데이터구조실습 3차 프로젝트 결과보고서

학　　과: 컴퓨터공학과

담당교수: 이형근 교수님

실습분반: 월요일 7,8 교시

학　　번: 2021202071

성　　명: 이민형

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-Ford 그리고 FLOYD 연산을 수행한다. 그래프 데이터는 방향성과 가중치를 모두 가지고 있으며, 데이터 형태에 따라 List 그래프와 Matrix 그래프로 저장한다. DFS와 BFS는 그래프의 방향성과 가중치를 고려하지 않고, 그래프 순회 또는 탐색 방법을 수행한다. BFS는 큐를 사용하여 구현하고, DFS는 스택을 사용하여 반복문으로 구현하는 방법과 재귀함수를 통해 재귀적으로 구현하는 두 가지 방법으로 구현한다. Kruskal 알고리즘은 최소 비용 신장 트리(Minimum Spanning Tree)를 만드는 방법으로 방향성이 없고, 가중치가 없는 그래프 환경에서 수행한다. Dijkstra 알고리즘은 정점 하나를 출발점으로 두고 다른 모든 정점을 도착점으로 하는 최단경로 알고리즘으로 방향성과 가중치가 모두 존재하는 그래프 환경에서 연산을 수행한다. 만약 가중치가 음수의 값을 가지면 에러코드를 출력한다. Bellman-Ford 알고리즘은 시작 Vertex와 도착 Vertex를 인자로 받으며 최단 경로와 거리를 구한다. 가중치가 음수인 경우에도 정상적으로 동작해야 하며, 음수 가중치에 사이클이 발생한 경우에는 알맞은 에러코드를 출력한다. 마지막으로 Floyd는 입력 인자가 따로 존재하지 않으며 모든 쌍에 대해서 최단 경로 행렬을 구한다. 가중치가 음수인 경우에도 정상적으로 동작해야 하며, 음수 가중치에 사이클이 방생한 경우에는 알맞은 에러코드를 출력한다.

다음으로 그래프 데이터에 대해서 알아보도록 하겠다. 본 프로젝트에서는 인접 리스트, 인접 행렬을 이용하여 그래프를 구축한다. 인접 리스트와 인접 행렬의 텍스트 파일의 첫 번째 줄에는 그래프의 형식 정보가 저장되어 있고, 두 번째 줄에는 그래프 크기가 저장되어 있다. 이후 데이터는 그래프 형식에 따라서 구분된다. 그래프 형식은 다음과 같다.

<인접 리스트 그래프의 데이터 형식>

형식	내용
1	시작 vertex
2	[도착 Vertex] [Weight]

<인접 행렬 그래프의 데이터 형식>

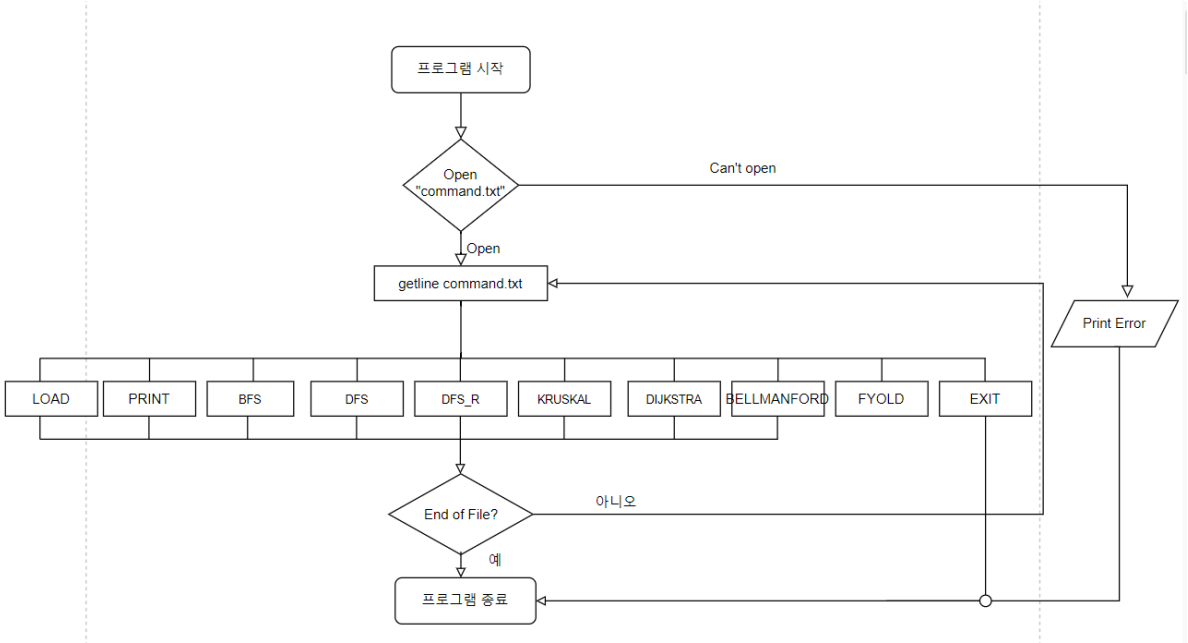
형식	내용
3	[Weight_1] [Weight_2] [Weight_3] [Weight_4] .....

마지막으로 정렬연산이다. 본 프로젝트에서 정렬 연산은 Kruskal 알고리즘의 edge를 오름차순으로 정렬하는 데에 사용한다. 연산의 효율을 향상하기 위해 segment size에 따라 정렬 알고리즘을 구현한다. 해당 프로젝트에서는 segment size가 7 이상 일때는 퀵 정렬을 수행하고, 6 이하일 경우에는 삽입 정렬을 수행한다.

Flow Chart

<RUN>

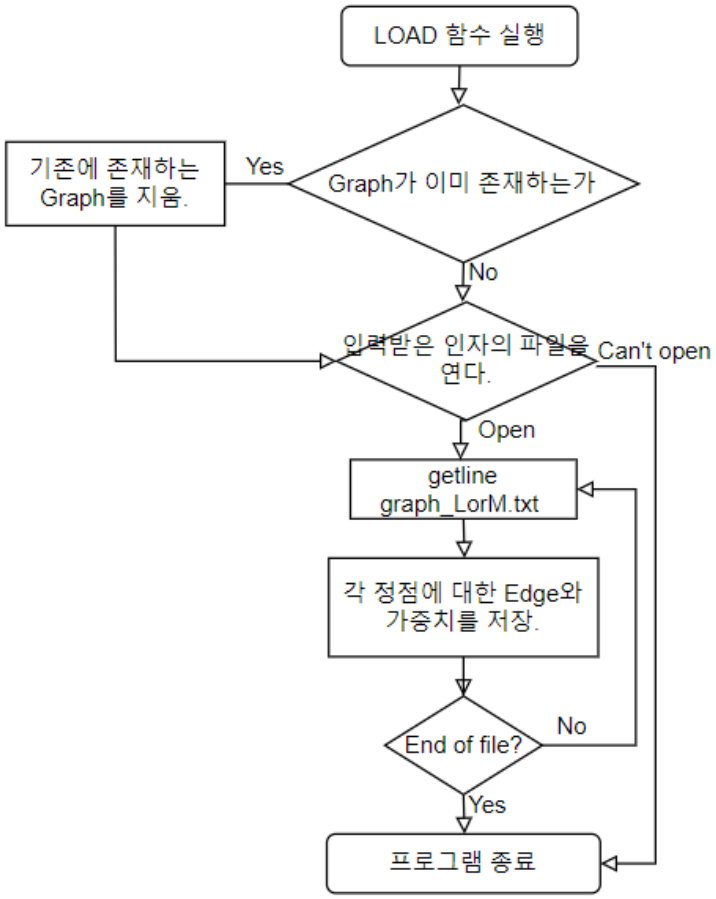
우선 모든 클래스들을 관리하는 Manager 헤더파일의 Run함수 Flow Chart는 다음과 같다.



프로젝트의 실행을 수행하는 RUN함수는 Command.txt 파일을 읽어서 각 명령어에 따른 연산을 수행한다. LOAD 명령어의 경우 인자로 파일 명을 같이 받아서 인접 행렬로 그래프를 구현할 지 아니면 인접 리스트로 그래프를 구현할 지 결정한다.

<LOAD>

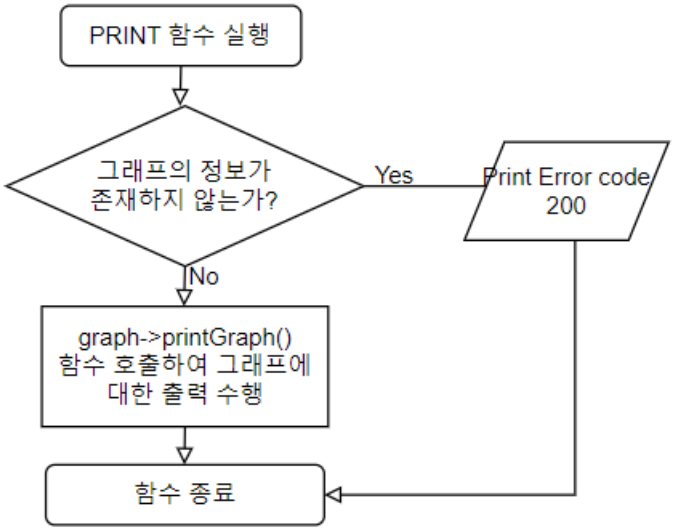
다음은 LOAD 명령어의 Flow Chart이다. LOAD 명령어는 인자로 받은 파일 명에 따라서 인접 행렬과 인접 리스트 중 하나로 그래프를 구현한다. 만약 이전에 생성된 그래프가 이미 존재한다면 존재하는 그래프를 삭제하고 다시 그래프를 만든다. 또한 텍스트 파일이 존재하지 않거나 비어있는 경우, 100의 오류코드를 출력한다.



<LOAD의 Flow Chart>

<PRINT>

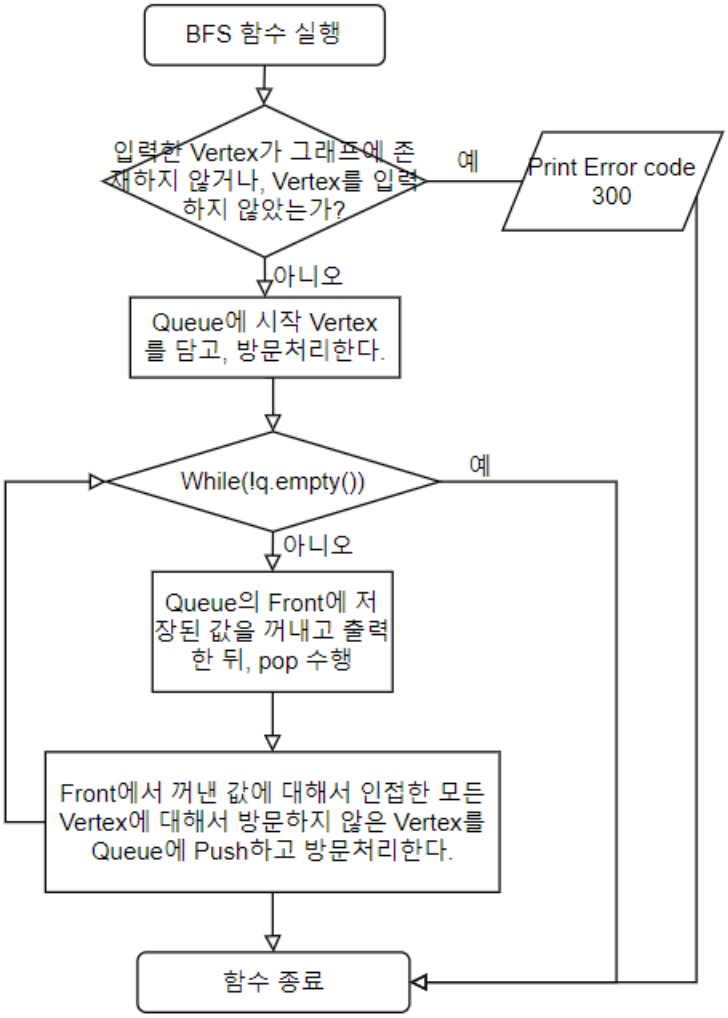
다음은 PRINT 명령어의 Flow Chart이다. PRINT 명령어는 그래프의 형식에 따라 그래프를 출력하는 명령어이다. 그래프가 존재하지 않을 경우 200의 에러코드를 출력하였다.



<PRINT의 Flow Chart>

<BFS>

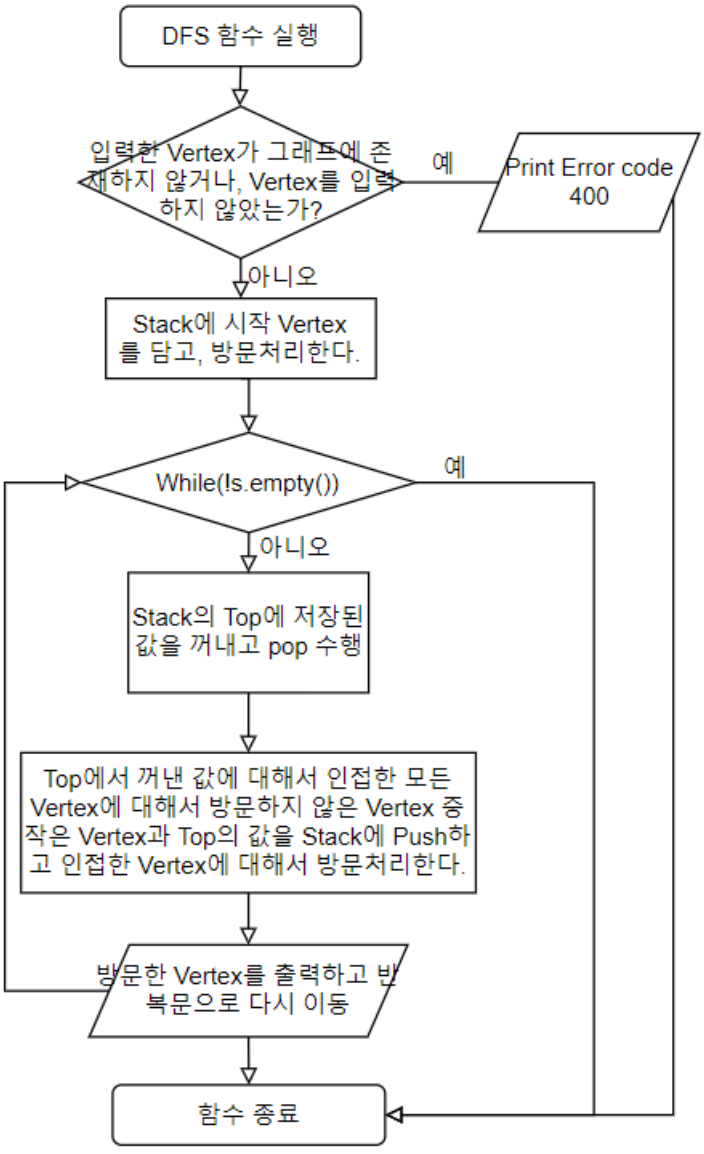
다음은 BFS 명령어이다. 인자로 시작 Vertex를 받아서 수행한다. 너비 우선 탐색 알고리즘으로 큐를 사용하여 구현하였다. 그래프의 정보가 존재하지 않거나, 인자로 Vertex를 입력하지 않거나, 입력한 Vertex가 존재하지 않을 경우에는 300의 에러 코드를 출력한다.



<BFS의 Flow Chart>

<DFS>

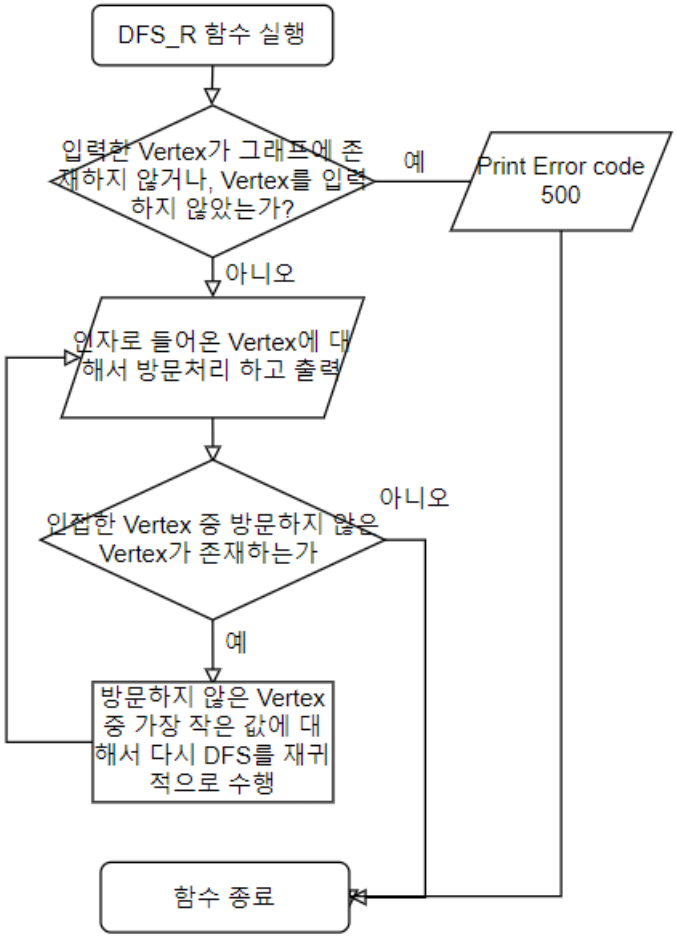
다음은 DFS 명령어이다. 깊이 우선 탐색 알고리즘으로 스택을 사용하여 구현하였다. 그래프이 정보가 존재하지 않거나, 인자로 Vertex를 입력하지 않거나 입력한 Vertex가 존재하지 않을 경우 400의 에러코드를 출력하였다.



<DFS의 Flow Chart>

<DFS_R>

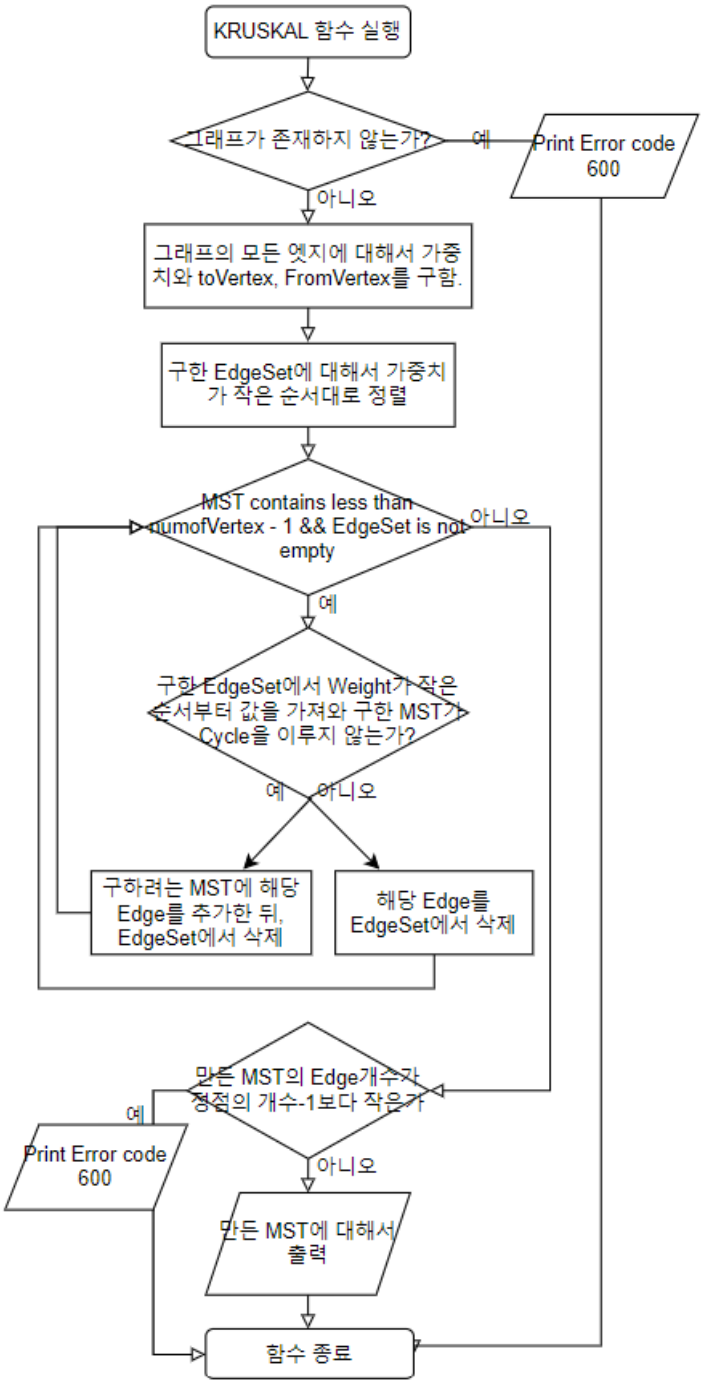
다음은 DFS_R 명령어이다. 이 명령어는 깊이 우선 탐색을 반복문을 사용하여 수행하지 않고 재귀함수를 통하여 구현하는 명령어이다.



<DFS_R의 Flow Chart>

<KRUSKAL>

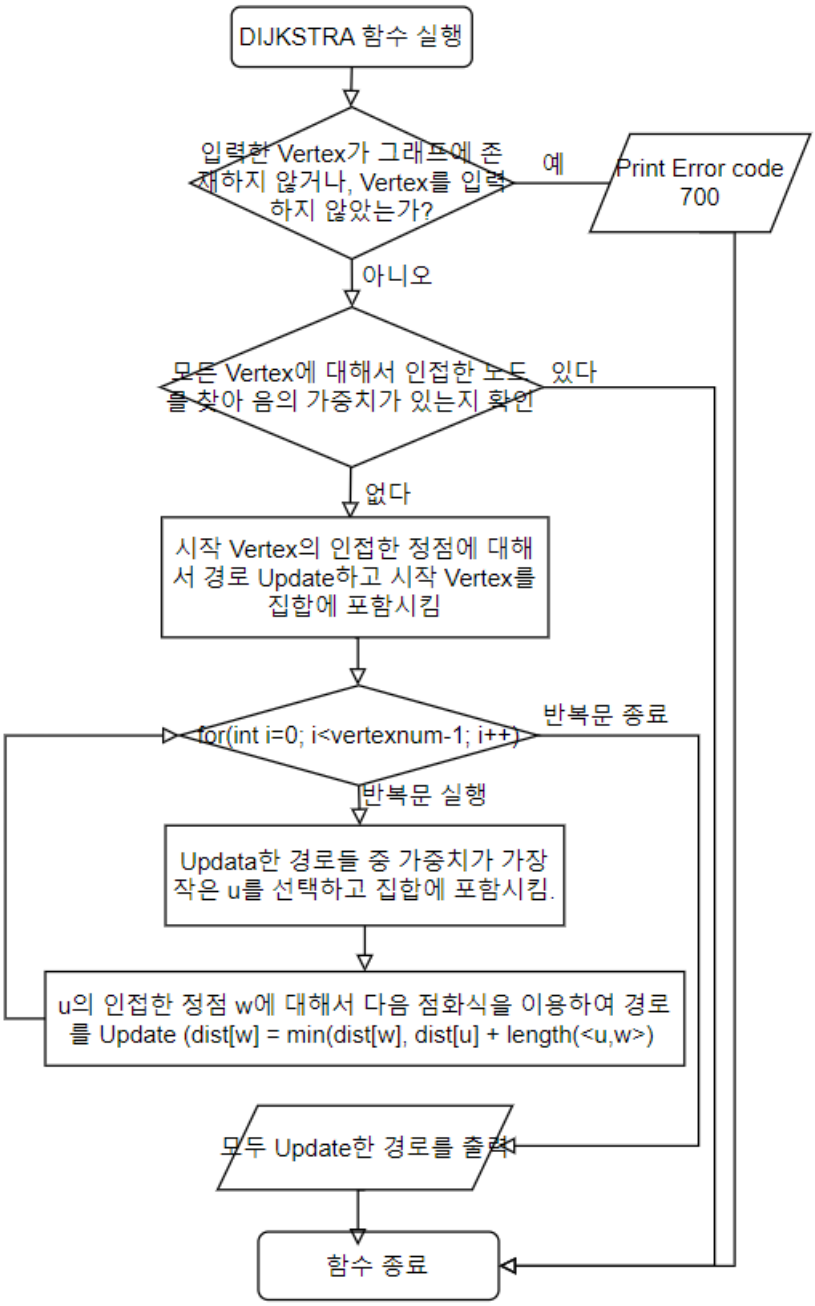
다음은 KRUSKAL 명령어이다. KRUSKAL 명령어는 현재 그래프의 MST를 구하고, MST를 구성하는 vertex들의 Weight와 Edge를 출력하고, weight들의 총합을 출력한다. 입력한 그래프가 MST를 구할 수 없는 경우, 명령어를 수행할 수 없는 경우 출력파일에 오류코드를 출력한다. 크루스칼을 구현하는데 사용한 알고리즘에 대해서는 Algorithm 파트에서 자세히 설명하도록 하겠다.



<KRUSKAL의 Flow Chart>

<DIJKSTRA>

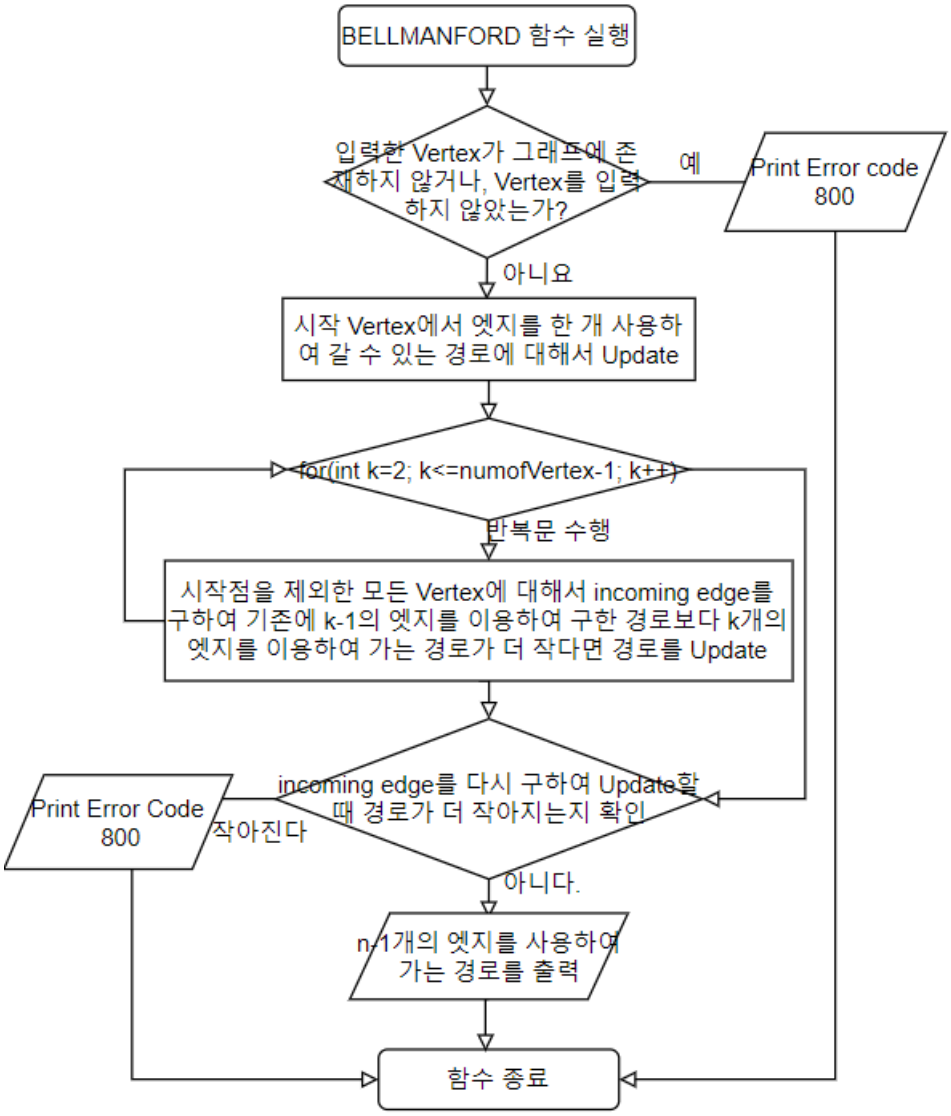
다음은 다익스트라 알고리즘을 수행하는 DIJKSTRA 명령어이다. 인자로 시작 Vertex를 받아서 동작을 수행한다. 명령어의 결과인 shortest path와 cost를 출력하며 vertex, shortest path, cost 순서로 출력한다. 기준 Vertex에서 도달할 수 없는 Vertex의 경우 'x'를 출력한다.



<DIJKSTRA의 Flow Chart>

<BELLMANFORD>

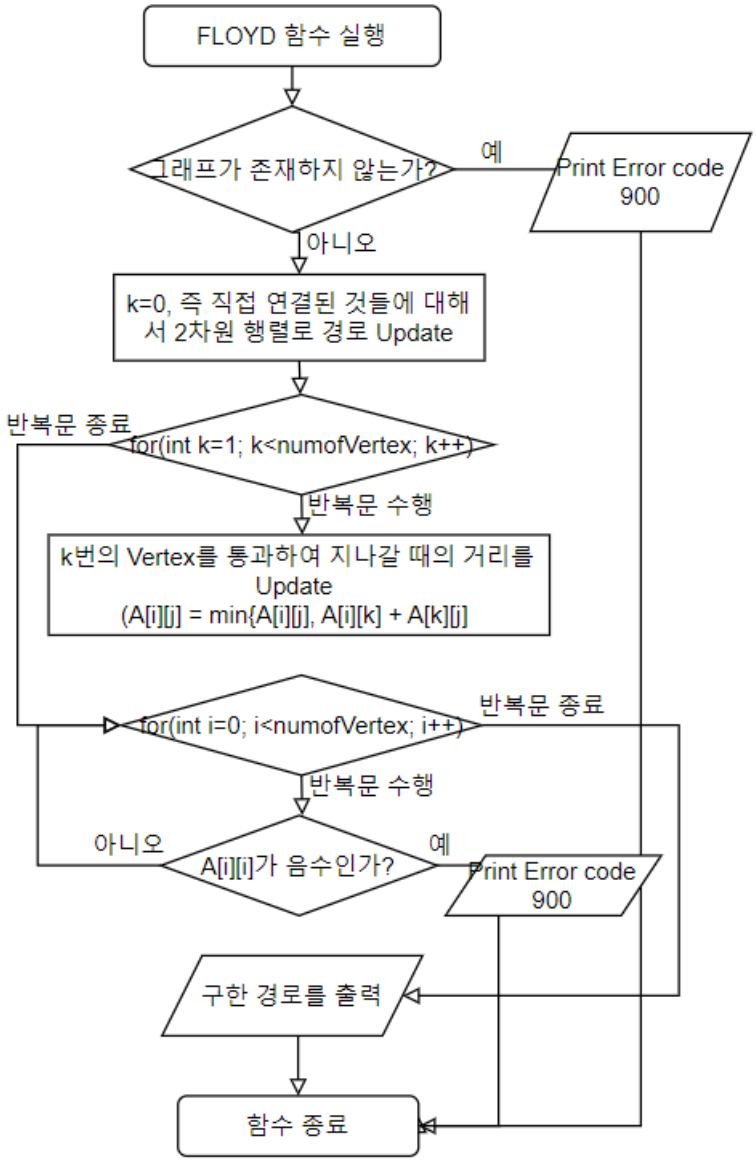
다음은 BELLMANFORD 명령어이다. 인자로 StartVertex와 EndVertex를 받아서 동작을 수행하여 StartVertex부터 EndVertex까지의 최단 경로와 거리를 구하는 명령어이다. 가중치가 음수인 경우에도 정상적으로 동작해야 하며, StartVertex에서 EndVertex까지 도달할 수 없는 경우에는 'x'를 출력한다. 또한 입력한 vertex가 그래프에 존재하지 않거나, 입력한 Vertex가 부족한 경우, 명령어를 수행할 수 없는 경우, 음수 사이클이 발생한 경우 에러코드를 출력한다.



<BELLMANFORD의 Flow Chart>

<FLOYD>

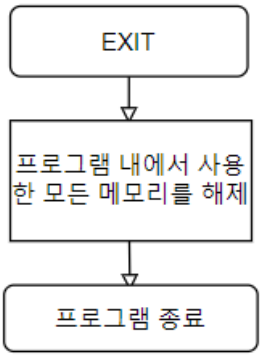
다음은 FLOYD 명령어이다. 모든 vertex 쌍에 대해서 StartVertex에서 EndVertex로 가는데 필요한 비용의 최솟값을 행렬형태로 출력한다. 기준 vertex에서 도달할 수 없는 vertex의 경우 'x'를 출력한다. 명령어를 수행할 수 없거나, 음수 사이클이 발생한 경우 에러코드를 출력한다.



<FLOYD의 Flow Chart>

<EXIT>

마지막으로 EXIT 명령어이다. 프로그램에서 사용한 메모리를 모두 해제하고 프로그램을 종료한다.



<EXIT의 Flow Chart>

Algorithm

다음은 본 프로젝트에서 사용한 알고리즘에 대해서 살펴보도록 하겠다. 본 프로젝트에서는 크게 그래프 연산 알고리즘과 정렬 알고리즘을 사용하였다. 그 중 먼저 그래프 연산 알고리즘에 대해서 살펴보도록 하겠다.

우선 BFS (Breath First Search), 너비 우선 탐색 알고리즘이다. 시작 정점를 방문한 후 시작 정점에 있는 인접한 모든 노드들을 우선 탐색하는 알고리즘이다. 너비 우선 탐색 알고리즘은 큐를 사용하여 구현한다. 우선 탐색 시작 정점을 큐에 넣고 큐에서 시작 정점을 꺼낸 뒤 해당 정점의 인접 노드 중 방문하지 않은 모든 노드를 큐에 삽입하고 방문 처리하는 알고리즘이다. 위를 수행한 뒤. 다시 큐를 꺼내서 해당 정점의 인접한 노드에 대해서 반복하는 알고리즘이다. DFS의 장점은 출발 노드에서 목표 노드까지 최단 길이의 경로를 보장한다는 점이다. 단점은 경로가 매우 길 경우에는 탐색 가지가 급격하게 증가함에 따라 보다 많은 기억 공간을 필요로 한다. 해가 존재하지 않는다면 유한 그래프의 경우, 모든 그래프를 탐색한 후 실패로 끝이 난다. 무한 그래프의 경우에는 결코 해를 찾지도 못하고, 끝내지도 못한다는 단점이 있다.

다음은 DFS (Depth First Search), 깊이 우선 탐색 알고리즘이다. 보통 재귀함수로 구현하거나 스택으로 반복문을 이용하여 구현한다. 먼저 탐색 시작 정점을 스택에 삽입하고 방문 처리한다. 스택의 최상단 정점에 방문하지 않은 인접한 노드가 하나라도 있으면 그 노드를 스택에 넣고 방문 처리하고, 방문하지 않은 인접한 노드가 없으면 스택에서 최상단 정점을 꺼낸다. 두 번째 과정을 수행할 수 없을 때까지 반복한다. DFS 방법의 장점은 다음과 같다. 현재 경로 상의 정점만 기억하면 되므로 저장공간의 수요가

비교적 적다. 목표 정점이 깊은 단계에 있을 경우 해를 빨리 구할 수 있다. 반면 단점으로는 해가 없는 경로에 깊이 빠질 가능성이 있다. 따라서 실제의 경우 미리 지정한 임의의 깊이까지만 탐색하고 목표 노드를 발견하지 못하면 다음 경로를 따라 탐색하는 방법이 유용할 수 있다. 또한, 얻어진 해가 최단 경로가 된다는 보장이 없다. 이는 목표에 이르는 경로가 다수인 문제에 대해 DFS는 해에 다다르면 탐색을 끝내 버리므로, 이때 얻어진 해는 최적이 아닐 수 있다.

다음으로는 모든 노드를 최소한의 비용으로 연결하는 알고리즘인 Kruskal 알고리즘에 대해서 살펴보도록 하겠다. n개의 정점이 있다고 할 때 최소 비용 신장 트리(MST)의 엣지는 n-1개를 사용해야 한다. 최소 비용 신장 트리(MST)는 최소 비용의 간선으로 구성되고 사이클을 포함하지 않는다. 크루스칼 알고리즘의 동작은 다음과 같다. 우선 모든 그래프의 간선들의 가중치를 오름차순으로 정렬한다. 정렬된 간선 리스트에서 순서대로 사이클을 형성하지 않는 간선을 선택한다. 즉, 가장 낮은 가중치를 먼저 선택한다. 사이클을 형성하는 간선은 제외한다. 해당 간선을 현재 MST의 집합에 추가한다. 여기서 가중치들의 오름차순 정렬은 Quick Sort 알고리즘과 Insertion Sort 알고리즘을 사용하는데 이는 정렬 알고리즘에서 다루도록 하겠다. 또한, 사이클이 형성되는지에 대한 판별 여부는 Union-Find 알고리즘을 사용하였다.

Union-Find 알고리즘이란 Disjoin Set의 개념을 이용한다. Disjoint Set이란 서로 중복되지 않은 부분 집합들로 나눠진 원소들에 대한 정보를 저장하고 조작하는 자료 구조이다. 즉 공통원소가 없는, 즉 "상토 배타적"인 부분 집합들로 나눠진 원소들에 대한 자료구조이다. Union-Find란 Disjoint Set을 표현할 때 사용하는 알고리즘으로 집합을 구현하는 데는 배열, 연결 리스트, 트리 등을 이용할 수 있으나 본 프로젝트에서는 배열로 Disjoint Set을 표현하였다. 배열로 Disjoint Set을 표현할 때는 Parent 배열을 하나 사용한다. Parent 배열은 각 인덱스가 자신의 부모의 정보를 저장하는 구조로 되어 있으며, Find 연산을 통해 루드 노드를 받고, 두 정점에 대해서 루트 노드가 같다면 현재 같은 집합에 존재하는 것이고, 그 번호가 다르면 현재 다른 집합에 존재하는 것이기 때문에 Union 연산을 수행하여 준다. Union 연산은 y를 x의 자손으로 넣어서 두 트리를 합하는 것이다.

다음으로는 다익스트라(Dijkstra) 알고리즘에 대해서 알아보도록 하겠다. 다익스트라 알고리즘은 이전에 살펴본 크루스칼 알고리즘과 같이 그래프에서 최소 비용을 구해야 하는 경우 유용하게 사용되는 알고리즘이다. 최소 비용 중에도, 주어진 두 노드(시작 노드, 도착 노드) 사이의 최소 비용인 경로를 찾을 때 유용하게 사용된다. 비용은 dist[]라는 배열에 저장되어 있다고 하고 초기 Dist 배열의 모든 값은 무한대(MAX)로 초기화 되어 있다. 다익스트라 알고리즘의 동작 원리를 순서대로 살펴보면 다음과 같다. 우선 시작 노드와 직접적으로 연결된 모든 정점들의 거리를 비교하여 업데이트 시켜주고, 시작 노드를 방문한 노드로 체크한다. 방문한 점점들과 연결되어 있는 정점들 중, 비용이 가장 적게 드는 정점을 선택하고, 해당 정점을 방문 처리한다. 이 과정에 의해서 비용이 가정 적게 드는 정점을 통해 갈 때 비용이 더 적게 드는 경우에 한해서 정점들의 거리를 업데이트 해 나가는 방식이다. 위의 과정을 반복해 나가서 Shortest Path와 Cost를 구하는 알고리즘이다. 다익스트라 알고리즘의 점화식은

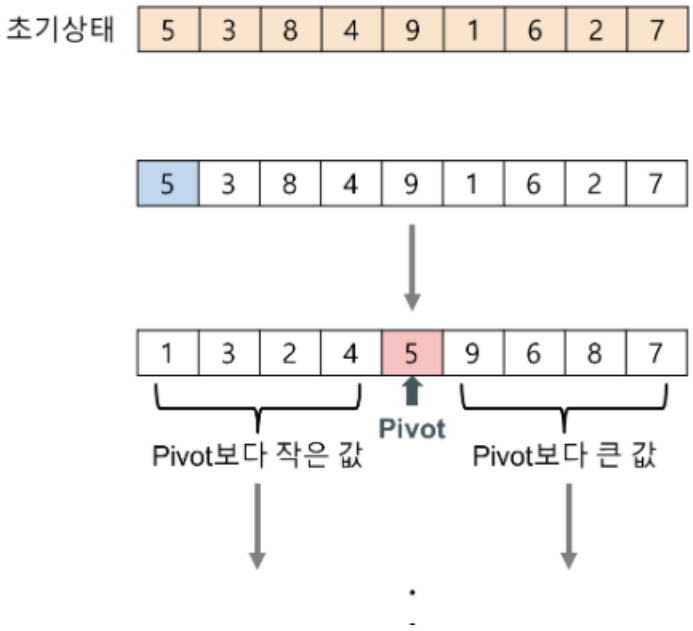
다음과 같다. dist[w] = min(dist[u] + length[u][w], dist[w]) 여기서 u는 방문한 정점들과 연결되어 있는 정점들 중 비용이 가장 적게 드는 정점이고, w는 u의 인접한 노드들이다.

다음으로는 Bellman-Ford 알고리즘이다. 벨만포드 알고리즘은 다익스트라 알고리즘과 마찬가지로 한 노드에서 다른 노드까지의 최단 거리를 구하는 알고리즘이다. 간선의 가중치가 음수일때는 올바르게 동작하지 않을 수도 있는 다익스트라 알고리즘에 비해 벨만포드 알고리즘은 간선의 가중치가 음수일때도 올바르게 최단 거리를 구할 수 있다. 우선 첫 번째는 출발 노드를 설정하고, 최단 거리 테이블을 초기화 한다. 두 번째로 모든 간선 E개를 하나씩 확인한다. 각 간선을 거쳐 다른 노드로 가는 비용을 계산하여 최단 거리 테이블을 갱신한다. 두 번째 과정을 정점의 개수 - 1만큼 반복하여 최단 거리 테이블을 구한 뒤, 음수 사이클이 발생하는 지 확인한다. 음수 사이클이 발생하는 지 확인하는 방법은 2번째 과정을 한번 더 수행하여 최단 거리 테이블이 갱신된다면 음수 간선 순환이 존재하는 것으로 판단할 수 있다.

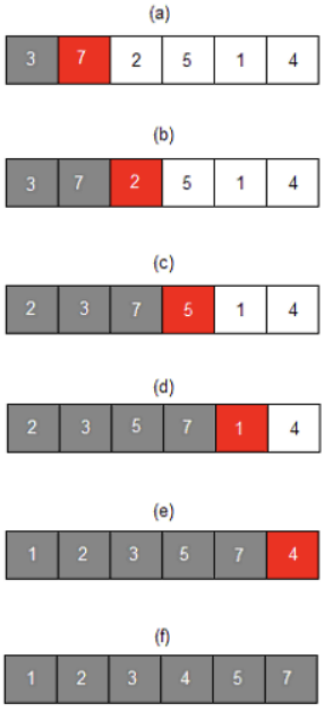
다음으로는 Floyd 알고리즘이다. 플로이드 알고리즘은 한 번 실행하여 모든 노드 간 최단 경로를 구할 수 있는 알고리즘이다. 모든 노드 간의 최단 거리를 구해야 하므로 2차원 인접 행렬을 구성한다. 초기 인접 행렬은 직접적으로 연결된 경로에 대해서 인접 행렬을 업데이트한다. 다음으로는 0번 노드를 새로운 중간 노드로 설정하여 중간 노드를 거쳐서 경로가 더 짧아진다면 경로를 업데이트한다. 위의 과정을 최대 노드 번호까지 수행하여 모든 노드 간 최단 경로를 구한다. 플로이드 알고리즘의 점화식은 다음과 같다. A[i][j] = min{A[i][j], A[i][k]+A[k][j]}, 즉 쉽게 말해서 플로이드 알고리즘은 노드에 번호를 매겨서 0부터 n까지의 노드들 중 사용할 수 있는 노드의 번호를 늘려가며 경로를 Update하는 것이고, 벨만포드 알고리즘은 사용할 수 있는 엣지의 개수를 늘려서 최단 거리 테이블을 갱신해 나가는 것이다.

마지막으로 정렬 알고리즘이다. 본 프로젝트에서는 크루스칼 알고리즘에서 엣지들의 가중치를 오름차순으로 정렬할 때 정렬 연산을 사용하였는데, 연산의 효율을 향상하기 위해서 segment size에 따라 정렬 알고리즘을 구현하였다. 퀵 정렬을 수행하며 정렬 수행 시 segment size가 6 이하일 경우 삽입 정렬을 수행하며, segment size가 7이상인 경우에는 분할하여 퀵소트를 수행한다.

Quick Sort는 In place하지만 Stable하지는 못하다. 즉, 인자로 들어온 리스트 이외에 추가로 사용하는 메모리가 상수크기의 메모리 거나 O(logn) 크기의 메모리 이지만, 같은 Key값이 존재하면 순서를 유지한다. 퀵 정렬의 과정은 다음과 같다. 우선 리스트 안에 있는 한 요소를 선택한다. 이렇게 고른 원소를 Pivot이라 한다. 피벗을 기준으로 피벗보다 작은 요소들은 모두 피벗의 왼쪽으로 옮겨지고, 피벗보다 큰 요소들은 모두 피벗의 오른쪽으로 옮겨진다. 피벗을 제외한 왼쪽 리스트와 오른쪽 리스트를 다시 정렬한다. 분할된 부분 리스트에 대해서는 재귀 호출을 사용하여 정렬을 반복한다. 부분 리스트에서도 다시 피벗을 정하고 피벗을 기준으로 2개의 부분 리스트로 나누는 과정을 반복한다. 부분 리스트들이 더 이상 분할이 불가능할 때까지 반복한다.



다음으로는 삽입정렬이다. 삽입 정렬의 과정은 다음과 같다. 우선 2번째 원소부터 비교를 시작한다. 이전 위치에 있는 원소들과 타겟이 되는 원소들을 비교한다. 여기서 타겟 변수는 임시변수에 저장한다. 타겟 원소가 이전 위치에 있는 원소보다 작다면 위치를 바꾼다. 이전 데이터도 차례대로 비교해나가며 정렬을 수행한다. 위의 과정을 반복 수행하여 정렬한다. 특징은 구현이 간단하지만, 배열이 길어질수록 효율이 떨어진다는 점이다. 선택 정렬이나 버블 정렬과 같은 알고리즘에 비해 빠르고 안정 정렬이다. 추가적인 공간을 필요로 하지 않고, Key가 같은 값이 있을 때 순서가 바뀌지 않으므로 Stable하며 In-place하다고 할 수 있다.



Result Screen

VMWare 우분투 18.04에서 동작을 확인하였다.

먼저 LOAD의 결과화면이다. 성공적으로 LOAD를 하였으면 Success를 출력하고, 파일이 존재하지 않거나 파일 내에 아무 것도 존재하지 않는다면 에러코드를 출력한다.

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

```
======== ERROR ========
100
=======================
```

다음은 PRINT 명령어이다. 그래프가 존재하지 않는다면 에러코드를 출력하고, 아니라면 생성한 그래프에 따라서 인접행렬과 인접 리스트 중 하나의 그래프를 출력한다.

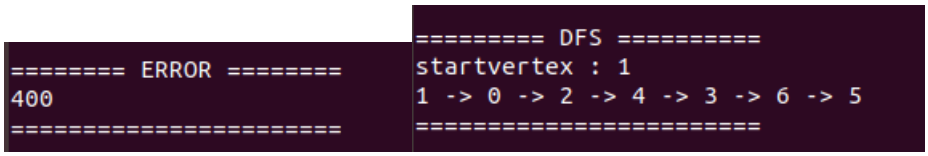
```
======== ERROR ========
200
=======================
```

```
========= PRINT =========
Graph is MatrixGraph!
       [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
=========================
```

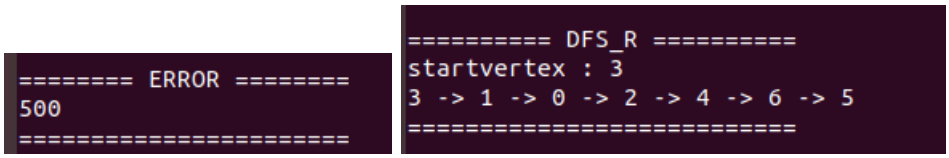
```
========= PRINT =========
Graph is ListGraph!
[0] -> (1,6) -> (2,2)
[1] -> (3,5)
[2] -> (1,7) -> (4,3) -> (5,8)
[3] -> (6,3)
[4] -> (3,4)
[5] -> (6,1)
[6] -> (4,10)
=========================
```

다음은 BFS 명령어이다. 입력한 Vertex를 기준으로 BFS를 수행하는 명령어로 BFS 결과를 출력 파일에 출력한다. 입력한 Vertex가 그래프에 존재하지 않거나, vertex를 입력하지 않은 경우, 에러코드를 출력한다.

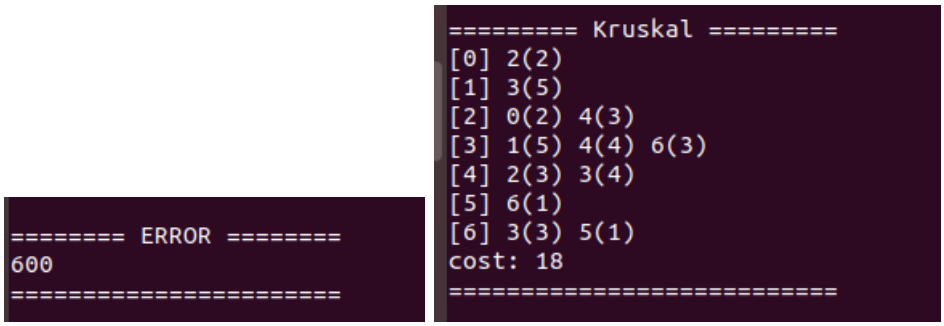
다음은 DFS 명령어이다. 입력한 Vertex를 기준으로 DFS를 수행하는 명령어로 BFS 결과를 출력 파일에 출력한다. 입력한 Vertex가 그래프에 존재하지 않거나, vertex를 입력하지 않은 경우, 에러코드를 출력한다.



다음은 DFS_R 명령어이다. 수행은 DFS와 같으며 재귀함수로 구현한 것이다.



다음은 KRUSKAL 명령어이다. 크루스칼 명령어는 현재 그래프의 MST를 구하고, MST를 구성하는 가중치 총합을 출력한다. 입력한 그래프가 MST를 구할 수 없는 경우, 즉 연결되어 있지 않은 경우, 명령어를 수행할 수 없는 경우 에러코드를 출력한다.



다음은 DIJKSTRA 명령어이다. 입력 받은 인자로부터 각 vertex들까지의 Shortest path를 구하고 cost와 함께 출력한다. 기준 Vertex에 도달할 수 없는 경우 'x'를 출력하고, 입력한 Vertex가 그래프에 존재하지 않거나 vertex를 입력하지 않은 경우, 명령어를 수행할 수 없는 경우 에러코드를 출력한다.

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

```
========= 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)
============================
```

다음은 BELLMANFORD 명령어이다. StartVertex를 기준으로 벨만포드 알고리즘을 수행하여 EndVertex까지의 최단 경로와 거리를 구하여 출력한다. 시작 노드에서 도착 노드로 도달할 수 없는 경우 'x'를 출력하고, 입력한 Vertex가 그래프에 존재하지 않거나 vertex를 입력하지 않은 경우, 명령어를 수행할 수 없는 경우 에러코드를 출력한다. 그리고 음의 사이클이 발생할 경우에 대해서도 에러코드를 출력한다.

```
======== ERROR ========
800
=======================
```

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

마지막으로 FLOYD 명령어이다. 모든 노드 쌍에 대해서 StartVertex에서 EndVertex로 가는데 필요한 비용의 최솟값을 행렬형태로 출력하는 명령어이다. 기준 Vertex에 도달할 수 없을 경우 'x'를 출력한다. 음수 사이클이 발생한 경우 출력파일에 에러코드를 출력한다.

```
======== ERROR ========
900
=======================
```

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

Consideration

이번 프로젝트를 구현함에 있어서 어려움이 있었던 부분은 먼저 그래프를 처음으로 다루다 보니 그래프의 개념 같은 것들이 머리속에 정확하게 잡혀 있지 않아서 그 부분에서 어려움이 많았다. 교수님이 수업시간에 진행한 강의자료에서 사용한 Psuedo Code와 강의자료를 이용하여 찬찬히 고찰해보니 구현에 성공할 수 있었다.

다음은 음의 사이클 발생 여부였다. 다익스트라는 음의 가중치가 발생하면 바로 에러코드를 출력하면 되기 때문에 문제가 발생하지 않았지만 음의 가중치가 존재하는 그래프에서 벨만포드와 플로이드 연산은 음의 가중치가 발생하였을 때만 에러코드를 출력해야 했기 때문에 어려움이 있었다. 벨만포드에서 음의 사이클이 발생하였다면, n-1개의 엣지를 사용하여 최단 경로 테이블을 작성하였을 때 다시 한번 엣지에 대한 연산을 수행하였을 때 값이 더 줄어들기 때문에 이 방법을 이용하여 음의 사이클 발생 여부를 확인하였다. 그리고 플로이드 연산에서는 음의 사이클이 발생하면 자기 자신으로 돌아오는 경로는 원래 0이 되어야 하는데 음의 사이클이 발생하면 자기 자신으로 돌아오면 경로가 음수의 값이 나오게 된다. 따라서 모든 노드들의 쌍에 대해서 경로를 구한 뒤, 인접 행렬에서 자기 자신으로 돌아오는 경로가 음수면 음의 사이클이 발생하였다고 판단하고 에러 코드를 출력하였다.

다음으로는 출력 문제였다. 다익스트라와 벨먼포드 알고리즘을 이용할 때 dist와 prev 배열을 이용하는데 도착 정점에서부터 prev로 따라가는 구조이기 때문에 일반적으로 반복문을 사용하여 구현할 수가 없었다. 해결 방법으로는 Stack Stl을 사용하여 LIFO 구조를 적극 이용하여 출력을 수행하니 올바르게 출력할 수 있었다.

이번 프로젝트를 구현하고 나니 객체지향프로그래밍에서 배운 다형성에 대해서 완벽하게 개념을 익힐 수 있었다. 객체지향 프로그래밍에서 다형성이란 간단히 말해 모습은 같은데 형태는 다른 것을 의미한다. 객체 지향에서 다향성은 부모 클래스의 객체 타입에서 자식 클래스의 객체가 할당되어 있다면 virtual 키워드를 사용하여 부모 클래스의 함수를 재정의함으로써 객체의 함수를 호출할 수 있는 것이다. 원래대로 다형성의 개념을 사용하지 않고 코드를 구현하였다면 인접행렬과 인접 리스트에 관한 2가지 형태에 대해서 모두 다르게 함수를 구현하여 사용해야 할 것이지만, 다형성의 개념을 이용해 구현하니 하나의 객체를 사용하여 모든 그래프 연산을 수행할 수 있었다.