University of Amsterdam

Encyclopedia of AI project

Tic-Tac-Toe

Authors:
Andreas van Cranenburgh
Ricus Smid

Supervisor:
Maarten van Someren

January 27, 2007

Encyclopedia of AI, assignment 5
Tic-tac-toe on fields of size n with m dimensions
–

0440949 Andreas van Cranenburgh
9660208 Ricus Smid
Fri Jan 26 16:03:15 CET 2007

Abstract

A Prolog implementation of the game of tictactoe, on fields of size n in m dimensions, using minimax.

1 Introduction

In the context of the first-year course Encyclopedia of AI, we have developed code to simulate playing the game of tic-tac-toe. This is to explore the use of mini-max and heuristics. In this paper some results will be presented and analyzed.

2 Implementation

With a 3x3 field, a very straightforward implementation can just store the state as a list of 9 elements, where each element is empty or contains either a naught or a cross. Then it follows that all the eight ways of having three-in-a-row can be checked, using pattern matching.

Obviously, this approach is less practical for 4x4, and even less so for 3x3x3. The number of rules to check for three-in-a-row with 3x3x3 would be 49. Also, to be able to play the game in different configurations without having to modify the program code, a different solution is called for.

Our code [1] stores the state as two lists, one for crosses and one for naughts. The lists consist of coordinates where a naught or cross has been placed. Each coordinate is a list, in turn, where the length depends on the number of dimensions currently in use. In two dimensions, a coordinate will thus contain X and Y, in three dimensions X, Y and Z, et cetera.

To generate a list of valid moves simply all possible coordinates are considered, and pruned by checking membership of the two lists in the state.

Checking if either player has won is doing by creating a list of all possible combinations of length n, where n is the size of the field, and checking every combination for characteristics of a three-in-a-row configuration. Before checking for these characteristics the coordinates are sorted, perhaps better described as aligned. Then each X coordinate is compared to the others, looking if they are all equal, or form a sequence (ie. 1,2,3 *or* 3,2,1). The same for each Y coordinate, and so forth till all coordinates have been compared.

Initially, a simple heuristic, employing a lookup table, was used. These pieces were then tied together with a standard minimax [3] implementation derived from [4]. A very simple user interface asks what size, dimensions, search depth and heuristic to use, and how many games to play. A so-called 'pretty print' procedure then outputs

all moves as they are being made. Unfortunately rendering a 3D representation of a 3x3x3 game was not feasible, so three flat representations will have to do.

3 The experiment

Currently our tic-tac-toe program can play the game in 2, 3 or 4 dimensions. The size of the field can be 3 or 4. Other sizes and dimensions could be added since the code is quite general, but this paper will be concerned with 3x3, 4x4, 3x3x3 and 4x4x4 only.

Another aspect is the heuristic. The reference heuristic is to assign each square a value based on the number of possible ways to win from that square, regardless of whether the other squares are already filled. This is heuristic 1. As a test, and also to be able to play on sizes and dimensions not supported by heuristic 1, there's a static heuristic which will always return 0, consequently called heuristic 0.

4 Results

dimension/size	depth/heuristic O - X	win O/X/draw	states O/X
2/3	2/1 - 2/1	0/0/5	825/600
	1/1 - 2/1	0/5/0	105/565
	1/1 - 1/1	0/5/0	105/90
2/4	2/1 - 2/1	0/0/5	4080/3400
	2/1 - 3/1	0/0/5	4080/34292
	2/1 - 4/1	0/0/5	4080/341680
	2/0 - 2/0	1/0/4	4053/3380
	2/0 - 3/0	2/0/3	
	2/0 - 4/0	1/1/3	4064/342473
3/3	2/1 - 2/1	5/0/0	12882/10280
	2/1 - 3/1	4/1/0	13211/224056
3/4	1/0 - 2/0	3/2/0	45651/196902
3/3 gravity	2/1 - 2/1	4/1/0	1988/1646
	2/1 - 3/1	2/3/0	1769/13206

5 Interpretation

5.1 Does it matter who plays first?

In three dimensions, the first player is the one who wins, given equal playing strength. In two dimensions there will be draw, given equal playing strength. These observations are confirmed by [2].

5.2 The effect of search depth

Naturally, the higher the search depth, the more states will be visited by the minimax algorithm. An upper bound (ie. worst case scenario) for the number of states that

will be visited, in Python code:

```
def maxstates(squares, depth):
#arguments: squares, the number of squares in the field/space
# depth, the search depth
#returns: upper bound of states that will be visited by minimax
    states = 0
    for a in range (squares, 0, -2): #eg. 9 7 5 3 1
        statesPerMove = 1
        for b in range(depth): #eg. 0 1 2 for depth=3
            statesPerMove *= (a - b)
        states += statesPerMove
    return states
```

So for example, if you play on a 3x3 field, and use search depth 2, the result would be:

$$9 * 8 + 7 * 6 + 5 * 4 + 3 * 2 = 140$$

But as said, this is only an upper bound. If one player wins, no further states will be made. Also, if you're the second player, you'll visit a little less states.

Whether it also gives any advantage to search deeper is a second question. From the results, it shows that on a 3x3 field, X wins if has more search depth. Yet with the 4x4 field, it doesn't matter, even when it searches 2 plies more than O, it's still draw! In three dimensions the difference is less clear. In 3x3x3, if X searches one ply deeper, it wins once out of five. A result which can probably only be explained by the fact that a random move is made if there are numerous moves with the same heuristic value.

5.3 Difference between 3x3 and 4x4

A peculiar difference shows between 3x3 and 4x4. With 3x3, if X searches deeper, it will defeat O. Yet with 4x4, O wins regardless of the search depth of X (at least with 2/3 and 2/4 – a search depth of 5 is currently beyond our patience). This might be explained by the fact that you have to get four-in-a-row with 4x4, which takes longer, and more importantly, gives more oppurtinity for the enemy to block you. That could explain the number of draws.

6 Other heuristics

The heuristic is the 'smart' part of the gameplay. It allows the computer to choose a move without searching the whole state space to the end. Currently the heuristic is not very good, and neither very flexible (it needs to be adapted to every size and dimension of playing field to be usable). It works by assigning a value to each location on the playing field, representing the number of possible wins in that location. So that means the middle square is always the most valuable, then the corners, then the rest. This does not take the already placed naughts and crosses into account. For

example putting your cross in a corner somewhere, even though it will be completely surrounded by naughts, still gives a high score by this heuristic.

A different heuristic might count the number of emtpy neighbor squares. After having implemented this heuristic, it turns out that this leads to the naughts and crosses being randomly scattered throughout the playing field, which is not really useful. A second version counts neighbor squares that are either empty or occupied by the player the heuristic is evaluated for. From preliminary experiments, it seems this heuristic is almost as good as the heuristic that assigns static values to locations – with the advantage that this one doesn't need a hard-coded lookup table. This heuristic is implemented as heuristic 2.

Another heuristic, heuristic 3, is based on the idea that the only way to win tic tac toe is to make a trap by making two rows of 2 naughts with no crosses in either row (in a game with size 3). This heuristic counts all the rows that has one naught and no cross and gives for every such a row a value of 1. It also counts the rows that have two naughts and no cross and gives those a value of 8. It will always choose an option with two rows with two naughts (the trap) above any other option except when there is a winning move or when a winning move has to be prevented.

Currently the heuristic only works for size 3 in 2 and 3 dimensions. It should however not be very difficult to make the code more generable. The current implementation performs better then the other heuristics we have implemented. It is however more computational expensive then heuristic 1.

7 Conclusion

Tic-tac-toe is a trivial game, a fact which can not be changed by playing it on more dimensions or a bigger playing field. Given a reasonable heuristic and sufficient search depth, minimax will converge to perfect play – that is, games result either in a draw or in the first player winning.

8 References

1. `https://unstable.nl/andreas/ai/encai/tictactoe.pl` - tested to work on SWI-Prolog

2. Weisstein, Eric W. "Tic-Tac-Toe." From MathWorld – A Wolfram Web Resource. `http://mathworld.wolfram.com/Tic-Tac-Toe.html`

3. See Wikipedia, Minimax, http://en.wikipedia.org/wiki/Minimax

4. Bratko, Ivan. "Prolog programming for AI", third edition; page 586.