데이터구조실습

Project3

담당교수 : 이형근 교수님

학 번 : 2021202058

성 명 : 송채영

1. Introduction

이번 3 차 프로젝트에서는 그래프를 활용하여 그래프 연산 프로그램을 구현하는 것이 목표이다. 그래프 정보가 저장되어 있는 텍스트파일을 통해 그래프를 구현하고, 그래프의 특성에 따라 BFS, DFS, Kruskal, Dijkstra, Bellman-Ford, FLOYD 연산을 수행하도록 한다. 프로그램의 동작은 명령어 파일에서 요청하는 명령에 따라 각각의 기능을 수행하고, 그 결과를 출력 파일인, "log.txt"파일에 저장한다. 각 그래프 연산들은 'Graph Method' 헤더 파일에 일반 함수로 존재하며 그래프 형식, 즉 List 와 Matrix 에 상관없이 그래프 데이터를 입력 받으면 동일한 동작을 수행하도록 일반화하여 구현하도록 한다. 그래프 데이터는 방향성인 direction 과 가중치인(weight)를 가지고 있으며, 데이터의 형태에 따라 List 그래프와 Matrix 그래프로 나누어 저장한다. 프로그램은 그래프 정보가 저장되어 있는 텍스트파일, 즉 "graph_L.txt", "graph_M.txt"을 읽어 해당 정보를 Adjacency List 혹은 Adjacency Matrix 에 저장한다. 텍스트 파일의 첫 번째 줄에는 그래프의 형태가 저장되어 있고, 두 번째 줄에는 그래프의 크기가 저장되어 있다. 이후의 데이터는 그래프 형식에 따라 구분된다.

형식	내용
1	시작 vertex
2	[도착 vertex][weight]

위 표는 List 그래프 데이터의 형식이다.

형식	내용
3	[weight_1] [weight_2] [weight_3] … [weight_n]

위 표는 Matrix 그래프 데이터의 형식이다.

행렬의 행과 열은 각각 to vertex 와 from vertex 를 의미하며, 행렬의 값은 from vertex 와 to vertex 사이를 잇는 edge 의 weight 를 의미한다. 이때 그래프의 모든 vertex 의 값은 1 이상의 정수이며, 1, 2, 3, …의 순으로 정해진다고 가정한다.

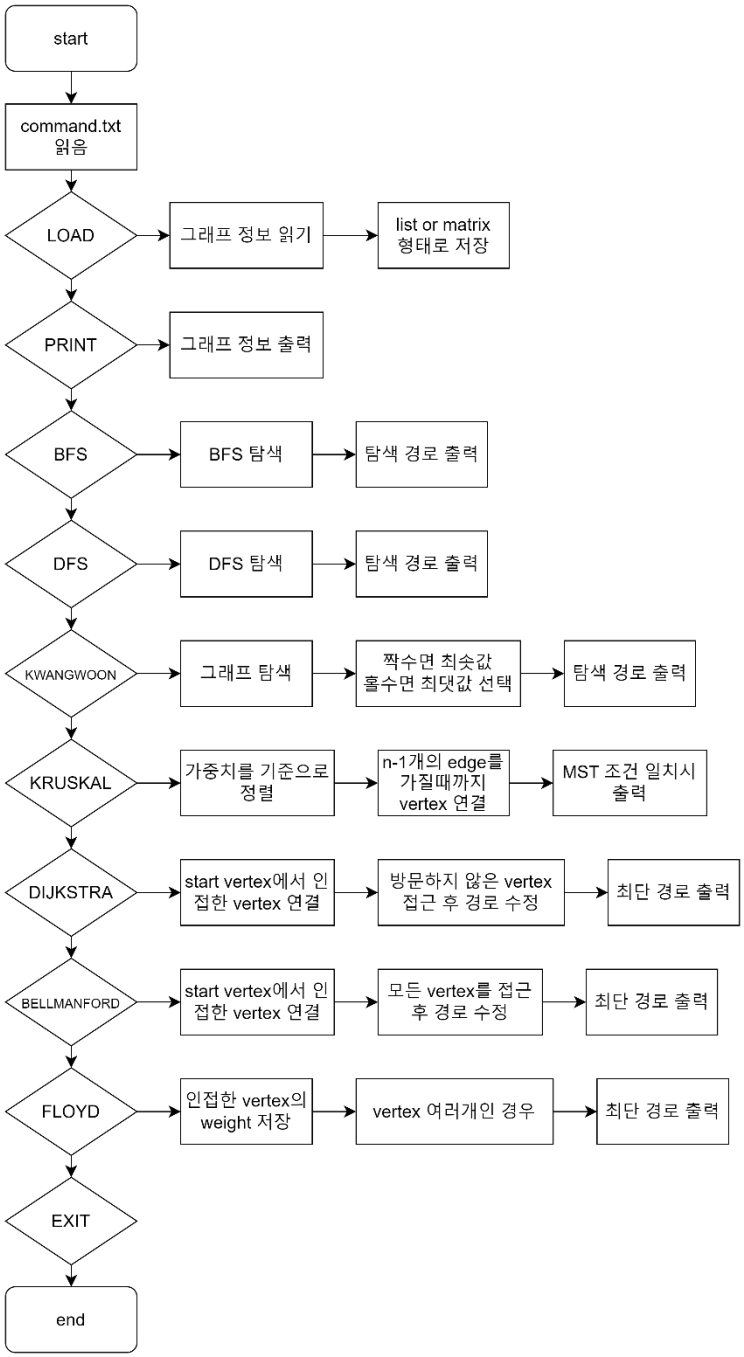
그래프 연산에 대해 간단하게 설명하겠다.

BFS 는 너비우선탐색으로 queue 를 활용하여 구현하고 DFS 는 깊이우선탐색으로 stack 을 활용하여 구현한다. KRUSKAL 알고리즘은 minimal spanning tree 를 구현하는 알고리즘이다. DIJKSTRA 알고리즘은 single source to all destination 으로 하나의 정점에서 모 든 정점까지의 최단경로를 구하는 알고리즘이다. BELLMANFORD 알고리즘은 시작 정점으로부터 도착 정점까지의 최단경로를 구 하는 알고리즘으로 weight 가 음수여도 최단경로를 구할 수 있으나 음수 사이클이 발생한 경우에는 최단경로를 구할 수 없다.

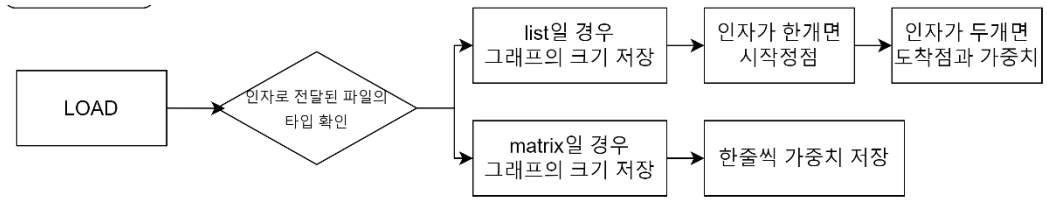
FLOYD 알고리즘은 모든 vertex 쌍간의 최단경로를 구하는 것이다. Weight 가 음수이더라도 최단경로를 구할 수 있으나 음수사이클이 발생한 경우에는 최단 경로를 구할 수 없다.

2. Flowchart

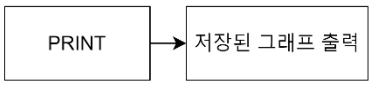
아래의 flowchart 사진은 명령어를 기반으로 한 동작들을 설명하였다.



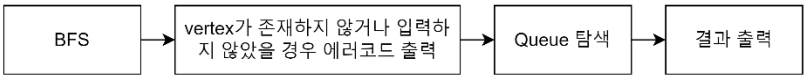
LOAD, PRINT, BFS, DFS, DFS_R, KRUSKAL, BELLMANCODE, FLOYD, EXIT 로 총 10 개의 명령어를 통해 프로젝트를 진행해 그래프를 이용한 그래프 연산 프로그램을 구현하였다. 전체적인 흐름은, 파일로부터 읽어온 명령어에 따른 함수를 실행시키고 성공 메시지 출력 또는 에러 메시지를 출력하며, 이것을 파일이 끝날 때까지 반복하였다. 이때 명령어를 제대로 수행하지 못했을 경우, 알맞은 에러코드를 출력하였다. 반대로 명령어를 제대 로 수행할 경우, 어떠한 작업이 진행되고 출력되는지 flowchart 에 명시하였다. 명령어 별로 나타낸 flow chart 이다. 먼저 Load 의 흐름도에 대해 자세히 살펴보면 다음 과 같다.



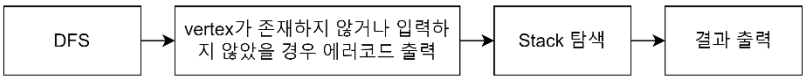
그래프의 형식이 list 인자 matrix 인지 확인한 후 strtok 함수를 사용해서 한 줄씩 정보를 저장해주었다. 먼저 list 일 경우 인자가 한 개면 시작 vertex, 인자가 두개면 첫 번째는 도착 vertex, 두번째는 가중치에 저장해주었다. Matrix 일 경우 for 문을 사용해서 한 줄씩 가중치를 저장하도록 구현하였다.



다음으로 PRINT 명령어의 흐름도이다. PRINT 명령어의 경우 저장된 그래프를 출력하도록 구현하였다.

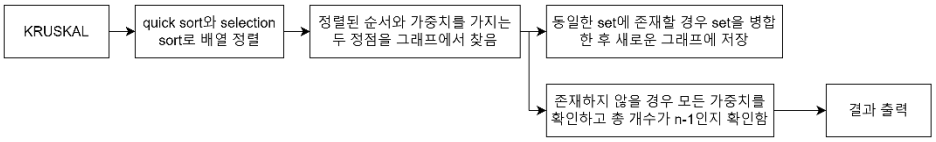


다음으로 BFS 명령어의 흐름도이다. BFS 는 queue 를 이용한 너비 탐색 방법이므로 인접한 노드들을 순차적으로 queue 에 삽입하도록 하였다. 또한 visited 를 이용하여 방문 표시를 해주었고 queue 에서 꺼낸 노드들을 접근하는 방식으로 구현하였다.

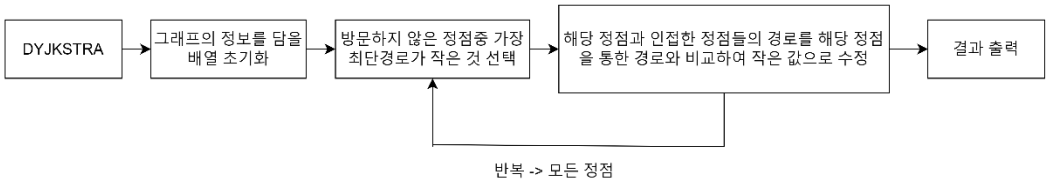


다음으로 DFS 명령어의 흐름도이다. DFS 는 stack 을 이용한 깊이 탐색 방법이므로 인접한 노드들을 순차적으로 stack 에 삽입하도록 하였다. 또한 visited 를 이용하여 방문 표시를 해주었고 stack 에서 꺼낸 노드들을 접근하는 방식으로 구현하였다.

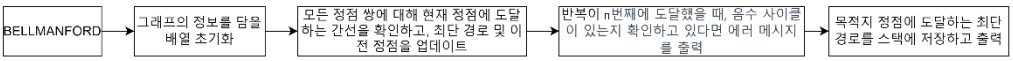
다음으로 KWANGWOON 명령어의 흐름도이다. 그래프를 순회하면서 방문한 정점들을 표시하였고 현재 방문할 수 있는 정점들이 홀수개면 가장 큰 정점 번호를 방문하도록, 현재 방문할 수 있는 정점들이 짝수개면 가장 작은 정점 번호를 방문하도록 구현하였다.



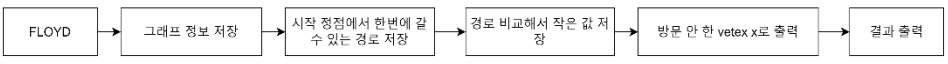
다음으로 KRUSKAL 명령어의 흐름도이다. Quicksort 함수와 selection sort 함수를 사용해 를 배열의 간선 가중치를 비내림차순으로 정렬한다. 이후 최소 신장 트리인 MST 를 구성하였다. MST 행렬은 for 문으로 반복하며 간선 가중치를 오름차순으로 탐색하며 정렬하였고 findset 과 unionset 함수를 사용해 disjoint-set 자료구조를 관리하며 간선을 MST 에 추가하도록 구현하였다. 이후 간선의 개수가 n-1 이 아닌 경우 연결되지 않은 그래프일 가능성이 있으므로 에러메시지를 출력해주었다.



다음으로 DIKSTRA 명령어의 흐름도이다. 그래프의 정보를 담을 배열을 초기화한 후 방문하지 않은 정점들 중 최단경로를 가진 정점을 선택한다. 이후 선택된 정점과 연결된 인접 정점들에 대한 최단 경로를 옵션에 따라 업데이트 하도록 구현하였다. 이때 음수 가중치가 있을 경우 negative weight 를 출력하도록 하였다.



다음으로 BELLMANFORD 명령어의 흐름도이다. 그래프의 정보를 담을 배열을 초기화하고 BELLMAN FORD 를 반복하며 수행하였다. 모든 정점 쌍에 대해 현재 정점에 도달하는 간선을 확인하고 최단 경로 및 이전 정점을 업데이트 하였으며 이를 반복하였다. n 번째 반복했을 때 음수 사이클이 있다면 에러 메시지를 출력하도록 하였다. 이후 목적지 정점에 도달하는 최단 경로를 스택에 저장하고 출력하도록 구현하였다.



마지막으로 FLOYD 명령어의 흐름도이다. 옵션에 따라 방향성을 고려하며 그래프의 입력한 간선 정보를 가져와 matrix 에 저장하였다. 이후 시작 정점에서 한 번에 갈 수 있는 경로를 저장하고 경로를 비교해서 더 짧은 경로로 업데이트 하였다. 방문 안 한 vertex 는 x 로 하여 결과를 출력하도록 구현하였다.

3. Algorithm

프로젝트에서 사용한 알고리즘의 동작을 설명하겠다.

BFS 의 경우 Breadth first search 로 너비 우선 탐색을 말한다. 현재 노드의 다음 레벨보다 인접한 노드들을 먼저 탐색하는 방법이다. GraphMethod.cpp 에서 BFS 의 코드를 짠 후 manager.cpp 의 mBFS 에서 실행하도록 한다. 우선 BFS 는 Queue 를 활용하여 동작을 수행하였다. Visit 을 통해 노드의 방문 여부를 알 수 있다. Queue 가 empty 가 될 때까지 while 문을 사용하여 코드를 구현하였다. 모든 연결된 vertex 에 대해서 vertex 의 순서대로 결과를 출력문에 맞게 출력해주었다. 이때 queue 가 empty 일 때와 아닐 때로 나누어 empty 면 vertex 를, empty 가 아니면 vertex 와 ->를 출력해주었다.

DFS 의 경우 Depth first search 로 깊이 우선 탐색을 말한다. 현재 노드와 인접한 노드들보다 현재 노드의 다음 노드를 먼저 탐색하는 방법이다. GraphMethod.cpp 에서 BFS 의 코드를 짠 후 manager.cpp 의 mDFS 에서 실행하도록 한다. 우선 DFS 는 Stack 을 활용하여 동작을 수행하였다. Visit 을 통해 노드의 방문 여부를 알 수 있다. DFS 와 마찬가지로 Stack 이 empty 가 될 때까지 while 문을 사용하여 코드를 구현하였다. 모든 연결된 vertex 에 대해서 vertex 의 순서대로 결과를 출력문에 맞게 출력해주었다. DFS 역시 출력할 때 stack 이 empty 일 때와 empty 가 아닐 때로 나누어 empty 면 vertex 를, empty 가 아니면 vertex 와 ->를 출력해주었다.

KWANGWOON 의 경우 원래는 세그먼트 트리를 구현한 후 그 트리를 활용하여 그래프를 구현하여야 하지만, 트리를 구현하지 않고도 알고리즘이 올바르게 동작할 수 있기 때문에 트리를 구현하지 않고 동작하게끔 하였다. KWANGWOON 알고리즘은 그래프의 정점을 탐색하면서 특정 규칙에 따라 이동한다. 특정 규칙은 정점은 항상 1 부터 시작하며 정점 1 부터 탐색의 순서를 출력한다. 현재 연결되어 있는 간선의 수가 짝수라면 가장 작은 정점 번호로 방문을 시작하고, 현재 연결되어 있는 간선의 수가 홀수라면 가장 큰 정점의 번호로 방문한다. 현재 정점에서 다른 정점으로 이동할 수 있는 이동 여부는 다른 알고리즘에서 구현한 방법처럼 배열을 활용해서 나타내었다.

KRUSKAL 의 경우 greedy method 를 이용하여 모든 경로를 그때 그때 최소비용으로 연결하는 알고리즘을 말한다. 그래프에서 서로 다른 두 정점간 정점을 사용하지 않은

경로의 가중치를 오름차순으로 정렬한 후 정렬된 경로들을 활용하여 사이클이 만들어지지 않게끔 하였고 모든 가중치를 검사하고 난 후 n-1 개의 간선이 만들어질 경우 MST 가 생성됨을 의미한다.

DIJKSTRA 의 경우 어떠한 정점을 기준으로 다른 정점까지의 최단 경로를 구하는 알고리즘을 말한다. 시작 정점에서부터 자신을 제외한 나머지 정점까지의 최단거리를 구하는데, 현재 정점에서 인접한 정점 중 방문하지 않는 정점을 구분하고 이 중 가장 최단 경로를 가진 정점부터 방문해서 최단경로를 업데이트 해야 하며 이때 가중치가 음수일 경우 사용이 불가능하다는 특징이 있다.

BELLMANFORD 의 경우 시작 정점으로부터 모든 정점까지의 경로를 구하는 알고리즘을 말한다. DIJKSTRA 알고리즘과 유사하지만 다른 점이 있다면 BELLMAN FORD 알고리즘은 음수 가중치를 가지더라도 적용이 가능하다는 점이다. 또한 최단경로를 갱신했을 때 모든 정점을 방문하기 때문에 방문 여부를 알지 않아도 된다.

FLOYD 는 k 이하의 정점을 사용해서 i 부터 j 까지 가는 경로의 최단거리는 k 1 이하의 vertex 를 사용하여 i 부터 k 까지 가는 경우와 k 부터 j 까지 가는 경우의 합과 k-1 이하의 vertex 를 사용하여 i 부터 j 까지 가는 경로 중 더 작은 값이다. 따라서 그래프의 정보를 담은 곳인, 코드에선 martirx[n][n]을 초기화 해준 후 시작 vertex 로부터 다른 vertex 를 거치지 않고 갈 수 있는 경로를 저장해주었다. 모든 vertex 에 대해서 위에 설명한 부분을 실행하고 출력문에 맞춰 출력해주었다. 이때 방문하지 않은 vertex 의 경우 'x'를 출력해주었다.

해당 그래프를 구현하면서 사용하였던 추가적인 함수에 대해서 설명하겠다.

Quicksort 함수는 divide and conquer 알고리즘으로 특정 pivot 을 기준으로 배열을 두 부분으로 분할하고 각 부분을 재귀적으로 정렬한다. Pivot 을 선택하고 배열을 분할하는 과정을 반복하며 최종적으로 정렬된 배열을 얻게 된다.

Insertion sort 함수는 각 요소를 이미 정렬된 부분과 비교하여 적절한 위치에 삽입하는 정렬 알고리즘이다. 배열의 각 요소를 하나씩 순회하며 해당 요소를 이미 정렬된 부분에 삽입한다.

Insertion 함수는 insertion sort 에서 사용되는 보조함수로, 특정 요소를 이미 정렬된 부분에 삽입하는 역할을 하며, 정렬된 부분에서 적절한 위치를 찾아 요소를 삽입한다.

Unionset 함수는 여러 집합들을 효율적으로 관리하는 자료구조로 unoinset 함수는 두 집합을 합치는 연산을 수행한다.

Findset 함수는 union find 자료구조에서 사용되는 함수로 특정 원소가 속한 집합의 대표 원소(루트)를 찾아 반환하도록 한다.

4. Result Screen

command.txt

```
1   LOAD graph_M.txt
2   PRINT
3   LOAD graph_L.txt
4   PRINT
5   BFS Y 1
6   DFS Y 1
7   KWANGWOON 1
8   KRUSKAL
9   DIJSKSTRA Y 5
10  BELLMANFORD Y 1 3
11  FLOYD Y
12  EXIT
```

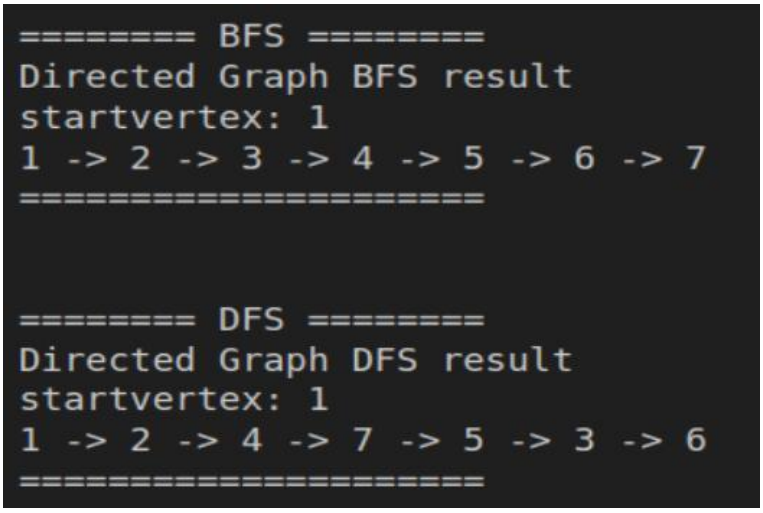
Command.txt 파일이다.

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


Graph is MatrixGraph!
========================== PRINT ==========================
    [1] [2] [3] [4] [5] [6] [7]
[1] 0   6   2   0   0   0   0
[2] 0   0   0   5   0   0   0
[3] 0   7   0   0   3   8   0
[4] 0   0   0   0   0   0   3
[5] 0   0   0   4   0   0   0
[6] 0   0   0   0   0   0   1
[7] 0   0   0   0   10  0   0
===========================================================
```

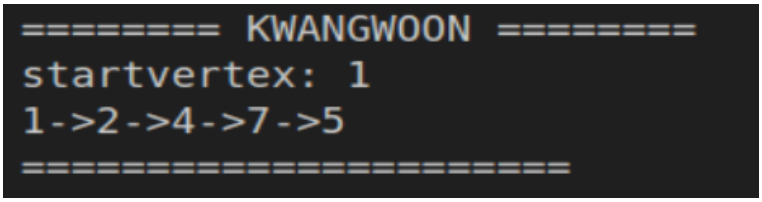
LOAD 가 성공적으로 된 것을 알 수 있으며 list 정보가 담긴 그래프 형식에 맞게 matrix 형식으로 PRINT 하는 것을 확인할 수 있다.

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


Graph is ListGraph!
======== 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)
======================
```

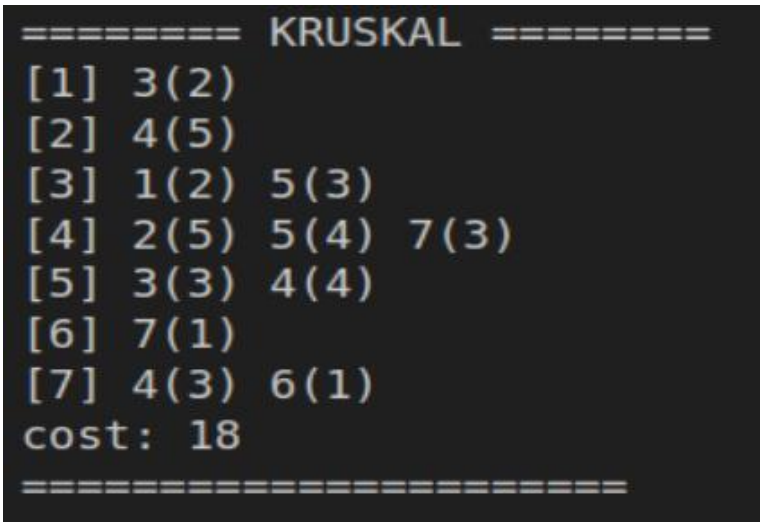
LOAD 가 성공적으로 되었으며 list 형식으로 PRINT 도 잘 되는 것을 확인할 수 있다. 이때 기존에 그래프 정보가 존재한 상태에서 LOAD 한 경우를 테스트한 것으로 기존 그래프 정보는 삭제되고 새로 생성된 그래프가 출력되는 것을 확인할 수 있다.



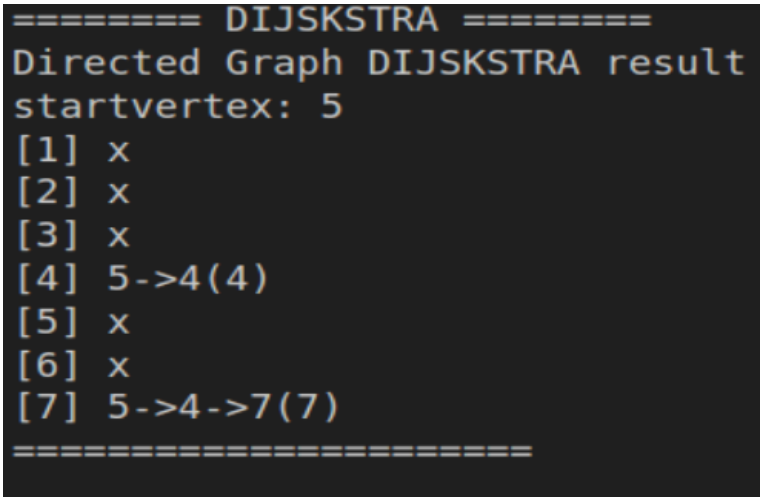
방향성이 있는 경우의 BFS, DFS 의 결과로 직접 stack 과 queue 에 넣어보았을 때 올바르게 작동하는 것을 확인하였다.



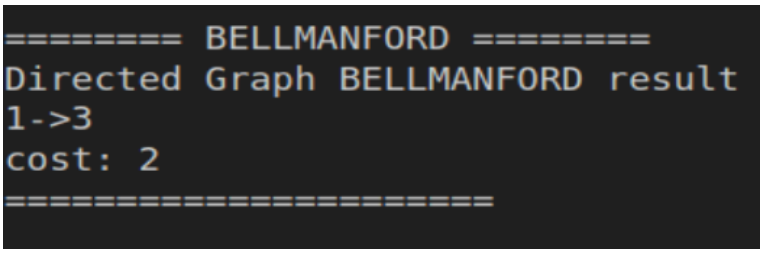
KWANGWOON 의 결과로 방향성이 있을 때 결과를 잘 출력하는 것을 확인할 수 있다.



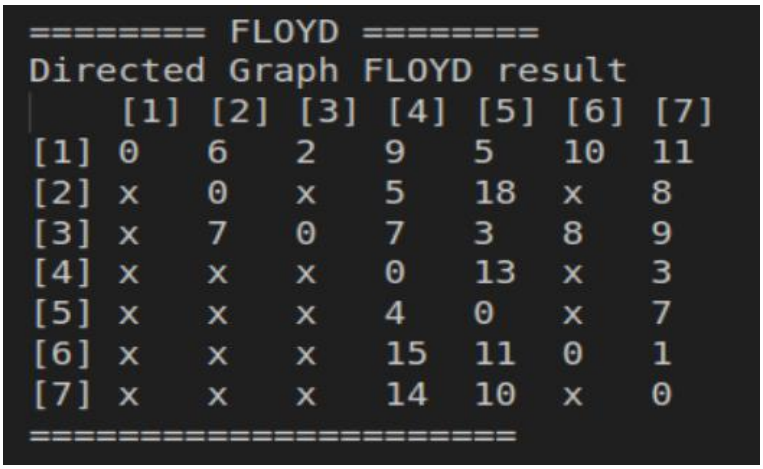
KRUSKAL 의 결과로 방향성이 있을 때 MST 를 구성하는 edge 들의 weight 값을 오름차순으로 잘 출력하였고 종합인 cost 도 알맞게 출력하는 것을 확인할 수 있다.



DIJSKSTR 의 결과로 도달할 수 없는 vertex 의 경우 x 를 출력하고 있으며 방향성이 있을 때의 shortest path 를 잘 출력하는 것을 확인할 수 있다.

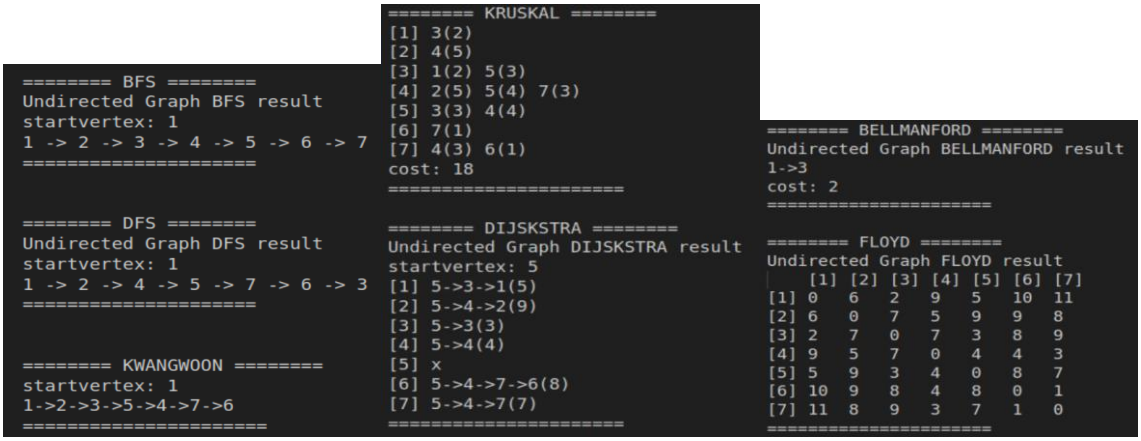


BELLMANFORD 의 결과로 방향성이 있을 때의 시작 정점에서 끝 정점까지의 최단경로와 cost 를 잘 출력하는 것을 확인할 수 있다.



FLOYD 의 결과로 방향성이 있을 때 시작 정점에서 끝 정점으로 가는데 필요한 비용의 최솟값을 잘 출력하는 것을 확인할 수 있다.

밑의 결과화면은 방향성이 없을 때를 출력해본 결과이다.



다음으로 10 개의 노드를 만들어서 결과를 확인해본 부분이다.

graph_L.txt

```
L
10
1
9 4
5 12
2
5 10
3
1 12
4
6 5
5
2 4
4 8
6
8 10
9 11
7
2 6
6 10
8
1 11
9
3 11
5 9
10
7 12
```

Graph_L.txt 부분이다.

실행한 log 결과화면은 아래와 같다.

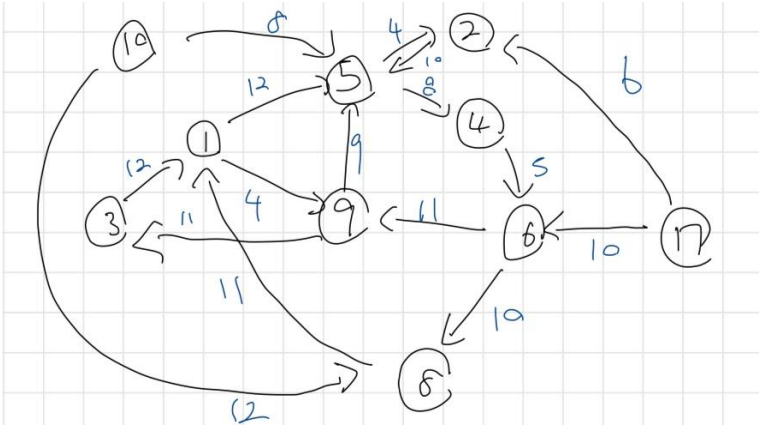
```
======== LOAD ========
Success
======================


Graph is ListGraph!
======== PRINT ========
[1] -> (5,12) -> (9,4)
[2] -> (5,10)
[3] -> (1,12)
[4] -> (6,5)
[5] -> (2,4) -> (4,8)
[6] -> (8,10) -> (9,11)
[7] -> (2,6) -> (6,10)
[8] -> (1,11)
[9] -> (3,11) -> (5,9)
[10] -> (5,8) -> (7,12)
======================
```

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


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


======== KWANGWOON ========
startvertex: 1
1->3->9->5->10->7->2
======================
```

```
======== KRUSKAL ========
[1] 9(4)
[2] 5(4) 7(6)
[3] 9(11)
[4] 5(8) 6(5)
[5] 2(4) 4(8) 9(9) 10(8)
[6] 4(5) 8(10)
[7] 2(6)
[8] 6(10)
[9] 1(4) 3(11) 5(9)
[10] 5(8)
cost: 65
======================
```

```
======== DIJSKSTRA ========
Undirected Graph DIJSKSTRA result
startvertex: 5
[1] 5->1(12)
[2] 5->2(4)
[3] 5->9->3(20)
[4] 5->4(8)
[5] x
[6] 5->4->6(13)
[7] 5->2->7(10)
[8] 5->1->8(23)
[9] 5->9(9)
[10] 5->10(8)
======================
```

```
======== BELLMANFORD ========
Undirected Graph BELLMANFORD result
1->3
cost: 12
======================


======== FLOYD ========
Undirected Graph FLOYD result
     [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1]  0   16  12  20  12  15  22  11  4   20
[2]  22  0   30  18  10  16  6   26  19  18
[3]  12  24  0   27  20  22  30  23  11  28
[4]  20  12  27  0   8   5   15  15  16  16
[5]  12  4   20  8   0   13  10  23  9   8
[6]  15  16  22  5   13  0   10  10  11  21
[7]  25  6   32  15  16  10  0   20  21  12
[8]  11  26  23  15  23  10  20  0   15  31
[9]  4   13  11  16  9   11  19  15  0   17
[10]     20  12  28  16  8   21  12  31  17  0
======================
```



직접 그림을 그려 확인해봤을 때 결과가 잘 출력되는 것을 확인하였다.

5. Consideration

이번 한 학기 동안 프로젝트를 진행하면서 작년보다 많이 성장했음을 느낄 수 있었다. 작년에는 1 차 반, 3 차 그리고 2 차는 구현하지도 못했지만 그래도 이번 학기는 세 과제 모두 구현해서 냈다는 점이 뿌듯했다. 이번 프로젝트를 진행하면서 작년 과제에서 방향성이 추가되었고 vertex 가 1 로 바뀌는 것 이외에는 별다른 점이 없어 복습하는 느낌으로 과제를 진행할 수 있었다. 특히 KWANGWOON 알고리즘은 작년과 다르게 새로운 알고리즘이었는데 세그먼트 트리를 이용해 현재 정점에서 다른 정점으로 이동할 수 있는 여부를 판단할 수 있지만, 알고리즘 구현 조건에 추가되어 있지 않았고 다른 알고리즘에서 구현했던 것처럼 visited 배열을 사용해 정점 여부를 판단하고 BSF 알고리즘처럼 구현하면 될 것 같았기에 새롭게 현재 정점에서 연결되어 있는 간선의 수가 짝수거나 홀수일 경우 작거나 큰 정점으로 이동하는 조건을 추가해서 구현해주었다. 처음에는 이 알고리즘이 구현하기 가장 어렵고 까다로울 것이라고 생각했지만 생각보다 KRUSKAL, BELLMANFORD, DIJSKSTRA 알고리즘을 구현하는 것이 가장 어려웠다. 하지만 데이터구조설계 ppt 를 참고하며 ppt 에 나와있는 코드를 그대로 구현하여 이번 프로젝트를 마무리할 수 있었다. 또한 다른 과목인 알고리즘에서 배웠던 내용을 기반으로 하여 자료구조를 배우니 훨씬 이해하기 쉬웠던 것 같다. 또한 음수 사이클 부분이 조금 이해하기 어려웠고 음수사이클의 경우 해당 경우에 어떤식으로 출력되는지 보고싶었지만, 실험해보지 못해 아쉬웠다.