An Exhaustive Explanation of Minimax, a Staple AI Algorithm

*10 January 2012*

Overview

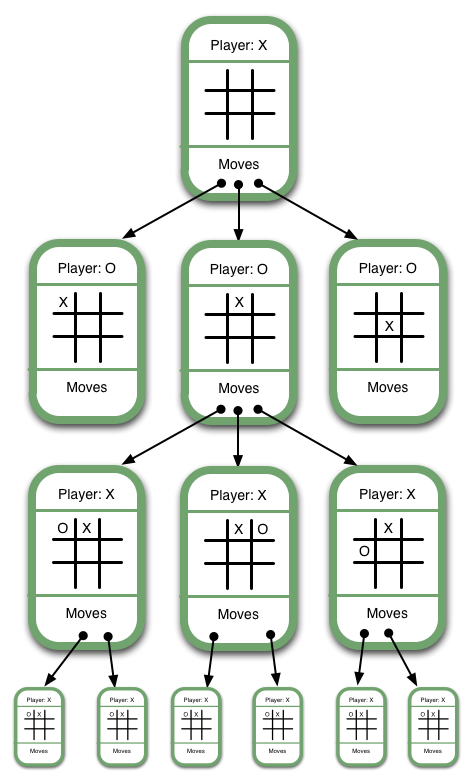
The minimax algorithm is used to determine which moves a computer player makes in games like tic-tac-toe, checkers, othello, and chess. These kinds of games are called games of perfect information because it is possible to see all possible moves. A game like scrabble is not a game of perfect information because there's no way to predict your opponent's moves because you can't see his hand.

You can think of the algorithm as similar to the human thought process of saying, "OK, if I make this move, then my opponent can only make two moves, and each of those would let me win. So this is the right move to make."

The rest of the article explains how to represent possible moves and evaluate them. An example command-line tic-tac-toe game using Ruby is given.

Representing Moves with the Game Tree Data Structure

Here's an example of a Game Tree for tic-tac-toe:



Note that this isn't a full game tree. A full game tree has hundreds of thousands of game states, so that's uh... not really possible to include in an image. Some of the discrepancies between the example above and a fully-drawn game tree include:

* The first Game State would show nine moves descending from it, one for each of the empty spaces on its board
* Similarly, the next level of Game States would show eight moves descending from them, and so on for each Game State

Representing the game as a game tree allows the computer to evaluate each of its current possible moves by determining whether it will ultimately result in a win or a loss. We'll get into how the computer determines this in the next section, Ranking. Before that, though, we need to clearly define the central concepts defining a Game Tree:

* The board state. In this case, where the X's and O's are.
* The current player - the player who will be making the next move.
* The next available moves. For humans, a move involves placing a game token. For the computer, it's a matter of selecting the next game state. As humans, we never say, "I've selected the next game state", but it's useful to think of it that way in order to understand the minimax algorithm.
* The game state - the grouping of the three previous concepts.

So, a Game Tree is a structure for organizing all possible (legal) game states by the moves which allow you to transition from one game state to the next. This structure is ideal for allowing the computer to evaluate which moves to make because, by traversing the game tree, a computer can easily "foresee" the outcome of a move and thus "decide" whether to take it.

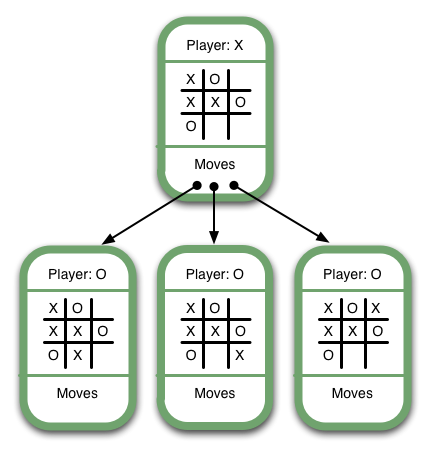
Next we'll go into detail about how to determine whether a move is good or bad.

Ranking Game States

The basic approach is to assign a numerical value to a move based on whether it will result in a win, draw, or loss. We'll begin illustrating this concept by showing how it applies to final game states, then show how to apply it to intermediate game states.

Final Game States

Have a look at this Game Tree:

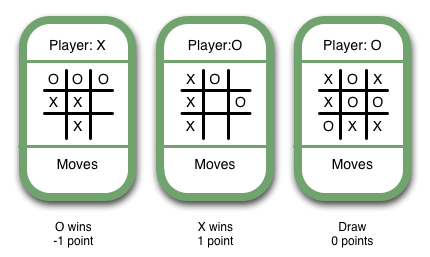


It's X's turn, and X has three possible moves, one of which (the middle one) will lead immediately to victory. It's obvious that an AI should select the winning move. The way we ensure this is to give each move a numerical value based on its board state. Let's use the following rankings:

* Win: 1
* Draw: 0
* Lose: -1

These rankings are arbitrary. What's important is that winning corresponds to the highest ranking, losing to the lowest, and a draw's between the two.

Since the lowest-ranked moves correspond with the worst outcomes and highest-ranked moves correspond with the best outcomes, we should choose the move with the highest value. This is the "max" part of "minimax". Below are some more examples of final game states and their numerical values:

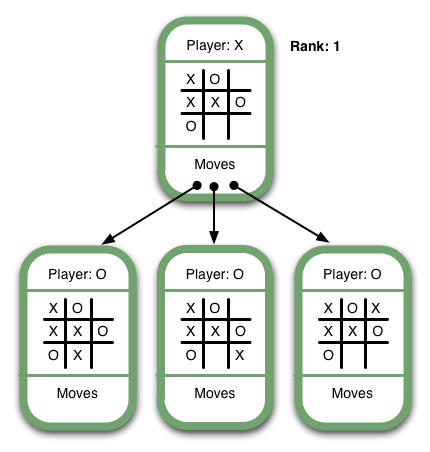


You might be wondering whether or not we should apply different rankings based on the player whose turn it is. For now, let's ignore the question entirely and only view things from X's perspective.

Of course, only the most boring game in the world would start out by presenting you with the options of "win immediately" and "don't win immediately." And an algorithm would be useless if it only worked in such a situation. But guess what! Minimax isn't a useless algorithm. Below I'll describe how to determine the ranks of intermediate Game States.

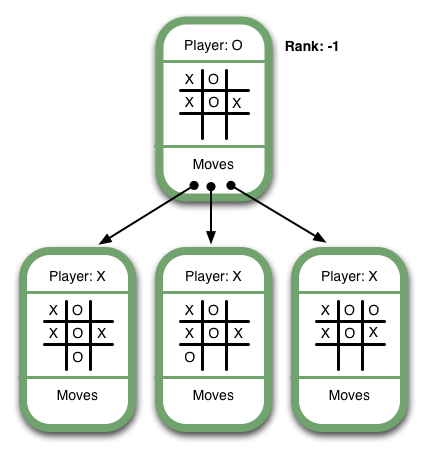
Intermediate Game States

Have another look at this game tree:



As you can see, it's X's turn in the top Game State. There are 3 possible moves, including a winning move. Since this Game State allows X to win, X should try to get it if possible. This Game State is as good as winning, so its rank should be 1. In general, we can say that the rank of an intermediate Game State where X is the current player should be set to the maximum rank of the available moves.

Now have a look at this game tree:



It's O's turn, and there are 5 possible moves, three of which are shown. One of the moves results in an immediate win for O. From X's perspective this Game State is equivalent to a loss, since it allows O to select a move that will cause X to lose. Therefore, its rank should be -1. In general, we can say that the rank of an intermediate Game State where O is the current player should be set to the *minimum* rank of the available moves. That's what the "mini" in "minimax" refers to.

By the way - the above game tree probably looks ridiculous to you. You might say, "Well of course you shouldn't make such a dumb move. Why would anyone give up a win and allow O to win?" Minimax is our way of giving the computer the ability to "know" that it's a dumb move, too.

To sum up:

* Final Game States are ranked according to whether they're a win, draw, or loss.
* Intermediate Game States are ranked according to whose turn it is and the available moves.
  + If it's X's turn, set the rank to that of the *maximum* move available. In other words, if a move will result in a win, X should take it.
  + If it's O's turn, set the rank to that of the *minimum* move available. In other words, If a move will result in a loss, X should avoid it.

And that's the minimax algorithm!

Ruby Example

Below is a Ruby implementation of Tic-Tac-Toe. It has a few limitations - for example, the computer always plays X. This is beacuse it's meant to illustrate the concepts described here.

To run it, copy it to something like "tictactoe.rb" and run ruby tictactoe.rb.

You can [view the repo on github](https://github.com/flyingmachine/minimax-tictactoe). In the future, I might add Common Lisp and C versions of Tic-Tac-Toe, and I'd welcome versions in other languages as well.

#!/usr/bin/env ruby

# This is a pretty naive implementation of Tic-Tac-Toe meant to

# demonstrate the minimax algorithm. Note that it takes awhile to

# start up - 30 seconds on a 2010 mac book air. Also note that the

# Ruby isn't all that great as Ruby, but I don't care as long as it

# illustrates the algorithm well.

#

# You can see that there's a hard-coded assumption that the AI is the

# X player, means that no human will ever be able to win. I'm positive

# that is only a small taste of what is to come, once robots take

# over the world.

class GameState

attr\_accessor :current\_player, :board, :moves, :rank

def initialize(current\_player, board)

self.current\_player = current\_player

self.board = board

self.moves = []

end

def rank

@rank ||= final\_state\_rank || intermediate\_state\_rank

end

# this is only ever called when it's the AI's (the X player) turn

def next\_move

moves.max{ |a, b| a.rank <=> b.rank }

end

def final\_state\_rank

if final\_state?

return 0 if draw?

winner == "X" ? 1 : -1

end

end

def final\_state?

winner || draw?

end

def draw?

board.compact.size == 9 && winner.nil?

end

def intermediate\_state\_rank

# recursion, baby

ranks = moves.collect{ |game\_state| game\_state.rank }

if current\_player == 'X'

ranks.max

else

ranks.min

end

end

def winner

@winner ||= [

# horizontal wins

[0, 1, 2],

[3, 4, 5],

[6, 7, 8],

# vertical wins

[0, 3, 6],

[1, 4, 7],

[2, 5, 8],

# diagonal wins

[0, 4, 8],

[6, 4, 2]

].collect { |positions|

( board[positions[0]] == board[positions[1]] &&

board[positions[1]] == board[positions[2]] &&

board[positions[0]] ) || nil

}.compact.first

end

end

class GameTree

def generate

initial\_game\_state = GameState.new('X', Array.new(9))

generate\_moves(initial\_game\_state)

initial\_game\_state

end

def generate\_moves(game\_state)

next\_player = (game\_state.current\_player == 'X' ? 'O' : 'X')

game\_state.board.each\_with\_index do |player\_at\_position, position|

unless player\_at\_position

next\_board = game\_state.board.dup

next\_board[position] = game\_state.current\_player

next\_game\_state = GameState.new(next\_player, next\_board)

game\_state.moves << next\_game\_state

generate\_moves(next\_game\_state)

end

end

end

end

class Game

def initialize

@game\_state = @initial\_game\_state = GameTree.new.generate

end

def turn

if @game\_state.final\_state?

describe\_final\_game\_state

puts "Play again? y/n"

answer = gets

if answer.downcase.strip == 'y'

@game\_state = @initial\_game\_state

turn

else

exit

end

end

if @game\_state.current\_player == 'X'

puts "\n==============="

@game\_state = @game\_state.next\_move

puts "X's move:"

render\_board

turn

else

get\_human\_move

puts "The result of your move:"

render\_board

puts ""

turn

end

end

def render\_board

output = ""

0.upto(8) do |position|

output << " #{@game\_state.board[position] || position} "

case position % 3

when 0, 1 then output << "|"

when 2 then output << "\n-----------\n" unless position == 8

end

end

puts output

end

def get\_human\_move

puts "Enter square # to place your 'O' in:"

position = gets

move = @game\_state.moves.find{ |game\_state| game\_state.board[position.to\_i] == 'O' }

if move

@game\_state = move

else

puts "That's not a valid move"

get\_human\_move

end

end

def describe\_final\_game\_state

if @game\_state.draw?

puts "It was a draw!"

elsif @game\_state.winner == 'X'

puts "X won!"

else

puts "O won!"

end

end

end

Game.new.turn

The End... or Is It???

I hope that explains the algorithm! If anything's unclear, please let me know in the comments below. I'd like for this post to explain the algorithm completely, and I'll use any feedback to revise it.

Also, you may have noticed that the algorithm, as written, requires a lot of memory. Tic-Tac-Toe, one of the simplest, most boring games in existence, requires hundreds of thousands of game states in the above, naive Ruby implementation. In the future, I may cover some optimization techniques, like alpha-beta programming. Look for it!