O-O Programming & Data Structure Lab. 13

13. Dijkstra 알고리즘

13.1 class Graph::Vertex

```
/** Graph.h */
#ifndef GRAPH_H
#define GRAPH_H

#include <list>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <limits>
#include <string>
using namespace std;

#define PLUS_INF INT_MAX
enum VrtxStatus { UN_VISITED, VISITED, VRTX_NOT_FOUND };
enum EdgeStatus { DISCOVERY, BACK, CROSS, EDGE_UN_VISITED, EDGE_VISITED, EDGE_NOT_FOUND };

class Graph    // Graph based on Adjacency Matrix
{
    . . . . .
public:
        class Vertex;
        class Edge;
        typedef std::list<Graph::Vertex> VrtxList;
        typedef std::list<Graph::Edge> EdgeList;
        typedef std::list<Vertex>::iterator VrtxItor;
        typedef std::list<Edge>::iterator EdgeItor;
public:
        class Vertex    // Graph::Vertex
        {
                friend ostream& operator<<(ostream& fout, Vertex v)
                {
                        fout << v.getName();
                        return fout;
                }
        public:
                Vertex() : name(), ID(-1) {}
                Vertex(string n, int id) : name(n), ID(id) { }
                Vertex(int id) : ID(id) {}
                string getName() { return name; }
                void setName(string c_name) { name = c_name; }
                int getID() { return ID; }
                void setID(int id) { ID = id; }
                void setVrtxStatus(VrtxStatus vs) { vrtxStatus = vs; }
                VrtxStatus getvrtxStatus() { return vrtxStatus; }
                bool operator==(Vertex v) { return ((ID == v.getID()) && (name == v.getName())); }
                bool operator!=(Vertex v) { return ((ID != v.getID()) || (name != v.getName())); }
        private:
                string name;
                int ID;
                VrtxStatus vrtxStatus;
        };    // end class Vertex
```

13.2 class Graph::Edge

```
class Graph    // Graph based on Adjacency Matrix
{
    . . . . .
public:
        class Edge    // Graph::Edge
```

```
    {
        friend ostream& operator<<(ostream& fout, Edge& e)
        {   . . . .             }
    public:
        Edge() : pVrtx_1(NULL), pVrtx_2(NULL), distance(PLUS_INF) {}
        Edge(Vertex& v1, Vertex& v2, int d)
            :distance(d), pVrtx_1(&v1), pVrtx_2(&v2), edgeStatus(EDGE_UN_VISITED)
        { }
        void endVertices(VrtxList& vrtxLst)   {   . . . . }
        Vertex opposite(Vertex v)  { . . . . }              }
        Vertex* getpVrtx_1() { return pVrtx_1; }
        Vertex* getpVrtx_2() { return pVrtx_2; }
        int getDistance() { return distance; }
        void setpVrtx_1(Vertex* pV) { pVrtx_1 = pV; }
        void setpVrtx_2(Vertex* pV) { pVrtx_2 = pV; }
        void setDistance(int d) { distance = d; }
        bool operator!=(Edge e) { return ((pVrtx_1 != e.getpVrtx_1()) || (pVrtx_2 != e.getpVrtx_2())); }
        bool operator==(Edge e) { return ((pVrtx_1 == e.getpVrtx_1()) && (pVrtx_2 == e.getpVrtx_2())); }
        void setEdgeStatus(EdgeStatus es) { edgeStatus = es; }
        EdgeStatus getEdgeStatus() { return edgeStatus; }
    private:
        Vertex* pVrtx_1;
        Vertex* pVrtx_2;
        int distance;
        EdgeStatus edgeStatus;
    }; // end class Edge
```

13.3 class Graph

```
/** Graph.h */
#ifndef GRAPH_H
#define GRAPH_H

#include <list>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <limits>
#include <string>
using namespace std;

#define PLUS_INF INT_MAX/2
enum VrtxStatus { UN_VISITED, VISITED, VRTX_NOT_FOUND };
enum EdgeStatus { DISCOVERY, BACK, CROSS, EDGE_UN_VISITED, EDGE_VISITED, EDGE_NOT_FOUND };

class Graph    // Graph based on Adjacency Matrix
{
public:
    class Vertex;
    class Edge;
    typedef std::list<Graph::Vertex> VrtxList;
    typedef std::list<Graph::Edge> EdgeList;
    typedef std::list<Graph::Vertex>::iterator VrtxItor;
    typedef std::list<Graph::Edge>::iterator EdgeItor;
public:
    class Vertex    // Graph::Vertex
    { . . . .   };   // end class Vertex

public:
    class Edge    // Graph::Edge
    {   . . . .   }; // end class Edge


public:
    Graph() : name(""), pVrtxArray(NULL), pAdjLstArray(NULL) {}    // default constructor
```

```
        Graph(string nm, int num_nodes) : name (nm), pVrtxArray(NULL), pAdjLstArray(NULL)
        { . . . .    }
        string getName() { return name; }
        void vertices(VrtxList& vrtxLst);
        void edges(EdgeList&);
        bool isAdjacentTo(Vertex v, Vertex w);
        void insertVertex(Vertex& v);
        void insertEdge(Edge& e);
        void eraseEdge(Edge e);
        void eraseVertex(Vertex v);
        int getNumVertices() { return num_vertices; }
        void incidentEdges(Vertex v, EdgeList& edges);
        Vertex* getpVrtxArray() { return pVrtxArray; }
        EdgeList* getpAdjLstArray() { return pAdjLstArray; }
        void fprintGraph(ofstream& fout);
        bool isValidvID(int vid);
private:
        string name;
        Vertex* pVrtxArray;
        EdgeList* pAdjLstArray;
        int num_vertices;
};

#endif
```

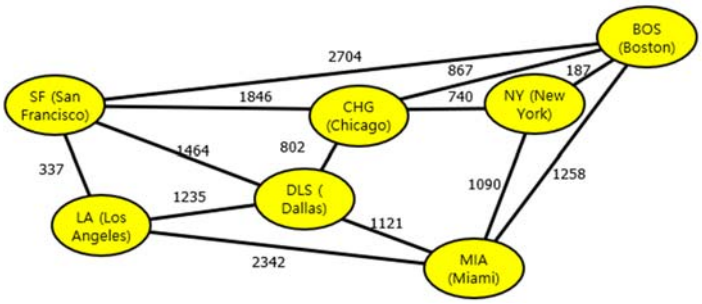
13.4 class BreadthFirstSearch with Dijkstra

```
/** BFS_Dijkstra.h */

#ifndef BFS_DIJKSTRA_H
#define BFS_DIJKSTRA_H

#include "Graph.h"
#include <fstream>
using namespace std;

typedef Graph::Vertex    Vertex;
typedef Graph::Edge    Edge;
typedef std::list<Graph::Vertex>    VrtxList;
typedef std::list<Graph::Edge>    EdgeList;
typedef std::list<Graph::Vertex>::iterator    VrtxItor;
typedef std::list<Graph::Edge>::iterator    EdgeItor;

class BreadthFirstSearch
{
protected:
        Graph& graph;
        bool done;          // flag of search done
        int **ppDistMtrx;    // distance matrix

protected:
        void initialize();
        bool isValidvID(int vid) { return graph.isValidvID(vid); }
        int getNumVertices() { return graph.getNumVertices(); }

public:
        BreadthFirstSearch(Graph& g) :graph(g) {
                int num_nodes;

                num_nodes = g.getNumVertices();
                // initialize DistMtrx
                for (int i = 0; i<num_nodes; i++)
                        ppDistMtrx = new int*[num_nodes];
                for (int i = 0; i<num_nodes; i++)
                        ppDistMtrx[i] = new int[num_nodes];
```

```
                for (int i = 0; i<num_nodes; i++) {
                        for (int j = 0; j<num_nodes; j++)
                        {
                                ppDistMtrx[i][j] = PLUS_INF;
                        }
                }
        }
        void initDistMtrx();
        void fprintDistMtrx(ofstream& fout);
        void DijkstraShortestPathTree(ofstream& fout, Vertex& s, int* pPrev);
        void DijkstraShortestPath(ofstream& fout, Vertex& s, Vertex& t, VrtxList& path);
        Graph& getGraph() { return graph; }
        int** getppDistMtrx() { return ppDistMtrx; }
};
#endif
```

13.5 main() function

```
/** main.cpp */

#include <iostream>
#include <fstream>
#include <string>
#include "Graph.h"
#include "BFS_Dijkstra.h"

using namespace std;
#define GRAPH_SIMPLE_USA_7_NODES

void main()
{
        ofstream fout;

        fout.open("output.txt");
        if (fout.fail())
        {
                cout << "Failed to open output.txt file !!" << endl;
                exit;
        }

#define NUM_NODES 7
#define NUM_EDGES 26
        Vertex v[NUM_NODES] = // 7 nodes
        {
                Vertex("SF", 0),        Vertex("LA", 1),        Vertex("DLS", 2),
                Vertex("CHG", 3),       Vertex("MIA", 4),       Vertex("NY", 5),
                Vertex("BOS", 6)

        };

        Graph::Edge edges[NUM_EDGES] =    // 70 edges
        {
                Edge(v[0], v[1], 337), Edge(v[1], v[0], 337),    Edge(v[0], v[2], 1464), Edge(v[2], v[0], 1464),
                Edge(v[0], v[3], 1846), Edge(v[3], v[0], 1846),   Edge(v[0], v[6], 2704), Edge(v[6], v[0], 2704),
                Edge(v[1], v[2], 1235), Edge(v[2], v[1], 1235), Edge(v[1], v[4], 2342), Edge(v[4], v[1], 2342),
                Edge(v[2], v[3], 802), Edge(v[3], v[2], 802),    Edge(v[2], v[4], 1121), Edge(v[4], v[2], 1121),
                Edge(v[3], v[5], 740), Edge(v[5], v[3], 740),    Edge(v[3], v[6], 867), Edge(v[6], v[3], 867),
                Edge(v[5], v[4], 1090), Edge(v[4], v[5], 1090),   Edge(v[5], v[6], 187), Edge(v[6], v[5], 187),
                Edge(v[4], v[6], 1258), Edge(v[6], v[4], 1258),
        };

        int test_start = 1;
        int test_end = 6;

        Graph simpleGraph("GRAPH_SIMPLE_USA_7_NODES", NUM_NODES);
```



```
	fout << "Inserting vertices .." << endl;
	for (int i = 0; i<NUM_NODES; i++) {
		simpleGraph.insertVertex(v[i]);
	}

	VrtxList vtxLst;
	simpleGraph.vertices(vtxLst);

	int count = 0;
	fout << "Inserted vertices: ";
	for (VrtxItor vItor = vtxLst.begin(); vItor != vtxLst.end(); ++vItor) {
		fout << *vItor << ", ";
	}
	fout << endl;

	fout << "Inserting edges .." << endl;
	for (int i = 0; i<NUM_EDGES; i++)
	{
		simpleGraph.insertEdge(edges[i]);
	}

	fout << "Inserted edges: " << endl;
	count = 0;
	EdgeList egLst;
	simpleGraph.edges(egLst);
	for (EdgeItor p = egLst.begin(); p != egLst.end(); ++p)
	{
		count++;
		fout << *p << ", ";
		if (count % 5 == 0)
			fout << endl;
	}
	fout << endl;

	fout << "Print out Graph based on Adjacency List .." << endl;
	simpleGraph.fprintGraph(fout);

	/* ========================================*/
	VrtxList path;
	BreadthFirstSearch bfsGraph(simpleGraph);

	fout << "\nTesting Breadth First Search with Dijkstra Algorithm" << endl;

	bfsGraph.initDistMtrx();

	//fout << "Distance matrix of BFS for Graph:" << endl;
	bfsGraph.fprintDistMtrx(fout);

	path.clear();
	fout << "\nDijkstra Shortest Path Finding from " << v[test_start].getName() << " to "
	     << v[test_end].getName() << " .... " << endl;
	bfsGraph.DijkstraShortestPath(fout, v[test_start], v[test_end], path);
	fout << "Path found by DijkstraShortestPath from " << v[test_end] << " to " << v[test_start] << " : ";
	for (VrtxItor vItor = path.begin(); vItor != path.end(); ++vItor)
	{
		fout << *vItor;
		if (*vItor != v[test_end])
			fout << " -> ";
	}
	fout << endl;

	fout.close();
}
```

13.6 Example output (output.txt)

```
Inserting vertices ..
Inserted vertices: SF, LA, DLS, CHG, MIA, NY, BOS,
Inserting edges ..
Inserted edges:
Edge(SF, LA, 337), Edge(SF, DLS, 1464), Edge(SF, CHG, 1846), Edge(SF, BOS, 2704), Edge(LA, SF, 337),
Edge(LA, DLS, 1235), Edge(LA, MIA, 2342), Edge(DLS, SF, 1464), Edge(DLS, LA, 1235), Edge(DLS, CHG, 802),
Edge(DLS, MIA, 1121), Edge(CHG, SF, 1846), Edge(CHG, DLS, 802), Edge(CHG, NY, 740), Edge(CHG, BOS, 867),
Edge(MIA, LA, 2342), Edge(MIA, DLS, 1121), Edge(MIA, NY, 1090), Edge(MIA, BOS, 1258), Edge(NY, CHG, 740),
Edge(NY, MIA, 1090), Edge(NY, BOS, 187), Edge(BOS, SF, 2704), Edge(BOS, CHG, 867), Edge(BOS, NY, 187),
Edge(BOS, MIA, 1258),
Print out Graph based on Adjacency List ..
GRAPH_SIMPLE_USA_7_NODES with 7 vertices has following adjacency lists :
 vertex ( SF) :  Edge(SF, LA, 337)  Edge(SF, DLS, 1464)  Edge(SF, CHG, 1846)  Edge(SF, BOS, 2704)
 vertex ( LA) :  Edge(LA, SF, 337)  Edge(LA, DLS, 1235)  Edge(LA, MIA, 2342)
 vertex (DLS) :  Edge(DLS, SF, 1464)  Edge(DLS, LA, 1235)  Edge(DLS, CHG, 802)  Edge(DLS, MIA, 1121)
 vertex (CHG) :  Edge(CHG, SF, 1846)  Edge(CHG, DLS, 802)  Edge(CHG, NY, 740)  Edge(CHG, BOS, 867)
 vertex (MIA) :  Edge(MIA, LA, 2342)  Edge(MIA, DLS, 1121)  Edge(MIA, NY, 1090)  Edge(MIA, BOS, 1258)
 vertex ( NY) :  Edge(NY, CHG, 740)  Edge(NY, MIA, 1090)  Edge(NY, BOS, 187)
 vertex (BOS) :  Edge(BOS, SF, 2704)  Edge(BOS, CHG, 867)  Edge(BOS, NY, 187)  Edge(BOS, MIA, 1258)

Testing Breadth First Search with Dijkstra Algorithm

Distance Matrix of Graph (GRAPH_SIMPLE_USA_7_NODES) :
       |   SF   LA  DLS  CHG  MIA   NY  BOS
-------+-----------------------------------
   SF  |    0  337 1464 1846  +oo  +oo 2704
   LA  |  337    0 1235  +oo 2342  +oo  +oo
  DLS  | 1464 1235    0  802 1121  +oo  +oo
  CHG  | 1846  +oo  802    0  +oo  740  867
  MIA  |  +oo 2342 1121  +oo    0 1090 1258
   NY  |  +oo  +oo  +oo  740 1090    0  187
  BOS  | 2704  +oo  +oo  867 1258  187    0


Dijkstra Shortest Path Finding from LA to BOS ....
Dijkstra::Least Cost from Vertex (LA) at each round :
           |   SF   LA  DLS  CHG  MIA   NY  BOS
-----------+-----------------------------------
round [ 1] |  337    0 1235 2183 2342  +oo 3041  ==> selected vertex : SF
round [ 2] |  337    0 1235 2037 2342  +oo 3041  ==> selected vertex : DLS
round [ 3] |  337    0 1235 2037 2342 2777 2904  ==> selected vertex : CHG
round [ 4] |  337    0 1235 2037 2342 2777 2904  ==> selected vertex : MIA
round [ 5] |  337    0 1235 2037 2342 2777 2904  ==> selected vertex : NY
round [ 6] |
reached to the target node (BOS) at Least Cost = 2904
Path found by DijkstraShortestPath from BOS to LA : LA -> DLS -> CHG -> BOS
```

<Oral Test 13>

(1) 그래프를 표현하기 위하여 사용되는 자료구조들을 그림으로 표현하고, 그 복잡도 (complexity)를 정점의 개수와 간선의 개수로 표현하라.

<Key Points>

(1) Vertex List, Edge List
(2) Adjacency List
(3) Adjacency Matrix
(4) 복잡도 분석표

(2) 2 x 3 격자형 그래프에 대한 깊이 우선 탐색 (Depth First Search) 알고리즘을 실행하기 위하여 구현되는 dfsTraversal() 멤버 함수를 pseudocode 로 표현하고, 상세 동작 절차를 주어진 2 x 3 격자형 그래프의 노드를 사용하여 설명하라.

<Key Points>

(1) 2 x 3 격자형 그래프 (vertex 0, 1, 2, 3, 4, 5)
(2) vertex 0으로 부터의 dfsTraversal() 실행에서 정점의 방문 순서와 간선의 구분 (discovery, back)
(3) DFS로 탐색된 vertex 0 -> vertex 5의 경로
(4) vertex 5로 부터의 dfsTraversal() 실행하는 경우 vertex 5 -> vertex 0의 경로
(5) 위 (3)과 (4) 경로에 대한 비교 분석

(3) 그래프에 대한 깊이 우선 탐색 (Depth First Search) 알고리즘, 넓이 우선 탐색 (Breadth First Search) 알고리즘, Dijkstra 알고리즘의 차이점에 대하여 설명하라.

<Key Points>

(1) Breadth First Search (BFS) 알고리즘의 동작 절차
(2) Breadth First Search (BFS)의 한 종류인 Dijkstra 알고리즘을 사용한 최단거리 경로 탐색 (shortest path search) 기능의 동작 절차
(3) Depth First Search (DFS)와 Breadth First Search (BFS) 알고리즘의 기능적 차이점과 주요 응용 분야에 대한 비교 설명

(4) 그래프의 자료구조의 주요 응용 분야에 대하여 상세하게 설명하라.

<Key Points>

(1) 자동차/스마트폰의 네비게이션
(2) 인터넷의 패킷 라우팅
(3) 전자회로 부품 배치
(4) 데이터 베이스의 연관 정보 검색을 통한 상관관계 분석

- 주요 응용 분야에서 정점과 간선은 어떤 정보를 의미하는가?
- 이 응용 분야에서 그래프 탐색의 목적은 무엇인가?