**Description of Design of Classes:**

* *Class Board:*
  + The major data structure I used was vectors to visualize the game’s holes. It would start with 0 being the pot and then each signifying the hole number.
  + I had private variables int m\_holes for the number of holes in the game (not including the pot) and int m\_beans for the number of beans in each hole. I also had private variables of 2 vectors as mentioned above, signifying the north and south.
* *Class Player & HumanPlayer & BadPlayer:*
  + For this, I just implemented the class using inheritance as SmartPlayer and BadPlayer were both derived from the class Player.
  + I had a private variable string m\_name to signify the name of the player for displaying purposes in class Player.
* *Class SmartPlayer:*
  + There were no major data structures I used, but if the minimax algorithm counts, it was used and it was the most trivial function.
  + I had 4 private member functions.
    - The first one is bool chooseMoveOptimizer which returns if we can make a move that can get us an additional move since this is the highest precedence in the kalah game (getting an additional move is the best move possible in kalah) If it could, then it would manipulate one of the parameters which is int &hole to that hole that gets us an additional move. If there is no move that can get us an additional move, we return false.
    - The second one was int evaluate which given a position, returns a number that measures how good that position is. It would return a high value if it is good for the south and a low value if it is a good position for the north (aka bad position for south).
    - The third was bool completedMove which returns if we can make a move or not for the SmartPlayer.
    - The last was void minimax which was the most complicated function and did most of the work. It practically finds what is considered to be the best play available using the evaluate function explained above. It then manipulates one of the input parameters which is int &bestHole which will then be used for the chooseMove function.
* *Class Game:*
  + The major data structure I used here was a pointer to the players in the south and north.
  + Private variables included Board m\_board to signify the game board, Player \*m\_south, Player \*m\_north, pointing to each player, and Side m\_side to signify who’s side the game is in.

**Description of Design for SmartPlayer::chooseMove**

* The heuristic I used to evaluate the board positions was comparing the number of beans on the north and south side, including the pots. Initially I was only checking the pots as it said to do so in the specs, but because we could not determine if it was a losing move or not (because we did not know which side we were on; status function this was in the game file which player did not include -> game includes player we don’t want circular) I decided to check the entire side. I would then compare the pots only when we run out of time or we reach the depth cap which is the base case of the recursion in the minimax function. In a sense, it is kind of like the final check of the status of the game.
* The SmartPlayer::chooseMove function utilizes all 4 of the private member functions.
  + It first checks if it can make a move to get itself an additional turn with the chooseMoveOptimize function. If it can, it will return that hole.
  + If it cannot, then it will refer to the minimax function using depth and timeLimit. The minimax function then follows the pseudocode and calls the evaluate function to see how well that position is and depending on if it is south or north, will modify the value and bestHole. If we can complete the move (aka call the completedMove function) we recursively call it with a deeper depth and less timeLimit (time limit for each branch) and when we run out of time or reach our depth cap (whichever comes first), we will return what was considered best at that point and use that.
  + I included both time and depth since depending on the scale of the game, it could go over 5 seconds if we only have depth as the parameter.
    - For example, a depth of 5 in a game with 3 holes may take 1 second but a depth of 5 in a game with 20 holes may take more than 5 seconds. The timer would then ensure it would not go past 5 seconds no matter what.
* *\*The pseudocode for this will be below in the non-trivial algorithm’s section*

**Pseudocode for Non-Trivial Algorithms:**

* *Class Board:*

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

{

if the hole is empty, invalid or a pot

return false;

store the number of beans in that hole in temp

take out all the beans in hole

if side is NORTH

set the current side as NORTH

set the current hole as hole - 1 (since we start with hole to the left)

while we have beans in temp

if we hit NORTH’s pot

add 1 bean to the pot

subtract a bean from temp

switch sides to SOUTH

set current hole as 1 (to start from 1)

if we are just on the NORTH side

add 1 bean to the hole

subtract a bean from temp

decrement current hole

if we hit SOUTH’s pot

switch sides to NORTH

set current hole to number of holes in game

if we are just on the SOUTH side

add 1 bean to the hole

subtract a bean from temp

increment current hole

if we end on SOUTH’s first hole

set endSide to NORTH

set endHole to 0

else if we end on NORTH

set endSide to NORTH

set endHole to the hole after our current hole

else if we end on SOUTH

set endSide to SOUTH

set endHole to the hole before our current hole

return true;

if side is SOUTH

apply similar logic as NORTH

}

* *Class HumanPlayer:*

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

{

initialize int moveChoice

set a boolean chosenMove to false

while chosenMove is false

cout << “Please choose a hole “ << name() << “: “;

cin >> moveChoice;

if there are no beans at that hole

cout << “No beans in this hole” << endl;

else if they chose an invalid hole or a pot

cout << “Invalid hole number!” << endl;

else (they chose a valid move)

set chosenMove to true)

return moveChoice;

}

* *Class BadPlayer:*

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

{

initialize int moveChoice

set a boolean chosenMove to false

while chosenMove is false

cout << “Please choose a hole “ << name() << “: “;

cin >> moveChoice;

if there are no beans at that hole

cout << “No beans in this hole” << endl;

else if they chose an invalid hole or a pot

cout << “Invalid hole number!” << endl;

else (they chose a valid move)

set chosenMove to true)

return moveChoice;

}

* *Class SmartPlayer:*

int SmartPlayer::evaluate(const Board& b)

{

initialize value to the difference of beans between SOUTH and NORTH

if value is less than -10

return negative infinity

if value is greater than -10

return positive infinity

else

return value

}

bool SmartPlayer::chooseMoveOptimize(const Board& b, Side s, int& hole)

{

for every hole on the board

create a copy of the original board

if we can sow on a that duplicate board & we can make an additional move

update hole to that current hole

return true

return false

}

bool SmartPlayer::completedMove(Board &b, Side s, int hole, Side & endSide, int & endHole) const

{

if we can sow for that position

if we get an additional turn

return false

if we can capture

do sweeping mechanism

return true

else (if we cannot sow)

return false

}

void SmartPlayer::minimax(const Board& b, Side s, int& evaluatedValue, int& bestHole, int depth, double timeLimit, JumpyTime & j)

{

initialize firstLegalMove to 1

if no move for player exist

initilialize bestHole to -1

initialize evaluatedValue to what the evaluate function gives us

return

if we run out of timeLimit or we reach the depth of 5 (recursion base case)

initialize bestHole to -1

initialize evaluatedValue to the difference of SOUTH and NORTH POT

return

for every hole h the player can choose

initialize thisBranchTimeLimit to timeLimit / branchesLeftToExplore

initialize statTime to JumpyTimer’s elapsed time

if there are no beans in that hole

increment firstLegalMove

continue

if we can complete the move

recursion the minimax function w/ depth + 1 & thisBranchTimeLimit

if we cannot complete the move

recursion the minimax function w/ same depth & thisBranchTimeLimit

if the current hole is the same as firstLegalmove

update bestHole to current hole

update evaluatedValue to the value retrieved from recursive call

if we are on SOUTH side & the retrieved evaluatedValue is > than original

update bestHole to current hole

update evaluatedValue to the value retrieved from recursive call

if we are on NORTH side & the retrieved evaluatedValue is < than original

update bestHole to current hole

update evaluatedValue to the value retrieved from recursive call

decrease timeLimit with elapsed time

}

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

{

int hole;

if we can chooseMoveOptimize (aka we can get an additional move)

return hole retrieved from chooseMoveOptimize

else (we cannot get an additional move)

initialize depth to 0

initialize timeLimit to 4990

generate a JumpyTimer

call minimax passing in timeLimit, depth, and JumpyTimer

return bestHole retrieved from minimax

}

* *Class Game:*

void Game::status(bool& over, bool& hasWinner, Side& winner) const

{

for every hole on the board

if there are beans on any holes

set over to false

return

set over to true

if SOUTH pot has more beans than NORTH pot

set hasWinner to true

set winner to NORTH

if NORTH pot has more beans than SOUTH pot

set hasWinner to true

set winner to SOUTH

else (case of a tie)

set hasWinner to false

}

void Game::move(Side s)

{

if we are on NORTH side and there are no beans in the NORTH

sweep all beans into the SOUTH pot

display the screen

return false

if we are on SOUTH side and there are no beans in the SOUTH

sweep all beans into the NORTH pot

display the screen

return false

if we are on NORTH

initialize selectedHole to m\_north’s chooseMove

if we are on SOUTH

initialize selectedHole to m\_south’s chooseMove

if we can sow using selectedHole

if are at capture

do sweeping mechanism

return true

if we are ad additional move

display()

recursively try the move again

return true

else (we cannot sow)

return false

}

**Notes about Bugs, Serious Inefficiencies, or Notable Problems:**

* The SmartPlayer artificial intelligence of course loses some cases because we are adding a cap to the depth search or time limit. If it were given infinite time to search through the entire tree, it would be most intelligent and make the best choice. (also as mentioned before, it would also depend on how big the board is and how many beans are in it but from testing, I found that depth of 5 is the best depth for a reasonable game board. If the board is any bigger to the point depth of 5 takes longer than 5 seconds, it would just test until 5 seconds.)

**Test Cases (& its purpose):**

* For test cases for playing and testing players, I mostly did this by playing the game for numerous hours. I broke it into every possible category of playing, which would be 1 human vs 1 human, 1 human vs 1 bad bot, 1 bad bot vs 1 bad bot, 1 smart bot vs 1 bad bot, 1 smart bot vs 1 smart bot, 1 human vs 1 smart bot.
* I specifically put my focus into 1 smart bot vs 1 bad bot as it should win more than losing. I adjusted the board’s parameters with different numbers of holes and different numbers of beans and recorded which cases led to me winning and which ones led to me losing. I also adjusted the depth of the search to find the optimal depth that leads to the most wins which I found out was 5.
* I also made sure that if the board was very complicated (aka large) that it would terminate at 5 seconds and nothing more as the specs said to do so.

**// testing for board**

void doBoardTests()

{

Board b(3, 2);

assert(b.holes() == 3 && b.totalBeans() == 12 && b.beans(SOUTH, POT) == 0 && b.beansInPlay(SOUTH) == 6);

b.setBeans(SOUTH, 1, 1);

b.moveToPot(SOUTH, 2, SOUTH);

assert(b.totalBeans() == 11 && b.beans(SOUTH, 1) == 1 && b.beans(SOUTH, 2) == 0 && b.beans(SOUTH, POT) == 2 && b.beansInPlay(SOUTH) == 3);

Side es;

int eh;

b.sow(SOUTH, 3, es, eh);

assert(es == NORTH && eh == 3 && b.beans(SOUTH, 3) == 0 && b.beans(NORTH, 3) == 3 && b.beans(SOUTH, POT) == 3 && b.beansInPlay(SOUTH) == 1 && b.beansInPlay(NORTH) == 7);

b.sow(NORTH, 3, es, eh);

assert(es == NORTH && eh == 0 && b.beans(NORTH, 3) == 0 && b.beans(NORTH, 2) == 3 && b.beans(NORTH, 1) == 3 && b.beans(NORTH, POT) == 1 && b.beansInPlay(SOUTH) == 1 && b.beansInPlay(NORTH) == 6);

// additional test cases

// board with a lot of beans

Board a(4, 20);

assert(a.holes() == 4 && a.totalBeans() == 160 && a.beans(SOUTH, POT) == 0 && a.beansInPlay(SOUTH) == 80);

Side as;

int ah;

a.sow(SOUTH, 4, as, ah);

assert(as == NORTH && ah == 4 && a.beans(NORTH, 4) == 23 && a.beans(NORTH, 3) == 22 && a.beans(NORTH, 2) == 22 && a.beans(NORTH, 1) == 22 && a.beans(NORTH, POT) == 0 && a.beans(SOUTH, 1) == 22 && a.beans(SOUTH, 2) == 22 && a.beans(SOUTH, 3) == 22 && a.beans(SOUTH, 4) == 2 && a.beans(SOUTH, POT) == 3 && a.beansInPlay(SOUTH) == 68 && a.beansInPlay(NORTH) == 89);

// test moveToPot function

a.moveToPot(NORTH, 1, NORTH);

assert(a.beans(NORTH, POT) == 22);

a.moveToPot(SOUTH, 4, SOUTH);

assert(a.beans(SOUTH, POT) == 5);

a.moveToPot(NORTH, 2, NORTH);

assert(a.beans(NORTH, POT) == 44);

a.moveToPot(NORTH, 3, SOUTH); // moving to opponent's pot

assert(a.beans(SOUTH, POT) == 27);

// board with 1 hole

Board c(1, 5);

assert(c.holes() == 1 && c.totalBeans() == 10 && c.beansInPlay(NORTH) == 5);

// board with 0 hole

Board d(0, 4);

assert(d.holes() == 1 && d.totalBeans() == 8);

// board with negative holes should act like 1

Board e(-1, 5);

assert(e.holes() == 1 && e.totalBeans() == 10);

// board with negative beans should act like 0

Board f(1, -3);

assert(f.holes() == 1 && f.totalBeans() == 0);

}

**// testing for player**

void doPlayerTests()

{

HumanPlayer hp("ME");

assert(hp.name() == "ME" && hp.isInteractive());

BadPlayer bp("DUMMY");

assert(bp.name() == "DUMMY" && !bp.isInteractive());

SmartPlayer sp("GENIUS");

assert(sp.name() == "GENIUS" && !sp.isInteractive());

Board b(6, 4);

b.setBeans(SOUTH, 5, 0);

b.setBeans(NORTH, 3, 0);

// HumanPlayer test (make sure it only chooses possible move and reprompts)

int n = hp.chooseMove(b, SOUTH);

assert(n == 1 || n == 2 || n == 3 || n == 4 || n == 6);

// BadPlayer test (make sure it only chooses possible moves)

n = bp.chooseMove(b, NORTH);

assert(n == 1 || n == 2 || n == 4 || n == 5 || n == 6);

// Smartplayer test (make sure it only choose possible moves and additional move if possible)

// it chooses the hole 4 since it gives us an additional move

n = sp.chooseMove(b, NORTH);

assert(n == 4);

}

**// testing for game**

void doGameTests()

{

BadPlayer bp1("Joseph");

BadPlayer bp2("Lenny");

Board b(5, 0);

b.setBeans(SOUTH, 1, 7);

b.setBeans(NORTH, 2, 4);

b.setBeans(NORTH, 3, 3);

b.setBeans(SOUTH, 4, 6);

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

bool over;

bool hasWinner;

Side winner;

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 0 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 4 && g.beans(NORTH, 3) == 3 && g.beans(NORTH, 4) == 0 && g.beans(NORTH, 5) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 6 && g.beans(SOUTH, 3) == 4 && g.beans(SOUTH, 4) == 3 && g.beans(SOUTH, 5) == 0);

g.move(SOUTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 1 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 5 && g.beans(NORTH, 3) == 4 && g.beans(NORTH, 4) == 0 && g.beans(NORTH, 5) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 7 && g.beans(SOUTH, 3) == 5 && g.beans(SOUTH, 4) == 4 && g.beans(SOUTH, 5) == 0);

g.move(NORTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 1 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 5 && g.beans(NORTH, 3) == 0 && g.beans(NORTH, 4) == 1 && g.beans(NORTH, 5) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 7 && g.beans(SOUTH, 3) == 5 && g.beans(SOUTH, 4) == 0 && g.beans(SOUTH, 5) == 1);

g.move(SOUTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 1 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 5 && g.beans(NORTH, 3) == 0 && g.beans(NORTH, 4) == 0 && g.beans(NORTH, 5) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 7 && g.beans(SOUTH, 3) == 5 && g.beans(SOUTH, 4) == 0 && g.beans(SOUTH, 5) == 0);

g.move(NORTH);

g.status(over, hasWinner, winner);

assert(over && g.beans(NORTH, POT) == 1 && g.beans(SOUTH, POT) == 2 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 6 && g.beans(NORTH, 3) == 1 && g.beans(NORTH, 4) == 0 && g.beans(NORTH, 5) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 6 && g.beans(SOUTH, 4) == 1 && g.beans(SOUTH, 5) == 0);

assert(hasWinner && winner == SOUTH);

BadPlayer bp1("Lucas");

BadPlayer bp2("Antonio");

Board b(4, 0);

b.setBeans(SOUTH, 1, 2);

b.setBeans(NORTH, 2, 1);

b.setBeans(NORTH, 3, 2);

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

bool over;

bool hasWinner;

Side winner;

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 0 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 2 &&

g.beans(SOUTH, 1) == 2 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

g.move(SOUTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 1 && g.beans(SOUTH, 3) == 0);

g.move(NORTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&

g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 1 && g.beans(SOUTH, 3) == 0);

g.move(SOUTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&

g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 1);

g.move(NORTH);

g.status(over, hasWinner, winner);

assert(over && g.beans(NORTH, POT) == 1 && g.beans(SOUTH, POT) == 4 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

assert(hasWinner && winner == SOUTH);

}

**// testing 1 human vs 1 human**

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

int main()

{

HumanPlayer hp1("AYAAN");

HumanPlayer hp2("MAX");

Board b(3, 2);

Game g(b, &hp1, &hp2);

g.play();

}

**// testing 1 human vs 1 bad bot**

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

int main()

{

BadPlayer bp1("BOT");

HumanPlayer hp2("MAX");

Board b(3, 2);

Game g(b, &bp1, &hp2);

g.play();

}

**// testing 1 bad bot vs 1 bad bot**

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

int main()

{

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board b(3, 2);

Board c(4, 2);

Board d(5, 2);

Board e(6, 2);

Board f(6, 3);

Board g(6, 4);

Board h(5, 3);

Board i(5, 4);

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

g.play();

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

g.play();

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

g.play();

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

g.play();

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

g.play();

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

g.play();

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

g.play();

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

g.play();

}

**// testing 1 smart bot vs 1 bad bot**

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

int main()

{

SmartPlayer sp("SMARTBOT");

BadPlayer bp("DUMBBOT");

Board b(6, 4);

Game g(b, &bp, &sp);

g.play();

}

**// testing 1 smart bot vs 1 smart bot**

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

int main()

{

SmartPlayer sp1("GENIUS 1");

SmartPlayer sp2("GENIUS 2");

Board b(6, 3);

Game g(b, &sp1, &sp2);

g.play();

}

**// testing 1 human vs 1 smart bot**

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

int main()

{

HumanPlayer hp("MAX");

SmartPlayer sp("GENIUS 1");

Board b(3, 3);

Game g(b, &hp, &sp);

g.play();

}