

# AI REPORT 1

Presented by Ioanna Avanidi(4977) & Ioannis kokkinis(5109)



## Εισαγωγή

Το ζητούμενο αυτής της εργασίας είναι η υλοποίηση ενός αλγορίθμου αναζήτησης για την πλοήγηση ενός ρομπότ σε έναν λαβύρινθο. Στόχος του αλγορίθμου είναι η εύρεση της βέλτιστης διαδρομής από ένα αρχικό σημείο(**S**) σε ένα τελικό σημείο(**G**). Το ρομπότ μας μπορεί να κινείται από και προς όλες τις κατευθύνσεις στις οποίες υπάρχει ελεύθερο γειτονικό κελί (αριστερά, δεξιά, πάνω, κάτω και διαγώνια) με κόστος 1 για κάθε κίνηση, ενώ μπορεί να κάνει και μια επιπλέον κίνηση από το κάτω αριστερό κελί στο πάνω δεξί κελί (και αντίστροφα) με κόστος 2. Η υλοποίηση έγινε με 2 διαφορετικές μεθόδους:

- Με αναζήτηση Ομοιόμορφου κόστους (UCS)
- Με αναζήτηση A\* με χρήση της βέλτιστης αποδεκτής ευρετικής συνάρτησης.

## Περιγραφή Υλοποίησης

### Αρχικοποίηση βασικών μεταβλητών

Η υλοποίηση έχει γίνει σε Java και η κλάση που υλοποιεί τους 2 αλγόριθμους είναι η `robotLab`. Μέσα σε αυτή την κλάση αρχικοποιούμε τα βασικά πεδία:

- **n** : το μέγεθος  $n \times n$  του πίνακα

```
private int n; //size of the array
```

- **p** : πιθανότητα εμποδίου

```
private double p; //propability to display an obstacle
```

- **labyrinth** : Ο λαβύρινθος

```
/**Create labyrinth with spots  
 *  
 * Free -> 0  
 * Reserved -> 1  
 *  
 */  
private int[][] labyrinth;
```

- **S** : Το αρχικό σημείο που βρίσκεται το ρομπότ

```
private int[] S = new int[2]; //start point
```

- **G** : Το τελικό σημείο στόχος που θα πάει το ρομπότ

```
private int[] G = new int[2]; //Goal Point
```

- **spotA** : Το σημείο του κάτω αριστερού κελιού του πίνακα

```
private int[] spotA = new int[2]; //position A for transition A->B or B->A
```

- **spotB** : Το σημείο του πάνω δεξιά κελιού του πίνακα

```
private int[] spotB = new int[2]; //position B for transition A->B or B->A
```

- **heurValues** : Πίνακας που περιέχει τις ευρετικές τιμές για κάθε σημείο

```
//for Heuristic values
private int[][] heurValues;
```

Για να αρχικοποιήσουμε αυτά τα πεδία, ζητάμε από τον χρήστη να δώσει τα απαιτούμενα δεδομένα.

```
//Give the n input
int n = 0;
try {
    System.out.print(s:"Give size N: ");
    n = scanner.nextInt();
    if (n<0){
        System.out.println(x:"Please give positive number for size N");
        return;
    }
} catch (InputMismatchException e1) {
    System.out.println(x:"Please give integer number for size N");
    return;
}
```

```
//Take probability
int p = 0;
try {
    System.out.print(s:"Give probability P: ");
    p = scanner.nextInt();
    if (p<0 || p>100){
        System.out.println(x:"The probability must be between 0-100%");
        return;
    }
} catch (InputMismatchException e2) {
    System.out.println(x:"The probability must be between 0-100%");
    return;
}
```

```

//Give the Start posotion
int spotS_X = 0;
int spotS_Y = 0;
System.out.println(x:"\nSet start position of robot");
try {
    System.out.print(s:"Give position X: ");
    spotS_X = scanner.nextInt();
    if (spotS_X<0 || spotS_X>(n-1)){
        System.out.println("Please give X between {0-"+(n-1)+"}");
    }
} catch (InputMismatchException e3) {
    System.out.println("Please give X between {0-"+(n-1)+"}");
    return;
}
try {
    System.out.print(s:"Give position Y: ");
    spotS_Y = scanner.nextInt();
    if (spotS_Y<0 || spotS_Y>(n-1)){
        System.out.println("Please give Y between {0-"+(n-1)+"}");
    }
} catch (InputMismatchException e4) {
    System.out.println("Please give Y between {0-"+(n-1)+"}");
    return;
}
}

```

```

//Give the Goal position
System.out.println(x:"\nSet Goal position of robot");
int spotG_X = 0;
int spotG_Y = 0;

try {
    System.out.print(s:"Give position X: ");
    spotG_X = scanner.nextInt();
    if (spotG_X<0 || spotG_X>100){
        System.out.println("Please give X between {0-"+(n-1)+"}");
    }
} catch (Exception e) {
    System.out.println("Please give X between {0-"+(n-1)+"}");
}

try {
    System.out.print(s:"Give position Y: ");
    spotG_Y = scanner.nextInt();
    if (spotG_Y<0 || spotG_Y>100){
        System.out.println("Please give Y between {0-"+(n-1)+"}");
    }
} catch (Exception e) {
    System.out.println("Please give Y between {0-"+(n-1)+"}");
}
}

```

Έπειτα με την χρήση του constructor της κλάσης robotLab αρχικοποιούμε τα απαραίτητα πεδία.

```

public robotLab(int updateN,int updateP, int spotS_X, int spotS_Y, int spotG_X, int spotG_Y){
    this.n = updateN;
    this.p = updateP/100.0;
    this.labyrinth = new int[updateN][updateN];
    this.heurValues = new int[updateN][updateN];

    //set spot A and spot B
    spotA[0] = updateN-1;
    spotA[1] = 0;

    spotB[0] = 0;
    spotB[1] = updateN-1;

    //set the Start and Goal point
    S[0] = spotS_X;
    S[1] = spotS_Y;

    G[0] = spotG_X;
    G[1] = spotG_Y;
}

```

```

robotLab RL = new robotLab(n, p, spotS_X, spotS_Y, spotG_X, spotG_Y);

```

Το βασικό μας αντικείμενο είναι το Node με το οποίο φτιάχνουμε το δένδρο και εκτελούμε τους αλγόριθμους UCS και A\*

## Κατασκευή Λαβυρίνθου

Ο λαβύρινθός μας είναι, ένας πίνακας με  $N \times N$  κελιά κάποια από τα οποία είναι ελεύθερα ενώ κάποια άλλα περιέχουν εμπόδια και δεν είναι επισκέψιμα.

Δημιουργούμε τον πίνακα, και τον γεμίζουμε με 0(ελεύθερο κελί) και 1(κελί με εμπόδιο). Τα εμπόδια δημιουργούνται με βάση την πιθανότητα  $p$  που ορίζουμε στην αρχή του προγράμματος. Για κάθε θέση του πίνακα που διατρέχουμε, ορίζουμε μια μεταβλητή **possibility** με πεδίο ορισμού  $[0,1]$ . Όταν η τιμή της μεταβλητής είναι μικρότερη της τιμής του  $p$ , στο συγκεκριμένο κελί υπάρχει εμπόδιο και είναι απροσπέλαστο. Η τιμή της **possibility** δίνεται τυχαία μέσω της συνάρτησης **random**. Προσέχουμε τα κουτάκια που αντιστοιχούν στο αρχικό και το τελικό σημείο, καθώς και τα κουτάκια στην κάτω αριστερή και στην πάνω δεξιά γωνία, να μην περιέχουν εμπόδια, για να μπορούν να γίνουν όλες οι δυνατές μετακινήσεις.

```

private void gerateRandomObstacles(){
    Random rand = new Random();

    for(int i=0; i<n; i++){
        for (int j=0; j<n; j++){
            if ( (i==S[0]) && (j==S[1]) || (i==G[0]) && (j==G[1]) ||
                (i==spotA[0]) && (j==spotA[1]) || (i==spotB[0]) && (j==spotB[1]) ){
                continue;
            }
            double possibility = rand.nextInt(bound:101)/100.0;
            if (possibility<=p){
                labyrinth[i][j] = 1;
            }else{
                labyrinth[i][j] = 0;
            }
        }
    }
}

```

## Μετακίνηση Ρομπότ

Από την εκφώνηση, γίνεται κατανοητό ότι υπάρχουν τριών ειδών μετακινήσεις, οι οριζόντιες, οι κάθετες και οι διαγώνιες, ενώ υπάρχει και η ειδική μετακίνηση από τη κάτω δεξιά στην πάνω αριστερή γωνιά. Η φιλοσοφία για την υλοποίηση των κινήσεων είναι ίδια και στις 10 περιπτώσεις. Αναθέτουμε σε μία καινούρια μεταβλητή, τις τιμές της θέσης του ρομπότ(x, y). Ανάλογα με τη φορά και το είδος της κίνησης ελέγχουμε αν η κίνηση είναι επιτρεπτή, δηλαδή αν με την πραγματοποίησή της, το ρομπότ παραμένει εντός των ορίων του λαβυρίνθου. Για παράδειγμα, για την οριζόντια κίνηση με φορά προς τα δεξιά, ελέγχουμε αν η y συντεταγμένη του ρομπότ είναι μικρότερη από το οριζόντιο μέγεθος του πίνακα. Ελέγχουμε επιπλέον αν το κελί στο οποίο γίνεται η μετακίνηση είναι εμπόδιο καθώς και αν ανήκει στη λίστα με κελιά που έχει ήδη επισκεφθεί (είναι δηλαδή προγονός του), έτσι ώστε να το αποφύγει. Ο έλεγχος αυτός γίνεται μέσω της μεθόδου isPrec της κλάσης Node.

```

public boolean isPrec(int[] checkPos){

    for(Node precNode: predecessors){
        if((precNode.getPos()[0]==checkPos[0]) && (precNode.getPos()[1]==checkPos[1])){
            return true;
        }
    }

    return false;
}

```

## Τελεστές Δράσης

Ορίζουμε 9 τελεστές δράσεις για να περιγράψουμε τις κινήσεις που μπορεί να κάνει το ρομπότ.

- **moveRight:** Μετακίνηση του ρομπότ μια θέση δεξιά

Προϋποθέσεις: Το κελί να μην είναι εμπόδιο και εκτός ορίων

Αποτέλεσμα: Το ρομπότ μετακινείται στο κελί (x, y+1)

```
private int[] moveRight(Node v){
    int[] checkPos = new int[2];
    checkPos[0] = v.getPos()[0];
    checkPos[1] = v.getPos()[1]+1;

    if ((checkPos[1]>9) || v.isPrec(checkPos) || (labyrinth[checkPos[0]][checkPos[1]]==1)){
        checkPos[0] = -1;
        checkPos[1] = -1;
        return checkPos;
    }

    return checkPos;
}
```

- **moveLeft:** Μετακίνηση του ρομπότ μια θέση αριστερά

Προϋποθέσεις: Το κελί να μην είναι εμπόδιο και εκτός ορίων

Αποτέλεσμα: Το ρομπότ μετακινείται στο κελί (x, y-1)

```
private int[] moveLeft(Node v){
    int[] checkPos = new int[2];
    checkPos[0] = v.getPos()[0];
    checkPos[1] = v.getPos()[1]-1;

    if ((checkPos[1]<0) || v.isPrec(checkPos) || (labyrinth[checkPos[0]][checkPos[1]]==1)){
        checkPos[0] = -1;
        checkPos[1] = -1;
        return checkPos;
    }

    return checkPos;
}
```

- **moveUp:** Μετακίνηση του ρομπότ μια θέση πάνω

Προϋποθέσεις: Το κελί να μην είναι εμπόδιο και εκτός ορίων

Αποτέλεσμα: Το ρομπότ μετακινείται στο κελί (x-1, y)

```
private int[] moveUp(Node v){
    int[] checkPos = new int[2];
    checkPos[0] = v.getPos()[0]-1;
    checkPos[1] = v.getPos()[1];

    if ((checkPos[0]<0) || v.isPrec(checkPos) || (labyrinth[checkPos[0]][checkPos[1]]==1)){
        checkPos[0] = -1;
        checkPos[1] = -1;
        return checkPos;
    }

    return checkPos;
}
```

- **moveDown:** Μετακίνηση του ρομπότ μια θέση κάτω

Προϋποθέσεις: Το κελί να μην είναι εμπόδιο και εκτός ορίων

Αποτέλεσμα: Το ρομπότ μετακινείται στο κελί (x+1, y)

```
private int[] moveDown(Node v){
    int[] checkPos = new int[2];
    checkPos[0] = v.getPos()[0]+1;
    checkPos[1] = v.getPos()[1];

    if ((checkPos[0]>9) || v.isPrec(checkPos) || (labyrinth[checkPos[0]][checkPos[1]]==1)){
        checkPos[0] = -1;
        checkPos[1] = -1;
        return checkPos;
    }

    return checkPos;
}
```

- **moveUpRight:** Μετακίνηση του ρομπότ μια θέση πάνω και δεξιά

Προϋποθέσεις: Το κελί να μην είναι εμπόδιο και εκτός ορίων

Αποτέλεσμα: Το ρομπότ μετακινείται στο κελί (x-1, y+1)

```
private int[] moveUpRight(Node v){
    int[] checkPos = new int[2];
    checkPos[0] = v.getPos()[0]-1;
    checkPos[1] = v.getPos()[1]+1;

    if ((checkPos[0]<0) || (checkPos[1]>9) || v.isPrec(checkPos) || (labyrinth[checkPos[0]][checkPos[1]]==1)){
        checkPos[0] = -1;
        checkPos[1] = -1;
        return checkPos;
    }

    return checkPos;
}
```

- **moveDownRight:** Μετακίνηση του ρομπότ μια θέση κάτω και δεξιά

Προϋποθέσεις: Το κελί να μην είναι εμπόδιο και εκτός ορίων

Αποτέλεσμα: Το ρομπότ μετακινείται στο κελί  $(x+1, y+1)$

```
private int[] moveDownRight(Node v){
    int[] checkPos = new int[2];
    checkPos[0] = v.getPos()[0]+1;
    checkPos[1] = v.getPos()[1]+1;

    if ((checkPos[0]>9) || (checkPos[1]>9) || v.isPrec(checkPos) || (labyrinth[checkPos[0]][checkPos[1]]==1)){
        checkPos[0] = -1;
        checkPos[1] = -1;
        return checkPos;
    }

    return checkPos;
}
```

- **moveUpLeft:** Μετακίνηση του ρομπότ μια θέση πάνω και αριστερά

Προϋποθέσεις: Το κελί να μην είναι εμπόδιο και εκτός ορίων

Αποτέλεσμα: Το ρομπότ μετακινείται στο κελί  $(x-1, y-1)$

```
private int[] moveUpLeft(Node v){
    int[] checkPos = new int[2];
    checkPos[0] = v.getPos()[0]-1;
    checkPos[1] = v.getPos()[1]-1;

    if ((checkPos[0]<0) || (checkPos[1]<0) || v.isPrec(checkPos) || (labyrinth[checkPos[0]][checkPos[1]]==1)){
        checkPos[0] = -1;
        checkPos[1] = -1;
        return checkPos;
    }

    return checkPos;
}
```

- **moveDownLeft:** Μετακίνηση του ρομπότ μια θέση κάτω και αριστερά

Προϋποθέσεις: Το κελί να μην είναι εμπόδιο και εκτός ορίων

Αποτέλεσμα: Το ρομπότ μετακινείται στο κελί  $(x+1, y-1)$

```
private int[] moveDownLeft(Node v){
    int[] checkPos = new int[2];
    checkPos[0] = v.getPos()[0]+1;
    checkPos[1] = v.getPos()[1]-1;

    if ((checkPos[0]>9) || (checkPos[1]<0) || v.isPrec(checkPos) || (labyrinth[checkPos[0]][checkPos[1]]==1)){
        checkPos[0] = -1;
        checkPos[1] = -1;
        return checkPos;
    }

    return checkPos;
}
```

- **checkIfNodeOnSpotAOrSpotB:** Μετακίνηση του ρομπότ από το σημείο A στο σημείο B ή μετακίνηση του ρομπότ από το σημείο B στο σημείο A.

Προϋποθέσεις: Το κελί να μην είναι εμπόδιο και εκτός ορίων

Αποτέλεσμα: Το ρομπότ μετακινείται στο κελί που βρίσκεται το σημείο B ή στο σημείο A.

```
private int[] checkIfNodeOnSpotAOrSpotB(Node v){  
  
    int[] checkPos = new int[2];  
  
    if((v.getPos()[0] == spotA[0]) && (v.getPos()[1] == spotA[1])){  
        checkPos[0] = spotB[0];  
        checkPos[1] = spotB[1];  
        return checkPos;  
    }else{  
        if ((v.getPos()[0] == spotB[0]) && (v.getPos()[1] == spotB[1])){  
            checkPos[0] = spotA[0];  
            checkPos[1] = spotA[1];  
            return checkPos;  
        }else{  
            checkPos[0] = -1;  
            checkPos[1] = -1;  
            return checkPos;  
        }  
    }  
}
```

## Συνάρτηση πραγματικού κόστους( $g(n)$ )

Ορίζουμε ότι το κόστος κάθε κίνησης είναι ίσο με 1, ενώ το κόστος της ειδικής κίνησης από το A στο B σημείο και πίσω είναι 2.

$g(n)$ : Άθροισμα βαρών των ακμών από την αφετηρία ως τον κόμβο  $n$

Στον κώδικά μας, η συνάρτηση πραγματικού κόστους υλοποιείται μέσω του πεδίου `cost` που υπάρχει σε κάθε αντικείμενο **Node**. Το πεδίο αυτό αποθηκεύει το συνολικό κόστος της διαδρομής από την αφετηρία προς τον αντίστοιχο κόμβο. Η τιμή του `cost` υπολογίζεται κάθε φορά που δημιουργείται ένα νέο παιδί κόμβος, μέσα στη συνάρτηση `generateKids()`. Δεδομένου ότι κάθε μετακίνηση από έναν κόμβο στον επόμενο, έχει κόστος ίσος με 1 και στην ειδική περίπτωση των σημείων A και B έχει κόστος ίσο με 2, το κόστος κάθε παιδιού υπολογίζεται :  $g(n) = (1 \text{ ή } 2 \text{ ανάλογα με την περίπτωση}) + (\text{κόστος του γονέα } v)$ . Με αυτό τον τρόπο χρησιμοποιείται η συνάρτηση πραγματικού κόστους μέσα στον κώδικα.

```
int newCost = v.getCost()+1;
```

```
int newCost = v.getCost()+2;
```

```
Node kid = new Node(nodeName,newCost,putPos);
```

## Ευρετική συνάρτηση( $h^*(n)$ ):

Δημιουργούμε, με τη χρήση της απόστασης Chebyshev, έναν πίνακα ευρετικών τιμών, όπου κάθε τιμή αντιστοιχεί στην απόσταση του συγκεκριμένου στοιχείου από τον τελικό κόμβο. Επειδή στην περίπτωση μας, έχουμε και την ειδική περίπτωση της μετακίνησης από το σημείο A στο σημείο B ή και το αντίστροφο, πρέπει να διαμορφώσουμε τον Chebyshev πίνακα.

Αρχικά, υπολογίζουμε την Chebyshev απόσταση όπως ακριβώς είναι ο τύπος.

```
//chebyshev distance to goal
int chebyshevDistanceGoal = Math.max(Math.abs(G[0] - i), Math.abs(G[1] - j));
```

Έπειτα εξετάζουμε το μονοπάτι από το κόμβο στο σημείο A, μετά στο σημείο B και από το σημείο B στον τελικό κόμβο. Με τον τροποποιημένο τύπο Chebyshev η απόσταση αυτή υπολογίζεται με τον παρακάτω τρόπο

```
//chebyshev distance to A->B->Goal
int toA = Math.max(Math.abs(spotA[0] - i), Math.abs(spotA[1] - j));
int fromB = Math.max(Math.abs(G[0] - spotB[0]), Math.abs(G[1] - spotB[1]));
int AtoBtoG = toA + 2 + fromB;
```

Με την ίδια ακριβώς λογική υπολογίζουμε και το μονοπάτι από το κόμβο στο σημείο B, μετά στο σημείο A και από το σημείο A στον τελικό κόμβο.

```
//chebyshev distance B->A->Goal
int toB = Math.max(Math.abs(spotB[0] - i), Math.abs(spotB[1] - j));
int fromA = Math.max(Math.abs(G[0] - spotA[0]), Math.abs(G[1] - spotA[1]));
int BtoAtoG = toB + 2 + fromA;
```

Τέλος για να υπολογίσουμε την ευρετική τιμή για το συγκεκριμένο σημείο, θα πάρουμε το ελάχιστο από τα 3 μονοπάτια

```
heurValues[i][j] = Math.min(chebyshevDistanceGoal, Math.min(AtoBtoG, BtoAtoG));
```

Η ευρετική συνάρτηση είναι παραδεκτή για τους εξής λόγους: Επιστρέφει τιμή ίση με το μηδέν όταν ο κόμβος αντιστοιχεί στον τελικό στόχο, όπως απαιτείται. Στην ειδική περίπτωση με τις μετακινήσεις A προς B ή B προς A, υπολογίζουμε όλες τις chebyshev αποστάσεις, εξετάζοντας όλα τα πιθανά μονοπάτια και επιλέγοντας το μικρότερο κόστος. Το πιο σημαντικό όμως είναι ότι, για κάθε κόμβο  $n$ , η ευρετική τιμή  $h^*(n)$  είναι μικρότερη ή ίση του  $h(n)$  (δηλαδή  $h^*(n) \leq h(n)$ ). Αυτό εξασφαλίζει ότι η ευρετική συνάρτηση είναι παραδεκτή.

```
private void GenerateHeurTable(){
    for (int i=0; i<n; i++){
        for (int j=0; j<n; j++){
            if (i==G[1] && j==G[0]) {
                heurValues[i][j]=0;
            }

            //chebyshev distance to goal
            int chebyshevDistanceGoal = Math.max(Math.abs(G[0] - i), Math.abs(G[1] - j));

            //chebyshev distance to A->B->Goal
            int toA = Math.max(Math.abs(spotA[0] - i), Math.abs(spotA[1] - j));
            int fromB = Math.max(Math.abs(G[0] - spotB[0]), Math.abs(G[1] - spotB[1]));
            int AtoBtoG = toA + 2 + fromB;

            //chebyshev distance B->A->Goal
            int toB = Math.max(Math.abs(spotB[0] - i), Math.abs(spotB[1] - j));
            int fromA = Math.max(Math.abs(G[0] - spotA[0]), Math.abs(G[1] - spotA[1]));
            int BtoAtoG = toB + 2 + fromA;

            heurValues[i][j] = Math.min(chebyshevDistanceGoal, Math.min(AtoBtoG, BtoAtoG));
        }
    }
}
```

## Υλοποίηση με αναζήτηση ομοιόμορφου κόστους (UCS)

Η UCS είναι μια στρατηγική αναζήτησης που βασίζεται στο συνολικό κόστος διαδρομής από ένα αρχικό σε ένα τελικό κόμβο. Σε θεωρητικό επίπεδο ο UCS βρίσκει πάντα λύση αν υπάρχει (είναι δηλαδή πλήρης αλγόριθμος) και ταυτόχρονα εγγυάται ότι η λύση που θα βρει είναι η βέλτιστη, αν το κόστος όλων των κινήσεων είναι θετικό.

Ο UCS σε κάθε βήμα, επιλέγει να αναπτύξει, εκείνο τον κόμβο που έχει το μικρότερο κόστος διαδρομής.

Σε ότι αφορά την υλοποίησή μας, έχουμε 2 βασικές λίστες:

- Μια λίστα με κόμβους, προς επίσκεψη, ταξινομημένη κατά κόστος, από δω και πέρα θα την ονομάζουμε ανοιχτή (στον κώδικα μας `openList`)
- Μια λίστα με κόμβους που έχουμε ήδη επισκεφθεί, από δω και πέρα θα την ονομάζουμε κλειστή (στον κώδικα μας `closeList`)

```
//For UCS and A* algorithms
List<Node> openList = new ArrayList<Node>();
List<Node> closeList = new ArrayList<Node>();
```

Στον κώδικα μας η συνάρτηση `UCS` επιστρέφει μία λίστα `List<String>` που περιέχει το μονοπάτι που βρήκε (Αν υπάρχει).

```
//this list is about path
List<String> pathToGoal = new ArrayList<String>();
```

Αρχικά δημιουργούμε ένα `Node root` που αντιπροσωπεύει την ρίζα του δένδρου που βρίσκεται στον χώρο αναζήτησης με κόστος 0. Τον προσθέτουμε στην λίστα `openList` που περιέχει τους κόμβους προς επίσκεψη. Επίσης, ορίζουμε το όνομα του κόμβου προκειμένου να το εκτυπώσουμε αργότερα, ως μέρος του μονοπατιού που ακολουθούμε για τον στόχο.

```
String nodeName = "(" + S[0] + ", " + S[1] + ")";
Node root = new Node(nodeName, updateCost:0, S);
openList.add(root);
```

Έπειτα ο αλγόριθμός UCS αναλαμβάνει τις παρακάτω διαδικασίες μέχρι να βρει το βέλτιστο μονοπάτι ή μέχρι η λίστα `openList` να γίνει κενή:

Αφαιρούμε από την `openList`, τον κόμβο `v`, με το μικρότερο κόστος.

```
removeVfromOpenList(v);
```

Προσθέτουμε τον κόμβο `v` που αφαιρέσαμε, στην λίστα `closeList`

```
closeList.add(v); //add it to close list
```

Δημιουργούμε τα παιδιά του κόμβου `v` με την συνάρτηση `generateKids()`. Η συνάρτηση αυτή ελέγχει όλες τις κινήσεις με την χρήση τελεστών δράσης που μπορεί να κάνει το ρομπότ και για κάθε κίνηση δημιουργεί ένα **Node** με όνομα την θέση του σημείο που μπορεί να επισκεφτεί. Υπολογίζει το κόστος του καινούριου κόμβου, με τον τρόπο της συνάρτησης πραγματικού κόστους που περιγράψαμε παραπάνω και το αναθέτει στο πεδίο `cost`. Επίσης, θέτει ως γονέα και προκάτοχο του καινούργιου κόμβου, το κόμβο `v` και στον κόμβο `v`, θέτει ως παιδί του τον καινούργιο κόμβο.

```

private Node generateKids(Node v, int ASTAR){
    int[] putPos = new int[2];
    /*
    * STEP RIGHT
    *
    */
    putPos = moveRight(v);
    if ((putPos[0] != -1) && (putPos[1] != -1)){
        //generate kid here
        String nodeName = "("+putPos[0]+","+putPos[1]+")";
        int newCost = v.getCost()+1;
        Node kid = new Node(nodeName,newCost,putPos);
        if(ASTAR==1){
            int herValue = herValues[putPos[0]][putPos[1]];
            kid.setHerValue(herValue);
        }
        kid.setParent(v);
        /**set the predecessors */
        for(Node precNode: v.getPrec()){
            kid.addPredec(precNode);
        }
        kid.addPredec(v);
        v.addKid(kid);
    }
}

```

```

/**
 *
 * STEP LEFT
 *
 */

putPos = moveLeft(v);
if ((putPos[0] != -1) && (putPos[1] != -1)){
    //generate kid here
    String nodeName = "("+putPos[0]+","+putPos[1]+")";
    int newCost = v.getCost()+1;
    Node kid = new Node(nodeName,newCost,putPos);
    if(ASTAR==1){
        int herValue = heurValues[putPos[0]][putPos[1]];
        kid.setHerValue(herValue);
    }
    kid.setParent(v);
    /**set the predecessors */
    for(Node precNode: v.getPrec()){
        kid.addPredec(precNode);
    }
    kid.addPredec(v);
    v.addKid(kid);
}

```

```

/**
 *
 * STEP DOWN
 *
 */

putPos = moveDown(v);
if ((putPos[0] != -1) && (putPos[1] != -1)){
    //generate kid here
    String nodeName = "("+putPos[0]+","+putPos[1]+")";
    int newCost = v.getCost()+1;
    Node kid = new Node(nodeName,newCost,putPos);
    if(ASTAR==1){
        int herValue = heurValues[putPos[0]][putPos[1]];
        kid.setHerValue(herValue);
    }
    kid.setParent(v);
    /**set the predecessors */
    for(Node precNode: v.getPrec()){
        kid.addPredec(precNode);
    }
    kid.addPredec(v);
    v.addKid(kid);
}

```

```

/**
 *
 * STEP UP
 *
 */
putPos = moveUp(v);
if ((putPos[0] != -1) && (putPos[1] != -1)){
    //generate kid here
    String nodeName = "("+putPos[0]+","+putPos[1]+")";
    int newCost = v.getCost()+1;
    Node kid = new Node(nodeName,newCost,putPos);
    if(ASTAR==1){
        int herValue = heurValues[putPos[0]][putPos[1]];
        kid.setHerValue(herValue);
    }
    kid.setParent(v);
    /**set the predecessors */
    for(Node precNode: v.getPrec()){
        kid.addPredec(precNode);
    }
    kid.addPredec(v);
    v.addKid(kid);
}

```

```

/**
 *
 * MOVE UP-RIGHT BOX
 *
 */

putPos = moveUpRight(v);
if ((putPos[0] != -1) && (putPos[1] != -1)){
    //generate kid here
    String nodeName = "("+putPos[0]+","+putPos[1]+")";
    int newCost = v.getCost()+1;
    Node kid = new Node(nodeName,newCost,putPos);
    if(ASTAR==1){
        int herValue = herValues[putPos[0]][putPos[1]];
        kid.setHerValue(herValue);
    }
    kid.setParent(v);
    /**set the predecessors */
    for(Node precNode: v.getPrec()){
        kid.addPredec(precNode);
    }
    kid.addPredec(v);
    v.addKid(kid);
}

```

Εάν ο κόμβος  $v$  που βγάλαμε από την **openList** είναι ο κόμβος στόχος , τότε η συνάρτηση τερματίζει και επιστρέφει το μονοπάτι με την χρήση της συνάρτησης **generatePath()** .

```
if(isGoal(v)){  
    /**  
     *  
     * make the return path  
     *  
     */  
    System.out.println(x:"FOUND GOAL!");  
    System.out.println("TOTAL COST: "+ v.getCost());  
    pathToGoal=v.generatePath();  
  
    //clear for next algorithm  
    openList = new ArrayList<Node>();  
    closeList = new ArrayList<Node>();  
    return pathToGoal;  
}
```

Αλλιώς επαναλαμβάνουμε τις ακόλουθες ενέργειες για κάθε παιδί  $u$  του κόμβου  $v$ :

```
List<Node> kids = v.getKids();
```

Ελέγχουμε αν το παιδί  $u$  ανήκει στην **openList** ή στην **closeList** με την χρήση της συνάρτησης **isInOpenOrCloseList()** που επιστρέφει τον κόμβο ή την τιμή -1 στην μεταβλητή **checkNodeFromLists**.

Αν το **checkNodeFromLists** έχει κόμβο και το **cost** του παιδιού  $u$  είναι μεγαλύτερο ή ίσο του **cost** του **checkNodeFromLists** τότε διαγράφουμε το παιδί. Αν το **checkNodeFromLists** έχει κόμβο και το **cost** του παιδιού  $u$  δεν είναι μεγαλύτερο ή ίσο του **cost** του **checkNodeFromLists** τότε αφαιρούμε το παιδί  $u$  από την λίστα που ανήκει (είτε στην **openList** είτε στην **closeList**) και προσθέτουμε το παιδί  $u$  στην ανοιχτή λίστα **openList**. Αν το **checkNodeFromLists** δεν έχει κόμβο τότε σημαίνει ότι το παιδί  $u$  δεν είναι μέσα σε καμία λίστα και το προσθέτουμε στην ανοιχτή λίστα **openList**.

```

for(int i=0; i<kids.size(); i++){
    Node checkNodeFromLists = isInOpenOrCloseList(kids.get(i));
    if ((checkNodeFromLists.getCost()!=-1)){
        /* node is in on open or close list */
        if (kids.get(i).getCost()>=checkNodeFromLists.getCost()){
            v.removeKid(kids.get(i));
            i--;
        }else{
            //remove from kids
            removeFromOpenOrCloseList(kids.get(i));
            openList.add(kids.get(i));
        }
    }else{
        /* node not in on open or close list*/
        openList.add(kids.get(i));
    }
}
}

```

Σε περίπτωση που η ανοιχτή λίστα `openList` είναι κενή , τότε σημαίνει ότι δεν υπάρχουν άλλοι κόμβοι να επισκεφτούμε, άρα δεν υπάρχει μονοπάτι προς τον κόμβο στόχο και έτσι σταματάμε την εκτέλεση του UCS.

```

//if open list is empty then there is no possible way
if(openList.isEmpty()){
    break;
}

```

Έπειτα, επιλέγουμε τον επόμενο κόμβο από την ανοιχτή λίστα `openList` με το μικρότερο κόστος και τον αναθέτουμε στην μεταβλητή `v` με σκοπό να επισκεφτούμε τον κόμβο αυτόν και να εκτελέσουμε πάλι τις παραπάνω διαδικασίες. Η επιλογή του επόμενου κόμβου `v` γίνεται με την συνάρτηση `pickTheNextNode()` όπου με βάση το επόμενο κόστος  $f(n) = g(n)$ .

Μόλις ολοκληρωθεί ο αλγόριθμος , τότε επιστρέφει την λίστα `pathToGoal` που περιέχει το μονοπάτι προς τον κόμβο στόχο.

```

private List<String> UCS(){
    //this list is about path
    List<String> pathToGoal = new ArrayList<String>();

    String nodeName = "("+S[0]+","+S[1]+")";
    Node root = new Node(nodeName, updateCost:0, S);
    openList.add(root);

    Node v = root;
    while(!(openList.isEmpty())){
        removeVfromOpenList(v);
        //open kids here
        v = generateKids(v,ASTAR:0);
        closeList.add(v); //add it to close list

        if(isGoal(v)){
            /**
             *
             * make the return path
             *
             */
            System.out.println(x:"FOUND GOAL!");
            System.out.println("TOTAL COST: "+ v.getCost());
            pathToGoal=v.generatePath();

            //clear for next algorithm
            openList = new ArrayList<Node>();
            closeList = new ArrayList<Node>();
            return pathToGoal;
        }

        List<Node> kids = v.getKids();

        for(int i=0; i<kids.size(); i++){
            Node checkNodeFromLists = isInOpenOrCloseList(kids.get(i));
            if ((checkNodeFromLists.getCost()!=-1)){
                /* node is in on open or close list */
                if (kids.get(i).getCost()>=checkNodeFromLists.getCost()){
                    v.removeKid(kids.get(i));
                    i--;
                }else{
                    //remove from kids
                    removeFromOpenOrCloseList(kids.get(i));
                    openList.add(kids.get(i));
                }
            }else{

```

```

    }else{
        /* node not in on open or close list*/
        openList.add(kids.get(i));
    }
}

//if open list is empty then there is no possible way
if(openList.isEmpty()){
    break;
}

//Select the minimun using UCS f(n)
v = openList.get(pickTheNextNode(UCS_OR_ASTAR:0));
}

pathToGoal.add(e:"PATH NOT FOUND!");

//clear for next algorithm
openList = new ArrayList<Node>();
closeList = new ArrayList<Node>();
return pathToGoal;
}

```

```

/* f(N) = g(n) */

min = openList.get(index:0).getCost();
for (int i=0; i<openList.size(); i++){
    if (openList.get(i).getCost()<min){
        min = openList.get(i).getCost();
        index = i;
    }
}
}

```

## Υλοποίηση με αναζήτηση A\*

Η συνάρτηση A\* υλοποιείται με τον ίδιο αλγόριθμο με την UCS με κύρια διαφορά ότι ο υπολογισμός του κόστους για την επιλογή επόμενου κόμβου, γίνεται και με την χρήση της ευρετικής τιμής.

Για αρχή όπως και στον UCS αρχικοποιούμε μία λίστα `List<String> pathToGoal` που θα κρατάει το μονοπάτι όταν ολοκληρωθεί ο αλγόριθμος.

```
List<String> pathToGoal = new ArrayList<String>();
```

Έπειτα, ακριβώς όπως με τον UCS, φτιάχνουμε έναν αντικείμενο `Node` που είναι ο `root`, περνάμε τις κατάλληλες παραμέτρους και τον προσθέτουμε στην ανοιχτή λίστα `openList`.

```
String nodeName = "(" + S[0] + ", " + S[1] + ")";  
Node root = new Node(nodeName, updateCost:0, S);  
root.setHeurValue(heurValues[S[0]][S[1]]);  
openList.add(root);
```

Στην συνέχεια όσο η ανοιχτή λίστα δεν είναι κενή εκτελεί τις παρακάτω ενέργειες:

Αφαιρεί τον κόμβο `v` από την ανοιχτή λίστα `openList`, παράγει τα παιδιά του και τοποθετεί τον κόμβο `v` στην κλειστή λίστα `closeList`.

```
removeVfromOpenList(v);  
//open kids here  
v = generateKids(v, ASTAR:1);  
closeList.add(v); //add it to close list
```

Έπειτα, αν ο κόμβος `v` είναι ο κόμβος στόχος, τότε τερματίζει ο αλγόριθμος, φτιάχνουμε το μονοπάτι με την χρήση της συνάρτησης `generatePath()` και επιστρέφουμε το μονοπάτι

```
if(isGoal(v)){  
    /**  
    *  
    * make the return path  
    *  
    */  
    System.out.println(x:"FOUND GOAL!");  
    System.out.println("TOTAL COST: "+ v.getCost());  
    pathToGoal=v.generatePath();  
    return pathToGoal;  
}
```

Για κάθε παιδί  $u$ , εξετάζουμε αν είναι μέσα στην ανοιχτή λίστα `openList` ή στην κλειστή λίστα `closeList`. Αν ανήκει σε κάποια λίστα, τότε παίρνουμε τον κόμβο που βρέθηκε στην λίστα και να τον αποθηκεύουμε στην μεταβλητή `checkNodeFromLists`. Στην συνέχεια, αν το `cost` του παιδί  $u$  είναι μεγαλύτερο ή ίσο του `cost` του κόμβου `checkNodeFromLists`, τότε αφαιρούμε το παιδί  $u$  από τον κόμβο  $v$ . Αν το κόστος είναι μικρότερο, αφαιρούμε τον κόμβο από την ανοιχτή ή την κλειστή λίστα και τον προσθέτουμε στην ανοιχτή λίστα.

Στην περίπτωση που δεν υπάρχει το παιδί  $u$  σε καμία λίστα, προσθέτουμε το παιδί  $u$  στην ανοιχτή λίστα ώστε να είναι υποψήφιο για επίσκεψη αργότερα.

```

for(int i=0; i<kids.size(); i++){

    Node checkNodeFromLists = isInOpenOrCloseList(kids.get(i));
    if ((checkNodeFromLists.getCost()!=-1)){
        /* node is in on open or close list */
        if (kids.get(i).getCost()>=checkNodeFromLists.getCost()){
            v.removeKid(kids.get(i));
            i--;
        }else{
            //remove from kids
            removeFromOpenOrCloseList(kids.get(i));
            openList.add(kids.get(i));
        }
    }else{
        /* node not in on open or close list*/
        openList.add(kids.get(i));
    }
}
}

```

Όπως ακριβώς και στην UCS, αν η λίστα είναι κενή τότε σταματάμε τον αλγόριθμο.

```

//if open list is empty then there is no possible way
if(openList.isEmpty()){
    break;
}

```

Η επιλογή του επόμενου κόμβου  $v$  από την ανοιχτή λίστα γίνεται με την συνάρτηση `pickTheNextNode()` όπως και στην UCS. Η κύρια διαφορά εδώ είναι ότι η επιλογή του επόμενου κόμβου γίνεται με τον τύπο  $f(n) = g(n) + h^*(n)$ . Υπολογίζει το κόστος που έχει πληρώσει ο κόμβος μέχρι εκεί και με την ευρετική συνάρτηση υπολογίζει το κόστους του στον επόμενο κόμβο. Έτσι, επιλέγουμε τον επόμενο κόμβο που πρέπει να επισκεφτούμε.

```

//if open list is empty then there is no possible way
if(openList.isEmpty()){
    break;
}

//Select the minimum using A* f(n)
v = openList.get(pickTheNextNode(UCS_OR_ASTAR:1));

/* f(n) = g(n) + h(n) */

min = openList.get(index:0).getCost()+openList.get(index:0).getHerValue();

for (int i=0; i<openList.size(); i++){
    if ((openList.get(i).getCost()+openList.get(i).getHerValue()) < min){
        min = (openList.get(i).getCost()+openList.get(i).getHerValue());
        index = i;
    }
}

```

Με αυτό τον τρόπο ολοκληρώθηκε η υλοποίηση μας, περιλαμβάνοντας τους αλγόριθμους A\* και UCS. Από την εκτέλεση και την σύγκριση των αποτελεσμάτων, παρατηρήσαμε ότι ο αλγόριθμος A\* είναι πιο αποδοτικός από την UCS, κυρίως λόγω της χρήσης ευρετικής συνάρτησης. Η ευρετική συνάρτηση επιτρέπει στον A\* για να κατευθύνει την αναζήτηση πιο στοχευμένα προς τον στόχο, μειώνοντας σημαντικά τον αριθμό των επεκτάσεων. Οι αλγόριθμοι UCS και A\* βρίσκουν την ίδια βέλτιστη λύση, απλά ο A\* αλγόριθμος το επιτυγχάνει με λιγότερο υπολογιστικό κόστος.

Παρακάτω σας δείχνουμε κάποια ενδεικτικά παραδείγματα εκτέλεσης:

```
user@user-IdeaPad-3-15ADA05: ~/Documents/AI/Project/AI-Project/lab1
user@user-IdeaPad-3-15ADA05:~/Documents/AI/Project/AI-Project/lab1$ java robotLab
Give size N: 10
Give probability P: 20

Set start position of robot
Give position X: 0
Give position Y: 0

Set Goal position of robot
Give position X: 5
Give position Y: 5
{
S | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | B |
0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
1 | 0 | 0 | 0 | 1 | G | 0 | 0 | 1 | 0 |
0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
A | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
}
UCS :
FOUND GOAL!
TOTAL COST: 6
PATH:

(0,0) --> (0,1) --> (1,2) --> (2,2) --> (3,3) --> (4,4) --> (5,5) --> END

A* :
FOUND GOAL!
TOTAL COST: 6
PATH:

(0,0) --> (0,1) --> (1,2) --> (2,2) --> (3,3) --> (4,4) --> (5,5) --> END
user@user-IdeaPad-3-15ADA05:~/Documents/AI/Project/AI-Project/lab1$
```



```
user@user-IdeaPad-3-15ADA05: ~/Documents/AI/Project/AI-Project/lab1
1 1 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 1 1 0 0
1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0
0 0 0 0 0 0 1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 1 1 1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
A 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
}
UCS :
FOUND GOAL!
TOTAL COST: 22
PATH:
(0,0) --> (1,1) --> (2,2) --> (3,3) --> (4,4) --> (5,5) --> (6,6) --> (7,7) --> (8,8) --> (8,9) --> (9,10) --> (10,11) --> (11,12) --> (12,13) --> (13,14) --> (13,15) --> (14,16) --> (15,16)
--> (16,16) --> (17,17) --> (18,18) --> (19,19) --> (20,20) --> END

A* :
FOUND GOAL!
TOTAL COST: 22
PATH:
(0,0) --> (1,1) --> (2,2) --> (3,3) --> (4,4) --> (5,5) --> (6,6) --> (7,7) --> (8,8) --> (9,9) --> (9,10) --> (10,11) --> (11,12) --> (12,13) --> (13,14) --> (14,14) --> (15,14) --> (16,15)
--> (16,16) --> (17,17) --> (18,18) --> (19,19) --> (20,20) --> END

user@user-IdeaPad-3-15ADA05:~/Documents/AI/Project/AI-Project/lab15
```