알고리즘

- HW09: Sequence Alignment,Bellman-Ford-

제 출 일	2017년 11월 30일
분 반	02반
담당교수	공은배
학 과	컴퓨터공학과
학 번	201302423
이 름	신종욱

ㄱ.Sequence Alignment-linear space

1. 해결 방법

```
static void Align(int a, int b, int c, int d, int[][] table) {// 0,0부터 x,y까지가지 최단경로 찾기
    if (d - b <= 1)
        return;//차이가 1이하면 종료
    int n = d + b;//
    int[] Yprefix = new int[c + 1 - a];
    int[] Ysuffix = new int[c + 1 - a];
    Yprefix = AllYprefixCosts(a, b, c, d, n / 2, table);//a,b부터 ?,n/2 까지의 차를 저장
    Ysuffix = AllYsuffixCosts(a, b, c, d, n / 2, table);//?,n/2부터 c,d까지의 차를 저장
    int mincost = 99999;
    int besty = b;//선정된 y좌표를 저장하기위한 변수
    int min = 99999;
    for (int p = a; p < c + 1; p++) {
        min = Math.min(min, table[p][n / 2]);
    }//해당라인에 있는 최소값을 저장
    for (int k = a; k < c + 1; k++) {
        if (Yprefix[k - a] >= 0 && Ysuffix[k - a] >= 0 && mincost >= Yprefix[k - a] + Ysuffix[k - a]//둘다 0이상이고 mincost보다 작고
                && k != table.length - 1 && table[k][n / 2] <= table[k + 1][n / 2]//선택된 배열의 오른쪽,아래,오른쪽대각선 아래보다 작거나 같아야 최소값이다.
                && table[k][n / 2] <= table[k + 1][n / 2 + 1] && table[k][n / 2] <= table[k + 1][n / 2 + 1]) {
            if (min == table[k][n / 2]) {//마지막으로 해당값이 최솟값인지 한번더 확인한다.
                mincost = Yprefix[k - a] + Ysuffix[k - a];//맞다면 MINCOST 갱신
                besty = k;//해당 k값을 저장
            }
        }
    }
    MinRoute.add(new node(besty, n / 2));//path에 추가
    System.out.println(besty + " " + n / 2);//선택된값 출력

    Align(a, b, besty, n / 2, table);//왼쪽도 진행
    Align(besty, n / 2, c, d, table);//오른쪽도 진행
}

static int[] AllYprefixCosts(int a, int b, int c, int d, int n, int[][] table) {
    int[] temp = new int[c + 1 - a];//k,n과 a,b배열의 차이를 정리하여서 리턴한다.
    for (int k = a; k < c + 1; k++)
        temp[k - a] = table[k][n] - table[a][b];
    return temp;
}

static int[] AllYsuffixCosts(int a, int b, int c, int d, int n, int[][] table) {
    int[] temp = new int[c + 1 - a];//k,n과 c,d배열의 차이를 정리하여서 리턴한다.
    for (int k = a; k < c + 1; k++)
        temp[k - a] = table[c][d] - table[k][n];//일반적으로 OPT테이블의 오른편이 값이 더큼으로 오른쪽에서 왼쪽을 빼줘야한다.
    return temp;
}
```

OPTTable이 x축으로 0..n이라는 배열이라면 무조건 n/2를 지나야하기 때문에 n/2를 지날 때 선택되는 y축을 골라내어서 Path를 추가해야한다.
Path는 ArrayList로 하여서 x,y값을 둘다 저장할 수 있는 node를 저장하도록 구현하였다.
0,0에서 n/2,(012345..)까지가는 Cost를 계산하고 나머지 오른편도 Cost를 계산후에 두 개의 합이 최솟값이 되는곳을 선택하면되는데 이때 문제점이 만약 최종 Cost는 같은경우일 경우 어떤 것을 선택해야하는지인데 그럴경우에는 해당 OPT값이 더 작은 값을 선택했다.
이런식으로 구현하면 또 하나의 문제점이 나왔는데 x값 1개당 하나의 좌표만 설정해서
같은 x값에 y값이 1증가하는 gap을 무시하고 넘어갔다.

```
Collections.sort(MinRoute);
MinRoute.add(0, new node(0, 0));
MinRoute.add(MinRoute.size(), new node(str1.length(), str2.length()));// 첫노드와 마지막 노드 추가
for (int z = 0; z < MinRoute.size() - 1; z++) {
    if (MinRoute.get(z + 1).i - MinRoute.get(z).i > 1) {
        int abc = MinRoute.get(z + 1).i - MinRoute.get(z).i;
        while (abc != 1) {
            MinRoute.add(z + 1, new node(MinRoute.get(z).i + 1, MinRoute.get(z).j));
            abc--;
            z++;
        }//선택안된 OPTPath를 선택하는 while문
    }
}
System.out.println();
System.out.println("최종 Path");
for (int w = 0; w < MinRoute.size(); w++)
    System.out.println(MinRoute.get(w).i + " " + MinRoute.get(w).j);
```

이문제를 해결하기위해 ArrayList에 만약 바로옆 x값과 차이가 클경우에는 사잇값을 추가하는식으루 구현했다.

2. 실행 결과

```
<terminated> Sequence [Java Application] C:₩Program Fil
String1 : ocurrance
String2 : occurrence
        o  c  c  u  r  r  e  n  c  e
     0  1  2  3  4  5  6  7  8  9 10
  o  1  0  1  2  3  4  5  6  7  8  9
  c  2  1  0  1  2  3  4  5  6  7  8
  u  3  2  1  2  1  2  3  4  5  6  7
  r  4  3  2  3  2  1  2  3  4  5  6
  r  5  4  3  4  3  2  1  2  3  4  5
  a  6  5  4  5  4  3  2  3  4  5  6
  n  7  6  5  6  5  4  3  4  3  4  5
  c  8  7  6  5  6  5  4  5  4  3  4
  e  9  8  7  6  7  6  5  4  5  4  3
4 5
2 2
1 1
2 3
3 4
5 7
5 6
7 8
8 9

최종 Path
0 0
1 1
2 2
2 3
3 4
4 5
5 6
5 7
6 7
7 8
8 9
9 10
```

```
<terminated> Sequence [Java Application] C:₩Program Fil
String1 : CTGACCTACCT
String2 : CCTGACTACAT
        C  C  T  G  A  C  T  A  C  A  T
     0  1  2  3  4  5  6  7  8  9 10 11
  C  1  0  1  2  3  4  5  6  7  8  9 10
  T  2  1  2  1  2  3  4  5  6  7  8  9
  G  3  2  3  2  1  2  3  4  5  6  7  8
  A  4  3  4  3  2  1  2  3  4  5  6  7
  C  5  4  3  4  3  2  1  2  3  4  5  6
  C  6  5  4  5  4  3  2  3  4  3  4  5
  T  7  6  5  4  5  4  3  2  3  4  5  4
  A  8  7  6  5  6  5  4  3  2  3  4  5
  C  9  8  7  6  7  6  5  4  3  2  3  4
  C 10  9  8  7  8  7  6  5  4  3  4  5
  T 11 10  9  8  9  8  7  6  5  4  5  4
4 5
1 2
1 1
2 3
3 4
8 8
5 6
7 7
9 9
9 10

최종 Path
0 0
1 1
1 2
2 3
3 4
4 5
5 6
6 6
7 7
8 8
9 9
9 10
10 10
11 11
```

최종 PATH에 뽑히는 순서와 정렬하고 앞뒤를 추가한 PATH를 출력하였다.

또다른 예

```
<terminated> Sequence [Java Application]
String1 : qwer
String2 : rasd
        r  a  s  d
     0  1  2  3  4
  q  1  2  3  4  5
  w  2  3  4  5  6
  e  3  4  5  6  7
  r  4  3  4  5  6
0 2
0 1
0 3

최종 Path
0 0
0 1
0 2
0 3
1 3
2 3
3 3
4 4
```

```
<terminated> Sequence [Java Application] C:₩Progra
String1 : qwer
String2 : asdf
        a  s  d  f
     0  1  2  3  4
  q  1  2  3  4  5
  w  2  3  4  5  6
  e  3  4  5  6  7
  r  4  5  6  7  8
0 2
0 1
0 3

최종 Path
0 0
0 1
0 2
0 3
1 3
2 3
3 3
4 4
```

missmatch가 아니고 gap으로 가는형태가 많지만 결국 최소경로로 길을 구한다.

ㄴ.Bellman-Ford

1. 해결 방법

x,에서 target을 향하는 OPT테이블을 구현하였다.
단 여기서 target은 마지막 인덱스여야된다.
그리고 일단 target으로 갈 때 노드가 중복선택되지않는다면 사용할수 있는 edge의 개수는
노드갯수 -1이다.
여기선 음의 사이클이 안일어나는 프로그램인걸 보여주기위해 edge사용갯수를 좀더 증가시켜서 출력하였다.

```
int[][] Edge = new int[NodeCount][NodeCount];

for (int i = 0; i < NodeCount; i++) {
    for (int j = 0; j < NodeCount; j++) {
        Edge[i][j] = TempEdge[i][j];
    }
} // 엣지옮기기

int[][] OPT = new int[NodeCount + 1][NodeCount];
boolean[][][] OPTPath = new boolean[NodeCount + 1][NodeCount][NodeCount];
for(int i=0;i<NodeCount + 1;i++) {
    for(int j=0;j<NodeCount;j++) {
        OPTPath[i][j][j]=true;
    }
}
for (int i = 0; i < NodeCount; i++) {
    OPT[0][i] = M;
}

OPT[0][Target] = 0;
```

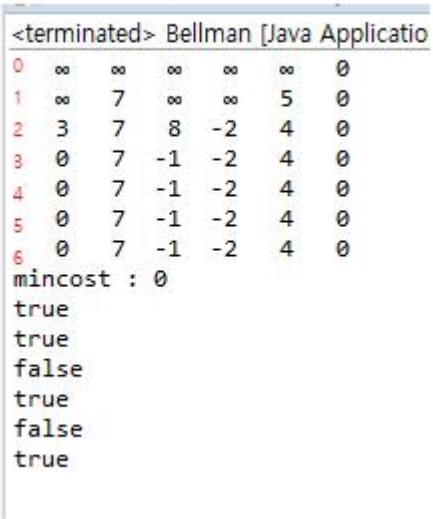
일단 파일들을 읽고 엣지를 저장하며 딱맞은 배열로 옮겨줬다.
그리고 OPT테이블을 생성하고 OPT테이블의 값을 만들면서 지나온 인덱스를 저장하기위한
3차원 boolean배열로 OPTPath를 하나 더 선언해줬다.
그리고 항상 자기자신은 지난다고 생각하여서 true로 초기화해줬다.

```
for (int i = 1; i < NodeCount + 1; i++) {
    for (int j = 0; j < NodeCount - 1; j++) {
        int temp = M;
        int MinI = M;
        int MinJ = M;
        int MinK = M;
        int flag = 0;
        for (int k = 0; k < NodeCount; k++) {
            if (Edge[j][k] != M)//엣지가 무한대이면 할필요가없다
                if (Edge[j][k] > 0 && OPT[i - 1][k] > 0 && Edge[j][k] + OPT[i - 1][k] < 0) {//양의 오버플로우를 체크
                } else if (temp > Edge[j][k] + OPT[i - 1][k]&&OPT[i-1][k]!=M) {//temp보다 작을경우
                    if(OPTPath[i-1][k][j]!=true)//사이클 확인 if문
                    {temp = Edge[j][k] + OPT[i - 1][k];
                    MinI = i-1;
                    MinJ = j;
                    MinK = k;
                    flag = 1;}//사이클이 아닐때 저장하여 기억함
                }

        }

        if (OPT[i - 1][j] <= temp) {
            OPT[i][j] = OPT[i - 1][j];
            OPTPath[i][j] = OPTPath[i-1][j];
        }//temp가 더클경우에는 그냥 이전의 i-1,j를 그대로 받아오면된다.
        else {//그외는 값을 바꿔야한다
            OPT[i][j] = temp;
            if(flag==1)//변화가 있을경우에만
                {if(i!=1) {
                for(int R=0;R<NodeCount;R++)
                    OPTPath[i][j][R] = OPTPath[MinI][MinK][R];//a->b--//->target으로 갈경우에
                //b--//-->target까지의 경로를 새롭게 저장한다.
                OPTPath[i][j][j] =true;}//혹시 초기값이 수정될수도있으니 해당 인덱스를 다시 초기화한다
            OPTPath[i][j][Target] =true;//새로운 값을 발견했다는건 target에 도착했다는 뜻임으로 true로한다
            OPTPath[i][j][MinK] =true; //새롭게 연결한값이 flag가 아닐가능성도 있으니 한번더 선언
            }
        }
    }
}
```

선택됐을 때 I,J,K값을 저장하기위해 따로 변수를 선언하였다.
일단 연결된 주변 이웃이 있는지 확인하기위해 Edge 배열을 이용하여서 최댓값이 아닌 경우에는 계산을 하여서 최솟값이 되는지 확인하였다. 여기서 오버플로우가 일어나서 출력값 에러가 뜰경우가 있어서 조건을 추가해줬다.
이 과정을 거쳐서 나온 MinI와 MinJ,MinK는 새로운 길을 찾은 경우인데 만약 I가 1일 경우에는
최초연결이라서 그이전의 OPTpath를 확인 할 필요가없지만 그 외의 경우에는
a에서 target을 간다고하면 연결된 이웃인 b로 가고 b에서 target으로가는 경로를 사용한다
a->b--//-->target
이런식이기 때문에 b에서 target으로가는 경로를 알아야해서
OPTPath[MinI][MinK]의 값들을 I,j로 다 복사하였다. 그럼 경로를 저장하면서 진행할 수 있다.
이값을 이용하여서 Edge를 돌아다닐 때 OPTPath[i-1][k][j] 값을 이용하여서 알아낼 수 있다.
(b-->taget으로 가는 경로중에 a가있다면 true이다.)

2.실행결과

엣지의 수

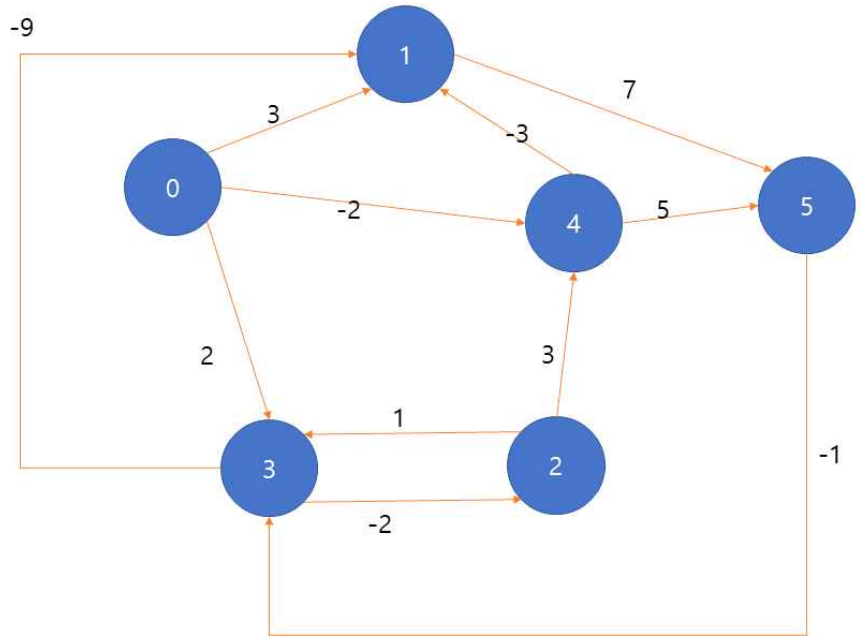


음의 사이클도 해결했다는걸 보여주기 위해 많은 엣지까지 선택하여서
테이블의 크기를 더 많이 만들어서 출력하였다.
노드는 가로로 출력하고 엣지의 개수가 늘어날수록 아래로 출력한다.

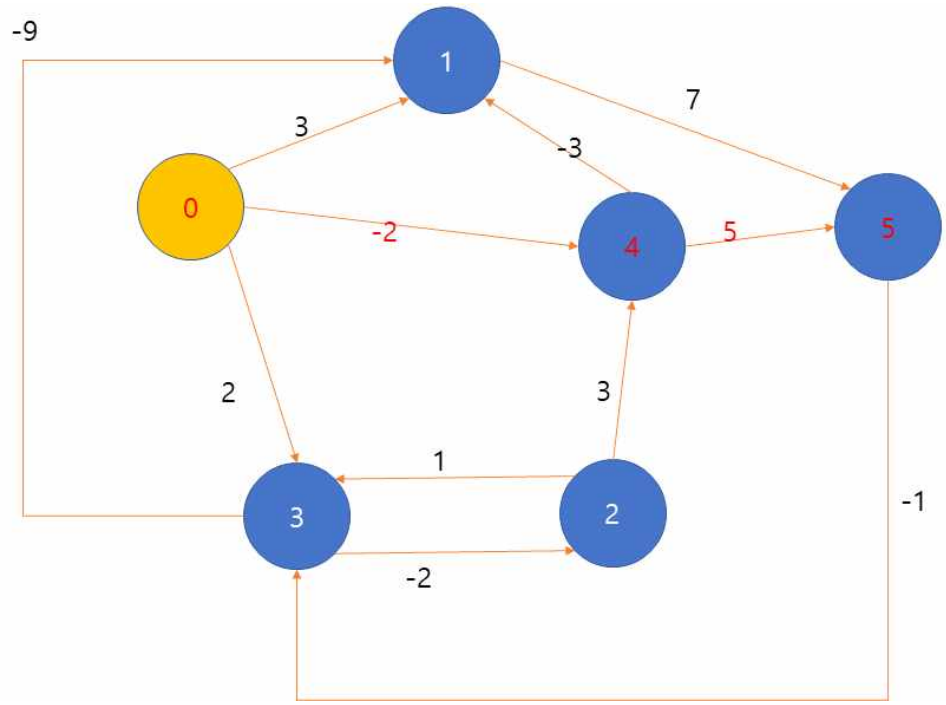
table과 start가 0인덱스일 때 mincost인 0을 출력 그리고 해당 path에 선택된 인덱스이다.
지나온 순서와는 무관하게 그냥 선택된 인덱스이다.

3. Path

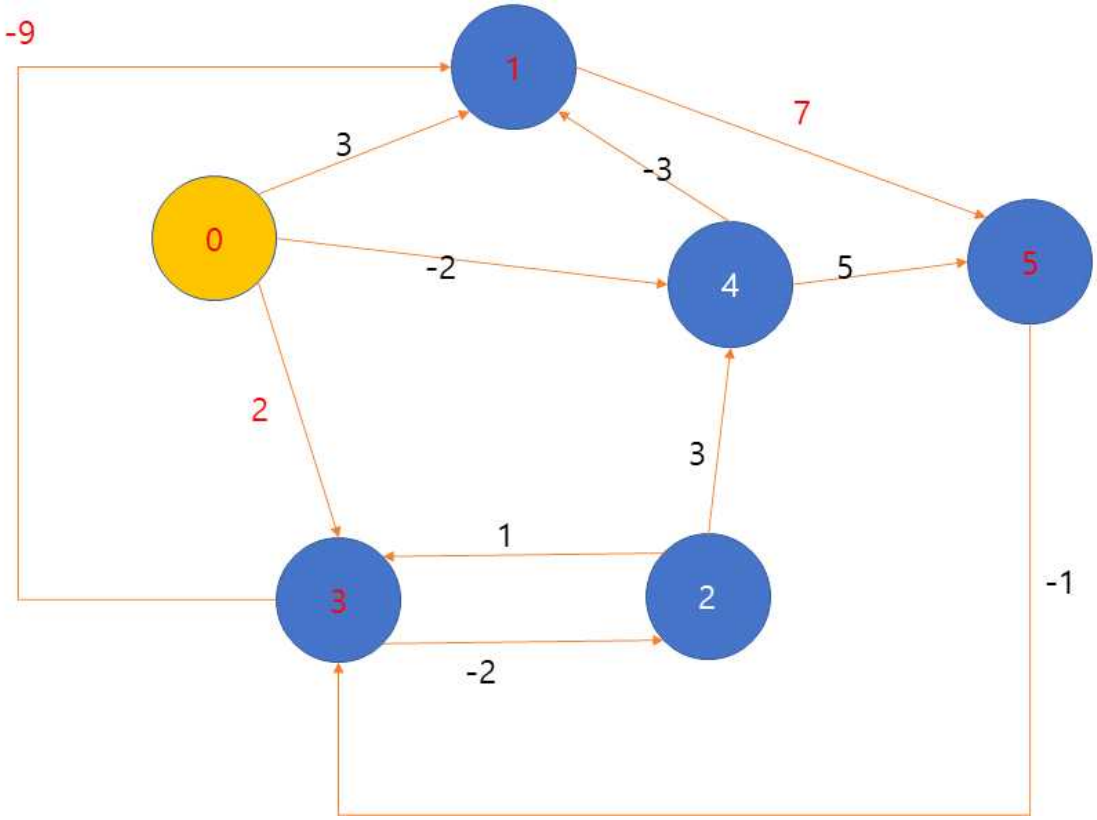
그래프화



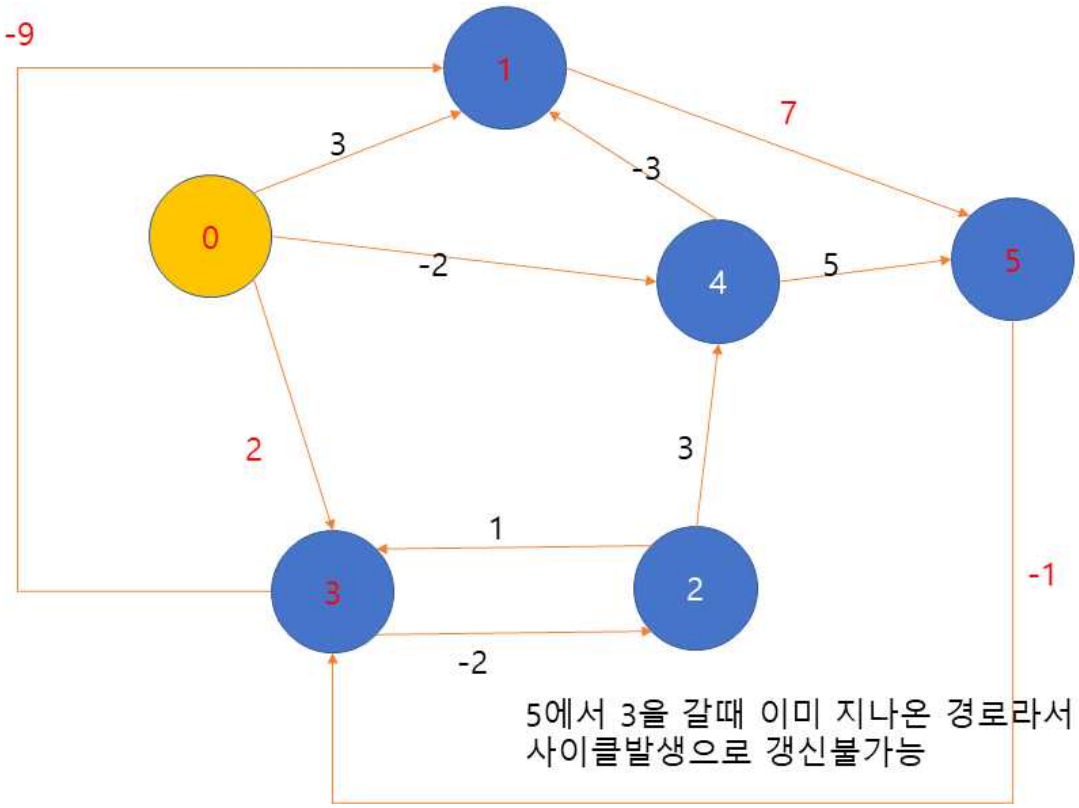
0에서 target까지가는 edge를 2개 사용하는 최소경로(0,4,5) cost : 3



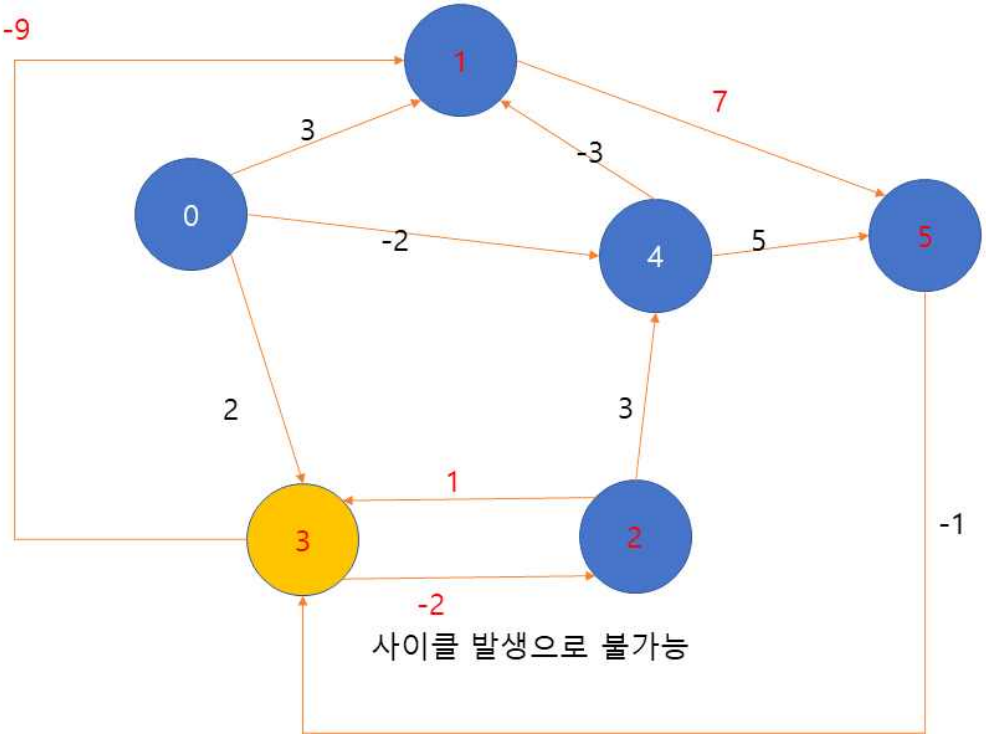
0에서 target까지가는 edge를 3개 사용하는 최소경로(0,3,1,5) cost : 0



만약 경로 확인이 없다면 edge가 6일때 0315315가 되면서 Cost가 –3이된다



사이클이 발생하는 다른 예)) 3에서 target을 4개의 egde사용(3,2,3,1,5)



3,2,3,1,5를하면 cost가 –3으로 이전값이 –2보다 작지만 사이클이 발생해서 갱신하면안된다.

4. 느낀 점

두 과제 모두다 예외와 조건이 매우 까다로워서 어려웠다.
Linear space는 하나의 x좌표만 선택하는걸로 이해하여서 잘못 구현했다가 출력을 보고 이상하다는걸 느껴서 수정하였다. 다르게하면 좀 더 깔끔하게 구현할 수 있을거같다.
특히 벨만포드는 target인덱스가 바뀔때도 구현하고 싶었지만 초석을 잘 못 깔아서 불가능했다.
그래도 OPT테이블은 출력하였다.