데이터구조 설계 및 실습

3차 프로젝트 보고서

제출일자: 2022년 12월 15일 (목)

학 과: 컴퓨터공학과

학 번: 2021202043

성 명: 이은서

1. Introduction

그래프를 이용하여 연산을 진행할 수 있는 프로그램을 구현하는 과제이다. 그래프 정보가 저장된 텍스트 파일을 읽어와서 그래프를 구현하고, 그래프의 특성에 따라 BFS, DFS, Kruskal, Dijkstra, Bellman-Ford, FLOYD 연산을 수행한다. 그래프이기 때문에 데이터는 방향성(Direction)과 가중치(Weight)를 모두 가지고 있으며, 데이터의 형태에 따라 List 그래프와 Matrix 그래프로 저장한다. 프로그램의 동작은 명령어 파일에서 요청한 명령에 따라 기능을 수행하고, 그 결과를 log.txt 파일에 저장하면 된다. 그래프 연산은 형식(List, Matrix)과 상관 없이 데이터를 입력받으면 동일한 동작을 수행하도록 일반화해야하며, 충분히 큰 그래프에서도 모든 연산이 정상적으로 수행되어야 한다.

프로젝트에서 요구하는 기능은 다음과 같다.

명령어	간단한 기능 설명
LOAD	그래프 정보가 저장된 텍스트 파일을 읽어 그래프를 정보를 불러와 그래프를 구성한다.
PRINT	그래프의 상태를 출력한다. List는 adjacency list, Matrixs는 adjacency matrix를 출력하며 vertex는 오름차순으로 출력한다.
BFS	입력한 vertex를 기준으로 BFS를 수행하여 방문하는 vertex의 순서로 결과를 출력한다.
DFS	입력한 vertex를 기준으로 DFS를 수행하여 방문하는 vertex의 순서로 결과를 출력한다.
DFS_R	입력한 vertex를 기준으로 DFS_R을 수행하여 방문하는 vertex의 순서로 결과를 출력한다.
KRUSKAL	MST를 구하고, MST를 구성하는 edge들의 weight값을 오름차순으로 출력하고 총합을 함께 출력한다.
DIJKSTRA	vertex, shortest path, cost 순서로 출력하며, path는 vertex에서 기준 vertex까지의 경로를 역순으로 출력한다.
BELLMANFORD	StartVertex를 기준으로 Bellman-Ford를 수행하여 EndVertex까지의 최단 경로와 거리를 구하여 결과를 출력한다.
FLOYD	모든 vertex의 쌍에 대하여 StartVertex에서 EndVertex로 가는데 필요한 cost의 최솟값을 행렬 형태로 출력한다.
EXIT	프로그램의 메모리를 해제하고 종료한다.

A. 데이터

텍스트 파일에서 데이터를 읽어 그래프를 형성해야 하므로 텍스트파일에 저장된 데이터에 대한 이해가 필요하다. 그래프의 형식은 두가지 (List, Matrix)가 있으므로 텍스트파일 역시 각 형식에 맞는 텍스트파일이 주어진다. 파일을 읽어서 얻어낸 정보

는 그래프 형식에 맞게 저장을 하면 된다. 첫 번째 줄에는 그래프의 형식 정보가 주어지고, 두 번째 줄에는 그래프의 크기가 저장되어 있으므로 1, 2번째 줄을 통해 그래프의 전체적인 틀을 결정하면 된다. 3번째 줄부터 데이터가 주어지는데, List 그래프의 경우에는 시작 vertex만 주어지거나 도착 vertex와 weight가 쌍으로 주어지는 2가지 형태가 존재한다. Edge가 없는 vertex의 경우에는 시작 vertex만 주어질 수 있다. 즉, List형태의 그래프가 크기가 3일 때 시작 vertex는 총 3개가 주어지고, 하나의 시작 vertex가 주어진 다음 줄에 바로 도착 vertex와 weight가 쌍으로 여러개가 주어질 수 있다. Matrix 그래프의 경우에는 2번째 줄에서 주어진 크기만큼 데이터가 한줄씩 주어지는데 weight의 값이 써져 있다. 행은 to vertex, 열은 from vertex이다. 즉 3번째 줄에 2번째로 작성된 값은 3번째 줄부터 데이터가 쓰여지고 인덱스는 0부터 시작하므로 (0,1)이라고 볼 수 있다.

B. LOAD

LOAD 명령어는 graph_L.txt 또는 graph_M.txt를 읽어 그래프 정보를 불러와 그래프를 구성하는 명령어이다. 텍스트 파일의 형식은 그래프 형태에 따라 다른데, 공통적인 것은 첫 번째 줄에는 그래프의 형식 정보가 저장되어 있고, 두 번째 줄에는 그래프 크기가 저장되어 있다는 것이다. 이 두 정보(형식과 크기)는 'Graph' class에 저장하며, 그래프 연결 정보는 형식에 따라 알맞은 클래스에 저장이 된다. List 그래프의 경우에는 3번째 줄부터 시작 vertex가 저장된 줄과 도착 vertex와 weight 정보가 쌍으로 저장되어 있는 줄이 번갈아 있다. 하나의 시작 vertex가 저장된 다음에는 여러개의 도착 vertex와 weight 정보가 여러줄로 저장되어 있을 수 있다. 그리고 이 연결 정보들은 'ListGraph' class에 저장한다. Matrix 그래프의 경우에는 3번째 줄부터 행렬 형식으로 저장되어 있다. 즉 3번째 줄은 시작 vertex, 하나의 줄에 첫번째 데이터부터 차례대로 도착 vertex라고 할 때 weight 값이 저장되어 있다. 이 정보들은 'MatrixGraph' class에 저장한다. 두개의 'ListGraph'와 'MatrixGraph' 클래스는 'Graph' 클래스를 상속받고 있다. 만약 텍스트 파일이 존재하지 않거나 비어있으면 오류 코드(100)을 출력하며, 기존에 그래프 정보가 이미 존재하고 있을 때 LOAD명령어가 입력되면 기존 그래프 정보는 삭제하고 새로운 그래프를 생성한다.

C. PRINT

PRINT 명령어는 그래프의 상태를 출력하는 명령어이다. List형 그래프의 경우에는 adjacency list를 출력하고 Matrix형 그래프의 경우에는 adjacency matrix를 출력하는데, 이때 vertex는 오름차순으로 출력한다. 만약 그래프 정보가 존재하지 않으면, 오류코드 (200)을 출력한다.

D. BFS

BFS 명령어는 vertex도 함께 입력받는데, 이 입력받은 vertex를 기준으로 BFS를 수행하는 명령어이다. BFS는 방향성과 가중치가 없는 그래프에서 수행이 가능하기 때문에 방향성과 가중치가 없다고 가정하고 수행해야 한다. BFS는 너비를 우선으로 탐색하여 vertex를 방문하고, queue를 이용하여 방문한 순서를 결과로 출력한다. Vertex는 값이 작은 vertex를 먼저 방문한다고 가정한다. 만약 입력한 vertex가 그래프에 존재하지 않거나, vertex를 입력하지 않은 경우에는 오류코드(300)을 출력한다.

E. DFS

DFS 명령어도 마찬가지로 vertex를 함께 입력받으며, 입력받은 값을 기준으로 DFS를 수행하는 명령어이다. DFS 역시 방향성과 가중치가 없는 그래프에서 수행이 가능하기 때문에 방향성과 가중치가 없다고 가정하고 수행해야 한다. DFS는 깊이를 우선으로 탐색하여 vertex를 방문하고, 재귀적 방법과 Stack을 모두 사용할 수 있지만 DFS 명령어에서는 Stack을 이용하여 동작한다. 방문하는 vertex의 순서로 결과를 출력하며 만약 입력한 vertex가 그래프에 존재하지 않거나, 입력하지 않은 경우에는 오류코드 (400)을 출력한다.

F. DFS_R

DFS_R 명령어도 BFS, DFS 명령어와 마찬가지로 vertex를 함께 입력받으며, 입력받은 값을 기준으로 DFS_R을 수행하는 명령어이다. DFS_R도 방향성과 가중치가 없는 그래프에서 수행이 가능하기 때문에 방향성과 그래프의 가중치가 없다고 가정하고 수행해야 한다. DFS_R은 DFS명령어와 마찬가지로 깊이를 우선으로 탐색하여 방문하는 대신, 방문하는 방법이 다르다. DFS에서는 Stack을 이용한 반면 DFS_R 명령어에서는 재귀적 방법을 이용하여 동작한다. 방문하는 vertex의 순서로 결과를 출력하며, 만약 입력한 vertex가 그래프에 존재하지 않거나, 입력하지 않은 경우에는 오류코드 (500)을 출력한다.

G. KRUSKAL

KRUSKAL 명령어는 현재 그래프의 MST를 구하고, MST를 구성하는 edge들의 weight 값을 오름차순으로 출력하고 weight의 총합을 출력하는 명령어이다. Kruskal 명령어는 방향성이 없고 가중치가 있는 그래프에서 수행이 가능하다. 저장되어 있는 그래프의 연결 정보를 이용하여 MST를 구하고, 구하는 과정에서 sub-tree가 cycle을 이루는지에 대한 검사도 같이 수행한다. 만약, 입력한 그래프가 MST를 구할 수 없는 경우이거나 명령어를 수행할 수 없는 경우에는 오류코드(600)를 출력한다.

H. DIJKSTRA

DIJKSTRA 명령어는 vertex, shortest path, cost 순서로 출력하며, shortes path는 해당 vertex에서 기준 vertex까지의 경로를 역순으로 출력하는 명령어이다. Dijkstra 명령어는 방향성과 가중치가 있는 그래프에서 수행이 가능하다. 명령어와 함께 입력받은 vertex를 기준으로 Dijkstra 알고리즘을 수행한 후, 모든 vertex에 대해 shortest path를 구하고, shortest path에 대한 cost값을 구한다. 기준 vertex에서 도달할 수 없는 vertex의 경우 'x'를 출력한다. 만약 입력한 그래프가 MST를 구할 수 없는 경우와 명령어를 수행할 수 없는 경우에는 오류코드(700)를 출력한다. Weight가 음수일 경우에도 오류코드(700)를 출력한다.

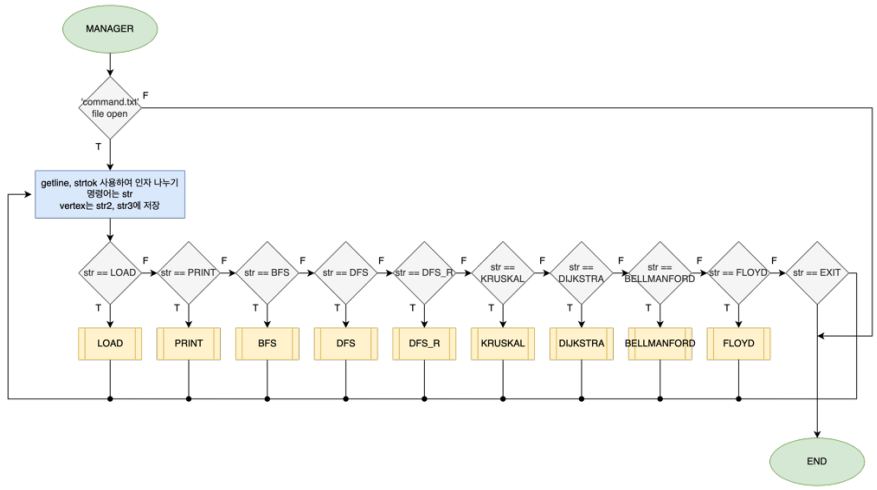
I. BELLMANFORD

BELLMANFORD 명령어는 StartVertex와 Bellman-Ford를 함께 입력받은데, 입력받은 StartVertex 값을 기준으로 Bellman-Ford를 수행하여 EndVertex까지의 최단 경로와 거리를 구하는 명령어이다. 음수인 weight가 있는 경우에도 동작해야 하고, 만약 음수 weight에 사이클이 발생한 경우에는 에러코드를 출력한다. 음수 weight가 있는데 사이클이 발생하지 않은 경우에는 정상 작동되어야 한다. StartVertex에서 Endvertex로 도달할 수 없는 경우에는 'x'를 출력하며, 입력한 vertex가 그래프에 존재하지 않거나, 입력한 vertex가 부족한 경우, 그리고 명령어를 수행할 수 없는 경우와 음수 사이클이 발생한 경우에 오류코드 (800)을 출력한다.

J. FLOYD

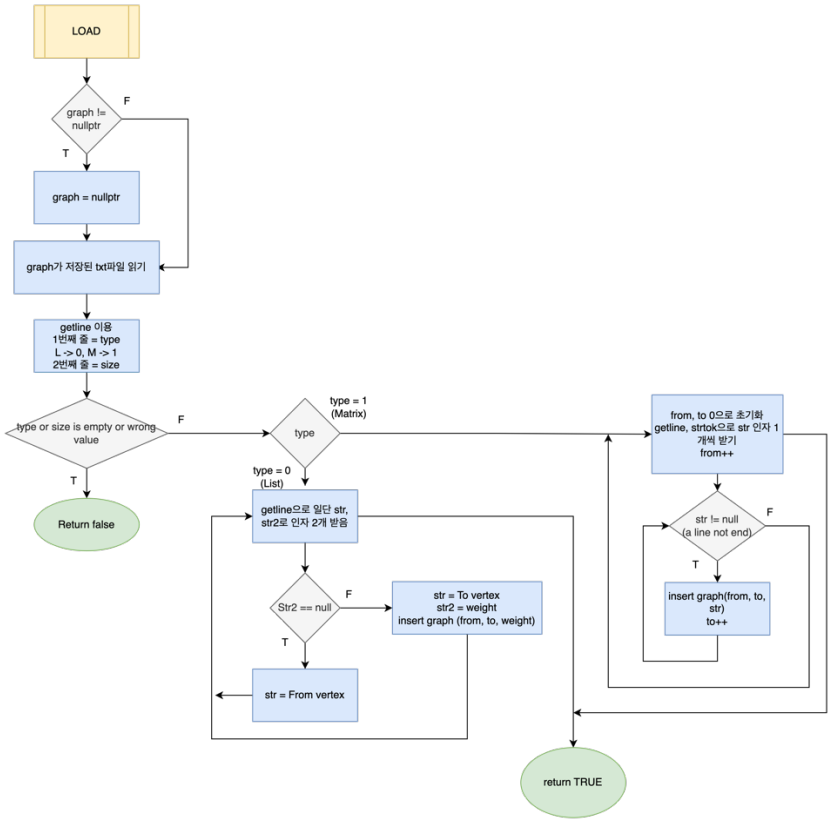
FLOYD 명령어는 입력 인자 없이 모든 쌍에 대하여 최단 경로 행렬을 구하는 알고리즘을 사용하여 StartVertex에서 EndVertex로 가는데 필요한 비용의 최솟값을 행렬 형태로 출력한다. 기준 vertex에서 도달할 수 없는 vertex의 경우 'x'를 출력하며, 명령어를 수행할 수 없는 경우와 음수 사이클이 발생한 경우 오류코드(900)을 출력한다. 음수 사이클만 허용하지 않으며 음수인 weight가 있는 경우에는 작동한다.

2. Flow Chart



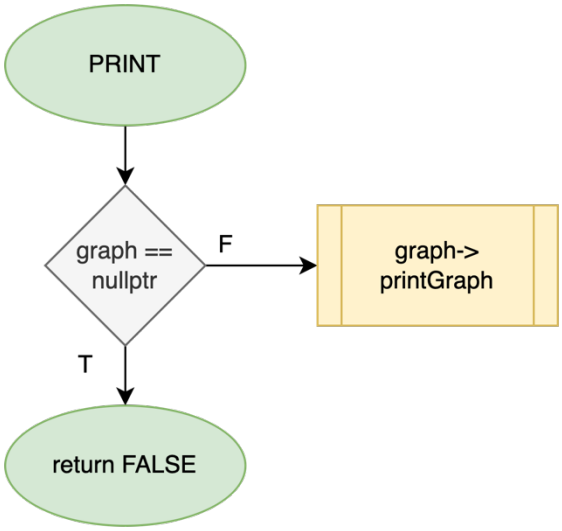
위 그림은 전체 명령어를 관리하고 있는 Manger 파일의 flow chart이다. LOAD, PRINT, BFS, DFS, DFS_R, KRUSKAL, DIJKSTRA, BELLMANFORD, FLOYD, EXIT 총 10개의 명령어를 관리하고 있다. Command.txt파일을 열고, getlined을 사용하여 한 줄 씩 읽은 뒤 strtok으로 공백을 기준으로 자른 다음 이를 인자로 하여 각각의 명령어에 필요한 인자만큼 넘겨 명령을 실행한다. EXIT의 경우 바로 break를 통해 while문을 탈출 후 종료하므로 해당 명령어를 제외한 총 9개의 명령어에 대한 flow chart를 보이며 설명을 진행하고자 한다.

1. LOAD



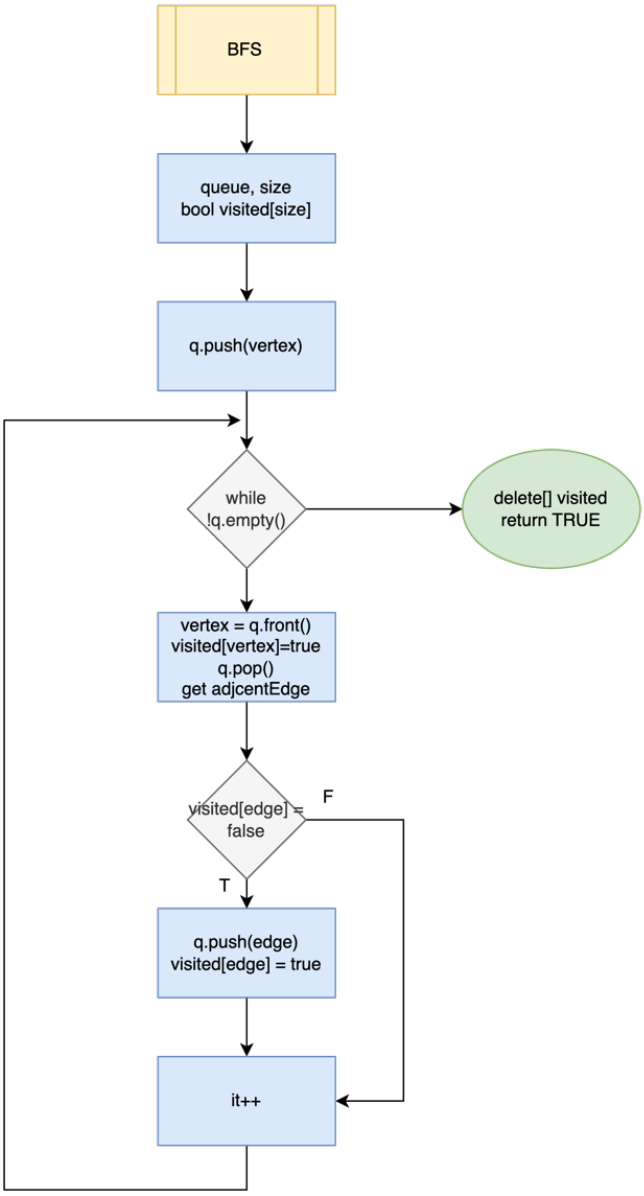
LOAD명령어의 Flow chart는 다음과 같다. 먼저 LOAD명령어가 불려지면 graph가 있는지 체크한다. Graph가 있으면 nullptr로 초기화해주고 LOAD 명령어와 함께 입력된 인자(파일 이름)을 통해 파일을 읽는다. 만약에 파일이 읽히지 않으면 에러 메세지를 출력하며 종료한다. Getline을 이용하여 1번째 줄과 2번째 줄을 미리 읽어 graph의 type과 size를 구축한다. L -> type = 0, M -> type = 1이다. 동적할당을 사용하여 해당 사이즈 크기만큼의 그래프를 만들고, 만약에 type과 size에 잘못된 값이 입력되었거나 아무런 값도 입력되지 않으면 false를 return한다. 이는 Manager 함수에서 false를 입력받은 경우 에러코드 100을 출력하도록 되어있다. 그 다음 type=0, 1에 따라 두가지 분기로 나뉜다. Type=0일 때는 그래프를 List 형식으로 입력받은 것이므로 getline으로 str과 str2 인자 2개를 입력받는다. 만약 str2가 null이면 인자 하나만을 받은 것이므로 이는 From vertex가 된다. 만약 str2에 값이 있으면 인자를 2개 받은 것이고 각각 To vertex, weight의 값이므로 아까 저장해둔 From vertex와 함께 그래프에 insert 함수를 사용하여 값을 저장한다. 만약 type=1이면 graph 형식이 matrix인 것이므로 getline을 통해 size만큼의 인자들이 나온다. Getline, strtok으로 나눈 인자들을 받고, 그 전에 0으로 초기화해둔 from, to가 각각 row, column이 되어 값을 증가시키며 입력받는다. 이때, 입력받는 값은 weight이며, insert 함수를 사용하여 matrix 형태로 저장한다. 모든 동작이 완료되면 true를 반환한다.

2. PRINT



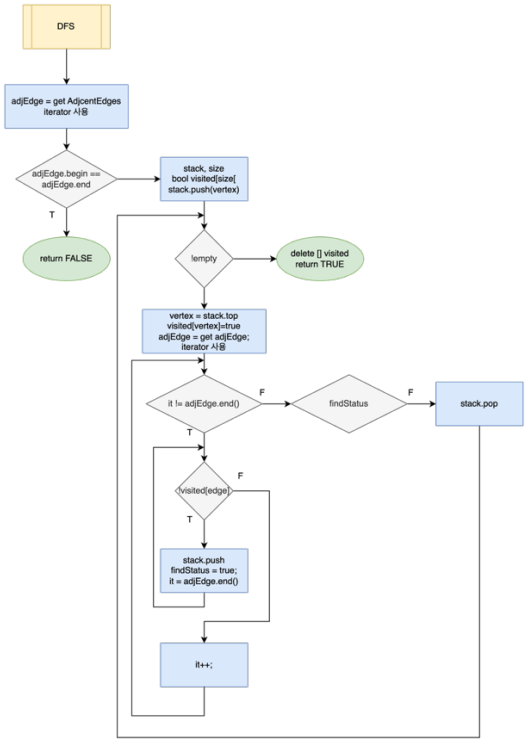
PRINT 명령은 다음과 같은 flow를 가진다. Graph가 없으면 FALSE를 반환하고, graph 정보가 저장되어 있으면 각 type에 맞는 printGraph를 통해 graph를 출력한다. Type=0(List) 이면 리스트 형태로 출력하고, type=1(matrix)이면 행렬 형태로 출력한다.

3. BFS



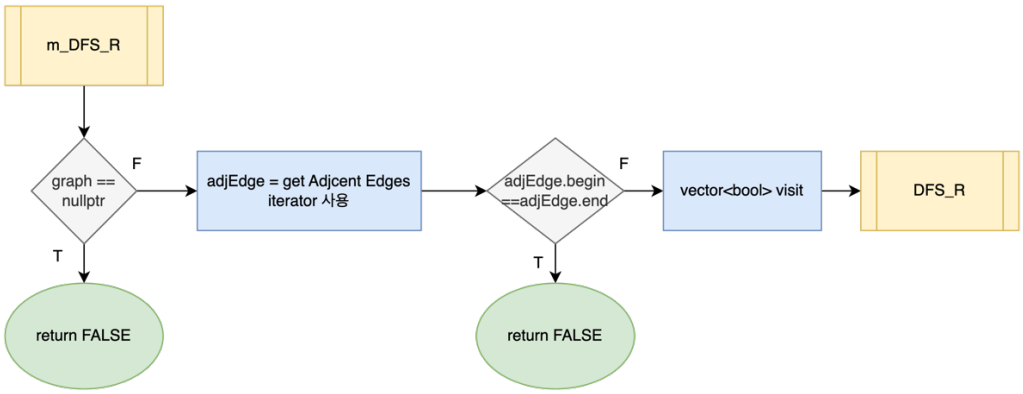
BFS명령어의 Flow chart는 다음과 같다. BFS명령어를 실행하면 queue, size, 그리고 bool 타입의 1차원 배열 visited를 선언한다. 그리고 인자로 받은 시작 vertex를 queue에 push한다. 그리고 queue가 비어있지 않는 동안 다음 동작을 반복한다. 먼저 queue의 맨 앞에 있는 값이자 인자로 입력받은 시작 vertex를 true로 변경하고, queue에서 뺀 다음, 해당 vertex의 adjcentEdge를 구한다. 인접한 edge에 차례로 접할 것이다. 인접한 edge가 방문했던 것인지 아닌지를 체크하고, 아직 방문하지 않은 vertex이면 그 edge를 queue에 넣고 해당 edge의 방문 확인을 true로 변경한다. 그리고 다음 인접한 edge에 대하여 체크를 진행한다. 모든 동작이 진행된 다음에는 bool 타입의 visitied 를 메모리해제 하고 TRUE를 반환한다.

4. DFS



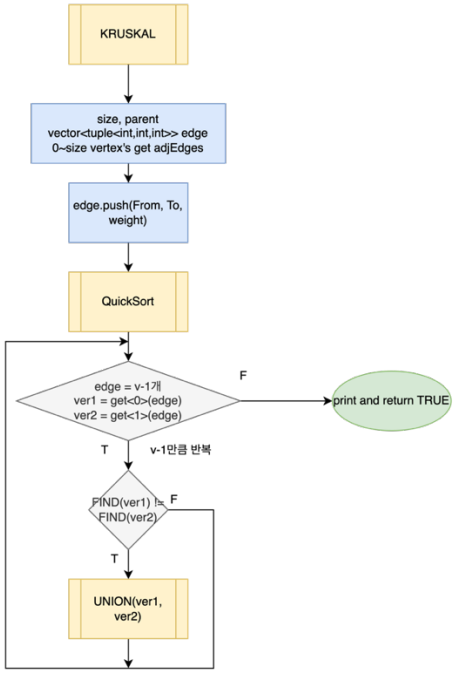
DFS의 flow chart는 다음과 같다. DFS 명령어와 함께 입력받은 vertex의 인접 edge들을 getAdjcentEdges 함수를 사용하여 얻는다. 그리고 만약 해당 adjEdge에 iterator를 사용하여 begin과 end가 같다면, 해당 vertex에 인접한 vertex가 없다는 뜻이고, 혹은 해당 vertex가 그래프에 없다는 뜻이므로 FALSE를 반환하도록 하였다. 만약 그렇지 않으면 다음 단계로 넘어간다. 다음 단계는 stack과 size를 정하고 해당 size만큼의 bool type visited [size]를 선언한 다음 stack에 vertex를 push한다. 그리고 stack이 비었는지 확인하여 빌 때까지 동작을 반복한다. Stack.top을 vertex에 넣고, visited[vertex]=true로 하여 방문하였다고 체크하고, 해당 vertex의 인접 edge들을 다시 구하여 iterator를 사용하여 다음으로 방문할 vertex를 선택한다. Graph의 edge들은 map에 저장되어 있기 때문에 key값을 기준으로 내림차순으로 자동 정렬이 되어있고, 따라서 가장 처음에 불러와지는 것이 가장 값이 작은 vertex이다. 해당 vertex의 visited를 확인하여 방문하지 않은 노드면 stack에 push를 통해 넣은 다음, 찾았다는 의미로 findStatus를 true로 변경한다. 그리고 더이상 이 vertex의 인접한 노드를 찾을 필요가 없으므로 iterator it = end로 변경한 다음 찾은 인접 edge를 기준으로 다시 방문하여 이 edge의 인접 vertex를 찾기 시작한다. 만약, 해당 vertex의 모든 인접 vertex가 방문한 상태이면 pop을 활용하여 해당 vertex를 stack에서 제거하고 이전 방문한 vertex로 이동하여 다음 인접 vertex에 대하여 조건을 판별한다. 만약, 모든 vertex에 방문한 상태라면 계속해서 pop 동작이 이루어질 것이고, 그렇게 empty 상태가 되면 visited를 메모리 해제하고 TRUE를 반환하며 다음 명령어를 입력받는다.

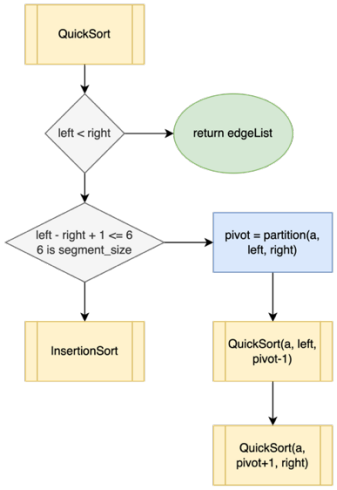
5. DFS_R



DFS_R의 Flow chart는 다음과 같다. DFS_R은 DFS와 기능은 같지만 DFS가 stack으로 구현하였으면, DFS_R은 Reculsive function을 사용하여 구현한 것이다. 즉, DFS_R의 flow를 자세히 설명해보자면, 가장 먼저 graph가 존재하는지 체크한 다음, 없으면 false를 반환하고 그렇지 않으면 다음 동작을 실행한다. 입력받은 vertex의 인접 edge들을 구한 다음, iterator를 사용하여 해당 vertex의 인접 edge가 있는지 체크한다. 여기서 getAdjacentEdges 함수는 방향성이 없는 인접 노드들을 모두 불러오는 것이므로, 해당 vertex에서 시작하는 노드와 해당 vertex로 들어오는 노드를 모두 포함한 것이다. 그렇기 때문에 iterator의 being과 end가 같다면 나가거나 들어오는 edge가 1개도 없다는 것이고, 이는 존재하지 않는 vertex이거나 인접한 edge가 없다는 뜻이므로 false를 반환하여 알맞은 오류코드(500)을 출력한다. 만약, 인접한 edge가 있으면 vector<bool> visit를 선언한 다음, 이 visit과 graph, vertex(인접한 vertex)로 들어가서 함수를 실행한다. 위의 DFS_R은 manager class의 mDFS_R 함수에서 DFS_R이 처음 선언되기 전까지의 흐름을 나타낸 것이고, DFS_R의 세부 흐름은 DFS와 비슷하다. 인자로 받은 vertex의 방문 여부를 체크하여, 아직 방문하지 않은 노드이면 해당 노드의 visit을 true로 바꾼 뒤 이를 인자로 하여 다시 DFS_R 함수를 호출하여 재귀적으로 구현하면 된다.

6. KRUSKAL



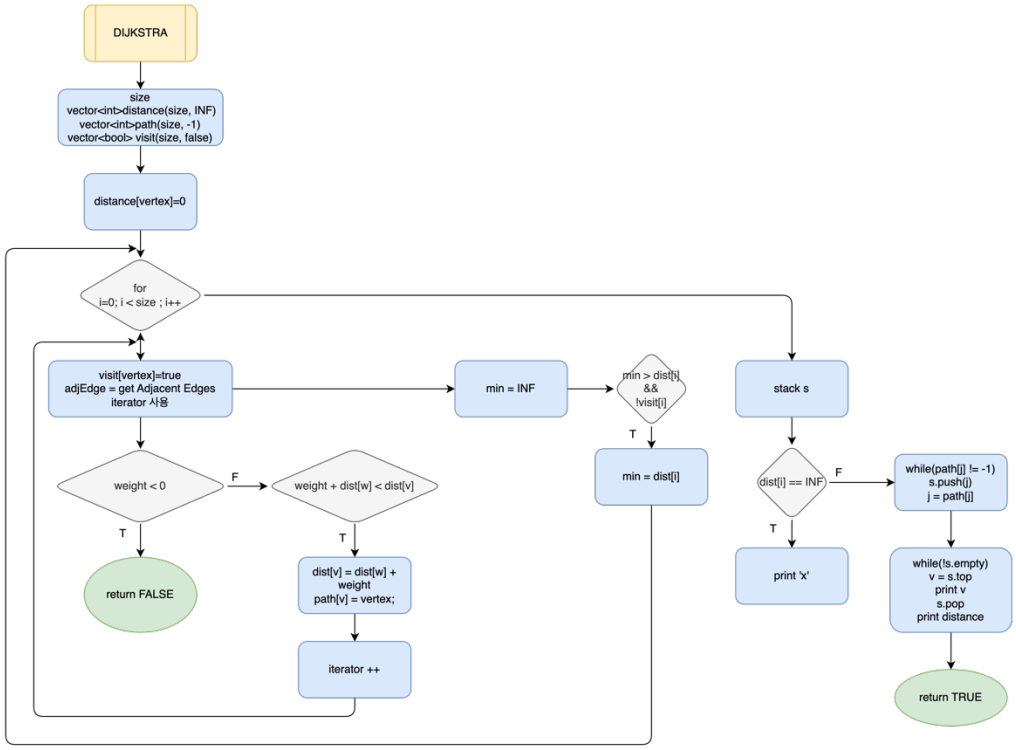


위 그림은 KRUSKAL의 flow chart와 Kruskal 명령어에서 사용하는 함수 중 QuickSoRt 함수에 대한 flow chart이다. 먼저, kruskal에서 size와 parent 변수를 설정한다. Parent는 각각의 vertex들이 edge에 의해 연결되면 하나의 vertex를 parent로 생성해 트리 형태처럼 만들 것이다. 0부터 size까지의 vertex들에 모두 접근하여 vertex의 인접 edge들을 모두 구한다. 이때의 edge 역시 방향성이 없기 때문에 시작하는 vertex, 들어오는 vertex 모두 구하여 edge로 가져온다. 그리고 0부터 size까지의 vertex들의 edge들을 모두 한 곳에 저장하기 위해 새로운 vector<tuple<int, int, int>> edge를 선언한다. Tuple은 3개의 값을 저장하기 위해 사용하였고, 맨 앞부터 차례대로 From vertex, To vertex, weight들을 저장한다. 방향성이 없는 edge들을 각각의 vertex입장에서 모두 가져온 것이므로 중복되는 edge들이 2개씩 존재한다. 해당 edge들이 저장된 상태에서 QuickSort 함수를 실행시키면, 오름차순으로 정렬된 상태이다. 이때 spanning tree는 vertex의 개수가 v라 할 때 v-1개의 edge를 가지므로, 총 v-1번 반복하여 spanning tree를 형성하면 된다. 정렬된 edge의 맨 처음부터 끝까지 접근한다. 접근한 다음 get함수를 사용하여 0번째 value, 1번째 value를 각각 ver1, ver2로 설정한다. 즉, ver1은 from, ver2는 to vertex이다. 그리고 ver1과 ver2의 find 함수를 통해 각각의 parent함수를 확인하고, 만약 서로 다른 parent를 가지고 있으면 union함수를 통해 ver1과 ver2를 하나의 set에 포함시킨다. 역시서 FIND와 UNION 함수에 대해 더 자세히 설명해보자면, FIND함수는 입력받은 vertex의 parent함수를 찾아서, parent값이 자기 자신이 될 때까지 vertex를 업데이트한다. 즉 vertex = parent[vertex]를 진행하는 것이다. 그러면 가장 상단에 있는 parent값이 불러와질 것이고, 이를 기준으로 FIND,

UNION 함수를 실행하는 것이다. UNION 함수는 입력받은 두 개의 인자 ver1, ver2를 하나의 Set으로 합치는 함수이다. Ver1을 기준으로 ver2가 들어가는 것이므로 parent[ver2]=ver1을 진행한다. 만약 find(ver1) == find(ver2)인 경우에는 이미 set인 상태이기 때문에 cycle이 형성되므로 이때는 union함수를 실행시키지 않는다. Union 함수가 실행될 때만 count와 cost를 계산하여 count가 v-1번 진행되면 지금까지 저장된 Edge(quickSort를 통해 오름차순으로 정렬되어 있음.)를 내림차순으로 출력한 다음, cost값도 함께 출력한다. 그리고 TRUE를 반환한다.

QuickSort함수에 대해 간략하게 설명해 보자면, 인자로 입력받은 left와 right값을 비교하여 left < right일 때만 함수를 실행한다. 만약 그렇지 않을 때는 edgeList를 반환한다. 이 edgeList는 정렬된 edge list 정보를 저장하고 있다. Left<right일때 함수가 실행되면 left – right + 1 <= segment_size 인지 체크한다. 만약 segment_size보다 작으면 시간복잡도를 고려하여 InsertionSort함수를 실행하고, 그렇지 않으면 pivot을 구하고 partition을 진행한 다음 두개의 파트로 나누어 quickSort함수를 recursive하게 다시 실행시킨다.

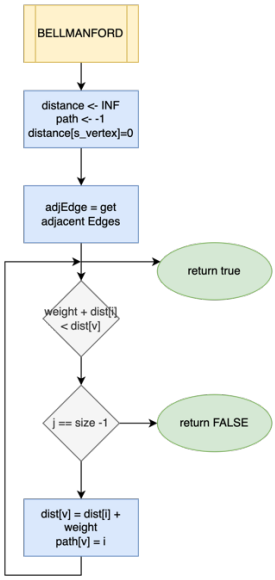
7. DIJKSTRA



DIJKSTRA 명령어에 대한 flow chart이다. 먼저, size를 받아오고, distance, path, visit에 대한 vector를 선언한다. Distance는 INF로 초기화하고, path는 -1로, visit은 false로 초기화한다. 시작 vertex의 distance=0으로 설정한 다음, 동작을 수행한다. Size만큼 반복

할 것이고, visit[vertex]=true로 해두고, 해당 vertex의 인접 edge를 불러온다. 그리고 iterator를 사용하여 모든 edge에 접근한다. 그리고 weight값이 음수인지 판별한 다음, 음수이면 FALSE를 반환하고 에러코드를 출력한다. 음수가 아니라면 weight와 저장된 w값을 더한 값이 이미 저장되어있는 v의 distance값보다 작으면 이 값으로 업데이트를 한다. 그리고 현재의 min값을 구하여 이 min값이 가리키고 있는 vertex에 접근하여 방금 설명한 동작을 size만큼 반복한다. 모든 반복이 끝나면 stack을 사용하여 도착 vertex부터 역순으로 경로를 찾아가기 위하여 path를 참고한다. 먼저 Stack에 넣고, path가 가리키는 곳으로 가 다시 stack에 넣는 것을 반복한다. 만약에 도착 vertex의 dist가 INF이면 시작부터 도착까지 연결되는 것이 없는 것이므로 x를 출력하고 그것이 아니라면 path를 참고하며 stack에 넣는다. 모두 넣은 다음에는 stack이 빌 때까지 top을 출력하고 pop을 진행하는것을 반복한다. Stack이 비면, 도착 vertex까지 걸리는 distance를 출력하고 TRUE를 반환한다.

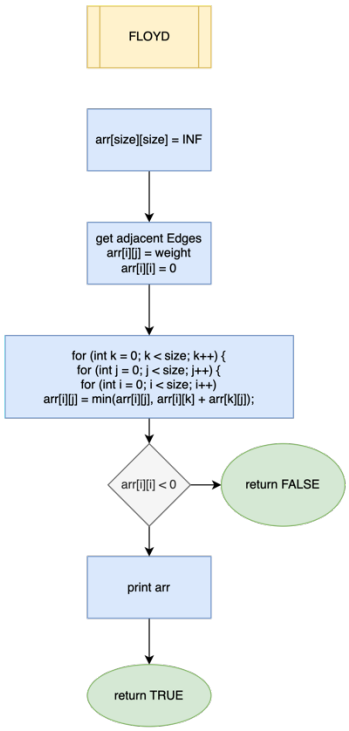
8. BELLMANFORD



BELLMANFORD명령어의 Flow chart는 다음과 같다. 먼저, distance를 모두 INF로 초기화하고 path를 모두 -1로 초기화한다. 그리고 시작 vertex의 distance=0으로 설정한다. 그리고 해당 vertex의 인접 edge들을 모두 구한다. 이때의 edge들은 방향성이 있기 때문에 adjacent Edge가 아닌 getAdjacentEdgesWithDirection 함수를 새로 생성하여 사용하였다. 이 함수는 입력받은 vertex에서 출발하는 edge들만 찾아 저장하는 함수이다. Bellman Ford 함수도 마찬가지로 weight와 dist[i]를 합한 값과 dist[v]를 비교하여 더 작은 값으로 distance를 변경한 다음 path[v]=i로 설정하여 다음 vertex에 대하여 탐색한다. 이때, 모든 vertex에 대해 size만큼 반복하는데, 그 이유는 시작 vertex가 무엇이냐에 따라 달라지기 때문이다. Start vertex=0이면 한번만 반복하면 되지만,

start_vertex = 6이면, 6번 vertex부터 시작하기 위해서는 앞서 5번의 반복이 있어야하기 때문이다. 즉, 이때는 j=4인 상태이고 6번 vertex를 돌기 위해 6번째의 반복, j=5가 된다. 최대 j=size-2인 값이면 모든 값이 업데이트가 되고 더이상의 변경이 이뤄지지 않는다. 하지만 음수 사이클이 발생하면 j=size-1일 때도 값이 업데이트가 된다. 따라서 해당 값이 업데이트 되기 위한 조건인 dist > dist + weight 값을 만족시켜서 업데이트가 되기 직전이라면 j의 값을 체크하여 음수싸이클이 발생했다고 보고 FALSE를 반환한다.

9. FLOYD

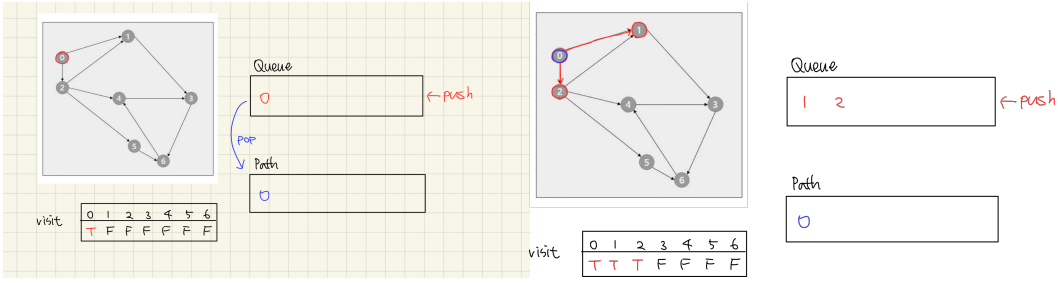


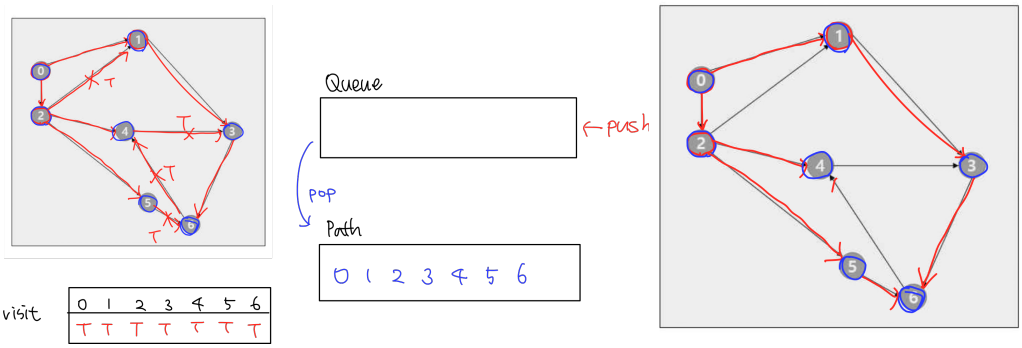
다음은 FLOYD 명령어의 Flow chart이다. Size만큼의 arr을 동적할당을 통해 생성한 다음 해당 값을 모두 INF로 초기화하였다. 그리고 0,0 1,1 처럼 자기 자신으로부터 시작해서 들어가는 값은 모두 0으로 설정하였다. 그다음 3중 for문을 통해 weight값을 업데이트하였다. 모든 업데이트가 끝나고 나더라도 정상적인 그래프는 자기 자신으로 들어가는 값들은 모두 0이어야 하는데, 여기서 값이 음수가 나와버리면 음수 싸이클이 발생했다고 볼 수 있다. 따라서 False를 리턴한다. 음수 싸이클 체크 후에는 arr을 행렬 형태로 출력하고 True를 리턴하여 명령어를 수행한다.

3. Algorithm

1. BFS

BFS는 Breadth First Search 의 약자로 너비 우선 탐색 알고리즘이다. Queue를 사용하는 방법이다. 아래의 그림은 이해를 돕기 위해 그린 그림이다. 시작 vertex가 0일때의 그림이다. 0부터 시작하여 queue에 넣는다. 그리고 0 vertex에 방문한 다음 visit을 True로 변경하고 queue에서 pop하여 path로 출력한다. 그리고 0에서 인접한 vertex들을 작은 순서대로 queue에 넣는다. 해당 그림에서는 1, 2가 순서대로 queue에 Push된다. 1과 2의 visit = true로 변경한다. 그리고 queue에서 가장 맨 앞에 있는 값인 1을 pop하면서 path로 출력하고, 1에 방문한다음 1에서 인접한 vertex를 찾아 똑같은 방식으로 queue에 삽입한다. 만약 인접한 vertex들 중에 visit=T인 값들은 queue에 넣지 않는다. F인 값들만 queue에 넣는다.

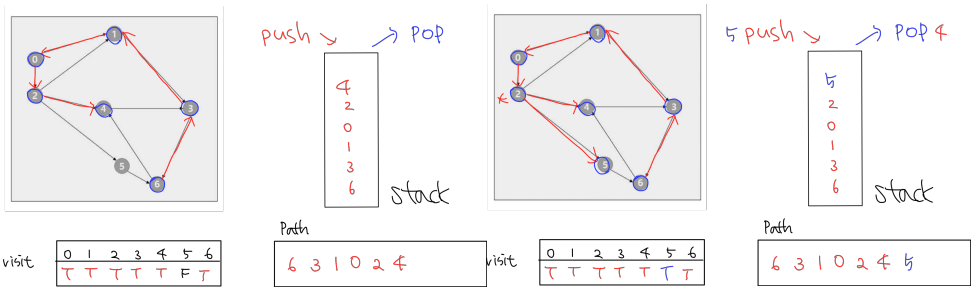




2. DFS

DFS는 Depth Frist Search (깊이 우선 탐색)의 약자로 재귀 함수를 사용하는 것과 stack을 이용하는 것 2가지 방법이 있다. 아래의 그림은 stack을 이용하여 dfs를 구하는 과정을 나타낸 그림이다. 시작vertex=6이라 할 때 6에서 인접한 vertex들 중 가장 작은 값을 기준으로 이동하고 stack에 push한다. 이때, dfs는 방향성이 없는 vertex를 기준으로 하기 때문에 화살표는 상관할 필요 없다. Stack에 push한 다음 그 값들이 있는 visit은 true로 변경을 한다. 그렇게 6, 3, 1, 0, 2, 4까지 path로 출력한다. 그다음 4의 입장에서는 인접한 vertex들이 모두 True의 상태로 갈 수 있는 곳이 없다. 이때는

stack에서 pop을 한 다음 stack의 top(2)를 참조하여 다시 그 vertex로 간다. 그리고 그 vertex(2)에서 인접한 vertex들 중 갈 수 있는 vertex를 선택하여 visit=true로 바꾸고 stack에 push한 다음 path로 출력한다. 모든 vertex를 방문한 상태이기 때문에 5에서는 갈 수 있는 vertex가 더이상 없다. 따라서 stack에서 pop이 이루어지고 마지막에 stack은 비어있는 상태가 된다.



3. Kruskal

Kruskal은 MST를 생성하는 알고리즘이다. MST는 Minimum Spanning Tree의 약자로 Spanning Tree에서 edge의 weight 합이 최소가 되는 트리이다. Spanning Tree이기 때문에 vertex가 N개라면 N-1개의 edge를 가지고 있고, 이 edge를 이용하여 모든 vertex를 연결하는 그래프이다. 모든 vertex가 연결되어있지만 사이클은 갖지 않는 그래프이다. Kruskal 알고리즘을 수행하는 방법은 다음과 같다.

1) 모든 edge를 weight 기준으로 오름차순 정렬
2) Weight가 작은 edge부터 그래프에 포함시킨다.
   A. 추가하는 edge로 인해 그래프에 사이클이 발생하는지 여부를 확인
   B. 추가한 edge로 인해 cycle이 발생하면 해당 edge는 그래프에 포함x
3) 모든 vertex가 edge로 연결될 때까지 혹은 N-1개의 edge가 그래프에 포함될 때까지 2)를 반복한다.

추가되는 edge가 사이클을 발생시키는지 확인하기 위해서는 Union, Find함수를 따로 써야한다. Union 함수는 서로 다른 두 set을 합치기 위해 사용한다. 1)에 의해 weight 기준으로 모든 edge들을 오름차순으로 정렬하였을 때는 모든 vertex들이 자기 자신만을 원소로 갖는 set의 형태이다. 그리고 작은 weight 값을 가진 edge부터 그래프에 포함시키는데, 이때 union을 사용하여 vertex1이 속한 set과 vertex2가 속한 set을 합치게 된다. 이때 싸이클이 발생하는지 검사하기 위해 Find를 사용하게되는데, root를 반환하여 서로 다른 두 edge가 같은 셋에 포함되었는지 확인한다.

4. Quick Sort

Quick Sort는 리스트 안에서 한 요소 Pivot을 선택하여 이를 기준으로 왼쪽은 pivot보다 작은 값, 오른쪽은 pivot보다 큰 값들로 구성한다. Pivot의 왼쪽 리스트와 오른쪽

리스트에 다시 Quick Sort를 수행한다. I는 왼쪽에서, j는 오른쪽에서 출발해가며 pivot 보다 작고 큰 값이 있으면 멈춰서 두 값을 swap하면서 계속 진행한다. 그러다가 j < i 가 되면 j를 pivot과 swap한 다음 pivot기준으로 분할하여 왼쪽 리스트와 오른쪽 리스트에 다시 Quick Sort를 수행한다. 더이상 분할이 불가능할 때까지 반복한다. 그러면 오름차순으로 Quick Sort를 진행할 수 있다.

5. Dijkstra

   Dijkstra는 방향성과 가중치가 있는 그래프에서 동작이 가능하며 음수 가중치를 허용하지 않는다. 주어진 vertex에서 다른 vertex 사이의 최단 경로를 구하는 알고리즘이다. 노드들을 방문하면서 획득하는 edge정보를 이용하여 최단거리와 경로를 업데이트하는 방식이다. 일단, 방문했는지 확인하기 위한 visit, 최단거리를 구하기 위해 저장해둘 distance, 그리고 경로를 저장하기 위한 path table들이 필요하다. Distance는 시작 vertex는 0, 그 외에는 모두 infinite값으로 초기화를 하고, visit은 아직 어느것도 방문하지 않았으므로 false(0)으로, path는 -1로 초기화를 한다. 그다음 visit을 기준으로 아직 방문하지 않은 vertex중 가장 가까운 vertex를 선택하고, 선택된 vertex에 연결된 edge정보를 이용하여 다른 vertex까지의 최단거리와 경로를 업데이트한다. 모든 vertex를 방문할 때까지 계속해서 위와 같은 동작을 반복한다. 만약 0에서 시작하면 0에서부터 인접한 vertex까지를 거리를 distance[vertex]에 저장한다. 그리고 distance 중에 가장 작은 값으로 방문하여 visit=1로 두고, path=0으로 한다. 이 vertex에서부터 인접한 vertex까지의 거리와 이 vertex까지 걸린 distance값을 더한 값이 원래 저장되어있던 distance[vertex]보다 작으면 해당 값으로 업데이트 한다. 그리고 path[vertex]를 이 vertex로 한다.그렇게 계속해서 업데이트 하다보면 주어진 vertex에서 다른 vertex까지의 최단 경로를 구할 수 있다.

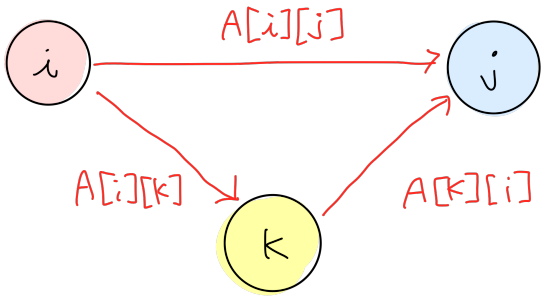
6. Bellman ford

   Bellman ford는 방향성과 가중치가 있는 그래프에서 동작하는 알고리즘이다. 주어진 vertex에서 다른 vertex사이의 최단 경로를 구하는 것으로 음수 가중치가 있어도 동작하지만 음수 사이클(Negative Cycle)이 발생한 경우에는 확인이 가능하다. Bellman Ford를 구현하는 과정은 다음과 같다. 일단 모든 vertex에 연결된 edge정보를 반복적으로 갱신하는 것이 주된 목적이다. 시작 vertex의 거리를 0으로 초기화하고, 이 외의 다른 vertex로의 거리는 무한대로 초기화한다. 그리고 각 Vertex에 대하여 주변 edge로의 거리를 보다 짧은 경로로 갱신한다. 이는 Dijkstra와 비슷하다. 이미 저장되어 있는 값과 해당 vertex를 지나쳐 들어가는 거리를 비교하여 더 짧은 쪽으로 갱신하는 것이다. 시작 vertex의 거리를 0으로 초기화 하고 모든 vertex에 대하여 접근해야 하기 때문에 최대 vertex의 수만큼 반복하여 값을 업데이트해야한다. 그리고 한번 더 반

복하였을 때는, 모든 값들이 업데이트 된 상태이므로 변화되는 것이 없어야하는데, 변화가 생기면 이는 음수 사이클이 발생한 것으로 볼 수 있다.

7. FLOYD

FLOYD 알고리즘은 방향성과 가중치가 있는 그래프에서 동작하는 알고리즘이다. 모든 vertex 쌍에 대한 최단 경로를 구하는 알고리즘으로 Bellman ford 알고리즘과 마찬가지로 음수 가중치가 있어도 동작이 가능하고, 만약 음수 사이클(Negative Cycle)이 발생한 경우에는 확인이 가능하다. Floyd 알고리즘을 구현하는 방법은 다음과 같다. 모든 vertex에 연결된 edge 정보를 반복적으로 갱신하는 것을 목표로 하며, 가장 먼저 모든 vertex를 순차적으로 선택하고, 선택된 vertex를 경유하는 모든 vertex 쌍의 최단 거리를 업데이트한다. 그리고 모든 vertex에 대하여 최단거리를 업데이트 한다.



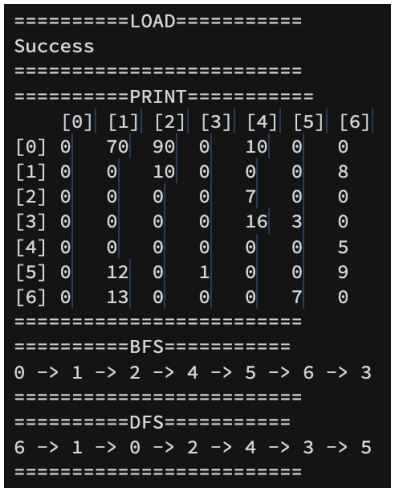
위 그림은 Floyd의 최단거리를 업데이트 하는 방법이다. i에서 j로 갈 때 바로 가는 a[i][j]와 k를 경유하여 가는 a[i][k] + a[k][i]를 비교하여 더 작은 값을 최단 거리로 업데이트 하는 것이다. 자기 자신에게 가는 최단거리는 항상 0이 되는데, 만약, 음수 사이클이 발생하면 자기 자신에게 가는 최단 거리가 음수 가중치를 갖게 되므로 자기 자신에게 가는 최단거리를 가지고 체크할 수 있다.

4. Result Screen

```
1   LOAD graph_M.txt
2   PRINT
3   BFS 0
4   DFS 6
5   LOAD graph_L.txt
6   PRINT
7   BFS 3
8   DFS_R 6
9   KRUSKAL
10  DIJKSTRA 5
11  BELLMANFORD 0 5
12  FLOYD
13  EXIT
```

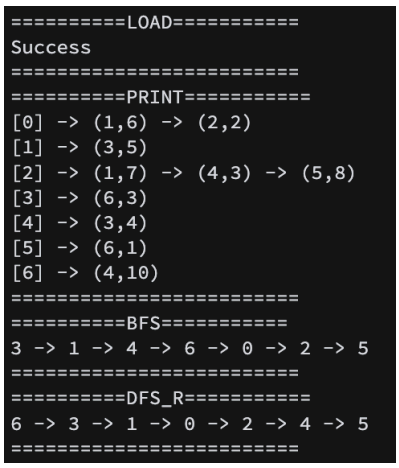
직접 사용한 command.txt의 내용은 다음과 같다. Matrix형태의 그래프를 먼저 불러와 print를 하고 bfs 0, dfs 6을 구한다. 그리고 List 형태의 그래프를 불러와 print를 하고 bfs 3, dfs_r 6, Kruskal, Dijkstra 5, bellmanford 0->5, floyd를 구한 다음

exit을 통해 해당 프로그램을 종료한다. 출력 결과는 log.txt파일에 저장되어있다.



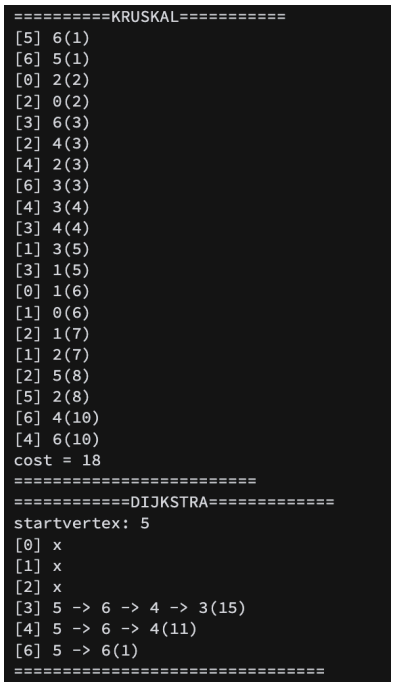
```
==========LOAD==========
Success
========================
==========PRINT==========
    [0] [1] [2] [3] [4] [5] [6]
[0] 0   70  90  0   10  0   0
[1] 0   0   10  0   0   0   8
[2] 0   0   0   0   7   0   0
[3] 0   0   0   0   16  3   0
[4] 0   0   0   0   0   0   5
[5] 0   12  0   1   0   0   9
[6] 0   13  0   0   0   7   0
========================
==========BFS==========
0 -> 1 -> 2 -> 4 -> 5 -> 6 -> 3
========================
==========DFS==========
6 -> 1 -> 0 -> 2 -> 4 -> 3 -> 5
========================
```

matrix 형태의 그래프를 불러와 load, print, bfs0, dfs6을 진행한 결과이다.



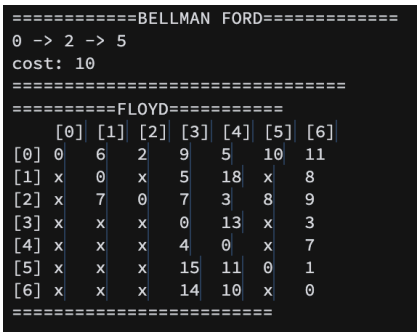
```
==========LOAD==========
Success
========================
==========PRINT==========
[0] -> (1,6) -> (2,2)
[1] -> (3,5)
[2] -> (1,7) -> (4,3) -> (5,8)
[3] -> (6,3)
[4] -> (3,4)
[5] -> (6,1)
[6] -> (4,10)
========================
==========BFS==========
3 -> 1 -> 4 -> 6 -> 0 -> 2 -> 5
========================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

list형태의 그래프를 불러와 load, print, bfs 3, dfs_r6을 한 결과이다. 위에서 matrix 형태의 그래프가 저장되어 있었음에도 불구하고 기존에 존재하던 그래프가 삭제된 후 새로운 그래프가 불려와져 저장된 것을 확인할 수 있다.



```
==========KRUSKAL==========
[5] 6(1)
[6] 5(1)
[0] 2(2)
[2] 0(2)
[3] 6(3)
[2] 4(3)
[4] 2(3)
[6] 3(3)
[4] 3(4)
[3] 4(4)
[1] 3(5)
[3] 1(5)
[0] 1(6)
[1] 0(6)
[2] 1(7)
[1] 2(7)
[2] 5(8)
[5] 2(8)
[6] 4(10)
[4] 6(10)
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)
================================
```

이어서 kruskal과 dijkstra를 출력한 결과이다. Weight값을 오름차순으로 출력한 결과이다. 즉 [5] 6 (1)은 5 -> 6으로 가는데 1만큼의 weight가 걸린다는 뜻이다. Weight를 기준으로 오름차순이 잘 된 것을 확인할 수 있고, cost값은 kruskal을 통해 구한 mst의 최단 경로이다. Dijkstra는 5를 startvertex로 하여 3까지는 연결이 되지만 0, 1, 2는 연결이 되지 않아 끊긴 것을 확인할 수 있다. 자기 자신으로 가는 값은 당연히 없기 때문에 생략하였다.



이어서 Bellman ford와 floyd를 수행하여 출력한 결과이다. Bellman ford는 0에서 시작하여 5까지 가는 vertex의 최단 경로와 최단 거리를 구한 값이다.
Floyd는 모든 vertex 쌍에 대하여 최단 거리를 구한 것이다. 방향성이 있기 때문에 대칭되지 않고, 자기 자신으로 들어가는 값은 항상 0이다.

5. Consideration

가장 먼저 난관을 겪었던 구간은 이번에도 스켈레톤 코드를 이해하는 부분이었다. Graph를 상속받아 ListGraph와 MatrixGraph에 있는 getAdjacentEdges 함수가 어떻게 쓰이는지 이해가 가지 않았다. insertEdge도 edge를 추가하는 함수인데 getAdjacentEdges는 이미 추가 된 그래프에서 vertex를 가져오는 것이 아닐까라는 추측을 하였지만, github에 이미 올려진 issue를 봤을 때 해당 함수가 vertex와 인접한 edge들을 저장하는 것이라는 답변이 있어 혼란스러웠다. 그렇다면 insertEdge와 getAdjacentEdges 모두 edge를 저장하는 것이라면 기능이 겹친다거나 중복되는 값들이 저장되는 것은 아닐까 싶었다. 그래서 오랜 시간 코드를 해석하는데 시간을 들였고, 결국 알아낸 것은 함수 밖에서 m_List 또는 m_Mat을 가지고와서 처리할 수 없기 때문에 인접한 vertex들을 저장하여 밖으로 꼬내는 역할을 하는 것이라는 거다. Edge를 저장한다고만 설명을 해서 너무 헷갈렸지만 코드의 이름을 기준으로 생각했으면 금방 풀릴 의문이었던 것 같다. 하지만, 여기서 시간을 많이 쏟아서 조금 아쉬운 부분이 있다. 그다음, BFS, DFS처럼 방향성이 필요 없는 경우 인접한 edge들을 구할 때 인자로 받은 vertex에서 출발하는 edge뿐만 아니라 vertex로 도착하는 edge들도 모두 구해야 했기 때문에 코드를 크게 수정하였었다. 처음에는 그저 출발하는 edge만 구하였기 때문이다. 고칠 때는 graph size만큼 0부터 size까지 vertex를 모두 접근하여 map<int, int>에서 value값이 입력받은 vertex와 같을 때의 vertex와 해당 vertex에서 접근 가능한 edge들의 모든 value값을 인자로 전달받은 새로만들어진 map<int, int>에 저장한 다음 iterator를 이용하여 사용하였다. 그 다음으로 어려움을 겪었더 부분은 Dijkstra이었다. Dijkstra에서는 방향성이 있기 때문에 현재 존재하는 getAdjacentEdges 함수를 사용할 수 없었다. 그래서 고안한 방법이 Graph에는 virtual로, 그리고 ListGraph와 MatrixGraph에는 함수로 getAdjacentEdgesWithDirection을 만드는 것이었다. 이 함수는 방향성을 가지는 edge들을 저장하고 있기 때문에 인자로 입력받은 vertex에서 출발할 수 있는 인접한 edge들만을 찾아서 weigth값과 함께 인자로 받은 새로운 map<int, int>에 저장한다. 이렇게 찾은 방향성이 있는 edge들을 가지고 최단 경로를 찾았다. 제일 어려웠

던 부분은 Kruskal이었다. Kruskal에서는 방향성이 없었지만 모든 vertex에 있는, 즉 그래프에 존재하는 모든 edge들을 한 곳에 저장하여 QuickSort와 InsertionSort를 진행해야 했다. Edge들을 한 곳에 저장하기 위해 처음 생각했던 것은 map<int, pair<int, int>>였다. 첫번째 value가 weight, second = from vertex, third = to vertex였다. 하지만 실패했다. 바로 Map의 특성을 잊고 있었기 때문이다. Map은 첫번째 값을 기준으로 자동으로 정렬하고 중복된 값을 허용하지 않는다. 따라서 weigth값은 중복되는 것이 분명 있을 수 있지만, 이를 허용하지 않기 때문에 값이 이상해진다. 그 다음으로 생각한 방법은 List<pair<int, int>, int>이다. First=From, second=To, Third=weight 값이다. 이 방법으로 구현을 진행하다가 tuple의 존재를 알게 되었다. Tuple은 값을 3개를 한꺼번에 넣을 때 사용할 수 있는 것이다. 따라서 vector<tuple<int, int, int>>를 사용하여 edge들을 한번에 관리하였고 이를 통해 Sort 함수들을 사용할 수 있었다.