**Lab13**

|  |
| --- |
| 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 | 2021.12.03.금 |

|  |
| --- |
| /\* main.cpp \*/  /\* Description  \* Graph 구현  \* Programmed by J. H. Kim  \* Last updated : 2021-12-03 \*/  #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\_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();  } |
| /\*\* 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 |
| /\*\* 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);  }  }  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 |
|  |

**2. 2021-2 객체지향형 프로그래밍과 자료구조 실습 Oral Test**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 학번 | 21812158 | 성명 | 김주환 | 점수 |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| (1) 그래프를 표현하기 위하여 사용되는 자료구조들을 그림으로 표현하고, 그 복잡도 (complexity)를 정점의 개수와 간선의 개수로 표현하라.  (1) Vertex List, Edge List    (2) Adjacency List    (3) Adjacency Matrix    (4) 복잡도 분석표 (vertex : n, Edge : m)   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | Vertex List | Edge List | Adjacency List | Adjacency Matrix | | 복잡도 (complexity) | O(n) | O(m) | O(n+m) | O(n\*n) | |
| (2) 2 x 3 격자형 그래프에 대한 깊이 우선 탐색 (Depth First Search) 알고리즘을 실행하기 위 하여 구현되는 dfsTraversal() 멤버 함수를 pseudocode 로 표현하고, 상세 동작 절차를 주어진 2 x 3 격자형 그래프의 노드를 사용하여 설명하라.   |  | | --- | | **Procedure dfsTraversal(v, target, path)**   1. // declare arguments : incidentEdges(EdgeList), pe(EdgeItor), e(Edge), eStat(EdgeStatus), w(Vertex) 2. v is visited 3. if v is target -> done is true, return 4. clear incidentEdges 5. graph.incidentEdges(v, incidentEdges) 6. pe is begin of incidentEdges 7. while (!isDone() && pe != incidentEdges.end()) 8. e = \*pe++ 9. eStat = status of e 10. if (eStat == EDGE\_UN\_VISITED) 11. e is visited 12. w = opposite of e’s v 13. if (!isVisited(w)) 14. path.push\_back(w) 15. e is discovery 16. if (!isDone()) 17. dfsTraversal(w, target, path) 18. if (!isDone()) 19. path.pop\_back() 20. else 21. e is back   **End of Procedure dfsTraversal(v, target, path)** |   (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)   |  | | --- | |  | | Vertex 방문순서   * A -> B -> C -> F -> E -> D * 연결된 vertex를 오름차순으로 방문   Edge 구분   * vertex가 unexplored인 경우 edge를 discovery로 변경한다. * vertex가 visited인 경우 edge를 back으로 변경한다. |   (3) DFS로 탐색된 vertex 0 -> vertex 5의 경로   |  |  | | --- | --- | |  | Vertex 방문순서   * A -> B -> C -> F -> E -> D   Edge 상태   * edge(A, B), edge(B, C), edge(C, F), edge(F, E), edge(E, D)는 discovery edge다. * edge(E, B), edge(D, A)는 back edge다. |   (4) vertex 5로 부터의 dfsTraversal() 실행하는 경우 vertex 5 -> vertex 0의 경로   |  | | --- | | Vertex 방문순서   * D -> A * Vertex 방문은 오름차순으로 방문하는 규칙을 가진다. 따라서, D와 연결된 A, E중 A가 낮기 때문에 A를 먼저 방문한다. |   (5) 위 (3)과 (4) 경로에 대한 비교 분석   |  | | --- | | 두 경로의 시작과 끝은 같은 vertex를 가진다.  하지만 실제 경로에는 차이가 발생하는데, 이는 traversal 방식 때문이다.  A가 시작인 경우, 다음 vertex는 D가 아닌 B로 선정된다.  D가 시작인 경우에는, 다음 vertex로 E가 아닌 A가 선정된다.  따라서, 위와 같은 경로로 traversal이 진행됐다. | |
| (3) 그래프에 대한 깊이 우선 탐색 (Depth First Search) 알고리즘, 넓이 우선 탐색 (Breadth First Search) 알고리즘, Dijkstra 알고리즘의 차이점에 대하여 설명하라.  (1) Breadth First Search (BFS) 알고리즘의 동작 절차   |  | | --- | |  | | (a) 탐색할 Graph 준비  (b) start vertex(A)를 선택하고 L0로 지정한다. A에서 갈 수 있는 모든 vertex를 L1으로 지정한다. Edge를 discovery로 연결한다.  (c) L1의 vertex중 작은 vertex(B)를 방문하고, B의 Edge를 discovery edge로 설정한다.  (d) B의 다음 vertex를 방문하지 않고, L1의 다른 vertex(D)를 방문한다. D에서 E로 방문할 수 있지만 이미 E는 visited다. 따라서, edge(D, E)는 cross edge로 설정한다.  (e) L1에서 갈 수 있는 모든 vertex를 L2로 지정하고, 작은 vertex(C)를 방문한다. C에서 갈 수 있는 vertex로 discovery edge를 연결한다.  (f) L2에서 아직 방문하지 않은 E에서도 edge를 연결해야한다. 하지만 이미 F는 visited기 때문에, cross edge로 연결해준다.  (g) L2에서 갈 수 있는 모든 vertex를 L3로 지정한다. |   (2) Breadth First Search (BFS)의 한 종류인 Dijkstra 알고리즘을 사용한 최단거리 경로 탐색 (shortest path search) 기능의 동작 절차   |  | | --- | |  | | (a) vertex 사이 distance가 포함된 Graph를 준비하고 start vertex(LA)를 선택합니다. 선택된 vertex를 Cloud에 포함시키고 상태를 VISITED로 설정합니다.  (b ~ f) Cloud에 포함된 vertex에서 방문 가능한 vertex 중에 가장 가까운 vertex를 방문하고 edge를 연결하여 Cloud에 포함시킨다.  이 과정에서 다른 vertex의 최단 거리 값이 작아졌다면 update 해준다.  (g) 마지막 vertex를 방문하고 edge를 연결하여 Cloud에 포함시키고 목표지점까지의 최단거리를 구할 수 있다. |   (3) Depth First Search (DFS)와 Breadth First Search (BFS) 알고리즘의 기능적 차이점과 주요 응 용 분야에 대한 비교 설명   |  |  |  |  | | --- | --- | --- | --- | |  | DFS | BFS | Dijkstra | | 경로나 사이클의 존재 여부 | ㅇ |  |  | | Spanning tree 기능 제공 | ㅇ | ㅇ |  | | 방향성이 있는 Edge |  |  | ㅇ | | Edge 가중치 |  |  | ㅇ | | 최단 경로 | 보장되지 않음 | edge의 수가 가장 적은 경로 | 가중치의 누적이 가장 적은 경로 | | 응용분야 | 미로 탐색 | 전자회로 부품 배치 | 네비게이션,  비행기 항로 | |
| (4) 그래프의 자료구조의 주요 응용 분야에 대하여 상세하게 설명하라.   |  |  |  | | --- | --- | --- | |  | Vertex와 Edge의 의미 | Graph 탐색의 목적 | | 자동차/스마트폰의 네비게이션 | Vertex : 위치  Edge : 도로 | 최단거리 탐색 | | 인터넷 패킷 라우팅 | Vertex : 라우터  Edge : 라우터간의 연결 | 라우터간의 최단거리 탐색 | | 전자회로 부품 배치 | Vertex : 소자  Edge : 점프선 | 긴밀한 통신  데이터 용량이 큰 전자부품은 근접한 위치에 배치, 통신량이 적은 부품은 멀리 배치 | | 데이터 베이스의 연관 정보 검색을 통한 상관관계 분석 | Vertex : 정보  Edge : 정보간의 연관성 | 정보들 간의 관련성을 그래프로 표시 | |