Data Structure Project

Project #3

담당교수 : 신 영 주 교수님

제출일 : 2019. 12. 06.

학과: 컴퓨터정보공학부

학번 : 2017202037

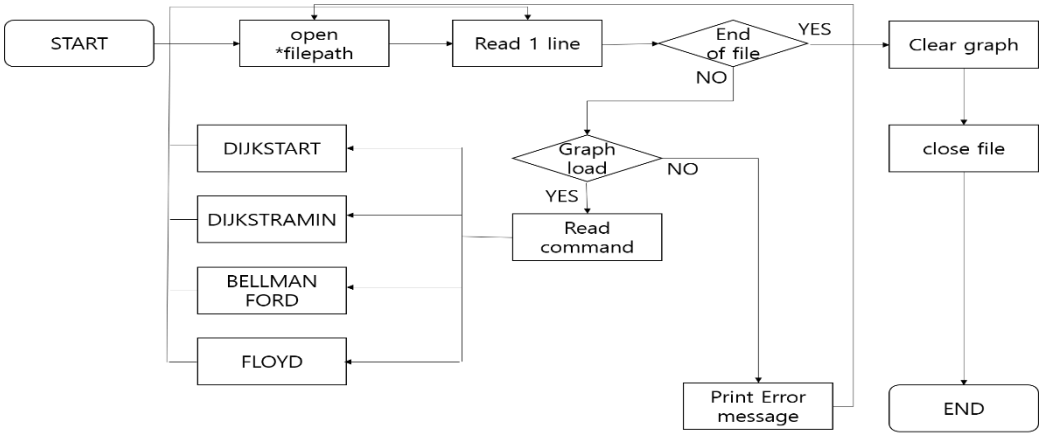
이름 : 오 민혁

1. Introduction

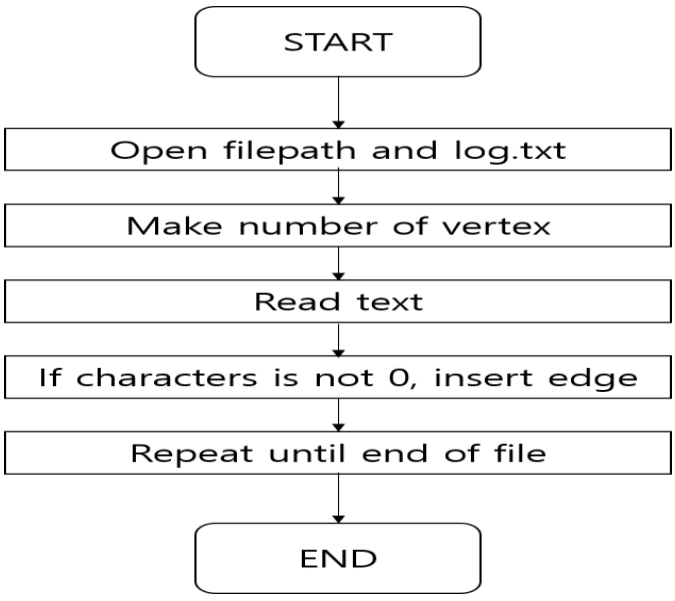
이번 프로젝트에서는 그래프를 이용해 최단경로 찾기 프로그램을 구현한다. 각 섬 간의 거리에 대한 정보가 저장되어 있는 텍스트 파일을 통해 그래프를 구현한 후, Dijkstra, Bellman-Ford, Floyd 알고리즘을 통해 최단 경로와 거리를 구하고 그 결과를 log.txt 파일에 저장한다. 섬의 그래프 정보는 방향성과 가중치를 모두 가지고 있고, Matrix 형태로 저장되어 있다.

2. Flow Chart

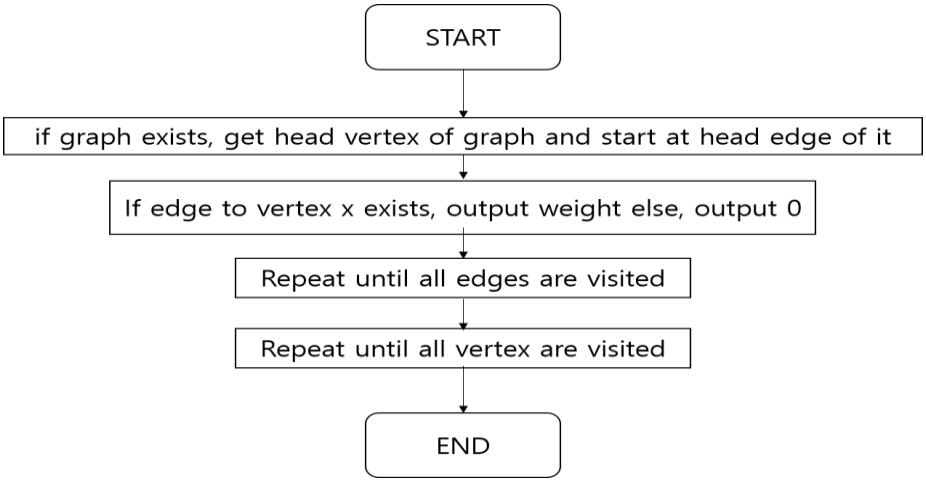
<Manager>



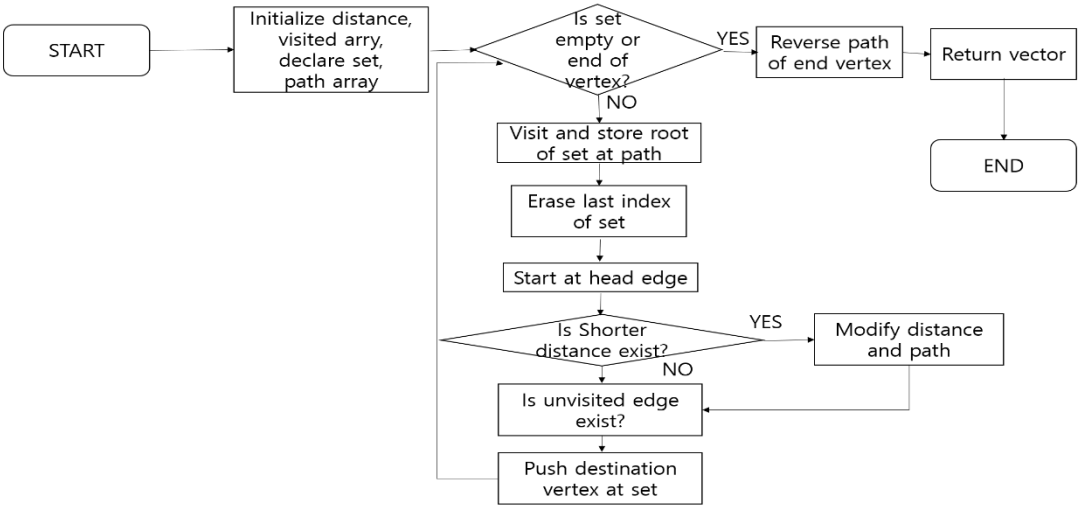
<LOAD>



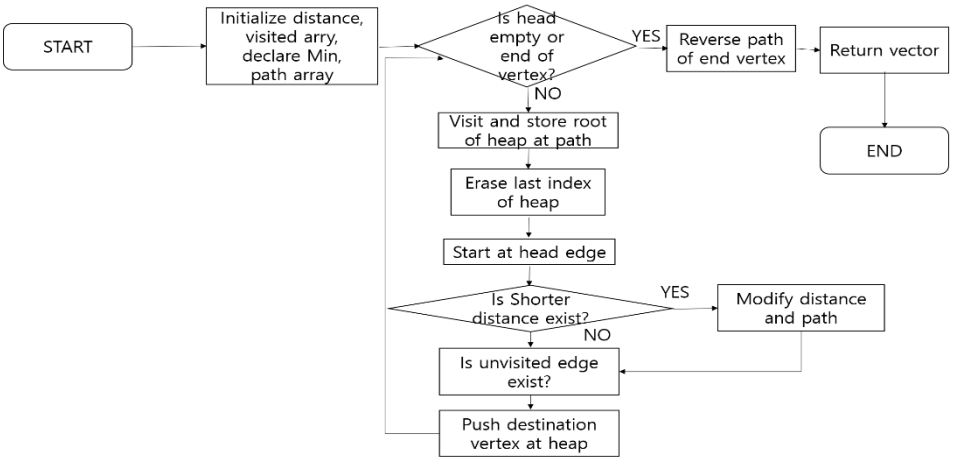
<PRINT>



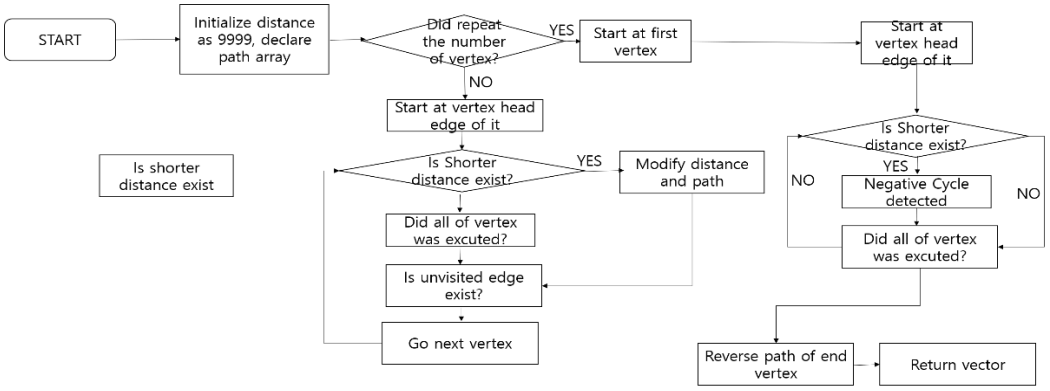
<DIJKSTRA>



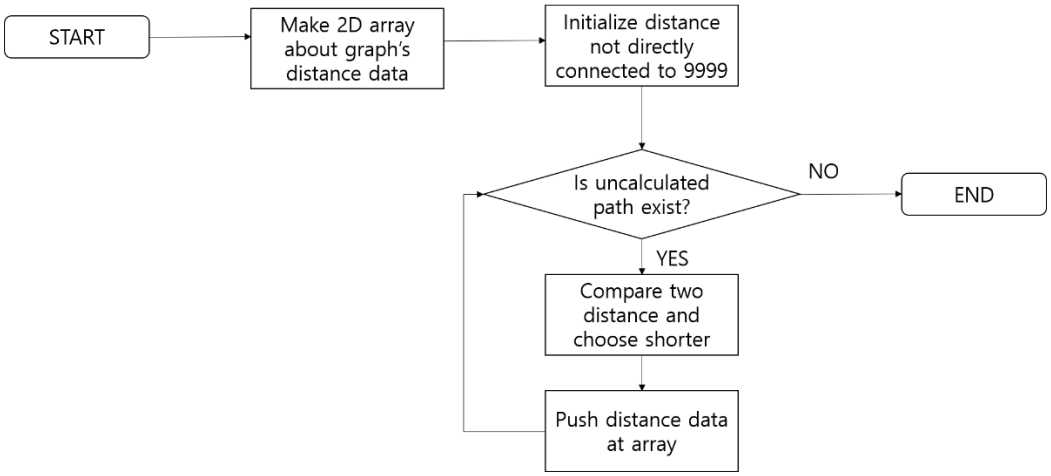
<DIJKSTRAMIN>



<BELLMANFORD>



<FLOYD>



3. Algorithm

Dijkstra

1. 모든 정점을 unvisited 상태로 표시한다. unvisited vertex의 집합(Set, Minheap)을 만든다.

2. 모든 vertex에 시험적 distance 부여한다. default를 0으로, 다른 모든 정점을 무한대로 설정한다. Default을 현재 위치로 설정한다.

3. 현재 vertex에서 unvisited 인접 vertex를 찾아 그 distance를 현재 vertex에서 계산한다. 새로 계산한 distance를 현재 distance와 min함수로 비교하여 더 작은 값을 선택하여 distance를 갱신해준다.

4. 만약 현재 노드에 인접한 모든 unvisited vertex를 계산했다면, 현재 vertex를 visited를 true로 변환하고 unvisited 집합에서 제거(Pop)한다. visited 정점은 다시 방문하지 않는다.

5. destination의 visited가 true가 되거나 (특정 두 정점 사이의 경로를 계획하고 있을 때) unvisited 집합에 있는 edge들의 distance 중 최솟값이 무한대면 알고리즘을 종료한다.

6. 아니면 distance가 가장 작은 다음 unvisited edge를 새로운 "현재 위치"로 선택하고 3단 계로 되돌아간다.

Bellman-Ford

1. 시작은 dijkstra 알고리즘과 동일하다. dijkstra 알고리즘으로 최단 경로를 구한다.

2. 음의 사이클의 유무를 확인하기위해 반복을 한번 더 한다.

3. 첫 번째 반복에서 구한 최단경로보다 더 짧은 경로가 존재한다면 음의 사이클이 존재함을 의미한다.

4. 모든 vertex를 방문했다면 end vertex의 path를 역추적한다.

FLOYD

플로이드 알고리즘은 각각의 edge 쌍을 지나는 그래프의 모든 path를 비교한다.

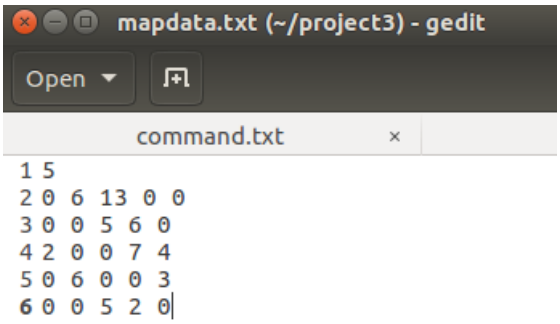
1. 2차원 배열을 만들고 그래프의 distance data를 저장한다. (두 정점이 직접적으로 연결되어 있지 않으면 무한대 값, 자기 자신으로 가는 거리는 0으로 한다.)

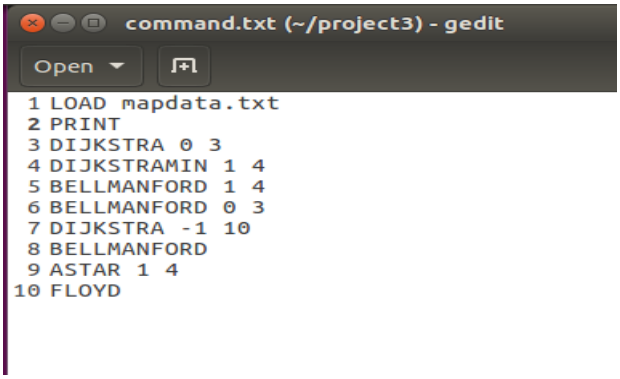
2. 경유지 1~n 까지 순회하여 2차원 테이블을 업데이트 한다. 경유지를 순회하는 알고리즘에 대한 반복문 코드는 아래와 같다.

```
for (int k = 1; k < n; ++k) {
        for (int i = 1; i < N; ++i) {
                for (int j = 1; j < N; ++j) {
                        if (dp[i][k] != INF && dp[k][j] != INF)
                        dp[i][j] = min(dp[i][k] + dp[k][j], dp[i][j]);
```

```
                    }
                }
}
```

4. Result Screen






```
1 ====== LOAD ======
2 Success
3 ====================
4
5 ==================
6 Error code:0
7 ================
8
```

LOAD 명령어이다. mapdata.txt를 인자로 입력해주어 mapdata.txt에 있는 데이터들로 graph를 생성해준다.

성공적으로 LOAD가 되면 Success가 출력되고 에러코드가 0이 출력된다.

```
 9 ====== PRINT ======
10 0 6 13 0 0
11 0 0 5 6 0
12 2 0 0 7 4
13 0 6 0 0 3
14 0 0 5 2 0
15 ==================
16
17 ==================
18 Error code:0
19 ================
20
```

PRINT 명령어이다. 성공적으로 그래프를 2차원 배열 형태로 출력하고, 아무 이상 없이 실행되어 에러 코드가 0이다.

```
21 ====== DIJKSTRA ======
22 shortest path: 0 1 3
23 sorted nodes: 0 1 3
24 path length: 12
25 ====================
26
27 ==================
28 Error code:0
29 ================
30
31 ====== DIJKSTRAMIN ======
32 shortest path: 1 2 4
33 sorted nodes: 1 2 4
34 path length: 9
35 ====================
36
37 ==================
38 Error code:0
39 ================
40
```

DIJKSTRA, DIJKSTRAMIN 명령어이다.

0에서 3으로 가는 최단 거리 경로와 길이, 1에서 4로 가는 최단 거리 경로와 길이가 잘 출력되고 있는 모습이다.

```
41 ====== BELLMANFORD ======
42 shortest path: 1 2 4
43 sorted nodes: 1 2 4
44 path length: 9
45 ==================
46
47 =================
48 Error code:0
49 =================
50
51 ====== BELLMANFORD ======
52 shortest path: 0 1 3
53 sorted nodes: 0 1 3
54 path length: 12
55 ==================
56
57 =================
58 Error code:0
59 =================
```

BELLMANFORD 명령어이다. 이 역시 0에서 3으로 가는 최단 거리 경로와 길이, 1에서 4로 가는 최단 거리 경로와 길이가 잘 출력되고 있는 모습이다. 위의 DIJKSTRA와 같은 결과 값들을 가지고 있는 부분까지 보아, 최단 거리 찾기가 정확함을 알 수 있다.

```
61 ====== DIJKSTRA ======
62 InvalidVertexKey
63 ==================
64
65 =================
66 Error code:201
67 ================
68
```

다음은 vertex 입력 오류에 관한 예외처리의 결과이다. 범위 밖의 parameter를 입력하면 에러 메세지가 잘 출력된다.

```
68
69 ====== BELLMANFORD ======
70 VertexKeyNotExist
71 ==================
72
73 =================
74 Error code:200
75 ================
76
77 ====== ASTAR ======
78 NonDefinedCommand
79 ==================
80
81 =================
82 Error code:300
83 ================
84
```

VertexKeyNotExist는 parameter가 없을 때 예외처리에 대한 결과이고, NonDefinedCommand는 잘못된 명령어를 입력했을 때의 예외처리 결과이다.

```
85 ===== FLOYD =====
86 0 6 11 12 15
87 7 0 5 6 9
88 2 8 0 6 4
89 10 6 8 0 3
90 7 8 5 2 0
91 ================
92
93 =================
94 Error code:0
95 ================
96
```

다음은 FLOYD 명령어 결과이다. N x N Matrix 형태로 결과값이 올바르게 출력되는 모습이다.

5. Consideration

이번 프로젝트에서는 stack과 heap을 직접 구현해보았는데, 평소 #include 하여 사용하다가 직접 구현해보니 stack과 heap의 개념과 구조에 대해 더 정확히 이해할 수 있었다. 그리고 manager 함수를 구현할 때, 윈도우에서는 strtok_s 문법으로 사용하는 것을 리눅스에서는 strtok_r로 사용해야 한다는 점을 알게 되었는데, 윈도우에서와 리눅스에서의 C++ 문법이 다를 수도 있다는 것을 처음 알게 된 계기였다. 그리고 가장 문제가 됐던 점은 윈도우에서는 올바르게 mapdata.txt를 열어서 성공적으로 LOAD를 하였는데, 리눅스에서는 되지 않았다. 몇 시간을 헤매다가 윈도우에서 만든 txt 파일을 리눅스에 drag and drop한 게 문제가 되었나 싶어, 리눅스에서 txt파일을 만들어 실행시키니 성공적으로 작동하였다. 마지막으로 최단 거리 구하는 알고리즘을 구현해보았는데, 2차 프로젝트때 했던 kruskal 알고리즘보다는 쉬웠던 것 같다.