Report

**Description**

**Board**

Board uses two vectors to store the beans for the holes on the north side and the south side (std::vector<int> m\_north and std::vector<int> m\_south). Additionally, it has four int variables that keep track of the total beans on the Board (m\_totalBeans), the number of holes on the sides not including the pots (m\_sideHoles), and the number of beans in the north and south pot (m\_northPot and m\_southPot).

**Player**

Player includes the base class Player as well as the derived classes HumanPlayer, BadPlayer, and SmartPlayer. The Player class uses a string variable to keep track of the name of the player (m\_name). SmartPlayer has two additional private helper functions chooseMove and evaluate to help calculate the most optimal move using the minimax algorithm.

**Game**

Game consists of a Board (m\_board), two players for the north and south side (m\_southPlayer and m\_northPlayer), a Player variable to keep track of the player currently acting (m\_activePlayer), and a Side variable to keep track of the side currently acting (m\_activeSide).

**SmartPlayer::chooseMove**

SmartPlayer’s chooseMove function calls a helper function chooseMove with additional parameters such as the original side, value, depth, time limit, and timer that help create a more comprehensive function. Ultimately, it returns the bestHole parameter obtained from the helper function.

In the helper function, we loop through the Board to determine the possible moves. We then evaluate the moves by giving them a score based on if we win (infinity), lose (-infinity), tie (0), and otherwise, the score is the number of beans in our side’s pot - the number of beans in the other side’s pot. We recursively call the helper function to determine how the opponent may move after our turn, and how we may move after the opponent, etc, only stopping once we reach a certain depth or if the timer exceeds five seconds.

**Psuedocode**

**bool Board::sow(Side s, int hole, Side& endSide, int& endHole)**

// if hole is out of bounds or empty

// return false

// if the side is north

// remove the beans from the hole

// while there are still beans

// if you’re at the north pot and about to go to the south side

// add a bean into the pot

// subtract a bean from hand

// if there are still more beans

// go to the south side’s first hole

// else, you’re in the pot

// if you’re about to go to the north side

// go to the north side’s last hole

// else

// if you’re currently on the north side

// add a bean to the hole

// move counterclockwise

// subtract a bean from hand

// set endSide and endHole to the right side and hole

// repeat for the south side

// basically the same thing but some things are reversed (some things are incremented on north’s side while it’s decremented on south’s side due to the counterclockwise nature of things)

**bool Game::move(Side s)**

// if the north side has no more beans

// sweep the remaining beans

// display the board

// if the south side has no more beans

// sweep the remaining beans

// display the board

// display whose turn it is

// allow activePlayer to choose which hole they wish to sow

// sow the beans from the hole

// display the board

// if the last bean lands in the respective pot

// go again

// if the last bean is placed in one of the player’s own holes that was just empty, and the opponent’s hole directly opposite is not empty

// capture the beans

// display the board

// change the activePlayer and activeSide

// return true

**void Game::play()**

// display the board

// while the player can move

// make a move

// check the status of the game

// if the game is over

// if the game has a winner

// display who won (one side, or draw)

**int BadPlayer::chooseMove(const Board& b, Side s) const**

// if on the north side

// loop through the side holes

// if the hole is not empty

// pick that hole

// if on the south side

// loop through the side holes

// if the hole is not empty

// pick that hole

**int HumanPlayer::chooseMove(const Board& b, Side s) const**

// if the game is over

// return -1

// prompt the player to select a move

// while the hole isn’t a valid hole

// display the error

// reprompt the user

**int SmartPlayer::chooseMove(const Board& b, Side s) const**

// set up the parameters for the helper function

// call the helper function

// return the best move found

**void SmartPlayer::chooseMove(const Board& b, Side s, Side og, int& bestHole, int& value, int depth, double timeLimit, JumpyTimer& timer) const**

// create a copy of the board

// if there are no more beans on the board

// return

// if the time limit is reached or the depth is exceeded

// return

// loop through all side holes

// if there is a possible move (the hole isn’t empty)

// evaluate the move

// if the side acting is the opponent

// take the negative of the value

// recursively call chooseMove to determine the opponent’s possible moves

// unmake the move on the copy of the board

// initialize the comparison value variable to the first node

// if the initial side has a greater value or if the opponent side has a smaller value

// set the best hole to this hole

// set the value to this value

// return

**int SmartPlayer::evaluate(Board& b, Side s, int hole) const**

// sow the beans in the given hole

// capture the beans at the end if conditions are met

// if the north side has no more beans

// sweep the remaining beans

// determine the value based on if the given side will win, lose, or draw

// if the south side has no more beans

// sweep the remaining beans

// determine the value based on if the given side will win, lose, or draw

// if the bean lands in the pot

// loop through all the possible holes

// if the hole isn’t empty

// evaluate the move

// if that move’s value is greater than the current best value

// set the best value to this value

// set the best copy of the board to this copy

// reset the copy of the board back to the original

// set the original board to the best copy of the board

// return the best value

// return the difference in beans between this side and the opponent’s side

**Problems**

As of this moment, there are no known bugs or serious inefficiencies. A minor improvement would be changing the SmartPlayer::chooseMove function so that it explores branches based on the time limit rather than a fixed depth. A notable problem while working on this project was implementing SmartPlayer::chooseMove and testing to see if the algorithm was working optimally.

**Test Cases**

In addition to the provided test cases which covered most Board functions, I also used the provided Board in the spec to test all spots in the provided game tree by iterating through SmartPlayer::chooseMove:

SmartPlayer bp1(“Human");

SmartPlayer bp2("Smart Bot”);

Board b(6, 4); // 6, 4

b.setBeans(NORTH, 0, 22);

b.setBeans(NORTH, 1, 0);

b.setBeans(NORTH, 2, 1);

b.setBeans(NORTH, 3, 0);

b.setBeans(NORTH, 4, 0);

b.setBeans(NORTH, 5, 2);

b.setBeans(NORTH, 6, 0);

b.setBeans(SOUTH, 0, 20);

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 0);

b.setBeans(SOUTH, 3, 2);

b.setBeans(SOUTH, 4, 0);

b.setBeans(SOUTH, 5, 1);

b.setBeans(SOUTH, 6, 0);

Game g(b, &bp2, &bp1);

g.play();

Board::sow was first tested by constructing Board(4, 4) and individually sowing each spot. Other tests were conducted by playing multiple games of Kalah, either by pitting two SmartPlayers against each other, or how a BadPlayer/SmartPlayer would react against a HumanPlayer (me).