Data Structure Project 3

컴퓨터정보공학부 2019202023 강병준

과목	데이터구조실습(수) 데이터구조설계(월 5 수 6)
교수	실습-공진흥 교수님 설계-이기훈 교수님
학번 및 이름	2019202023 강병준
전공	컴퓨터정보공학부
제출일자	2022/12/15

Introduction

데이터구조설계 과목의 3 차 프로젝트에 대한 설명이다. 그래프 이론을 바탕으로 BFS, DFS, DFS_R, KRUSKAL, DIJKSTRA, BELLMAN-FORD, FLOYD 알고리즘을 구현한다. 그래프의 구축은 graph_L.txt 혹은 graph_M.txt 를 바탕으로 구축하는데, 해당 파일은 command.txt 에서 LOAD 명령어를 수행시에 인자로 전달함으로써 해당 그래프 txt 파일에 있는 vertex 와 해당 vertex 부터 다른 vertex 로의 weight 를 저장하여 그래프를 구축한다. 그래프를 구축하기 위한 vertex 는 중간중간 띄는 값이 없이 0 부터 1, 2, 3 순으로 차례대로 1 씩 올라가는 것으로 가정이 되어 있다. 또한 edge 가 없는 vertex 또한 존재하므로 해당 vertex 에 대한 예외 또한 고려해야 한다. 각 그래프 연산 알고리즘에 쓰이는 그래프의 형태(방향성의 유무)와 weight 의 양음 여부에 따라서 그래프 구축의 여부가 달라지기 때문에 이를 고려하여 구축하도록 한다. 3 차 프로젝트에 대한 Introduction 은 구축해야 하는 그래프 연산인 BFS, DFS, DFS_R, KRUSKAL, DIJKSTRA, BELLMAN-FORD, FLOYD 를 기준으로 진행할 것이다.

BFS

BFS 는 너비 우선 탐색 방식으로, 특정 vertex 부터 주변 vertex 들을 queue 에 저장하여 First In First Out 구조로 탐색을 진행한다. BFS 는 그래프의 방향성과 가중치를 고려하지 않고 구축해야 하므로, 두 vertex 양쪽에서 반대 vertex 로 모두 접근이 가능해야 하도록 그래프를 구축해야 할 것이다.

DFS

DFS 는 깊이 우선 탐색 방식으로, 특정 vertex 에서 접근할 수 있는 주변 vertex 중 가장 작은 vertex 로 이동하는 방식이며 vertex 들을 stack 에 저장하여 Last In Frist Out 구조로 탐색을 진행한다. DFS 는 BFS 와 마찬가지로 그래프의 방향성과 가중치를 고려하지 않고 그래프를 구축해야 한다.

DFS_R

DFS_R 은 앞서 stack 구조로 구현한 깊이 우선 탐색 방식을 재귀적으로 호출하여 구현하는 방식이다. 그래프는 마찬가지로 방향성과 가중치를 고려하지 않고 구축한다.

KRUSKAL

KRUSKAL 알고리즘은 그래프를 최소한의 비용으로 구축할 수 있는 minimum spanning tree 를 구축하는 알고리즘이다. KRUSKAL 알고리즘은 방향성과 가중치가 있는 그래프를 구축하되, 정렬 연산은 Insertion sort 와 Quick sort 를 합하여 hybrid 한 sort 방식을 사용한다. 정렬 이후에 가중치가 작은 edge 들부터 차례대로 구축한다.

DIJKSTRA

DIJKSTRA 알고리즘은 single source all destination 최단 경로 알고리즘이다. 즉, 하나의 특정 vertex 로부터 다른 모든 vertex 로의 최단 경로를 구축하는 것이다. 최단경로를 구해야 하기 때문에 방향성과 가중치 모두 존재하는 그래프를 구축해서 진행해야 한다. 다익스트라 알고리즘에서는 가중치가 음수인 경우 정상적으로 진행할 수 없다.

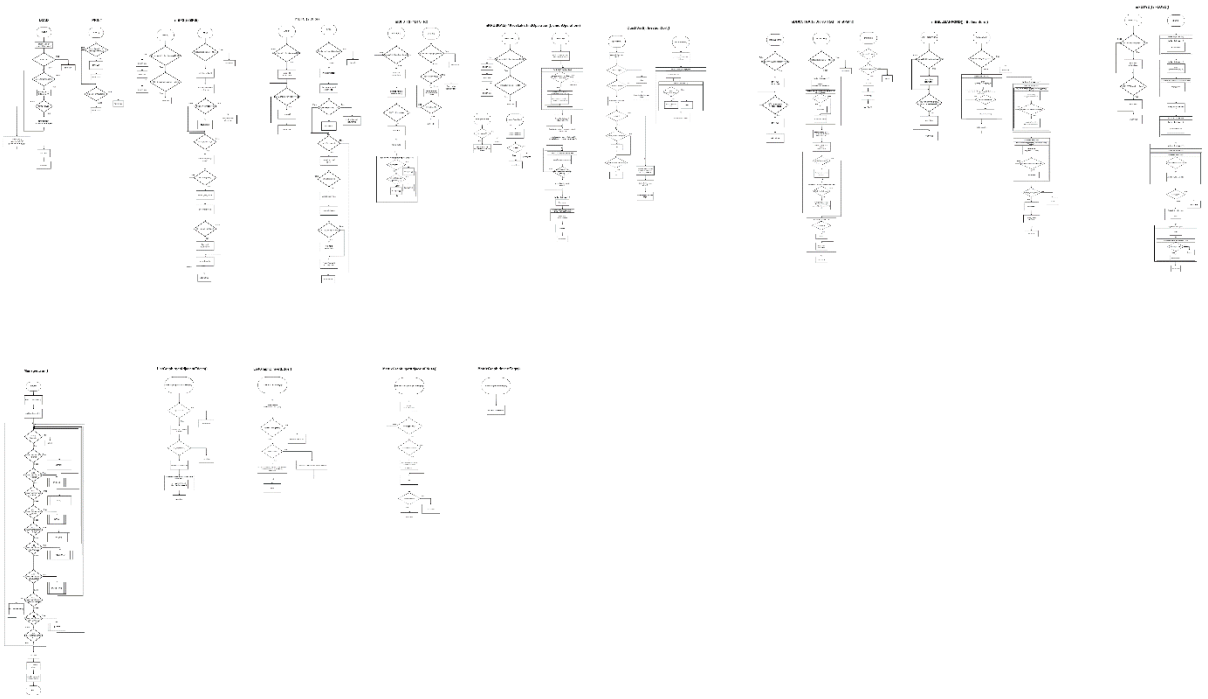
BELLMAN-FORD

BELLMAN-FORD 알고리즘은 DIJKSTRA 알고리즘과 마찬가지로 최단 경로 알고리즘이다. 하지만 음수 가중치가 존재할 수 있다는 차이점이 있다. 음수 가중치가 있는 edge 는 존재할 수 있으나, 이로 인해 음수 cycle 이 발생하면 정상적으로 진행이 불가능하다.

FLOYD

FLOYD 알고리즘은 모든 vertex 에서 다른 모든 vertex 로의 최단 경로를 구할 수 있는 알고리즘이다. 벨만 포드와 마찬가지로 음수 가중치가 존재할 수 있지만 음수 사이클이 발생할 경우 진행이 불가능하다.

Flowchart



(위 이미지는 draw.io 로 직접 그린 전체 Flowchart 입니다. 모든 함수를 포함해 세로로 flow 를 만들면 매우 길어지므로 명령어 flowchart 및 함수 서브루틴 기호를 사용하여 flowchart 를 작성했습니다.)

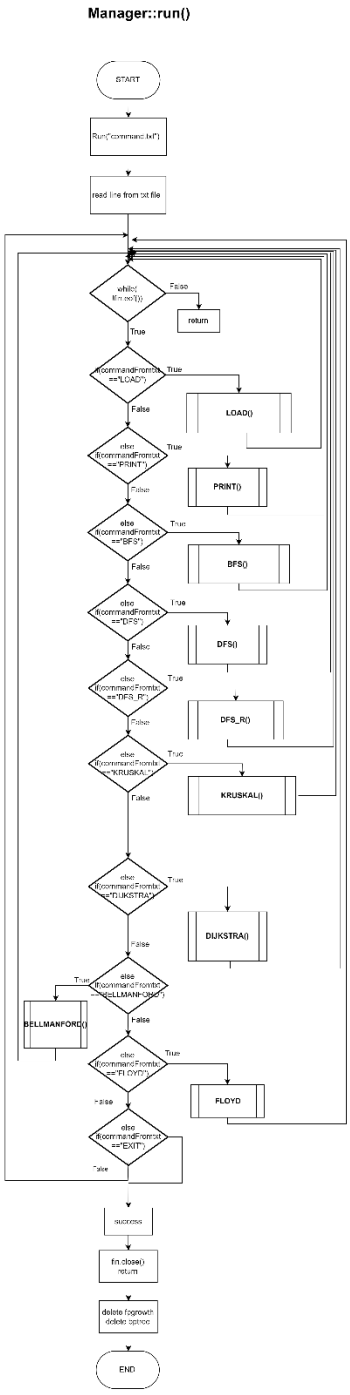
아래는 제가 직접 작성한 draw.io flowchart 링크입니다. flowchart 에 대해 자세하게 확인할 수 있습니다.

https://drive.google.com/file/d/1fOx_Jq17Yoh8lOb1kWsVxj5GMGwS5FEx/view?usp=sharing

+실제 알고리즘에 대한 자세한 설명은 대부분 Algorithm 쪽에 작성하였기 때문에 flowchart 에서는 해당 알고리즘을 구축하기 위해 설계한 flow 위주로 설명을 진행하였습니다.

+Algorithm 부분에서 이기훈 교수님의 데이터구조설계 강의자료를 통해 배운 개념과 코드를 사용한 것에 대한 언급이 나와 있습니다.

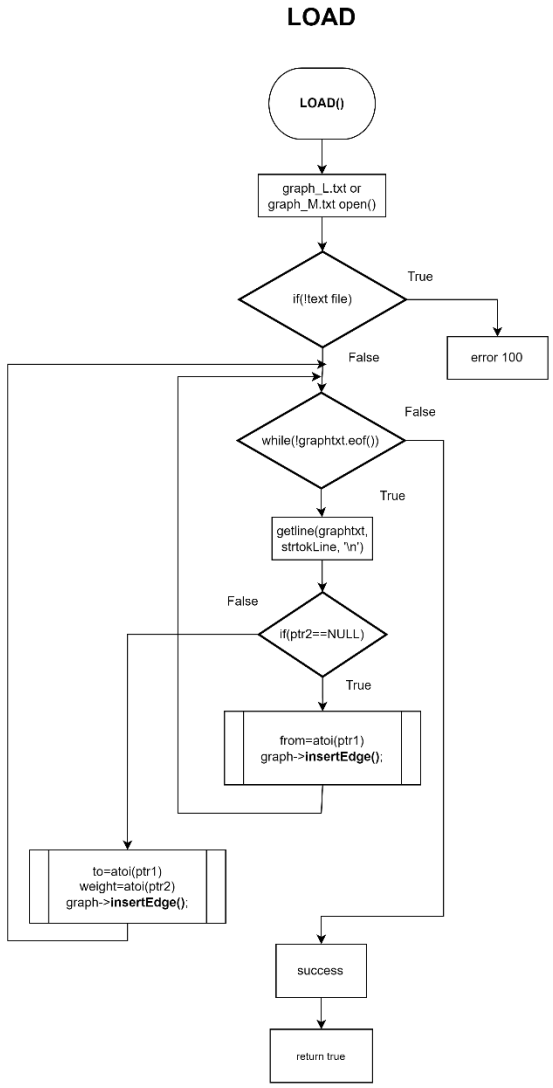
전체 흐름도-Run()



main.cpp 파일에서 제일 먼저 실행되는 Manager 클래스의 Run() 함수이다. Run() 함수는 txt 파일을 인자로 가지고 있기 때문에 command.txt 파일을 인자로 전달하여 실행한 것을 flowchart 에서 확인할 수 있다. command.txt 파일에서 getline 으로 읽어온 정보를 바탕으로 어떤 명령어가 실행될지 알 수 있다. command.txt 파일 안의 명령어는 하나만 존재하는 것이 아니므로 루프문을 통해 지속적으로 명령어의 기능을 수행하게 되는 것이다. 각 명령어마다 서브루틴

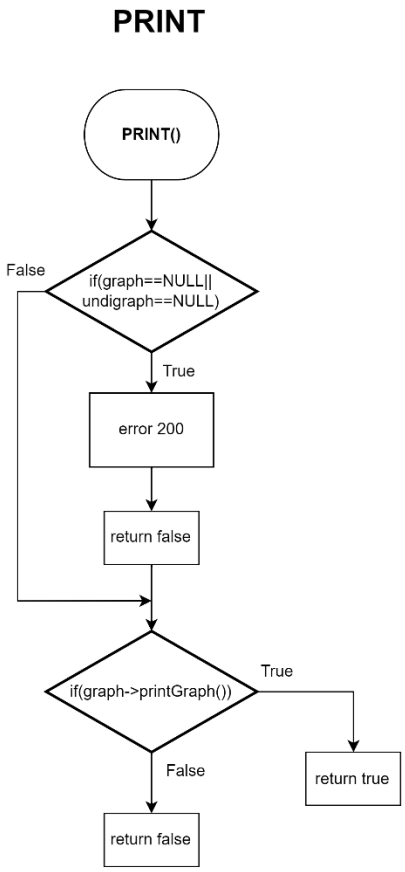
기호로 그렸기 때문에 flowchart 상에서 각 명령어에 대한 함수와 연결된다. 이를 바탕으로 마지막으로 EXIT() 함수가 실행되면 종료되도록 flowchart 의 구성을 완료하였다.

LOAD()



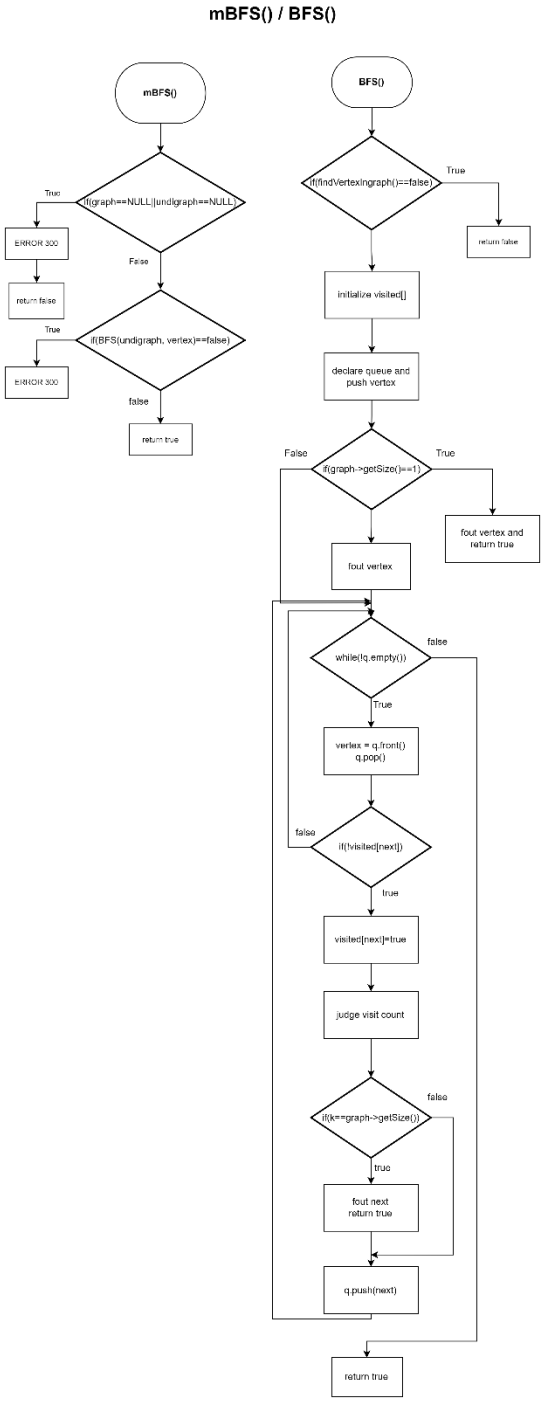
command.txt 로부터 LOAD 와 함께 list 그래프 혹은 matrix 그래프 정보가 담겨 있는 파일을 읽는다. 그래프 파일에 따라서 구축 방법이 달라지기 E 때문에 if 문으로 어떤 그래프를 구축할 지 분기를 나누어 주었다. 먼저, graph_L.txt 의 경우 파일이 존재하지 않을 경우 오류 100 처리를 해주었다. 또한 방향성이 있는 그래프, 방향성이 없는 그래프를 구축하기 위해 2 개의 graph 를 동시에 동적 할당으로 생성해주었다. 해당 알고리즘에 대한 자세한 설명은 후에 기술되어 있다. list 그래프 기준으로 구축할 시에 vertex 를 읽고, 해당 vertex 에 대해 연속적으로 이어져 있는 edge 들이 존재하는 경우도 존재하고, edge 가 없이 연결되지 않은 vertex 또한 존재하기 때문에 분기문으로 char to int(atoi)를 사용하여 strtok 를 동시에 사용함으로써 그래프를 구축하는 flow 를 구성하였다.

PRINT()



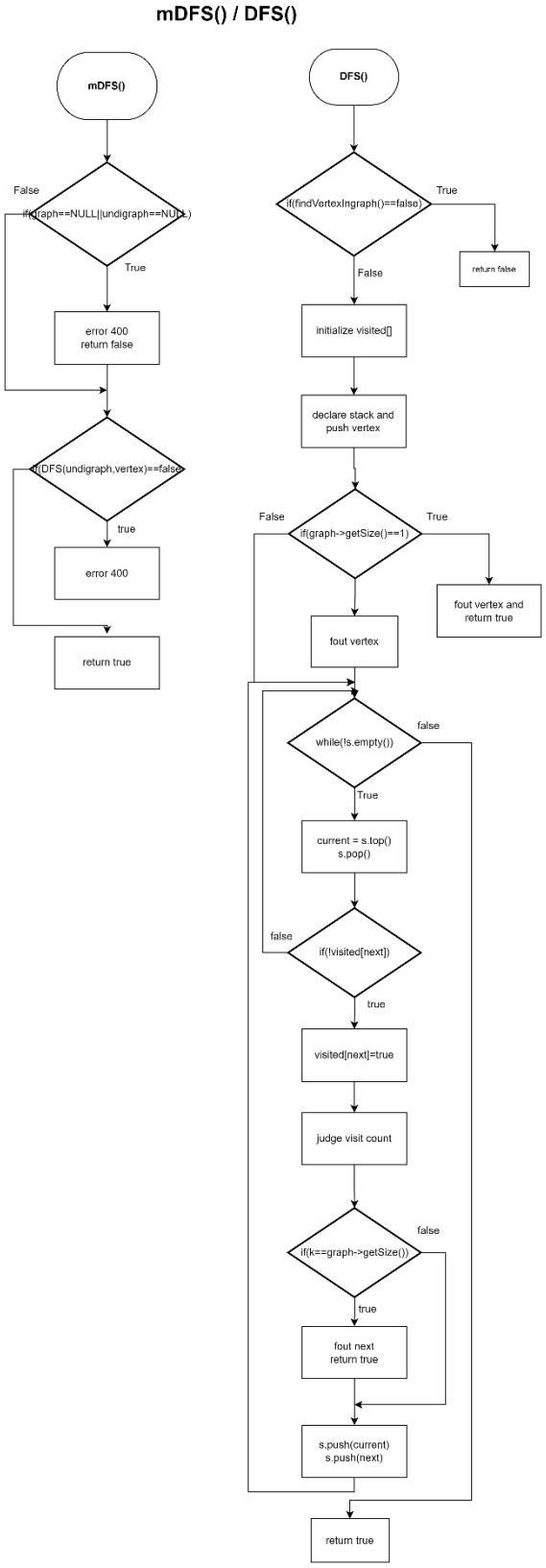
LOAD 명령어로 구축한 그래프에 담겨있는 정보들을 출력하는 명령어이다. 출력 형식은 그래프에 따라 달라지게 되는데, PRINT 를 하기 전, 그래프가 존재하지 않는 경우에는 error 200 을 출력하게 하여 예외처리를 하였다. 그래프가 존재한다면 graph->printGraph() 함수를 호출하여 각 그래프에 맞는 다형성이 적용된 출력 함수가 호출되도록 flow 를 구성하였다.

mBFS() / BFS()



mBFS() 명령어는 manager.cpp 내부에서 구축되어 있는 bfs 에 대한 함수이다. 하지만 manager.cpp 에서 mBfs() 명령어를 호출하고, GraphMethod.cpp 내부에 실제 구현된 BFS()를 호출하는 형태로 구축하였기 때문에 두 flow chart 를 동시에 그렸다. 즉, BFS()함수는 GraphMethod.cpp 내부의 구축되어 있는데, queue 자료구조를 활용하여 너비 우선 탐색을 위해 push 와 pop 을 반복함으로써 탐색할 수 있도록 flow 를 구성하였다

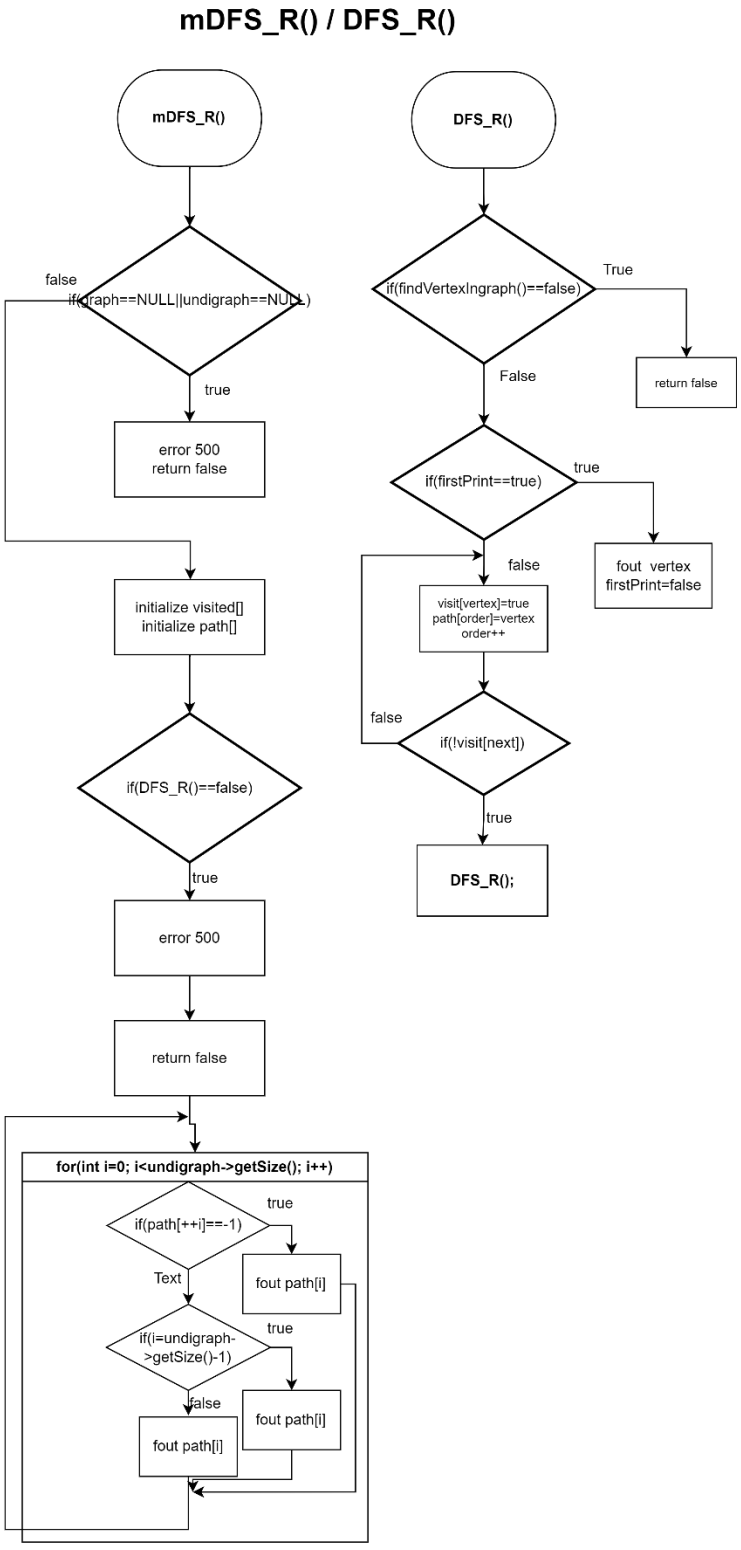
mDFS() / DFS()



mDFS()에서 호출한 DFS() 함수이다. 깊이 우선 탐색이므로, 처음에 지나갔던 vertex 를 나중에 다시 확인해야 하므로 stack 구조로 처음 들어간 것을 나중에 꺼내 쓸 수 있도록 설계하였다.

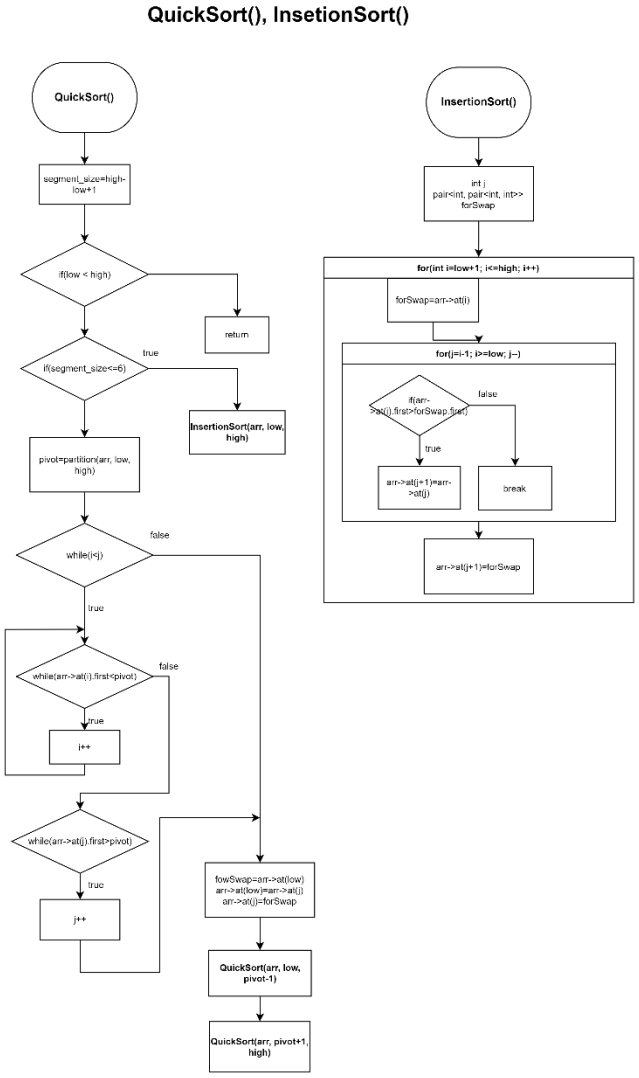
만약 stack 이 비었다면 더 이상 탐색할 vertex 가 없는 것이므로 종료될 수 있도록 flow 을 구성하였다.

mDFS_R() / DFS_R()



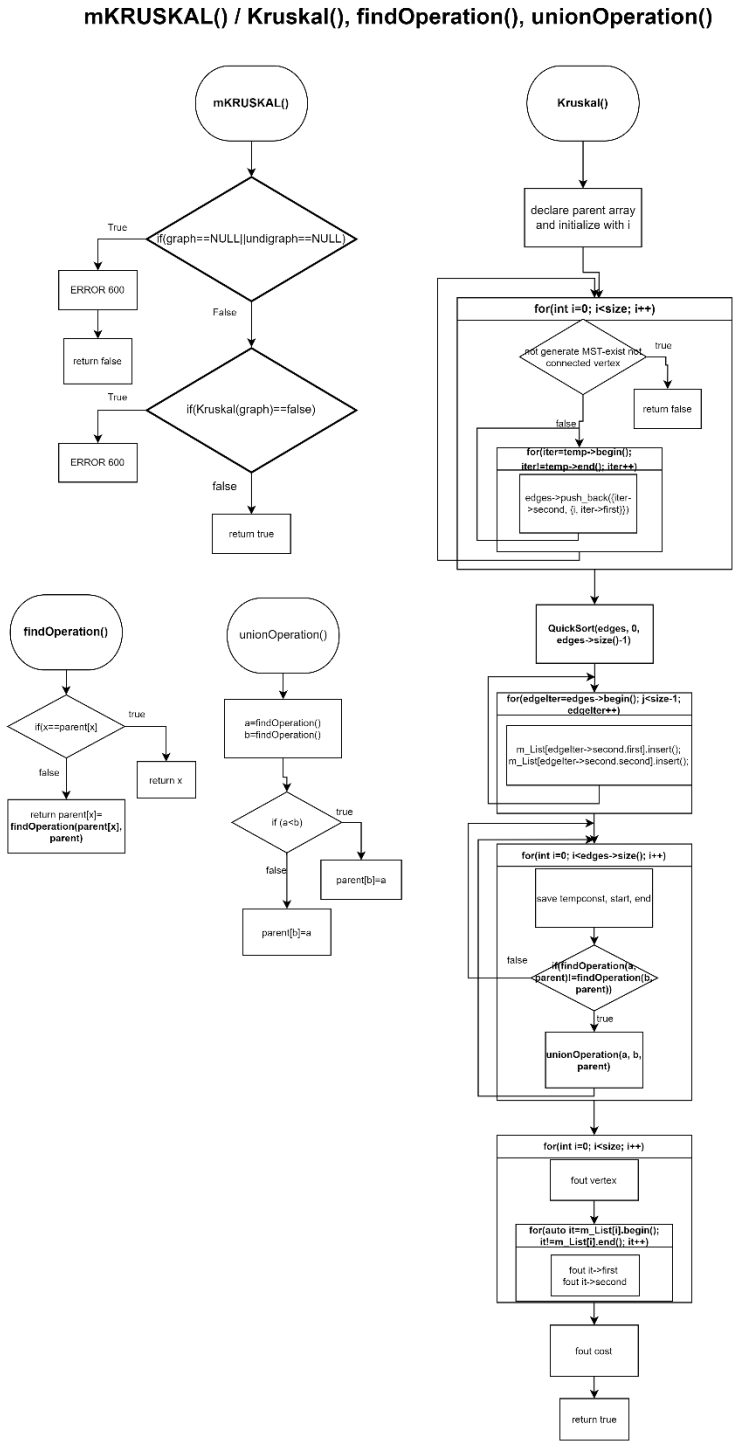
DFS_R()은 앞서 구축한 DFS 명령어를 재귀적 호출 방법을 사용하여 구조를 변경한 형태이다. 즉, 결과적으로 두 함수를 각각 실행했을 때 같은 결과가 나와야 한다. 우선 재귀적 호출로 dfs 를 구현하기 위해서 vertex 방문 여부를 확인하는 배열을 선언해주었다. 만약 방문하지 않은 vertex 라면, 해당 vertex 를 바탕으로 함수를 재귀적으로 호출한다. 재귀를 진행하면 방문 여부 확인 배열이 다 true 로 바뀌어 재귀적 호출이 종료되도록 flow 를 구성하였다.

QuickSort(), InsertionSort()



QuickSort()와 InsertionSort()는 바로 뒤에 작성할 Kruskal() 구축에서 사용할 정렬 방식이다. spec 에서 제공된 의사코드 기반으로 알고리즘을 구축하기 위해 QuickSort()를 처음으로 실행하도록 하고 segment_size 를 구해주었다. 만약 6 이하라면 InsertionSort()를 진행하고, 6 보다 크다면 QuickSort()를 그대로 진행한다. InsertionSort()의 경우 전달받은 배열과 low, high 를 기준으로 반복문을 통해 배열 arr 이 임시 변수인 forSwap 을 바탕으로 swap 이 이루어지는 것으로 flow 를 구축하였으며, QuickSort()는 pseudo code 를 기반으로 제일 왼쪽 element 를 pivot 으로 정해주어 bigelement 와 smallelement 를 교환(swap)하는 방식으로 flow 을 구축하였다.

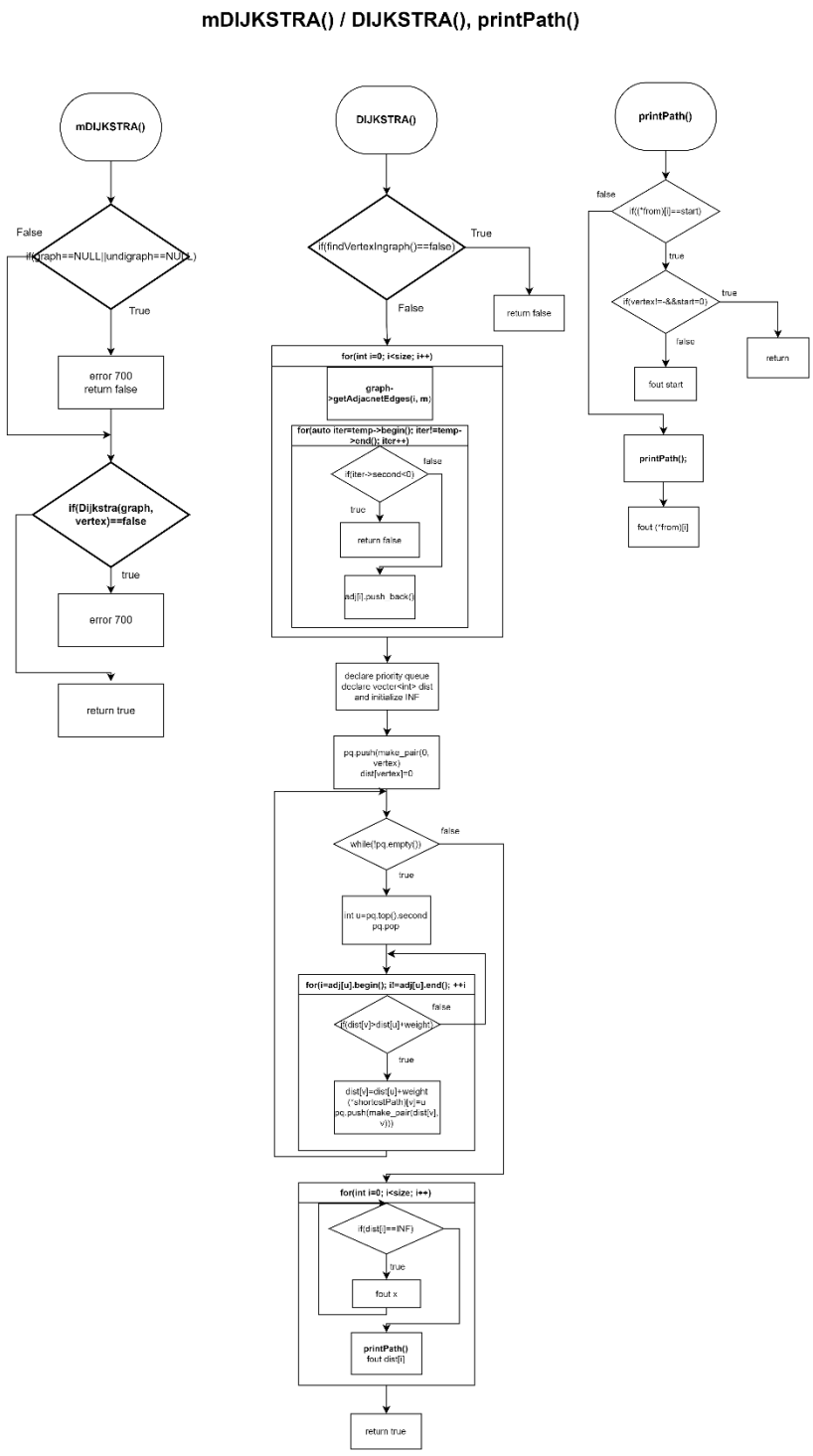
mKRUSKAL() / Kruskal(), findOperation(), unionOperation()



Kruskal()의 경우 함께 사용하는 함수들을 함께 flow 차트를 작성하였다. 우선 과제 spec 에 따르면 정렬 연산의 경우 segment size 에 따라 다른 sort 방식을 사용해야 한다. 해당 sort 방식은 Insertion Sort 와 Quick Sort 가 segment size 에 따라 다르게 실행되도록 구축되어 있으며, 이는 앞에서 flowchart 를 작성하였다. 우선 Kruskal() 실행 후 모든 edge 들을 vector 에 담고, QuickSort()를 호출하여 weight 순으로 sort 를 진행한다. sort 진행 후에 같은 disjoint set 에

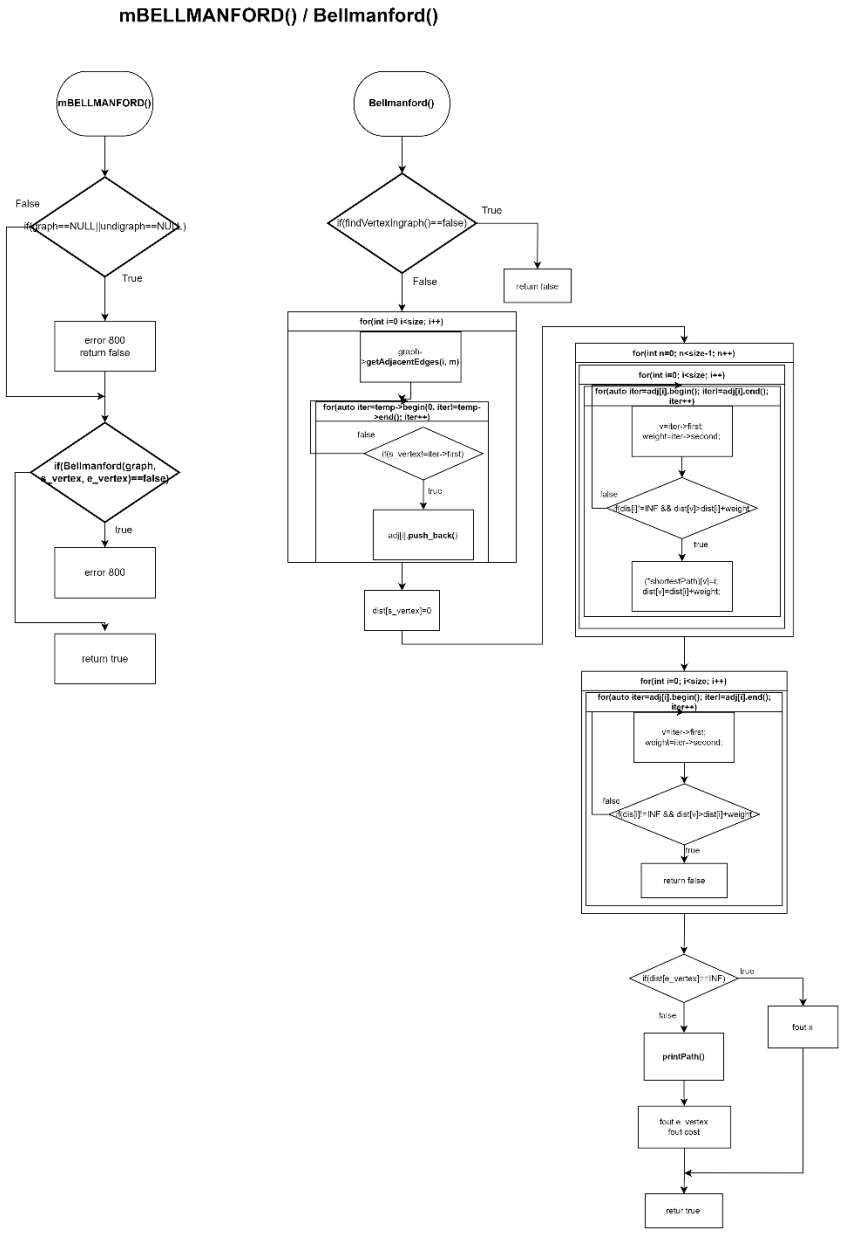
있는지 확인하는 findOperation()을 사용하고, 만약 두 vetex 에 대해 부모가 다르다면 unionOperation()으로 연산이 수행되게 하여 하나의 set 으로 구축하고 cost 를 더해주는 flow 로 구성을 완료하였다.

mDIJKSTRA() / DIJKSTRA(), printPath()



DIJKSTRA()는 하나의 vertex 로부터 나머지 모든 vertex 로의 최단 경로를 구하는 함수이다. 최단 경로를 구하기 위해 edge 에 대한 정보들을 모두 가지고 있어야 하므로 getAdjacentEdges() 함수를 사용하여 가져온 map 컨테이너를 바탕으로 list<> 구조에 push_back 하고 해당 구조를 바탕으로 min 거리 구하는 공식을 적용하여 최단 경로를 구하는 flow 를 구성하였다.

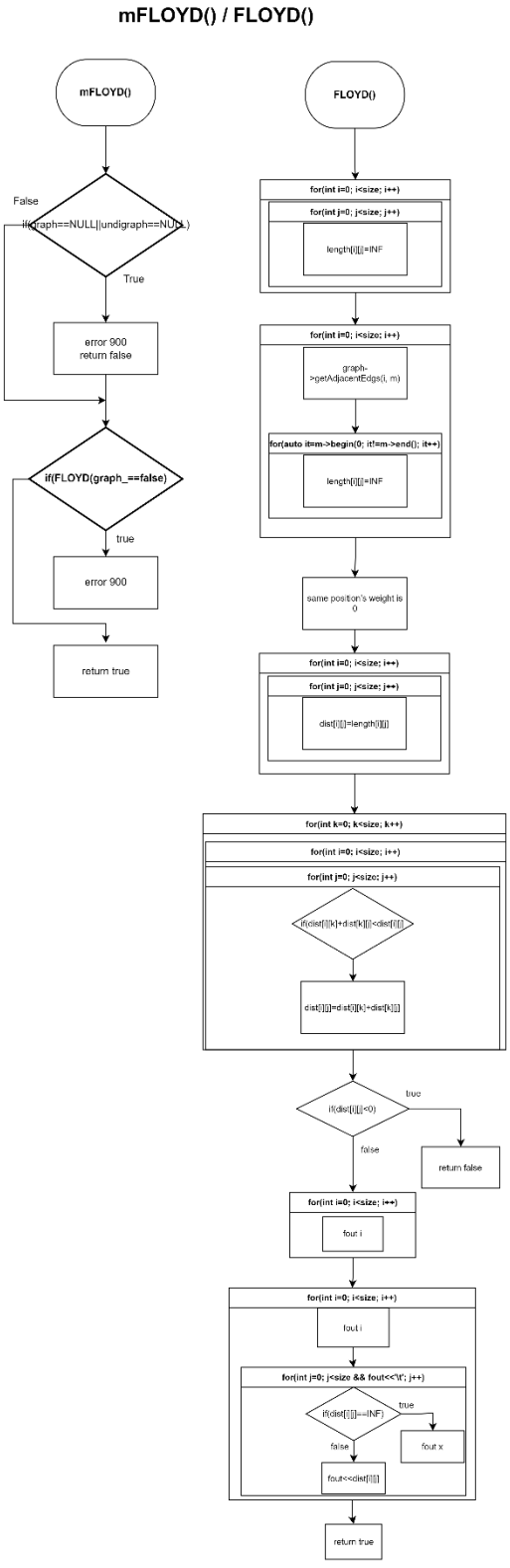
mBELLMANFORD() / Bellmanford()



Bellmanford()는 다익스트라와 마찬가지로 하나의 vertex 로부터 나머지 vertex 로의 최단 경로를 구하는 알고리즘이고, 음수 weight 가 존재할 수 있으며, 음수 cycle 을 존재하면 안된다. 따라서 다익스트라와 마찬가지로 map 컨테이너에 인접한 edge 들을 가져와 저장하여 해당 vertex 로의 최단 길이가 담겨있는 dist[] 배열과 함께 최단 경로 구하는 공식으로 dist[i]!=INF &&

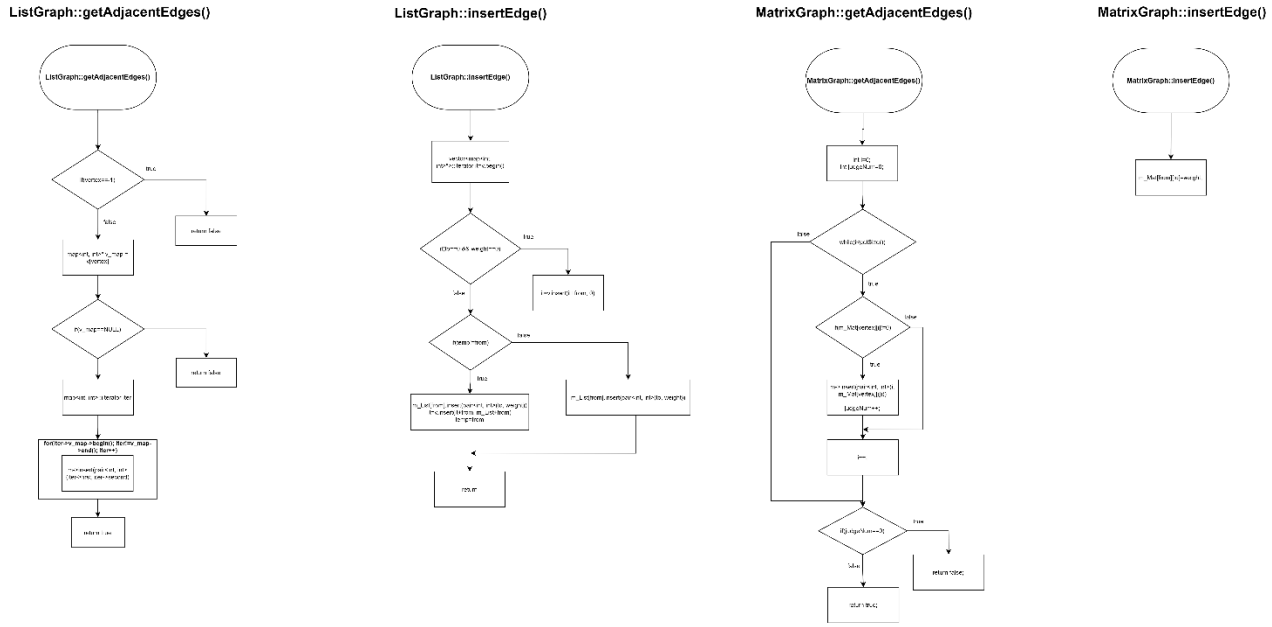
dist[v]>dist[i]+weight 를 적용하여 flow 를 구축하였다. INF 는 0x3f3f3f3f 값으로 define 하였으며 해당 값은 이기훈 교수님의 데이터구조설계 강의자료를 바탕으로 저장해주었다.

mFLOYD() / FLOYD()



FLOYD()는 모든 vertex 에서 다른 모든 vertex 로의 최단 경로를 구하는 알고리즘을 작성한 함수이다. dist[i][k] + dist[k][j] < dist[i][j]일 때 최단 경로를 update 하는 구조로 작성하였다. 해당 공식을 적용하기 전 과정은 앞서 다익스트라와 벨만포드와 동일하기 때문에 생략하였다.

ListGraph::getAdjacentEdges() / ListGraph::InsertEdge() / MatrixGraph::getAdjacentEdges() / MatrixGraph::insertEdge()



이번 프로젝트에서는 그래프를 구축하는 구조가 총 2 개이다. 따라서 그래프 연산을 수행할 때 두 그래프에서 인접한 edge 들을 가져오는 함수가 필요한데 해당 함수가 getAdjacentEdges()이다. listGraph 에서의 getAdjacentEdges 는 처음에 vertex 값이 음수가 들어오면, 존재하지 않는 vertex 이므로 return false 를 하여 manager.cpp 에서 error 를 출력하도록 해주었고, vector 로부터 인접 edge 들을 가져와 getAdjacentEdges()에 인자로 전달한 map 컨테이너에 저장되도록 하여 구성하였고, MatrixGraph 에서의 getAdjacentEdges 또한 map 컨테이너에 저장되도록 하는데, MatrixGraph 는 2D array 구조이므로 인덱스를 참조하여 map 컨테이너에 저장하는 방법으로 flow 를 구축하였다.

Algorithm

방향성 있는 그래프, 방향성 없는 그래프 구축 알고리즘

이번 프로젝트에서는 그래프 연산에 따라 방향성이 있는 그래프 혹은 방향성이 없는 그래프를 사용한다. 그런데 그래프 정보가 담겨 있는 파일에서는 방향성이 있는 것처럼 고려가 되어 있기 때문에 방향성이 없는 그래프를 구축하기 위해서는 별도의 작업이 필요하다. 여러가지 방법이 있겠지만, 나의 경우에는 그래프를 하나만 생성하는 것이 아닌, 방향성이 없는 그래프와 방향성이 있는 그래프를 각각 생성하여 2 개의 그래프를 만드는 것이다. 방향성이 없는 그래프는 txt 파일에서 읽어온 그대로 구축하면 되고, 방향성이 없는 그래프의 경우 뱡향성이 1 -> 2 로 주어졌을 때 이를 그대로 삽입 후, 출발 vertex 와 도착 vertex 를 반전시킴으로써 2 -> 1 또한 삽입해주었다. 이렇게 구현하게 되면 vertex 1 에서도 vertex 2 로 이동할 수 있고, vertex 2 에서도 vertex 1 로 이동할 수 있게 되므로 사실상 방향성이 없는 그래프가 구축되는 것이다.

edge 가 없는 vertex 구축 알고리즘

과제 spec 2 페이지에 따르면 List 그래프에서 edge 가 없는 vertex 가 존재하는 경우가 있을 수 있다는 것을 알게 되었다. 따라서 edge 가 없는 vertex 또한 구축을 해주어야 하는데 LOAD 에서 이를 처리해주었다. 이를 구축하는 알고리즘은 우선 strtok 로 인자들을 잘라오는 것이 핵심이다. 인자를 잘라올 때 만약 하나만 존재한다면 해당 vertex 가 있다는 의미이므로 ptr2==NULL 인 경우로 들어가 insertEdge 를 한다. vector 의 경우 모든 값을 채우지 않아도 빈 값으로 들어가지기 때문에 vertex 만 넣을 수 있게 되는 것이다. 이후 파일을 읽어 2 4 처럼 weight 가 4 이고 도착 vertex 가 2 인 경우에는 ptr2!=NULL 인 경우로 들어가 insertEdge 를 한다. 해당 경우에는 이미 전에 읽으면서 ptr1 값이 저장되어 있는 상태이기 때문에 시작 vertex, 도착 vertex, weight 를 한 번에 삽입할 수 있게 된다. 따라서 edge 가 없는 vertex 는 ptr2==NULL 로 insertEdge 가 되기 때문에 도착 vertex, weight 가 없는, 즉 edge 가 없는 vertex 로써 vector 에 저장되게 되는 것이다.

QuickSort & InsertionSort 정렬 알고리즘(Hybrid)

이번 과제에서는 크루스칼 알고리즘에 대한 정렬 연산은 슈도 코드에 따라 QuickSort 를 호출하되, QuickSort 내부에서 segment_size 에 따라 InsertionSort 로 진행할 지, QuickSort 로 진행할 지 결정하게 된다. 해당 알고리즘을 사용하는 이유는 기본적으로 QuickSort 가 시간복잡도 상 더 효율적이긴 하지만 일정 segment 이하에서는 InsetionSort 가 효율적인 경우가 존재하기 때문에 같이 hybrid 로 사용하는 것이다. InsertionSort 의 경우 인자로 배열, 최소 Index, 최대 Index 를 넣어주었다. 함수 내부에서는 2 중 for 문을 사용해 배열의 특정 위치의 값이 바꿀 값보다 큰 경우에 swap 을 완료하는 형태로 알고리즘을 작성하였고, QuickSort 의 경우 이기훈 교수님의 데이터 구조 설계 과목 시간에 배운 알고리즘을 바탕으로 작성하였다. 그런데 hybrid 로

진행하려고 하니, stl swap 함수를 사용하게 되면 정렬이 제대로 진행되지 않아서 기존에 임시로 저장할 곳을 따로 선언해 swap 을 하는 방식으로 진행해주었다. 또한 pivot 을 기준으로 재귀적으로 호출해야 하기 때문에 왼쪽 부분에 대해, 오른쪽 부분에 대해 따로 QuickSort 를 재귀적으로 호출함으로써 과제 spec 에서 주어진 의사코드에 기반하여 알고리즘을 구축 완료하였다.

BFS

BFS 는 STL queue 자료 구조를 활용하였다. 함수 초기의 예외처리로, BFS 를 진행할 vertex 가 그래프 내에 존재하지 않는 경우 return false 를 통해 manager 로 돌아가서 error 를 출력하도록 해주었다. 만약 에러가 발생하지 않을 경우 queue 구조에 vertex 를 push 하고, queue 가 empty 일 때까지 방문한 노드를 체크하는 배열을 사용하여 출력을 진행함으로써 너비를 우선적으로 탐색할 수 있게 해주었다. 해당 과정은 이기훈 교수님의 데이터 구조설계 강의자료를 참고하여 구현을 완료하였다.

DFS

DFS 는 STL stack 자료 구조를 활용하였다. BFS 와 마찬가지로 함수 초기의 예외처리로, vertex 가 존재하지 않는 경우를 처리해주었으며 시작 vertex 를 stack 에 넣고 방문했음을 의미하도록 true 로 설정한다. 이후 해당 stack 의 top 에 있는 vertex 에 대해 방문하지 않은 인접 vertex 가 존재한다면 해당 vertex 를 stack 에 넣고 마찬가지로 방문했음을 의미하도록 true 로 처리해주었다. vertex 출력의 경우 visited 배열에 true 로 바뀔 때 출력하게 함으로써 깊이를 우선적으로 탐색할 수 있도록 알고리즘 구축을 완료하였다. 해당 과정은 이기훈 교수님의 데이터 구조설계 강의자료를 참고하여 구현을 완료하였다.

DFS_R

DFS_R 은 stack 자료 구조 대신 재귀적인 호출을 기반으로 알고리즘을 작성하였다. DFS_R 을 처음에 호출하였던 Manager.cpp 에서 경로를 저장하기 위한 배열과 시작 vertex 를 DFS_R 로 처음에 넘겨준다. 함수 초기 예외 처리를 마치고, 해당 vertex 와 인접한 edge 들을 map 컨테이너에 담아와서 만약 인접한 edge 가 방문하지 않은 상태면 재귀적으로 DFS_R 을 다시 호출하여 방문과 동시에 경로 배열에는 vertex 를 저장하고, 방문 체크 배열에는 true 로 설정해줌으로써 진행을 해주었다. 해당 과정을 반복 후, 더 이상 vertex 가 없다면 함수가 종료되도록 알고리즘을 작성하였다.

KRUSKAL

크루스칼 알고리즘은 disjoint set 과 앞에서 설명한 sort 알고리즘을 바탕으로 알고리즘을 작성하였다. 먼저, disjoint set 에 이용하기 위한 parent 배열을 만들고 초기화 해주었다. 크루스칼은 그래프에서 cost 가 가장 적은 edge 부터 연결하여 MST 를 구축하는 것이므로 vector 를 새로 생성하여 모든 edge 들에 대한 정보를 넣어주었다. 그리고 구현한 sort 알고리즘을 바탕으로 cost 를 오름차순으로 정렬하였다. 정렬 후에는 m_List 에 삽입을 진행하고 난 후,

cycle 이 생기는 것을 MST 상에서 방지하기 위해 union 연산을 구현한 unionOperation() 함수와 find 연산을 구현한 findOperation() 함수를 사용하였다. 만약 find 연산으로 두 vertex 에 대해 같은 값이 저장되었다면, cycle 이 발생하는 것이므로 find 연산으로 두 vertex 에 대해 다른 결과가 나왔을 때 union 연산을 진행함으로써 cycle 이 발생하지 않도록 MST 를 생성해줌과 동시에 union 이 진행될 때 cost 를 합산하여 MST 생성 및 MST 생성 시의 cost 를 구하였고, 이를 출력함으로써 알고리즘을 구축하였다.

DIJKSTRA

다익스트라 알고리즘은 시간 복잡도를 고려하여 우선 순위 큐를 사용하여 진행하였고, 해당 과정은 이기훈 교수님께 배운 데이터 구조 설계 강의자료를 바탕으로 진행하였다. 우선 한 vertex 에서 나머지 다른 vertex 들에 대한 최단경로를 구해야 하기 때문에 별도로 해당 경로를 저장하기 위한 vector 를 선언해주었고, 각 vertex 에 인접한 vertex 들은 list 자료구조를 사용하여 각각 저장해주었다. 저장 후에는 pq(우선 순위 큐)에 시작값을 push 하는 것으로 시작하여 pq 가 empty 일때가지 계속 수행하게 된다. 다익스트라의 최소 경로를 update 하기 위한 조건은 dist[v] > dist[u] +weight 일 때, 기존의 v 로의 경로보다 u 를 거쳐서 갈 때의 값이 작게 되면, 새로운 weight 로 거리를 갱신해주고 pq 에 다시 삽입을 해줌으로써 진행을 완료하였다. 출력 포맷의 경우 startvertex 는 출력하지 말아야 하기 때문에 같은 경우에는 continue 로 출력하지 않았으며 만약 거리가 초기화 값과 같은 INF 와 같게 된다면 기준 vertex 에서 도달할 수 없는 vertex 이므로 x 를 출력하게 해줌으로써 알고리즘 구축을 완료하였다.

BELLMANFORD

벨만포드 알고리즘은 다익스트라와 마찬가지로 하나의 vertex 에 대하여 다른 모든 vertex 로의 최단 경로를 구하되, 음수 값을 가진 weight 가 존재할 수 있다. 초기 구조는 다익스트라와 동일하게 각 vertex 에 인접한 vertex 들은 list 를 사용하여 저장해주었다. 또한 최단 경로를 갱신하며 저장하는 dist 배열을 INF 로 초기화해주었다. s_vertex 부터 시작할 때, 저장된 인접 vertex 까지의 거리보다 현재 vetex 를 거치고 인접 vertex 를 도달하는게 더 짧을 경우 값을 update 하도록 하였다. 해당 조건은 dist[i]!=INF && dist[v]>dist[i]+weight 로 설정해주었다.

FLOYD

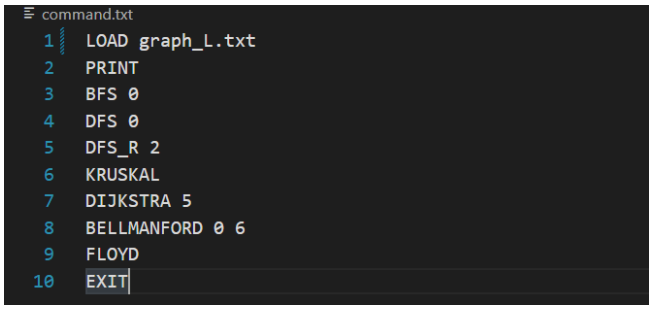
플로이드 알고리즘은 최단 경로 알고리즘이며, 모든 vertex 에서 다른 모든 vertex 로의 최단 경로를 구할 수 있다. 음수 weight 가 존재하며, 음수 cycle 이 존재하면 안 된다. length[][] 배열에 초기값으로 INF 로 설정하고, getAdjacentEdges()를 통해 각 vetex 에 인접한 edge 들에 대한 정보를 length 에 전부 담아주었다. 또한 자기 자신으로 가는 경우는 0 이므로 length[i][i]=0 으로 설정해주었다. dist[][] 배열에 length[][]를 저장하고, dist[i][k] +dist[k][j] < dist[i][j] 조건으로 최단 경로를 update 하도록 해주었다. update 를 전부 완료한 상황에서 만약 음수 cycle 이 발생하면 예외처리를 하기 위해 dist[i][i]<0 조건을 수행해주었다. 이후 출력 과정은 Matrix 그래프의 출력

함수의 스켈레톤 코드를 사용함으로써 출력 format 동안 동일하게 구성해줌으로써 알고리즘을 작성하였다.

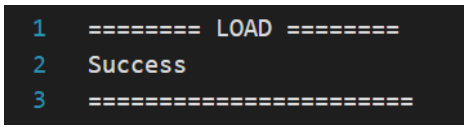
Result Screen

작성에 앞서 result screen 을 통한 검증을 위해 graph_L.txt 로 진행을 하였고, 제안서의 예시처럼 아래와 같이 구성하였다.

log.txt 에 출력되는 결과를 얻기 위해 작성한 command.txt 제안서와 동일하게 아래와 같다.



LOAD Result Screen (LOAD)



위는 command.txt 의 line 1 을 실행한 LOAD 명령어의 Result Screen 이다.

graph_L.txt 파일이 존재하기 때문에 ERROR 100 이 출력되지 않고, LOAD 의 기능을 성공적으로 구현하여 Success 가 출력된 것을 확인할 수 있다.

PRINT Result Screen (PRINT)

```
5   ========PRINT========
6   [0] -> (1,6) -> (2,2)
7   [1] -> (3,5)
8   [2] -> (1,7) -> (4,3) -> (5,8)
9   [3] -> (6,3)
10  [4] -> (3,4)
11  [5] -> (6,1)
12  [6] -> (4,10)
13  ====================
```

위는 command.txt 의 line 2 를 실행한 PRINT 명령어의 Result Screen 이다.
앞서 LOAD 로 구축한 그래프의 구축 정보가 정상적으로 출력되는 것을 확인할 수 있다. 해당 그래프를 바탕으로 연산을 수행하게 된다.

BFS Result Screen (BFS 0)

```
15  ======== BFS ========
16  startvertex: 0
17  0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
18  ====================
```

위는 command.txt 의 line 3 을 실행한 BFS 명령어의 Result Screen 이다.
너비 우선 탐색 방식인 BFS 가 startvertex 가 0 으로 정상적으로 출력되고, 탐색 경로 또한 정상적으로 출력 format 에 맞게 출력되는 것을 확인할 수 있다.

DFS Result Screen (DFS 0)

```
20  ======== DFS ========
21  startvertex: 0
22  0 -> 1 -> 2 -> 4 -> 3 -> 6 -> 5
23  ====================
```

위는 command.txt 의 line 4 을 실행한 DFS 명령어의 Result Screen 이다.
깊이 우선 탐색 방식인 DFS 가 startvertex 가 0 으로 정상적으로 출력되고, 탐색 경로 또한 정상적으로 출력 format 에 맞게 출력되는 것을 확인할 수 있다.

DFS_R Result Screen (DFS_R 2)

```
25  ======== DFS_R ========
26  startvertex: 2
27  2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
28  ====================
```

위는 command.txt 의 line 5 을 실행한 DFS_R 명령어의 Result Screen 이다.
앞선 DFS 와 다르게 startvertex 는 2 이며, 탐색 경로 또한 정상적으로 출력 format 에 맞게 출력되는 것을 확인할 수 있다. 또한 위의 DFS 명령어와 동일한 연산 결과가 나와야 하기 때문에

별도로 startvertex 를 동일하게 진행하였는데, 같은 결과가 나오게 된 것을 통해 DFS, DFS_R 연산이 정상적으로 구현이 되었다는 것을 알 수 있었다.

KRUSKAL Result Screen (KRUSKAL)

```
30  ====== Kruskal =======
31  [0] 2(2)
32  [1] 3(5)
33  [2] 0(2) 4(3)
34  [3] 1(5) 4(4) 6(3)
35  [4] 2(3) 3(4)
36  [5] 6(1)
37  [6] 3(3) 5(1)
38  cost: 18
39  ====================
```

위는 command.txt 의 line 6 을 실행한 KRUSKAL 명령어의 Result Screen 이다.
그래프에 구축된 정보를 바탕으로 minimum spanning tree 를 구축한 후에 해당 MST 의 구성 edge 들을 weight 와 함께 오름차순으로 정상적으로 출력하는 것을 확인할 수 있다. 또한 MST 구성 후 최종 cost 를 얻은 것을 확인할 수 있다.

DIJKSTRA Result Screen (DIJKSTRA 5)

```
41  ========= Dijkstra =========
42  startvertex: 5
43  [0] x
44  [1] x
45  [2] x
46  [3] 5 -> 6 -> 4 -> 3 (15)
47  [4] 5 -> 6 -> 4 (11)
48  [6] 5 -> 6 (1)
49  ==========================
```

위는 command.txt 의 line 7 을 실행한 DIJKSTRA 명령어의 Result Screen 이다.
startvertex 자신에 대한 것은 출력하지 않고, 도달할 수 없는 vertex 는 x 로 정상적으로 출력되는 것을 확인할 수 있으며, 도달할 수 있는 경우 최단 경로와 함께 cost 를 정상적으로 출력하는 것을 확인할 수 있다.

BELLMANFORD Result Screen (BELLMANFORD 0 6)

```
51  ====== Bellman-Ford ======
52  0 -> 2 -> 5 -> 6
53  cost: 11
54  ==========================
```

위는 command.txt 의 line 8 을 실행한 BELLMANFORD 명령어의 Result Screen 이다.

0 부터 6 까지의 최단 경로와 함께 계산된 cost 또한 정상적으로 출력하고 있는 것을 확인할 수 있다.

FLOYD Result Screen (FLOYD)

```
56  ======== FLOYD ========
57      [0] [1] [2] [3] [4] [5] [6]
58  [0] 0   6   2   9   5   10  11
59  [1] x   0   x   5   18  x   8
60  [2] x   7   0   7   3   8   9
61  [3] x   x   x   0   13  x   3
62  [4] x   x   x   4   0   x   7
63  [5] x   x   x   15  11  0   1
64  [6] x   x   x   14  10  x   0
65  ======================
```

위는 command.txt 의 line 9 을 실행한 FLOYD 명령어의 Result Screen 이다.

제안서 8 페이지의 표 4. 출력 형식에서와 다른 것은 제안서의 출력 형식의 오타이기 때문이다.

따라서 수동으로 계산한 결과 위의 result screen 과 동일한 결과를 얻은 것을 확인할 수 있었다.

EXIT (EXIT)

command.txt 의 line 10 을 실행한 EXIT 명령어이다.

이번 EXIT 명령어는 이전 프로젝트와는 다르게 출력 형식이 제안서의 표 5. 동작 예시에 따르면 출력을 하고 있지 않아 이와 동일하게 아무 출력도 하지 않고, 메모리 누수 방지를 위한 동적 할당 해제만 진행하였다.

Consideration

이번 DS 3 차 프로젝트를 통해 두 종류의 그래프에 대한 구축을 완료하고, 그래프 연산을 직접 구현함으로써 과제 spec 에 만족하도록 성공적으로 모든 명령어를 구현 완료 및 예외 처리를 하였다. 이번 과제를 처음 시작하면서 처음 겪었던 어려운 점은 두 형태의 그래프가 존재하고 있는 상황에서 BFS, DFS, DFS_R, KRUSKAL, DIJKSTRA, BELLMAN-FORD, FLOYD 총 7 개의 그래프 연산을 수행하기 위해서 LIST 그래프 전용으로 7 개의 알고리즘 작성 + MATRIX 그래프 전용으로 7 개의 알고리즘 작성으로 총 14 개의 알고리즘 코드를 작성해야 하는지에 대한 것이었다. 하지만 프로젝트 구조 상에서 상당히 비효율적인 방식이라는 생각이 들었고, 주어진 스켈레톤 코드에 대해서 분석을 시작하였다. 그러던 중, MatrixGraph.cpp 와 ListGraph.cpp 에서 공통적으로 getAdjacentEdges 라는 함수를 발견하였다. 해당 함수를 직역하자면 인접한 edge 들을 가져오는 함수라고 생각했다. 그리고 매개변수로 map 컨테이너가 <int, int> 형태로 주어진 것을 통해 각 그래프에서 해당 코드를 작성하면 map 컨테이너를 통해 동일한 인접한 값들이 가져와질 것으로 예측하기까지 초기에 많은 시간이 소요되었다. 두 번째 이슈로는 DFS_R() 명령어에 대한 것이었다. DFS_R()의 구조상 재귀적으로 함수를 호출해야 하기 때문에 재귀적으로 출력을 하려고 했었다. 하지만 어떤 경우에는 끝 노드에 ->가 같이 출력되어서 나오고, 이를 수정하면 끝 노드에는 ->가 출력되지 않는데 앞의 노드와 같이 붙어서 출력되는 것이었다. 즉, 1 -> 2 -> 3 -> 4 -> 5 -> 로 출력이 되는 것을 수정하고 나니 1 -> 2 -> 3 -> 45 로 출력이 되었다. 어느 한 쪽을 고치면 다른 한 쪽에 영향이 가게 되어서 들게 된 생각은 manager 에서 DFS_R 을 호출하는 부분에서 경로만을 저장하기 위한 배열을 DFS_R 로 전달해주는 것이었다. 즉, DFS_R 에서 경로가 생길 때마다 해당 vertex 를 배열에 저장하고 난 후 DFS_R 이 종료되면 다시 manager 에서 DFS_R 을 호출하는 부분으로 돌아와 해당 배열을 출력해주는 것으로 해결할 수 있었다. 재귀 구조상 출력 부분에 어려움을 많이 겪게 되었지만, 재귀 내부 구조에서 해결하기 힘든 경우에는 임시로 저장 후, 재귀가 끝난 후에 출력을 진행해도 된다는 아이디어를 얻게 되었다. 마지막으로 가장 큰 이슈는 단기간에 배운 MST 알고리즘과 Shortest path 알고리즘의 혼동이었다. 데이터구조설계 수업 시간에 배운 BFS 와 DFS 까지는 혼동이 되지 않았으나 이후에 프림, 크루스칼에 대한 MST 알고리즘을 배우고 나서 다익스트라, 벨만 포드, 플로이드까지 전부 배우고 나니 5 가지 알고리즘에 대한 개념이 혼동이 오기 시작했다. 그러다 보니 내가 작성한 코드에 대해 시간이 좀 지나 다시 보았을 때 이게 다익스트라 코드인지, 벨만 포드 코드인지 헷갈렸었다. 따라서 단기간에 배운 개념을 코드로 프로젝트를 수행하여 개념의 기반이 단단하게 자리 잡히지 않은 상태이다. 게다가 해당 알고리즘들은 추후에 코딩 테스트에서도 자주 출제가 되는 핵심 알고리즘들인 만큼 종강 이후에 이를 복습할 계획을 수립하였다. 마지막으로, 알고리즘 구현에 대한 아이디어는 이기훈 교수님의 데이터 구조 설계 강의자료에서 공부한 내용이 많이

적용되었다. 따라서 이후 복습 과정에서 시간 복잡도를 고려하여 다른 방식으로 처음부터 혼자 구현하는데 많은 시간을 들일 것으로 예상된다.