Kristianstad University Sweden

Kristianstad University
SE-291 88 Kristianstad
Sweden
+46 44 250 30 00
www.hkr.se

Analysis Report

The one with the labyrinth

Anton Carlsson
June 15, 2022

Contents

1 Introduction

This report aim to solve a simple problem utilizing Polya's Problem Solving Technique. This method consists of first gathering an understanding of the problem, then a plan of how to solve it is devised. These steps constitutes the main focus of this report. Additionally with a plan at hand the last steps consists of execute according to the plan and reflecting whether the strategy sufficed to achieve a suitable solution to the problem.

2 Step 1

2.1 Rephrasing the problem

The task consists of developing a labyrinth simulation game where the player starts at a specific square. Initially the player is greeted with a message explaining that they can type "w" to move one step upwards, "a" one step to the left, "s" one step down and "d" one step to the right. In the labyrinth it's however completely dark meaning the player can at no stage of the game know the current state or where he or she is located.

The user has 20 moves to find the exit which will terminate the game and let the user know that the game is finished. However if the user does not make it in time a message stating this will instead be shown. If the player tries to walk into a wall this will cost one move, but the player will remain on the same square and be alerted that there is a wall. Similarly the player can find power ups (adding 15 lives) and traps which sets the player back to the starting position and in both cases a suitable message explaining the situation will be shown to the player.

2.2 Understanding the words

There was no vague or unclear phrasing and no research was needed to solve this task. There was however some freedom in interpretation of certain situation, for example what happens if the player tries to double back to an already visited trap or power up square. In this case a decision was made to set the power up square to a normal path square (one time use), while the trap remains and can be triggered multiple times.

3 Devising a plan

3.1 Approach of choice

The idea is to break the task up into smaller subtasks and try to solve these in individual functions to then assemble everything and present a combined solution. Sometimes it can be nice to start writing a main function containing the structure of the program and then fill in any vague functions with what they're supposed to accomplish afterwards. For this task a good way to start can be to construct a representation of the labyrinth and this function can for example be called *create_board*(). Before the player gets to start, the instructions of how to play needs to be provided which we can accomplish with a function *print_startup_msg*(). Then essentially the program is going to run until either the lives run out or the exit has been found. The pseudo code representation could look something like:

```
def main():
    board = create_board()
    print_startup_msg()
    run_program(board)
    if get_board_val(board, curr_pos) not EXIT then
        print("Game over! You did not reach the exit in time.")
```

With this structure it is assumed that the *run_program*() function is run while the player has not yet found the exit and still has lives left. Afterwards the state of the game is examined using the *get_board_val*() with the board state *board* and current position *curr_pos* to determine whether the player has won or lost.

The board is initialized and represented as a $10x10$ matrix where the value at position i, j determines whether the square is starting position, exit, path, wall, power up or a trap.

```
def create_board() -> np.ndarray:
    # Initiating board with only walls
    board := np.zeros((10,10))
    pths := [[1,0], [2,0], [3,0], [4,0], [5,0], [6,0], [7,0],
   [8,0], [9,0], [1,1], [9,1], [1,2], [2,2], [3,2], [4,2], [5,2],
   [6,2], [9,2], [3,3], [7,3], [3,4], [5,4], [6,4], [7,4], [1,5],
   [2,5], [3,5], [3,6], [4,6], [5,6], [6,6], [7,6], [8,6], [8,7],
   [2,8], [3,8], [4,8], [8,8], [4,9], [5,9], [6,9], [7,9], [8,9]]
    start := [7,2]
    exit := [1,8]
    traps := [[9,3], [1,6], [2,9]]
    power_ups := [[1,4], [6,8]]
    # Setting up paths
    for path in pths
        board[path[0]][path[1]] := PATH
    # Setting up traps
    for trap in traps
        board[trap[0]][trap[1]] := TRAP
    # Setting up power ups
    for power in power_ups:
        board[power[0]][power[1]] := POWER
    # Select start position
    board[start[0]][start[1]] := START
    # Select exit position
    board[exit[0]][exit[1]] := EXIT

    return board
```

For actually running the program an easy implementation could be to use a while loop that remains active while there still are lives left and the player hasn't reached the exit. At the beginning of the while loop it could be a good idea to let the user choose a move and if the player chooses a legal move (w, a, s or d) then one move can be subtracted from the 20 lives. Depending on which direction was chosen, a new position can be found which then needs to be investigated further.

```
def run_program():
    input:
        board   (Matrix representation of the state of the game)

    # Continue while we have lives left and not yet found the exit
    while life > 0 and get_board_val(board, curr_pos) not EXIT
   then

        choice := input("Enter direction: ")

        # Controls for the game w=up, a=left, s=down, d=right
        if choice = "w" then
            life -= 1
            new_pos := [curr_pos[0] - 1, curr_pos[1]]
            move(board, new_pos)
        else if choice = "a" then
            life -= 1
            new_pos := [curr_pos[0], curr_pos[1] - 1]
            move(board, new_pos)
        else if choice = "s" then
            life -= 1
            new_pos := [curr_pos[0] + 1, curr_pos[1]]
            move(board, new_pos)
        else if choice = "d" then
            life -= 1
            new_pos := [curr_pos[0], curr_pos[1] + 1]
            move(board, new_pos)
        else
            pass         (No need to handle exceptions, the user
                          knows what to do.)
```

If the player tries to move outside the labyrinth (not exit), then this should be treated as a wall and the current position should not be changed. In any other case the current position needs to be updated to the new position. Then for every other possible square the board needs to be updated according to the task description. If the player reaches a power up this square needs to be set to a path square so the player can't obtain multiple power ups from the same square. If the player reaches the exit, then a victory message should be printed and the functioned can be returned since the game is over.

```
def move():
    input:
        board   (Matrix representation of the state of the game)
        pos     (New potential position to be considered)

    # Makes sure we cannot move outside board
    if -1 in pos or 10 in pos then
        print("Ouch! I can not walk through walls...")
        return null

    val := get_board_val(board, pos)    (Finds the value of the
                                         square to the  pos in the
                                         matrix)

    # Updates the game stage based on what square we are moving to
    if val is WALL then
        print("Ouch! I can not walk through walls...")
    else if val is PATH or val is START then
        curr_pos := pos
    else if val is TRAP then
        print("Oh no, a trap!")
        curr_pos := ORIGIN
    else if val is POWER then
        print("A chocolate bar, I feel stronger.")
        curr_pos := pos
        board[pos[0]][pos[1]] := PATH
        life += 15
    else
        print("You survived! Well done adventurer!")
        curr_pos := pos
        return null
```

There's also a few global variables, but they are self explanatory (hint: life, WALL, PATH, START, TRAP, POWER are all integers *curr_pos* and ORIGIN are both 1x2 vectors containing coordinates in the board matrix)