데이터 구조 설계, 실습 3차 프로젝트

보고서

2021202085 전한아솔

1. Introduction-프로젝트 소개

이번 프로젝트는 그래프를 이용하여 그래프 연산 프로그램을 구현하는 프로젝트이다. 그래프의 정보가 저장된 테스트 파일을 통해 그래프를 구현하고 연산을 수행하면 된다.

그래프는 데이터의 형태에 따라서 List와 Matrix 그래프로 저장한다.

<List 형태 – graph_L.txt>

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

List 형태의 그래프는 첫 번째 줄에 그래프의 정보인 L, 두 번째 줄에는 그래프의 크기가 저장되어 있다.

해당 파일은 시작 vertex만 존재하는 1번 형식과 도착 vertex weight로 이루어진 2번 형식으로 구성된다.

제안서에 있는 텍스트 파일로 설명을 한다면,

0

1 6

2 2

의 부분은 0번 vertex가 1번 vertex와 6의 가중치를 가진 간선으로 연결, 2번 vertex가 2의 가중치를 가진 간선으로 연결되어 있다고 생각할 수 있다.

<Matrix 형태 – graph_M.txt>

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

List 형태와 마찬가지로 그래프의 정보인 M, 그래프의 크기가 1, 2번째 줄에 명시되어 있고 행렬의 형태로 데이터가 저장되어 있다.

N행 m열의 값이 k라면, n번 vertex가 k의 가중치를 가진 간선으로 m번 vertex와 연결되어 있다고 생각할 수 있다.

List와 Matrix 형태 모두, 방향과 가중치를 가지는 그래프를 저장하고 있는 형태이므로 각 명령어에 맞추어 그래프의 방향, 가중치를 고려하여 연산을 수행할 수 있고 List와 Matrix의 형태 모두 연산이 가능하도록 일반화하여 프로그램을 구현해야 한다.

<LOAD 명령어>

프로그램에 그래프의 정보를 불러오는 명령어로, 위에서 설명한 2가지 텍스트 파일 중 하나를 인자로 선택하여 그래프를 구성한다. 기존 그래프의 정보가 존재하는 상태에서 LOAD 명령어가 수행되면 기존의 그래프 정보는 삭제하고 새로운 그래프를 생성한다.

<PRINT 명령어>

그래프의 상태를 출력하는 명령어로, List 형태의 그래프는 adjacency list, Matrix 형태의 그래프는 adjacency matrix를 출력한다.

이때, vertex는 오름차순으로, edge는 vertex를 기준으로 오름차순으로 출력한다.

<BFS 명령어>

방향성과 가중치가 없는 그래프에서 수행 가능하므로, 방향성과 가중치가 없다고 가정하고 인자로 전달된 vertex를 기준으로 너비 우선 탐색을 수행하는 명령어이다.

모든 vertex를 Queue를 사용하는 너비 우선 탐색 알고리즘을 사용하여 방문하고, 방문 순서를 명령어의 결과로 출력한다.

<DFS, DFS_R 명령어>

BFS와 마찬가지로 방향성과 가중치가 없다고 가정하고 인자로 전달된 vertex를 기준으로 깊이 우선 탐색을 수행하는 명령어이다.

DFS 명령어는 stack을, DFS_R 명령어는 재귀적 호출 방법을 사용하여 모든 vertex를 깊이 우선 탐색 알고리즘을 사용하여 방문하고, 방문 순서를 명령어의 결과로 출력한다.

<KRUSKAL 명령어>

KRUSKAL 명령어는 방향성이 없고 가중치가 존재하는 그래프에서 수행 가능하다. Kruskal 알고리즘을 사용하여 그래프에 대한 MST(minimum spanning tree, 최소 신장 트리)를 구하고, MST를 구성하는 edge의 weight 값을 vertex의 오름차순으로 출력하고 weight의 총합을 출력한다. 이때, MST를 구하는 과정에서 sub-tree가 사이클을 이루는지에 대한 검사도 수행한다.

Kruskal 알고리즘에서 edge를 정렬할 때 연산의 효율을 향상하기 위해서 segment size에 따른 정렬 알고리즘을 구현한다.

이번 프로젝트에서는 퀵 정렬을 수행하며 정렬을 수행하는 과정에서 segment size를 재귀적으로 분할할 때, segment size가 6 이하인 경우에 삽입 정렬을 수행하도록 하는 정렬 함수를 따로 정의하여 사용한다.

<DIJKSTRA 명령어>

DIJKSTRA 명령어는 방향성과 가중치가 있는 그래프에서 수행 가능하다. 인자로 전달된 vertex를 기준으로 Dijkstra 알고리즘을 수행하여 모든 vertex에 대한 최단 경로를 출력한다. 경로는 기준 vertex(인자로 전달된 vertex)에서 해당 vertex까지의 경로로 출력하며 해당 경로의 cost도 같이 출력한다.

기준 vertex에서 도달할 수 없는 경우에는 x를 대신 출력한다.

<BELMANFORD 명령어>

BELMANFORD 명령어는 방향성과 가중치가 있는 그래프에서 수행 가능하며 weight가 음수인 경우에도 정상 동작한다.

인자로 전달되는 2개의 vertex (시작 vertex, 도착 vertex)에서 시작 vertex를 기준으로 Bellman-Ford 알고리즘을 수행하여 도착 vertex까지의 최단 경로와 cost를 출력한다. 이때, 도달할 수 없는 경우라면 x를 출력한다.

Weight의 값이 음수가 가능하므로 음수 사이클이 발생할 수 있다. 따라서 음수 사이클이 발생한 경우에는 에러 코드를 출력한다.

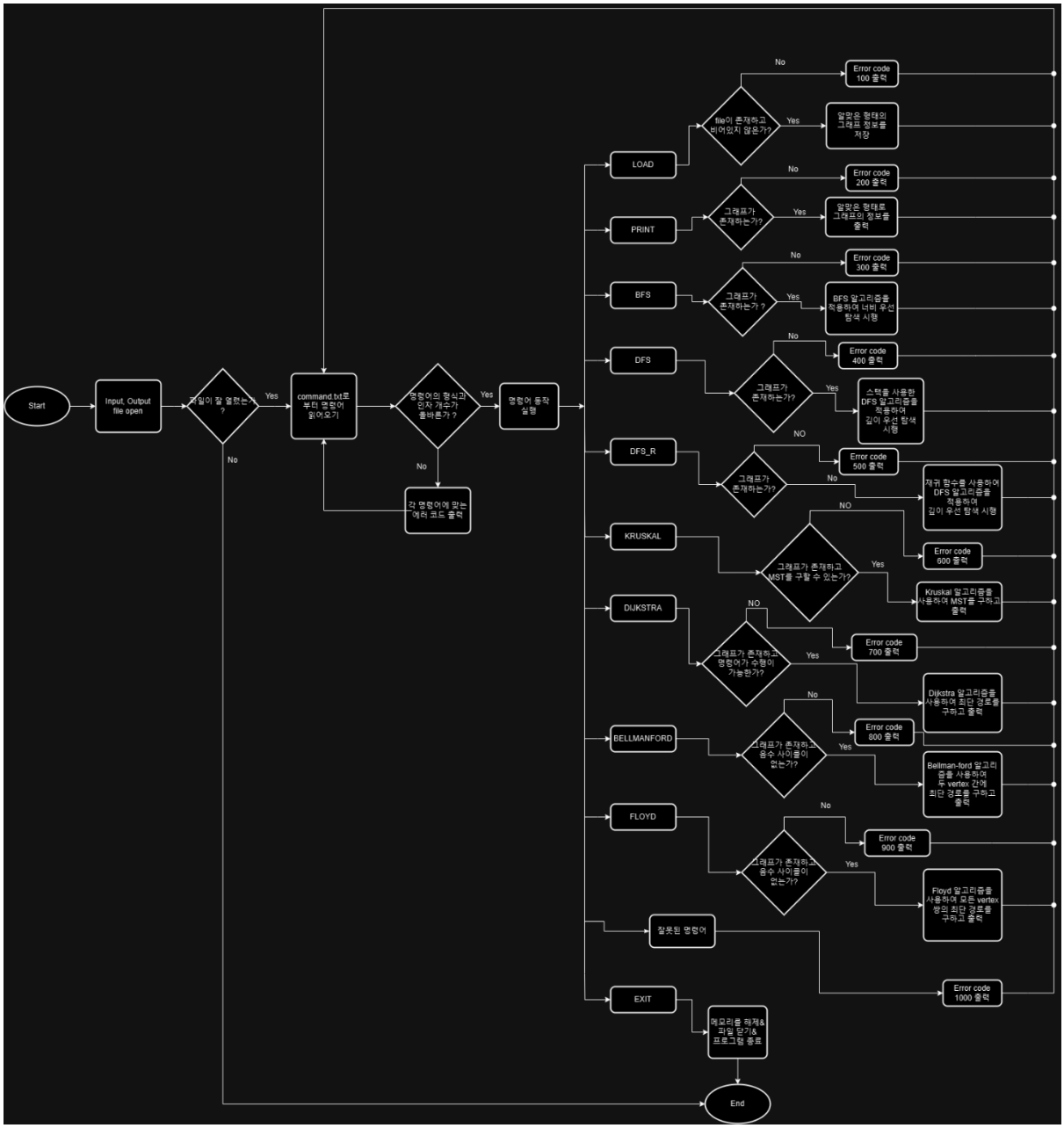
<FLOYD 명령어>

FLOYD 명령어는 방향성과 가중치가 있는 그래프에서 수행 가능하며 weight가 음수인 경우에도 정상 동작한다.

인자가 존재하지 않으며 모든 vertex 쌍에 대해서 시작 vertex에서 도착 vertex로 가는데 필요한 cost의 최솟값을 행렬의 형태로 출력한다.

도달할 수 없는 경로는 x를 출력하며 Bellman-Ford 명령어와 마찬가지로 음수 사이클이 발생한 경우에는 에러 코드를 출력한다.

2. Flowchart-프로젝트의 전반적인 흐름과 동작을 설명



파일이 제대로 열리지 않았다면 프로그램은 종료된다.

Command.txt 로부터 한 줄씩 명령어를 읽어와, 해당 명령어를 수행한다. 해당 명령어의 인자 개수가 맞지 않거나 형식이 맞지 않는다면 알맞은 에러코드를 출력하고 다음 명령어를 실행한다. 특히 그래프의 정보가 없다면 명령어를 수행할 수 없으므로 에러를 출력하도록 구현했다.

3. Algorithm – 프로젝트에서 사용한 알고리즘의 세부적인 아이디어를 설명

<BFS>

BFS 는 너비 우선 탐색 알고리즘으로, 시작 정점으로부터 가까운 정점을 먼저 방문하는 방식으로 동작하는 완전 탐색 알고리즘이다.

BFS 는 Queue 를 사용하여 동작하므로 FIFO 의 원칙을 가지고 탐색을 진행한다.

탐색을 시작할 정점을 큐에 삽입하고 해당 정점의 번호를 방문했다는 의미로 bool 형 배열을 true 로 바꾸고 시작한다. While 문을 통해 queue 가 빌 때까지 동작하며 큐에서 front 에 존재하는 정점을 꺼내고 pop 연산을 통해 큐에서 제거한다. 해당 정점과 연결되어 있는 정점 중, 방문하지 않는 정점이 존재한다면 큐에 push 하고 해당 정점의 방문을 체크한다.

<DFS>

DFS는 깊이 우선 탐색 알고리즘으로, 시작 정점으로부터 한 방향으로 끝까지 탐색하고, 더 이상 탐색할 곳이 없다면 다른 탐색을 진행하는 방식으로 동작하는 완전 탐색 알고리즘이다.
DFS는 재귀 함수로 동작하거나 stack을 사용하여 동작한다.

재귀 함수로 동작하는 코드는 시작 정점에 방문했음을 체크하고 정점과 연결된 다른 정점을 탐색한다. 이때, 방문하지 않았다면 함수의 인자로 해당 정점을 전달하여 다시 함수를 호출하는 방식으로 동작한다.

스택을 사용하여 동작하는 코드는 BFS와 비슷하게 작성하는데 큐 대신 스택을 사용한다. 시작 정점을 스택에 저장 후, while문을 통해서 스택이 빌 때까지 동작하며 스택에서 top에 존재하는 정점을 꺼내고 제거한다. 해당

정점이 이미 방문 된 정점이라면 continue를 이용하여 다음 반복을 수행하고 아니라면 해당 노드와 연결된 노드를 스택에 push한다. 스택은 LIFO의 구조이므로, 예를 들어서 0번 정점이 1, 2번 정점과 연결되어 있다면 스택에는 1, 2번의 순서로 저장되므로 다시 꺼낼 땐 2, 1번의 순서로 탐색이 수행된다. 따라서 이번 프로젝트의 출력 양식을 맞추기 위해서 reverse_iterator를 사용하여 역순으로 정점을 스택에 push하여, 탐색이 수행되는 정점은 원래의 순서로 출력할 수 있도록 했다.

<KRUSKAL>

Kruskal 알고리즘은 greedy 알고리즘을 사용하여 최소 신장 트리(MST, Minimum Spanning Tree)를 찾기 위한 알고리즘이다.

greedy 알고리즘은 현재의 최적해를 선택하여 전체의 최적해를 만들어내는 알고리즘이다. Kruskal 알고리즘에서는 매 단계에서 가장 비용이 적은 간선을 선택하는 방식으로 사용한다.

먼저, 가중치의 오름차순으로 간선들을 정렬한다. 이때, 프로젝트 제안서에 명시되어 있는 방식으로 segment size가 6 이하인 경우는 삽입 정렬을 수행하고 segment size가 7 이상인 경우에 분할하도록 구현한 정렬 알고리즘을 사용하여 정렬했다.

가중치가 가장 낮은 간선부터 순서대로 집합에 추가하는데, find 연산을 사용하여 해당 간선을 추가해도 사이클이 생기지 않는지 판단했고, Union 연산을 사용하여 두 집합을 하나의 집합으로 만들어서 MST를 구성하도록 했다.

Find 연산은 주어진 원소가 속한 집합의 루트를 반환하는 연산인데, 같은 집합에 속한 원소는 동일한 루트를 가지므로 두 정점의 find 연산 값이 다르다면 사이클이 생기지 않았음을 의미하므로 간선을 추가한다.

Union 연산은 두 집합을 하나의 집합으로 합치는 연산인데, 두 집합의 루

트 중 한쪽의 루트를 다른 쪽 루트의 자식으로 연결하여 합친다.

Union과 find 연산을 사용하여 Kruskal 알고리즘에서 사이클을 형성하지 않으면서 가중치가 제일 작은 간선을 추가하여 MST를 완성할 수 있었다.

<DIJKSTRA>

Dijkstra 알고리즘은 greedy 알고리즘을 하나의 출발 정점에서 최단 경로를 찾는 알고리즘이다. 해당 알고리즘은 음의 가중치를 가지는 간선이 없는 그래프에서 동작할 수 있다.

시작 정점은 자기 자신이므로 경로를 0으로 설정하고 다른 모든 정점과의 경로는 무한대로 초기화한다.
아직 방문하지 않은 정점 중에서 현재까지의 최단 경로 비용이 가장 작은 정점을 선택하는 방식으로 다음 경로를 찾는데, 기존 경로의 비용보다 작은 경우에는 더 작은 값으로 설정한다.

시작 정점부터 w까지 가는 경로의 비용을 dist[w] 라고하고 u부터 w까지의 가중치를 length<u, w>라고 한다면
Dist[w] = min(dist[w], dist[u]+length<u, w>)이다. 즉, w까지 바로 갈 수 있는 경로의 비용과 u를 거쳐서 w까지 가는 새로운 경로가 생성되면 해당 비용을 비교하여 더 작은 비용을 가지는 경로를 택하여 기존 경로에 더하는 방식으로 구현한다.

<BELLMANFORD>

Bellman-ford 알고리즘은 두 정점 간에 최단 경로를 찾는 알고리즘으로 Dijkstra 알고리즘과 다르게 음수를 포함하는 가중치에도 동작할 수 있다는 특징이 존재한다.

시작 정점과의 거리는 0, 나머지 모든 정점과의 거리는 무한대로 초기화한다. D(v, k)를 시작 정점부터 v 정점까지 k개의 edge를 사용하여 도달할 수

있는 경로라고 하자. 그렇다면 d(v, k)는 k-1개의 edge만 사용하여 도달하는 경우인 d(v, k-1)과 w의 정점에 k-1개의 edge를 사용하고 1개의 edge를 더 사용하여 총 k개의 edge로 v에 도달하는 경우인 d(w, k-1)+length(w, v) 가 존재한다. 두 경로의 비용 중 더 작은 값을 선택해야 하므로 d(v, k) = min(d(v, k-1), min(d(w, k-1) + length(w, v)))이다.

W 정점에서 v까지 1개의 edge를 사용해서 도달하는 경우 중에서도 최소 비용을 가지는 경로를 선택해서 d(v, k-1)과 비교해야 하므로 min(d(w, k-1) + length(w, v))로 계산해야 한다. 해당 방법을 e(정점 개수)-1번 반복하여 최단 경로를 구할 수 있다.

간선의 가중치 중 음수가 존재하므로 음수 사이클이 존재할 수 있다. 정상적으로 음수 사이클이 존재하지 않는 그래프라면 최단 경로가 바뀌지 않을 것이다. 하지만 음수 사이클이 존재한다면 한 번 더 반복했을 때 경로가 더 작아질 것이다. 따라서 e-1번의 반복이 끝난 후, 한번의 반복을 더 시행했을 때, 경로가 바뀐다면 음수 사이클이 존재함을 의미하므로 에러를 출력한다.

<FLOYD>

Floyd 알고리즘은 DP(동적 계획법, Dynamic Programming)를 사용하여 그래프의 모든 정점 간의 최단 경로를 구하는 알고리즘이다. Bellman-ford 알고리즘과 마찬가지로 음수의 가중치가 존재해도 동작한다.

동적 계획법은 전체 문제를 작은 하위 문제로 나누어 해결하는 알고리즘으로, 중복되는 하위 문제를 재사용하여 해답을 찾는 방식이다.

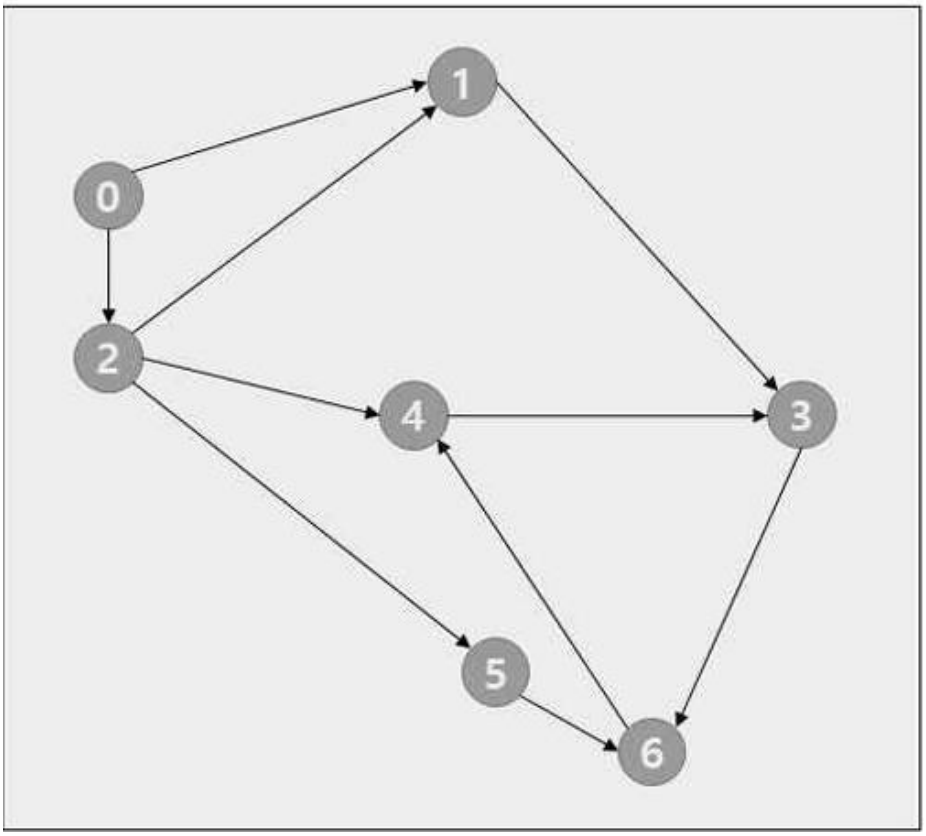
Dist[i][j]를 i에서 j로 가는 경로의 비용이라고 하자. Dist[i][i]는 자신과의 경로이므로 0으로 초기화하고 i에서 j로 가는 경로는 i에서 j로 직접 가는 경로, i에서 k를 거쳐 k에서 j로 가는 경로, 총 2가지 경로가 존재한다. 따라서 dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])의 점화식을 도출할 수 있고, k는 0부터 e-1까지 순회하면서 모든 i와 j쌍의 최단 경로를 구할 수 있

다.

음수 사이클이 존재하는 경우에, 음수 사이클을 거쳐 자기 자신으로 돌아오는 경로가 더 줄어들 수 있다. 따라서 모든 경로를 구한 뒤, 자기 자신과의 거리가 0이 아니라면 음수 사이클이 존재함을 의미하므로 에러를 출력한다.

4. Result – 모든 명령어의 결과화면과 예외 처리의 동작을 설명

<LOAD & PRINT>



제안서에 주어진 그래프를 List, Matrix의 형태로 txt파일에 저장하여 그래프 객체를 생성함을 보이겠다.

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

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

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

Graph_L.txt에 저장된 그래프의 정보를 Adjacency List의 형태로 저장하고 출력한 결과이다. 제안서의 그림처럼 각 정점과 가중치가 잘 연결됨을 볼 수 있다.

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

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

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

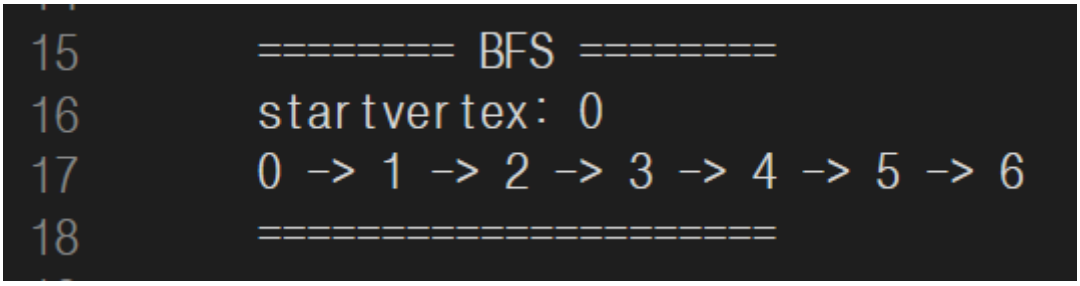
Graph_M.txt의 저장된 그래프의 정보를 Matrix의 형태로 저장하고 출력한 결과, txt 파일과 같은 형태로 저장됨을 알 수 있다.

```
LOADgraph_T.txt
PRINT
BFS    0
DFS    0
DFS_R        0
KRUSKAL
DIJKSTRA    0
BELLMANFORD    0        5
FLOYD
EXIT
```

```
======== ERROR ========
100
======================

======== ERROR ========
200
======================

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

======== ERROR ========
400
======================

======== ERROR ========
500
======================

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

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

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

======== ERROR ========
900
======================

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

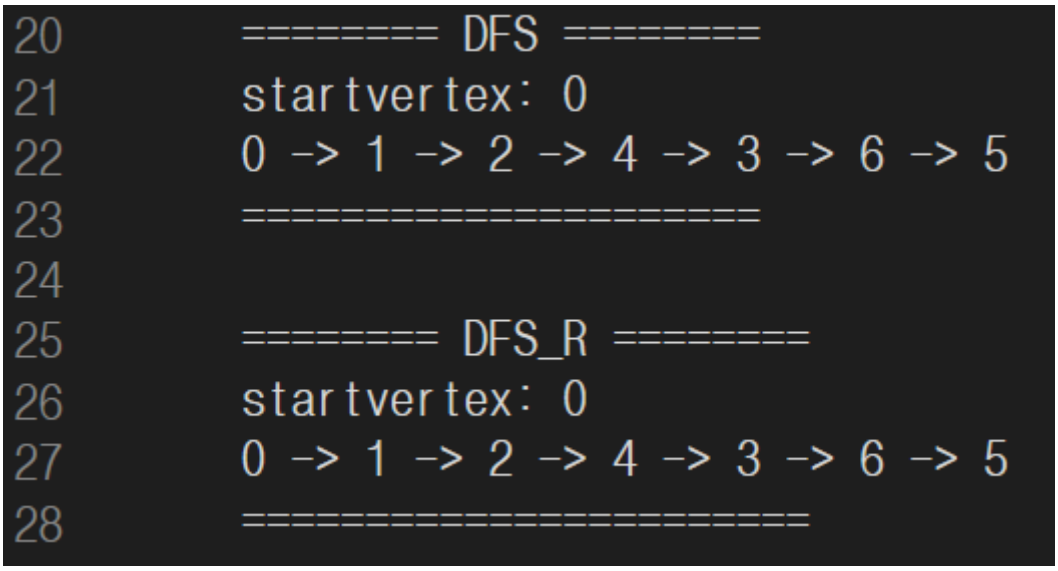
이렇게 존재하지 않는 텍스트 파일을 LOAD하면 그래프에 저장된 정보가 없으므로 모든 명령어에서 에러를 출력하도록 구현했다.

<BFS>



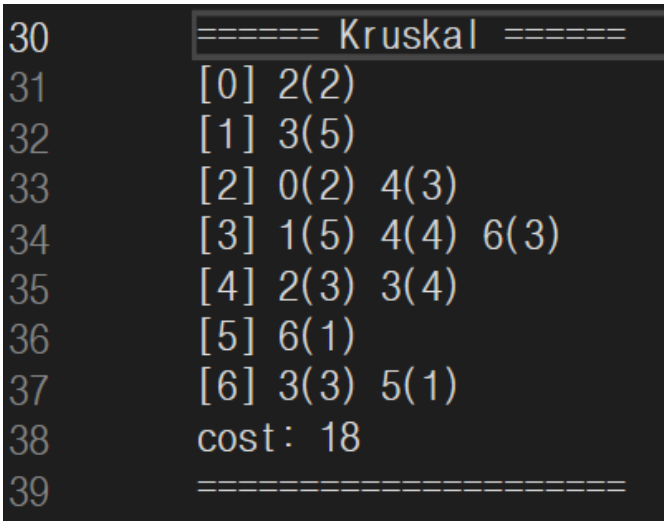
0번 정점부터 시작하여 BFS 알고리즘을 수행한 결과이다.
방향성이 없다고 가정하지만 양방향 연결처럼 생각하고 수행하므로 0 -> 1 -> 3 -> 6 -> 4 -> 2 -> 5와 같은 순서가 아니라 사진과 같은 순서로 탐색된다.

<DFS & DFS_R>



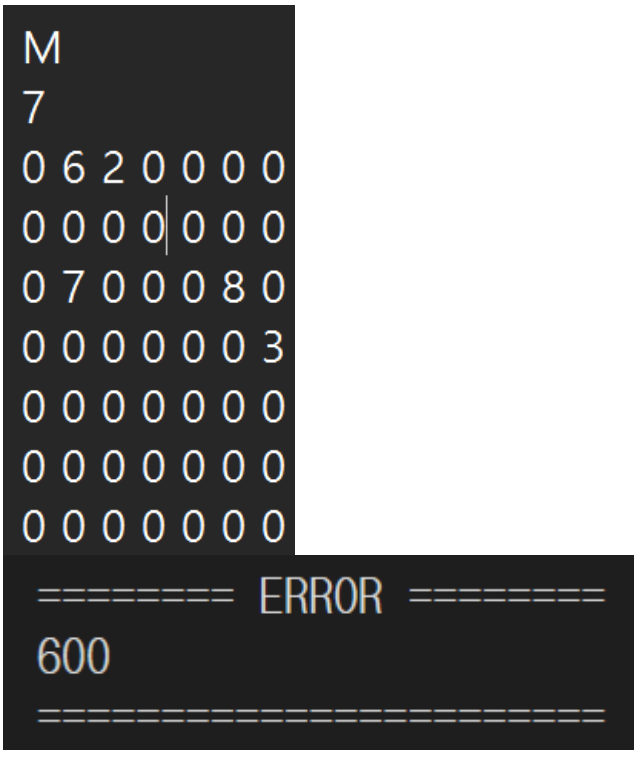
0번 정점부터 시작하여 DFS 알고리즘을 수행한 결과이다.
BFS 알고리즘과 마찬가지로 방향이 없다고 가정하므로 양방향 연결처럼 생각하고 수행한다. 따라서 0 -> 2 -> 5 -> 6 -> 4 -> 3 -> 1와 같은 순서가 아니라 사진과 같은 순서로 탐색된다.

<KRUSKAL>



제안서의 명시된 그래프에서 MST를 구한 결과이다.

MST에서 각 정점과 연결된 정점을 정점 번호 오름차순으로 정렬하여 출력했다. 또한 MST를 구성하는 총 비용도 출력하도록 했다.



해당 그래프에서는 MST를 구할 수 없으므로 에러를 출력하는 결과를 볼 수 있다.

<DIJKSTRA>

```
41          ====== Dijkstra ======
42          startvertex :0
43          [1] 0 -> 1 (6)
44          [2] 0 -> 2 (2)
45          [3] 0 -> 2 -> 4 -> 3 (9)
46          [4] 0 -> 2 -> 4 (5)
47          [5] 0 -> 2 -> 5 (10)
48          [6] 0 -> 2 -> 5 -> 6 (11)
49          ======================
```

시작 정점인 0번 정점부터 자신을 제외하고 도달할 수 있는 정점에 최단 경로를 역순으로 출력한 결과이다.

```
M
7
0 6 2 0 0 0 0
0 0 0 0 0 0 0
0 7 0 0 0 8 0
0 0 0 0 0 0 3
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
 ====== Dijkstra ======
 startvertex :0
 [1] 0 -> 1 (6)
 [2] 0 -> 2 (2)
 [3] x
 [4] x
 [5] 0 -> 2 -> 5 (10)
 [6] x
 ======================
```

도달할 수 없는 정점은 x로 표시하도록 예외 처리를 진행했다.

<Bellman-ford>

```
====== Bellman-Ford ======
0 -> 2 -> 5 -> 6
cost: 11
==========================
```

0번 정점에서 6번 정점까지의 최단 경로를 출력하고 해당 경로의 비용도 출력하는 결과를 볼 수 있다.

```
====== Bellman-Ford ======
Same vertex
cost: 0
==========================
```

만약 두 정점이 동일하다면 같은 정점임을 알려주고 cost는 0으로 출력하도록 구현했다.

```
====== Bellman-Ford ======
x
==========================
```

두 정점 간에 이어지는 경로가 없어서 도달할 수 없다면 X를 출력하도록 구현했다.

<FLOYD>

```
======== FLOYD ========
     [0] [1] [2] [3] [4] [5] [6]
[0]  0   6   2   9   5   10  11
[1]  x   0   x   5   18  x   8
[2]  x   7   0   7   3   8   9
[3]  x   x   x   0   13  x   3
[4]  x   x   x   4   0   x   7
[5]  x   x   x   15  11  0   1
[6]  x   x   x   14  10  x   0
======================
```

FLOYD 알고리즘은 방향성이 존재한다고 가정하고 수행하므로 도달할 수 있는 정점은 최단 거리의 비용을 출력하고 도달할 수 없다면 X 표시를 나타낸 결과를 볼 수 있다.

<EXIT>

```
kw2021202085@ubuntu:~/Downloads/DS_Project_3(1)/DS_Project_3$ ./run
kw2021202085@ubuntu:~/Downloads/DS_Project_3(1)/DS_Project_3$ valgrind ./run
==9090== Memcheck, a memory error detector
==9090== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==9090== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==9090== Command: ./run
==9090==
==9090==
==9090== HEAP SUMMARY:
==9090==     in use at exit: 0 bytes in 0 blocks
==9090==   total heap usage: 573 allocs, 573 frees, 2,264,043 bytes allocated
==9090==
==9090== All heap blocks were freed -- no leaks are possible
==9090==
==9090== For lists of detected and suppressed errors, rerun with: -s
==9090== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
kw2021202085@ubuntu:~/Downloads/DS_Project_3(1)/DS_Project_3$
```

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

Graph에 객체로 할당했던 메모리와 명령어의 동작을 수행하는 과정에서 할당했던 메모리를 모두 해제하여 메모리 누수가 일어나지 않도록 하고 프로그램을 종료한 결과이다.

<고찰>

메모리 누수가 발생하여 GetMethod 클래스에서 할당한 메모리의 delete 를 모두 확인했으나 발견하지 못했다. 그래프 형태에 따라서 다른 객체를 동적으로 할당하는데, 두 객체 모두 graph 클래스를 상속하므로 graph 클래스의 소멸자를 virtual로 선언하여 그래프 형태에 맞는 소멸자가 호출되도록 구현하여 메모리 누수를 방지했다.