California State University, Fresno

Computer Science Department

CSCI 264

Artificial Intelligence

Project Documentation on Connect Four

To

Prof. David Ruby

Submitted by

Leeladhar Reddy Munnangi 109510225

Yasasvi Yeleswarapu 109475008

Sri Satya Sai Kamal Atluri 109479597

Contents

Introduction

Connect Four (also known as Captain's Mistress, Four Up, Plot Four, Find Four, Fourplay, Four in a Row, Four in a Line and Gravitrips (in Soviet Union)) is a two-player connection game in which the players first choose a color and then take turns dropping colored discs from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the next available space within the column. The objective of the game is to connect four of one's own discs of the same color next to each other vertically, horizontally, or diagonally before your opponent. Connect Four is a strongly solved game. The first player can always win by playing the right moves.

Procedure

Connect Four is a two-player game with "perfect information". This term describes games where one player at a time plays, players have all the information about moves that have taken place, and all moves that can take place, for a given game state. Connect Four also belongs to the classification of an adversarial, zero-sum game, since a player's advantage is an opponent's disadvantage.

One measure of complexity of the Connect Four game is the number of possible games board positions. For classic Connect Four played on 6 high, 7 wide grid, there are 4,531,985,219,092 positions[3] for all game boards populated with 0 to 42 pieces.

The game was first solved by James Dow Allen (October 1, 1988), and independently by Victor Allis (October 16, 1988). Allis describes a knowledge based approach, with nine strategies, as a solution for Connect Four. Allen also describes winning strategies in his analysis of the game. At the time of the initial solutions for Connect Four, brute force analysis was not deemed feasible given the game's complexity and the computer technology available at the time.

Connect Four has since been solved with brute force methods beginning with John Tromp's work in compiling an 8-ply databases (Feb 4, 1995). The artificial intelligence algorithms able to strongly solve Connect Four are minimax or negamax, with optimizations that include alpha-beta pruning, dynamic history ordering of game player moves, andtransposition tables. The code for solving Connect Four with these methods is also the basis for the Fhourstones integer performance benchmark.

The solved conclusion for Connect Four is first player win. With perfect play, the first player can force a win, on or before the 41st move by starting in the middle column. The game is a theoretical draw when the first player starts in the columns adjacent to the center. For the edges of the game board, column 1 and 2 on left, and column 7 and 6 on right, the exact move-value score for first player start is loss on the 40th move, and loss on the 42nd move, respectively. In

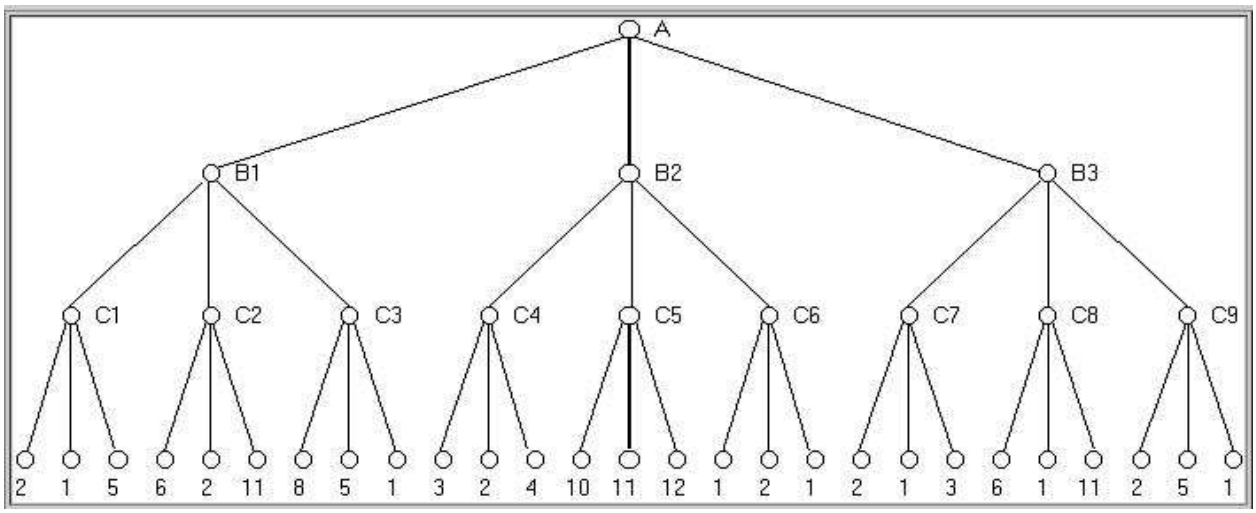
other words, by starting with the four outer columns, the first player allows the second player to force a win.

Minimax Algorithm

Search algorithms tend to utilize a cause-and-effect concept--the search considers each possible action available to it at a given moment; it then considers its subsequent moves from each of those states, and so on, in an attempt to findterminal states which satisfy the goal conditions it was given. Upon finding a goal state, it then follows the steps it knows are necessary to achieve that state.

However, in a competitive multi-player game, when other agents are involved and have different goals in mind (and agents usually in fact have goals which directly oppose each other), things get more complicated. Even if a search algorithm can find a goal state, it usually cannot simply take a set of actions which will reach that state, since for every action our algorithm takes towards its goal, the opposing player can take an action which will alter the current state (presumably in a way unfavorable to our algorithm.) This does not mean searches are useless in finding strategies for multi-player games; they simply require additional tactics to be effective.

For two player games, the minimax algorithm is such a tactic, which uses the fact that the two players are working towards opposite goals to make predictions about which future states will be reached as the game progresses, and then proceeds accordingly to optimize its chance of victory. The theory behind minimax is that the algorithm's opponent will be trying to minimize whatever value the algorithm is trying to maximize (hence, "minimax"). Thus, the computer should make the move which leaves its opponent capable of doing the least damage.



In the above example assuming normally alternating turns, if the computer has the choice at level A, its opponent will have the choice at B, and the computer will again have the choice at C. Since the computer is trying to maximize its score, it can know ahead of time what it would choose should any given C node be reached. C1 thus effectively has a score of 5, C2, 11, C3, 8,

and so on. When the opponent has a choice to make at B, however, they will choose the path that leads to the C node with the lowest effective score. Thus, the score of each B node can be thought to be the minimum of the effective scores of the C-nodes it leads to. For example, B1's score would be 5 (the minimum of 5, 11, and 8, as calculated above). B2 and B3 can be calculated in a similar fashion. Finally, we are back to the current turn. The computer now knows what will come of choosing B1, B2, or B3; and, though the actual endgame is many turns away, it will choose the maximum of those three for the best possible result. Note that, if the opponent does not behave as predicted, the calculation can simply be re-run, taking the current state as the starting node, and a result as good (or better) than what was predicted will still be achieved.

A major problem with this approach is how pessimistic it is. In a trivial example like the one above, minimax is useful because it is a reasonable expectation that the computer's opponent can figure out what its best options are; in more complex games, however, this will not be so clear, and a computer running the minimax algorithm may sacrifice major winnings because it assumes its opponent will "see" a move or series of moves which could defeat it, even if those moves are actually quite counterintuitive--in short, the computer assumes it is playing an opponent as knowledgeable as itself.

In reality, however, an exhaustive use of the minimax algorithm, as shown above, tends to be hopelessly impractical--and, for many win-or-lose games, uninteresting. A computer can compute all possible outcomes for a relatively simple game like tic-tac-toe (disregarding symmetric game states, there are 9! = 362,880 possible outcomes; the X player has a choice of 9 spaces, then the O has a choice of 8, etc.), but it won't help. As veteran tic-tac-toe players know, there is no opening move which guarantees victory; so, a computer running a minimax algorithm without any sort of enhancements will discover that, if both it and its opponent play optimally, the game will end in a draw no matter where it starts, and thus have no clue as to which opening play is the "best." Even in more interesting win-or-lose games like chess, even if a computer could play out every possible game situation (a hopelessly impossible task), this information alone would still lead it to the conclusion that the best it can ever do is draw (which would in fact be true, if both players had absolutely perfect knowledge of all possible results of each move). It is only for games where an intelligent player is guaranteed victory by going first or second (such as Nim, in many forms) that minimax will prove sufficient.

Alpha-Beta Pruning

Alpha-beta pruning is an improvement over the minimax algorithm. The problem with minimax is that the number of game states it has to examine is exponential in the number of moves. While it is impossible to eliminate the exponent completely, we are able to cut it in half. It is possible to compute the correct minimax decision without looking at every node in the tree. Borrowing the idea of pruning, or eliminating possibilities from consideration without having to examine them, the algorithm allows us to discard large parts of the tree from consideration.

hen applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

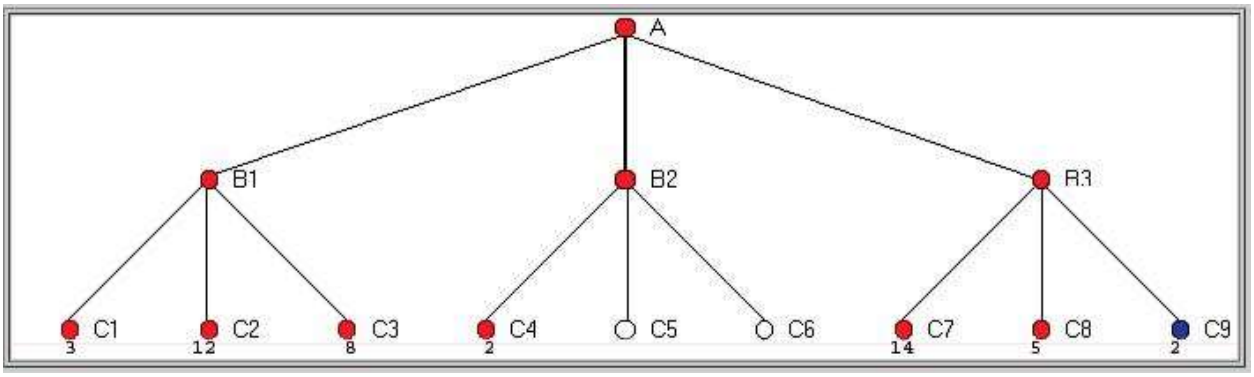
Alpha-beta pruning can be applied to trees of any depth and it often allows to prune away entire subtrees rather than just leaves. Here is the general algorithm:

1. Consider a node *n* somewhere in the tree, such that one can move to that node. 2. If there is a better choice *m* either at the parent of the node *n* or at any choice point further

3. Once we have enough information about n to reach this conclusion, we can prune it.

Alpha-Beta pruning gets its name from the following parameters: a = the value of the best choice we have found so far at any choice point along the path for MAX.

ß = the value of the best choice we have found so far at any choice point along the path for MIN



Alpha-Beta search updates the values of a and ß as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the

current values of a and ß. The success of the algorithm depends on the order in which the successors are examined.

Transportation tables have been used as a way to keep track of the information about individual nodes. Using such a table can have a dramatic effect, sometimes even doubling the Reachable search depth. However, if a million nodes are evaluated every second, keeping all the information is a table is not practical.

Program Code

Board.java

```
import java.io.*;

public interface Board extends Cloneable {

        public BoardStats getBoardStats();

        public boolean move(Move m);

        public void undoLastMove();

        public Move[] getPossibleMoves(Player aPlayer);

        public boolean isGameOver();

        public void gameStarted();

        public void gameRestarted();

        public void gameStoped();

        public Object clone();

 }
```

Boardstats.java

```
public interface BoardStats {

        public int getScore(Player aPlayer);

        public int getStrength(Player aPlayer);

        public int getMaxStrength();
```

```
        public int getMinStrength();
}//end interface BoardStats
```

Applet.Java

```
import java.applet.*;
import java.awt.*;

public class C4Applet extends Applet
{

        final static String ABOUT = "Four in a row, created by Sean Bridges,
www.geocities.com/sbridges.geo";

        private final static String L1 = "Level 1";
        private final static String L2 = "Level 2";
        private final static String L3 = "Level 3";
        private final static String L4 = "Level 4";
        private final static String L5 = "Level 5";
        private final static String L6 = "Level 6";
        private final static String L7 = "Level 7";
        private final static String L8 = "Level 8";

        private final static String STARTING_LEVEL = L4;
        private final static int STARTING_DEPTH = 4;

        private final static String RED_PIECE_IMAGE_NAME = "redpiece.gif";
        private final static String BLACK_PIECE_IMAGE_NAME = "blackpiece.gif";
```

```
private final static String BOARD_IMAGE_NAME = "board.jpg";



private Button restart;
private Choice levels;
private ImagePanel imagePanel;

private MinimaxPlayer computer;
private AsynchronousPlayer human;
private C4Board board;
private GameMaster gameMaster;



public void init ()
{
	System.out.println("C4Applet initializing");
	System.out.println(ABOUT);

	Image blackPieceImage = null;
	Image redPieceImage = null;
	Image boardImage = null;



	try
	{
		System.out.println("Loading images...");
		blackPieceImage = this.getImage(getCodeBase(),BLACK_PIECE_IMAGE_NAME);
		redPieceImage = this.getImage(getCodeBase(),RED_PIECE_IMAGE_NAME);
```

```
                boardImage = this.getImage(getCodeBase(), BOARD_IMAGE_NAME);


                MediaTracker imageTracker = new MediaTracker(this);

                imageTracker.addImage(boardImage, 1);
                imageTracker.addImage(redPieceImage, 2);
                imageTracker.addImage(blackPieceImage, 3);
                imageTracker.waitForAll();
                System.out.println("Images loaded");
            }
            catch (Exception e)
            {
                System.out.println("Error loading images");
                e.printStackTrace();
            }

            removeAll();

            human = new AsynchronousPlayer("human",C4Board.FIRST_PLAYER_NUMBER);
            computer = new MinimaxPlayer("computer", C4Board.SECOND_PLAYER_NUMBER,
human);
            computer.setDepth(STARTING_DEPTH);
            board = new C4Board(human, computer);
            Player[] players = new Player[2];
            players[0] = human;
            players[1] = computer;
```

```
                gameMaster = new GameMaster(board, players);
                this.setLayout(new BorderLayout());
                Panel controlPanel = new Panel();
                controlPanel.setBackground(Color.white);
                controlPanel.setLayout(new FlowLayout() );

                restart = new Button("restart");
                levels = new Choice();
                levels.add(L1);
                levels.add(L2);
                levels.add(L3);
                levels.add(L4);
                levels.add(L5);
                levels.add(L6);
                levels.add(L7);
                levels.add(L8);
                levels.select(STARTING_LEVEL);

                controlPanel.add(restart);



                this.add(controlPanel, BorderLayout.SOUTH);

                imagePanel = new ImagePanel(gameMaster, board, computer, human,boardImage,
redPieceImage, blackPieceImage);
                this.add(imagePanel, BorderLayout.CENTER);

        }
```

```
public void start()
{
	System.out.println("C4Applet starting");
	gameMaster.startGame();
	this.repaint();
}

public void stop()
{
	System.out.println("C4Applet stopping");
	gameMaster.stopGame();
}



public String getAppletInfo()
{
	return ABOUT;
}
	public boolean action (Event evt, Object arg)
{
	if(arg.equals("restart"))
	{
		gameMaster.restartGame();
	}
	else if(evt.target.equals(levels))
	{
```

```
if(levels.getSelectedItem() == L1)
{
	computer.setDepth(1);
}
if(levels.getSelectedItem() == L2)
{
	computer.setDepth(2);
}
if(levels.getSelectedItem() == L3)
{
	computer.setDepth(3);
}
if(levels.getSelectedItem() == L4)
{
	computer.setDepth(4);
}
if(levels.getSelectedItem() == L5)
{
	computer.setDepth(5);
}
if(levels.getSelectedItem() == L6)
{
	computer.setDepth(6);
}
if(levels.getSelectedItem() == L7)
{
	computer.setDepth(7);
}
```

```
                    if(levels.getSelectedItem() == L8)
                    {
                        computer.setDepth(8);
                    }

                }
                else
                {
                    return super.action(evt,arg);
                }
                return true;

            }



}//end class C4Applet

class ImagePanel extends Panel implements GameEventListener
{
    private final static int BOARD_TOP_X = 5;
    private final static int BOARD_TOP_Y = 25;

    private final static int BOARD_WIDTH = 420;
    private final static int BOARD_HEIGHT = 320;

    private final static int COLUMN_WIDTH = 50;
    private final static int ROW_HEIGHT = 50;
```

```
private final static int X_OFFSET = 15;
private final static int Y_OFFSET = 15;

private final static int TEXT_TOP_X = 80;
private final static int TEXT_TOP_Y = 80;
private final static Font TEXT_FONT = new Font("SansSerif", Font.BOLD, 36);

private final static int TIP_DIAMETER = 20;
private final static int TIP_TOP_X = 5;
private final static int TIP_TOP_Y = 5;
private final static int BLACK_TIP_OFFSET = 15;
private final static int RED_TIP_OFFSET = 35;
private Image offscreenImage;
private Graphics offscreenGraphics;

Image boardImage;
Image blackPieceImage;
Image redPieceImage;

private Move lastComputerMove;

private GameMaster game;
private Player computer;
private AsynchronousPlayer human;
private C4Board board;
int blackColumn = -1;
```

```
    ImagePanel(GameMaster aGame, C4Board aBoard, Player computer, AsynchronousPlayer
human,
                                Image boardImage, Image redPieceImage, Image blackPieceImage)
    {
        super();

        this.setBackground(Color.white);

        this.redPieceImage = redPieceImage;
        this.blackPieceImage = blackPieceImage;
        this.boardImage = boardImage;

        game = aGame;
        board = aBoard;
        this.computer = computer;
        this.human = human;
        game.addListener(this);
    }

    public void paint(Graphics g)
    {
        g.drawImage(offscreenImage,0,0, this);
    }

    public void update()
    {
        Graphics g = this.getGraphics();
        g.drawImage(offscreenImage,0,0, this);
```

```
}

public boolean mouseMove(Event evt,int x, int y)
{
	if((BOARD_TOP_X < x) & (x < (BOARD_TOP_X + BOARD_WIDTH)))
	{
		if((TIP_TOP_Y < y) & (y < (BOARD_TOP_Y + BOARD_HEIGHT)))
		{
			x = x - BOARD_TOP_X - 10;
			int column = x /COLUMN_WIDTH;
			if(column >= C4Board.NUMBER_OF_COLUMNS)
			{
				column = -1;
			}

			setBlackColumn(column);
		}
	}
	else
	{
		setBlackColumn(-1);
	}

	return true;
}

private void setBlackColumn(int newColumn)
{
```

```
        if(newColumn != blackColumn)
        {
                if(newColumn != -1)
                {
                        drawBlackTip(newColumn);
                }
                if(blackColumn != -1)
                {
                        clearBlackTip(blackColumn);
                }
                blackColumn = newColumn;
                update();

        }

}

public boolean mouseDown(Event evt, int x, int y)
{
        if(blackColumn != -1)
        {
                C4Move move = new C4Move(human, blackColumn);
                human.makeMove(move);
        }
        return true;
}

private void resetOffScreen()
```

```
	{
		offscreenImage = createImage(this.getSize().width, this.getSize().height);
		offscreenGraphics = offscreenImage.getGraphics();
		drawBoard();
		update();
	}

	private void drawBoard()
	{
		offscreenGraphics.drawImage(boardImage, BOARD_TOP_X,BOARD_TOP_Y,this);
	}


	private void drawRedTip(int column)
	{
		offscreenGraphics.setColor(Color.red);
		offscreenGraphics.fillOval((column * COLUMN_WIDTH) + (TIP_TOP_X +
RED_TIP_OFFSET), TIP_TOP_Y, TIP_DIAMETER, TIP_DIAMETER);
	}
	private void drawBlackTip(int column)
	{
		offscreenGraphics.setColor(Color.black);
		offscreenGraphics.fillOval((column * COLUMN_WIDTH) + (TIP_TOP_X +
BLACK_TIP_OFFSET), TIP_TOP_Y, TIP_DIAMETER, TIP_DIAMETER);
	}

	private void clearBlackTip(int column)
	{
```

```
            offscreenGraphics.setColor(Color.white);

            offscreenGraphics.fillOval((column * COLUMN_WIDTH) + (TIP_TOP_X +
BLACK_TIP_OFFSET), TIP_TOP_Y, TIP_DIAMETER, TIP_DIAMETER);

        }


        private void clearRedTip(int column)

        {

            offscreenGraphics.setColor(Color.white);

            offscreenGraphics.fillOval((column * COLUMN_WIDTH) + (TIP_TOP_X +
RED_TIP_OFFSET), TIP_TOP_Y, TIP_DIAMETER, TIP_DIAMETER);

        }


        private void drawBlackToken(int row, int column)

        {

            int xPos = BOARD_TOP_X + (column * COLUMN_WIDTH) + X_OFFSET;

            int yPos = BOARD_TOP_Y + ((C4Board.NUMBER_OF_ROWS - 1 - row) * ROW_HEIGHT) +
Y_OFFSET;

            offscreenGraphics.drawImage(blackPieceImage,xPos,yPos,this);

        }


        private void drawRedToken(int row, int column)

        {

            int xPos = BOARD_TOP_X + (column * COLUMN_WIDTH) + X_OFFSET;

            int yPos = BOARD_TOP_Y + ((C4Board.NUMBER_OF_ROWS - 1 - row) * ROW_HEIGHT) +
Y_OFFSET;

            offscreenGraphics.drawImage(redPieceImage,xPos,yPos,this);

        }


        public void gameStarted()
```

```
{
        lastComputerMove = null;
        resetOffScreen();
}

public void gameStoped()
{
        if(board.isGameOver())
        {
                offscreenGraphics.setFont(TEXT_FONT);
                offscreenGraphics.setColor(Color.black);
                if(board.getBoardStats().getScore(human) != 0)
                {
                        offscreenGraphics.drawString("You win", TEXT_TOP_X, TEXT_TOP_Y);
                }
                else if(board.getBoardStats().getScore(computer) != 0)
                {
                        offscreenGraphics.drawString("You lose", TEXT_TOP_X, TEXT_TOP_Y);
                }
                else
                {
                        offscreenGraphics.drawString("Tie game", TEXT_TOP_X, TEXT_TOP_Y);
                }
                update();
        }

}
```

```
public void gameRestarted()
{
	lastComputerMove = null;
	resetOffScreen();
}

public void moveMade(Move aMove)
{
	int column = aMove.toInt();
	int row = board.numerOfChipsInColumn(column) - 1;

	if(aMove.maker().getNumber() == C4Board.FIRST_PLAYER_NUMBER)
	{
		drawBlackToken(row,column);
		update();
	}
	else
	{
		drawRedToken(row, column);

		if(lastComputerMove != null)
		{
			clearRedTip(lastComputerMove.toInt());
		}

		drawRedTip(aMove.toInt());
		lastComputerMove = aMove;
		update();
```

```
                }
        }
}
```

MiniMax.Java

```
public class MinimaxPlayer extends DefaultPlayer
{
        private int depth = 1;
        private Player minPlayer;
        public MinimaxPlayer(String name, int number, Player minPlayer)
        {
                super(name, number);

                this.minPlayer = minPlayer;
        }


        public int getDepth()
        {
                return depth;
        }


        public void setDepth(int anInt)
        {
                depth = anInt;
```

```
        }


        public Move getMove(Board b)
        {
                MinimaxCalculator calc = new MinimaxCalculator(b,this,minPlayer);
                return calc.calculateMove(depth);
        }


}
final class MinimaxCalculator
{
        private int moveCount = 0;
        private long startTime;

        private Player minPlayer;
        private Player maxPlayer;
        private Board board;

        private final int MAX_POSSIBLE_STRENGTH;
        private final int MIN_POSSIBLE_STRENGTH;


        MinimaxCalculator(Board b, Player max, Player min)
        {
                board = b;
                maxPlayer = max;
```

```
        minPlayer = min;

        MAX_POSSIBLE_STRENGTH = board.getBoardStats().getMaxStrength();
        MIN_POSSIBLE_STRENGTH = board.getBoardStats().getMinStrength();
}


public Move calculateMove(int depth)
{
        startTime = System.currentTimeMillis();

        if(depth == 0)
        {
                System.out.println("Error, 0 depth in minumax player");
                Thread.dumpStack();
                return null;
        }

        Move[] moves = board.getPossibleMoves(maxPlayer);
        int maxStrength = MIN_POSSIBLE_STRENGTH;
        int maxIndex = 0;

        for(int i = 0; i < moves.length; i++)
        {
                if(board.move(moves[i]))
                {
                        moveCount++;
```

```
                    int strength = expandMinNode(depth -1, maxStrength);
                    if(strength > maxStrength)
                    {
                        maxStrength = strength;
                        maxIndex = i;
                    }
                    board.undoLastMove();
                }//end if move made


                if(Thread.currentThread().isInterrupted())
                {
                    return null;
                }

            }//end for all moves

            long stopTime = System.currentTimeMillis();
            System.out.print("MINIMAX: Number of moves tried:" + moveCount);
            System.out.println(" Time:" + (stopTime -  startTime) + " milliseconds");

            return moves[maxIndex];

}


private int expandMaxNode(int depth, int parentMinimum)
{
```

```
//base step
if(depth == 0 || board.isGameOver())
{
	return board.getBoardStats().getStrength(maxPlayer);
}

//recursive step
Move[] moves = board.getPossibleMoves(maxPlayer);
int maxStrength = MIN_POSSIBLE_STRENGTH;

for(int i = 0; i < moves.length; i++)
{
	if(board.move(moves[i]))
	{
		moveCount++;
		int strength = expandMinNode(depth -1, maxStrength);

		if(strength > parentMinimum)
		{
			board.undoLastMove();
			return strength;
		}
		if(strength > maxStrength)
		{
			maxStrength = strength;
		}
		board.undoLastMove();
	}//end if move made
```

```
        }//end for all moves

        return maxStrength;

}//end expandMaxNode



private int expandMinNode(int depth, int parentMaximum)
{
        //base step
        if(depth == 0 || board.isGameOver())
        {
                return board.getBoardStats().getStrength(maxPlayer);
        }

        //recursive step
        Move[] moves = board.getPossibleMoves(minPlayer);
        int minStrength = MAX_POSSIBLE_STRENGTH;

        for(int i = 0; i < moves.length; i++)
        {
                if(board.move(moves[i]))
                {
                        moveCount++;
                        int strength = expandMaxNode(depth -1, minStrength);
```

```
                    if(strength < parentMaximum)
                    {
                        board.undoLastMove();
                        return strength;
                    }
                    if(strength < minStrength)
                    {
                        minStrength = strength;
                    }
                    board.undoLastMove();
                }//end if move made

            }//end for all moves

            return minStrength;

        }//end expandMaxNode

}
```

GameMaster.java

```
import java.util.*;

public class GameMaster
{
    private Player[] players; //the players of the game
    private Board board;
```

```
	private int currentPlayerIndex = 0;
	private GameThread gameThread;
	public GameMaster(Board board, Player[] players )
	{

		if(players.length == 0)
		{
			System.err.println("Error, player array is empty, GameMaster() ");
			throw new ArrayIndexOutOfBoundsException("Game Master created with
empty array");
		}

		this.board = board;
		this.players = players;

	}

	public synchronized void addListener(GameEventListener l)
	{
		listeners.addElement(l);
	}

	public synchronized void removeListener(GameEventListener l)
	{
		listeners.removeElement(l);
	}
```

```
private synchronized Enumeration enumerateListeners()
{
	return ((Vector) listeners.clone() ).elements();
}



private void notifyListenersMoveMade(Move aMove)
{
	Enumeration e = enumerateListeners();
	while(e.hasMoreElements())
	{
		((GameEventListener) e.nextElement()).moveMade(aMove);
	}
}

private void notifyListenersGameReStarted()
{
	Enumeration e = enumerateListeners();
	while(e.hasMoreElements())
	{
		((GameEventListener) e.nextElement()).gameRestarted();
	}
}



private void notifyListenersGameStarted()
{
	Enumeration e = enumerateListeners();
```

```
        while(e.hasMoreElements())
        {
                ((GameEventListener) e.nextElement()).gameStarted();
        }
}

private void notifyListenersGameStoped()
{
        Enumeration e = enumerateListeners();
        while(e.hasMoreElements())
        {
                ((GameEventListener) e.nextElement()).gameStoped();
        }
}

private synchronized Player getCurrentPlayer()
{
        return players[currentPlayerIndex];
}
private synchronized void advancePlayer()
{
        currentPlayerIndex ++;
        if(currentPlayerIndex >= players.length)
        {
                currentPlayerIndex = 0;
        }
}
```

```
public synchronized void startGame()
{
	if(gameThread != null)
	{
		gameThread.stopThread();
		gameThread = null;
	}

	board.gameStarted();
	currentPlayerIndex = 0;
	gameThread = new GameThread();
	gameThread.start();
	notifyListenersGameStarted();

}

public synchronized void stopGame()
{
	if(gameThread != null)
	{
		gameThread.stopThread();
		gameThread = null;
	}

	board.gameStoped();
	notifyListenersGameStoped();
}
```

```
public synchronized void restartGame()
{
	if(gameThread != null)
	{
		gameThread.stopThread();
		gameThread = null;
	}

	board.gameRestarted();
	currentPlayerIndex = 0;
	gameThread = new GameThread();
	gameThread.start();
	notifyListenersGameReStarted();

}

class GameThread extends Thread
{

	public boolean active = true;

	public void stopThread()
	{
		if(active)
		{
			active = false;
			this.interrupt();
		}
```

```
}

public  void run()
{
        while( active && ! board.isGameOver() )
        {

                //get the players move.
                Player player = getCurrentPlayer();
                Move move = player.getMove((Board) board.clone());
                if(move == null)
                {
                        //ignore null moves
                        continue;
                }

                //if we are still active, make the move
                boolean moveMade = false;

                if(active)
                {
                        moveMade = board.move(move);
                }

                //if we are still active, and the move was made, notify listeners
                if(active && moveMade)
                {
                        advancePlayer();
```

```
                                        notifyListenersMoveMade(move);
                                }



                        }//end while


                        if(active && board.isGameOver())
                        {
                                notifyListenersGameStoped();
                        }


                        active = false;
                }//end run



        }//end class GameThread


}//end class GameMaster
```

Output: