Data Structures_Porject_03

학　　과: 컴퓨터공학과

담당교수: 최상호 교수님

학　　번: 2020202037

성　　명: 염정호

1. Introduction

1) 목적

본 프로젝트는 graph데이터를 이용해 다양한 방법으로 그래프 탐색을 수행하는 프로그램을 구현한다.

2) 세부사항

그래프는 정보가 저장된 텍스트 파일을 통해 구현되며, 그래프의 특성에 따라 BFS, DFS, Kruskal, Dijkstra, Bellman-Ford, FLOYD 그리고 KwangWoon 총 7 가지 알고리즘을 통해 그래프 탐색을 수행한다. 그래프 데이터는 방향성과 가중치를 가지고 있으며 데이터의 형태에 따라 List또는 Matrix로 구현된다. 그래프 탐색마다 방향성을 고려하거나 고려하지 않는 조건이 존재한다.

7가지 연산종류는 다음과 같다.

- BFS&DFS

->방향성과 무방향성 모두 가능하며 그래프의 가중치는 고려하지 않고 순회/탐색을 실행한다.

- Kruskal

무방향성 그래프만 탐색 가능하며 MST를 만드는 방법으로 방향성이 없고 가중치가 있는 그래프 환경에서 수행된다.

-Dijkstra

정점 하나를 출발점으로 두고 다른 모든 정점을 도착점으로 하는 최단경로 알고리즘으로 방향성과 가중치 모두 존재하는 그래프 환경에서 연산을 수행한다. 만약 weight가 음수일 경우 Dijkstra알고리즘은 에러를 출력하게 된다.

-Bellman-Ford

정점 하나를 출발점으로 두고 다른 모든 정점을 도착점으로 하는 최단경로 알고리즘으로 방향성과 가중치 모두 존재하는 그래프 환경에서 연산을 수행한다. 만약 weight가 음수일 경우 Bellman-Ford음수 가중치는 허용되지만 음수사이클이 발생하는 경우 에러를 출력하고 발생하지 않은 경우 최단 경로와 거리를 구한다.

-FLOYD-

방향성, 무방향성 모두 가능하며 음수사이클이 발생한 경우에는 에러를, 발생하지 않은 경우에는 최단경로 행렬을 구한다.

-KwangWoon

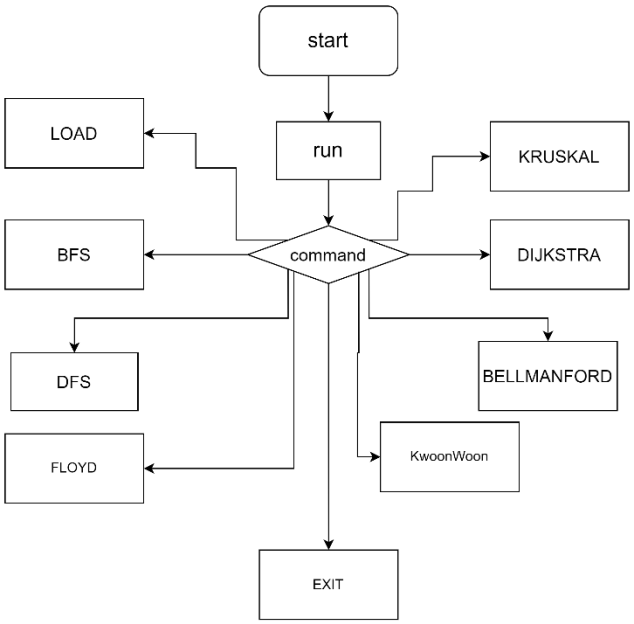
무방향성만 가능, list graph를 이용해 구현 가능하다. 현재 정점에서 방문할 수 있는 정점들이 홀수개면 탐색할 수 있는 정점 번호들의 가장 큰 정점 번호로 방문하고, 짝수일 경우 가장 작은 정점 번호로 방문을 시작한다.

헤더파일	명령어	내용
Manager	run	프로그램 실행 명령어, command.txt파일을 열어 안에 있는 명령어들을 순차적으로 실행하고, log.txt파일에 수행 결과를 쓴다.
	LOAD	graph_L또는 graph_M 파일로부터 데이터를 읽어오며 그래프 형태 L 또는 M에 따라 다른 형태로 데이터를 내부에 저장한다. txt파일이 존재하지 않거나 내부에 데이터가 없는 경우 오류코드를 출력한다.
	PRINT()	현재 저장하고 있는 그래프 데이터를 그래프 타입에 따라 출력한다 현재 데이터가 없는 경우 에러코드를 출력한다.
	mBFS	저장된 그래프를 BFS방식으로 탐색한다. 인자가 부족하거나 vertex로 이상한 값이 들어오면 에러를 출력한다.
	mDFS	저장된 그래프를 DFS방식으로 탐색한다. 인자가 부족하거나 vertex로 이상한 값이 들어오면 에러를 출력한다.
	mDIJKSTRA	저장된 그래프를 DIJKSTRA방식을 이용해 탐색한다(미구현)
	mKRUSKAL	저장된 그래프를 KRUSKAL방식을 이용해 탐색한다(미구현)
	mBELLMANFORD	저장된 그래프를 BELLMANFORD방식을 이용해 탐색한다(미구현)
	mFLOYD	저장된 그래프를 FLOYD방식을 이용해 탐색한다(미구현)
	mKwoonWoon	저장된 그래프를 mKwoonWoon 방식을 이용해 탐색한다(미구현)
	printErrorCode	명령어 실행결과 에러가 발생하는 경우 해당하는 에러코드를 출력한다.
Graph	getType	해당 그래프의 저장형태를 출력한다. L또는 M
	getSize	해당 그래프의 크기 즉 vertex 개수를 출력한다.

ListGraph	getAdjacentEdges	리스트형태로 저장된 그래프에서 무방향성 edge를 반환한다.. 선택된 vetex에서 들어오는 간선과 나가는 간선 모두를 확인한다.
	getAdjacentEdges Direct	리스트형태로 저장된 그래프에서 방향성이 있는 edge를 반환한다. 선택된 vertex에서 output edge만을 확인한다.
	insertEdge	리스트그래프에서 간선을 추가하는 것으로 load명령어를 통해 읽어온 데이터를 이용해 간선을 추가한다.
	printGraph	현재 저장된 그래프를 출력한다.
MatrixGraph	getAdjacentEdges	행렬 형태로 저장된 그래프에서 무방향성 edge를 반환한다. 해당 vertex에서 들어오는 간선과 나가는 간선 모두를 확인한다.
	getAdjacentEdges Direct	행렬 형태로 저장된 그래프에서 방향성이 있는 edge를 반환한다. 선택된 vertex에서 output edge만을 반환한다.
	insertEdge	행렬그래프에서 간선을 추가하는 것으로 load명령어를 통해 읽어온 데이터를 이용해 간선을 추가한다.
	printGraph	현재 저장된 그래프를 출력한다.
GraphMethod	BFS	너비우선탐색을 이용해 해당 그래프를 탐색한다. 그래프는 사용자 입력에 따라 방향성 무방향성을 선택해 탐색을 진행하게 된다. 큐를 이용해 탐색을 진행하며 시작 vertex로부터 갈 수 있는 모든 간선을 데이터에 저장 후 해당 간선은 큐에서 삭제하며 다음 간선을 탐색한다.
	DFS	깊이우선탐색을 이용해 그래프를 탐색한다. 그래프는 사용자 입력에 따라 뱡항성 무방향성을 선택해 탐색을 진행하게 된다. 스택을 이용해 탐색을 진행하며 시작 vertex로부터 다음 벌텍스로 계속 이동하다가 더 이상 이동할 수 없을 경우 이전 정점으로 이동 후 새로운 정점을 탐색한다.
	KWANGWOON	미구현
	Kruskal	kruskal알고리즘을 이용해 그래프를 탐색해 MSP를 구하는 것으로 방향성이 없는 그래프를 탐색한다. 이어진 모든 간선에서 가중치가 제일 작은순으로 간선을 선택하며 disjoint set을 이용해 사이클 여부를 확인한다. 탐색 이후 간선의 개수가 n-1개 미만인 경우 spanning tree가 존재하지 않는다.
	Dijkstra	미구현
	Bellmanford	미구현
	FLOYD	미구현

2. Flowchart

Main&Manager flowchart



LOAD에서 입력 데이터 형태에 따라 행렬 또는 리스트 형태로 배열 저장.
커멘드 입력에 따라 GraphMethod에서 탐색 방법을 선택한다.

3. ALOGRITHM

MANNAGER{

Run:

Command.txt 파일을 불러오고

Log.txt파일을 생성한다.

LOAD:

해당파일을 읽어와 명령어 수행

파일이 존재하지 않을 경우 오류 출력.

명령어 수행 과정에서 오류가 나타날 경우 명령어 별로]

에러코드 출력.

PRINT:

저장된 그래프를 형태에 따라 출력.

-탐색 방법 별 알고리즘은 그래프 매소드에서 설명-

}

Graph{

getType:

현재 저장된 그래프의 타입을 반환

getSize :

현재 저장된 그래프의 사이즈를 반환
getAdjacentEdges:
방향성이 있는 그래프의 간선을 반환할 때 사용하는 가상함수

getAdjacentEdgesDirect:
방향성이 없는 그래프의 간선을 반환할 때 사용하는 가상함수.

insertEdge:
그래프에 간선을 추가할 때 이용하는 가상함수.

printGraph:
그래프 데이터 출력을 위해 사용하는 가상함수.
}

ListGraph{

getAdjacentEdges:
입력받은 vertex와 연결된 간선데이터를 저장하고
무방향성 데이터이기 때문에
모든 데이터를 순회하며 해당 데이터로 들오는 간선 또한
맵 데이터로 추가한다.

getAdjacentEdgesDirect:
해당 데이터로부터 나가는 간선을
입력 받은 맵에 저장한다.

insertEdge:
시작점, 끝점, 가중치를 입력받아.
해당 데이터를 리스트 형태로
추가한다.

printGraph:
리스트 형태로 데이터를 출력한다.
}

Matrix Graph{

getAdjacentEdges:
입력받은 vertex로부터 진행하는 무방향성 간선들을
입력받은 맵 주소에 저장한다.
무방향성 배열이기 때문에 I 에서 vertex로 들어오는 데이터가 없으면
vertex에서 i로 향하는 데이터를 확인하고 해당 데이터가 존재하면 데이터를 추가한다.

getAdjacentEdgesDirect:
입력받은 vertex로부터 진행하는 방향성이 있는 간선들은
입력받은 맵에 저장한다.
방향성이 존재하는 배열이기 때문에 vertex에서 출발하는
모든 간선들을 탐색하여 맵 데이터에 저장한다.

insertEdge:
입력받은 메트릭스 데이터를 매트릭스 형태로 저장한다.

printGraph:
현재 저장된 배열을 매트릭스 형태로 출력한다.
}

GraphMethod{
BSF :
처음 입력 받은 데이터를 큐에 저장한다.
해당 큐로부터 갈 수 있는 모든 정점을 Q에 저장한다.(정점이 작은 순서대로)
옵션에 따라 방향성 있는 데이터 또는 방향성 없는 edge를 불러온다.
큐에 가장 처음 저장된 정점을 삭제하고 큐에 저장된 다음 정점을 불러온다.
큐에 저장된 모든 정점이 소모될 때 까지 해당 수행을 반복한다.

DFS :
처음 입력된 데이터를 스택에 저장한다.
해당 vertex로부터 갈수 있는 가장 작은 정점을 스택에 삽입 후
해당 정점으로 이동한다.
계속 해서 이동하다가 더 이상 이동이 불가능 할 경우 이전 정점으로 이동 후
탐색하지 않은 정점으로 이동한다.
스택이 비어있을 경우 종료한다.

KRUSKAL :
그래프에서 탐색 가능한 모든 간선을 불러온다.
그래프의 간선들을 weight순서대로 데이터를 정렬한다.
저장된 간선들을 차례대로 판별하며 disjoint set에 데이터를
추가한다 이미 추가된 간선의 경우 데이터를 넘긴다.
최종적으로 추가된 edge의 개수가 n -1개인 경우
해당 그래프의 spanning tree가 완성된 것이가 n-1개 미만인 경우
Spanning tree가 생성되지 않는다.

KWANGWOON :

Dijkstra :

Bellmanford :

FLOYD :

}

4. Result Screen

```
LOAD graph_M.txt
PRINT
BFS Y 1
BFS N 1
DFS Y 1
DFS N 1
KRUSKAL
LOAD graph_L.txt
PRINT
BFS Y 1
BFS N 1
DFS Y 1
DFS N 1
```

LOAD

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

PRINT

```
============print==============
     [ 1 ] [ 2 ] [ 3 ] [ 4 ] [ 5 ] [ 6 ] [ 7 ] [ 8 ] [ 9 ] [ 10 ] [ 11 ] [ 12 ] [ 13 ] [ 14 ] [ 15 ] [ 16 ] [ 17 ] [ 18 ]
[ 1 ]   0     3     0     0     2     0     0     5     0     0     0     0     0     0     0     0     0     0
[ 2 ]   0     0     4     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
[ 3 ]   0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
[ 4 ]   0     0     2     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
[ 5 ]   0     1     0     0     0     4     0     0     1     0     0     0     0     0     0     0     0     0
[ 6 ]   0     0     3     0     0     0     0     0     7     4     0     0     0     0     0     0     0     0
[ 7 ]   0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
[ 8 ]   0     0     0     0     0     0     0     0     4     0     0     3     0     0     0     0     0     0
[ 9 ]   0     0     0     0     0     0     0     0     0     1     0     0     0     0     0     0     0     0
[ 10 ]  0     0     0     0     0     0     3     0     0     0     0     0     0     0     0     0     0     0
[ 11 ]  0     0     0     10    0     0     1     0     0     6     0     0     0     0     0     0     0     0
[ 12 ]  0     0     0     0     0     0     0     0     5     0     0     0     0     0     9     0     0     0
[ 13 ]  0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     8     3     0
[ 14 ]  0     0     0     0     0     0     0     0     0     2     0     0     1     0     0     0     0     0
[ 15 ]  0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
[ 16 ]  0     0     0     0     0     0     0     0     0     0     0     0     11    0     0     0     0     0     15
[ 17 ]  0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
[ 18 ]  0     0     0     20    0     0     0     0     0     0     10    0     0     0     0     0     0     0     0
================================
```

BFS Y & N

```
======== BFS ========
Directed Graph BFS result
start vertex : 1
1->2->5->8->3->6->9->12->10->15->7->
======================

======== BFS ========
UnDirected Graph BFS result
start vertex : 1
1->2->5->8->3->6->9->12->4->10->15->16->11->18->7->14->13->17->
======================
```

DFS Y & N

```
======== DFS ========
Directed Graph BFS result
start vertex : 1
1->2->3->5->6->9->10->7->8->12->15->
=====================

======== DFS ========
UnDirected Graph BFS result
start vertex : 1
1->2->3->4->11->7->10->6->5->9->8->12->15->16->13->14->17->18->
=====================
```

KRUSKAL

```
====== Kruskal =======
[1]    5(2)
[2]    3(4) 5(1)
[3]    2(4) 4(2) 6(3)
[4]    3(2)
[5]    1(2) 2(1) 9(1)
[6]    3(3)
[7]    10(3) 11(1)
[8]    9(4) 12(3)
[9]    5(1) 8(4) 10(1)
[10]    7(3) 9(1) 14(2)
[11]    7(1) 18(10)
[12]    8(3) 15(9)
[13]    14(1) 16(8) 17(3)
[14]    10(2) 13(1)
[15]    12(9)
[16]    13(8)
[17]    13(3)
[18]    11(10)
====================
```

PRINT_L

```
=============print==============
[1] -> (2,3) -> (5,2) -> (8,5)
[2] -> (3,4)
[3]
[4] -> (3,2)
[5] -> (2,-2) -> (6,4) -> (9,1)
[6] -> (3,3) -> (9,7) -> (10,4)
[7]
[8] -> (9,4) -> (12,-2)
[9] -> (10,1)
[10] -> (7,3)
[11] -> (4,10) -> (7,1) -> (10,6)
[12] -> (9,5) -> (15,9)
[13] -> (16,8) -> (17,3)
[14] -> (10,2) -> (13,1)
[15]
[16] -> (12,11) -> (18,15)
[17]
[18] -> (4,20) -> (11,10)
================================
```

BFS & DFS

```
======== BFS ========
UnDirected Graph BFS result
start vertex : 1
1->2->5->8->3->6->9->12->4->10->15->16->11->18->7->14->13->17->
=====================

======== DFS ========
Directed Graph BFS result
start vertex : 1
1->2->3->5->6->9->10->7->8->12->15->
=====================

======== DFS ========
UnDirected Graph BFS result
start vertex : 1
1->2->3->4->11->7->10->6->5->9->8->12->15->16->13->14->17->18->
=====================
```

해당 그래프 파일은 홍왕기님의 파일을 참고하여 작성하였습니다.

5. Consideration

1. 과제를 진행하면서 무방향성 데이터 처리를 할 때 역삼각형 형태로 데이터가 들어올 경우 반대쪽 삼각형에 있는 데이터 또한 불러오는 부분에서 고민을 많이 해야 했다.

2. DFS구현 시 모든 인접한 모든 VERTX를 가져오지 않고 크기가 가장 작은 정점 외의 것을 불러와 데이터를 처리하는 알고리즘을 구현할 때 방문한 노드를 기록하는 배열을 추가함으로써 해결할 수 있었다.

3. 이번 과제를 통해 다양한 탐색 방법과 그래프에 대해 이해할 수 있었다 기존에 알고있던 트리구조, 배열, 링크드 리스트 이외에도 많은 데이터 표현방법과 정렬 방법들이 존재한다는 것을 알 수 있었던 과제였다.
   이번 과제에서 미구현한 부분도 방학을 이용해 탐색알고리즘을 구현해봐야겠다.