Quarto

Tobias Dreher, Dmitriy Zharkov

Oktober 2012

Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering
Norwegian University of Science and Technology

Contents

1 Player Strategies

In this section the players' strategies are briefly described. For the whole document there are some definitions to be explained:

- **Make move:** Placing a given piece on a cell of the board
- **Select piece:** Choosing a piece for the opponent
- **Perform action:** Making move first, then choosing a piece

1.1 Random Player

The most trivial player. Performs his actions randomly with only one restriction: He can't make a move on a taken cell and he can't select a piece that has already been chosen.

1.2 Novice Player

Acts the same way as Random Player does, bit in cases, where he can make a move in order to win, he will make this move. The same applies to the chosen a piece, as he won't select a piece for his opponent that might make the opponent win with only one move. If there are no such pieces left, he chooses a piece randomly.

1.3 Monte Carlo Player

Every time this player has to perform an action, he simulates hundreds of games for each possible action he can perform. He chooses the action which has led to the most wins during the simulations. Usually, Monte Carlo simulations are totally random, but we used the Novice Player for the simulations instead. To speed the simulations up, they are performed parallely.

1.4 MinMax-D Player

The MinMax-D Player looks D actions into future. One action means that only one player selects a piece and makes a move, but not both of them. So for $D = 3$ the player simulates all possible actions he can perform, all possible counteractions to these actions and all possible counteractions to the counteractions. We use α-β-pruning in order to reduce the number of nodes being evaluated.

We use the Monte Carlo strategy before the seventh piece is set on the board.

During the search, if the MinMax player comes upon a state, where the game is either won, lost or tied it's easy to evaluate such a state. But when a leaf in the search tree is a intermediate state, it has to be evaluated otherwise.

1.4.1 Evaluation functions

In order to test the further evaluation functions we used a trivial function that always return 0 regardless of the state of the board. A MinMax-D Player with such an evaluation function behaves the like a Novice Player with D actions foresight.

Almost Completed Attributes

This evaluation function determines how many attributes there are which are almost completed. For an almost completed attribute there must be a row with three pieces which have this attribute in common. Consider figure 1.1. Here, there are three almost completed attributes. The pieces in the row share the attribute *BIG* and *SOLID*, the pieces in the column are *BLUE* and the diagonal consits of *CIRCLULAR* pieces.

The evaluation function counts all almost completed attributes and returns this number.

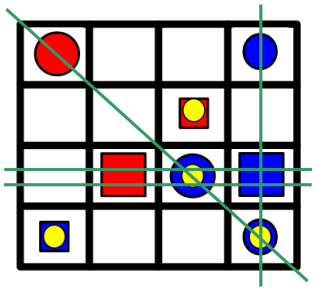

Figure 1.1: A board with four almost completed attributes

Completing Pieces: Absolute

In the previous evaluation function the remaining pieces aren't taken into account. Now, we want to correct this. The reason why this is important is shown in figure 1.2. Here, there are two almost completed attributes (*BIG*) but no pieces left to complete these attributes. Therefore, these attributes don't give any advantage to the player.

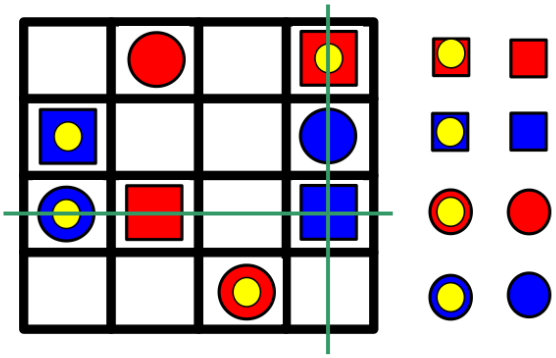

Figure 1.2: A board with two almost completed attributes but no pieces to complete these attributes

Similarly to the previous evaluation function we determine which attributes are almost completed. But now we also iterate through the remaining pieces and count the pieces which can complete one or more of the attributes. If a piece completes more than one attribute, it is counted more than once. The figure 1.3 show a board with 15 completing pieces. There are 4 pieces to complete the attribute *BIG*, 4 pieces to complete *SOLID*, 3 pieces to complete *BLUE* and 4 pieces for *CIRCULAR*.

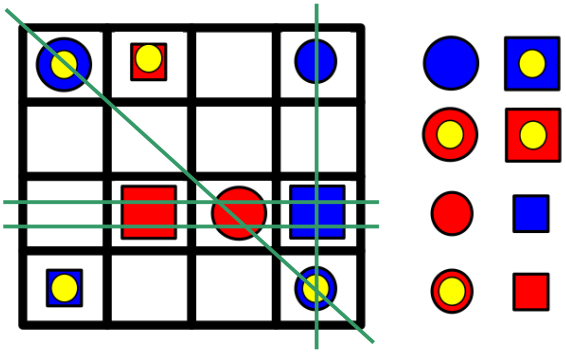

Figure 1.3: A board with three almost completed attributes and several pieces which complete an attribute

Completing Pieces: Relative

In the previous evaluation function the pieces which complete an attribute are taken into account, but the pieces which don't complete anything are left out. It is interesting to see if these pieces would improve the estimation of a board state.

Therefore, we don't only determine the number of completing pieces but also set this number in relation to the amount of remaining pieces. The value h returned by the evaluation function is shown in equation 1.1. C is a constant stretching factor.

$$h = \frac{\#\ completing\ pieces}{\#\ remaining\ pieces} * C \tag{1.1}$$

Figure 1.4 illustrates the implications that the equation 1.1 has. The evaluation function returns $h = 0.27$ for the left and $h = 0.5$ for the right board. That means that the right board is considered more desirable. The reason for this is, that the evaluation happens at a moment when the opponent has made a move. In the next step the opponent selects a piece for us. The right state in figure 1.4 has less pieces that can't complete a row, which is good for us.

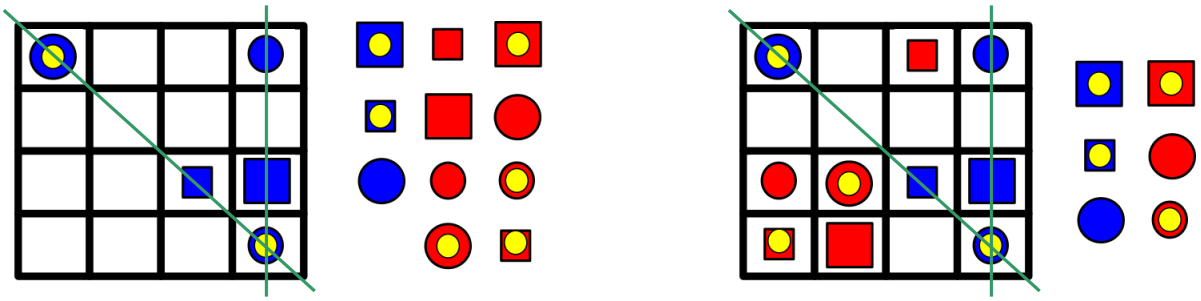

Figure 1.4: Comparison of two board states. The right board has a higher value h returned by the evaluation function

Conclusion

To test the introduced heuristics we let them play against the trivial evaluation function which always returns 0. All of the heuristics described above were better then the trivial one which means that they indeed help to estimate a boards state and improve the performance of the MinMax Player.

A competition between the evaluation functions where all heuristics had to compete against each other has shown that the absolute completing pieces evaluation is the best. Therefore, the MinMax Player described in Chapter 2 and 3 uses this evaluation function.

2 Player Evaluations

To evaluate the player strategies we simulated 1000 games per test case. The Monte Carlo Player performs 1000 simulations per move and piece.

Random Player vs. Novice Player

Player	Wins	Ties
Random Player	21	3
Novice Player	976	

Novice Player vs. Monte Carlo Player

Player	Wins	Ties
Novice Player	99	6
Monte Carlo Player	895	

Novice Player vs. MinMax-3 Player

Player	Wins	Ties
Novice Player	34	14
MinMax-3 Player	952	

Monte Carlo Player vs. MinMax-3 Player

Player	Wins	Ties
Monte Carlo Player	215	139
MinMax-3 Player	646	

MinMax-3 Player vs. MinMax-4 Player

Player	Wins	Ties
MinMax-3 Player	78	476
MinMax-4 Player	446	

3 Tournament

3.1 Performance

The tournament conists of five groups competing 950 times against each other. Every group uses a MinMax-3 Player with their own evaluation function.

Player	**Wins**
Annika & Tim	2429
Nils	2218
Dmitriy & Tobias	2117
Alessio & Vincent	1370
Magnus	16

Dmitriy & Tobias vs.	**Wins**	**Losses**	**Ties**
Annika & Tim	272	400	278
Nils	223	240	487
Alessio & Vincent	682	153	115
Magnus	942	4	4

3.2 Experiences

Right after the first lecture we formed a tournament group of 5 teams with 8 students from 4 countries in total. The first topic to discuss was the interaction between the programs of the teams. We came up with two approaches. The first one was to keep the source code of all teams in one Java project. This would be very easy to implement but there was a team which was uncertain about the language they will use for the project. Furthermore the source code of each team would be visible to all of the other teams.

The solution we finally chose was to interact between the programms per Std-In/Std-Out. This approach doesn't force the teams to use the same language but needs an additional program (GameMaster) that synchronizes the communication between the teams' programs.

Unfortunately it turned out that this approach slows down the execution time significantly, because Std-In/Std-Out communication seems to be quite slow.

Implementing the GameMaster was no big deal but it took some time for the teams to implement the protocol in their projects correctly. One major problem was the fact that the teams used different Java versions. There is no backwards compatibility between Java 6 and 7 so if a project was compiled with Java 7 it couldn't be executed on machines with older Java versions. Therefore it was necessary for all teams to upgrade their Java versions.

After all the problems have been eliminated a bash script was written to execute the GameMaster for all given programs. There were 10 competitions with 1000 games each. The execution time of the tournament script was 6 hours.

Although there weren't many points of contact between the teams during the implementation period it was still a lot of fun to develop a protocol collectively and finding bugs in different projects.

It was a great idea to organize a tournament between different teams. This way you can see the result of your efforts directly and compare it with other teams. Also, working with students from other countries and cultures gave us an insight into their different way of thinking.