



# Data structure

## Project 3

Professor	이기훈 교수님
Department	Computer engineering
Student ID	2013722082 이형민
	2013722068 김동현
Class	2013722080 김동민
Date	2016. 12. 16

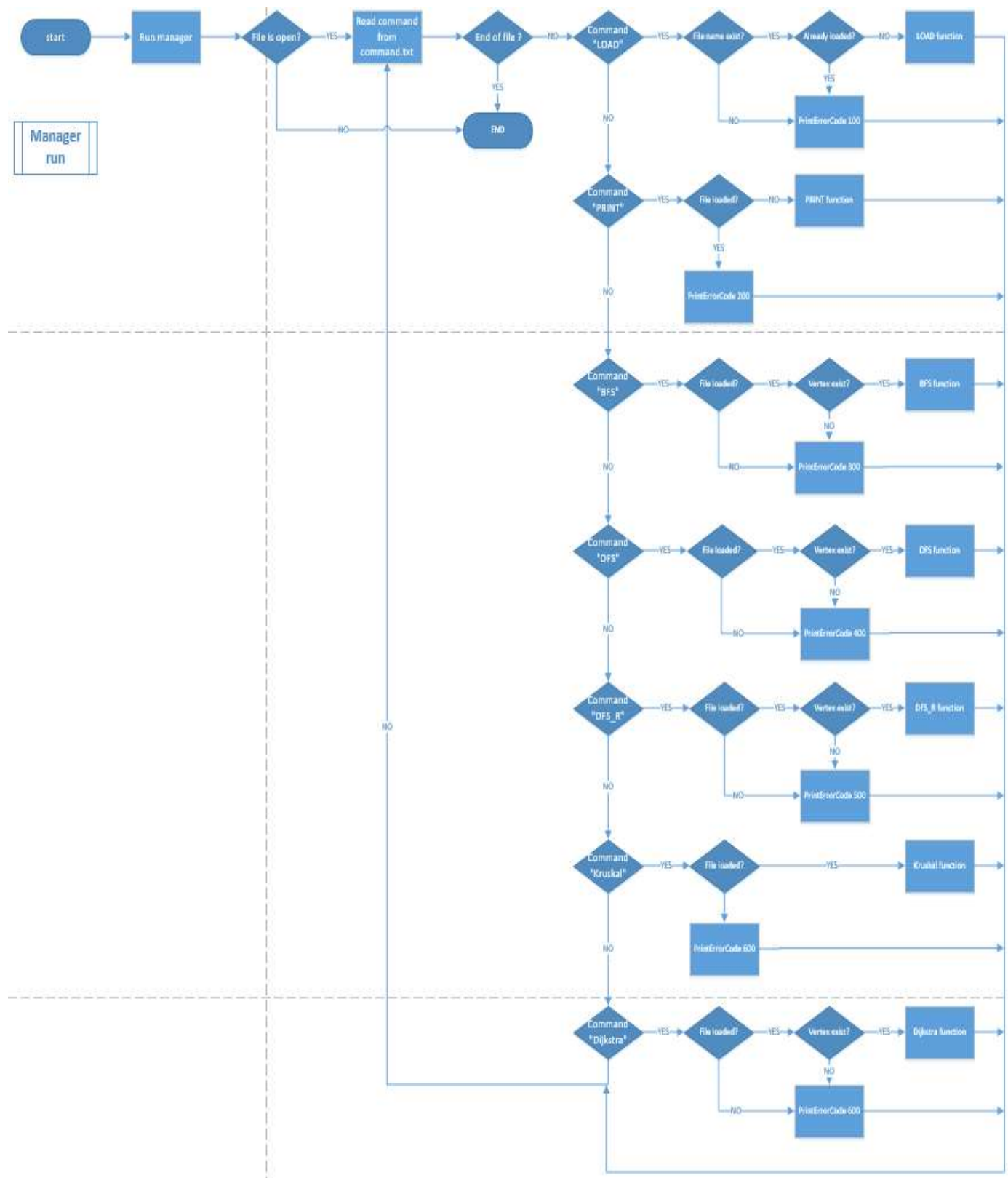


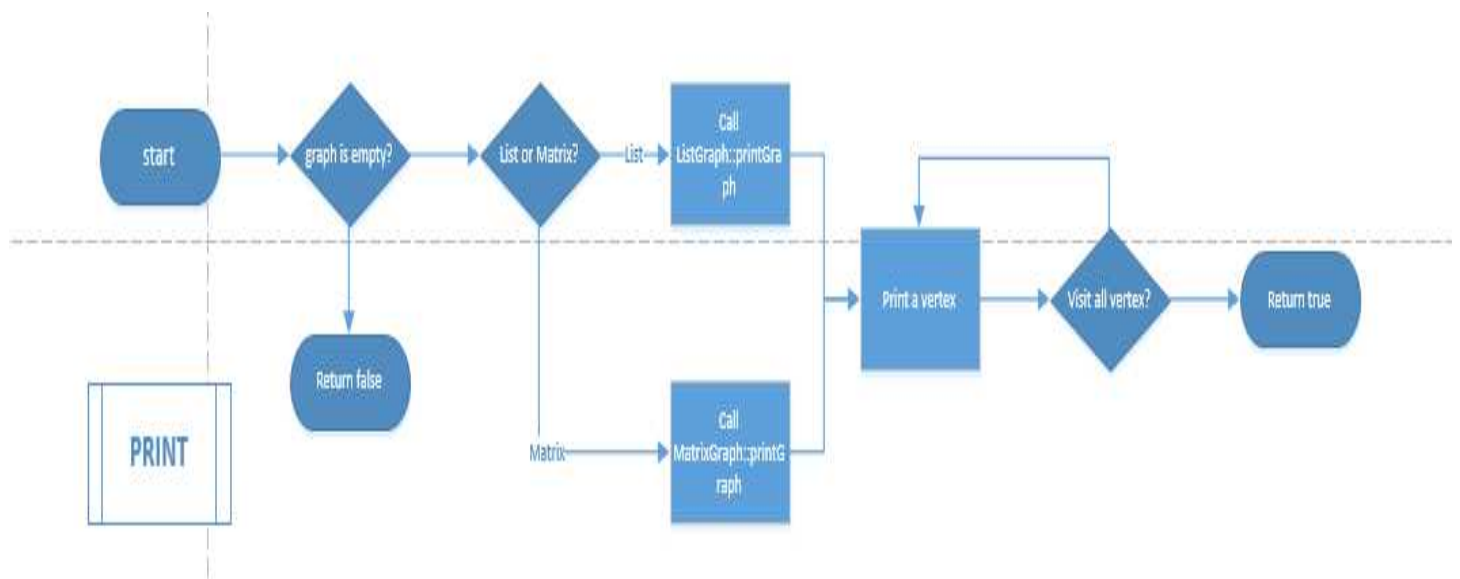
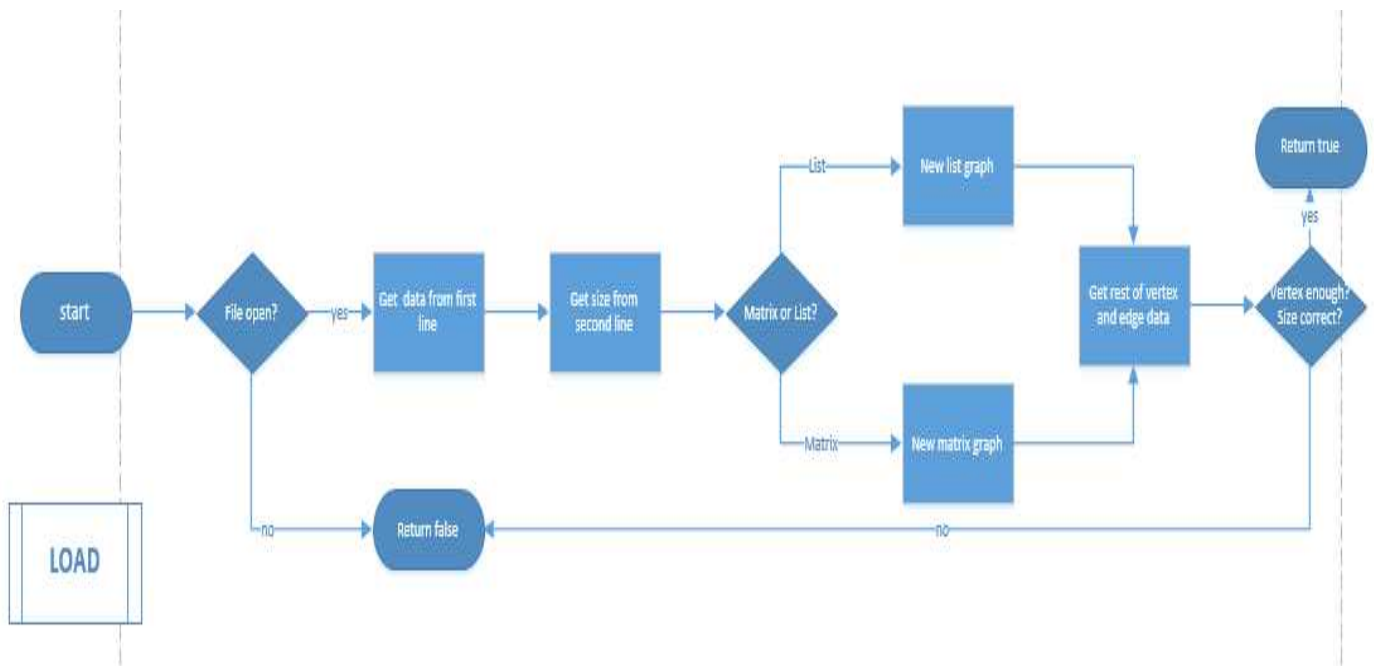
## Introduction

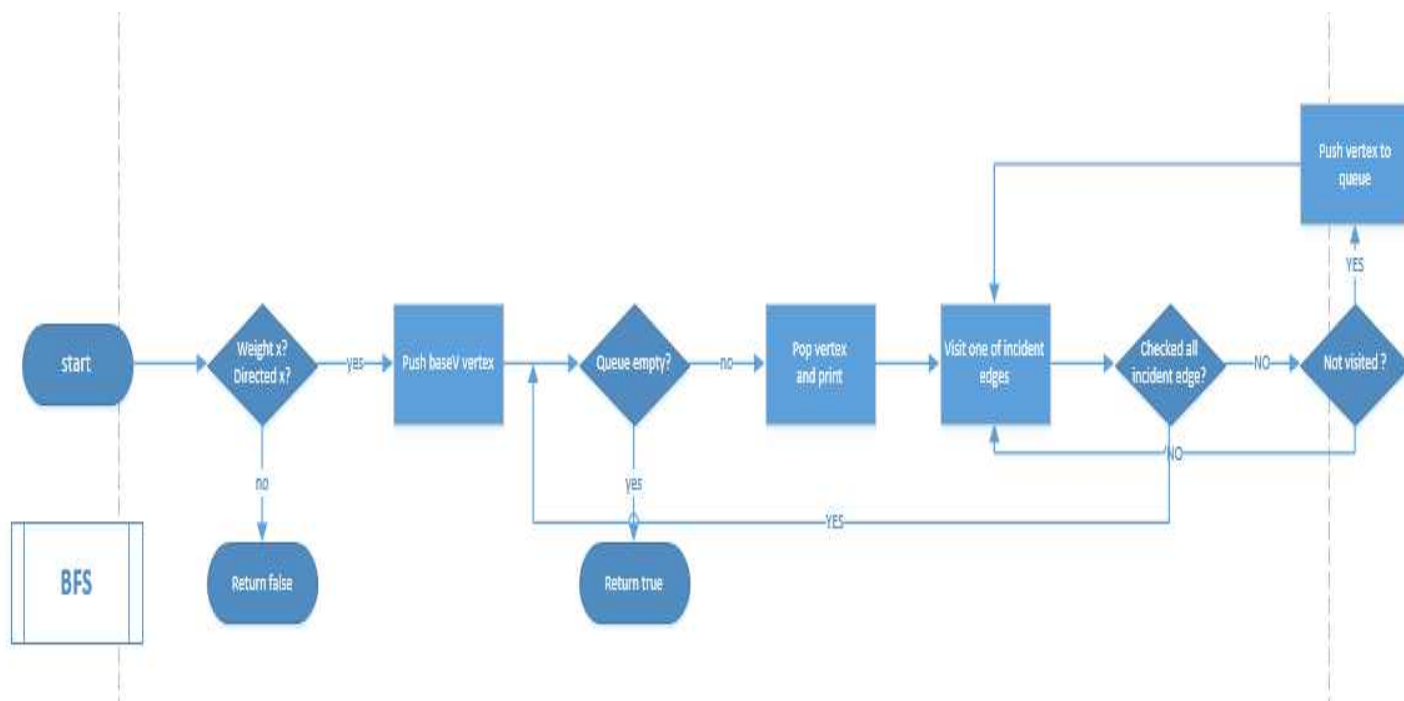
기존까지는 tree 형태의 cycle 이 없는 tree 형태의 자료구조에 대한 연산 알고리즘을 구현했다면 이번 프로젝트는 graph를 이용한 연산 알고리즘을 수행하는 프로그램을 구현하는 것이다. 기존의 프로젝트들과 같이 text 파일에서 graph 정보를 읽어와서 해당 command 에 맞게 연산을 수행하는 프로그램을 구현하는 것이다. 이때 graph는 list 형태이거나 matrix 형태이고 첫째 줄과 둘째 줄에 해당 graph가 어떤 형태인지, 방향성이 있는지, 가중치가 있는지 vertex가 몇 개 인지에 대한 정보를 주고 그 아래에 graph의 edge 들에 대한 정보들이 담겨 있다.

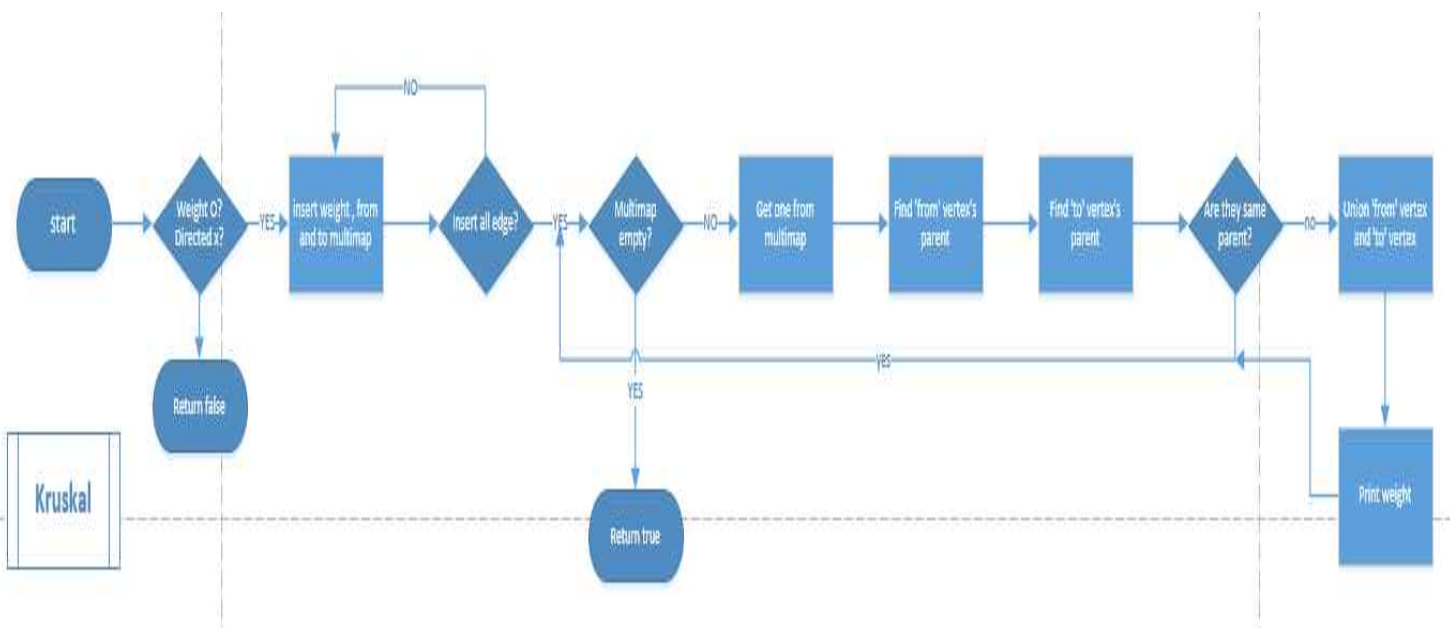
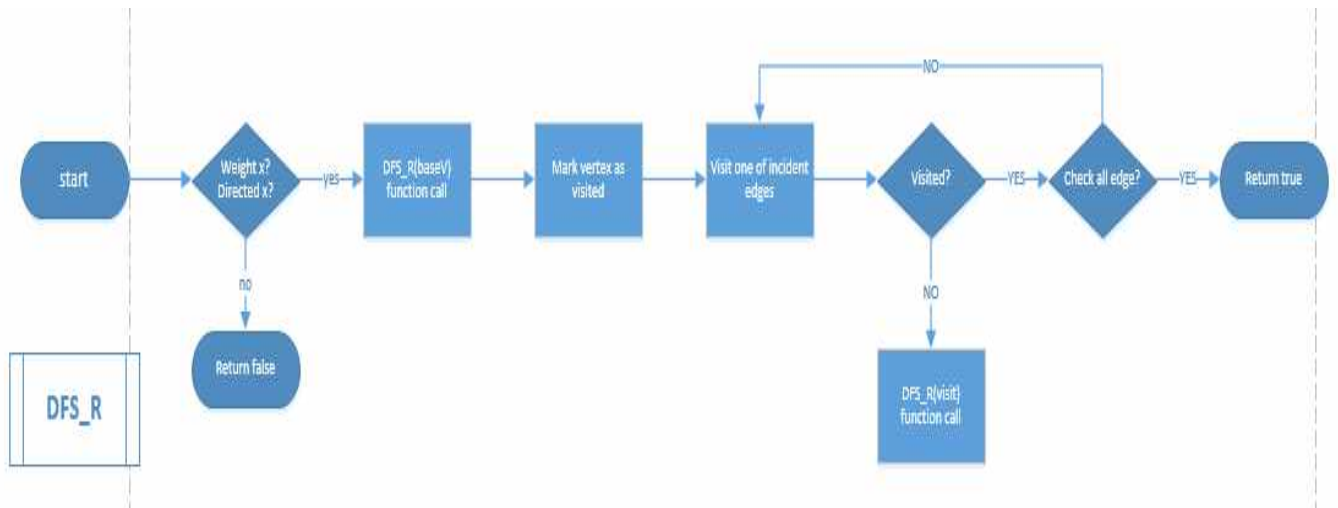
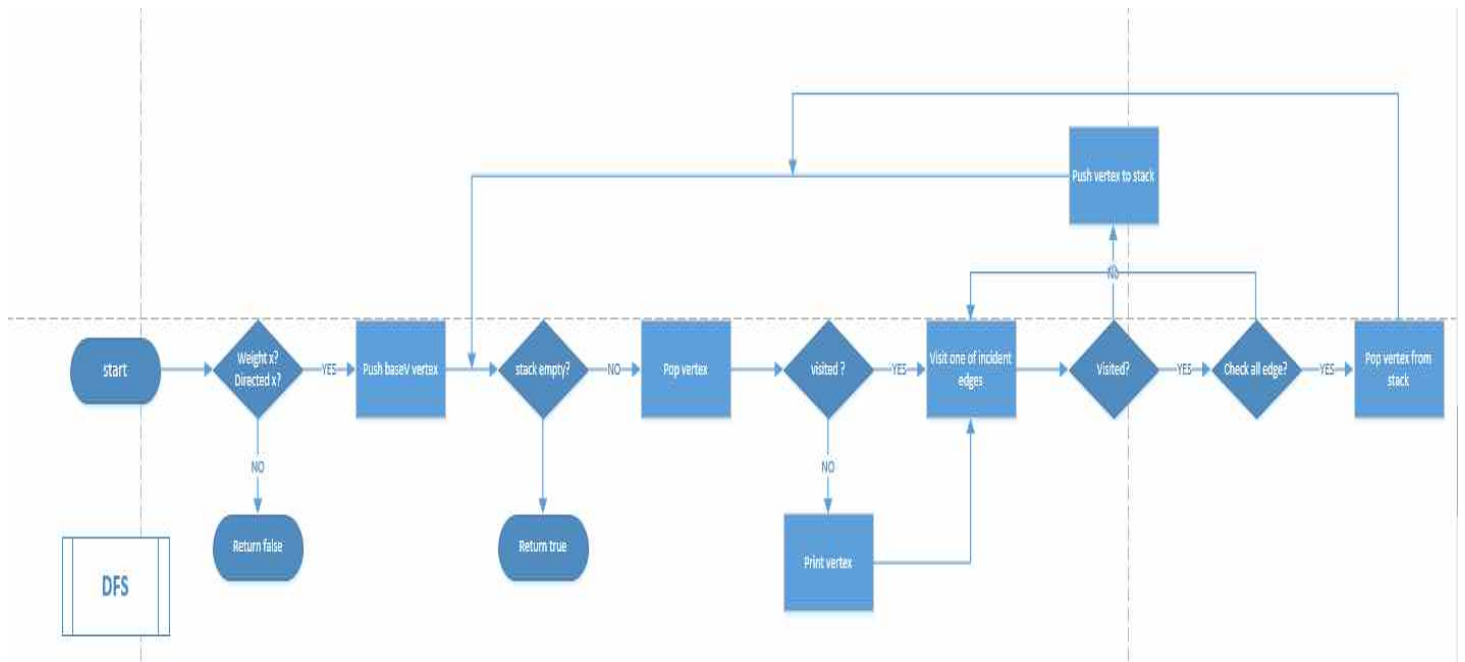
텍스트 파일에서 성공적으로 그래프를 읽어오면 각 command 에 맞는 알고리즘 연산을 수행한다. 이때 BFS DFS DFS\_R은 방향성과 가중치가 모두 없는 그래프 환경에서 수행되어야 한다. Kruskal 알고리즘은 최소 비용 신장 트리를 만드는 방법으로 방향성이 없고 가중치가 있는 그래프 환경에서 수행되어야 한다. Dijkstra 알고리즘은 정점 하나를 출발점으로 받고 다른 모든 정점을 도착점으로 하는 최단 경로 알고리즘으로 방향성과 가중치가 모두 있는 그래프 환경에서만 연산이 수행되어야 한다.

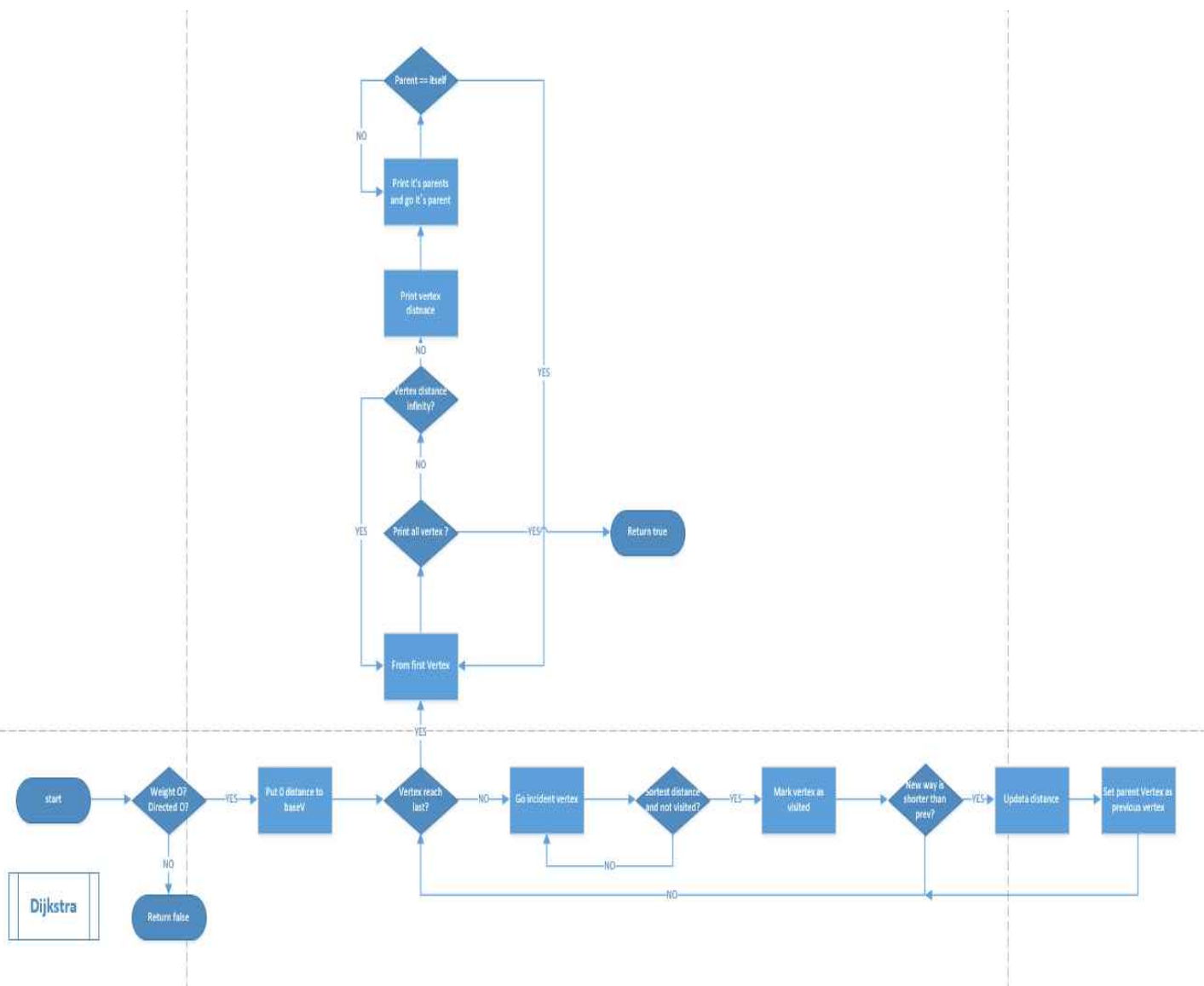
## Flow chart











## Algorithm

### BFS

BFS는 너비우선 탐색으로서 방향성과 무게가 없으며 현재 vertex에서 연결된 모든 vertex를 방문, 모두 방문하면 연결된 vertex들의 또 다른 연결된 vertex를 방문하며 진행합니다. BFS를 구현하는데 Queue를 사용하였습니다.

처음 방문한 vertex를 확인할 visit 배열을 graph size 만큼 동적할당 하였고 0으로 초기화하였습니다. vertex를 탐색하는데 사용할 Queue q, 그리고 vertex의 인접행렬 정보를 받아올 map p를 선언하였습니다.

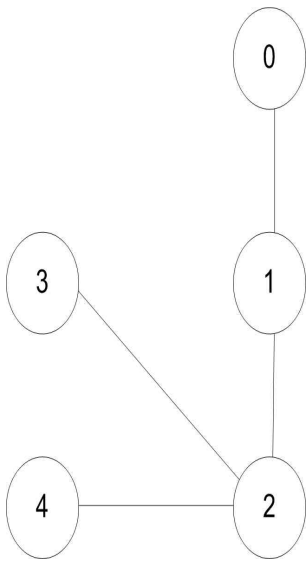
우선 반복문 while(1) 하여 무한반복 하도록 설정하였습니다.

그다음 getIncidentEdges(baseV, &p)를 통해 baseV의 인접행렬 정보를 p에 저장하였습니다. 이것으로 baseV를



방문한것이므로  $visit[baseV] = 1$  로 하였습니다. 방문한  $baseV$ 를 출력. 다음 반복문 `while` 안에서 `map p`의 `iterater`가 `begin`에서 `end`까지 반복하도록 설정하였으며 그 안에  $visit[iterator->first] != 1$ 이면 `q.push(iterator->first)` 하도록 하였습니다.

`iterator first`에는  $baseV$ 의 인접 `vertex` 정보가 있으므로 아직 방문하지 않았다면 `queue q`에 삽입 될것입니다. 다음 `while`문을 나와 무한반복문에서 `map p.clear()` 하여 `map`을 비우고, `queue q`가 비어있다면 `break` 하여 무한반복문 탈출, 아니라면  $baseV = q.front$ 하여  $baseV$ 를 `queue`의 첫 번째로 바꾸었습니다. 그다음 `queue pop`하여 현재의  $baseV$ 를 `queue`에서 빼고 무한반복문을 닫습니다.



그래프가 다음과 같고 모든 `weight`를 1이라하면 `getincident(baseV, &p)` 선언 이후 `map` 안에는 (1,1) (3,1) (4,1) 순으로 정보가 저장되어있으며  $visit[2] = 1$ 입니다. 그리고  $baseV(2)$  출력.

다음 `queue` 에는 1,3,4 순으로 정보가 저장됩니다. `map p.clear`이후  $baseV$ 는 `q.front`이므로  $baseV = 1$ . `q.pop` 하여 `q`에는 3,4 저장된상태.

그다음 다시 `for`문을 반복하여 `getIncident(1,&p)`하면

`map p` 안에는 (0,1) (2,1) 저장.

1을 방문하였으므로  $visit[1] = 1$  , 1출력.

$visit[0]=0$  이므로 `q.push(0)`.

$visit[2] = 1$  이므로 `q.push`하지 않습니다.

현재 `queue q` 안에는 (3,1)(4,1)(0,1) 이 저장되었습니다. 동일한 방법으로 다음  $baseV$ 는 3이 되고 연결된 것은 `vertex 2`지만 2는 방문하였으므로 `q`에 `qush` 되지 않으며  $baseV$ 는 4가 되고 4또한 2와 연결되 있으나 2는 `push`되지 않습니다. 마지막으로 0은 1과 연결되어 있으나 1또한 이미 방문하였으므로 `queue`에 들어가지 않습니다. 그리고 `q`가 비어있으므로 반복문을 탈출합니다.

BFS를 구현하여  $baseV$ 가 2일 때 출력되는 순서는 2, 1, 3, 4, 0입니다.



## Dijkstra

다익스트라는 방향성과 무게가 있는 알고리즘으로서 graph에서 입력한 vertex baseV에서 각 지점으로 갈 때의 shortest path 즉, 최단거리를 구하는 알고리즘입니다.

또한 이번 프로젝트에서는 해당 vertex로부터 기준 vertex base 까지 역순으로 출력하며, 만약 도달할수 없는 경우 X를 출력합니다.

방문할 vertex를 표시할 배열 visit를 graph size N 만큼 동적할당하고 0으로 초기화합니다.

baseV에서 해당 vertex까지 도달하는 거리 distance 배열을 N만큼 동적할당하고 충분히 큰수 100000000으로 초기화합니다.

다음 가는 경로를 저장할 2차원 배열 arr[i][j]를 N\*N의 형태로 동적할당 후 -1로 초기화 합니다.

for(int c=0; c < graph size ; c++) 하여 graph size만큼 반복하도록 합니다.

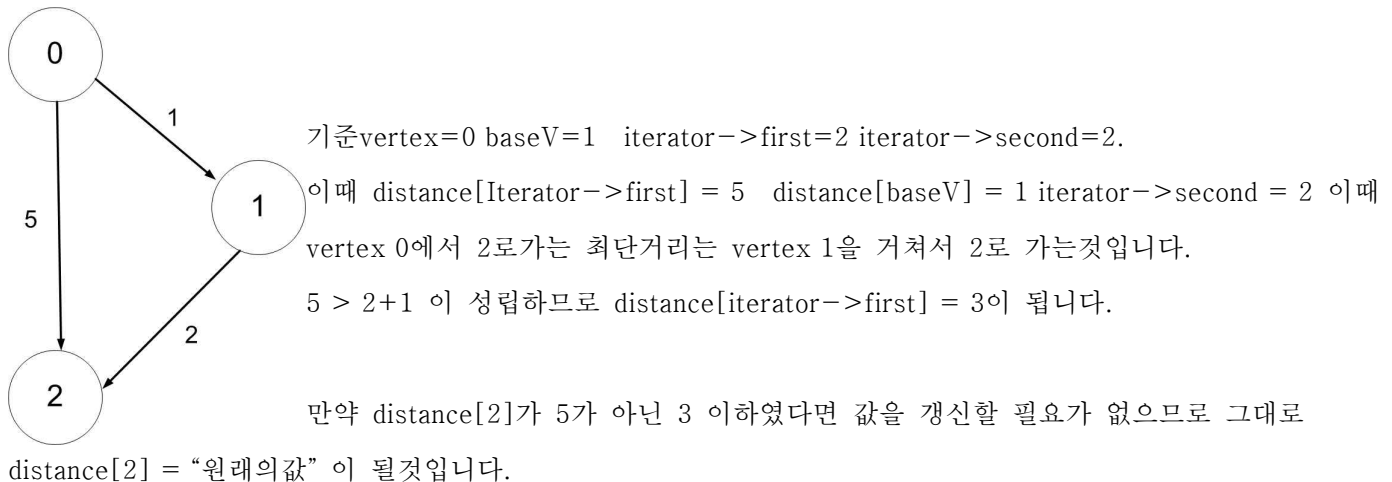
그다음 getincidentEdges(baseV, &p) 하여 baseV의 인접행렬 정보를 map p에 저장합니다.

다음 iterator을 map p begin으로 설정 후 iterator가 p.end 일때까지 반복문 while을 설정합니다.

그 안에 if(distance[Iterator->first] > distance[baseV] + iterator->second)

distance[iterator->first] = distance[baseV] + iterator->second

이것이 의미하는 것은 다음과 같습니다.



그리고 배열에 경로를 저장하기 위해 arr[i][j]에서 I는 from j는 to 라 할 때  
값이 갱신 될 때만

```

for(int k=0;k <graph->getSize();k++){
    if(arr[baseV][k]!= -1)
        arr[Index->first][k] = arr[baseV][k];
    else{
        arr[Index->first][k] = baseV;
        break;}
} Index++;

```

합니다.

밖의 for 문을 반복하면서 arr[A][k]에서 A는 도착 vertex까지의 최단 경로가 저장됩니다.

그다음 iterator++ 후 while문을 닫습니다.

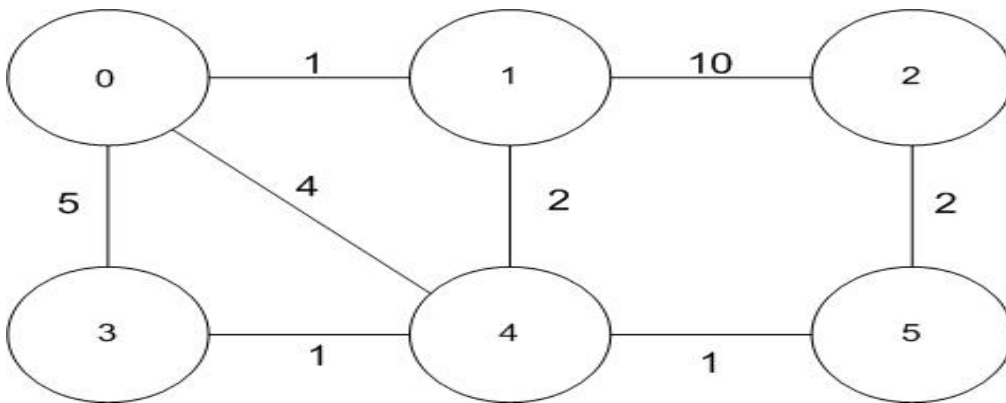
baseV를 방문한 것이므로 visit[baseV] = 1로 설정합니다.

그리고 다음 baseV를 설정하기 위해 이미 방문한 vertex를 제외하고 distance[vertex] 값이 가장 작은 vertex를 baseV로 설정합니다.

distance가 가장 작은 것을 택하는 이유는 다른 vertex를 먼저 하게 되면 뒤에는 다시 값을 갱신해야 하기 때문입니다.

```
baseV = k {(min =distance[k]) && (visit[k] == -1)}
```

따라서 baseV를 다음과 같이 재설정 한후 반복문을 종료합니다.



다음 그래프에서 baseV가 0이고 각 vertex까지의 최단거리를 구할 때

방문한 vertex를 V , 방문하지 않은 vertex는 Q라하고 최단거리를 d[vertex]라 할 때

처음 baseV를 방문하면  $V\{0\}$   $Q\{1,2,3,4,5\}$   $d[1]=1$   $d[2]=\text{inf}$   $d[3]=5$   $d[4]=4$   $d[5]=\text{inf}$ 입니다.

그 다음 방문하지 않은 노드 중 가장 작은 distance를 갖는 것은 1이므로 baseV는 1이 됩니다. 이때 현재의  $d[4]$ 는 4인데 0에서 1을 거쳐 4로갈 때 weight가 3이므로

$V\{0,1\}$   $Q\{2,3,4,5\}$   $d[1]=1$   $d[2]=11$   $d[3]=5$   $d[4]=3$   $d[5]=\text{inf}$ 입니다.

그다음 baseV는 4가 되고  $V\{0,1,4\}$   $Q\{2,3,5\}$  distance는 순서대로  $\{1/11/4/3/4\}$ 입니다.

계속해서 baseV가 3으로 5로 2로 바뀌는데 3에서 5로 바뀔 때 distance 2가 0,4,5,2를 거쳐가는 것이 6으로 원래의 11보다 weight가 더 작으므로  $\text{distance}[2]=6$ 으로 갱신됩니다.

그 이외의 변화는 없습니다.

2차원 동적할당 배열 `for(j = size-1; j>=0 ; j++)`

```
if(arr[i][j]!=1)
```

```
cout<<arr[i][j];
```

하면 0에서 vertex I까지의 경로가 역순으로 출력됩니다.

만약 `visit[i]`가 0이라면 vertex 0과 연결되어 있지 않은것이므로 X를 출력합니다.

## Matrix

매트릭스 그래프는 graph 의 vertex가 N개일 때  $N*N$ 의 2차원 배열을 선언해 `arr[i][j]`에서 I는 fromVertex j는 toVertex, 그때의 `arr[i][j]`는 weight의 값을 저장하는 배열입니다.

따라서 Matrix graph의 생성자에서 `m_Mat`는 2차원 배열 동적할당을 하였고 모든 배열의 값을 0으로 초기화 하였습니다.

Matrix graph의 소멸자에서는 동적할당 하였던 `m_Mat`를 동적할당 해제를 하였습니다.

`getIncidentEdges(int vertex, map<int,int>*m)`이란 vertex에서 연결되어 있는 인접행렬들의 정보를 맵에 저장하는데 이때 first에는 toVertex, second 에는 weight를 저장하므로 `m->clear()`

```
for(int a=0; i< graphSize; a++)
```

```
if(m_Mat[vertex][a] !=0)
```

```
m->insert(make_pair(a, m_Mat[vertex][a]))
```

을 구현하여 인접행렬의 정보를 map에 저장하였습니다.

처음 map을 clear하는 이유는 이전 vertex 인접행렬정보가 들어있을수 있기 때문입니다.

`InsertEdge(int from, int to, int weight)`는 `m_Mat`의 배열에 text파일에서 읽어온 값들을 저장할 때 사용합니다.

따라서 `m_Mat[from][to] = weight`.

`printGraph`는 matrix형식의  $N*N$ 의 꼴로 각 가로 세로 줄의 from, to에 맞춰 weight를 출력하는 것입니다.

연결되지 않은곳은 0으로 초기화 되어 있으므로 모두 출력합니다.

```

따라서 for(int I=0;i<graphsize; I++)
        for(int j =0; j<graphsize; j++)
            cout<<m_Mat[i][j]

```

하여 모든 배열의 weight값을 출력하였습니다.

## vertexSet

### Find

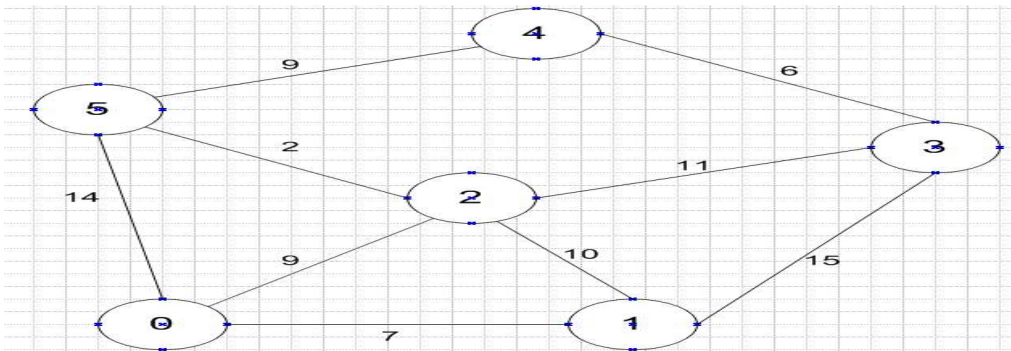
vertexSet class에서의 Find함수는 graphMethod에서 Kruskal 알고리즘을 구현하는데 있어서 사용되어집니다. Find함수의 역할은 전달 받은 vertex의 연결되어있는 부모 vertex를 반환해주는 역할을 실시하게 됩니다. 이와 같은 역할을 구현하기 위해서 vertexSet class에서 기본적으로 선언되었던 m\_Parent를 생성자에서 graph의 사이즈 만큼 동적할당을 받은 배열로 만들고 배열의 각값을 -1로 초기화 시켰습니다. Find함수에서는 전달 받은 vertex 최상위 부모 노드를 반환 해야 하므로 m\_Parent[ver]의 값이 음수를 가질 때까지 계속해서 반복문을 돌며 탐색하다가 m\_Parent[ver]의 값이 음수를 가질때의 ver의 값을 반환 하도록 설계했습니다.(m\_Parent를 -1로 초기화 했으므로 부모vertex의 m\_Parent[vert] 값은 음수이기 때문입니다.)

### Union

vertexSet class의 Union함수의 경우도 Find 함수와 마찬가지로 Kruskal 알고리즘을 구현하는데 있어서 꼭 필요한 함수입니다. Kruskal 알고리즘을 구현하는데 있어서 꼭 필요한 vertex끼리의 불필요한 사이클을 판단하고 방지하는 역할을 수행하면서 이어주는 역할을 같이 수행해 주게됩니다. Union함수의 기본 원리는 Find함수와 상당히 유사합니다. 먼저 전달받은 v1, v2 수를 각각 Find함수의 넣은 것처럼 v1, v2의 부모의 값을 v1, v2의 값으로 만들어 준 뒤 m\_Parent[v1] = v2; 표현을 사용하여 2개의 vertex를 이어주는 역할을 해주므로써 Kruskal알고리즘이 정상적으로 동작할 수 있게 해줍니다.

### DFS

DFS함수의 경우에는 깊이 우선 탐색 알고리즘에 해당하는 graph Method입니다.



해당 그래프에 대한 vertex 0을 기준으로 DFS알고리즘을 적용한다고 할 경우

0 -> 1 -> 2 -> 3-> 4-> 5 와 같이 기준 vertex를 기준으로 방문하지 않은 veretex중에서 가장 작은

vertex부터 이동을 실시하면서 탐색을 하게 되는 알고리즘을 DFS알고리즘이라고 합니다. 먼저 각 vertex에 대한 방문을 표시할 visit배열을 graph의 size만큼 동적할당 받은뒤 0의 값으로 초기화를 실시합니다. stack을 사용한 DFS알고리즘의 구성이므로 사용할 stack을 미리 선언해준 뒤 넘겨받은 기준 vertex인 baseV의 값을 stack에 push 하고 반복문에 들어가게 됩니다. stack이 비어 있지 않을 경우 getIncidentEdges함수를 이용하여 baseV vertex에 연결된 edge들을 m2로 불러온 뒤 baseV의 값이 방문했던 vertex의 값이 아니라면 baseV를 출력하고 방문표시를 실시합니다. 그 뒤에 for문을 사용하여 불러온 m2에 대하여 baseV와 연결된 edge들을 돌면서 만약 방문하지 않은 vertex가 있다면 그 즉시 stack에서 s.push(it->first)표현을 사용하여 stack에 방문하지 않은 vertex를 넣게 되고 반복문을 탈출하게 됩니다. 만약 iterator가 m2의 끝까지 돌아서 it == m2.end()에 도달했다면 stack에서 pop을 실시하게 됩니다. 그리고 마지막으로 stack이 비지 않았으면 baseV = s.top()이라는 표현을 사용하여 baseV를 stack에 저장되었던 값으로 최신화 해준뒤 다시 반복문을 도는 것을 이용하여 DFS알고리즘을 구현했습니다.

## DFS\_R

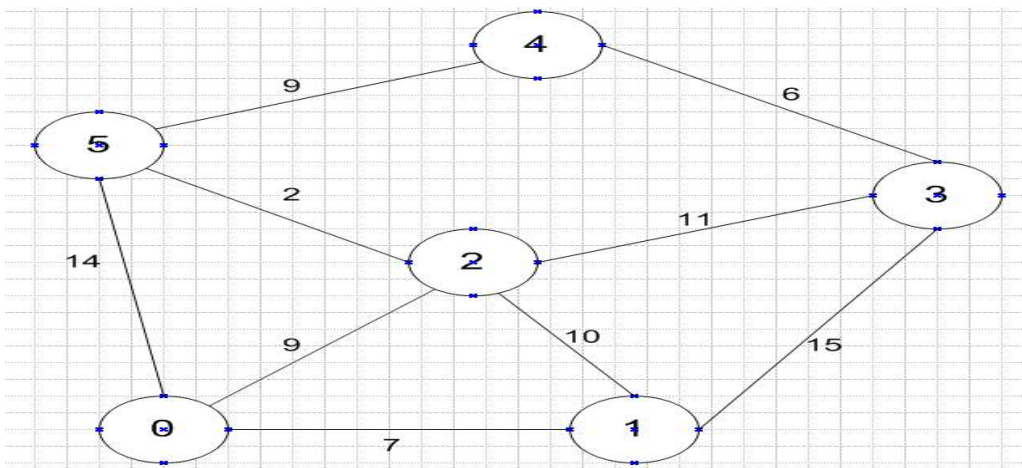
DFS와 마찬가지로 깊이 우선탐색알고리즘이지만 stack을 사용하여 구현하는 것이 아니라 재귀를 이용한 알고리즘으로 깊이 우선탐색 알고리즘을 구현하는 방법을 사용하여 알고리즘을 구현했습니다. 재귀를 통한 알고리즘의 구현이므로 먼저 넘겨받은 baseV 값에 대한 출력을 먼저 실시하고 DFS\_R에서는 방문한 vertex에 대한 표현으로 STL vector를 사용하여 나타냈습니다. visit배열과 같은 원리 이므로 visit->at(baseV) = 1 이라는 표현을 사용하여 방문을 표시한뒤 getIncidentEdges함수를 통하여 현재 baseV와 연결된 edge들을 m2로 불러오고 반복문을 들어 가게 됩니다. 첫 번째 for문에 해당하는 it의 위치의 first의 값과 두 번째 for문의 a 값이 같게 된다면(현재 baseV와 연결되어있는 vertex에 해당된다면) 그리고 그 vertex가 방문한 vertex가 아니라면 그때의 a 값을 baseV의 값으로 최신화 시켜준뒤 다시 DFS\_R함수를 호출하는 재귀적알고리즘을 사용하여 DFS\_R 깊이 우선탐색알고리즘을 구현했습니다.

## Kruskal

Kruskal 알고리즘은 주어진 vertex들을 최소한의 weight를 가진 edge로 모든 vertex를 연결 짓는 방식을 구현하는 알고리즘입니다. Kruskal알고리즘을 구현하기 위해서 vertexSet class를 사용했습니다. vertexSet class의 사용을 위해서 먼저 graph size만큼 vertexSet의 생성자로 전달한뒤 vertexSet class함수의 사용의 위해 vertexSet class의 이름을 set이라고 정의 했습니다. Kruskal알고리즘은 크게 3부분으로 나뉩니다. 첫 번째는 모든 vertex의 연결된 edges들을 multimap에 inster하여 정렬하기, 둘째 정렬된 multimap을 처음부터 끝까지 돌면서 weight가 작은 부분부터 차근 차근 사이클이 발생하지 않게 이어주기, 세 번째 모든 vertex의 부모가 같은지 확인한후 errorcode 출력하기 이렇게 3부분으로 나누어서 생각했습니다. 먼저 첫 번째 부분부터 살펴 보면 반복문을 사용하여 graph->getIncidentEdges(a, &m2)라는 표현을 사용하여 a가 0부터 graph size보다 1 작은 값까지 계속해



서 인접행렬을 불러 오게 하고 그러한 인접행렬을 불러 올때마다 iterator it을 사용하여 불러온 인접행렬의 begin부터 end 바로 전까지 돌게 하면서 미리 multimap으로 선언해 두었던 weight에 weight.insert(make\_pair(weight의 값, make\_pair(from vertex의 값, to vertex의 값)))으로 insert하게 됩니다. multimap으로 선언하고 insert한 이유는 같은 값의 weight를 가진 edges가 있기 때문에 multimap으로 하게 되었습니다. 이렇게 weight로 모든 인접행렬들의 weight의 값을 넣어주었다면 두 번째 단계로 넘어 가게 됩니다. 두 번째 단계에서는 weight의 저장된 값을 탐색하는 iterator를 사용해서 weight의 begin부터 end한칸 전까지 탐색을 시작하면서 weight의 second에 저장된 from 과 to값을 set.Find함수로 넘겨 주어서 두 vertex의 부모 vertex가 다르다면(사이클이 발생하지 않는다면), setUnion을 사용하여 이어주고, 같은 부모 vertex를 같은 다면 이어주지 않는 방식을 사용하여 연결짓기를 수행했습니다. 마지막 3번째 부분에서는 for문을 사용하여 a가 0부터 graph size보다 1 작은 값까지 반복하면서 set.Find(a)값의 비교를 수행하게 되는데 완벽한 MST의 연산을 수행한 경우에는 부모 vertex의 반환 값이 모두 같아야 하므로 만약 다르다면 false를 출력해서 MST연산의 실패를 나타내는 방식으로 켜 Kruskal알고리즘의 구현을 완성하게 되었습니다. 이렇게 구현되어진 알고리즘을 예시그림을 살펴보면서 부가 설명하겠습니다.



해당 그래프에대한 Kruskal알고리즘을 구현한다고 하였을 때 위에서 나눈 단계별로 생각하여 예시를 들겠습니다. 먼저 각 vertex에 대한 모든 edge들을 weight에 weight, from, to 순서로 insert하는 첫 번째 단계까지 완료되었을 경우에는

weight	[18]((2, (2, 5
[comp]	less
[0]	(2, (2, 5))
[1]	(2, (5, 2))
[2]	(6, (3, 4))
[3]	(6, (4, 3))
[4]	(7, (0, 1))
[5]	(7, (1, 0))
[6]	(9, (0, 2))
[7]	(9, (2, 0))
[8]	(9, (4, 5))
[9]	(9, (5, 4))
[10]	(10, (1, 2))
[11]	(10, (2, 1))
[12]	(11, (2, 3))
[13]	(11, (3, 2))
[14]	(14, (0, 5))
[15]	(14, (5, 0))
[16]	(15, (1, 3))
[17]	(15, (3, 1))

weight의 값으로 이러한 값들이 저장되게 됩니다. 이때 weight의 iterator를 사용하여 weight의 begin부터 end한 칸 전인 weight[0]~weight[17]까지 돌면서 그때의 from과 to 에 대한 Union연산의 실시여부를 판단하게 됩니다. 이러한 판단을 실시하면서 연결지을때의 weight의 값을 그때 마다 출력을 해주게 되고 이러한 출력값 겹 겹 과 값을 살펴보면

```
=====Kruskal=====
2 6 7 9 9
33
=====
```

이 그림의 위의 제시된 graph와 연결지어서 생각한다면 총 6개의 vertex에 땡나 vertex 0부터 5 까지를 모두 연결짓는 edge중 가장작은 weight의 합을 보이면서 연결시킬 수 있는 방법이 2, 6, 7, 9, 9의 weight를 가진 edge를 연결지으면 모든 vertex가 연결되는 것을 확인하고 Kruskal알고리즘에 문제가 없는 것을 확인 할 수 있습니다.

## ListGraph

ListGraph class서 사용되는 m\_List의 경우에는 graph size만큼 동적할당 받는 맵형 배열로써 생성자에서 동적할당을 실시했습니다. 소멸자에서는 동적할당을 해제해주었습니다. ListGraph의 insertEdge함수는 전달인자로 전달 받은 from 에대한 to의 연결과 그때의 weight의 값이므로 m\_List[from]에 해당하는 값을 insert(make\_pair(to, weight))표현을 사용하여 나타냈습니다.

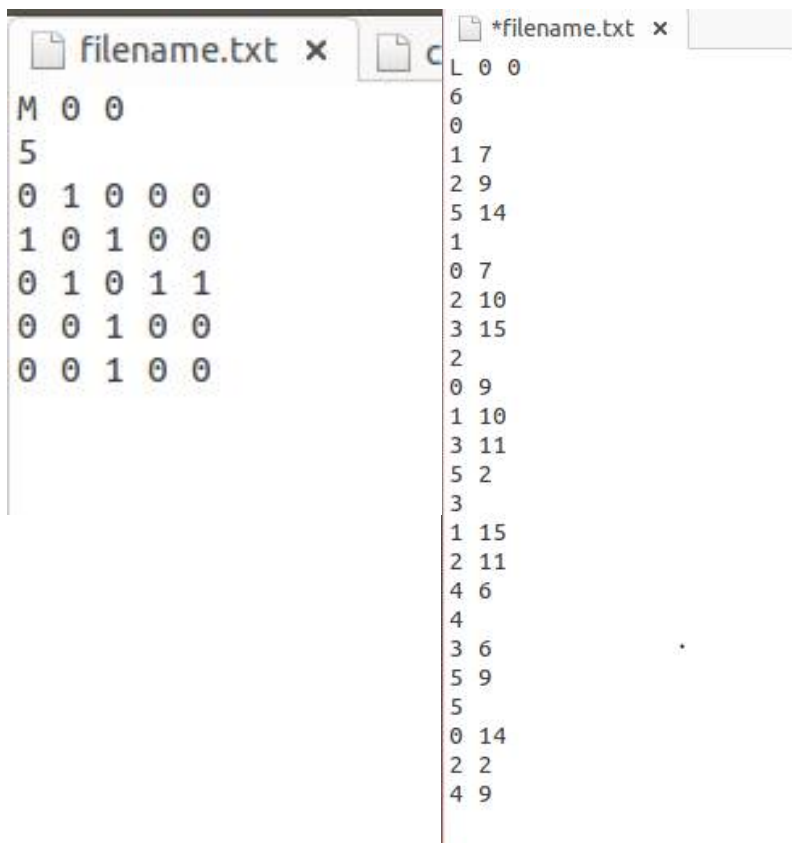
getIncidentEdges함수에서는 insertEdges함수에서 넣어준 값들을 map<int, int>\* m의 값으로 받아 오게 되는데 전달인자로 받은 vertex와 연결된 정보를 m의 값으로 넘겨 주는 것이므로 \*m = m\_List[vertex]라는 표현을 사용하여 m이 전달 받은 vertex와 연결된 정보를 가지고 있는 map을 가리킬 수 있도록 설정했습니다.



printGraph 함수에서는 m\_List에 저장된 모든 값들을 출력해주는 함수입니다. 따라서 입력받은 graph가 없을 경우(getSize() == 0) false를 출력하지만 그밖의 경우에는 m\_List의 각각의 vertex에 연결되어있는 edge와 weight를 출력해야 합니다. 따라서 m\_List[a].begin()부터 end한칸 전까지 도는 iterator를 선언하고 a가 0부터 graph size의 1작은 값을 가질 때 까지 반복하면서 그때의 a의 값과 iterator를 활용하여 a값에 해당하는 edge, weight를 출력해주는 방식으로써 Listgraph의 printgraph를 구현했습니다.

## Result Screen

graph.txt



```
filename.txt x  *filename.txt x
M 0 0
5
0 1 0 0 0
1 0 1 0 0
0 1 0 1 1
0 0 1 0 0
0 0 1 0 0

L 0 0
6
0
1 7
2 9
5 14
1
0 7
2 10
3 15
2
0 9
1 10
3 11
5 2
3
1 15
2 11
4 6
4
3 6
5 9
5
0 14
2 2
4 9
```

결과 화면

```
===== ERROR =====  
100  
===== ERROR =====  
100  
===== ERROR =====  
200  
===== ERROR =====  
300  
===== ERROR =====  
400  
===== ERROR =====  
600  
===== ERROR =====  
700  
=====
```

```
=====LOAD=====
Success
=====
=====PRINT=====
0 1,7 2,9 5,14
1 0,7 2,10 3,15
2 0,9 1,10 3,11 5,2
3 1,15 2,11 4,6
4 3,6 5,9
5 0,14 2,2 4,9
=====
=====BFS=====
0 1 2 5 3 4
=====
=====DFS=====
0 1 2 3 4 5
=====
=====DFS_R=====
0 1 2 3 4 5
=====
===== ERROR =====
700
=====
===== ERROR =====
600
=====
```

cd c:\program files\project34\run

```
===== LOAD =====  
Success  
===== PRINT=====  
0 1 0 0 0  
1 0 1 0 0  
0 1 0 1 1  
0 0 1 0 0  
0 0 1 0 0  
===== BFS =====  
3 2 1 4 0  
===== DFS =====  
1 0 2 3 4  
===== DFS_R=====  
1 0 2 3 4  
===== ERROR =====  
600  
===== ERROR =====  
700  
=====
```

```

===== ERROR =====
300
=====
===== ERROR =====
400
=====
===== ERROR =====
500
=====
===== ERROR =====
600
=====
===== Dijkstra =====
0
[1] 0 (1)
[2] 1 0 (3)
[3] 0 X
[4] 0 X
=====

```

```

===== ERROR =====
300
=====
===== ERROR =====
400
=====
===== ERROR =====
500
=====
===== ERROR =====
600
=====
===== Dijkstra =====
0
[1] 0 (7)
[2] 0 (9)
[3] 2 0 (20)
[4] 5 2 0 (20)
[5] 2 0 (11)
=====

```

```
===== ERROR =====  
300  
=====   
===== ERROR =====  
400  
=====   
===== ERROR =====  
500  
=====   
===== Kruskal =====  
2 6 7 9 9  
33  
=====   
===== ERROR =====  
700  
=====
```

## Consideration

### 김동민 고찰

이번 프로젝트를 구현하면서 맵의 포인터라는 개념이 어려웠습니다.

처음 ListGraph와 MatrixGraph를 보면서 getIncident는 어디에 쓰며 어떻게 구현하는지 생각하는데 굉장히 오랜시간이 걸렸습니다.

Manager를 어느정도 코드 구현을 하면서 insertEdge를 사용한 뒤에야 getIncident를 어떻게 사용해야 할지 감이 왔습니다.

본격적으로 GraphMethod를 구현할 때 BFS와 DFS는 비교적 쉬웠던 것 같습니다.

익숙했던 queue와 stack을 사용했고 함수의 개념또한 잘 알고있었습니다.

하지만 DFS\_R에서 조금 헛갈렸습니다. 구현하고 나서 보면 정말 당연한 코드인데 구현할 당시에는 idea가 쉽게 떠오르지 않았습니다. 위에 구현한 DFS를 보며 다시한번 개념을 생각하고 나서 어느 정도 후에 구현이 되었고 3개중 가장 간단한 코드였습니다.

그다음 kruskal은 굉장히 어려웠습니다. 개념을 정확히 알지 못한채 막연히 구현해보면 나오겠다는 생각에 코드작성을 시작했습니다.

처음엔 Union이 넣고나서 cycle이 도는지 확인 후에 cycle이 돌지 않으면 pass, 돌면 다시 parent[]에 -1을 하도록 구현했습니다. 이러한 방식으로 하다보니 구현 후에도 오류가 많았습니다.

먼저 root가 명확하지 않았고 1이 root라고 가정해도 1,2 가 연결 1,3이 연결되어 있다면 parent[2] = 1 p[3]=1 후에 2와 3이 연결된다면 어떻게 해야하나..고민하며 이런경우는 cycle 비교를 위해 parent[1]=2로 바꾼뒤 parent[2] = 3 으로 저장하여 root도 꼬이고 그때의 cycle 비교 또한 굉장히 까다로웠습니다. 어찌어찌 구현하였다고 생각했는데 다른 예시의 textfile을 입력하면 또 오류가 났습니다,

다시 강의자료를 보며 Find와 Union의 개념을 정립하였습니다.

알고나니 정말 5가지의 함수중 가장 간단한 문제였던 것 같습니다.

다만 아쉬운 것은 모든 vertex가 연결되지 않을 때 MST를 구현하지 못하므로 error code를 출력하도록 하였는데 코드를 다시 수정하며 그 부분을 깜박하고 삭제하였습니다.

Dijkstra도 구현하는데 적지않은 시간이들었습니다.

처음 distance 배열을 infinite가 아닌 0으로 초기화 하였더니 weight를 비교할 때 처음에는 어떠한 값도 저장되지 않았습니다. 또한 그 경로를 저장하는데 있어서 2차원 배열을 사용할지 vector나 deque 등 다른 방법을 사용할지 감이 오지 않아 익숙한 2차원 배열을 사용하였는데 저장하는 방법이 생각보다 어려웠습니다.



## 팀원들의 소스코드에서 배운점, 고칠점(구체적 사례)

### 이형민

:kruskal 에서 map<int,pair<int,int>>를 사용하여 weight와 from,to를 하나의 map에 표현하였는데 전 이 방법을 생각하지 못하고 두 개의 맵 map<int,int>을 이용하여 weight,from/ weight,to 이렇게 사용하였습니다. 이형민 조원의 코드를 보니 제가 한 방법보다 더 간결하고 kruskal을 구현하는데 있어서도 더 좋을것 같습니다.

Dijkstra 구현시 저는 모든 경로를 2차원 배열에 저장후에 출력을 하였는데 이형민 조원은 경로 중간에 출력이 있어서 코드를 읽는데에는 조금 불편했습니다.

이형민 조원의 코드는 주석과 코드를 구분짓는 가독성의 부분에 있어서 상당히 모호해서 코드를 알아보는데 조금 힘든감이 있었습니다. 특히 Dijkstra와 Kruskal알고리즘을 해석하고 이해하는데 있어서 각각의 알고리즘 덩어리 들이 어떠한 역할을 수행하고 있는지를 잘 알 수 없었습니다. 이러한 부분은 본인 스스로 프로젝트에 역할에 부합하는 코드를 설계하는데에는 문제가 없을지 모르지만 여러사람이 자신의 코드를 공유하게 될 경우 다른사람의 눈에서는 자신의 코드에 대한 가독성이 떨어지기 때문에 이형민 조원에게 있어서 꼭 고쳐야할 점이라고 생각합니다.

### 김동현:

이형민조원과 마찬가지로 kruskal에서 weight, from, to를 하나의 맵에 저장하여 더 효율적으로 map을 사용하였습니다.

kruskal을 구현할 때 MST를 형성하지 못하는 경우 false를 반환하여 에러코드를 출력해야하는데 그 부분이 생략되어 있습니다. 프로젝트 문서에 명시되어 있는 예외처리를 하지 않았을 경우에는 이번과제에서 감점이 될 수도 있고 예외처리를 하지 않는 예외적 상황이 프로그램의 input값으로 들어 왔을 때 각각의 프로그램 알고리즘이 어떻게 동작할지를 모르기 때문에 이러한 예외처리는 꼭 필요한 예외처리라고 생각합니다.

### 김동현 고찰

먼저 2번의 프로젝트를 진행하면서 stl 에 대한 이해와 사용능력을 키워와서 이번에 stl을 이용해서 좀더 수월하게 문제들을 해결 할 수 있었다. 이번에는 map 의 포인터를 사용해서 값을 저장하고 넘겨주었다. 흔히 우리가 1학년때 수행했던 call by reference를 이용해서 map 에 값을 넘겨주었다. 처음엔 다소 난해 했지만 기존의 데이터 형식의 포인터에 대한 내용을 다시 복습하고 접근한 결과 int \*와 다를 것이 없다는 것을 알았다. 그리고 처음에는 map에 incident를 받아올 때 map을 동적 할당 시켜서 새로운 기존의 vertex 정보들을 다 map에 받아와서 문제를 해결하는 식으로 하려고 했다. 이렇게 하면 BFS 나 DFS 에서 visit 이라는 배열을 만들어서 굳이 vertex를 방문했는지 안했는지 사용하지 않고 BFS 나 DFS는 가중치가 없기때문에 map에 weight 부분을 0으로 표시함으로써 visit의 여부를 판단하는 식으로 코드를 작성하였으나 코드를 작성하고 함수를 만들어가면서 굳이 이럴 필요가 없다는 것을 깨달았다. 나의 방식대로 한다면 먼저 map 을 다 받아 와야 하기 때문에 적어도

$size^2$  만큼의 연산을 수행하고 함수를 실행하는 것과 같다. 따라서 만약 vertex 의 사이즈가 매우 커지게 된다면 map 을 받아오는 연산만으로도 많은 시간이 소요되기 때문에 성능에 매우 악영향을 미친다는 느꼈다. 따라서 코드를 수정했는데 최종적으로는 map 배열에 모든 요소들을 받아 오지만 map 하나를 필요할 때 불러서 사용하는 식으로 수정을 했다. 따라서 기존에  $size^2$  만큼의 시간은 걸리는 것을 피했지만 불필요한 메모리들을 저장하고 있어야 한다는 점에서 아직 나의 코드엔 단점이 남아있다고 볼 수 있다.

그리고 kruskal 의 union 과 find의 개념이 잡히지 않아 vertex 가 루프를 형성하는지 안 하는지 확인하는데 매우 큰 어려움이 있었다. 따라서 만약 이러한 개념을 몰랐다면 size 만큼의 포문을 돌려서 중복되는 값이 나오는지 확인해보는 과정을 거쳐야 했을 것이다. 하지만 각종 사이트와 google을 통해 union find를 이해하게 되었고 좀더 빠른 시간에 vertex 가 루프를 형성하는지 안 하는지 확인할 수 있었다.

Dijkstra도 구현하면서 인터넷을 찾아보면서 기존의 방법은  $n^2$  의 복잡도 이지만 heap 을 이용하면  $n \log n$ 으로 단축시킬 수 있다는 것을 알게 되었다. 프로젝트는 비록  $n^2$  의 방법으로 구현을 했지만 복잡도를 줄이는 방향으로도 다시 한번 코딩해볼 가치가 있다고 판단이 든다.

이형민 조원의 코드를 보면 다익스트라 알고리즘에서 매우 많은 반복 문을 사용한 것을 볼 수 있다. 5중 반복 문을 사용하였는데 아주 최악의 경우  $n^5$ 의 복잡도가 된다는 말이다. 물론 정확히 측정해 보진 않았지만 이렇게 코드를 작성하면 성능 면에 있어서 매우 안좋은 영향을 줄 것이다.

```
while(1)
{
    int count = 0;
    for(int a = 0; a<graph->getSize(); a++)
    {
        .....(생략)
    }
    if(count == graph->getSize())//
    {
        .....(생략)
        for(int a = 0; a<graph->getSize() ; a++)// if visited all vertex
        {
            .....(생략)
            while(temp2 != origin)
            {
                .....(생략)
                for(
```

.....(생략)

}

물론 다익스트라 알고리즘을 제외하면 나보다 좀더 다양한 상황에 대해서 예외처리를 해줬다는 것이 본받을 점이였다. Kruskal 의 경우 모든 vertex가 연결되어있지 않은 상황에 대해서도 예외처리를 해주는 등 꼼꼼하게 예외처리를 잘 해주었다.

김동민 조원의 코드를 보면 먼저 일단 주석이 깔끔하게 달려 있어서 한눈에 보기 편했다. Kruskal 에서 multimap을 사용해서 문제를 해결해 나갔는데 이에 비해 나는 priority queue를 이용해서 문제를 해결했다. Multimap을 사용함에 따라 둘다 장단점이 있겠지만 굳이 우선순위 큐까지 쓰지 않아도 됐었다 라는 것을 느꼈다. 그리고 많은 동적 할당을 해주었는데 모두 꼼꼼하게 해제를 해 준 것을 볼 수 있었다.

```
str=NULL;
```

```
char* str2=NULL;
```

```
char* rest = NULL;
```

```
char buf[128]={0};
```

```
char buf2[128]={0};
```

```
char buf3[128]={0};
```

```
char* type = 0;
```

```
int directed = 0;
```

```
int weighted = 0;
```

```
int size = 0;
```

```
int vertex_to = 0;
```

```
int vertex_from = 0;
```

```
int graph_size = 0;
```

```
int graph_weight = 0;
```

```
int count1 = 0;//
```

```
int count2 = 0;//
```

```
int count3 = 0;//
```

```
int count4 = 0;//
```

다만 로드에서 불필요하거나 아니면 그전의 변수를 사용해도 충분했던 경우가 있었는데 그럴 때 마다 새로운 변수를 선언해서 사용하 다보니 다량의 변수가 선언된 것을 볼 수 있다.

## 이형민 고찰

이번 프로젝트를 진행하면서 LsitGraph, MatrixGraph class를 설계하는 부분과 graphMethod중 다익스트라 알고리즘을 설계하는 부분이 제일 시간이 많이 들었습니다. 어떤 방식으로 Manager의 LOAD에서 전달인자를 전달시켜서 insert시켜주는 방법부터 시작해서 getIncidentEdges함수의 용도와 STL map을 2차원배열처럼 사용하기 위한 사용법등을 깨닫고 코딩할 수 있을만큼의 이해도를 가지는데 까지 오랜시간이 걸렸습니다. 2차 프로젝트에 비해서 코딩의 양은 상당히 줄었지만 1차 프로젝트에 이어서 트리를 설계하는 것이었던 2차프로젝트와는 다르게 3차프로젝트는 처음보는 graph를 코딩으로써 설계를 하려고 하니 처음에는 어떤식으로 각각의 함수들을 정의해야 되는지 아는데 까지 오랜시간이 걸렸다고 생각합니다. 이 함수의 용도가 무엇인지 왜 이러한 전달인자를 가지고 정의가 되어있는건지 왜 형식이 void, bool형인지를 먼저 이해하기위해 노력했습니다. 실습시간 조교님들한테 함수의 용도에 대해서도 여쭙보기도 하고 왜이러한 전달인자를 가지고 있어야 하는 지도 여쭙보면서 class에 정의된 함수하나하나를 이해해 나갔습니다. 이러한 함수의 용도를 알고 나니 어떤식으로 함수의 내용을 작성해야하고 그래프를 설계하는데 있어서 어떠한 방식으로 이용해야되는지를 깨닫게 되었고 그래프를 만드는 기본함수들의 이해를 바탕으로 코딩을 수행한 결과 여러 가지 그래프들을 구현하는데 기초적인 이해를 가지고 설계를 할 수 있었습니다. 다른 여러 가지 graph들을 구현하는데도 많은 시간이 들었지만 특히 다익스트라 알고리즘을 구현하고 최단경로로 거쳐가는 vertex를 출력하는데 상당히 많은 시간과 이해도가 필요했습니다. 맨 처음 다익스트라 알고리즘을 구현할때는 최단경로로 지나가는 각각의 vertex에 대한 weight의 합만 출력하는 방식으로 먼저 구현을 했습니다. 각각의 vertex에 대한 경로의 합을 비교하고 더 작은 합의 weight를 가진다면 그 값을 경로 weight의 합으로 바꾸어주는 방식으로 설계를 완료했지만 프로젝트 스펙문서의 Dijkstra print형식을 보니 최단경로로써 방문하게되는 vertex들을 추가로 출력하는 방식이 사용된다는 것을 확인했습니다. 제가 설계한 알고리즘 상으로는 당연히 거쳐가는 vertex들이므로 그러한 vertex들을 출력해주면 되겠구나 라고 쉽게 생각했었지만 막상 코딩으로써 적용할 수 있는 설계를 실시하려고 해보니 어떤 부분에서 출력을 실시해야하는건지 그리고 stack을 써서 해야되는지 배열을 사용해서 값을 미리 저장을 해줘야 되는지를 선택하는 판단이 잘 되지 않았습니다. 처음 알고리즘을 설계할 때 stack을 이용했습니다. 지나는 vertex에서 최단 경로가 최신화 될 때마다 현재 vertex의 값을 stack에 넣어주어서 나중에 한번에 출력하는 방식으로 설계를 하려고 했으나 설계 후에 print를 해서 살펴보니 각각의 vertex에 대한 최단 경로의 vertex의 수가 정해져 있지 않고 최단경로가 바뀌는 그때의 vertex를 stack에 넣다 보니 방문하지 않는 vertex가 insert되어 있는 것을 발견하게 되었습니다. 따라서 stack에 넣는 것을 관두고 2차원 배열에 각각의 경로들을 표시하는 방식으로 설계를 시작하게 되었습니다. graphSize보다 사이즈가 1정도 큰 2차원 배열을 동적할당 받아서 각각의 weight의 합이 바뀔때 마다 course[][]라는 2차원 배열에 경로를 표시하는 즉 vertex 1에서 2로 이동했을때의 weight의 값이 바뀌게 되었다면 course[1][2] = 1 라는 표현을 사용하여 나타내고 원래 있던값에서 바뀌었을 때는 최신화 된 그부분의 경로만 1로 설정하고 나머지 경로는 0으로 초기화하는 방식을 통해 course배열을 구현하였고 print부분에서는 각각의 vertex에서 거꾸로 값을 찾아 가면서 경로의 vertex를 출력해주는 방식으로써 구현했습니다. 이번 프로젝트에서는 예외처리를 하는 부분에 있어서 지금까지의 프로젝트중에서 가장많은 경우의 수를 생각하면서 예외처리를 했습니다. 그중에서도 각각

의 vertex가 모두 이어진 graph가 아닌 graph를 크루스칼, 다익스트라 알고리즘으로 나타낼 때 errorcode와, x 표시를 출력하는 부분에서 다익스트라에 해당하는 x표시 출력을 하는 부분에서 처음 설계당시 제가 하지 못하는 수준의 예외처리라고 생각했습니다. 가는 경로를 2차원배열을 사용하여 출력한 것도 겨우겨우 알고리즘을 계속해서 수정해 가면서 있는 힘껏 다한 설계 였는데 갈 수 없는 vertex에 대한 x표시 출력은 처음 설계구현당시에도 아무런 감이 잡히지 않았습니다. 갈 수 없는 vertex를 가진 graph를 지금까지 설계했던 다익스트라 알고리즘으로 실행시켰을 때 무한루프를 도는 것을 시작으로하여 넘겨줘야 할 baseV의 값을 넘겨주지 못해서 계속해서 같은 baseV를 넘겨주어서 무한루프를 도는 것이었는데 이러한 문제를 시작으로 경로를표시하는 2차원배열에서의 x표시 출력을 어떻게 해주어야 하는지를 포함해서 수많은 경우의 수를 생각하고 예외처리에 대한 예외처리를 실시해가면서 프로그램 x표시 예외처리를 구현했습니다. 프로그램 코딩실력은 물론 어떠한 알고리즘을 가지고 효율적인 프로그램코딩을 실시하냐도 중요한 문제이지만 얼마나 세세한 예외처리를 해주는지에 따라 설계자가 잘하는 설계자인지 못하는 설계자 인지를 알 수 있다는 말을 들었습니다. 이 부분에서는 이번 프로젝트를 설계하는데 있어서 정말 도움될만한 많은 경우의 수에 대한 생각들을 실시했다고 생각합니다.

#### -자신의 코드에 대한 고찰

이번 프로젝트는 다른 여러 가지 프로젝트를 마무리 짓고 코딩을 실시하다 보니 촉박한 시간에 프로그램을 구현했습니다. 그래서 이번 보고서 작성시에 제 코드를 다시한번 살펴보니 프로그램은 정상적으로 동작을 실시하지만 제3자의 입장에서 보았을 때 왜 이러한 알고리즘을 굳이 거쳐야 되는지를 명확히 설명이 안되는 부분이 몇 가지 있었습니다. 특히 이번 graphMethod 다익스트라에서 갈수 없는 경로를 x로 표시하는 예외처리의 부분이 다시 보니 이해하기 힘든 부분이 있었습니다. 이번 프로젝트에서는 프로그램의 수행시간도 채점기준이 되는 것으로 알고있습니다. 하지만 제가 적용한 알고리즘의 수행시간을 살펴보니 예외처리를 실시하는 부분에서 쓸모없는 반복문이 상당히 자주 사용된 것 같다는 느낌을 받게 되었습니다. 굳이 for문 이나 while문 같은 반복문을 사용하지 않고도 같은 역할을 수행하는 알고리즘을 수행할 수 있었는데 반복문을 적용시켜서 프로그램을 구현하다보니 이런식으로 프로그램이 구현되었다고 생각하는데 비록 이번 학기 DS과목은 끝이 났지만 항상 이런부분들을 염두해두고 프로그램을 구현해야 된다는 깨달음을 얻게 되었습니다. 여러 가지 그래프들을 구현하면서 모든 배열을 설계하는데 있어서 동적할당을 사용해서 이용했습니다. 이부분에 있어서는 어떠한 그래프를 설계하게 될지도 모르고 몇 개의 vertex들을 이용해서 알고리즘을 구성해야 될지를 모르므로 이런부분에 있어서는 제코드의 장점이 있다고 생각합니다. 비록 작은 범위의 그래프라면 굳이 동적할당을 실시하지 않고 선언 및 초기화를 해도 되지만 그래프의 범위가 커지게 된다면 프로그램이 동작하지 않기 때문입니다.

#### - 김동민 조원에 대한 고찰

가장 기본적인 getIncidentEdges 함수로의 전달인자와 저와는 다르게 map<int,int>\* 형인 것을 확인했습니다. 물론 같은 반환형을 가지기 때문에 상관 없는 표현이라고 생각하지만 다른 조원의 코드를 확인하게 되니 상당히 기초적인 부분부터 나오는 다른 생각을 가지는 코딩을 수행할 수 있구나 라는 생각을 하게 되었습니다. 대부분의



코드에서 저와는 다른 알고리즘을 가지고 같은 역할을 수행하고 있었지만 BFS, DFS, DFS\_R과 같은 경우는 그나마 유사한 방식으로 수행되어있었습니다. 하지만 Kruskal, 과 Dijkstra알고리즘의 경우에는 대부분의 코드에서 다른 방식으로 구현되어있었는데 Kruskal알고리즘의 경우에는 제코드와 비교해 보았을 때 불필요한 반복문이 있다고 생각합니다. vertexSet class의 내용도 똑같지만 알고리즘을 구성하는 부분에서 저와는 다른 단계를 설계한 뒤 프로그램을 구현해서 그런지 코드의 길이도 더 길지만 같은 역할을 수행하고 있습니다. 이런 부분에 있어서는 프로그램의 수행시간과 코드의 길이 두부분 모두 커지게 되므로 수정해야 된다고 생각합니다. 하지만 Dijkstra알고리즘을 확인해 본 결과 제 3자의 입장으로 보았을 때 저보다 훨씬 깔끔한 코드로 실행된다는 생각을 했습니다. 특히 최단거리의 경로를 지나가면서 vertex를 출력해주는 부분에 있어서 제가 구현한 방식과 같은 방식인 똑같이 2차원 배열을 가지고 설계를 했지만 훨씬더 깔끔한 출력결과와 알고리즘을 확인 할 수 있었는데 비록 같은 형식의 틀을 가지고 설계했지만 이런식으로의 알고리즘의 구상도 생각해 볼 수 있구나 라는 생각을 하게 되었고 제가 구현한 방식이 얼마나 복잡하고 제3자가 보기에 이해하기 힘든 알고리즘이었구나 라는 생각을 하게 되었습니다. 다익스트라 알고리즘과 같은 경우에는 제가 배워야 할 부분이라고 생각합니다.

#### - 김동현 조원에 대한 고찰

김동현 조원과 같은 경우에는 제 3자의 입장에서 코드를 확인해 봤을 때 간결한 코드의 구성이 눈에 띄었습니다. 같은 역할을 수행하는 Dijkstra 알고리즘에서 가장 눈에 띄는 차이점을 나타냈지만 Kruskal 알고리즘에서도 약간의 차이점을 나타냈습니다. Dijkstra의 최단경로를 출력하는 부분에서 경로를 parent와 distance라는 2개의 배열을 이용해서 구현했는데 이러한 방법은 강의자료에서 나온 방법대로 설계한 것인데 훨씬 더 간결하고 보기 편한 코드구현법이라는 생각이들었습니다. 제가 구현한대로 2차원배열을 사용하여 설계하게 되면 독립 vertex가 있을 때의 출력표현인 x print를 구현하는 부분에서의 어려운 부분이 상당히 많게 되는데 김동현 조원과 같이 2개의 배열을 사용하여 구현을 하게 된다면 훨씬 더 쉽고 간결한 알고리즘의 구성이고 배울점이 많다고 생각합니다. 주석을 다는 방식에 있어서도 쓸데 없는 부가 설명이 들어가 있는 것이아니라 정말로 주석이 필요한 문장에만 그에 대한 설명을 달아서 어떠한 반복문이 어떠한 역할을 수행하고 있는지를 보기 편하게 표현했습니다. 지금까지 제가 주석을 달았던 경우에는 어떠한 부분이 중요한지를 콕콕 찌는 것이 아닌 거의 대부분의 문장에 주석을 달았는데 그와 같이 할 경우 제 3자의 입장에서 코드를 봤을 때 코드와 주석의 경계가 상당히 모호해 지기 때문에 가독성에 문제가 있다고 생각했습니다. 따라서 이러한 주석의 요약도 배울점이라고 생각합니다. 하지만 김동현 조원이 구현한 Kruskal알고리즘에서 MST를 구현하지 못할 때 errorcode를 출력하는 예외처리에서 이러한 예외처리를 해주지 않은 것을 발견했습니다. 이는 프로젝트 스펙에 나와있는 예외처리 중에하나로 독립된 개체가 있어서 혹은 다른 이유로 MST를 구할 수 없는 경우 errorcode를 출력해야 하지만 그러한 예외처리가 없기 때문에 어떤 방식으로 Kruskal알고리즘이 동작할지는 아무도 모르기 때문에 이러한 예외처리는 김동현 조원의 코드에서 부족한 부분이라고 생각합니다.

인증샷!!

