Robert P. Baumeister

Design document

First, some explanation of the way this program works.

There are two variants of this program, serial and parallel.

The serial variation is only slightly different from the parallel version, in that the alpha-beta-minimax function is initiated in parallel at the top of the recursive calls in the parallel version. The majority of the code will be explained in the context of the serial variation, as the parallel variation requires knowledge of the serial algorithm.

The game being played is Konane. The rules are outlined in the user document. In essence, the game ends when there are no more moves possible. The loser being the person with the next move.

The Alpha-Beta-Minmax algorithm is a recursive algorithm to evaluate the best move based on board states. The recursive calls iterate through every possible move on a board and further calls evaluate more moves, up to a predefined depth (governed by the variable depth), which must be an even number. This function operates in serial and, through alpha-beta cutoffs, is capable of using board states passed up from the leaves to determine whether to explore moves further. This function is called on a board object, which creates recursive nodes passing down the tree

The SEF, or State Evaluation Function, is executed at the leaves of this tree, once it reaches the desired depth. It evaluates the board according to the number of possible moves for the AI minus the number possible for its opponent, and the piece balance of the board (how many pieces are on the board for the AI, less the number available to the other player.). The ideal state has more AI pieces on the board than opponent pieces and/or more possible moves for the AI than for the opponent. The piece balance and total moves are added together and returned to the node which called it. The maximum number found, and the move made to gain it at that point in the tree is chosen. At the next level the minimum is passed up (representing the opponent making the ideal move). This continues to the root of the tree, which calculates one final maximum and the AI chooses the max move.

The majority of the functions within the program are helper-functions for these two functions.

The Alpha Beta Cutoff is a key element of the algorithm, enabling reductions in moves explored by “remembering” previous leaves that will not be explored in the game, or, if the player explores them, they will be making a mistake. Alpha and Beta are the minimum and maximum scores that the maximizer node(Alpha) and minimizing node(Beta) can be certain of. Alpha is initially set to MIN, which is -999, Beta is initially set to MAX, which is 999, scores that will assure that higher and lower values will be selected. If Beta becomes greater than Alpha, the recursive nodes connected to that node will not be examined as (with optimal play), these numbers will not be examined.

The parallel program replaces the root of the recursive tree with a function that initiates a thread on each row of the game board, rather than use one thread for the entire board. The thread then launches the serial alpha-beta-minimax algorithm as described before. The function returns a board object which is compared with all other board objects, returning the one with the best SEF value.

Depth (how many moves ahead the AI looks) is determined at compile time from const.h, this makes testing at different depths difficult, but removes a source of user error and ensures that depth is not set to some undesired level i.e. 18 or 10, which take too long to be useful and too short to be effective respectively.

There are no private variables or functions inside the board class. Everything is public. This is a deliberate design choice, as there is only one class running the game and the object stores all important information. Getters and setters would be redundant and likely result in slower execution time due to the number of times such functions would be called.

This is defined as a God Class in O.O.P., and will be avoided in future code due to the serious difficulty of maintaining the code.

variables in class.

bval records best value, used for parallel root

zpgame stores value for whether or not a zero player game has been chosen.

The two 4 element arrays bestmove and playermove store the values for the best move on a given board at a given time and the best move for a player at a given time.

There is an 8x8 2D array representing the board, Black and white pieces have distinct integer values. Empty spaces are turned into zeros.

The Boolean function setZpgame() is used to toggle whether a zero player game is desired or whether the user wishes to play the AI (and likely lose to it).

The Boolean function setColor() sets the color and the first player (black as per konane rules). The Boolean function guardRails() prevents illegal moves by player and the make\_move() function, which receives an evaluated move, if called on user, prevents out of bounds values by calling the Boolean legal\_move() and passing the move to it. This function is also used by the AI, though in a different capacity. There is a void function that displays the board called display(). manualOvveride allows the user to set up the game and correct any mistakes that may arise in the game (that ability is less necessary since the Boolean guardRails was implemented, which takes a move and determines the legality of that move. The Boolean function isFull is a helper to other functions which returns whether a given space is full. This is used everywhere in the program that such checks are needed. The Boolean legal\_move tests whether a move can be made, and returns true if the move being considered is legal. Legal\_SEF is a similar boolean function that determines whether a move passed into it is possible inside the State Evaluation Function, but it is calibrated to explore opponent moves as well.

The void function makeMove uses the array currentmove (which exists only inside of the AlphaBeta function) to make the move determined by the player. makeMove will always choose double jumps if available and considered optimal.

This is to explore moves on the board (at each depth), and also to make moves on the board when the AlphaBeta, and its parallel variation

Unmake move undoes this in reverse as each node of the recursive tree passes up their integer value to the next node.

The serial variation of alphaBetaMinimax is quite complex. It may only be possible to understand with the explanation in this portion of the document and reference to code and comments inside the listed code itself. It also exists inside the program itself.

Alpha and beta cutoff values are passed into recursive calls, depth is the depth the simulation is at, levelColor is the color being examined at a given level, and a boolean tells the function what kind of level it is at (maximizing or minimizing).

//State Evaluation Function. Does not work alone, uses legal\_SEF pass is for future modifications

The SEF evaluates the board at the lowest level according to moves avialible to each player and the number of pieces on the board.

Selection checks if the move chosen is on the board.

void selection(int S);

displayMove prints off the move numerically.

The board comparison function compares 8 boards at root of tree, this is needed for the serial implementation. The ideal board with the ideal move stored in it is returned and set equal to the board object used by the game. The move stored is then executed on the board according to the value bval, which is the integer value passed up by the AlphaBeta function.

The function threadKonane is called inside main by individual threads. It mimics a maximizing node at the root of the tree and returns the best value for the row of moves on the board being examined. It initiates a minimizing node for the Alpha Beta Minimax function (serial function that returns an integer value). This isolates threads and eliminates all sources of deadlock as the threads will never examine the same portion of the same board.

int calibrate calibrates the for loop for looping through columns so that only black or only white squares are examined (as those are the only important squares for that player).

User Document

The rules of Konane are that every move must be a capture move. Moves can only be vertical or horizontal. A move consists of a black or white piece (X for Black in the program terminal, O for white) moving to an empty space by jumping over a piece with the opposite color. A second, or third jump, is allowed if possible and if desired. Second or third jumps must be in the same direction as the first.

The first player to begin a turn with no possible moves loses. To start, the white player should remove two adjacent tiles (black and white) in either the corners or the center. Black has first move.

There are two modes for this program, zero-player game, and playing the AI.

For zero-player games, you can assign the AI a color, which is not actually relevant, and the AI plays itself to completion of the game. Over the course of the game a CSV will be outputted with the times of each turn taken by the AI.

For a human versus AI game, you can assign the AI a color and you will be able to play by entering in moves as row-column numbers (the upper corner is 1,1. The lower corner is 8,8). Over the course of this game a CSV will be outputted with the times of each turn taken by the AI. The user is not tracked.

After the game finishes the program will exit.

Code with internal documentation

standard.h

#ifndef standard\_h

#define standard\_h

/\*seems like a good way to do this... might want to do in the future.

All includes for the program, and the namespace.

There is possibly an argument for reducing includes and removing the namespace.

This can happen in future iterations of the program

Possibly a few too many includes, and the namespace has to go.

\*/

#include <iostream>

#include <cstdlib>

#include <string>

#include <fstream>

#include <vector>

#include <utility>

#include <algorithm>

#include <chrono>

#include <random>

#include <ratio>

#include <thread>

using namespace std;

using std::chrono::duration;

using std::chrono::duration\_cast;

using std::chrono::high\_resolution\_clock;

using std::milli;

#endif // standard\_h

const.h

#ifndef const\_h

#define const\_h

/\*varied constants for use in Konane\*/

static const int MAX = 999;

static const int MIN = -999;

static const int LOSE = -300;

static const int WIN = 300;

static const int B = 2;

static const int W = 1;

static const int GAME\_OVER\_WHITE= 600;

static const int GAME\_OVER\_BLACK = -600;

/\*\* CHANGE \DEPTH AS NEEDED/DESIRED \*\*/

/\*\* DEPTH MUST BE EVEN, AS PER CONVENTION \*\*/

/\*\*CONFIRMED \EFFECTIVE AT 12, AT 16 \PROBLEMATIC\*\*/

static const int depth = 14;

#endif

Konane-parallel.cpp

/\*Order of Game Play\*/

/\*Game is lost when there are no longer any valid moves.\*/

/\* 1. determine who plays Black and is thus first.

\* 2. choose to remove 2 pieces if not first (another variant has the removal of one piece by each player).

\*

\* \*\*GAME LOOP\*\* \*

\* 3. wait for input or make input

\* 4. upon AI move:

\* run legalmove on each position in array when is turn (and board is accurate.)

\* N,S,E,W determines legalmove direction.

\*

\* the following bounds due to no move possible:

\* if i<2 (row) don't run N

\* if i>5 don't run S,

\*

\* if j<2 (column) don't run W,

\* if j>5 don't run E.

\*

\* so the first square in a column doesn't evaluate the Northern most move and the last square will not

\* evaluate the Southernmost... etc

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* RECURSIVE CALLS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 5. if legalmove returns true, form node (recursive call with new board) with move and repeat process board to obtain number.

\* it is a DFS.

\* 6. the calls eventually reach depth d and the SEF is run. It is notable that the alpha and beta cutoffs will reduce the number of boards evaluated

\* WITHOUT any data loss.

\* (this is a less intuitive action of the recursive function, no data is lost, but branches that the AI will NEVER select are not explored further.

\* essentially, a node which comes out too high or too low relative to the node examined will not be selected)

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\* State Evaluation function (SEF)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* 7. The SEF evaluates partly by counting legal moves for player in that state.

\* Additionally, it counts pieces and estimates based on expected piece balance (each move must remove one piece, if more pieces are missing a

\* double move has occurred. The SEF assumes that this is bad for the player who had this happen to it.)

\*

\* The SEF value is passed up to the top of the tree with varied modifications, minimizing levels take the lowest value to pass, maximizing

\* will take up the largest values

\*

\* Dir can = 0 for N (up), 1 for E(right), 2 for S (down), 3 for W (left)

\*/

/\* STANDARD INCLUDES FROM LIBRARY FILES AND NAMESPACE \*/

#include "standard.h"

/\*Include Constants, depth is defined at compile time, as are some\*/

#include "const.h"

/\*\*REMEMBER TO USE THE DIAGNOSTIC PRINTOUTS FOR FUTURE CHANGES\*\*/

/\*\*FUNCTIONAL WITH 16 DEPTH, EASE OF TESTING AT 14 OR 12 DEPTH, GAMEPLAY IS DIFFERENT DEPENDING ON DEPTH\*\*/

/\*\*color is AI, humanColor is opponent\*\*/

/\*\*\*MUST HAVE THESE GLOBALS, COULD PERHAPS EDIT INTO MAIN?\*\*\*/

int AIcolor, humanColor;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*DUE TO FOLLOWING OFFICIAL RULES, COLOR HAS BEEN MERGED WITH FIRST\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*(setters and getters, removed from code at this time)\*\*\*/

/\*int getFirst(){return first;}

\*int getColor() {return AIcolor;}

\*int getPlayerColor() {return playerColor;}

\*/

class player{

public:

void jokes();

void detectWL(int value);

};

void player::jokes()

{

int jokes;

std::cout << "Would you like to play a game? \n";

std::cout << "1. Konane \n2. ALSO Konane\n3. Global Thermonuclear War\n";

std::cin >> jokes;

if (jokes == 3) {

std::cout << "That game is SOOOOOOOO boring,let's play Konane!!!\n";

}

}

void player::detectWL(int value){}

class board {

public:

//records best value, used for parallel root

int bval;

//2D array, Black and white pieces value B= 2 and W= 1 respectively

//empty spaces will be treated as zeros.

int objBoard[8][8] = {

{B,W,B,W,B,W,B,W},

{W,B,W,B,W,B,W,B},

{B,W,B,W,B,W,B,W},

{W,B,W,B,W,B,W,B},

{B,W,B,W,B,W,B,W},

{W,B,W,B,W,B,W,B},

{B,W,B,W,B,W,B,W},

{W,B,W,B,W,B,W,B}

};

//is always maximizing

//sets zero player game

bool zpgame;

bool setZpgame();

//Sets both color and first player.

bool setColor();

//returns first and color

bool guardRails();

//returns first and color

void setValues(int b, int w,int x,int dir);

//displays board

void display();

//piece removal, for fixing board errors and for setup

void manualOverride();

//returns True if a space is full, False if it is not.

bool isFull(int i, int j);

//test for legality of move

bool legal\_move(int currentmove[], int i, int j, int Dir);

//tests for legality of move inside the SEF (uses different logic) Uses isFull to do checks

bool legal\_SEF(int i, int j, int Dir);

/\*

\*makes a move at each node and also makes the chosen move in the game.

\*currentmove[] is passed along with player color to make the move.

\*makeMove MUST receive correct input, as it assumes such

\*this is handled by other functions in the case of a user.

\*/

void makeMove(int currentmove[], int playerColor);

//unmakes a move after a move is chosen.

void unmakeMove(int currentmove[], int playerColor, int opColor);

//serial variation of alphaBetaMinimax Check design document for more information

//level is the level it is at, depth is the level to which it will travel

//levelColor is the color being examined at this level

//maximizing is whether or not it is maximizing (true = max, false = min)

int alphaBetaMinimax(int alpha, int beta, int level, int depth, int levelColor, bool maximizing);

//State Evaluation Function. Does not work alone, uses legal\_SEF pass is for future modifications

int SEF(int pass);

//Checks if chosen move by player is on the board.

void selection(int S);

//Displays move numerically

void displayMove();

//Compares 8 boards at root of tree (needed, serial implementation) returns board with highest bvalue (bval) after each board is done

board comparison (board board1, board board2, board board3, board board4, board board5, board board6, board board7, board board8);

//Called on each thread, imitates the maximizing node of the tree.

void threadKonane(int i,int alpha, int beta, int depth, int levelColor);

//Calibrates the for loop that loops through the board for moves. Checks moves that player

//or opponent will make based on the node.

int calibrate(int i, int playerColor);

//RECORDS MOVES BESTMOVE HOLDS FIRST MOVE AND THEN ANY MOVE THAT IS BETTER.

int bestmove[4];

int playermove[4];

};

void board::threadKonane(int i,int alpha, int beta, int depth, int levelColor){

int k, m, temp;

int opColor = levelColor%2;

opColor++;

/\*\*RESOLVE MATH AFTER SUCCESSFUL TEST\*\*/

int currentmove[4];

int val = MIN;

bval= MIN;

for (int j = calibrate(i, levelColor); j < 8; j = j + 2)

{

if (objBoard[i][j] > 0)

{

for (int Dir = 0; Dir < 4; Dir++) {

if (legal\_move(currentmove, i, j, Dir)) {

/\*\*\*\*\*MOVE PIECE\*\*\*\*\*/

makeMove(currentmove, levelColor);

//calls for minimization using the opposing color

temp = val;

//needs some work.

//calling serial variation

val = max(val, alphaBetaMinimax(alpha, beta, 1, depth, opColor, false));

unmakeMove(currentmove, levelColor, opColor);

/\*\*AT ROOT OF RECURSIVE TREE, IS CURRENT MOVE BEST?\*\*/

/\*\*IF CURRENT MOVE BEST, CHANGE MOVE\*\*/

//This normally happens at depth zero

//This IS "depth zero" as far as the code

//is concerned, hence this can happen here.

//could remove from other code.

alpha = max(alpha, val);

if (temp < val) {

bval=val;

//the best move is the highest scoring move selected

bestmove[0] = currentmove[0];

bestmove[1] = currentmove[1];

bestmove[2] = currentmove[2];

bestmove[3] = currentmove[3];

}

}

}

}

}

//cout<<"SEF of thread "<<i<<" = "<<bval<<endl;

return;

}

/\*\*\*ALPHABETA MINIMAX BELOW\*\*\*/

/\*\*\*NEEDS FULL REFACTOR FOR THE PURPOSE OF ZERO PLAYER GAME\*\*\*/

//for clarifications, refer to user doc.

int board::alphaBetaMinimax(int alpha, int beta, int level, int depth, int levelColor, bool maximizing)

{

int k, m, tshoot, temp;

/\* Initial values of

\* Alpha and Beta

\* Terminating condition. i.e

\*leaf node, is reached\*/

int opColor = levelColor%2;

opColor++;

int currentmove[4];

if (level == depth) {

tshoot = SEF(levelColor);

//std::cout << "\nSEF= " << tshoot << "\n";

return tshoot;

}

else if (maximizing == true)

{

int val = MIN;

for (int i = 0; i < 8; i++)

{

for (int j = calibrate(i, levelColor); j < 8; j = j + 2)

{

if (objBoard[i][j] > 0)

{

for (int Dir = 0; Dir < 4; Dir++) {

if (legal\_move(currentmove, i, j, Dir)) {

/\*\*\*\*\*MOVE PIECE\*\*\*\*\*/

makeMove(currentmove, levelColor);

//calls for minimization using the opposing color

temp = val;

val = std::max(val, alphaBetaMinimax(alpha, beta, level + 1, depth, opColor, false));

unmakeMove(currentmove, levelColor, opColor);

/\*\*AT ROOT OF RECURSIVE TREE, IS CURRENT MOVE BEST?\*\*/

/\*\*IF CURRENT MOVE BEST, CHANGE MOVE\*\*/

if (level == 0 && temp < val) {

bestmove[0] = currentmove[0];

bestmove[1] = currentmove[1];

bestmove[2] = currentmove[2];

bestmove[3] = currentmove[3];

}

alpha = std::max(alpha, val);

//Alpha cutoff

if (beta <= alpha) {

return alpha;

}

}

}

}

}

}

/\*\*Program can predict loss, and will end properly if loss.\*\*/

if (val == MIN)

{

return LOSE;

}

return val;

}

else

{

//std::cout << "minimizing \n";

int val = MAX;

for (int i = 0; i < 8; i++)

{

for (int j = calibrate(i, levelColor); j < 8; j = j + 2)

{

if (objBoard[i][j] > 0)

{

for (int Dir = 0; Dir < 4; Dir++) {

if (legal\_move(currentmove, i, j, Dir)) {

/\* std::cout << "\n ";

for (int z = 0; z < 4; z++) {

std::cout << currentmove[z] + 1;

}

std::cout << "\n ";

\*/

makeMove(currentmove, levelColor);

val = std::min(val, alphaBetaMinimax(alpha, beta, level + 1, depth, opColor, true));

beta = std::min(beta, val);

unmakeMove(currentmove, levelColor, opColor);

// Alpha Beta Pruning

if (beta <= alpha) {

//Beta cutoff

return beta;

}

}

}

}

}

}

if (val == MAX)

{

/\*ran out of moves on mimimzer

\*\*The SEF needs tuning, but MAY work here

\*\*(some assumptions have been made here)

\*\*once tuned, it should perform ideally\*/

return WIN;

}

return val;

}

std::cout << "ERROR";

return -1;

}

// returns best board, to be set as correct board. Resource intensive. Has to be a better way.

board board::comparison (board board1, board board2, board board3, board board4, board board5, board board6, board board7, board board8)

{

if (board1.bval<board2.bval)

{

cout<<board1.bval<<endl;

board1=board2;

}

if (board1.bval<board3.bval)

{

cout<<board1.bval<<endl;

board1=board3;

}

if (board1.bval<board4.bval)

{

cout<<board1.bval<<endl;

board1=board4;

}

if (board1.bval<board5.bval)

{

cout<<board1.bval<<endl;

board1=board5;

}

if (board1.bval<board6.bval)

{

cout<<board1.bval<<endl;

board1=board6;

}

if (board1.bval<board7.bval)

{ cout<<board1.bval<<endl;

board1=board7;

}

if (board1.bval<board8.bval)

{

cout<<board1.bval<<endl;

board1=board8;

cout<<board1.bval<<endl;

}

return board1;

}

//for for loops, to allow for checking less squares and to ensure the right squares on

//a checkerboard are checked.

int board::calibrate(int i, int playerColor)

{

if (playerColor == 2)

{return i % 2;}

//starting index of an even i will be 0

else

//starting index of an even i will be 1

{return (i + 1) % 2;}

}

/\*\*ZERO PLAYER GAME SELECTION\*\*/

//Sets up a zero player game for the AI

bool board::setZpgame(){

char ans;

do{

std::cout<<"Will I be playing Myself?\ny/n\n";

std::cin>>ans;

if (ans=='y'){

return true;

}

else if(ans == 'n'){

return false;

}

else{

std::cout<<"choose only y or n please";

}

}while (true);

}

/\*\*SAFETY FOR USER ERRORS\*\*/

bool board::guardRails(){

bool correct=false;

int i,j,k,m;

i=bestmove[0];

j=bestmove[1];

k=bestmove[2];

m=bestmove[3];

//check for horizontal movement

if(i==k||j==m){

//check for even number of squares traversed, if zero, then move is probably legal.

i=(abs(i-k))%2;

j=(abs(j-m))%2;

if (i==0&&j==0)

{

return true;

}

}

std::cout<<"incorrect move";

return false;

}

/\*\*\*ASSISTS MANUAL OVVERIDE\*\*\*/

void board::selection(int S) {

int i, j;

std::cout << "\n column (A-H = 1-8): ";

std::cin >> j;

std::cout << "\n row (1-8): ";

std::cin >> i;

i--;

j--;

if (i > 7 || i < 0 || j>7 || j < 0) {

std::cout << "\n invalid choice, restarting selection";

}

else {

objBoard[i][j] = S;

}

return;

}

/\*\*\*FOR SETUP, TESTING, AND MANUAL BOARD CORRECTIONS IF ERRORS MADE,\*\*\*

\*\*\*ASSUMES SOME INTELLIGENCE ON PART OF USER\*\*\*/

/\*\*\*ZERO PLAYER GAME COMPATIBLE\*\*\*/

void board::manualOverride() {

int p = 0;

int i, j;

char remadd, ans;

while (ans != 'y') {

display();

std::cout<< "\n\nX= Black, O = White, Spaces for empty squares\n\n";

std::cout << "r = Removal of piece, b = add black, w = add white e = exit \n";

std::cin >> remadd;

while (remadd != 'e') {

if (remadd == 'e'){}

else if (remadd == 'r')

{

std::cout << "What pieces will be removed? choose row and column for each. (NO adjacency test)\n";

selection(0);

}

else if (remadd == 'b')

{

std::cout << "What black pieces will be added? choose row and column for each. (NO adjacency test)\n";

selection(2);

}

else if (remadd == 'w') {

std::cout << "What white pieces will be added? choose row and column for each. (NO adjacency test)\n";

selection(1);

}

else{

std::cout <<"\n invalid choice, restarting selection\n";

display();

std::cout<< "\n\nX= Black, O = White, Spaces for empty squares\n\n";

std::cout << "r = Removal of piece, b = add black, w = add white e = exit \n";

std::cin >> remadd;

}

display();

std::cout << "r = Removal of piece, b = add black, w = add white e = exit \n";

std::cin >> remadd;

}

display();

std::cout << "\n is board correct? y for yes, n for no \n \n";

std::cin >> ans;

}

}

/\*\*\* CRITICAL FOR ENTIRE GAME, SET COLOR AND FIRST GLOBALS HERE \*\*\*/

/\*\*\*GIVEN CURRENT USAGE, IS NOT INCOMPATIBLE WITH ZERO PLAYER GAME\*\*\*/

bool board::setColor() {

char ans='n';

bool first;

while (ans != 'y') {

std::cout << "\nWhat color should 'I' play as? \n 1 for white and 2 for black:\n(black goes first per Konane rules \n black is X, white is O)\n ";

std::cin >> AIcolor;

if (AIcolor == 2)

{

humanColor = 1;

first = true;

std::cout<< "\nI play as black and go first.";

}

else if (AIcolor == 1)

{

humanColor = 2;

first = false;

std::cout<< "\nI play as white and go second.";

}

else {

std::cout << "\nerror, restarting";

}

std::cout << "\nIs this correct? y for yes:\n";

std::cin >> ans;

}

return first;

}

/\*\*\*SPACE TEST FOR PRESENCE OF PIECE\*\*\*/

/\*\*\* COLOR AGNOSTIC \*\*\*/

bool board::isFull(int i, int j) {

if (objBoard[i][j] == 0){

return false;

}

else {

return true;

}

}

/\*\*\*ASSISTS WITH CALIBRATION OF SQUARES, DEALS WITH DOUBLE WHITE/BLACK BOARD EDGE ISSUE\*\*\*/

/\*\*\*FAIRLY SIMPLE DISPLAY FUNCTION\*\*\*/

void board::display() {

int row = 0;

std::cout << " A B C D E F G H \n";

std::cout << " 1 2 3 4 5 6 7 8 ";

std::cout<< "\n \n";

for (int i = 0; i < 8; i++) {

std::cout << ++row <<" ";

for (int j = 0; j < 8; j++) {

if (objBoard[i][j] > 1) {

std::cout << "X ";

}

else if (objBoard[i][j] > 0) {

std::cout << "O ";

}

else {

std::cout << " ";

}

}

std::cout << "\n";

}

}

//displays move as a numerical value corresponding to the numerical indexes on the board.

void board::displayMove(){

std::cout<<"\n"<<bestmove[0]+1<<", ";

std::cout<<bestmove[1]+1<<", to ";

std::cout<<bestmove[2]+1<<", ";

std::cout<<bestmove[3]+1<<", \n your move\n";

}

/\*\*\* TESTS LEGALITY OF MOVE FOR SEF \*\*\*/

/\*\*\*State Evaluation helper, a form of Legal Move.\*\*\*

\*\*\* Exists for the purposes of tallying possible \*\*\*

\*\*\* moves at depth of tree \*\*\*/

/\*\*CLUMSY, REFACTOR\*\*/

bool board::legal\_SEF(int i, int j, int Dir)

{

int k = i;

int m = j;

bool state;

if (Dir == 0) //N

{

if (i > 1) {

k--;

state = isFull(k, m);

if (state == true) {

k--;

state = isFull(k, m);

if (state == false) {

return true;

}

}

}

}

if (Dir == 1) //S

if (i < 5) {

{

k++;

state = isFull(k, m);

if (state == true) {

k++;

state = isFull(k, m);

if (state == false) {

return true;

}

}

}

}

if (Dir == 2) //W

{

if (j > 1) {

m--;

state = isFull(k, m);

if (state == true) {

m--;

state = isFull(k, m);

if (state == false) {

return true;

}

}

}

}

if (Dir == 3) //E

{

m++;

state = isFull(k, m);

if (state == true) {

m++;

state = isFull(k, m);

if (state == false) {

return true;

}

}

}

return false;

}

/\*\*\*ACCOMODATING NEW COORDINATE SYSTEM ANNOUNCED

\*\*\*MESSY, REFACTOR IF TIME, IT IS COLOR AGNOSTIC

\*\*Takes advantage of array being altered and then

\*\*passed out of function

\*\*i, j = source co-ords. k,m = destination coords

\*\*/

bool board::legal\_move(int currentmove[], int i, int j, int Dir)

{

bool state;

int k = i;

int m = j;

if (Dir == 0) //N

{

if (k > 1) {

k--;

//looks for single jump, calls isFull to check if that space is empty or full.

state = isFull(k, m);

if (state == true) {

k--;

state = isFull(k, m);

if (state == false) {

currentmove[0] = i;

currentmove[1] = j;

currentmove[2] = k;

currentmove[3] = m;

if(k>1){

k--;

//looks for double jump

state = isFull(k, m);

if (state == true) {

k--;

state = isFull(k, m);

if (state == false) {

currentmove[2] = k;

currentmove[3] = m;

}

}

}

return true;

}

}

}

}

if (Dir == 1) //S

{

if (k < 6) {

k++;

state = isFull(k, m);

if (state == true) {

k++;

state = isFull(k, m);

if (state == false) {

currentmove[0] = i;

currentmove[1] = j;

currentmove[2] = k;

currentmove[3] = m;

if(k<6){

k++;

state = isFull(k, m);

if (state == true) {

k++;

state = isFull(k, m);

if (state == false) {

currentmove[2] = k;

currentmove[3] = m;

}

}

}

return true;

}

}

}

}

if (Dir == 2) //W

{

if (m > 1) {

m--;

state = isFull(k, m);

if (state == true) {

m--;

state = isFull(k, m);

if (state == false) {

currentmove[0] = i;

currentmove[1] = j;

currentmove[2] = k;

currentmove[3] = m;

if(m>1){

m--;

state = isFull(k, m);

if (state == true) {

m--;

state = isFull(k, m);

if (state == false) {

currentmove[2] = k;

currentmove[3] = m;

}

}

}

return true;

}

}

}

}

if (Dir == 3) //E

{

if (m <6) {

m++;

state = isFull(k, m);

if (state == true) {

m++;

state = isFull(k, m);

if (state == false) {

currentmove[0] = i;

currentmove[1] = j;

currentmove[2] = k;

currentmove[3] = m;

if(m<6){

m++;

state = isFull(k, m);

if (state == true) {

m++;

state = isFull(k, m);

if (state == false) {

currentmove[2] = k;

currentmove[3] = m;

}

}

}

return true;

}

}

}

}

//if no move at all

return false;

}

/\*\*\*ASSUMES LEGAL MOVE HAS BEEN EXECUTED,\*\*\*

/\*\*\*MAKES MOVE RECORDED BY LEGAL MOVE \*\*\*/

/\*\*

\*i,j is indexes of source of piece. k, m is destination (i,j coord)

\*currentmove[] is the move being investigated by the Minimax algorithim.

\*playerColor is the color of the moving piece.

\*\*/

void board::makeMove(int currentmove[], int playerColor)

//takes

{ //std::cout<<"making move\n";

int i,j,k,m;

bool neq;

i = currentmove[0];

j = currentmove[1];

k = currentmove[2];

m = currentmove[3];

//std::cout<<" "<<i+1<<", "<<j+1<<", "<<k+1<<", "<<m+1<<"\n";

neq = true;

do {

objBoard[i][j] = 0;

if (k < i)

{

objBoard[i - 1][j] = 0;

objBoard[i - 2][j] = playerColor;

i=i-2;

}

if (k > i)

{

objBoard[i + 1][j] = 0;

objBoard[i + 2][j] = playerColor;

i=i+2;

}

if (m < j)

{

objBoard[i][j - 1] = 0;

objBoard[i][j - 2] = playerColor;

j=j-2;

}

if (m > j)

{

objBoard[i][j + 1] = 0;

objBoard[i][j + 2] = playerColor;

j=j+2;

}

if((i==k)&&(m==j)){neq=false;}

} while (neq);

return;

}

/\*\*UNMAKE MOVE IS COLOR-AGNOSTIC, ZERO PLAYER GAME WILL NOT REQUIRE A CHANGE HERE\*\*/

//Same effect as make move, only in reverse.

void board::unmakeMove(int currentmove[],int playerColor, int opColor) {

int i,j,k,m;

bool neq=true;

i = currentmove[0];

j = currentmove[1];

k = currentmove[2];

m = currentmove[3];

while (neq){

//std::cout << "\nunmaking move: ";

objBoard[k][m] = 0;

if (k > i)

{

objBoard[k - 1][m] = opColor;

objBoard[k - 2][m] = playerColor;

k=k-2;

if(k==i)

{

break;

}

}

if (k < i)

{

objBoard[k + 1][m] = opColor;

objBoard[k + 2][m] = playerColor;

k=k+2;

if(k==i)

{

break;

}

}

if (m > j)

{

objBoard[k][m-1] = opColor;

objBoard[k][m-2] = playerColor;

m=m-2;

if(m==j)

{

break;

}

}

if (m < j)

{

objBoard[k][m+1] = opColor;

objBoard[k][m+2] = playerColor;

m=m+2;

if(m==j)

{

break;

}

}

}

}

/\*\*DEBRIS COMMENTS, NEED TO RECONCILE WITH CURRENT VERSION\*\*/

/\*AI Game Playing Functions, in a loop every turn

void ABmax(); //maximizer for AI sim, may want it to be separate from this class?

void ABmin(); //minimizer for AI sim, may want it to be separate from this class?

Make a move in-game

void chosenMove(); //states move chosen, implements... probably unneccessary\*/

/\*

The minimax algorithm begins at the current state of the game recursively searching the

game tree for the optimal move depth-first, which means exploring a branch as far as it goes and

then backtracking until a path that has not been explored is found. Once the algorithm reaches a

leaf node, which indicates a terminal state, or reaches the specified search depth in the game tree

the utility(best/worst) value of the node is returned.

\*/

int board::SEF(int playerColor) {

int sum=0; //sum of total moves for SEF player

int eb=0; //deprecated but potentially useful

int pb=0; //Player pieces

int oPiece=0; //Opponent pieces

//pb=30;

//eb=(pb-(turn+pass));

//ABOVE NEEDS SOME WORK

int opColor = playerColor%2;

opColor++;

/\*\*PIECE BALANCE ADJUSTMENT MAy BE NEEDED BASED ON WHO IS FIRST IF PLAYER IS SECOND, INCREMENT PIECES\*\*/

for (int i = 0; i < 8; i++)

{

for (int j = calibrate(i, playerColor); j < 8; j = j + 2)

{ //If there is a AI piece on the board, increment pb

if (objBoard[i][j] > 0)

{

pb++;

//check moves for AI, increment sum if legal\_SEF returns true

for (int Dir = 0; Dir < 4; Dir++) {

if (legal\_SEF(i, j, Dir)) {

sum++;

}

}

}

}

}

for (int i = 0; i < 8; i++)

{

for (int j = calibrate(i, opColor); j < 8; j = j + 2)

{ //If there is a Player piece on the board here, decrement pb

if (objBoard[i][j] > 0)

{

oPiece++;

//check moves for opponent, decrement sum if legal\_SEF returns true

for (int Dir = 0; Dir < 4; Dir++) {

if (legal\_SEF(i, j, Dir)) {

sum--;

}

}

}

}

}

//std::cout<<"SEF EXECUTED ON BELOW BOARD\n";

//display();

return (sum+(pb-oPiece));

}

// std::pair

void write\_csv(double input[], int totalMoves){

// Make a CSV file with one or more columns of integer values

// Each column of data is represented by the pair <column name, column data>

// The dataset is represented as a vector of these columns

// Create an output filestream object

std::ofstream csv("parallel-moves-14-1.csv");

csv<<"Turn, Time \n";

for(int i=0;i<totalMoves;i++){

csv<<i+1<<","<<input[i]<<"\n";

}

// Close the file

csv.close();

}

/\*\*TODO:: TOO MUCH IN MAIN!! REFACTOR!!\*\*/

int main() {

int Dir = 0;

int level=0;

int turn=0;

int i,j,k,m,z, alpha, beta, state;

double tc;

//for recording time

double csv[50];

duration<double> timeCount;

char correct, ans;

//who is first, whose turn is it?

bool first, AIturn, right;

//

bool cor=true;

player player;

board board, board1, board2, board3, board4, board5, board6, board7, board8;

/\*\*odd numbered depths end on min nodes, even on max. This doesn't work. This is to check for that\*\*/

std::cout<<"depth = "<< depth<<". (This should be an even number)\n";

std::cout<<"\n";

player.jokes();

//Zero Player Game?

right=board.setZpgame();

if (right){

board.zpgame=true;

}

//Who is black, goes first?

first=board.setColor();

if (first == true) {

AIturn = true;

}

else{

AIturn=false;

}

/\*\*SETS BOARD UP\*\*/

board.manualOverride();

/\*\*SETS UP BOARDS FOR THREADS\*\*/

board1=board;

board2=board;

board3=board;

board4=board;

board5=board;

board6=board;

board7=board;

board8=board;

//Starts the clock.

high\_resolution\_clock::time\_point t1 = high\_resolution\_clock::now();

while (true) {

//if all moves lose.

if(board.bval<-500){

std::cout<<"Game over, White loses";

break;

}

if (AIturn) {

/\*\*This should be refactored into a function.

\*This initializes every board object to the same state, starts the clock on the move and

\*makes the \*\*/

high\_resolution\_clock::time\_point tpre = high\_resolution\_clock::now();

board1=board;

board2=board;

board3=board;

board4=board;

board5=board;

board6=board;

board7=board;

board8=board;

//Initiates threads, calls them on the rows of the board

//Alpha and Beta NOT set to Min and Max at first call. This has improved efficiency and reduced error.

//They ARE set to MAX and MIN later.

thread th1( &board::threadKonane, &board1, 0, 1,2,depth,AIcolor);

thread th2( &board::threadKonane, &board2, 1, 1,2,depth,AIcolor);

thread th3( &board::threadKonane, &board3, 2, 1,2,depth,AIcolor);

thread th4( &board::threadKonane, &board4, 3, 1,2,depth,AIcolor);

thread th5( &board::threadKonane, &board5, 4, 1,2,depth,AIcolor);

thread th6( &board::threadKonane, &board6, 5, 1,2,depth,AIcolor);

thread th7( &board::threadKonane, &board7, 6, 1,2,depth,AIcolor);

thread th8( &board::threadKonane, &board8, 7, 1,2,depth,AIcolor);

//Joins threads to main program after program run (some efficiencies to realize here).

th1.join();

th2.join();

th3.join();

th4.join();

th5.join();

th6.join();

th7.join();

th8.join();

//makes a comparison between the 8 boards and selects the ideal board from them

board = board.comparison(board1, board2, board3, board4, board5, board6, board7, board8);

//displays prior board

board.display();

//if no valid moves

if(board.bval<-500){

cout<<"Game Over, Black Loses";

break;

}

//Uses the "bestmove" array to make the move chosen by algorithim

board.makeMove(board.bestmove, AIcolor);

//displays the move in start i, start j, destination i, destination j, format.

board.displayMove();

//displays the board

board.display();

//Passes turn

AIturn = false;

//increments turn counter

turn++;

//stops clock and counts value

high\_resolution\_clock::time\_point tpost = high\_resolution\_clock::now();

timeCount = duration\_cast<duration<double>>(tpost-tpre);

tc=timeCount.count();

//gives amount of time taken

std::cout<<"\n"<<tc<<"\n";

//records time and turn to array

csv[turn-1]=tc;

}

/\*\* if PLAYING AGAINST ITSELF OR A SIMILAR AI W A DIFFERENT SEF\*\*/

//same as above

if((right) && (AIturn==false)){

high\_resolution\_clock::time\_point tpre = high\_resolution\_clock::now();

board1=board;

board2=board;

board3=board;

board4=board;

board5=board;

board6=board;

board7=board;

board8=board;

thread th1( &board::threadKonane, &board1, 0, 1,2,depth,humanColor);

thread th2( &board::threadKonane, &board2, 1, 1,2,depth,humanColor);

thread th3( &board::threadKonane, &board3, 2, 1,2,depth,humanColor);

thread th4( &board::threadKonane, &board4, 3, 1,2,depth,humanColor);

thread th5( &board::threadKonane, &board5, 4, 1,2,depth,humanColor);

thread th6( &board::threadKonane, &board6, 5, 1,2,depth,humanColor);

thread th7( &board::threadKonane, &board7, 6, 1,2,depth,humanColor);

thread th8( &board::threadKonane, &board8, 7, 1,2,depth,humanColor);

th1.join();

th2.join();

th3.join();

th4.join();

th5.join();

th6.join();

th7.join();

th8.join();

board = board.comparison (board1, board2, board3, board4, board5, board6, board7, board8);

board.display();

//if no valid moves

if(board.bval<-500){

std::cout<<"Game over, White loses";

break;

}

board.makeMove(board.bestmove, humanColor);

board.displayMove();

board.display();

AIturn = true;

turn++;

//record timing.

high\_resolution\_clock::time\_point tpost = high\_resolution\_clock::now();

timeCount = duration\_cast<duration<double>>(tpost-tpre);

tc=timeCount.count();

std::cout<<"\n"<<tc<<"\n";

csv[turn-1]=tc;

}

/\*\* if PLAYING A HUMAN\*\*/

if((!right) && (AIturn==false)) {

do{

std::cout << "\nenter piece to move column, 1-8: ";

//source column

std::cin >> j;

//accomodate indexing

j--;

//enters player move into array

board.bestmove[1]=j;

std::cout << "\nenter piece to move row, 1-8: ";

std::cin >> i;

i--;

//source row

board.bestmove[0]=i;

std::cout << "\nenter piece destination column, 1-8: ";

std::cin >> m;

m--;

//destination column

board.bestmove[3]=m;

std::cout << "\nenter piece destination row, 1-8: ";

std::cin >> k;

k--;

//destination row

board.bestmove[2]=k;

}while(! cor);

//is this move legal?

cor=board.guardRails();

board.makeMove(board.bestmove,humanColor);

board.displayMove();

board.display();

turn++;

AIturn=true;

}

std::cout<< "\nassuming correct board and continue\n";

}

high\_resolution\_clock::time\_point t2 = high\_resolution\_clock::now();

duration<double> time\_span = duration\_cast<duration<double>>(t2 - t1);

std::cout<<"finished in "<<turn<<" turns";

std::cout << "It took me " << time\_span.count() << " seconds.\n";

for(z=0;z<turn;z++)

{std::cout<<"turn " <<z+1<< " took "<<csv[z]<<" seconds \n";}

//write\_csv(csv, turn);

return 0;

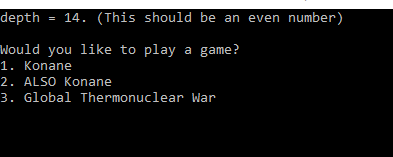
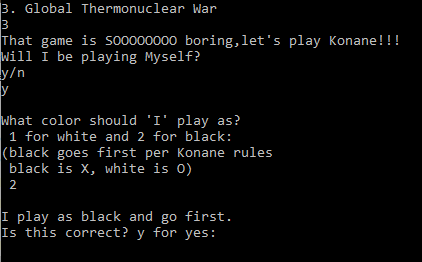
}

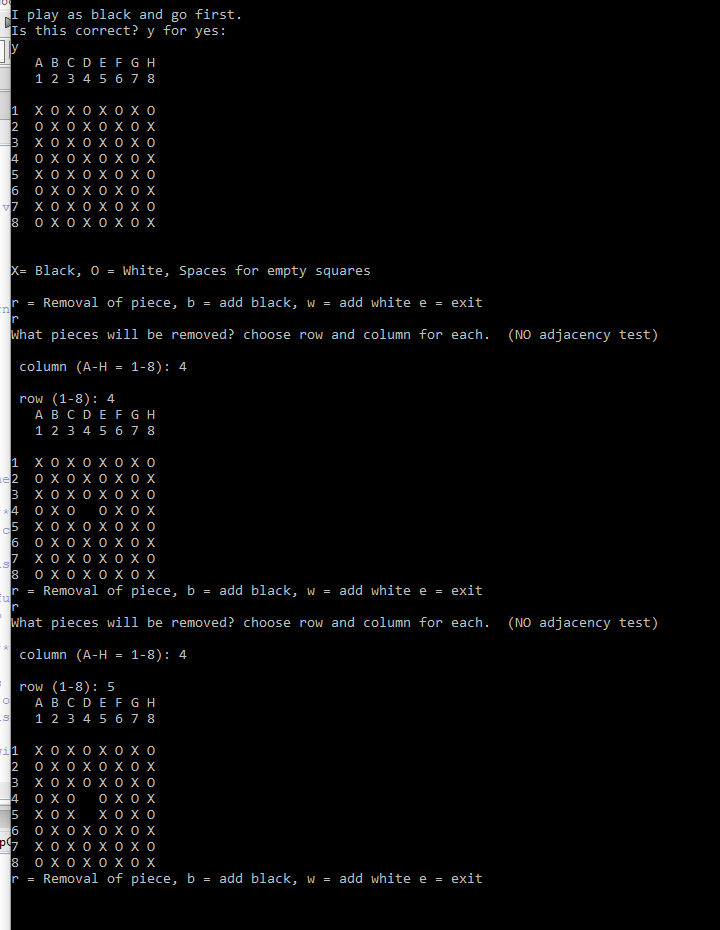
Testing doc

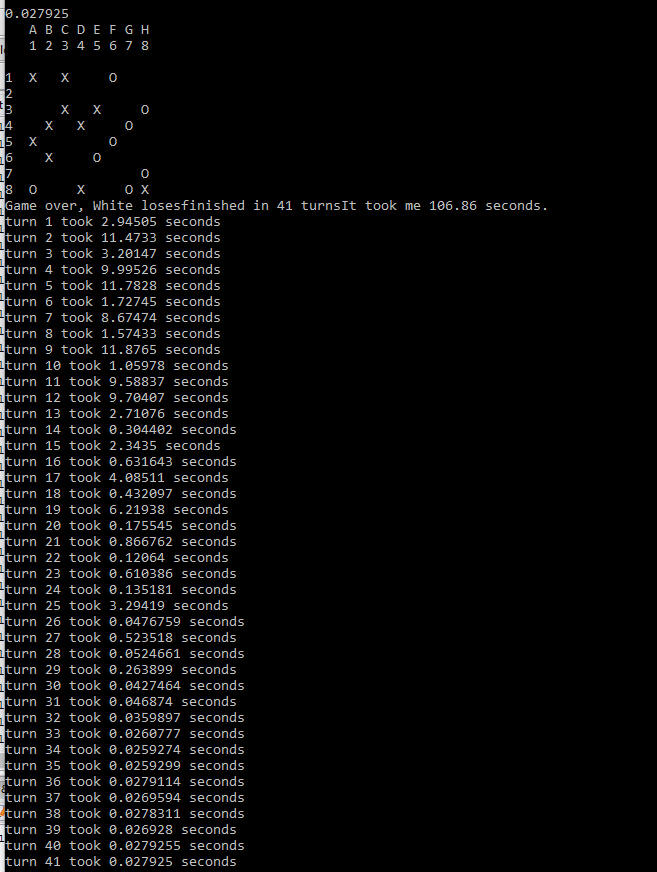
Several tests were performed. These are the Excel comparisons of time taken by the serial and the parallel algorithms. There was not much of a net increase for the Parallel algorithm, but it may have been due to hardware deficiencies.

The graphs included demonstrate that the parallel algorithm leads to minimal payoffs in time taken. This is very likely due to cutoffs happening at the highest level of the tree via the serial algorithm which are not possible with the parallel algorithm, though the AI takes 4 more moves, the mean move value does not yield useful information.

The following are screenshots of testing.





At this point it outputs to a CSV, using Excel on both variations of the program, the graph shown below was made as a comparison of performance.