LASCA GAME WITH MINMAX AI

Pietro Pepe

1 Proposal

This is an implementation of the board game Lasca, using OpenGL and Glut. In this draughts variant, pieces are not removed from the board when they are jumped. Instead, they are placed under the piece that jumped them, forming a stack.

This is a single human player game with an opponent AI implemented with MinMax tree, where the player controls the blue pieces, while the computer is responsible for the red ones. Both of them **NEED** to jump a piece, if there is one available, and one piece can and also need to make consecutive jumps.

2 General Architecture

The following diagram represents the software modules:

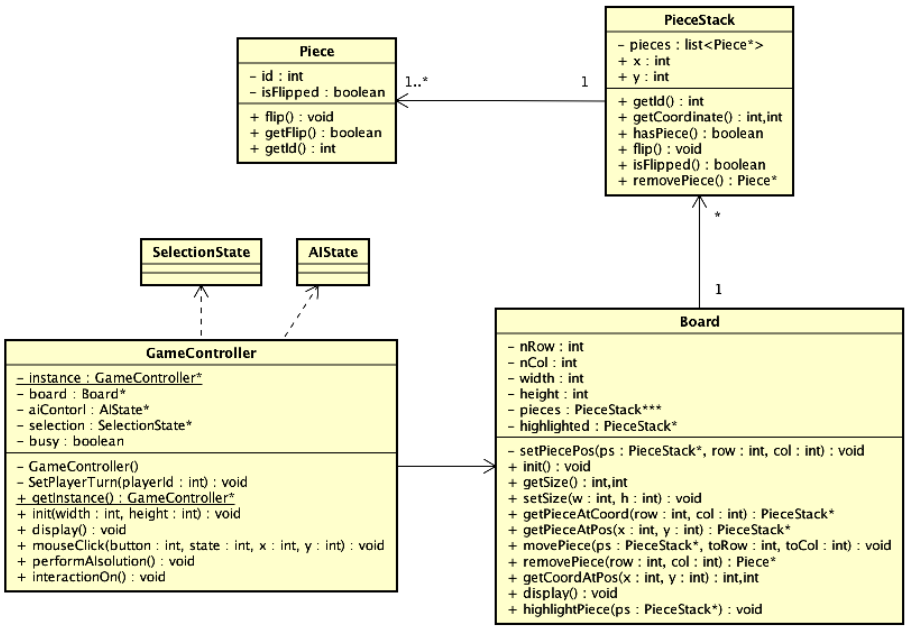

Fig. 1: Lasca game modules

A general overview:

Piece	basic representation of a single game piece
PieceStack	manages a piece stack, with add/removal functionalities
Board	responsible to manage the game board, moving and getting pieces at specific screen or table coordinates, as required.
SelectionState	auxiliary module to control human player movement selection with mouse clicks
AIState	implementation of MinMax search tree algorithm for computer movement selection
GameController	main applicatino module, generates game and implement game rules, connecting other modules

3 Rules

The following rules are implemented:

- PieceStack move accordingly to top piece
- pieces can only move diagonally. They move forward and can only go backwards if they are flipped (represented by a star drawing on them)
- the top piece of a stack gets flipped if and only if they reach the last opposite row
- the player can only make single moves if there is no jumping move available

3.1 AI logic

AI is implemented with a MinMax search tree. This method consists in generating a tree of all game states (i.e. board configurations) that can be generated by considering every possible movement sequence of a given length. Each node of the tree represents a state, and they have one child node for each possible movement, leading to next player turn.
MinMax applies a scoring function to represent how good a state is for the human player. Considering both players play optimally, the algorithm aims to choose a path where the computer minimizes the maximum possible score the human can have.

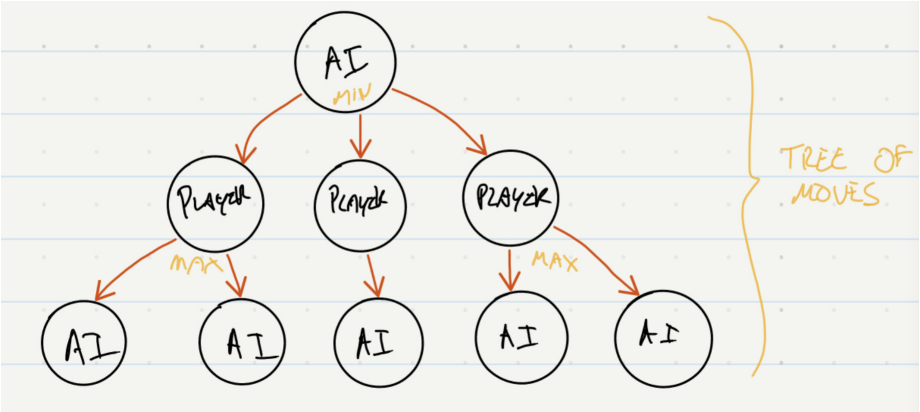

Fig. 2: MinMax tree

To simplify, we use the whole Board class to represent a state, but we avoid generating a whole tree of **Board**s, since that would grow memory use exponentially, and limit our tree depth. The algorithm run by running through the tree with a single **Board**, applying and reversing movements as it travels:

1. If the board has possible movements, and hasn't reached max depth, it lists those. Otherwise, returns the current scoring + movement
2. For each movement:

   Apply the movement to the board, modifying its state and scoring

   Perform step 1, gets the score, and verifies if its maximum/minimum

   Revert the movement
3. Returns the best scoring + movement

Using depth 8, the algorithm run almost instantly at the beginning of the game, but the search gets more intensive as it progresses, since pieces have more space on the board, and then, more possible movements, affecting the number of branches and the exponential growth of the search. Even though, in end states of the game, depth 8 runs usually in less than 1 second (not including the 1 second delay for computer movement applied between player's turns).

The scoring function can be tuned to make the AI more or less competitive. In this implementation the function is basically proportional to, and only to, the number of piece stacks currently belonging to each player. A more competitive approach, would be also considering weights to certain advantageous board configurations, and to reward flipped pieces, considering the flexibility they give to the player.