객체지향프로그래밍과 자료구조

Ch 13. 그래프 자료구조와 관련 알고리즘

정보통신공학과

교수 김 영 탁

(Tel : +82-53-810-2497; E-mail : ytkim@yu.ac.kr)

Outline

- **Graphs**
- **Data Structures for Graphs**
- **Graph Traversal and Search**
  - Depth First Search (DFS)
  - Breadth First Search (BFS)
- **Directed Graphs**
- **Shortest Paths**
  - Dijkstra's Algorithm
- **Minimum Spanning Tree (MST)**

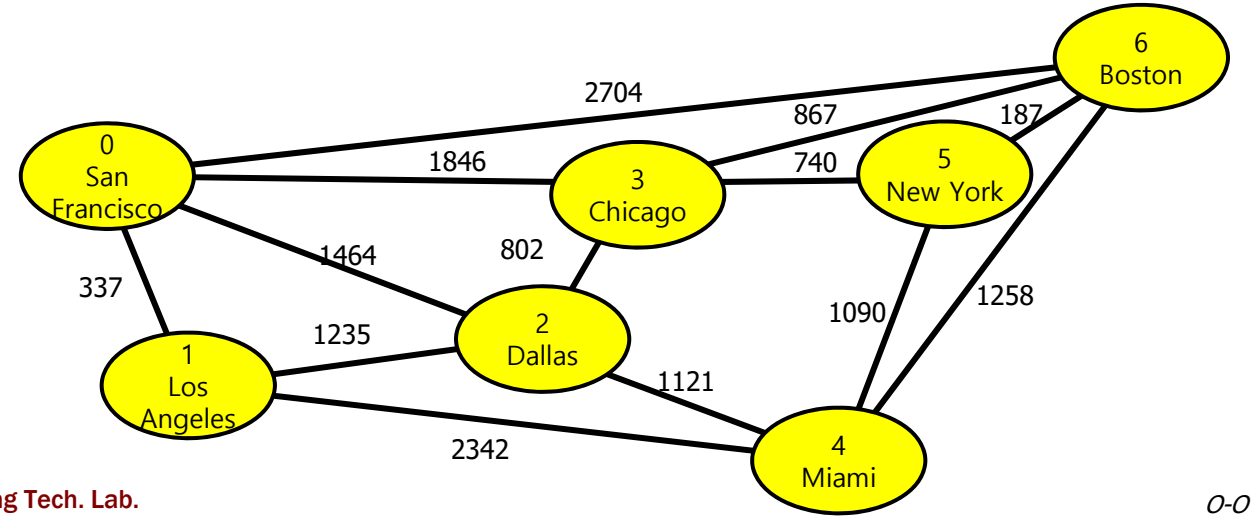
Graphs

A graph is a pair (*V*, *E*), where

- ***V*** is a set of nodes, called **vertices (정점, 노드)**
- ***E*** is a collection of pairs of vertices, called **edges (간선)**
- Vertices and edges are positions, and store elements (information)

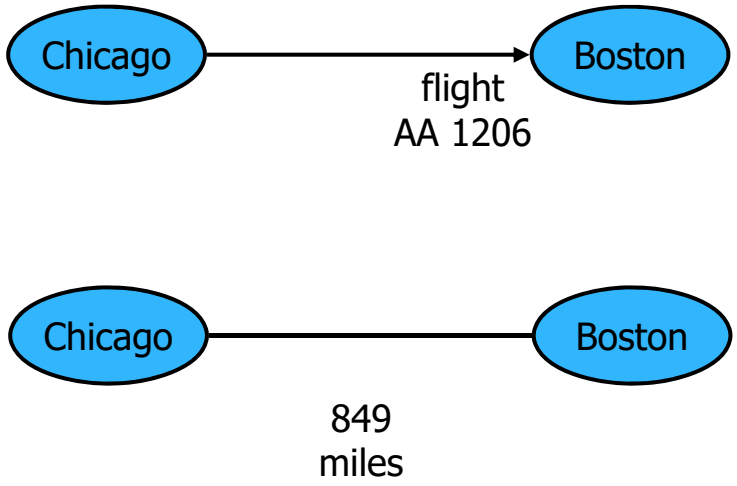
Example:

- A **vertex** (정점, 노드) represents an airport and stores the three-letter airport code
- An **edge** (간선) represents a flight route between two airports and stores the mileage of the route



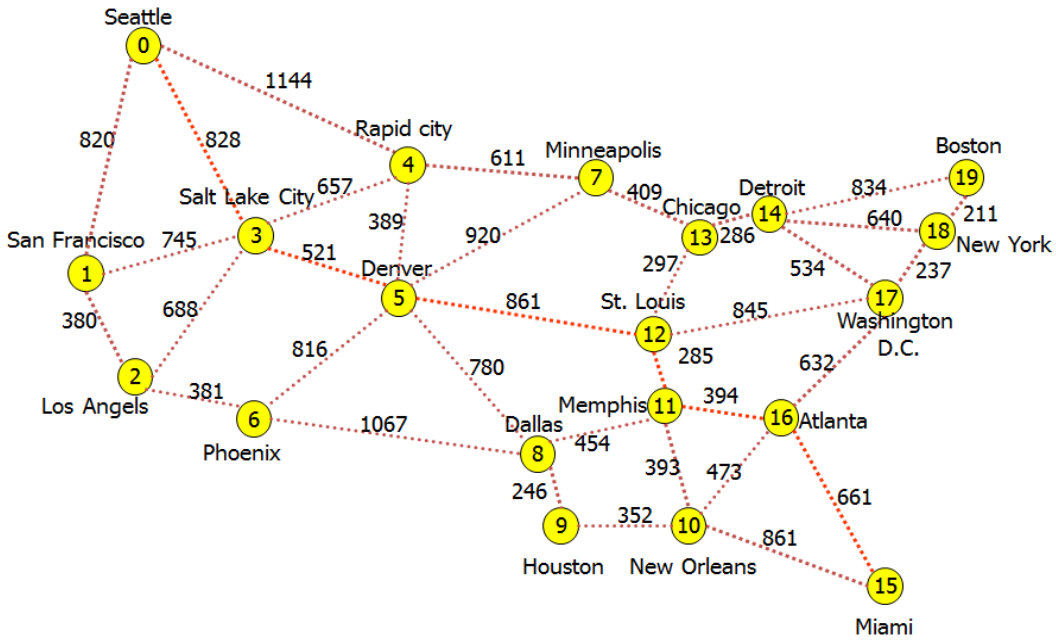
Edge Types

- **Directed edge**
  - ordered pair of vertices (***u***,***v***)
  - first vertex ***u*** is the origin
  - second vertex ***v*** is the destination
  - (u, v) is different from (v, u)
  - e.g., a flight
- **Undirected edge**
  - unordered pair of vertices (***u***,***v***)
  - (u, v) is same as (v, u)
  - e.g., a flight route
- **Directed graph**
  - all the edges are directed
  - e.g., route network
- **Undirected graph**
  - all the edges are undirected
  - e.g., flight network



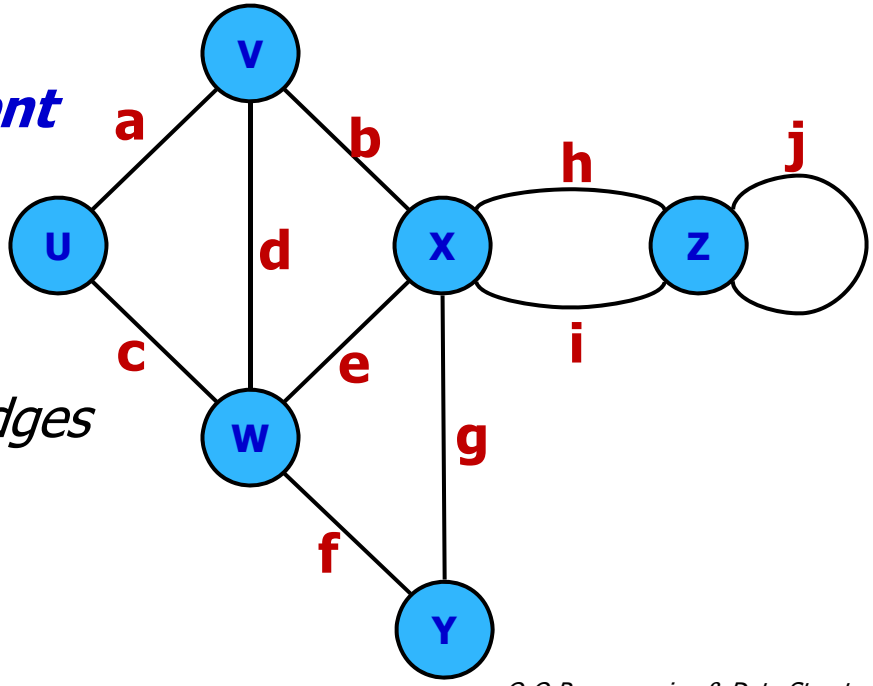
Applications

- **Electronic circuits**
  - Printed circuit board
  - Integrated circuit
- **Transportation netwo**
  - Highway network
  - Flight network
- **Computer networks**
  - Local Area Network (LAN)
  - Internet
  - Web
- **Databases, Machine Learning**
  - Entity-relationship diagram
  - Relationship among knowledge



그래프 관련 용어 (terminology)

- **정점 (Vertex)**
  - U and V are the vertices (*endpoints*) of edge a
  - U and V are ***adjacent*** (인접) each other
- **간선 (Edge)**
  - a, b, and d are ***incident*** (falling or striking, 입사) on vertex V
- **인접 정점(adjacent vertex)**
  - Vertex U and vertex V are ***adjacent***
- **Degree of a vertex**
  - Vertex X has ***degree*** 5
- **Parallel edges**
  - Edge h and edge i are ***parallel*** *edges*
- **Self-loop**
  - Edge j is a ***self-loop***



그래프 관련 용어 (terminology)

◆ 경로 (Path)

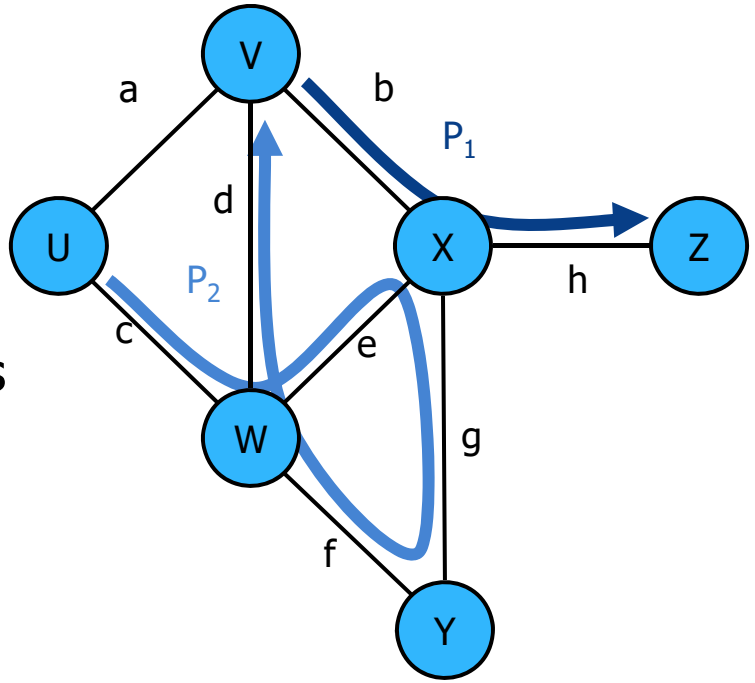
- sequence of alternating vertices and edges
- begins with a vertex, and ends with a vertex
- each edge is preceded and followed by its endpoints (vertices)

◆ 단순 경로 (Simple path)

- path such that all its vertices and edges are distinct without repeated visit (i.e., without loop)

◆ Examples

- P_1=(V,b,X,h,Z) is a simple path
- P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



그래프 관련 용어 (terminology)

◆ 순환(Cycle)

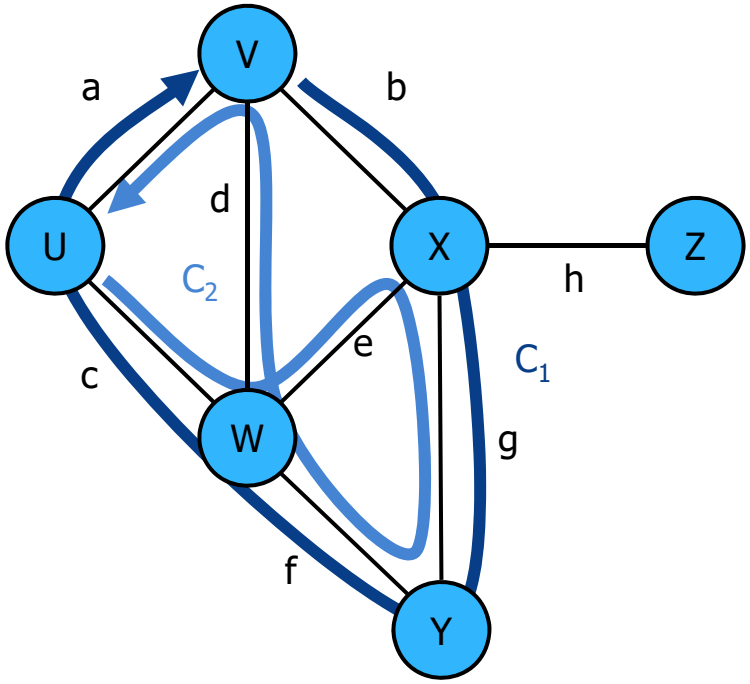
- circular sequence of alternating vertices and edges
- each edge is preceded and followed by its endpoints

◆ 단순 순환 (Simple cycle)

- cycle such that all its vertices and edges are distinct

◆ Examples

- C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
- C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



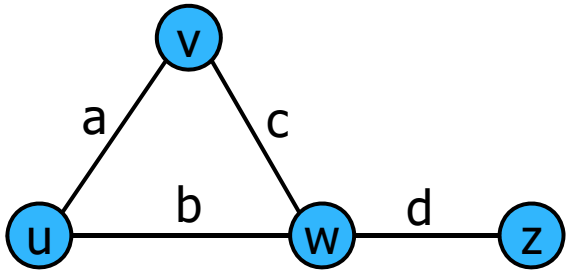
그래프 자료구조

그래프 표현을 위한 자료구조

◆Data structures to represent graphs

- Vertex array / list (정점 배열 또는 리스트)
- Edge list (간선 리스트)
- Adjacency list (인접 리스트) – incident edge list
- Adjacency matrix (인접행렬)

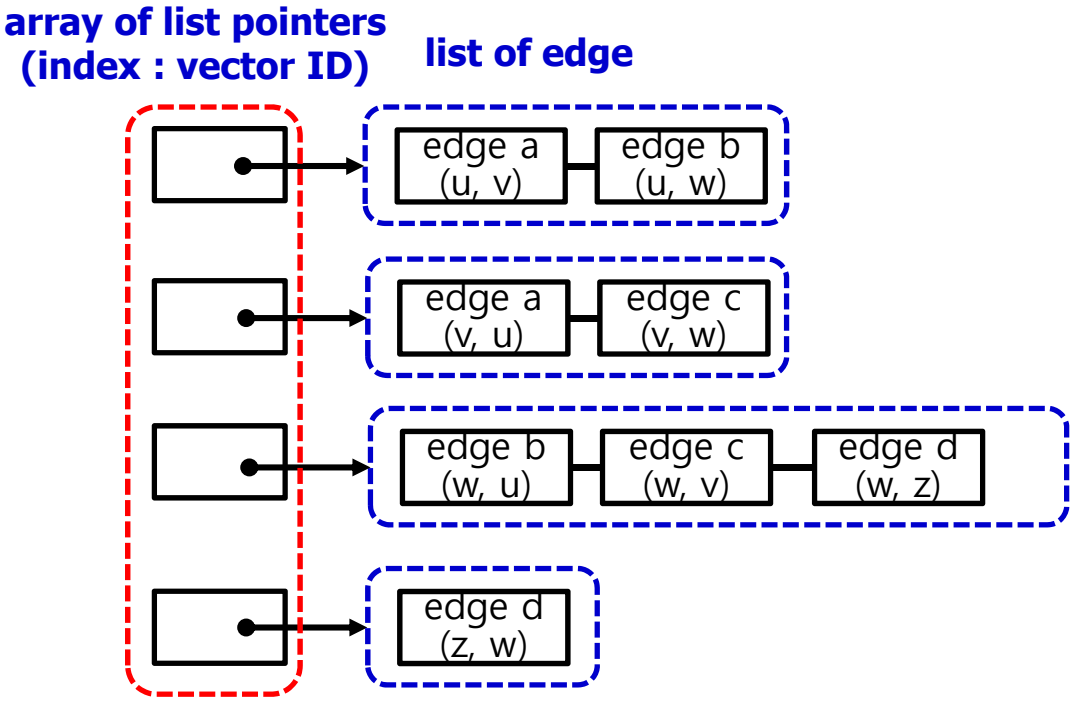
Vertex Array and Adjacency List Array



(a) Graph Topology

Vertex (u, 0)
Vertex (v, 1)
Vertex (w, 2)
Vertex (z, 3)

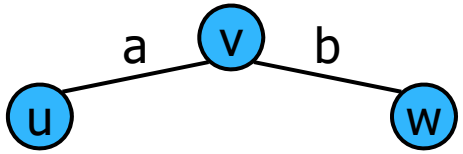
(b) Vertex Array

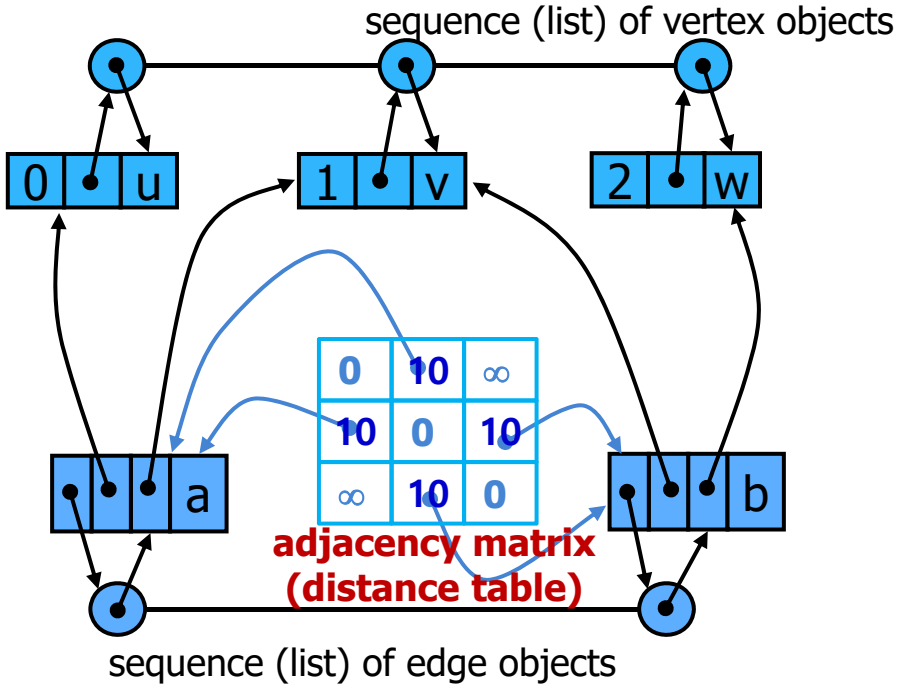


(c) Adjacency List Array

Adjacency Matrix (인접행렬)

- **Edge list (간선리스트) structure**
- **Augmented vertex objects**
  - Integer key (index) associated with vertex
- **2D-array adjacency list array**
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- **Distance table**
  - 0 : same vertex
  - ∞ : not connected
  - non-zero value : distance of edge or pointer to edge





Implementation of Graph in C++

```
/** Graph.h  (1) */
#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<Vertex>::iterator VrtxItor;
        typedef std::list<Edge>::iterator EdgeItor;
```

```
/** Graph.h  (2) */

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
```

```
/** Graph.h  (3) */

public:
        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)
                {
                        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);
                        }
                }
```

```
/** Graph.h  (4) */

            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

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];
            for (int i = 0; i<num_nodes; i++)
                  pVrtxArray[i] = NULL;
            pAdjLstArray = new EdgeList[num_vertices];
            for (int i = 0; i<num_vertices; i++)
                  pAdjLstArray[i].clear();
      }
```

```
/** Graph.h  (5) */
        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 isValidVrtxID(int vid);
private:
        string name;
        Vertex* pVrtxArray; //array of pointers of vertex
        EdgeList* pAdjLstArray; // array of adjacent lists
        int num_vertices;
};
#endif
```

```
/** Graph.cpp (1) */
#include <iostream>
#include "Graph.h"
using namespace std;

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

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]);
}
```

```
/** Graph.cpp (2) */

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);

}
```

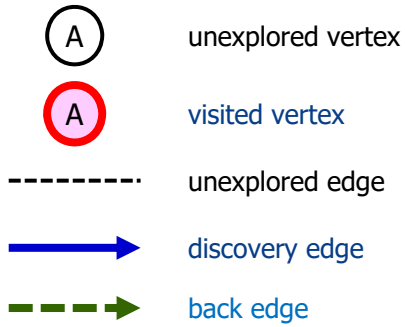
```
/** Graph.cpp (3) */

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_back(e);
			eItor++;
		}
	}
}
```

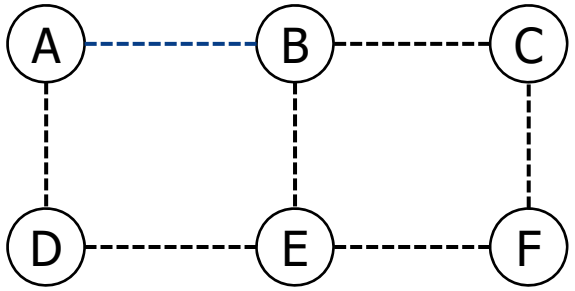
```
/** Graph.cpp (4) */

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::isValidVrtxID(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;
	}
}
```

```
/** Graph.cpp (4) */

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;
        }
}
```

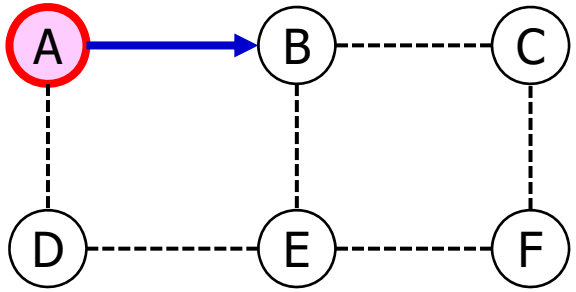
그래프의 깊이 우선 탐색 (DFS)

Depth First Search (1)

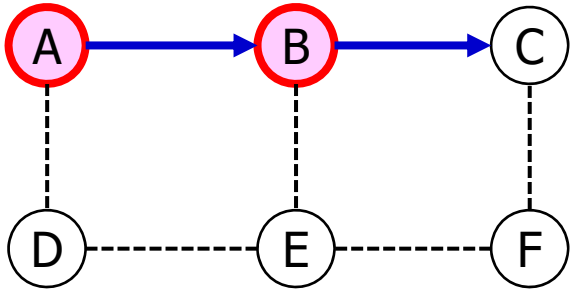




(a) Graph to be searched

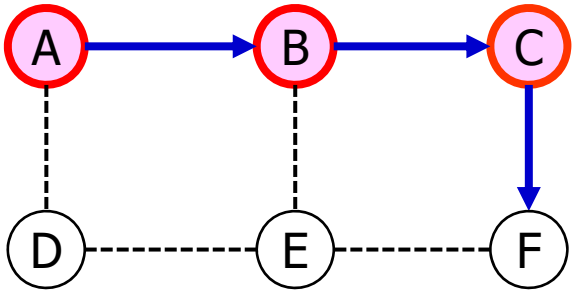


(b) Vertex A selected,
Edge (A-B) searched

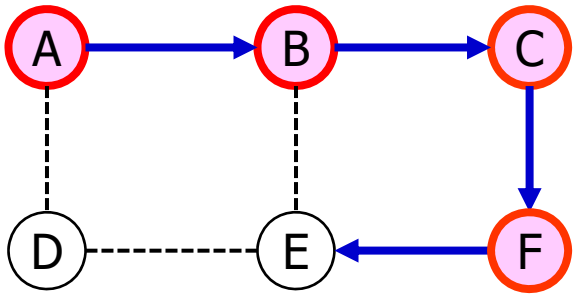


(c) Vertex B selected,
Edge (B-C) searched

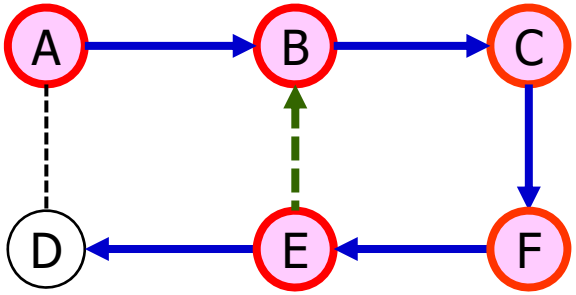
Depth First Search (2)



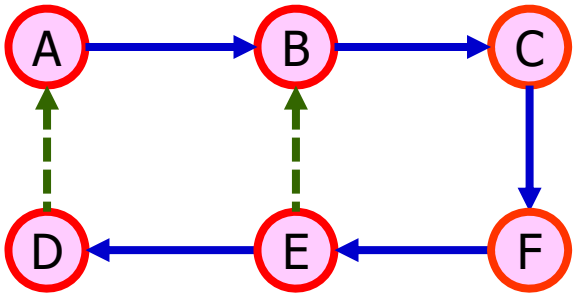
(d) Vertex C selected,
Edge (C-F) searched



(e) Vertex F selected,
Edge (F-E) searched



(f) Vertex E selected,
Edge (B-E) and Edge (E-D) searched



(g) Vertex D selected,
Edge (D-A) searched

깊이우선탐색 (Depth-First Search, DFS)

- **Depth-first search (DFS)**
  - a general technique for traversing a graph
- **A DFS traversal of a graph G**
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
  - Does not guarantee shortest path
- **DFS on a graph with n vertices and m edges takes $O(n + m)$ time**
- **DFS can be further extended to solve other graph problems**
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- **Depth-first search is to graphs what Euler tour is to binary trees**

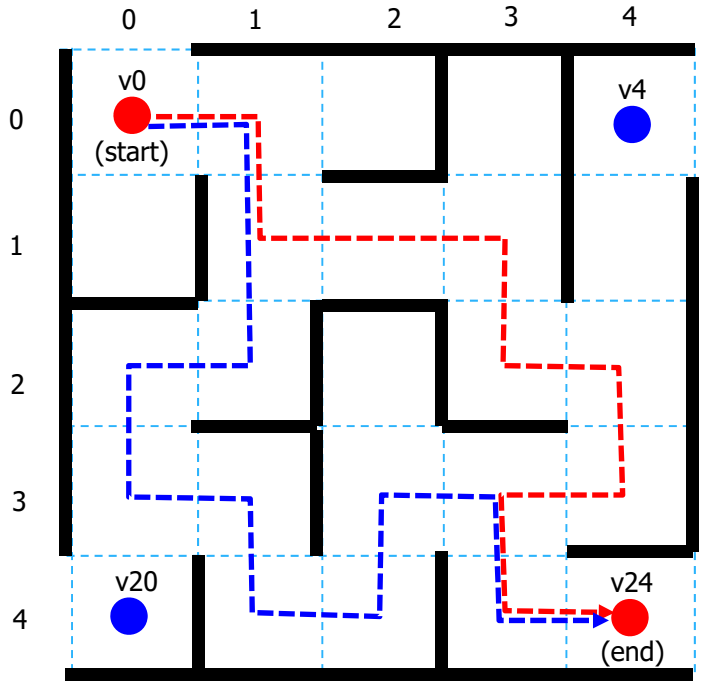
DFS Algorithm

- **The algorithm uses a mechanism for setting and getting "labels" of vertices and edges**

```
Algorithm DFS(G, v)  // depth first search from vertex v in graph G
    Input graph G and a start vertex v of G
    Output labeling of the edges of G in the connected component of v
        as discovery edges and back edges
    label v as VISITED
    for all  edge e in v.incidentEdges() do
        if  edge e is UNEXPLORED
            w ← e.opposite(v)
            if vertex w is UNEXPLORED
                label e as DISCOVERY
                recursive call DFS(G, w)
            else  // vertex w is VISITED
                label e as BACK  // vertex w was visited before
```

DFS과 미로탐색 (Maze Traversal)

- **The DFS algorithm is similar to a classic strategy for exploring a maze**
  - traverse each intersection, corner and dead end (vertex) and mark as "visited"
  - for each discovery edge, mark as "discovery" (traversed)
  - keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



Depth First Search in C++

```
/** DepthFirstSearch.h (1)*/
#include <iostream>
#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, VertexList& 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;}
```

```
/** DepthFirstSearch.h (2)*/

    // 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, VertexList& path);
    Graph& getGraph() {return graph;}
    void showConnectivity();

private:
    VrtxStatus* pVrtxStatus;
    EdgeStatus** ppEdgeStatus;
    int** ppConnectivity; // two dimensional array; table of distance[v1][v2]

};  // end of class DepthFirstSearch
#endif
```

```
/** DepthFirstSearch.cpp (1)*/
#include <iostream>
#include <list>
#include <algorithm>
#include "Graph.h"
#include "DepthFirstSearch.h"
using namespace std;

DepthFirstSearch::DepthFirstSearch(Graph& g) :graph(g)
{
        int num_nodes = 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
```

```
/** DepthFirstSearch.cpp (2)*/

        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
}
```

```
/** DepthFirstSearch.cpp (2)*/

void DepthFirstSearch::initialize()
{
        int num_nodes = 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;

}
```

```
/** DepthFirstSearch.cpp (2)*/

void DepthFirstSearch::showConnectivity(ofstream& fout)
{
        int num_nodes = 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
}
```

```
Connectivity of graph:
     |    A    B    C    D    E    F    G    H    I    J    K    L    M    N    O    P
-----+---------------------------------------------------------------------------------
   A |    0   10  +00  +00   10   15  +00  +00  +00  +00  +00  +00  +00  +00  +00  +00
   B |   10    0   10  +00  +00   10  +00  +00  +00  +00  +00  +00  +00  +00  +00  +00
   C |  +00   10    0   10  +00  +00   10  +00  +00  +00  +00  +00  +00  +00  +00  +00
   D |  +00  +00   10    0  +00  +00   15   10  +00  +00  +00  +00  +00  +00  +00  +00
   E |   10  +00  +00  +00    0   10  +00  +00   10  +00  +00  +00  +00  +00  +00  +00
   F |   15   10  +00  +00   10    0  +00  +00   15  +00  +00  +00  +00  +00  +00  +00
   G |  +00  +00   10   15  +00  +00    0  +00  +00   15   10   15  +00  +00  +00  +00
   H |  +00  +00  +00   10  +00  +00  +00    0  +00  +00  +00   10  +00  +00  +00  +00
   I |  +00  +00  +00  +00   10   15  +00  +00    0   10  +00  +00   10   15  +00  +00
   J |  +00  +00  +00  +00  +00  +00   15  +00   10    0   10  +00  +00  +00  +00  +00
   K |  +00  +00  +00  +00  +00  +00   10  +00  +00   10    0  +00  +00   15   10  +00
   L |  +00  +00  +00  +00  +00  +00   15   10  +00  +00  +00    0  +00  +00  +00   10
   M |  +00  +00  +00  +00  +00  +00  +00  +00   10  +00  +00  +00    0   10  +00  +00
   N |  +00  +00  +00  +00  +00  +00  +00  +00   15  +00   15  +00   10    0  +00  +00
   O |  +00  +00  +00  +00  +00  +00  +00  +00  +00  +00   10  +00  +00  +00    0   10
   P |  +00  +00  +00  +00  +00  +00  +00  +00  +00  +00  +00   10  +00  +00   10    0
```

```
/** DepthFirstSearch.cpp (2)*/

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
```

```
/** DepthFirstSearch.cpp (2)*/

                                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);
}
```

```
/** DepthFirstSearch.cpp (2)*/

void DepthFirstSearch::visit(Vertex& v)
{
        Graph::Vertex* pVtx;
        int numNodes = getGraph().getNumVertices();
        int vID = v.getID();

        if (isValidVrtxID(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 (isValidVrtxID(vID_1) && isValidVrtxID(vID_2))
        {
                ppEdgeStatus[vID_1][vID_2] = EDGE_VISITED;
        }
}
```

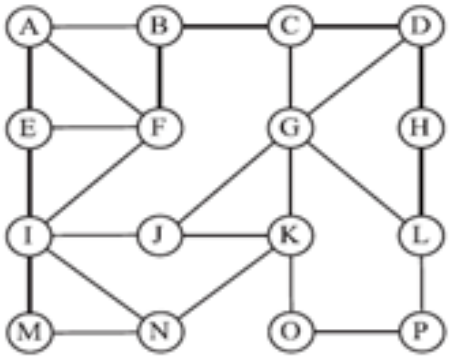
```
/** DepthFirstSearch.cpp (2)*/

void DepthFirstSearch::unvisit(Vertex& v)
{
	Graph::Vertex* pVtx;
	int numNodes = getGraph().getNumVertices();
	int vID = v.getID();

	if (isValidVrtxID(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 (isValidVrtxID(vID_1) && isValidVrtxID(vID_2))
	{
		ppEdgeStatus[vID_1][vID_2] = EDGE_UN_VISITED;
	}
}
```

```
/** DepthFirstSearch.cpp (2)*/

bool DepthFirstSearch::isVisited(Vertex& v)
{
        Graph::Vertex* pVtx;
        int numNodes = getGraph().getNumVertices();
        int vID = v.getID();

        if (isValidVrtxID(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 (isValidVrtxID(vID_1) && isValidVrtxID(vID_2))
        {
                eStat = ppEdgeStatus[vID_1][vID_2];
                if ((eStat == EDGE_VISITED) || (eStat == DISCOVERY) || (eStat == BACK))
                        return true;
                else
                        return false;
        }
        return false;
}
```

```
/** DepthFirstSearch.cpp (2)*/

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 (isValidVrtxID(vID_1) && isValidVrtxID(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 (isValidVrtxID(vID_1) && isValidVrtxID(vID_2))
	{
		eStat = ppEdgeStatus[vID_1][vID_2];
		return eStat;
	}  else {
		cout << "Edge (" << e << ") was not found from AdjacencyList" << endl;
		return EDGE_NOT_FOUND;
	}
}
```

```
/** main.cpp (1)   */

#include <iostream>
#include <fstream>
#include <string>
#include "Graph.h"
#include "DepthFirstSearch.h"
#include "BreadthFirstSearch.h"
#define NUM_NODES 16
#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);
        }
```

```
/** main.cpp (2)   */
      Vertex v[NUM_NODES] =
     {
         Vertex("A",  0), Vertex("B",  1),
         Vertex("C",  2), Vertex("D",  3),
         Vertex("E",  4), Vertex("F",  5),
         Vertex("G",  6), Vertex("H",  7),
         Vertex("I",  8), Vertex("J",  9),
         Vertex("K", 10), Vertex("L", 11),
         Vertex("M", 12), Vertex("N", 13),
         Vertex("0", 14), Vertex("P", 15)
     };
```

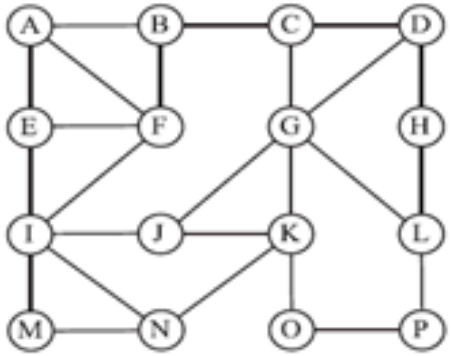



```
/** main.cpp (3)   */
  Edge edges[NUM_EDGES] =
  {
       Edge(v[0], v[1], 10),     Edge(v[1], v[0], 10),
       Edge(v[0], v[4], 10),     Edge(v[4], v[0], 10),
       Edge(v[0], v[5], 15),     Edge(v[5], v[0], 15),
       Edge(v[1], v[2], 10),     Edge(v[2], v[1], 10),
       Edge(v[1], v[5], 10),     Edge(v[5], v[1], 10),
       Edge(v[2], v[3], 10),     Edge(v[3], v[2], 10),
       Edge(v[2], v[6], 10),     Edge(v[6], v[2], 10),
       Edge(v[3], v[6], 15),     Edge(v[6], v[3], 15),
       Edge(v[3], v[7], 10),     Edge(v[7], v[3], 10),
       Edge(v[4], v[5], 10),     Edge(v[5], v[4], 10),
       Edge(v[4], v[8], 10),     Edge(v[8], v[4], 10),
       Edge(v[5], v[8], 15),     Edge(v[8], v[5], 15),
       Edge(v[6], v[9], 15),     Edge(v[9], v[6], 15),
       Edge(v[6], v[10], 10),    Edge(v[10], v[6], 10),
       Edge(v[6], v[11], 15),    Edge(v[11], v[6], 15),
       Edge(v[7], v[11], 10),    Edge(v[11], v[7], 10),
       Edge(v[8], v[9], 10),     Edge(v[9], v[8], 10),
       Edge(v[8], v[12], 10),    Edge(v[12], v[8], 10),
       Edge(v[8], v[13], 15),    Edge(v[13], v[8], 15),
       Edge(v[9], v[10], 10),    Edge(v[10], v[9], 10),
       Edge(v[10], v[13], 15),   Edge(v[13], v[10], 15),
       Edge(v[10], v[14], 10),   Edge(v[14], v[10], 10),
       Edge(v[11], v[15], 10),   Edge(v[15], v[11], 10),
       Edge(v[12], v[13], 10),   Edge(v[13], v[12], 10),
       Edge(v[14], v[15], 10),   Edge(v[15], v[14], 10)
  };
```

```
/** main.cpp (3)   */

    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);
```

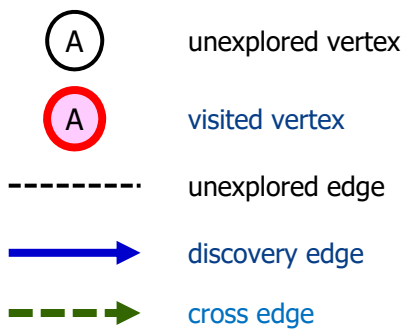
```
/** main.cpp (4)   */
    for (EdgeItor p = egLst.begin(); p != egLst.end(); ++p)
    {
        count++;
        cout << *p << ", ";
        if (count % 5 == 0)
            cout << endl;
    }
    cout << endl;

    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[15], path);
    cout << "₩nPath (" << v[0] << " => " << v[15] << ") : ";
    for (VrtxItor vItor = path.begin(); vItor != path.end(); ++vItor)
        cout << *vItor << "    ";
    cout << endl;

    dfsGraph.findPath(v[15], v[0], path);
    cout << "₩nPath (" << v[15] << " => " << v[0] << ") : ";
    for (VrtxItor vItor = path.begin(); vItor != path.end(); ++vItor)
        cout << *vItor << "    ";
    cout << endl;
}
```

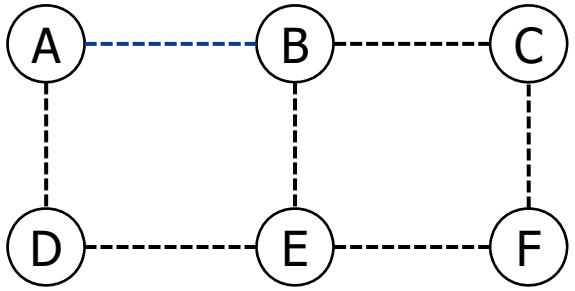
```
Inserting vertices ..
Inserted vertices: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P,
Inserting edges ..
GRAPH_SQUARE_16_NODES with 16 vertices has following connectivity :
 vertex (  A) :  Edge( A,  B,   10)  Edge( A,  E,   10)  Edge( A,  F,   15)
 vertex (  B) :  Edge( B,  A,   10)  Edge( B,  C,   10)  Edge( B,  F,   10)
 vertex (  C) :  Edge( C,  B,   10)  Edge( C,  D,   10)  Edge( C,  G,   10)
 vertex (  D) :  Edge( D,  C,   10)  Edge( D,  G,   15)  Edge( D,  H,   10)
 vertex (  E) :  Edge( E,  A,   10)  Edge( E,  F,   10)  Edge( E,  I,   10)
 vertex (  F) :  Edge( F,  A,   15)  Edge( F,  B,   10)  Edge( F,  E,   10)  Edge( F,  I,   15)
 vertex (  G) :  Edge( G,  C,   10)  Edge( G,  D,   15)  Edge( G,  J,   15)  Edge( G,  K,   10)  Edge( G,  L,   15)
 vertex (  H) :  Edge( H,  D,   10)  Edge( H,  L,   10)
 vertex (  I) :  Edge( I,  E,   10)  Edge( I,  F,   15)  Edge( I,  J,   10)  Edge( I,  M,   10)  Edge( I,  N,   15)
 vertex (  J) :  Edge( J,  G,   15)  Edge( J,  I,   10)  Edge( J,  K,   10)
 vertex (  K) :  Edge( K,  G,   10)  Edge( K,  J,   10)  Edge( K,  N,   15)  Edge( K,  O,   10)
 vertex (  L) :  Edge( L,  G,   15)  Edge( L,  H,   10)  Edge( L,  P,   10)
 vertex (  M) :  Edge( M,  I,   10)  Edge( M,  N,   10)
 vertex (  N) :  Edge( N,  I,   15)  Edge( N,  K,   15)  Edge( N,  M,   10)
 vertex (  O) :  Edge( O,  K,   10)  Edge( O,  P,   10)
 vertex (  P) :  Edge( P,  L,   10)  Edge( P,  O,   10)

Testing dfsGraph...
Connectivity of graph:
     I    A    B    C    D    E    F    G    H    I    J    K    L    M    N    O    P
-----+-------------------------------------------------------------------------------
   A I    0   10  +oo  +oo   10   15  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo
   B I   10    0   10  +oo  +oo   10  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo
   C I  +oo   10    0   10  +oo  +oo   10  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo
   D I  +oo  +oo   10    0  +oo  +oo   15   10  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo
   E I   10  +oo  +oo  +oo    0   10  +oo  +oo   10  +oo  +oo  +oo  +oo  +oo  +oo  +oo
   F I   15   10  +oo  +oo   10    0  +oo  +oo   15  +oo  +oo  +oo  +oo  +oo  +oo  +oo
   G I  +oo  +oo   10   15  +oo  +oo    0  +oo  +oo   15   10   15  +oo  +oo  +oo  +oo
   H I  +oo  +oo  +oo   10  +oo  +oo  +oo    0  +oo  +oo  +oo   10  +oo  +oo  +oo  +oo
   I I  +oo  +oo  +oo  +oo   10   15  +oo  +oo    0   10  +oo  +oo   10   15  +oo  +oo
   J I  +oo  +oo  +oo  +oo  +oo  +oo   15  +oo   10    0   10  +oo  +oo  +oo  +oo  +oo
   K I  +oo  +oo  +oo  +oo  +oo  +oo   10  +oo  +oo   10    0  +oo  +oo   15   10  +oo
   L I  +oo  +oo  +oo  +oo  +oo  +oo   15   10  +oo  +oo  +oo    0  +oo  +oo  +oo   10
   M I  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo   10  +oo  +oo  +oo    0   10  +oo  +oo
   N I  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo   15  +oo   15  +oo   10    0  +oo  +oo
   O I  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo   10  +oo  +oo  +oo    0   10
   P I  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo   10  +oo  +oo   10    0

Path found by DFS (A => F) : A    B    C    D    G    J    I    E    F

Path found by DFS (F => A) : F    A
```



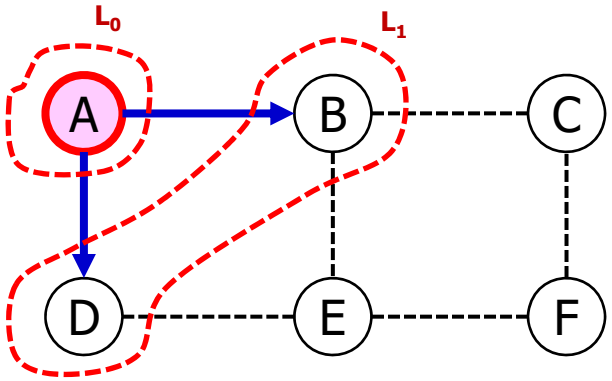
그래프의 넓이 우선 탐색 (BFS)

그래프의 넓이 우선 탐색 (Breadth First Search) (1)

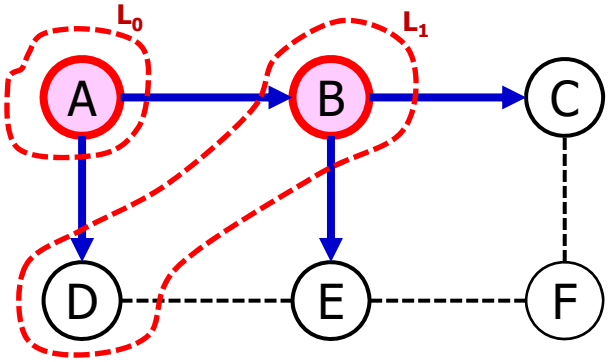




(a) Graph to be searched

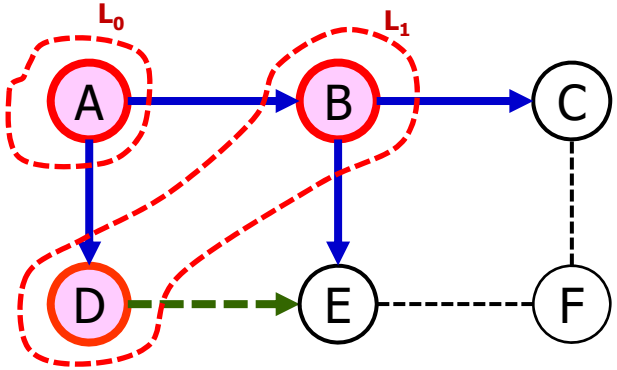


(b) Vertex A selected,
Edges (A-B), (A_D) searched

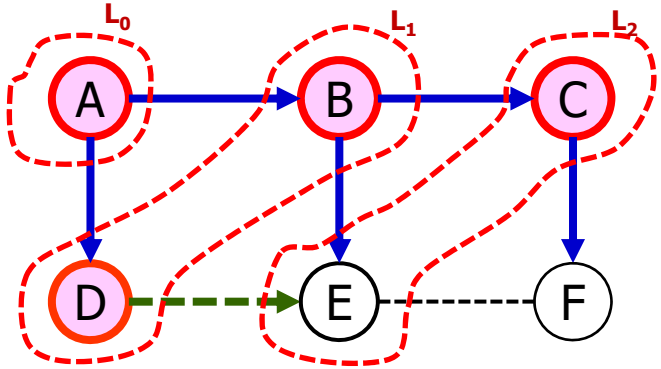


(c) Vertex B selected,
Edges (B-C), (B-E) searched

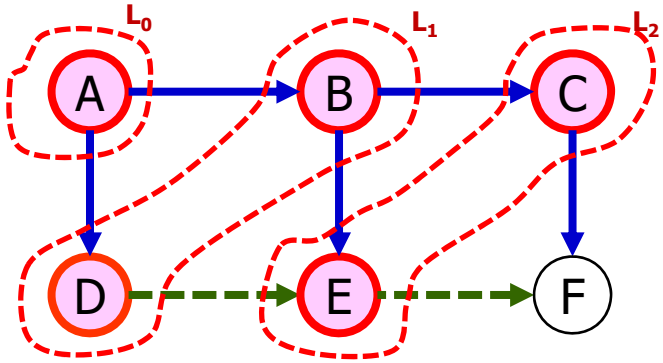
그래프의 넓이 우선 탐색 (Breadth First Search) (2)



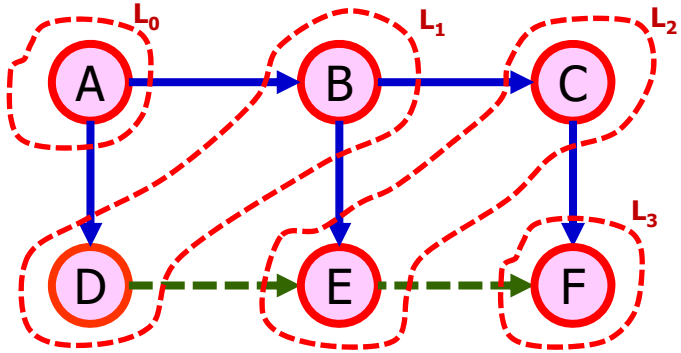
(d) Vertex D selected,
Edges (D-E) searched



(e) Vertex C selected,
Edges (C-F) searched



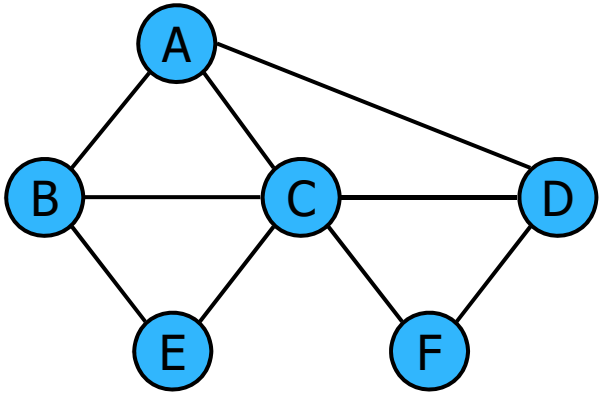
(f) Vertex E selected,
Edges (B-C), (B-E), (D-E) searched

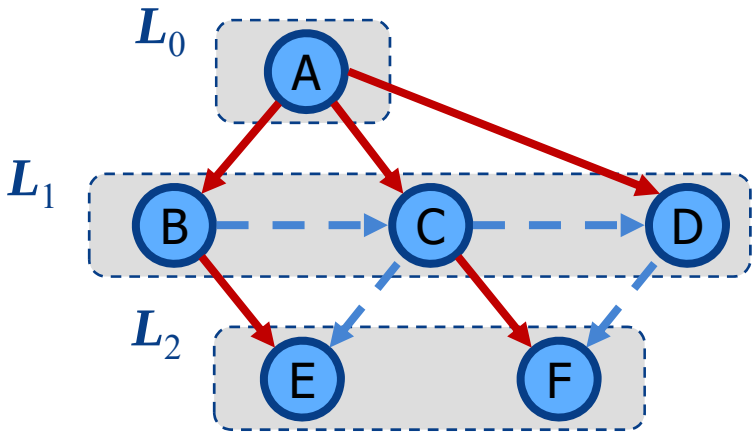


(g) Vertex F selected

Properties

- **Notation**
  - G_s: connected component of start vertex s
- **$BFS(G, s)$ visits all the vertices and edges of G_s**
- **The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s**
- **For each vertex v in L_i**
  - The path of T_s from s to v has i edges
  - Every path from s to v in L_i has at least i edges
- **Each edge is labeled**
  - DISCOVERY: when an unvisited vertex is connected
  - CROSS: when an already visited vertex is connected





Applications

- **Using the template method pattern, we can specialize the BFS traversal of a graph *G* to solve the following problems in $O(n + m)$ time**
  - Compute the connected components of ***G***
  - Compute a spanning tree/forest of ***G***
  - Find a simple cycle in ***G***, or report that ***G*** is a tree/forest
  - Given two vertices of ***G***, find a path in ***G*** between them with the minimum number of edges (shortest path), or report that no such path exists

C++ Implementation of Breath First Search

```
/** BFS_Dijkstra.h (1)*/

#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(); }
```

```
/** BFS_Dijkstra.h (2)*/

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
```

```
/** BFS_Dijkstra.cpp  (1) */

#include <iostream>
#include <iomanip>
#include <list>
#include <algorithm>
#include "Graph.h"
#include "BFS_Dijkstra.h"
using namespace std;

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);
}
```

```
/** BFS_Dijkstra.cpp  (2) */

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;
        }
}
```

```
/** BFS_Dijkstra.cpp  (3) */

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;
```

```
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
```

```
/** BFS_Dijkstra.cpp  (4) */

        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;
}
```

```
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
```

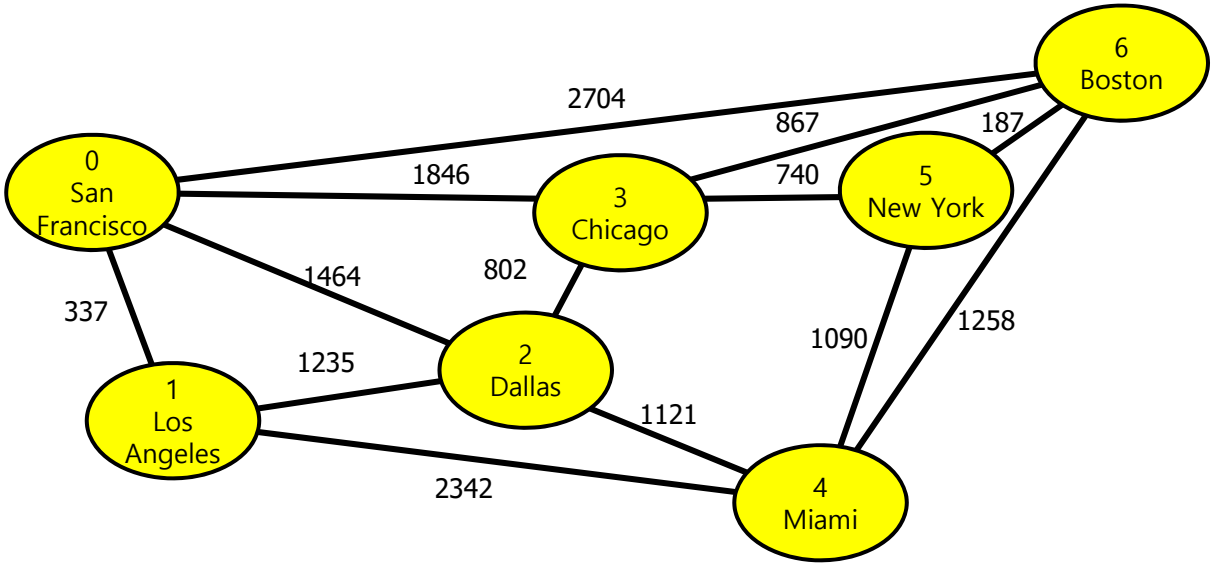
그래프의 최단거리 경로 (Shortest Path) 탐색
- Dijkstra Algorithm

Shortest Paths

- **Weighted graphs**
  - Shortest path problem
  - Shortest path properties
  - Shortest paths in DAGs (Directed Acyclic Graphs)
- **Dijkstra's algorithm**
  - Algorithm
  - Edge relaxation
- **The Bellman-Ford algorithm**
- **All-pairs shortest paths**

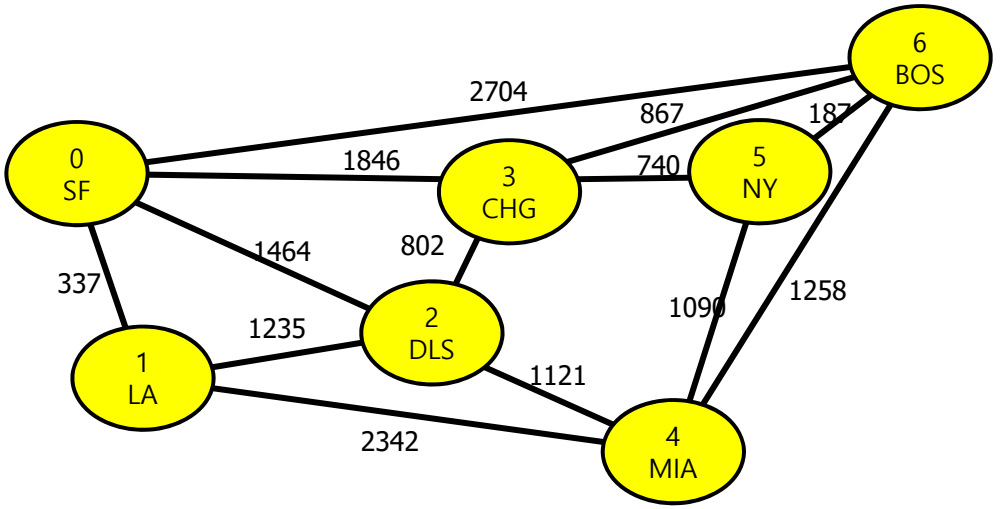
Weighted Graphs

- **In a weighted graph, each edge has an associated numerical value, called the weight of the edge**
- **Edge weights may represent distances, time, costs, etc.**
- **Example:**
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Finding Shortest Path in Weighted Graph

- **Given a weighted graph and two vertices *u* and *v*, we want to find a path of minimum total weight between *u* and *v*.**
  - Length of a path is the sum of the weights of its edges.
- **A subpath of a shortest path is itself a shortest path**
- **There is a tree of shortest paths from a start vertex to all the other vertices**
- **Applications**
  - Internet packet routing
  - Flight reservations
  - Driving directions



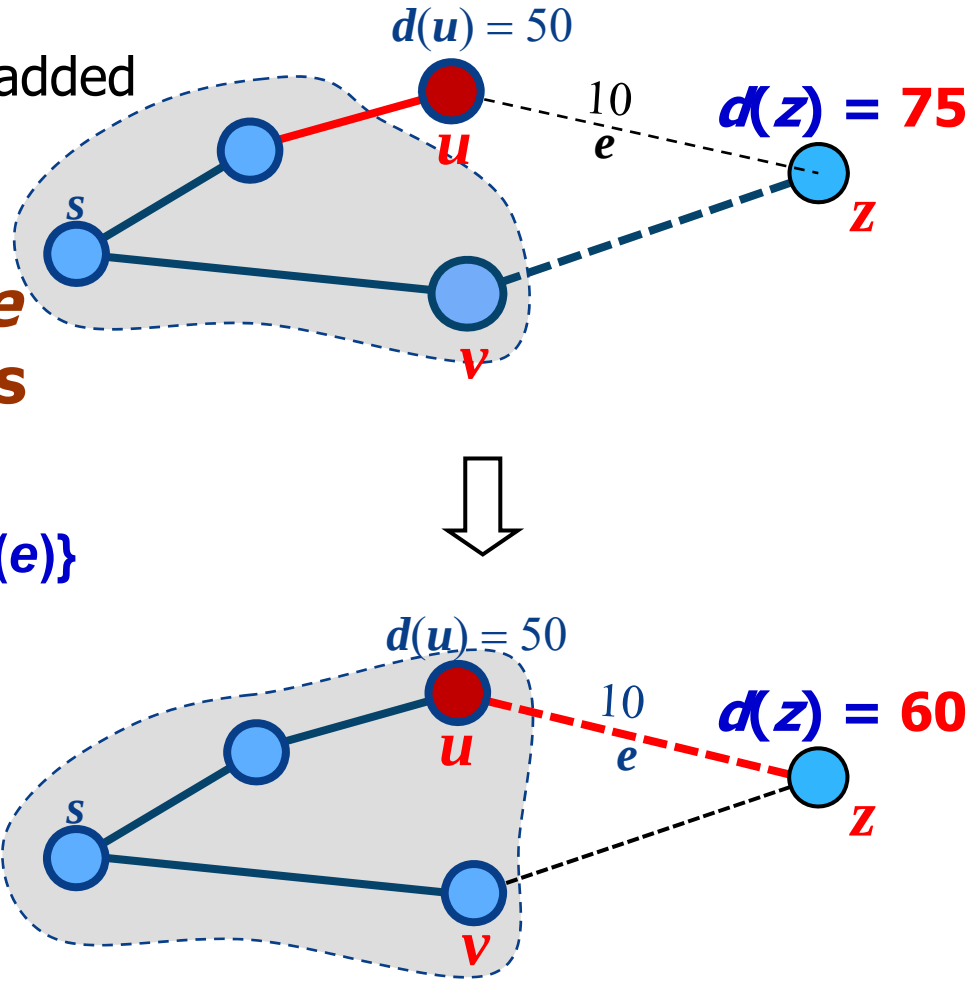
Dijkstra's Algorithm

- The distance of a vertex *v* from a vertex *start* is the length of a shortest path between *start* and *v*
- Dijkstra's algorithm computes the distances of all the vertices from a given *start* vertex
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- We grow a "cloud" of vertices, beginning with *start* and eventually covering all the vertices
- We store with each vertex *v* a label *d*(*v*) representing the distance of *v* from *start* in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex *u* outside the cloud with the smallest distance label, *d*(*u*)
  - We update the labels of the vertices adjacent to *u*

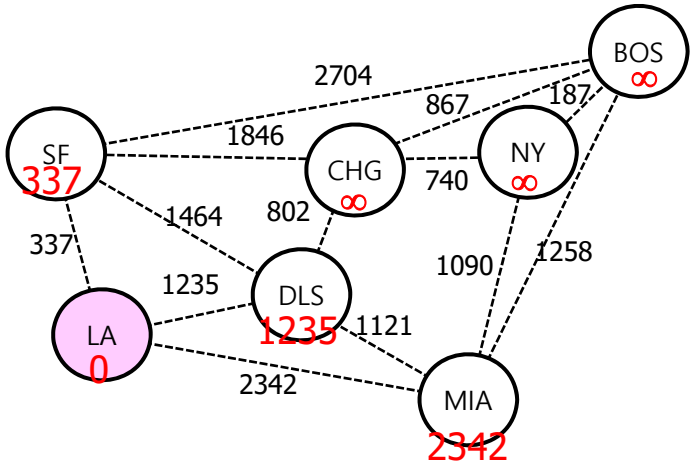
Edge Relaxation

- **Consider an edge *e* = (*u*, *z*) such that**
  - *u* is the vertex most recently added to the cloud
  - *z* is not in the cloud
- **The *relaxation* of edge *e* updates distance *d*(*z*) as follows:**

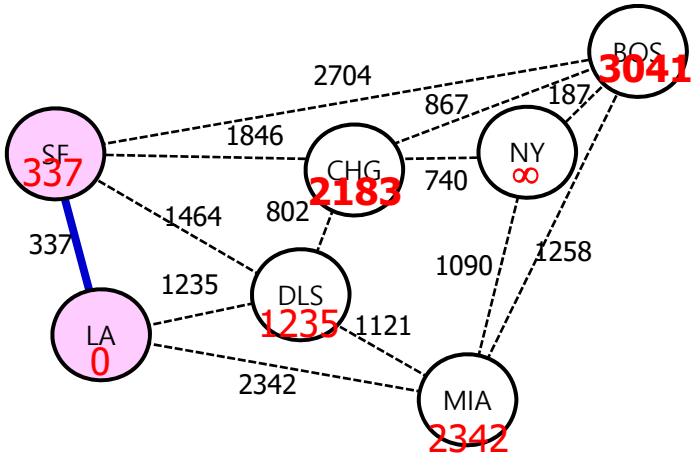
$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$



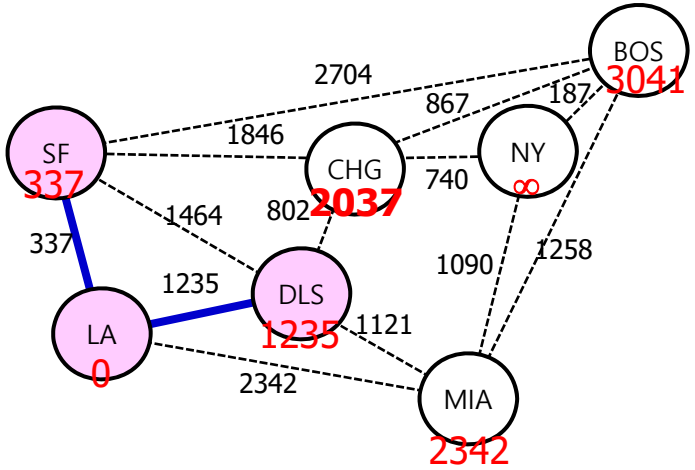
An execution of Dijkstra's algorithm on a weighted graph.



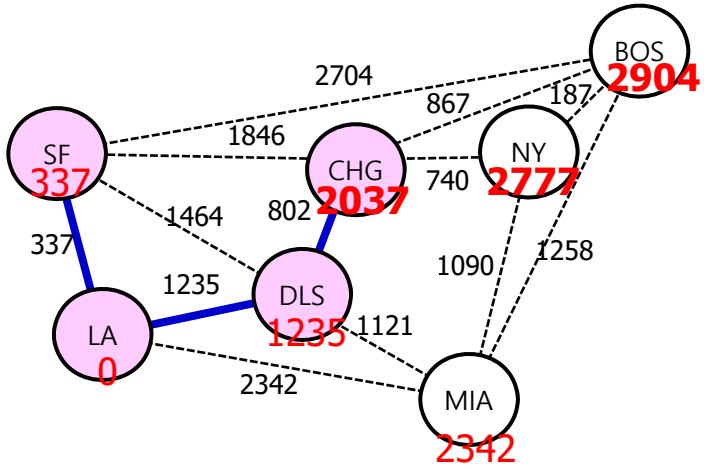
(a) round 0



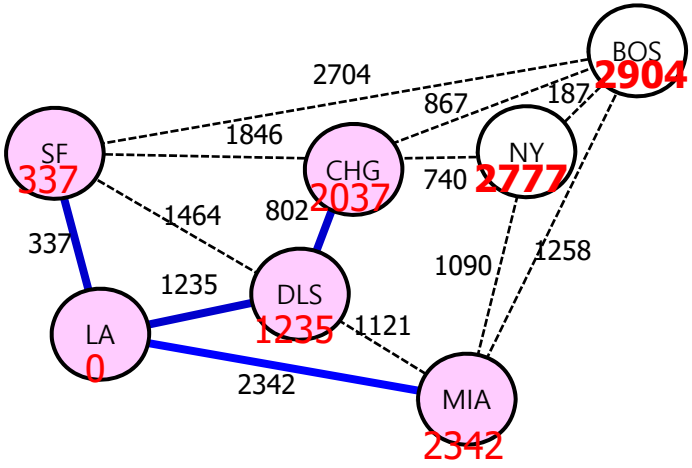
(b) round 1



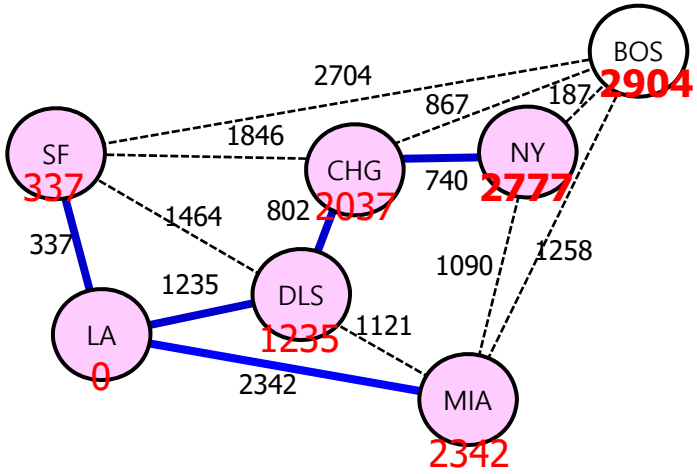
(c) round 2



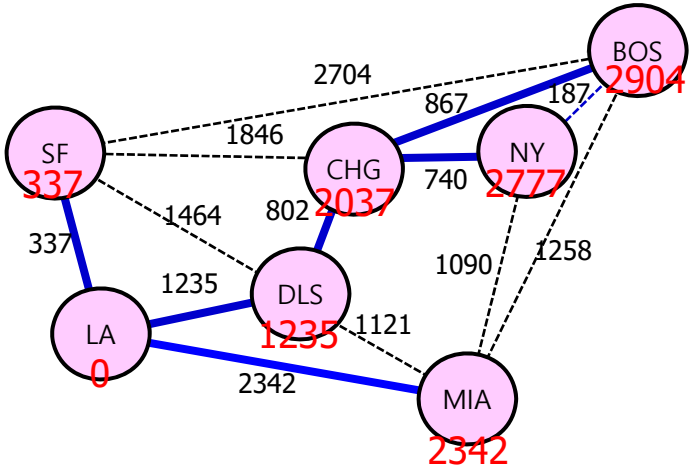
(d) round 3



(e) round 4



(f) round 5



(g) round 6

```
/** BFS_Dijkstra.cpp  (5) */

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];
```

```
/** BFS_Dijkstra.cpp  (6) */

        // initialize L(n) = w(start, n);
        for (int i = 0; i< num_nodes; i++)
        {
                pLeastCost[i] = ppDistMtrx[start_vID][i];
                pPrev[i] = start_vID;
                pBFS_Process_Stat[i] = NOT_SELECTED;
        }
        pBFS_Process_Stat[start_vID] = SELECTED;
        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;
```

```
/** BFS_Dijkstra.cpp  (7) */

        while (num_selected < num_nodes)
        {
                round++;
                fout << "round [" << setw(2) << round << "] |";
                minID = -1;
                minCost = PLUS_INF;
                for (int i = 0; i<num_nodes; i++) // find a node with LeastCost  {
                        if ((pLeastCost[i] < minCost) && (pBFS_Process_Stat[i] != SELECTED)) {
                                minID = i;
                                minCost = pLeastCost[i];
                        }
                }
                if (minID == -1) {
                        fout << "Error in Dijkstra() -- found not connected vertex !!" << endl;
                        break;
                } else {
                        pBFS_Process_Stat[minID] = SELECTED;
                        num_selected++;

                        if (minID == target_vID)
                        {
                                fout << endl << "reached to the target node ("
                                   << pVrtxArray[minID].getName() << ") at Least Cost = " << minCost << endl;
                                vID = minID;
                                do {
                                        vrtx = pVrtxArray[vID];
                                        path.push_front(vrtx);
                                        vID = pPrev[vID];
                                } while (vID != start_vID);
                                vrtx = pVrtxArray[vID];
                                path.push_front(vrtx); // start node
                                break;
                        }
                }
```

```
/** BFS_Dijkstra.cpp  (7) */

            /* 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()
```

```
/** main.cpp  (1) */

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

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)
	};
```

```
/** main.cpp  (2) */

	Graph::Edge edges[NUM_EDGES] =
	  // 26 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;
```

```
/** main.cpp  (3) */

	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]);
	}
```

```
/** main.cpp  (3) */

        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);

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

        fout << "\nTesting Breadth First Search with Dijkstra Algorithm" << endl;
        bfsGraph.initDistMtrx();
        bfsGraph.fprintDistMtrx(fout);
```

```
/** main.cpp  (3) */

        VrtxList path;

        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 << "  ->   ";
        fout << endl;

        fout.close();
}
```

◆ Execution results (1)

```
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

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
```

◆ Execution results (2)

```
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 LA to BOS : LA -> DLS -> CHG -> BOS
```

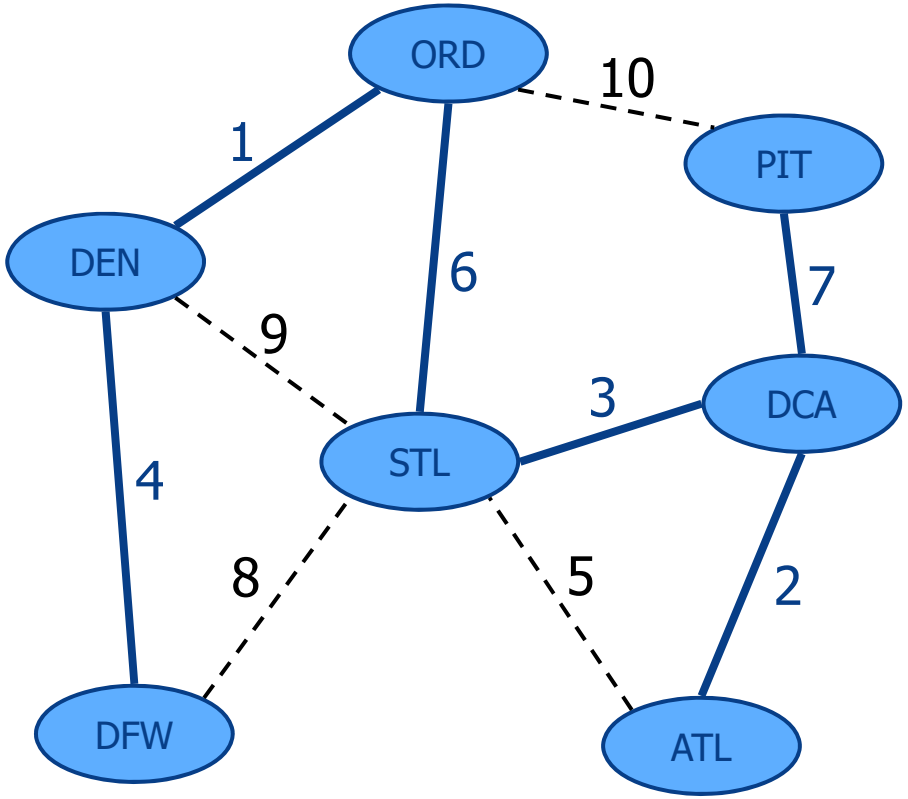
DFS, BFS, Dijkstra

	DFS	BFS	Dijkstra
Check existence of path or cycle (bi-connected elements)	✓		
Spanning tree/forest, connected components, paths	✓	✓	
Directional Edge, Digraph			✓
Weighted Edge			✓
Shortest paths	not guaranteed	path with smallest number of edges	path with shortest accumulated distance

Minimum Spanning Tree
- Prim-Jarnik's Algorithm

Minimum Spanning Tree

- **Spanning subgraph**
  - Subgraph of a graph ***G*** containing all the vertices of ***G***
- **Spanning tree**
  - Spanning subgraph that is itself a (cycle-free) tree
- **Minimum spanning tree (MST)**
  - Spanning tree of a weighted graph with **minimum total edge weight**
- **Applications**
  - Communications networks
  - Transportation networks



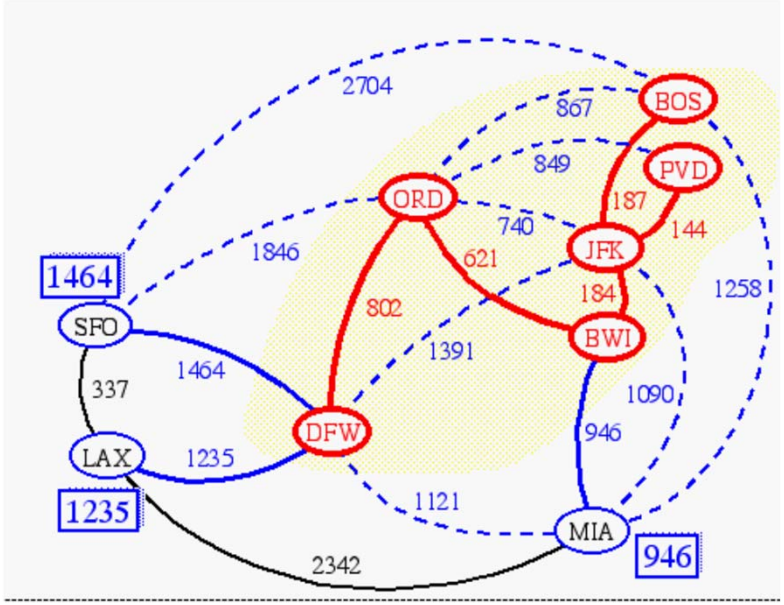
Prim-Jarnik's Algorithm

Finding MST with Prim-Janik's Algorithm

- Similar to Dijkstra's algorithm (for a connected graph)
- We pick an arbitrary vertex ***s*** and we grow the MST as a cloud of vertices, starting from ***s***
- For each vertex ***v***, **maintain** an array ***dist[v]*** = the smallest weight of an edge connecting ***v*** to any vertex in the cloud

At each step:

- We add to the cloud a vertex ***u*** outside the cloud with the smallest distance label
- We update the labels of the vertices adjacent to ***u***



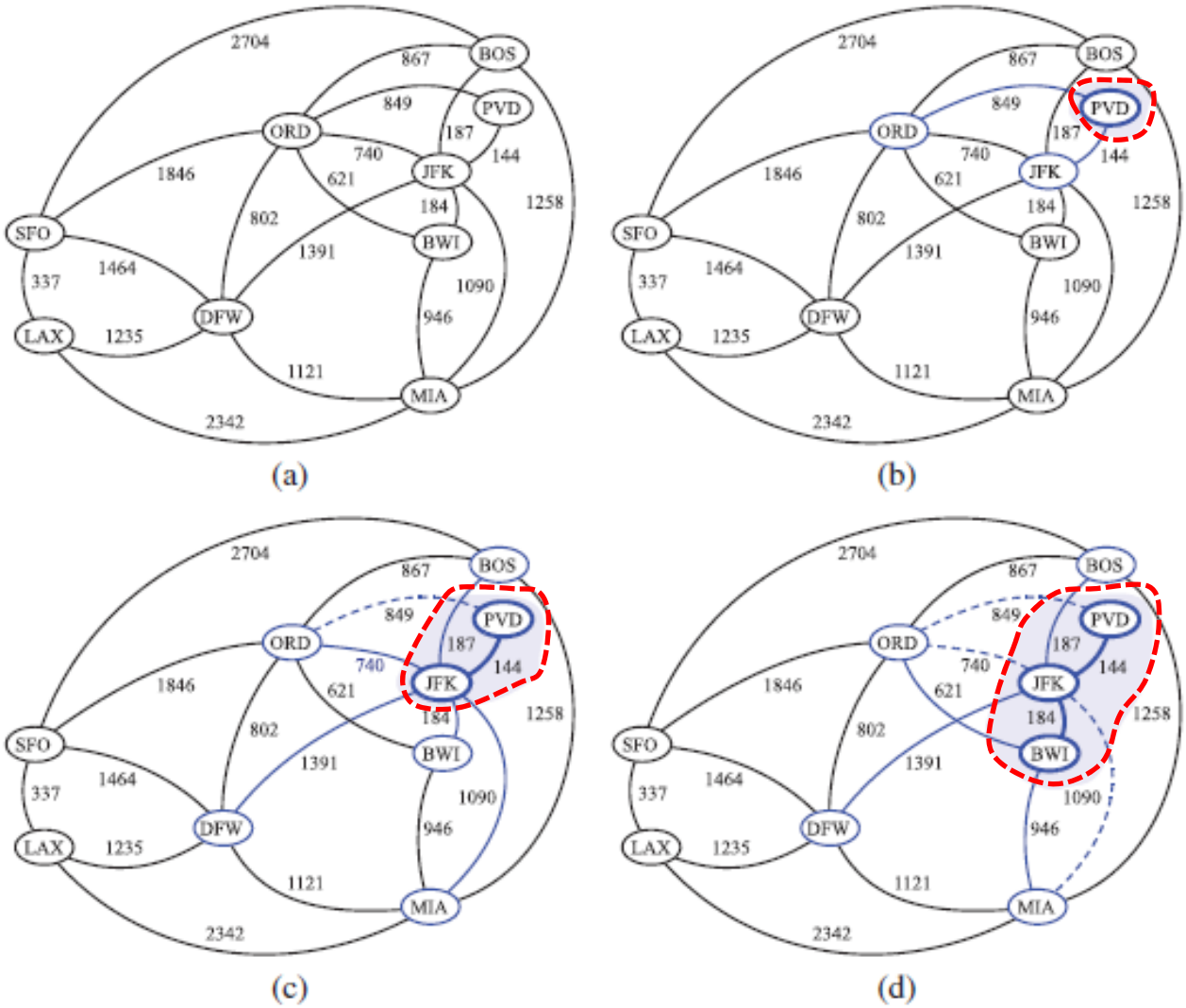
Prim-Jarnik's Algorithm (cont.)

Internal Data Structures

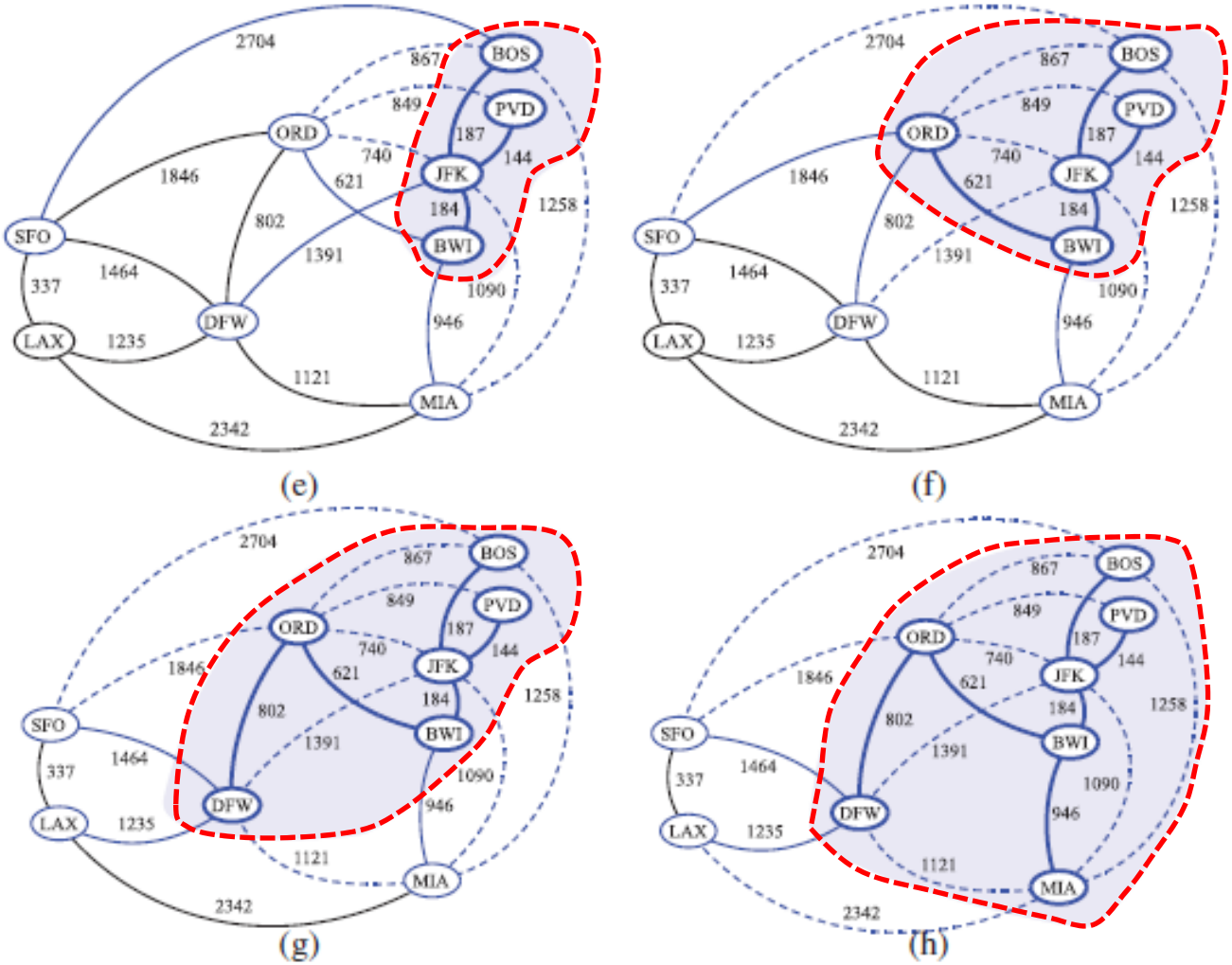
- VrtxArray[] : array of vertex
- EdgeList : Adjacent List Array
- DistMtrx[][] : distance table
- s: start vertex
- u: vertex
- VertexStatus[u] : status of vertex u (e.g., SELECTED, NOT_SELECTED)
- ParentEdge[u] : parent edge that connects u to the currently selected cloud
- Dist[u] : distance of the parent edge
- SelectedEdgeList : list of selected edges

```
Algorithm PrimJarnik(G):
  Input: A weighted connected graph G with n vertices
      and m edges
  Output: A minimum spanning tree
          (selected edge lists) for G

  Pick any vertex s of G
  Dist[s]←0
  VertexStatus[s] = SELECTED
  SelectedEdgeList.clear()
  for each vertex u ≠ s do
      Dist[u]←+∞
      Mark vertex u as NOT_SELECTED
  while not all vertices are selected do
      Select a vertex u with minimum Dist[u]
        and its parent edge pe
      Mark vertex u as SELECTED
      Include parent edge pe into the SelectedEdgeList
      for each vertex z adjacent to u
          such that z is NOT_SELECTED
          do
            perform the edge_relaxation procedure
            on edge (u, z), update Dist[z], ParentEdge[z],
  return SelectedEdgeList
```

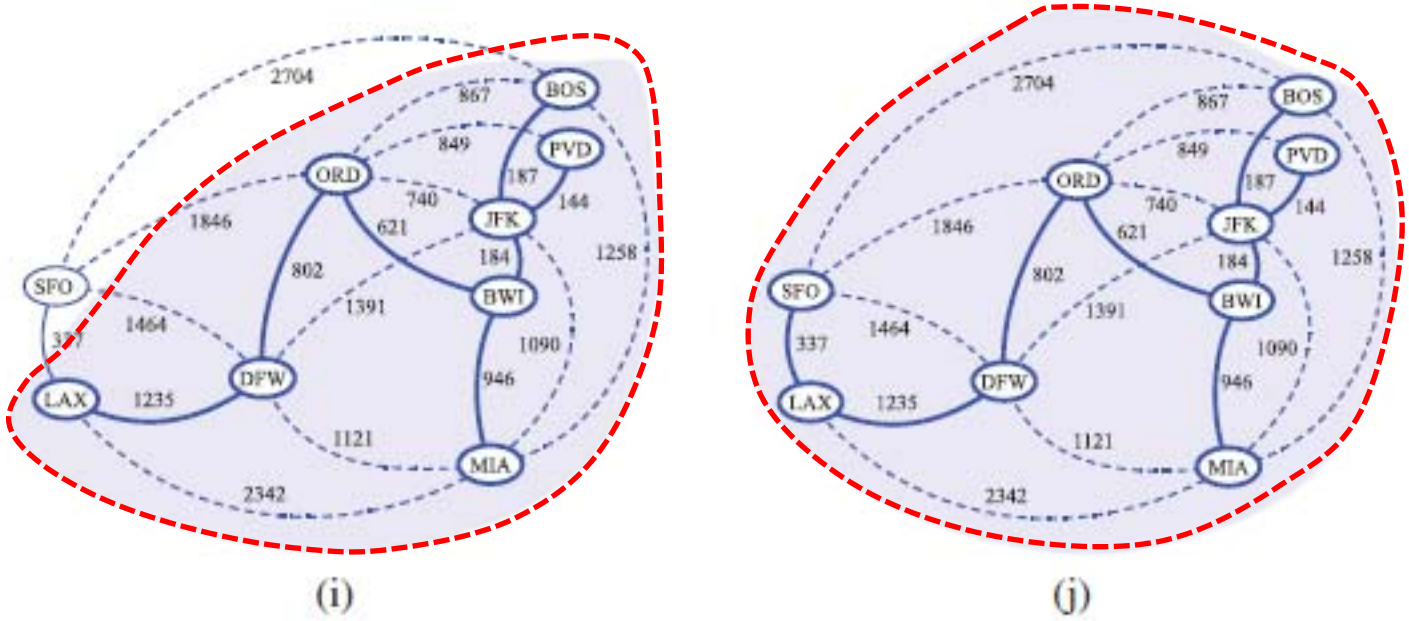
◆ Example of Minimum Spanning Tree with Prim-Jarnik's Algorithm (1)



◆ Example of Minimum Spanning Tree with Prim-Jarnik's Algorithm (2)



◆ Example of Minimum Spanning Tree with Prim-Jarnik's Algorithm (3)



C++ implementation of Prim-Jarnik's Algorithm

```
/** MinimumSpanningTree::PrimJarnikMST()  (1)  */

void MinimumSpanningTree::PrimJarnikMST()
{
    int num_nodes, 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;
    Graph::Edge *pParentEdge, edge, min_edge; // edge that connects this node to the cloud

    std::list<Graph::Edge> selectedEdgeLst;
    std::list<Graph::Edge>::iterator edgeItor;

    num_nodes = graph.getNumVertices();
    pVrtxArray = graph.getpVrtxArray();
    pAdjLstArray = graph.getpAdjLstArray();

    initDistMtrx();
    ppDistMtrx = getppDistMtrx();
```

```
/** MinimumSpanningTree::PrimJarnikMST()  (2)  */

    pDist = new int[num_nodes];
    pVrtxStatus = new VertexStatus[num_nodes];
    pEdge = new Graph::Edge[num_nodes];
    for (int i=0; i<num_nodes; i++) {
        pDist[i] = PLUS_INF;
        pVrtxStatus[i] = NOT_SELECTED;
        pParentEdge[i] = Graph::Edge(NULL, NULL, PLUS_INF);
    }

    srand(time(0));
    start = rand() % num_nodes;  // randomly select start node

    cout << "Start node : " << start << endl;
    pDist[start] = 0;

    selectedEdgeLst.clear();
```

```
/** MinimumSpanningTree::PrimJarnikMST()  (3)  */

    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;
```

```
/** MinimumSpanningTree::PrimJarnikMST()  (4)  */
        // edge relaxation
        EdgeItor pe = pAdjLstArray[min_id].begin();
        while (pe != pAdjLstArray[min_id].end())
        {
            end_ID = ((*pe).getVertex_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
```

```
/** MinimumSpanningTree::PrimJarnikMST()  (5)  */


    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;
}
```

◆ Example of execution result

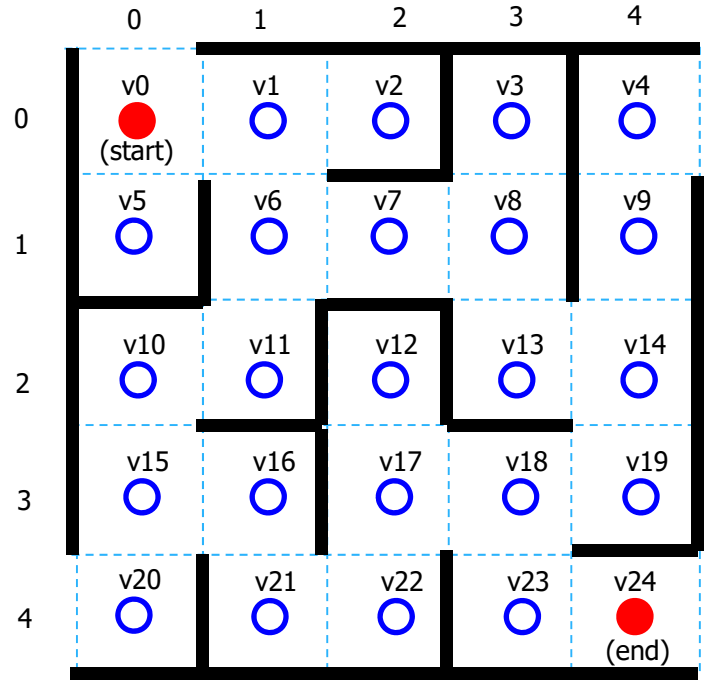
```
Testing Prim-JarnikMST()
Start node : 12
Dist after round [ 0] :  +oo  +oo  +oo  +oo  +oo  861  +oo  +oo  +oo  +oo  +oo  285    0  297  +oo  +oo  +oo  845  +oo  +oo
Dist after round [ 1] :  +oo  +oo  +oo  +oo  +oo  861  +oo  +oo  454  +oo  393  285    0  297  +oo  +oo  394  845  +oo  +oo
Dist after round [ 2] :  +oo  +oo  +oo  +oo  +oo  861  +oo  409  454  +oo  393  285    0  297  286  +oo  394  845  +oo  +oo
Dist after round [ 3] :  +oo  +oo  +oo  +oo  +oo  861  +oo  409  454  +oo  393  285    0  297  286  +oo  394  534  640  834
Dist after round [ 4] :  +oo  +oo  +oo  +oo  +oo  861  +oo  409  454  352  393  285    0  297  286  861  394  534  640  834
Dist after round [ 5] :  +oo  +oo  +oo  +oo  +oo  861  +oo  409  246  352  393  285    0  297  286  861  394  534  640  834
Dist after round [ 6] :  +oo  +oo  +oo  +oo  +oo  780 1067  409  246  352  393  285    0  297  286  861  394  534  640  834
Dist after round [ 7] :  +oo  +oo  +oo  +oo  +oo  780 1067  409  246  352  393  285    0  297  286  661  394  534  640  834
Dist after round [ 8] :  +oo  +oo  +oo  +oo  611  780 1067  409  246  352  393  285    0  297  286  661  394  534  640  834
Dist after round [ 9] :  +oo  +oo  +oo  +oo  611  780 1067  409  246  352  393  285    0  297  286  661  394  534  237  834
Dist after round [10] :  +oo  +oo  +oo  +oo  611  780 1067  409  246  352  393  285    0  297  286  661  394  534  237  211
Dist after round [11] :  +oo  +oo  +oo  +oo  611  780 1067  409  246  352  393  285    0  297  286  661  394  534  237  211
Dist after round [12] : 1144  +oo  +oo  657  611  389 1067  409  246  352  393  285    0  297  286  661  394  534  237  211
Dist after round [13] : 1144  +oo  +oo  521  611  389  816  409  246  352  393  285    0  297  286  661  394  534  237  211
Dist after round [14] :  828  745  688  521  611  389  816  409  246  352  393  285    0  297  286  661  394  534  237  211
Dist after round [15] :  828  745  688  521  611  389  816  409  246  352  393  285    0  297  286  661  394  534  237  211
Dist after round [16] :  828  380  688  521  611  389  381  409  246  352  393  285    0  297  286  661  394  534  237  211
Dist after round [17] :  820  380  688  521  611  389  381  409  246  352  393  285    0  297  286  661  394  534  237  211
Dist after round [18] :  820  380  688  521  611  389  381  409  246  352  393  285    0  297  286  661  394  534  237  211
Dist after round [19] :  820  380  688  521  611  389  381  409  246  352  393  285    0  297  286  661  394  534  237  211

End of finding Minimum Spanning Tree by Prim-Jarnik's Algorithm; selectedEdgeLst.size = 19
Selected edges:
Edge(12, 11, d(285)), Edge(12, 13, d(297)), Edge(13, 14, d(286)), Edge(11, 10, d(393)), Edge(10, 9, d(352)),
Edge(9, 8, d(246)), Edge(11, 16, d(394)), Edge(13, 7, d(409)), Edge(14, 17, d(534)), Edge(17, 18, d(237)),
Edge(18, 19, d(211)), Edge(7, 4, d(611)), Edge(4, 5, d(389)), Edge(5, 3, d(521)), Edge(16, 15, d(661)),
Edge(3, 2, d(688)), Edge(2, 1, d(380)), Edge(2, 6, d(381)), Edge(1, 0, d(820)),
```

Homework 13

Homework 13-1

13.1 그래프 깊이 우선 탐색과 미로 탐색 (Maze Traversal)

- Graph representation for Maze
  - vertex: cross point
  - edge: distance between the cross points
- Find the path from v0 (start) to v24 (end)

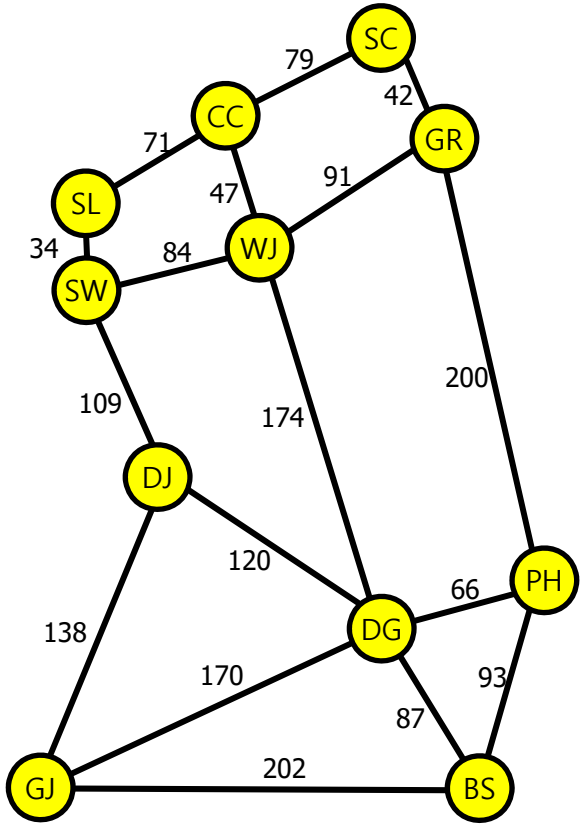


v0 ~ v24: cross points
distance(v0, v1): 1
distance(v0, v5): 1
distance(v2, v3): +∞

Homework 13-2

13.2 Dijkstra 알고리즘을 사용한 최단거리 경로 탐색

- 다음과 같은 그래프가 주어지고, 출발지(start)와 목적지(destination)이 주어질 때, 최단 거리의 경로를 탐색하는 알고리즘을 구현하라.



Homework 13-3

13.3 Minimum Spanning Tree 산출

- 다음과 같은 그래프에서 모든 노드를 연결하는 minimum spanning tree를 구성하여 출력하라.