데이터 구조 설계 보고서

제목: DS_Project3

학　　과: 컴퓨터정보공학부

학　　번: 2022202064

성　　명: 최봉규

실습 분반: 금 5교시, 6교시

1. Introduction

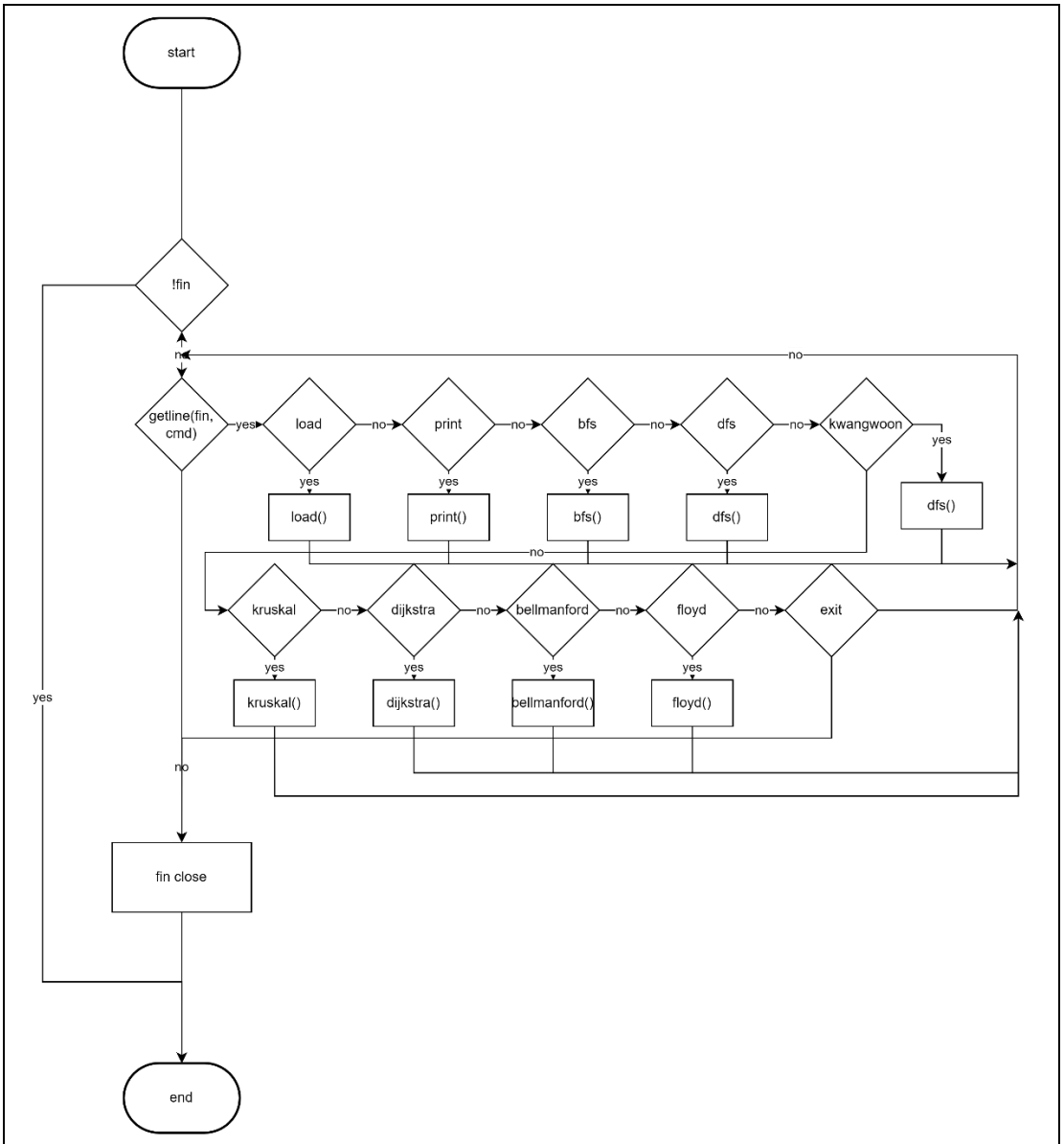
본 프로젝트는 그래프를 이용해 그래프 연산 프로그램을 구현한다. 이 프로그램은 그래프 정보가 저장된 텍스트 파일을 통해 그래프를 구현하고, 그래프의 특성에 따라 BFS, DFS, KRUSKAL, DIJKSTRA, BELLMAN-FOR, FLOYD 그리고 KWANGWOON 총 7가지 알고리즘을 통해 그래프 연산을 수행한다. 주어지는 그래프 데이터는 기본적으로 방향성과 가중치를 모도 가지고 있으며, 데이터 형태에 따라 List 그래프와 Matrix 그래프로 저장한다. 하지만 그래프 탐색 알고리즘 마다 방향성으로 고려해야 하기도 하고 고려하지 않아도 된다는 조건이 존재한다.

- BFS & DFS
  방향과 무방향이 가능하다. 그래프의 방향성과 가중치를 고려하고, 그래프 순회 또는 탐색 방법을 수행한다.
- Kruskal
  무방향만 가능하다. 알고리즘은 최소 비용 신장 트리 (MST)를 만드는 방법으로 방향성이 없고, 가중치가 있는 그래프 환경에서 수행한다.
- Dijkstra & Bellman Ford
  방향과 무방향 모두 가능하다. Dijkstra 알고리즘은 정점 하나를 출발점으로 두고 다른 모든 정점을 도착점으로 하는 최단경로 알고리즘은 방향성과 가중치 모두 존재하는 그래프 환경에서 연산을 수행한다. 만약 가중치가 음수일 경우 Dijkstra는 에러 코드를 출력하며, Bellman-Ford 에서는 음수 사이클이 발생한 경우 에러, 음수 사이클이 발생하지 않았을 겨우 최단 경로와 거리를 구한다.
- FLOYD
  방향과 무방향이 가능하다. Floyd 에서는 음수 사이클이 발생한 경우 에러, 음수 사이클이 발생하지 않았을 경우 최단 경로 행렬을 구한다.
- KwangWoon
  무방향만 가능하며, List graph만 이용해 구현한다. Kwangwoon이라는 그래프 탐색법은 다음과 같은 조건을 만족하며 그래프를 탐색하게 된다. 현재 정점에서 방문할 수 있는 정점(방문한 적이 없는 정점들) 들이 홀수개면 탐색할 수 있는 정점 번호들의 가장 큰 정점 번호로 방문을 시작하고, 짝수개면 가장 작은 정점 번호로 방문을 시작한다.

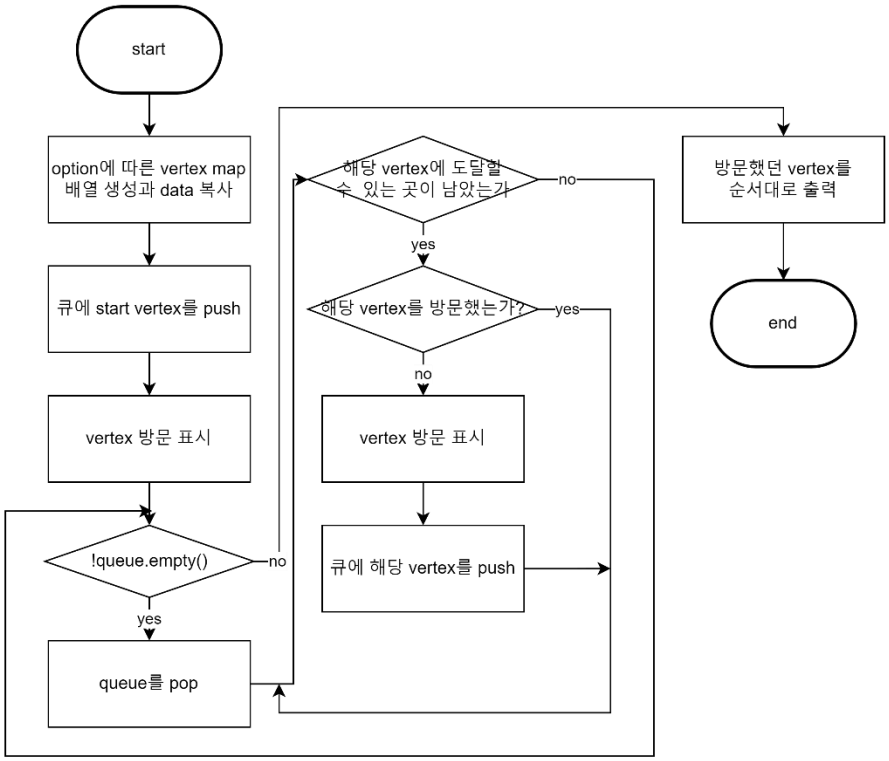
각 그래프 연산들은 'GraphMethod' 헤더 파일에 일반 함수로 존재한다. 그래프 형식(List, Matrix)에 상관없이 그래프 데이터를 입력 받으면 동일한 동작을 수행한도록 일반화하여 구현한다. 또한 충분히 큰 그래프에서도 모든 연산을 정상적으로 수행할 수 있

도록 구현한다. 프로그램의 동작은 명령어 파일에서 요청하는 command에 따라 각각의 기능을 수행하고, 그 결과를 출력 파일(log.txt)에 저장한다.

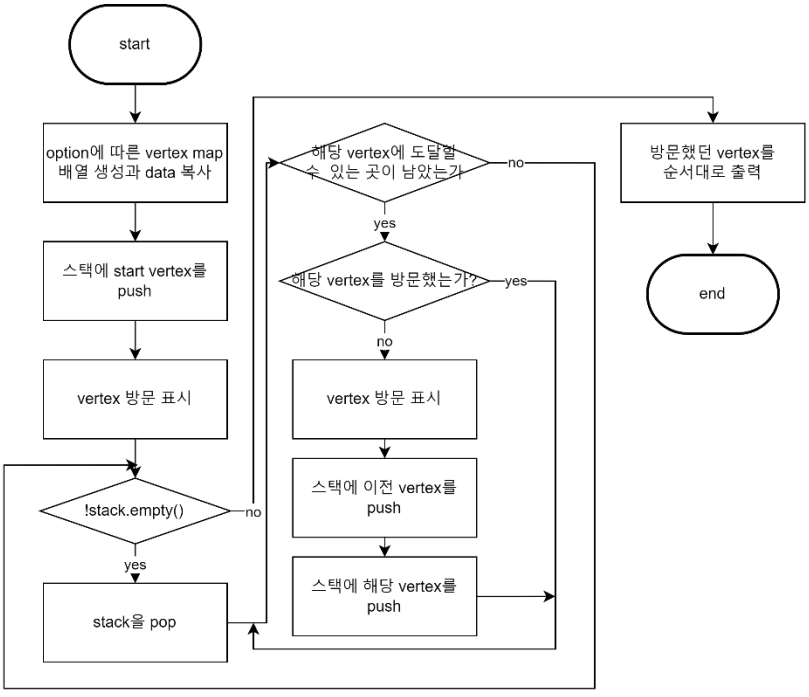
2. Flowchart



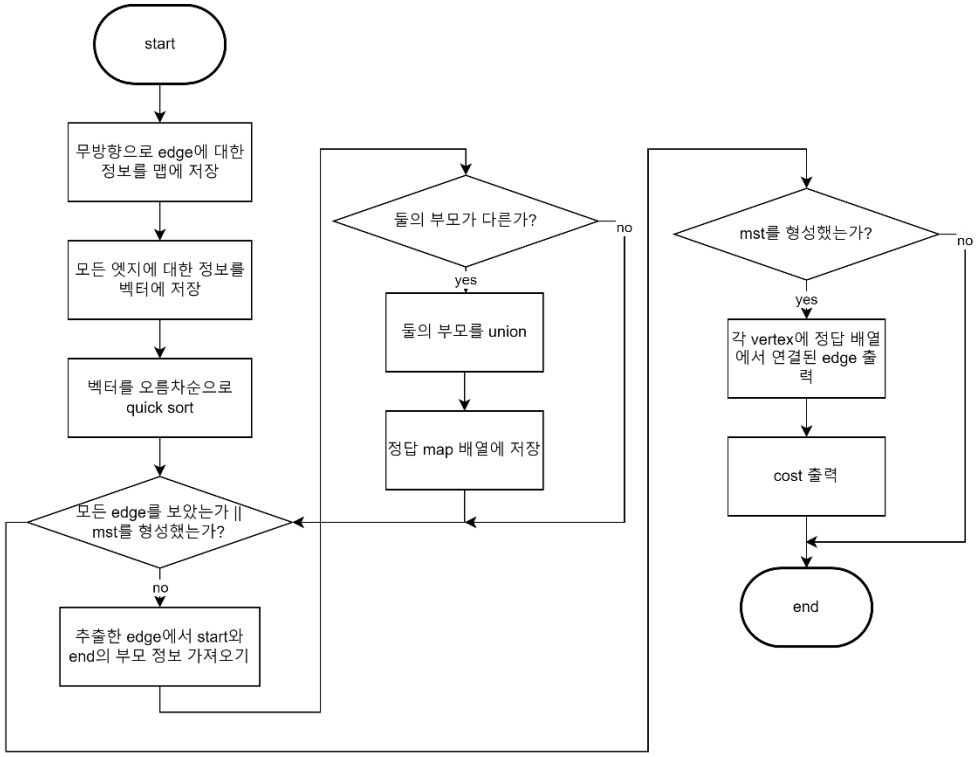
해당 flow chart는 manager.cpp의 run함수로 각각의 커맨드를 받아 커맨드에 따른 함수를 실행한다.



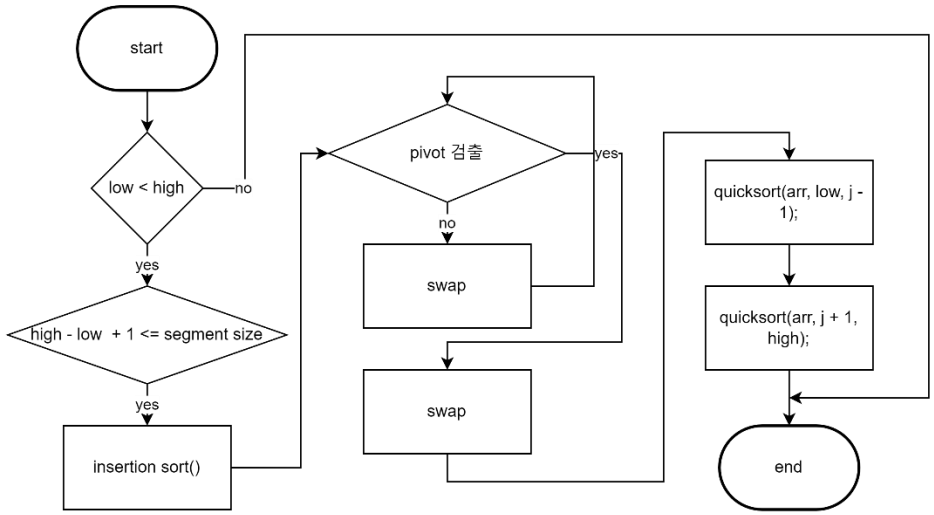
해당 flow chart는 BFS것으로 큐를 이용하여 해당 알고리즘을 수행, 방문한 vertex를 순서대로 출력하는 기능을 수행한다.



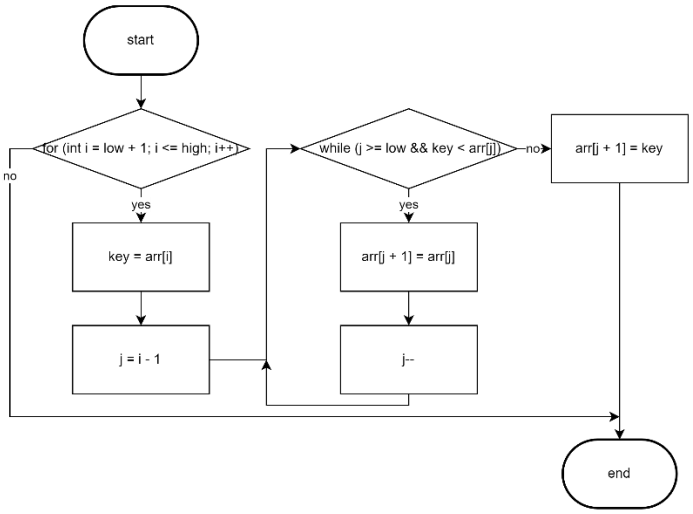
해당 flow chart는 DFS의 것으로 스택을 이용하여 해당 알고리즘을 수행, 방문한 vertex를 순서대로 출력하는 기능을 수행한다.



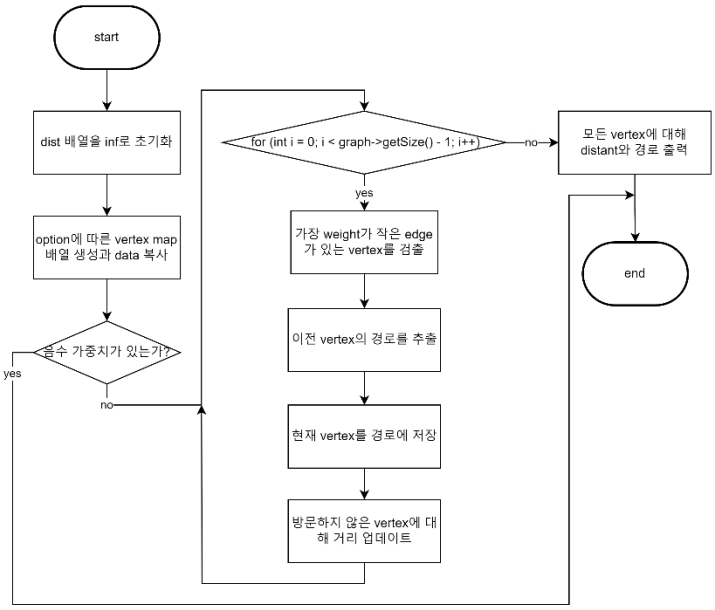
해당 flow chart는 kruskal algorithm의 것으로 map 배열과 find, union algorithm을 활용했다.



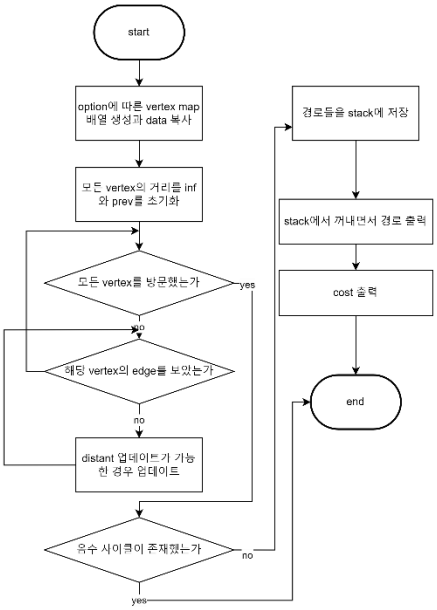
해당 quicksort의 것으로 재귀적으로 시행하며 segment 기준으로 insertion sort와 같이 수행한다.



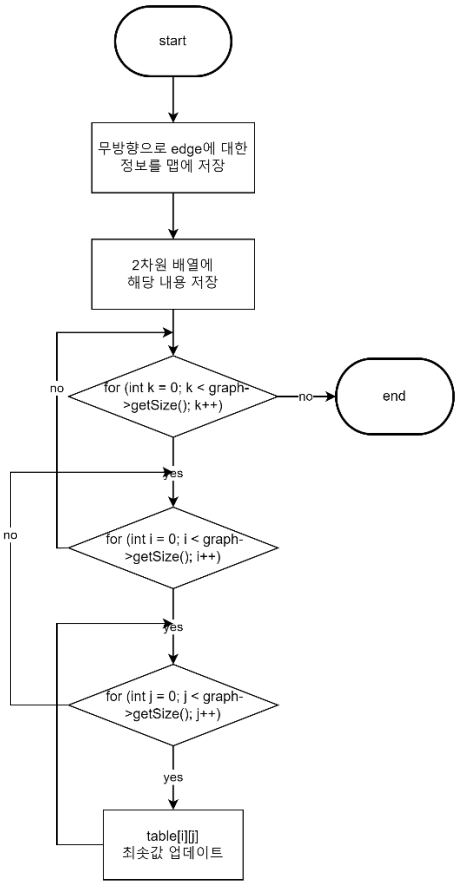
해당 flow chart는 insertion sort의 것으로 key에 대해서 제 자리를 찾을 때까지 배열에 저장된 수를 뒤로 미루며 이동하다가 찾은 경우 해당 자리에 key를 넣는 방식을 취한다.



해당 flow chart는 dijkstra의 것으로 음수 가중치가 있는 경우의 예외처리와 모든 vertex를 돌면서 startvertex로부터의 최단 경로와 거리를 출력한다.



해당 flow chart는 belllman-ford 알고리즘으로 모든 vertex와 vertex가 가지고 있는 모든 edge를 확인하면서 distant를 확인하고 음수 사이클을 확인, 경로 출력과 cost를 출력한다.



해당 flow chart는 floyd 알고리즘의 것이다.

3. Algorithm

-BFS

너비 우선 탐색은 시작 노드에서부터 거리에 따라 차례로 인접한 노드들을 탐색하는 방식이다. BFS는 queue를 활용하여 구현할 수 있다. 탐색을 시작할 노드를 선택한다. 큐에 시작 노드로 삽입 후, 방문한 노드를 표시하고 큐에서 해당 노드를 제거한다. 현재 노드의 인접한 노드들을 모두 큐에 삽입한다. 이미 방문한 노드의 경우에는 다시 큐에 추가하지 않도록 한다. 이를 큐가 완전히 비어 있을 때까지 반복한다.

-DFS

깊이 우선 탐색은 한 경로를 따라 끝까지 탐색한 후, 다음 경로로 넘어가며 탐색을 진행한다. 이를 통해 깊숙히 탐색하고자 하는 목적지까지 도달한 후, 되돌아와 가능한 경로를 탐색하게 된다. 해당 알고리즘은 스택이나 재귀를 활용하여 구현할 수 있다. 우선 탐색을 시작할 노드를 선택한다. 시작 노드를 방문한 것으로 표시하고, 해당 노드를 스택에 넣는다. 현재 노드의 인접한 노드 중에서 아직 방문하지 않은 노드를 선택하고, 해당 노드를 방문한 것으로 표시한다. 선택한 인접 노드로 이동하여 반복한다. 이때, 스택을 이용하여 깊이를 우선적으로 탐색하게 되고, 스택이 완전히 비어 있을 때까지 이를 반복한다.

-Kruskal

크루스칼 알고리즘은 MST를 찾는 그래프 알고리즘이다. 그래프의 모든 노드를 포함하면서 사이클이 없고 간선의 가중치 합이 최소인 부분 그래프를 찾는 것이 목표이다. 다음과 같은 순서로 동작한다. 그래프의 모든 간선의 가중치를 기준으로 오름차순으로 정렬한다. 정렬된 간선 중에서 가장 가중치가 작은 간선을 선택한다. 선택한 간선이 현재까지 선택한 간선들과 사이클을 형성하는 지 확인한다. 이때 union & find 알고리즘을 이용하여 set에 대해 파악한다. 모든 간선을 확인하거나 MST가 만들어진 경우의 해당 알고리즘을 종료한다.

-**Dijkstra**

다익스트라 알고리즘은 모든 노드까지의 최단 경로를 찾는 알고리즘이다. 주로 가중치가 있는 그래프에서 각 간선의 가중치의 합이 최소가 되도록 하는 최단 경로를 찾을 때 사용된다. 시작 노드를 선택하고, 시작 노드로부터의 최단 경로를 나타내는 배열을 초기화한다. 해당 배열은 출발 노드로부터의 각 노드까지의 현재까지 발견된 최단 경로의 길이를 저장한다. 시작 노드의 최단 경로 값을 0으로 설정한다. 이외의 경우에는 전부 INF로 설정한다. 가장 짧은 최단 경로를 가진 노드를 하나씩 찾으며 해당 노드에 연결된 간선을 통해 다른 노드들까지의 최단 경로를 갱신한다. 만약 새로운 경로가 더 짧다면 업데이트

한다. 다익스트라 알고리즘은 음의 가중치를 갖는 간선이 없을 때에만 사용이 가능하다.

-**Bellman Ford**

벨만포드 알고리즘은 그래프에서 음의 가중치를 포함하는 최단 경로를 찾는 알고리즘이다. 벨만포드 알고리즘은 다음과 같이 동작한다. 시작 노드를 선택하고, 시작 노드로부터의 최단 경로를 나타내는 배열을 초기화 한다. 이 배열은 출발 노드로부터 각 노드까지의 현재까지 발견된 최단 경로의 길이를 저장한다. 모든 간선에 대해 출발 노드에서 도착 노드까지의 경로를 현재까지의 초단 경로와 비교하여 갱신한다. 만약 새로운 경로가 더 짧다면 해당 노드의 최단 경로 값을 갱신한다. 정점의 수만큼 이전의 과정을 반복한다. 다만 마지막 순서에서 최단 경로가 갱신된다면 음수사이클로 판명한다.

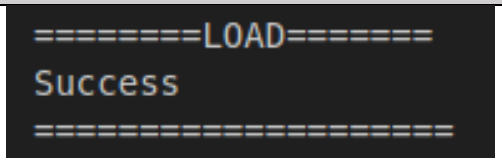
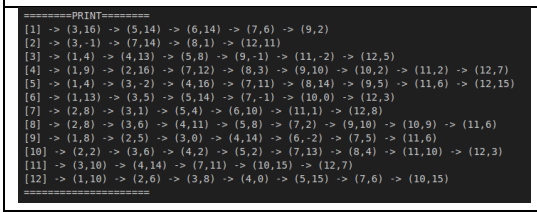
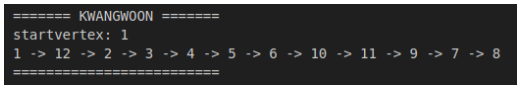
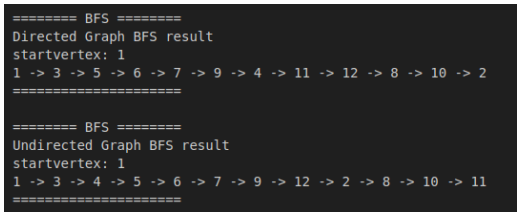
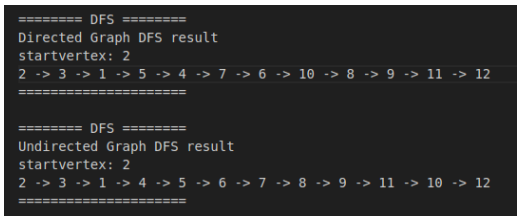
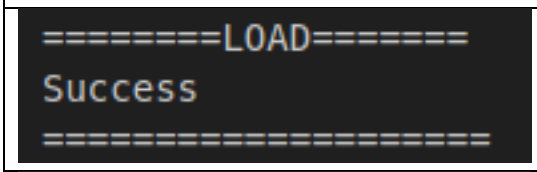
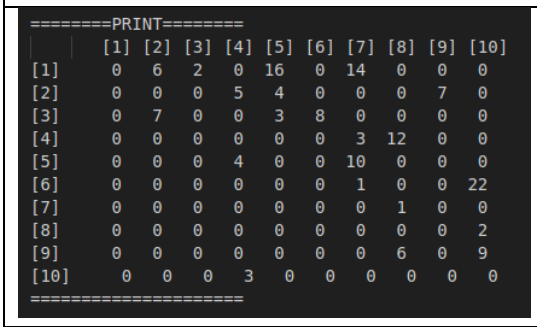
-FLOYD

모든 정점 간의 최단 경로를 찾는 알고리즘이다. 음의 가중치가 있어도 가능하다. 거쳐가는 정점을 기준으로 최단 경로를 찾는다. 모든 정점을 시작점, 중간 정점, 도착점으로 선택하여 모든 쌍의 최단 경로를 찾는다. 인접 행렬을 사용하여 그래프의 간선 정보를 표현하고 각 정점 쌍에 대한 최단 경로를 저장할 행렬을 초기화 한다. 초기화 단계에서는 직접적인 간선이 있는 경우에는 해당 가중치를, 직접적인 간선이 없는 경우에는 inf를 할당한다. 각 정점을 중간 정점으로 선택하여 시작 정점과 도착 정점 간의 최단 경로를 업데이트한다. 모든 정점을 시작 정점, 거쳐가는 정점, 도착 정점으로 선택하여 최단 경로를 업데이트 한다. 여러 번 이를 반복하면 최종적으로 모든 쌍의 최단 경로를 계산할 수 있다. 자기 자신으로 가는 것이 업데이트 된다면 음수 사이클로 간주한다.

-KwangWoon

항상 정점 "1"부터 탐색을 시작한다. 따라서 정점 1부터 탐색을 시작하여 탐색의 순서를 출력하면 된다. 현재 연결되어 있는 간선의 수가 짝수라면 가장 작은 정점 번호로 방문을 시작하고, 홀수라면 가장 큰 정점의 번호로 방문하도록 한다. 세그먼트 트리는 현재 정점에서 다른 정점으로 이동할 수 있는 이동 여부를 위해 사용된다. 리프 노드는 다른 정점으로의 이동 여부를 담고 있다. 만약 1번 정점이 없는 경우 에러를 출력한다. 세그먼트 트리를 이용하며 각 노드는 트리를 갖는다. 트리의 루트는 현재 갈 수 있는 vertex의 수를 가지고 있으며, 이를 통해 다음 노드를 갈 수 있는 가를 결정한다.

4. Result Screen

결과 화면	동작 설명
	Graph_L을 로드했을 때의 결과이다.
	해당 그래프를 출력했을 때의 결과이다.
	해당 그래프에 대해서 광운 서치 알고리즘을 적용한 모습이다.
	해당 그래프에 대해서 startvertex를 1로 설정후, BFS를 한 결과이다. 방향성이 있을 때와 없을 때를 본 결과이다. 방향성이 있을 때는 연결이 되더라도 정점을 가지 못할 수 있다. 그렇기 때문에 탐색 순서가 바뀔 수 있다.
	startvertex를 2로 설정후, DFS를 한 결과이다. 방향성이 있을 때와 무방향일 때 둘 모두 확인한 결과로 BFS와 마찬가지로 연결이 되어있으나, 방향이 있는 경우 정점을 가지 못할 수 있다. 그렇기 때문에 탐색 순서가 바뀔 수 있다.
	Graph_M을 로드했을 때의 모습이다.
	해당 그래프에 대해 출력한 모습이다.

<table>
<tr>
<td>

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

</td>
<td>해당 그래프에 대해 방향성을 가지며, startvertex를 2로 설정 후, Dijkstra 알고리즘을 사용한 결과화면이다.</td>
</tr>
<tr>
<td>

```
======= Kruskal =======
[1]     3(2)
[2]     5(4)
[3]     1(2)5(3)
[4]     5(4)10(3)
[5]     2(4)3(3)4(4)
[6]     7(1)
[7]     6(1)8(1)
[8]     7(1)9(6)10(2)
[9]     8(6)
[10]     4(3)8(2)
cost: 26
======================
```

</td>
<td>해당 그래프에 대해 Kruskal 알고리즘을 돌린 화면이다.</td>
</tr>
<tr>
<td>

```
=======PRINT=======
      [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1]     0   6   2   0  16   0  14   0   0   0
[2]     0   0   0   5   4   0   0   0   7   0
[3]     0   7   0   0   1   8   0   0   0   0
[4]     0   0   0   0   0   0   3  12   0   0
[5]     0   0   0   4   0   0  10   0   0   0
[6]     0   0   0   0   0   0   1   0   0  22
[7]     0   0   0   0   0   0   0   1   0   0
[8]     0   0   0   0   0   0   0   0   0   2
[9]     0   0   0   0   0   0   0   6   0   9
[10]    0   0   0   3   0   0   0   0   0   0
==================
```

</td>
<td>가중치를 살짝 수정해보았다. 3->5로 가는 것을 기존 3을 1로 수정했다.</td>
</tr>
<tr>
<td>

```
======= Kruskal =======
[1]     3(2)
[2]     5(4)
[3]     1(2)5(1)
[4]     5(4)7(3)
[5]     2(4)3(1)4(4)
[6]     7(1)
[7]     4(3)6(1)8(1)
[8]     7(1)9(6)10(2)
[9]     8(6)
[10]     8(2)
cost: 24
======================
```

</td>
<td>현재 값이 수정되면서 cost의 값이 기존 26에서 24로 바뀐 것을 볼 수 있다. 또한 3->5의 가중치도 1로 바뀐 것을 확인할 수 있다.</td>
</tr>
</table>

<table>
<tr>
<td>

```
======= Bellman-Ford =======
Undirected Graph Bellman-Ford result
1 -> 2 -> 9
cost: 13
======================

======= Bellman-Ford =======
Directed Graph Bellman-Ford result
1 -> 2 -> 9
cost: 13
======================

======= Bellman-Ford =======
Undirected Graph Bellman-Ford result
9 -> 2 -> 1
cost: 13
======================

======= Bellman-Ford =======
Directed Graph Bellman-Ford result
x
======================
```

</td>
<td>해당 그래프에 대해서 bellman ford로 방향과 무방향으로 1에서 9를 향하는 것과 9에서 1로 가는 것을 확인해 보았다. 1에서 9, 9에서 1로 가는 것은 비용이 같다. 하지만 4번째의 결과를 확인하면 방향성이 있는 것으로 9에서 1로 가는 길이 없어 x로 표현한다.</td>
</tr>
<tr>
<td>

```
======= FLOYD =======
Directed Graph FLOYD result
      [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1]     0   6   2   7   3  10  10  11  13  13
[2]     x   0   x   5   4   x   8   9   7  11
[3]     x   7   0   5   1   8   8   9  14  11
[4]     x   x   x   0   x   x   3   4   x   6
[5]     x   x   x   4   0   x   7   8   x  10
[6]     x   x   x   7   x   0   1   2   x   4
[7]     x   x   x   6   x   x   0   1   x   3
[8]     x   x   x   5   x   x   8   0   x   2
[9]     x   x   x  11   x   x  14   6   0   8
[10]    x   x   x   3   x   x   6   7   x   0
====================

======= FLOYD =======
Undirected Graph FLOYD result
      [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1]     0   6   2   7   3  10  10  11  13  10
[2]     6   0   5   5   4   9   8   9   7   8
[3]     2   5   0   5   1   8   8   9  12   8
[4]     7   5   5   0   4   4   3   4  10   3
[5]     3   4   1   4   0   8   7   8  11   7
[6]    10   9   8   4   8   0   1   2   8   4
[7]    10   8   8   3   7   1   0   1   7   3
[8]    11   9   9   4   8   2   1   0   6   2
[9]    13   7  12  10  11   8   7   6   0   8
[10]    10   8   8   3   7   4   3   2   8   0
====================
```

</td>
<td>해당 그래프에 대해 방향성을 갖고 Floyd 알고리즘을 적용한 결과화면이다. 위는 방향성을 갖고, 아래는 무방향이다. 무방향에 대해서는 floyd는 symmetric한 matrix로 표현되는 거에 반해 방향으로 찾은 것은 연결되지 못했다는 표시의 x표시가 많은 것을 확인할 수 있다.</td>
</tr>
<tr>
<td>

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

</td>
<td>Exit을 하여 프로그램을 종료한다.</td>
</tr>
</table>

5. Consideration

이번 프로젝트를 진행하면서 그래프에 대해 막연하게 생각하던 과거의 틀을 깰 수 있었다. 직접 구현하면서 생각보다 어려운 알고리즘이 아니었다는 것을 느꼈고, 활용 방안에 대해서도 많이 생각해 볼 수 있는 시간이었다고 생각한다. 광운 알고리즘을 진행하면서 세그먼트 트리를 공부하면서 굉장히 쓸모 있는 알고리즘을 배웠다.

크루스칼에 대한 결과를 친구와 비교하면서 엣지 수는 같으나 코스트가 다른 경우를 확인할 수 있었다. 이에 대해 조금 생각해 보니, 크루스칼이 그리디하게 움직인다고 생각하게 되었다. 당장 가장 작은 것을 고려해서 union을 하지만, 거시적으로 보면 최단 경로가 아닐 수도 있다는 것이다. 이에 대해 검색을 해 본 결과 크루스칼 알고리즘은 최적 부분 구조를 가지고 있어 각 단계에서의 최적해를 선택한다. 이로 인해 최종적으로 얻어지는 해가 전체적으로 최적인지는 보장되지 않는다. 모든 간서의 가중치가 서로 다르다면 크루스칼 알고리즘을 사용하여 항상 최적의 해를 찾을 수 있지만, 가중치가 같은 간선이 여러 개 존재하는 경우에 대해 최적해를 보장하지 못할 수 있다.