University of Amsterdam

Encyclopedia of AI project

Tic-Tac-Toe

Tic-tac-toe on fields of size n with m dimensions
Comparisons and analysis of different sizes, dimensions, heuristics and search depths

Authors:
Andreas van Cranenburgh
Ricus Smid

Supervisor:
Maarten van Someren

January 31, 2007

Abstract

We made a Prolog implementation of the game of tictactoe, on fields of size n in m dimensions, using minimax. This paper shows measurements and presents two new heuristics.

Contents

1 Introduction **2**

2 The experiment **2**
2.1 Implementation 2
2.2 Heuristics 3

3 Analysis of results **4**
3.1 Search depth and number of states 4
3.2 Gametype: 3x3 5
3.2.1 Does it matter who plays first? 5
3.2.2 The effect of search depth 5
3.2.3 Number of states 5
3.2.4 Heuristics 6
3.3 Gametype: 3x3x3 without gravity 7
3.3.1 Does it matter who plays first? 7
3.3.2 Number of states 7
3.3.3 Heuristics 8
3.4 Gametype: 3x3x3 with gravity 8
3.4.1 Does it matter who plays first? 8
3.4.2 Number of states 9
3.4.3 Heuristics 9
3.5 Gametype: 4x4 10
3.5.1 Does it matter who plays first? 10
3.5.2 Number of states 10
3.5.3 Heuristics 10

4 Conclusion **10**

5 References **12**

6 Appendix **12**
6.1 Pseudocode for upper bound on number of states 12

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[1]. This is to explore the use of mini-max and heuristics. In this paper we will analyse what happens when search depth, heuristics, the size and the number of dimensions in the games are varied.

2 The experiment

Our experiment will be concerned with the following questions:

1. Does it matter who plays first?
2. What is the effect of search depth?
3. What is the difference between 2D and 3D games?
4. What is the difference between 3x3 and 4x4 games?
5. What is the effect of different heuristics on gameplay?
6. What is the effect of search depth on the number of states?

This paper evaluates these questions for 3x3, 4x4 and 3x3x3 games. The approach was to run a number of games, varying paramaters such as size, dimensions, heuristics and search depth. Time allowing, we tried to run as much games as possible, to make up for random variations introduced when different possible moves of equal value occur. The results consist of the number of games won, the amount of states visited by each player, and the time per game.

2.1 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 nought 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

[1] In British, the game is known as noughts and crosses.

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 Prolog code [1] stores the state as two lists, one for crosses and one for noughts. The lists consist of coordinates where a nought 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. For 3x3x3 games we added an option to simulate 'gravity', so that pieces will fall like in the game of four-in-a-row.

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.

2.2 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. The existing heuristic we used wasn't very good, and neither very flexible (it needed to be adapted to every size and dimension of playing field to be usable). It worked 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 noughts and crosses into account. For example putting your cross in a corner somewhere, even though it will be completely surrounded by noughts, still gives a high score by this heuristic.

We developed a different heuristic that counts the number of empty neighbor squares. After having implemented this heuristic, it turns out that this leads to the noughts 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 noughts with no crosses in either row (in a game with size 3). This heuristic counts all the rows that has one nought and no cross and gives for every such a row a value of 1. It also counts the rows that have two noughts and no cross and gives those a value of 8. It will always choose an option with two rows with two noughts (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.

3 Analysis of results

For detailed results, refer to the appendix. What follows are selected results that answer our research questions.

3.1 Search depth and number of states

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 can be found in the appendix, in pseudocode.

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. In the following paragraphs the upper bound will also be discussed seperately for each gametype.

3.2 Gametype: 3x3

In the original tic tac toe there can be no winner when the game is played properly[2]. The results show indeed that with the same depth (higher then 1) and a proper heuristic all the games end in a draw.

3.2.1 Does it matter who plays first?

With heuristic 0, that only checks for a winning state and is random otherwise, the results from tests with different combinations of depths, show that the starting player has a great advantage over the other player.

Heur o/x	Win % o	x	draw	States/game x	o	Time
0/0	52.2	19.4	21.3	369	236	0.07

Table 1: Results for gametype 3x3

3.2.2 The effect of search depth

Considering heuristic 1, the results show that when one player has a depth of 2 or higher and the other player has depth 1, the player with the higher depth will win, regardless of who starts the game. When both players have a depth of at least 2, the game will end in a draw.

There are some exceptions however. When a player uses search depth 3, somehow this is disadvantageous for the player. We currently have no explanation for this. We only found this effect in the game of 3x3.

Considering heuristic 3 the only game that didn't always end in a draw, was the game where the starting player had depth 2 and the other player a depth of 1. Apparently heuristic 3 plays this game smart, since smart play will always end in a draw.

3.2.3 Number of states

heur	depth	Win o	Win x	draw	States o	States x	Time/game
1/1	9/9	0	0	100	278190	28112	217.00
2/2	9/9	0	0	100	278254	28224	313.00
3/3	9/9	0	0	100	279198	32659	349.16

Table 2: Maximum number of states

We looked at the number of states with a number of games with search depths from 1 to 9; see table 2.

A search depth of 9 covers the whole state space, so you get the maximum number of states.

The upper bound for the number of states of tic tac toe is $(m^n)!$, with m = game size and n = dimensions. For a 3x3 game this is 9!

With a search depth of 9 (for the starting player) the number of states is given by:

$$9! + 7! + 5! + 3! + 1! = \ 9! = \ 37000$$

In practice, there a are a lot of forbidden states within this upper bound and the amount will be lower. The results show that the starting player only needs to make 28000 states to cover the whole game; see table 3.

The second player, whose upper bound is $8! = \ 40.000$, only needs to make around 30.000 states. So the number of states is around 75% of the upper bound for both players, when searching the whole game tree.

When the search depth is 1, the first player needs to make

$$9 + 7 + 5 + 3 + 1 = 25$$

...states for a full play. This is about the same as our results (some of the games didn't end up with a full board).

depth	states
1	23
2	162
3	929
4	4753
5	20900
6	75434
7	114914
8	214983
9	278538

Table 3: Number of states

3.2.4 Heuristics

We let the heuristics play against each other. Heuristic 3 performed best as is shown in the table. With a search depth of 4 against 1 it won all the games

Overall Heur 1-2:	h1	h2	draw
.	42.5	0	57.5
Overall Heur 2-3:	h2	h3	draw
.	0	46	54
Overall Heur 1-3:	h1	h3	draw
.	0	8.3	91.7

Table 4: Comparison of heuristics

against heuristic 1 even though heuristic 1 started the game. The other way around all the games ended in a draw.

Heuristic 2 performed worse than the other two. It lost all games against both heuristic 1 and 3 when playing both with a search depth of 4.

3.3 Gametype: 3x3x3 without gravity

When 3x3x3 is played well, the starting player should always be able to win in 4 moves.

3.3.1 Does it matter who plays first?

The starting player wins around 70 % of the games, which is no surprise. Heuristic 1 performs best with 85 % of wins for the starting player. Heuristic 3 performs badly when a player has a search depth of 1.

For any other search depth, the starting player, or the one with the higher depth wins all games, so the heuristic performs very well for search depths above 1.

3.3.2 Number of states

The upper bound for the number of states is:

$$(m^n)! = (3^3)! = 27!$$

The real amount might be 75% of this according to the number of states of a 3x3 game. The number in practice will always be a lot lower, since most games are won pretty quickly, and no game ends in a draw,

The amount of states created for depths from 1 to 3 is shown in the table below.

Depending on the length of the game, the upper amount of states created with a depth of 1 should be: $27 + 25 + 23 + 21 + ..$

depth	states
1	85
2	2542
3	43195

Table 5: Number of states

With depth 2: $27 * 26 + 25 * 24 + 23 * 22$ ...
and depth 3: $27 * 26 * 25 + 25 * 24 * 23 + 23 * 22 * 21$ ...

The mean amount of states made in our tests is of course lower than the upper bound, but still this amount can get pretty high pretty quickly with greater search depths.

3.3.3 Heuristics

Overall Heur 1-2:	h1	h2	draw
.	55	45	0
Overall Heur 2-3:	h2	h3	draw
.	50	50	0
Overall Heur 1-3:	h1	h3	draw
.	50	50	0

Table 6: Comparison of heuristics

Heuristic 2 performs slightly worse than the other heuristics. Heuristic 1 and 3 perform the same, but heuristic 3 is the only heuristic that won all games in 4 moves, where heuristic 1 needed sometimes more moves.

3.4 Gametype: 3x3x3 with gravity

The concept of gravity here refers to the idea that a nought or cross will fall to the ground, or on another nought or cross – similar to how the pieces fall in four-in-a-row. 3x3x3 With gravity is a more interesting game than 3x3x3 without gravity, since it is more difficult to win (ie. to make a trap).

3.4.1 Does it matter who plays first?

The starting player has, again, a pretty big advantage winning around 70% of the time in all variants.

Again, depth 1 doesn't work too well for heuristic 3. Still it has the highest amount of wins for the starting player. Above depth 1, all starting players or the player with the higher depth wins the game.

3.4.2 Number of states

The amount of possible states for the first move is the same as for 3x3, which is 9.

After the first move, still 9 options are available, since the other player can build on the first move.

The amount of possible moves goes down to 8 as soon as one vertical row is full. Our guess is that the bottom of the board will become full twice as fast as the second layer and the second layer twice as fast as the third layer.

So after 7 moves, 4 moves are made in the first layer, 2 in the second layer and 1 in the third layer. After 14 moves, two moves are made in the third layer. After that, the first layer is almost full and the other layers start getting full more quickly.

For a search depth of 1 the number of states made by the first player is approximated by:

$$9 + 9 + 9 + 9 + 8 + 8 + 8 + 7 + 7 + 6 + 6 + 5 + 3 + 1$$

... in case the game is played until the board is full.

But with rising search depths, every virtual move can be followed by another, so the complexity grows fast with growing depths, as we can see in the table. The values are pretty much the same as in 3x3x3 without gravity. But it should be taken into consideration that the games for that gametype were much shorter.

depth	states
1	80
2	2500
3	45000

Table 7: Number of states

3.4.3 Heuristics

For this gametype, heuristic 1 outperforms both other heuristics. When the depth is higher than 3 however, heuristic 3 outperforms heuristic 1, both in wins and in the amount of moves needed to win. (see table in appendix)

3.5 Gametype: 4x4

3.5.1 Does it matter who plays first?

The difference between the starting player and the other player is smaller than in the other games. The starting player wins around 24 % of the games, the other player around 14%. all the other games ended in a draw.

3.5.2 Number of states

The upper bound is:

$$(4^2)! \ = 2 * 10^{13}$$

Since most games end in a draw, the number of states is pretty high in practice.

For depth 1 the starting player should make around $16 + 14 + 12 + 10 + 8 + 6 + 4 + 2 = 72$ states.

The complexity seems to be approximately $70 * 10^n$ as can be seen in our results.

depth	states
1	68
2	750
3	7600
4	84000

Table 8: Number of states

3.5.3 Heuristics

Since heuristic 3 was only implemented in games with size 3, only heuristic 1 and 2 were tested (besides heuristic 0).

In this case, heuristic 2 performs the same as heuristic 1. A game can according to our tests only be won, when there is a big difference in search depth. Heuristic 2 managed to beat heuristic 1 in all games when it was the starting player and had a depth of 4 against 1.

4 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.

It can be concluded that the first player has a clear advantage, because each move will improve its position. Higher search depths improve the chances of winning, provided a proper heuristic is used. With sufficient search depths, in 3D games the first player will win; in 2D games, there will always be a draw. To win in a 4x4 game, the difference in search depth needs to be bigger than for 3x3 games. The average number of states per game as a function of the search depth is biggest in 3x3x3 games without gravity, and smallest in 3x3 games.

Heuristic 3 performs best (in gametypes it supports), provided the search depth is 2 or greater. Heuristic 2 works in all gametypes, but performs worse than 1 and 3, except in 4x4 games. Heuristic 1 performs better in terms of speed and efficiency.

5 References

1. `https://unstable.nl/andreas/ai/encai/tictactoe.pl` - tested to work on SWI-Prolog
2. 2006: Weisstein, Eric W. "Tic-Tac-Toe." From MathWorld – A Wolfram Web Resource. `http://mathworld.wolfram.com/Tic-Tac-Toe.html`
3. 2007: See Wikipedia, Minimax, http://en.wikipedia.org/wiki/Minimax
4. 2001: Bratko, Ivan. "Prolog programming for AI", third edition; page 586.

6 Appendix

6.1 Pseudocode for upper bound on number of states

```
MaxNumberOfStates(squares, depth):
    # squares: 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 legalMoves = squares to 0 step -2:   #eg. 9 7 5 3 1
        statesPerMove = 1
        for a = 0 to depth - 1:
            statesPerMove *= (legalMoves - a)
        states += statesPerMove
    return states
```