NTNU

Norwegian University of
Science and Technology

Artificial intelligence

IE502014

Assignment 1

Author:
Nicolò Pinciroli

February 2, 2020

1 Single-player challenge on CodinGame

1.1 Code

```
import java.util.*;
import java.io.*;
import java.math.*;

class Coordinate {
    private int x;
    private int y;

    public Coordinate(int x, int y) { // defines a coordinate
        (x, y)
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }

    public int hashCode(){
        int hashcode = 0;
        hashcode = x+y*1000;
        return hashcode;
    }

    public boolean equals(Object obj){ // implements the
       equality for two elements of the coordinate class
        if (obj instanceof Coordinate) {
            Coordinate coo = (Coordinate) obj;
            return (coo.getY() == this.y && coo.getX() ==
               this.x);
        } else {
            return false;
        }
    }

    public int distance(Object x) { // calculates the (
       Manhattan) distance between two coordinates
```

```
        if(x instanceof Coordinate) {
            Coordinate a = (Coordinate) x;
            return Math.abs(a.getX() - this.getX()) + Math.
                abs(a.getY() - this.getY());
        }
        return -1;
    }

    public double correctedDistance(Object x) {
        if(x instanceof Coordinate) {
            Coordinate a = (Coordinate) x;
            return Math.abs(a.getX() - this.getX()) + Math.
                abs(a.getY() - this.getY())+0.2*Math.signum(a.
                getX() - this.getX())+0.2*Math.signum(a.getY()
                 - this.getY());
        }
        return -1;
    }

    public String toString() { // can bse used to print a
        coordinate, mainly for debug purposes
        return "<x, y> = <" + this.getX() + ", " + this.getY
            () + ">";
    }
}

class Player {

    private static Vector<String> matrix = new Vector<String
        >(); // the game field, containing the labyrinth
    private static Vector<Coordinate> questionMarks = new
        Vector<Coordinate>(); // question marks locations

    private static Coordinate controlRoom = new Coordinate
        (-1,-1);
    // the position of the control room
    // Note: the initial coordinate of the control room is
        (-1, -1) since all the positions inside the labyrinth
    // are positive

    private static boolean runAway = false;
    // runAway becomes true when the control room has been
        reached and
    // the countdown has started. In such a situation Kirk
        has to find the
```

```
// best path to reach the starting position

private static Coordinate startingPoint = new Coordinate
    (-1,-1);
// The starting point is also inizialized using (-1,-1),
    but its value
// is changed immediately when the game loop starts

private static Vector<Coordinate> pathToC = new Vector<
    Coordinate>();
// pathToC is a vector that contains the path to the
    control room

// pathFromHere is used to store the path from a given
    position to
// the control room
private static Vector<Coordinate> pathFromHere = new
    Vector<Coordinate>();

// flag that will be used to determine the correct
    behaviour
// when the control room is discovered and reached
private static boolean firstFindC = true;

// cumQM is used to store the total number of question
    marks found along
// a path. This is useul because in this way the best
    choice can be selected,
// when the control room cannot be reached
private static HashMap<Coordinate, Integer> cumQM = new
    HashMap<Coordinate, Integer>();

// distPath is used to store the distance of a node from
    the source node
// during a research. It would be possible to introduce a
     correction that
// gives a different weight to the nodes that Kirk has
    already visited/discovered
private static HashMap<Coordinate, Integer> distPath =
    new HashMap<Coordinate, Integer>();

private static Vector<Coordinate> visitedNodes = new
    Vector<Coordinate>();
// visitedNodes contains the list of all the nodes Kirk
    has visited so far
```

```
private static void printMatrix() {
    for(String row:matrix) {
        System.err.println(row);
    }
}

private static char what(Coordinate xy) {
    try {
        char out = matrix.get(xy.getY()).charAt(xy.getX()
            );
        return out;
    }
    catch(Exception e) {
        // If the coordinate is outside of the field,
            then
        // it is equivalent to a wall, since it cannot be
             reached
        return '#';
    }
}

private static boolean isControlRoom(Coordinate position)
    {
    return what(position) == 'C';
}

 private static boolean isHollow(Coordinate position) {
    return what(position) == '.' || what(position) == 'T'
        ;
}

public static Coordinate seeRight(Coordinate coor) {
    int x = coor.getX();
    int y = coor.getY();
    Coordinate out = new Coordinate(x+1, y); // right =
        increase the x coordinate
    return out;
}

public static Coordinate seeLeft(Coordinate coor) {
    int x = coor.getX();
    int y = coor.getY();
    Coordinate out = new Coordinate(x-1, y); // left =
        decrease the x coordinate
```

```
    return out;
}

public static Coordinate seeUp(Coordinate coor) {
    int x = coor.getX();
    int y = coor.getY();
    Coordinate out = new Coordinate(x, y-1); // up =
        decrease the y coordinate
    return out;
}

public static Coordinate seeDown(Coordinate coor) {
    int x = coor.getX();
    int y = coor.getY();
    Coordinate out = new Coordinate(x, y+1); // down =
        decrease the y coordinate
    return out;
}

private static void moveKirk(Coordinate from, Coordinate
    to) {

    visitedNodes.add(from);
    visitedNodes.add(to);

     if(from.getX() == to.getX()) { // Kirk moves
         vertically

         if(from.getY() == to.getY() + 1) {
             System.out.println("UP");
         }
         else if(from.getY() == to.getY() - 1) {
             System.out.println("DOWN");
         }
     }
    else if(from.getY() == to.getY()) { // Kirk moves
        horizontally
         if(from.getX() == to.getX() + 1) {
             System.out.println("LEFT");
         }
         else if(from.getX() == to.getX() - 1) {
             System.out.println("RIGHT");
         }
    }
}
```

```
private static Vector<Coordinate> canBeWalked(int R, int
    C) {
     // This method returns a vector containing all the
        cells on which
     // it is possible to walk, since they are hollow (or
        the starting position)
     Vector<Coordinate> walkable = new Vector<Coordinate
        >();
     for(int i=0; i<R; ++i) {
         for(int j=0; j<C; ++j) {
             if(isHollow(new Coordinate(j, i))) {
                 walkable.add(new Coordinate(j, i));
             }
         }
     }
     return walkable;
}


private static Vector<Coordinate> getQuestionNeighbours(
    Coordinate position) {
     Vector<Coordinate> neighbours = new Vector<Coordinate
        >();

      for(int i=position.getX()-1;i<=position.getX()+1;++i
         ) {
         for(int j=position.getY()-1;j<=position.getY()
            +1;++j) {
             if(position.distance(new Coordinate(i, j))
                ==1) {
                 if(isQuestion(new Coordinate(i, j))) {
                     neighbours.add(new Coordinate(i, j));
                 }
             }
         }
     }
     return neighbours;
}

private static Vector<Coordinate>
    getExtendedQuestionNeighbours(Coordinate position) {
     Vector<Coordinate> neighbours = new Vector<Coordinate
        >();
```

```
        for(int i=position.getX()-2;i<=position.getX()+2;++i
            ) {
            for(int j=position.getY()-2;j<=position.getY()
                +2;++j) {
                    if(isQuestion(new Coordinate(i, j))) {
                        neighbours.add(new Coordinate(i, j));

                    }
            }
        }

        return neighbours;
    }

    private static Vector<Coordinate> getWalkableNeighbours(
        Coordinate position) {
        Vector<Coordinate> neighbours = new Vector<Coordinate
            >();

        for(int i=position.getX()-1;i<=position.getX()+1;++i)
            {
            for(int j=position.getY()-1;j<=position.getY()
                +1;++j) {
                if(position.distance(new Coordinate(i, j))
                    ==1) {
                    if(isHollow(new Coordinate(i, j))) {
                        neighbours.add(new Coordinate(i, j));
                    }
                }
            }
        }
        return neighbours;
    }

    private static Map<Coordinate, Coordinate> BFS(Coordinate
        source, int R, int C, boolean considerC) {

        // Breadth-first-search algorithm to find the best
            path from a position
        // to all the others

        // Returns the distances from the source to all the
            other reachable cells
        // as a list of predecessors
```

```
Map<Coordinate, Coordinate> predecessors = new
    HashMap<Coordinate, Coordinate>();

// adjacent is used to store the adjacent vertices
    with respect to a
// given vertex (the position under analysis at a
    given moment)
Vector<Coordinate> adjacent = new Vector<Coordinate
    >();

// This data structure is used to associate a flag to
     every coordinate
// that can be possibily visited (the hollow cells)
// Once a cell has been considered, its flag goes to
    true
HashMap<Coordinate, Boolean> flag = new HashMap<
    Coordinate, Boolean>();

// The queue used by the BFS algorithm
Queue<Coordinate> Q = new LinkedList<Coordinate>();

// This is the vector containing all the coordinates
    that represent
// hollow cells, independently from the fact that
    they are reachable.
// In a graph representation the elements of this
    vector are the
// nodes of the graph
Vector<Coordinate> canBeWalked = new Vector<
    Coordinate>();

// The vector containing the cells that have been
    visited by the BFS
// algorithm (not necessarily by Kirk). The set of
    cells visited by BFS
// is a subset of the hollow cells whose content is
    known, but there is the possibiliy
// that Kirk hasn't visited directly the considered
    cells.
Vector<Coordinate> visited = new Vector<Coordinate>()
    ;

cumQM.clear();
distPath.clear();
```

```
// Get all the hollow cells discovered so far
canBeWalked = canBeWalked(R, C);

 if(considerC) { // if the control room has already
     been visited
     canBeWalked.add(controlRoom);
}
// Note: in case the control room has already been
    visited it is always not convenient
// for Kirk to go there another time, since a loop
    would occur. However, this coordinate
// has been considered since Kirk starts from there,
    and therefore it is part of his
// best path

for(Coordinate v:canBeWalked) {
     flag.put(v, false);

     // A position v can have some predecessors, that
         allow to reconstruct
     // the best path found by BFS. Before the actual
         execution of BFS, however,
     // there aren't paths and, therefore, there aren'
         t predecessors (yet)
     predecessors.put(v, null);
}

flag.put(source, true); // Since Kirk starts from the
     source, it has been visited
Q.add(source); // add the source to the BFS queue

// associate the number of question marks nearby to
    the source node
cumQM.put(source, getExtendedQuestionNeighbours(
    source).size());

// The distance of the source from itself is 0
distPath.put(source, 0);

while(!Q.isEmpty()) { // Not all the reachable
    positions have been considered yet
     Coordinate v = Q.poll(); // extract the position
         that entered the queue first
     adjacent = getWalkableNeighbours(v); // get the
         adjacent positions that can be visited
```

```
 if(considerC && (isControlRoom(seeDown(v)) ||
     isControlRoom(seeUp(v)) || isControlRoom(
     seeRight(v)) || isControlRoom(seeLeft(v)))) {
     adjacent.add(controlRoom); // and add the
         control room if allowed
}

for(Coordinate w:adjacent) { // for every
    adjacent position
     if(flag.get(w) == false) { // if that
         position has not been visited yet

          flag.put(w, true); // now it has been
              visited
          predecessors.put(w, v); // its
              predecessor is the last position
              having been extracted
          Q.add(w); // this new position can be
              added to the queue

          // associate to that position a value
              that is the sum of the
          // predecessor value and the question
              marks nearby
          cumQM.put(w, cumQM.get(v) +
              getExtendedQuestionNeighbours(w).size
              ());

          // the distance from the source is
              determined by the predecessor's
          // distance and by the fact that the node
               had been visited (by Kirk!)
          // previously
          distPath.put(w, distPath.get(v) + 1 +
              visitedNode(v));

          // Note: visitedNode can be used to
              change the distance on the basis
          // of Kirk's previous behaviour, but in
              this implementation visitiedNode
          // always return 0. It has been kept in
              the code for any future implementation
          // and/or optimization
     }
}
```

```
    }
    return predecessors;

}

private static int visitedNode(Coordinate v) {
    for(Coordinate c:visitedNodes) {
        if(v.equals(c)) {
            return 0;
        }
    }
    return 0;
}

private static boolean isQuestion(Coordinate coor) {
    return what(coor) == '?';
}

private static void findQuestionMarks(int R, int C) {
    for(int i=0; i<R; ++i) {
        for(int j=0; j<C; ++j) {
            if(isQuestion(new Coordinate(j, i))) {
                questionMarks.add(new Coordinate(j, i));
            }
        }
    }
}

private static boolean isBorderDot(Coordinate coor) {
    Vector<Coordinate> neighbours = getQuestionNeighbours
       (coor);
    if(neighbours.size()==0) {
        return false;
    }
    else {
        return true;
    }
}

private static Vector<Coordinate> findBestChoice(
   Coordinate currentPosition, int R, int C) {

    // This method returns the path to the best choice
       from the current position
    // The best choice is determined by considering some
```

```
    parameters related to the
// number of question marks and to the distance.
// In particular, the goal is to maximize the number
    of discovered question
// marks in the shortest time possible (=> shortest
    distance traveled bt Kirk)

// List of predecessors given a position
Map<Coordinate, Coordinate> pr = new HashMap<
    Coordinate, Coordinate>();

// Path from the current position to the best choice
Vector<Coordinate> path = new Vector<Coordinate>();

// The best choice hasn't been found yet
boolean found = false;

// The parameter used to determine the quality of a
    choice is initially
// set to - infinity, since the highest value
    corresponds to the best choice
Float tempNum = Float.NEGATIVE_INFINITY;

// The best choice is outside the field, but it will
    be set during the fist iteration
Coordinate bestChoice = new Coordinate(-1, -1);

// Gets the list of predecessors reachable from the
    current position
pr = BFS(currentPosition, R, C, false);

for (Map.Entry<Coordinate, Coordinate> entry : pr.
    entrySet()) {
     if(entry.getValue() != null) {
         // A border dot is an hollow cell that has at
             least one
         // question mark as a neighbour. It is part
             of the fringe, if
         // this structured is considered as a graph
         if(isBorderDot(entry.getKey())) {
             // Calculation of the "score" related to
                 the considered cell
             float num = cumQM.get(entry.getKey())-12*
                 distPath.get(entry.getKey());
```

```
                // See if this score is better than the
                    previous highest score...
                if(num > tempNum) {
                    // ...and in case update the highest
                        score and
                    // the best choice coordinates
                    bestChoice = entry.getKey();
                    tempNum = num;
                }
            }
        }
    }

    // Now, given the best choice, find the path to it
    Coordinate pathElement = bestChoice;
    while(!found) {
        path.add(pathElement);
        pathElement = pr.get(pathElement);
        // Note: it is possible to use == because equals
            has been implemented
        // in the Coordinate class
        if(pathElement == currentPosition) {
            found = true;
        }
    }
    path.add(currentPosition);
    return path;
}

private static boolean isThereC(int R, int C) {
    // This methods checks if the control room location
        has been
    // found by Kirk (this does not necessarily mean that
         such a
    // position is reachable by Kirk)
    for(int i=0; i<R; ++i) {
        for(int j=0; j<C; ++j) {
            if(isControlRoom(new Coordinate(j, i))) {
                controlRoom = new Coordinate(j, i);
                return true;
            }
        }
    }
    return false;
}
```

```
private static Vector<Coordinate> findRoute(Coordinate
    start, int R, int C) {

            // Given a position start, this methods
                returns the path from
            // the control room to that position, if such
                 a position exists

            // The output is given as a vector
                representing a sequence of
            // adjacent positions from start to the
                control room or null in
            // case such a sequence does not exist

            // Find the list of all the predecessors to
                all the destinations
            Map<Coordinate, Coordinate> predecC = BFS(
                start, R, C, true);

            // Vector containing the shortest path from
                the control room to
            // the coordinate "start"
            Vector<Coordinate> ptc = new Vector<
                Coordinate>();

            // Find the node before the control room in
                the path starting from
            // the "start" coordinate
            Coordinate predecessor = predecC.get(
                controlRoom);

            // Add to the path to the control room the
                control room itself...
            ptc.add(controlRoom);

            // ... and its predecessor
            ptc.add(predecessor);

            boolean found = false;
            while(!found) {
                predecessor = predecC.get(predecessor);
                ptc.add(predecessor);

                // When the starting point is reached,
```

```
                    then the goal has been reached
                // so found = true
                if(predecessor == start) {
                    found = true;
                }

                // If the predecessor is null, then the
                    node is not reachable
                if(predecessor == null) {
                    return null;
                }
            }

            return ptc;
}

public static void main(String args[]) {

    Scanner in = new Scanner(System.in);
    int R = in.nextInt(); // number of rows.
    int C = in.nextInt(); // number of columns.

    int A = in.nextInt(); // number of rounds between the
        time the alarm countdown is activated and the
       time the alarm goes off.

    // game loop
    while (true) {

        int KR = in.nextInt(); // row where Kirk is
            located.
        int KC = in.nextInt(); // column where Kirk is
            located.

        // Note: the matrix is in fact a vector of
            strings and it represents the
        // game field, divided in cells. A cell,
            therefore, is represented by a
        // character, while a string is a row of the
            matrix, that is an horizontal
        // 'slice' of the labyrinth

        matrix.clear();
        for (int i = 0; i < R; i++) {
            String ROW = in.next(); // C of the
```

```
        characters in '#.TC?' (i.e. one line of
        the ASCII maze).
    matrix.add(ROW);
}

// Set the current poition to the correct value
Coordinate currentPosition = new Coordinate(KC,
   KR);

boolean noPath = false;

// Set the current position to the correct value
if(what(currentPosition) == 'T') {
    startingPoint = currentPosition;
}

if(runAway) { // the countdown has started
    pathToC.remove(0); // the first element is
        the current position

    // move to the next position along the path
        from the control room
    // to the starting position T
    moveKirk(currentPosition, pathToC.get(0));
}
if(!runAway) { // the countdown has not started
   yet
     if(isThereC(R, C)) { // the control room
        position has been found
        if(firstFindC) {
            // find the shortest route from C to
                the starting point
            pathToC = findRoute(startingPoint, R,
                C);
        }

        // If the path from C to the starting
            point exists
        // and if it is short enough
        if(pathToC != null && pathToC.size() <= A
            +1) {

                if(firstFindC) { // it is the
                    first time that it is computed
                     // find the shortest path
```

```
                    from the current position
                pathFromHere = findRoute(
                    currentPosition, R, C);

                // the next time won't be the
                    first anymore
                firstFindC = false;
            }

            if(pathFromHere.get(pathFromHere.
                size()-1) == currentPosition)
                {
                pathFromHere.remove(
                    pathFromHere.size()-1);
            }

            // If Kirk is in the control room
                , he has to run away
            if(isControlRoom(pathFromHere.get
                (pathFromHere.size()-1))) {
                runAway = true;
            }

            // Move Kirk to the next position
            moveKirk(currentPosition,
                pathFromHere.get(pathFromHere.
                size()-1));

            // Remove the last position in
                the path
            pathFromHere.remove(pathFromHere.
                size()-1);

        }
        else { // otherwise ...
            noPath = true; // a good enough path
                from C to T does not exist
            firstFindC = true; // next time I
                have to look for such a path again
        }
    }

    // In case the contro room position is
        unknown or
    // there isn't a good path from it to the
```

```
                    starting position
                if(!isThereC(R, C) || noPath) {
                    // Determine the path to the best
                        reachable cell
                    Vector<Coordinate> myPath =
                        findBestChoice(currentPosition, R, C);

                    // Move Kirk along that path
                    moveKirk(currentPosition, myPath.get(
                        myPath.size()-2));

                    // Note: for each loop the best choice is
                         determined again, since
                    // for every movement of Kirk the
                        situation on the game field
                    // changes, since new cells are
                        discovered
                }

                printMatrix(); // shows the field status (for
                    debug purposes, to visualize
                // the explored cells easily)
            }
        }
    }
}
```

1.1.1 The `Coordinate` class

I have defined the `Coordinate` class in order to perform operations related to coordinates (for instance, distance calculation and comparisons) easily, without having to redefine those operations every time I needed them.

In particular, a coordinate is defined by two components, x and y. x refers to the the column, while y refers to the row in the labyrinth. It is possible to use two getters (`getX()` and `getY()` to access the components of a coordinate.

Moreover, in order to have a more compact and intuitive code afterwards, I have redefined `equals`, so that it is possible to see if two coordinates are in fact the same coordinate (thus, if the components are the same, the two coordinates are the same coordinate, even if they are located in different spaces in memory). In order to implement this it has been necessary to redefine the method `hashCode`.

It is also possible to calculate the (Manhattan) distance between two coor-

dinates. This is not the diagonal distance, since Kirk cannot move diagonally, but it is defined as:

$$d(i,j) =\mid \Delta x \mid + \mid \Delta y \mid=\mid i_x - j_x \mid + \mid i_y - j_y \mid$$

If the programmer tries to calculate the distance between a coordinate and an object that is not a coordinate, the returned value is -1. This value is never returned otherwise, since the distance is always greater or equal than 0 because of the absolute value introduced in the formula.

Moreover, there exist a method called `correctedDistance(Object x)`, that is not used. The reason why it had been introduced was to give priority to a specific direction in the movement of Kirk, but the overall performance of the game has not improved. I have decided to leave it in the code because it might be useful for a future development of the algorithm.

The `toString()` method can be used mainly for debug purposes, since, given a coordinate as an input, it provides a string representing the coordinate components as an output. In this way it is possible to use `System.err.println(coordinate)` and it will print the coordinate as `<x, y> = <coordinate.getX(), coordinate.getY()>`.

1.1.2 The `Player` class

This class is the most important class of the program, and it contains its business logic.

The `printMatrix()` method is used for debug purposes and it prints the field at a given moment. It has been necessary to iterate using strings because the field is defined as a sequence of strings. This implementation has been the reason why part of the code has become less readable and intuitive, while the use of a matrix would have simplified the implementation.

In order to read the character in a given position I have defined the `what` method. It takes a coordinate as an input and returns the character found in that position only in case the field has been defined in that position. In case it is not possible to access the given position (for instance because one of the coordinate components is negative) this method returns the wall character, defined as `'#'`. Although there isn't a real wall in that position, the definition of a wall is in practice equivalent to the definition of a new type of cell, which would have the same characteristics of the wall, for the purpose of this algorithm.

The `what` method is used by some functions that checks if a cell contains a specific character (for instance, if a cell contains `C`, than it is the control room cell.

Moreover, Kirk has the ability to see, even without visiting, the surrounding cells, therefore the the methods `seeX`, where X is right, left, up and down, have been defined. This is useful, for instance, in order to prevent Kirk from hitting a wall (or the control room before having visited enough cells[1]).

Given two coordinates, Kirk should be able to move from one coordinate to another. If the coordinates are not adjacent[2], it is necessary to look for a path to get from a coordinate to another, but if the coordinates are adjacent it is possible to move Kirk without any further calculation. The method `moveKirk` defines the four possible movements (up, down, left and right) according to the different possibilities. If `moveKirk` is applied to two coordinates with a distance that is not 1, no output is printed.

It is also very important to know where Kirk can walk. In particular, it is always possible for Kirk to walk safely on the hollow cells, represented by a dot. The method `canBeWalked` returns a list containing all the hollow coordinates.

Given a position, Kirk can move to one of the neighbours. In order to minimize the fuel consumption, this algorithm proposes an approach based on the movement of Kirk towards cells trying to reach the highest number of question marks[3], therefore it is handy to have a method able to identify which neighbours have an unknown content. Given a position, a neighbour is defined as a cell with distance equal to 1, therefore for each position there exist 4 neighbours. The method `getQuestionNeighbours` gives the list of neighbours as an output.

Similarly, `getExtendedQuestionNeighbours` gives the list of cells distant 1 or 2 from the given position. Those cells are the cells whose content is discovered when Kirk is in the given position.

In order to find the best path from a position A to a position B, it is necessary to use a search algorithm. The labyrinth can be modelled as a graph, even if it hasn't been modelled explicitly as a graph in this algorithm. However, the idea is that there exist a labyrinth fringe, that consists of all the coordinates in which there is an hollow space (represented by a dot or

[1]More details about how this has been implemented are presented later

[2]That is, if the distance, as defined before, is not 1

[3]More details later and in the comments in the code

by T, that is the starting point) that have at least an unknown neighbour (represented by a question mark). In such a graph the vertices are represented by the coordinates, while the arcs can be thought as the equivalent of the border shared between a cell and another adjacent cell. Given two adjacent cells, the distance between them is always 1, therefore all the arcs will have a weight of 1. This means that it is not necessary to use Dijkstra or Bellman-Ford: BFS is able to find the best path, if it exists. The main differences with respect to the original BFS algorithm are related to the identification of the adjacent nodes (the neighbours that can be visited) and to the memorization of the number of question marks that can be discovered during a given path[4]. Moreover, the distance to a given position is memorized.

A central part of the algorithm consists in the definition of the best choice (that is, the definition of the best node to reach, given a position, in order to minimize the fuel consumption before having reached the control room). Performing the BFS search gives a list of possible paths to reach all the reachable positions in the labyrinth. For each destination, the number of discovered question marks and the distance from the current position are known. The goal would be to maximize the number of discovered question marks minimizing the distance to discover them. Considering the approaches more in detail:

- A first approach I had considered was to determine the best path on the basis of the destination which had the highest number of question marks as neighbours. Even if this approach has worked for all the test cases, it is rather inefficient, since it often happens that Kirk goes back and forth, walking on the same cells several times. The results for all the test cases, from the first to the last, have been, respectively, 14, 42, 716, 266, 680, 450, 452 and 754 steps.

- Then I have considered that the best choice has to maximize the ratio between the number of question marks and the distance. The result have significantly improved: 14, 42, 178, 142, 92, 140, 294 and 288 steps.

[4]It is not the exact number of question marks, since for a given path it is the sum of the question marks discovered for each position, without considering that the same question mark can be discovered in two different cells on the same path. The reason why this is not considered is that the algorithm would have become more complex without a significant advantage.

- I have tried several slight adjustments on the basis of this formula, but the results have remained almost unchanged.
- Then I have tried to maximize the difference between the number of question marks and the distance and I have obtained 14, 42, 318, 172, 160, 222, 348 and 394 steps, that is almost always worse than the ratio case.
- Similarly, I have tried to maximize the difference between the number of question marks and four times the distance and I have obtained 14, 42, 198, 170, 96, 156, 252 and 262 steps, that is almost always worse than the ratio case, even if it is slightly better than the previous case[5].
- Similarly, I have tried to maximize the difference between the number of question marks and 12 times the distance and I have obtained 14, 42, 130, 144, 148, 74, 212 and 256 steps, that is almost always better than the ratio case [6].
- Similarly, I have tried to maximize the difference between the number of question marks and 12 times the distance and I have obtained 14, 42, 114, 158, 148, 118, 224 and 246 steps, that is almost the same as the previous case.
- Finally, I have tried to use $\frac{QM}{dist} - dist$, where QM is the number of question marks and dist is the distance to the destination. In this case I have obtained slight improvements for some cases (with respect to the ratio case, but not to the $QM - 12 * dist$ case): 14, 42, 138, 144, 148, 74, 258 and 258 steps.

Once the control room has been located, before reaching it the algorithm checks if there exist a path to the starting point that is short enough (this means that it has to be shorter than the alarm time). If it exists, Kirk finds the best path to the control room and then the shortest path from the control room to the initial position, otherwise he continues the exploration, checking every time if he can now reach the control room.

[5]My hypothesis is that this can be explained by considering that the total number of discovered question marks is closer to $4 * distance$ than to $distance$.

[6]My hypothesis is, again, that this can be explained by considering that the total number of discovered question marks is closer to $12 * distance$ than to $distance$. Since every time 24 new cells are analysed, this proposal is based on the assumption that half of them hasn't been visited yet

1.2 Analysis

1.2.1 Categorisation of the intelligent agent

The agent is not a **reflex** agent, since it does not act only on the basis of the current percept (the current cell content and the immediate neighbours content), but also on the basis of the past interactions, since the result of new discoveries is always memorized. The agent always has a map of the labyrinth based on the previous exploration.

The agent is not a **model-based reflex** agent because the considered agent is also able to reach a goal (finding the control room, finding a question mark, finding the shortest path from the control room to the initial position).

The agent is able to determine whether a status is a goal status or not (actually, the definition of "goal status" changes during the game, since in some phases the goal is to reach a question mark on the basis of some parameters, in other phases it consists in reaching the control room and finally it consists in reaching the initial position in the quickest way). Because of the introduction of an heuristic to find the most desirable question mark in the first phase, this agent is not simply a **goal-based** agent, but it behaves as a **utility-based** agent. However, during the second and third phase it behaves as a goal-based agent, since there is not an actual distinction between the desirability of different destinations (even because there exist only one control room and only one initial position).

The agent is not a **learning** agent, since it does not improve over time and there is no evolution of its behaviour, that is determined *a priori*.

The agent, therefore is a **utility-based agent**, even if it shows the characteristics of a goal-based agent when it cannot choose among different possibilities.

1.2.2 PEAS

It is possible to perform a PEAS analysis:

- **Performance measure**: number of steps to reach an intermediate (or final) destination, backpack energy used
- **Environment**: walls, control room, starting position, unknown cells, hollow cells
- **Actuators**: move right, move left, move up, move down

- **Sensors**: tricoder[7], input stream (to receive the instructions related to movement).

1.2.3 Categorisation of the environment

The environment is **partially observable**, since the agent knows only what is being detected by the sensors and what has been detected in the past. In some particular cases all the environment is observed, since the agent might visit the whole labyrinth, but in general the environment remains partially observable.

There is only one agent, therefore this is a **single-agent environment**. Differently from chess, for instance, there aren't multiple agents trying to pursue a goal. However, the alarm can be considered as a very elementary reflex agent, even if in the proposed implementation it has been considered only as a constraint to respect in the algorithm.

Competition and cooperation are defined only for multi-agent systems. If the alarm is considered as an agent, then the environment can be regarded as **competitive**.

The environment is **deterministic**, since the result of an action is known. It may be possible to change the definition of the problem by introducing, for instance, moving walls, and the environment would become stochastic.

The environment is **sequential**, since a movement affects the next movements and since every movement reduces the backpack energy and/or the time left to return to the initial position, once the control room has been reached.

The environment is **static**: while an agent is deliberating the labyrinth doesn't change and the current view of the labyrinth, from the agent perspective, changes only once a new cell has been visited.

The environment is **discrete**, since the environment can be in a finite number of different states (the cells are in a finite number, their content can assume only finite values and the agent can be in a finite number of positions).

The environment is **known**, since the agent knows what are the "laws" that regulate it and their consequences (for instance, hitting a wall means that the agent has failed, and this is a known information).

[7]Tool able to see the content of the cells in a 5-cell wide square centered on the current position

1.2.4 Problem formulation

In general, a **state** can be described as the current position, the map of the labyrinth as it is seen by the agent and the current phase of the game (the goal can be reaching the control room or reaching again the initial position and on the basis of the phase the agent will behave differently).

As for the **initial state**, it consists of the initial position of the agent and the map of the cells detected by the tricoder from that position. The agent can perform four **actions**: move right, move left, move up, move down.

As for the **transition model and successor function**, it can be described using the code provided in the `moveKirk` method: given a coordinate and one of the actions, the agent will move consequently and it will find itself in a new coordinate, whose components are described in that method. For instance, `Result(Coordinate <x, y>, moveLeft) = Coordinate <x-1, y>`.

As for the **goal test**, the problem can be split in two parts, with different goals. In the first part the goal is to reach the control room if and only if there is a path from the control room to the initial position that allows to reach it before the alarm. In the second part the goal is to reach the initial position. In the first case, therefore, it is necessary to check (a) if the control room is reachable and (b) if a good[8] path to the initial position exists, while in the second case it is only necessary to check if the agent is in the initial position after having reached the control room. This last control is hidden by the program interface, since once Kirk is arrived at the initial position the program automatically terminates with success, without the need of any further action.

The **step cost** is 1 to move from a coordinate to an adjacent coordinate, while the **path cost** is just the sum of all the step costs. Considering the heuristic described before, it is also possible to consider, for the first phase (before reaching the control room) that as the cost, since the best path is chosen on the basis of that. However, it is not a proper cost, since it does not represent exactly the "effort" needed to reach a destination. Alternatively, if the cost is just the number of steps, this coincides with the backpack energy calculation.

[8]That allows Kirk to reach the starting position before the countdown ends.

1.2.5 Description of the algorithm

A description of the algorithm has already been provided after the code section. The proposed algorithm is **informed**, since an heuristic is used to determine the best choice to reach a certain (intermediate) destination. The agent can estimate the "distance" from the current position to such a destination by using the heuristic, trying to minimize or maximize a value (depending on how the heuristic is being formulated).

I haven't used **local search strategies**, since the implemented search strategies point to the boundaries of the labyrinth in all the situations in which a search is performed. There aren't **adversarial search strategies** either.

The algorithm is a modified version of **BFS** and it is **complete**, since it always find a solution if it exists, it is **optimal** during the path from the control room to the initial position and when it has to reach an intermediate position (given a coordinate A and a coordinate B, it finds the best path from A to B). However, it does not necessary find the optimal solution, since the algorithm has to be based on the number of question marks, that are associated to an higher probability of finding walkable (hollow) cells, but this is an element of uncertainty, from which different possible choices can arise on the basis of probabilistic estimates. This doesn't contradict the determinism of the environment, since the labyrinth does not change: what changes is the perception the agent has of it. The stochastic element, moreover, is related to what the agent knows about the labyrinth, not to the labyrinth itself.

The algorithm might be improved by defining a more precise measure of the number of question marks discovered along the path, by considering that a question mark that has already been visited before in the path cannot be discovered again. Moreover, finding a better heuristic, after more attempts and possibly more precise probabilistic calculations, can speed up the control room reseach.

The proposed algorithm is able to find the shortest path in a reasonable time, that is always[9] less than 22% of the maximum time allowed. However, it is necessary to perform BFS searches quite frequently, while another approach might result in a faster computation (and, probably, longer paths).

Another change can involve the introduction of a penalty in case the agent visits again the already visited cells, for instance by introducing an higher

[9] In the test cases provided

step cost for such situations. This possibility has been introduced in the code, although now it is not active.