**Homework 13**

|  |
| --- |
| 그림입니다. 원본 그림의 이름: YU_UI_RGB-10.png 원본 그림의 크기: 가로 2256pixel, 세로 3047pixel 프로그램 이름 : Adobe ImageReady |

|  |  |
| --- | --- |
| 과목명 | 객체지향프로그래밍과자료구조 |
| 교수님 | 김영탁 교수님 |
| 이 름 | 김주환 |
| 학 번 | 21812158 |
| 일 자 | 2021.12.07.수 |

**13.1 그래프 깊이 우선 탐색과 미로 탐색 (Maze Traversal)**

|  |
| --- |
| /\* main.cpp \*/  /\* Description  \* Graph 구현및 기능 검사 (DFS)  \* Programmed by J. H. Kim  \* Last updated : 2021-12-07 \*/  #include <iostream>  #include <fstream>  #include <string>  #include "Graph.h"  #include "DepthFirstSearch.h"  #define NUM\_NODES 25  #define NUM\_EDGES 50  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;  void main()  {  ofstream fout;  fout.open("output.txt");  if (fout.fail())  {  cout << "Fail to open output.txt file !!" << endl;  exit(1);  }  Vertex  v[NUM\_NODES] =  {  Vertex("0", 0), Vertex("1", 1),  Vertex("2", 2), Vertex("3", 3),  Vertex("4", 4), Vertex("5", 5),  Vertex("6", 6), Vertex("7", 7),  Vertex("8", 8), Vertex("9", 9),  Vertex("10", 10), Vertex("11", 11),  Vertex("12", 12), Vertex("13", 13),  Vertex("14", 14), Vertex("15", 15),  Vertex("16", 16), Vertex("17", 17),  Vertex("18", 18), Vertex("19", 19),  Vertex("20", 20), Vertex("21", 21),  Vertex("22", 22), Vertex("23", 23),  Vertex("24", 24)  };  Edge edges[NUM\_EDGES] =  {  Edge(v[0], v[1], 1), Edge(v[1], v[0], 1),  Edge(v[0], v[5], 1), Edge(v[5], v[0], 1),  Edge(v[1], v[2], 1), Edge(v[2], v[1], 1),  Edge(v[1], v[6], 1), Edge(v[6], v[1], 1),  Edge(v[3], v[8], 1), Edge(v[8], v[3], 1),  Edge(v[4], v[9], 1), Edge(v[9], v[4], 1),  Edge(v[6], v[7], 1), Edge(v[7], v[6], 1),  Edge(v[6], v[11], 1), Edge(v[11], v[6], 1),  Edge(v[7], v[8], 1), Edge(v[8], v[7], 1),  Edge(v[8], v[13], 1), Edge(v[13], v[8], 1),  Edge(v[9], v[14], 1), Edge(v[14], v[9], 1),  Edge(v[10], v[11], 1), Edge(v[11], v[10], 1),  Edge(v[10], v[15], 1), Edge(v[15], v[10], 1),  Edge(v[12], v[17], 1), Edge(v[17], v[12], 1),  Edge(v[13], v[14], 1), Edge(v[14], v[13], 1),  Edge(v[14], v[19], 1), Edge(v[19], v[14], 1),  Edge(v[15], v[16], 1), Edge(v[15], v[16], 1),  Edge(v[15], v[20], 1), Edge(v[15], v[20], 1),  Edge(v[16], v[21], 1), Edge(v[16], v[21], 1),  Edge(v[17], v[18], 1), Edge(v[18], v[17], 1),  Edge(v[17], v[22], 1), Edge(v[22], v[17], 1),  Edge(v[18], v[19], 1), Edge(v[19], v[18], 1),  Edge(v[18], v[23], 1), Edge(v[23], v[18], 1),  Edge(v[21], v[22], 1), Edge(v[22], v[21], 1),  Edge(v[23], v[24], 1), Edge(v[24], v[23], 1)  };  Graph simpleGraph("GRAPH\_SQUARE\_16\_NODES", NUM\_NODES);  cout << "Inserting vertices .." << endl;  for (int i = 0; i < NUM\_NODES; i++) {  simpleGraph.insertVertex(v[i]);  }  VrtxList vrtxLst;  simpleGraph.vertices(vrtxLst);  int count = 0;  cout << "Inserted vertices: ";  for (VrtxItor vItor = vrtxLst.begin(); vItor != vrtxLst.end(); ++vItor) {  cout << \*vItor << " ";  }  cout << endl;  cout << "Inserting edges .." << endl;  for (int i = 0; i < NUM\_EDGES; i++)  {  simpleGraph.insertEdge(edges[i]);  }  cout << "Inserted edges: " << endl;  count = 0;  EdgeList egLst;  simpleGraph.edges(egLst);  for (EdgeItor p = egLst.begin(); p != egLst.end(); ++p)  {  count++;  cout << \*p << ", ";  if (count % 5 == 0)  cout << endl;  }  cout << endl;  cout << "Print out Graph based on Adjacency List .." << endl;  simpleGraph.printGraph();  cout << "Testing dfsGraph..." << endl;  DepthFirstSearch dfsGraph(simpleGraph);  VrtxList path;  dfsGraph.findPath(v[0], v[24], path);  cout << endl << "Path(" << v[0] << " = > " << v[24] << ") : ";  for (VrtxItor vItor = path.begin(); vItor != path.end(); ++vItor)  cout << \*vItor << " ";  cout << endl;  dfsGraph.findPath(v[24], v[0], path);  cout << endl << "Path(" << v[24] << " = > " << v[0] << ") : ";  for (VrtxItor vItor = path.begin(); vItor != path.end(); ++vItor)  cout << \*vItor << " ";  cout << endl;  } |
| /\*\* 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 // 오버플로우 방지  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  class Edge // Graph::Edge  {  friend ostream& operator<<(ostream& fout, Edge& e)  {  fout << "Edge(" << setw(2) << \*e.getpVrtx\_1() << ", " << setw(2)  << \*e.getpVrtx\_2() << ", " << setw(4) << e.getDistance() << ")";  return fout;  }  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) // 시작과 끝을 설정하는데 어쩌라는 거지  {  vrtxLst.push\_back(\*pVrtx\_1);  vrtxLst.push\_back(\*pVrtx\_2);  }  Vertex opposite(Vertex v)  {  if (v == \*pVrtx\_1)  return \*pVrtx\_2;  else if (v == \*pVrtx\_2)  return \*pVrtx\_1;  else {  //cout << "Error in opposite()" << endl;  return Vertex(NULL);  }  }  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; // 최신화된 데이터를 사용하기 위해 포인터 사용  Vertex\* pVrtx\_2;  int distance;  EdgeStatus edgeStatus;  }; // end class Edge  public:  Graph() : name(""), pVrtxArray(NULL), pAdjLstArray(NULL) {} // default constructor  Graph(string nm, int num\_nodes) : name(nm), pVrtxArray(NULL), pAdjLstArray(NULL)  {  num\_vertices = num\_nodes;  pVrtxArray = new Graph::Vertex[num\_vertices]; // 개수가 num\_vertices인 이유  for (int i = 0; i < num\_nodes; i++)  pVrtxArray[i] = NULL;  pAdjLstArray = new EdgeList[num\_vertices]; // 개수가 num\_vertices인 이유  for (int i = 0; i < num\_vertices; i++)  pAdjLstArray[i].clear();  }  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 printGraph();  bool isValidvID(int vid);  private:  string name;  Vertex\* pVrtxArray;  EdgeList\* pAdjLstArray;  int num\_vertices;  };  bool Graph::isAdjacentTo(Vertex v, Vertex w)  {  return true;  }  void Graph::eraseEdge(Edge e)  {  }  void Graph::eraseVertex(Vertex v)  {  }  void Graph::insertVertex(Vertex& v)  {  int vID;  vID = v.getID();  if (pVrtxArray[vID] == NULL) {  pVrtxArray[vID] = v;  }  }  void Graph::vertices(VrtxList& vrtxLst)  {  vrtxLst.clear();  for (int i = 0; i < getNumVertices(); i++)  vrtxLst.push\_back(pVrtxArray[i]);  }  void Graph::insertEdge(Edge& e)  {  Vertex vrtx\_1, vrtx\_2;  Vertex\* pVtx;  int vID\_1, vID\_2;  vrtx\_1 = \*e.getpVrtx\_1(); vID\_1 = vrtx\_1.getID();  vrtx\_2 = \*e.getpVrtx\_2(); vID\_2 = vrtx\_2.getID();  if (pVrtxArray[vID\_1] == NULL) {  pVrtxArray[vID\_1] = vrtx\_1;  }  if (pVrtxArray[vID\_2] == NULL) {  pVrtxArray[vID\_2] = vrtx\_2;  }  e.setpVrtx\_1(&pVrtxArray[vID\_1]);  e.setpVrtx\_2(&pVrtxArray[vID\_2]);  pAdjLstArray[vID\_1].push\_back(e); // 해당 vertex에 edge 추가  }  void Graph::edges(EdgeList& edges)  {  EdgeItor eItor;  Graph::Edge e;  edges.clear();  for (int i = 0; i < getNumVertices(); i++)  {  eItor = pAdjLstArray[i].begin();  while (eItor != pAdjLstArray[i].end())  {  e = \*eItor;  edges.push\_front(e);  eItor++;  }  }  }  void Graph::incidentEdges(Vertex v, EdgeList& edgeLst)  {  Graph::Edge e;  EdgeItor eItor;  int vID = v.getID();  eItor = pAdjLstArray[vID].begin();  while (eItor != pAdjLstArray[vID].end())  {  e = \*eItor;  edgeLst.push\_back(e);  eItor++;  }  }  bool Graph::isValidvID(int vid)  {  if ((vid >= 0) && (vid < num\_vertices))  return true;  else  {  cout << "Vertex ID (" << vid << ") is invalid for Graph (" << getName()  << ") with num\_vertices (" << num\_vertices << ")" << endl;  }  }  void Graph::printGraph()  {  int i, j;  EdgeItor eItor;  Graph::Edge e;  int numOutgoingEdges;  cout << this->getName() << " with " << this->getNumVertices()  << " vertices has following connectivity :" << endl;  for (i = 0; i < num\_vertices; i++)  {  cout << " vertex (" << setw(3) << pVrtxArray[i].getName() << ") : ";  numOutgoingEdges = pAdjLstArray[i].size();  eItor = pAdjLstArray[i].begin();  while (eItor != pAdjLstArray[i].end())  {  e = \*eItor;  cout << e << " ";  eItor++;  }  cout << endl;  }  }  #endif |
| /\*\* DepthFirstSearch.h (1)\*/  #ifndef DFS\_H  #define DFS\_H  #include <iostream>  #include <algorithm>  #include "Graph.h"  using namespace std;  typedef Graph::Vertex Vertex;  typedef Graph::Edge Edge;  typedef std::list<Graph::Vertex> VrtxList;  typedef std::list<Graph::Vertex>::iterator VertexItor;  typedef std::list<Graph::Edge> EdgeList;  typedef std::list<Graph::Edge>::iterator EdgeItor;  class DepthFirstSearch  {  protected:  Graph& graph;  Vertex start;  bool done; // flag of search done  protected:  void initialize();  void dfsTraversal(Vertex& v, Vertex& target, VrtxList& path);  virtual void traverseDiscovery(const Edge& e, const Vertex& from) { }  virtual void traverseBack(const Edge& e, const Vertex& from) { }  virtual void finishVisit(const Vertex& v) {}  virtual bool isDone() const { return done; }  // marking utilities  void visit(Vertex& v);  void visit(Edge& e);  void unvisit(Vertex& v);  void unvisit(Edge& e);  bool isVisited(Vertex& v);  bool isVisited(Edge& e);  void setEdgeStatus(Edge& e, EdgeStatus es);  EdgeStatus getEdgeStatus(Edge& e);  public:  DepthFirstSearch(Graph& g);  void findPath(Vertex& s, Vertex& t, VrtxList& path);  Graph& getGraph() { return graph; }  void showConnectivity(ofstream& fout);  private:  VrtxStatus\* pVrtxStatus;  EdgeStatus\*\* ppEdgeStatus;  int\*\* ppConnectivity; // two dimensional array; table of distance[v1][v2]  }; // end of class DepthFirstSearch  DepthFirstSearch::DepthFirstSearch(Graph& g) :graph(g)  {  int num\_nodes = graph.getNumVertices();  pVrtxStatus = new VrtxStatus[num\_nodes];  for (int i = 0; i < num\_nodes; i++)  pVrtxStatus[i] = UN\_VISITED;  ppEdgeStatus = new EdgeStatus \* [num\_nodes];  for (int i = 0; i < num\_nodes; i++)  ppEdgeStatus[i] = new EdgeStatus[num\_nodes];  for (int i = 0; i < num\_nodes; i++)  for (int j = 0; j < num\_nodes; j++)  ppEdgeStatus[i][j] = EDGE\_UN\_VISITED;  ppConnectivity = new int\* [num\_nodes];  for (int i = 0; i < num\_nodes; i++)  ppConnectivity[i] = new int[num\_nodes];  for (int i = 0; i < num\_nodes; i++)  for (int j = 0; j < num\_nodes; j++)  ppConnectivity[i][j] = PLUS\_INF; // initially not connected  Vertex vrtx\_1, vrtx\_2;  int vID\_1, vID\_2;  EdgeList edges;  edges.clear();  graph.edges(edges);  for (EdgeItor pe = edges.begin(); pe != edges.end(); ++pe)  {  vrtx\_1 = \*(\*pe).getpVrtx\_1(); vID\_1 = vrtx\_1.getID();  vrtx\_2 = \*(\*pe).getpVrtx\_2(); vID\_2 = vrtx\_2.getID();  ppConnectivity[vID\_1][vID\_2] = (\*pe).getDistance();  }  for (int i = 0; i < num\_nodes; i++)  ppConnectivity[i][i] = 0; // distance of same node  }  void DepthFirstSearch::initialize()  {  int num\_nodes = graph.getNumVertices();  VrtxList verts;  graph.vertices(verts);  Vertex vrtx\_1, vrtx\_2;  int vID\_1, vID\_2;  done = false;  for (int i = 0; i < num\_nodes; i++)  pVrtxStatus[i] = UN\_VISITED;  for (int i = 0; i < num\_nodes; i++)  for (int j = 0; j < num\_nodes; j++)  ppEdgeStatus[i][j] = EDGE\_UN\_VISITED;  }  void DepthFirstSearch::showConnectivity(ofstream& fout)  {  int num\_nodes = graph.getNumVertices();  int dist;  Graph g = getGraph();  Vertex\* pVrtxArray = g.getpVrtxArray();  fout << "Connectivity of graph: " << endl;  fout << " |";  for (int i = 0; i < num\_nodes; i++)  fout << setw(5) << pVrtxArray[i].getName();  fout << endl;  fout << "-----+";  for (int i = 0; i < num\_nodes; i++)  fout << "-----";  fout << endl;  for (int i = 0; i < num\_nodes; i++) {  fout << " " << pVrtxArray[i].getName() << " | ";  for (int j = 0; j < num\_nodes; j++) {  dist = ppConnectivity[i][j];  if (dist == PLUS\_INF)  fout << " +oo";  else  fout << setw(5) << dist;  } // end inner for  fout << endl;  } // end outer for  }  void DepthFirstSearch::dfsTraversal(Vertex& v, Vertex& target, VrtxList& path)  {  //startVisit(v);  visit(v);  if (v == target) {  done = true;  return;  }  EdgeList incidentEdges;  incidentEdges.clear();  graph.incidentEdges(v, incidentEdges);  EdgeItor pe = incidentEdges.begin();  while (!isDone() && pe != incidentEdges.end())  {  Edge e = \*pe++;  EdgeStatus eStat = getEdgeStatus(e);  if (eStat == EDGE\_UN\_VISITED)  {  visit(e);  Vertex w = e.opposite(v);  if (!isVisited(w))  {  //traverseDiscovery(e, v);  path.push\_back(w);  setEdgeStatus(e, DISCOVERY);  if (!isDone()) {  dfsTraversal(w, target, path); // recursive call  if (!isDone()) {  //traverseBack(e, v);  // check whether node w is already in path as a cycle  Vertex last\_pushed = path.back(); // for debugging  path.pop\_back();  }  }  }  else // w is VISITED  {  setEdgeStatus(e, BACK);  }  } // end if (eStat == EDGE\_UN\_VISITED)  } // end of while - processing of all incedent edges  }  void DepthFirstSearch::findPath(Vertex& start, Vertex& target, VrtxList& path)  {  initialize();  path.clear();  path.push\_back(start);  dfsTraversal(start, target, path);  }  void DepthFirstSearch::visit(Vertex& v)  {  Graph::Vertex\* pVtx;  int numNodes = getGraph().getNumVertices();  int vID = v.getID();  if (graph.isValidvID(vID))  {  pVrtxStatus[vID] = VISITED;  }  }  void DepthFirstSearch::visit(Edge& e)  {  Vertex vrtx\_1, vrtx\_2;  int vID\_1, vID\_2;  int numNodes = getGraph().getNumVertices();  vrtx\_1 = \*e.getpVrtx\_1(); vID\_1 = vrtx\_1.getID();  vrtx\_2 = \*e.getpVrtx\_2(); vID\_2 = vrtx\_2.getID();  if (graph.isValidvID(vID\_1) && graph.isValidvID(vID\_2))  {  ppEdgeStatus[vID\_1][vID\_2] = EDGE\_VISITED;  }  }  void DepthFirstSearch::unvisit(Vertex& v)  {  Graph::Vertex\* pVtx;  int numNodes = getGraph().getNumVertices();  int vID = v.getID();  if (graph.isValidvID(vID))  {  pVrtxStatus[vID] = UN\_VISITED;  }  }  void DepthFirstSearch::unvisit(Edge& e)  {  Vertex vrtx\_1, vrtx\_2;  int vID\_1, vID\_2;  int numNodes = getGraph().getNumVertices();  vrtx\_1 = \*e.getpVrtx\_1(); vID\_1 = vrtx\_1.getID();  vrtx\_2 = \*e.getpVrtx\_2(); vID\_2 = vrtx\_2.getID();  if (graph.isValidvID(vID\_1) && graph.isValidvID(vID\_2))  {  ppEdgeStatus[vID\_1][vID\_2] = EDGE\_UN\_VISITED;  }  }  bool DepthFirstSearch::isVisited(Vertex& v)  {  Graph::Vertex\* pVtx;  int numNodes = getGraph().getNumVertices();  int vID = v.getID();  if (graph.isValidvID(vID))  {  return (pVrtxStatus[vID] == VISITED);  }  }  bool DepthFirstSearch::isVisited(Edge& e)  {  Vertex vrtx\_1, vrtx\_2;  int vID\_1, vID\_2;  EdgeStatus eStat;  int numNodes = getGraph().getNumVertices();  vrtx\_1 = \*e.getpVrtx\_1(); vID\_1 = vrtx\_1.getID();  vrtx\_2 = \*e.getpVrtx\_2(); vID\_2 = vrtx\_2.getID();  if (graph.isValidvID(vID\_1) && graph.isValidvID(vID\_2))  {  eStat = ppEdgeStatus[vID\_1][vID\_2];  if ((eStat == EDGE\_VISITED) || (eStat == DISCOVERY) || (eStat == BACK))  return true;  else  return false;  }  return false;  }  void DepthFirstSearch::setEdgeStatus(Edge& e, EdgeStatus es)  {  Vertex vrtx\_1, vrtx\_2;  int vID\_1, vID\_2;  int numNodes = getGraph().getNumVertices();  vrtx\_1 = \*e.getpVrtx\_1(); vID\_1 = vrtx\_1.getID();  vrtx\_2 = \*e.getpVrtx\_2(); vID\_2 = vrtx\_2.getID();  if (graph.isValidvID(vID\_1) && graph.isValidvID(vID\_2))  {  ppEdgeStatus[vID\_1][vID\_2] = es;  }  }  EdgeStatus DepthFirstSearch::getEdgeStatus(Edge& e)  {  Vertex vrtx\_1, vrtx\_2;  int vID\_1, vID\_2;  int numNodes = getGraph().getNumVertices();  EdgeStatus eStat;  vrtx\_1 = \*e.getpVrtx\_1(); vID\_1 = vrtx\_1.getID();  vrtx\_2 = \*e.getpVrtx\_2(); vID\_2 = vrtx\_2.getID();  if (graph.isValidvID(vID\_1) && graph.isValidvID(vID\_2))  {  eStat = ppEdgeStatus[vID\_1][vID\_2];  return eStat;  }  else {  cout << "Edge (" << e << ") was not found from AdjacencyList" << endl;  return EDGE\_NOT\_FOUND;  }  }  #endif |
|  |

**13.2 Dijkstra 알고리즘을 사용한 최단거리 경로 탐색**

|  |
| --- |
| /\* main.cpp \*/  /\* Description  \* Graph 구현및 기능 검사 (BFS\_Dijkstra)  \* Programmed by J. H. Kim  \* Last updated : 2021-12-07 \*/  #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 11  #define NUM\_EDGES 34  Vertex v[NUM\_NODES] = // 7 nodes  {  Vertex("SL", 0), Vertex("CC", 1), Vertex("SC", 2),  Vertex("SW", 3), Vertex("WJ", 4), Vertex("GR", 5),  Vertex("DJ", 6), Vertex("DG", 7), Vertex("PH", 8),  Vertex("GJ", 9), Vertex("BS", 10)  };  Graph::Edge edges[NUM\_EDGES] = // 70 edges  {  Edge(v[0], v[1], 71), Edge(v[1], v[0], 71),  Edge(v[0], v[3], 34), Edge(v[3], v[0], 34),  Edge(v[1], v[2], 79), Edge(v[2], v[1], 79),  Edge(v[1], v[4], 47), Edge(v[4], v[1], 47),  Edge(v[2], v[5], 42), Edge(v[5], v[2], 42),  Edge(v[3], v[4], 84), Edge(v[4], v[3], 84),  Edge(v[3], v[6], 109), Edge(v[6], v[3], 109),  Edge(v[4], v[5], 91), Edge(v[5], v[4], 91),  Edge(v[4], v[7], 174), Edge(v[7], v[4], 174),  Edge(v[5], v[8], 200), Edge(v[8], v[5], 200),  Edge(v[6], v[7], 120), Edge(v[7], v[6], 120),  Edge(v[6], v[9], 138), Edge(v[9], v[6], 138),  Edge(v[7], v[8], 66), Edge(v[8], v[7], 66),  Edge(v[7], v[9], 170), Edge(v[9], v[7], 170),  Edge(v[7], v[10], 87), Edge(v[10], v[7], 87),  Edge(v[8], v[10], 93), Edge(v[10], v[8], 93),  Edge(v[9], v[10], 202), Edge(v[10], v[9], 202),  };  int test\_start = 0;  int test\_end = 10;  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\_start] << " to " << v[test\_end] << " : ";  for (VrtxItor vItor = path.begin(); vItor != path.end(); ++vItor)  {  fout << \*vItor;  if (\*vItor != v[test\_end])  fout << " -> ";  }  fout << endl;  fout.close();  } |
| /\*\* 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 // 오버플로우 방지  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  class Edge // Graph::Edge  {  friend ostream& operator<<(ostream& fout, Edge& e)  {  fout << "Edge(" << setw(2) << \*e.getpVrtx\_1() << ", " << setw(2)  << \*e.getpVrtx\_2() << ", " << setw(4) << e.getDistance() << ")";  return fout;  }  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) // 시작과 끝을 설정하는데 어쩌라는 거지  {  vrtxLst.push\_back(\*pVrtx\_1);  vrtxLst.push\_back(\*pVrtx\_2);  }  Vertex opposite(Vertex v)  {  if (v == \*pVrtx\_1)  return \*pVrtx\_2;  else if (v == \*pVrtx\_2)  return \*pVrtx\_1;  else {  //cout << "Error in opposite()" << endl;  return Vertex(NULL);  }  }  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; // 최신화된 데이터를 사용하기 위해 포인터 사용  Vertex\* pVrtx\_2;  int distance;  EdgeStatus edgeStatus;  }; // end class Edge  public:  Graph() : name(""), pVrtxArray(NULL), pAdjLstArray(NULL) {} // default constructor  Graph(string nm, int num\_nodes) : name(nm), pVrtxArray(NULL), pAdjLstArray(NULL)  {  num\_vertices = num\_nodes;  pVrtxArray = new Graph::Vertex[num\_vertices]; // 개수가 num\_vertices인 이유  for (int i = 0; i < num\_nodes; i++)  pVrtxArray[i] = NULL;  pAdjLstArray = new EdgeList[num\_vertices]; // 개수가 num\_vertices인 이유  for (int i = 0; i < num\_vertices; i++)  pAdjLstArray[i].clear();  }  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;  };  bool Graph::isAdjacentTo(Vertex v, Vertex w)  {  return true;  }  void Graph::eraseEdge(Edge e)  {  }  void Graph::eraseVertex(Vertex v)  {  }  void Graph::insertVertex(Vertex& v)  {  int vID;  vID = v.getID();  if (pVrtxArray[vID] == NULL) {  pVrtxArray[vID] = v;  }  }  void Graph::vertices(VrtxList& vrtxLst)  {  vrtxLst.clear();  for (int i = 0; i < getNumVertices(); i++)  vrtxLst.push\_back(pVrtxArray[i]);  }  void Graph::insertEdge(Edge& e)  {  Vertex vrtx\_1, vrtx\_2;  Vertex\* pVtx;  int vID\_1, vID\_2;  vrtx\_1 = \*e.getpVrtx\_1(); vID\_1 = vrtx\_1.getID();  vrtx\_2 = \*e.getpVrtx\_2(); vID\_2 = vrtx\_2.getID();  if (pVrtxArray[vID\_1] == NULL) {  pVrtxArray[vID\_1] = vrtx\_1;  }  if (pVrtxArray[vID\_2] == NULL) {  pVrtxArray[vID\_2] = vrtx\_2;  }  e.setpVrtx\_1(&pVrtxArray[vID\_1]);  e.setpVrtx\_2(&pVrtxArray[vID\_2]);  pAdjLstArray[vID\_1].push\_back(e); // 해당 vertex에 edge 추가  }  void Graph::edges(EdgeList& edges)  {  EdgeItor eItor;  Graph::Edge e;  edges.clear();  for (int i = 0; i < getNumVertices(); i++)  {  eItor = pAdjLstArray[i].begin();  while (eItor != pAdjLstArray[i].end())  {  e = \*eItor;  edges.push\_front(e);  eItor++;  }  }  }  void Graph::incidentEdges(Vertex v, EdgeList& edgeLst)  {  Graph::Edge e;  EdgeItor eItor;  int vID = v.getID();  eItor = pAdjLstArray[vID].begin();  while (eItor != pAdjLstArray[vID].end())  {  e = \*eItor;  edgeLst.push\_back(e);  eItor++;  }  }  bool Graph::isValidvID(int vid)  {  if ((vid >= 0) && (vid < num\_vertices))  return true;  else  {  cout << "Vertex ID (" << vid << ") is invalid for Graph (" << getName()  << ") with num\_vertices (" << num\_vertices << ")" << endl;  }  }  void Graph::fprintGraph(ofstream& fout)  {  int i, j;  EdgeItor eItor;  Graph::Edge e;  int numOutgoingEdges;  fout << this->getName() << " with " << this->getNumVertices()  << " vertices has following connectivity :" << endl;  for (i = 0; i < num\_vertices; i++)  {  fout << " vertex (" << setw(3) << pVrtxArray[i].getName() << ") : ";  numOutgoingEdges = pAdjLstArray[i].size();  eItor = pAdjLstArray[i].begin();  while (eItor != pAdjLstArray[i].end())  {  e = \*eItor;  fout << e << " ";  eItor++;  }  fout << endl;  }  }  #endif |
| /\*\* BFS\_Dijkstra.h \*/  #ifndef BFS\_DIJKSTRA\_H  #define BFS\_DIJKSTRA\_H  #include <algorithm>  #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; }  };  void BreadthFirstSearch::initialize()  {  Vertex\* pVrtx = getGraph().getpVrtxArray();  VrtxList vrtxLst;  graph.vertices(vrtxLst);  int num\_vertices = graph.getNumVertices();  for (int vID = 0; vID < num\_vertices; vID++)  pVrtx[vID].setVrtxStatus(UN\_VISITED);  EdgeList edges;  graph.edges(edges);  for (EdgeItor pe = edges.begin(); pe != edges.end(); ++pe)  pe->setEdgeStatus(EDGE\_UN\_VISITED);  }  void BreadthFirstSearch::initDistMtrx()  {  int\*\* ppDistMtrx;  int\* pLeaseCostMtrx;  int num\_nodes;  Vertex\* pVrtxArray;  EdgeList\* pAdjLstArray;  int curVID, vID;  num\_nodes = getNumVertices();  pVrtxArray = graph.getpVrtxArray();  pAdjLstArray = graph.getpAdjLstArray();  ppDistMtrx = getppDistMtrx();  for (int i = 0; i < num\_nodes; i++)  {  curVID = pVrtxArray[i].getID();  EdgeItor pe = pAdjLstArray[curVID].begin();  while (pe != pAdjLstArray[curVID].end())  {  vID = (\*(\*pe).getpVrtx\_2()).getID();  ppDistMtrx[curVID][vID] = (\*pe).getDistance();  pe++;  }  ppDistMtrx[curVID][curVID] = 0;  }  }  void BreadthFirstSearch::fprintDistMtrx(ofstream& fout)  {  int\*\* ppDistMtrx;  Vertex\* pVrtxArray;  int num\_nodes;  int dist;  int vID;  string vName;  pVrtxArray = graph.getpVrtxArray();  num\_nodes = getNumVertices();  ppDistMtrx = getppDistMtrx();  fout << "\nDistance Matrix of Graph (" << graph.getName() << ") :" << endl;  fout << " |";  for (int i = 0; i < num\_nodes; i++) {  vName = pVrtxArray[i].getName();  fout << setw(5) << vName;  }  fout << endl;  fout << "-------+";  for (int i = 0; i < num\_nodes; i++) {  fout << "-----";  }  fout << endl;  for (int i = 0; i < num\_nodes; i++) {  vName = pVrtxArray[i].getName();  fout << setw(5) << vName << " |";  for (int j = 0; j < num\_nodes; j++) {  dist = ppDistMtrx[i][j];  if (dist == PLUS\_INF)  fout << " +oo";  else  fout << setw(5) << dist;  }  fout << endl;  }  fout << endl;  }  enum BFS\_PROCESS\_STATUS { NOT\_SELECTED, SELECTED };  void BreadthFirstSearch::DijkstraShortestPath(ofstream& fout, Vertex& start, Vertex& target,  VrtxList& path)  {  int\*\* ppDistMtrx;  int\* pLeastCost;  int num\_nodes, num\_selected;  int minID, minCost;  BFS\_PROCESS\_STATUS\* pBFS\_Process\_Stat;  int\* pPrev;  Vertex\* pVrtxArray;  Vertex vrtx, \* pPrevVrtx, v;  Edge e;  int start\_vID, target\_vID, curVID, vID;  EdgeList\* pAdjLstArray;  pVrtxArray = graph.getpVrtxArray();  pAdjLstArray = graph.getpAdjLstArray();  start\_vID = start.getID();  target\_vID = target.getID();  num\_nodes = getNumVertices();  ppDistMtrx = getppDistMtrx();  pLeastCost = new int[num\_nodes];  pPrev = new int[num\_nodes];  pBFS\_Process\_Stat = new BFS\_PROCESS\_STATUS[num\_nodes];  // initialize L(n) = w(start, n);  for (int i = 0; i < num\_nodes; i++)  {  pLeastCost[i] = ppDistMtrx[start\_vID][i]; // 시작점으로부터 목적지까지의 edge weight 반환  pPrev[i] = start\_vID;  pBFS\_Process\_Stat[i] = NOT\_SELECTED;  }  pBFS\_Process\_Stat[start\_vID] = SELECTED; // 클라우드 선택  num\_selected = 1;  path.clear();  int round = 0;  int cost;  string vName;  fout << "Dijkstra::Least Cost from Vertex (" << start.getName() << ") at each round : " << endl;  fout << " |";  for (int i = 0; i < num\_nodes; i++)  {  vName = pVrtxArray[i].getName();  fout << setw(5) << vName;  }  fout << endl;  fout << "-----------+";  for (int i = 0; i < num\_nodes; i++)  {  fout << setw(5) << "-----";  }  fout << endl;  while (num\_selected < num\_nodes)  {  round++;  fout << "round [" << setw(2) << round << "] |";  minID = -1;  minCost = PLUS\_INF;  for (int i = 0; i < num\_nodes; i++)  {  if ((pLeastCost[i] < minCost) && (pBFS\_Process\_Stat[i] != SELECTED)) {  minID = i;  minCost = pLeastCost[i];  }  }  if (minID == -1) { // 연결되지 않은 vertex 존재  fout << "Error in Dijkstra() -- found not connected vertex !!" << endl;  break;  }  else  {  pBFS\_Process\_Stat[minID] = SELECTED; // 가장 작은 vertex cloud 선택  num\_selected++;  if (minID == target\_vID) // 목적지 도착  {  fout << endl << "reached to the target node ("  << pVrtxArray[minID].getName() << ") at Least Cost = " << minCost << endl;  vID = minID;  do { // 목적지부터 출발지까지 역추적  vrtx = pVrtxArray[vID];  path.push\_front(vrtx);  vID = pPrev[vID];  } while (vID != start\_vID);  vrtx = pVrtxArray[vID];  path.push\_front(vrtx); // start node  break;  }  }  /\* Edge relaxation \*/  int pLS, ppDistMtrx\_i;  for (int i = 0; i < num\_nodes; i++)  {  pLS = pLeastCost[i];  ppDistMtrx\_i = ppDistMtrx[minID][i];  if ((pBFS\_Process\_Stat[i] != SELECTED) && (pLeastCost[i] >  (pLeastCost[minID] + ppDistMtrx[minID][i])))  {  pPrev[i] = minID;  pLeastCost[i] = pLeastCost[minID] + ppDistMtrx[minID][i];  }  }  // print out the pLeastCost[] for debugging  for (int i = 0; i < num\_nodes; i++)  {  cost = pLeastCost[i];  if (cost == PLUS\_INF)  fout << " +oo";  else  fout << setw(5) << pLeastCost[i];  }  fout << " ==> selected vertex : " << pVrtxArray[minID] << endl;  } // end while()  } // end DijkstraShortestPath()  #endif |
|  |

**13.3 Minimum Spanning Tree 산출**

|  |
| --- |
| /\* main.cpp \*/  /\* Description  \* Graph 구현및 기능 검사 (MST)  \* Programmed by J. H. Kim  \* Last updated : 2021-12-07 \*/  #include <iostream>  #include <fstream>  #include <string>  #include "Graph.h"  #include "MinimumSpanningTree.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 11  #define NUM\_EDGES 34  Vertex v[NUM\_NODES] = // 7 nodes  {  Vertex("SL", 0), Vertex("CC", 1), Vertex("SC", 2),  Vertex("SW", 3), Vertex("WJ", 4), Vertex("GR", 5),  Vertex("DJ", 6), Vertex("DG", 7), Vertex("PH", 8),  Vertex("GJ", 9), Vertex("BS", 10)  };  Graph::Edge edges[NUM\_EDGES] = // 70 edges  {  Edge(v[0], v[1], 71), Edge(v[1], v[0], 71),  Edge(v[0], v[3], 34), Edge(v[3], v[0], 34),  Edge(v[1], v[2], 79), Edge(v[2], v[1], 79),  Edge(v[1], v[4], 47), Edge(v[4], v[1], 47),  Edge(v[2], v[5], 42), Edge(v[5], v[2], 42),  Edge(v[3], v[4], 84), Edge(v[4], v[3], 84),  Edge(v[3], v[6], 109), Edge(v[6], v[3], 109),  Edge(v[4], v[5], 91), Edge(v[5], v[4], 91),  Edge(v[4], v[7], 174), Edge(v[7], v[4], 174),  Edge(v[5], v[8], 200), Edge(v[8], v[5], 200),  Edge(v[6], v[7], 120), Edge(v[7], v[6], 120),  Edge(v[6], v[9], 138), Edge(v[9], v[6], 138),  Edge(v[7], v[8], 66), Edge(v[8], v[7], 66),  Edge(v[7], v[9], 170), Edge(v[9], v[7], 170),  Edge(v[7], v[10], 87), Edge(v[10], v[7], 87),  Edge(v[8], v[10], 93), Edge(v[10], v[8], 93),  Edge(v[9], v[10], 202), Edge(v[10], v[9], 202),  };  int test\_start = 0;  int test\_end = 10;  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;  MinimumSpanningTree MSTGraph(simpleGraph);  MSTGraph.initDistMtrx();  //fout << "Distance matrix of BFS for Graph:" << endl;  MSTGraph.fprintDistMtrx(fout);  MSTGraph.PrimJarnikMST();  fout << endl;  fout.close();  } |
| /\*\* 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 // 오버플로우 방지  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  class Edge // Graph::Edge  {  friend ostream& operator<<(ostream& fout, Edge& e)  {  fout << "Edge(" << setw(2) << \*e.getpVrtx\_1() << ", " << setw(2)  << \*e.getpVrtx\_2() << ", " << setw(4) << e.getDistance() << ")";  return fout;  }  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) // 시작과 끝을 설정하는데 어쩌라는 거지  {  vrtxLst.push\_back(\*pVrtx\_1);  vrtxLst.push\_back(\*pVrtx\_2);  }  Vertex opposite(Vertex v)  {  if (v == \*pVrtx\_1)  return \*pVrtx\_2;  else if (v == \*pVrtx\_2)  return \*pVrtx\_1;  else {  //cout << "Error in opposite()" << endl;  return Vertex(NULL);  }  }  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; // 최신화된 데이터를 사용하기 위해 포인터 사용  Vertex\* pVrtx\_2;  int distance;  EdgeStatus edgeStatus;  }; // end class Edge  public:  Graph() : name(""), pVrtxArray(NULL), pAdjLstArray(NULL) {} // default constructor  Graph(string nm, int num\_nodes) : name(nm), pVrtxArray(NULL), pAdjLstArray(NULL)  {  num\_vertices = num\_nodes;  pVrtxArray = new Graph::Vertex[num\_vertices]; // 개수가 num\_vertices인 이유  for (int i = 0; i < num\_nodes; i++)  pVrtxArray[i] = NULL;  pAdjLstArray = new EdgeList[num\_vertices]; // 개수가 num\_vertices인 이유  for (int i = 0; i < num\_vertices; i++)  pAdjLstArray[i].clear();  }  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;  };  bool Graph::isAdjacentTo(Vertex v, Vertex w)  {  return true;  }  void Graph::eraseEdge(Edge e)  {  }  void Graph::eraseVertex(Vertex v)  {  }  void Graph::insertVertex(Vertex& v)  {  int vID;  vID = v.getID();  if (pVrtxArray[vID] == NULL) {  pVrtxArray[vID] = v;  }  }  void Graph::vertices(VrtxList& vrtxLst)  {  vrtxLst.clear();  for (int i = 0; i < getNumVertices(); i++)  vrtxLst.push\_back(pVrtxArray[i]);  }  void Graph::insertEdge(Edge& e)  {  Vertex vrtx\_1, vrtx\_2;  Vertex\* pVtx;  int vID\_1, vID\_2;  vrtx\_1 = \*e.getpVrtx\_1(); vID\_1 = vrtx\_1.getID();  vrtx\_2 = \*e.getpVrtx\_2(); vID\_2 = vrtx\_2.getID();  if (pVrtxArray[vID\_1] == NULL) {  pVrtxArray[vID\_1] = vrtx\_1;  }  if (pVrtxArray[vID\_2] == NULL) {  pVrtxArray[vID\_2] = vrtx\_2;  }  e.setpVrtx\_1(&pVrtxArray[vID\_1]);  e.setpVrtx\_2(&pVrtxArray[vID\_2]);  pAdjLstArray[vID\_1].push\_back(e); // 해당 vertex에 edge 추가  }  void Graph::edges(EdgeList& edges)  {  EdgeItor eItor;  Graph::Edge e;  edges.clear();  for (int i = 0; i < getNumVertices(); i++)  {  eItor = pAdjLstArray[i].begin();  while (eItor != pAdjLstArray[i].end())  {  e = \*eItor;  edges.push\_front(e);  eItor++;  }  }  }  void Graph::incidentEdges(Vertex v, EdgeList& edgeLst)  {  Graph::Edge e;  EdgeItor eItor;  int vID = v.getID();  eItor = pAdjLstArray[vID].begin();  while (eItor != pAdjLstArray[vID].end())  {  e = \*eItor;  edgeLst.push\_back(e);  eItor++;  }  }  bool Graph::isValidvID(int vid)  {  if ((vid >= 0) && (vid < num\_vertices))  return true;  else  {  cout << "Vertex ID (" << vid << ") is invalid for Graph (" << getName()  << ") with num\_vertices (" << num\_vertices << ")" << endl;  }  }  void Graph::fprintGraph(ofstream& fout)  {  int i, j;  EdgeItor eItor;  Graph::Edge e;  int numOutgoingEdges;  fout << this->getName() << " with " << this->getNumVertices()  << " vertices has following connectivity :" << endl;  for (i = 0; i < num\_vertices; i++)  {  fout << " vertex (" << setw(3) << pVrtxArray[i].getName() << ") : ";  numOutgoingEdges = pAdjLstArray[i].size();  eItor = pAdjLstArray[i].begin();  while (eItor != pAdjLstArray[i].end())  {  e = \*eItor;  fout << e << " ";  eItor++;  }  fout << endl;  }  }  #endif |
| /\*\* MinimumSpanningTree.h \*/  #ifndef MST\_H  #define MST\_H  #include <algorithm>  #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 MinimumSpanningTree  {  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:  MinimumSpanningTree(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);  Graph& getGraph() { return graph; }  int\*\* getppDistMtrx() { return ppDistMtrx; }  void PrimJarnikMST();  };  void MinimumSpanningTree::initialize()  {  Vertex\* pVrtx = getGraph().getpVrtxArray();  VrtxList vrtxLst;  graph.vertices(vrtxLst);  int num\_vertices = graph.getNumVertices();  for (int vID = 0; vID < num\_vertices; vID++)  pVrtx[vID].setVrtxStatus(UN\_VISITED);  EdgeList edges;  graph.edges(edges);  for (EdgeItor pe = edges.begin(); pe != edges.end(); ++pe)  pe->setEdgeStatus(EDGE\_UN\_VISITED);  }  void MinimumSpanningTree::initDistMtrx()  {  int\*\* ppDistMtrx;  int\* pLeaseCostMtrx;  int num\_nodes;  Vertex\* pVrtxArray;  EdgeList\* pAdjLstArray;  int curVID, vID;  num\_nodes = getNumVertices();  pVrtxArray = graph.getpVrtxArray();  pAdjLstArray = graph.getpAdjLstArray();  ppDistMtrx = getppDistMtrx();  for (int i = 0; i < num\_nodes; i++)  {  curVID = pVrtxArray[i].getID();  EdgeItor pe = pAdjLstArray[curVID].begin();  while (pe != pAdjLstArray[curVID].end())  {  vID = (\*(\*pe).getpVrtx\_2()).getID();  ppDistMtrx[curVID][vID] = (\*pe).getDistance();  pe++;  }  ppDistMtrx[curVID][curVID] = 0;  }  }  void MinimumSpanningTree::fprintDistMtrx(ofstream& fout)  {  int\*\* ppDistMtrx;  Vertex\* pVrtxArray;  int num\_nodes;  int dist;  int vID;  string vName;  pVrtxArray = graph.getpVrtxArray();  num\_nodes = getNumVertices();  ppDistMtrx = getppDistMtrx();  fout << "\nDistance Matrix of Graph (" << graph.getName() << ") :" << endl;  fout << " |";  for (int i = 0; i < num\_nodes; i++) {  vName = pVrtxArray[i].getName();  fout << setw(5) << vName;  }  fout << endl;  fout << "-------+";  for (int i = 0; i < num\_nodes; i++) {  fout << "-----";  }  fout << endl;  for (int i = 0; i < num\_nodes; i++) {  vName = pVrtxArray[i].getName();  fout << setw(5) << vName << " |";  for (int j = 0; j < num\_nodes; j++) {  dist = ppDistMtrx[i][j];  if (dist == PLUS\_INF)  fout << " +oo";  else  fout << setw(5) << dist;  }  fout << endl;  }  fout << endl;  }  enum VertexStatus { NOT\_SELECTED, SELECTED };  void MinimumSpanningTree::PrimJarnikMST()  {  int num\_nodes;  int num\_edges;  Vertex\* pVrtxArray;  EdgeList\* pAdjLstArray;  int curVrtx\_ID, vrtxID;  int\*\* ppDistMtrx;  int\* pDist;  int start, min\_id, dist, min\_dist, min\_dist\_org, min\_dist\_end, end\_ID;  VertexStatus\* pVrtxStatus;  Edge\* pParentEdge;  Edge edge, min\_edge; // edge that connects this node to the cloud  std::list<Edge> selectedEdgeLst;  std::list<Edge>::iterator edgeItor;  num\_nodes = graph.getNumVertices();  pVrtxArray = graph.getpVrtxArray();  pAdjLstArray = graph.getpAdjLstArray();  initDistMtrx();  ppDistMtrx = getppDistMtrx();  pDist = new int[num\_nodes];  pVrtxStatus = new VertexStatus[num\_nodes];  pParentEdge = new Edge[num\_nodes];  for (int i = 0; i < num\_nodes; i++) {  pDist[i] = PLUS\_INF;  pVrtxStatus[i] = NOT\_SELECTED;  pParentEdge[i] = Edge();  }  srand(time(0));  start = rand() % num\_nodes; // randomly select start node  cout << "Start node : " << start << endl;  pDist[start] = 0;  selectedEdgeLst.clear();  for (int round = 0; round < num\_nodes; round++) {  min\_dist = PLUS\_INF;  min\_id = -1;  for (int n = 0; n < num\_nodes; n++)  {  if ((pVrtxStatus[n] == NOT\_SELECTED) && (pDist[n] < min\_dist)) {  min\_dist = pDist[n];  min\_id = n;  } // end if  } // end for  if (min\_id == -1)  {  cout << "Error in finding Prim-Jarnik's algorithm !!";  break;  }  pVrtxStatus[min\_id] = SELECTED;  // edge relaxation  EdgeItor pe = pAdjLstArray[min\_id].begin();  while (pe != pAdjLstArray[min\_id].end())  {  end\_ID = ((\*pe).getpVrtx\_2())->getID();  dist = (\*pe).getDistance();  if ((pVrtxStatus[end\_ID] == NOT\_SELECTED) && (dist <= pDist[end\_ID])) {  pDist[end\_ID] = dist;  pParentEdge[end\_ID] = \*pe;  }  pe++;  } // end while  if (min\_id != start) {  min\_edge = pParentEdge[min\_id];  selectedEdgeLst.push\_back(min\_edge);  }  cout << "Dist after round [" << setw(2) << round << "] : ";  for (int i = 0; i < num\_nodes; i++) {  if (pDist[i] == PLUS\_INF)  cout << " +oo ";  else  cout << setw(4) << pDist[i] << " ";  }  cout << endl;  } // end for  cout << "\nEnd of finding Minimum Spanning Tree by Prim-Jarnik's Algorithm";  cout << "selectedEdgeLst.size = " << selectedEdgeLst.size() << endl;  cout << "Selected edges: " << endl;  edgeItor = selectedEdgeLst.begin();  int cnt = 0;  while (edgeItor != selectedEdgeLst.end())  {  cout << \*edgeItor << ", ";  edgeItor++;  if ((++cnt % 5) == 0)  cout << endl;  }  cout << endl;  }  #endif |
|  |