COMP332 – Assignment 1: Frogs and Toads

Jake Garth – 44887841

Report Structure:

The report will cover the three following topics:

- **Introduction**: This covers the aim of the project and the context surrounding the project.
- **Design and Implementation**: This covers the overarching strategy implemented in the code, this is in addition to explaining the specifics of the methods that were created.
- **Testing**: What tests were used and the reasoning behind their completeness .

Introduction

The Problem

The goal of this project was to be able to solve the "Frogs and Toads" problem, by providing a solution to the problem if possible.

The "Frogs and Toads" problem is as follows:
There is a board of 'n' amount of frogs, 'm' amount of toads and one empty spot on a 1 dimensional board.

Initially, the frogs are all on the left-side of the board and all the toads are on the right-hand side of the board with the empty cell in between the frogs and toads.

e.g. A board of 2 frogs and 3 toads, where green represents frogs and blue represents toads.

The objective of the game is to be able to rearrange the frogs and toads such that all the frogs end up on the right-hand side of the board and all the toads end up on the left-hand side of the board. This is called the terminal state.

e.g.

However, the frogs and toads are bound by certain movements. The four movement options are as follows:

- Frog Slides Right into an Empty spot
- Toad Slides Left into an Empty spot
- Frog Jumps Right over one Toad into an Empty spot
- Toad Jumps Left over one Toad into an Empty spot

If the board gets into a state such that none of these movements are possible and the board is not in the terminal state, then the game is lost.

In the Context of Scala and Functional Programming

The restrictions on the creator of the program is that they must use the programming language Scala and functional programming techniques to find a solution and then display the solution of steps as an image.

In particular, recursion and pattern matching are necessities for building this project, while loops are to not be used.

Design and Implementation

Overarching Strategy

1. Making movements: The first thing made was the “toadLeft”, “frogRight”, “toadJumpLeft”, and “frogJumpRight” methods. Essentially, these methods rearrange the board of the PuzzleState and also update the location of the Empty cell.
2. Solving for a solution: The “solve” method is given an initial PuzzleState and then will check every possible combination of legal movements until it arrives at a solution. The sequence of PuzzleStates that end up at a terminal state is then returned. If no solution is valid, null is returned.
3. Visualizing Cells: Different squares were made to represent a Toad, Frog and Empty Cell.
4. Visualizing the Solution: The animate method is the result of two different recursive methods. The “seqImageMaker” and “builder” method.
   The “builder” method will take in a PuzzleState and then recursively drew a Cell at every position of the PuzzleState board. This is used to draw an individual PuzzleState returns an image. The “seqImageMaker” will then use the “builder” method to make a sequence of Images, representing all the different PuzzleStates that make up the solution discovered by the “solve” method.

Explanation of Methods

Movements

The movements were written by first checking if the movement is possible. This includes checking if an out of bounds exception will be caused by the swap and then also if the Cells that are being swapped are in the right position. These methods returned an Option[PuzzleState], because, in the event that the movement is invalid, None is returned, just in case that movement was incorrectly called.

For example, in the toadLeft movement, it first checks that the Empty Cell isn’t at the very right of the board, and then also if the Cell to the right of the Empty Cell is a Toad. If all these are true, then, a new PuzzleState is returned that has a board which maintains original board, however, the position of the Empty Cell is switched with the Toad to the right. The reasoning behind the other three different movement types is the same.

```
def toadLeft(): Option[PuzzleState] = {
  val thisBoard: Vector[PuzzleState.Cell] = getBoard()
  val emptyIndex: Int = thisBoard.indexOf(Empty)
  if( emptyIndex != size-1){                  //checking if the swap would cause an outofbounds exception
  if (thisBoard(emptyIndex + 1) == Toad) {    //checking if there is a Toad directly to the right of Empty
  return  Some(new PuzzleState(               //if so, return a new PuzzleState by swapping Empty and Toad
     thisBoard.take(loc).++(Vector(Toad)).++(Vector(Empty)).++(thisBoard.takeRight(size-loc-2)), loc+1))
} else {
    return None                              //else, Return None
  }
  } else {
    return None                               //else, Return None
  }
}
```

Solve for a Solution

The “solve” method is the method which given an initial PuzzleState, will return a sequence of PuzzleStates that lead to a terminal PuzzleState, recursively.

The base case is that if the most recent PuzzleState is a terminal state, then, return the sequence for that PuzzleState.

If the PuzzleState is not at a terminal state, then it checks if a move is possible. If this move is not possible, return None. If it is possible, then make this move and continue until either a terminal state can be achieved or null is returned. If a terminal state can be reached, build a sequence of PuzzleStates comprising of all the PuzzleStates necessary to reach the terminal state.

```
def solve(start: PuzzleState): Seq[PuzzleState] ={
  if(start.isTerminalState()){
    return Seq(start)
  }
  start.frogJumpRight() match {
    case Some(newState) => solve(newState) match {
      case null =>
      case sequen => return sequen++Seq(start)
    }
    case None =>
  }
   start.toadJumpLeft() match {
    case Some(newState) => solve(newState) match {
      case null =>
      case sequen => return sequen++Seq(start)}
    case None =>
   }
   start.toadLeft() match {
    case Some(newState) => solve(newState) match {
      case null =>
      case sequen => return sequen++Seq(start)
    }
    case None =>
   }
  start.frogRight match {
    case Some(newState) => solve(newState) match {
        case null =>
        case sequen => return sequen++Seq(start)
    }
    case None =>
  }
null
 }
```

If no solution can be found from the given PuzzleState, null is returned. The null is then caught later, and is then displayed as an empty screen to indicate no solution. This has been tested with a board of the form ‘Empty | Frog |Frog | Toad’.

Visualizing the Cells

In the PuzzleState object, three different squares were defined, which each represent the different Cells. These are then used in the “builder” method to display the positions of Frog, Toad and Empty Cells on the screen.

```
val squareboiGreen =
  Image.square( side = 100)
    .fillColor(Color.green)
    .strokeColor(Color.black)
    .strokeWidth( width = 4)

val squareboiBlue =
  Image.square( side = 100)
    .fillColor(Color.blue)
    .strokeColor(Color.black)
    .strokeWidth( width = 4)

val squareboiWhite =
  Image.square( side = 100)
    .fillColor(Color.white)
    .strokeColor(Color.black)
    .strokeWidth( width = 4)
```

Visualizing the Solution

Two helper methods were used to make up the contents of the “animate” method.

Firstly, the “builder” method was made to visualize one PuzzleState.
The “builder” method uses recursion to make a visual representation of a PuzzleState using the defined squares. This method will take in a PuzzleState and then interpret it visually. It does this by looking at the right-most (at position n) Cell and then drawing that Cell adjacent to “builder(n-1, PuzzleState)”. Then, this recursion would stop when it reached the left-most Cell (when n = 0).

```
def builder(count: Int, start: PuzzleState): Image =
  count match {
    case n if(n==0 && start.getBoardState( pos = 0) == Toad) => squareboiBlue
    case n if(n==0 && start.getBoardState( pos = 0) == Frog) => squareboiGreen
    case n if(n==0 && start.getBoardState( pos = 0) == Empty) => squareboiWhite
    case n if start.getBoardState(n) == Toad =>
      val here = squareboiBlue
      builder(n-1,start).beside(here)
    case n if start.getBoardState(n) == Frog =>
      val here = squareboiGreen
      builder(n-1,start).beside(here)
    case n if start.getBoardState(n) == Empty =>
      val here = squareboiWhite
      builder(n-1,start).beside(here)
  }
```

Now that an individual PuzzleState can be drawn, a sequence of these must be made. The

“seqImageMaker” method will take in the sequence of PuzzleStates made by the “solve” method. Then, it will use this sequence to build a new sequence of Images, via the “builder” method.
It does this by using pattern matching and recursion.
The base case is when the puzzle sequence is empty, which will return the image sequence.
Else, return “seqImageMaker”, while parsing through a puzzle sequence without the left-most PuzzleState and also an Image Sequence that now includes the image of the PuzzleState that was just removed.

```
//the seqImageMaker returns the sequence of images for the animate method to display
def seqImageMaker(PuzzleSeq: Seq[PuzzleState], ImageSeq: Seq[Image]):
Seq[Image] = PuzzleSeq match{
case PuzzleSeq if (PuzzleSeq.isEmpty) => ImageSeq
case _ => seqImageMaker(PuzzleSeq.take(PuzzleSeq.size-1),
  ImageSeq:+builder(PuzzleSeq.takeRight(1).head.size-1,PuzzleSeq.takeRight(1).head))
}
```

Then, the “animate” method simply just returns seqImageMaker, parsing through the sequence of PuzzleStates and also an empty sequence to put the images into. However, if solve returns a null (i.e. no solution is found), then a blank screen is drawn.

```
def animate(start: PuzzleState): Seq[Image] = {

 if(solve(start) == null){ //if the solve method returns null because it doesn't find a solution, draw an empty screen
   Seq()
 } else {
   seqImageMaker(solve(start), Seq())
 }//the animate method is essentially the "seqImageMaker" method
}
```

This concludes the key parts of the code.

Testing

As the boards can only have a maximum of 10 frogs and 10 toads at a time, and a minimum of 1 frog or 1 toad, there are only 100 different starting positions. So, for each test written, a nested for loop from 1-10 frogs and 1-10 toads will ensure completeness.

Checking Size

These check that in the initial state and in the terminal state, the amount of PuzzleState Cells is the same size as the amount of frogs plus the amount toads plus one empty.

```
  for (x <- 1 to 10) {
    for (y <- 1 to 10) {
      "A puzzle state with " + x + "frogs and " + y +" toads:" should
        "have " + x+ " + " + y +" + " +1 + " cells" in {
        assert(PuzzleState(x, y).size == x+y+1)
      }
    }
  }

  for (x <- 1 to 10) {
    for (y <- 1 to 10) {
      "A puzzle state with " + x + "frogs and " + y +" toads:" should
        "have " + x+ " + " + y +" + " +1 + " cells in its terminal state" in {
        assert(solve(PuzzleState(x, y)).takeRight(1).head.size == x+y+1)
      }
    }
  }
```

Checking Empty Position

These check that in the initial and terminal state, that the PuzzleState has the empty location in the correct position. The empty location at initial state should simply by the amount frogs on the board, at termination it should be the amount of toads on the board.

```
//checks position of empty cell at initial position
for (x <- 1 to 10) {
  for (y <- 1 to 10) {
    it should "have its empty cell at position" + x +" for frogs " + x + "toads" + y in {
      assertResult(x) {
        PuzzleState(x, y).emptyLoc
      }
    }
  }
}

//checks position of empty cell at terminal position
for (x <- 1 to 10) {
  for (y <- 1 to 10) {
    it should "have its empty cell at position " + (PuzzleState(x,y).size-x-1) +" at terminal state for frogs " + x + " and toads" + y in {
      assertResult(y) {
        solve(PuzzleState(x, y)).head.emptyLoc
      }
    }
  }
}
```

Checking that the Initial Position isn't the Terminal Position

This simply checks that the initial position was constructed correctly by checking it is not initially in the terminal state.

```
//checks that initial position != terminal position
for (x <- 1 to 10) {
  for (y <- 1 to 10) {
    it should "not be constructed in the terminal puzzle state for frogs " + x + " and toads " + y in {
      assert(!PuzzleState(x, y).isTerminalState())
    }
  }
}
```

Checking that the Solve Method returns the Terminal State

This method checks the solve() method by checking that the left-most PuzzleState in the sequence returned by the solve() method is the Terminal State.

```
//checks that the solve function returns a sequence such that the last PuzzleState in the sequence is in the terminal position. i.e. Did solve work?
for (x <- 1 to 10){
  for( y <- 1 to 10) {
    it should "finish in terminal state for frogs: " +x+" and toads: "+y in {
      assert(solve(PuzzleState(x, y)).head.isTerminalState())
  }
  }
}
```

In addition to this, to check that there are a reasonable amount of PuzzleStates in the sequence returned by the solve() method, the code below was used to ensure that the solve() method actually returned a sequence, not just the terminal PuzzleState.

```
//checks if there are multiples PuzzleStates in the solution
for (x <- 1 to 10){
  for( y <- 1 to 10) {
    it should "return a sequence of more than 3 PuzzleStates: for " +x+" frogs: and "+y+" toads" in {
      assert(solve(PuzzleState(x, y)).size > 3)
    }
  }
}
```

With this, the tests comprehensively check every combination possible.