6.1 Graph 용어

그림 6.1처럼 그래프 G는 vertices의 집합 V(G)와 edge들의 집합 E(G)에 대하여 G = (V, E)로 정의한다. undirected graph는 (u, v) 또는 (v, u)로 표기하며 같은 edge를 나타낸다. directed graph는 <u, v>로 표기하며 u를 tail이라 하고 v를 edge의 head라고 부른다.

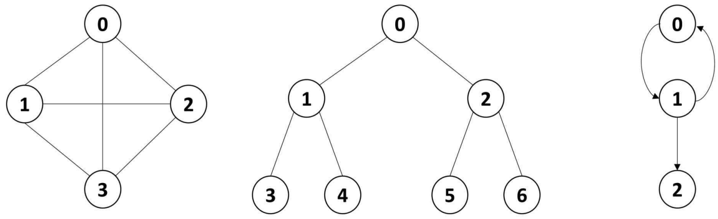


그림 6.1 그래프.

E(G)의 한 edge가 (u, v)일 때 u와 v는 *adjacent* 하다고 한다. edge (u,v)는 u, v에 대하여 *incident* 하다고 한다. <u, v>가 directed edge이므로 u 는 v에 adjacent to라고 하고 v는 u로 부터 adjacent from이라고 한다. edge <u, v>는 u와 v에 대하여 incident라고 한다.

u로 부터 v로 가는 path는 $<u, i_1, i_2, \dots, i_k, v>$이며 path length는 path의 # of edges 이다. simple path는 출발점과 도착점이 다르고 중간 vertices들은 모두 다른(중간 경유 verices가 같은 것이 없는 것) path를 말한다. cycle은 출발점과 도착점이 같은 simple path를 말한다. 그래프 G에서 u로 부터 v가 가는 path가 존재하면 u와 v는 connected 라고 한다. undirected graph의 connected component는 maximal connected graph 를 말한다.

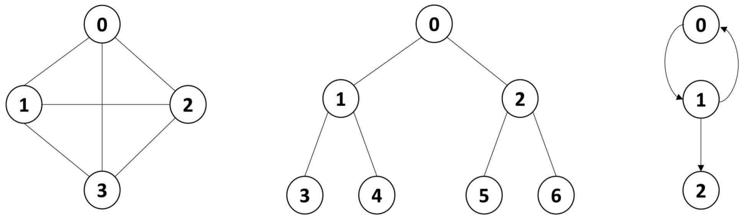
tree는 connected acyclic graph이다. directed graph G의 모든 vertices u와 v에 대하여 u -> v 또는 v -> u로 가는 directed path가 존재하면 strongly connected라고 한다. strongly connected component는 strongly connected인 maximal subgraph 이다.

vertex의 degree는 vertex에 incident 한 # of edges이다. G가 directed graph일 때 vertex v의 in-degree는 v가 head인 # of edges이다. v의 out-degree는 v가 tail인 # of edges이다.

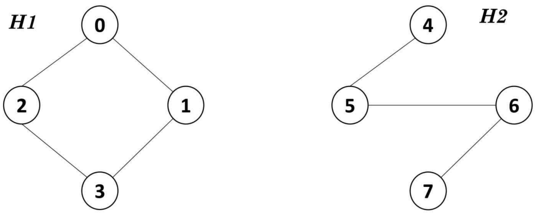
6.2 Graph 표현 구조

6.2.1 Adjacency matrix

그래프 G의 adjacency matrix[1]는 그림 6.2처럼 2차원의 n x n 배열로 표현한다. edge (i, j)가 E(G)에 있으면 A[i][j] = 1로 표현한다. 방향성 그래프와 비방향성 그래프 모두 배열로 표현하는 방법이다.



(a) G_1 (b) G_2 (c) G_3



G_4

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

(a) G_1

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{array}$$

(b) G_3

$$\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

(c) G_4

그림 6.2 그래프의 행렬 표현.

6.2.2 Adjacency lists

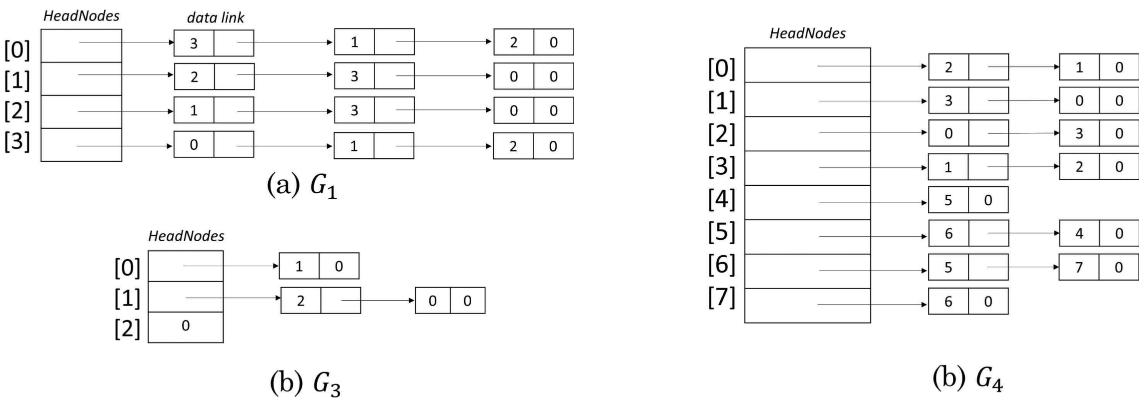
adjacency matrix의 n row가 n linked list로 표현하는 방법이다. 그래프 G에서 각 vertex를 위한 하나의 list가 있다. list i의 노드는 vertex i으로부터 adjacent from인 vertices를 나타낸다. 각 list node는 data와 link를 포함한다. 각 노드의 data field는 vertex i에 adjacent to인 vertex index를 저장한다.

n vertices와 e edge를 갖는 undirected graph에 대하여 adjacency list[1]는 그림 6.3처럼 n head node와 2e list node를 갖는다. undirected graph의 vertex degree는 해당 adjacency list의 node 개수를 세면 된다.

directed graph는 list node 개수는 e이다. vertex의 out-degree는 adjacency list의 노드 개수를 세면 된다. vertex의 in-degree는 계산이 복잡하다. 어떤 vertex에 대한

adjacent to인 vertex를 구하기 위해서는 inverse adjacency list를 만드는 것이 더 편리하다. inverse adjacency list에서 각 list는 vertex에 adjacent to인 vertex 들을 나타낸다.

graph의 edge가 weight를 가질 수 있다. 예를 들면 edge weight는 vertex 간의 거리 등을 표현할 수 있다. adjacency list의 각 list node에 weight field를 포함하는 것을 weight edge라 한다. weight edge로 표현된 graph를 network이라 한다.



(a) G_1 (b) G_3 (b) G_4

그림 6.3 그래프의 Adjacency list 표현.

```
//소스 코드 6.1: Graph Representation
//Adjacency Lists + BFS + DFS
import java.util.*;

class ListNode {
    int data;
    ListNode link;

    ListNode(int data) {
        this.data = data;
        this.link = null;
    }
}

class LinkedList {
    ListNode first = null;

    void insert(int data) {
        ListNode newNode = new ListNode(data);
        newNode.link = first;
        first = newNode;
    }

    boolean contains(int data) {
```

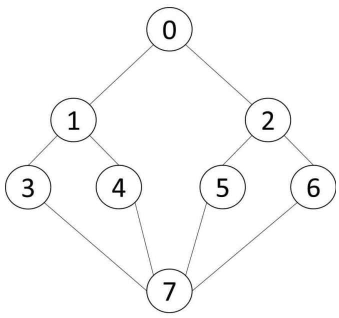
```
        ListNode current = first;
        while (current != null) {
            if (current.data == data) return true;
            current = current.link;
        }
        return false;
    }

    void printList() {
        if (first == null) {
            System.out.println("null");
            return;
        }
        ListNode current = first;
        while (current != null) {
            System.out.print(current.data);
            if (current.link != null) System.out.print(" -> ");
            current = current.link;
        }
        System.out.println();
    }

    ListNode getFirst() {
        return first;
    }
}

class Graph {
    private LinkedList[] headNodes;
    private int n;
    private boolean[] visited;

    public Graph(int vertices) {
        this.n = vertices;
        headNodes = new LinkedList[n];
        for (int i = 0; i < n; i++) {
            headNodes[i] = new LinkedList();
        }
    }

    public void insertEdge(int start, int end) {
        if (start < 0 || start >= n || end < 0 || end >= n) {
            System.out.println("Node index out of bounds");
            return;
        }
```

}

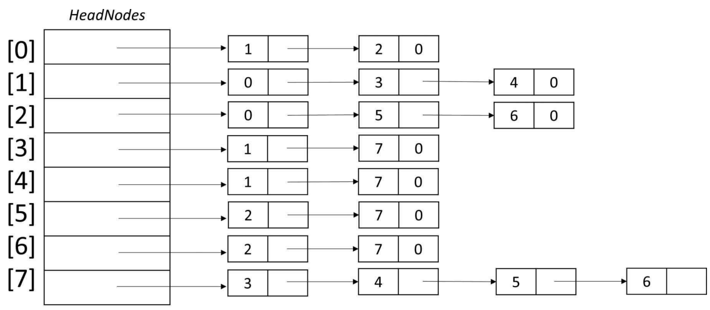
```java
        if (!headNodes[start].contains(end)) {
            headNodes[start].insert(end);
            headNodes[end].insert(start); // Undirected graph
        }
    }

    public void displayAdjacencyLists() {
        for (int i = 0; i < n; i++) {
            System.out.print(i + " -> ");
            headNodes[i].printList();
        }
    }

    public void bfs(int v) {
        visited = new boolean[n];
        Queue<Integer> q = new LinkedList<>();
        visited[v] = true;
        System.out.print(v + ", ");
        q.offer(v);

        while (!q.isEmpty()) {
            int curr = q.poll();
            ListNode node = headNodes[curr].getFirst();
            while (node != null) {
                int w = node.data;
                if (!visited[w]) {
                    visited[w] = true;
                    System.out.print(w + ", ");
                    q.offer(w);
                }
                node = node.link;
            }
        }
        System.out.println();
    }

    public void dfs(int v) {
        visited = new boolean[n];
        dfsRecursive(v);
        System.out.println();
    }

    private void dfsRecursive(int v) {
        visited[v] = true;
```

```
        System.out.print(v + ", ");
        ListNode node = headNodes[v].getFirst();
        while (node != null) {
            int w = node.data;
            if (!visited[w]) dfsRecursive(w);
            node = node.link;
        }
    }
}

public class GraphTraversal {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int select = -1, n, startEdge = -1, endEdge = -1;
        int startBFSNode = Integer.MAX_VALUE;

        System.out.print("Input the total node number: ");
        n = sc.nextInt();
        Graph g = new Graph(n);

        while (select != 5) {
            System.out.println("\nSelect command: 1: Add edges, 2: Display
Adjacency Lists, 3: BFS, 4: DFS, 5: Quit");
            select = sc.nextInt();
            switch (select) {
                case 1:
                    System.out.print("Input start node: ");
                    startEdge = sc.nextInt();
                    System.out.print("Input end node: ");
                    endEdge = sc.nextInt();

                    if (startEdge < 0 || startEdge >= n || endEdge < 0 || endEdge
>= n) {
                        System.out.println("The input node is out of bound.");
                        break;
                    }

                    if (startEdge < startBFSNode) startBFSNode = startEdge;
                    if (endEdge < startBFSNode) startBFSNode = endEdge;

                    g.insertEdge(startEdge, endEdge);
                    break;

                case 2:
                    g.displayAdjacencyLists();
```

```
                    break;

                case 3:
                    System.out.println("Start BFS from node: " + startBFSNode);
                    g.bfs(startBFSNode);
                    break;

                case 4:
                    System.out.println("Start DFS from node: " + startBFSNode);
                    g.dfs(startBFSNode);
                    break;

                case 5:
                    System.out.println("Exiting...");
                    break;

                default:
                    System.out.println("Wrong input. Re-enter.");
            }
        }
        sc.close();
    }
}
```



(a)



(b)

그림 6.4 그래프의 List 표현.

6.3 DFS와 BFS

그래프 G = (V, E)에서 V(G)의 한 vertex v에 대하여 v로 부터 reachable 한 모든 vertices를 방문하는 알고리즘을 만든다. 그림 6.4에 대하여 vertex v에 연결된 모든 vertices를 방문하는 방법으로 depth-first search와 breadth-first search[1]가 있다.

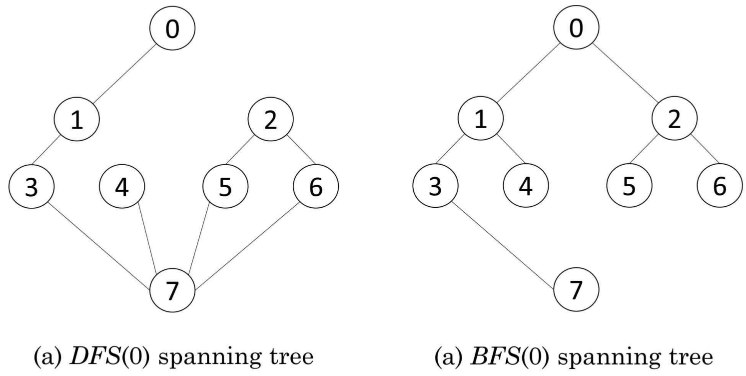


그림 6.5 DFS와 BFS.

6.3.1 Depth-first search

시작 vertex를 v라고 하자. 그림 6.5처럼 v에 adjacent to인 미방문 vertex w를 선택한 다음에 w에 또 다시 adjacent to인 미방문 노드를 선택한다. 만일 w에 인접한 미방문 노드를 방문한 다음에 아직도 미방문 노드가 남아 있으면 v로 돌아가(backtrack) v에 인접한 미방문 노드를 찾아가게 된다. v로부터 reachable 한 모든 노드가 방문하게 될 때 종료된다. depth-first search의 driver는 void Graph::DFS(int v)이며 workhorse는 void Graph::_DFS(const int v)으로 구현한다.

6.3.2 Breadth-first search

그림 6.5처럼 vertex v에서 시작할 때 v에 adjacent to인 모든 미방문 노드를 방문한다. v에 인접한 미방문 노드를 모두 queue에 넣고 방문 처리한 후에 queue에서 하나씩 꺼내 다음 vertex를 처리하는 방식이다. void Graph::BFS(int v)가 queue를 사용하여 non-recursive 알고리즘으로 동작한다. vertex v에 대한 connected components는 DFS(v) 또는 BFS(v)를 처리하여 찾을 수 있다.

6.3.3 Spanning tree

DFS 또는 BFS는 그래프 G를 2개의 set, T(tree edges)와 N(nontree edges)을 partition 한다. spanning tree는 그림 6.6처럼 tree edges[1]의 집합인 T를 말한다. T는 T = T + {(v,w)}으로 set에 추가되며 G의 모든 vertex를 포함한다.

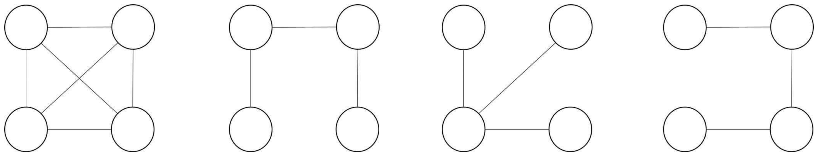


그림 6.6 여러 가지 spanning tree.

depth-first spanning tree는 DFS에 의해서 breadth-first spanning tree는 BFS에 의해서 만들어진다. 그래프 G의 minimal spanning tree는 G의 minimal subgraph, G'로서 V(G') = V(G)이고 G'가 connected인 것을 말한다.