| ECE30011-01 알고리듬분석 | Student number | 21200446 |
|---|---|---|
| Homework 2 | Name | 여 은 동 |

# 알고리즘 final 과제

## 1. DFS

# CODE

```
import java.util.List;
import java.util.ArrayList;
import java.util.Stack;

	class vertex {
		public int color;
		public int startTime;		//시작 시간
		public int finishTime;		//완료 시간
		public char name;		//출력을 위한 노드 네임
	vertex(){				//컨스트럭터
			this.color = 0;
			this.startTime=0;
			this.finishTime=0;
		}
	}

	public class dfs {

		static int[][] edge =		//엣지 정보를 담고있는 array 테이블
			//a, b, c, d, e, f, g, h
			{{0, 1, 0, 0, 0, 0, 0, 0	}, //a
			{ 0, 0, 1, 0, 1, 1, 0, 0	}, //b
			{ 0, 0, 0, 1, 0, 0, 1, 0	}, //c
			{ 0, 0, 1, 0, 0, 0, 0, 1	}, //d
			{ 1, 0, 0, 0, 0, 1, 0, 0	}, //e
			{ 0, 0, 0, 0, 0, 0, 1, 0	}, //f
			{ 0, 0, 0, 0, 0, 1, 0, 1			}, //g
			{ 0, 0, 0, 0, 0, 0, 0, 1			}   //h
			};
```

```
static int[][] transEdge = new int[8][8]; //역방향 엣지를 담을 trans array 테이블
        static int time=-1;
public static List<String> scc = new ArrayList<String>();   //scc 출력을 위한 리스트
public static String Str="";
public static void main(String argv[]){
        int i, j, k;
        //a, b, c, d, e, f, g, h

        //dfs in order
        vertex[] v = new vertex[8];
        vertex[] v2 = new vertex[8];
        for(i=0;i<8;i++)
                v2[i] = new vertex();
        for( i = 0 ; i < 8 ; i++)
        v[i] = new vertex();
        // 초기화
        for(i = 0 ; i< 8 ; i ++)
                for(j =0 ; j < 8 ; j++)
                        transEdge[j][i] = edge[i][j];

        v[0].name='a';
        for(i=1;i<8;i++)
                v[i].name=(char) (v[i-1].name +1);
        v2[0].name='a';
        for(i=1;i<8;i++)
                v2[i].name=(char) (v2[i-1].name +1);    //이름 초기화

        for(i=0;i<8;i++)
                if(v[i].color==0)
                        search(i, v, edge);

        for( i= 0; i< 8 ; i ++)
        {
                System.out.print("vertex["+ i +"] = "+v[i].name+" "+"  ₩n ");
                System.out.println("start =" + v[i].startTime);
                System.out.println("finish =" +v[i].finishTime);

        }

        // sorting

        vertex temp = new vertex(); // bubble sort로 vertex finish time 정렬
        for(j=0; j< 7 ; j++)
```

```
		for(i=0 ; i < 7 ; i++)
		{
			if(v[i].finishTime < v[i+1].finishTime)
			{temp=v[i];
			v[i] = v[i+1];
			v[i+1]=temp;
				}

		}


		time= -1; // 시간 다시 초기화/
		Str = ""; // str 초기화
		int count=0;
		System.out.println("==============================");

		//for( i = 0 ; i < 8 ; i ++)
		//		v2[i].name = v[i].name;
		String[] sccList = new String[4];

		System.out.println("Strongly connected component  :  ");
		while(count<8){
			int sema=0;		//세마포를 통해서 f[n] 내림차순으로 DFS( trans)
			for(i=0 ; i < 8 ; i ++)
			if(v2[i].color==0 && v[count].name==v2[i].name)
			{		search(i, v2, transEdge);
		// time 역방향으로 솔팅 후에 DFS
				sema=1;
			}
			if( sema ==1)
			System.out.println(Str);
			Str="";
			count ++;
		}

		for(i=0 ; i < 8 ; i++)
		{
			System.out.print("vertex["+ i +"] = "+v2[i].name+" "+"  ₩n ");
			System.out.println("start =" + v2[i].startTime);
			System.out.println("finish =" +v2[i].finishTime);
		}

}
```

```
	public static void search(int i, vertex[] v3, int[][] e){
		int j;
		time ++;
		v3[i].color=1;		//detected
		v3[i].startTime=time;		//stamp current time
		Str= Str+v3[i].name;
		while(true){
			for(j=0;j<8;j++)
				if(e[i][j]==1)
					if(v3[j].color==0){
					search(j, v3, e);
						//white vertex를 발견.
						// recursive dfs.
					}
			time ++;
			v3[i].finishTime = time;		//finish time stamp
			return;
		}

	} // method dfs

}		//class dfs
```

# 출력 결과

```
vertex[0] = a
 start =0
finish =15
vertex[1] = b
 start =1
finish =14
vertex[2] = c
 start =2
finish =11
vertex[3] = d
 start =3
finish =6
vertex[4] = e
 start =12
finish =13
vertex[5] = f
 start =8
finish =9
vertex[6] = g
 start =7
finish =10
vertex[7] = h
 start =4
finish =5
--------------------------------
```

처음 해당 그래프에 대한 DFS 출력화면.

```
vertex[0] = a
 start =0
finish =5
vertex[1] = b
 start =2
finish =3
vertex[2] = c
 start =6
finish =9
vertex[3] = d
 start =7
finish =8
vertex[4] = e
 start =1
finish =4
vertex[5] = f
 start =11
finish =12
vertex[6] = g
 start =10
finish =13
vertex[7] = h
```

그래프의 모든 edge를 역방향으로 바꾼 다음 DFS 실행.

```
Strongly connected component  :
aeb
cd
gf
h
```

Trans 된 graph를 dfs하면서 각 vertex들을 스트링에 저장. 해당 dfs가 끝나면 모든 스트링에 저장된 vertex를 scc로 묶고 다음 dfs를 실행.

# discussion

우선적으로 SCC를 구현하는 데에 있어서는 링크드 리스트가 아닌 구현에 쉬운 어레이를 사용하여 구현하였다. 어레이는 각 vertex들 간의 edge정보를 다루고 있으며, 행의 인덱스에서 열의 인덱스로 이동한다고 생각하면 된다. 각 vertex들은 class로 하나로 묶어 오브젝트로 관리하였다. 또한 dfs를 사용하는 데에 있어서 스택구조로 구현하기 위하여, 재귀함수의 구조로 search() 메소드를 구현하였다.

구현에 크게 막히는 부분은 없었지만, 다 구현한 후 scc가 어떻게 추출되는지 자세히 몰라 다시 공부해야 했다. 결과적으론 finish time이 내림차순으로 dfs된 것들이 각각 scc를 이루는 것을 손으로 그려보고 알았다.

## 2.8 puzzle 알고리즘

처음 8 puzzle 알고리즘을 짜려고 했을 때, a*로 짜려고 했으나 제대로 구현되지 않았다. 우선 각 확장(search)마다 distance를 나눠서 구분해야 하는데, recursive로 짜면 각 search당 distance가 하나씩 늘어나서 depth별로 distance를 구별하는 것이 쉽지 않아서 포기하고 BFS로 짜려고 했다.

BFS로 짰을 때는 queue를 사용하여 구현하였는데, 예상했던 대로 O(9!)의 시간복잡도를 가지기 때문에 버퍼 용량이 계속 터졌고, 어느정도 추정치를 사용한 BFS를 사용하려 했으나(a*의 방식을 혼합한) 실패해서 다시 a* 알고리즘으로 돌아와서 짜게 되었다.

# CODE

```
package newpuzzle;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.List;

import java.util.Queue;


class node{

	int parent;
```

```
        int child;
}       //각 바꿀 내용에 대한 node( child와 parent 유지)
public class lastpuzzle {

        int[] goal = {1,2,3,  //최종 목표 map
                        8,-1,4,
                        7,6,5};
        int[] start= {1,3,2,   //초기 map
                -1,7,8,
                5,6,4};
        List<node> list = new ArrayList<node>();        //search 전 나아갈 방향들의 노드를 저장하는 list
        Queue<int[]> duplicate = new LinkedList<int[]>();          //이전 4상태 저장하는 list
        int direction=-1; // 이전에 있었던 장소. 다시 안가기 위해 막아둠.
        int depth=0;                    //현재 search한 node들의 search depth.
        public void addNode(int newparent, int newchild){
                node tmp = new node();
                tmp.parent= newparent;
                tmp.child= newchild;

                if(direction==newchild)
                        return; //이전에 있던 곳에 가지 않기.
                else
                        list.add(tmp);        //해당 방향으로 나아갈 node 저장.
                return;
        }
        public void search(){
                list.clear();           //시작전 list 초기화
                int i = 0;
                int empty =0;                   //빈 칸
```

```
int from=0;                              //현재 빈칸의 정보
int to=0;                          //바꿔질 칸의 정보
depth += 1;                              //깊이를 +1 추가.
empty = findEmpty();          //지금 빈칸이 어디있는지 check
System.out.println("empty = "+empty);
if( empty != -1)
{
        if( empty > 2 )                    // 위
        addNode(empty, empty-3);
if( empty < 6 )              //  아래
addNode(empty, empty+3);
if( (empty % 3) != 0)       // 좌
     addNode(empty, empty-1+0);
if( (empty % 3) != 2 )      // 우
addNode(empty, empty+1);
//각 방향에 대해 list에 노드 추가.

direction= empty;  // 현재 있는 빈칸을 이전 방향으로 저장하고, 막아두는 작업.
to = heuristic();

System.out.print("₩n₩n selected direction : "+ to +"₩n₩n"); //나아갈 방향 to에 저장
int k=0;
while(true)
{
        if(to==list.get(k).child|| k==list.size())
        {from = list.get(k).parent;                    //저장된 node 중 나아갈 방향
인 node를 찾음
        break;                                                              //
부모 노드를 from에 저장.
        }
```

```
                k++;
        }
        this.start[from] = this.start[to];                    //swap(from,to);
        this.start[to]=-1;     //최종 swap;
        }

        if(duplicate.size()>4)
                duplicate.remove(1);
        else if(this.start.equals(duplicate.poll()))
        {
                return; //무한 루프 시 빠지기..
        }
        else
                duplicate.add(this.start);
        //중복 체크 4개만 저장. 4개 전의 상태가 반복 시 무한루프 탈출.
}
public int heuristic (){
        int node[] = new int[9];
        int i,j;
        for(i=0 ; i<9 ; i ++)
                node[i]=0; //
        int costtogoal=0;
        int totalcost=0;
        int shortest = -1;
        int selected=0;
        //list.get(0);
        int k=0;
        while(true){                                    //각 노드에 대해 코스트 가장 적은 것 찾는 과정.
                if(list.isEmpty()||k==list.size())
```

```
                    break;
                else{
                    costtogoal=0;
                    //node에 임시 카피
                    int a;
                    for(a=0;a<9;a++)
                    node[a] = this.start[a];

                    node[list.get(k).child] = node[list.get(k).parent];
                node[list.get(k).parent]= -1; //임시 node를 만든 다음 start를 넣고 스왑시켜봄.

                    //costtogoal 계산.
                    for(j =0 ; j< 9; j++)
                        if(node[j]!= goal[j])
                            costtogoal++;
                    totalcost = depth + costtogoal;          //total cost는 (depth)+(<현재-
목표>상태가 다른 정도)

                //최적의 길 찾기. 토탈코스트가 가장 적은 것 중 가장 빨리 찾아지는 것.
                    if(shortest ==-1||shortest>totalcost)
                    {
                        shortest = totalcost;
                        selected = list.get(k).child;
                    }
                    k++;
                }
            }

            return selected;
```

```
}
public int findEmpty(){
        int i;
        for(i=0; i<9 ; i++)
                if(start[i]==-1)
                        return i;
        return -1;
}


public static void main(String args [])
{       int i;
        lastpuzzle puzzle = new lastpuzzle();

        int check=0;
        for( i =0 ; i< 9 ; i ++)
        {
                if(i%3 ==0)
                        System.out.println();
                if(puzzle.start[i]==-1)
                        System.out.print(" ");
                else
                        System.out.print(puzzle.start[i]);
        }
        System.out.println();
        while(true){
                check=0;
                System.out.println("");
                for(i=0; i< 9; i ++)
```

```
                    if(puzzle.start[i]!=puzzle.goal[i])
                            check=1;
                if(check ==0){
                    System.out.println(" we find goal!!");
                    break;
                }

                puzzle.search(); // 수정 필요.
                //mapping tile.

                for( i =0 ; i< 9 ; i ++)
                {
                    if(i%3 ==0)
                            System.out.println();
                    if(puzzle.start[i]==-1)
                            System.out.print(" ");
                    else
                            System.out.print(puzzle.start[i]);
                }
                System.out.println("₩n depth = "+ puzzle.depth);
            }
        }
} }
```

# 출력

```
<terminated> lastpuzzle [Java Application] C:₩Program Files₩Java₩jre1.8.0_151₩bin₩javaw.exe (2017. 6. 5. 오후 10:29:20)
132
 78
564

empty = 3


 selected direction : 4


132
7 8
564
 depth = 1

empty = 4


 selected direction : 1


1 2
738
564
```

처음 시작 부분.

```
empty = 2


 selected direction : 1


1 3
824
765
 depth = 116

empty = 1


 selected direction : 4


123
8 4
765
 depth = 117

 we find goal!!
```

찾는 부분.

# Discussion

우선 앞에서 말했듯이 a*를 처음 짤 때 recursive로 짜려고 했기 때문에 감을 못 잡아서, 각 나아가는 방향이 어떻게 depth정보를 count하는지 고민하여야 했다. 그래서 각 depth를 count하지않고 차례만 지키면 되는 BFS를 사용한 puzzle 알고리즘을 짜려고 했다. BFS로 짜려고 했을 때는 각 깊이마다 차례대로 구현할 수 있도록 Queue를 사용하여 구현하려 했는데, 버퍼 문제로 인해 지속적으로 스택이 터지는 현상이 벌어졌다. 이를 해결하기 위해 추정치 값을 어느정도 이용하여 bfs를 짜려했지만, queue의 내용들을 모두 비교하기 위해서는 큐의 값들을 모두 꺼냈다가 다시 넣어야 하기 때문에 사용하지 않고 다시 a*로 짰다.

앞에서 말한 depth에 대한 문제는 main에서 search를 지속적으로 콜함으로써 한번의 루프당 한번의 깊이가 추가되게 구현하였다. 또한 a*알고리즘을 공부하면서 total cost값이 같은 것들은 따로 저장하면서 병렬적으로 해당 깊이의 방향들이 동시적으로 search되는데, 그에 대해 병렬적으로 다시 리스트를 만들어 구현하기에는 힘들어서, 따로 그 부분은 구현하지 못하고 가장 처음 search되는 부분을 선택하였다. 따라서 만약 방향이 잘못 선택될 시에는 무한 루프나 많이 돌아가야 하는 상황이 발생할 수 있었다.

또한 무한 루프를 막기위해 중복된 이전 정보를 저장하여야 하는데, 우선 최근 4개의 map 정보를 저장하

고 그것을 지속적으로 search되는 값들과 비교하여 만약 같으면 return하게 하였다. (아쉽게도 이전 깊이로 돌아가는 backtracking은 구현을 하지 못하였고, return을 사용하여 돌아가게만 하였다.

방향에 대해서는 이전에 움직였던 방향의 정보를 기억해 돌아가는 것을 막고자 하였다. 이를 위해 direction이라는 변수를 따로 놔두었다.