알고리즘 과제 보고서

- AL09, 동등 클래스

충남대학교 컴퓨터공학과
분반: 알고리즘 04 분반
학번: 201701975
이름: 구건모

과제에서 해야할 일과 주요이론

이번 과제의 주요 이론은 EquivalanceClass 로, 그래프에서 동등한 관계를 가지는 원소들을 하나로 묶은 동등 클래스들을 찾는 프로그램을 만드는 것이 목표입니다.
프로그램에서 원소들의 관계쌍을 추가하여 추가된 graph를 바탕으로 동등한 클래스에 해당하는 원소들을 찾아 EquivalanceClass로 묶어 저장한 후, 찾아진 EquivalanceClass들을 EquivalanceClassList에 저장하여 이후에 결과에서 출력해주도록 구현하는 것이 과제에서 해야할 일이라고 볼 수 있습니다.

주요 구현 내용

Class UndirectedAdjacencyListGraph

```java
package graph;

public class UndirectedAdjacencyListGraph<E extends Edge> extends DirectedAdjacencyListGraph<E> {

    public UndirectedAdjacencyListGraph(int givenNumberOfVertices) {
        super(givenNumberOfVertices);
    }

    public boolean addEdge(E anEdge) {
        if (this.edgeIsValid(anEdge) && (!this.edgeDoesExist(anEdge))) {
            this.neighborListOf(anEdge.tailVertex()).add(anEdge);
            @SuppressWarnings("unchecked")
            E reversedEdge = (E)anEdge.reversed();
            this.neighborListOf(reversedEdge.tailVertex()).add(reversedEdge);
            this.setNumberOfEdges(this.numberOfEdges() + 1);
            return true;
        }
        return false;
    }

}
```

Class UndirectedAdjacencyListGraph

UndirectedAdjacenyListGraph는 동등 클래스 분류에 사용할 관계쌍과 그 원소들을 표현할 그래프로 기존에 구현하였던 DirectedAdjacencyListGraph를 상속받아 구현됩니다.

Undirected와 Directed는 방향성의 존재 유무 이외의 차이가 없으므로 UndirectedAdjacencyListGraph 를 구현할 때에 다른 메서드들은 그대로 사용하고 Edge를 추가하는 addEdge만 Override 하여, Edge가 추가될 때 반대방향도 추가되도록 하는 방식으로 구현하면 UndirectedAdjacencyListGraph의 구현이 완료됩니다. 메서드 구현을 보면 추가하려는 Edge와 reversed(역방향) 관계에 있는 Edge를 Edge 클래스의 reversed() 메서드를 이용해 다시 추가해주므로써 무방향 그래프를 구현합니다.

주요 구현 내용

Class UndirectedAdjacencyMatrixGraph

_Main_AL09_S... | AppView.java | AppController... | EquivalanceCl... | ArrayStack.java | DirectedAdja... | UndirectedAdj... | LinkedNode.java

```java
package graph;

public class UndirectedAdjacencyMatrixGraph<E extends Edge> extends DirectedAdjacencyMatrixGraph<E> {
    public UndirectedAdjacencyMatrixGraph(int givenNumberOfVertices) {
        super(givenNumberOfVertices);
    }

    public boolean addEdge(E anEdge) {
        if (this.edgeIsValid(anEdge) && (!this.edgeDoesExist(anEdge))) {
            int tailVertex = anEdge.tailVertex();
            int headVertex = anEdge.headVertex();

            this.setAdjacencyOfEdgeAsExist(tailVertex, headVertex);
            this.setAdjacencyOfEdgeAsExist(headVertex, tailVertex);
            this.setNumberOfEdges(this.numberOfEdges() + 1);
            return true;
        }
        return false;
    }

}

```

Class UndirectedAdjacencyMatrixGraph

Graph를 Matrix로 구현한 것입니다. 이번 과제에서는 ListGraph와 MatrixGraph 두개를 모두 사용해보는 과정이 포함되어 있기 때문에 구현하였으며, DirectedAdjacencyMatrixGraph에서 addEdge 부분만 Override 하였습니다 Edge가 추가될 때 양방향 모두 추가되도록 구현하여 Undirected 특성을 가지도록 하였습니다. 이후에 MatrixGraph나 ListGraph나 어떤것을 이용하던지, 복잡도는 다르겠지만 결과는 동일합니다.

주요 구현 내용

```java
package list;

public class ArrayStack<E> implements Stack<E> {
    private static final int DEFAULT_CAPACITY = 100;

    private int _capacity;
    private int _top;
    private E[] _elements;

    public ArrayStack() {
        this(ArrayStack.DEFAULT_CAPACITY);

    }

    @SuppressWarnings("unchecked")
    public ArrayStack(int givenCapacity) {
        this.setCapacity(givenCapacity);
        this.setElements((E[]) new Object[this.capacity()]);
        this.setTop(-1);
    }

    private int capacity() {
        return this._capacity;
    }

    private void setCapacity(int newCapacity) {
        this._capacity = newCapacity;
    }

    private int top() {
        return this._top;
    }

    private void setTop(int newTop) {
        this._top = newTop;
    }

    protected E[] elements() {
        return this._elements;

    }

    private void setElements(E[] newElements) {
        this._elements = newElements;
    }

```

```java
    public void reset() {
        // TODO Auto-generated method stub
        this.setTop(-1);
    }

    @Override
    public int size() {
        // TODO Auto-generated method stub
        return this.top() + 1;
    }

    @Override
    public boolean isEmpty() {
        // TODO Auto-generated method stub
        if (this.size() == 0) {
            return true;
        }
        return false;
    }

    @Override
    public boolean isFull() {
        // TODO Auto-generated method stub
        if (this.size() == this.capacity()) {
            return true;
        }
        return false;
    }

    @Override
    public boolean push(E anElement) {
        // TODO Auto-generated method stub

        if (!this.isFull()) {
            this.elements()[this.top() + 1] = anElement;
            this.setTop(this.top() + 1);

            return true;
        }
        return false;
    }

    @Override
    public E pop() {
        // TODO Auto-generated method stub
        if (!this.isEmpty()) {
            E top_element = this.elements()[this.top()];
            this.setTop(this.top() - 1);
            return top_element;

        }

        return null;
    }

    @Override
    public E peek() {
        // TODO Auto-generated method stub
        if (!this.isEmpty()) {
            return this.elements()[this.top()];
        }
        System.out.println("*** pop() ***");

        return null;
    }
}
```

*코드의 일부입니다.

Class ArrayStack

동등클래스 찾기 과정에서 사용될 자료구조인 ArrayStack 입니다.
stack을 array를 이용해 구현한 것으로 멤버변수인 _elements 에
실질적인 값이 담기게 되고 그 값을 stack의 구조에 맞게 사용하도록
push, pop 등 stack 관련 method들이 구현되어 있습니다.
Stack은 LIFO(Last in First out) 의 방식으로 값을 사용하므로
값을 push하면 그 값이 top이 되며 pop하면 가장 최근에 들어온 값인
top을 꺼낸 후 Stack에서 제거합니다. peek 메소드는 가장 최근에
추가된 값을 return 해주는 메소드 입니다. 이후에 동등 클래스를 찾는 과정에서
ArrayStack에 동등 클래스의 원소들을 담아 사용하게 됩니다.

주요 구현 내용

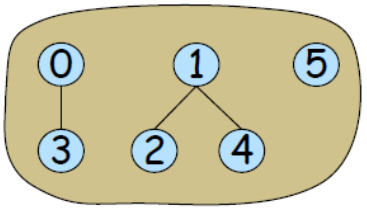
```
public boolean solve(AdjacencyGraph<E> aGraph) {
    this.setGraph(aGraph);
    if (this.graph().numberOfVertices() < 1) {
        return false;
    }
    this.setFound(new boolean[this.graph().numberOfVertices()]);
    this.setEquivalanceClassList(new LinkedList<List<Integer>>());
    this.setSameClassMembers(new ArrayStack<Integer>());
    EquivalanceClasses.showDebugMessage("\n");
    for (int vertex = 0; vertex < this.graph().numberOfVertices(); vertex++) {
        if (!this.found()[vertex]) {
            EquivalanceClasses.showDebugMessage("[Debug] 새로운 동등 클래스: {");
            List<Integer> newEquivalanceClass = new LinkedList<Integer>();
            this.equivalanceClassList().add(newEquivalanceClass);

            this.found()[vertex] = true;
            newEquivalanceClass.add(vertex);
            EquivalanceClasses.showDebugMessage(" "+vertex);
            this.sameClassMembers().push(vertex);

            while (!this.sameClassMembers().isEmpty()) {
                int tailVertex = this.sameClassMembers().pop();
                Iterator<E> iterator = this.graph().neighborIteratorOf(tailVertex);
                while (iterator.hasNext()) {
                    E edge = iterator.next();
                    int headVertex = edge.headVertex();
                    if (!this.found()[headVertex]) {
                        this.found()[headVertex] = true;
                        newEquivalanceClass.add(headVertex);
                        EquivalanceClasses.showDebugMessage(", " + headVertex);
                        this.sameClassMembers().push(headVertex);
                    }
                }
            }
            EquivalanceClasses.showDebugMessage(" }\n");
        }

    }
    return true;
}
```

Class EquivalanceClasses - Solve() Method

Equivalance Method의 solve 메소드는 동등 클래스를 찾아주는
알고리즘을 담당하는 부분으로, sameClassmembers 라는 arrayStack에
동등 클래스 원소들을 담았다가 하나씩 pop() 하면서
리스트인 newEquivalanceClass에 모두 넣어주고,
동등클래스 하나를 다 찾으면 다음 newEquivalanceClass를 같은 방법으로
찾아서 equivalanceClassList에 하나씩 add 해줍니다 .
따라서 각각의 equivalanceClass들은 equivalanceClassList에 저장되게 됩니다.

결과화면 1

```
<terminated> _Main_AL09_S1_201701975_구건모 (1) [Java Application]
<<< 동등 클래스 찾기 찾기 프로그램을 시작합니다 >>>
> 입력할 동등 관계의 원소의 수와 관계 쌍의 수를 먼저 입력
? 원소의 개수를 입력하시오: 6
? 관계쌍의 개수를 입력하시오: 3

> 이제부터 관계 쌍을 주어진 수 만큼 입력합니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 0
? 관계쌍의 두번째 원소를 입력하시오: 3
!새로운 관계 쌍 (0,3) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 1
? 관계쌍의 두번째 원소를 입력하시오: 2
!새로운 관계 쌍 (1,2) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 1
4? 관계쌍의 두번째 원소를 입력하시오:
!새로운 관계 쌍 (1,4) 가 그래프에 삽입되었습니다.

> 입력된 관계 쌍의 그래프는 다음과 같습니다:
[0] -> 3
[1] -> 4 2
[2] -> 1
[3] -> 0
[4] -> 1
[5] ->


[Debug] 새로운 동등 클래스: { 0, 3 }
[Debug] 새로운 동등 클래스: { 1, 4, 2 }
[Debug] 새로운 동등 클래스: { 5 }

> 찾아진 동등 클래스는 다음과 같습니다.
[동등 클래스  0]  { 5 }
[동등 클래스  1]  { 2 , 4 , 1 }
[동등 클래스  2]  { 3 , 0 }

<<< 동등 클래스 찾기 프로그램을 종료합니다 >>>
```



* 결과1

실습 자료에 예시로 나와있던 출력값과 동일하게 출력했을 때의 결과입니다.

우측의 그래프와 동일하게 동등클래스가 찾아지는 것을

볼수 있으며 이로써 정상적으로 프로그램이 동작하는것을 확인해 볼

수 있습니다.

결과화면 2

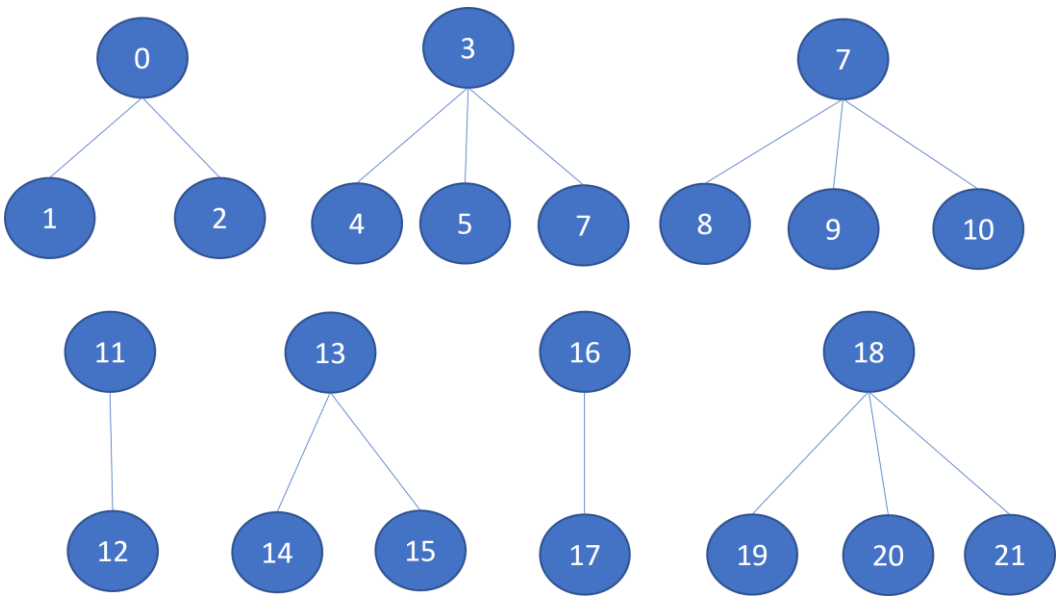
```
<terminated> _Main_AL09_S1_201701975_구건모 (1) [Java Applicatior
<<< 동등 클래스 찾기 찾기 프로그램을 시작합니다 >>>
> 입력할 동등 관계의 원소의 수와 관계 쌍의 수를 먼저 입
? 원소의 개수를 입력하시오: 5
? 관계쌍의 개수를 입력하시오: 0

> 이제부터 관계 쌍을 주어진 수 만큼 입력합니다.

> 입력된 관계 쌍의 그래프는 다음과 같습니다:
[0] ->
[1] ->
[2] ->
[3] ->
[4] ->


[Debug] 새로운 동등 클래스: { 0 }
[Debug] 새로운 동등 클래스: { 1 }
[Debug] 새로운 동등 클래스: { 2 }
[Debug] 새로운 동등 클래스: { 3 }
[Debug] 새로운 동등 클래스: { 4 }

> 찾아진 동등 클래스는 다음과 같습니다.
[동등 클래스  0]  { 4 }
[동등 클래스  1]  { 3 }
[동등 클래스  2]  { 2 }
[동등 클래스  3]  { 1 }
[동등 클래스  4]  { 0 }

<<< 동등 클래스 찾기 프로그램을 종료합니다 >>>
```

* 결과2 - 관계쌍의 개수를 0으로 설정할 경우

관계쌍의 개수를 0으로 설정하면 각각의 원소들이 동등클래스로 묶여있는 경우가 없는 것을 의미하므로 다음과 같이 자기자신만을 원소로 하는 동등클래스들이 원소의 개수만큼 나오게 됩니다.

결과화면 3

[결과3그래프]



*** 결과3 - 원소 22개, 관계쌍 15개**

결과3 에서는 원소의 수가 22개이고 관계쌍의 수가 15개인
그래프를 가지고 동등클래스를 찾아보았습니다.
추가한 그래프는 위와 같고, 결과 화면을 다음장에 첨부하였습니다.

결과화면 3

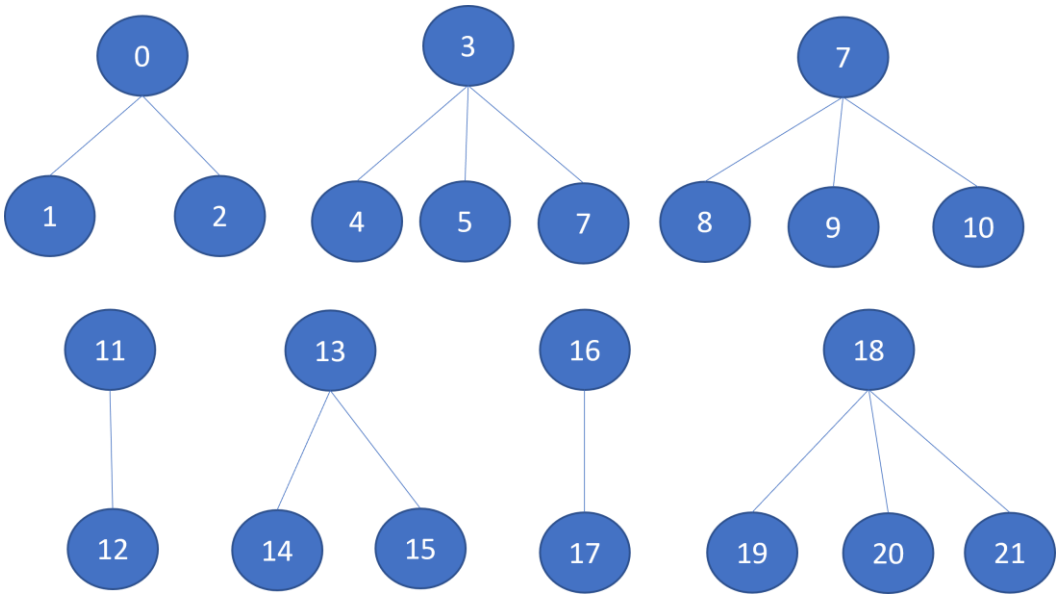
```
<terminated> _Main_AL09_S1_201701975_구건모 (1) [Java Application] C:₩Users₩
<<< 동등 클래스 찾기 찾기 프로그램을 시작합니다 >>>
> 입력할 동등 관계의 원소의 수와 관계 쌍의 수를 먼저 입력해야 합니
? 원소의 개수를 입력하시오: 22
? 관계쌍의 개수를 입력하시오: 15

> 이제부터 관계 쌍을 주어진 수 만큼 입력합니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 0
? 관계쌍의 두번째 원소를 입력하시오: 1
!새로운 관계 쌍 (0,1) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 0
? 관계쌍의 두번째 원소를 입력하시오: 2
!새로운 관계 쌍 (0,2) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 3
? 관계쌍의 두번째 원소를 입력하시오: 4
!새로운 관계 쌍 (3,4) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 3
? 관계쌍의 두번째 원소를 입력하시오: 5
!새로운 관계 쌍 (3,5) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 3
? 관계쌍의 두번째 원소를 입력하시오: 6
!새로운 관계 쌍 (3,6) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 7
? 관계쌍의 두번째 원소를 입력하시오: 8
!새로운 관계 쌍 (7,8) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 7
? 관계쌍의 두번째 원소를 입력하시오: 9
!새로운 관계 쌍 (7,9) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 7
? 관계쌍의 두번째 원소를 입력하시오: 10
!새로운 관계 쌍 (7,10) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 11
? 관계쌍의 두번째 원소를 입력하시오: 12
!새로운 관계 쌍 (11,12) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 13
? 관계쌍의 두번째 원소를 입력하시오: 14
!새로운 관계 쌍 (13,14) 가 그래프에 삽입되었습니다.
- 입력할 관계 쌍의 두 원소를 차례로 입력해야 합니다.
? 관계쌍의 첫번째 원소를 입력하시오: 13
```

*** 결과3 - 원소 22개, 관계쌍 15개**

```
> 입력된 관계 쌍의 그래프는 다음과 같습니다:
[0] -> 2 1
[1] -> 0
[2] -> 0
[3] -> 6 5 4
[4] -> 3
[5] -> 3
[6] -> 3
[7] -> 10 9 8
[8] -> 7
[9] -> 7
[10] -> 7
[11] -> 12
[12] -> 11
[13] -> 15 14
[14] -> 13
[15] -> 13
[16] -> 17
[17] -> 16
[18] -> 21 20 19
[19] -> 18
[20] -> 18
[21] -> 18


[Debug] 새로운 동등 클래스: { 0, 2, 1 }
[Debug] 새로운 동등 클래스: { 3, 6, 5, 4 }
[Debug] 새로운 동등 클래스: { 7, 10, 9, 8 }
[Debug] 새로운 동등 클래스: { 11, 12 }
[Debug] 새로운 동등 클래스: { 13, 15, 14 }
[Debug] 새로운 동등 클래스: { 16, 17 }
[Debug] 새로운 동등 클래스: { 18, 21, 20, 19 }

> 찾아진 동등 클래스는 다음과 같습니다.
[동등 클래스  0]  {19 ,20 ,21 ,18 }
[동등 클래스  1]  {17 ,16 }
[동등 클래스  2]  {14 ,15 ,13 }
[동등 클래스  3]  {12 ,11 }
[동등 클래스  4]  { 8 , 9 ,10 , 7 }
[동등 클래스  5]  { 4 , 5 , 6 , 3 }
[동등 클래스  6]  { 1 , 2 , 0 }

<<< 동등 클래스 찾기 프로그램을 종료합니다 >>>
```

그래프 edge, vertex 추가 과정이 길어
일부 생략하였습니다.

결과화면 3




```
[Debug] 새로운 동등 클래스: { 0, 2, 1 }
[Debug] 새로운 동등 클래스: { 3, 6, 5, 4 }
[Debug] 새로운 동등 클래스: { 7, 10, 9, 8 }
[Debug] 새로운 동등 클래스: { 11, 12 }
[Debug] 새로운 동등 클래스: { 13, 15, 14 }
[Debug] 새로운 동등 클래스: { 16, 17 }
[Debug] 새로운 동등 클래스: { 18, 21, 20, 19 }

> 찾아진 동등 클래스는 다음과 같습니다.
[동등 클래스  0]  {19 ,20 ,21 ,18 }
[동등 클래스  1]  {17 ,16 }
[동등 클래스  2]  {14 ,15 ,13 }
[동등 클래스  3]  {12 ,11 }
[동등 클래스  4]  { 8 , 9 ,10 , 7 }
[동등 클래스  5]  { 4 , 5 , 6 , 3 }
[동등 클래스  6]  { 1 , 2 , 0 }

<<< 동등 클래스 찾기 프로그램을 종료합니다 >>>
```

*** 결과3 - 원소 22개, 관계쌍 15개**

추가된 관계쌍들과

그에 따라 찾아진 동등클래스가

그래프의 상태와 동일하게 찾아지는 것을

확인해볼 수 있습니다.

생각할 점

> 각 Class의 사용자 관점에서, 사용자의 코드가 구현에 영향을 받지 않도록 class가 잘 설계되었는지?

이번 과제에서도 Dirceted Graph 구현이 Matrix와 List 두가지로 구현하였는데, graph 구현에 대한 자료구조가 변경되어도 내부의 값을 순환할 때에 각각의 Iterator를 통해 순회하기 때문에 기존의 다른 주요 구현들은 변경될 필요가 없습니다. 따라서 코드가 구현에 영향을 받지 않도록 잘 설계되었다고 생각합니다.

> Class 와 Interface 는 적절하게 구분되어 사용되고 있는지?

Interface들이 전반적으로 공통된 기능들을 명시하고 있고 그에 따라 필요한 Class 에서 implement하여 구현되고 있는것은 적절하게 사용되고 있다고 생각합니다. ArrayStack을 구현할 때 Stack interface를 이용하여 명시되어 있는 method 들을 overriding 하여서 구현하였고, UndirectedAdjacencyListGraph를 구현할 때에도 기존에 구현하였던 DirectedAdjcencyListGraph의 기능과 addEdge 부분을 제외하고는 동일하기 때문에 상속받아 구현됩니다.
따라서 적절하게 구분되어 사용되고 있다고 생각합니다.

> 스택을 선언할 때 그 자료형을 interface Stack<> 으로 하는 이유

Stack interface를 generic 사용하여 만들경우 이후에 사용되어질 자료형을 미리 정하지 않으므로써 사용자가 구현시 사용할 자료형에 제한을 받지 않는다는 장점이 있습니다.

- private ArrayStack<Integer> _sameClassMembers 이렇게 하면 어떤 단점이?

만약 위와같이 Stack이 아닌 ArrayStack 으로 선언할 경우에는 Stack의 구현에 사용되는 자료구조가 바뀌는 경우에 종속적으로 변경해주어야 한다고 생각하여 해당 부분이 단점이라고 생각합니다.

생각할 점

> 그래프에 사이클이 있으면 문제가 되는지?

현재 프로그램에서는 동등 클래스를 찾는 과정에서 loop을 그래프의 원소 개수 만큼만 수행하기 때문에 사이클이 생겨도 문제가 되지 않을 것이라고 생각합니다.

> 그래프에 Edge가 하나도 없으면 어떻게 되는지?

그래프에 Edge가 없다는 것은 관계쌍이 없다는 것으로, 각각의 원소들이 서로 동등한 관계를 가지는 경우가 없다는 것을 의미합니다. 따라서 프로그램 상에서는 그래프의 원소 개수만큼 하나의 원소만 가지고 있는 동등클래스가 만들어지게 됩니다.

> 시간 복잡도와 공간 복잡도

*** 시간 복잡도:** 시간 복잡도는 알고리즘에서의 수행시간을 분석한 지표로, 실질적으로 연산횟수로 생각하면 직관적이며 최악의 경우를 계산합니다.
*** 공간 복잡도:** 공간 복잡도는 알고리즘에서 메모리 사용량에 대한 지표입니다.

사용자가 graph에 추가한 vertex 수가 n개 그리고 edge 수가 e개라고 가정했을 때, ListGraph를 사용하였을 때와 MatrixGraph를 사용하였을 때의 복잡도가 다르게 산출되게 되는데,
ListGraph의 경우 n개의 Vertex를 가지는 그래프는 n개의 공간만 할당하고 이에 대해서만 순회하지만, MatrixGraph를 사용할 경우 더 많은 공간을 할당하여 사용하기 때문에 n개의 vertex를 가지는 그래프의 경우 n^2의 공간복잡도와 이에 따라서 순회하는 데에도 제곱의 시간이 소요됩니다. 따라서 ListGraph를 이용한 것이 MatrixGraph를 이용한 것보다 복잡도가 낮게 나오게 됩니다.