O-O Programming & Data Structure

Lab. 13 (보충설명) Dijkstra's Shortest Path Algorithm

정보통신공학과

교수 김 영 탁

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

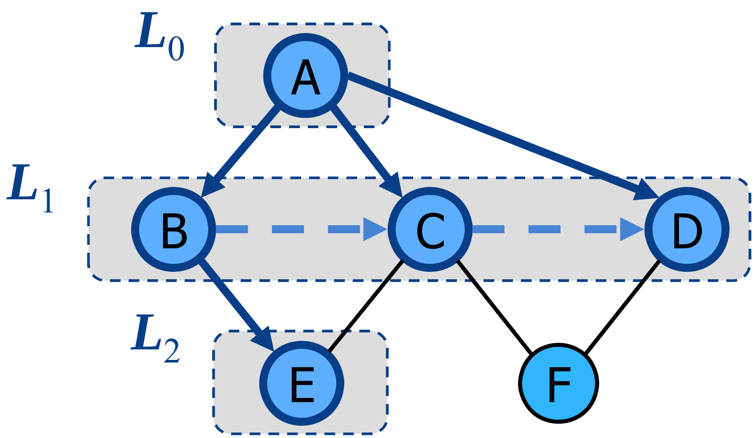
Breadth-First Search

◆ Breadth-first search

- Breadth-first search (BFS) is a general technique for traversing a graph

◆ A BFS 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



Breadth-First Search

- **BFS on a graph with n vertices and m edges takes $O(n + m)$ time**
- **BFS can be further extended to solve other graph problems**
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

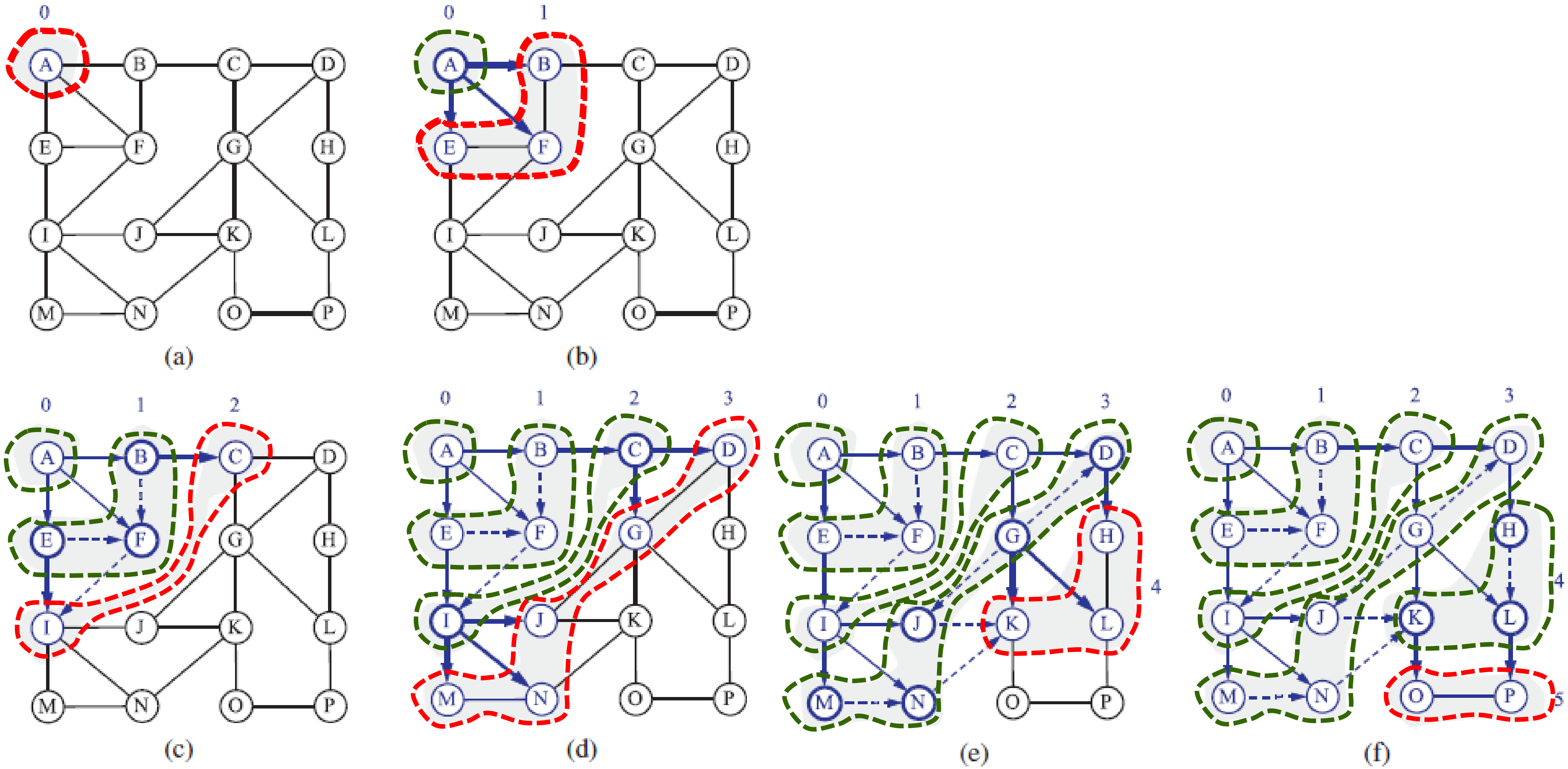
BFS Algorithm

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

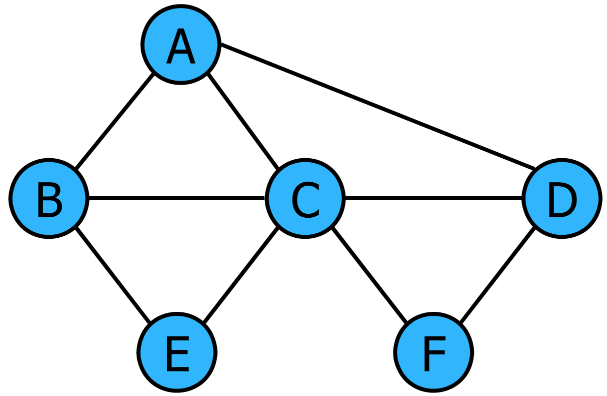
```
Algorithm BFS(G)
  Input graph G
  Output labeling of the edges
     and partition of the
     vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      BFS(G, v)
```

```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.insertLast(s)
  setLabel(s, VISITED)
  i ← 0
  while (!Li.isEmpty())
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            Li+1.insertLast(w)
          else // w is in same or next level
            setLabel(e, CROSS)
    i ← i +1
```

Example of BFS



Properties



Notation

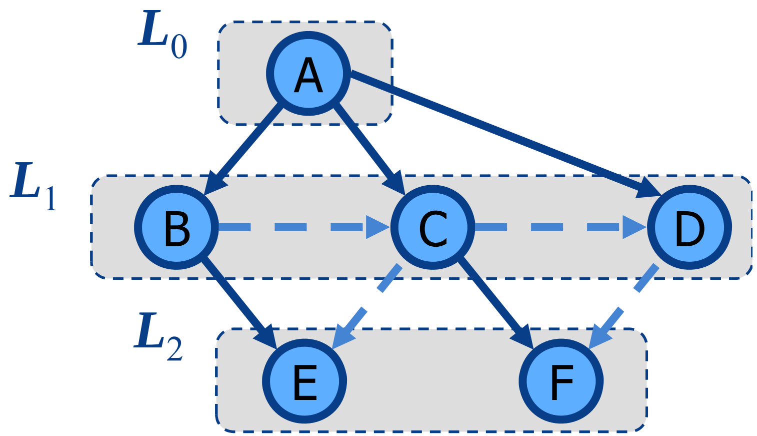
G_s: connected component of s

Property 1

***BFS*(*G*, *s*) visits all the vertices and edges of G_s**

Property 2

The discovery edges labeled by *BFS*(*G*, *s*) form a spanning tree T_s of G_s



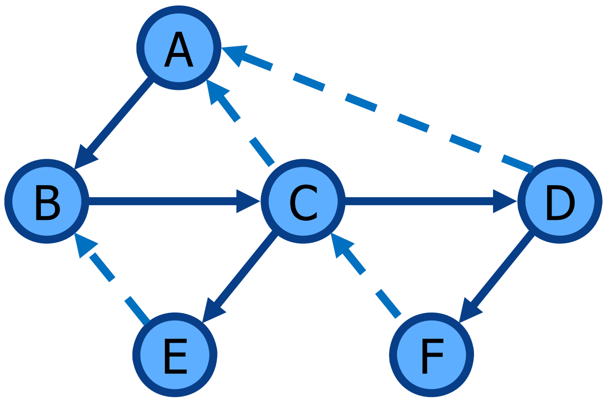
Property 3

For each vertex *v* in L_i

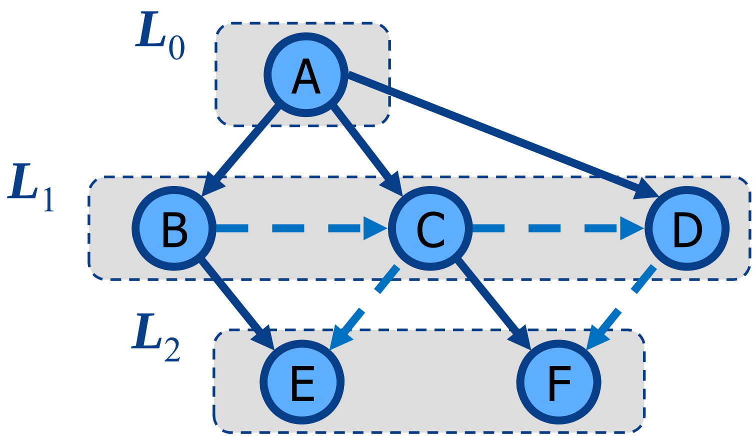
- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



DFS

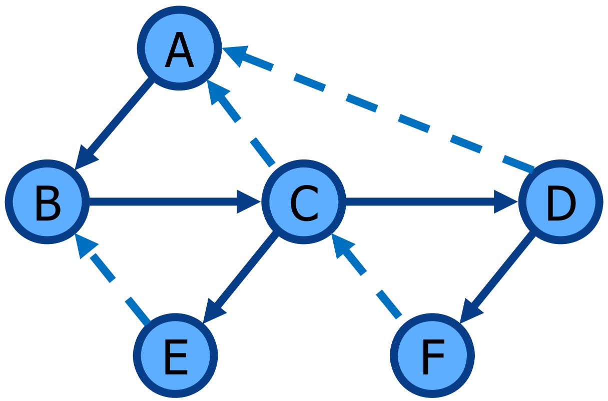


BFS

DFS vs. BFS (cont.)

Back edge (*v,w*)

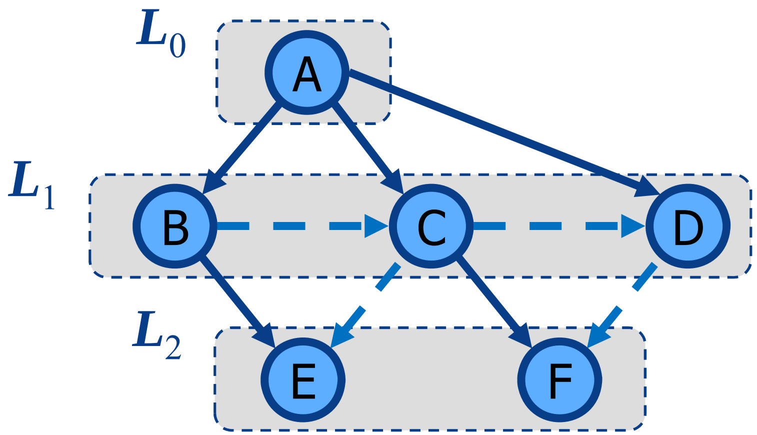
- ***w*** is an ancestor of ***v*** in the tree of discovery edges



DFS

Cross edge (*v,w*)

- ***w*** is in the same level as ***v*** or in the next level in the tree of discovery edges



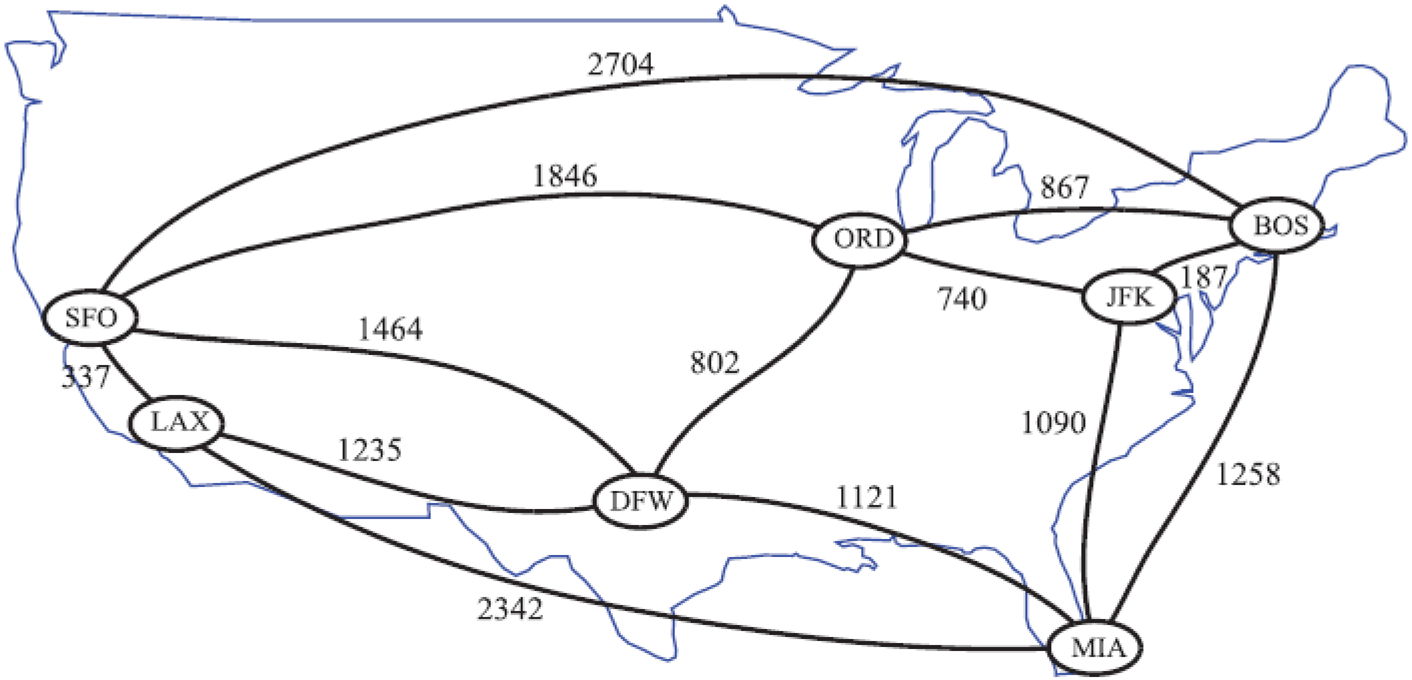
BFS

Shortest Paths

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

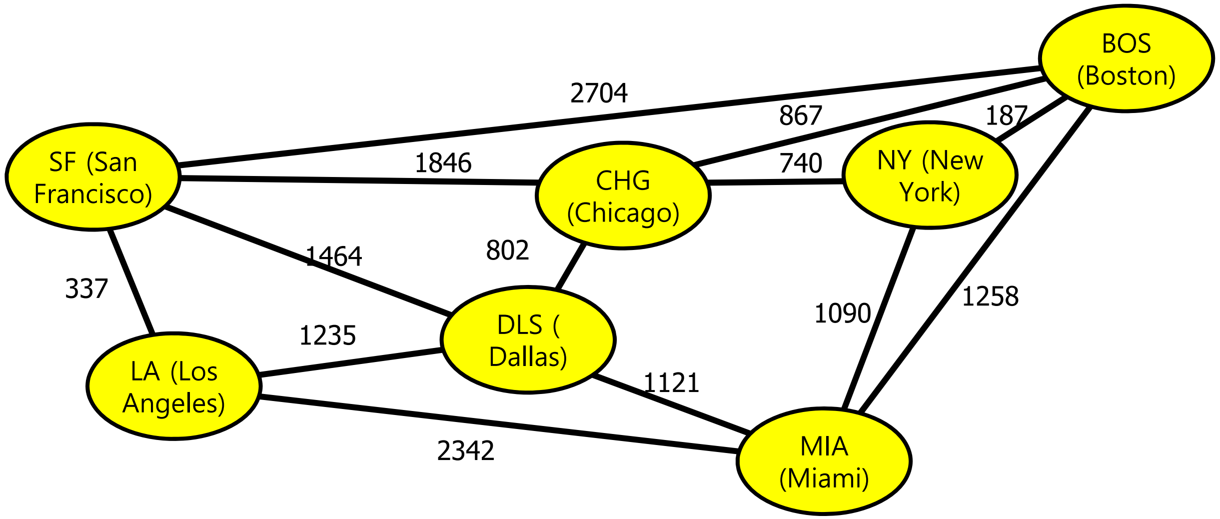
Weighted Graphs

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



Shortest Path Problem

- **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.
- **Example:**
  - Shortest path between Providence (Rhode Island, USA) and Honolulu (Hawaii, USA)
- **Applications**
  - Internet packet routing
  - Flight reservations
  - Driving directions



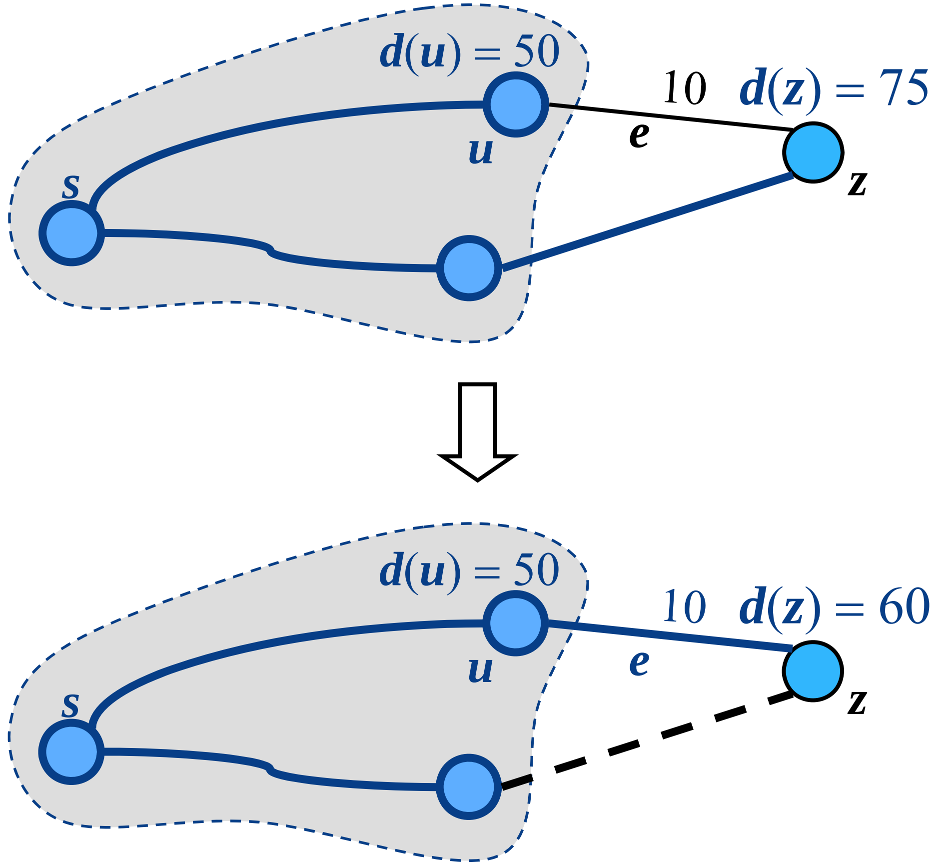
Dijkstra's Algorithm

- **The distance of a vertex *v* from a vertex *s* is the length of a shortest path between *s* and *v***
- **Dijkstra's algorithm computes the distances of all the vertices from a given start vertex *s***
- **Assumptions:**
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- **We grow a "cloud" of vertices, beginning with *s* and eventually covering all the vertices**
- **We store with each vertex *v* a label *d*(*v*) representing the distance of *v* from *s* 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:**

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



class BreadthFirstSearch

```
/** 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

	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
```

```
/** DijkstraShortestPath()  (1) */
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];
```

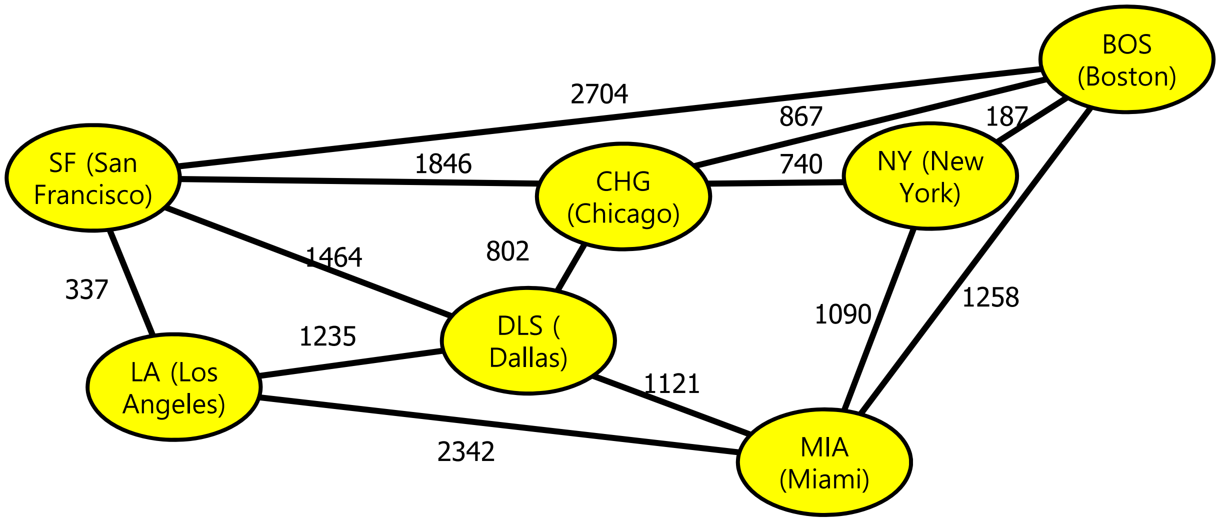
```
/** DijkstraShortestPath()  (2) */

	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;
```

```
/** DijkstraShortestPath()  (3) */

        while (num_selected < num_nodes)
        {
                round++;
                fout << "round [" << setw(2) << round << "] |";
                // find current node with LeastCost
                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) {
                        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;
                        }
                }
```

```
/** DijkstraShortestPath()  (4) */

                /* 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];
                fout << endl;

        }  // end while()

}
```

Sample Graph

◆ Simplified USA Topology



```
/** GRAPH_USA_7_NODES (1) */
#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_USA_20_NODES (2) */

Edge edges[NUM_EDGES] =
{
        Edge(v[0], v[1], 820), Edge(v[1], v[0], 820),
        Edge(v[0], v[3], 828), Edge(v[3], v[0], 828),
        Edge(v[0], v[4], 1144), Edge(v[4], v[0], 1144),
        Edge(v[1], v[2], 380), Edge(v[2], v[1], 380),
        Edge(v[1], v[3], 745), Edge(v[3], v[1], 745),
        Edge(v[2], v[3], 688), Edge(v[3], v[2], 688),
        Edge(v[2], v[6], 381), Edge(v[6], v[2], 381),
        Edge(v[3], v[4], 657), Edge(v[4], v[3], 657),
        Edge(v[3], v[5], 521), Edge(v[5], v[3], 521),
        Edge(v[4], v[5], 389), Edge(v[5], v[4], 389),
        Edge(v[4], v[7], 611), Edge(v[7], v[4], 611),
        Edge(v[5], v[6], 816), Edge(v[6], v[5], 816),
        Edge(v[5], v[7], 920), Edge(v[7], v[5], 920),
        Edge(v[5], v[8], 780), Edge(v[8], v[5], 780),
        Edge(v[5], v[12], 861), Edge(v[12], v[5], 861),
        Edge(v[6], v[8], 1067), Edge(v[8], v[6], 1067),
        Edge(v[7], v[13], 409), Edge(v[13], v[7], 409),
        Edge(v[8], v[9], 246), Edge(v[9], v[8], 246),
        Edge(v[8], v[11], 454), Edge(v[11], v[8], 454),
        Edge(v[9], v[10], 352), Edge(v[10], v[9], 352),
        Edge(v[10], v[11], 393), Edge(v[11], v[10], 393),
        Edge(v[10], v[15], 861), Edge(v[15], v[10], 861),
        Edge(v[10], v[16], 473), Edge(v[16], v[10], 473),
```

```
/** GRAPH_USA_20_NODES (3) */

        Edge(v[11], v[12], 285), Edge(v[12], v[11], 285),
        Edge(v[11], v[16], 394), Edge(v[16], v[11], 394),
        Edge(v[12], v[13], 297), Edge(v[13], v[12], 297),
        Edge(v[12], v[17], 845), Edge(v[17], v[12], 845),
        Edge(v[13], v[14], 286), Edge(v[14], v[13], 286),
        Edge(v[14], v[17], 534), Edge(v[17], v[14], 534),
        Edge(v[14], v[18], 640), Edge(v[18], v[14], 640),
        Edge(v[14], v[19], 834), Edge(v[19], v[14], 834),
        Edge(v[15], v[16], 661), Edge(v[16], v[15], 661),
        Edge(v[16], v[17], 632), Edge(v[17], v[16], 632),
        Edge(v[17], v[18], 237), Edge(v[18], v[17], 237),
        Edge(v[18], v[19], 211), Edge(v[19], v[18], 211)
};

int test_start = 0;
int test_end = 15;
```

```
/** main.cpp  (1) */
. . . .  // include necessary header files and definitions

void main()
{
      . . . . .  // include necessary elements
      Graph simpleGraph("GRAPH_SIMPLE_USA_7_NODES", NUM_NODES);

      fout << "Inserting vertices .." << endl;
      . . . .  // insert vertices

      VrtxList vtxLst;
      simpleGraph.vertices(vtxLst);

      fout << "Inserted vertices: ";
       . . .   // printout inserted vertices

      fout << "Inserting edges .." << endl;
      . . . . // insert edges

      fout << "Inserted edges: " << endl;
      . . . . // printout inserted edges

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

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

```
    /* ==========================================*/
    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();
}
```

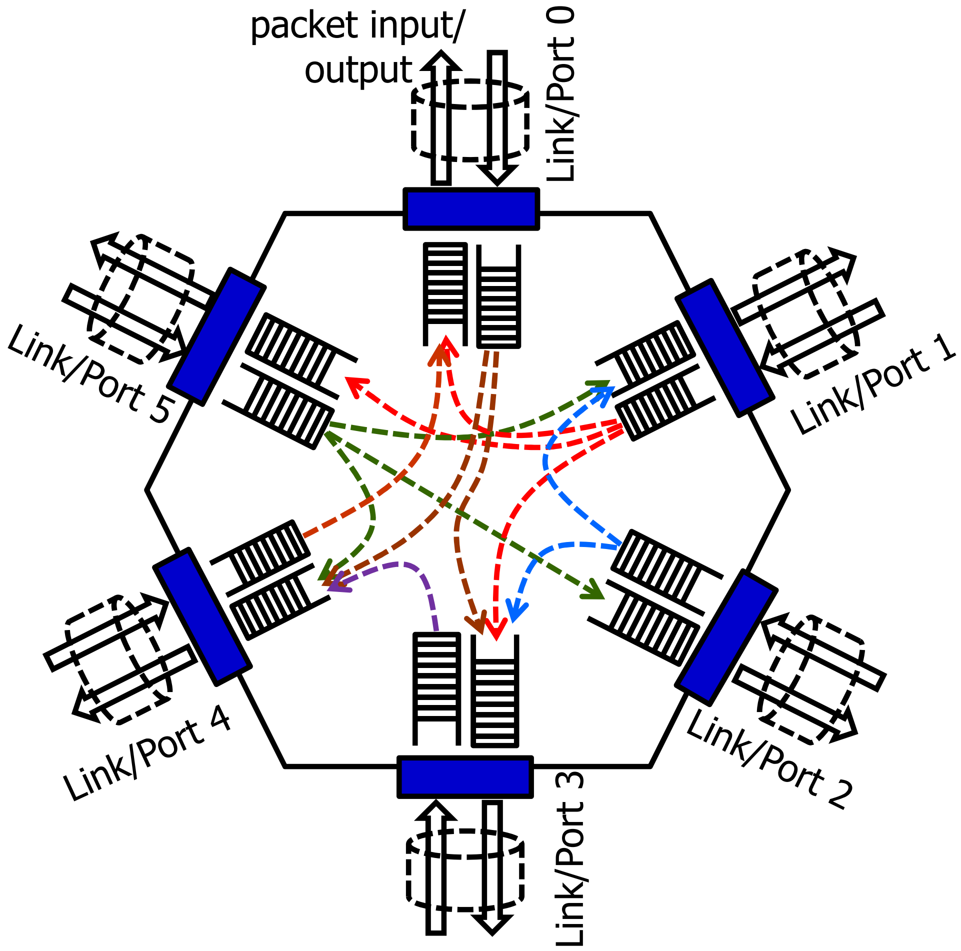
◆ Execution results (1)

```
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 with Dijkstra Algorithm

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

◆ 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
```

Internet Protocol (IP) Router Model



Packet Forwarding Table

Dest IP address	Next hop	Tx Link/ Port No	. . . .

Implementation of Router

◆ Basic Operation of Router

- obtain the network topology data: nodes, links
- calculate **shortest paths tree** to all other nodes from this node
- configure **forwarding table** (next hop for a target address)

- (optional for simulator) generate packets to all nodes (including itself)

- check the input queue for packet arrivals
- obtain the target address of the arrived packet
- if a packet to this node is arrived:
  print the route, and delete
- if a transit packet (to other node) is arrived:
  push the router ID into the packet to record route
  forward the packet to "next_hop" router

Packet Forwarding Table

◆ Initialization of Packet Forwarding Table

- for each destination address, the next hop (from this node) is found from the shortest paths tree (calculated by Dijkstra's algorithm)

◆ Packet Forwarding at each Router

- for each arrived packet, obtain the target address
- if a packet to this node is arrived:
  print the route, and delete
- if a transit packet (to other node) is arrived:
  push the router ID into the packet to record route
  forward the packet to "next_hop" router

Oral Test 13 (1)

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

<Key Points>

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

13.2 2 x 3격자형 그래프에 대한 깊이 우선 탐색 (Depth First Search) 알고리즘을 실행하기 위하여 구현되는 dfsTraversal() 멤버 함수를 pseudocode로 표현하고, 상세 동작 절차를 주어진2 x 3격자형 그래프의 노드를 사용하여 설명하라.

<Key Points>

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

Oral Test 13

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

<Key Points>

(1) Breadth First Search (BFS) 알고리즘의 동작 절차

(2) Breadth First Search (BFS)의 한 종류인 Dijkstra 알고리즘을 사용한 최단거리 경로 탐색 (shortest path search) 기능의 동작 절차

(3) Depth First Search (DFS)와 Breadth First Search (BFS) 알고리즘의 기능적 차이점과 주요 응용 분야에 대한 비교 설명

13.4 그래프의 자료구조의 주요 응용 분야에 대하여 상세하게 설명하라.

<Key Points>

(1) 자동차/스마트폰의 네비게이션

(2) 인터넷의 패킷 라우팅

(3) 전자회로 부품 배치

(4) 데이터 베이스의 연관 정보 검색을 통한 상관관계 분석

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