알고리즘 과제 보고서

- AL07, ShortestPath

충남대학교 컴퓨터공학과
분반: 알고리즘 04 분반
제출일: 2021-11-02
학번: 201701975
이름: 구건모

과제에서 해야할 일과 주요이론

이번 과제에 주요 개념인 Dijkstra 알고리즘을 이용해 최단경로를 구하는 것이 주요 내용 입니다.
Dijkstra 알고리즘은 시작 vertex로 부터 Edge의 weight의 합이 최소가 되는 경로를 찾는 알고리즘 입니다. 최초에는 시작 vertex 에서부터 시작하여 직접적으로 이어지는 Edge가 없을 경우 INIFINITE 값으로 나타내고 매 단계마다 현재까지 Found 하지 않은 Edge 중에서 가장 작은 weight 를 가진 Edge를 찾아 Vertex를 추가합니다. 또한, 매 단계가 진행되면서 Vertex들이 추가됨에 따라서 변화되는 distance를 기존의 distance 와 비교하여 더 작은 값을 distance 값으로 설정해 주게 됩니다. 이 과정을 반복하여 최단거리를 찾게되는 알고리즘 입니다. 이번과제에서는 다익스트라 알고리즘을 구현하기 위해 , WeightedDirectedAdjacencyListGraph를 이용하고,
최단 경로를 찾는 알고리즘을 구현한 클래스인 ShortestPaths 클래스를 구현하게 됩니다.
ShortestPaths 클래스에서는 위에서 언급한 다익스트라 알고리즘의 과정을 수행하는 method가 구현됩니다.

주요 구현 내용

*** Interface Stack**

```
package list;

public interface Stack<T> {
    public void reset();

    public int size();

    public boolean isEmpty();

    public boolean isFull();

    public boolean push(T anElement);

    public T pop();

    public T peek();
}
```

*** Interface Stack**

Linked Stack 을 구현할 때 사용할 Stack interface 입니다. stack을 구현하기 위해
필요한 기본적인 stack 관련 메소드들이 명시되어 있고 이후 이것을 implement 하여
LinkedStack을 구현합니다. LinkedStack 에서 위의 메서드들을 오버라이드하여 구현합니다.

주요 구현 내용

*** LinkedStack Class**

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

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

        }

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

        private IteratorForLinkedStack() {
            this.setNextNode(LinkedStack.this.top());
        }

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

        }

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

        }
    }
}
```

*** Class LinkedStack**

LinkedStack 클래스는 interface Stack을 implements 하여 구현되며, stack 인터페이스에 명시된 메서드들이 override 되어 구현됩니다. 또한 inner class로
LinkedStack을 위한 iterator가 구현되어 있어 이후에 LinkedStack을 순회할 때 해당 iterator를 이용하여 순회하게 됩니다.

주요 구현 내용

*** ShortestPaths Class의 Solve method**

```
public boolean solve(AdjacencyGraph<WE> aGraph, int aSource) {
    if (aGraph == null || aGraph.numberOfVertices() < 2) {
        return false;

    }
    if (!aGraph.vertexDoesExist(aSource)) {
        return false;

    }
    this.setGraph(aGraph);
    this.setSource(aSource);
    this.setDistance(new int[this.graph().numberOfVertices()]);
    this.setPath(new int[this.graph().numberOfVertices()]);

    boolean[] found = new boolean[this.graph().numberOfVertices()];
    for (int vertex = 0; vertex < this.graph().numberOfVertices(); vertex++) {
        found[vertex] = false;
        this.distance()[vertex] = ((SupplementForWeightedGraph<WE>) this.graph()).weightOfEdge(this.source(),
                vertex);
        this.path()[vertex] = this.source();
    }
    found[this.source()] = true;
    this.distance()[this.source()] = 0;
    this.path()[this.source()] = -1;

    ShortestPaths.showDebugMessage("\n[DEBUG] 최단경로 찾기 반복 과정:\n");
    this.debug_showIteration(0, this.source());
    for (int i = 1; i < this.graph().numberOfVertices() - 1; i++) {
        int u = chooseVertexForNextShortestPath(found);
        found[u] = true;
        Iterator<WE> iterator = this.graph().neighborIteratorOf(u);
        while (iterator.hasNext()) {
            WE edge = iterator.next();
            int w = edge.headVertex();
            if (!found[w]) {
                if (this.distance()[w] > this.distance()[u] + edge.weight()) {
                    this.distance()[w] = this.distance()[u] + edge.weight();
                    this.path()[w] = u;
                }
            }
        }
        this.debug_showIteration(i, u);

    }
    ShortestPaths.showDebugMessage("[DEBUG] 반복 과정 보여주기를 마칩니다. \n");
    return true;
}
```

***ShortestPaths Class**

이번 과제의 이론인 다익스트라 알고리즘을 구현한 ShortestPaths 클래스의 solve method 입니다. 그래프와 시작 vertex를 설정해주고, 방문한 vertex를 나타내기 위한 boolean type의 found를 그래프의 vertex 개수만큼 선언합니다. 초기값으로 시작 vertex의 distance 와 path를 각각 0,-1 을 부여하며 시작 vertex를 제외한 numberOfVertex - 1 번 순회하면서 각 vertex를 방문하며, chooseVertexForNextShortestPath 메소드를 통해 distance가 작은 vertex를 선택하게 됩니다.

또한 vertex가 추가되면 기존 distance 와 비교후 작은 값으로 다시 distance를 설정해주는 과정을 반복하며 최단경로를 구하게 됩니다.

주요 구현 내용

*** WeightedDirectedAdjacencyListGraph Class 중 일부 .**

```java
package graph;

import list.Iterator;

public class WeightedDirectedAdjacencyListGraph<WE extends WeightedEdge> extends DirectedAdjacencyListGraph<WE>
        implements SupplementForWeightedGraph<WE> {

    private static final int WEIGHT_INFINITE = Integer.MAX_VALUE / 2;

    public WeightedDirectedAdjacencyListGraph(int givenNumberOfVertices) {
        super(givenNumberOfVertices);

    }

    protected int adjacencyOfEdge(int aTailVertex, int aHeadVertex) {
        Iterator<WE> iterator = (Iterator<WE>) this.neighborIteratorOf(aTailVertex);
        while (iterator.hasNext()) {
            WE neighborEdge = (WE) iterator.next();
            if (aHeadVertex == neighborEdge.headVertex()) {
                return neighborEdge.weight();

            }
        }
        return WeightedDirectedAdjacencyListGraph.WEIGHT_INFINITE;
    }

    public boolean edgeDoesExist(int aTailVertex, int aHeadVertex) {
        if (this.edgeIsValid(aTailVertex, aHeadVertex)) {
            return (this.adjacencyOfEdge(aTailVertex,
                    aHeadVertex) < WeightedDirectedAdjacencyListGraph.WEIGHT_INFINITE);

        }
        return false;

    }

    public int weightOfEdge(int aTailVertex, int aHeadVertex) {
        if (this.edgeIsValid(aTailVertex, aHeadVertex)) {
            return this.adjacencyOfEdge(aTailVertex, aHeadVertex);

        }
        return WeightedDirectedAdjacencyListGraph.WEIGHT_INFINITE;
    }

    public int weightOfEdge(WE anEdge) {
        if (this.edgeIsValid(anEdge)) {
            return this.adjacencyOfEdge(anEdge.tailVertex(), anEdge.headVertex());
        }

        return WeightedDirectedAdjacencyListGraph.WEIGHT_INFINITE;
```

*** WeightedDirectedAdjacencyListGraph Class**

WeightedDirectedAdjacencyListGraph 클래스에서는 존재하지 않는 Edge의 weight를 표현하기 위한 WEIGHT_INIFITE 상수가 선언되어 있고, Edge의 Weight을 반환할 때 존재하지 않는 Edge의 Weight을 INFINITE 값으로 반환됩니다.

이후 최단경로를 찾을 그래프로 사용됩니다.

주요 구현 내용

* AppController 클래스의 solveAndShowShortestPaths() Method

```
private void solveAndShowShortestPaths() {

    AppView.outputLine("");
    AppView.outputLine("> 주어진 그래프에서 최단 경로를 찾습니다:");
    if (this.graph().numberOfVertices() <= 1) {
        AppView.outputLine("[오류] vertex 수 (" + this.graph().numberOfVertices()
                + ") 가 너무 적어서, 최단경로 찾기를 하지 않습니다 . 2개 이상이어야 합니다. ");
    } else {
        AppView.outputLine("> 출발점을 입력해야 합니다: ");
        int sourceVertex = this.inputSourceVertex();
        if (this.shortestPaths().solve(this.graph(), sourceVertex)) {
            AppView.outputLine("");
            AppView.outputLine("> 최단 경로별 비용과 경로는 다음과 같습니다: ");
            AppView.outputLine("출발점=" + sourceVertex + ":");
            for (int destination = 0; destination < this.graph().numberOfVertices(); destination++) {
                if (destination != sourceVertex) {
                    AppView.output("  [목적점=" + destination + "]");
                    AppView.output("최소비용=" + this.shortestPaths().minCostOfPathToDestination(destination) + ", ");
                    AppView.output("경로:");
                    LinkedStack<Integer> pathToDestination = this.shortestPaths().pathToDestination(destination);
                    LinkedStack<Integer>.IteratorForLinkedStack iterator = pathToDestination.iterator();
                    while (iterator.hasNext()) {
                        AppView.output(" -> " + iterator.next());

                    }
                    AppView.outputLine("");
                }
            }
        } else {
            AppView.outputLine("[오류] 최단경로 찾기를 실패하였습니다.");
        }

    }
}
```

***AppController Class**

앞서 구현했던 ShortestPaths 클래스의 solve 메소드를 이용해서 AppController 클래스에서 최단경로 찾기를 수행하고, minCostOfPathToDestination 메서드를 통해 최소비용을 출력하고, 수행한 결과로 나온 경로에 대한 Stack의 값을 iterator를 통해 보여주므로써 경로를 사용자에게 출력해줍니다.

결과화면 1

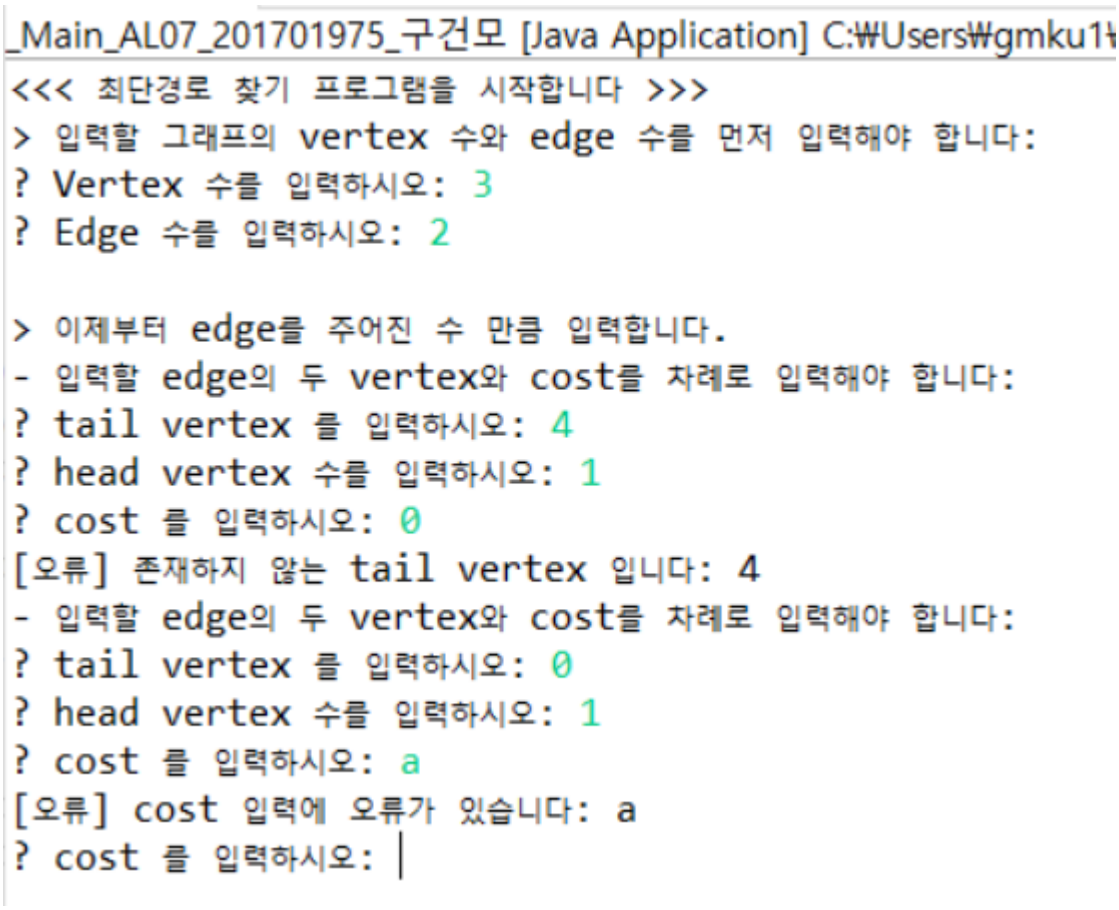
```
<terminated> _Main_AL07_201701975_구건모 [Java Application] (
<<< 최단경로 찾기 프로그램을 시작합니다 >>>
> 입력할 그래프의 vertex 수와 edge 수를 먼저 입력해야 합니다:
? Vertex 수를 입력하시오: 5
? Edge 수를 입력하시오: 7

> 이제부터 edge를 주어진 수 만큼 입력합니다.
- 입력할 edge의 두 vertex와 cost를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 1
? cost 를 입력하시오: 10
!새로운 edge (0,1, (10)) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex와 cost를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 3
? cost 를 입력하시오: 30
!새로운 edge (0,3, (30)) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex와 cost를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 4
? cost 를 입력하시오: 100
!새로운 edge (0,4, (100)) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex와 cost를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 1
? head vertex 수를 입력하시오: 2
? cost 를 입력하시오: 50
!새로운 edge (1,2, (50)) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex와 cost를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 2
? head vertex 수를 입력하시오: 4
? cost 를 입력하시오: 10
!새로운 edge (2,4, (10)) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex와 cost를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 3
? head vertex 수를 입력하시오: 2
? cost 를 입력하시오: 20
!새로운 edge (3,2, (20)) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex와 cost를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 3
? head vertex 수를 입력하시오: 4
? cost 를 입력하시오: 60
!새로운 edge (3,4, (60)) 가 그래프에 삽입되었습니다.
```

```
> 입력된 그래프는 다음과 같습니다:
[0] -> 4(100) 3(30) 1(10)
[1] -> 2(50)
[2] -> 4(10)
[3] -> 4(60) 2(20)
[4] ->


> 주어진 그래프에서 최단 경로를 찾습니다:
> 출발점을 입력해야 합니다:
? 출발 vertex를 입력하시오: 0

[DEBUG] 최단경로 찾기 반복 과정:
[DEBUG] Iteration_0: (u=0): d[0]=0 d[1]=10 d[2]=1073741823 d[3]=30 d[4]=100
[DEBUG] Iteration_1: (u=1): d[0]=0 d[1]=10 d[2]=60 d[3]=30 d[4]=100
[DEBUG] Iteration_2: (u=3): d[0]=0 d[1]=10 d[2]=50 d[3]=30 d[4]=90
[DEBUG] Iteration_3: (u=2): d[0]=0 d[1]=10 d[2]=50 d[3]=30 d[4]=60
[DEBUG] 반복 과정 보여주기를 마칩니다.

> 최단 경로별 비용과 경로는 다음과 같습니다:
출발점=0:
  [목적점=1]최소비용=10, 경로: -> 0 -> 1
  [목적점=2]최소비용=50, 경로: -> 0 -> 3 -> 2
  [목적점=3]최소비용=30, 경로: -> 0 -> 3
  [목적점=4]최소비용=60, 경로: -> 0 -> 3 -> 2 -> 4

<<< 최단경로 찾기 프로그램을 종료합니다 >>>
```

결과 1, 최단거리 경로

다익스트라 알고리즘을 이용해 ShortestPaths 를 구해 출력된 모습입니다. 사용자가 추가한 Graph 에서 0번 vertex를 시작 vertex로 하여 다익스트라 알고리즘을 이용해 최단거리의 경로를 출력하는 결과입니다.

결과화면 2



결과화면2 - 예외상황에 대한 결과

범위를 벗어난 vertex 를 추가하려고 하거나, cost로 들어온 값이 적절하지 않은 등의 예외상황에 대한 결과입니다. 각각의 경우에 알맞는 오류 메시지를 출력하고 있는 것을 확인하실 수 있습니다.

결과화면 3

```
<terminated> _Main_AL07_201701975_구건모 [Java Application] C:₩Users₩gmku1₩.p2₩pool₩p
<<< 최단경로 찾기 프로그램을 시작합니다 >>>
> 입력할 그래프의 vertex 수와 edge 수를 먼저 입력해야 합니다:
? Vertex 수를 입력하시오: 1
? Edge 수를 입력하시오: 0

> 이제부터 edge를 주어진 수 만큼 입력합니다.

> 입력된 그래프는 다음과 같습니다:
[0] ->


> 주어진 그래프에서 최단 경로를 찾습니다:
[오류] vertex 수 (1) 가 너무 적어서, 최단경로 찾기를 하지 않습니다 . 2개 이상이어야 합니다.

<<< 최단경로 찾기 프로그램을 종료합니다 >>>
```

결과화면3 - 예외상황에 대한 결과

vertex 가 1 개일 때의 결과입니다. 최단경로 찾기는 최소 2개 이상의 vertex가 있어야만 실행되도록 하였기 때문에 실행되지 않는 모습을 보실 수 있습니다.

결과화면 3

```
<terminated> _Main_AL07_201701975_구건모 [Java Application] C:₩Users₩gmku1₩.p2₩pool₩p
<<< 최단경로 찾기 프로그램을 시작합니다 >>>
> 입력할 그래프의 vertex 수와 edge 수를 먼저 입력해야 합니다:
? Vertex 수를 입력하시오: 1
? Edge 수를 입력하시오: 0

> 이제부터 edge를 주어진 수 만큼 입력합니다.

> 입력된 그래프는 다음과 같습니다:
[0] ->


> 주어진 그래프에서 최단 경로를 찾습니다:
[오류] vertex 수 (1) 가 너무 적어서, 최단경로 찾기를 하지 않습니다 . 2개 이상이어야 합니다.

<<< 최단경로 찾기 프로그램을 종료합니다 >>>
```

결과화면3 - 예외상황에 대한 결과

vertex 가 1 개일 때의 결과입니다. 최단경로 찾기는 최소 2개 이상의 vertex가 있어야만 실행되도록 하였기 때문에 실행되지 않는 모습을 보실 수 있습니다.

결과화면 4

```
> 입력된 그래프는 다음과 같습니다:
[0] -> 1(10)
[1] -> 4(3) 7(4) 6(4) 15(2) 2(1
[2] -> 19(23) 6(44) 3(13)
[3] -> 5(4) 8(3) 6(555) 4(15)
[4] -> 5(2)
[5] -> 12(30) 19(9) 6(12)
[6] -> 13(55) 9(1) 7(11)
[7] -> 19(30) 16(21) 8(40)
[8] -> 13(20) 9(25)
[9] -> 10(51)
[10] -> 11(40)
[11] ->
[12] ->
[13] ->
[14] ->
[15] -> 17(12)
[16] ->
[17] -> 19(20)
[18] ->
[19] -> 18(33)
```

```
> 주어진 그래프에서 최단 경로를 찾습니다:
> 출발점을 입력해야 합니다:
? 출발 vertex를 입력하시오: 0

[DEBUG] 최단경로 찾기 반복 과정:
[DEBUG] Iteration_0: (u=0): d[0]=0 d[1]=10 d[2]=1073741823 d[3]=1073741823 d[4]=10737418
[DEBUG] Iteration_1: (u=1): d[0]=0 d[1]=10 d[2]=24 d[3]=1073741823 d[4]=13 d[5]=10737418
[DEBUG] Iteration_2: (u=15): d[0]=0 d[1]=10 d[2]=24 d[3]=1073741823 d[4]=13 d[5]=1073741
[DEBUG] Iteration_3: (u=4): d[0]=0 d[1]=10 d[2]=24 d[3]=1073741823 d[4]=13 d[5]=15 d[6]=
[DEBUG] Iteration_4: (u=6): d[0]=0 d[1]=10 d[2]=24 d[3]=1073741823 d[4]=13 d[5]=15 d[6]=
[DEBUG] Iteration_5: (u=7): d[0]=0 d[1]=10 d[2]=24 d[3]=1073741823 d[4]=13 d[5]=15 d[6]=
[DEBUG] Iteration_6: (u=5): d[0]=0 d[1]=10 d[2]=24 d[3]=1073741823 d[4]=13 d[5]=15 d[6]=
[DEBUG] Iteration_7: (u=9): d[0]=0 d[1]=10 d[2]=24 d[3]=1073741823 d[4]=13 d[5]=15 d[6]=
[DEBUG] Iteration_8: (u=2): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7]=
[DEBUG] Iteration_9: (u=17): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7]
[DEBUG] Iteration_10: (u=19): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7
[DEBUG] Iteration_11: (u=16): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7
[DEBUG] Iteration_12: (u=3): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7]
[DEBUG] Iteration_13: (u=8): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7]
[DEBUG] Iteration_14: (u=12): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7
[DEBUG] Iteration_15: (u=18): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7
[DEBUG] Iteration_16: (u=13): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7
[DEBUG] Iteration_17: (u=10): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7
[DEBUG] Iteration_18: (u=11): d[0]=0 d[1]=10 d[2]=24 d[3]=37 d[4]=13 d[5]=15 d[6]=14 d[7
[DEBUG] 반복 과정 보여주기를 마칩니다.

> 최단 경로별 비용과 경로는 다음과 같습니다:
출발점=0:
  [목적점=1]최소비용=10, 경로: -> 0 -> 1
  [목적점=2]최소비용=24, 경로: -> 0 -> 1 -> 2
  [목적점=3]최소비용=37, 경로: -> 0 -> 1 -> 2 -> 3
  [목적점=4]최소비용=13, 경로: -> 0 -> 1 -> 4
  [목적점=5]최소비용=15, 경로: -> 0 -> 1 -> 4 -> 5
  [목적점=6]최소비용=14, 경로: -> 0 -> 1 -> 6
  [목적점=7]최소비용=14, 경로: -> 0 -> 1 -> 7
  [목적점=8]최소비용=40, 경로: -> 0 -> 1 -> 2 -> 3 -> 8
  [목적점=9]최소비용=15, 경로: -> 0 -> 1 -> 6 -> 9
  [목적점=10]최소비용=66, 경로: -> 0 -> 1 -> 6 -> 9 -> 10
  [목적점=11]최소비용=106, 경로: -> 0 -> 1 -> 6 -> 9 -> 10 -> 11
  [목적점=12]최소비용=45, 경로: -> 0 -> 1 -> 4 -> 5 -> 12
  [목적점=13]최소비용=60, 경로: -> 0 -> 1 -> 2 -> 3 -> 8 -> 13
  [목적점=14]최소비용=1073741823, 경로: -> 0 -> 14
  [목적점=15]최소비용=12, 경로: -> 0 -> 1 -> 15
  [목적점=16]최소비용=35, 경로: -> 0 -> 1 -> 7 -> 16
  [목적점=17]최소비용=24, 경로: -> 0 -> 1 -> 15 -> 17
  [목적점=18]최소비용=57, 경로: -> 0 -> 1 -> 4 -> 5 -> 19 -> 18
  [목적점=19]최소비용=24, 경로: -> 0 -> 1 -> 4 -> 5 -> 19

<<< 최단경로 찾기 프로그램을 종료합니다 >>>
```

결과화면4 - 그래프 크기가 큰 예시

Vertex 20개 , Edge 30개로 이루어진 그래프 입니다. 규모가 큰 경우에도 ShortestPaths 가 정상적으로 찾아지고 있는 것을 보실 수 있습니다.

* 앞의 입력부분이 너무 길어서 결과만 첨부하였습니다.

Iteration Debug message의 경우도 너무 길어 일부 생략하였습니다.

생각할 점

> 이번 과제에서 그래프의 정의에 사용된 interface 와 class 를 어떤 경우에 어느 것을 사용하면 좋은지?

이번 과제에서 최단경로를 구현하기 위해 사용된 interface 에는 iterator, stack, Graph, SupplementForWeightedGraph 등이 있습니다. iterator 는 구현이 달라질 때마다 해당 구조에 적합한 순회방법을 구현할 때 사용되고 stack은 이번 과제의 LinkedStack 클래스를 구현되는데 사용됩니다. LinkedStack의 경우 ShortestPaths를 저장하는 자료구조로 쓰이게 됩니다. Graph의 경우 어떤 종류의 그래프인지에 관계없이 공통적으로 필요한 부분들을 기술하며 SupplementForWeightedGraph는 그래프에 Weight 개념이 추가되는 그래프인 경우 사용하게 됩니다. 따라서 이번 과제의 주요 구현중 하나인 WeightedDirectedAdjacencyListGrpah 클래스는 interface Graph를 이용해 만든 DirectedAdjacecnyListGraph에서 Weight 개념이 추가된 것으로 SupplmentForWeightedGraph 까지 이용하여 만들어 지게 됩니다.
이렇게 interface는 알고리즘을 구현하는데 필요한 class에 맞게 기존의 class 들과 interface 들을 선택해서 사용하는 것이 좋다고 생각합니다.