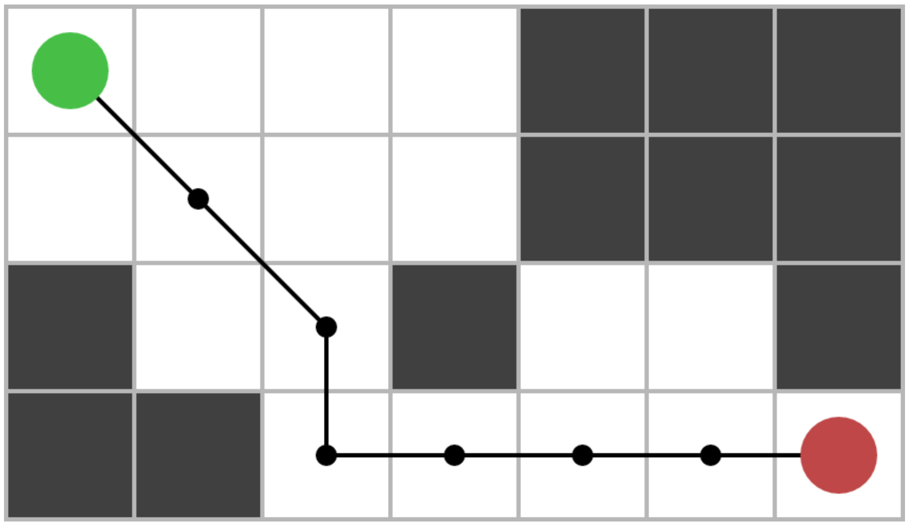
Mini-project
Autumn Semester 2023

Random Pits. Path finding

Made by:
Vladyslav Horbatenko, 76508


Growing with the Web: A* pathfinding algorithm

Roskilde University
November 2023

Contents

1 Problem Description

Main goal of this mini project is to implement and test Java program. It must be a single-player game, where player need to reach the goal, walking around 20x20 grid using Up, Down, Left, Right movements. On his way, in each new round (step) randomly pits will appear and block his way to the goal point.

Rules:

1. The game is played on a board (grid) consisting of 20×20 cells.
2. Initially, all cells are free, and the player is located in cell (0, 0).
3. In each turn, the player can move in one of the four cardinal directions (north, east, south, west).
4. In each turn, a random number (between 1 and 5) of pits is placed on random of the board that don't already have a pit.
5. The game is won if the player reaches the goal cell (19, 19).
6. The game is over and the player loses, if they stand on a pit.
7. The game is also lost if the player get surrounded by pits or is not able to reach the goal location anymore.

2 Solution

In this section I will explain my reasoning behind my solution, and also provide sources I used for different parts of the assignment.

2.1 Initial Variables and Constants

First, we create our main *class PitGame*. Throughout the game we going to use lots of variables, but some of them will be the most used. Program have this variables and constancts:

```java
    // constants
    private static final int BOARD_SIZE = 20;
    private static final char FREE_CELL = '.';
    private static final char PLAYER = 'X';
    private static final char PIT = 'o';
    private static final char GOAL = 'G';
    private static Random random = new Random();
    private static Scanner scanner = new Scanner(System.in);

    // game variables
    private char[][] board; // 2dm board array
    private boolean visited; // visited cells
    private int playerRow; // player row number
    private int playerCol; // player col number
    private int goalRow; // goal row number
    private int goalCol; // goal col number
    private boolean fellIntoPit = false; // checks if player fell into pit
```

Listing 1: Initial variables and constants

The variables and constants are self-explanatory. So lets move to board initiation.

2.2 Board Initiation

Out task it to create 20x20 cells grid (board). For that we going to use 2D array, as we also practiced in classes. For it, we'll update **board** variable with **BOARD_SIZE** constant that equals to 20.

```java
public PitGame() {
    board = new char[BOARD_SIZE][BOARD_SIZE]; // initiating board with 20x20 size
    // calling methods to fill all empty spaces with "o" and also to place player and goal on board
    initializeBoard();
    placePlayer();
    placeGoal();
}
```

Listing 2: Initiating the grid

And immediately after that we'll create a method to fill add cells with ".", as free cells:

```java
//method to fill each cell with free_cell symbol using nested loop
private void initializeBoard() {
    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            board[i][j] = FREE_CELL;
        }
    }
}
```

Listing 3: Initial cells

2.3 Placing Entities

This game have 3 entities that we'll have to implement:

- Player
- Goal
- Pits

One of the most important part of this game is player, which position will change over time. But at this point we will put them in their initial positions:

```java
    //method to put player on board
    private void placePlayer() {
        playerRow = 0; //players coordinates can be changed
        playerCol = 0;
        board[playerRow][playerCol] = PLAYER; //assign player's position
    }

    //method to put a goal on board
    private void placeGoal() {
        goalRow = BOARD_SIZE - 1; //assign goal's position to the last cell in our board, no
    matter of its size
        goalCol = BOARD_SIZE - 1;
        board[goalRow][goalCol] = GOAL; //assigning goal position
    }


    //puts random amount of pits in random positions, that are not player or goal, for each round
    private void placePits() {
        int numPits = random.nextInt(5) + 1; //random number from 1 to 5
        for (int i = 0; i < numPits; i++) {
            int row;
            int col;
            do { //using do while loop from classes to assign random positions to pits ONLY IF
    they are not player\goal\pit
                row = random.nextInt(BOARD_SIZE);
                col = random.nextInt(BOARD_SIZE);
            } while (board[row][col] != FREE_CELL);

            board[row][col] = PIT; //new pit
        }
    }
```

Listing 4: Initial entities position

Later we will use *placePits()* for each new round, to place 1-5 random pits in random positions around grid. This function creates random number between 1 and 5 and using for loop, creates pits in available positions until random number of pits is reached.

2.3.1 Visual

As we have a human player, we need to provide some kind of visualization of what is happening on grid.

```
    //method to display board game status (positions) in terminal
    //using nested loop to print each cell (we have used it in a class)
    private void printBoard() {
        for (int i = 0; i < BOARD_SIZE; i++) {
            for (int j = 0; j < BOARD_SIZE; j++) {
                System.out.print(board[i][j] + "  ");
            }
            System.out.println();
        }
    }
```

Listing 5: Displaying grid to user

For this, I used nested for loop, as we did several times on our classes. Like this, we can print all rows and columns, no matter what grid size we have.

2.4 Move Validation

We can only move somewhere if its legal, like in any videogame, we have boundaries and limits. In the below section I implemented rules for when game is won, lost, and where player can make a move.

```
    //checking if player is not trying to move outside of the board
    private boolean isValidMove(int newRow, int newCol) {
        return newRow >= 0 && newRow < BOARD_SIZE && newCol >= 0 && newCol < BOARD_SIZE;
    }

    //checking weather is the players position equal to the goal position = game is won
    private boolean isGameWon() {
        return playerRow == goalRow && playerCol == goalCol;
    }

    private boolean isGameLost() {
        // return board[playerRow][playerCol] == PIT;
        return fellIntoPit;
    }
```

```java

    //method to assist new position to player
    private void makeMove(int newRow, int newCol) {
        if (board[newRow][newCol] == PIT) { // check if the new position contains a pit
            fellIntoPit = true;
        } else {
            board[playerRow][playerCol] = FREE_CELL;
            playerRow = newRow;
            playerCol = newCol;
            board[playerRow][playerCol] = PLAYER;
        }
    }
```

Listing 6: Initial variables and constants

In the last method *makeMove()* I check if new position has a pit, if case if it is - I send this to *isGameLost()* method, that will inform main loop about it and end the game. Otherwise - player makes a move.

3 BFS Algorithm For Path Finding

One of the hardest part of this project for me was how to check if goal is reachable. From previous experience I went to investigate how path finding algorithms can help me with that. My main sources are:

- YouTube video that explain difference between differnt path finding alrotihms
- Geeks for geeks website blog
- CodePal version for AStart algorithm

With this algorithm we can check each round if goal is reachable from current player position. For this we create similar to grid 20x20 boolean 2D array that specifies which cells have been visited already and adding current players positions as initial = visited.

```java
    private boolean isGoalReachable() {
        boolean[][] visited = new boolean[BOARD_SIZE][BOARD_SIZE];
        visited[playerRow][playerCol] = true;
```

Listing 7: Boolean grid

After we create a directional vectors that will help us move around grid and gather data about possible paths to goal:

```java
    int[] dr = {-1, 1, 0, 0}; //rows
    int[] dc = {0, 0, -1, 1}; //columns
```

Listing 8: Directions

Now we implement Queue and update it with initial player position:

```java
    // implementing a queue
    java.util.Queue<int[]> queue = new java.util.LinkedList<>(); //creating a queue
    queue.offer(new int[]{playerRow, playerCol}); // enqueue the current player position
```

Listing 9: Queue

3.1 BFS While Loop

From now on, we will update this queue using a while loop. This while loop will expand and check most recently added neighbors around visited cells and check if the new position is within the board size, has not been visited, and is not a pit. In this case, if the algorithm reaches the goal, we can still play; otherwise, the player loses.

```java
// Running BFS until the queue is empty
while (!queue.isEmpty()) {
    // Dequeue the current position
    int[] current = queue.poll();
    int row = current[0];
    int col = current[1];

    // Checking all four directions
    for (int i = 0; i < 4; i++) {
        int newRow = row + dr[i];
        int newCol = col + dc[i];

        // Checking if the new position is within the board size, has not been visited, and not a pit
        if (newRow >= 0 && newRow < BOARD_SIZE && newCol >= 0 && newCol < BOARD_SIZE && !visited[newRow][newCol] && board[newRow][newCol] != PIT) {
            // Check if the goal is reached
            if (newRow == goalRow && newCol == goalCol) {
                return true; // Goal is reachable
            }

            // Marking the new position as visited and enqueuing
            visited[newRow][newCol] = true;
            queue.offer(new int[]{newRow, newCol});
        }
    }
}
return false; // If the goal is not reachable
```

Listing 10: While loop BFS

4 Main Loop

There is not much to say about the main loop in this game. It's quite simple and just calls previously described methods in the right order and uses some if conditions to notify the player if he wins/loses. The only thing that might be important is the moving by input system, where I used a *switch statement* to select one of the optional moves:

```java
switch (move) {
    case "W":
        newRow--; // Go up
        break;
    case "D":
        newCol++; // Go right
        break;
    case "S":
        newRow++; // Go down
        break;
    case "A":
        newCol--; // Go left
        break;
    default: // If the user inputs something other than W\A\S\D
        System.out.println("Wrong key pressed. You can enter W, A, S, or D-");
        continue; // Continue the loop
```

Listing 11: Initial variables and constants

5 Conclusion

Overall, the game functions well and completes all the required tasks. However, it can be optimized, for example, by using the A* algorithm instead of BFS. The structure of the code could also be better written and split into separate Java files and multiple Class use. And program still have some bugs: new pits appear even if player make a wrong move or tries to go out of the grid. As a possible future development, the game could transition from terminal visualization to a PC application.