CS420 - Artificial Intelligence

# Lego the Maze Runner


| Tran Thi Anh Thu | Le Pham Ngoc Yen | Do Tri Nhan | Nguyen Minh Tri |
|---|---|---|---|
| 1751036 | 1751028 | 1751087 | 1751109 |
| ttathu@apcs.vn | lpnyen@apcs.vn | dtnhan@apcs.vn | nmtri@apcs.vn |

Advanced Program in Computer Science
Department of Information Technology
University of Science, VNU-HCM


May 9, 2023

## 1 Conceive:

In practice, there are many cases where we wish to find the least-cost paths such as traffic navigation, logistics, or circuit designing. In this project, our group want to illustrate such problem in a smaller scale which is least-cost maze solving with the Lego Mindstorms Robot as the solver (agent).

## 2 Design:

### 2.1 Product design

#### 2.1.1 Software design

We formally formulate the problem as a search problem where our agent is trying to search for the least-cost path from the entrance to the exit of a maze:

- The maze will not be guaranteed to have available path. It will first have a fixed sized $10 \times 10$. If the algorithm works fluently, we will let the robot determines the size of the maze whilst reading pixel from each row and column.
- The entrance and exit will not be fixed at the top left corner and the bottom right corner but rather they will be picked randomly within the maze.

Formal problem statement:

- Initial state: The robot at the entrance pixel (colored Yellow).
- Actions: $MoveLeft$, $MoveRight$, $MoveUp$, $MoveDown$, $LiftPen$ (to stop drawing), $DipPen$ (to start drawing).
- Transition model: At each position, the robot will have at most 4 adjacent pixels to choose. If one of the available pixels has the least cost, the current pixel will be updated to that pixel. The process will be carried out until the exit pixel is found or all available pixels have been through but no exit pixel is found.
- Goal test: The robot at the exit pixel (colored Red).
- Path cost: 1 between 2 adjacent pixels.

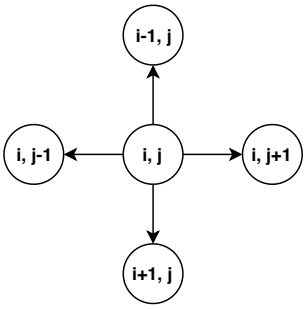


Figure 1: Transition Model

#### 2.1.2 Hardware design

There are 6 total features:

- Start button to run robot.
- The motors to control the robot goes vertically or horizontally, the robot will move in the $10 \times 10$ matrix with each $2 \times 2$ $cm^2$ for each pixel, the total area is 400 $cm^2$.
- Sensor to read information of maze on paper: maze design includes a total of 4 colors, white for paths, yellow for entrance gate, red for exit gate and black for obstacles (or walls). The robot is responsible for reading information from the matrix and processing to solve the maze.
- Notify signal lights if the problem is solved or not.
- The motor to control the pencil to draw the result path of the maze paper.
- Sounds to announce the completion of solving.

### 2.2 Accessories/motors/sensors

| Index | Components | Amount |
|---|---|---|
| 1 | Electric Mindstorms NXT | 1 |
| 2 | Electric 7.4V Battery Rechargeable NXT | 1 |
| 3 | Electric Mindstorms NXT Touch Sensor | 1 |
| 4 | Electric Mindstorms NXT Color Sensor | 1 |
| 5 | Electric Mindstorms NXT Motor | 1 |
| 6 | Electric Cable NXT | 3 |
| 7 | Some other basic components: Axles, Beams, Bricks, Gears... | 5 |

### 2.3 Solution design:

#### 2.3.1 Intelligence algorithm and pseudocode:

- To look for the least-cost way out of the maze, we use A* search algorithm with Manhattan distance heuristic to guarantee the admissibility of our solution.
- Pseudocode:

```
// Pixel class
Pixel {
    // backtracking by ID, implement using suitable data structure.
    int ID;
    int parentID;

    // keep coordinates (x, y)
    // to determine the position {above, below, left, right}
    int x;
    int y;

    int fValue;

    // in case we want to know how much the actual cost is
    int actualCost;
}

// this function returns movement {up, down, left, right} instruction to the robot
function getInstruction(int[][] map, Pixel start, Pixel goal) {

    Pixel[] expandedPixels;
    int[] instruction;

    boolean[][] visited;
    Pixel current = start;
    PriortyQueue pQueue (Pixel, f_Value);

    // start pixel get expanded
    expandedPixels.add(start);

    // keep searching until goal is found or cannot find the way out
    while (current != goal and instruction.isEmpty()) {
        // for each current pixel, expanding its 4 orthogonal neighbors
        // if they are not wall pixels and are not visited yet
        int above = expand (current, current.above_cordinates, pQueue, map, visited, goal);
        int below = expand (current, current.below_cordinates, pQueue, map, visited, goal);
        int left  = expand (current, current.left_cordinates,  pQueue, map, visited, goal);
        int right = expand (current, current.right_cordinates, pQueue, map, visited, goal);

        // if the number of expanding pixels is 0, this means we have stuck, no way out
        // therefore, end searching and return this instruction
        if (above + below + left + right == 0) {
            instruction.append (NO_WAY_OUT);
            break;
        }
        // if does not stuck, keep searching
        else {
            current = pQueue.dequeue()
            expandedPixels.add(current);
        }
    }


```

```
    // if reach goal state, backtrack a path from goal pixel to start pixel
    // then reverse the instruction list
    if (instruction.isEmpty()) {

        instruction.append (SOLVED);

        Pixel backTrack = current;

        while (backTrack.ID != backTrack.parentID) {

            // backtracking
            Pixel next = expandedPixels.find(backTrack.parentID);

            if (next == backTrack.above_pixel) instruction.append (DOWN);
            if (next == backTrack.below_pixel) instruction.append (UP);
            if (next == backTrack.left_pixel) instruction.append (RIGHT);
            if (next == backTrack.right_pixel) instruction.append (LEFT);

            backTrack = next;
        }
    }

    // reverse instruction list
    instruction = instruction.reverseList();

    return instruction;
}

function expand (current, coordinates, pQueue, map, visited, goal) {

    if (coordinates.valid
        and is_not_wall (coordinates, map)
        and is_not_visited (coordinates, visited)) {

            int manhattan_d = getManhattan (coordinates, goal)

            Pixel p = new Pixel;
            p.ID = current.ID + 1;
            p.parentID = current.ID;
            p.x = coordinates.x;
            p.y = coordinates.y;
            p.actualCost = current.actualCost + 1;
            p.fValue = manhattan_d + p.actualCost;

            // add the pixel to the priority queue
            pQueue.enqueue (p, p.fValue);

            // successfully visit a pixel
            return 1;
    }

    // failed visiting a pixel
    return 0;
}
```

#### 2.3.2 Risks and back-up plan:

- If the robot can not draw the path, the robot will display the answer on its screen and signal whether the maze is solvable or not.

### 2.4 Mechanism explanation

- Initially we have a matrix of 100 pixels, each pixel contains one of four color: red, yellow, black and white with encode: 3,2,1,0.
- For each row, the robot uses the horizontal-motor to move back and forth, using the color sensor to identify the color then store in the array.
- After finishing a row, the robot using vertical-motor moves color-sensor vertically, continue reading next line.
- After reading all the pixels, the robot will process the path with the algorithm, the algorithm returns the sequence of the directions of the pencil movement, if it is not solved, returns None.
- The robot returns to its original position, if the maze solvable, uses the third motor to put the pen down on the paper, and uses two motors to control motor following the returned sequence of direction, otherwise, signaling the problem cannot be solved.
- After finishing, the robot signaled its completion.

## 3 Implement and Operate:

### 3.1 Robot's capability:

Our product's ability:

- Movement: move forwards and backwards vertically using its wheels. The moving distance is preprogrammed regards to the maze size. The sensor and the pen are attached to a horizontal conveyor belt moving with maximum distance of $8cm$ (3 maze pixels).
- Perception: use color sensor to read color of each square of the maze. After reading the whole maze, a 2D-array containing encoded values is passed to the searching process. The searching process now calculates and returns the solvable signal and the solution path if exists. To instruct the robot, the program will give it a list of values to indicate whether it should turn left, turn right, go forward, or go backward.
- Output search result: play sound to signal if the maze is solvable or not. If it is, the robot uses a pen to draw the solution path.

### 3.2 Assemble hardships and solution

There are several differences between our design and the reality product as follows:

- Movement component:
  - We first reference some design about the Lego Mindstorm's vertical movement. However, it is too complicated and misleading to assemble so our group came up with a different design. We ended up not using gears to transmit rotation from the motor but rather directly benefit from it. This design turned out working nicely.

  - The horizontal movement of the sensor and the pen is too short comparing to our printed maze on paper. However, we decided not to resize the maze as it might affect our color sensor reading (the sensor needs approximately $2cm \times 2cm$ in area per pixel). Thus, we tried to lengthen the moving distance by replacing the old design with a conveyor belt controlled by a motor. This design allowed us to move horizontally up to 4 pixels. We choose 3 pixels for reading precision.

- Perception component: the color sensor was the main sensor of our robot. However, the hardware in reality were quite old and erroneous which made it impossible to collect information about the maze. We first tried to read in the color to test if the sensor was working, it did not respond. We testes 2 functions $get\_sample(), get\_lightness()$ but none of them returned any values when we scanned through different colors. We also asked teaching assistance to give us a new color sensor but it still did not work. After many attempts to fix this, we finally decided to handle the issues by bypassing the perception steps and manually setting up the maze in our code before loading into the robot.

- As we were having troubles in the perception component, our robot can only perform drawing the solution path. We then try to perfect the movement of the robot in drawing out the path. However, we encounter another issue is that our development software library was missing with unknown errors. We had tried to reload the library, read multiple threads on the Internet to fix this bug. Thus, despite having worked early on, our final robot was not ready to perform in the presentation.

# 4 Evaluations

## 4.1 Pros and cons in implementation methods:

**Pros:**

- We choose A* search algorithm with Manhattan distance heuristic, which guarantees our solution to be optimal since for each stroke, the robot can only draw from one square to it adjacent squares. This meets our problem statement that the robot should intelligently choose the least-cost way out if one exists.

- The algorithm, in fact, does not require the entrance and exit squares to be on the first row or last row of the maze but they can be anywhere within the $10 \times 10$ maze.

- We choose to use Python Development for NXT Lego Mindstorms which should allow us to give complex instruction (running A* algorithm) rather than doing intensive work of drag and drop.

**Cons:**

- Although we pay attention in designing the search algorithm, we have not focus carefully on the hardware design part, especially on the backup plan for the color sensor. This has such major impact on our final product since without the color sensor we could not conduct the next step.

- We relied on the sample instructions in building parts of the robot and spent quite a lot of time following it. The instruction was misleading since the pictures were not clear and the angles of the the Lego in the pictures made it hard to imagine the space in between each part of the robot. It was also time-consuming to figure it out, thus we improvise our construction.

### 4.2 Lessons learned:

- Given such time box and requirements, we should reference works of some previous years to determine our scope appropriately. Since we did not do that, in the end, we cannot achieve the goal that we had set in time.
- We should not rely too much on sample instructions, and then try to remodel it. Since that model is created to do different things than ours, it missed a lot of components that we need to accomplish our goal.
- We should have a more detailed backup plan for possible worse cases.
- We should record our trials (when our robot was still functioning properly) for backup presentation.
- We should prioritize using common and reliable development tools or frameworks to avoid library bugs and have more time to focus on the product logic section.