ΟΡΙΣΜΟΣ ΛΑΒΥΡΙΝΘΟΥ

Για τον ορίσμο του λαβυρίνθου δημιουργήσαμε μια κλάση(*labyrinth.java)* η οποια ειναι υπεύθυνη για την αρχικοποίηση των κελιών του λαβυρίνθου και των τιμών τους.

```
public class Labyrinth{
    private int[][] lab;
    private int arraySize;

    public Labyrinth(int arraySize) {
        Random rand = new Random();
        this.arraySize=arraySize;
        int max = 4;
        lab = new int[arraySize][arraySize];
        for(int i= 0; i<arraySize ;i++){
            for(int j= 0; j<arraySize ;j++){
                //Random number from 0.0-1.0
                float p=rand.nextFloat();
                //Random number from 0-3 adding 1 is random number from 1-4
                int value = rand.nextInt(max)+1;
                if(p<=0.1){
                    lab[i][j]=-1;
                }
                else{
                    lab[i][j]=value;
                }
            }
        }
    }
```

Με τη βοήθεια της συνάρτηση rand δημιουργούμε μια μεταβλητη τύπου float για να δουμε αν στο συγκεκριμένο κελί θα εχουμε καποιο εμπόδιο(πιθανότητα 10%) και στη συνέχεια προσθέτουμε ενα random value σε καθε κελι που δεν περιέχει εμπόδιο.

ΚΛΑΣΗ MOVEMENT

Η κλάση *movement.java* δημιουργήθηκε για να μας επιστρέφει το κόστος της μετακίνησης του robot απο το ένα κελί στο άλλο.

```java
public class Movement {
    private int x1;
    private int y1;
    private int x2;
    private int y2;
    private boolean typeOfMovement;
    public Movement(int x1, int y1,int x2, int y2){
        this.x1=x1;
        this.y1=y1;
        this.x2=x2;
        this.y2=y2;
        if(x1==x2 ||y1==y2){
            typeOfMovement= true;
        }
        else{
            typeOfMovement= false;
        }
    }
    public double movementCost(Labyrinth lab){
        double cost=0;
        int val1 =lab.getValue(x1,y1);
        int val2 =lab.getValue(x2,y2);
        if(typeOfMovement){
            if(val1>=val2){
                cost=val1-val2+1;
            }
            else{
                cost=val2-val1+1;
            }
        }
        else{
            if(val1>=val2){
                cost=val1-val2+0.5;
            }
            else{
                cost=val2-val1+0.5;
            }
        }
        return cost;
    }
```

ΚΛΑΣΗ NODE

Η κλάση node είναι υπεύθυνη για την διαχείρηση των κόμβων των δέντρων που δημιουργούνται κατα την εκτέλεση των αλγορίθμων . Περιέχει constructors για την εκτέλεση των αλγορίθμων A* και UCS ,τους αντιστοιχους getters για πρόσβαση στα στοιχεία καθως και μια συνάρτηση findNeighbours η οποία βρίσκει τα γειτονικά στοιχεια του καθε κόμβου.

```
public class Node {
    private double totalCost;
    private double pathCost;
    private String path;
    private int [][] neighbours;
    private int posX;
    private int posY;
    private Node parent;
    private double value;
    private int arraySize;
    private Labyrinth lab;
    private int rows=0;
    private double movementCost;
    private String name;
```

```
    public void findNeighbours(){
        int startPosX = (posX - 1 < 0) ? posX : posX-1;
        int startPosY = (posY - 1 < 0) ? posY : posY-1;
        int endPosX =   (posX + 1 > arraySize-1) ? posX : posX+1;
        int endPosY =   (posY + 1 > arraySize-1) ? posY : posY+1;

        int neighbour[][]= new int [8][2];
        rows=0;

        for (int rowNum=startPosX; rowNum<=endPosX; rowNum++) {
            for (int colNum=startPosY; colNum<=endPosY; colNum++) {
                if(!(rowNum==posX && colNum==posY)){
                    if(lab.getValue(rowNum,colNum)!=(-1)){
                        neighbour[rows][0]=rowNum;
                        neighbour[rows][1]=colNum;
                        rows++;
                    }
                }
            }
        }
        neighbours = new int[rows][2];
        for(int i=0; i<rows; i++){
            for(int j=0; j<2; j++){
                neighbours[i][j]=neighbour[i][j];
            }
        }
    }
```

```
    public int getNeighbourRows(){
        return rows;
    }

    public int getNeighbourX(int i) {
        return neighbours[i][0];
    }
    public int getNeighbourY(int i) {
        return neighbours[i][1];
    }

    public String getPath() {
        return path;
    }
    public double getPathCost() {
        return pathCost;
    }
    public double getTotalCost() {
        return totalCost;
    }
    public int getX(){
        return posX;
    }
    public int getY(){
        return posY;
    }

    public String getName() {
        return name;
    }
}
```

ΑΛΓΟΡΙΘΜΟΣ UNIFORMCOSTSEARCH

Για τον αλγόριθμο UCS χρησιμοποιήσαμε 2 hashmaps

```
private HashMap<String,Node> openNodes = new HashMap<String,Node>();
private HashMap<String,String> closedNodes = new HashMap<String,String>();
```

Εκ των οποίων το πρώτο αποθηκεύει τους «ανοιχτούς» κόμβους του δέντρου, δηλαδή τους κόμβους τους οποίους δεν εχει εξετάσει άκομα ο αλγόριθμος και το δεύτερο το οποίο αποθηκεύει τους κλείστους κομβους, δηλαδή τους κόμβους τους οποίους εχει εξετάσει ο αλγόριθμος.

Η κλάση UniformCostSearch.java περιέχει 3 συναρτήσεις την StartTheUcs() που αρχικοποιεί το δεντρο, το String με τα πατήματα του αλγορίθμου και τον πρώτο κομβο (με το στοιχειο S)

```
public void startUCS(){
    Node s = new Node(lab, sx, sy, parent: null, Name(sx, sy));
    openNodes.put(s.getName(),s);
    expansion= s.getName();
    createTree(s);
    doTheUCS(s);
}
```

Την createTree που δέχεται ως όρισμα έναν κόμβο και τσεκάρει αν είναι ενας απο τους κόμβους στόχους αλλίως τον τοποθετεί στους κλειστούς κόμβους και βάζει στους ανοιχτούς τους γειτονές του

```
public void createTree(Node a){
    if(a.getName().equals(Name(g1x, g1y))){
        openNodes.clear();
        printPath( end: "G1",a);
        return;
    }
    else if(a.getName().equals(Name(g2x, g2y))){
        openNodes.clear();
        printPath( end: "G2",a);
        return;
    }
    closedNodes.put(a.getName(),a.getPath());
    openNodes.remove(a.getName());
    int nRows = a.getNeighbourRows();
    for(int i=0;i<nRows;i++){
            int x = a.getNeighbourX(i);
            int y = a.getNeighbourY(i);
            Node cur = new Node(lab, x, y, a, Name(x, y));
            if(!closedNodes.containsKey(Name(x, y))){
                if(!openNodes.containsKey(Name(x, y))){
                    openNodes.put(Name(x, y), cur);
                }
                else{
                    Node prev = openNodes.get(Name(x, y));
                    if(cur.getPathCost() < prev.getPathCost()){
                        openNodes.replace(Name(x, y),cur,prev);
                    }
                }
            }
    }
    return;
}
```

Και την doTheUcs που ειναι η συνάρτηση που αναδρομικά τρέχει τον αλγόριθμο

```
public void doTheUCS(Node prev){
    double cost=prev.getPathCost();
    Node   next = prev;
    while(openNodes.size()>0){
       for(String key : openNodes.keySet()){
            Node cur = openNodes.get(key);
            if(cost==prev.getPathCost()){
                if(next==prev){
                    cost = cur.getPathCost();
                    next = cur;
                }

            }
            else if(cost>cur.getPathCost()){
                cost = cur.getPathCost();
                next = cur;
            }

       }
       expansion += " " + next.getName();
       createTree(next);
       doTheUCS(next);
    }
}
```

Στην Node.java ο constructor υπεύθυνος για τον UCS είναι:

```
public Node(Labyrinth lab,int x, int y, Node parent, String name){
    posX = x;
    posY = y;
    this.lab=lab;
    this.parent=parent;
    this.name=name;
    arraySize=lab.getArraySize();
    findNeighbours();
    if(!(parent==null)){
        path = parent.getPath()+" , "+ name;
        Movement move= new Movement(parent.getX(), parent.getY(), x, y);
        movementCost = move.movementCost(lab);
        pathCost = parent.getPathCost()+movementCost;
    }
    else{
        path = name;
        movementCost=0;
        pathCost = 0;
    }
}
```

Αλγοριθμος Α*

Για τον αλγόριθμο Α* αρχίκα φτιάξαμε την ευρετική του συνάρτηση. Δημιουργήσα με μια κλάση Heurestic.java:

```
public class Heuristic {
    private int g1x;
    private int g1y;
    private int g2x;
    private int g2y;

    public Heuristic(int g1x,int g1y,int g2x,int g2y){
        this.g1x=g1x;
        this.g1x=g1y;
        this.g2x=g2x;
        this.g2y=g2y;
    }
    public double findCost(int x,int y){
        double a=0;
        double b=0;
        double c=0;
        double d=0;

        if(x-g1x<0){
            if(y-g1y<0){
                a=g1x-x;
                b=g1y-y;
            }
            else{
                a=g1x-x;
                b=y-g1y;
            }
        }
        else{
            if(y-g1y<0){
                a=x-g1x;
                b=g1y-y;
            }
            else{
                a=x-g1x;
                b=y-g1y;
            }
        }
        if(x-g2x<0){
            if(y-g2y<0){
                c=g2x-x;
                d=g2y-y;
            }
            else{
                c=g2x-x;
                d=y-g2y;
            }
        }
```

```
        else{
            if(y-g2y<0){
                c=x-g2x;
                d=g2y-y;
            }
            else{
                c=x-g2x;
                d=y-g2y;
            }
        }
        double g1 = calcCost(a,b);
        double g2 = calcCost(c,d);
        double cost=0;
        if(g1>g2){
            cost = g2;
        }
        else{
            cost = g1;
        }
        return cost;
    }


    public double calcCost(double a , double b){
        double straight=0;
        double diagonal=0;

        if(a>b){
            straight=a-b;
            diagonal=a-straight;
        }
        else{
            straight=b-a;
            diagonal=b-straight;
        }
        return (diagonal*0.5 + straight);
    }
```

Η οποία βρίσκει το συντομότερο μονοπάτι προς τον κοντινότερο κόμβο στόχο ανέξαρτήτως εμποδίων.Το βάρος (cost) των πατημάτων είναι ίσο με την ελάχιστη τίμη που μπόρει να πάρει κατα την μετακίνηση(δηλαδή 1 για κάθετες και οριζόντιες μετακινήσεις και 0.5 για διαγώνιες)

Κλαση A*

Τέλος δήμιουργήσαμε την κλάση A_Star.java για την υλοποίηση του αλγορίθμου A* η οποία δουλεύει με παρόμοιο τρόπο με την UCS.java με την διαφόρα ότι το κριτήριο τερματισμού της είναι η πρώτη εμφάνιση κόμβου στόχου και του constructor στην συνάρτηση Node.java

```
public Node(Labyrinth lab,int x, int y, Node parent, String name, double h){
    posX = x;
    posY = y;
    this.lab=lab;
    this.parent=parent;
    this.name=name;
    arraySize=lab.getArraySize();
    findNeighbours();
    if(!(parent==null)){
        path = parent.getPath()+" , "+ name;
        Movement move= new Movement(parent.getX(), parent.getY(), x, y);
        movementCost = move.movementCost(lab);
        pathCost = parent.getPathCost()+movementCost;
        totalCost = parent.getPathCost()+movementCost + h;
    }
    else{
        path = name;
        movementCost=0;
        pathCost = 0;
        totalCost = h;
    }
}
```

```java
import java.util.HashMap;

public class A_Star {
    private HashMap<String,Node> openNodes = new HashMap<String,Node>();
    private HashMap<String,String> closedNodes = new HashMap<String,String>();
    private String expansion;
    private Labyrinth lab;
    private int sx;
    private int sy;
    private int g1x;
    private int g1y;
    private int g2x;
    private int g2y;
    private Heuristic h;

    public A_Star(Labyrinth lab,int sx,int sy,int g1x,int g1y,int g2x,int g2y){
        this.lab=lab;
        this.sx =sx;
        this.sy=sy;
        this.g1x=g1x;
        this.g1y=g1y;
        this.g2x=g2x;
        this.g2y=g2y;
        h = new Heuristic(g1x, g1y, g2x, g2y);
    }
    public void startA_Star(){
        Node s= new Node(lab, sx, sy, parent: null, Name(sx, sy),h.findCost(sx, sy));
        openNodes.put(s.getName(),s);
        expansion= s.getName();
        createTree(s);
        doTheA_Star(s);

    }
```

```java
public void doTheA_Star(Node prev){
    double cost=prev.getTotalCost();
    Node   next = prev;
    while(openNodes.size()>0){
       for(String key : openNodes.keySet()){
            Node cur = openNodes.get(key);
            if(cost==prev.getTotalCost()){
                if(next==prev){
                    cost = cur.getTotalCost();
                    next = cur;
                }

            }
            else if(cost>cur.getTotalCost()){
                cost = cur.getTotalCost();
                next = cur;
            }

        }
        expansion += " " + next.getName();
        createTree(next);
        doTheA_Star(next);
    }
}
```

```java
public void createTree(Node a){
    closedNodes.put(a.getName(),a.getPath());
    openNodes.remove(a.getName());

    int nRows = a.getNeighbourRows();
    for(int i=0;i<nRows;i++){
            int x = a.getNeighbourX(i);
            int y = a.getNeighbourY(i);
            Node cur = new Node(lab, x, y, a, Name(x, y),h.findCost(x, y));

            if(cur.getName().equals(Name(g1x,g1y))){
                openNodes.clear();
                expansion += " " + cur.getName();
                printPath( end: "G1", cur);
                return;
            }
            else if(cur.getName().equals(Name(g2x,g2y))){
                openNodes.clear();
                expansion += " " + cur.getName();
                printPath( end: "G2", cur);
                return;
            }

            if(!closedNodes.containsKey(Name(x, y))){
                if(!openNodes.containsKey(Name(x, y))){
                    openNodes.put(Name(x, y), cur);
                }
                else{
                    Node prev = openNodes.get(Name(x, y));
                    if(cur.getTotalCost() < prev.getTotalCost()){
                        openNodes.replace(Name(x, y),cur,prev);
                    }
                }
            }
    }
    return;
}
```

Η Main

Για να τρέξουμε τον κωδικά μας φτιάξαμε μια κλάση Project.java η οποία εχει τον ρολο της main