Date Structure

3차 프로젝트

과목: 데이터 구조 설계

교수: 이기훈 교수님

실습 분반: 수요일

학번: 2020202034

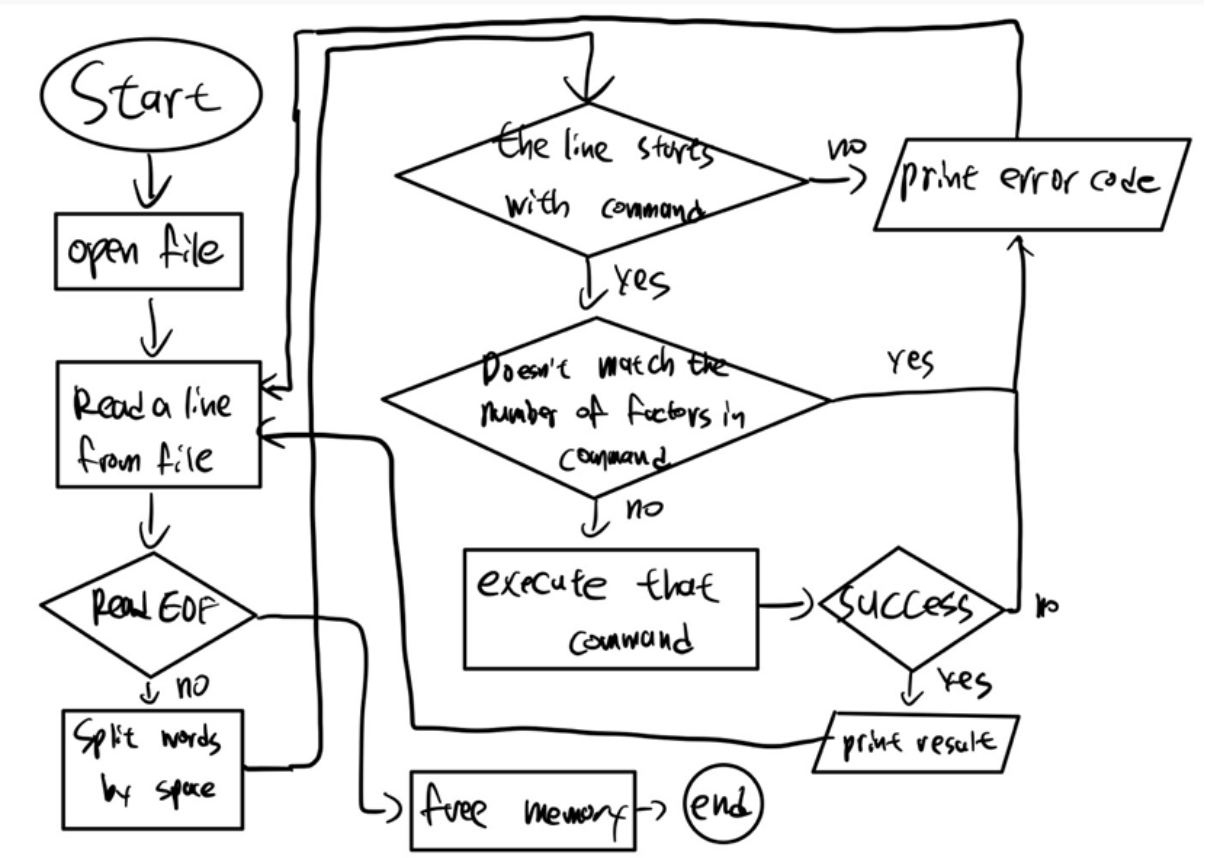
이름: 김태완

1. Introduction

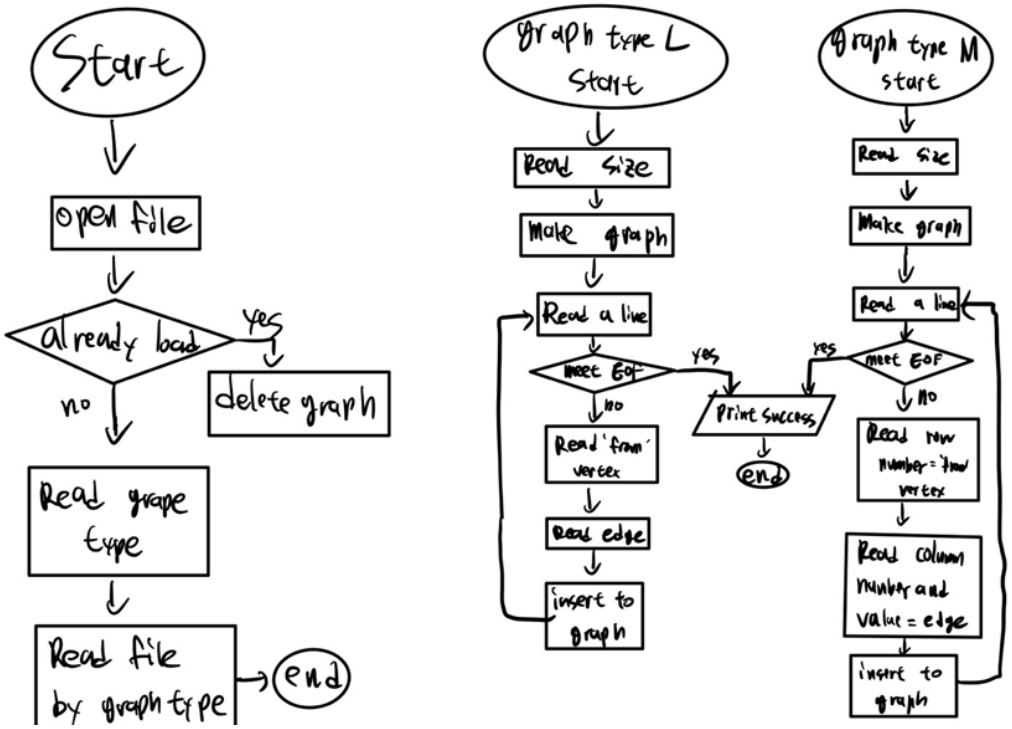
   A. 이번 3차 프로젝트는 그래프 탐색 알고리즘을 구현해보는 내용입니다. 먼저 인접 리스트 형태 또는 인접 행렬 형태의 그래프가 주어지면 이를 저장해 명령어로 주어지는 탐색 알고리즘에 따라 그 결과를 출력합니다. 수업시간에 배운 BFS, DFS, Dijkstra, Bellman-Ford, Floyd, Kruskal 알고리즘, 그리고 segment tree를 이용하는, 이번 과제를 위해 만들어진 Kwangwoon 알고리즘을 직접 구현해보는 과제입니다. 그래프 구현 알고리즘마다 탐색에 필요한 요소(예: 방향성 여부, 시작점, 종료점 등)가 다르기 때문에 명령어를 더 꼼꼼히 파싱해야 합니다.

2. Flowchart

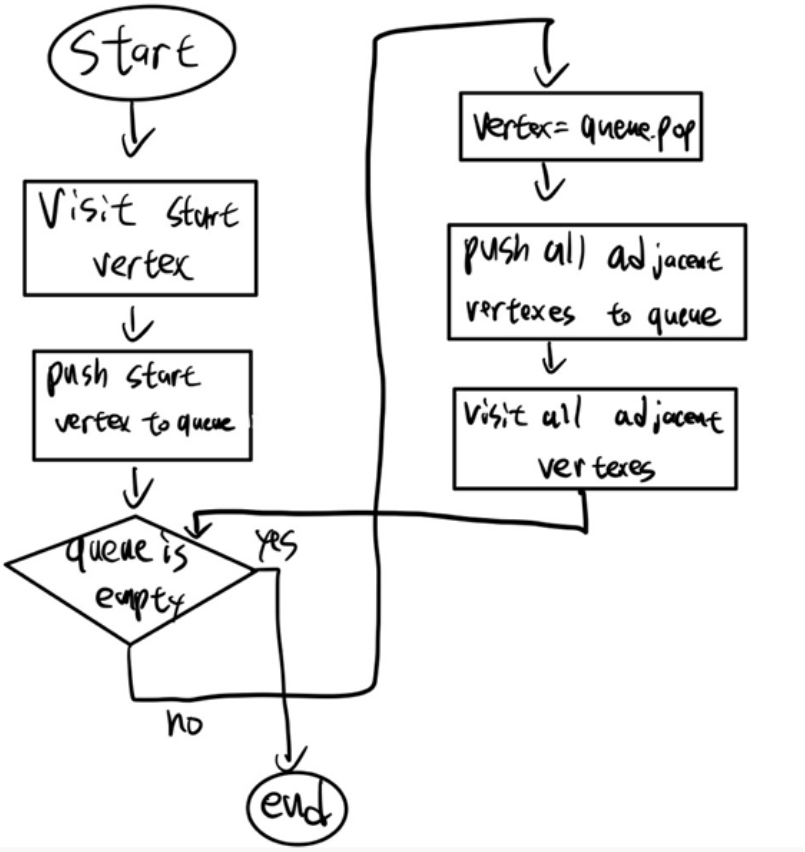
   A. RUN(main)



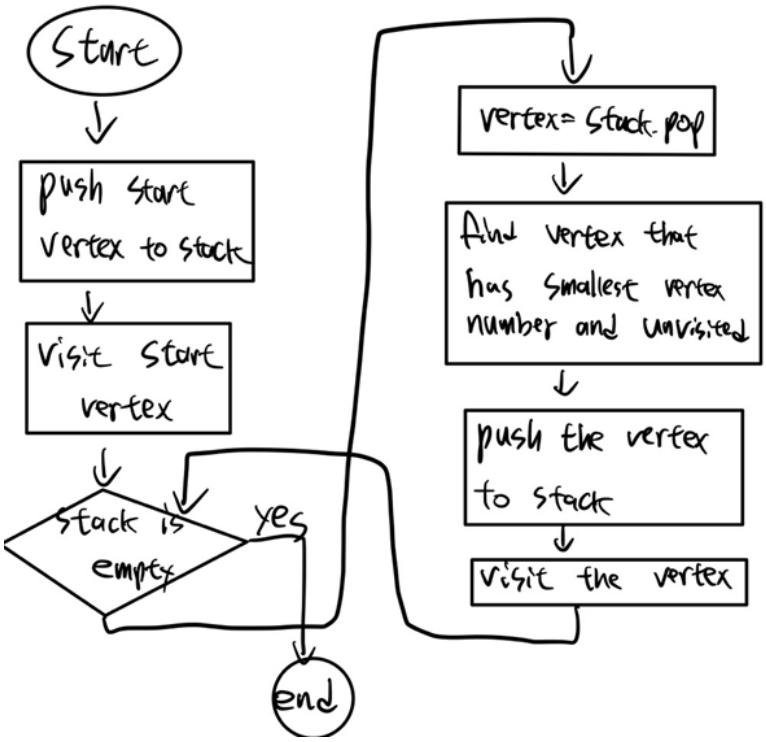
B. LOAD


Start
open file
already load
yes
delete graph
no
Read grape type
Read file by graph type
end
graph type L start
Read size
Make graph
Read a line
meet EOF
yes
no
Read 'from' vertex
Read edge
insert to graph
graph type M start
Read size
Make graph
Read a line
meet EOF
yes
no
Read row number = 'from' vertex
Read column number and value = edge
insert to graph
Print success
end

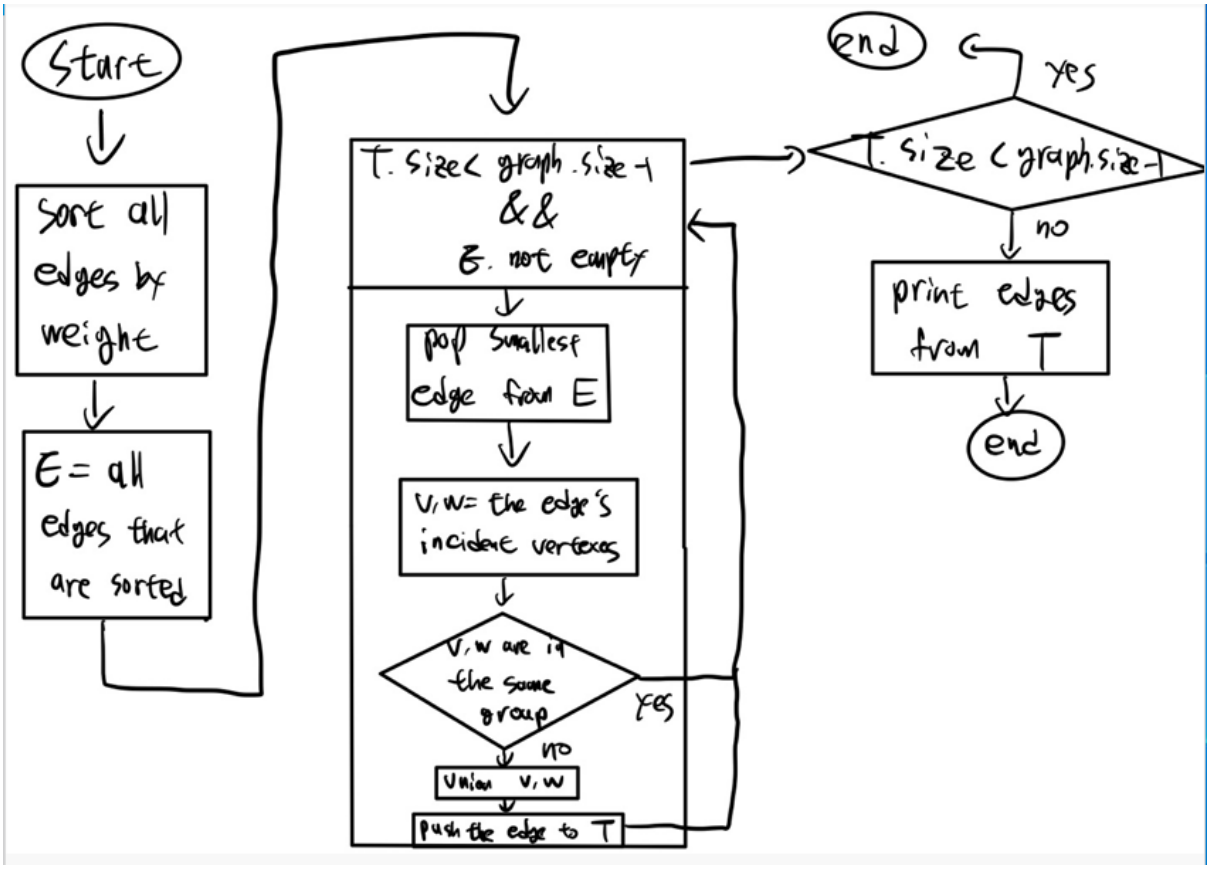
C. BFS


Start
Visit start vertex
push start vertex to queue
queue is empty
yes
no
end
Vertex = queue.pop
push all adjacent vertexes to queue
visit all adjacent vertexes

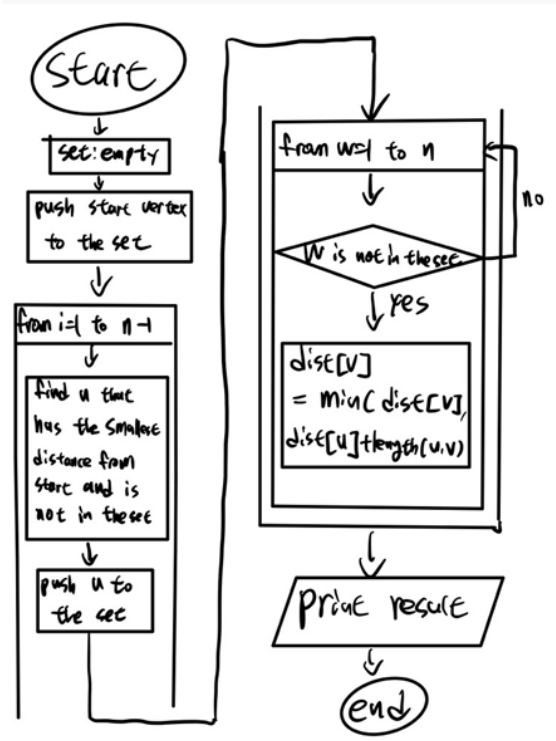
D. DFS



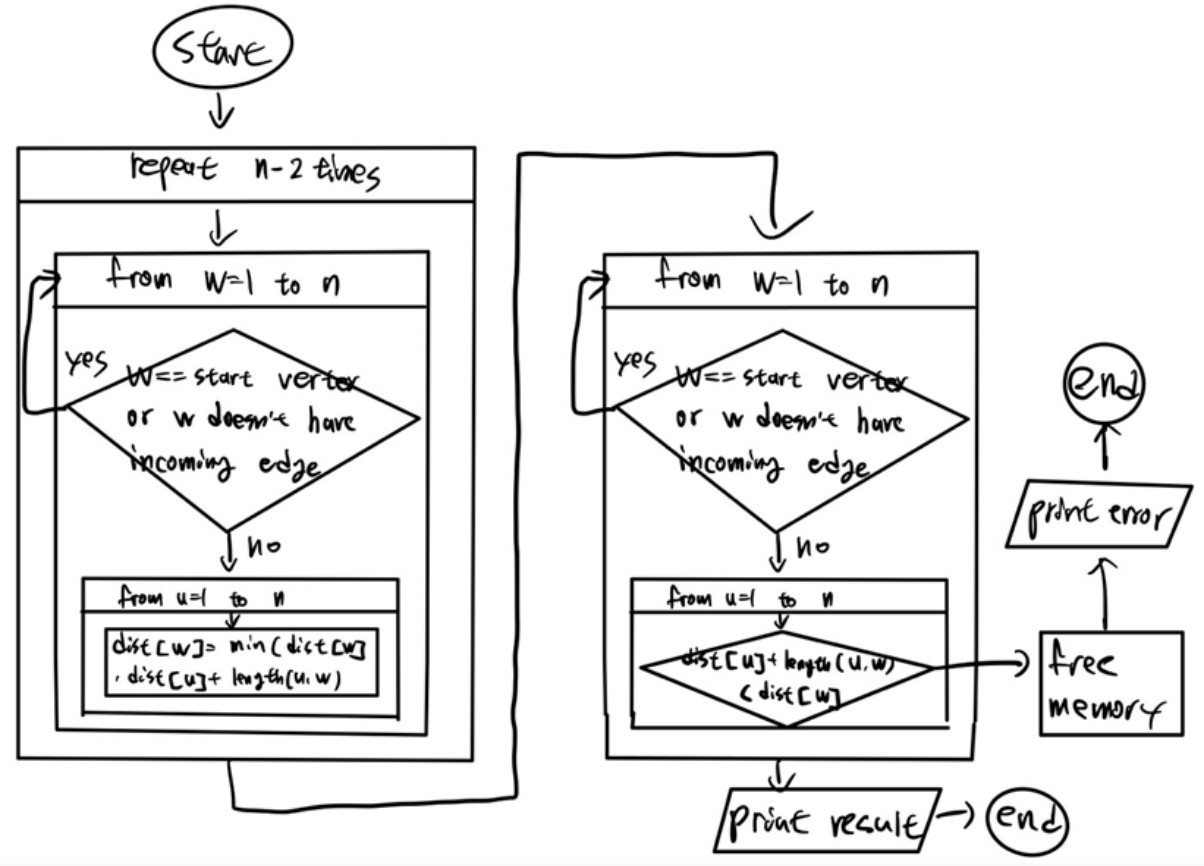
E. KRUSKAL



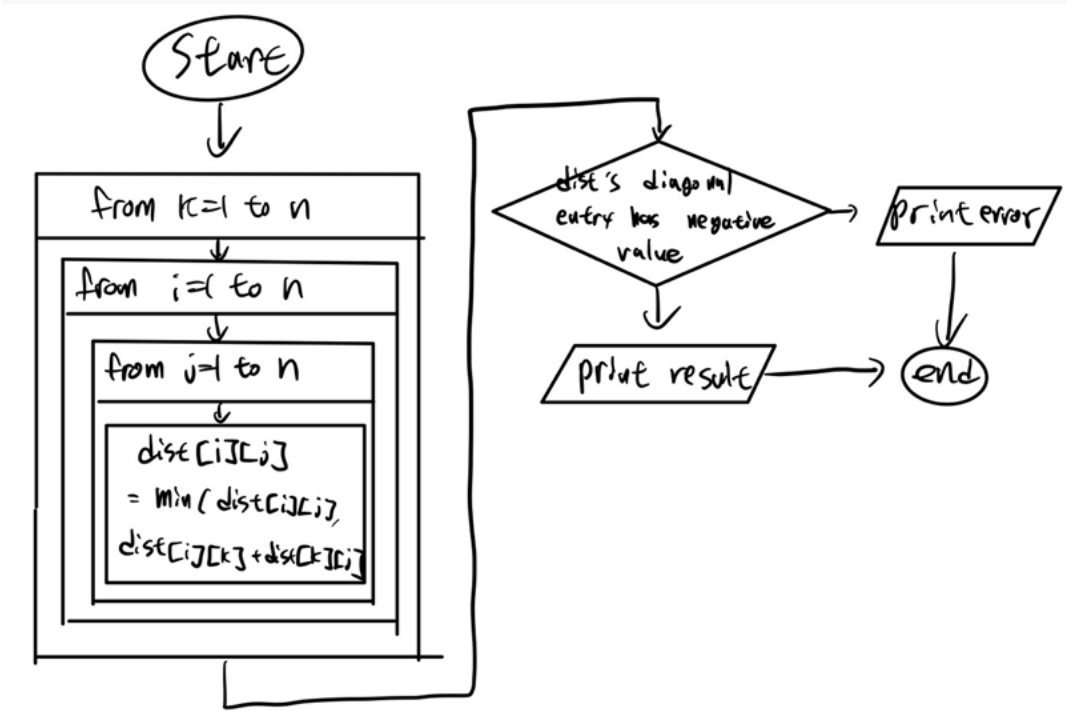
F. DIJKSTRA



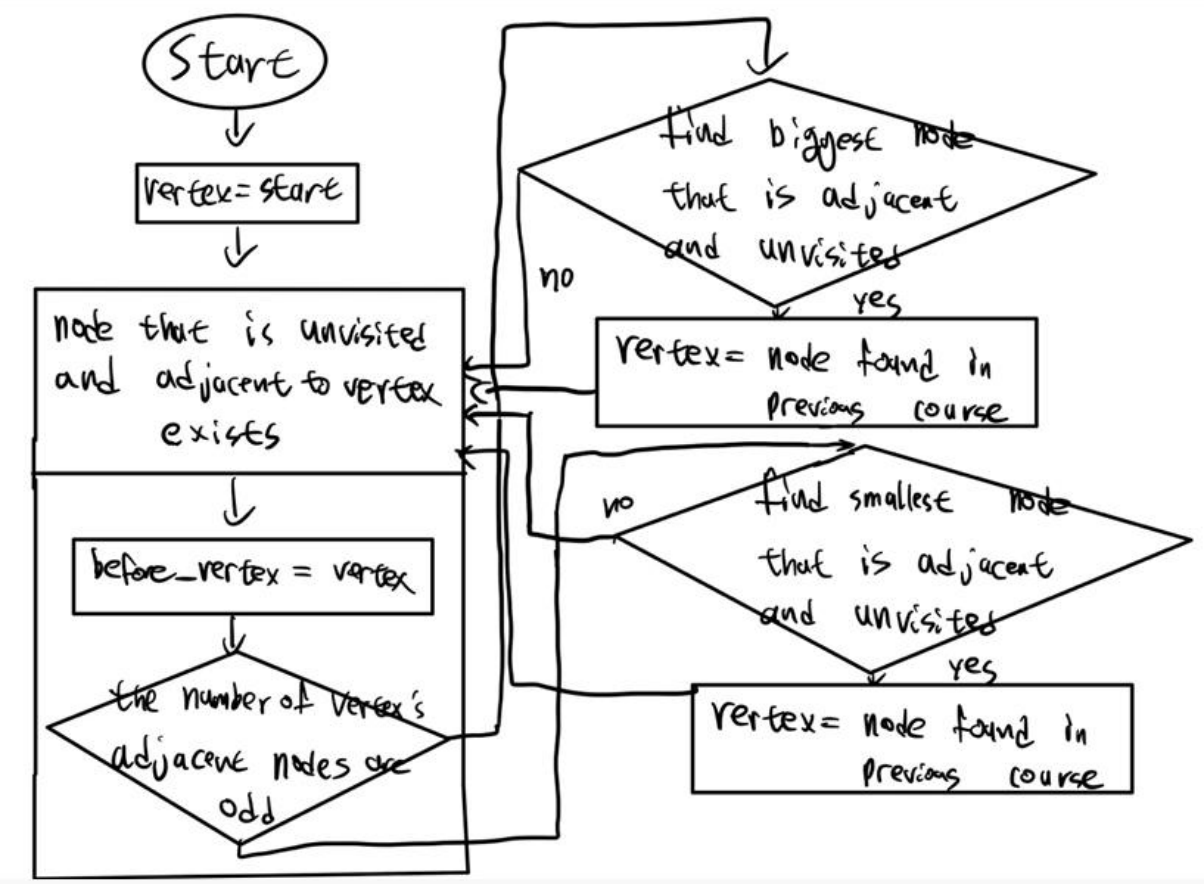
G. BELLMANFORD

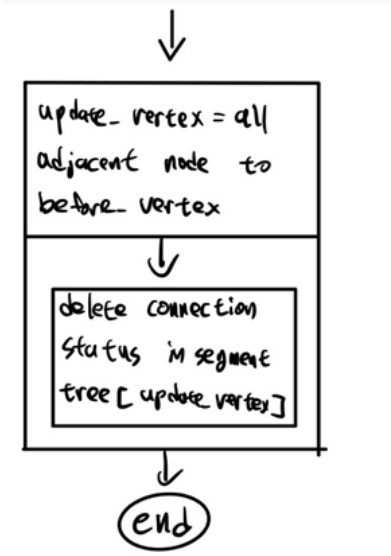


H. FLOYD



I. KwangWoon





3. Algorithm

A. BFS & DFS

i. BFS와 DFS는 시작 노드부터 방문 가능한 모든 지점을 방문할 수 있는 탐색 방식들입니다.

ii. BFS는 Breadth-First-Search(너비 우선 탐색)로, 현재 노드에서 다음에 방문 가능한 노드들을 순차적으로 방문하는 탐색 방식입니다. 기본적으로 Queue를 사용합니다.

iii. BFS의 탐색 방식:

1. 시작 노드 방문 및 큐에 삽입
2. 큐가 빌 때까지 아래 과정 반복
3. 큐에서 노드 하나 추출
4. 해당 노드에서 방문 가능한 노드 모두 큐에 삽입 & 방문

iv. DFS는 Depth-First-Search(깊이 우선 탐색)로, 현재 노드에서 방문 가능한 노드가 있으면 그 중 하나를 골라 방문을 반복 후, 방문 가능한 노드가 없으면 이전 노드로 돌아와 다시 방문 가능한 노드가 있는 지를 검사하며 탐색하는 방식입니다. 기본적으로 이전에 방문했던 노드로 다시 돌아올 수 있어야 하기 때문에 Stack을 사용합니다.

v. DFS의 탐색 방식:

1. 시작 노드 방문 및 스택에 삽입

2. 스택이 빌 때까지 아래 과정 반복
3. 스택에서 노드 하나 추출
4. 해당 노드에서 방문 가능한 노드가 있다면 다시 스택에 삽입
5. 위 노드에서 방문 가능한 노드도 방문 후 스택에 삽입

B. Insertion sort & Quick sort

i. 삽입 정렬은, 정렬이 되어 있지 않은 부분의 데이터를 하나 추출해 정렬이 된 부분의 적절한 위치에 추가하는 과정을 반복하는 정렬 방식입니다.

ii. 퀵 정렬은, 데이터를 pivot을 기준으로 둘로 나누고 나눈 데이터를 다시 pivot으로 나누어 정렬하기에 충분히 작은 수가 될 때까지 위 과정을 반복하는 정렬 방식입니다.

iii. 삽입 정렬은 평균, 최악의 경우에 O(n^2)의 시간 복잡도를 가지지만, 퀵 정렬은 데이터가 정렬되어 있을 때를 최악의 경우로 가지는데, 이때의 시간 복잡도는 O(n^2)이지만, 최선, 평균의 경우에는 O(nlogn)의 시간 복잡도를 가집니다.

iv. 퀵 정렬은 pivot을 어떻게 정하느냐에 따라 최악의 경우를 피할 수 있는데, 쪼갤 데이터의 왼쪽, 중간, 오른쪽 값을 뽑아 그 중 가장 가운데 값으로 pivot을 정하면 최악의 경우를 평균의 경우로 바꿀 수 있습니다.

C. Union & Find

i. Union과 Find는 Kruskal 알고리즘을 사용할 때, 추가된 edge가 사이클을 생성하는지 판별하기 위해 구현한 함수입니다.

ii. 시간 복잡도를 고려하지 않은 union함수의 시간 복잡도는 O(1)이고, find함수의 시간 복잡도는 O(h) (h는 group tree의 높이)입니다.

iii. Union함수는 한 그룹 트리의 루트 노드를 다른 그룹 트리의 루트 노드의 자식으로 연결해 구현할 수 있고, Find 함수는 루트 노드를 찾을 때까지 부모 노드를 따라 올라가 루트 노드를 반환하는 방식으로 구현할 수 있습니다.

D. Kruskal

i. 위에서 구현한 삽입 정렬, 퀵 정렬, Union, Find 함수를 모두 사용해 구현하였습니다.

ii. Kruskal은 Minimum Spanning Tree를 만드는 알고리즘으로, edge를 weight

순으로 정렬해 하나씩 선택하면서 n(노드의 개수) - 1개가 선택될 때까지 반복하는 방식입니다. 이때, 추가한 edge가 cycle을 만든다면 선택하지 않아야 하며, edge를 모두 사용했는데도 선택된 edge의 개수가 n - 1개가 되지 않는다면 이는 MST가 존재하지 않는 그래프입니다.

iii. Kruskal의 수도 코드는 다음과 같습니다.

1. 그래프의 edge를 모두 E에 추가
2. E를 quick sort로 정렬
3. T에 추가된 edge가 n – 1개가 되거나 E가 비었을 때까지 아래 과정 반복
4. E에서 가중치가 가장 낮은 edge 추출
5. Edge의 두 vertex가 같은 그룹이 아니면(이때, Find함수로 두 vertex의 root를 비교)
6. 두 vertex를 union
7. T에 edge 추가
8. 3.에서 반복문 탈출 후 T에 n – 1개 미만의 edge가 있으면 MST 생성 실패
9. 아니면 성공

E. Dijkstra

i. Dijkstra는 최단 경로를 Greedy하게 찾는 알고리즘입니다. 최단 경로를 찾았음을 저장하는 set에 시작점부터 최단 경로를 찾은 노드를 추가하면서, 해당 set과 가장 인접한 노드의 거리를 수정하는 방식으로 최단 경로를 업데이트합니다.

ii. Dijkstra는 greedy한 탐색방식 때문에 시간복잡도 측면에서 나쁘지 않지만, 음수 가중치가 포함된 그래프에서 제대로 동작하지 않는다는 단점이 존재합니다.

iii. Dijkstra의 수도 코드

1. S에 시작점 추가
2. 3. - 5. N – 1번 반복
3. S에 포함되어 있지 않으면서 시작점과의 거리가 제일 작은 노드(u)

찾기

4. U를 S에 추가

5. 모든 노드에서 기존 시작점과의 거리가 시작점에서 u를 거쳐 각 노드로 들어오는 거리보다 길다면 거리를 업데이트

F. Bellman-Ford & Floyd

i. Bellman-Ford와 Floyd는 둘 다 최단 거리를 탐색하지만 greedy하게 탐색하지 않아 음수 가중치를 고려할 수 있는 알고리즘입니다.

ii. Bellman-Ford는 시작 점에서의 최단 거리를 찾을 때 사용 가능한 edge를 0 ~ n – 1개까지 사용 가능하게 하며 최단 거리를 업데이트하는 방식입니다. 음수 사이클이 있을 때는 최단 거리를 찾을 수 없으며, 해당 알고리즘을 이용해 음수 사이클 여부를 찾을 수 있습니다.

iii. 다음은 Bellman-Ford의 수도코드입니다.

1. 시작 노드와 연결된 노드들은 그의 가중치 값으로, 나머지 노드는 0으로 dist 배열 초기화

2. 3.-5. n – 2번 반복

3. For j = 1 to n

4. J가 시작점이 아니고, incoming edge가 있을 때 다음 과정 실행

5. 모든 노드를 반복하며, 노드 번호가 k일 때 dist[j] = min(dist[j], dist[k] + length(k, j) 실행

6. 3. – 5. 과정 한 번 더 반복해 dist값이 업데이트되는 경우가 있다면 이는 음수 사이클이 있는 것이므로 에러 출력

iv. Floyd는 시간 복잡도가 O(n^3)으로 매우 크지만, 이 시간 복잡도로 그래프 상의 모든 지점 간의 최단 경로를 찾아준다는 장점이 있습니다.

v. Bellman-Ford와 유사하게, 한 지점에서 다른 지점으로 최단 경로를 찾을 때, 중간에 거쳐갈 노드를 0개 ~ n – 2 개까지 사용하여 최단 경로를 업데이트하는 방식입니다.

vi. 다음은 Floyd의 수도코드입니다.

1. Dist[v][w] <- v, w와 인접한 edge가 있으면 그의 weight로 초기화, 없으면 inf로 초기화

2. 노드의 개수만큼 3. – 5. 반복
3. For I = 1 to n
4. For j = 1 to n
5. Dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
6. Dist의 대각 성분 검사해 음수가 존재하면 음수 사이클이 존재한다는 것이므로 에러 출력

G. Init, update, sum in segment tree

i. Segment tree의 초기화, 노드 값 업데이트, 부분 합 출력을 수행하는 함수들입니다. 제공된 segment tree 코드를 약간 변경해 추가하였습니다.

ii. Init은 segment tree를 초기화하는 함수로, segment tree의 범위의 시작과 끝이 같지 않으면 두 자식 노드의 합으로 해당 노드를 초기화하고, 시작과 끝이 같으면 1(연결 여부를 나타냄)로 초기화하는 방식으로 구현하였습니다. 이때, 루트 노드가 시작점이기 때문에 값을 초기화하기 전에 범위의 시작과 끝이 같으면 1로 초기화하고 함수를 끝냈고, 그렇지 않을 때 절반씩 나누어 재귀적으로 Init을 호출해 leaf node까지 이동하도록 하였습니다.

iii. Update는 segment tree의 leaf 노드 값을 수정하기 위해 루트 노드부터 재귀적으로 호출하며 부분 합의 값을 수정하는 함수입니다. Target index가 범위 내에 있으면 업데이트해줄 값을 더해주고, 아니라면 그냥 함수를 끝내게 했습니다. 그리고, leaf 노드가 아니라면 범위를 반으로 나눠주어 재귀적으로 호출하게 했습니다.

iv. Sum은 segment tree의 전체 합 또는 부분 합을 출력해주는 함수입니다. 전체 범위에서 부분 범위가 포함되면 그 결과를 반환하고, 범위를 벗어나면 0을 반환합니다. 그리고 범위가 부분적으로 포함되면 범위를 반으로 나누어 재귀적으로 합 결과를 호출합니다.

H. Kwangwoon

i. Kwangwoon의 수도코드입니다.

1. 1번 vertex 방문
2. 방문한 vertex에서 갈 수 있는 vertex가 없을 때까지 3. – 5. 반복
3. 갈 수 있는 정점이 홀수 개면 가장 큰 정점 방문

4. 짝수 개면 가장 작은 정점 방문

5. 직전에 방문한 정점과 연결된 정점들의 연결을 끊음

I. Dijkstra, Bellman-Ford 경로 출력

i. Parent 배열에는 최단 경로 상에서 각 노드의 부모 노드가 저장되어 있기 때문에, 도착 지점에서 출발 지점까지 경로를 거슬러 올라갈 수 있습니다.

ii. 그러나, 출력 결과는 출발점에서 도착점까지로 되어야 하기 때문에 이를 거꾸로 출력해야 합니다.

iii. 그래서, 가장 나중에 들어온 데이터를 가장 먼저 출력해주는 stack을 이용해야겠다고 생각했습니다.

iv. 도착점을 i라 할 때, i = parent[i]로 갱신해주며 i 값이 -1이 아닐 때까지 stack에 넣어주었습니다.

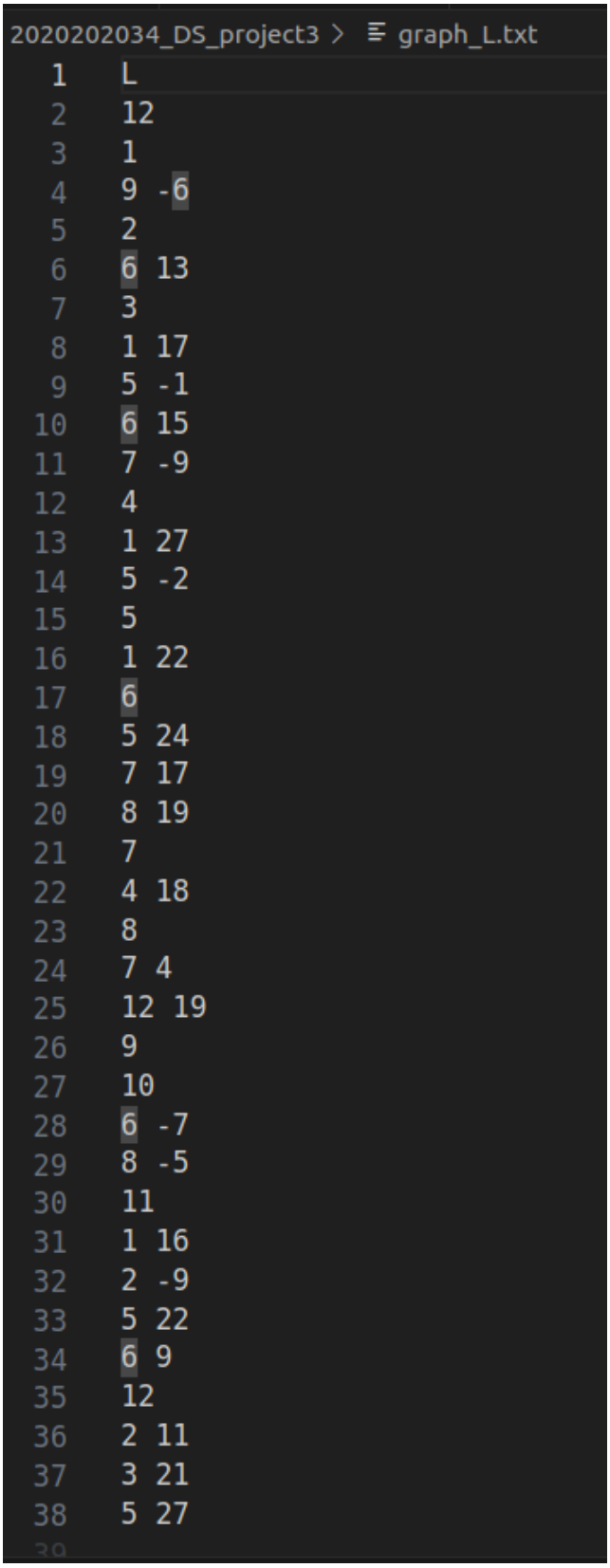
v. 그리고 stack의 top이 시작점이 아니라면 경로가 끊긴 것이므로 'x'를 출력하고, 그게 아니라면 stack을 pop하며 시작점부터 도착점까지 경로를 출력해주었습니다.

4. Result Screen

020202034_DS_project3 > command.txt

```
1   LOAD graph_L.txt
2   PRINT
3   BFS N 4
4   DFS Y 9
5   DIJKSTRA Y 1
6   BELLMANFORD Y 5 9
7   FLOYD Y
8   KRUSKAL
9   KWANGWOON
10  LOAD graph_M.txt
11  PRINT
12  BFS Y 4
13  DFS N 9
14  BELLMANFORD Y 2 8
15  FLOYD N
16  LOAD graph.txt
17  LOAD graph_L_pos.txt
18  DIJKSTRA Y 2
19  DIJKSTRA N 2
20  LOAD graph_M_neg_cycle.txt
21  BELLMANFORD Y 3 6
22  EXIT
```

명령어들이 쓰여있는 Command.txt입니다.



12개의 정점과 음수 가중치를 가지고 음수 사이클을 가지지 않는 adjacency list graph입니다.

2020202034_DS_project3 > graph_M.txt

```
M
12
0 0 0 0 0 0 0 0 -6 0 0 0
0 0 0 0 0 13 0 0 0 0 0 0
17 0 0 0 -1 15 -9 0 0 0 0 0
27 0 0 0 -2 0 0 0 0 0 0 0
22 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 24 0 17 19 0 0 0 0
0 0 0 18 0 0 0 0 0 0 0 0
0 0 0 0 0 0 4 0 0 0 0 19
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 -7 0 -5 0 0 0 0
16 -9 0 0 22 9 0 0 0 0 0 0
0 11 21 0 27 0 0 0 0 0 0 0
```

Graph_L.txt와 같은 내용을 가지는 adjacency matrix graph입니다.

2020202034_DS_project3 > graph_L_pos.txt

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

Dijkstra검사를 위해 만든 음수 가중치가 없고, 10개의 정점을 가지는 그래프 데이터입니다.

2020202034_DS_project3 > graph_M_neg_cycle.txt

```
M
10
0 0 0 0 0 0 0 0 0 5
0 0 7 0 0 0 0 0 0 -3
0 0 0 0 0 0 0 0 -6 0
0 0 4 0 0 8 0 -5 0 0
0 0 0 0 0 0 -6 0 0 -2
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 4 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0
```

3 -> 9 -> 4에서 음수 사이클을 가지는 그래프 데이터입니다.

2020202034_DS_project3 > log.txt

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

========PRINT========
[1]->(9,-6)
[2]->(6,13)
[3]->(1,17)->(5,-1)->(6,15)->(7,-9)
[4]->(1,27)->(5,-2)
[5]->(1,22)
[6]->(5,24)->(7,17)->(8,19)
[7]->(4,18)
[8]->(7,4)->(12,19)
[9]
[10]->(6,-7)->(8,-5)
[11]->(1,16)->(2,-9)->(5,22)->(6,9)
[12]->(2,11)->(3,21)->(5,27)
====================

========BFS========
Undirected Graph BFS result
startvertex: 4
4->1->5->7->3->9->11->6->12->8->2->10
====================

========DFS========
Directed Graph DFS result
startvertex: 9
9
====================
```

처음에 LOAD가 성공하였고, LOAD한 그래프를 출력하였습니다. 위에 캡쳐한 graph_L.txt와 비교하면 같은 것을 볼 수 있습니다. 해당 그래프는 모두 연결되어 있기 때문에 Undirected BFS로는 모든 점을 방문할 수 있지만, Directed DFS를 9에서 시작하면 9에서는 outgoing edge가 없기 때문에 9에서 시작 후 종료합니다. BFS를 추가로 보면, 4와 1, 5, 7이 adjacent 하므로 4, 1, 5,7 순으로 방문 후 1과 연결된 3, 9, 11을 방문하고 5와 연결된 6, 12를 방문함을 볼 수 있습니다.

```
========ERROR=======
700
====================

========Bellman-Ford========
Directed Graph Bellman-Ford result
5->1->9
cost: 16
====================

========FLOYD========
Directed Graph FLOYD result
        [1]    [2]    [3]    [4]    [5]    [6]    [7]    [8]    [9]    [10]    [11]    [12]
[1]      0      x      x      x      x      x      x      x     -6      x       x       x
[2]     59      0     72     48     37     13     30     32     53      x       x      51
[3]     17     64      0      9     -1     15     -9     34     11      x       x      53
[4]     20      x      x      0     -2      x      x      x     14      x       x       x
[5]     22      x      x      x      0      x      x      x     16      x       x       x
[6]     46     49     59     35     24      0     17     19     40      x       x      38
[7]     38      x      x     18     16      x      0      x     32      x       x       x
[8]     42     30     40     22     20     43      4      0     36      x       x      19
[9]      x      x      x      x      x      x      x      x      0      x       x       x
[10]    37     25     35     17     15     -7     -1     -5     31      0       x      14
[11]    16     -9     63     39     22      4     21     23     10      x       0      42
[12]    38     11     21     30     20     24     12     43     32      x       x       0
====================
```

그래프에 음수 가중치가 포함되어 있으므로 DIJKSTRA는 에러 코드를 반환합니다. Bellman-ford와 FLOYD의 결과가 같은 것으로 보아(가중치가 같음) 잘 작동한 것을 볼 수 있습니다. FLOYD의 출력에서 갈 수 없는 경로는 'x'로 표현되었습니다.

```
========Kruskal========
[1]     9(-6)11(16)
[2]     11(-9)12(11)
[3]     5(-1)7(-9)
[4]     5(-2)
[5]     3(-1)4(-2)
[6]     10(-7)11(9)
[7]     3(-9)8(4)
[8]     7(4)10(-5)
[9]     1(-6)
[10]    6(-7)8(-5)
[11]    1(16)2(-9)6(9)
[12]    2(11)
cost: 1
====================

========KWANGWOON========
startvertex: 1
1->11->6->2->12->8->7->3->5->4
====================

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

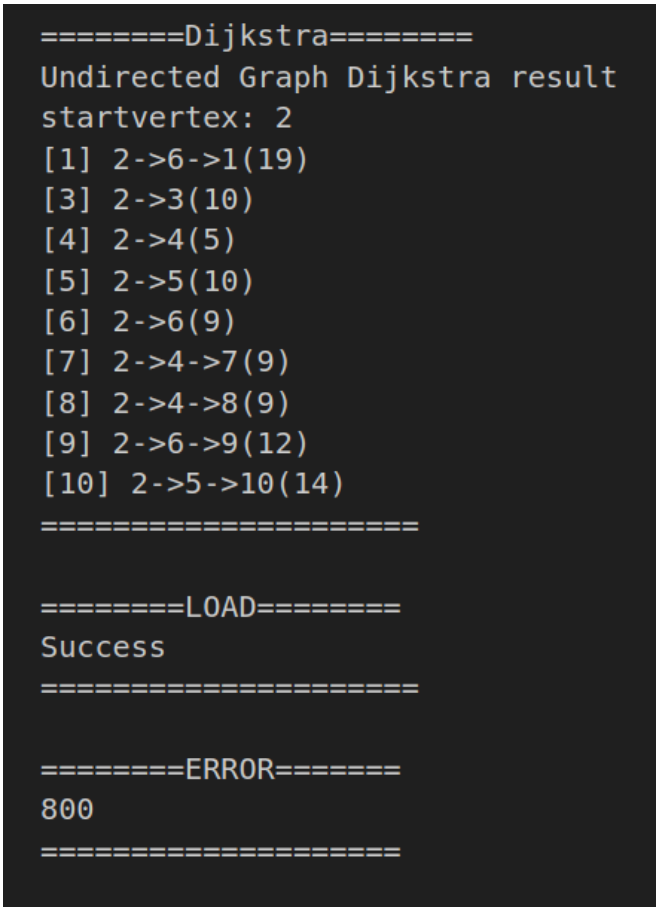
Kruskal의 결과를 보면 Edge가 총 11개(총 개수는 22개로 보이지만, 이는 1 -> 2, 2 ->1을 다른 edge로 본 것이므로 총 개수에서 2를 나누면 edge의 개수)이므로 12 – 1 == 11을 만족해 Minimal spanning tree를 만들었다고 할 수 있습니다. KWANGWOON 알고리즘의 결과를 보면 정점 1에서 시작했고, 1에서 갈 수 있는 정점이 3, 4, 5, 9, 11 총 5개, 즉 홀수이기 때문에 가장 큰 정점인 11로 이동했고 11에서 갈 수 있는 정점은 2, 5, 6(1은 이미 방문함), 총 3개, 홀수이기 때문에 가장 큰 6으로 이동하였습니다. 그리고 6에서 갈 수 있는 정점은 2, 3, 5, 7, 8, 10(11은 방문함) 총 6개, 짝수이므로 가장 작은 2번으로 이동한 것을 볼 수 있습니다. 그리고 이미 graph_L.txt를 Load했지만, graph_M.txt를 다시 Load하였고, 문제없이 이뤄진 것을 볼 수 있습니다.

```
========PRINT========
        [1]    [2]    [3]    [4]    [5]    [6]    [7]    [8]    [9]    [10]    [11]    [12]
[1]      0      0      0      0      0      0      0      0     -6      0       0       0
[2]      0      0      0      0      0     13      0      0      0      0       0       0
[3]     17      0      0      0     -1     15     -9      0      0      0       0       0
[4]     27      0      0      0     -2      0      0      0      0      0       0       0
[5]     22      0      0      0      0      0      0      0      0      0       0       0
[6]      0      0      0      0     24      0     17     19      0      0       0       0
[7]      0      0      0     18      0      0      0      0      0      0       0       0
[8]      0      0      0      0      0      0      4      0      0      0       0      19
[9]      0      0      0      0      0      0      0      0      0      0       0       0
[10]     0      0      0      0      0     -7      0     -5      0      0       0       0
[11]    16     -9      0      0     22      9      0      0      0      0       0       0
[12]     0     11     21      0     27      0      0      0      0      0       0       0
=====================

========BFS========
Directed Graph BFS result
startvertex: 4
4->1->5->9
=====================

========DFS========
Undirected Graph DFS result
startvertex: 9
9->1->3->5->4->7->6->2->11->12->8->10
=====================
```

Graph_M.txt는 adjacency matrix graph이므로 이에 맞게 형식을 출력하였습니다. 그래프의 내용은 같은 것을 확인할 수 있습니다. BFS와 DFS를 전과 달리 각각 Direct, Undirect로 수행하였습니다. 모든 정점이 연결되어 있으므로 Undirected DFS로는 모두 방문 하였지만, Directed BFS로는 4와 연결된 1, 5를 방문 후 1과 연결된 9를 방문한 후에는 9에서 나가는 edge가 없어(5와 연결된 정점은 이미 방문한 1밖에 없음) 전부 탐색하지 못했습니다.

```
========Bellman-Ford========
Directed Graph Bellman-Ford result
2->6->8
cost: 32
======================

========ERROR=======
900
====================

========ERROR=======
100
====================

========LOAD========
Success
======================

========Dijkstra========
Directed Graph Dijkstra result
startvertex: 2
[1] x
[3] 2->3->1(10)
[4] 2->4(5)
[5] 2->3->9->10->5(33)
[6] x
[7] 2->7->6(10)
[8] 2->3->9->10->5->8(43)
[9] 2->3->9(20)
[10] 2->3->9->10(29)
======================
```

Bellman ford 알고리즘의 시작점과 끝점을 2 -> 8로 바꿔보았습니다. 위에 FLOYD로 전체 최단 경로를 구한 결과를 보면 가중치가 32로 이와 같은 것을 확인할 수 있습니다. 그리고 FLOYD N으로 무방향일 때 FLOYD를 구했는데 에러 코드가 나왔습니다. 이는 음수 가중치가 있는 그래프를 무방향으로 보면 음수 가중치의 edge를 계속 왔다갔다 하면서 음수 사이클이 무조건 생기기 때문에 위에서 FLOYD Y가 제대로 되었음에도(음수 사이클이 없음) 무방향일 때 음수 사이클이 있다고 판단되는 이유입니다. 그리고 LOAD graph.txt 명령어를 실행하게 했는데, graph.txt 명령어는 존재하지 않았기 때문에 에러 코드를 출력하였으며, 그 후 양수 가중치만 존재하는 graph_L_pos.txt를 load하였습니다. 그 후 Dijkstra를 실행하였는데, 갈 수 없는 경로는 x로, 갈 수 있는 경로는 path 상의 모든 vertex와 그의 cost가 잘 출력되는 것을 볼 수 있습니다.



이번엔 같은 graph에서 Dijkstra를 무방향으로 실행하였습니다. 이전에 Direct로 실행한 Dijkstra와 결과가 다른 것을 볼 수 있는데, 이는 방향성이 사라져 기존에 갈 수 없던 정점을 갈 수 있게 된 것이라고 할 수 있습니다. 그리고 음수 사이클을 Bellman ford가 찾아낼 수 있는지를 검사하기 위해 음수 사이클이 있는 graph_M_neg_cycle.txt를 load하고 Bellman ford 36을 실행하였더니, 에러 코드가 출력되었습니다. 이는 해당 그래프가 3 -> 9 -> 4에서 음수 사이클을 가지기 때문에 이를 찾아낸 것이라 할 수 있습니다. 그후 EXIT 명령어를 실행해 프로그램을 종료해 더 결과가 출력되지 않았습니다.

5. Consideration

처음에는 방향성이 있는 그래프를 어떻게 무방향으로 읽어야 하나 고민할 때 무방향 그래프를 따로 만들어줘야 하나 싶었지만, 어차피 그래프의 정점에서 갈 수 있는 다른 정점, 간선을 찾을 때 method로 찾으므로 이를 고려한 method만 잘 만들어주면 되겠다고 생각했습니다. 그리고 Bellman ford에서 음수 사이클을 찾을 때 탐색 코드와 같이 자기 자신으로 가는 거리를 업데이트하지 않았더니 시작점이 음수 사이클에 포함되었을 때 이를 찾지 못했습니다. 그래서 음수 사이클을 찾을 때는 이 조건을 빼 문제를 해결하였습니다.