Problem Solving Techniques 문제해결

Jinkyu Lee

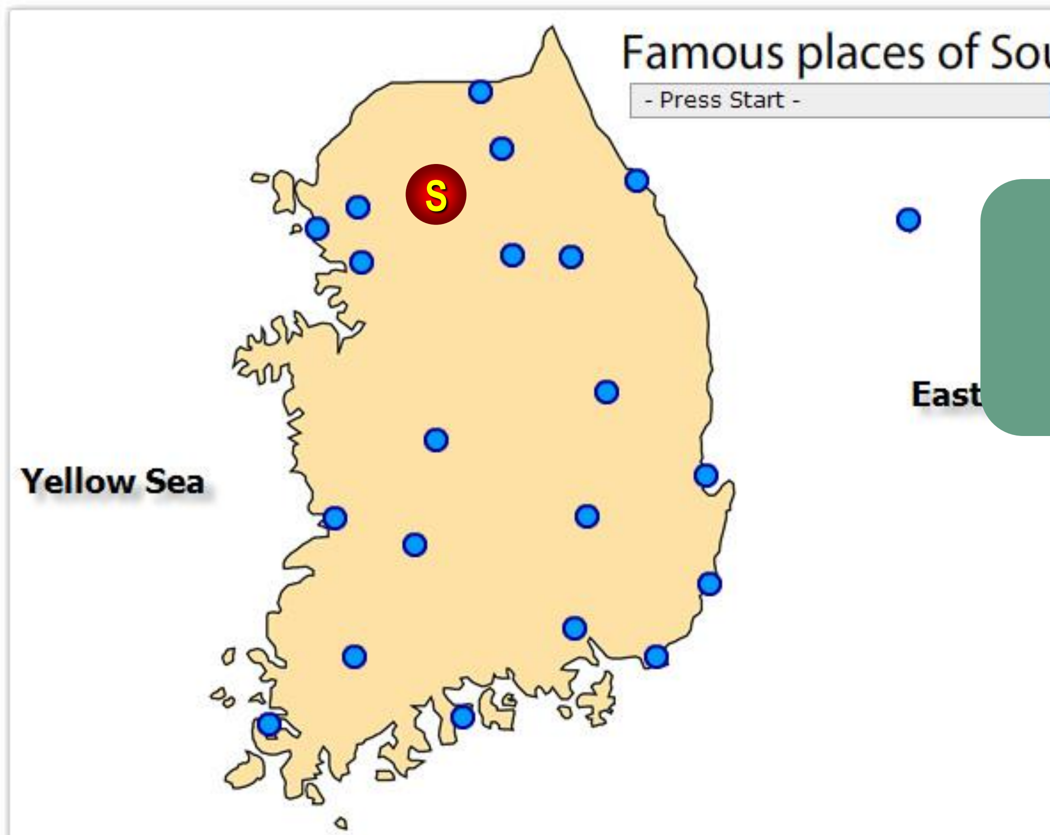
Dept. of Computer Science and Engineering,
Sungkyunkwan University (SKKU)

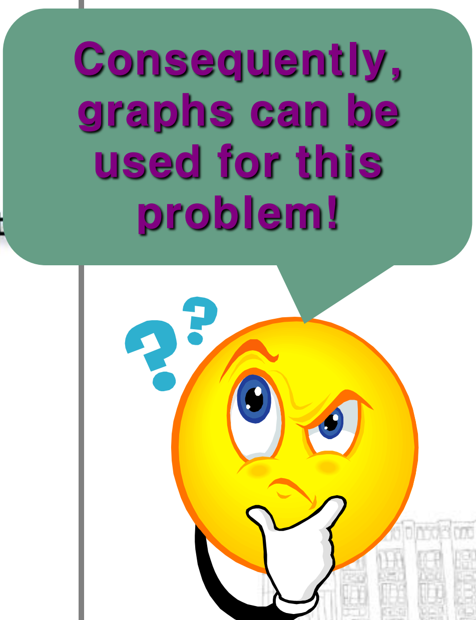
Contents

- Chapter 9 – Graph Traversal

1. Graphs: Prologue

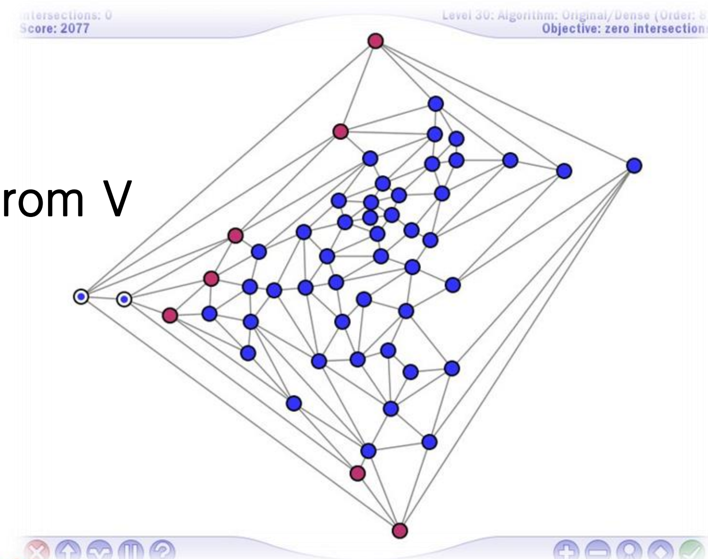
- You would like to visit famous historic sites in Korea, starting from Seoul.
- All the information on distance, highway, and travel cost is available.
- You are trying to write a program for planning your trip.
- Prior to programming, you need to find a proper **Representation** method.



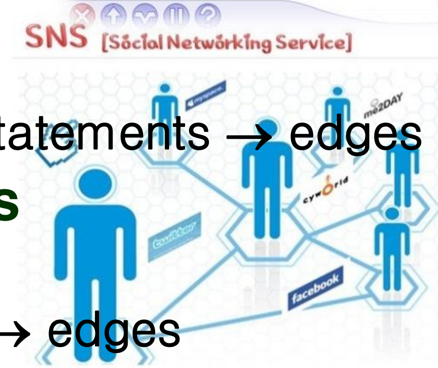


2. Flavors of Graphs <1>

- **Graph G = (V, E)**
  - **V: a set of vertices (or nodes)**
  - **E: a set of edges**
    - E = (x, y) where x, y ∈ V
    - ordered/unordered pairs of vertices from V



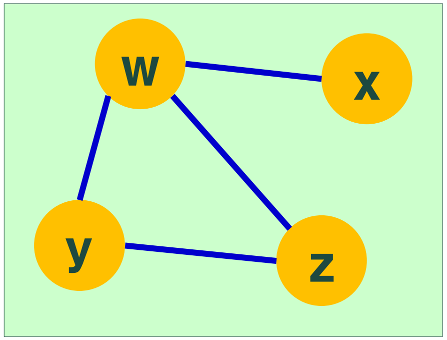
- **Graph Applications**
  - **Modeling a road network**
    - Cities or Junctions → vertices
    - Roads between them → edges
  - **Analyzing a source code**
    - Lines of code → vertices
    - Connecting lines on consecutive statements → edges
  - **Analyzing human interactions**
    - People → vertices
    - Connecting pairs of related souls → edges
  - **Design of scheduling algorithms, logic circuits, communication networks, etc.**

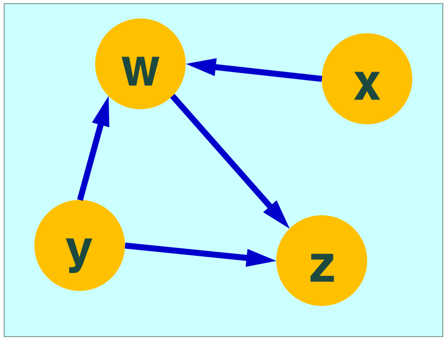


2. Flavors of Graphs 〈2〉

❖ Undirected vs. Directed : A graph G = (V, E)

- is **undirected** if edge (x, y) ∈ E implies that (y, x) ∈ E, too.
- is **directed** if not.



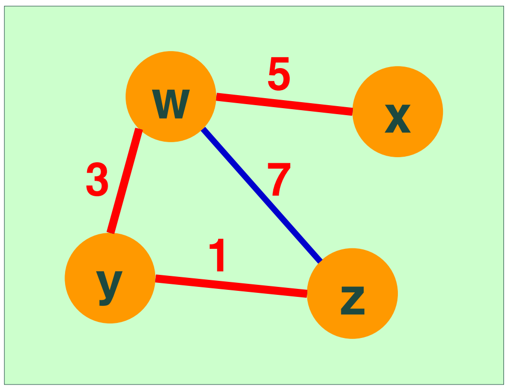


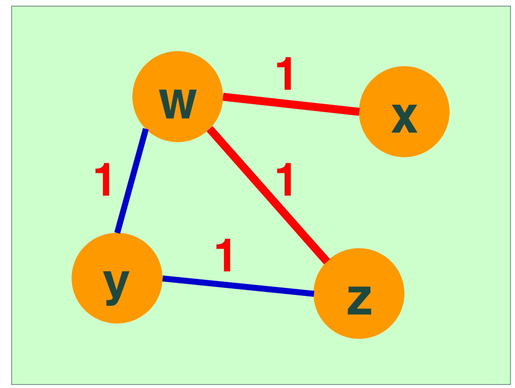
- ex.
  - Road networks *between* cities → undirected
  - Road networks *within* cities → directed because of some one-way streets
  - Program-flow graphs → directed
- Most graphs of graph-theoretic interest are **undirected.**

2. Flavors of Graphs 〈3〉

❖ Weighted vs. Unweighted

- **each edge (or vertex) of G is assigned a numerical value or weight → weighted graph**



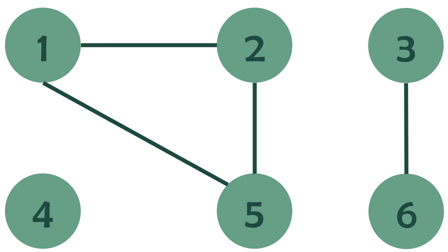


- **They become particularly apparent in finding the shortest path between two vertices.**
  - unweighted : the fewest number of edges
    - the shortest path from x to z in the graph: x->w->z
  - weighted : the smallest sum of the weights on the path
    - the shortest path from x to z in the graph: x->w->y->z

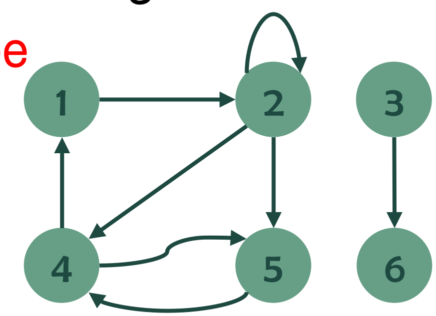
2. Flavors of Graphs <4>

❖ Degree of a vertex

- **undirected**
  - the number of edges incident on it.
    ex) vertex 2 in the graph has degree 2.
  - A vertex whose degree is 0,
    i.e., vertex 4 in the graph, is *isolated*.



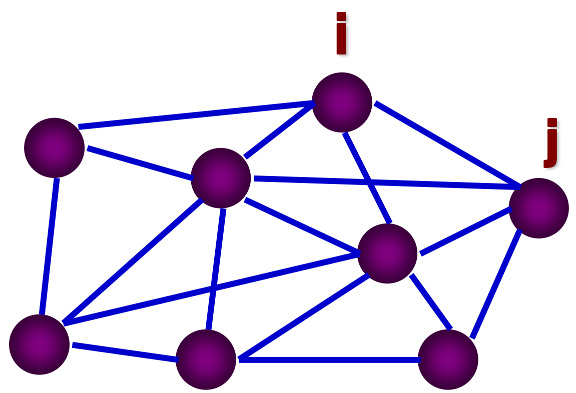
- **directed**
  - out-degree of a vertex : the number of edges leaving it
  - in-degree of a vertex : the number of edges entering it
  - degree of a vertex : its in-degree + out-degree
  - ex) vertex 2 in the right graph
    - in-degree = 2
    - out-degree = 3
    - degree = 2+3 = 5



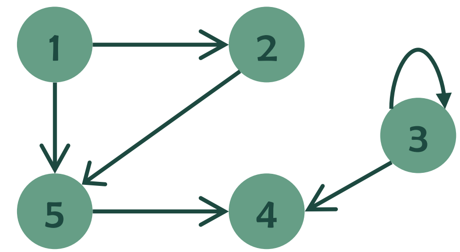
3. Data Structures of Graphs <1>

- G = (V, E), |V|=n and |E|=m

- Adjacency Matrices
  - Represent G using n × n matrix M
  - Each element M[i, j] =
    - 1, if (i, j) ∈ E
    - 0, if (i, j) ∉ E
  - Advantage
    - Fast answers to “is (i, j) in G?”
    - Rapid updates for edge insertion/deletion
  - Disadvantage
    - When n >> m : it uses excessive space (i.e., wasting memory)



3. Data Structures of Graphs <2>



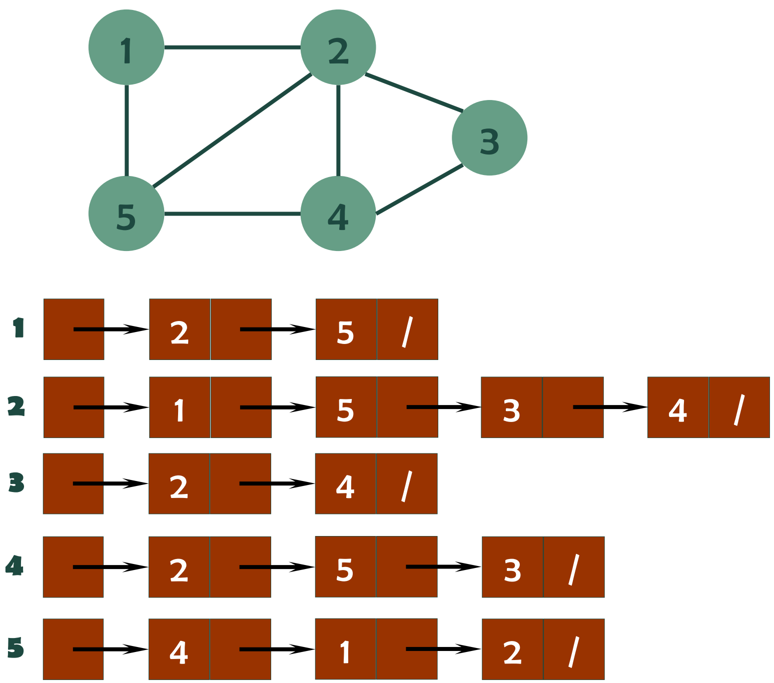
	1	2	3	4	5
1	0	1	0	0	1
2	0	0	0	0	1
3	0	0	1	1	0
4	0	0	0	0	0
5	0	0	0	1	0

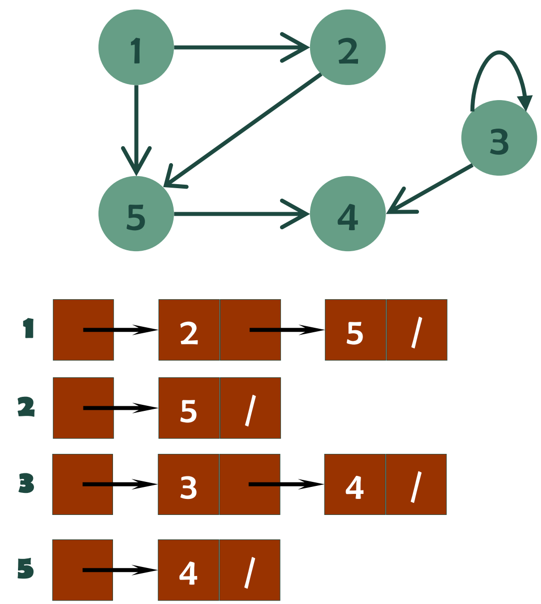
- **A graph of the street map of Manhattan in NY City**
  - Manhattan is a grid of 15 avenues, each crossing 200 streets
  - Every junction is a vertex, with neighboring junctions connected by edges
    - This gives 3000 vertices and 6000 edges
    - The adjacency matrix has 3,000 × 3,000 = 9,000,000 cells (almost all of them empty!)

3. Data Structures of Graphs <3>

Adjacency Lists in Lists

- Use **linked lists** to store the neighbors adjacent to each vertex
- Require **Pointers**
- **Efficient** to represent **Sparse graphs**
- **But, harder to ask whether a given edge (i, j) is in G!**
  since we have to search through the appropriate list to find it

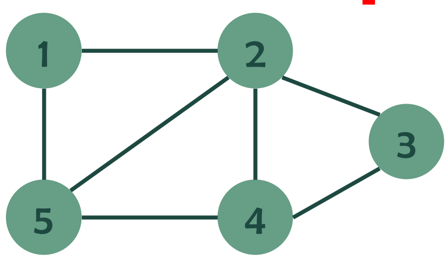




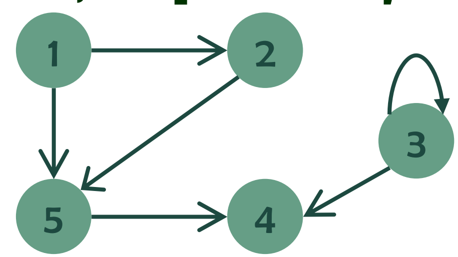
3. Data Structures of Graphs ‹4›

❖ Adjacency Lists in Matrices (it is used in all our examples)

- Represent a list in an array by keeping a count k for no. of elements
- Thus, visit successive elements from the first to last just like a list, but by incrementing an index in a loop
- It seems to combine the worst properties of adjacency matrices (large space) and adjacency lists (the need to search for edges)
- But it is the simplest data structure to program, for particularly static graphs



	1	2	3	4	
1	2	5			2
2	1	5	4	3	4
3	2	4			2
4	2	5	3		3
5	1	2	4		3



	1	2	
1	2	5	2
2	5		1
3	3	4	2
			0
5	4		1

4. Adjacency List Representation of Graph <1>

❖ We represent a graph using Adjacency lists.

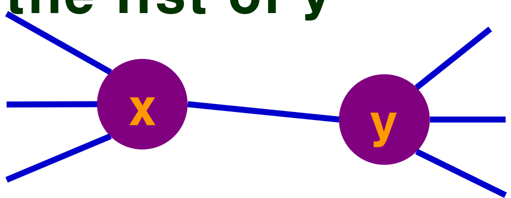
- We keep count of the number of vertices
- We assign each vertex a unique number from 1 to nvertices
- We represent the edges in an MAXV × MAXDEGREE array
  - so, each vertex can be adjacent to MAXDEGREE others.
- cf) When setting MAXV × MAXV, wasteful of space of low-degree graphs!

```
#define MAXV            100             /* maximum number of vertices */
#define MAXDEGREE       50              /* maximum vertex outdegree */

typedef struct {
        int edges[MAXV+1][MAXDEGREE];   /* adjacency info */
        int degree[MAXV+1];             /* outdegree of each vertex */
        int nvertices;                  /* number of vertices in graph */
        int nedges;                     /* number of edges in graph */
} graph;
```

4. Adjacency List Representation of Graph <2>

- We represent a directed edge (x, y)…
  - by the integer y in the adjacency list of x, which is located in the subarray graph -> edges[x]
- The degree field counts…
  - number of meaningful entries for the given vertex
- For a undirected graph, the edge (x, y) appears twice in any adjacency structure;
  - once as y in the list of x
  - once as x in the list of y



If edge(x,y) is the kth edge of vertex x, it becomes graph->edge[x][k] = y.
Since undirected, the process is done in the vertex y.
It becomes graph->edge[y][.] = x.

4. Adjacency List Representation of Graph <3>

❖ How to read in a graph from a file?

```
#define MAXV          100        /* maximum number of vertices */
#define MAXDEGREE     50         /* maximum vertex outdegree */

typedef struct {
        int edges[MAXV+1][MAXDEGREE];  /* adjacency info */
        int degree[MAXV+1];            /* outdegree of each vertex */
        int nvertices;                 /* number of vertices in graph */
        int nedges;                    /* number of edges in graph */
} graph;
```

```
read_graph(graph *g, bool directed)
{
        int i;                          /* counter */
        int m;                          /* number of edges */
        int x, y;                       /* vertices in edge (x,y) */

        initialize_graph(g);

        scanf("%d %d",&(g->nvertices),&m);

        for (i=1; i<=m; i++) {
                scanf("%d %d",&x,&y);
                insert_edge(g,x,y,directed);
        }
}
```

No. of vertices and No. of edges in a graph.

All edge information on each vertice

4. Adjacency List Representation of Graph <4>

❖ Initialization

```
initialize_graph(graph *g)
{
        int i;                          /* counter */

        g -> nvertices = 0;
        g -> nedges = 0;

        for (i=1; i<=MAXV; i++) g->degree[i] = 0;
}
```

❖ Printing the graph is a matter of nested loops:

```
print_graph(graph *g)
{
        int i,j;                        /* counters */

        for (i=1; i<=g->nvertices; i++) {
                printf("%d: ",i);
                for (j=0; j<g->degree[i]; j++)
                        printf(" %d",g->edges[i][j]);
                printf("\n");
        }
}
```

4. Adjacency List Representation of Graph <5>

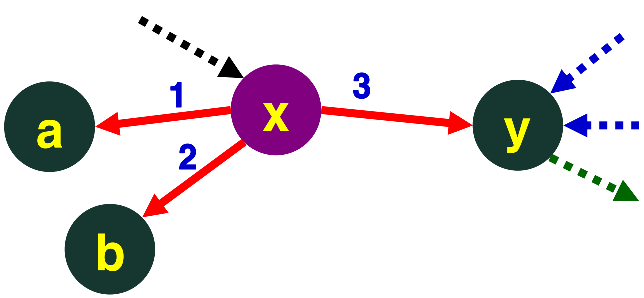
The critical routine is insert_edge()

- **Inserting two copies of each edge or only one by the use of recursion**

```
insert_edge(graph *g, int x, int y, bool directed)
{
        if (g->degree[x] > MAXDEGREE)
            printf("Warning: insertion(%d,%d) exceeds max degree\n",x,y);

        g->edges[x][g->degree[x]] = y;
        g->degree[x] ++;

        if (directed == FALSE)
                insert_edge(g,y,x,TRUE);
        else
                g->nedges ++;
```



directed = true

g->edges[x][0]=a
g->edges[x][1]=b

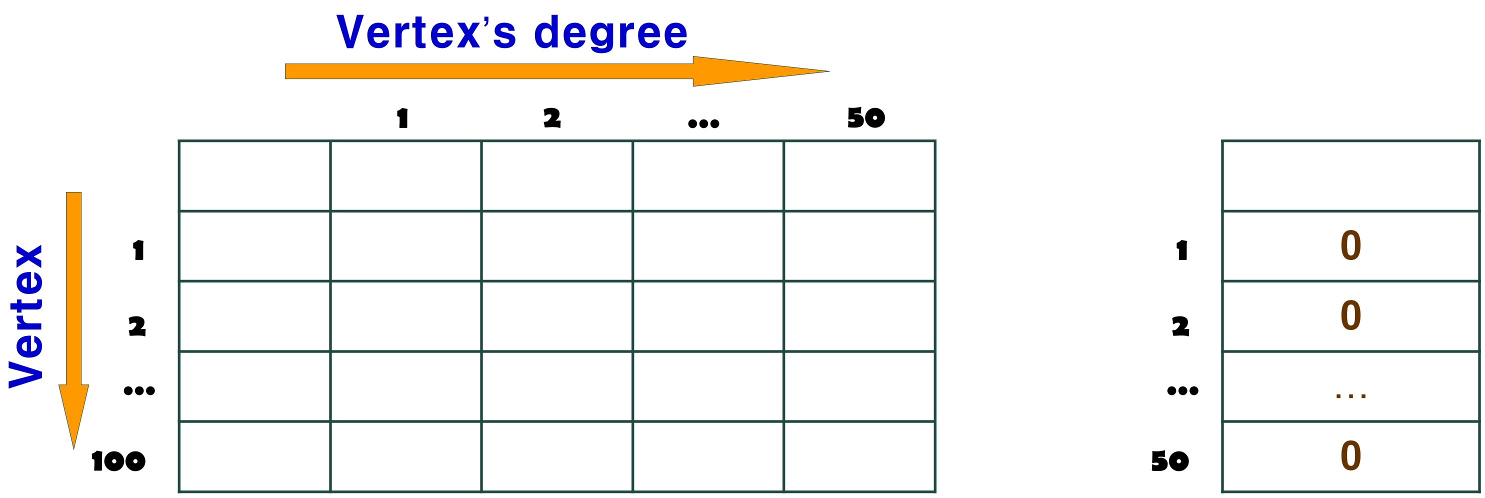
g->degree[x]=2
g->nedges=2

g->edges[x][2]=y

g->degree[x]=3
g->nedges=3

4. Adjacency List Representation of Graph 〈6〉

❖ **The adjacency list matrix after initialize_graph(g)**



edges[101][51]

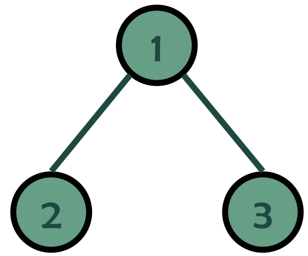
degree[51]

g->nvertices = 0

g->nedges = 0

4. Adjacency List Representation of Graph <7>

- **Ex) Read in this graph!**
  - **g->nvertices = 3**
  - **m = 2**




```
read_graph(graph *g, bool directed)
{
        int i;                          /* counter */
        int m;                          /* number of edges */
        int x, y;                       /* vertices in edge (x,y) */

        initialize_graph(g);

        scanf("%d %d",&(g->nvertices),&m);

        for (i=1; i<=m; i++) {
                scanf("%d %d",&x,&y);
                insert_edge(g,x,y,directed);
        }
}
```

```
insert_edge(graph *g, int x, int y, bool directed)
{
        if (g->degree[x] > MAXDEGREE)
            printf("Warning: insertion(%d,%d) exceeds max degree\n",x,y);

        g->edges[x][g->degree[x]] = y;
        g->degree[x] ++;

        if (directed == FALSE)
                insert_edge(g,y,x,TRUE);
        else
                g->nedges ++;
```

- **for (i=1; i<=2, i++)**
  - **x = 1, y = 2, directed = false**
  - **insert_edge(g, 1, 2, false)**
    - g->edges[1][g->degree[1]] = g->edges[1][0] = 2
    - g->degree[1]++

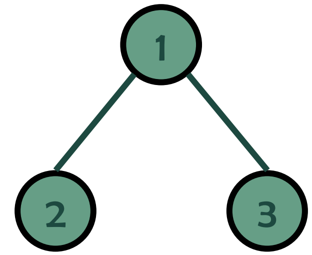
		1	2	…	50
1	2				
2					
…					
100					

edges[101][51]

1	1
2	0
…	…
50	0

degree[51]

4. Adjacency List Representation of Graph <8>



❖ for – cont'd

- **if (directed == FALSE)**
  - insert_edge(g, 2, 1, TRUE)
    - g->edges[2][g->degree[2]] = g->edges[2][0] = 1
    - g->degree[2] ++

	0	1	2	…	50
0					
1	2				
2	1				
…					
100					

1	1
2	1
…	…
50	0

    - else
      - » g->nedges ++ (i.e., 0 → 1)

❖ Next…

- insert_edge(g, 1, 3, FALSE)
- insert_edge(g, 3, 1, TRUE)