Principles of Programming Languages

Final assignment
“Santorini”

Announcement date: 13/01/2021
Deadline: 20/02/2021

Introduction

The assignment’s main requirement is the implementation of the board game “Santorini”.

“Santorini” is a board game played on a 5x5 grid. For this project, we will have two players, but in general there can be up to four. From now on, we will assume there is a blue and a red player, and the blue player gets to play the first turn.

Initially, the blue player places two pawns anywhere on the board (but not in the same cell). After that, the red player does the same, ensuring that no two pawns are placed on the same cell. Then, at every turn, the current player moves one of their pawn to one of the 8 neighbouring cells. After that, they also build up a level on one of the new cell’s neighbouring cells.

In any cell, there can be at most one pawn at any time. Pawns can go down as many levels as we want during a move, but may only go up at most one level. For example, they can move from a cell with two levels down to one with zero levels, but not the other way around. We do not have a restriction on where a pawn can build (except that it must build on a neighbouring cell).

When a player builds on a cell with three levels, we think of a dome being put on top of that cell. This means that for the rest of the game, no pawn can move to or build upon this cell. It’s as if this cell has been sent to the shadow realm. A player can win by moving a pawn to a cell with three levels, or by eliminating their opponent’s legal moves (like a checkmate).

Visualization – Internal Representation

To ease the process of implementing and debugging the game, we provide a visualization kit which brings your data structures to life in a graphical environment where you can actually play the game, provided you have implemented enough of the requirements to do so.

For this scheme to work, we define a certain back-end to front-end communication protocol, which is described below. Your implementation must abide by this protocol.

Required Data Structure

You are expected to implement a **Game** data structure, which will maintain the information that you deem necessary for representing the state of the game. Through this data structure, you must be able to provide all the information needed by the visualization kit.

Required Functions

To make the functions more intuitive, we define some type synonyms here that your are free to use in your own implementation, although it is not mandatory.

```
type RowPos = Int
type ColumnPos = Int
type Position = (RowPos, ColumnPos)

type PlayerPositions = (Position, Position)
type BluePlayerPositions = PlayerPositions
type RedPlayerPositions = PlayerPositions

type Height = Int
type Building = (Height, Position)
type BuildingsList = [Building]

type Turn = Char
type Depth = Int
```

Game modeling

1. **initializeGame** ::
BluePlayerPositions → RedPlayerPositions → Game (20%)

In this function, you will be given two pairs of initial positions of the pawns in the form ((Int, Int), (Int, Int)). You must properly initialize the **Game** data structure you have defined and return the created object. Positions are given in the from (row, col). The position (0, 0) refers to the leftmost, topmost cell while (4, 4) refers to the rightmost, bottommost cell.

2. **tryMove** :: Game → (Position, Position, Position) → Game (40%)

The first **Position** tells us where the pawn to move is currently located. The second one tells us where we would like to move this pawn to. Finally, the last position specifies the cell upon which we plan to build a level. This function should apply the move if and only if it is “legal”. We return the **Game** instance that we should have if this move is indeed applied to the input **Game**. If the move is “illegal”, we should get back the original **Game**.

A move is “legal” if and only if:

- The moving pawn (if indeed one exists where specified) belongs to the player in turn.
- The cell to which the pawn moves must be adjacent to the initial cell of the pawn (at most 1 unit away in either or both directions). Additionally, this cell must not already contain a pawn or have a dome.
- The same restrictions apply to the third position, with regard to the second one.
- The moving pawn rises at most one level. Thus, if the initial cell has a level of x, the cell where it lands cannot have a level higher than x + 1.

Recall that a player can win by moving a pawn to a three-level cell or by eliminating the opponent’s legal moves for the next turn. At every turn, you must check if any of these conditions apply and update your **Game** data structure accordingly.

3. **screenshotGame** :: Game → (Bool, Turn, BluePlayerPositions, RedPlayerPositions, BuildingsList) (10%)

This function is necessary for the visualization kit to work and for your implementation to be assesed. It essentially implements the communication protocol we talked about earlier.
If the game has ended, the Bool value should be true and Turn specifies the player that won. If the game is still in play, Turn specifies the player to play in this turn. If blue is playing we return ‘B’, else ‘R’.
The BluePlayerPositions refer to the current positions of blue’s two pawns. The same applies to red.
For every cell with at least one level, data in the form of (Int, (Int, Int)) is included into the BuildingsList. The first Int specifies the level, while the pair of Ints denotes the position of the cell. A level of 4 means we have a dome.

4. **undoMove** :: Game → Game (20%)

This function must revert the changes made by the last move applied (either by **tryMove** or by **redoMove**) in the **Game** given as input. We must be able to call it multiple times, even so far as to go back to the state return by **initializeGame**. If there is no move to undo, simply return the input **Game**.

5. **redoMove** :: Game → Game (10%)

It is pretty simple to guess what this function must do. We must be able to call it as many times as we called **undoMove**. However, if after a corresponding call of **undoMove**, we have **applied** (and not ignored due to illegality) some move through **tryMove**, then **redoMove** is not expected to do anything (and would not be able to as there is an inherent ambiguity).

In essence these two functions implement the functionality of the ctrl + Z and ctrl + Y shortcuts. You will be able to call them in the same way in the visualization but without the ctrl, for technical reasons. You may find “Zippers” helpful, which are described in detail here.

AI player implementation (Bonus 20%)

In the code given, an option to play against the computer has been partially implemented. The Minimax algorithm is given, and you are expected to fill in the essential parts. These are described below.

6. **possibleMoves** :: Game → [(Position, Position, Position)] (Bonus 10 %)

This function returns all the legal moves available to the player currently in turn. It supplies the Minimax algorithm with the way to branch out different scenarios. The complexity of this functions directly affects Minimax's complexity, so try to implement it as efficiently as possible.

7. **evaluateState** :: Turn → Game → Int (Bonus 10%)

This function is expected to be a heuristic function that estimates, given the current **Game's** setup, how likely the player in turn is to win. If for example the blue player is playing and they have a pawn next to a three-level cell, it should return a very large value, since blue has pretty much already won.
As you may realize, this is essentially the "brains" of your AI player. Altough the design of heuristic functions can vary greatly depending on your imagination, a hint would be to look at how high the player's pawns are currently, how many and how high buildings are near, the same information for the opponent and so on.
Our goal here is to balance efficiency with accuracy. In principle, one could check what happens in every possible outcome of the game and simply return 0 or 1 depending on whether the player has a winning strategy. This however is obviously going to take an exponential amount of time and thus is not suitable. On the flip side, a very simplistic and easy to compute function most probably won't capture the essence of what it means for a player to be close to winning.
In order to receive full marks, the heuristic must not be very slow. The Minimax algorithm should be able to terminate within 1-2 minutes for depth 3. Also, the function must not be very simplistic. If you just follow the hints given (not in a very trivial way), you can say you've done enough. There's always room for improvement for the contest though.

How to play the game

To play the game, you must first compile the visualization with the command "ghc visualization". The gloss library is necessary, which you can install through cabal. Cabal is included in the haskell-platform. To install gloss you simply give the command "cabal install gloss".

After that, just run the executable (./visualization). Initially, you click on 4 cells which will specify the initial positions of the pawns (**initializeGame's** input). Then, every triplet of clicks (on cells) specifies an input **tryMove**. With z, y you can undo and redo respectively, given that you have implemented the corresponding functions.
By pressing a you can toggle between Human vs Human and Human vs Computer. Don't be discouraged if your AI player does not do that well, since the current implementation of Minimax is far from optimized and only lets the AI shine so much. The default depth given is 3. By pressing + or – you may increase or decrease the depth.

Best Heuristic Contest

After the initial grading, we will hold a contest (for those who wish to participate) where we will essentially compare the implementations of the function **evaluateState**.
A "match" will go down as follows. The teams get to pick the initial positions. The movement of pawns will be carried out based on the Minimax algorithm. For depth 2 and 3, we will have two games (with alternating first team). If one team wins more times than the other, they take the match. In case of a tie, the first team to play will be decided on a dice roll, and the tie-breaker will be carried out with a depth of 3 or 4, depending on the turnaround times of Minimax. Finally, the contest will be similar to an elimination tournament.

1st place : Bonus 10%
2nd place : Bonus 8%
3rd place : Bonus 5%

Some execution examples

```
*StudentCode> game1 = initializeGame ((0, 0), (1, 1)) ((2, 2),(3, 3))

*StudentCode> screenshotGame game1
(False,'B',((0,0),(1,1)),((2,2),(3,3)),[])

 --legal move
*StudentCode> game2 = tryMove game1 ((0, 0), (1, 0), (2, 0))

 --state changed
*StudentCode> screenshotGame game2
(False,'R',((1,0),(1,1)),((2,2),(3,3)),[(1,(2,0))])

--illegal move since pawn would land on another pawn (at (1, 1))
*StudentCode> game3 = tryMove game2 ((1, 0), (1, 1), (1, 2))

--no change of state
*StudentCode> screenshotGame game3
(False,'R',((1,0),(1,1)),((2,2),(3,3)),[(1,(2,0))])


*StudentCode> game4 = undoMove game3

*StudentCode> screenshotGame game4
(False,'B',((0,0),(1,1)),((2,2),(3,3)),[])


*StudentCode> game5 = redoMove game4

*StudentCode> screenshotGame game5
(False,'R',((1,0),(1,1)),((2,2),(3,3)),[(1,(2,0))])
```

```
*StudentCode> possibleMoves game5
[((2,2),(1,2),(0,1)),((2,2),(1,2),(0,2)),((2,2),(1,2),(0,3)),((2,2),(1,2),(1,3)),((2,2),
(1,2),(2,1)),((2,2),(1,2),(2,2)),((2,2),(1,2),(2,3)),((2,2),(1,3),(0,2)),((2,2),(1,3),
(0,3)),((2,2),(1,3),(0,4)),((2,2),(1,3),(1,2)),((2,2),(1,3),(1,4)),((2,2),(1,3),(2,2)),
((2,2),(1,3),(2,3)),((2,2),(1,3),(2,4)),((2,2),(2,1),(1,2)),((2,2),(2,1),(2,0)),((2,2),
(2,1),(2,2)),((2,2),(2,1),(3,0)),((2,2),(2,1),(3,1)),((2,2),(2,1),(3,2)),((2,2),(2,3),
(1,2)),((2,2),(2,3),(1,3)),((2,2),(2,3),(1,4)),((2,2),(2,3),(2,2)),((2,2),(2,3),(2,4)),
((2,2),(2,3),(3,2)),((2,2),(2,3),(3,4)),((2,2),(3,1),(2,0)),((2,2),(3,1),(2,1)),((2,2),
(3,1),(2,2)),((2,2),(3,1),(3,0)),((2,2),(3,1),(3,2)),((2,2),(3,1),(4,0)),((2,2),(3,1),
(4,1)),((2,2),(3,1),(4,2)),((2,2),(3,2),(2,1)),((2,2),(3,2),(2,2)),((2,2),(3,2),(2,3)),
((2,2),(3,2),(3,1)),((2,2),(3,2),(4,1)),((2,2),(3,2),(4,2)),((2,2),(3,2),(4,3)),((3,3),
(2,3),(1,2)),((3,3),(2,3),(1,3)),((3,3),(2,3),(1,4)),((3,3),(2,3),(2,4)),((3,3),(2,3),
(3,2)),((3,3),(2,3),(3,3)),((3,3),(2,3),(3,4)),((3,3),(2,4),(1,3)),((3,3),(2,4),(1,4)),
((3,3),(2,4),(2,3)),((3,3),(2,4),(3,3)),((3,3),(2,4),(3,4)),((3,3),(3,2),(2,1)),((3,3),
(3,2),(2,3)),((3,3),(3,2),(3,1)),((3,3),(3,2),(3,3)),((3,3),(3,2),(4,1)),((3,3),(3,2),
(4,2)),((3,3),(3,2),(4,3)),((3,3),(3,4),(2,3)),((3,3),(3,4),(2,4)),((3,3),(3,4),(3,3)),
((3,3),(3,4),(4,3)),((3,3),(3,4),(4,4)),((3,3),(4,2),(3,1)),((3,3),(4,2),(3,2)),((3,3),
(4,2),(3,3)),((3,3),(4,2),(4,1)),((3,3),(4,2),(4,3)),((3,3),(4,3),(3,2)),((3,3),(4,3),
(3,3)),((3,3),(4,3),(3,4)),((3,3),(4,3),(4,2)),((3,3),(4,3),(4,4)),((3,3),(4,4),(3,3)),
((3,3),(4,4),(3,4)),((3,3),(4,4),(4,3))]

*StudentCode> evaluateState 'R' game5
-3
```

Submission guidelines

In the files given you can find StudentCode.hs. This is where you will fill in the requirements of the assignment. Do not change this file's name, since it will break the visualization. This file is your entire submission.

Questions

For any sort of problem or question, do not hesitate to contact sdi1700069@di.uoa.gr.

Konstantinos Lakis