Data Structure Project #3

학과: 컴퓨터정보공학부

학번: 2022202065

이름: 박나림

1. Introduction

사용자의 명령어에 따라 해당하는 그래프 연산을 수행하는 프로그램을 만드는 것이 본 프로젝트의 목표이다. 그래프 정보가 저장된 텍스트 파일을 읽어서 그래프를 구성하고, 명령어에 따라 BFS, DFS. Kruskal, Dijkstra, Bellman-Ford, FLOYD, KwangWoon으로 7가지의 알고리즘을 수행한다. 대부분의 그래프는 방향성과 가중치를 모두 가지고 있으며, List 또는 Matrix 그래프로 저장된다. 명령어 설명은 다음과 같다.

LOAD: graph_L.txt 또는 graph_M.txt 파일을 읽어서 해당 그래프의 정보를 바탕으로 새로운 그래프를 구성한다. 이때 기존 그래프 정보가 존재하는 경우에는 초기화 하여 새로 생성한다.

PRINT: 각각의 그래프 종류에 맞게 저장된 그래프를 출력하는 명령어이다. 모든 그래프의 vertex는 1부터 시작하며, 이에 맞게 오름차순으로 출력한다. edge도 vertex에 맞게 오름차순으로 출력하는 형식이다.

BFS: 추가 인자로 방향성과 시작 vertex를 전달받으면 그에 맞는 BFS 연산을 수행한다. 현재 명령을 수행하면서 방문하는 vertex의 순서대로 결과를 출력한다.

DFS: 추가 인자로 방향성과 시작 vertex를 전달받으면 그에 맞는 DFS 연산을 수행한다. 현재 명령을 수행하면서 방문하는 vertex의 순서대로 결과를 출력한다.

KRUSKAL: 저장된 그래프에서 MST를 구하고 이를 구성하는 edge들의 vertex 값을 오름차순으로 출력한다. 추가로 weight들의 총합을 마지막에 출력하는 형식이다.

DIJKSTRA: 추가 인자로 방향성과 시작 vertex를 전달받으면 그에 맞는 DIJKSTRA 연산을 수행한다. 그 결과인 vertex, shortest path, cost를 순서대로 출력하며, 이때 shortest path는 해당 vertex에서 기준 vertex까지의 경로를 역순으로 출력하는 형식이다. 기준 vertex에서 도달할 수 없는 경우에는 'x'를 출력하도록 한다.

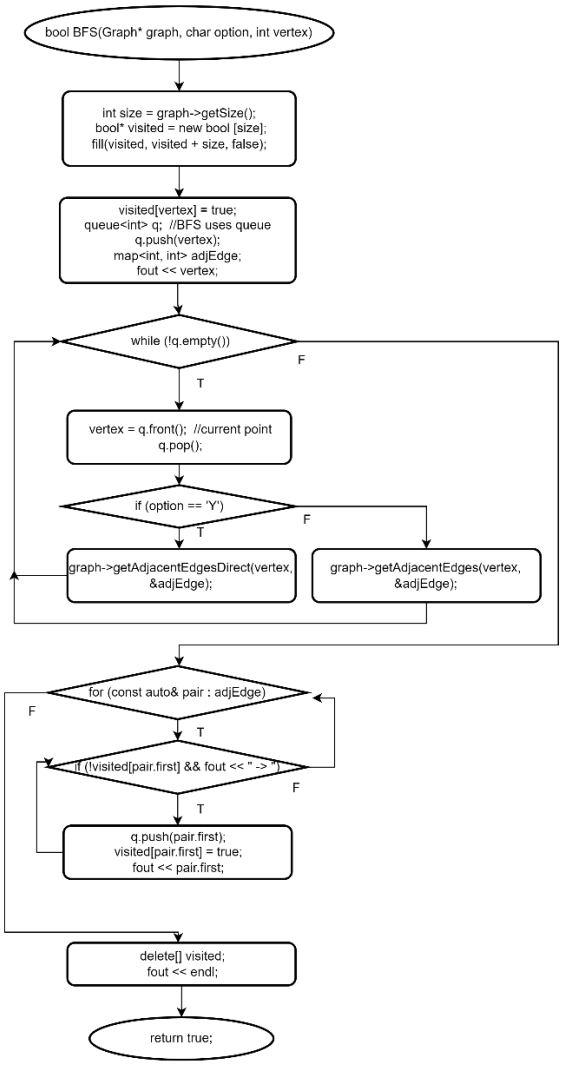
BELLMANFORD: 추가 인자로 방향성과 start vertex, end vertex를 전달받으면 그에 맞는 BELLMANFORD 연산을 수행한다. 음수인 weigth가 있는 경우에도 동작해야 하며, 음수 cycle때는 오류 코드를 출력한다. 위와 마찬가지로 도달할 수 없는 경우엔 'x'를 출력한다.

FLOYD: 추가 인자로 방향성을 전달받으면 그에 맞는 FLOYD 연산을 수행한다. 모든 vertex의 쌍에 대해서 시작점에서 끝점으로 가는 데 필요한 비용의 최솟값을 행렬 형태로 출력하는 형식이다. 도달할 수 없는 경우에는 'x'를 출력한다.

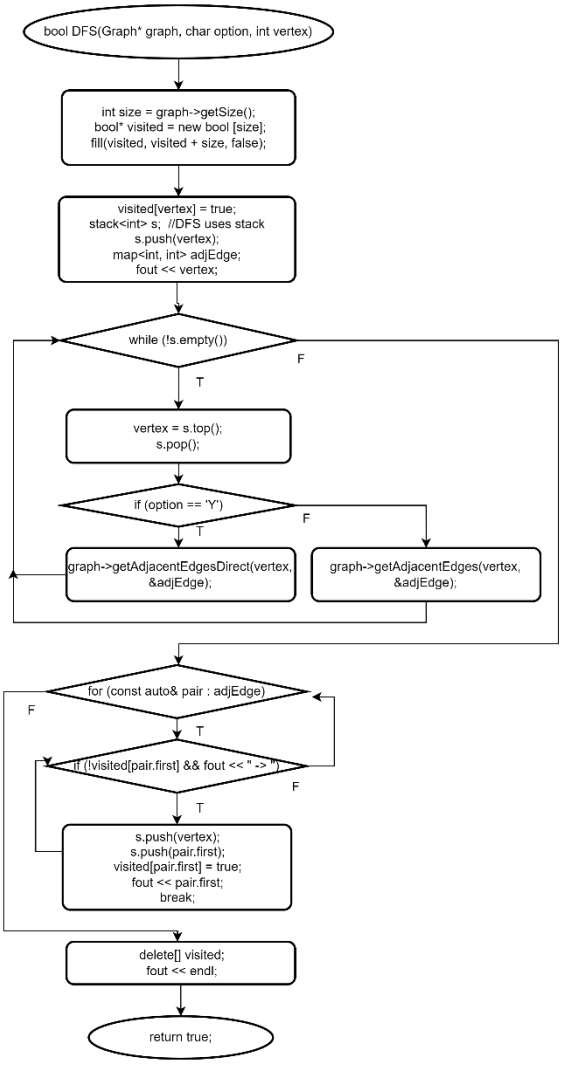
KWANGWOON: 현재 정점에서 방문할 수 있는 정점들이 홀수 개일 경우 해당 번호들의 가장 큰 번호로 방문을 시작하고, 짝수 개면 가장 작은 번호로 방문을 시작하여 결과를 출력한다. 이 명령어는 List 그래프에서만 동작한다.

2. Flowchart

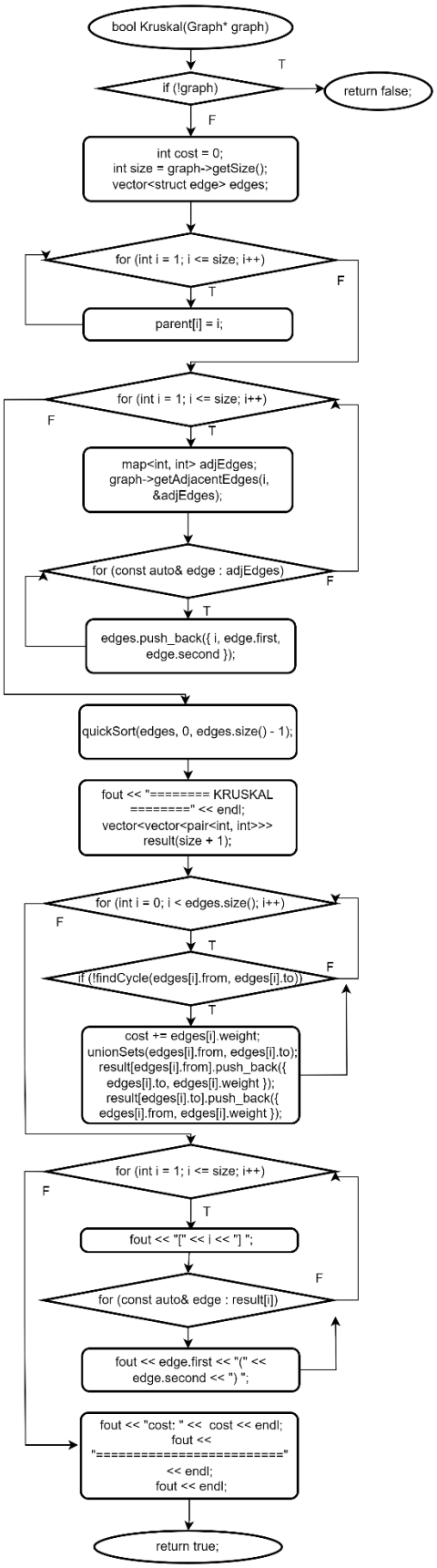
1) BFS



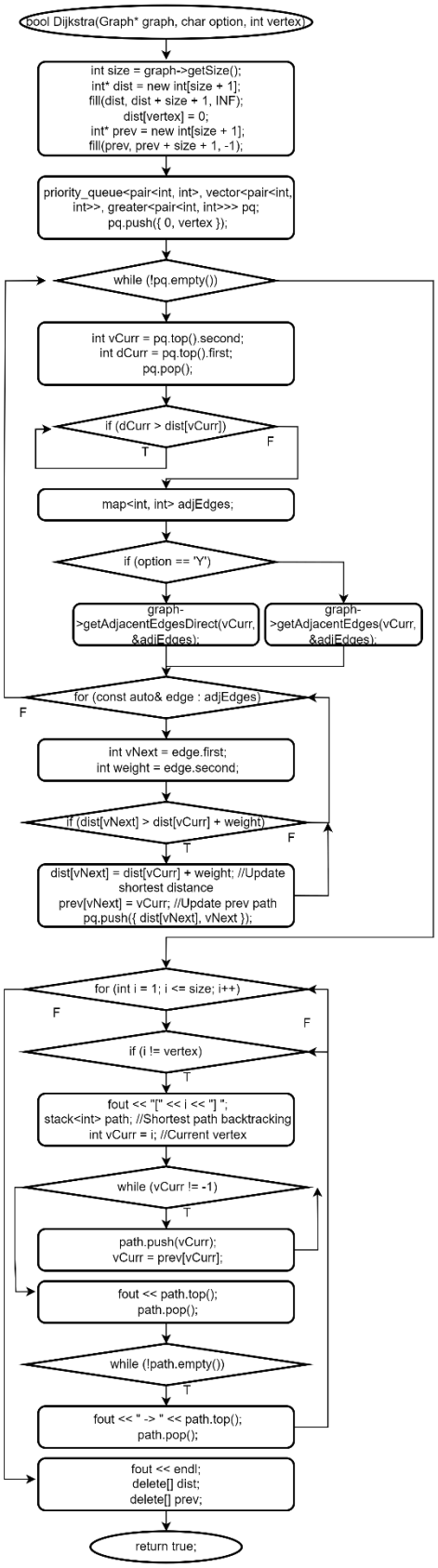
2) DFS



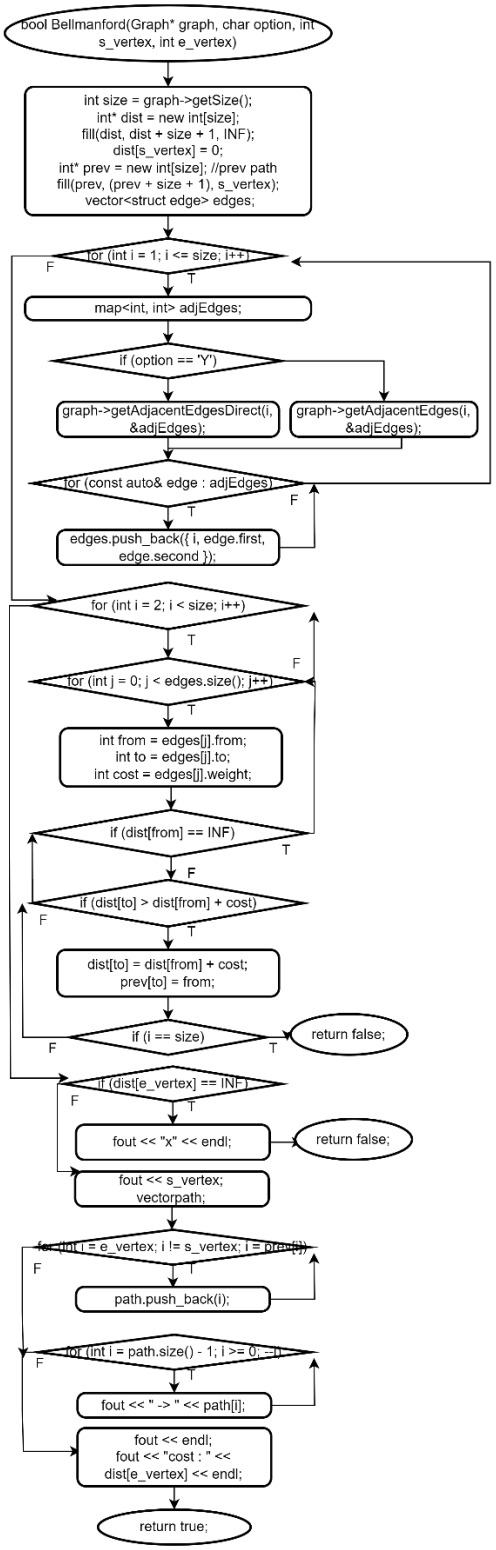
3) Kruskal



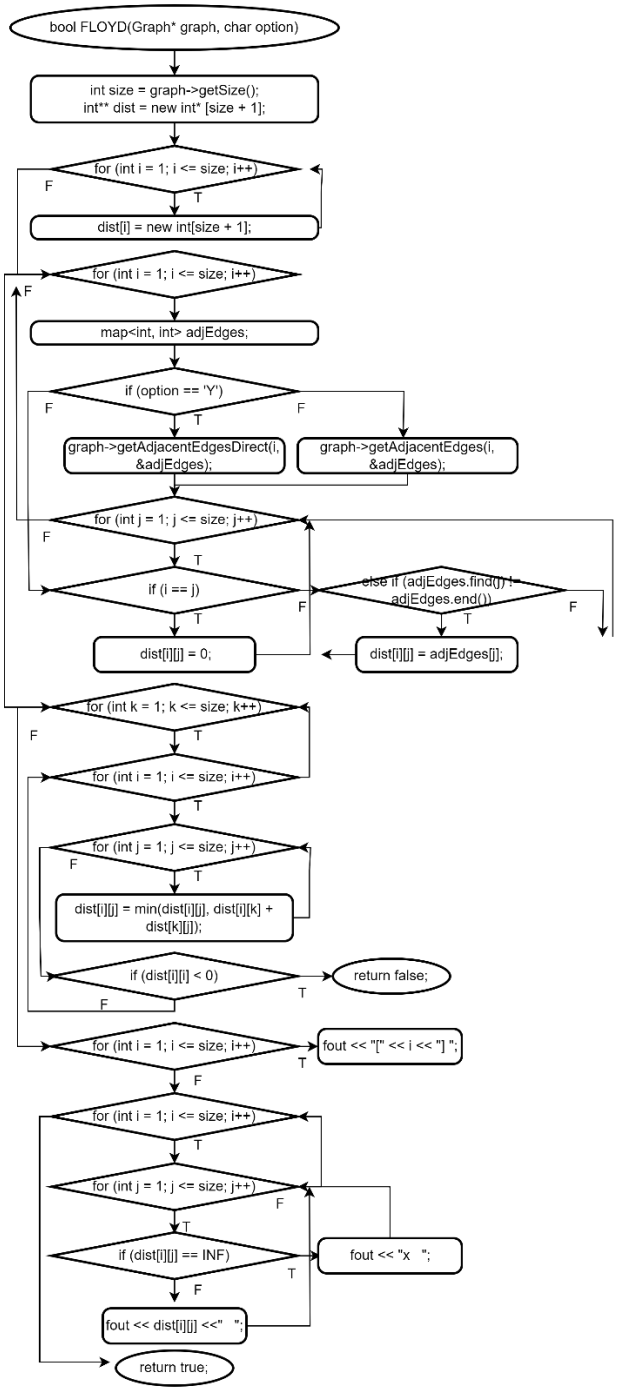
4) Dijkstra



5) Bellman-Ford



6) FLOYD



3. Algorithm

1) BFS

BFS(Breadth-First Search)는 그래프의 각 노드들을 넓이를 우선으로 하여 순차적으로 탐색해 최단 경로를 찾는 알고리즘이다. 시작 정점을 방문한 다음 해당 정점과 인접한 정점들을 모두 방문한 후, 그 다음에 해당 정점들과 연결된 인접 정점들을 차례로 방문하는 형식이다. 동작 순서는 다음과 같다.

각 정점의 방문 여부를 저장하는 배열을 visited로 동적할당 하고, false로 초기화를 한다. BFS는 queue를 사용하여 구현되므로, queue를 생성한 뒤 시작 정점을 queue에 추가하고 visited에도 true로 저장한다. 그리고 map을 이용하여 인접 정점에 대한 간선들을 저장한 뒤, while문을 통해 queue가 비어 있지 않은 동안 queue에서 정점을 하나씩 꺼내어 방문을 시작한다. 인접 정점이 방문 되지 않았다면 queue에 삽입하고 방문 여부를 갱신하며 해당 간선을 출력한다.

2) DFS

DFS(Depth-First Search)는 그래프의 각 노드들을 깊이를 우선으로 하여 순차적으로 탐색하는 알고리즘이다. 시작 정점을 방문한 다음 그 정점에서 null이 아닐 때까지 계속 아래 단계로 탐색한다. DFS는 재귀함수 또는 stack으로 구현된다. 동작 순서는 다음과 같다.

각 정점의 방문 여부를 저장하는 배열을 visited로 동적할당 하고, false로 초기화를 한다. 여기서는 stack으로 구현되므로, stack을 생성한 뒤 시작 정점을 추가하고 visited에도 true로 저장한다. while문으로 stack이 비어 있지 않은 동안 stack에서 정점들을 하나씩 꺼내어 방문을 시작한다. 현재 정점과 연결된 간선들에 대한 for문을 시작하고, 해당 간선의 다음 정점이 방문 되지 않았다면 stack에 추가하고 visited로 방문 여부를 갱신한다. 이때 해당 간선을 출력하고 for문을 종료한다. 깊이 우선 탐색이므로 한 간선만을 선택하기 때문이다. 이러한 과정을 모든 정점에 대해 반복한 뒤 동적 배열을 해제하며 마친다.

3) Kruskal

Kruskal은 그래프의 최소 비용 신장 트리인 MST(Minimum Spanning Tree)를 찾는 알고리즘이다. 그래프의 간선들을 가중치의 오름차순으로 정렬한 뒤 첫번째 가중치부터 선택하여 MST에 추가한다. 이때 cycle이 형성되지 않는지 확인하며 추가한다. Tree에서는 cycle이 생기면 안되기 때문이다. 동작 순서는 다음과 같다.

모든 간선들을 저장할 struct vector를 생성한 뒤 저장한다. 또한 각 정점의 부모를 자기 자

신으로 초기화하는 Union, Find 함수를 위해 parent 배열을 동적할당 한다. 그리고 간선들을 가중치의 오름차순으로 정렬하기 위해 quickSort 함수를 사용한다. 이때 segment size에 따라 기준치보다 작으면 insertionSort, 크면 quickSort함수를 사용하도록 구성한다. MST를 저장할 새로운 vector를 생성한 뒤 간선들을 순회하면서 저장한다. findCycle 함수를 통해 선택한 간선이 cycle을 이루지 않는지 확인하고, cycle이 안된다면 cost에 가중치들을 더한 후 unionSets 함수를 통해 선택한 간선의 양 끝 정점들을 하나의 집합으로 합친다. 이러한 간선을 MST에 저장하는 과정을 반복적으로 수행하여 결과를 출력한 뒤 마친다.

4) Dijkstra

Dijkstra는 그래프의 시작 정점으로부터 다른 모든 정점들까지의 최단 경로와 그에 대한 최단 거리를 찾는 알고리즘이다. 우선 순위 큐를 이용하여 구현될 수 있다. 동작 순서는 다음과 같다.

각 정점까지의 최단 거리를 저장할 dist 배열과 각 정점까지의 최단 경로를 저장할 prev 배열을 동적 할당한 뒤 초기화 한다. priority_queue(pq)를 이용하여 최단 거리와 정점을 저장한다. 이때 비교 타입은 greater를 사용하여 가중치가 적은 간선이 우선 순위가 높게 설정되도록 만든다. 시작 정점을 pq에 추가한 뒤 while문으로 pq가 비어 있지 않은 동안 탐색을 반복한다. 먼저 pq에서 현재 정점과 현재 정점까지의 최단 거리를 가져온 뒤 pop으로 제거한다. 그리고 현재 정점까지의 최단 거리가 이미 갱신된 상태라면 continue하고, 아니라면 인접한 정점들을 순회하면서 갱신한다. 다음 정점까지 한번에 가는 것 보다 현재 정점을 거쳐서 다음 정점까지 가는 게 더 빠르다면 dist[vNext]=dist[vCurr]+weight;를 통해 최단 거리를 갱신한다. prev[vNext]=vCurr;로 최단 경로도 갱신한다. 이렇게 갱신된 정보는 pq에 추가한다.

위 과정이 다 끝나면 최단 경로를 출력하기 위해 stack을 생성한다. 최단 경로는 1차원 배열에 덮어씌워진 상태이므로, backtracking을 하여 경로를 stack에 저장하는 것이다. 이후 각 정점에 대한 최단 경로를 출력하며 마친다.

5) Bellman-Ford

Bellman-Ford는 그래프의 시작 정점으로부터 다른 모든 정점들까지의 최단 경로와 최단 거리를 찾는 알고리즘이다. 간선들의 개수에 제한을 걸면서 경로를 찾아가는 형식이다. Dijkstra와 달리 음의 가중치를 포함하는 그래프에서도 동작하지만 음의 사이클이 있을 때는 동작할 수 없다. 따라서 이를 확인하는 과정이 들어가 있다. 동작 순서는 다음과 같다.

최단 거리와 최단 경로를 저장할 dist와 prev 배열을 동적 할당한 뒤 초기화 한다. 모든 간선 정보를 저장할 vector를 생성한 뒤 저장하며, 탐색을 반복한다. 이때 정점이 총 n개 있을

때 간선은 n-1개가 존재하므로 그만큼만 반복한다. for문을 통해 모든 간선들을 순회하면서 각 간선의 from, to, weight 정보들을 저장해 놓은 뒤 if문으로 확인한다. dist[from]이 INF(무한대, 경로 없음)라면 continue하고, 현재 최단 거리보다 더 짧은 거리를 확인할 경우 dist[to]=dist[from]+cost;를 통해 갱신한다. prev[to]=from;으로 최단 거리도 갱신한다. 만약 그래프의 size번째 반복에서도 갱신이 된다면 이는 음의 사이클이 존재한다는 뜻이므로 false를 반환하며 종료한다. 정상적으로 끝났다면 stack으로 최단 경로를 저장하고 출력하며 마친다.

6) FLOYD

FLOYD는 그래프의 시작 정점으로부터 다른 모드 정점들까지의 최단 경로와 최단 거리를 찾는 알고리즘이다. 동적 프로그래밍을 기반으로 하여 모든 정점 쌍에 대한 최단 거리를 구한다. 방문할 수 있는 정점 개수에 제한을 걸면서 탐색을 하는 형식이다. 동작 순서는 다음과 같다.

최단 거리를 저장할 dist를 2차원 배열로 동적 할당하고 map으로 간선 정보를 가져와서 for문을 통해 초기화 한다. 자기 자신으로의 거리는 0으로, 간선이 존재하는 경우 해당 가중치로, 없는 경우 INF로 초기화 한다. 이후 3중 for문을 통해 알고리즘을 수행한다. 중간 정점 k, 시작 정점 i, 도착 정점 j에 대해 반복하며 i부터 j까지 한번에 가는 거리와 k를 거쳐가는 거리 중 짧은 경로로 갱신하여 저장한다. (dist[i][j]=min(dist[i][j], dist[i][k]+dist[k][j]);) 이때도 음의 사이클이 존재한다면 false를 반환하며 종료한다. 해당 과정을 반복한 뒤 끝나면 dist 배열의 정보를 출력하며 마친다.

4. Result Screen

10개의 정점에 대해 그래프를 구성하여, 모든 명령어에 대한 결과를 확인하였다. 그래프의 방향성에 대한 차이는 완벽히 구현을 하지 못하였기에 Directed 그래프 기준으로 출력된다.

1) LOAD

그래프 정보가 저장된 텍스트 파일을 읽어서 그래프를 구성했을 때 결과를 출력한 모습이다.

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

2) PRINT

graph_L.txt, graph_M.txt의 그래프 정보들을 각각 인접리스트, 인접행렬로 print한 결과이다.

```
========PRINT========
[1] -> (2,6) -> (3,2)
[2] -> (4,5)
[3] -> (2,7) -> (5,3) -> (6,8)
[4] -> (7,3)
[5] -> (4,4)
[6] -> (7,1)
[7] -> (5,10) -> (8,12)
[8] -> (9,23)
[9] -> (7,16) -> (10,20)
[10] -> (7,30)
=====================
```

```
========PRINT========
    [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1]  0   6   2   0   0   0   0   0   0   0
[2]  0   0   0   5   0   0   0   0   0   0
[3]  0   7   0   0   3   8   0   0   0   0
[4]  0   0   0   0   0   0   3   0   0   0
[5]  0   0   0   4   0   0   0   0   0   0
[6]  0   0   0   0   0   0   1   0   0   0
[7]  0   0   0   0   10   0   0   12   0   0
[8]  0   0   0   0   0   0   0   0   23   0
[9]  0   0   0   0   0   0   16   0   0   20
[10]  0   0   0   0   0   0   30   0   0   0
====================
```

3) BFS

BFS 알고리즘에 따라 방문한 순서대로 출력한 결과이다.

```
========BFS========
Directed Graph BFS result
startvertex: 1
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10
====================

========BFS========
Undirected Graph BFS result
startvertex: 1
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10
====================
```

4) DFS

DFS 알고리즘에 따라 방문한 순서대로 출력한 결과이다.

```
========DFS========
Directed Graph DFS result
startvertex: 1
1 -> 2 -> 4 -> 7 -> 5 -> 8 -> 9 -> 10 -> 3 -> 6
====================

========DFS========
Undirected Graph DFS result
startvertex: 1
1 -> 2 -> 4 -> 7 -> 5 -> 8 -> 9 -> 10 -> 3 -> 6
====================
```

5) KRUSKAL

MST를 구성하는 edge들의 vertex값을 오름차순으로 출력하고, weight들의 총합을 cost로 출력한 결과이다. cycle이 생기는 edge는 제외된 것을 확인할 수 있다.

```
======== KRUSKAL ========
[1] 3(2)
[2] 4(5)
[3] 1(2) 5(3)
[4] 7(3) 5(4) 2(5)
[5] 3(3) 4(4)
[6] 7(1)
[7] 6(1) 4(3) 8(12) 9(16)
[8] 7(12)
[9] 7(16) 10(20)
[10] 9(20)
cost: 66
=========================
```

6) DIJKSTRA

shortest path와 cost를 출력한 결과이다. 이때 최단 경로는 역추적한 순서대로 출력된다.

```
======== DIJKSTRA ========
Directed Graph Dijkstra result
startvertex: 1
[2] 1 -> 2 (6)
[3] 1 -> 3 (2)
[4] 1 -> 3 -> 5 -> 4 (9)
[5] 1 -> 3 -> 5 (5)
[6] 1 -> 3 -> 6 (10)
[7] 1 -> 3 -> 6 -> 7 (11)
[8] 1 -> 3 -> 6 -> 7 -> 8 (23)
[9] 1 -> 3 -> 6 -> 7 -> 8 -> 9 (46)
[10] 1 -> 3 -> 6 -> 7 -> 8 -> 9 -> 10 (66)

=========================

======== DIJKSTRA ========
Undirected Graph Dijkstra result
startvertex: 1
[2] 1 -> 2 (6)
[3] 1 -> 3 (2)
[4] 1 -> 3 -> 5 -> 4 (9)
[5] 1 -> 3 -> 5 (5)
[6] 1 -> 3 -> 6 (10)
[7] 1 -> 3 -> 6 -> 7 (11)
[8] 1 -> 3 -> 6 -> 7 -> 8 (23)
[9] 1 -> 3 -> 6 -> 7 -> 8 -> 9 (46)
[10] 1 -> 3 -> 6 -> 7 -> 8 -> 9 -> 10 (66)

=========================
```

7) BELLMANFORD

Bellman-Ford 알고리즘에 따라 시작점부터 도착점까지의 최단 경로를 출력한 결과이다. 총 cost도 같이 출력된다.

```
======== Bellman-Ford ========
Directed Graph Bellman-Ford result
1 -> 3 -> 6 -> 7 -> 8 -> 9 -> 10
cost : 66
==========================

======== Bellman-Ford ========
Undirected Graph Bellman-Ford result
1 -> 3 -> 6 -> 7 -> 8 -> 9 -> 10
cost : 66
==========================
```

8) FLOYD

시작점에서 도착점까지 가는 데 필요한 cost의 최솟값을 행렬 형태로 출력한 결과이다. 자기 자신으로의 cost는 0이며, 도달할 수 없는 정점에 대해서는 x를 출력한 것을 확인할 수 있다.

```
======== FLOYD ========
Directed Graph FLOYD result
    [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1]  0   6   2   9   5   10   11   23   46   66
[2]  x   0   x   5   18   x   8   20   43   63
[3]  x   7   0   7   3   8   9   21   44   64
[4]  x   x   x   0   13   x   3   15   38   58
[5]  x   x   x   4   0   x   7   19   42   62
[6]  x   x   x   15   11   0   1   13   36   56
[7]  x   x   x   14   10   x   0   12   35   55
[8]  x   x   x   53   49   x   39   0   23   43
[9]  x   x   x   30   26   x   16   28   0   20
[10]  x   x   x   44   40   x   30   42   65   0
==========================

======== FLOYD ========
Undirected Graph FLOYD result
    [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1]  0   6   2   9   5   10   11   23   46   66
[2]  x   0   x   5   18   x   8   20   43   63
[3]  x   7   0   7   3   8   9   21   44   64
[4]  x   x   x   0   13   x   3   15   38   58
[5]  x   x   x   4   0   x   7   19   42   62
[6]  x   x   x   15   11   0   1   13   36   56
[7]  x   x   x   14   10   x   0   12   35   55
[8]  x   x   x   53   49   x   39   0   23   43
[9]  x   x   x   30   26   x   16   28   0   20
[10]  x   x   x   44   40   x   30   42   65   0
==========================
```

9) KWANGWOON

해당 알고리즘은 구현을 하지 못하여 에러 코드가 출력된다.

```
========ERROR=======
500
====================
```

10) EXIT

프로그램 상의 메모리를 해제하며 프로그램이 종료된 결과이다.

```
========EXIT========
Success
====================
```

5. Consideration

LOAD 함수를 작성할 때 getline()을 쓰는 과정에서 여러 시행착오를 겪었었다. 입력객체로써 불러온 것이 아니라 단순히 getline만을 사용했을 때 파일을 정상적으로 읽어들이지 못하였다. 윈도우에서 사용했을 때는 괜찮았으나 리눅스로 옮길 때 발생한 일이었다. 이에 while문을 통하여 입력객체로 전체 파일을 읽어오는 식으로 수정하였더니 명령어를 정상적으로 받아올 수 있었다. 그리고 문자의 크기에 대해서도 윈도우와 리눅스의 차이점이 있다는 것을 깨달았다. list 그래프 파일을 읽어올 때, from과 to weight 라인을 구별하기 위해서 size()함수로 크기를 구분하였는데 윈도우에서는 공백을 제외하고 계산하였으나 리눅스에서는 뒤에 포함되어 있는 '₩n'도 같이 계산되는 것 같았다. 그래서 size도 한 칸씩 올리는 것으로 수정하였다.

크루스칼 알고리즘을 진행할 때도 리눅스에서는 segment fault 오류가 발생하였는데, 이는 parent 배열 선언에 문제가 있었었다. find, getParent, union같은 함수에서도 사용하기 위해 처음에는 전역 변수로써 parent 배열을 만들었다. 하지만 오류가 발생하고 확인하는 과정에서 전역 변수 사용이 문제가 있음을 깨달았다. 처음에 초기화를 해줘야 하는데 전역 변수로 선언하면 그래프의 크기를 모르니 끝까지 전부 초기화를 해줄 수 없었던 것이다. 따라서 크루스칼 함수 안에서 동적할당으로 만들고 다른 함수에다 배열을 인자로 같이 보내는 형식으로 수정하였더니 정상적으로 돌아갈 수 있었다.

방향, 무방향 그래프에 대한 차이를 구현하려 했는데 이는 제대로 완성하지 못하였다. 들어오는 간선과 나가는 간선을 모두 저장하는 형태를 생각했는데 잘 안 되었다. 출력 결과도 방향성을 고려한 형태로 출력되었다. 이는 map에 대해 더 공부를 해봐야 할 것 같다.