COMP332 Programming Languages – Assignment 1 Report

Name: George Hayward
Student Number: 43692486

Introduction:

The aim of this assignment was to develop a function in Scala that would find a solution to the specified frogs and toads' game. The solution must implement a recursive depth-first search algorithm that would return a solution represented by a sequence of puzzle states spanning from initial state to terminal state. Furthermore, all values used must be immutable in order to fully operate within functional programming.

The following report will detail the implementation of:

1. *The frogs and toads game moves:* slideFromLeft(), slideFromRight(), jumpFromLeft(), jumpFromRight()
2. *The solution to frogs and toad game:* solve()
3. *The visual representation of the solution:* animate(), generateBoard()
4. *The testing of all implementations*

Design and Implementation:

Game Moves:
The games moves are specified as so:

- slideFromLeft(): If there is a frog to the left of the empty space then the frog and empty space swap place to create a new PuzzleState
- slideFromRight(): If there is a toad to the right of an empty space then the toad and empty space swap place to create a new PuzzleState
- jumpFromLeft(): If there is sequence of '|F|T| |' on the board then the frog can swap place with the empty space '| |T|F' to create a new PuzzleState.
- jumpFromRight(): If there is a sequence of '| |F|T|' on the board then the toad can swap place with the empty space '|T|F| |' to create a new PuzzleState.

All of the movement functions contain conditionals to check whether or not the move is legal. These conditionals include the rules specified above and also conditionals that check for problematic puzzle states where the empty space is at the border of the board. The new puzzle states are created by using the '.slice' function to extract the 2-3 cells which will be altered and replacing them with the cells from the moves effect.

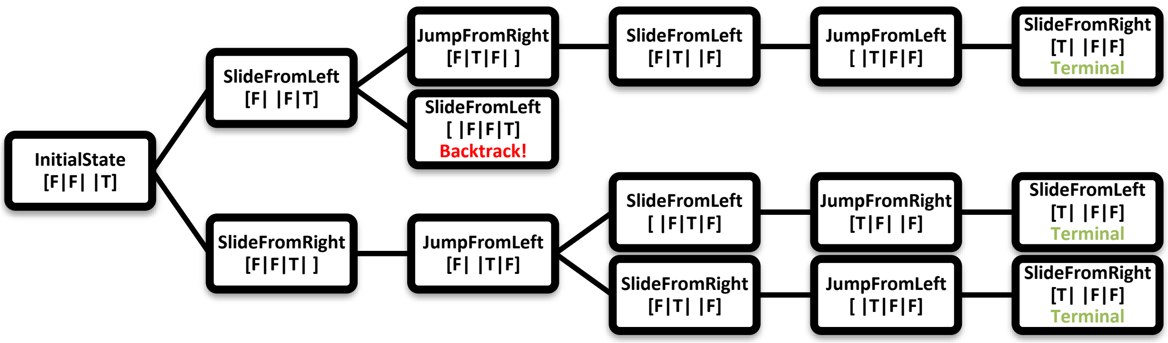
E.g. slideFromLeft() will extract **|F| |** from |F**|F| |**T|T| and replace with **| |F|** to make a new puzzle state board |F**| |F|**T|T|
In order to determine whether or not a move is legal, the return type of these functions is Option[PuzzleState]. If there is a legal move then the function returns Some(newPuzzleState) and otherwise returns None.

Solution:

The solve() function implements a recursive depth first search. In this DFS, each node represents a PuzzleState object with differing game boards. The edges of the DFS represent moves made between the nodes. E.g. A PuzzleState will connect to PuzzleState.slideFromLeft() if the move is legal. The **solve()** algorithm works as follows:

1. Create a new sequence newStateSolution containing the input 'start' PuzzleState object.
2. Check if the input PuzzleState is terminal, if it is return newStateSolution. This step will end the recursion of later steps.
3. Input PuzzleState isn't terminal, which means the terminal state must be more moves away. Try every legal move for this puzzle state (slideFromLeft, slideFromRight, jumpFromLeft, jumpFromRight)
4. A move is legal if the case-match finds Some state from a game move. If the move is legal create a new sequence futureStates = newStateSolution ++ solve(newState).
    a. solve(newState) is called recursively to do step 1-4 for the next state of this legal move. These steps will continue until a terminal state is found or every possible node is visited.
5. If futureStates last entry is a terminal state, it means that this recursive call of solve(newState) has found a terminal state, because the conditional from 2. Returned the terminal state instead of continuing through that solve() call. Therefore, return futureStates as it will contain a path of puzzleStates that will lead to the terminal state.
6. If all moves return None, then return an empty Seq() which will cause previous solve()'s to backtrack as no such terminal state was found. From then that backtracked solve() will continue to try moves.

Below is a tree demonstrating how a very simple solution to the game of frogs and toad will function:



The returning object for Solve() will be a sequence of PuzzleStates which will now need to be visually represented.

Visual Representation:

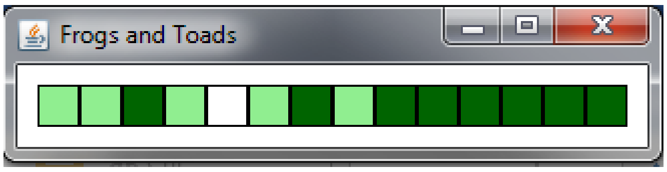
The **animate()** is a recursive function that works by generating a sequence of Images (done by helper method generateBoard()) based on the solved solution of the input PuzzleState. It actually works very similarly to solve(). The animate() algorithm works as follows:

1. Generate a game solution – sequence of PuzzleStates that leads to terminal state from solve(inputState)
2. Make value imageSeq which will hold this call of animate()'s PuzzleState image. (it is rotated due to the implementation of generateBoard())
3. Check if this call of animate(start) is terminal. If it is return imageSeq which will break the upcoming recursion and yield a Seq of images representing the solution.
4. If not terminal. Return a value of newImageSeq = imageSeq ++ animate(solution(1))
    a. Solution(1) is animated because it is the next PuzzleState in the sequence.

The **generateBoard()** algorithm is private, so it references this instance of the PuzzleState. It works as follows:

1. If the input Int is -1, that means that every other box has been added recursively. So just return an empty image to stop the recursion.
2. Otherwise, Check what type of cell is located at the Int (iter). Make a new rectangle with the right color (lightgreen = frog etc) and put it beside a recursive call of generateBoard() which will generate the next cell image.

The animate() function will be called from the projects Main() and will generate an animated representation of a solution to the input instantiation of a frogs and toad game. See below:



Other functions

These functions exist for convenience in testing and development but are not essential to the core implantation of this assignment:

- Overide of toString() – This will convert the board to a readable string of F's and T's.
- simpleChar() – Converts strings to simple chars. E.g. Frog > F
- apply(String) – Creates PuzzleState object based on a string input. E.g. |F|F| |T|
- recursiveAddVector – A helper method for above apply to join all new vectors together.

Testing:

Below is a set of tests designed to properly cover all areas of the project's implementation.

No.	Test Details	Rationale
1	Included **Skeleton Tests**	Useful for beginning of project
2	**Construct PuzzleState from String tests**. Check input string is the same as the PuzzleStates output. toString()	Useful for other testing. Make sure the constructor (apply(String)) is working correctly.
3	**Game move tests.** Includes board boundary tests, valid move tests and next state after move test.	It is important that all game moves are working correctly before moving onto any other parts of the project. This includes situations where index out of bounds might occur at the boundaries of the game board.
4	**solve() tests.** Check the returned result gives a PuzzleState Sequence with both the initial/terminal states. Also make sure solve() functions when given a non-initial yet solvable state.	These are the only specified resulting conditions that we are able to check regarding solve(). It behaves the way we want.
5	**Animate() tests.** Make sure that the output from animate corresponds to the correct image.	Make sure what's being visually represented is correct. (unable to get these tests functioning correctly.