알고리즘 과제 보고서

- AL08, 위상정렬 과제

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

과제에서 해야할 일과 주요이론

이번 과제의 이론인 위상정렬은 DirectedGraph에서 Vertex와 Vertex 사이의 위상, 즉 Edge가 연결된 방향성을 잃지 않고 정렬을 수행하는 것으로, predecessor를 가지지 않는 vertex 들을 stack에 추가하여 하나씩 pop 하면서 pop 결과로 꺼내진 vertex를 graph에서 제거한 후, 다시 graph에서 predecessor가 없는 vertex들을 stack에 push 하고 이 과정을 모든 vertex 들이 정렬될 때까지 반복합니다. 위상정렬은 그래프에 cycle이 존재하지 않는 acycle graph에 대해서만 수행할 수 있습니다. 이번 과제에서는 방향성을 가지는 Directed Graph를 사용하고, 실질적인 알고리즘을 담당하는 TopologicalSort에서 알고리즘 수행과정에서 그래프에 존재하는 각각의 Vertex 들의 선행자의 개수를 판별하거나 선행자가 없는 vertex들을 찾기 위한 method를 구현 및 사용하여 결과적으로 solve method 에서 실질적인 TopologicalSort 기능을 하도록 구현되게 됩니다.

주요 구현 내용

```java
package list;

public class ArrayList<T> extends List<T> {
    private static final int DEFAULT_CAPACITY = 10;

    private int _capacity;
    private int _size;
    private T[] _elements;

    public ArrayList() {
        this(ArrayList.DEFAULT_CAPACITY);
    }

    @SuppressWarnings("unchecked")
    public ArrayList(int givenCapacity) {
        this.setCapacity(givenCapacity);
        this.setElements((T[]) new Object[this.capactiy()]);
        this.setSize(0);
    }

    private int capactiy() {
        return this._capacity;

    }

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

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

    private void setSize(int newSize) {
        this._size = newSize;
```

*코드의 일부입니다.

Class ArrayList

위상정렬의 결과를 저장하기 위해 구현한 arrayList 입니다.

TopologicalSort 클래스의 solve 메소드에서 stack에서 pop한 vertex를 add ArrayList 타입 변수에 add 메서드를 통해 담아서 결과를 저장합니다.

주요 구현 내용

```
private class IteratorForArrayList implements Iterator<T> {
    private int _nextPosition;

    private int nextPosition() {
        return this._nextPosition;
    }

    private void setNextPosition(int newPosition) {
        this._nextPosition = newPosition;

    }

    private IteratorForArrayList() {
        this.setNextPosition(0);
    }

    public boolean hasNext() {
        return (this.nextPosition() < ArrayList.this.size());

    }

    public T next() {
        T nextElement = ArrayList.this.elements()[this.nextPosition()];
        this.setNextPosition(this.nextPosition() + 1);
        return nextElement;
    }

}
```

*코드의 일부입니다.

Class IteratorForArrayList

ArrayList 를 순회하기 위한 Iterator로 Class ArrayList의 inner class로 정의되어 있습니다. iterator의 기본적인 구현을 명시해놓은 interface Iterator 를 Implement 하여 구현되며 ,ArrayList 값을 출력할 때 IteratorForArrayList 인스턴스를 생성하여 ArrayList를 순회하고 값을 출력하게 됩니다. AppController에서 TopologicalSort의 결과가 저장된 ArrayList를 출력할 때 사용합니다.

주요 구현 내용

```java
package list;

public class LinkedStackWithIterator<E> implements StackWithIterator<E> {
    private LinkedNode<E> _top;
    private int _size;

    public LinkedStackWithIterator() {
        this.reset();
    }

    private LinkedNode<E> top() {
        return this._top;
    }

    private void setTop(LinkedNode<E> newTop) {
        this._top = newTop;
    }

    @Override
    public void reset() {
        // TODO Auto-generated method stub
        this.setSize(0);
        this.setTop(null);
    }

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

    private void setSize(int newSize) {
        this._size = newSize;
    }

    @Override
    public boolean isEmpty() {
```

*코드의 일부입니다.

Class LinkedStackWithIterator

LinkedStackWithIterator 클래스는 위상정렬에 사용될 stack으로 interface StackWithIterator 를 implement 하여 구현되어 있습니다. stack의 기본적인 메서드들과 inner class로 iterator가 구현되어 있습니다. 위상정렬시 선행자가 없는 vertex 들을 push 하거나, stack의 top을 pop 하여 정렬 결과를 저장하는 arrayList에 저장할 때 해당 stack이 사용되게 됩니다.

주요 구현 내용

```
private class IteratorForLinkedStack implements Iterator<E> {
    private LinkedNode<E> _nextNode;

    private LinkedNode<E> nextNode() {
        return this._nextNode;

    }

    private void setNextNode(LinkedNode<E> newNextNode) {
        this._nextNode = newNextNode;
    }

    private IteratorForLinkedStack() {
        this.setNextNode(LinkedStackWithIterator.this.top());

    }

    public boolean hasNext() {
        return (this.nextNode() != null);
    }

    public E next() {
        E nextElement = this.nextNode().element();
        this.setNextNode(this.nextNode().next());
        return nextElement;
    }
}
```

*코드의 일부입니다.

Class IteratorForLinkedStack

Class IteratorForLinkedStack는 LinkedStack을 순회하기 위한 iterator로 Class LinkedStackWithIterator의 inner class로 구현됩니다.

Interface Iterator를 implement 하여 구현하며, 다음 노드값을 반환하는 next, 다음 노드가 있는지 확인하는 hasNext 메서드가 구현되어 있습니다.

위상정렬시에 stack의 값을 출력할 때 해당 iterator를 이용하여 stack을 순회하며 출력합니다.

주요 구현 내용

```java
package graph;

import list.LinkedList;

public class DirectedAdjacencyListGraph<E extends Edge> extends AdjacencyGraph<E> {
    private LinkedList<E>[] _adjacency;

    protected LinkedList<E>[] adjacency() {
        return this._adjacency;

    }

    protected void setAdjacecny(LinkedList<E>[] newAdjacency) {
        this._adjacency = newAdjacency;
    }

    protected LinkedList<E> neighborListOf(int aTailVertex) {
        return this.adjacency()[aTailVertex];

    }

    protected int adjacencyOfEdge(int aTailVertex, int aHeadVertex) {
        if (this.vertexDoesExist(aTailVertex) && this.vertexDoesExist(aHeadVertex)) {
            Iterator<E> iterator = this.neighborIteratorOf(aTailVertex);
            while (iterator.hasNext()) {
                E neighborEdge = iterator.next();
                if (aHeadVertex == neighborEdge.headVertex()) {
                    return AdjacencyGraph.EDGE_EXIST;
                }
            }
        }
        return AdjacencyGraph.EDGE_NONE;
    }

    @SuppressWarnings("unchecked")
    public DirectedAdjacencyListGraph(int givenNumberOfVertices) {
        this.setNumberOfVertices(givenNumberOfVertices);
```

Class DirectedAdjacencyListGraph

위상정렬에 사용되는 그래프인 Directed Adjacency Graph를 List를 이용하여 구현한 DirectedAdjacencyGraph 입니다. LinkedList 인스턴스인 adjacency를 멤버변수로 가지고 해당 LinkedList 를 이용하여 graph 를 저장합니다.

주요 구현 내용

```java
package graph;

import list.Iterator;

public class DirectedAdjacencyMatrixGraph<E extends Edge> extends AdjacencyGraph<E> {
    protected int[][] _adjacency;

    protected DirectedAdjacencyMatrixGraph() {

    }

    public DirectedAdjacencyMatrixGraph(int givenNumberOfVertices) {
        this.setNumberOfVertices(givenNumberOfVertices);
        this.setNumberOfEdges(0);
        this.setAdjacency(new int[givenNumberOfVertices][givenNumberOfVertices]);
        for (int tailVertex = 0; tailVertex < this.numberOfVertices(); tailVertex++) {
            for (int headVertex = 0; headVertex < this.numberOfVertices(); headVertex++) {
                this.setAdjacencyOfEdgeAsNone(tailVertex, headVertex);
            }
        }
    }

    public boolean adjacencyOfEdgeDoesExist(int tailVertex, int headVertex) {
        return (this.adjacencyOfEdge(tailVertex, headVertex) != AdjacencyGraph.EDGE_NONE);
    }

    protected int[][] adjacency() {
        return this._adjacency;
    }

    protected void setAdjacency(int[][] newAdjacency) {
        this._adjacency = newAdjacency;
    }

    protected int adjacencyOfEdge(int tailVertex, int headVertex) {
        return this.adjacency()[tailVertex][headVertex];
    }
```

*코드의 일부입니다.

Class DirectedAdjacencyMatrixGraph

위상정렬에 사용되는 그래프인 Directed Adjacency Graph를 Matrix를 이용하여 구현한 DirectedAdjacencyMatrixGraph 입니다.

2차원 배열인 adjacency를 멤버변수로 가지고 해당 Matrix를 이용하여 graph 를 저장합니다.

주요 구현 내용

```
private void initPredecessorCounts() {
    this.setPredecessorCounts(new int[this.graph().numberOfVertices()]);
    for (int tailVertex = 0; tailVertex < this.graph().numberOfVertices(); tailVertex++) {
        this.predecessorCounts()[tailVertex] = 0;
    }
    for (int tailVertex = 0; tailVertex < this.graph().numberOfVertices(); tailVertex++) {
        Iterator<E> iterator = this.graph().neighborIteratorOf(tailVertex);
        while (iterator.hasNext()) {
            E edge = iterator.next();
            this.predecessorCounts()[edge.headVertex()]++;
        }

    }
    TopologicalSort.showDebugMessage("\n[Debug] 각 Vertex의 초기 선행자 수는 다음과 같습니다:\n-->");
    for (int vertex = 0; vertex < this.graph().numberOfVertices(); vertex++) {
        TopologicalSort.showDebugMessage(" [" + vertex + "]=" + this.predecessorCounts()[vertex]);

    }
    TopologicalSort.showDebugMessage("\n");
}

private void initZeroCountVertices() {
    this.setZerocountVertices(new LinkedStackWithIterator<Integer>());
    TopologicalSort.showDebugMessage("\n[Debug] 그래프에 선행자가 없는 vertex들은 다음과 같습니다:\n--> ( ");
    for (int vertex = 0; vertex < this.graph().numberOfVertices(); vertex++) {
        if (this.predecessorCounts()[vertex] == 0) {
            this.zeroCountVertices().push(vertex);
            TopologicalSort.showDebugMessage(vertex + " ");

        }
    }
    TopologicalSort.showDebugMessage(")\n");
}
```

*코드의 일부입니다.

TopologicalSort Class - initPredecessorCounts, initZeroCountVertices

initPredecessorCounts method 는 그래프에서 각 Vertex 들이 가지는 선행자의 개수를 Iterator를 이용하여 순회하면서 initialize 해줍니다.
위상정렬을 구현할 때 Vertex 중 선행자가 없는 Vertex 들을 먼저 Stack 추가하는데, initZeroCountVertices method 에서 선행자가 없는 Vertex들을 찾아 zeroCountVertices로 push 해줍니다.
initZeroCountVertices method 는 predecessorCounts 값들 중 0 값을 가지는 Vertex 인 경우 zeroCountVertex로 인식하기 때문에, 사용할 때 우선적으로 initPredecessorCounts를 통해 predecessorCount를 초기화 후 사용하여야 합니다.

주요 구현 내용

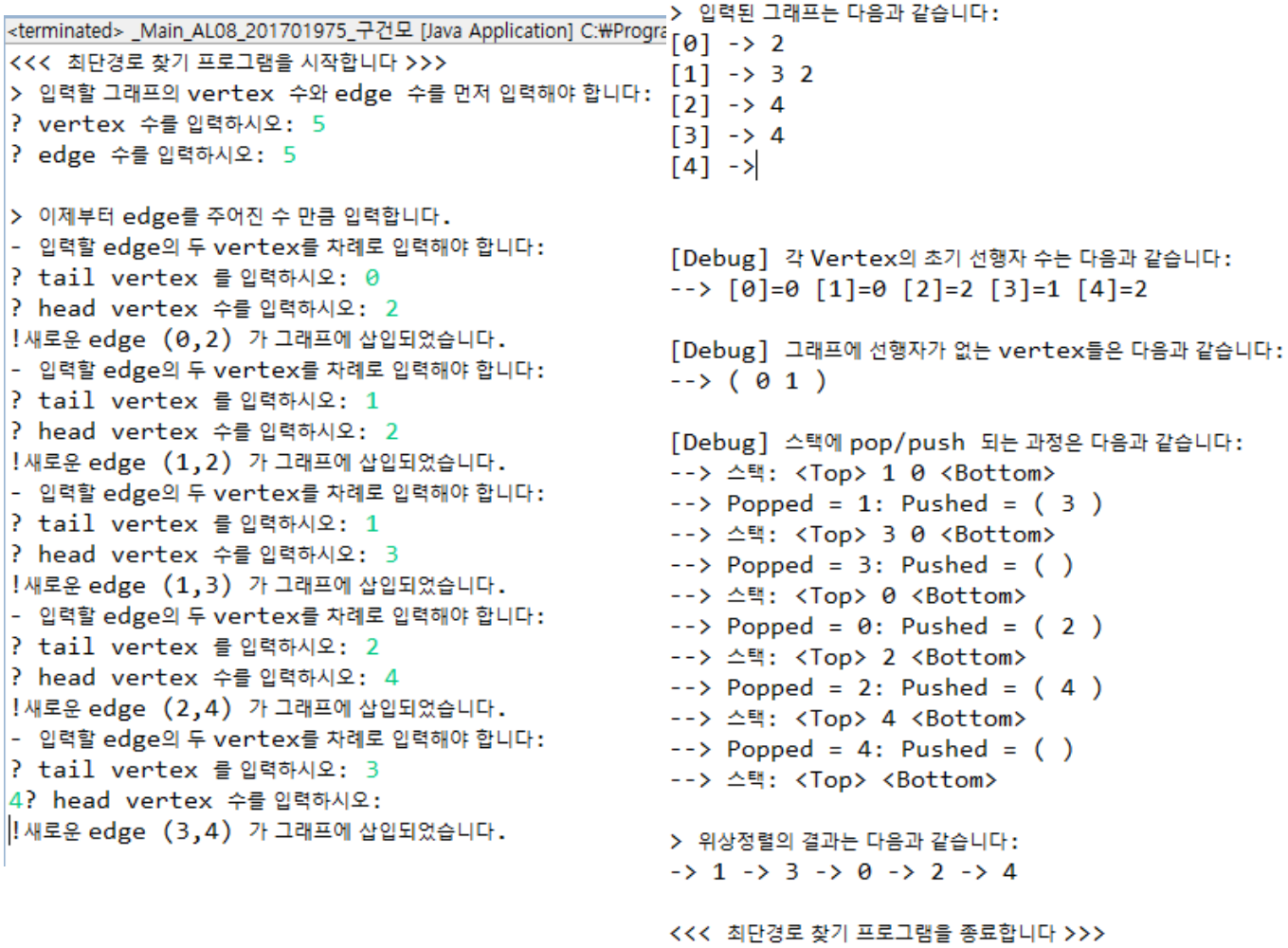
```
public boolean solve(AdjacencyGraph<E> aGraph) {
    this.setGraph(aGraph);
    this.initPredecessorCounts();
    this.initZeroCountVertices();
    this.setSortedList(new ArrayList<Integer>(this.graph().numberOfVertices()));

    TopologicalSort.showDebugMessage("\n[Debug] 스택에 pop/push 되는 과정은 다음과 같습니다:\n");
    this.showZeroCountVertices();
    while (!this.zeroCountVertices().isEmpty()) {
        int tailVertex = this.zeroCountVertices().pop();
        TopologicalSort.showDebugMessage("--> Popped = " + tailVertex + ": Pushed = ( ");
        this.sortedList().add(tailVertex);
        Iterator<E> iterator = this.graph().neighborIteratorOf(tailVertex);
        while (iterator.hasNext()) {
            E edge = iterator.next();
            --this.predecessorCounts()[edge.headVertex()];
            if (this.predecessorCounts()[edge.headVertex()] == 0) {
                this.zeroCountVertices().push(edge.headVertex());
                TopologicalSort.showDebugMessage(edge.headVertex() + " ");
            }
        }
        TopologicalSort.showDebugMessage(")\n");
        this.showZeroCountVertices();
    }
    return (this.sortedList().size() == this.graph().numberOfVertices());
}
```

*코드의 일부입니다.

Class TopologicalSort – solve()

solve method 는 실질적으로 위상정렬을 수행하는 알고리즘이 구현된 메소드로, 앞서 구현한 init method 들을 이용해 선행자 개수와 선행자가 없는 vertex를 초기화하고 초기화된 값들을 이용하여 Iteration 하면서 정렬하게 됩니다.

결과화면 1



```
<terminated> _Main_AL08_201701975_구건모 [Java Application] C:₩Progra
<<< 최단경로 찾기 프로그램을 시작합니다 >>>
> 입력할 그래프의 vertex 수와 edge 수를 먼저 입력해야 합니다:
? vertex 수를 입력하시오: 5
? edge 수를 입력하시오: 5

> 이제부터 edge를 주어진 수 만큼 입력합니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 2
!새로운 edge (0,2) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 1
? head vertex 수를 입력하시오: 2
!새로운 edge (1,2) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 1
? head vertex 수를 입력하시오: 3
!새로운 edge (1,3) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 2
? head vertex 수를 입력하시오: 4
!새로운 edge (2,4) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 3
4? head vertex 수를 입력하시오:
!새로운 edge (3,4) 가 그래프에 삽입되었습니다.

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

[Debug] 각 Vertex의 초기 선행자 수는 다음과 같습니다:
--> [0]=0 [1]=0 [2]=2 [3]=1 [4]=2

[Debug] 그래프에 선행자가 없는 vertex들은 다음과 같습니다:
--> ( 0 1 )

[Debug] 스택에 pop/push 되는 과정은 다음과 같습니다:
--> 스택: <Top> 1 0 <Bottom>
--> Popped = 1: Pushed = ( 3 )
--> 스택: <Top> 3 0 <Bottom>
--> Popped = 3: Pushed = ( )
--> 스택: <Top> 0 <Bottom>
--> Popped = 0: Pushed = ( 2 )
--> 스택: <Top> 2 <Bottom>
--> Popped = 2: Pushed = ( 4 )
--> 스택: <Top> 4 <Bottom>
--> Popped = 4: Pushed = ( )
--> 스택: <Top> <Bottom>

> 위상정렬의 결과는 다음과 같습니다:
-> 1 -> 3 -> 0 -> 2 -> 4

<<< 최단경로 찾기 프로그램을 종료합니다 >>>
```

*** 결과1**

vertex 5개와 edge 5개에 대한 실행결과 입니다.

최초에 선행자가 없는 vertex 들이 stack에 들어가고 stack에서 하나씩 pop 하면서 pop된 vertex를 graph에서 제거한 상태에서의 predecessor가 없는 vertex를 다시 stack에 push 합니다. 이 과정을 모든 vertex들이 위상정렬 될때까지 반복하게 됩니다. 위상정렬의 결과를 확인해보면 각 vertex들의 위상이 유지된 상태로 정렬된 것을 볼 수 있습니다.

결과화면 2

```
<terminated> _Main_AL08_201701975_구건모 [Java Application] C:₩Program F
<<< 최단경로 찾기 프로그램을 시작합니다 >>>
> 입력할 그래프의 vertex 수와 edge 수를 먼저 입력해야 합니다:
? vertex 수를 입력하시오: 11
? edge 수를 입력하시오: 18

> 이제부터 edge를 주어진 수 만큼 입력합니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 3
!새로운 edge (0,3) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 4
!새로운 edge (0,4) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 5
!새로운 edge (0,5) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 1
? head vertex 수를 입력하시오: 4
!새로운 edge (1,4) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 1
? head vertex 수를 입력하시오: 5
!새로운 edge (1,5) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 1
? head vertex 수를 입력하시오: 6
!새로운 edge (1,6) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 1
? head vertex 수를 입력하시오: 7
!새로운 edge (1,7) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 2
? head vertex 수를 입력하시오: 5
!새로운 edge (2,5) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 2
? head vertex 수를 입력하시오: 6
!새로운 edge (2,6) 가 그래프에 삽입되었습니다.
```

결과 2에서는

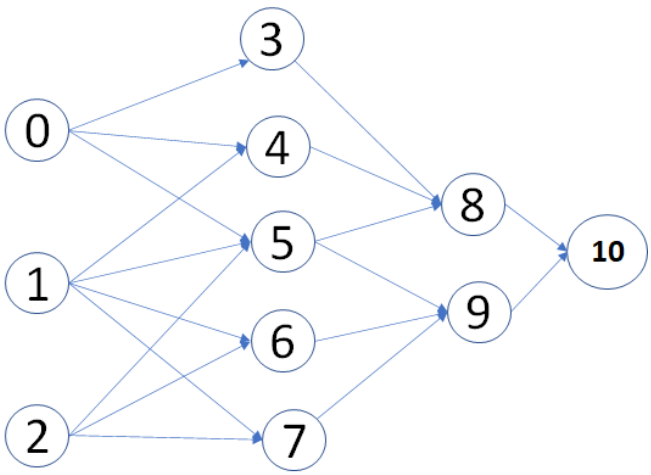
vertex 11 개, edge 18 개인

DAG를 만들어 사용하였습니다.

추가한 그래프를 도식화 해보면

아래와 같습니다.

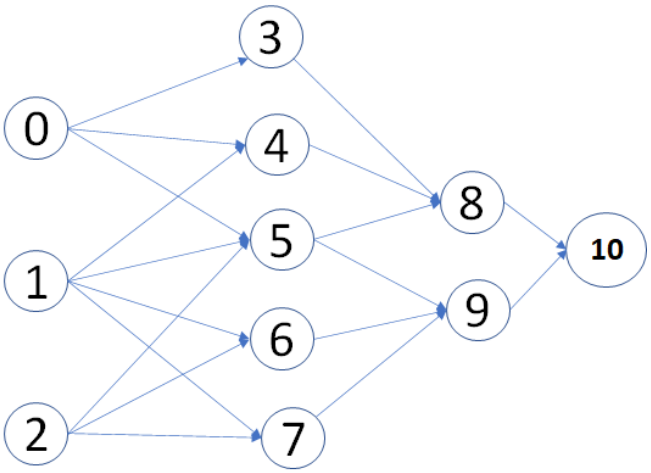
[결과2 그래프]



그래프 edge,vertex 추가 과정이 길어 일부 생략하였고,
다음장에 결과를 첨부하였습니다.

결과화면 2

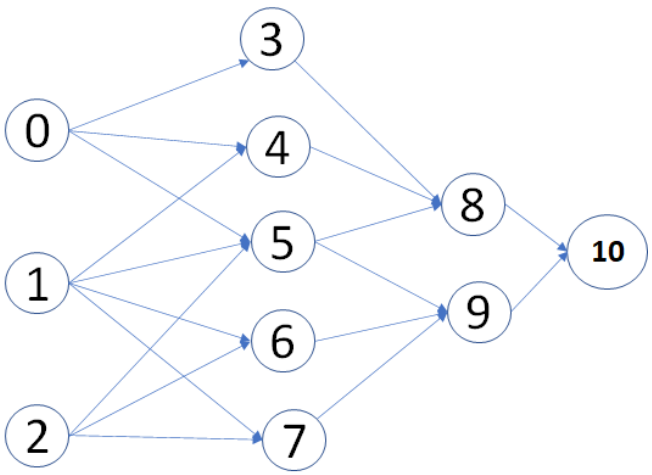
[결과2 그래프]




```
> 입력된 그래프는 다음과 같습니다:
[0] -> 5 4 3
[1] -> 7 6 5 4
[2] -> 7 6 5
[3] -> 8
[4] -> 8
[5] -> 9 8
[6] -> 9
[7] -> 9
[8] -> 10
[9] -> 10
[10] ->

[Debug] 각 Vertex의 초기 선행자 수는 다음과 같습니다:
--> [0]=0 [1]=0 [2]=0 [3]=1 [4]=2 [5]=3 [6]=2 [7]=2 [8]=3 [9]=3 [10]=2

[Debug] 그래프에 선행자가 없는 vertex들은 다음과 같습니다:
--> ( 0 1 2 )

[Debug] 스택에 pop/push 되는 과정은 다음과 같습니다:
--> 스택: <Top> 2 1 0 <Bottom>
--> Popped = 2: Pushed = ( )
--> 스택: <Top> 1 0 <Bottom>
--> Popped = 1: Pushed = ( 7 6 )
--> 스택: <Top> 6 7 0 <Bottom>
--> Popped = 6: Pushed = ( )
--> 스택: <Top> 7 0 <Bottom>
--> Popped = 7: Pushed = ( )
--> 스택: <Top> 0 <Bottom>
--> Popped = 0: Pushed = ( 5 4 3 )
--> 스택: <Top> 3 4 5 <Bottom>
--> Popped = 3: Pushed = ( )
--> 스택: <Top> 4 5 <Bottom>
--> Popped = 4: Pushed = ( )
--> 스택: <Top> 5 <Bottom>
--> Popped = 5: Pushed = ( 9 8 )
--> 스택: <Top> 8 9 <Bottom>
--> Popped = 8: Pushed = ( )
--> 스택: <Top> 9 <Bottom>
--> Popped = 9: Pushed = ( 10 )
--> 스택: <Top> 10 <Bottom>
--> Popped = 10: Pushed = ( )
--> 스택: <Top> <Bottom>

> 위상정렬의 결과는 다음과 같습니다:
-> 2 -> 1 -> 6 -> 7 -> 0 -> 3 -> 4 -> 5 -> 8 -> 9 -> 10

<<< 최단경로 찾기 프로그램을 종료합니다 >>>
```

결과화면 2

```
[Debug] 각 Vertex의 초기 선행자 수는 다음과 같습니다:
--> [0]=0 [1]=0 [2]=0 [3]=1 [4]=2 [5]=3 [6]=2 [7]=2 [8]=3 [9]=3 [10]=2

[Debug] 그래프에 선행자가 없는 vertex들은 다음과 같습니다:
--> ( 0 1 2 )

[Debug] 스택에 pop/push 되는 과정은 다음과 같습니다:
--> 스택: <Top> 2 1 0 <Bottom>
--> Popped = 2: Pushed = ( )
--> 스택: <Top> 1 0 <Bottom>
--> Popped = 1: Pushed = ( 7 6 )
--> 스택: <Top> 6 7 0 <Bottom>
--> Popped = 6: Pushed = ( )
--> 스택: <Top> 7 0 <Bottom>
--> Popped = 7: Pushed = ( )
--> 스택: <Top> 0 <Bottom>
--> Popped = 0: Pushed = ( 5 4 3 )
--> 스택: <Top> 3 4 5 <Bottom>
--> Popped = 3: Pushed = ( )
--> 스택: <Top> 4 5 <Bottom>
--> Popped = 4: Pushed = ( )
--> 스택: <Top> 5 <Bottom>
--> Popped = 5: Pushed = ( 9 8 )
--> 스택: <Top> 8 9 <Bottom>
--> Popped = 8: Pushed = ( )
--> 스택: <Top> 9 <Bottom>
--> Popped = 9: Pushed = ( 10 )
--> 스택: <Top> 10 <Bottom>
--> Popped = 10: Pushed = ( )
--> 스택: <Top> <Bottom>

> 위상정렬의 결과는 다음과 같습니다:
-> 2 -> 1 -> 6 -> 7 -> 0 -> 3 -> 4 -> 5 -> 8 -> 9 -> 10

<<< 최단경로 찾기 프로그램을 종료합니다 >>>
```



*** 결과2 (vertex 11개, edge 18개)**

결과2의 그래프를 살펴보면 최초에 선행자가 없는 vertex인 0,1,2 가 stack에 추가되어 시작되며 하나씩 pop 하여 그래프에서 제거되었다고 생각했을 때의 선행자가 없는 vertex들을 stack에 push 하는데 최초에 스택의 top인 2를 pop 하면 6,7이 선행자가 없는 vertex가 되므로 스택에 추가됩니다.

다시 스택에서 top인 6을 pop 했을 때 추가적으로 선행자가 없어지는 vertex가 없으므로 아무것도 stack에 push 하지 않고, top인 7을 pop 합니다. 7 또한 stack에 push되는 값이 없고, 다음으로 0이 pop 됩니다. 0이 pop되면서 3,4,5가 선행자가 없는 vertex가 되므로 모두 push 합니다. 이 과정을 모든 vertex들이 정렬될 때까지

반복하게 되는데 아래 위상정렬의 결과를 확인해 보면 vertex들의 위상이 유지된 채로 정렬된 것을 볼 수 있습니다.

생각할 점

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

Dirceted Graph 구현이 Matrix와 List 두가지로 구현하였는데, graph 구현에 대한 자료구조가 변경되어도 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 Class에서 implement하여 구현되고 있는것은 적절하게 사용되고 있다고 생각합니다. iterator 인터페이스의 경우로 예를 들면 각각의 다른 자료구조들에 대해서 순회하는 구현은 다르지만 구현해야하는 기능적인 요소들은 동일합니다. 따라서 iterator 라는 interface를 생성하여 공통적으로 가지는 기능들을 명시하면 이후에 iterator를 사용해야 하는 자료구조를 구현할 때에 편리하게 implement 하여 구현할 수 있습니다.

> 그래프에 사이클이 있으면 프로그램에서의 대처 방법은?

사이클이 존재하는 그래프에 대해서 위상정렬을 할 경우 모든 vertex가 아직 정렬되지 않았는데 stack이 비어있는 상황이 발생합니다. 따라서 stack에 더 이상 값이 없는데 정렬되지 않는 vertex가 존재할 경우 cycle이 있는 그래프임을 의미하므로 위상정렬을 할 수 없다는 메시지를 출력하여 대처할 수 있습니다. 또는 그래프를 입력받을 때부터 사용자가 추가하려는 edge가 cycle을 발생시키는지에 대한 여부를 체크하도록하여 사이클이 있는 그래프가 추가되는것을 방지하는 것도 하나의 방법이 될 수 있습니다.

생각할 점

> 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.

```
public boolean solve(AdjacencyGraph<E> aGraph) {
    this.setGraph(aGraph);
    this.initPredecessorCounts();
    this.initZeroCountVertices();
    this.setSortedList(new ArrayList<Integer>(this.graph().numberOfVertices()));

    TopologicalSort.showDebugMessage("\n[Debug] 스택에 pop/push 되는 과정은 다음과 같습니다:\n");
    this.showZeroCountVertices();
    while (!this.zeroCountVertices().isEmpty()) {
        int tailVertex = this.zeroCountVertices().pop();
        TopologicalSort.showDebugMessage("--> Popped = " + tailVertex + ": Pushed = ( ");
        this.sortedList().add(tailVertex);
        Iterator<E> iterator = this.graph().neighborIteratorOf(tailVertex);
        while (iterator.hasNext()) {
            E edge = iterator.next();
            --this.predecessorCounts()[edge.headVertex()];
            if (this.predecessorCounts()[edge.headVertex()] == 0) {
                this.zeroCountVertices().push(edge.headVertex());
                TopologicalSort.showDebugMessage(edge.headVertex() + " ");
            }
        }
        TopologicalSort.showDebugMessage(")\n");
        this.showZeroCountVertices();
    }
    return (this.sortedList().size() == this.graph().numberOfVertices());
}
```

사용자가 graph에 추가한 vertex 수가 n개 그리고 edge 수가 e개라고 가정했을 때,
solve method 의 복잡도를 구해보면 내부에서 init 과정에서 연산량이 가장 많이 발생하므로 해당 과정에 대한 복잡도를 구해보면 되는데, graph를 다루므로 graph를 구현한 자료구조에 따라 연산량이 달라지게 됩니다.

- DirectedAdjacencyListGraph 사용

만약 DircetedAdjacencyListGraph를 사용할 경우 initProdecessorCounts에서 배열을 초기화하는데
n 그리고 edge를 이용해서 count 배열을 초기화하는데 e 만큼의 연산을 하므로 소요되므로
시간복잡도는 O(n+e) 를 가집니다.

- DirectedAdjacencyMatrixGraph 사용

DirectedAdjacencyMatrixGraph를 사용할 경우 공간복잡도는
그래프를 저장할 때 2차원 배열을 사용하므로 vertex의 개수가 n개라고 가정하면 matrix는 공간은 n^2 만큼
이 할당되므로 initProdecessorCounts에서 counts를 초기화 할때 n^2에 대해서 scan이 이루어집니다.
따라서 시간복잡도는 O(n^2)가 됩니다.