Data Structure Project #3

2023202027 정정윤

1. Introduction

본 프로젝트는 그래프 탐색 알고리즘인, DFS, BFS, KRUSKAL, DIJKSTRA, BELLMAN-FORD, FLOYD에 대한 프로젝트이다. 이를 적용하기 위해 C++에서 그래프를 저장하는 방법과 그 정점, 간선들의 정렬 방식을 알고 구현할 필요가 있다.

그래프 탐색 알고리즘을 수행하는 프로그램은 하나의 C++프로그램에서 명령형으로 동작하며, 명령과 입력 데이터, 출력은 텍스트 파일로 주어진다. 명령은 리스트 형태 또는 행렬 형태로 그래프의 정점 개수와 간선들을 입력받고, 그 형태에 따라 저장하는 LOAD, 저장된 그래프를 출력하는 PRINT, 프로그램을 종료하는 EXIT의 3가지 프로그램 제어 명령이 있고, 그래프 탐색 알고리즘을 수행하는 위의 알고리즘들이 명령으로써 주어진다. DFS는 구현 방식에 따라 재귀적이지 않은 DFS와 재귀적인 DFS_R을 따로 구현하여야 한다.

각 명령은 정해진 개수의 인자를 받으며, 인자 개수가 부족하거나 많아 명령을 수행하기에 부적합하다면 프로그램은 적절한 오류 코드를 출력하여야 한다. 또, 해당 명령을 수행할 수 없는 경우에도 적절한 오류 코드를 출력하여야 한다.

프로그램은 메모리 누수가 발생하여서는 안되며, Linux에서 컴파일 및 동작 가능해야 하고, 주어진 스켈레톤 코드의 형태에서 구현할 수 있도록 하여야 한다.

DFS와 BFS는 그래프의 순회 알고리즘으로 인자로 입력된 해당 알고리즘에 따라 방문한 정점들의 순서를 출력하여야 한다. DFS는 어떤 정점으로 간 경우, 그 정점에서 방문할 수 있는 하나의 다른 정점을 방문하는 것을 반복한다. 방문할 수 없는 경우 이전 정점으로 돌아가 가능한 다른 정점이 있는지 확인한다. BFS는 어떤 정점으로 간 경우, 그 정점에서 방문할 수 있는 모든 정점을 방문하고 각 정점에서 방문할 수 있는 다른 정점들을 방문한다.

KRUSKAL은 가능한 한 적은 비용을 가지는 간선들으로 최소 신장 트리를 구하기 위하 알고리즘이다. 탐욕법을 사용한 알고리즘으로, 가장 비용이 적은 간선부터 확인하며, 순환을 형성하지 않는 한 그 간선을 추가한다.

DIJKSTRA는 빠른 시간에 최소 비용 경로를 탐색하기 위한 알고리즘이다. 각 정점에 시작 정점에서 방문할 수 있는 최소 비용을 기록해두고, 방문할 수 있는 각 정점을 방문하며, 해당 정점을 통해 가는 비용과 원래 비용을 비교하며, 최소 비용을 탐색한다. 기반으로 하여, 한 번 탐색한 정점은 방문하지 않으므로, 사용 불가능한 몇몇 조건이 있다.

BELLMAN-FORD는 더 적은 제약 조건 하에 최소 비용 경로를 탐색하기 위한 알고리즘이다. 동적 프로그래밍 기법을 사용하며, 음수 순환이 발생한 것을 감지할 수 있다. 다익스트라와 다르게, 탐색한 정점에 대해서도 다시 방문할 수 있다.

FLOYD는 모든 정점쌍에 대해 경로를 탐색하는 알고리즘이다. 동적 프로그래밍 기법을 사용하며, BELLMAN-FORD를 모든 정점에 대해 수행한 것과 비슷한 방식을 가진다. 마찬가지로 음수 순환을 감지할 수 있다.

본 프로젝트를 아래 표를 기준으로 프로그램을 작성하였다.

알고리즘	목표	조건
BFS	모든 정점 탐색	무방향, 무가중치
DFS	모든 정점 탐색	무방향, 무가중치
KRUSKAL	최소 신장 트리 구성	방향성이 없을 것
DIJKSTRA	최소 비용 경로 탐색	음수 가중치가 없을 것
BELLMAN-FORD	최소 비용 경로 탐색	음수 순환이 없을 것
FLOYD	모든 정점쌍에 대해 최소 비용 경로 탐색	음수 순환이 없을 것

Table 1 각 알고리즘 별 목표와 조건

2. Flowchart

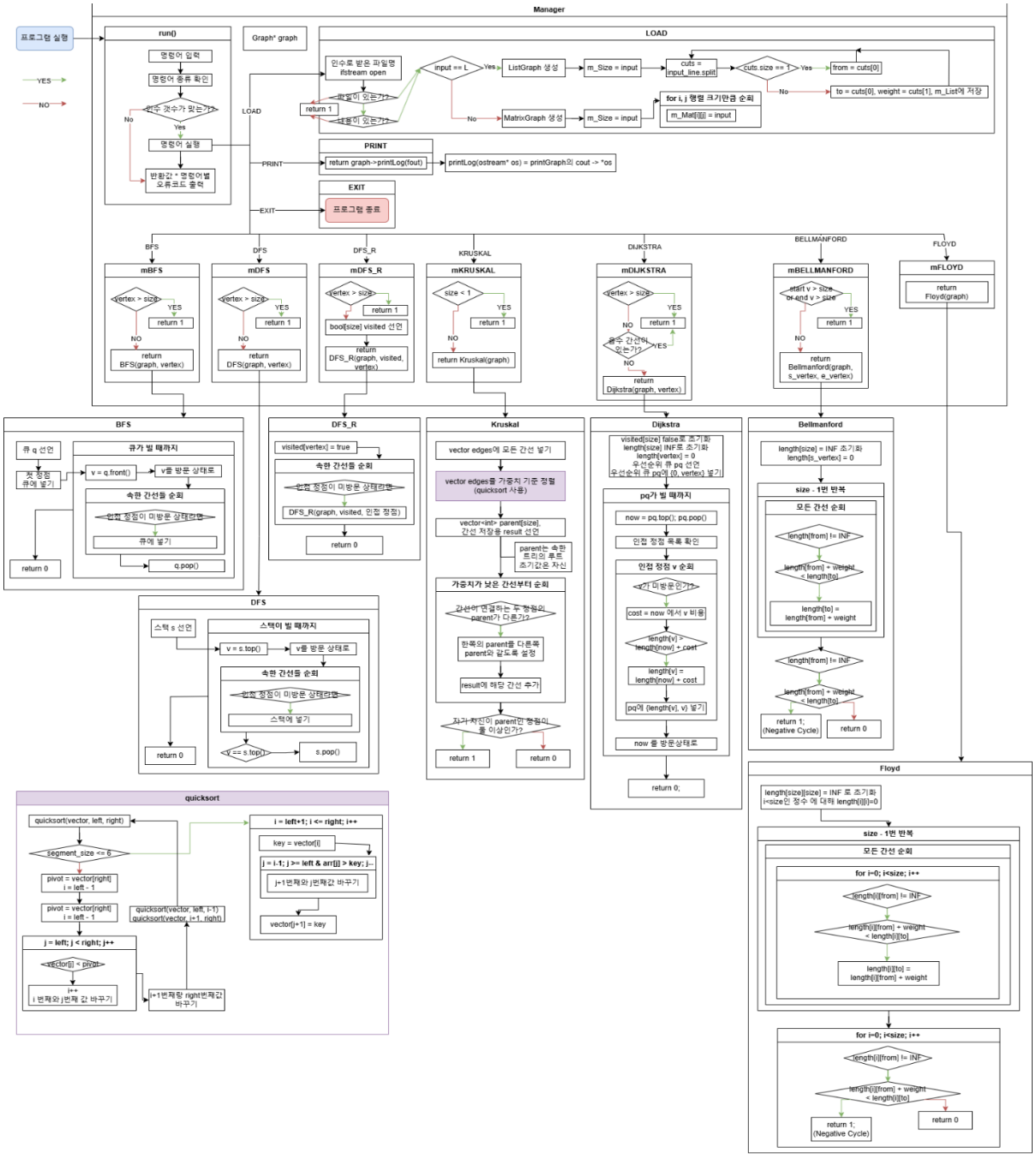

Figure 1 Flowchart

명령형 프로그램 구조와, 각 알고리즘 실행이 Graph 내부의 멤버 함수가 아닌 GraphMethod에 따로 선언된 함수가 실행되는 구조를 고려하여 Flowchart를 구성하였다. 순회 부분에 대해서는 상세가 필요한 경우 for문의 구조, 반복자를 이용한 반복의 경우 무엇을 순회하는지 의미를 기록하였다. 순서도 상세는 다음 링크를 통해 확인할 수 있다. https://drive.google.com/file/d/1zUKXu6zWNsjqCUvQUSiLMDWu774KrmkH/view?usp=sharing

3. Algorithm

Manager.run → Manager 멤버 함수 → GraphMethod 구조를 가지고 있어, 명령어 구분 및 적절한 함수로 이동 과정을 다소 코드를 읽기 쉽도록 개선하였다. 명령어의 인자 개수 확인 까지만 명령어 구분단계에서 수행하고, 이후의 조건에 따른 예외처리를 멤버 함수에서, 탐색 알고리즘 실행 도중의 오류는 GraphMethod 내의 함수에서 반환하도록 하였다. 오류 반환의 경우, 프로그램 실행 종료와 같이 0일 때 오류 없음, 1 이상의 경우 오류가 있다는 표시로, 구현 단계에서 각 함수는 오류가 발생하면 1을 반환하도록 하여, 각 기능의 오류 코드와 곱하여 출력하였다.

BFS는 큐를 이용하여 구현하였다. 첫 정점을 큐에 넣은 후, 큐의 front에서 정점을 불러오고 그 정점을 방문한 상태로 만든다. 이후, 그 정점과 인접한 방문하지 않은 모든 정점을 큐에 넣는다. 이후 큐의 맨 앞에서 값을 제거하는 것을 큐가 빌 때까지 반복하면 시작 정점에서 더 적은 간선을 지나는 정점 순서로 방문하게 된다.

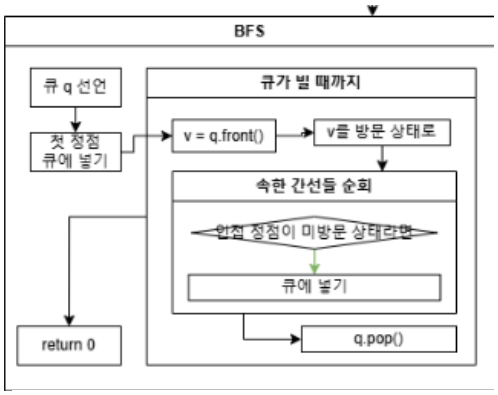

Figure 2 BFS Flowchart

DFS는 스택을 이용하여 구현하였다. 첫 정점을 스택에 넣은 후, 스택의 top에서 정점을 불러온다. 이후 그 정점을 방문 상태로 만들고, 정점과 이어진 간선들을 순회하며, 인접한 미방문 상태의 정점을 스택에 넣는다. 만약 스택에 넣은 정점이 없어 현재 정점과 top이 같다면 스택의 top을 제거한다. 이를 스택이 빌 때까지 반복하는 것으로 깊이 우선 탐색을 수행할 수 있다.

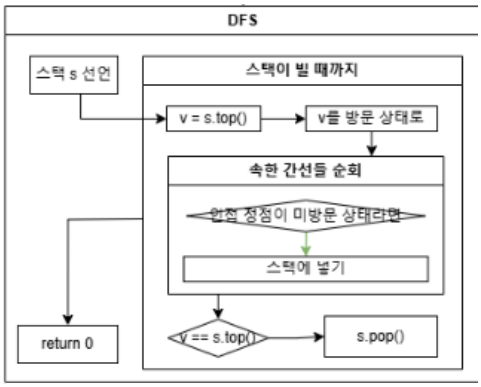

Figure 3 DFS Flowchart

DFS_R은 깊이 우선 탐색을 재귀적 방법으로 구현한 것이다. 현재 정점과 정점의 방문 상태가 매개변수로 주어진다. 함수를 통해 정점을 방문하며, 이 때, 해당 정점을 방문 상태로 만든다. 이후, 인접한 정점들을 순회하며, 해당 정점이 미방문 상태라면 같은 함수에 방문 정점으로 지정하여 실행한다. 모든 정점이 방문 상태라면 하나씩 return 되어 처음 함수가 return 되면 깊이 우선 탐색이 종료된다.

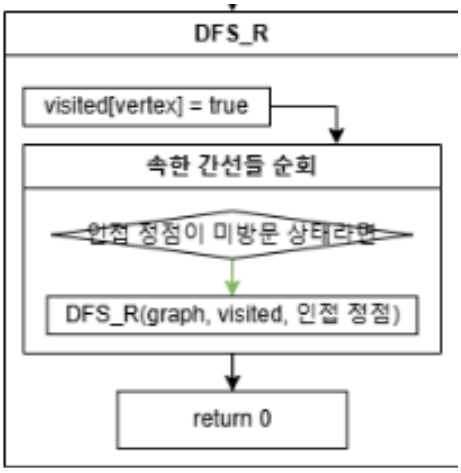

Figure 4 DFS_R Flowchart

Kruskal은 최소 신장 트리를 구성하기 위한 알고리즘으로, 가장 비용이 적은 간선부터 순회하며, 순환이 발생하지 않는 한 모든 간선을 추가하는 방식으로 구성하게 된다.

추가하는 것에서 해당 간선이 순환을 발생시키지 않는지 확인하는 것이 중요한 부분이 되는데, Disjoint Set을 트리를 이용해 구성하여 이를 확인하도록 하였다. 그래프의 size와 같은 길이의 parent 배열을 선언하여, 각 Node가 parent를 가리키도록 트리를 구성한다. 이유는 Disjoint Set은 Child를 찾는 것보다 최종적인 root가 같은지 확인하여 동일한 트리 안에 있는지 체크하는 것이 목표이기 때문이다. 따라서, root가 같은 두 정점을 잇는 간선의 경우에는 결과에 포함되지 않으며, 다른 정점을 잇는 경우에는 결과에 포함하며, 그 두 정점의 트리를 Union하게 된다. Union은 한 쪽의 parent를 다른쪽의 root 또는 다른쪽 그 자체로 정하는 것으로 수행할 수 있다.

Kruskal의 경우, 최소 신장 트리를 구성할 수 없는 경우 오류를 출력해야 하므로, Disjoint Set이 두 개 이상 나오게 되는 경우에 오류를 반환한다. 따라서, 초기값인 자기 자신이 parent에 있는 정점이 두 개 이상인 경우 오류를 반환하도록 하였다. Disjoint Set이 하나라면 Union과정에서 결국 하나의 정점만 parent를 자신으로 가지게 된다.

Dijkstra는 최소 비용 경로를 탐색하기 위한 알고리즘으로, 방문하지 않은 정점들을 순회하며, 시작 정점에서 각 정점까지의 최소 비용을 기록하며, 그래프를 순회한다. 각 정점 방문의 최소 비용을 MAX로 초기화하고, 우선순위 큐를 이용해 현재 가장 비용이 작은 정점부터 방문하여, 인접한 미방문 상태의 정점들의 최소 비용을 갱신한다. 이를 모든 정점을 방문할 때까지 반복하게 되면, 각 정점으로 가기 위한 최소 비용들이 기록된다. 경로까지 파악하기 위해서는 parent[size] 배열을 선언한 후, 최소 비용을 위해 방문하는 경로 상 직전 정점을 기록하면 이를 경로를 찾고자 하는 최종 정점부터 역추적하는 것으로 경로를 찾을 수 있다. 출력 시에는 이를 스택에 넣고 다시 출력하는 것으로 처음 정점부터 출력하도록 하였다.

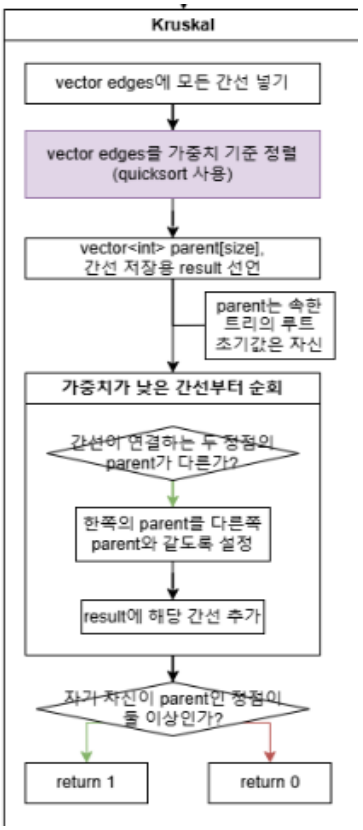

Figure 5 Kruskal Flowchart

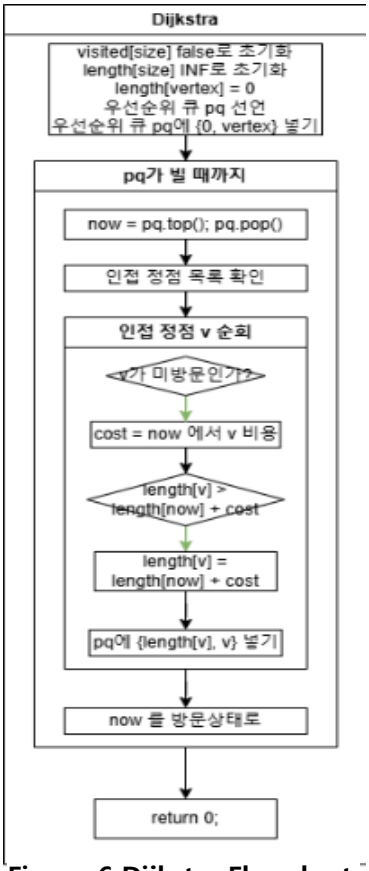

Figure 6 Dijkstra Flowchart

Dijkstra의 경우 음수 가중치를 가지는 간선이 있을 경우 올바르게 작동하지 않는데, 그 이유는 이미 방문한 정점을 다시 방문하지 않기 때문이다. 탐욕법의 특징과도 같은데, 가중치가 크게 증가하였다가 크게 감소하는 경로의 경우, 다른 경로가 먼저 탐색되어 그 경로로 방문하는 정점을 방문 상태로 만들어버리기에 해당 현상이 발생하게 된다. 작성한 프로그램에서는 Manager의 멤버 변수가 모든 간선을 확인하고 음수 가중치의 간선이 있을 경우 오류 코드를 반환하도록 한다.

Bellman-ford는 음수 가중치를 가지는 그래프에서도 작동할 수 있도록 만들어진 알고리즘으로 동적 프로그래밍을 기반으로 한다. 개중 반복 수행에 있어 다음 연산에 사용하는 부분을 기록해두어 다음 연산에 사용하는 방식을 사용하게 된다.

시작 정점에서 각 정점까지 가는 거리를 MAX(INF)로 초기화하고, 시작 정점은 0으로 둔다. 이후 모든 간선들을 순회하며, 각 간선의 <u, v>에 있어 u의 시작 정점으로부터의 비용이 INF가 아니라면, v의 최소 비용을 갱신하게 된다. 이 때, v가 이미 방문되었는지는 중요하지 않으며, 각 반복마다 방문한 정점이 기준이 아닌 모든 간선을 기준으로 반복한다. 따라서, 정점의 개수 -1 만큼 반복을 수행하면 모든 경로를 탐색하게 되며, 최소 비용을 파악할 수 있다. 음수 가중치가 존재하더라도 방문한 정점의 최소 비용을 변경할 수 있기에 처리할 수 있다. 만약 음수 사이클이 발생하게 될 경우에는 이미 최적해가 구해진 상태인 size-1회 반복 이후에도 최소 비용 갱신되어야 하므로, 한 번 더 모든 간선을 수행하여 비용이 갱신되는 경우에는 오류 코드를 반환하도록 하였다.

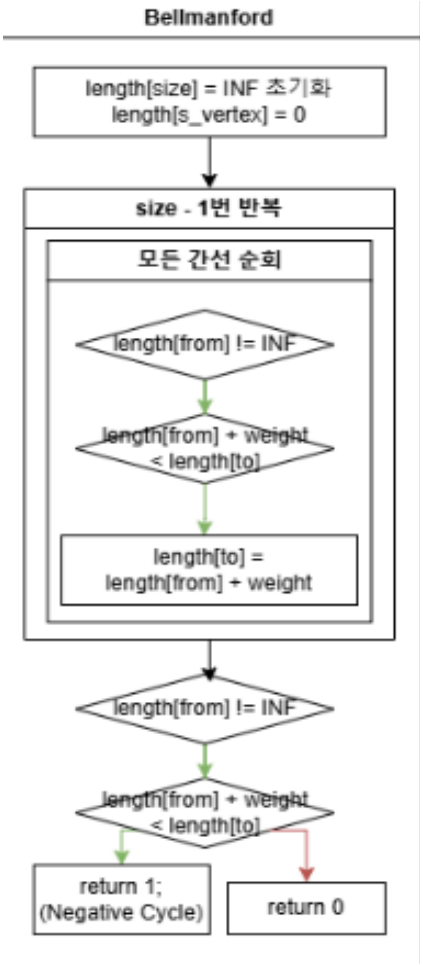

Figure 7 Bellman-ford Flowchart

Floyd 알고리즘은 Bellman-ford 알고리즘을 그래프 내 모든 정점쌍에 대해서 수행하는 것으로 length[size] 배열을 length[size][size] 2차원 배열로 확장하여 [i][j] 요소에 대해 i정점을 시점으로 하는 각 j정점으로의 최소 비용을 탐색하는 것이라 볼 수 있다. 따라서, 모든 간선 순회 과정에 있어 모든 정점에 대해 순회하며 length[i] 에서 각 간선 <u, v>를 확인하고 비용을 갱신한다. 음수 사이클에 대한 확인은 Bellman-ford와 같이 size-1 반복 이후 한 번 더 갱신을 수행하여 갱신되는 수치가 있을 경우 음수 사이클이 발생한 것으로 판단할 수 있다.

4. Result Screen

첫번째 테스트 케이스는 프로젝트 설명 자료에 주어진 그래프이다.

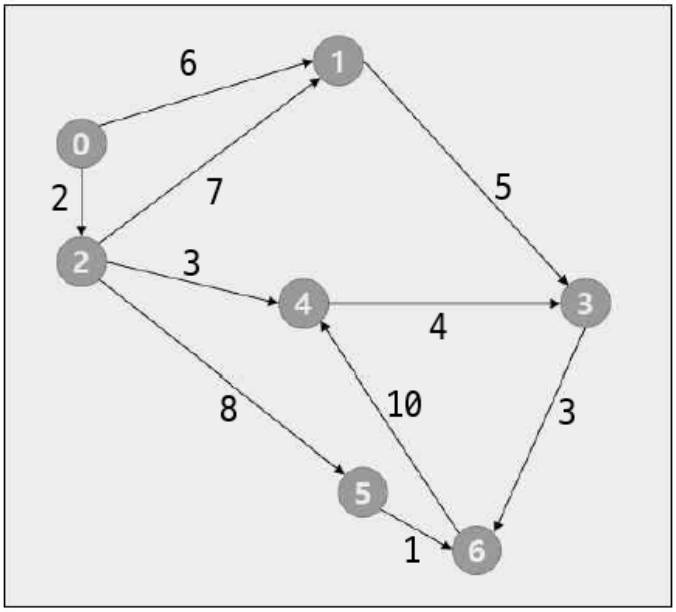

Figure 11 Testcase 1 Visualized (Source : DS_Project3_v2.pdf)

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

Figure 9 Command.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
```

Figure 8 loadfile.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)
================
===== DFS =====
startVertex : 0
0 -> 1 -> 2 -> 4 -> 3 -> 6 -> 5
===============
===== BFS =====
startVertex : 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
===============
===== DFS_R =====
startVertex: 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
================
```

Figure 10 Result1

```
===== 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] 6 -> 4 -> 3 (15)
[4] 6 -> 4 (11)
[6] 6 (1)
==================
===== BELLMAN-FORD =====
0 -> 2 -> 5 -> 6
cost: 11
=======================
```

Figure 12 Result 2

```
===== 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
```

Figure 13 Result 3

이 테스트 케이스에 대한 출력값은 PRINT부분을 제외하면 모두 프로젝트 자료에 포함되어 있다. 모든 출력이 제시된 값과 일치함을 알 수 있다.

두번째 테스트 케이스는 ChatGPT를 이용해 Directed Weighted Graph의 생성을 요청하여 생성하였다.

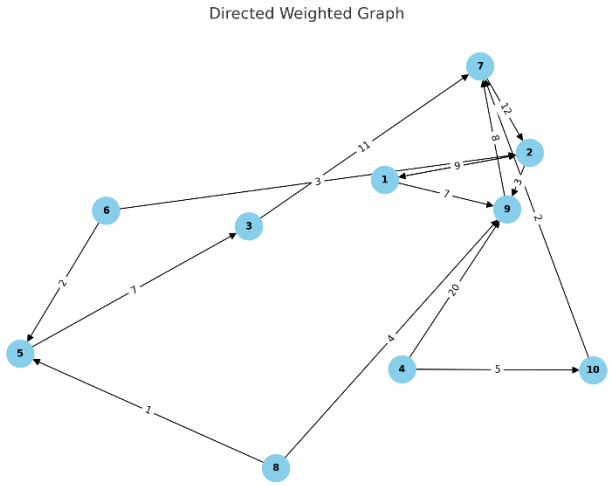

Figure 15 Testcase 2 Visualized (From : ChatGPT)

```
M
10
0 7 0 0 0 0 0 0 7 0
9 0 0 0 0 0 0 0 3 0
0 0 0 0 0 0 11 0 0 0
0 0 0 0 0 0 0 0 20 5
0 0 7 0 0 0 0 0 0 0
0 3 0 0 2 0 0 0 0 0
0 12 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 4 0
0 0 0 0 0 0 8 0 0 0
0 0 0 0 0 0 2 0 0 0
```

Figure 14 loadfileM.txt

Testcase 2에서의 입력 파일은 Figure 9에서 loadfile만 loadfileM으로 바꾼 것을 사용하였다.

```
[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
================
===== LOAD =====
Success
================
===== PRINT =====
        [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]     [8]     [9]
[0]     0       7       0       0       0       0       0       0       7       0
[1]     9       0       0       0       0       0       0       0       3       0
[2]     0       0       0       0       0       0       11      0       0       0
[3]     0       0       0       0       0       0       0       0       20      5
[4]     0       0       7       0       0       0       0       0       0       0
[5]     0       3       0       0       2       0       0       0       0       0
[6]     0       12      0       0       0       0       0       0       0       0
[7]     0       0       0       0       1       0       0       0       4       0
[8]     0       0       0       0       0       0       8       0       0       0
[9]     0       0       0       0       0       0       2       0       0       0
================
===== DFS =====
startVertex : 0
0 -> 1 -> 5 -> 4 -> 2 -> 6 -> 8 -> 3 -> 9 -> 7
===============
===== BFS =====
startVertex : 0
0 -> 1 -> 8 -> 5 -> 6 -> 3 -> 7 -> 4 -> 2 -> 9
===============
===== DFS_R =====
startVertex: 2
2 -> 4 -> 5 -> 1 -> 0 -> 8 -> 3 -> 9 -> 6 -> 7
================
```

Figure 16 Case2 Result1

Figure 16는 log.txt의 일부분을 가져온 것이다. Testcase 1 실행 후 log.txt를 초기화하지 않았기 때문에, 해당 부분의 뒤에서부터 log가 작성된다.

Adjacency Matrix로 데이터를 입력하였기에 PRINT는 해당 가중치들을 출력한다.

DFS와 BFS의 출력 결과는 그래프 이미지를 바탕으로 손으로 추적해본 결과 정상 출력임을 알 수 있었다.

처음에는 의도된 설계는 아니었으나, 간선을 불러오는 것에 있어 map에 {vertex, weight}로 저장하게 될 경우, 자동으로 key인 vertex를 기준으로 오름차순 정렬되기에 vertex가 더 작은 vertex를 먼저 방문하게 된다.

```
===== KRUSKAL =====
[0] 8(7)
[1] 5(3) 8(3)
[2] 4(7)
[3] 9(5)
[4] 2(7) 5(2) 7(1)
[5] 1(3) 4(2)
[6] 8(8) 9(2)
[7] 4(1)
[8] 0(7) 1(3) 6(8)
[9] 3(5) 6(2)
cost: 38
====================
```

Figure 18 Case2 Result 2

```
===== FLOYD =====
      [0]   [1]   [2]   [3]   [4]   [5]   [6]   [7]   [8]   [9]
[0]   0     7     x     x     x     x     15    x     7     x
[1]   9     0     x     x     x     x     11    x     3     x
[2]   32    23    0     x     x     x     11    x     26    x
[3]   28    19    x     0     x     x     7     x     20    5
[4]   39    30    7     x     0     x     18    x     33    x
[5]   12    3     9     x     2     0     14    x     6     x
[6]   21    12    x     x     x     x     0     x     15    x
[7]   33    24    8     x     1     x     12    0     4     x
[8]   29    20    x     x     x     x     8     x     0     x
[9]   23    14    x     x     x     x     2     x     17    0
================
```

Figure 17 Case2 Result3

Figure 18은 Kruskal을 수행한 결과, Figure 17은 Floyd를 수행한 결과이다. Kruskal의 경우에도 이미지를 바탕으로 직접 구해본 결과 정확한 결과를 내는 것을 알 수 있었다.

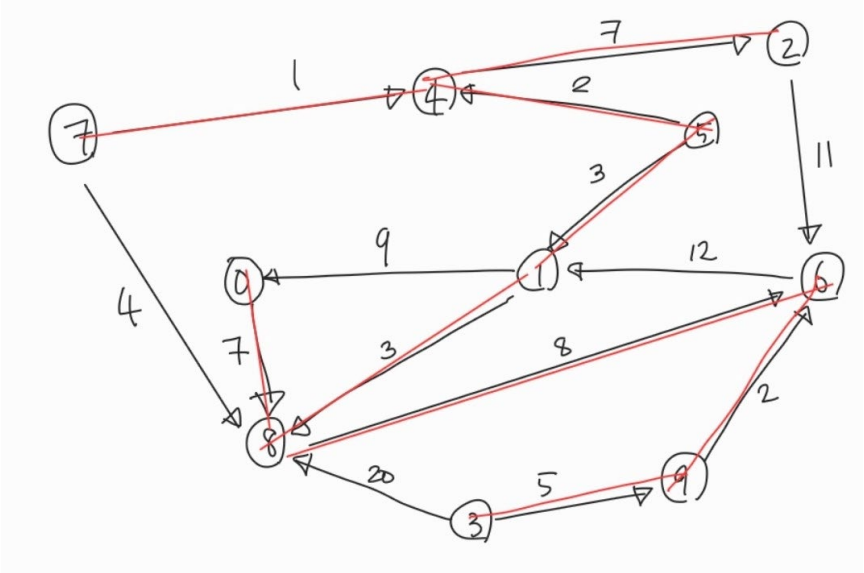

Figure 19 Kruskal Calculated by hand

Floyd의 경우에도 Floyd 연산 계산기를 작동시킨 결과 정확한 결과가 도출되었음을 알 수 있었다.

Matrix 10

	A	B	C	D	E	F	G	H	I	J
A	-	7	Infinity	Infinity	Infinity	Infinity	15	Infinity	7	Infinity
B	9	-	Infinity	Infinity	Infinity	Infinity	11	Infinity	3	Infinity
C	32	23	-	Infinity	Infinity	Infinity	11	Infinity	26	Infinity
D	28	19	Infinity	-	Infinity	Infinity	7	Infinity	20	5
E	39	30	7	Infinity	-	Infinity	18	Infinity	33	Infinity
F	12	3	9	Infinity	2	-	14	Infinity	6	Infinity
G	21	12	Infinity	Infinity	Infinity	Infinity	-	Infinity	15	Infinity
H	33	24	8	Infinity	1	Infinity	12	-	4	Infinity
I	29	20	Infinity	Infinity	Infinity	Infinity	8	Infinity	-	Infinity
J	23	14	Infinity	Infinity	Infinity	Infinity	2	Infinity	17	-

Figure 20 Floyd Calculated by Floyd-Warshall Solver

-는 0인 점을 제외하면 결과가 동일한 것을 알 수 있다. 여기서 구한 결과를 바탕으로 Dijkstra와 Bellman-ford의 결과 또한 검증할 수 있다.

Dijkstra의 경우, 5번 Vertex 즉 F Vertex의 Floyd결과를 통해 최소 비용을 확인함으로써 일부 검증이 가능하다. 경로의 경우에는 해당 경로를 직접 이미지 상에서 통해보면 올바른 경로를 출력하였는지 확인할 수 있다.

Bellman-ford의 경우에도 같은 방식으로 검증이 가능하다.

```
===== DIJKSTRA =====
startvertex: 5
[0] 1 -> 0 (12)
[1] 1 (3)
[2] 4 -> 2 (9)
[3] x
[4] 4 (2)
[6] 1 -> 8 -> 6 (14)
[7] x
[8] 1 -> 8 (6)
[9] x
====================
===== BELLMAN-FORD =====
0 -> 8 -> 6
cost: 15
========================
```

Figure 21 Case2 Result4

위 케이스에서 만약 Vertex 9를 향한 모든 간선을 제거하고 테스트를 실행하게 될 경우에는 다음과 같은 결과가 나옴을 알 수 있다. 다음은 log.txt의 일부를 가져온 것이다.

```
===== PRINT =====
        [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]     [8]     [9]
[0]     0       7       0       0       0       0       0       0       7       0
[1]     9       0       0       0       0       0       0       0       3       0
[2]     0       0       0       0       0       0       11      0       0       0
[3]     0       0       0       0       0       0       0       0       20      0
[4]     0       0       7       0       0       0       0       0       0       0
[5]     0       3       0       0       2       0       0       0       0       0
[6]     0       12      0       0       0       0       0       0       0       0
[7]     0       0       0       0       1       0       0       0       4       0
[8]     0       0       0       0       0       0       8       0       0       0
[9]     0       0       0       0       0       0       0       0       0       0
================
===== DFS =====
startVertex : 0
0 -> 1 -> 5 -> 4 -> 2 -> 6 -> 8 -> 3 -> 7
===============
===== BFS =====
startVertex : 0
0 -> 1 -> 8 -> 5 -> 6 -> 3 -> 7 -> 4 -> 2
===============
===== DFS_R =====
startVertex: 2
2 -> 4 -> 5 -> 1 -> 0 -> 8 -> 3 -> 6 -> 7
=================
======== ERROR ========
600
======================
```

Figure 22 Case2-2 Result1

Vertex 9에 대한 모든 연결이 사라졌기에 Kruskal은 최소 신장 트리를 생성할 수 없으므로, 오류 코드 600을 출력한다. 그리고 DFS, BFS의 경우 9번 정점을 향할 수 없기에 9번 정점을 제외한 탐색 경로만이 출력된다.

Figure 11의 그래프에서 3, 4, 6번의 순환 과정에서 음수 사이클을 형성하게 될 경우 (새로운 테스트케이스에서는 <6,4>의 가중치를 -10으로 설정하였다. 이 테스트케이스에서는 log.txt가 아래와 같이 출력된다.

```
======== ERROR ========
700
======================
======== ERROR ========
800
======================
======== ERROR ========
900
======================
```

Figure 23 Case1-2 Result1

음수 가중치가 있으며, 음수 사이클이 발생하기에 각각 Dijkstra, Bellman-ford, Floyd의 오류 코드인 700, 800, 900을 출력하는 것을 볼 수 있다.

5. Consideration

Kruskal 구현 시기, Disjoint Set의 Tree로의 구현을 사용하기 전에 구현을 하였을 때는, set STL을 이용하여 구현하였다. set에서 데이터 탐색이 시간 소요가 많이 안되기도 하고, set은 반복자를 이용하여 서로 합치는 것이 가능하였기에 이 방법을 선택하였다. Disjoint Set의 Tree 방식으로의 구현을 알고 난 이후에 다시 해당 방식으로 구현을 하였다. 구현 이전과 이후의 코드를 확인한 결과 코드의 시인성이 많이 개선되었고, 수행 과정에서 더 적은 테스트를 필요로 하였다. 필요한 정보에 따라 구현 방식을 바꾸는 것이 많은 도움이 된다는 사실을 알 수 있었다.

Dijkstra 구현 과정에서 우선순위 큐의 우선순위가 어떤 순서로 적용되는 지 알고 있지 않아 해당 구현에서 어려움이 있었으나, 확인해본 결과 Value의 내림차순으로 정렬된다는 것을 확인하였다. 이에 Value에 음수를 곱하는 것으로 내림차순 정렬을 수행하더라도 Value가 가장 작은 값을 먼저 가져올 수 있도록 하고자 하였고, 이를 기준으로 코드를 작성하였다. 이후에 정렬 기준을 조정할 수 있는 방법을 파악하고 해당 방식으로 우선순위 큐를 수정하였는데, STL 라이브러리의 사용 방법을 구체적으로 파악하고 있는 것이 중요하다는 사실을 알 수 있었다.

테스트케이스용 그래프 생성에 있어, 직접 그래프를 그려보고 생성하고자 하였으나, 결국 테스트케이스로 적합하지 않거나, 오류나 결함 발견에 있어 프로그래머가 직접 작성한 테스트케이스만으로는 검증이 힘들다 판단하여 ChatGPT에게 적절한 그래프 생성과 시각화를 요청하여 테스트케이스로 사용하였다. 이에 DFS 과정과 Dijkstra에서 있었던 일부 문제를 발견하고 수정할 수 있었고, 위 결과 자료에서도 해당 자료가 사용되었다. AI를 이용하여 적절한 테스트 케이스를 생성하고 검증하는 것 또한 많은 도움이 될 수 있다는 사실을 알 수 있게 되었다.

정렬 알고리즘 구현 시, 데이터 정렬 비교 연산을 단순히 반복자 내부에서 불러오고 map에서 불러와 사용하였으나, 원하는 방식으로 정렬을 더 간편하게 하기 위해서는 비교 연산은 일반적으로 크기 비교가 가능한 자료형에 대해서 사용하듯이 하고, 자신이 원하는 대상에 있어서는 클래스로 선언하여 연산자 오버로딩을 통해 해당 코드의 입력값만 바꾸어 그대로 사용할 수 있다는 사실을 알게 되었다. 결국 템플릿을 더욱 폭넓게 사용하는 방법이라 생각이 들었고, 재사용 가능한 코드를 작성하는 요령을 하나 더 알게 되었다.