HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Artificial Intelligence

Breakthrough - Adversarial Search

September 13, 2017

Jón Steinn Elíasson

jonse07@ru.is

Contents

1 Introduction **2**

2 Board evaluation **2**
2.1 Terminal evaluation . . . 2
2.2 Player evaluation . . . 2
2.3 Heuristic . . . 4

3 Experiments **4**
3.1 Head to head . . . 4
3.2 Expansions . . . 5
3.2.1 No ordering . . . 5
3.2.2 Order of column position . . . 5
3.2.3 Order by heuristic . . . 6
3.2.4 Using transposition table . . . 6
3.3 Playing against GameController's random agents . . . 6
3.4 Results . . . 7

4 Bonus improvements **7**

A Output examples of a head to head test **8**

B Game play **9**

C JUnit tests **10**

List of Tables

1 Improved values for estimates . . . 5
2 No ordering . . . 5
3 Simple ordering (advanced first) . . . 5
4 Heuristic ordering . . . 6
5 caption . . . 6
6 Game results using GameController . . . 6

List of Figures

1 Count value evaluation . . . 2
2 Furthest value evaluation . . . 2
3 Can move value evaluation . . . 3
4 Unhindered value evaluation . . . 3
5 Lane control value evaluation . . . 3
6 Win next special case . . . 4
7 Win next for both players . . . 4
8 Playing as white against a random agent on a 3×5 board without a depth tax . . . 9
9 Playing as black against a random agent on a 3×5 board without a depth tax . . . 9
10 Playing as white against a random agent on a 3×5 board with a depth tax . . . 9
11 Playing as black against a random agent on a 3×5 board with a depth tax . . . 9
12 Code coverage . . . 10

Listings

1 Output of head to head experiment . . . 8

1 Introduction

This project is about implementing an agent to play a variation of Breakthrough. This version only allows diagonal kills which introduces the notion of draws. The board size is also not fixed.

A game state is represented by a set of positions for each color. Some other ways of representing the states such as bitboards were explored but seeing as our table size varies, the more simplistic approach of sets was used. A move is represented by a source and destination position, both a two element integer tuple. The agent searches, within the given time limit, until all leaves are terminal leaves.

A Negamax variant of alpha beta pruning was used, where (α, β) are passed down as $(-\beta, -\alpha)$ such that each player is always the maximizer when evaluating. He tries to maximize the value which negation is passed to the other player above in the tree.

2 Board evaluation

2.1 Terminal evaluation

Since white can't move into a black winning state and vice versa, there is no need to check if the current player has won. We only need to check if he has lost or if he has any moves.

For the white player we do so by iterating over the set of black positions and check if any has reached a goal position, if so a `LOST` value is returned. If not, we check if white has any possible movements and if so, return a `NON_TERMINAL` value. Otherwise a `DRAW` variable is returned. Evaluating terminal state for black works similarly. In any expansion, we check for these values to see if we should continue.

2.2 Player evaluation

Evaluating the board for each player depends on several variables.

i. **Pawn count**. The pawns of each player are counted and multiplied with a count value. In figure 1, the white player would get a score of `3 * count_value`, while the black would get `2 * count_value`.

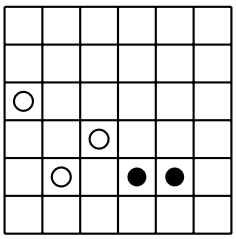

Figure 1: Count value evaluation

ii. **Furthest pawn**. The furthest pawn is multiplied with a furthest value. In figure 2, the white player has a score of `4 * furthest_value` while the black has a value of `5 * furthest_value`. The value of black is calculated by subtracting its minimum y position from the table's height plus one while the value for white is his maximum y position.

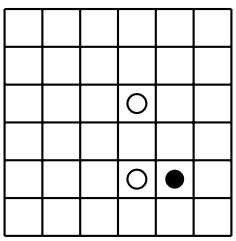

Figure 2: Furthest value evaluation

iii. **Movable pawns**. For each player, we count the number of pawns he can move and multiply it by a can move value. In figure 3, black can only move the upper left pawn while the white player can move 2, so there evaluation are `1 * can_move_value` and `2 * can_move_value` respectively.

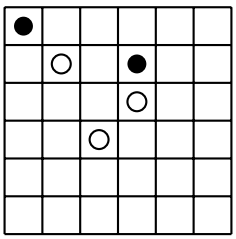

Figure 3: Can move value evaluation

iv. **Unhindered pawns**. Here we check whether a pawn has a clear path to the other end. This factor is multiplied with how far it is so the closer to goal the better. It checks each pawn current column (not those behind it) as well as both the columns to the left and right. In figure 4 we have 2 black pawns and 3 white. The black pawn at $(1, 5)$ does not have a clear path because of the white pawn at $(2, 1)$ and vice versa. The black pawn at $(4, 3)$ does have a clear path to the goal and he is on the forth row counting from his start so we would add 4 to a variable that is ultimately multiplied with the `unhindered_value`. The white pawns at $(4, 4)$ and $(6, 1)$ have a clear path so we would add 4 and 1 to this variable for white. In the end, this would evaluate to `4 * unhindered_value` and `(4+1) * unhindered_value` for the black and white players respectively.

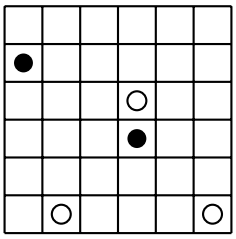

Figure 4: Unhindered value evaluation

v. **Lane control**. Here we check how many of the columns are occupied by our pawns. In figure 5 the black has pawns on 5 columns while white only at 4. This would evaluate to `5 * lane_control_value` and `4 * lane_control_value` for black and white respectively.

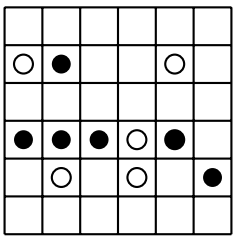

Figure 5: Lane control value evaluation

vi. **Win next**. Here we check if the players can win next turn. The value of this attribute is always greater then the maximum gained from the others combined but still less than a negative loss value (and $-\infty$). This takes into account who's turn it is so that the player who's turn it is dominates the other in case both players win next. The value is doubled for the player who's turn it is (in both cases this would dominate all other attributes so this is just for this single scenario). We also take into account that if a player who's turn it is not, has only a single move to a winning position and that being an attack move. If so, we do not look at it as a win next scenario since we assume the other person would kill his piece.

If however two such scenario were to arise on the table, the other player could only eliminate one. We must however check, in this case, if we only have two pawns attacking the same piece.

The special case is shown for black in figure 6. If we would not check for this scenario, our evaluation of the state would be that black wins next but in fact this is a winning position for white, starting by moving pawn $(3, 1)$ to $(3, 2)$ and then just advancing the right most pawn.

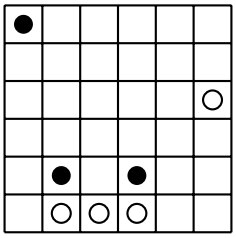

Figure 6: Win next special case

In the state shown in figure 7, both players are in a 'win next' position. If it would be white to play, we would evaluate his score as `2 * win_next_value` while only `1 * win_next_value` for the black.

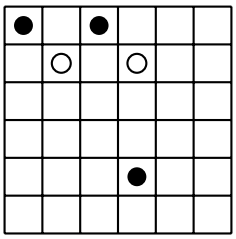

Figure 7: Win next for both players

2.3 Heuristic

To evaluate a state for white, we subtract the board evaluation for black if it's not his turn from the evaluation for white if it's his turn. For black, it would be the evaluation for white if not to play subtracted from the evaluation for black if it's his turn.

The values assigned to each part of the valuation was done by making two agent compete against each other and the one who lost changed one of his values. All values were kept in $[1, 500]$. More details about that can be found in subsection 3.1.

3 Experiments

3.1 Head to head

A static variable is used to store an instance of values for our heuristic and this variables is used by the agent. Initially all values were set to 1, then two agent would play against each other, one using the stored values and the other choosing one of it's values and increasing it by one. This should be repeated until increasing any value results in a loss.

A 8×8 board was used to play on and the time limit set to 1 second so that the agents would need to relay on their estimate. To make the experiment more reliable, a best out of three system was used as well as choosing color at random. These experiment took very long since the agents do not favor early terminals and in total, several hours went into this (although other work was done while they played out). At the time of the hand in, a search for such a set of values was still ongoing. The method for this experiment (`testSingleEstimate()`) can be found in the test class MyAgentTest. The resulting values the experiment yielded (so far) can be seen in table 1.

Variable	count	furthest	movable	unhindered	laneControl
Value	5	8	4	10	2

Table 1: Improved values for estimates

After finding values for our heuristic, another experiment was conducted where the agent played against another with random values. This was done on a 5×5 board with 1 second time limit and again, using random colors and a best out of three system. An output example can be seen in listing 1, appendix A. The output for the first experiment was identical (but a single entry). The latter experiment method (`testEstimates()`) can be found in the same class.

3.2 Expansions

The alpha beta search was run with no order, simple ordering (priority queue with most advanced pieces first), sorted by heuristic and than using a transposition table and heuristic ordering. All tests were done on a 8×8 board with a time limit of 10 seconds.

3.2.1 No ordering

Iteration	Expansions	Depth	Time
1	9	1	0
2	24	2	0
3	103	3	16
4	246	4	16
5	1210	5	31
6	4598	6	122
7	24368	7	288
8	90737	8	842
9	540912	9	2499

Table 2: No ordering

3.2.2 Order of column position

Iteration	Expansions	Depth	Time
1	9	1	88
2	24	2	15
3	110	3	12
4	379	4	18
5	2031	5	47
6	7148	6	104
7	37232	7	461
8	135318	8	2121
9	748961	9	3104

Table 3: Simple ordering (advanced first)

3.2.3 Order by heuristic

Iteration	Expansions	Depth	Time
1	6	1	115
2	16	2	5
3	58	3	13
4	168	4	23
5	997	5	54
6	2595	6	94
7	5630	7	182
8	23386	8	558
9	44115	9	1548
10	234104	10	1657
11	602457	11	3597

Table 4: Heuristic ordering

3.2.4 Using transposition table

Iteration	Expansions	Depth	Time
1	6	1	114
2	16	2	6
3	70	3	23
4	131	4	13
5	561	5	31
6	1552	6	83
7	2479	7	111
8	6200	8	242
9	16082	9	363
10	88619	10	2228
11	269017	11	2242

Table 5: caption

3.3 Playing against GameController's random agents

Games	Color	Map	Wins	Draws	Losses
5	White	3×5	5	0	0
5	Black	3×5	5	0	0
3	White	5×5	3	0	0
3	Black	5×5	3	0	0
2	White	6×6	2	0	0
2	Black	6×6	2	0	0
1	White	7×7	1	0	0
1	Black	7×7	1	0	0
1	White	8×8	1	0	0
1	Black	8×8	1	0	0
1	White	9×9	1	0	0
1	Black	9×9	1	0	0

Table 6: Game results using GameController

3.4 Results

Its unlikely that our experiment of determining which values to use in our evaluation is great. If a best choice does exists, it's not in $\mathbb{Z}_n^5$ (the set of 5 positive integers modulo n) for $n \in \mathbb{N}$ and even if it were, we do assume that winning a best out of three makes the heuristic better in general which is very unlikely. It is still better than a personal choice and at least, has something to back it up. A nice approach would be to make the agent reconfigure the relation between values dynamically.

As we can see, the agent does get deeper into the tree using the heuristic ordering since it will cut off more subtrees. It does expand less states per second since it takes some time to calculate the heuristic (although it should be fairly fast). We will go with more pruning over more state expansions as getting deeper into the tree should be more important and therefore The final version uses the heuristic ordering.

It's hard to say how useful the transposition table is by one search since it will re-use it's entries again in the next. Even so, we can see that we have a lot less expansion. Since the total time of the depth we managed to finish is a lot less than otherwise, we assume that we are closer to reaching depth 12 and benefit from using considerably from transposition table.

Playing against a random agent is mostly about heaving a good enough heuristic not to arrive at an impossible position in the end game. There is of course a change that he plays perfectly but almost always some mistakes are made and our agent should be able to use that to his advantage. One should not read to much into games against random agents.

4 Bonus improvements

The agent uses a priority queue to order which states to expand next. The comparator used compares the negative heuristic values since Java's priority queue has an ascending order.

A data structure for transposition table was implemented but wasn't completed until a few hours away from the deadline so there was some hesitation including it. It is set to contain a maximum of 100000 entries. The agent did however start to lose more often when using it so there is probably some bug in it. For that reason and lack of time, it was omitted.

A Output examples of a head to head test

An experiment output for testing our evaluation values against random values.

```
AGENT 1 WINS! (Black)
Win:
count: 5
furthest: 8
movable: 4
unhindered: 10
lane control: 2
nextWin: 491
--------------------
Lost:
count: 26
furthest: 99
movable: 131
unhindered: 441
lane control: 303
nextWin: 19016

AGENT 1 WINS! (Black)
Win:
count: 5
furthest: 8
movable: 4
unhindered: 10
lane control: 2
nextWin: 491
--------------------
Lost:
count: 424
furthest: 394
movable: 473
unhindered: 75
lane control: 423
nextWin: 15681

AGENT 2 WINS! (White)
Win:
count: 38
furthest: 194
movable: 351
unhindered: 440
lane control: 117
nextWin: 20846
--------------------
Lost:
count: 5
furthest: 8
movable: 4
unhindered: 10
lane control: 2
nextWin: 491

AGENT 1 WINS! (White)
Win:
count: 5
furthest: 8
movable: 4
unhindered: 10
lane control: 2
nextWin: 491
--------------------
Lost:
count: 54
furthest: 178
movable: 433
unhindered: 11
lane control: 376
nextWin: 8026
```

Listing 1: Output of head to head experiment

B Game play

Figures 8 and 9 show a game where the agent does not value winning early over winning late, while figures 10 and 11 show a game where the agent uses a 'depth tax' by multiplying the value passed up with 0.99. It seemed that changing all values to double and putting a depth tax resulting in a shallower search (by 2 levels) so this approach was canceled.

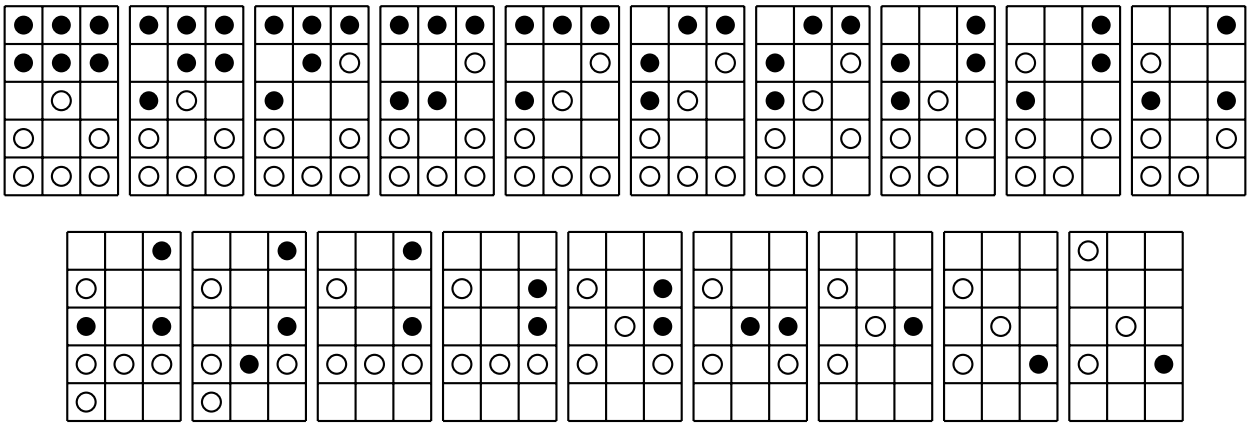

Figure 8: Playing as white against a random agent on a 3×5 board without a depth tax

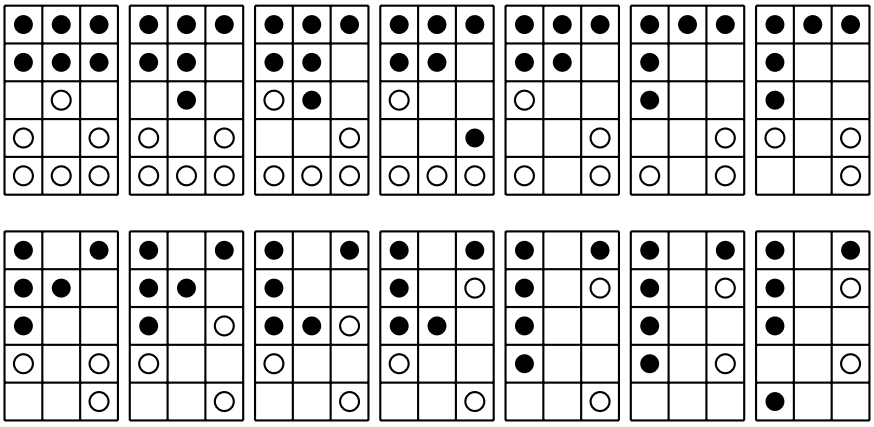

Figure 9: Playing as black against a random agent on a 3×5 board without a depth tax

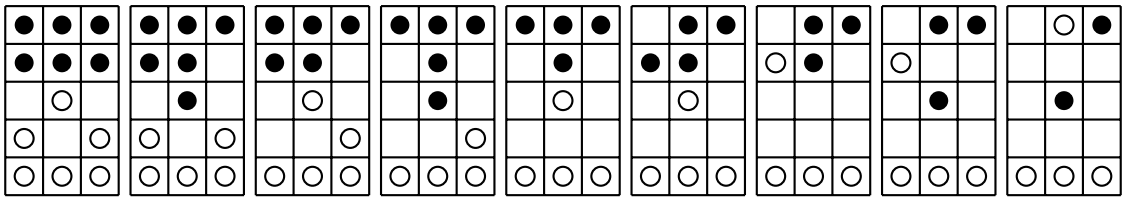

Figure 10: Playing as white against a random agent on a 3×5 board with a depth tax

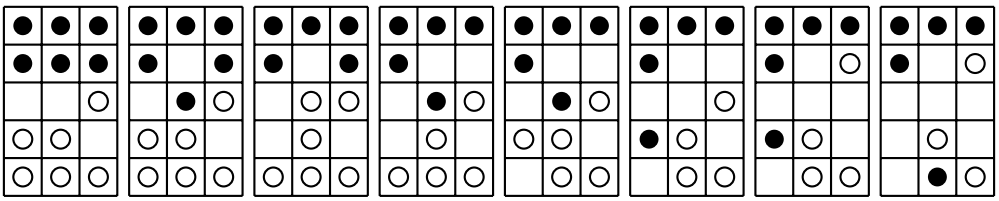

Figure 11: Playing as black against a random agent on a 3×5 board with a depth tax

C JUnit tests

The code is thoroughly tested with 100% coverage for everything not included in the project originally. A total of 77 tests although few of which are experiments rather than test. Testing ranges from testing single methods to playing entire games. As would be expected with such tests, they can take some time.

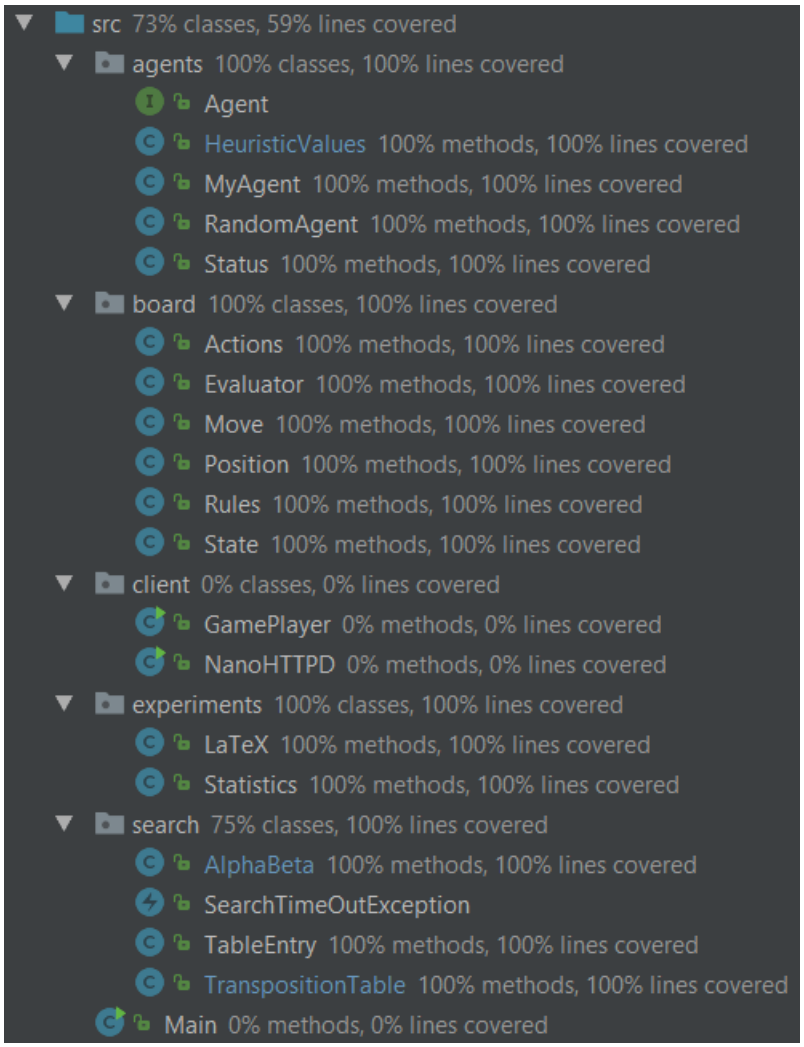

Figure 12: Code coverage

The following lists the highlights of the tests.

- Play as white against a random agent 10 times on a 3×5 board and checking if score is at least 8 (getting 1 for a win, 0 for draw, -1 for a loss). Another test does the same for black.
- Play as white against a random agent 3 times on a 9×9 board and checking if we win all. Another test does the same for black.
- Playing until a game is over as a random agent and check if we return valid commands.
- Symmetry test for playing as white (white starts) and black (black starts) for two states that are a reflection of each other (about $y = h/2$ or the diagonal), testing heuristics, best move, etc.
- Testing special case for 'win next' as mentioned in subsection 2.2.
- Testing if search passes time limit for multiple boards.
- If various obvious best moves are chosen.
- All collision cases for transposition tables.