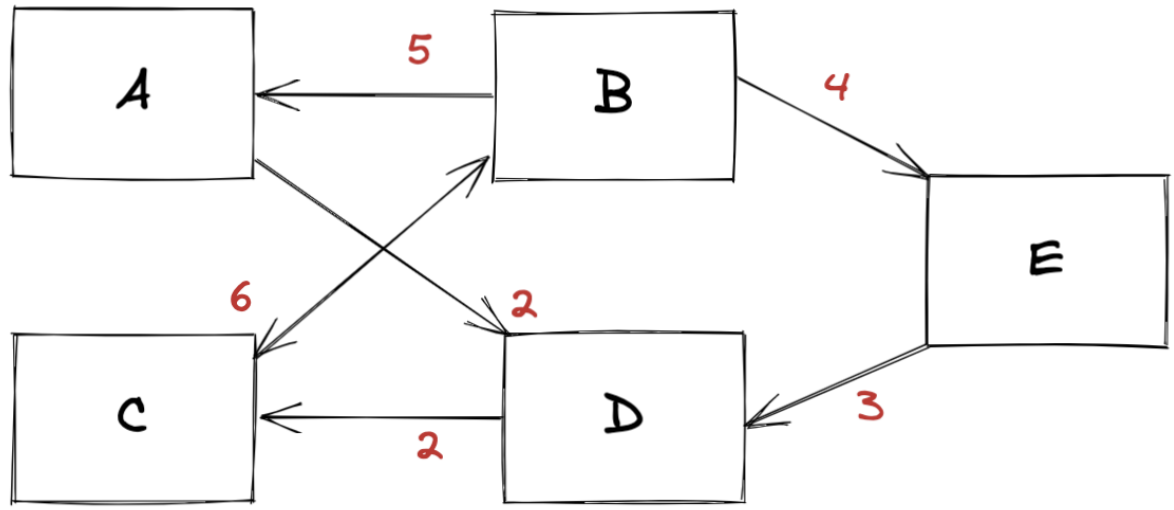
[읽을거리] 8. 그래프

방향있는 가중치 그래프



위 그림과 같은 방향성있고, 가중치가 주어진 그래프를 만들고 A 에서 E 까지 찾아가는 방법을 구현해 봅시다.

Connection

가중치는 Node와 Node 사이의 연결에 주어지는 것이니까 이 연결을 표현하는 Connection이라는 클래스를 만들겠습니다.

```
class Connection {
    Node node;
    int weight;

    public Connection(Node node, int weight) {
        this.node = node;
        this.weight = weight;
    }
}
```

그러면 Node는 이렇게 변경될 것입니다.

```
class Node {
    String name;
    List<Connection> links;
    boolean visited;

    public Node(String name) {
        this.name = name;
        this.links = new LinkedList<>();
    }
```

```
    void link(Node node, int weight) {
        links.add(new Connection(node, weight));
    }

    void visit() {
        this.visited = true;
    }

    boolean isVisited() {
        return this.visited;
    }

    @Override
    public String toString() {
        return name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Node node = (Node) o;
        return Objects.equals(name, node.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }
}
```

Node의 연결

방향성을 고려해서 가중치와 함께 연결을 만들어 냅니다.

```
Node a = new Node("A");
Node b = new Node("B");
Node c = new Node("C");
Node d = new Node("D");
Node e = new Node("E");

a.link(d, 2);
b.link(a, 5);
b.link(c, 6);
b.link(e, 4);
c.link(b, 6);
d.link(c, 2);
e.link(d, 3);
```

BFS 탐색

A에서 E를 찾아가는 탐색과 그 과정까지의 총 weight 를 계산해 봅시다.

```
Node target = e;

// BFS
```

```
Queue<Connection> queue = new LinkedList<>();
queue.offer(new Connection(a, 0));

while (!queue.isEmpty()) {
    Connection con = queue.poll();
    Node n = con.node;
    int weight = con.weight;
    n.visit();
    System.out.println(n + " (" + weight + ")");

    if (n.equals(target)) {
        System.out.println("FOUND!!");
        break;
    }

    n.links.stream()
            .filter(i -> !i.node.isVisited())
            .filter(i -> !queue.contains(i))
            .map(i -> new Connection(i.node, i.weight + weight))
            .forEach(queue::offer);
}
```

결과

```
A (0)
D (2)
C (4)
B (10)
E (14)
FOUND!!
```

이동가능 경로를 따라 이동하면서 가중치가 누적되어 최종으로 14의 가중치와 함께 E를 찾아낸 결과입니다.

Queue를 Stack으로 변경하면 DFS가 되죠? 이 코드를 DFS로 변경해서 실행해 봐도 실행 결과는 동일하게 출력됩니다. 방향성에 따라서 이동 경로가 자유롭지 못하게 되면서 BFS와 DFS의 경로상의 차이도 없어지게 된 것입니다.

인접행렬을 사용한 그래프 생성

각 Node 간의 이동가능 여부와 가중치를 표로 만들어보면 이렇게 표시할 수 있습니다.

	A 로	B 로	C 로	D 로	E 로
A 에서				2	
B 에서	5		6		4
C 에서		6			
D 에서			2		
E 에서				3	

숫자는 가중치 이고, 빈 칸은 이동할 수 없는 경로입니다.

이 표를 array로 이렇게 구성할 수 있겠죠.

```
int[][] matrix = {
        {0, 0, 0, 2, 0},
        {5, 0, 6, 0, 4},
        {0, 6, 0, 0, 0},
        {0, 0, 2, 0, 0},
        {0, 0, 0, 3, 0},
};
```

이제 이 matrix 배열만 있으면 그래프를 동적으로 구성할 수 있습니다.

```
List<Node> nodes = new ArrayList<>() {{
    add(new Node("A"));
    add(new Node("B"));
    add(new Node("C"));
    add(new Node("D"));
    add(new Node("E"));
}};

int[][] matrix = {
        {0, 0, 0, 2, 0},
        {5, 0, 6, 0, 4},
        {0, 6, 0, 0, 0},
        {0, 0, 2, 0, 0},
        {0, 0, 0, 3, 0},
};

generateGraph(nodes, matrix);
```

```
void generateGraph(List<Node> nodes, int[][] matrix) {
    for (int i = 0; i < matrix.length; i++) {
        int[] row = matrix[i];
        for (int j = 0; j < row.length; j++) {
            if (row[j] == 0) continue;
            nodes.get(i).link(nodes.get(j), row[j]);
        }
    }
}
```

Generic Graph

제너릭 타입을 사용하는 그래프 클래스를 만들고 재사용가능한 자료구조를 만들어 봅시다.

Connection.java

```
public class Connection<T> {
    private Node<T> node;
    private int weight;

    public Connection(Node<T> node, int weight) {
        this.node = node;
        this.weight = weight;
    }
```

```
    public Node<T> getNode() {
        return node;
    }

    public int getWeight() {
        return weight;
    }
}
```

Node.java

제너릭 타입 T 를 사용하도록 변경하고, `resetVisit` 로 visited를 초기화 하여 재사용 할 수 있도록 했습니다.

```
import java.util.LinkedList;
import java.util.List;
import java.util.Objects;

public class Node<T> {
    private T data;
    private List<Connection<T>> links;
    private boolean visited;

    public Node(T name) {
        this.data = name;
        this.links = new LinkedList<>();
    }

    public List<Connection<T>> connections() {
        return links;
    }

    public void link(Node<T> node, int weight) {
        links.add(new Connection(node, weight));
    }

    public void resetVisit() {
        this.visited = false;
    }

    public void visit() {
        this.visited = true;
    }

    public boolean isVisited() {
        return this.visited;
    }

    @Override
    public String toString() {
        return data.toString();
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Node node = (Node) o;
        return Objects.equals(data, node.data);
    }
```

```
    @Override
    public int hashCode() {
        return Objects.hash(data);
    }
}
```

Graph.java

bfs, dfs 함수를 내장했고, 각 탐색을 시작하기 전에 reset을 통해 visited를 초기화 합니다.

```
import java.util.*;

public class Graph<T> {
    private List<Node<T>> nodes = new ArrayList<>();

    public void addNode(Node<T> node) {
        nodes.add(node);
    }

    public Node<T> getNode(int index) {
        return nodes.get(index);
    }

    public void generate(int[][] matrix) {
        for (int i = 0; i < matrix.length; i++) {
            int[] row = matrix[i];
            for (int j = 0; j < row.length; j++) {
                if (row[j] == 0) continue;
                nodes.get(i).link(nodes.get(j), row[j]);
            }
        }
    }

    public void reset() {
        nodes.forEach(Node::resetVisit);
    }

    public Connection<T> bfs(Node<T> start, Node<T> target) {
        reset();

        Queue<Connection<T>> queue = new LinkedList<>();
        queue.offer(new Connection(start, 0));

        while (!queue.isEmpty()) {
            Connection<T> con = queue.poll();
            Node<T> n = con.getNode();
            int weight = con.getWeight();
            n.visit();

            if (n.equals(target)) {
                return new Connection<>(target, weight);
            }

            n.connections().stream()
                    .filter(i -> !i.getNode().isVisited())
                    .filter(i -> !queue.contains(i))
                    .map(i -> new Connection(i.getNode(), i.getWeight() + weight))
                    .forEach(queue::offer);
        }

        return null;
```

```
    }

    public Connection<T> dfs(Node<T> start, Node<T> target) {
        reset();

        Stack<Connection<T>> stack = new Stack<>();
        stack.push(new Connection(start, 0));

        while (!stack.isEmpty()) {
            Connection<T> con = stack.pop();
            Node<T> n = con.getNode();
            int weight = con.getWeight();
            n.visit();

            if (n.equals(target)) {
                return new Connection<>(target, weight);
            }

            n.connections().stream()
                    .filter(i -> !i.getNode().isVisited())
                    .filter(i -> !stack.contains(i))
                    .map(i -> new Connection(i.getNode(), i.getWeight() + weight))
                    .forEach(stack::push);
        }

        return null;
    }
}
```

Graph 사용하기

Node를 관리하는 Graph 객체를 통해서 연결과 가중치를 동적으로 생성할 수 있습니다.

```
public class Main {
    public static void main(String[] args) {
        Graph<String> graph = new Graph<>();
        graph.addNode(new Node("A"));
        graph.addNode(new Node("B"));
        graph.addNode(new Node("C"));
        graph.addNode(new Node("D"));
        graph.addNode(new Node("E"));

        graph.generate(new int[][]{
                {0, 0, 0, 2, 0},
                {5, 0, 6, 0, 4},
                {0, 6, 0, 0, 0},
                {0, 0, 2, 0, 0},
                {0, 0, 0, 3, 0},
        });

        Node a = graph.getNode(0);
        Node target = graph.getNode(4);

        var answer1 = graph.bfs(a, target);
        System.out.println(String.format("BFS : %s (%d)", answer1.getNode(), answer1.getWeight()));
        // BFS : E (14)

        var answer2 = graph.dfs(a, target);
        System.out.println(String.format("DFS : %s (%d)", answer2.getNode(), answer2.getWeight()));
        // DFS : E (14)
```

```
    }
}
```