데이터 구조 설계 프로젝트3 보고서
제출 일자: 2022년 12월 15일(목)

학과: 컴퓨터정보공학부
담당교수: 이기훈교수님
학번: 2021202045
이름: 김예은

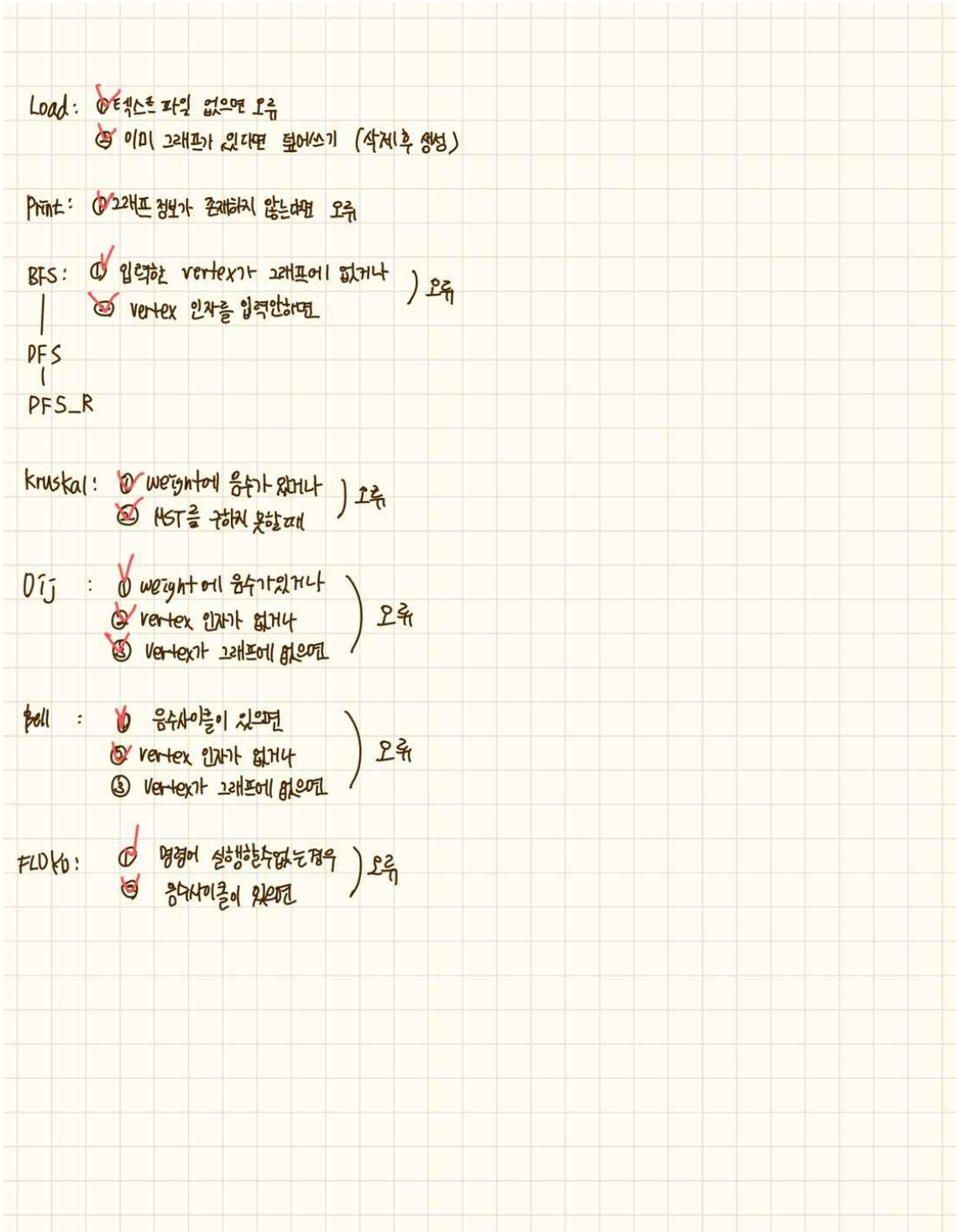
1. Introduction

이번 3차 프로젝트는 수업시간에 배운 거의 모든 그래프 탐색 방법과 mst구하기, 최단경로 구하기 등 그래프에 대한 알고리즘을 직접 구현해보는 것이 목적이다. 그래프 탐색 방법엔 DFS(깊이 우선 탐색)과 BFS(넓이 우선 탐색)이 있는데 DFS는 찾으려는 노드가 깊게 위치해 있을 때 빠르게 찾을 수 있다는 장점이 있지만, 해당 경로에 찾고자 하는 노드가 없다면 너무 깊게 빠진다는 단점과 탐색 경로가 항상 최선의 방법이 아니라는 한계점을 가진다. BFS는 최적의 방법을 가르쳐주는 장점을 가지지만, node의 가짓수가 많아지면 complexity가 커진다는 단점이 있다. MST란 minimum spanning tree로, 최소의 weight를 거쳐서 spanning tree를 만드는 것을 말한다. prim's, kruskal's method가 있으나 이번 프로젝트에서는 kruskal's method만 구현하도록 한다. 지금까지 설명 세 개의 그래프 관련 연산은 이번 프로젝트에서 undirected graph라고 가정하고 구현한다. 크루스칼의 경우 만약 vertex 0->1로 가는 edge가 있고, 1->0으로 가는 edge가 있을 때, 둘 중 weight값이 더 작은 것을 기억하여 두 vertex를 undirected 선분으로 이어줬다. BFS, DFS는 방향성과 가중치를 고려하지 않고 그래프를 순회 및 탐색한다. 만약 edge가 없고 vertex만 떨어져 있을 때도 고려한다. 방향성과 양수 가중치를 고려하는 다익스트라 방법은 start vertex로부터 모든 end vertex까지의 최단경로를 저장하여 출력해준다. 만약 가중치가 음수값이 있다면 다익스트라는 에러코드가 뜬다. bellman-ford란, 음수 가중치까지 고려한 다익스트라 방식이라고 생각하면 된다. 단, 음수 사이클이 있다면 계속 최솟값이 갱신되면서 무한 루프를 돌 가능성이 있기때문에 음수 사이클이 발생한다면 에러코드가 나오게 하였다. bellmanford에서는 거쳐가야 하는 노드의 개수를 지정해준다. floyd의 경우도 방향성 및 음수를 고려하므로 음수 사이클이 생긴다면 에러코드가 나오게끔 해주었다. floyd의 경우는 거쳐야 하는 노드를 숫자로 지정해준다. 예를 들어, k=0이라면 0번노드를 지나는 path가 있는지 없는지 확인해야한다. 각 그래프에 대한 연산은 manager class에서 command.txt에서 명령어를 읽고, 명령어에 맞게 연산을 해준 후 그래프 출력 형식에 맞게 출력을 해준다.

2. Algorithm

(1) Manager

먼저 command.txt를 읽어서 명령어에 맞게 if~else if문에 들어간다. load가 성공하였다면 load 변수값을 0->1로 바꾸어준다. 이미 load=1인데 다시 한번 LOAD명령어가 입력이 된다면 기존에 있던 그래프를 delete한 다음, 다시 load를 해주었다. ~Graph()는 virtual로 해주어서 main함수가 끝날 때 소멸자가 자동 소환되면서, 사용한 graph의 객체를 삭제하게 해주었다. start vertex를 parameter로 입력받는 명령어가 있는데 맨처음에 start vertex를 -1로 초기화 해주었다가 start vertex가 계속 -1이면 start vertex가 update가 되었지 않다는 뜻이므로, 이는 parameter를 입력을 받지못했다고 간주하여 에러처리를 할 수 있도록 했다. graph의 size는 0으로 초기화해주어 만약 그래프가 생성되지 않았다면 (graph->size가 0일 때) 그래프 연산을 할 때 에러를 뜨게 해주었다. 그래프의 type에 따라 구분하여 load받은 정보들을 List의 경우 map형태에 insert해주었고, Matrix의 경우 2차원 int형에 저장해주었다. 주의할 점은, 모두 new를 이용하여 동적할당을 해주었기 때문에 메모리 할당 해제를 잘 해줘야한다는 점이 있다. 다음 그림은 전체적으로 구현해줘야 하는 에러 처리 사항을 정리해놓은 것이다.



(2) BFS

BFS는 queue를 이용하여 구현할 수 있다. bool형 배열 visited를 선언해주고, 모든 요소를 false로 초기화시켜준다. 시작 vertex를 queue에 push해주고, pop하는 동시에 해당 vertex와 인접한 edge들을 모두 queue에 push해준다. queue에 넣어진 vertex들에 대한 visited[vertex]는 true로 update되면서 이미 그 노드는 방문했다는 정보를 저장한다. 그 다음 다시 queue.pop을 진행하여 queue에서 front노드를 빼주고 그 front노드와 인접한 노드를 다시 queue에 push, visited[vertex]=true를 해

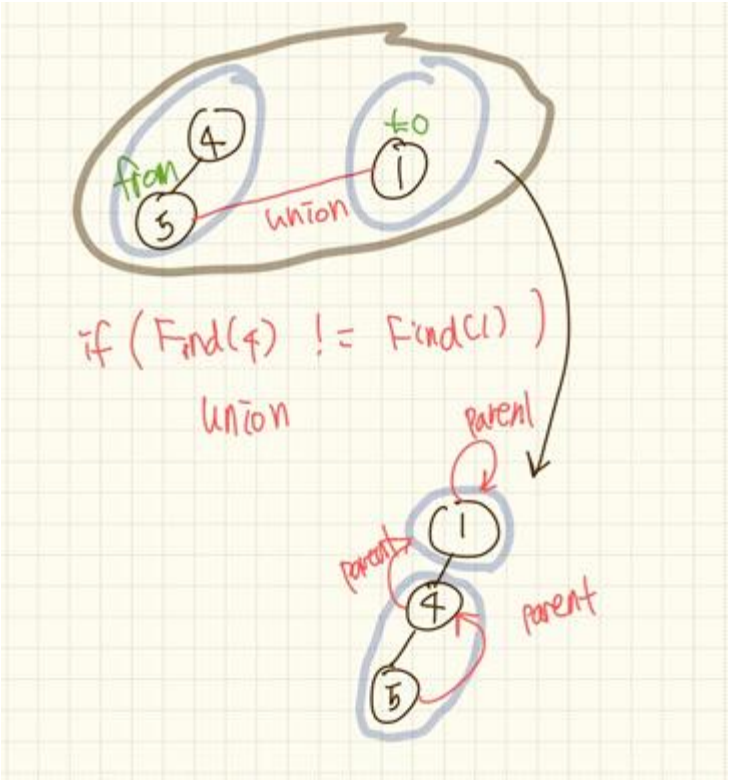
준다. q가 empty가 될 때까지 이 과정을 반복한다. 출력형식에 맞게 출력하기 위해 vector를 하나 선언하여 q.pop이 된 노드들을 vector에 저장시키고 vecotr를 iterator를 사용하여 순회하면서 BFS경로를 출력해주었다. 여기서 인접한 노드들을 구하는 함수는 MatrixGraph.cpp 또는 ListGraph.cpp에 구현해놓았다. map이나 int 형 배열을 모두 순회하다가 방향성을 고려해야 하는 경우 입력받은 vertex에서 나가는 방향의 노드들만 구해 map형에 넣어주었고, 비방향성일 때는 입력받은 vertex에서 나가는 vertex 및 가중치값과 입력받은 vertex로 들어오는 edge를 가진 vertex의 값 및 가중치를 저장해주었다. 여기서 까다로운 점은 서로를 가리킬 때이다. 예를 들어 0->1과 1->0이 둘다 있을 시에는 비방향성이므로 둘 중에 하나의 가중치값만 기억할 수 있는데 이때, 둘 중에 작은 가중치값만 min algorithm으로 가져와 그 값만을 저장할 수 있도록 구현해주었다.

(3) DFS, DFS_R

DFS는 stack을 이용하여 구현한 방법 1, 재귀함수를 이용하여 구현한 방법 2가 있다. stack의 경우 위에서 설명한 DFS와 전체적인 흐름은 비슷하다. 다른 점은, BFS에서는 인접한 노드들을 모두 한번에 queue에 넣어준 반면, DFS의 경우 인접한 노드들 중에서 방문하지 않는 노드를 stack에 push한 다음 해당 노드를 방문 처리한 다음 break문을 만나 다시 바로 그 노드와 인접한 노드를 stack에 넣는 방식으로 진행된다는 점이다. 재귀함수의 경우 Manager.cpp의 mDFS_R에서 재귀함수를 호출하도록 하였다. 재귀함수의 인자로 들어온 vertex를 true처리하고 다시 vertex의 인접한 노드중에 방문하지 않은 노드를 다시 DFS_R함수에 넣어주었다. 이 함수에서 queue를 쓴 이유는 고찰에 쓰도록 하겠다.

(4) Kruskal's method

크루스칼은 방향성을 고려하지않은 그래프로, 가중치에 음수값이 없다고 가정한다. 만약 있으면 에러처리를 한다. 크루스칼은 가중치값을 sorting하여 작은 순서대로 vertex끼리 이어준 후, 만약 cycle을 형성하면 pass하도록 해야한다. 이를 구현하기 위해 quick sort, insertion sort, union, find함수를 따로 구현해주었다. 우선 insertion sort는 정렬하고자하는 배열의 크기가 7보다 작으면 insertion sort를 하게 해주었다. insertion sort는 배열 사이에 벽을 세워서 벽 왼쪽은 정렬을 하고 벽을 오른쪽으로 한칸씩 밀면서 오른쪽의 수를 왼쪽으로 넣어 정렬해주는 방식이다. quick sort는 맨 왼쪽 배열값을 pivot으로 잡아서 재귀적으로 분할하여 sorting해주었다. do while문 세 개를 써서 구현해주었다. 구현 코드는 데구설 자료를 참고하였다. left와 right을 가리키는 값이 서로 switch할때까지 재귀함수가 호출된다. 맨 처음에 parent배열을 선언하여 초기화를 모든 노드가 각자 자기 숫자를 루트로 가지게 해주었다.
Find함수에서 둘의 parnet값이 다르면 같은 집합에 있지않다고 생각하여 Union을 해준다. Union을 해줄 때는 더 큰 root값을 가진 트리가 작은 트리의 subtree로 들어가게끔 구현해주었다. count값을 이용하여 union이 될 때마다(union이 되면 edge가 설정되었다는 뜻이므로) count값을 ++해주었는데 만약 count값이 graph->getSize()-1이 아니라면 mst가 만들어지지않았다는 의미이므로 에러처리를 해주었다.



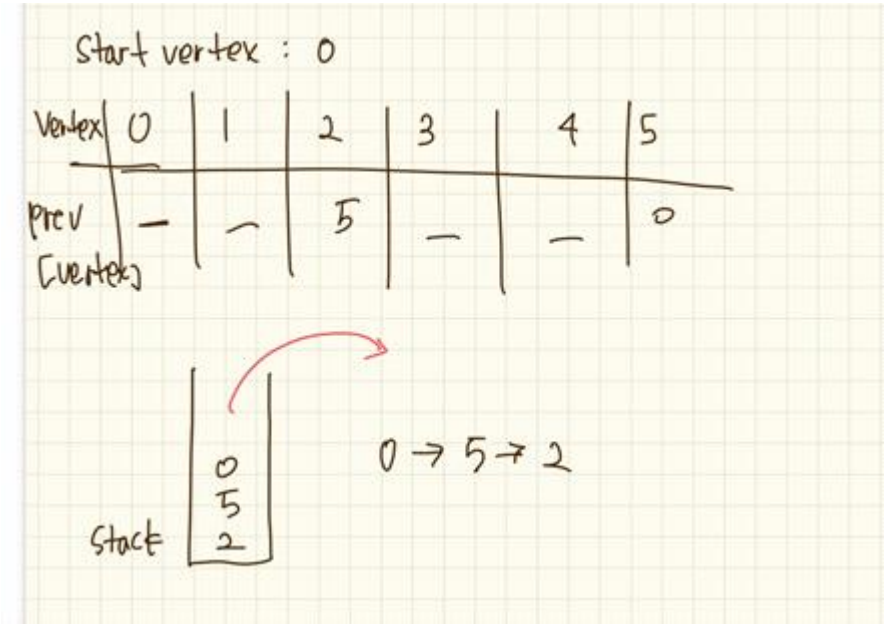
5. 다익스트라

다익스트라는 방향성을 고려하고, 가중치에 negative값이 있으면 에러처리를 해주었다. 즉 양수의 가중치만 있다는 가정 하에 최단거리를 구해주는 알고리즘인데 핵심 코드는 다음과 같다.

```
if (visited[w] == false)
{
    int past = dist[w];
    dist[w] = min(dist[idx] + len[idx][w], dist[w]);
    if (dist[w] < past) {
        if (w != vertex)
        {
            prev[w] = idx;
        }
    }
}
```

위의 코드처럼 w로 가는 거리가 기존의 dist[w]라고 하고 idx를 거쳐서 가는 방법을 dist[idx]+len[idx][w]라고 한다면 둘 중에 작은 거리를 취하여 갱신하는 것이다. 거리 갱신과 동시에 prev(자신이 어디vetex로부터 왔는지 경로를 저장해주는 배열)도 갱신시켜준다. 만약 w가 start vertex라면 바꾸지않는다. 이 부분을 추가해주지 않았더니 0->6(4), 6->0(5)의 edge가 있을 때 무한반복으로 순환해주어서 힘들었다.

이렇게 prev에 저장된 경로를 출력을 해줄 때는 stack을 이용하여 출력해주었다. 다음 그림을 참고하면 된다.
다음 그림은 0에서 출발하여 2까지 가는 경로를 출력해주는 방식을 stack으로 어떻게 구현하였는지 간단히 그려본 것이다. while문을 이용하여 prev[vertex]가 MAX를 만나기 전까지 stack에 경로값을 넣고 stack이 빌 때까지 pop을 해준 것이다.



6. bellman-ford

밸만 포드 알고리즘은 다익스트라와 달리 음수 가중치값이 있다고 가정하고 최단경로를 구해주는 것인데, 이때 음수 사이클이 발생하면 무한 반복적으로 최단 거리를 갱신시켜주므로 이때는 에러처리를 해준다. 이는 최단 경로를 구해주는 for 문을 한번 더 돌 때 또 갱신하면 음수사이클이 있다는 것으로 간주하여 코드를 짜면 된다. 이는 데구실 수업 자료에서 아이디어를 얻은 것으로 다음과 같다.

Bellman-Ford DS Lab.

☐ 모든 vertex에 연결된 edge 정보를 반복적으로 갱신

1. 시작 vertex의 거리를 0으로 초기화, 다른 vertex로의 거리를 무한대로 초기화
2. 각 vertex에 대하여 주변 edge로의 거리를 보다 짧은 경로로 갱신
3. 2.를 vertex의 수만큼 반복
4. 2.를 한 번 더 수행했을 때, 거리가 갱신되는 경우가 있다면 음수 사이클 발생

```
void MatrixWDigraph::BellmanFord(const int n, const int s)
{
    // initialize dist
    for (int i = 0; i < n; i++) dist[i] = length[s][i];

    for (int k = 2; k <= n-1; k++)
        for (each v such that v != s and
                              v has at least one incoming edge)
            for (each <w, v> in the graph)
                dist[v] = min(dist[v], dist[w] + length[w][v]);
}
```

그 외는 다익스트라 알고리즘과 비슷하게 구현하였고, visited 배열이 필요가없다는 장점이 있다. 출력형식도 위에서 설명했듯이 stack을 이용하여 출력해주었는데 이 알고리즘의 경우 start vertex와 end vertex를 정해주므로 출력과정이 더 간단하다.

7. Floyd

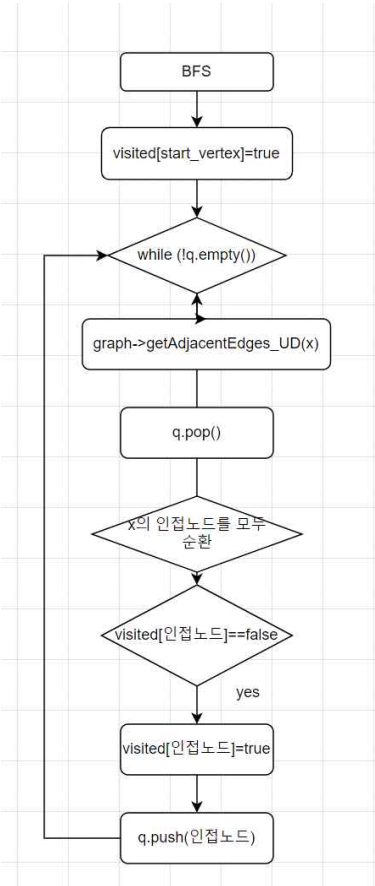
Floyd의 경우 모든 정점에서 모든 정점으로 가는 최단 경로를 matrix형태로 print하는 것으로 출력 형식은 matrix print형식을 그대로 가져왔다. floyd의 경우 지나가야하는 노드번호를 지정해주는 알고리즘으로 3중 while문을 이용하여 구현해주었다. 이

```
if (len[i][k] + len[k][j] < len[i][j]) {  //distance[i,j] = min(distance[i,j], distance[i,n] + distance[n,j])
    if (len[i][k] == MAX || len[k][j] == MAX) {
        continue;
    }
    len[i][j] = len[i][k] + len[k][j];
}
```

때,
위의 그림처럼 k를 거쳐서 오는 경로중에 끊긴 경로(MAX값)이 있으면 continue를 시켜줘야한다. 아니면 가중치값에 -1이 있다면 MAX-1값이 최종 출력값에 나오기 때문이다. FLOYD의 경우 음수 사이클을 확인할 때 Bellman ford때처럼 한번 더 3중 for문을 돌려 거리가 다시 갱신되면 음수사이클이 되었다고 간주하여 에러코드가 나오게 했다.

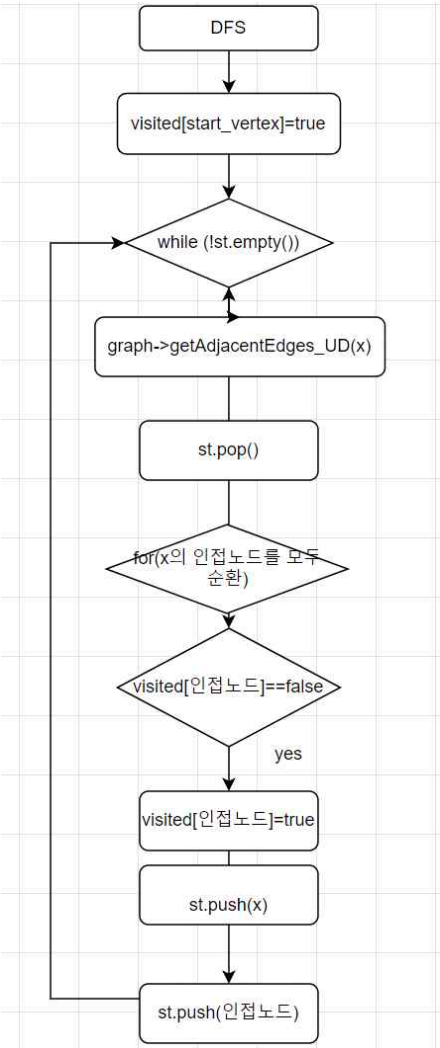
3. flow chart

(1) BFS



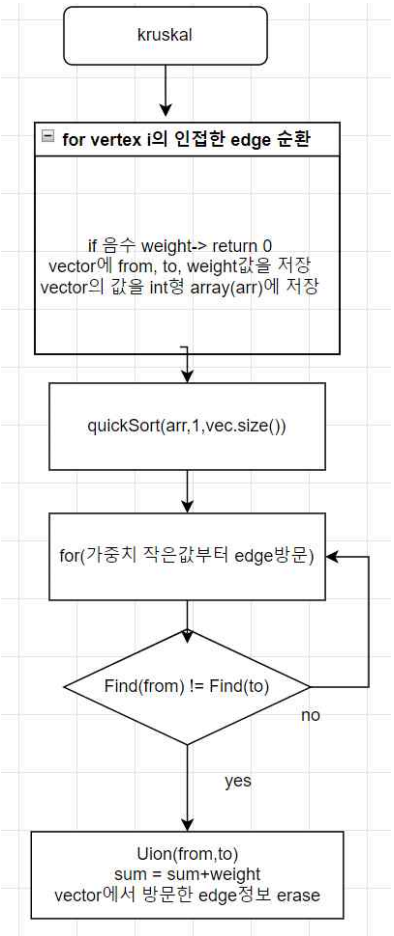
인자로 받은 start_vertex값의 방문처리를 해준 후, queue가 empty일 때까지 x의 인접노드를 구하여 map에 저장한 다음, pop과 동시에 인접노드를 저장한 map을 순회하면서 인접노드들을 모두 순환한다. 이때 인접노드의 방문처리가 false라면 아직 방문하지않았다는 뜻이므로 queue에 push 후 방문 처리해준다.

(2) DFS



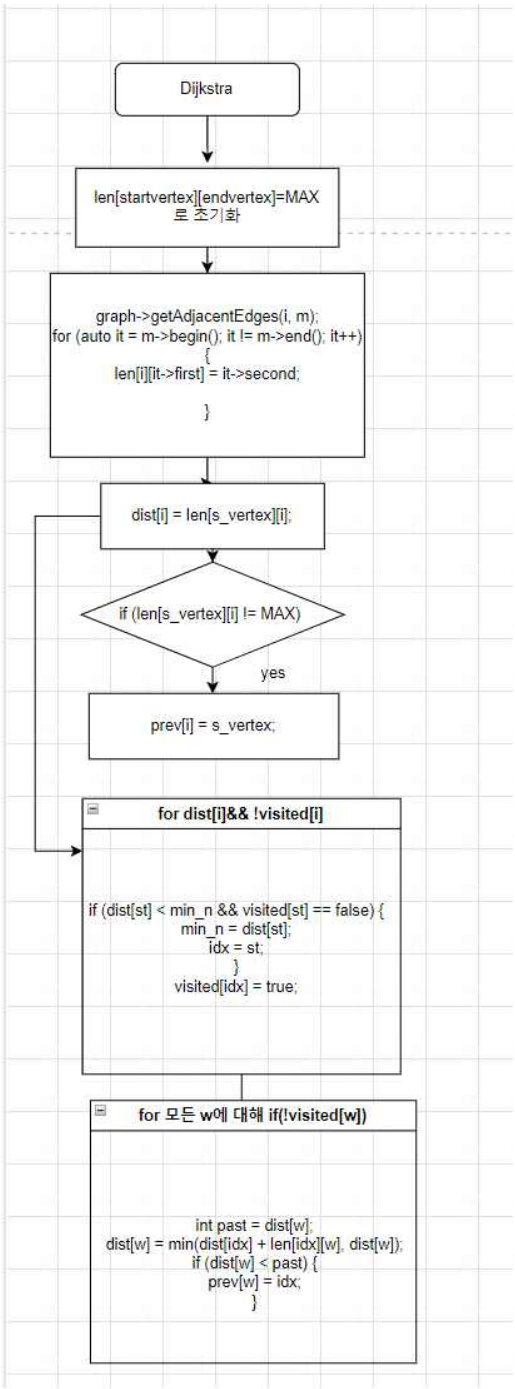
인자로 받은 start_vertex값의 방문처리를 해준 후, stack이 empty일 때까지 x의 인접노드를 구하여 map에 저장한 다음, pop과 동시에 인접노드를 저장한 map을 순회하면서 인접노드들을 모두 순환하다가 인접노드의 방문처리가 false라면 아직 방문하지않았다는 뜻이므로 queue에 push 후 break를 한다. 이것이 BFS와의 차이점이다.

(3) Kruskal



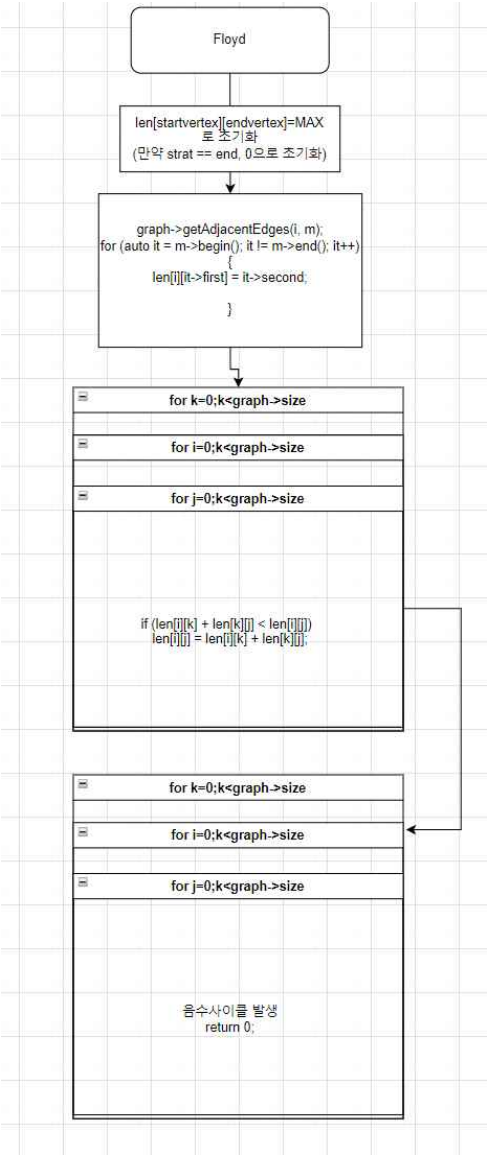
for문을 통해 vertex의 인접한 edge를 순환하다가 음수인 가중치를 발견하면 에러처리를 한다. vecotr에 from, to, weight값을 저장하고, 가중치값끼리 sorting을 위해 vecotr의 second값을 int형 array에 저장하여 quickSort함수의 인자로 넘긴다. 가중치가 작은순서대로 정렬된 arr와 가중치값과 from,to값의 정보를 저장하고 있는 vector<pair<pair<int, int>, int>> vec을 match 시키면서 둘이 같은 set에 없으면 union을 해준다. 이때 vector에서 방문한 edge는 erase시킨다.

(4) Dijkstra


Dijkstra
len[startvertex][endvertex]=MAX
로 초기화
graph->getAdjacentEdges(i, m);
for (auto it = m->begin(); it != m->end(); it++)
{
len[i][it->first] = it->second;
}
dist[i] = len[s_vertex][i];
if (len[s_vertex][i] != MAX)
yes
prev[i] = s_vertex;
for dist[i]&& !visited[i]
if (dist[st] < min_n && visited[st] == false) {
min_n = dist[st];
idx = st;
}
visited[idx] = true;
for 모든 w에 대해 if(!visited[w])
int past = dist[w];
dist[w] = min(dist[idx] + len[idx][w], dist[w]);
if (dist[w] < past) {
prev[w] = idx;
}

len[i][j]값을 Max로 다 초기화를 해준다. graph의 정보에 맞게 len[from][to] 값을 update해준다. dist[i]또한 초기화해준다. 만약 start vertex에서 I까지의 경로가 있다면 prev[i]의 값을 start vertex로 초기화해준다. dist[i]가 있고 visited[i]가 방문하지 않았다면 dist[i]끼리 비교하여 최소값을 choose하여 그를 idx값이라고 한다. 그 다음 idx값을 거쳐서 w로 가는 경로와 기존의 dist[w]의 값을 비교하여 min값을 update해준다. update된다면 prev[w]도 idx로 갱신되어야한다.

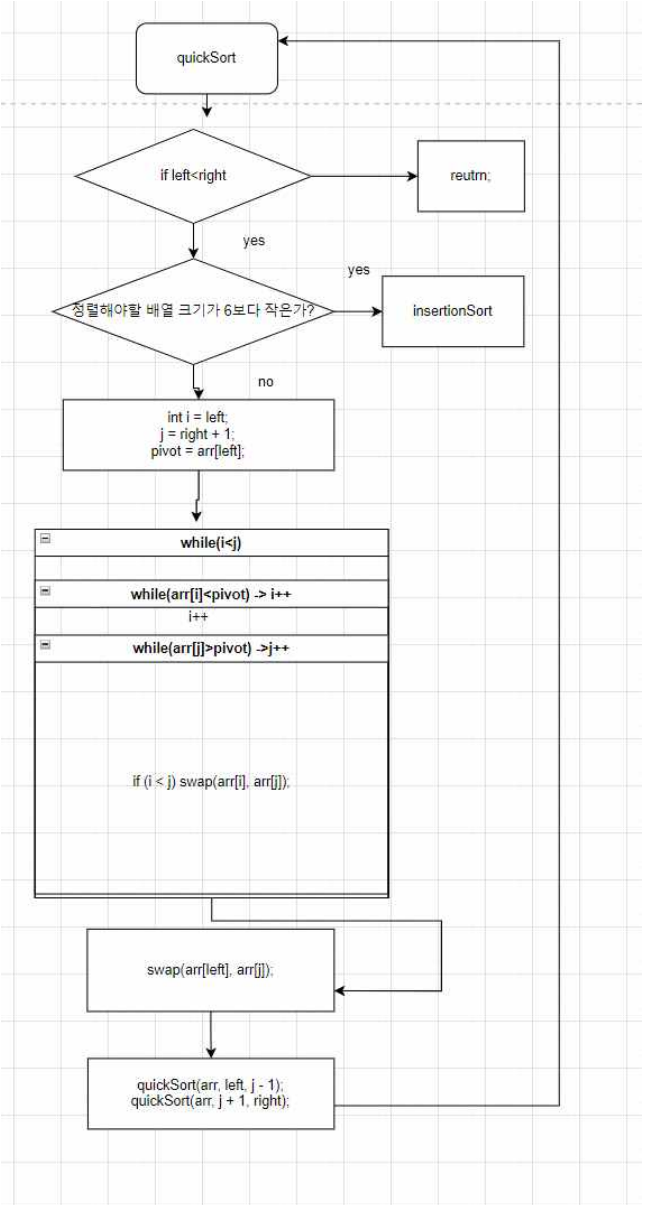
(5) Floyd



len[i][j]값을 우선 Max로 다 초기화를 해준다. graph의 정보에 맞게 len[from][to]값을 update해준다. dist[i]또한 초기화해준다. 만약 start vertex와 end vertex가 같다면 그 경로는 0으로 초기화해준다. 예를 들어,

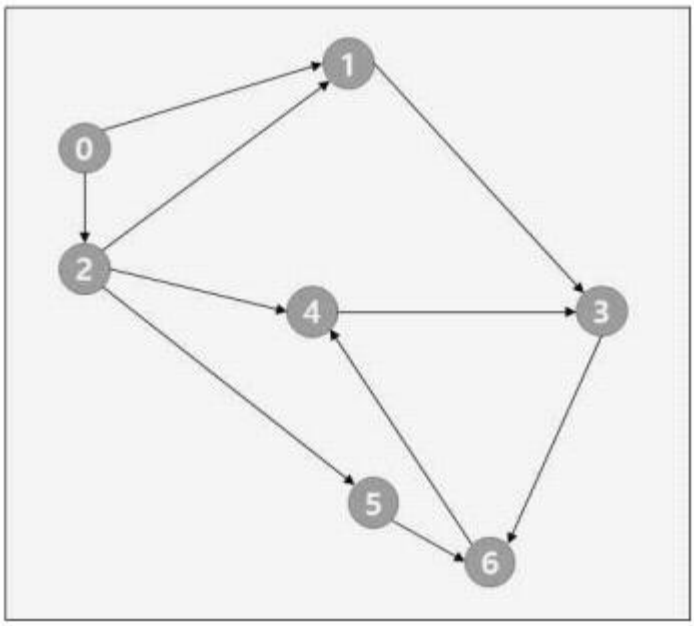
len[1][1]=0, for문을 3개(k,j,i) 돌면서 k를 거쳐 간 거리와 기존 거리를 비교하여 갱신시켜주었다. 3중 for문을 다시 한번 돌려서 음수사이클이 발생되었는지 또한 확인한다.

(6) quick sort



인자값으로 받은 left와 right값이 서로 switch가 되지 않는 동안 정렬해야할 배열의 크기에 맞게 insertion sort를 할지, quick sort를 진행할지 정한다. I는 left, j는 right을 가리키게하고, pivot은 해당 배열의 가장 왼쪽 요소로 정하여 i<j인동안 I는 점점 오른쪽으로 j는 왼쪽으로 가면서 pivot보다 작은수나 큰수를 조건에 맞게 swap한다. pivot을 기준으로 왼쪽 segment는 작은 수, 오른쪽 segment는 큰 수가 나열되는데, pivot기준 또 왼쪽과 오른쪽 배열로 나누어 quickSort를 재귀적으로 호출하여 나머지도 정렬해준다.

4. Result Screen



command.txt는 다음과 같다.
LOAD graph_L.txt
PRINT
BFS 3
DFS 2
DFS_R 2
KRUSKAL
DIJKSTRA 0
BELLMANFORD 0 6
FLOYD
EXIT

(1)LOAD 및 PRINT

```
========LOAD========
Success
====================
==========PRINT==========
[0] -> (1,6) -> (2,2)
[1] -> (3,5)
[2] -> (1,7) -> (4,3) -> (5,8)
[3] -> (6,3)
[4] -> (3,4)
[5] -> (6,1)
[6] -> (4,10)

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

(2)BFS, DFS, DFS_R

```
======BFS=======
startvertex : 3
3 -> 1 -> 4 -> 6 -> 0 -> 2 -> 5
=============
======DFS=======
startvertex : 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
=============
======DFS_R=======
startvertex : 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
=============
```

(3) KRUSKAL

```
=============
======Kruskal=======
[0] 2(2)
[1] 3(5)
[2] 0(2) 4(3)
[3] 1(5) 4(4) 6(3)
[4] 2(3) 3(4)
[5] 6(1)
[6] 3(3) 5(1)
cost:18
===============
```

(4) DIJKSTRA

```
======Dijkstra=======
startvertex : 0
[1] 0->1(6)
[2] 0->2(2)
[3] 0->2->4->3(9)
[4] 0->2->4(5)
[5] 0->2->5(10)
[6] 0->2->5->6(11)
==================
```

다익스트라의 경우 위 그래프를 바탕으로 최단 경로를 구하는 것인데, 결과가

잘 출력되는 것을 볼 수 있다.

(5) BELLMANFORD

```
======Bellman-Ford=======
0->2->5->6
cost :11
====================
```

(6) FLOYD

```
--------------------
======FLOYD=======
        [0]     [1]     [2]     [3]     [4]     [5]     [6]
[0]     0       6       2       9       5       10      11
[1]     x       0       x       5       18      x       8
[2]     x       7       0       7       3       8       9
[3]     x       x       x       0       13      x       3
[4]     x       x       x       4       0       x       7
[5]     x       x       x       15      11      0       1
[6]     x       x       x       14      10      x       0
=================
```

프로젝트3 제안서 pdf와 완전히 같은 예시로 다익스트라 외 모든 결과 값이 조교님 파일과 동일한 결과를 보인다.
floyd의 부분에서 [1][1]로 가는 경우 0으로 출력하는 것이 맞으며, [1][2]의 경우도 edge가 없는 경우이므로, x로 출력되는 것이 맞다.

(7) graph_M.txt를 읽었을 때

```
========LOAD========
Success
====================
===========PRINT==========
        [0]     [1]     [2]     [3]     [4]     [5]     [6]
[0]     0       6       2       0       0       0       0
[1]     0       0       0       5       0       0       0
[2]     0       7       0       0       3       8       0
[3]     0       0       0       0       0       0       3
[4]     0       0       0       4       0       0       0
[5]     0       0       0       0       0       0       1
[6]     0       0       0       0       10      0       0
=============
```

```
------------
======BFS=======
startvertex : 3
3 -> 1 -> 4 -> 6 -> 0 -> 2 -> 5
=============
======DFS=======
startvertex : 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
=============
======DFS_R=======
startvertex : 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
=============
======Kruskal=======
[0] 2(2)
[1] 3(5)
[2] 0(2) 4(3)
[3] 1(5) 4(4) 6(3)
[4] 2(3) 3(4)
[5] 6(1)
[6] 3(3) 5(1)
cost:18
===============
======Dijkstra=======
startvertex : 5
[0] x
[1] x
[2] x
[3] 5->6->4->3(15)
[4] 5->6->4(11)
[6] 5->6(1)
=================
```

```
======Bellman-Ford=======
0->2->5->6
cost :11
=====================
======FLOYD=======
        [0]     [1]     [2]     [3]     [4]     [5]     [6]
[0]     0       6       2       9       5       10      11
[1]     x       0       x       5       18      x       8
[2]     x       7       0       7       3       8       9
[3]     x       x       x       0       13      x       3
[4]     x       x       x       4       0       x       7
[5]     x       x       x       15      11      0       1
[6]     x       x       x       14      10      x       0
=================
```

이 또한 그래프 형태가 matrix로 바뀐 것뿐이지 명령어는 프로젝트 제안서에 나와있는 것과 동일하므로 제안서 pdf파일 결과와 동일하게 결과가 나오는 것을 확인할 수 있다.

5. Consideration

DFS_R부분에서 cout으로 출력할 때는 count값이 0에서 재귀함수를 호출할 때마다 증가하게 만들어서 count가 그래프의 사이즈값과 동일해지면 "->"을 출력하지 않도록 구현해주었는데 ofstream flog1으로 cout을 대체해주면서 cout값이 0부터 초기화 해줘도 재귀함수 내에서 그래프사이즈부터 count값이 줄어들면서 결론적으로 0까지 줄어드는 것을 확인하였다. 따라서 거꾸로 출력해주기 위해 stack을 이용해주었는데 출력방식이 바뀌지 않아 큐형태로 값을 저장하고 마지막에 큐에서 원소들을 pop하면서 file에 출력해주었더니 원하는 대로 출력할 수 있었다. cout으로는 잘 되던 과정이 filestream을 쓰면서 이상해져서 출력에 맞게 코드를 짰다. 이 부분은 아직도 왜 queue를 써서 출력해줘야하는지, count값이 거꾸로 줄어드는지 모르겠다.

비방향성 그래프에서 인접 노드를 map에 저장할 때 만약 두 노드가 서로를 가리키는 arc가 있다면, 둘 중에 더 작은 가중치값만 비방향성 선분으로 남겨야하는데 이 부분을 for문 네 개를 이용하여 구하니까 계속 update를 해주지 않고, 마지막값만 저장하여 반영하는 것을 확인하였다. 이를 고치기 위해 for문 두 개만을 이용하여 바로바로 가중치값을 비교하여 더 작은 값이 map에 insert되도록 고쳐주었다.

pdf예시에서 0->6(5) , 6->0(4)부분을 넣어줬는데 출력할 때 무한반복하는 것을 확인하였다. 이는 startvertex가 0일 때 오류가 났는데 그 이유가 start vertex로 들어오는 edge가 있으면 start vertex의 prev가 즉, prev[0]=6으로 바뀌기 때문에 prev를 순환할 때 0과 6을 반복해서 출력하는 것이다.(while문 탈출 불가) 따라서 start vertex로 들어오는 edge가 있다면 prev[startvertex]는 갱신을 무시하도록 코드를 고쳐주었다.

다익스트라 부분도 아직 방문하지 않은 노드까지의 가장 작은 거리 값을 choose하는 부분에서 맨처음 min변수를 dist[0]으로 초기화해버리니까, 이미 0번째 노드는 방문했음에도 불구하고, dist[0]보다 작은 거리가 더 이상 없는 경우 0은 이미 방문했으니까 0을 제외하고 min값을 계속 갱신시키지 못하고, 계속 dist[0]을 min값으로 가지는 것을 볼 수 있었다. 그래서 다익스트라 최단경로를 잘못 구하는 것을 확인하였다. min변수의 초기화 부분을 고쳐주었더니

이 부분의 오류를 해결할 수 있었다.
이번에는 수업시간에 배운 알고리즘을 직접 구현해보고 결과를 확인할 수 있어서 흥미로웠다. 자잘자잘한 에러처리나, 예외상황에 에러가 나는 것을 고쳐줘야하는 부분이 의외로 많았지만 2차 프로젝트에서 경험했던 map이나 vector를 능숙하게 사용하여 코드를 구현함으로써 성장함을 느꼈다.