# Artificial Intelligence, Present and Future 2018/19
# Assignment: Game Playing in Prolog

Jake Palmer Jacques Fleuriot Alan Smaill

March 11, 2019

## Introduction

The practical assignment for students on the AIPF course involves using Prolog for representing and reasoning about game playing. You will be required to implement an extension of the minimax algorithm to deal with random branching – expectiminimax – and use it to solve a stochastic variation on Tic-tac-toe (aka Noughts and Crosses). To get started, download all of the Prolog files (into a single directory) from:

`http://www.inf.ed.ac.uk/teaching/courses/aipf/#coursework`

You may run `swipl ttt_engine.pl` then execute the command `start` to play an ordinary game of Tic-tac-toe. By the end of this assignment you will have an implementation which extends this for the stochastic version. The following files are the *only* files you will be modifying for this coursework:

- `expectiminimax.pl`
- `expectiminimax_tree.pl`
- `stoch_tictactoe.pl`

You should *not* modify any of the other files provided.

## Prolog

Prolog is a programming language borrowing its basic constructions from logic. As a programming language, it departs from the traditional imperative approach to programming – which reproduces at a higher level the underlying von Neumann architecture and low level engineering constraints of computers – and instead presents its programs as a series of true statements. Its programs are like a set of axioms, and its computations are constructive proofs of goal statements using the program.

Sufficiently nested expressions are elided when printing, which will make debugging predicates involving the board tricky or impossible. To make Prolog display the full contents of lists (and all highly nested expressions), execute the following two commands:

```
set_prolog_flag(answer_write_options,[max_depth(0)]).
set_prolog_flag(debugger_write_options,[max_depth(0)]).
```

Alternatively, you can press the `w` key during tracing or when presented with a solution (if Prolog does not immediately drop back to the prompt) to achieve the same thing.

## Essential Reading and Exercises

As you will be using Prolog, you will need to be familiar with the system before you start. You will find this assignment much easier if you have completed the formative exercises and asked questions about using Prolog before you start. It is recommended that you read the relatively short text *Learn Prolog Now!* located at:

http://www.learnprolognow.org/

You should refer to the Prolog exercises and external material on Prolog such as *Learn Prolog Now!* and `http://www.swi-prolog.org/` to discover some of the commands and techniques available for tracing, debugging, and testing.

You may use the full SWI Prolog library as installed on DICE. You should not use other people's existing implementations of predicates.

## Notes on Prolog Style

Marks will also be allocated for good Prolog style, including good explanatory comments above the predicates you add and modify. Exercise caution when using cuts. It is fine at times to prioritise declaratively written predicates over ones full of optimising cuts. Cuts are often essential for efficiency, but death by a thousand cuts can take on a new meaning in Prolog.

## Part 1: Extend Minimax with Chance Nodes [25%]

For the first part of this assignment, you will extend minimax to account for "move by nature" or chance nodes, over which neither the maximising nor minimising player have control but which both players know the probability of going down each branch. The expecti- prefix comes from the switch from merely maximising (or minimising) the final value achieved to maximising *expected value* (see lecture 7, *Adversarial Search*).

### Binary trees with chance *(5 marks)*

Before that, to be able to test your implementation, you will complete an implementation of a generator for binary game trees with chance. A partial solution has been provided, with explanatory comments, in `expectiminimax_tree.pl`. You should complete the following predicates:

1. `gen_tree/2`, the main predicate for generating the tree. **(4 marks)**
2. `chance_of/3`, a predicate which takes a position, a possible next position, and puts the probability of transition in the third argument. **(1 mark)**

An implementation for working with binary game trees *without* chance has been provided in the file `minimax_tree.pl`. You should **not** edit `minimax.pl` or `minimax_tree.pl`. Note the use of `mod`, which you can look up in the SWI Prolog documentation[1], to ensure the strategies alternate between `min` and `max`, ending with maximisation over the leaf nodes. Your chance-endowed game trees should alternate in much the same way, except with chance taking turns between them, i.e. working form the bottom it should go `max->chance->min->chance->max->...` and so on.

### Expectiminimax *(20 marks)*

Now you are ready to implement expectiminimax, which you can test using your implementation from the previous sub-part. A skeleton implementation is provided in `expectiminimax.pl`. Your tasks:

1. Implement `expectiminimax/3`, which takes a position and places in the second and third arguments the best next position and the associated *value* respectively. **(10 marks)**

[1] `http://www.swi-prolog.org/pldoc/man?function=mod%2f2`

2. Implement `expectedVal/2`, which takes a chance-governed position and returns the *expected value* associated with it. **(8 marks)**

3. Explain, in a comment above it, why `expectedVal` does not have an argument which represents the best next position. **(2 marks)**

*Implementation hint: these two predicates are mutually dependent.*

Add comments above each of these predicates explaining in procedural terms how the predicate works. See below for an example illustrating declarative and procedural descriptions using Prolog's `member` predicate. You may, of course, implement additional helper predicates; please add comments above these explaining their purpose and how they work as well. For your implementation you will need to make use of `chance_to_move/1` and `chance_of/3`. The former was provided for you and the latter you implemented in the previous sub-part, both in `expectiminimax_tree.pl`.

```
% Declaratively:
% X is a member of a list starting with X, and
% if X is a member of a list T, then it is a member of a list with T
% as a tail.
% Procedurally:
% Nothing needs to be done to compute whether X is a member
% of [X|_], otherwise
% to compute whether X is a member of the list see if it is a member
% of the tail.
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

## Part 2: Complete the Implementation of the Game *Stochastic Tic-tac-toe* [30%]

For the second part of this assignment, you will complete the implementation characterising a stochastic twist on Tic-tac-toe[2] in the file `stoch_tictactoe.pl`. This will provide expectiminimax with the predicates it needs to work with the game.

[2]Our stochastic variation on the traditional game is related to but distinct from *"Wild Tic-tac-toe"*: `https://en.wikipedia.org/wiki/Wild_Tic-tac-toe`

For both the normal and extended game each board's cells are named from left to right, top to bottom like so:

```
 1 | 2 | 3
-----------
 4 | 5 | 6
-----------
 7 | 8 | 9
```

Stochastic Tic-tac-toe is a twist on the traditional game where a *coin flip* determines whose marker you put down on each turn (or in other words who you're playing as). You still choose at the start which marker (`x` or `o`) you will be trying to win with throughout. So if you choose to play as `x` and the coin is flipped and you have to play as `o`, you will want to place the `o` in as *least good* a position as possible for `o`.

You should refer to the file `tictactoe.pl` to see what a similar implementation for the standard game of Tic-tac-toe looks like, as well as load up the file `ttt_engine.pl` to see what playing against an optimal player employing minimax is like. The engines implement the concrete game-playing, while the file you will be editing only does things like characterise legal moves.

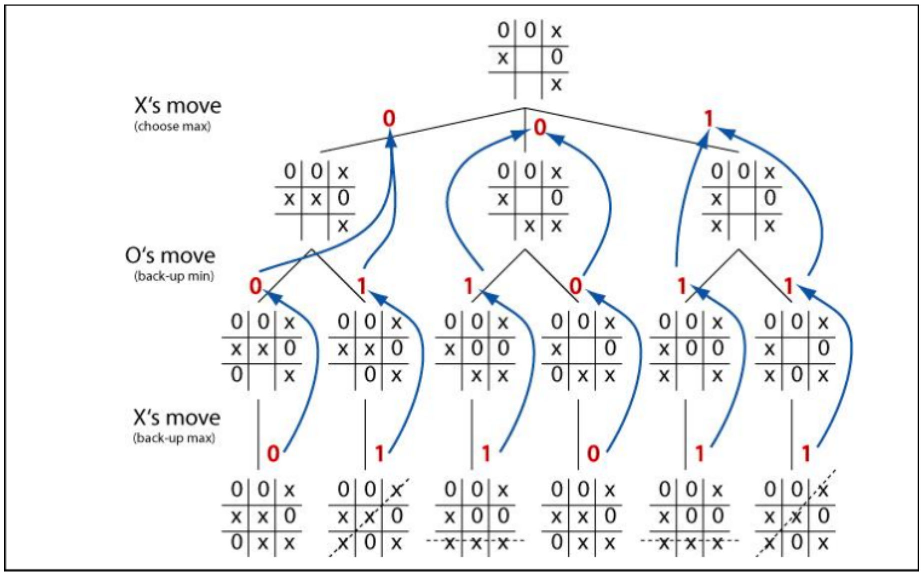

Figure 1: Partial minimax game tree for standard Tic-tac-toe.

You should **not** edit *any* of the following related files: `minimax.pl`, `tictactoe.pl`, `ttt_engine.pl`, `stoch_ttt_engine.pl`. Please make sure you understand Stochastic Tic-tac-toe before continuing. Do not change the representation of the game state or the board as the engine relies on it. Your tasks:

1. There are a few predicates in `tictactoe.pl` that take advantage of the predictability of the standard game in a few ways. Explain (in the indicated part of the file `stoch_tictactoe.pl`) what the assumptions being made are, and why they will not hold when we move to Stochastic Tic-tac-toe. They are:

   (a) The cut in `move` after `winPos`, which also affects the implementation of `drawPos` by allowing a simpler definition, and

   (b) the `utility` predicate. **(5 marks)**

2. Implement `move/2`, which characterises the legal moves available from the position supplied as the first argument. **(15 marks)**

3. Implement `strat_at/2`, which takes a position and puts the strategy for the player at that position in the second argument. **(1 mark)**

4. Implement `chance_to_move/1`, which is true if chance is to "move" at the given position. **(1 mark)**

5. Implement `chance_of/3`, which takes a position, a possible next position, and puts the probability of transition in the third argument. **(1 mark)**

6. Implement `utility/2`, which takes a leaf node position and returns the utility. **(7 marks)**

Note that the game state (aka "position") in the standard implementation (`tictactoe.pl` etc.) has the form `[Player, State, Board]`; in the stochastic extension (`stoch_tictactoe.pl` etc.) it has the form `[Player, PlayingAs, State, Board, BoardSize, InARowNeeded]`. The latter two do not change over the course of a game, and `State` is one of `win`, `draw`, or `play`. Again, see `tictactoe.pl` for examples applied to the standard game.

To test your implementation (and have fun with it) you can load up the provided `stoch_ttt_engine.pl`. This file will make use of your implementation of expectiminimax in the previous part and of your characterisation of Stochastic Tic-tac-toe in this part. You should *not* modify the engine. You call `start` to start the game, or `start(Seed)` for reproducible runs of the game. Here is an example run to compare your results to (note the fixed seed value):

```
?- start(98765).

==============================
= Prolog Stochastic TicTacToe =
==============================

Each turn, the player’s mark to
place will be decided by coin flip.

Marker for human player?
(x, o, or [w]atch)
|    w.

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



Coin flip! x will play o
Computer plays:
Running expectiminimax...
Move has value -0.2578125.
     |   |
  -----------
     |   |
  -----------
     | o |

Coin flip! o will play o
Computer plays:
Running expectiminimax...
Move has value -0.5234375.
     |   |
  -----------
     | o |
  -----------
     | o |

Coin flip! x will play x
Computer plays:
Running expectiminimax...
Move has value -0.234375.
     | x |
  -----------
     | o |
  -----------
     | o |

Coin flip! o will play o
Computer plays:
Running expectiminimax...
Move has value -0.5625.
     | x | o
  -----------
     | o |
  -----------
     | o |

Coin flip! x will play o
Computer plays:
Running expectiminimax...
Move has value -0.875.
     | x | o
  -----------
     | o |
  -----------
     | o | o

Coin flip! o will play o
Computer plays:
Running expectiminimax...
Move has value -1.
   o | x | o
  -----------
     | o |
  -----------
     | o | o

End of game: o wins!
```

If your implementation is taking a long time to run (e.g. more than 5 minutes), load up the engine using `swipl stoch_ttt_engine.pl`, take one of the later game states (called positions in the files) in the run above, and run the provided `test/3` predicate on it (which simply calls expectiminimax). For example, for the move from the board with three `o`'s:

```
test([x, o, play, [0,x,o, 0,o,0, 0,o,0], 3, 3], BestPos, Val).
```

where zeros denote empty cells on the board. The last two arguments to the game state is the board dimension (3x3) and the number in a row required for a win (which is 3). You can ignore the role of these game state/position "arguments" for now.

## Part 3: Implement a Cut-off Mechanism Using Heuristic Evaluation [45%]

For the third part of this assignment, you will add an additional argument `Cutoff` to `expectiminimax` and other necessary predicates so that when computing the best move at position `Pos`, calls to `expectiminimax` only go as deep as `Cutoff` levels. So, if `Cutoff = 6` expectiminimax should only look 6 moves ahead. Remember to take into account chance "moves" when setting the depth limit.

You will have probably felt the need for such a depth-limited search when running your experiments in the previous part. The slowness will only become more pronounced when we increase the size of the board, as we will do in this section. Everything should work for the larger board as before without modification, only now waiting for the best moves to be computed will be prohibitively long[3]. To work with a larger board, instead of simply calling `start` as in the previous section, you should now call `start_cutoff(4, 4, 6)`. This will initialise a board of size 4x4 which requires 4 of the same mark in a row for a win, i.e. the format is `start_cutoff(+BoardDimension, +InARow, +Cutoff)`. You may call `start_cutoff(+BoardDimension, +InARow, +Cutoff, +Seed)` to start the game with a fixed seed and hence a reproducible sequence of coin flips.

The cut-off point turns partially completed games into pseudo-leaf nodes. This means that you will need to add an additional predicate for evaluating these partially completed games. This predicate should ideally have at least the following properties:

[3]Even if you see potential optimisations to make running expectiminimax fast enough for a 4x4 board, please do **not** make modifications other than what is asked for here. Of course you may implement your own extensions and optimisations for fun outside of the context of this coursework.

- When `x` is certain to win, the evaluation should be $= 1$.
- When `o` is certain to win, the evaluation should be $= -1$.
- When `x` is more likely to win, the evaluation should be $> 0$ and $< 1$.
- When `o` is more likely to win, the evaluation should be $< 0$ and $> -1$.
- When `x` and `o` are certain to draw or equally likely to win, the evaluation should be $= 0$.
- Replacing every `x` on the board with an `o` and vice versa should negate the evaluation, e.g. 0.25 becomes -0.25.

Your tasks for this part are as follows:

1. In the indicated part of the file `stoch_tictactoe.pl`, describe and justify *two* different predicates for evaluating the utility of an unfinished game. Include comments on the strategies that your evaluation predicates encourage the computer player to take. The evaluation predicates do not need to be perfect, but they should be reasonable. **(10 marks)**.
2. Compare them with each other, make a claim about which you think is best, and justify that claim. **(5 marks)**
3. Implement and test your predicates by doing the following:
   (a) Extend `expectiminimax.pl` to include cut-offs. The engine expects the cutoff as the second argument to `expectiminimax`, which should now take four arguments in total. You should call a new predicate `eval/2` inside `expectiminimax` when the cut-off argument is 0. You will also need to pass the cut-off argument to other predicates. You will need to export the new `expectiminimax/4` at the top of the file. **(10 marks)**
   (b) Implement your two utility evaluation predicates in `stoch_tictactoe.pl`. When testing one or the other make sure the one being tested is named `eval` to link-in with the new `expectiminimax`. **(20 marks)**

You can assume the board dimensions are equal to the number of marks in a row needed for a win[4], and for this part you may wish to use similar predicates to those

[4]There are facilities for more general games (e.g. `winPosGen`) but we will not be using them in this assignment. You may of course play with this after submission.

in `winpos.pl` but remember that you should not modify this file. Finally, make sure to test your predicates on various board states (and sizes); you may use these tests as part of – but not as a substitute for – your justifications.

## Demonstrator Hours and Help

The demonstrator, Jake Palmer (s1673264@sms.ed.ac.uk), will be available to give advice on **Mondays, 9am-11am** in **6.06, Appleton Tower**.

You are strongly encouraged to make use of the Piazza forum for discussion of general problems and for sharing any queries that you may have.

Note that, although we encourage discussions about the assignment, you must not discuss or share **actual** solutions to any of the problems with fellow students.

## Submission

By **4pm, 15th March** you must submit all of your solution files in electronic form. These can be submitted (on DICE) using the following commands:

```
cd $HOME$/path/to/your/files
zip files.zip expectiminimax.pl expectiminimax_tree.pl stoch_tictactoe.pl
submit aipf cw1 files.zip
```

**Please ensure that your zip file contains all of your solution files.**

Late coursework will be penalised in accordance with the Informatics standard policy (see `http://edin.ac/1LRblYG`). Please consult your degree guide for specific information about this. Also note that, while we encourage students to discuss the practical among themselves, we take plagiarism **seriously** and any suspected case will be treated appropriately. Please remember the University requirements as regards all assessed work. Details about this can be found at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

Furthermore, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself).