Using Minimax with Alpha-Beta pruning to play Quarto

An IT3105 Project

Sondre Engebråten and Kristoffer Hagen

Index

1. The State evaluation function

The state evaluation function is a crucial part of the minmax algorithm, it enables the agent to differentiate between states that are not final. This happens when the depth of the minmax tree reaches it maximum limit. The function simply takes in a BoardState and then returns a value that represents the utility, or value, of this state. Having a good state evaluation function is what will separate a great minmax agent from a mediocre one.

1.1 How we handle final states

Our state evaluation function is split up into two parts. First, in the BaseMinMax algorithm itself will check whether the game is over or not in its current state. If the game is over; a 0 will be returned if its a draw, 1000 will be returned if we are in a max-node (we win) and -1000 if we are in a min-node (opponent wins).

1.2 How we handle intermediate states

For any intermediate state, this is when we get to the bottom of the search-tree and the algorithms has not found a final (win/draw/loss) state. The current BoardState is sent to an evaluation method that calculates the value of the this state.

1.3 How it works

There are ten potential winning lines, the methods counts up the pieces that are occupying any these positions. Then it calculates how many (out of the eight) attributes that all the pieces in the current line has in common. This we call the numberOfFeatureMatches.

After the pieces on the board has been categorized we look at the pieces yet to be placed. We count how many of the pieces still available that also share the feature(s) that all the pieces in the winning lines have in common.

We then return a value for each of the ten winning lines based on how many features the pieces of that line have in common and how many of the unused pieces can be used to complete this line.

A winning row (that still has the potential to end the game and has not yet ended the game), can have 1,2 or 3 pieces in it, seen on the y-axis of the table. These pieces can share 0,1,2 or 3 features. The “# pieces” in the tables represents how many different pieces out of the available ones that can be used to complete the line.

Value of a line:

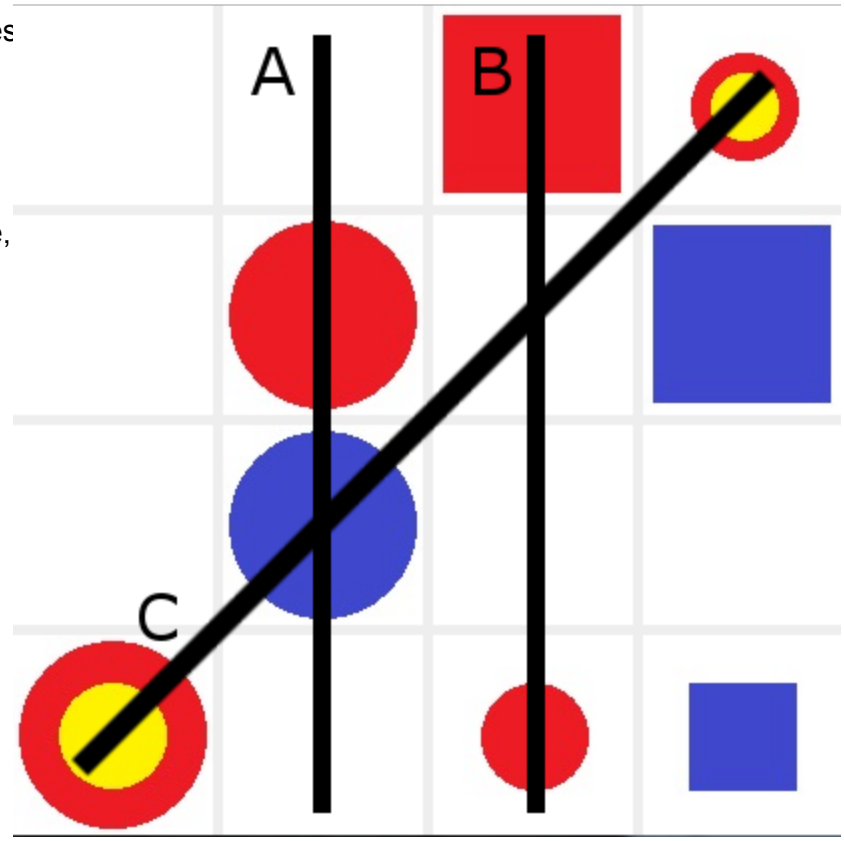
Pieces \ Features	1	2	3
1	1	-	-
2	10 + # pieces	20 + 2* # pieces	30 + 4* # pieces
3	100 + 2* # pieces	200 + 4* # pieces	-

Here is an example with three lines evaluated:

Line A:
Has 2 pieces in the line and 3 features in common (Large, Circle, No-Hole)

Line B:
Has 2 pieces in the line and 2 features in common (Red, No-Hole)

Line C:
Has 3 pieces in the line and one feature in common (Circle)



Looking at Line C. There are 3 pieces in the line, they all share one attribute and out of the remaining pieces, 3 will share that attribute. That will give this line a score of 100 + 2*3 = 106.

All ten of the winning lines will be evaluated and the sum will be returned. This sum represents the value of the board.

1.4 Potential value of a open slot

In addition to we have another method that takes every open slot on the board and figures out how many potential winner lines this slot is a member of. The value of each slot is can be seen in the figure below:

3	2	2	3
2	3	3	2
2	3	3	2
3	2	2	3

This method gives a value, depending on the BoardState of how many winning lines the player can manipulate. We consider it to be of higher utility if more of the slots that are a part of 3 winning lines are open.

1.5 The end of the evaluator

We sum up the value of the open slots and add this to the value of the board. After this we make sure that the value isn't higher than that of a win or a loss and it is returned to the MinMax algorithm.

2. Random AI versus Novice AI

A table showing the results of 100 runs of our system in which the novice agent plays against the random AI.

100 games was played, the results are:

players.ai.RandomAI won:	2	(2%)
players.ai.NoviceAI won:	96	(96%)
Draws:	2	(2%)

Since this was a rather fast simulation we also ran it with 10000 games.

10000 games was played, the results are:

players.ai.RandomAI won:	46	(0%)
players.ai.NoviceAI won:	9882	(99%)
Draws:	72	(1%)

3. MinMax(3) AI versus Novice AI

A table showing the results of 20 runs of our system in which the novice agent plays against the minimax-3 agent.

20 games was played, the results are:

players.ai.minmax.MinMaxAI(3) won:	13	(65%)
players.ai.NoviceAI won:	2	(10%)
Draws:	5	(25%)

4. MinMax(3) AI versus MinMax(4) AI

A table showing the results of 20 runs of our system in which a minimax-3 agent plays against a minimax-4 agent.

20 games was played, the results are:

players.ai.minmax.MinMaxAI(3) won:	0	(0%)
players.ai.minmax.MinMaxAI(4) won:	20	(100%)
Draws:	0	(0%)

5. The Tournament

How to read the results:
On the y-axis you can find the groups, on the x-axis the groups they played against. In parenthesis the amount of draws in the matchup. The percentage shown is the amount of matches won.

5.1 The Results

MinMax 2 ply, 200 games:

Losses Wins	Sondre / Kristoffer	Sindre	Bart	Marien
Sondre / Kristoffer	**x**	**192 (0) 96%**	**111 (5) 56%**	**183 (2) 92%**
Sindre	**8 (0) 4%**	**x**	**30 (0) 15%**	**36 (0) 18%**
Bart	**84 (5) 42%**	**170 (0) 85%**	**x**	**128 (26) 64%**
Marien	**15 (2) 8%**	**159 (0) 80%**	**46 (26) 23%**	**x**

MinMax 3 ply, 20 games:

Losses Wins	Sondre / Kristoffer	Sindre	Bart	Marien
Sondre / Kristoffer	**x**	**4 (15) 20%**	**15 (4) 80%**	**13 (5) 70%**
Sindre	**1 (15) 5%**	**x**	**3 (5) 15%**	**10 (2) 50%**
Bart	**1 (4) 5%**	**12 (5) 60%**	**x**	**18 (2) 90%**
Marien	**2 (5) 10%**	**8 (2) 40%**	**0 (2) 0%**	**x**

5.2 Our experience with the tournament

5.2.1 Infrastructure

We quite early got in touch with another student who had written code for a server and a protocol for communication. This was very helpful as it enables many of the groups to finish all their coding before meeting up and running the tournament. The server-code had support for running the tournament over a network but after various IP-issues and firewall problems we decided to export our code into executables and run it locally instead.

5.2.2 The tournament, take one

The first time we ran our tournament we decided to run one thousand games with our minmax-2 playing against each other and then another 20 games where minmax-3 plays. This turned out to be very time consuming. Some of the longest runs lasted for over an hour. 3 out of the 4 groups completed the tournament while the last one had to do some last-minute coding.

We got all our results done but we later found out that there was a bug somewhere in the server-code and the server was unable to generate draws. This was quite devastating after hours of running the tournament. Even so, after fixing the bug we decided to run it again. Mostly because we just really wanted to know if anything would change. We decided to cut down the games to 200 and 20 to make it execute faster.

5.2.3 The tournament, take two

We ran the tournament a second time with the fixed server code. It turned out that results from the first tournament didn't change a lot when it came to our 2-ply games. However when we ran our 3-ply games again we came out with many more draws. We were really happy with this change as it is a sign that our and the rest of the groups agents are performing how they should. With sufficient lookahead draws should be common. Given perfect play every game will be a draw.

5.2.4 Conclusion

Programming and participating in the tournament was a blast. We might be slightly competative of nature but the possibility of measuring the performance of our AI to that of other students was really enjoyable and inspired us to create better agents. We happen to achieve really good results as well but the best part about it was sitting down with the other students. Talking about our solutions and problems and just running our agents against each other was a lot of fun.