Data Structure Project

Project #3

담당교수 : 신영주 교수님

제출일 : 2019. 12. 06.

학과:컴퓨터정보공학부

학번 :2018202074

이름 : 김상우

1. Introduction

- 위 프로젝트는 텍스트 파일에서 섬의 거리 데이터를 읽은 후 명령어를 받고 그에 따라 알맞은 그래프 연산을 수행하는 코드를 완성하는 것이 목표인 프로젝트이다.

  최종적으로 DIJKSTRA알고리즘, BELLMANFORD 알고리즘, FLOYD 알고리즘을 구현하여야 한다. 제작해야 하는 알고리즘은 아래의 명령어에 대해 알맞은 작동을 할 수 있어야 하며 추가적으로 에러가 발생한다면 해당 에러에 대해 알맞은 값들을 출력해야 한다.

  LOAD(텍스트 파일로부터 값을 읽어오고 이를 그래프화 한다.)

  PRINT(LOAD한 값을 출력한다.)

  DIJKSTRA(STL set으로 Dijkstra를 실행한다.)

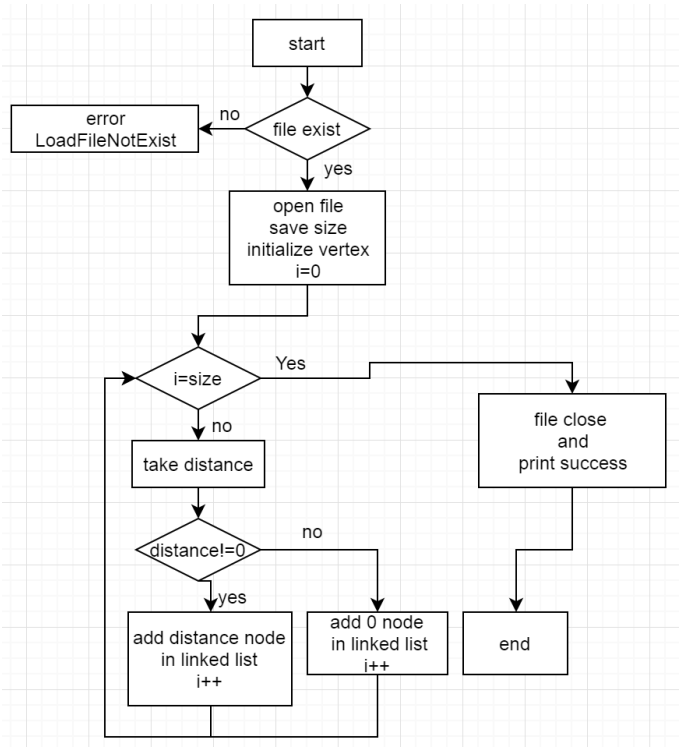
  DIJKSTRAMIN(Min-Heap으로 Dijstra를 실행한다.)

  BELLMANFORD(BELLMANFORD 알고리즘을 실행한다.)

  FLOYD(FLOYD 알고리즘을 실행한다.)

2. Flow Chart

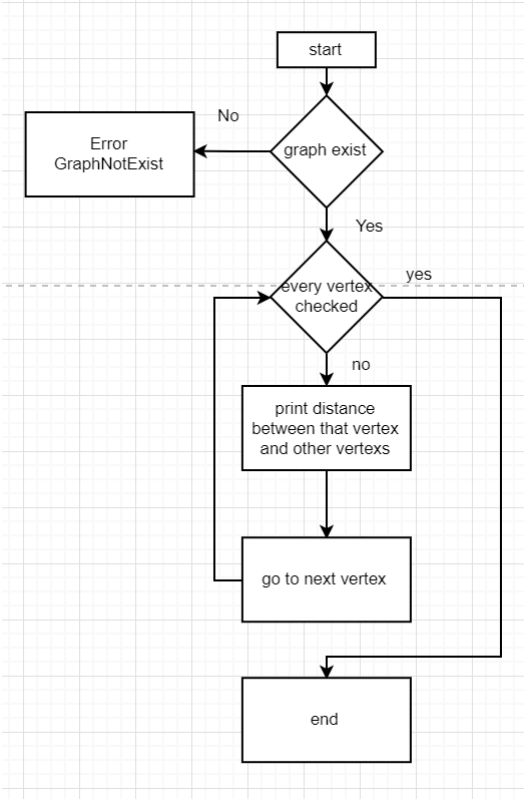
- LOAD



텍스트 파일을 열고 크기를 확인한다. 그리고 입력된 거리를 순서대로 입력받는다. 이때 크기만큼 진행될 때마다 줄을 바꿔준다. 줄을 바꿔준 횟수가 크기만큼 되었다면 이를 종료한다.

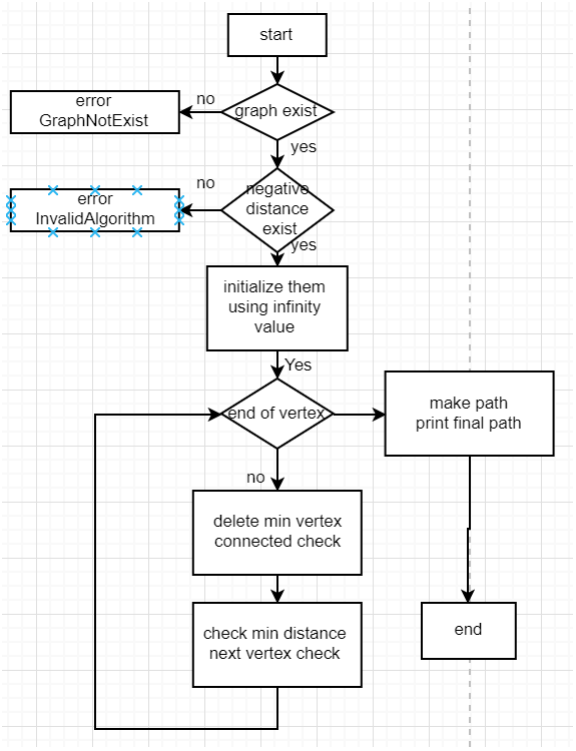
이때 파일이 없다면 알맞은 error를 출력한다.

- PRINT



모든 vertex에 방문하고 그 vertex와 다른 vertex간의 길이들을 모두 확인하고 출력한다.

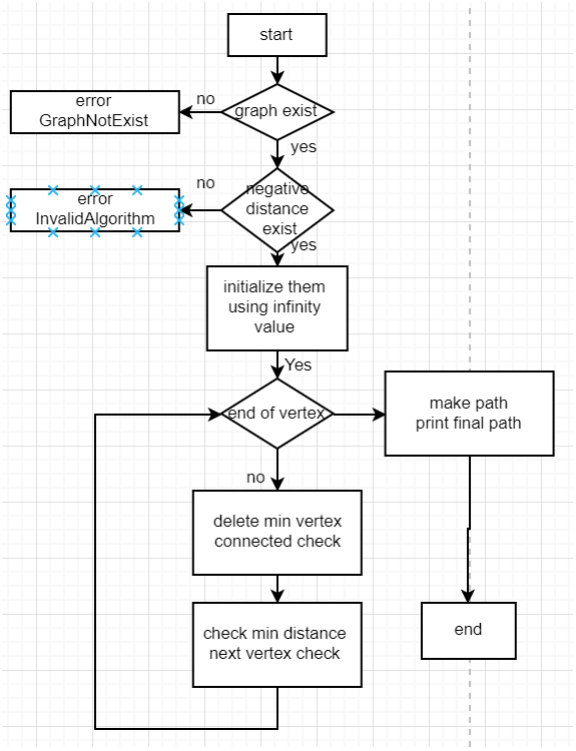
- DIJKSTRA



Graph가 존재하지 않거나 음수인 거리가 입력되었을 경우 알맞은 에러를 출력한다.

만약 에러가 감지되지 않았다면 첫번째 Vertex로 향한다. 첫 번째 Vertex와 연결된 노드들과의 거리들을 확인하고 저장한 후 Vertex를 제거한다. 이 중 가장 적은 거리를 가진 distance를 체크하고 Vertex를 확인한다. 이 때 거리들은 저장된다. 이후 다음 Vertex가 존재한다면 그 연결된 Vertex로 이동한다. 이후 그 Vertex와 연결된 노드들과의 거리들을 확인하고 이전에 연결된 값들과 비교해 더 적은 값을 저장한 후 Vertex를 제거한다. 이 중 가장 적은 거리를 가진 distance를 체크하고 Vertex를 확인한다. 이후 Vertex가 존재하지 않는다면 이를 끝낸다.

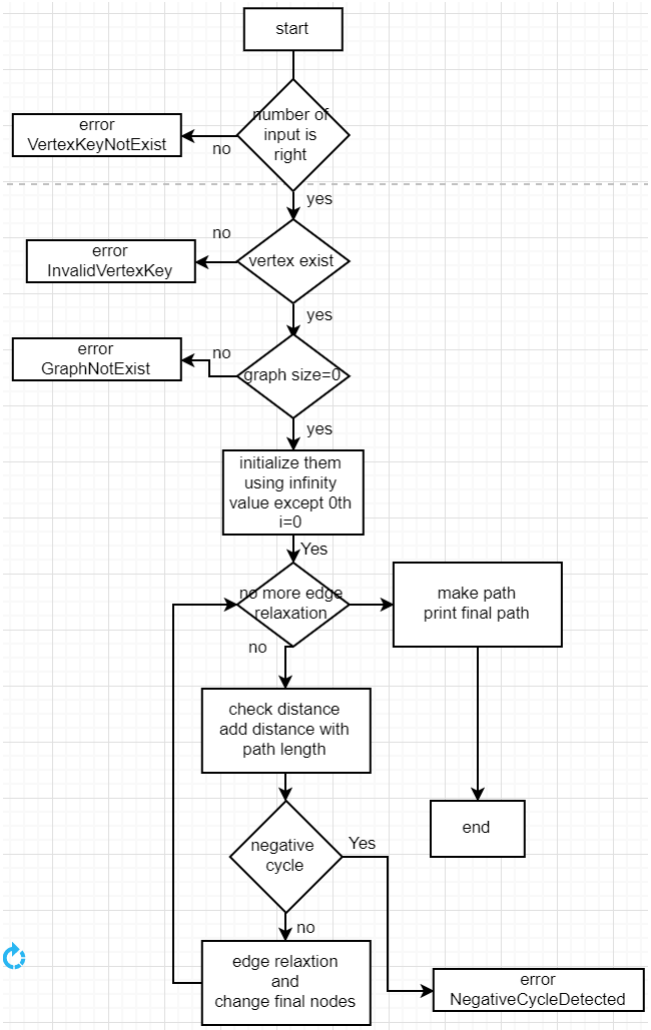
- DIJKSTRAMIN



Graph가 존재하지 않거나 음수인 거리가 입력되었을 경우 알맞은 에러를 출력한다.

만약 에러가 감지되지 않았다면 첫번째 Vertex로 향한다. 첫 번째 Vertex와 연결된 노드들과의 거리들을 확인하고 저장한 후 Vertex를 제거한다. 이 중 가장 적은 거리를 가진 distance를 체크하고 Vertex를 확인한다. 이 때 거리들은 저장된다. 이후 다음 Vertex가 존재한다면 그 연결된 Vertex로 이동한다. 이후 그 Vertex와 연결된 노드들과의 거리들을 확인하고 이전에 연결된 값들과 비교해 더 적은 값을 저장한 후 Vertex를 제거한다. 이 중 가장 적은 거리를 가진 distance를 체크하고 Vertex를 확인한다. 이후 Vertex가 존재하지 않는다면 이를 끝낸다.

- BELLMANFORD

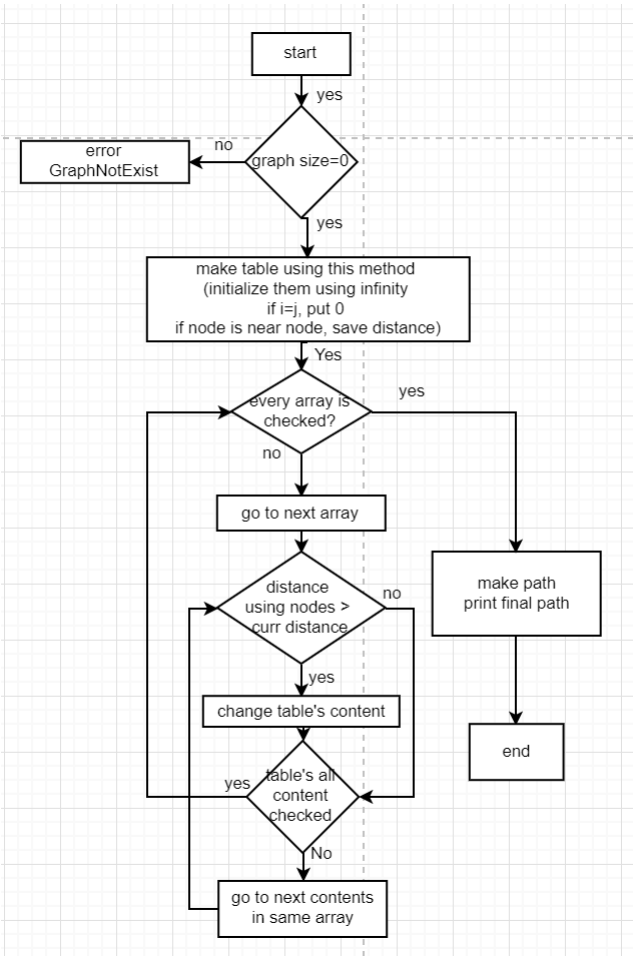


명령어 뒤에 입력된 값의 개수가 다르거나 Vertex가 존재하지 않거나 graph size가 0 이면 그에 대해 알맞은 Error를 출력한다.

만약 Error가 없다면 연결된 노드들과의 거리를 확인한다. 이후 이전까지의 거리를 더하고 이가 음수 사이클을 만드는지 확인한다. 만약 음수 사이클을 만든다면 Error 를 출력한다. 만약 음수 사이클을 형성하지 않는다면 최종 노드들의 값들을 지금까지의 거리로 변경한다. 이를 더 이상 edge들이 이를 감소시킬 수 없을 때까지 진행한다.

이후 이를 기반으로 path를 만들고 종료한다.

- FLOYD



만약 GraphSize가 0일 경우 알맞은 Error를 출력해준다.

이 에러에 해당되지 않는다면 같은 노드를 가리키는 거리는 직접적으로 연결되어 있는 node끼리의 거리는 해당 거리만큼, 이외의 노드 간에는 무한대의 값을 준 표를 만든다. 이후 특정 경유지들에 저장된 거리 값들을 여러 노드를 거쳤을 때의 값들과 비교한다. 이때 여러 노드를 거치는 경우가 더 짧다면 값을 바꾸어준다. 이를 모든 값들이 확인되었을 때까지 하고 이 값에 따라 path를 만들고 종료한다.

3. Algorithm

- 전체 알고리즘

명령어를 텍스트 파일을 통해 받고 이를 판단한 후 이에 맞는 실행을 하는 것이 본적인 알고리즘이다. 이는 strcmp를 통해 구현하였다. 만약 input값이 여러 개라면 strtok을 통해서 여러 값들을 모두 저장하였다. 각 값들을 PRINT, LOAD등은 위의 설명과 같은 방식으로 구현되었다. PRINT의 경우 ofstream을 이용해 각 결과에서 값들

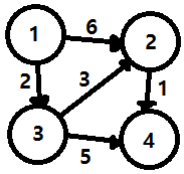
을 받아오는 형식으로 진행하였다. LOAD의 경우 텍스트 파일을 읽고 getline을 통해 한 줄씩 값을 받는 형식으로 진행되었다. 이 외의 명령어들은 이후 상세 서술한다. Ofstream을 통해 생성된 값을 이후 log.txt로서 내보내주며 코드는 마무리된다.

- DIJKSTRA, DIJKSTRAMIN

  Graph가 존재하지 않거나 음수인 거리가 입력되었을 경우를 graph의 size와 음수 사이클 발생으로 구별한다. 만약 에러가 발생했다면 알맞은 에러를 출력한다.

  만약 에러가 감지되지 않았다면 첫번째 Vertex로 향한다. 첫 번째 Vertex와 연결된 노드들과의 거리들을 확인하고 저장한 후 Vertex를 제거한다. 이 중 가장 적은 거리를 가진 distance를 체크하고 Vertex를 확인한다. 이 때 거리들은 저장된다. 이후 다음 Vertex가 존재한다면 그 연결된 Vertex로 이동한다. 이후 그 Vertex와 연결된 노드들과의 거리들을 확인하고 이전에 연결된 값들과 비교해 더 적은 값을 저장한 후 Vertex를 제거한다. 이 중 가장 적은 거리를 가진 distance를 체크하고 Vertex를 확인한다. 이후 Vertex가 존재하지 않는다면 이를 끝낸다.

  DIJKSTRA 알고리즘의 예시는 다음과 같다.



다음과 같은 그래프가 있다고 가정하자.

	1	2	3	4
{1}	0	6	**2**	∞

1과의 연결은 다음과 같다. 본인을 제외한 값 중 가장 가까운 건 3이므로 3을 1과 연결한다.

	1	2	3	4
{1}	0	6	**2**	∞
{1,3}		**5**		5

{1,3}과의 연결은 다음과 같다. 본인을 제외한 값 중 가장 가까운 것 중 낮은 값을 가진 Vertex는 2이므로 2을 {1,3}과 연결한다.

	1	2	3	4
{1}	0	6	**2**	∞

{1,3}		**5**		5
{1,3,2}				**1**

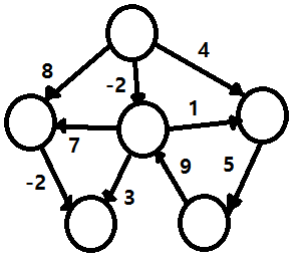
이후 4와의 연결을 통해 마무리 되었다.

- BELLMANFORD

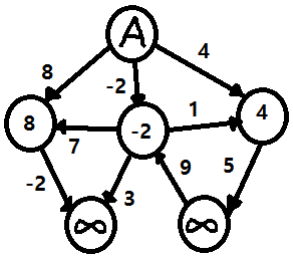
  위 알고리즘의 경우 입력된 값의 수나 Vertex의 존재 유무, graph size등을 판단해 error가 존재하는지 판별한다. 만약 error가 존재하지 않는다면 처음에 각 노드 간 거리가 적힌 Table을 반복문을 통해 구하고 음수 사이클이 발생하는지 확인한다. 음수 사이클 발생의 경우 Vertex값을 넘어간 경우에도 값들이 줄어들 경우가 생기는지 확인하여 발생 여부를 판단한다.

  이후 이를 기반으로 path를 만들고 종료한다.

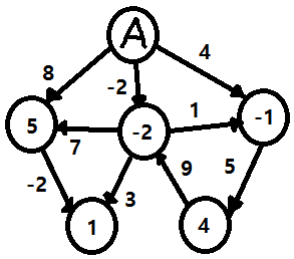
  BELLMANFORD 알고리즘의 예시는 다음과 같다.



다음과 같은 그래프가 있다고 가정하자.



가장 위 Vertex를 기준으로 잡고 근접한 Vertex의 값들을 거리만큼 준다. 이후 더 많은 노드를 거쳤을 때의 거리 중 최솟값을 Vertex에 저장한다.



위의 알고리즘을 따를 시 1개의 노드를 중간에 거칠 시 왼

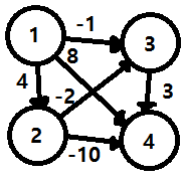
쪽의 8->5, 가장 오른쪽의 4->-1, 아래의 노드들이 왼쪽부터 1, 9가 된다. 이후 2개의 노드를 중간에 거칠 시 우측 하단의 9의 값이 4가 되면서 이후 값들이 바뀌지 않는다. 최종적으로 위와 같은 결과가 된다.

- FLOYD

만약 GraphSize가 0일 경우 알맞은 Error를 출력해준다.

이 에러에 해당되지 않는다면 같은 노드를 가리키는 거리는 직접적으로 연결되어 있는 node끼리의 거리는 해당 거리만큼, 이외의 노드 간에는 무한대의 값을 준 표를 만든다. 이에 저장된 거리 값들을 특정 경유지에 대해 여러 노드를 거쳤을 때의 값들과 비교한다. 이때 여러 노드를 거치는 경우가 더 짧다면 값을 바꾸어준다. 이를 모든 값들이 확인되었을 때까지 하고 이 값에 따라 path를 만들고 종료한다.

FLOYD 알고리즘의 예시는 다음과 같다.



다음과 같은 그래프가 존재한다고 가정하자.

위 그래프를 이용해 다음과 같이 그래프를 초기화 할 수 있다.

	1	2	3	4
1	0	4	-1	8
2	∞	0	-2	-10
3	∞	∞	0	3
4	∞	∞	∞	0

이후 1번을 지나가는 경로는 없으며 2번 경유지를 지나는 경우를 확인하면 1-2-4는 -6임을 알 수 있다. 이는 8보다 작아 이를 표에 적용하면

	1	2	3	4
1	0	4	-1	-6
2	∞	0	-2	-10
3	∞	∞	0	3
4	∞	∞	∞	0

다음과 같은 표가 생기게 된다. 이후 경유지 4를 지날 때 까지 변화는 없으며 이것이 최종값이 됩니다.

4. Result Screen

- 전체 명령어 코드

```
LOAD mapdata.txt
PRINT
DIJKSTRA 0 3
BELLMANFORD 1 4
DIJKSTRA -1 10
BELLMANFORD
ASTAR 1 4
FLOYD
```

예시에서 사용한 다음 코드를 가지고 우선 진행한다.

- LOAD

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

================
Error code: 0
================
```

LOAD의 경우 Success를 통해 성공했음을 알 수 있고 뒤에 나오는 Error code: 0를 통해 정상적으로 작동함을 알 수 있다

- PRINT

```
===== PRINT =====
0 6 13 0 0
0 0 5 6 0
2 0 0 7 4
0 6 0 0 3
0 0 5 2 0
===============

================
Error code: 0
================
```

Load된 값들이 Print로 작동된다. LOAD가 성공했음을 이를 통해 재확인할 수 있고 뒤에 나오는 Error code: 0를 통해 정상적으로 작동함을 알 수 있다

- DIJKSTRA

```
===== DIJKSTRA =====
shortest path: 0 1 3
sorted nodes: 0 1 3
path length: 12
===============

================
Error code: 0
================
```

위의 값들을 바탕으로 DIJKSTRA가 진행된 모습이다. LOAD를 통해 입력된 값들이 잘 진행되고 있으며 Error code:0 또한 정상적으로 작동함을 알 수 있다.

- BELLMANFORD

```
===== BELLMANFORD =====
shortest path: 1 2 4
sorted nodes: 1 2 4
path length: 9
===============

================
Error code: 0
================
```

위의 값들을 바탕으로 BELLMANFORD가 진행된 모습이다. LOAD를 통해 입력된 값들이 잘 진행되고 있음을 알 수 있다.

- FLOYD

```
===== FLOYD =====
0 6 11 12 15
7 0 5 6 9
2 8 0 6 4
10 6 8 0 3
7 8 5 2 0
===============

================
Error code: 0
================
```

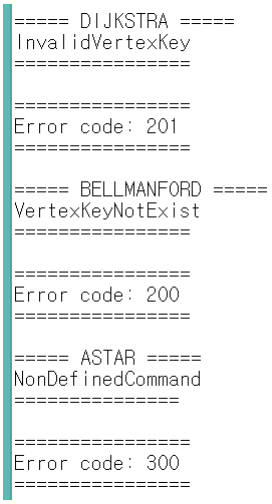
위의 값들을 바탕으로 FLOYD가 진행된 모습이다. LOAD를 통해 입력된 값들이 잘 진행되고 있음을 알 수 있다.

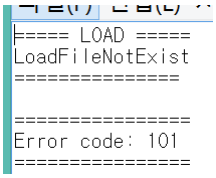
- DIJKSTRAMIN

```
===== DIJKSTRAMIN =====
shortest path: 0 1 3
sorted nodes: 0 1 3
path length: 12
===============

================
Error code: 0
================

===== BELLMANFORD =====
shortest path: 1 2 4
sorted nodes: 1 2 4
path length: 9
===============

================
Error code: 0
================
```

DIJKSTRA의 값을 DIJKSTRAMIN으로 바꾸어 진행한 모습이다. LOAD를 통해 입력된 값들이 잘 진행되고 있으며 Error code:0 또한 정상적으로 작동함을 알 수 있다. 이후의 값들도 일정한 것을 보았을 때 정상적으로 작동함을 알 수 있다.

- ERROR





위에서 언급된 ERROR들의 결과이다. 각각 음수가 입력되었을 때, 입력값의 개수가 맞지 않았을 때, 명령어에 해당되지 않는 값이 들어왔을 때 알맞은 ERROR 코드가 출력된다. 그 외의 오른쪽 사진과 같이 파일의 유무 등을 통한 에러도 잡는 것을 확인하였다. 이 외의 에러들도 적절히 작동되는 것을 확인할 수 있었다.

5. Consideration

이번 프로젝트를 통해 가장 크게 깨달은 것은 같은 알고리즘이라도 다양한 방식으로 코드를 짤 수 있다는 것이었습니다. 대표적으로 다익스트라 알고리즘에서 set을 사용한 것과 min heap을 구현하였을 때, 같은 방식이더라도 어떠한 형태로 작업하느냐에 따라 큰 차이가 발생함을 알 수 있었습니다. 이를 유의해서 가장 효율적인 코드를 작성하는 것이 코더에게 있어서 매우 중요할 것입니다.

또한 Error를 작업하는 것도 중요함을 알았습니다. 실제로 저의 코드에서는 경고가 많은데 이 같은 경우들도 Error로 표현하는 것은 프로그램을 발전시키는 데 큰 도움이 되지 않을까 생각하게 되었습니다.

다만 Stack등의 코드 파일을 어떻게 사용하는 것인지 아직까지 의문이 남습니다. 작업자의 의도를 알았더라면 Stack을 이용해 좀더 효율적인 방법의 코딩도 가능했을 것이라고 저는 생각합니다. 이를 통해 기획자의 의도를 파악하는 것이 얼마나 중요한지 또한 알 수 있었습니다.