California State University, Fresno

Computer Science Department

CSCI 264

Artificial Intelligence

Assignment 2

Submitted by: **Prof. David Ruby**

Sri Satya Sai Kamal Atluri (109479597)

Jeevana Kala Bommanagari (109504843)

Sri Ramya Nimmagadda (109492883)

Leeladhar Reddy Munnangi (109510225)

Yasasvi Yeleswarupu (109475008)

Contents

Introduction

Tic-tac-toe (also known as Noughts and crosses or Xs and Os) is a paper-and-pencil game for two players, *X* and *O*, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.

Tic-Tac-Toe is a relatively simple game which can be analyzed and completely solved. Since it is a finite-length 2-player game, one of the following must be true: Player 1 has a winning strategy, Player 2 has a winning strategy, or optimal play results in a tie. One way of analyzing a game such as this is to create a game tree, i.e. map out every possible game. However, as the board size increases, this becomes less and less feasible as the number of possible games increases exponentially.

History

An early variant of Tic-tac-toe was played in the Roman Empire, around the first century BC. It was called Terni Lapilli and instead of having any number of pieces, each player only had three, thus they had to move them around to empty spaces to keep playing. The game's grid markings have been found chalked all over Rome. However, according to Claudia Zaslavsky's book Tic Tac Toe: And Other Three-In-A Row Games from Ancient Egypt to the Modern Computer, Tic-tac-toe could originate back to ancient Egypt. Another closely related ancient game is Three Men's Morris which is also played on a simple grid and requires three pieces in a row to finish.

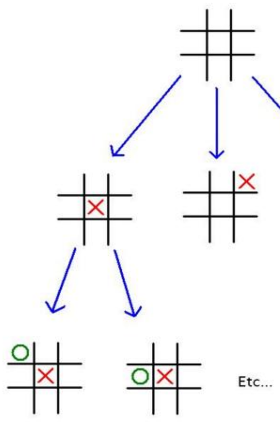
The different names of the game are more recent. The first print reference to "Noughts and crosses", the British name, appeared in 1864. In his novel "Can You Forgive Her", 1864, Anthony Trollope refers to a clerk playing "tit-tat-toe". The first print reference to a game called "tick-tack-toe" occurred in 1884, but referred to "a children's game played on a slate, consisting in trying with the eyes shut to bring the pencil down on one of the numbers of a set, the number hit being scored". "Tic-tac-toe" may also derive from "tick-tack", the name of an old version of backgammon first described in 1558. The U.S. renaming of Noughts and crosses as Tic-tac-toe occurred in the 20th century.

In 1952, OXO (or Noughts and Crosses) for the EDSAC computer became one of the first known video games. The computer player could play perfect games of Tic-tac-toe against a human opponent.

In 1975, Tic-tac-toe was also used by MIT students to demonstrate the computational power of Tinkertoy elements. The Tinkertoy computer, made out of (almost) only Tinkertoys, is able to play Tic-tac-toe perfectly.[5] It is currently on display at the Museum of Science, Boston.

Procedure

The “brute force” method of determining the outcome of a game based on optimal play is by mapping a game tree. This is done by first drawing how the game looks when it starts. You then draw all of the next possible states the game board could be in (in other words, all of the moves the first player can make). On the first move, we only have to draw 3 different states, since choosing one corner is the same as choosing any other, by reflection and rotation. Note that the same will not be true on subsequent moves. From each of these we can expand each of the second player’s possible moves, and so on, until we have mapped every possible game. You may wish to leave out trivial choices of moves (e.g. the player has the chance to win but picks another space).



Once you have mapped out a whole game tree, the game becomes easier to analyze. If you can show that no matter how Player 2 picks his/her moves, Player 1 has some way of winning, then you are guaranteed Player 1 has a winning strategy. However, if Player 2 can always pick a move which will lead to a tie, then you know the game will result in a tie given optimal play.

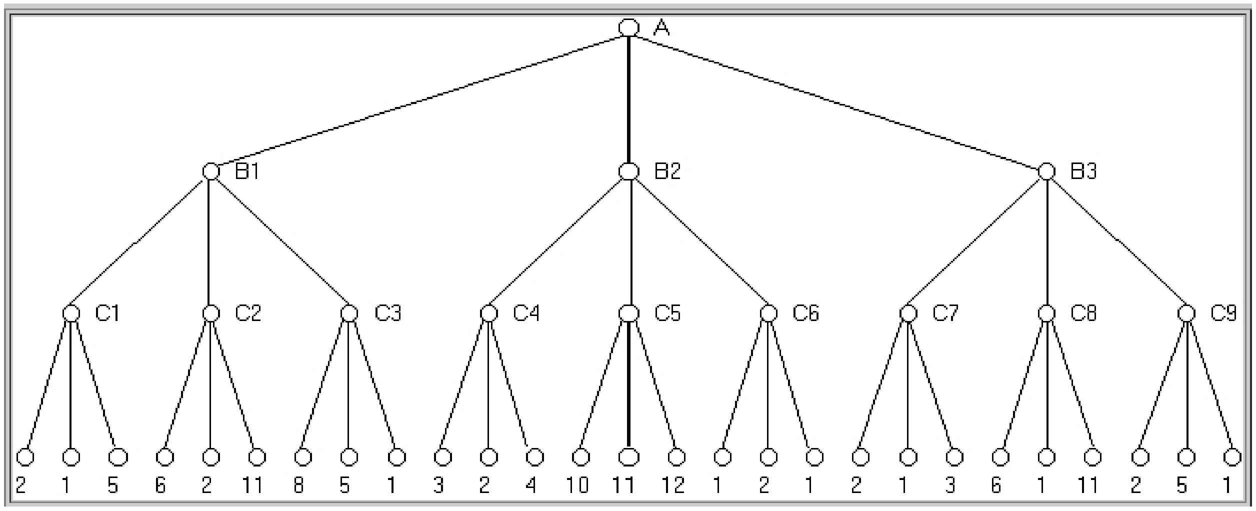
Mapping out a game tree takes a lot of time and space, and is not feasible for larger boards. However, by mapping out games on smaller boards, you may be able to develop a general strategy for any size board. Alternatively, you may wish to develop a strategy without mapping every possible game. Be sure that this strategy still covers every possibility.

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.

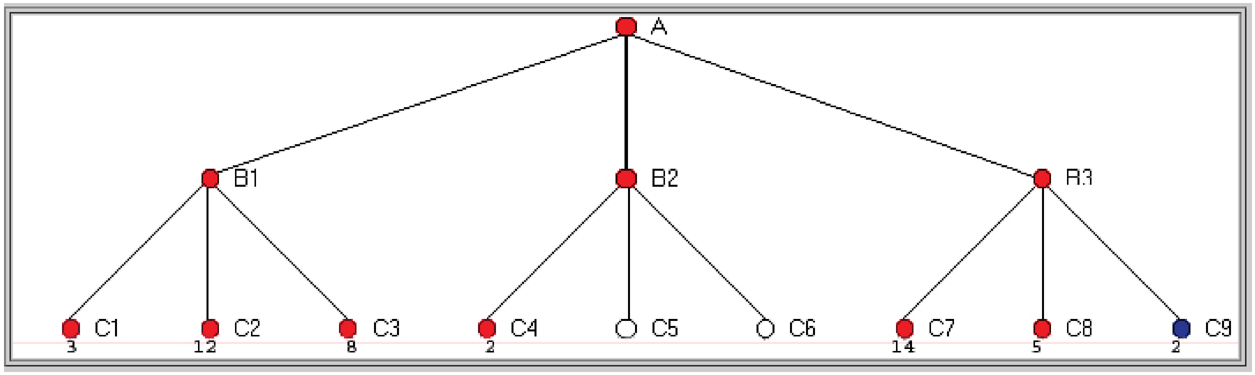
When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

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 up, *n* will never be reached.
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

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.Scanner;

class Point {

    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "[" + x + ", " + y + "]";
    }
}

class PointsAndScores {

    int score;
    Point point;

    PointsAndScores(int score, Point point) {
        this.score = score;
        this.point = point;
    }
}

class Board {

    List<Point> availablePoints;
    Scanner scan = new Scanner(System.in);
    int[][] board = new int[3][3];
```

```
List<PointsAndScores> rootsChildrenScore = new ArrayList<>();

public int evaluateBoard() {
    int score = 0;

    //Check all rows
    for (int i = 0; i < 3; ++i) {
        int blank = 0;
        int X = 0;
        int O = 0;
        for (int j = 0; j < 3; ++j) {
            if (board[i][j] == 0) {
                blank++;
            } else if (board[i][j] == 1) {
                X++;
            } else {
                O++;
            }

        }
        score+=changeInScore(X, O);
    }

    //Check all columns
    for (int j = 0; j < 3; ++j) {
        int blank = 0;
        int X = 0;
        int O = 0;
        for (int i = 0; i < 3; ++i) {
            if (board[i][j] == 0) {
                blank++;
            } else if (board[i][j] == 1) {
                X++;
            } else {
                O++;
            }
        }
        score+=changeInScore(X, O);
    }

    int blank = 0;
    int X = 0;
    int O = 0;

    //Check diagonal (first)
    for (int i = 0, j = 0; i < 3; ++i, ++j) {
        if (board[i][j] == 1) {
            X++;
        } else if (board[i][j] == 2) {
            O++;
        } else {
            blank++;
        }
    }

    score+=changeInScore(X, O);

    blank = 0;
    X = 0;
    O = 0;

    //Check Diagonal (Second)
    for (int i = 2, j = 0; i > -1; --i, ++j) {
        if (board[i][j] == 1) {
            X++;
        } else if (board[i][j] == 2) {
            O++;
        } else {
            blank++;
        }
    }

    score+=changeInScore(X, O);

    return score;
}
private int changeInScore(int X, int O){
    int change;
    if (X == 3) {
        change = 100;
    } else if (X == 2 && O == 0) {
        change = 10;
    } else if (X == 1 && O == 0) {
        change = 1;
    } else if (O == 3) {
        change = -100;
    } else if (O == 2 && X == 0) {
        change = -10;
    } else if (O == 1 && X == 0) {
        change = -1;
    } else {
        change = 0;
    }
    return change;
}

//Set this to some value if you want to have some specified depth limit
for search
int uptoDepth = -1;

public int alphaBetaMinimax(int alpha, int beta, int depth, int turn){

    if(beta<=alpha){ System.out.println("Pruning at depth =
"+depth);if(turn == 1) return Integer.MAX_VALUE; else return
Integer.MIN_VALUE; }

    if(depth == uptoDepth || isGameOver()) return evaluateBoard();

    List<Point> pointsAvailable = getAvailableStates();

    if(pointsAvailable.isEmpty()) return 0;

    if(depth==0) rootsChildrenScore.clear();

    int maxValue = Integer.MIN_VALUE, minValue = Integer.MAX_VALUE;

    for(int i=0;i<pointsAvailable.size(); ++i){
        Point point = pointsAvailable.get(i);

        int currentScore = 0;

        if(turn == 1){
            placeAMove(point, 1);
            currentScore = alphaBetaMinimax(alpha, beta, depth+1, 2);
            maxValue = Math.max(maxValue, currentScore);

            //Set alpha
            alpha = Math.max(currentScore, alpha);

            if(depth == 0)
                rootsChildrenScore.add(new PointsAndScores(currentScore,
point));
        }else if(turn == 2){
            placeAMove(point, 2);
            currentScore = alphaBetaMinimax(alpha, beta, depth+1, 1);
            minValue = Math.min(minValue, currentScore);

            //Set beta
            beta = Math.min(currentScore, beta);
        }
        //reset board
        board[point.x][point.y] = 0;

        //If a pruning has been done, don't evaluate the rest of the
sibling states
        if(currentScore == Integer.MAX_VALUE || currentScore ==
Integer.MIN_VALUE) break;
    }
    return turn == 1 ? maxValue : minValue;
}

public boolean isGameOver() {
    //Game is over is someone has won, or board is full (draw)
    return (hasXWon() || hasOWon() || getAvailableStates().isEmpty());
}

public boolean hasXWon() {
    if ((board[0][0] == board[1][1] && board[0][0] == board[2][2] &&
board[0][0] == 1) || (board[0][2] == board[1][1] && board[0][2] ==
board[2][0] && board[0][2] == 1)) {
        //System.out.println("X Diagonal Win");
        return true;
    }
    for (int i = 0; i < 3; ++i) {
        if (((board[i][0] == board[i][1] && board[i][0] == board[i][2] &&
board[i][0] == 1)
                || (board[0][i] == board[1][i] && board[0][i] ==
board[2][i] && board[0][i] == 1))) {
            // System.out.println("X Row or Column win");
            return true;
        }
    }
    return false;
}

public boolean hasOWon() {
    if ((board[0][0] == board[1][1] && board[0][0] == board[2][2] &&
board[0][0] == 2) || (board[0][2] == board[1][1] && board[0][2] ==
board[2][0] && board[0][2] == 2)) {
        // System.out.println("O Diagonal Win");
        return true;
    }
    for (int i = 0; i < 3; ++i) {
        if ((board[i][0] == board[i][1] && board[i][0] == board[i][2] &&
board[i][0] == 2)
                || (board[0][i] == board[1][i] && board[0][i] ==
board[2][i] && board[0][i] == 2)) {
```

```
            //  System.out.println("O Row or Column win");
            return true;
        }
    }

    return false;
}

public List<Point> getAvailableStates() {
    availablePoints = new ArrayList<>();
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (board[i][j] == 0) {
                availablePoints.add(new Point(i, j));
            }
        }
    }
    return availablePoints;
}

public void placeAMove(Point point, int player) {
    board[point.x][point.y] = player;   //player = 1 for X, 2 for O
}

public Point returnBestMove() {
    int MAX = -100000;
    int best = -1;

    for (int i = 0; i < rootsChildrenScore.size(); ++i) {
        if (MAX < rootsChildrenScore.get(i).score) {
            MAX = rootsChildrenScore.get(i).score;
            best = i;
        }
    }

    return rootsChildrenScore.get(best).point;
}

void takeHumanInput() {
    System.out.println("Your move: ");
    int x = scan.nextInt();
    int y = scan.nextInt();
    Point point = new Point(x, y);
    placeAMove(point, 2);
}

public void displayBoard() {
        System.out.println();

        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                System.out.print(board[i][j] + " ");
            }
            System.out.println();

        }
    }

    public void resetBoard() {
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                board[i][j] = 0;
            }
        }
    }
}

public class AlphaBetaMinimaxTTT {

    public static void main(String[] args) {
        Board b = new Board();
        Random rand = new Random();

        b.displayBoard();

        System.out.println("Who's gonna move first? (1)Computer (2)User: ");
        int choice = b.scan.nextInt();
        if (choice == 1) {
            Point p = new Point(rand.nextInt(3), rand.nextInt(3));
            b.placeAMove(p, 1);
            b.displayBoard();
        }

        while (!b.isGameOver()) {
            System.out.println("Your move: ");
            Point userMove = new Point(b.scan.nextInt(), b.scan.nextInt());

            b.placeAMove(userMove, 2); //2 for O and O is the user
            b.displayBoard();
            if (b.isGameOver()) break;

            b.alphaBetaMinimax(Integer.MIN_VALUE, Integer.MAX_VALUE, 0, 1);
            for (PointsAndScores pas : b.rootsChildrenScore)
```

```
                System.out.println("Point: " + pas.point + " Score: " +
pas.score);

            b.placeAMove(b.returnBestMove(), 1);
            b.displayBoard();
        }
        if (b.hasXWon()) {
            System.out.println("Unfortunately, you lost!");
        } else if (b.hasOWon()) {
            System.out.println("You win!");
        } else {
            System.out.println("It's a draw!");
        }
    }
}
```

Output

```
0 0 0
0 0 0
0 0 0
Who's gonna move first? (1)Computer (2)User:
2
Your move:
2 2

0 0 0
0 0 0
0 0 2
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 5
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 5
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 4
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 5
Pruning at depth = 7
Pruning at depth = 5
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 4
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 4
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 4
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 5
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 5
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 5
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 5
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 6
Pruning at depth = 4
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 7
Pruning at depth = 5
Pruning at depth = 7
Pruning at depth = 5
Pruning at depth = 7
Pruning at depth = 6
Pruning at depth = 7
Pruning at depth = 5
Pruning at depth = 6
Pruning at depth = 7

Point: [0, 0] Score: -90
Point: [0, 1] Score: -90
Point: [0, 2] Score: -90
Point: [1, 0] Score: -91
Point: [1, 1] Score: 0
Point: [1, 2] Score: 0
Point: [2, 0] Score: 0
Point: [2, 1] Score: 0

0 0 0
0 1 0
0 0 2
Your move:
0 0

2 0 0
0 1 0
0 0 2
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 3
Pruning at depth = 5
Pruning at depth = 3
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 3
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 3
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 2
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 3
Pruning at depth = 4
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 2
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 5
Pruning at depth = 5
Pruning at depth = 3
Pruning at depth = 4
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 5
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 2
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 2
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 4
Pruning at depth = 2
Point: [0, 1] Score: 0
Point: [0, 2] Score: 0
Point: [1, 0] Score: 0
Point: [1, 2] Score: 0
Point: [2, 0] Score: -90
Point: [2, 1] Score: 0

2 1 0
0 1 0
0 0 2
Your move:
2 1

2 1 0
0 1 0
0 2 2
Pruning at depth = 3
Pruning at depth = 3
Point: [0, 2] Score: -109
Point: [1, 0] Score: -100
Point: [1, 2] Score: -100
Point: [2, 0] Score: 0

2 1 0
0 1 0
1 2 2
Your move:
0 2

2 1 2
0 1 0
1 2 2
Point: [1, 0] Score: -100
Point: [1, 2] Score: 0

2 1 2
0 1 1
1 2 2
Your move:
1 0
```

```
2 1 2
2 1 1
1 2 2
It's a draw!
```

Conclusion

All the searches we discuss have their niche in the body of problems we ask computer to solve- this is evidenced by their continued use and continued efforts to optimize them. There is no "ideal" search algorithm for every situation, or even a large number of situations. This, in part, is because questions which are answered by search are far from homogenous in nature. Two player games cannot be solved in the same way one would find an ideal route for a map; and even among such categories, the same search may not be as useful for one game (say, chess) as another (say, backgammon, which includes chance). A more interesting optimization is that of the heuristic devices common to intelligent search. One search may be slightly more efficient than another in a certain scenario, but designing a better heuristic will always speed things up- and even make things possible that otherwise were not.

In a field as broad and complex as artificial intelligence, it is often hard to have a good place to establish foundation for a basic understanding. These searches, though at the core of the artificial intelligence's most advanced theories and most notable achievements, are nonetheless comprehensible to relatively inexperienced computer scientists. We hope that from this basic understanding, knowledge of the deepest parts of this vast area can begin to grow.