Data Structure Lab. Project #3

과목명	데이터구조실습
담당 교수	공진흥
학과	컴퓨터정보공학부
학번	2021202078
이름	최경정

1. Introduction

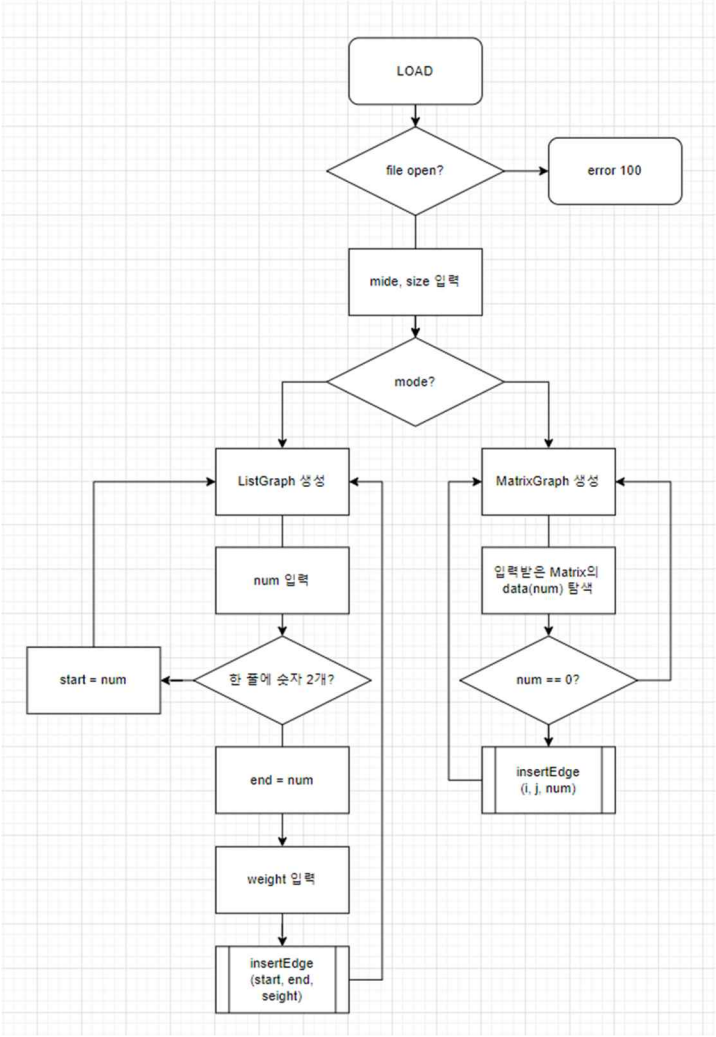
본 프로젝트에서는 그래프를 이용한 다양한 연산 알고리즘을 C++로 구현하는데 목적이 있다. 이 프로그램은 우선 그래프의 정보가 저장된 텍스트 파일 두 종류를 이용해 명령어에 맞게 그래프를 구현한다. 그래프 데이터는 방향성과 가중치를 저장하고 있으며, 데이터의 형태에 따라 맵 형태인 List 그래프와 행렬 형태인 matrix 그래프로 나뉜다. 이렇게 생성한 그래프의 특성과 명령어에 따라 BFS, DFS, Kruskal, Dijkstra, Bellman-Ford 그리고 FLOYD 연산을 수행한다.

우선 BFS와 DFS, DFS_R은 그래프의 방향성과 가중치를 고려하지 않고 비방향 그래프로 하여 그래프 순회와 탐색을 진행한다. Kruskal 알고리즘은 Minimum Spainning Tree(최소 신장 트리)를 만드는 방식으로 진행하며, 방향성은 없지만 가중치를 고려하는 그래프 환경에서 수행한다. Dijkstra 알고리즘은 입력받은 정점을 출발점으로 하여 다른 모든 정점을 도착점으로 하는 최단 경로를 알아내는 알고리즘으로, 방향성과 가중치를 모두 고려하는 그래프 환경에서 연산을 수행하야 한다. 그러나 다익스트라 알고리즘은 가중치가 양수인 경우에만 적용이 가능하기 때문에 음수 사이클이 발생할 경우 Bellman-Ford 알고리즘을 사용해야 한다. 벨만 포드 알고리즘에서는 음수 사이클이 발생한 경우 에러 출력을 진행하고, 나머지 경우에는 최단 경로와 그 거리를 구한다.

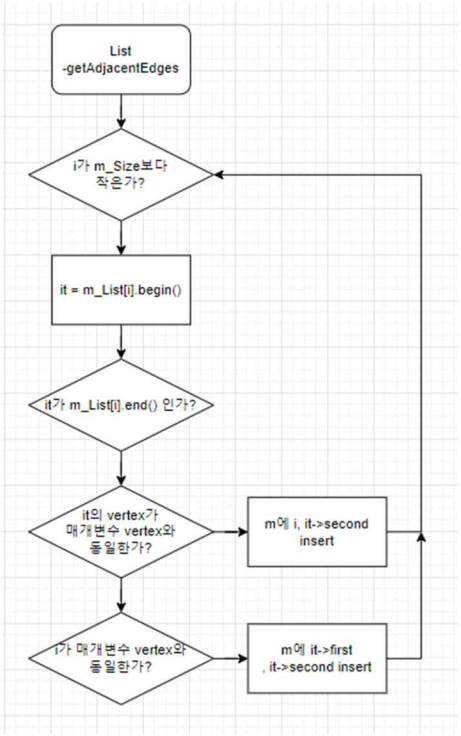
프로그램의 동작은 command.txt 파일에서 입력받는 명령어와 부가 조건에 따라 각각의 기능을 수행하는 방식으로 이루어지고, 연산의 결과는 출력 파일(log.txt)에 저장된다. 각 그래프 연산은 GraphMethod 헤더 파일 및 cpp 파일에 일반 함수로 존재하며 그래프 형식이 다를지라도 동일한 데이터 입력 시 동일한 동작이 수행되도록 이를 일반화하여 구현한다. 또한 대형 그래트에서도 모든 연산이 정상적으로 동작하도록 프로그램의 성능을 높이도록 한다.

2. Flowchart

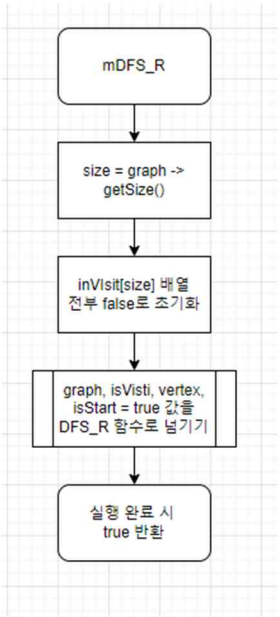
Manager.cpp 파일의 명령어들과 알고리즘의 핵심이 되는 함수들로 하여 flowchart를 구성했다. 뒷 부분의 flowchart는 시간 관계상 손으로 직접 그린 것으로 대체한다. 또한 알고리즘 함수들의 자세한 동작은 Algorithm 파트에서 상세히 후술하도록 한다.



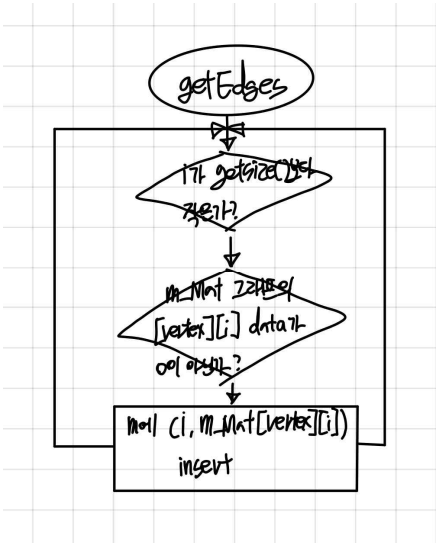
▲ LOAD 명령어 실행 시 수행되는 동작. 파일을 열어 그래프의 형태와 사이즈를 입력받고, 입력받은 형태에 따라 각각 ListGraph 또는 MatrixGraph를 생성



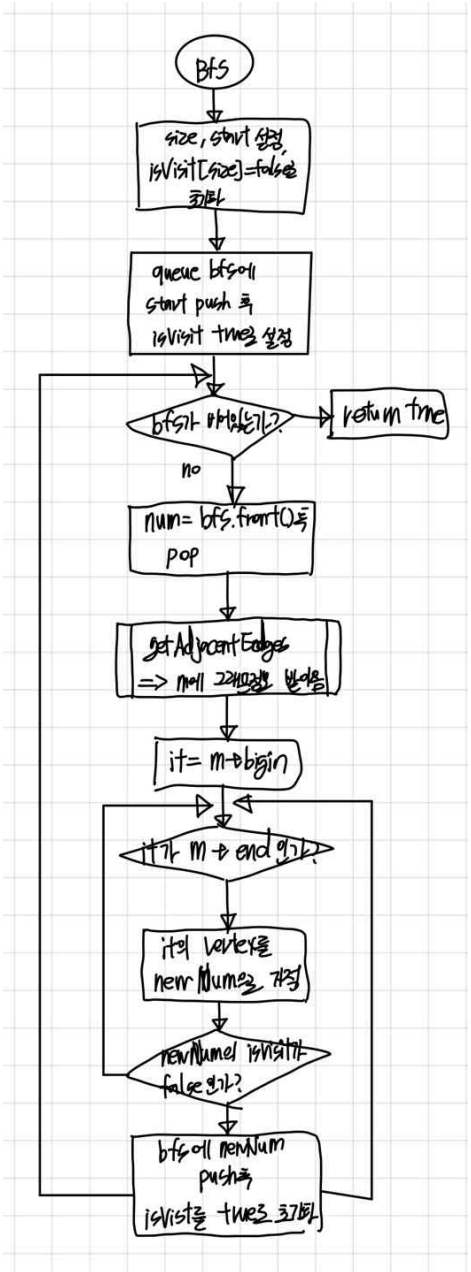
▲ LishGraph의 getAdjacent 함수 동작. getAdjacent 함수는 ListGraph나 MatrixGraph 클래스 모두에 존재하지만, 예시로 List만 생성했다. getEdge 함수와는 달리 이 함수는 BFS, DFS 등에 쓰이는 함수로, 방향성이 없을 경우를 생각해서 만든 함수이다. 모든 vertex의 인접 vertex는 쌍방으로 저장되어 있다.



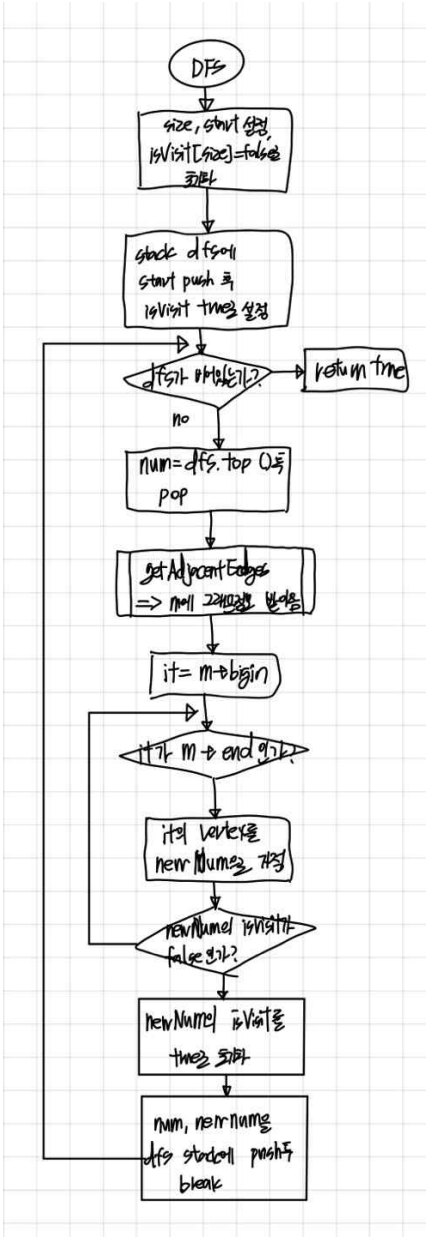
▲ mDFS_R 함수의 동작. DFS와 달리 재귀함수를 통해 기능이 수행되므로, DFS_R 함수를 호출하기 전에 미리 isVisit 배열(방문 여부를 확인하는 배열)이나 size 등을 측정하고 DFS_R 함수를 호출하여 값을 넘겨줘야만 그 값들이 다음 재귀 때에도 유지된다.



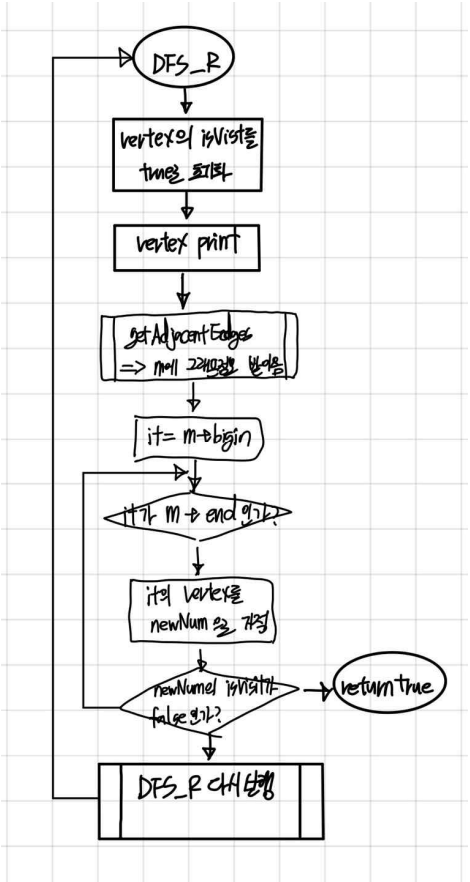
▲ MatrixGraph의 getEdge함수의 동작. getEdge 함수는 ListGraph나 MatrixGraph 클래스 모두에 존재하지만, 예시로 MatrixGraph의 동작만 가져왔다. getAdjacent 함수와는 달리 방향성을 고려하여 만든 함수이다.



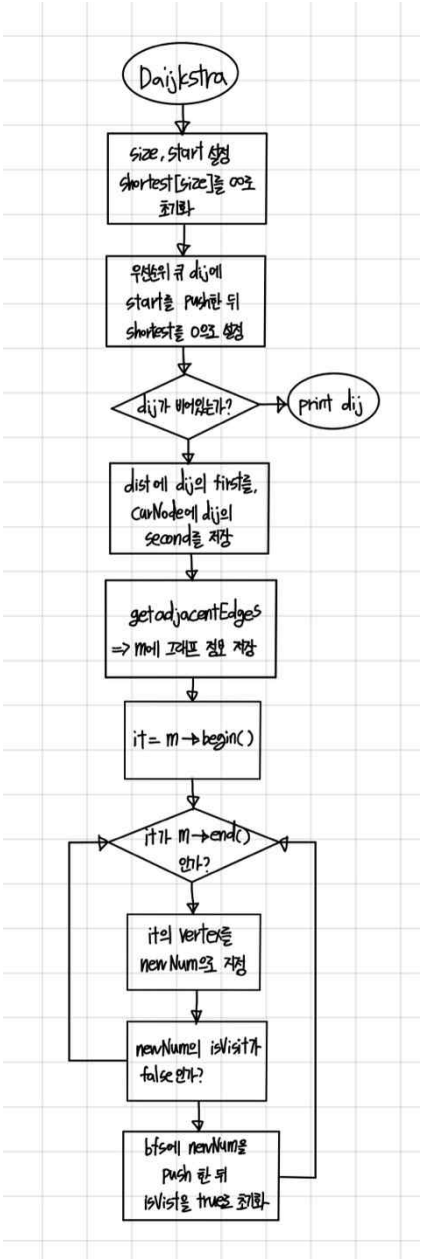
▲ BFS 알고리즘의 동작이다. Queue를 사용했으며, 자세한 내용은 Algorithm 파트에서 후술한다.



▲ DFS 알고리즘의 동작이다. Stack을 사용했으며, 자세한 내용은 Algorithm 파트에서 후술한다.



▲ DFS_R 알고리즘의 동작이다. DFS와는 달리 재귀함수의 동작을 통해 작동한다. 처음 시작 때 inVisit[size] 배열을 false로 초기화해 매개변수로 받고, 같이 받은 vertex를 출력한다(이 과정은 Manager.cpp의 mDFS_R에서 수행한다). 여기서 print는 stack의 pop과 같은 역할을 한다고 생각할 수 있다 그리고 getAdjacentEdges 함수를 실행해 m에 해단 vertex graph의 정보를 받고, iter를 통해 받은 m map을 순회한다. M의 end가 될 때까지 iter의 vertex를 newNum으로 지정 후, isVisti[newNum]이 true인지(이미 방문하여 프린트한 vertex인지) 확인한다. 그리고 만약 아니라면 DFS_R를 재귀적으로 다시 실행하여 이 과정을 반복한다. 만약 방문한 그래프라면 for 순환문을 계속 end가 될 때까지 돌고, 반복문이 끝난 경우 true를 return한다.



▲ DIJKSTRA 알고리즘의 동작이다. 우선순위 큐를 사용했으며, 자세한 내용은 Algorithm 파트에서 후술한다.

3. Algorithm

- BFS

1. ofstream fout의 선언
2. log.txt 파일을 열어 텍스트 파일의 가장 끝부분부터 쓰기를 진행
3. graph의 getSize 함수를 통해 size를 반환받음
4. bool형의 isVisit 배열을 size만큼 할당하고, false로 초기화. 이 배열은 vertex 방문 여부를 판별하는 데 사용됨
5. start를 주어진 vertex로 지정
6. 큐를 선언하여 start data를 push함
7. start의 isVisit 배열을 true로 설정(방문했음을 표기) 후 start를 출력
8. bfs라는 큐가 empty 상태가 될 때까지 반복문 돌기
8-1. bfs 큐의 front를 num으로 지정 후 bfs pop 진행
8-2. num 출력
8-3. m이라는 map을 할당하여 num에 대해 m에 정보를 받아오도록 getAdjacentEdges 함수 실행. 이때 getAdjacentEdges 함수는 방향성을 고려하지 않은 그래프(한 vertex의 인접 vertex에 대하여 강한 연결 존재).
8-4. map에 대한 iter 선언하여 m의 begin부터 end까지 돌게 함.
9. newNum을 it의 first(vertex)로 선언.
10. 만약 newNum을 방문한 적이 없다면, bfs에 newNum push.
10-1. isVisit의 newNum 자리를 true로 설정

- DFS

1. ofstream fout의 선언
2. log.txt 파일을 열어 텍스트 파일의 가장 끝부분부터 쓰기를 진행
3. graph의 getSize 함수를 통해 size를 반환받음
4. bool형의 isVisit 배열을 size만큼 할당하고, false로 초기화. 이 배열은 vertex 방문 여부를 판별하는 데 사용됨
5. start를 주어진 vertex로 지정
6. stack를 선언하여 start data를 push함
7. start의 isVisit 배열을 true로 설정(방문했음을 표기) 후 start를 출력
8. dfs라는 stack이 empty 상태가 될 때까지 반복문 돌기
8-1. dfs stack의 top을 num으로 지정 후 dfs pop 진행
8-2. m이라는 map을 할당하여 num에 대해 m에 정보를 받아오도록 getAdjacentEdges 함수 실행. 이때 getAdjacentEdges 함수는 방향성을 고려하지 않은 그래프(한 vertex의 인접 vertex에 대하여 강한 연결 존재).
8-3. map에 대한 iter 선언하여 m의 begin부터 end까지 돌게 함.
9. newNum을 it의 first(vertex)로 선언.
10. 만약 newNum을 방문한 적이 없다면, newNum을 출력

10-1. isVisit의 newNum 자리를 true로 설정
10-2. dfs에 num과 newNum을 차례대로 push후 break를 통해 반복문 탈출

- Dijkstra
1. ofstream fout의 선언
2. log.txt 파일을 열어 텍스트 파일의 가장 끝부분부터 쓰기를 진행
3. graph의 getSize 함수를 통해 size를 반환받음
4. int형의 shortest 배열을 size만큼 할당하고, 999(Infinite number을 임의로 999로 설정)로 초기화. 이 배열은 vertex 방문까지의 최단 거리를 저장하는 데 사용됨
5. start를 주어진 vertex로 지정
6. 우선순위 큐를 선언하여 start를 push
7. shortest의 start를 0으로 설정(자기 자신으로 가는 최단거리는 0이므로)
8. dij라는 우선순위 큐가 empty 상태가 될 때까지 반복문 돌기
8-1. dist를 dij의 top의 first(vertex)의 음수로 저장
8-2. curNode를 dij top의 second(가중치)로 저장
8-3. pop 진행
9. 만약 dist가 shortest[curNode]보다 더 길 경우에는 이후 실행이 의미가 없으므로 continue를 통해서 다음으로 넘어감
10. m이라는 map을 할당하여 num에 대해 m에 정보를 받아오도록 getAdjacentEdges 함수 실행. 이때 getAdjacentEdges 함수는 방향성을 고려하지 않은 그래프(한 vertex의 인접 vertex에 대하여 강한 연결 존재).
11. map에 대한 iter 선언하여 m의 begin부터 end까지 돌게 함.
12. cost를 dist에 it->second(가중치)를 더한 값으로 초기화
12-1. 만약 새롭게 초기화한 cost가 현재 vertex의 shortest로 지정된 경로보다 짧다면, shortest를 cost로 초기화.
12-2. 아까 음수로 저장되어 있던 cost와 it의 first를 dij에 push
13. 최단거리 계산이 모두 끝난 후, start vertex를 제외한 각 vertex를 돌면서 출력 진행
14. 만약 거리가 999(infinite num, 연결되지 않은 vertex)라면 x를 출력
14-1. 아닌 경우, 최단 경로를 출력

4. RESULT SCREEN

각 명령어별로의 결과 화면을 정리했다. 모든 결과는 콘솔창이 아닌 log.txt에 출력된다.
기본적인 Command의 순서는 다음과 같다.

DS_Project_3_2022_2 > command.txt

```
1  LOAD graph_L.txt
2  PRINT
3  BFS 0
4  DFS 0
5  DFS_R 2
6  DIJKSTRA 0
```

▲ 첫 번째 실행

DS_Project_3_2022_2 > command.txt

```
1  LOAD graph_M.txt
2  PRINT
3  BFS 3
4  DFS 1
5  DFS_R 4
6  DIJKSTRA 2
```

▲ 두 번째 실행

1. LOAD & PRINT

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

========PRINT========
Graph is ListGraph!
[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)
```

▲ ListGraph를 출력했을 때의 모습

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

========PRINT========
Graph is MatrixGraph!
    [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

```

▲ MatrixGraph를 출력했을 때의 모습. ListGraph를 출력했을 때와 마찬가지로 잘 출력됨을 확인해볼 수 있다.

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

▲ 파일을 열리지 않게 했을 때의 모습. 에러가 뜬다.

2. BFS

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

▲ 시작을 0으로 하고 BFS를 진행했을 때의 결과. 직접 계산했을 때와 동일한 결과가 나온다.

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

▲ 시작을 3으로 하고 BFS를 진행했을 때도 마찬가지로 결과가 잘 나옴을 확인할 수 있다.

4. DFS

```
21    ========DFS========
22    startvertex: 0
23    0 -> 1 -> 2 -> 4 -> 3 -> 6 -> 5
24    ====================
```

▲ 시작을 0으로 하고 DFS를 진행했을 때의 결과. 직접 계산했을 때와 동일한 결과가 나온다.

```
21    ========DFS========
22    startvertex: 1
23    1 -> 0 -> 2 -> 4 -> 3 -> 6 -> 5
24    ====================
```

▲ 시작을 1으로 하고 DFS를 진행했을 때도 마찬가지로 결과가 잘 나옴을 확인할 수 있다.

5. DFS_R

```
26    ========DFS_R========
27    startvertex: 2
28
29    ====================
```

```
choikyeongjeong@DESKTOP-AVBHKU1:~/DS_Project_3_2022_2$ ./run
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
```

▲ DFS_R을 실행하면 cout으로 했을 때는 제대로 된 결과가 나오나, 파일에는 제대로 된 출력이 되지 않는다. 다른 것들은 아무것도 바꾸지 않고 cout을 fout을 바꿨을 뿐인데도 이상한 출력이 뜬다. 디버깅을 하면 제대로 된 결과가 나오는데, 파일 출력만 이상하게 나와서 부득이하게 cout으로 진행했다.

```
26    ========DFS_R========
27    startvertex: 4
28
29    ====================
```

```
choikyeongjeong@DESKTOP-AVBHKU1:~/DS_Project_3_2022_2$ ./run
4 -> 2 -> 0 -> 1 -> 3 -> 6 -> 5
```

▲ DFS_R을 시작 vertex를 4로 하고 실행한 결과. cout에서는 결과가 잘 나옴을 확인해볼 수 있다.

6. Dijkstra

```
========DIJKSTRA========
startvertex: 0
[1] 6
[2] 2
[3] 9
[4] 5
[5] 10
[6] 11
====================
```

▲ 시작 vertex를 0으로 했을 때의 다익스트라 알고리즘 실행 결과. 원래는 지나가는 vertex도 모두 출력했어야 하나 시간상의 문제로 각 vertex의 최단 경로만 출력했다.

```
31    ========DIJKSTRA========
32    startvertex: 2
33    [0] x
34    [1] 7
35    [3] 7
36    [4] 3
37    [5] 8
38    [6] 9
39    ====================
```

▲ 시작 vertex를 2로 했을 때의 다익스트라 알고리즘 실행 결과. 0의 경우에는 나가는 방향만 존재하고 들어오는 방향은 존재하지 않는 vertex이므로 최단 거리가 무한(999), x가 나와야 맞다.

5. Consideration

우선 끝까지 해결이 안 된 문제로는 DFS_R의 fout 문제가 있다. 사실 fout은 초반부터 지금까지 계속 문제가 되어 왔다. 저번에는 스켈레톤 코드에서 fout을 넘겨 주어 큰 문제가 되지 않았지만, 스스로 해 보려 하니 그 부분이 잘 되지 않았다. 우선 저번 2차 과제와 같이 &, *을 이용해 fout 자체를 클라스 사이사이로 넘겨 보려고 했지만 알 수 없는 오류로 인해 잘 되지 않았다. 따라서 차선책으로 fout이 필요한 함수마다 ofstream을 선언해 log.txt를 계속 여는 식으로 코드를 짰다. 그런데 그런 경우 함수가 종료되자마자 log.txt가 초기화되는 문제가 존재했다. 따라서 이런 문제를 없애기 위해 스켈레톤 코드에 있던 ios::app을 사용했다. 그랬더니 이번에 생긴 문제는, 출력이 텍스트 파일의 제일 끝부분에 추가되는 점은 좋은데 프로그램이 전체 종료되어도, 그리고 다시 실행되어도 log.txt 파일의 내용이 사라지지 않는다는 점이었다. 그러나 이 부분은 그냥 보기에만 좀 불편하지 기능상 큰 문제는 없어서 넘어갈 수 있었다. 가장 큰 문제는 DFS_R에서 fout을 이용한 출력이 이상하다는 점이었다. 다른 함수들은 다 멀쩡하고, DFS_R을 cout으로 출력했을 때는 정상적으로 잘 출력이 되는 것을 보면 재귀를 여러 번 하면서 fout을 여러 번 선언할 때 문제가 생긴 것이 아닌가 싶은데, 이러한 문제를 결국 해결하지 못했다. 지금 고찰을 쓰면서 매개변수로 fout을 넘겨보는 것이 어떤가라는 생각이 들었는데, 시간이 없어서 해결하지 못할 것 같다. 다음에 시간이 생기면 ios::app에 대해서 조금 더 찾아봐야 할 것 같다.

두 번째로 골머리를 앓았던 문제는 LOAD에서 List 모드로 숫자를 받을 때 숫자가 1개인지 2개인지 구분하는 것에 대한 문제였다. 원래는 줄 단위로 문자열로 받아서 스페이스바 단위로 문자열을 나눈 뒤에 숫자를 확인하려고 했었다. 스페이스바가 나왔을 때 그 전의 문자열을 NULL로 바꾸고, 그 다음 포인터가 가리키는 게 NULL이라면 숫자가 1개라는 뜻이고, 그렇게 되면 받은 숫자는 start vertex가 된다는 의미이기 때문이다. 그런데 줄 단위로 문자열이 받아지지 않았다. 여러 시도 끝에 알아낸 점은, 바로 getline과 fin의 차이 문제였다. 처음에 모드와 그래프 사이즈를 받을 때는 fin을 이용해서 그냥 받았는데, fin은 개행 문자를 포함하지 않고 문자를 받는다는 것이었다. 따라서 그다음에 getline로 다음 줄을 받으려고 하니 ""만 뜨고 문자열이 받아지지 않는 것이었다. 따라서 getline을 fin 다음에 실행하여 아무것도 없는 개행 문자를 받고(getline은 받을 수 있다고 한다), 그 다음 숫자가 존재하는 줄을 반복문을 통해 받도록 설정했다.

그리고 또 고민했던 것은 BFS, DFS는 방향이 없는 그래프 환경이라는 것이었다. 이를 어떻게 해야 하나 고민하던 중, getAdjacentEdges 함수가 떠올랐다. 굳이 그래프를 2개 만들 필요 없이 getAdjacentEdges를 실행할 때 순환을 2번 돌아 해당 vertex에서 나가는 방향의 vertex를 push하고, 해당 vertex로 들어오는 방향의 vertex도 map에 같이 push를 해 주면 쌍방으로 연결된 형식이 되므로(실제 그래프 상에서는 변화가 없다. map에서만 그렇게 확인된다) 비방향 그래프처럼 사용할 수 있는 것이다. 대신 getEdges 함수를 새로 하나 만들어 방향 그래프에 대해서도 사용할 수 있는 함수로 만들었다(해당 vertex에서 나가는 방향의 vertex만 push).

이번 과제는 아쉬움이 많이 남는다. 1차 2차 과제에 비해 난이도가 확실히 쉬워진 게 느껴졌으나 안타깝게도 시간 문제 때문에 끝까지 구현하지 못한 게 아깝다. 앞으로 C++을 쓸 일이 얼마나 있을지는 모르겠지만 이번 학기 3개의 프로젝트를 통해 많은 것을 배워가는 것 같다. 1학기때 나름 C++을 할 줄 안다고 자만했는데, stl에는 내가 모르는 정말 수많은 기능이 있었고, 간단한 코드 구현이라고 생각했던 것들도 구현하는데 꽤 애를 많이 먹을 적도 있었다.

비주얼 스튜디오 코드도 처음 사용해 봤고, 디버깅을 하는 법도 이제야 제대로 배운 것 같다.
앞으로도 꾸준히 공부를 열심히 해야 할 것 같다.