

# DS Project 보고서

2021202054 송예준

## 1. Introduction

### - 요약

이번 3차 프로젝트에서는 그래프를 이용해 그래프 연산 프로그램을 구현한다. 이 프로그램은 그래프 정보가 저장된 텍스트 파일을 통해 그래프를 구현하고, 그래프의 특성에 따라 BFS, DFS, Kruskal, Dijkstra, Bellman-Ford 그리고 FLOYD 연산을 수행한다. 그래프 데이터는 방향성(Direction)과 가중치(Weight)를 모두 가지고 있으며, 데이터 형태에 따라 List 그래프와 Matrix 그래프로 저장한다. BFS와 DFS는 그래프의 방향성과 가중치를 고려하지 않고, 그래프 순회 또는 탐색 방법을 수행한다. Kruskal 알고리즘은 최소 비용 신장 트리(MST)를 만드는 방법으로 방향성이 없고, 가중치가 있는 그래프 환경에서 수행한다. Dijkstra 알고리즘은 정점 하나를 출발점으로 두고 다른 모든 정점을 도착점으로 하는 최단경로 알고리즘으로 방향성과 가중치 모두 존재하는 그래프 환경에서 연산을 수행한다. 만약 weight가 음수일 경우 Dijkstra는 에러를 출력하며, Bellman-Ford에서는 음수 사이클이 발생한 경우 에러, 음수 사이클이 발생하지 않았을 경우 최단 경로와 거리를 구한다. FLOYD에서는 음수 사이클이 발생한 경우 에러, 음수 사이클이 발생하지 않았을 경우 최단 경로 행렬을 구한다. 프로그램의 동작은 명령어 파일에서 요청하는 명령에 따라 각각의 기능을 수행하고, 그 결과를 출력 파일(log.txt)에 저장한다. 각 그래프 연산들은 'GraphMethod' 헤더 파일에 일반 함수로 존재하며, 그래프 형식(List, Matrix)에 상관없이 그래프 데이터를 입력 받으면 동일한 동작을 수행하도록 일반화하여 구현한다. 또한 충분히 큰 그래프에서도 모든 연산을 정상적으로 수행할 수 있도록 구현한다.

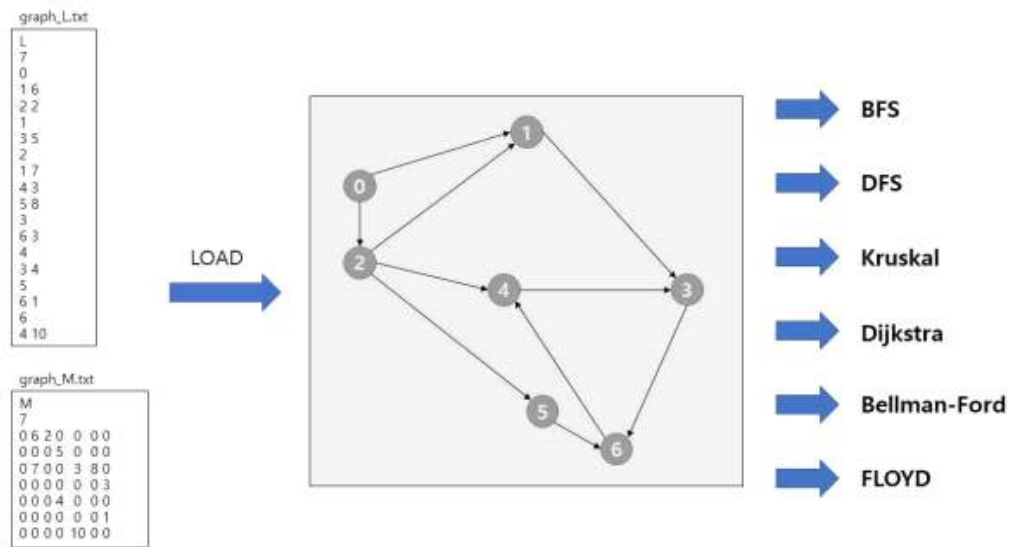


그림 1. 프로그램 구성

## ※ 구현할 명령어 종류 및 설명

### ① LOAD

- 프로그램에 그래프 정보를 불러오는 명령어로, 그래프 정보가 저장되어 있는 텍스트 파일 (graph\_L.txt 또는 graph\_M.txt)을 읽어 그래프를 구성한다.
- 텍스트 파일이 존재하지 않거나, 그래프가 이미 구성되어 있는 경우, 출력 파일 (log.txt)에 오류 코드(100)를 출력한다.
- 기존에 그래프 정보가 존재할 때 LOAD가 입력되면 기존 그래프 정보는 삭제하고 새로 그래프를 생성한다.

### ② PRINT

- 그래프의 상태를 출력하는 명령어로, List형 그래프일 경우 adjacency list를, Matrix형 그래프일 경우 adjacency matrix를 출력한다.
- vertex는 오름차순으로 출력한다. edge는 vertex를 기준으로 오름차순으로 출력한다.
- 만약 그래프 정보가 존재하지 않을 경우, 출력 파일(log.txt)에 알맞은 오류 코드를 출력한다.(200)

### ③ BFS

- 현재 그래프에서 입력한 vertex를 기준으로 BFS를 수행하는 명령어로, BFS 결과를 출력 파일(log.txt)에 출력한다.
- BFS 명령어를 수행하면서 방문하는 vertex의 순서로 결과를 출력한다.
- 입력한 vertex가 그래프에 존재하지 않거나, vertex를 입력하지 않은 경우, 출력 파일(log.txt)에 알맞은 오류 코드를 출력한다.(300)

#### ④ DFS

- 현재 그래프에서 입력한 vertex를 기준으로 DFS를 수행하는 명령어로, DFS 결과를 출력 파일(log.txt)에 출력한다.
- DFS 명령어를 수행하면서 방문하는 vertex의 순서로 결과를 출력한다.
- 입력한 vertex가 그래프에 존재하지 않거나, vertex를 입력하지 않은 경우, 출력 파일(log.txt)에 알맞은 오류 코드를 출력한다.(400)

#### ⑤ DFS\_R

- 현재 그래프에서 입력한 vertex를 기준으로 DFS\_R 수행하는 명령어로, DFS\_R 결과를 출력 파일(log.txt)에 출력한다.
- DFS\_R 명령어를 수행하면서 방문하는 vertex의 순서로 결과를 출력한다.
- 입력한 vertex가 그래프에 존재하지 않거나, vertex를 입력하지 않은 경우, 출력 파일(log.txt)에 알맞은 오류 코드를 출력한다.(500)

#### ⑥ KRUSKAL

- 현재 그래프의 MST를 구하고, MST를 구성하는 edge들의 weight 값을 오름차순으로 출력하고 weight의 총합을 출력 파일(log.txt)에 출력한다.
- 입력한 그래프가 MST를 구할 수 없는 경우, 명령어를 수행할 수 없는 경우, 출력 파일(log.txt)에 알맞은 오류 코드를 출력한다.(600)

#### ⑦ DIJKSTRA

- 명령어의 결과인 shortest path와 cost를 출력 파일(log.txt)에 출력한다. 출력 시 vertex, shortest path, cost 순서로 출력하며, shortest path는 해당 vertex에서 기준 vertex까지의 경로를 역순으로 출력한다.
- 기준 vertex에서 도달할 수 없는 vertex의 경우 'x'를 출력한다.
- 입력한 vertex가 그래프에 존재하지 않거나, vertex를 입력하지 않은 경우, 명령어를

수행할 수 없는 경우, 출력 파일(log.txt)에 알맞은 오류 코드를 출력한다.(700)

## ⑧ BELLMANFORD

- StartVertex를 기준으로 Bellman-Ford를 수행하여 EndVertex까지의 최단 경로와 거리를 구하는 명령어로, Bellman-Ford 결과를 출력 파일 (log.txt)에 저장한다.
- 음수인 weight가 있는 경우에도 동작해야한다.
- StartVertex에서 EndVertex로 도달할 수 없는 경우 'x'를 출력한다.
- 입력한 vertex가 그래프에 존재하지 않거나, 입력한 vertex가 부족한 경우, 명령어를 수행할 수 없는 경우, 음수 사이클이 발생한 경우 출력 파일 (log.txt)에 알맞은 오류 코드를 출력한다. (800)

## ⑨ FLOYD

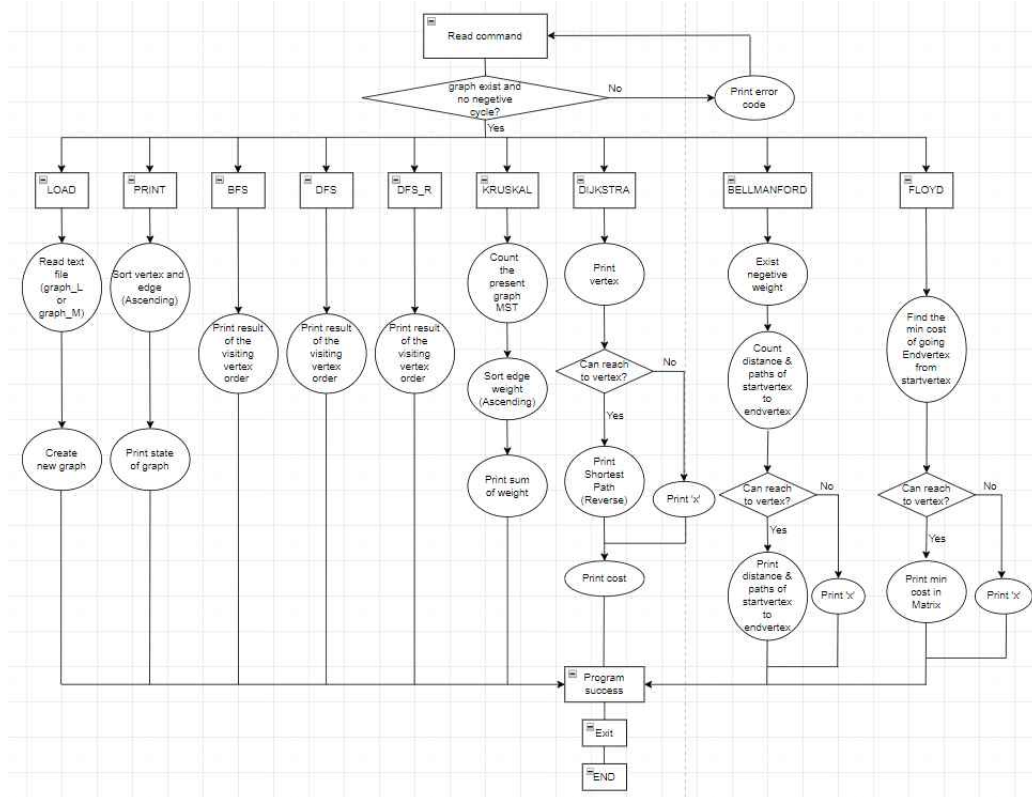
- 모든 vertex의 쌍에 대해서 StartVertex에서 EndVertex로 가는데 필요한 비용의 최소값을 행렬 형태로 출력한다.
- 기준 vertex에서 도달할 수 없는 vertex의 경우 'x'를 출력한다.
- 명령어를 수행할 수 없는 경우, 음수 사이클이 발생한 경우 출력 파일(log.txt)에 알맞은 오류 코드를 출력한다.(900)

## ⑩ EXIT

프로그램 상의 메모리를 해제하며, 프로그램을 종료한다.

## 2. Flowchart (draw.io 이용)

이번 3차 프로젝트의 전체적인 flowchart는 아래와 같다.



## 3. Algorithm

### ① BFS, DFS 알고리즘

#### - BFS(Breadth-First Search)

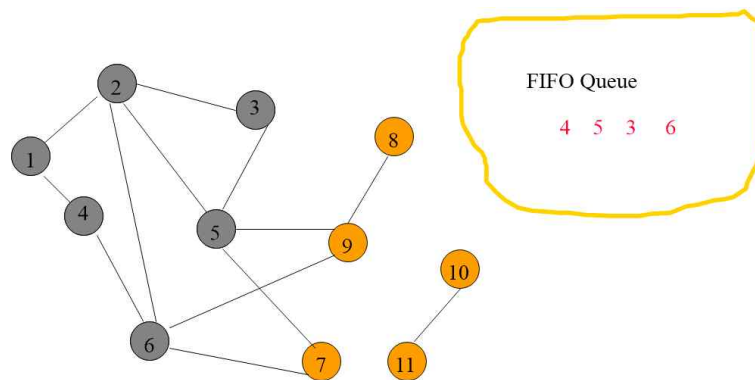
너비 우선 탐색이라고 부르는데 뜻은 root node(혹은 다른 임의의 node)에서 시작하여 인접한 node를 먼저 탐색하는 방법이다.

#### ※ 특징

1. 재귀적으로 동작하지 않는다.
2. 방문한 노드들을 차례로 저장한 후 꺼낼 수 있는 자료 구조인 큐(Queue)를 사용한다. (FIFO 원칙으로 탐색, 반복적인 형태 구현)

※ 작동 과정

1. 시작 node를 방문한다.  
큐에 방문된 node를 삽입한다.(enqueue)
2. 큐에서 꺼낸 node와 인접한 노드들을 모두 순서대로 방문한다.
  - 인접한 노드가 없으면 큐의 앞에서 node를 꺼낸다.(dequeue)1번 과정 반복
3. 큐가 소진될 때까지 계속한다.



참고 출처: <https://gmlwjd9405.github.io/2018/08/15/algorithm-bfs.html>

사진 출처: Graph\_Search\_Methods part 2 강의자료

- DFS(Depth-First Search)

우리말로 깊이 우선 탐색이라고 불리는데 뜻은 root node(혹은 다른 임의의 node)에서 시작하여 다음 branch로 넘어가기 전에 해당 branch를 완벽하게 탐색하는 방법이다.

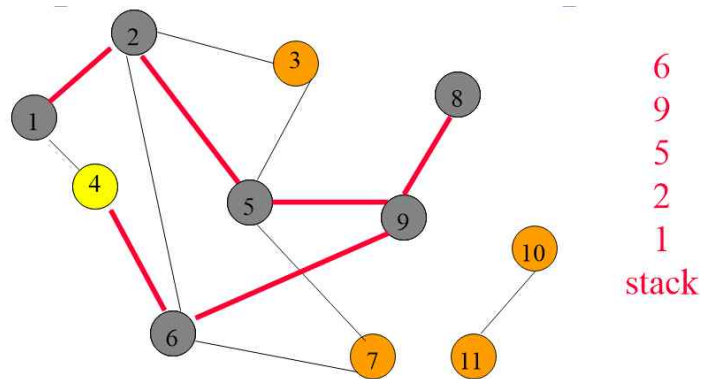
※ 특징

1. 자기 자신을 호출하는 순환 알고리즘의 형태를 띠고 있다.
2. 전위 순회(Pre-Order Traversal)를 포함한 다른 형태의 트리 순회도 DFS의 한 종류이다.

※ 작동 과정

1. 시작 node를 먼저 방문한다.
2. 시작 node와 인접한 node들을 차례대로 순회한다.

3. 시작 node와 이웃한 노드 b를 방문했으면 그 시작 node와 인접한 또 다른 node를 방문하기 전에 b에 이웃한 node들을 모두 방문한다.
4. b의 branch를 모두 완벽히 탐색했으면 다시 시작 node에 인접한 점들 중 아직 방문 안 된 점들을 찾는다.



참고 출처: <https://gmlwjd9405.github.io/2018/08/14/algorithm-dfs.html>

사진 출처: Graph\_Search\_Methods part 1 강의자료

## ② Kruskal 알고리즘

### - 신장 트리(Spanning Tree)

모든 정점들을 포함하고, 정점들 간에 서로 연결되면서 사이클이 존재하지 않는 그래프를 의미한다.

### - 최소신장트리(Minimum Spanning Tree)

간선의 가중치가 최소인 신장트리를 의미한다.

### - Kruskal's Algorithm

사이클을 만들지 않고 가중치가 가장 작은 간선을 하나씩 선택하는 방법

※ 작동 과정

1. 최소간선을 선택
2. 간선을 선택했을 때 사이클이 생성되는지 확인
3. 생성되지 않을 때 간선을 선택하고 1번 과정부터 반복  
생성되면 다른 간선 선택
4. 간선의 개수가 'node 개수-1'을 만족할 때까지 반복

참고 출처: <https://kosaf04pyh.tistory.com/314>

### ③ Dijkstra 알고리즘

시작 정점부터 나머지 각 정점까지의 최단 경로(Shortest Path)를 탐색하는 알고리즘을 의미한다.

※ 작동 과정

1. 방문하지 않은 정점 중 가장 가중치 값이 작은 정점을 방문한다.
2. 해당 정점을 거쳐서 갈 수 있는 정점의 거리가 이전에 기록한 값보다 작다면 그 거리를 갱신한다.

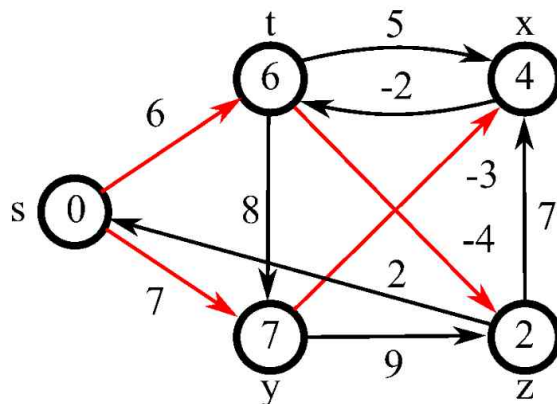
참고 출처: <https://code-lab1.tistory.com/29>

### ④ Bellman-Ford 알고리즘

한 node에서 다른 node까지의 최단 거리를 구하는 알고리즘  
간선의 가중치가 양수, 음수 관계없이 최단 거리를 구할 수 있다.

※ 작동 과정

1. 출발 node 선정
2. 최단 거리 table 초기화
3. 아래 과정을 (정점-1)번 반복하기
  - 모든 간선 E개를 하나씩 확인
  - 각 간선을 거쳐 다른 node로 가는 비용을 계산하여 최단 거리 table 갱신





참고 출처:

<https://velog.io/@kimdukbae/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EB%B2%A8%EB%A7%8C-%ED%8F%AC%EB%93%9C-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-Bellman-Ford-Algorithm>


## ⑤ FLOYD 알고리즘

가중치 포함 그래프의 각 정점에서 가는 모든 정점까지의 최단거리를 구할 때 쓰는 알고리즘  
음수 가중치가 없이 항상 양수의 그래프에서 동작한다.

참고 출처: <https://nolzaheo.tistory.com/10>

## 4. Result Screen

### ① Command.txt(Graph\_M.txt 이용)

```
Open ▾  command.txt
~/Downloads/code3-5

LOAD graph_M.txt
PRINT
BFS 0
DFS 0
DFS_R 2
KRUSKAL
DIJKSTRA 5
BELLMANFORD 0 6
FLOYD
EXIT
```

-> log.txt & terminal Result

```
Open ▾  log.txt
~/Downloads/code3-5

===== 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

===== BFS =====
startvertex: 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6

===== DFS =====
startvertex: 0
0 -> 1 -> 2 -> 4 -> 3 -> 6 -> 5

===== DFS_R =====
startvertex: 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5

===== Kruskal =====
[0] 2(2)
[1] 3(5)
[2] 0(2) 4(3)
[3] 1(5) 4(4) 6(3)
[4] 2(3) 3(4)
[5] 6(1)
[6] 3(3) 5(1)
cost: 18

===== Dijkstra =====
startvertex: 5
[0] x
[1] x
[2] x
[3] 5 -> 6 -> 4 -> 3 (15)
[4] 5 -> 6 -> 4 (11)
[5] 5 (0)
[6] 5 -> 6 (1)

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

===== 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

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

```

joon0723@ubuntu:~/Downloads/code3-5$ ./run
joon0723@ubuntu:~/Downloads/code3-5$ cat log.txt
===== 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
=====
===== BFS =====
startvertex: 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
=====
===== DFS =====
startvertex: 0
0 -> 1 -> 2 -> 4 -> 3 -> 6 -> 5
=====
===== DFS_R =====
startvertex: 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
=====
===== Kruskal =====
[0] 2(2)
[1] 3(5)
[2] 0(2) 4(3)
[3] 1(5) 4(4) 6(3)
[4] 2(3) 3(4)
[5] 6(1)
[6] 3(3) 5(1)
cost: 18
=====

```

```

===== Dijkstra =====
startvertex: 5
[0] x
[1] x
[2] x
[3] 5 -> 6 -> 4 -> 3 (15)
[4] 5 -> 6 -> 4 (11)
[5] 5 (0)
[6] 5 -> 6 (1)
=====
===== Bellman-Ford =====
0 -> 2 -> 5 -> 6
cost: 11
=====
===== 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
=====
===== EXIT =====
Success
=====

```

## ② Command.txt(Graph\_L.txt 이용)

```

command.txt
~/Downloads/code3-5

Open ▾ [icon]

LOAD graph_L.txt
PRINT
BFS 0
DFS 0
DFS_R 2
KRUSKAL
DIJKSTRA 5
BELLMANFORD 0 6
FLOYD
EXIT

```

-> log.txt & terminal Result

```

log.txt
~/Downloads/code3-5

Open ▾ [icon]

===== 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)
=====

===== BFS =====
startvertex: 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
=====

===== DFS =====
startvertex: 0
0 -> 1 -> 2 -> 4 -> 3 -> 6 -> 5
=====

===== DFS_R =====
startvertex: 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
=====

===== Kruskal =====
[0] 2(2)
[1] 3(5)
[2] 0(2) 4(3)
[3] 1(5) 4(4) 6(3)
[4] 2(3) 3(4)
[5] 6(1)
[6] 3(3) 5(1)
cost: 18
=====

```

```

===== Dijkstra =====
startvertex: 5
[0] x
[1] x
[2] x
[3] 5 -> 6 -> 4 -> 3 (15)
[4] 5 -> 6 -> 4 (11)
[5] 5 (0)
[6] 5 -> 6 (1)
=====

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

===== 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
=====

===== EXIT =====
Success
=====

```

```

joon0723@ubuntu:~/Downloads/code3-5$ ./run
joon0723@ubuntu:~/Downloads/code3-5$ cat log.txt
===== 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)
=====

===== BFS =====
startvertex: 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
=====

===== DFS =====
startvertex: 0
0 -> 1 -> 2 -> 4 -> 3 -> 6 -> 5
=====

===== DFS_R =====
startvertex: 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
=====

===== Kruskal =====
[0] 2(2)
[1] 3(5)
[2] 0(2) 4(3)
[3] 1(5) 4(4) 6(3)
[4] 2(3) 3(4)
[5] 6(1)
[6] 3(3) 5(1)
cost: 18
=====

```

```

===== Dijkstra =====
startvertex: 5
[0] x
[1] x
[2] x
[3] 5 -> 6 -> 4 -> 3 (15)
[4] 5 -> 6 -> 4 (11)
[5] 5 (0)
[6] 5 -> 6 (1)
=====

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

===== 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
=====

===== EXIT =====
Success
=====

```

Command 텍스트에서 load 시킨 graph 텍스트에 따라 Print 결과가 달라지는 것을 위의 결과를 통해 확인할 수 있었다.

## 5. Consideration

이번 3차 프로젝트에서 구현하다가 가장 많이 고민했었던 부분은 'KRUSKAL'이었다. 문제 조건대로 처음에 구현하다가 조금 더 효율적인 방법을 고려해본 결과 vertexSet 소스 파일 및 헤더 파일을 추가하는 것이 좋을 것 같다는 생각이 들었다. 3차 플젝 과제가 막 나왔을 때, vertexSet class를 활용하라고 되어있었다가 삭제되었는데 이 class를 이용하여 KRUSKAL 명령어 구현해서 오히려 더 쉽고 편리했다.

처음에는 VertexSet class 역할이 뭔지 몰라서 찾아본 결과 최소 신장트리를 위한 정점 집합 class라는 것을 바로 확인할 수 있었다.

Dijkstra, Bellmanford, Floyd 부분에서는 코드 구현하는 과정에서는 크게 어려움은 없었으나 차이점을 구별하기가 매우 힘들었다. 그래서 구글링을 따로 해본 결과 Dijkstra 알고리즘과 Bellmanford 알고리즘의 차이는 시간 복잡도, 음수 간선의 존재와 관계없이 최적의 해 찾기 등 여러 가지가 존재한다는 것을 확인할 수 있었다. 그리고 매 반복할 때마다 모든 간선을 확인하는 것도 차이점이라는 것을 알 수 있었다.

Dijkstra 알고리즘과 Floyd-Warshall 알고리즘의 차이도 인터넷으로 검색해본 결과 Dijkstra 알고리즘의 경우 한 정점에서 모든 정점까지의 거리를 알고 싶을 때 사용하는 것이고 Floyd-Warshall 알고리즘은 모든 정점에서 모든 정점까지의 거리를 알고 싶을 때 사용한다는 사실을 알 수 있었다.

프로젝트 진행하면서 command와 graph 텍스트 입력을 무엇을 할까 고민해본 결과 graph\_L.txt, graph\_M.txt는 문제에서 제시한 예시 그대로 사용하였고 Command.txt도 그대로 사용했는데 Dijkstra, Bellmanford 명령어 부분에서 값을 일부 변경하여 문제 예시와 같이 출력되게 하였다.

문제에서 특별히 Exit 부분 설명이 따로 없어서 manager.cpp에 추가로 구현하여 깔끔하게 프로그램이 종료될 수 있도록 하였다.

(Exit 명령어는 error code 따로 만들지 않음)

이번 프로젝트 실습을 통해 각 명령어의 알고리즘의 역할, 특징, 알고리즘 과정을 정확히 배울 수 있어서 좋았다. 그리고 각 알고리즘의 차이점도 개념만으로 이해하기 어려웠는데 출력 결과 확인 과정을 거치면서 쉽게 알 수 있었다.