# SISP project 1

Johan Lund Neergaard & Christian Nøhr Rasmussen

March 2016

## 1 Search algorithm

### 1.1 Minimax algorithm

Our solution is based on the Minimax algorithm. The algorithm assumes that there are two players and that the first player (called max) will make the first move, followed by the other player (called min). It is assumed that this will continue until a **terminal** state is reached. A terminal state is defined as a state where either one of the players has won or no further moves are possible. The algorithm works by calculating all possible states, which can be reached by taking all possible moves at each state. This results in a rather large tree of paths to a terminal state.

When all terminal states have been reached, they are each given a value determining their worth to the player. Then the value is propagated up through the branches. For each node the following happens:

If it is max's turn to move, then he will select the move, which results in the largest value.

If it is min's turn to move, then he will select the move, which results in the lowest value.

It is from this, that the algorithm gets its name. Min will always select the minimum value, while max will always select the maximum value.

### 1.2 Alpha-Beta pruning

Alpha-Beta pruning is a modification of the Minimax algorithm, which aims to reduce the number of branches searched. The basic idea is, that since we have knowledge about which turn it is, we can safely say that for a given node, where min is to decide, if min finds a value, which is lower than the largest value found so far in its parent (which is a max node), we do not need to continue the search from the min node, since the parent will always select the higher value, and thus we need not expand the branches from this min node.

The algorithm is mirrored for when the current node is a max node. The algorithm is equal to the Minimax algorithm, except that the best and worst value found so far need to be passed along and compared within the nodes.

Alpha-Beta pruning allowed us to reach a depth, which was two levels deeper than before, and therefore significantly increased the **intelligence** of our AI.

### 1.3 Iterative deepening

We also added iterative deepening to ensure that we would always use all available time. By using iterative deepening, we could quickly calculate the best move for a small depth and remember that move. Then we could iteratively improve our result by going deeper and deeper, and overriding the previous best move. We run this iterative function in a separate thread, as a java Future, and we simply time out the thread if our time runs out.

We cut off when we reach the max depth allotted for the given search. This is accomplished by passing the current depth in the min-max algorithm and simply using the heurstic for evaluation if we reach the maximum depth.

We use a starting depth of 8, to a maximum of 30. Furthermore, we use a timeout of 10 seconds, to ensure that we use no more than a maximum of 10 seconds per turn.

### 1.4 Possible improvements

A clear improvement would have been to use a transposition table to store computations for states. It is quite likely that the same state would be reached through different paths, and if that happens, there is no reason to do the heuristic computation again.

Also, we do not check if the result of a given depth indicates a terminal depth, so in the case that we just reached the end, and the end is closer than the maximum depth, we will run more computations until we run out of time. This is unnecessary, since we will arrive at the same result.

## 2 Evaluation algorithm

The Heuristic runs through all possible combinations of four in a row, and assigns points to the players depending on how many they have in that row. If both or neither player have a token in the specific combination, no points are awarded. Every token in the combination awards 1 point, while every empty space awards 0.5 points. That means a combination with one token will award 2.5 points, a row with 2 tokens 3 points, and a row with 3 tokens 3.5 points. We added the empty space value because we wanted the algorithm to favour blocking the opponents setups over playing as far away as possible to have more possible combinations of getting four in a row. Because a token close to the middle are included in more winning combinations, it will naturally have a higher value than a token near the edge of the board.

To give an example how how points are counted, consider blue token in column 2 row 3 in Figure 1. The token is included in 4 possible winning combinations: 2 horizontal (column 1-4 and 2-5), one vertical (row 3-6), and one

diagonal (column 2, row 3 to column 5, row 6). Note that the remaining combinations (vertical row 1-4 and 2-5 and diagonal column 1, row 4 to column 4 row 1) will not be counted, as they are blocked by red tokens.

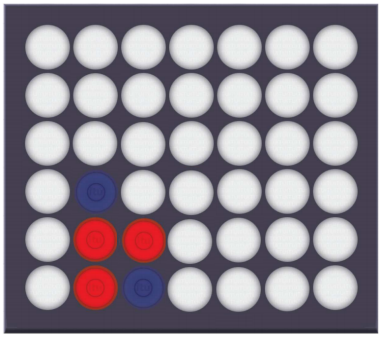

Figure 1: Example of a for in a row game board.

## 2.1 Traps

While the Heuristic runs through all the possible combinations of four in a row, it will also save any cases where a trap exists. A trap is defined as three out of four in a row, where the fourth token cannot be placed yet, because the slot below it is empty. After evaluating the possible winning combinations as explained above, the heuristic will then go through all the traps, and award points depending on the value of the trap. A trap will be considered less valuable if one of the following conditions holds true: The trap is not the first (that is, lowest) in its column, the trap overlaps the opponents trap, or the trap does not allow the player to switch initiative. A trap that allows the player to switch initiative is a trap where when the player is forced to break their own trap because all other moves would result in a loss, filling out the rest of the column will result in the initiative being switched to the other player. This is the case when there is an odd amount of empty spaces above the trap. For instance, if player 1 has a trap in the top row, breaking in would result in player 2 placing a token in the top row, filling out the column, and player 1 would have the initiative again.

Another thing that we tried to implement but ended up discarding was attempting to predict the winner based on all the traps, and how many moves were left. However, because the game is often decided relatively early on, by the time there was enough traps to correctly predict the winner, it was too late.