COMP332 - Assignment 1

Frogs and Toads

Sophie Kaelin (45198543)

Introduction

The aim of this assignment is to visually demonstrate solutions to the “Frogs and Toads” puzzle given various cell length inputs. The sub goals in order to achieve this include: generating an algorithm to solve the puzzle with a given input of frogs and toads, creating a visual display of this solution once it is found. In this report I will review and justify the code design and how each element of the solution is implemented, as well as the tests created to ensure the program works as it is designed.

Design and Implementation

As there are two distinct goals of the assignment, I will address how each of them was designed and implemented individually.

Algorithm to solve puzzle

In order to solve the puzzle I had to attempt every possible combination of slides and jumps until I found one which completed the puzzle (i.e. it was in it’s terminal state). This was achieved through a depth first search of a tree structure which contained every possible move that could be made. In order to achieve this, I implemented various helper methods to test if potential moves were legal. These will be discussed below.

> slideFromRight(), slideFromLeft(), jumpFromRight() & jumpFromLeft()
> These helper methods are responsible for assessing whether a given move is legal from a certain PuzzleState. These methods have an optional return type, meaning they can either return “Some” or “None” value. A “None” value would be returned if the move was illegal, or if the index of the moving cell or the empty cell is out of the range. Otherwise, a new PuzzleState would be created and returned representing the successful outcome of an attempted move. This method is called by the solve function when determining the correct path from the initial state to the terminal state.
>
> Solve()

This method operated recursively to return the path of PuzzleStates from the initial to the terminal. The method would first check if the parameter PuzzleState was the terminal state, and in that case it would return the Sequence of PuzzleStates. It will then perform a depth-first-search for the terminal PuzzleState, and return an amended Seq() list from the recursive calls when this is achieved. This method will attempt each possible move (listed above) in the depth-first-search, and only when it has attempted each move with none of them proving successful will it return an empty sequence up the trail of recursive calls to signify that this sequence would not be successful. By assigning the recursive statement to a variable, I was able to check whether a particular path was successful or not, and continue recursing if a potential path was found. An example of this can be seen for the jumpFromLeft move:

```
if(start.jumpFromLeft() != None){
  val jumpLeft = solve(start.jumpFromLeft().get)
  if (jumpLeft != Seq()) {
    return solve(start.jumpFromLeft().get) ++ Seq(start)
  }
}
```

Algorithm to create visual animation of the solved puzzle

This element of the solution was only achievable once the recursive solution to solve the puzzle was achieved. Once a list/sequence of moves was found, the animation functions were able to recurse through the given sequence to create an animation of the found solution. Helper methods were also created to aid this functionality, which will be discussed below.

createState()

The purpose of this helper function was to create the individually images for any given PuzzleState. This function operates recursively, as it iterates over each of the PuzzleState cells and their contents, and is called by the animate function, when various PuzzleState's images must be created. I achieved this recursion using a match expression that altered the colour of a cell depending on whether it currently contained a frog, toad or was empty. The parameter "idx" kept track of which cell was currently being drawn and ensured there were no out of bounds exception errors.

animate()

The animate function takes the successful sequence of PuzzleStates and generates a visual representation of the sequence of moves it takes from the initial to terminal state.

Testing

Because the program allows for an inputted value to dynamically change the solution to the problem, various tests were created to ensure that the program worked for every potential edge

case. In this section of my report, I will justify the tests I included as being a proper representative set of test cases for the given puzzle.

Testing Potential Moves

In testing whether potential moves were valid or not, there were a lot of edge cases to consider. Some of these boundary cases included:

- ➔ Returns Some(_) when a move is legal
- ➔ Returns None when a move is illegal
- ➔ Returns None when a move is made out of bounds of the board.

These tests check whether the empty cell is in the position it should be after the operation. The downside to this is that there is no check if the moved cell is in the correct position. If time permitted, I would have included tests such as this.

Considering these edge cases, a suitable number of tests were created to ensure that the functions worked successfully for a PuzzleState at any stage of the sequence. Only checking PuzzleStates at the beginning of their sequence would not capture a large enough variety of potential inputs. An example test for the edge case that checks if the method will return “None” when a toad will jump left into a location outside of the PuzzleState boundaries goes as follows:

```
"A puzzle state with 1 frogs and 3 toads using slideFromRight, jumpFromLeft, jumpFromRight:" should
  "return None because the move is illegal (Out of Bounds Exception)" in {
  assert(PuzzleState(1, 3).slideFromRight.get.jumpFromLeft.get.jumpFromRight == None)
}
```