M.txt

command.txt

```
LOAD graph_L.txt
BELLMANFORD O 4 2
BELLMANFORD X 4 2
LOAD graph_M.txt
BELLMANFORD O 4 1
BELLMANFORD X 4 1
EXIT
```

graph_L.txt

```
L
10
0
1 2
2 5
1
3 3
4 10
2
4 2
5 7
3
6 1
4
3 1
6 8
7 4
5
7 2
6
8 5
7
8 1
9 6
8
9 3
9
0 15
9
```

graph_M.txt

```
M
10
0 2 5 0 0 0 0 0 0 0
0 0 0 3 10 0 0 0 0 0
0 0 0 0 2 7 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 8 4 0 0
0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 0 0 5 0
0 0 0 0 0 0 0 0 1 6
0 0 0 0 0 0 0 0 0 3
15 0 0 0 0 0 0 0 0 0
```

각 파일들을 위처럼 설정하고 실행했을때의 log.txt 사진입니다

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

========BELLMANFORD========
Directed Graph Bellman-Ford
4 -> 7 -> 8 -> 9 -> 0 -> 2
Cost: 28
============================

========BELLMANFORD========
Undirected Graph Bellman-Ford
4 -> 2
Cost: 2
============================

========LOAD========
Success
====================

========BELLMANFORD========
Directed Graph Bellman-Ford
4 -> 7 -> 8 -> 9 -> 0 -> 2
Cost: 28
============================

========BELLMANFORD========
Undirected Graph Bellman-Ford
4 -> 2
Cost: 2
============================

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

제대로 값이 나옴을 확인했습니다.

이후 음의 사이클 판단을 제대로 하는지 확인하기 위해서 graph_L.txt 파일의 4번 노드에서 3번노드로 향하는 가중치를 -12로 설정해서 실행해봤습니다.

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

========BELLMANFORD========
Directed Graph Bellman-Ford
4 -> 3 -> 6 -> 8 -> 9 -> 0 -> 2
Cost: 17
============================

========ERROR========
700
====================

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

ERROR가 제대로 출력됨을 확인했습니다.

FLOYD

command.txt	graph_L.txt	graph_M.txt

command.txt

```
LOAD graph_L.txt
BELLMANFORD O 4 2
BELLMANFORD X 4 2
LOAD graph_M.txt
BELLMANFORD O 4 1
BELLMANFORD X 4 1
EXIT
```

graph_L.txt

```
L
10
0
1 2
2 5
1
3 3
4 10
2
4 2
5 7
3
6 1
4
3 1
6 8
7 4
5
7 2
6
8 5
7
8 1
9 6
8
9 3
9
0 15
9
```

graph_M.txt

```
M
10
0 2 5 0 0 0 0 0 0 0
0 0 0 3 10 0 0 0 0 0
0 0 0 0 2 7 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 8 4 0 0
0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 0 0 5 0
0 0 0 0 0 0 0 0 1 6
0 0 0 0 0 0 0 0 0 3
15 0 0 0 0 0 0 0 0 0
```

각 파일들을 위처럼 설정하고 실행했을때의 log.txt 사진입니다

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

========FLOYD========
Directed Graph Floyd
    [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[0] 0   2   5   5   7   12  6   11  11  14
[1] 27  0   32  3   10  39  4   14  9   12
[2] 25  27  0   3   2   7   4   6   7   10
[3] 24  26  29  0   31  36  1   35  6   9
[4] 23  25  28  1   0   35  2   4   5   8
[5] 21  23  26  26  28  0   27  2   3   6
[6] 23  25  28  28  30  35  0   34  5   8
[7] 19  21  24  24  26  31  25  0   1   4
[8] 18  20  23  23  25  30  24  29  0   3
[9] 15  17  20  20  22  27  21  26  26  0
====================

========FLOYD========
Undirected Graph Floyd
    [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[0] 0   2   5   5   6   12  6   10  11  14
[1] 2   0   6   3   4   10  4   8   9   12
[2] 5   6   0   3   2   7   4   6   7   10
[3] 5   3   3   0   1   7   1   5   6   9
[4] 6   4   2   1   0   6   2   4   5   8
[5] 12  10  7   7   6   0   8   2   3   6
[6] 6   4   4   1   2   8   0   6   5   8
[7] 10  8   6   5   4   2   6   0   1   4
[8] 11  9   7   6   5   3   5   1   0   3
[9] 14  12  10  9   8   6   8   4   3   0
====================
```

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

========FLOYD========
Directed Graph Floyd
    [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[0] 0   2   5   5   7   12  6   11  11  14
[1] 27  0   32  3   10  39  4   14  9   12
[2] 25  27  0   3   2   7   4   6   7   10
[3] 24  26  29  0   31  36  1   35  6   9
[4] 23  25  28  1   0   35  2   4   5   8
[5] 21  23  26  26  28  0   27  2   3   6
[6] 23  25  28  28  30  35  0   34  5   8
[7] 19  21  24  24  26  31  25  0   1   4
[8] 18  20  23  23  25  30  24  29  0   3
[9] 15  17  20  20  22  27  21  26  26  0
====================

========FLOYD========
Undirected Graph Floyd
    [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[0] 0   2   5   5   6   12  6   10  11  14
[1] 2   0   6   3   4   10  4   8   9   12
[2] 5   6   0   3   2   7   4   6   7   10
[3] 5   3   3   0   1   7   1   5   6   9
[4] 6   4   2   1   0   6   2   4   5   8
[5] 12  10  7   7   6   0   8   2   3   6
[6] 6   4   4   1   2   8   0   6   5   8
[7] 10  8   6   5   4   2   6   0   1   4
[8] 11  9   7   6   5   3   5   1   0   3
[9] 14  12  10  9   8   6   8   4   3   0
====================
```

값이 모두 제대로 출력됨을 확인했으며, 두 사진의 값이 같은걸로 확인해서, FLOYD 알고리즘 구현 부분의 오류는 없다고 판단하였습니다.

CENTRALITY

command.txt

```
LOAD graph_L.txt
CENTRALITY
LOAD graph_M.txt
CENTRALITY
EXIT
```

graph_L.txt

```
L
10
0
1 2
2 5
1
3 3
4 10
2
4 2
5 7
3
6 1
4
3 1
6 8
7 4
5
7 2
6
8 5
7
8 1
9 6
8
9 3
9
0 15
9
```

graph_M.txt

```
M
10
0 2 5 0 0 0 0 0 0 0
0 0 0 3 10 0 0 0 0 0
0 0 0 0 2 7 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 8 4 0 0
0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 0 0 5 0
0 0 0 0 0 0 0 0 1 6
0 0 0 0 0 0 0 0 0 3
15 0 0 0 0 0 0 0 0 0
```

각 파일들을 위처럼 설정하고 실행했을때의 log.txt 사진입니다

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

========CENTRALITY========
[0] 9/71
[1] 9/58
[2] 9/50
[3] 9/40
[4] 9/38 <- Most Central
[5] 9/61
[6] 9/44
[7] 9/46
[8] 9/50
[9] 9/74
==========================
```

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

========CENTRALITY========
[0] 9/71
[1] 9/58
[2] 9/50
[3] 9/40
[4] 9/38 <- Most Central
[5] 9/61
[6] 9/44
[7] 9/46
[8] 9/50
[9] 9/74
==========================
```

분수 형태로 제대로 출력 됨을 확인했으며, 최대값의 중심성에 "<- Most Central 이 출력되는것 까지 확인했습니다.

5. Consideration

본 프로젝트는 graph_L.txt, graph_M.txt 둘 중 하나의 파일을 LOAD 명령어를 이용해서 인접 리스트 형태 혹은 인접 행렬 형태로 graph를 불러온 뒤, DFS, BFS를 이용해서 그래프를 탐색하거나, Dijkstra, Bellman-ford, Centrality, Kruskal, Floyd 알고리즘을 구현해서, 최소 신장 트리를 구축하거나, 특정 노드에서 특정 노드까지의 최소 거리, 정점의 중요성 을 파악할 수 있도록 하는 프로그램을 만드는 프로젝트였다.

프로그램을 구성하면서 있었던 어려움으로는 처음에 파일 입출력 구성을 잘못 해서, 명령어를 구분해주는 "======"가 DFS와 BFS를 사용하는 경우에 이전 명령어의 끝을 구분해주는 부분을 DFS와 BFS 시행 이후에 출력되는 문제가 있었어서, 이를 해결하기 위해서 명령어 안에서 파일을 열고 닫고 수행을 하도록 수정했다.

그리고 이번 프로젝트에서는 STL 컨테이너를 많이 사용했는데, 특히 map 컨테이너를 사용을 이전에 많이 해본적이 없는데, 이번 프로젝트에서 구현한 알고리즘을 구성하는데, map 컨테이너가 효율적이라고 교수님께서도 수업에서 알려 주셨기 때문에, 프로그램을 구현하는데, map 컨테이너를 사용하려고 노력했다.

이번 프로젝트에는 구현해야하는 알고리즘이 많다보니, 본 프로젝트를 구성하면서 각 알고리즘의 특성, 예를 들어서 어떤 알고리즘이 음수 가중치를 허용하는지, Floyd은 어떤 값을 기대하고 사용하는지, 숙지하지 못하고 있는 경우가 많았는데, 프로그램을 구현하면서, 구현하기 이전에 알고리즘의 작동 원리를 공부하면서, 알고리즘들의 작동원리와 , 예외처리 사항에 대해서 공부 할 수 있었다.