# Artificial Intelligence - Implementing Minimax with Prolog

## Mines Saint-Étienne - Toolbox ICM

### Table of Contents



## Introduction

The goal of this tutorial is to understand how to implement the `minimax` algorithm as described in AIMA. We take inspiration here from the algorithm provided by Bratko. The slides containing the pseudocode are available here.

## Minimax step-by-step

The `minimax` algorithm is recursive by nature. The terminal case (second `minimax` rule) is triggered when the analyzed state (`Pos`) has no successors. In this case, the evaluation of the state is the utility of this leaf. The recursive case (first `minimax` rule) is triggered when the analyzed state has successors. In this case, `minimax` generates the list[1] of all the successors obtained by applying all the possible moves from `Pos` (`NextPosList`), and looks for the best successors in this list (`BestNextPos`) with the best value (`Val`).

```
minimax(Pos, BestNextPos, Val) :-                     % Pos has successors
    bagof(NextPos, move(Pos, NextPos), NextPosList),
    best(NextPosList, BestNextPos, Val), !.

minimax(Pos, _, Val) :-                     % Pos has no successors
    utility(Pos, Val).
```

Notice that rules `utility` and `move` are problem-dependent, and will need to be defined when implementing a particular game. However, we can define the `best` predicate like this:

```
best([Pos], Pos, Val) :-                              % There is no more position to compare
    minimax(Pos, _, Val), !.

best([Pos1 | PosList], BestPos, BestVal) :-           % There are other positions to compare
    minimax(Pos1, _, Val1),
    best(PosList, Pos2, Val2),
    betterOf(Pos1, Val1, Pos2, Val2, BestPos, BestVal).
```

The first rule concerns the case when there is only one state (`Pos`) to compare: its value is its `minimax` evaluation. The second rule corresponds to the case when there are still states (`PosList`) to analyze after the current one (`Pos1`). In this case, `Pos1` is evaluated and compared (using the `betterOf` rule, see later) to the best one from the `PosList`. The best one from these two is returned.

To compare two states given their evaluation, we define the `betterOf` predicate, as follows:

```
betterOf(Pos0, Val0, _, Val1, Pos0, Val0) :-   % Pos0 better than Pos1
    min_to_move(Pos0),                         % MIN to move in Pos0
    Val0 > Val1, !.                            % MAX prefers the greater value

betterOf(Pos0, Val0, _, Val1, Pos0, Val0) :-   % Pos0 better than Pos1
    max_to_move(Pos0),                         % MAX to move in Pos0
    Val0 < Val1, !.                            % MIN prefers the lesser value

betterOf(_, _, Pos1, Val1, Pos1, Val1).        % Otherwise Pos1 better than Pos0
```

Now, we have a generic `minimax` engine (we can download it as a prolog module here) that can be used for developing any game, by providing the proper definitions for the following rules :

- `move(+Pos, -NextPos)` : states that `NextPos` is a legal move from `Pos`
- `utility(+Pos, -Val)` : states that `Pos` as a value equal to `Val`
- `min_to_move(+Pos)` : states that the current player in `Pos` is *min*
- `max_to_move(+Pos)` : states that the current player in `Pos` is *max*

## Tic-tac-toe instantiation

As aforementioned, if we want to develop an AI for playing a specific game, we need to define some predicates. We are interested here to develop the well-known *tic-tac-toe* game.

### Game engine

As to test your implementation, we provide here a game engine[2] that will drive the game between a human and the computer.

To use it, launch prolog, consult `tictactoc-game.pl` and type :

```
?- play.
```

You will be asked for interaction. Follow the game, until the end :

```
====================
= Prolog TicTacToe =
====================

Rem : x starts the game

Color for human player ? (x or o)
|: x.

    |   |
 -----------
    |   |
 -----------
    |   |


Next move ?
|: 1.

  x |   |
 -----------
    |   |
 -----------
    |   |

Computer play :

  x |   |
 -----------
    | o |
 -----------
    |   |

Next move ?
|: 2.

  x | x |
 -----------
    | o |
 -----------
    |   |

Computer play :

  x | x | o
 -----------
    | o |
 -----------
    |   |

Next move ?
|: 4.

  x | x | o
 -----------
  x | o |
 -----------
    |   |

Computer play :

  x | x | o
 -----------
  x | o |
 -----------
  o |   |

End of game : o win !

true.
```

To use this engine, we will develop the `tictactoe.pl` module defining all the required predicates.

### State representation

The engine relies on a specific state representation that we will use in our predicates.

We represent a game position by a list `[Player, State, Board]`, where

- `Player` is the next player to play
- `State` is equal to '`play`' if not final state, '`win`' if win or '`draw`' if draw
- `Board` is the actual board of the game

The board is represented by a list of 9 elements (the first 3 elements are the first line of the board, …). An empty case is represented by '`0`'. We choose `x` to be the MAX player and `o` the MIN player.

### Tic-tac-toe predicates

As aforementioned, we need to define `move/2`, `min_to_move/1`, `max_to_move/1`, `utility/2` (which are required by `minimax` module) and we also need to define `winPos/2` and `drawPos/2` that are used by the game engine.

`min_to_move/1` and `max_to_move/1` are easy to implement, because of our state representation (it only depends on the first element of the state, a.k.a. the current player) :

```
min_to_move([o, _, _]).

max_to_move([x, _, _]).
```

Defining utility is quite straightforward too since we are developing the entire game tree and evaluating only leafs (terminal states) :

```
utility([o, win, _], 1).       % Previous player (MAX) has win.
utility([x, win, _], -1).      % Previous player (MIN) has win.
utility([_, draw, _], 0).
```

Let's now define what are terminal states (win or draw positions). A draw position consists in a state in which the board is full, i.e. contains no `0` :

```
drawPos(_,Board) :-
    \+ member(0, Board).
```

A win position consists in a state where three `x` or three `o` are aligned (in row, in column or in diagonal) on the board :

```
winPos(P, [X1, X2, X3, X4, X5, X6, X7, X8, X9]) :-
    equal(X1, X2, X3, P) ;    % 1st line
    equal(X4, X5, X6, P) ;    % 2nd line
    equal(X7, X8, X9, P) ;    % 3rd line
    equal(X1, X4, X7, P) ;    % 1st col
    equal(X2, X5, X8, P) ;    % 2nd col
    equal(X3, X6, X9, P) ;    % 3rd col
    equal(X1, X5, X9, P) ;    % 1st diag
    equal(X3, X5, X7, P).     % 2nd diag

equal(X, X, X, X).
```

Now we have enough the material to develop the most difficult predicate, `move/3` :

```
move([X1, play, Board], [X2, win, NextBoard]) :-
    nextPlayer(X1, X2),
    move_aux(X1, Board, NextBoard),
    winPos(X1, NextBoard), !.

move([X1, play, Board], [X2, draw, NextBoard]) :-
    nextPlayer(X1, X2),
    move_aux(X1, Board, NextBoard),
    drawPos(X1,NextBoard), !.

move([X1, play, Board], [X2, play, NextBoard]) :-
    nextPlayer(X1, X2),
    move_aux(X1, Board, NextBoard).

move_aux(P, [0|Bs], [P|Bs]).

move_aux(P, [B|Bs], [B|B2s]) :-
    move_aux(P, Bs, B2s).
```

The three first rules correspond to the following cases:

1. a move from a `Board` to a terminal winning state for player `X1`
2. a move from a `Board` to a terminal draw state
3. a move from a `Board` to a non-terminal state

The `move_aux` rules only states `NextBoard` is `Board` with an empty case replaced by `Player` mark.

We are now done! The `minimax` engine can now work fine with all these rules defined.

You can download the `minimax.pl`, `tictactoe.pl` and `tictactoe-game.pl` files to play with the computer.

## Exercises

Let's now extend our game in different directions.

### Exercise 1

By extending the `minimax` engine we developed, implement the **alpha-beta** pruning technique presented in the course. Compare it with the classical `minimax`.

### Exercise 2

By extending the `minimax` engine we developed (or the `alphabeta` algorithm you developed), implement the **cutting off** technique presented in the course. Compare it with the classical `minimax` and `alphabeta`.

### Exercise 2

By taking inspiration from the tic-tac-toe game we developed, implement another two-player game of your choice.

## Footnotes:

1

the `bagof` rule is standard in SWIProlog: http://www.swi-prolog.org/pldoc/man?predicate=bagof/3

2

Inspired by http://www.montefiore.ulg.ac.be/~lens/prolog/tutorials/tictactoe.pl

---

Gauthier Picard