데이터 구조 실습 보고서

-3차-

학 번: 2018202014

이 름: 박지원

실습 분반: 목 34

Introduction

이 프로젝트는 도로에 대한 정보를 가지고 있는 텍스트 파일을 읽어와 그래프를 통한 최단 경로 탐색 프로그램을 구현하고 Report 제목에 대한 정보를 가지고 있는 텍스트 파일을 읽어와 라빈 카프 알고리즘을 이용한 문자열 비교 프로그램을 구현한다.

도로 정보가 담긴 텍스트 파일에는 장소의 이름과 N x N 행렬로 표현된 거리의 weight들이 담겨 있다. 이 중 첫번째 줄에는 자신의 회사가, 마지막 줄에는 도착해야하는 회사의 정보가 담겨있다. 파일에서 장소의 이름과 거리의 weight들을 가져와 그래프에 2차원 linked list 형태로 저장한다.

이 그래프를 이용하여 최소 비용 경로를 BFS, DIJKSTRA, BELLMANFORD, FLOYD 알고리즘을 통해 탐색한다.

BFS는 breadth first search의 약자로 인접한 경로부터 탐색하는 알고리즘이다. 이 프로젝트에서는 큐를 이용하여 구현하였고 큐에 넣는 순서는 회사들의 Index 순서대로 들어가야한다.

음수 사이클이 발생하는 경우, 에러를 출력하고, Weight가 음수인 간선은 제거하고 최단 경로를 구한다.

DIJKSTRA는 최단 경로 탐색 알고리즘 중 하나로 BFS와 마찬가지로 음의 간선을 포함 할 수 없기에 Weight가 음수인 간선은 제거하고 최단 경로를 구한다. DIJKSTRA는 최단 거리는 여러 개의 최단 거리로 이루어져 있다고 가정하고, 여러 노드를 거치며 최단 거리를 갱신한다.

BELLMANFORD는 최소 비용을 구하는 그래프 알고리즘으로 DIJKSTRA와 다르게 연결된 정점만을 이용하여 Greedy Search를 하는 것이 아닌 가능한 모든 경우의 수를 탐색한다. 또한, Weight가 음수인 간선이 있어도 정상 작동을 한다.
하지만 음수 Weight에 사이클이 발생하면 에러 코드를 출력한다.

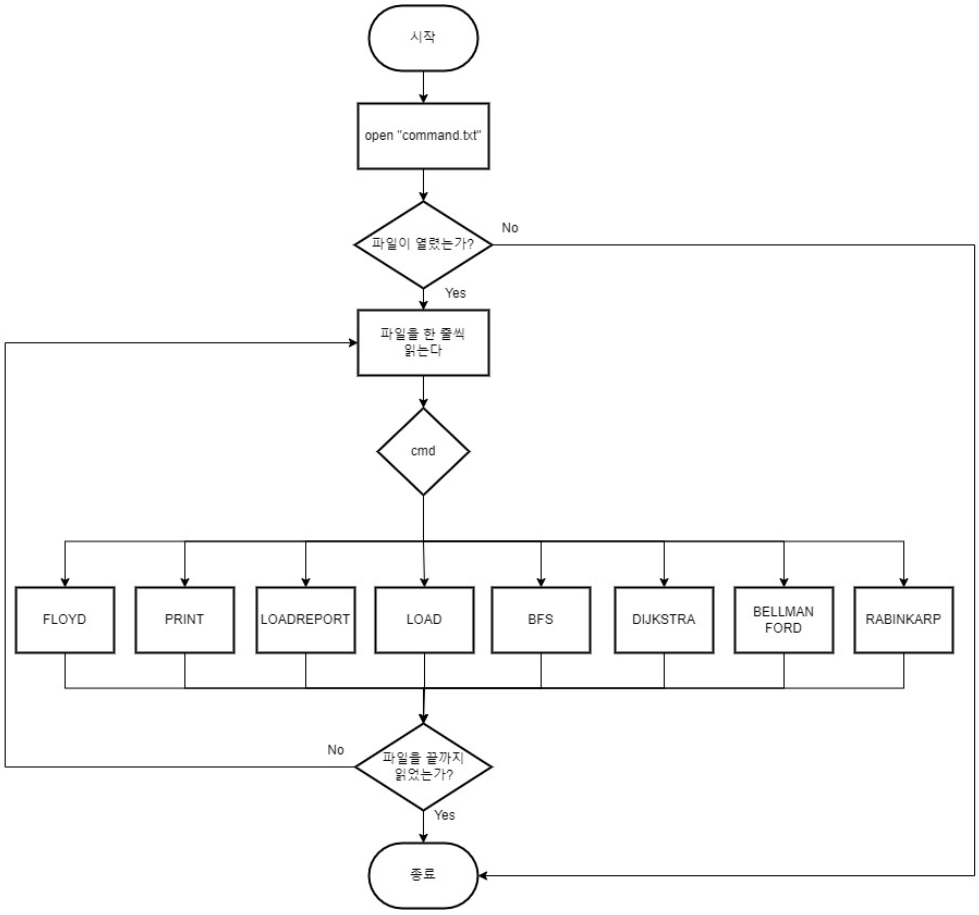
FLOYD는 다른 알고리즘과 달리 시작점에서의 최단 거리를 구하는 것이 아닌 모든 점에서 다른 점으로 가는 최단 경로를 구하는 알고리즘이다. 또한, 2차원 행렬을 통하여 결과를 저장한다.
BELLMANFORD와 같이 음수인 간선이 존재해도 정상 작동을 하지만, 음수 Weight에 사이클이 발생하면 에러 코드를 출력한다.

라빈 카프 알고리즘은 해시 기법을 통한 문자열 알고리즘으로 각 아스키 코드 값에 2의 제곱수를 차례대로 곱하여 모두 더한 값을 비교하여 문자열을 비교한다.

Flowchart

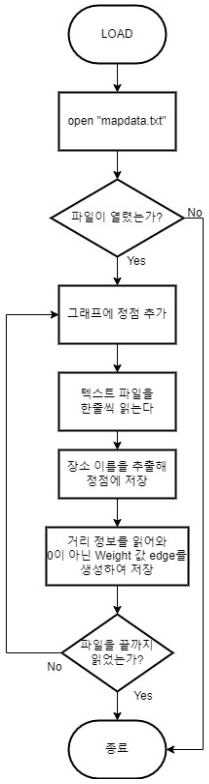
<run>

프로그램을 전체적으로 관리하는 Manager 클래스의 run은 "command.txt"에서 명령어를 순차적으로 읽어와 그에 해당하는 동작을 수행한다. Flow chart는 아래와 같다.



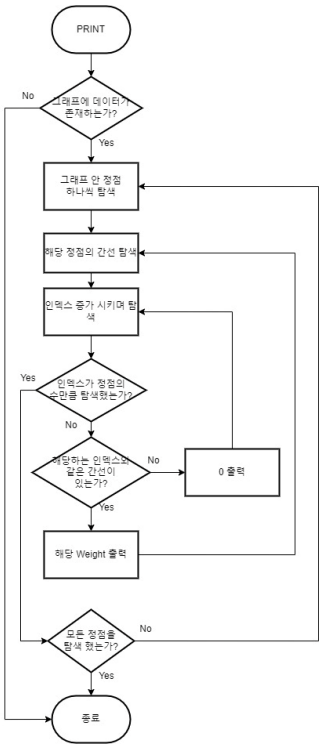
<LOAD>

LOAD 명령어는 도로 정보가 담긴 텍스트 파일을 읽어와 그래프에 저장하는 명령어이다.
파일이 열리지 않으면 함수를 바로 종료하고 에러 코드를 출력한다.
Flow chart는 아래와 같다.



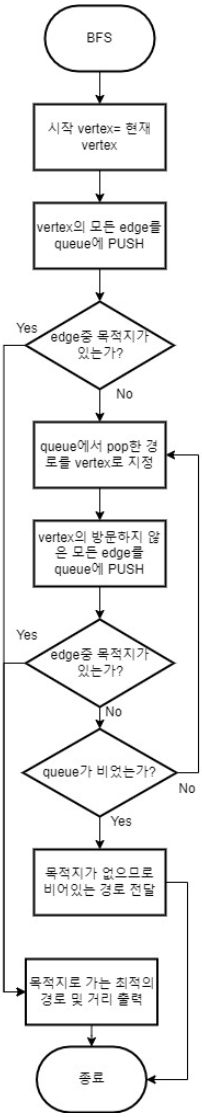
<PRINT>

PRINT 명령어는 그래프 안의 도로 정보를 2차원 행렬로 출력해주는 명령어 이다. 정점 간에 연결되어 있지 않아 간선이 없는 경우에는 0을 출력하고 간선이 있다면 해당하는 간선의 Weight를 출력한다. Flow chart는 아래와 같다.



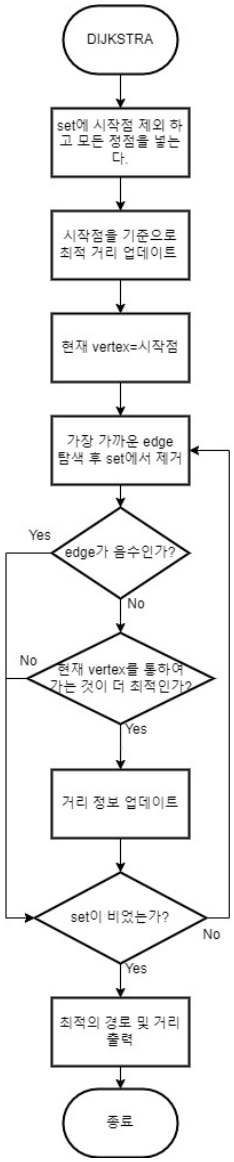
<BFS>

BFS명령어는 BFS 알고리즘을 통해 최적의 경로를 찾는다. 큐를 이용하여 인접한 정점부터 탐색을 하고 목적지가 없다면 비어있는 경로를 전달하고 에러 코드를 출력하게 된다.
Flow chart는 아래와 같다.



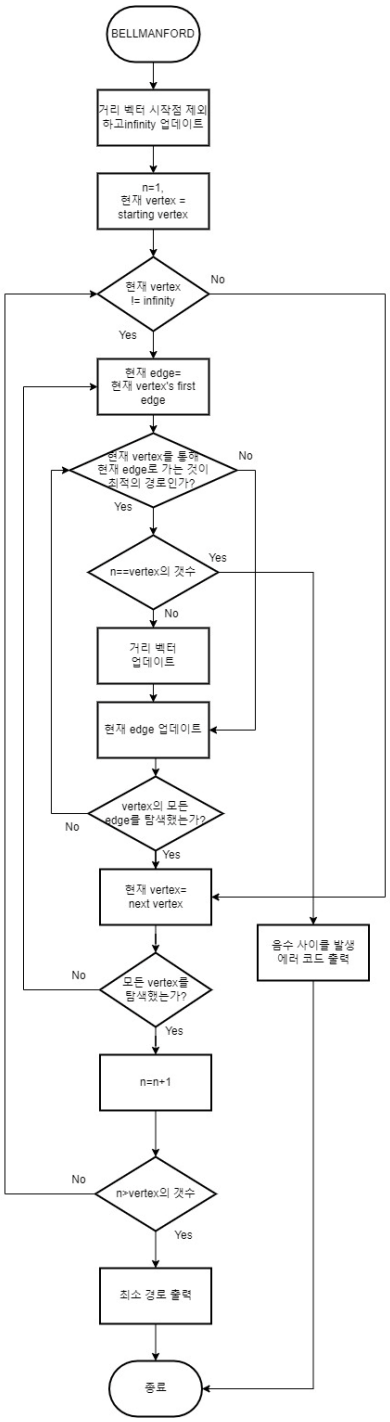
<DIJKSTRA>

DIJKSTRA 명령어는 다익스트라 알고리즘을 이용하여 최소 경로를 찾는다. 음의 간선이 있으면 제거하고 최단 경로를 구한다. Flow chart는 아래와 같다.



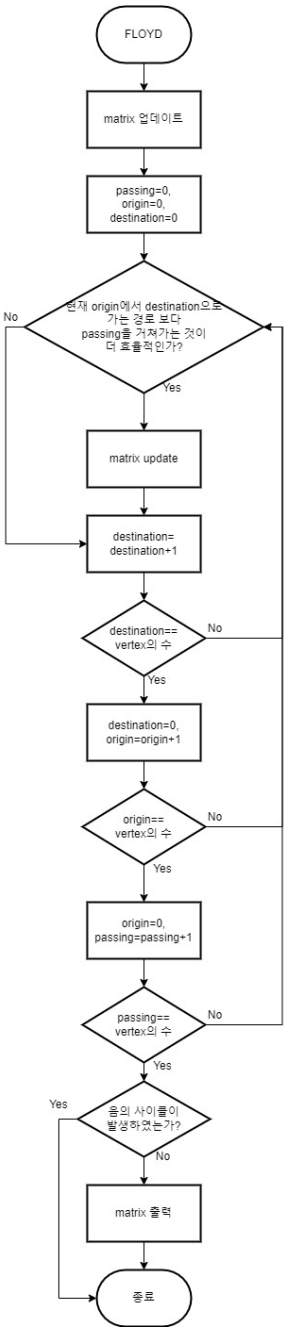
<BELLMANFORD>

BELLMANFORD 명령어는 bellmanford 알고리즘을 사용하여 최단 거리를 찾는 것으로, 위에서 나온 다익스트라나 BFS와 다르게 음의 간선이 있어도 정상적으로 작동을 한다.
Flow chart는 아래와 같다.



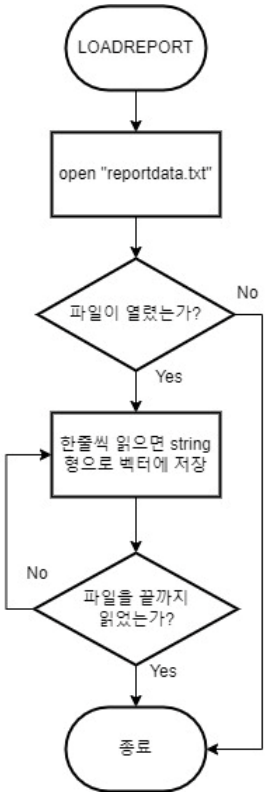
<FLOYD>

FLOYD 명령어는 FLOYD 알고리즘을 사용한다. FLOYD는 기존의 다른 명령어와 다르게 한 정점에서 다른 정점으로의 경로만 출력하는 것이 아닌 모든 경로에 대한 최소의 경로를 matrix로 출력한다. Flow chart는 아래와 같다.



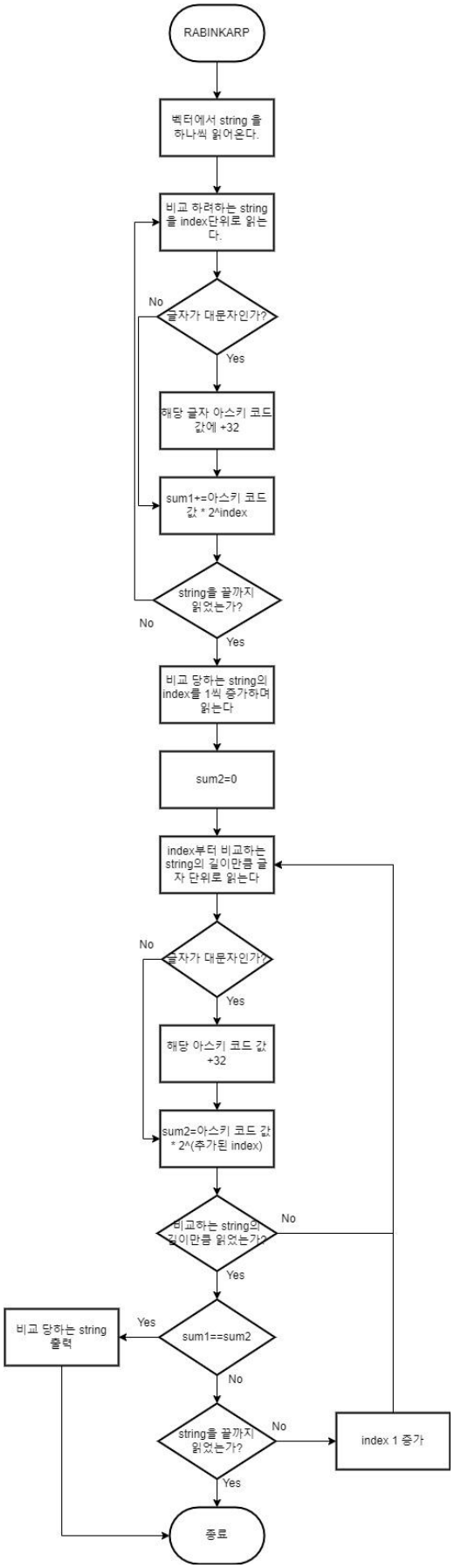
<LOADREPORT>

LOADREPORT는 report의 제목이 담겨있는 텍스트 파일을 열어 한 줄씩 읽어 벡터에 저장하는 명령어이다. Flow chart는 아래와 같다.



RABINKARP>

RABINKARP는 두 문자열을 해시 값을 통해서 비교하여 문자열이 동일한지 확인하는 알고리즘으로 이 프로젝트에는 대소문자를 구별하지 않고 해당 String에 찾고자 하는 Key string이 있다면 String 전체를 출력한다 Flow chart는 아래와 같다.


RABINKARP
벡터에서 string 을 하나씩 읽어온다.
비교 하려하는 string 를 index단위로 읽는다.
No
글자가 대문자인가?
Yes
해당 글자 아스키 코드 값에 +32
sum1+=아스키 코드 값 * 2^index
string을 끝까지 읽었는가?
No
Yes
비교 당하는 string의 index를 1씩 증가하며 읽는다
sum2=0
index부터 비교하는 string의 길이만큼 글자 단위로 읽는다
No
글자가 대문자인가?
Yes
해당 아스키 코드 값 +32
sum2=아스키 코드 값 * 2^(추가된 index)
비교하는 string의 길이만큼 읽었는가?
No
Yes
비교 당하는 string 출력
Yes
sum1==sum2
No
string을 끝까지 읽었는가?
No
index 1 증가
Yes
종료

Algorithm

이 프로젝트에서 사용한 알고리즘은 크게 최단 거리 탐색 알고리즘과 문자열 비교 알고리즘으로 분류 할 수 있다.

이 중 최단 거리 탐색 알고리즘에는 BFS, DIJKSTRA, BELLMANFORD, FLOYD가 있고
문자열을 비교 할 때는 RABINKARP 알고리즘을 사용한다.

<BFS>

BFS(Breadth First Search)는 너비 우선 탐색 기법으로 시작 정점을 기점으로 가까운 정점을 우선 방문하고 멀리 있는 정점을 나중에 방문하는 방법이다. 주로, weight가 없는 그래프에서 최단 경로를 찾을 때 사용된다.

BFS의 장단점은 아래와 같다.

장점

- 목표 정점이 시작 정점에 가까울 경우 빠르게 찾을 수 있다.
- 큐를 이용하여 비교적 간단하게 구현할 수 있다.

단점

- 검색한 정점의 최단 경로는 Weight가 존재하지 않을 때만 보장 받을 수 있다.
  Weight가 존재하면 최단 경로가 아닌 단순히 거쳐간 정점이 적은 경로를 찾게 된다.

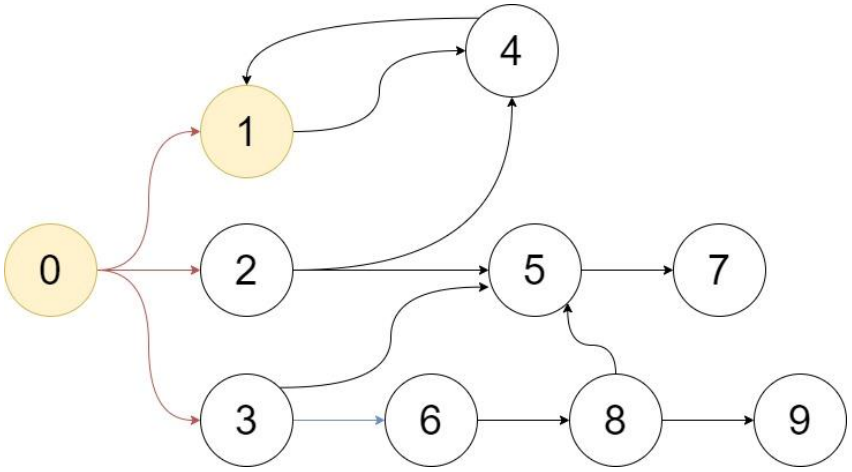
BFS는 너비 우선 탐색 기법이기 때문에 시작점에서 가까운 점부터 탐색을 시작한다. 시작점을 Level 0으로 가정하고 인접한 정점부터 Level 1, Level 2라 하면 BFS는 Level을 증가시키며 정점을 탐색한다.

BFS 알고리즘은 queue를 사용하여 구현하고 동작 원리는 아래와 같다.

1. 시작점을 방문하고 인접한 모든 정점 들을 차례대로 방문
2. 인접한 정점을 기준으로 인접하지만 방문하지 않는 정점 queue에 PUSH
3. 인접한 정점을 방문 한 후 방문한 인접한 정점에서 가장 인접한 정점을 queue를 이용해 방문
4. 목표 정점에 방문 할 때까지 2,3번을 반복한다.

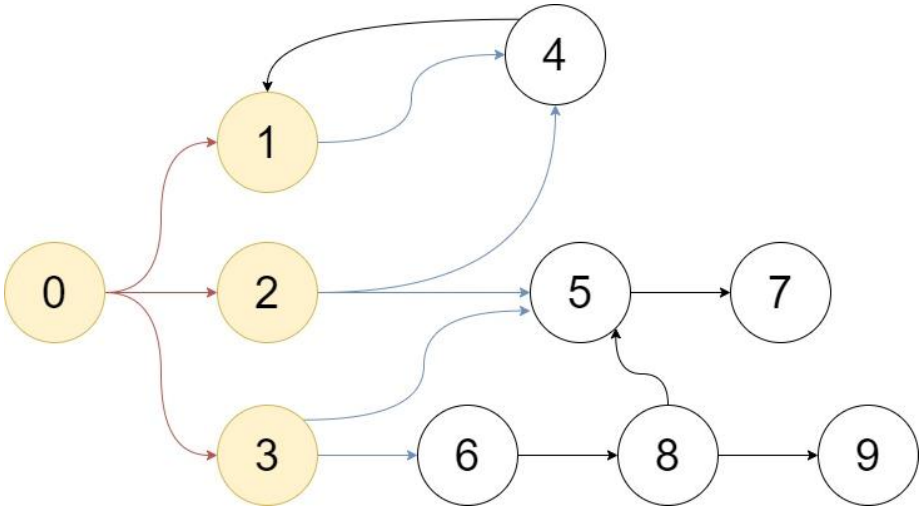
1) 시작점을 방문하고 인접한 모든 정점 들을 차례대로 방문

아래 그림과 같은 그래프가 있고 0번 정점에서 5번 정점을 찾는다고 가정하자 노란 색으로 칠해진 정점은 방문한 정점이고 색깔이 다른 경로는 해당 경로에 있는 정점을 큐에 PUSH를 한다.



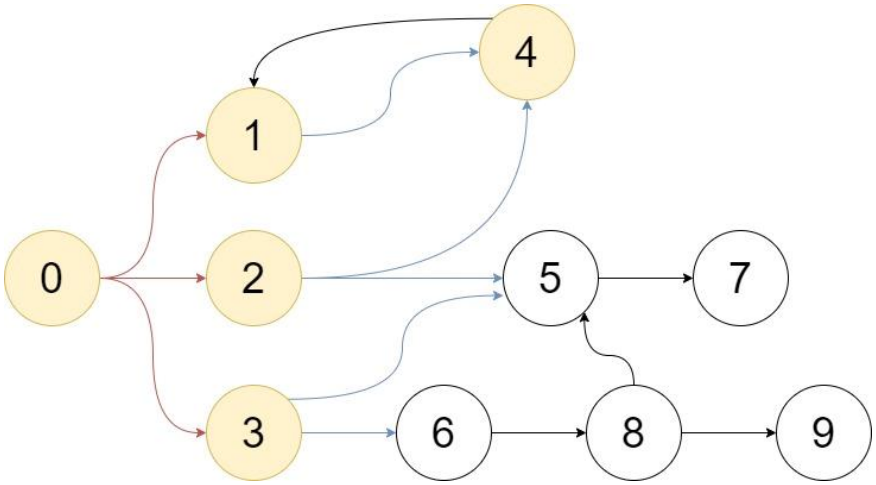
2) 인접한 정점을 기준으로 인접하지만 방문하지 않는 정점 queue에 PUSH

Level 1에 해당하는 정점을 방문하며 queue에 PUSH 해놓은 정점을 POP하며 방문하고 다시 다음 Level에 해당하는 정점을 queue에 PUSH한다.



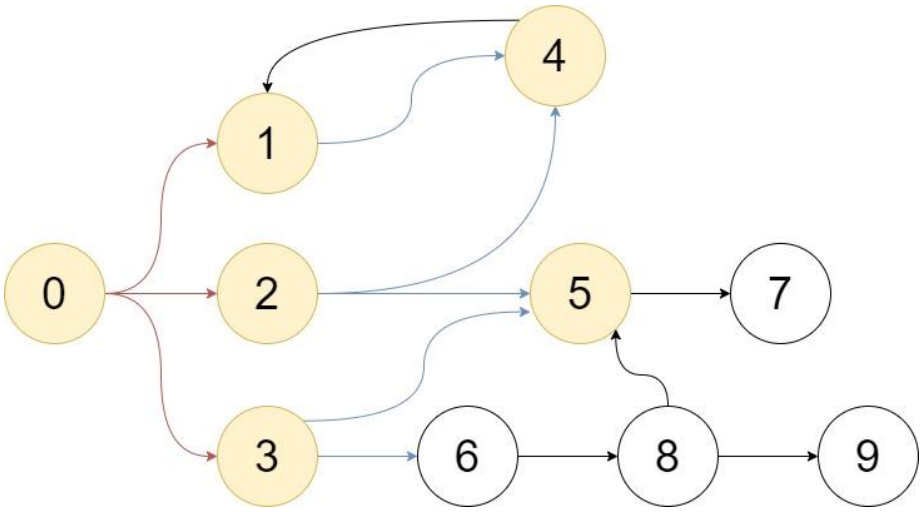
위 그림에서는 순서대로 0,1,2,3 정점을 방문하고 queue에 4,5,6을 PUSH 해놓은 상태이다.

3) 인접한 정점을 방문 한 후 방문한 인접한 정점에서 가장 인접한 정점을 queue를 이용해 방문한 Level에 해당하는 정점을 모두 방문하면 queue에서 POP하며 다음 Level을 정점을 방문한다. 아래 그림에서 4번 정점을 방문했으나, 1번 정점을 queue에 PUSH를 하지않은 이유는 1번 정점이 이미 방문한 정점이기 때문이다.



4) 목표 정점에 방문 할 때까지 2,3번을 반복한다.

이를 반복하여 아래와 같이 목표 정점을 찾으면 종료한다. 아래 그림에서 최단 경로는 2가지 이지만 이 프로젝트에서는 인덱스가 적은 정점부터 방문하므로 최단 경로는 0->2->5가 된다.



<DIJKSTRA>

DIJKSTRA는 대표적인 최단 경로 탐색 알고리즘으로 최단 거리는 여러 개의 최단 거리로 이루어져 있다고 가정하고 여러 경로를 거칠 때 그 경로가 최적이면 거리 정보를 업데이트 한다. 단, 음의 간선이 있을 경우는 다익스트라 알고리즘을 사용할 수 없지만 이 프로젝트에는 음의 간선이 존재할 경우 음의 간선을 제거하고 최단 경로를 구한다.

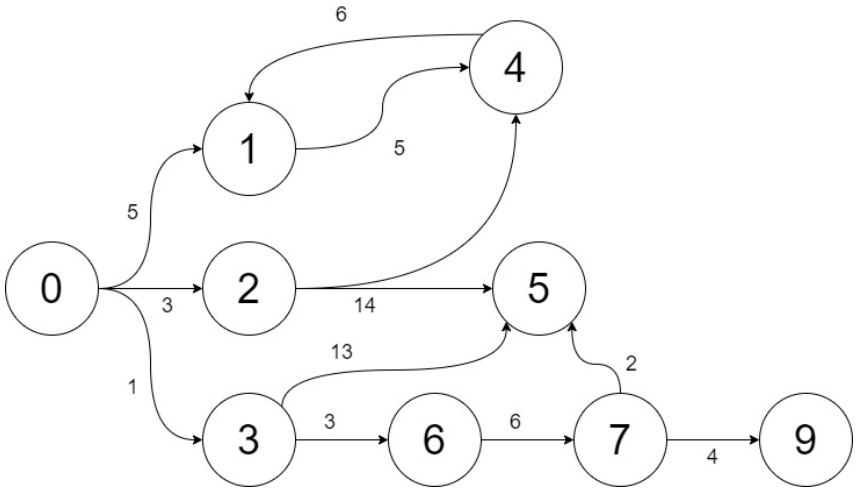
다익스트라는 Greedy 기반으로 만들어진 알고리즘으로 매 주어진 상황에서 최적의 선택을 한다.

다익스트라의 장점은 BFS와 다르게 Weight가 존재하여도 최적의 경로를 보장 할 수 있다는 것이고 단점은 만약 Weight가 음수 라면 최적의 경로를 보장받지 못한다.

다익스트라 알고리즘의 동작원리는 아래와 같다.

1. 출발 정점을 기준으로 거리 정보를 업데이트하고 연결되어 있지 않은 간선은 Infinity로 설정
2. 방문하지 않은 정점 중 가장 인접한 정점을 방문
3. 방문한 정점을 거쳐 주변 정점으로 가는 최소 거리 업데이트
4. 2,3 반복

아래 그림과 같은 그래프가 있다고 가정 했을 때 0에서 5로가는 최단 거리를 구해보자



우선 출발 정점을 기준으로 거리 정보를 업데이트 하면 아래와 같다.

0	5	3	1	INF	INF	INF	INF	INF

이후 출발 점에서 가장 인접한 정점인 3번 정점을 방문하여 3번 정점을 거쳤을 경우의 거리 정보를 업데이트 하면 아래와 같다.

0	5	3	1	INF	14	4	INF	INF

같은 방법으로 모든 노드를 방문하여 업데이트를 하면 순서대로 아래 표와 같이 업데이트가 된다

0	5	3	1	17	14	4	INF	INF
0	5	3	1	17	14	4	10	INF
0	5	3	1	10	14	4	10	INF
0	5	3	1	10	12	4	10	13
0	5	3	1	10	12	4	10	13
0	5	3	1	10	12	4	10	13
0	5	3	1	10	12	4	10	13

위와 같이 최종적으로 0번 정점에서 5번정점의 최소 거리는 12가 된다.

<BELLMANFORD>

벨만 포드 알고리즘은 최소 비용 탐색 알고리즘으로 다익스트라와 달리 Greedy하게 인접한 정점을 이용하여 찾는 것이 아닌 모든 경우의 수를 다 탐색해 가며 최소 비용을 찾는다.

장점으로는

- Greedy하게 동작하는 것이 아닌 모든 경우에 대하여 동작한다.
- 음의 간선이 있어도 정상 동작을 한다.

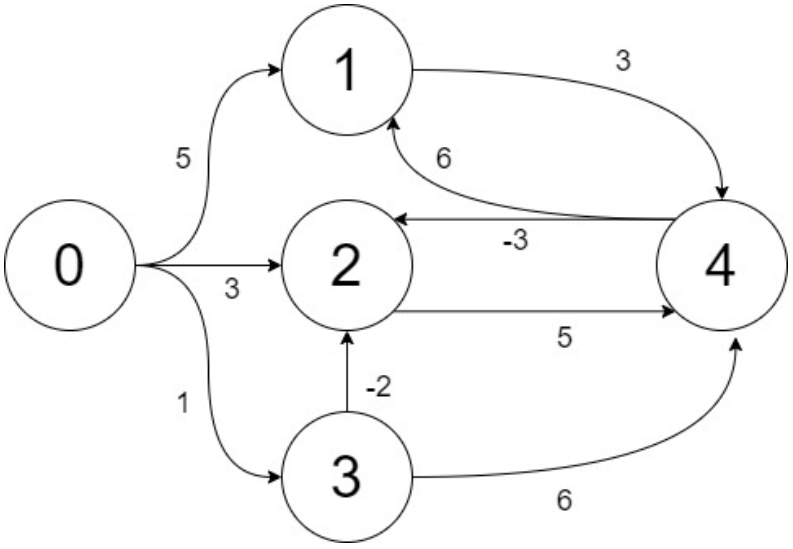
가 있고, 단점으로는

- 모든 경우의 수를 탐색하여 다익스트라 탐색법보다 시간이 오래걸린다.
- 음의 사이클이 발생하면 동작이 불가능하다.

이 있다.

단, 모든 간선들을 탐색하지만 간선을 잇는 출발 정점은 한번이라도 계산된 정점 이어야 한다. 즉, infinity이면 안된다.

예시로 아래 같은 그림이 있고 0번 정점에서 4번 정점의 최소 비용을 벨만 포드 알고리즘으로 구해보자.



우선 거리 정보를 시작점을 기준으로 업데이트 하면 아래와 같다.

0	5	3	1	INF

우선, 1번 정점을 거쳐가는 모든 경우의 수에 대해서 계산하여 거리 정보를 업데이트 하면 1번 정점을 거치는 정점은 0번과 4번 이지만 4번 정점은 아직 한번도 계산되지 않은 정점으로 값이 INF 이므로 제외하고 계산을 진행하여야 하지만, 0번 정점이 시작점 일 때, 계산을 하면 0 -> 1 -> 4가 성립하기 때문에 4번 정점이 계산된 정점으로 바뀌어 계산이 가능해진다.

최적의 거리 정보를 업데이트 하면 아래와 같다.

0	5	3	1	8

위와 같이 2번 정점을 기준으로 최적의 거리 정보를 업데이트 하면 아래와 같다. 하지만 현재 거리 정보 보다 최적의 경로가 없으므로 업데이트가 되지 않는다.

0	5	3	1	8

3번 정점

0	5	-1	1	7

4번 정점

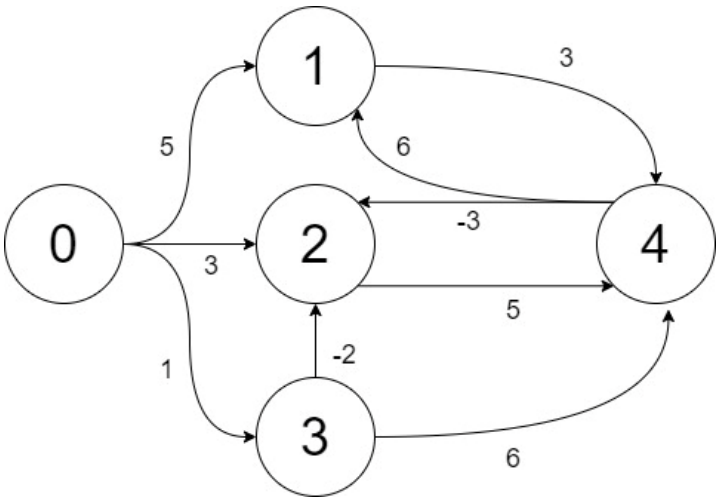
0	5	3	1	7

이렇게 모든 정점을 기준으로 최소 비용을 탐색하면 위와 같다. 하지만 위 표는 최소 비용이 아니다. 최소 비용을 구하려면 0~4번 정점을 기준으로 탐색하는 것을 총 정점의 개수 -1인 4번을 더 진행하여야 한다. 그러면 0->4의 최소 거리인 4를 얻을 수 있다. 만약, 정점의 개수만큼 진행하였는데도 거리의 업데이트가 일어난다면 음의 사이클이 존재하는 것이므로 더 이상 벨만 포드 알고리즘으로 구한 최소 거리는 의미가 없어진다.

<FLOYD>

FLOYD는 다른 최단 경로를 구하는 알고리즘과 달리 3중 반복문을 이용하여 한번에 모든 노드 간 최단 경로를 구한다. 또한, 벨만 포드 알고리즘과 같이 음의 간선이 있어도 정상 작동하지만, 음의 사이클이 존재하면 동작하지 않는다. 최단 거리 정보는 2차원 Matrix에 저장한다.

아래 그래프를 기준으로 FLOYD 알고리즘 동작 예시는 다음과 같다.



우선 Matrix를 아래 와 같이 업데이트 한다.

0	5	3	1	INF
INF	0	INF	INF	3
INF	INF	0	INF	5
INF	INF	-2	0	6

INF	6	-3	INF	0

그리고 모든 정점에 대해 거쳐가는 경우와 모든 정점에서 시작하는 경우 마지막으로 모든 정점에서 도착하는 경우는 고려하여야 한다. 따라서, 정점의 개수가 위와 같이 5개라면 5x5x5 총 125번을 반복한다.
먼저 거쳐가는 정점을 정하면 0번 정점은 0번 정점으로 향하는 간선이 없으므로 넘어가게 되고 1번 정점을 거쳐가는 모든 경우를 업데이트 하면 아래와 같다.

0	5	3	1	8
INF	0	INF	INF	3
INF	INF	0	INF	5
INF	INF	-2	0	6
INF	6	-3	INF	0

이 후 2번 정점을 지나치는 경우를 업데이트 하면

0	5	3	1	4
INF	0	0	INF	3
INF	INF	0	INF	5
INF	INF	-2	0	3
INF	6	-3	INF	0

이런 식으로 모든 정점을 거쳐가는 모든 시작점과 도착점의 경우를 진행하면 음의 사이클이 없다면 모든 정점에서 다른 모든 정점으로의 최소 비용을 구할 수 있다.
음의 사이클의 존재 유무는 3중 반복 문을 마치고 자기 자신에 대한 거리 비용이 0이 아니라면 음의 사이클이 발생한 것이다.

<RABINKARP>
라빈 카프 알고리즘은 두 문자열을 비교하는 알고리즘으로 문자열의 아스키 코드 값에 2의 n제곱수를 곱한 합으로 해시 구조를 만든다. 이 값을 기반으로 두 문자열의 일치 여부를 알 수 있다.
해시로 만드는 방법은 각 문자의 아스키 코드 값에 2^index값을 곱한 뒤 전부 더하면 된다.
예를 들어, "ABC"의 해시 값은 A,B,C의 아스키 코드 값은 각각 65,66,67 이므로
"ABC" 해시 값= 65* 2^2 + 66* 2^1 +67*2^0=459 이다. 이 해시 값은 일반적으로 문자가 다르면 다르게 나오므로 이 해시 값을 통하여 문자열 비교가 가능하다.

위에서 설명한 최단 거리 탐색 기법은 BFS, Dijstra, Bellmanford, Floyd가 있다.
이 들의 시간 복잡도는 아래와 같다.

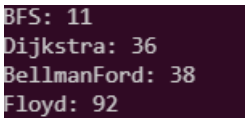
name	time complexity
BFS	O(V+E)
Dijstra	O(V^2)
Bellmanford	O(V*E)
Floyd	O(V^3)

BFS의 시간 복잡도는 인접 리스트와 인접 행렬 일 때로 나뉜다. 인접 행렬 일 때에는, 연결 되어 있지 않는 점도 확인 하므로 O(V^2)의 시간이 생기고, 인접 리스트는 간선으로 연결된 정점에 대해서만 확인 하므로 V+E의 시간이 발생한다.

Dijstra의 시간 복잡도는 O(V^2)이다. 이 경우에는 매번 반복문을 통해 방문하지 않은 정점을 판별 및 탐색하기 때문에 O(V^2+E) 따라서, O(V^2)이 된다. 하지만 우선 순위 큐를 이용하면 우선 순위 큐에 추가되는 정점의 최대 개수는 O(E)의 시간이 걸리고, 우선 순위 큐에 원소를 이동하는 데 logV가 걸리므로 총 시간은 O(E*logV)가 걸린다.

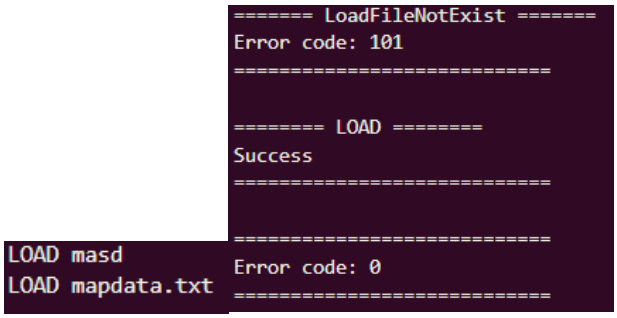
Bellmanford의 시간 복잡도는 모든 정점에서 연결되어 있는 모든 간선을 탐색해주어야 하므로 O(VE)이다.

Floyd는 모든 정점에 대하여 시작점, 거쳐가는 점, 도착점으로 3중 반복문으로 구현하므로 O(V^3)의 시간 복잡도를 갖는다.
따라서, 대략적인 성능이 뛰어난 순서는 BFS>=Dijstra>=Bellmanford>=Floyd이다.
이는 간선의 수인 E에 따라서 차이가 발생 할 수 있다. 하지만 대략적으로 E가 V보다 클 경우가 많다. 또한, BFS는 Weight 값이 1일 때만 동작하기 때문에 다른 알고리즘과 성능 비교 대상으로 적합하지 않다.
아래는 이번에 구현한 프로젝트에서 실행 시간을 측정한 수치이다. 이론 상의 성능과 비슷한 결과가 나온 것을 볼 수 있다.

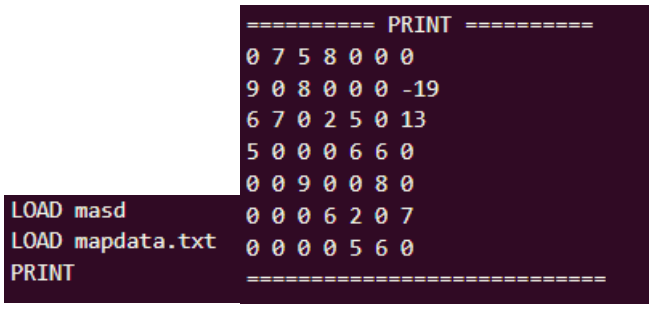


Result

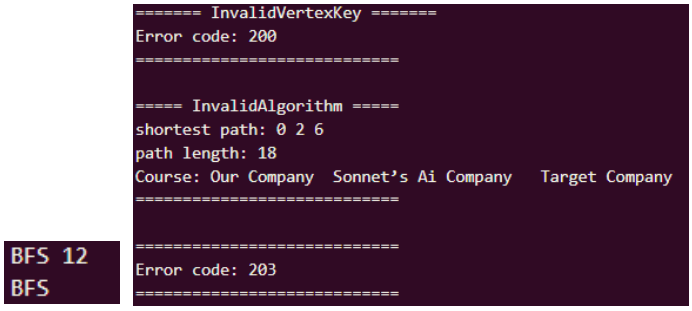
다음 LOAD 명령어의 성공적 예시와 에러 예시이다. 존재하지 않은 파일 명을 입력 하였을 때, 오류가 출력 되었고 적절한 파일 명을 입력 했을 때, 명령어가 성공적으로 수행 되었다.



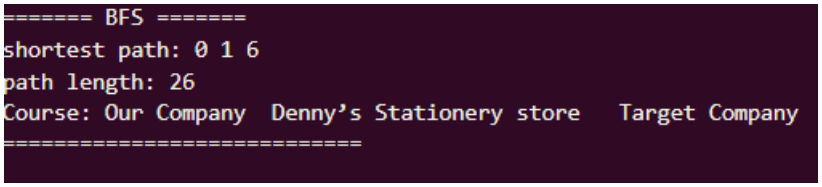
다음은 PRINT 명령어이다. LOAD를 한 후의 거리 정보가 성공적으로 출력된 것을 볼 수 있다.



BFS는 우선 에러가 발생 하였을 경우를 먼저 보면 아래와 같다. 첫번째 명령어는 잘못된 인자를 입력하여 에러가 발생 하였고, 두 번째는 그래프에 음의 간선이 존재하여 BFS가 동작은 하지만 에러 코드가 출력 되었다.



다음은 BFS의 성공적인 예시이다.



다음은 Dijkstra의 에러 예시이고 조건은 BFS와 동일한 조건이다.

```
DIJKSTRA 12
DIJKSTRA
```

```
======= InvalidVertexKey =======
Error code: 200
============================


===== InvalidAlgorithm =====
shortest path: 0 2 6
path length: 18
Course: Our Company  Sonnet's Ai Company   Target Company
============================

============================
Error code: 203
============================
```

다음은 Dijkstra의 성공 예시이다.

```
====== DIJKSTRA ======
shortest path: 0 2 6
path length: 18
Course: Our Company  Sonnet's Ai Company   Target Company
============================
```

다음은 Bellmanford의 에러 예시이다.

인자가 잘못 들어간 경우

```
======= InvalidVertexKey =======
Error code: 200
============================
```

음의 사이클이 발생한 경우

```
Our Company / 0 7 5 8 0 0 0
Denny's Stationery store / 9 0 8 0 0 0 19
Sonnet's Ai Company / -6 7 0 2 5 0 13
Tom's Accountant office / 5 0 0 0 6 6 0
Harry's Computer Repair / 0 0 9 0 0 8 0
Lin's Medical Company / 0 0 0 6 2 0 7
Target Company / 0 0 0 0 5 6 0
```

```
======= NegativeCycleDetected =======
Error code: 204
============================
```

목적지가 연결이 안되어 있는 경우

```
Our Company / 0 7 5 8 0 0 0
Denny's Stationery store / 9 0 8 0 0 0 0
Sonnet's Ai Company / 6 7 0 2 5 0 0
Tom's Accountant office / 5 0 0 0 6 6 0
Harry's Computer Repair / 0 0 9 0 0 8 0
Lin's Medical Company / 0 0 0 6 2 0 0
Target Company / 0 0 0 0 5 6 0
```

```
======= VertexKeyNotExist =======
Error code: 201
============================
```

Bellmanford 성공 예시

아래를 보면 Bellmanford는 음의 간선이 있어도 정상적을 동작함을 알 수 있다.

```
========== BELLMANFORD ==========
shortest path: 0 1 6
path length: -12
Course: Our Company  Denny's Stationery store  Target Company
============================
```

다음은 FLOYD 에러 예시이다.

음수 사이클이 존재하는 경우

```
========== PRINT ==========
0 7 5 8 0 0 0
9 0 8 0 0 0 -19
-6 7 0 2 5 0 13
5 0 0 0 6 6 0
0 0 9 0 0 8 0
0 0 0 6 2 0 7
0 0 0 0 5 6 0
============================

======= NegativeCycleDetected =======
Error code: 204
============================
```

다음은 FLOYD 성공적 예시이다.

```
========== PRINT ==========
0 7 5 8 0 0 0
9 0 8 0 0 0 -19
6 7 0 2 5 0 13
5 0 0 0 6 6 0
0 0 9 0 0 8 0
0 0 0 6 2 0 7
0 0 0 0 5 6 0
============================

========== FLOYD ==========
0 7 2 0 -7 -6 -12
-2 0 -5 -7 -14 -13 -19
5 7 0 0 -7 -6 -12
5 12 7 0 -2 -1 -7
14 16 9 9 0 3 -3
11 18 11 6 2 0 -1
17 21 14 12 5 6 0
============================
```

또한 Floyd는 끊겨 있는 정점이 존재하여도 출력이 가능하다.

```
========== PRINT ==========
0 7 5 8 0 0 0
9 0 8 0 0 0 0
6 7 0 2 5 0 0
5 0 0 0 6 6 0
0 0 9 0 0 8 0
0 0 0 6 2 0 0
0 0 0 0 5 6 0
============================

========== FLOYD ==========
0 7 5 7 10 13 INF
9 0 8 10 13 16 INF
6 7 0 2 5 8 INF
5 12 10 0 6 6 INF
15 16 9 11 0 8 INF
11 18 11 6 2 0 INF
17 21 14 12 5 6 0
============================
```

만약 거리 정보 데이터가 없는 경우에 아래 모든 명령어가 동일한 에러코드를 출력한다.

```
PRINT
FLOYD
BFS
BELLMANFORD
DIJKSTRA
```

```
======= GraphNotExist =======
Error code: 202
============================

======= GraphNotExist =======
Error code: 202
============================

======= GraphNotExist =======
Error code: 202
============================

======= GraphNotExist =======
Error code: 202
============================

======= GraphNotExist =======
Error code: 202
============================
```

LOADREPORT 명령어의 예시는 아래와 같다. LOAD와 비슷하게 동작하는 것을 볼 수 있다.

```
LOADREPORT sfkja
LOADREPORT reportdata.txt
```

```
======= FaildtoUpdatePath =======
Error code: 5
============================

======== LOADREPORT ========
Success
============================

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

RABINKARP의 에러 예시는 아래와 같다.

너무 많은 인자가 들어온 경우

```
RABINKARP asdasfsafasdf
```

```
======= InvalidOptionNumber =======
Error code: 1
============================
```

찾고자 하는 문자열이 없을 경우

```
RABINKARP asdfxd
```

```
========== RABINKARP ==========
NO DUPLICATE TITLE EXISTS
============================
```

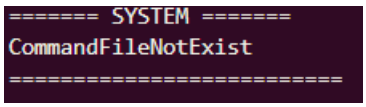
성공한 예시는 아래와 같다.

```
RABINKARP big data
```

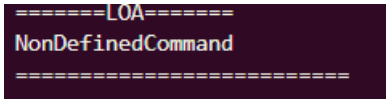
```
========== RABINKARP ==========
Big Data wlasdld dsfsd
Seoul Metropolitan Government's Big Data Promotion Project Big data
Plans to promote the pet plant big data
============================
```

이 외의 에러 예시는 다음과 같다.

run함수 인자로 받은 커맨드 파일이 존재하지 않을 경우



잘못된 명령어를 입력한 경우



Consideration

이 프로젝트는 다양한 최단 거리 탐색 알고리즘과 문자열 비교 알고리즘인 Rabinkarp를 구현하였다. 이 프로젝트를 구현하며 가장 시간을 많이 쓰고 가장 어렵게 느껴졌던 부분은 BFS이다. 물론 실제 알고리즘 난이도는 BFS가 다른 알고리즘에 비하여 낮지만 BFS는 이 프로젝트에서 처음으로 구현하는 최단 거리 탐색 알고리즘인 것도 있었지만, 기존 배경지식에 의존하면 BFS는 넓이 탐색이기 때문에 정점을 찾는 용도로만 쓰이고 최단 거리는 구하지 못할 것이라고 생각하였다.
BFS를 진행하며 목표 정점을 찾는 것은 간단 하였지만 목표 정점을 찾았을 경우 시작점에서 목표 점까지의 경로를 찾는 방법을 찾지 못하여 난항을 겪었다. 이를 해결한 방법은 정점의 수만큼의 크기를 같는 배열을 생성하여 BFS를 진행하며 해당 배열의 Index를 현재 정점이라 가정하고 안의 요소는 이전 경로의 정점의 Index를 넣어주는 방법으로 해결하였다. 그 이후, 다른 최단 거리 알고리즘들은 BFS를 기반으로 조금씩 변형하며 만들어 실제 알고리즘 난이도에 비하여 쉽게 느껴졌다. 두 번째 문제는 프로젝트를 완성하고 검증하는 과정에서 발견하였다. Rabinkarp를 구현하였을 때, 결과가 제대로 나왔지만 인덱스에 2의 제곱수를 곱하는 것이 아닌 아스키 코드 값을 제곱하여 더하였다. 이 후, 잘못됨을 알고 고쳤으나 이번에는 2의 제곱수를 8,4,2,1 순으로 곱하여 한다면 반대로 1,2,4,8로 곱하였다. 물론 이는 이론 상 Rabinkarp와 동일한 동작을 하고 실제로 결과에 차이는 없었다. 이는 나의 학습의 부족함을 보여 주었고, 문제의 정확한 이해의 중요성을 알려주었다.
이 프로젝트를 진행하며 알게 된 점이 있다면 궁금한 점도 많이 생기게 한 프로젝트였다.
첫 번째는 최단 거리 알고리즘이 항상 최적의 거리를 보장하는지 이다. 이는 따로 학습을 통하여 해결할 예정이다. 그리고 두 번째는 Rabinkarp의 정확도에 대해서다. 물론 강의 자료 등에서도 일반적이라고 명시가 되어 있지만 이는 찾고자 하는 문자열의 길이가 짧을수록 정확도의 신뢰성이 떨어진다. 실제로 "he"와 "as"는 Rabinkarp 알고리즘을 통해 해시 값으로 변경하면 309로 동일한 해시 값은 갖는다. 실제 as를 검색 하였을 때, 자주 쓰이는 단어인 the가 들어간 문장은 모두 출력 되었다. 다음에 기회가 된다면 Rabinkarp 알고리즘 보다 높은 정확도를 보장하는 알고리즘이 있는지 학습을 할 예정이다.