Introducing the Min-Max Algorithm

Paulo Pinto

28 July 2002

Submited to the *AI Depot* article contest.

Contents

1 Introduction

There are plenty of applications for AI, but games are the most interesting to the public.

Nowadays every major OS comes with some games.

So it is no surprise that there are some algorithms that were devised with games in mind.

2 The Min-Max Algorithm

The Min-Max algorithm is applied in two player games, such as tic-tac-toe, checkers, chess, go, and so on.

All these games have at least one thing in common, they are logic games. This means that they can be described by a set of rules and premisses. With them, it is possible to know from a given point in the game, what are the next available moves. So they also share other characteristic, they are 'full information games'. Each player knows everything about the possible moves of the adversary.

Before explaining the algorithm, a brief introduction to search trees is required. Search trees are a way to represent searches. In Figure 1 you can a representation of a search tree. The squares are known as nodes and they represent points of the decision in the search. The nodes are connected with branches. The search starts at the root node, the one at the top of the figure. At each decision point, nodes for the available search paths are generated, until no more decisions are possible. The nodes that represent the end of the search are known as leaf nodes.

There are two players involved, MAX and MIN. A search tree is generated, depth-first, starting with the current game position upto the end game position. Then, the final game position is evaluated from MAX's point of view, as shown in Figure 1. Afterwards, the inner node values of the tree are filled bottom-up with the evaluated values. The nodes that belong to the MAX player receive the maximun value of it's children. The nodes for the MIN player will select the minimun value of it's children. The algorithm is described in Listing 1.

So what is happening here? The values represent how good a game move is. So the MAX player will try to select the move with highest value in the end. But the MIN player also has something to say about it and he will try to select the moves that are better to him, thus minimizing MAX's outcome.

```
MinMax (GamePosition game) {
  return MaxMove (game);
}

MaxMove (GamePosition game) {
  if (GameEnded(game)) {
    return EvalGameState(game);
  }
  else {
    best_move <- {};
    moves <- GenerateMoves(game);
    ForEach moves {
       move <- MinMove(ApplyMove(game));
       if (Value(move) > Value(best_move)) {
          best_move <- move;
       }
    }
    return best_move;
  }
}

MinMove (GamePosition game) {
  best_move <- {};
  moves <- GenerateMoves(game);
  ForEach moves {
     move <- MaxMove(ApplyMove(game));
     if (Value(move) > Value(best_move)) {
        best_move <- move;
     }
  }

  return best_move;
}
```

Table 1: Basic Min-Max Algorithm

3 Optimization

However only very simple games can have their entire search tree generated in a short time. For most games this isn't possible, the universe would probably vanish first. So there are a few optimizations to add to the algorithm.

First a word of caution, optimization comes with a price. When optimizing we are trading the full information about the game's events with probabilities and shortcuts. Instead of knowing the full path that leads to victory, the decisions are made with the path that might lead to victory. If the optimization isn't well choosen, or it is badly applied, then we could end with a dumb AI. And it would have been better to use random moves.

One basic optimization is to limit the depth of the search tree. Why does this help? Generating the full tree could take ages. If a game has a branching factor of 3, which means that each node has tree children, the tree will have the folling number of nodes per depth:

The sequence shows that at depth n the tree will have 3^n nodes. To know the total number of generated nodes, we need to sum the node count at each level. So the total number of nodes for a tree with depth n is $\sum_{n=0}^{n} 3^n$. For many games, like chess that have a very big branching factor, this means that the tree might not fit into memory. Even if it

Depth	Node Count
0	1
1	3
2	9
3	27
...	...
n	3^n

Table 2: Node Count per Tree Depth

did, it would take to long to generate. If each node took 1s to be analyzed, that means that for the previous example, each search tree would take $\sum_{n=0}^{n} 3^n * 1s$. For a search tree with depth 5, that would mean 1+3+9+27+81+243 = 364 * 1 = 364s = 6m! This is too long for a game. The player would give up playing the game, if he had to wait 6m for each move from the computer.

The second optimization is to use a function that evaluates the current game position from the point of view of some player. It does this by giving a value to the current state of the game, like counting the number of pieces in the board, for example. Or the number of moves left to the end of the game, or anything else that we might use to give a value to the game position.

Instead of evaluating the current game position, the function might calculate how the current game position might help ending the game. Or in another words, how probable is that given the current game position we might win the game. In this case the function is known as an estimation function.

This function will have to take into account some heuristics. Heuristics are knowledge that we have about the game, and it can help generate better evaluation functions. For example, in checkers, pieces at corners and sideways positions can't be eaten. So we can create an evaluation function that gives higher values to pieces that lie on those board positions thus giving higher outcomes for game moves that place pieces in those positions.

One of the reasons that the evaluation function must be able to evalute game positions for both players is that you don't know to which player the limit depth belongs.

However having two functions can be avoided if the game is symetric. This means that the loss of a player equals the gains of the other. Such games are also known as ZERO-SUM games. For these games one evalution function is enough, one of the players just have to negate the return of the function.

The revised algorithm is described in Listing 3.

Even so the algorithm has a few flaws, some of them can be fixed while other can only be solved by choosing another algorithm.

One of flaws is that if the game is too complex the answer will always take too long even with a depth limit. One solution it limit the time for search. If the time runs out choose the best move found until the moment.

A big flaw is the limited horizon problem. A game position that appears to be very good might turn out very bad. This happens because the algorithm wasn't able to see that a few game moves ahead the adversary will be able to make a move that will bring him a great outcome. The algorithm missed that fatal move because it was blinded by the depth limit.

4 Speeding the algorithm

There are a few things can still be done to reduce the search time. Take a look at figure 2. The value for node A is 3, and the first found value for the subtree starting at node B is 2. So since the B node is at a MIN level, we know that the selected value for the B node must be less or equal than 2. But we also know that the A node has the value 3, and both A and B nodes share the same parent at a MAX level. This means that the game path starting at the B node wouldn't be selected because 3 is better than 2 for the MAX node. So it isn't worth to pursue the search for children of the B

node, and we can safely ignore all the remaining children.

This all means that sometimes the search can be aborted because we find out that the search subtree won't lead us to any viable answer.

This optimization is know as alpha-beta cuttoffs and the algorithm is as follows:

- Have two values passed around the tree nodes:
  - the alpha value which holds the best MAX value found;
  - the beta value which holds the best MIN value found.
- At MAX level, before evaluating each child path, compare the returned value with of the previous path with the beta value. If the value is greater than it abort the search for the current node;
- At MIN level, before evaluating each child path, compare the returned value with of the previous path with the alpha value. If the value is lesser than it abort the search for the current node.

The Listing 4 shows the full pseudocode for MinMax with alpha-beta cuttoffs. How better does a MinMax with alpha-beta cuttoffs behave when compared with a normal MinMax? It depends on the order the search is searched. If the way the game positions are generated doesn't create situations where the algorithm can take advantage of alpha-beta cutoffs then the improvements won't be noticible. However, if the evaluation function and the generation of game positions leads to alpha-beta cuttoffs then the improvements might be great.

5 Adding Alpha-Beta Cutoffs

With all this talk about search speed many of you might be wondering what this is all about. Well, the search speed is very important in AI because if an algorithm takes too long to give a good answer the algorithm may not be suitable.

For example, a good MinMax algorithm implementation with an evaluation function capable to give very good estimatives might be able to search 1000 positions a second. In tourament chess each player has around 150 seconds to make a move. So it would probably be able to analyze 150 000 positions during that period. But in chess each move has around 35 possible branchs! In the end the program would only be able to analyze around 3, to 4 moves ahead in the game[1]. Even humans with very few pratice in chess can do better than this.

But if we use MinMax with alpha-beta cutoffs, again a decent implementation with a good evaluation function, the result behaviour might be much better. In this case, the program might be able to double the number of analyzed positions and thus becoming a much toughter adversary.

6 An example implementation

An example is always a good way to show how an algorithm might be implemented. Back in 1999, I and a friend of mine have implemented a checkers game as a Java application for the AI class in the university. I have recently ported the game to C#.

The MinMax algortihm isn't a great implementation. In fact I should mention that the best thing about it is that it works. However I think that it presents a way that the algorithm might be implemented and as an example it is good enough.

The game uses MinMax with alpha-beta cutoffs for the computer moves. The evaluation function is an weighted average of the positions occupied by the checker pieces. The figure 3 shows the values for each board position. The value of each board position is multiplied by the type of the piece that rests on that position, described in Table 6. The Listing 5 shows the Java implementation of the evaluation function. It has been slightly modified for the article.

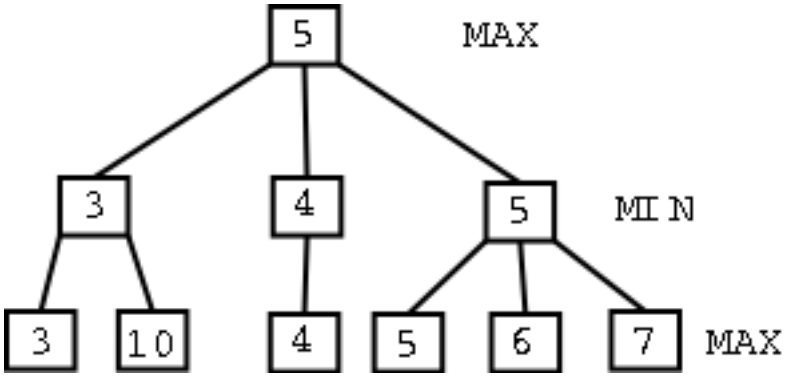

Figure 1: A search tree

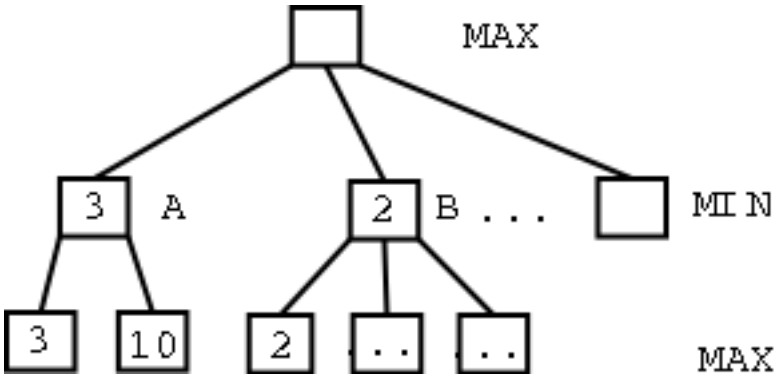

Figure 2: A cutted search tree

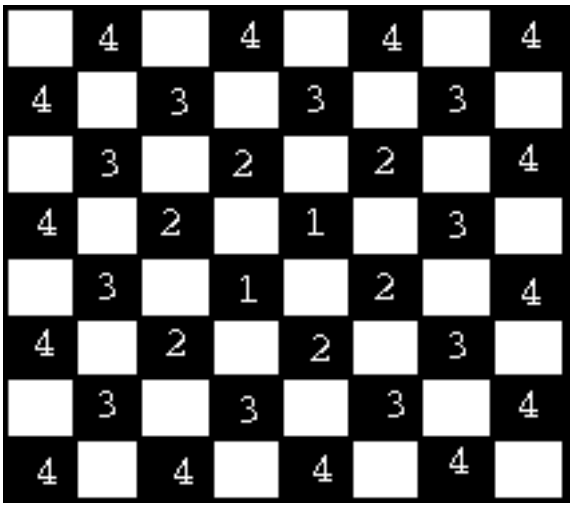

Figure 3: Board Values

Please note that the code uses a vector, 0-31 based, for the board game positions.

The game code is available at:

- Java version - http://www.progtools.org/graphics/projects/checkers
- C# version - http://www.progtools.org/graphics/projects/sharp_checkers

7 Conclusion

The MinMax might not be the best answer for all kind of computer games that need to have AI that resembles human behaviour. But given a good implementation it can create a tought adversary. I hope that this article has gave you some insight on the MinMax algorithm and how you might use it on your games.

8 References

1 - Russell Stuart J., Norvig Peter. "Artificial Intelligence. A modern approach.". Prentice Hall, 1995. ISBN 0-13-103805-2.

2 - Bratko Ivan. "PROLOG. Programming for artificial intelligence" Addison-Wesley, 1990. ISBN 0-201-41606-9

3 - Rich Elaine, Knight Kevin. "Artificial Intelligence". McGraw-Hill Inc., 1991. ISBN 0-07-100894-2

4 - Analytical Reasoning FAQ. Available at http://www.west.net/ stewart/lwfaq.htm

```
MinMax (GamePosition game) {
  return MaxMove (game);
}

MaxMove (GamePosition game) {
  if (GameEnded(game) || DepthLimitReached()) {
    return EvalGameState(game, MAX);
  }
  else {
    best_move <- {};
    moves <- GenerateMoves(game);
    ForEach moves {
       move <- MinMove(ApplyMove(game));
       if (Value(move) > Value(best_move)) {
          best_move <- move;
       }
    }
    return best_move;
  }
}

MinMove (GamePosition game) {
  if (GameEnded(game) || DepthLimitReached()) {
    return EvalGameState(game, MIN);
  }
  else {
    best_move <- {};
    moves <- GenerateMoves(game);
    ForEach moves {
       move <- MaxMove(ApplyMove(game));
       if (Value(move) > Value(best_move)) {
          best_move <- move;
       }
    }
    return best_move;
  }
}
```

Table 3: Min-Max with Depth Limit

```
MinMax (GamePosition game) {
  return MaxMove (game);
}

MaxMove (GamePosition game, Integer alpha, Integer beta) {
  if (GameEnded(game) || DepthLimitReached()) {
    return EvalGameState(game, MAX);
  }
  else {
    best_move <- {};
    moves <- GenerateMoves(game);
    ForEach moves {
       move <- MinMove(ApplyMove(game), alpha, beta);
       if (Value(move) > Value(best_move)) {
          best_move <- move;
          alpha <-  Value(move);
       }

       // Ignore remaining moves
       if (beta > alpha)
         return best_move;
    }
    return best_move;
  }
}

MinMove (GamePosition game) {
  if (GameEnded(game) || DepthLimitReached()) {
    return EvalGameState(game, MIN);
  }
  else {
    best_move <- {};
    moves <- GenerateMoves(game);
    ForEach moves {
       move <- MaxMove(ApplyMove(game), alpha, beta);
       if (Value(move) > Value(best_move)) {
          best_move <- move;
          beta <- Value(move);
       }

       // Ignore remaining moves
       if (beta < alpha)
         return best_move;
    }
    return best_move;
  }
}
```

Table 4: Min-Max with Alpha-Beta Cuttoffs

```
// Snippet from Computer.java.
// Contains the evaluation function

/**
 * Evaluation function.
 */
private int eval (CheckersBoard board) {
    int colorKing;

    // Finds out who is the current player
    if (color == CheckersBoard.WHITE)
      colorKing = CheckersBoard.WHITE_KING;
    else
      colorKing = CheckersBoard.BLACK_KING;


    int colorForce = 0;
    int enemyForce = 0;
    int piece;

    try {
      // Searchs all board positions for pieces
      // and evaluates each position.
      for (int i = 0; i < 32;  i++) {
        piece = board.getPiece (i);

        if (piece != CheckersBoard.EMPTY)
          if (piece == color || piece == colorKing)
            colorForce += calculateValue (piece, i);
          else
            enemyForce += calculateValue (piece, i);
      }
    }
    catch (BadCoord bad) {
      bad.printStackTrace ();
      System.exit (-1);
    }

    return colorForce - enemyForce;
}

/**
 * Measures the value of a checkers piece, given
 * it's position in the board
 */
private int calculateValue (int piece, int pos) {
  int value;

  if (piece == CheckersBoard.WHITE ) //Simple piece
   if (pos >= 4 && pos <= 7) // White pieces are more
     value = 7;              // valuable the closer they get
   else                      // to the oponent
     value = 5;
  else if (piece != CheckersBoard.BLACK) //Simple piece
     if (pos >= 24 && pos <= 27) // White pieces are more
        value = 7;               // valuable the closer they get
     else                        // to the oponent
        value = 5;
  else // King piece
    value = 10;                  // King pieces are always the
                                 // most valuable

  return value * tableWeight[pos];
}
```

Piece	Value
Normal	5
Normal but close to being promoted	7
King	10

Table 6: Piece Values