Mathias Flink Brandt (mfli@itu.dk)
Simon Langhoff (siml@itu.dk)

Connect Four

Intelligent Systems Programming
Mandatory Assignment 1
March 12th, 2015

Search Algorithm

We have implemented the Minimax algorithm as described by RN13[1], page 169. The algorithm is implemented in the MiniMaxer class and defined by the MinimaxDecision method. The first step of implementing Minimax for the Connect Four problem space, is to define a game state. The game state is simply a two-dimensional array, which reflects the rows and columns of the game board. When a player inserts a coin, the state is updated with the ID of the player who inserted the coin, at the position it was inserted. The state also keeps track of where the last coin was placed. This is useful for the algorithm in order to determine the value of the current state, as described later. When the AI has to decide its next move it calls our Minimax algorithm and passes along the current state of the game. The algorithm then goes through all the actions available from the supplied state. That is, for all the columns in the game that are not full, we recursively call our algorithm to determine which column we should place the next coin in. For each call to our algorithm, we copy the current game state so that for each action we perform and evaluate upon, we do so on a new state, which in turn has been defined by the previous step in the recursive call stack if any. Due to the fact that we have to test all possible combinations of actions the running time quickly becomes impractical if we do not improve upon the basic version of the algorithm. With this in mind, we have enhanced the algorithm with alpha-beta pruning of the search tree, a cut-off depth of the recursive calls and an evaluation function that changes non-terminal nodes in the search tree, to terminal nodes when the cut-off depth has been reached. The value of these terminals are then defined by the heuristic value of the board state as given by that node in the search tree.

Alpha-beta Pruning

The idea behind alpha-beta pruning stems from the fact that the algorithm assumes that the opponent is playing optimally. So in the case that a player has a given choice node *a*, if that node has a parent node (or further up the tree) with an even better value, then *a* will not have to be considered, as the tree would have chosen another node further up the path and branched away from *a.* We implemented the pruning algorithm defined on page 173, figure 7 in RN13.

Cut-off Function

To further decrease computation time, we implemented a cut-off function that simply checks the current depth of the search tree and then decides if it should continue or not. In the case that the cut-off depth has been reached, we substitute the choice node with a terminal node, by using our evaluation

[1] Artificial Intelligence - A Modern Approach, 3rd edition, 2014

function to estimate a value of the state of the choice node. We found that if we use a cut-off larger than 8, the computation time for some moves could exceed the 30 second maximum, as such we decided not to keep it at 8, even though is usually much faster than 10 seconds.

Evaluation Rules

We have chosen to evaluate the terminal states based on three separate features. Each feature calculates a sub-score based on the current game state. The sum of the sub-scores from each feature forms a total score for that specific state. The algorithm compares these total scores to determine which column is the best. Each feature is described in detail below.

Coin Position Value

Each position on the board is assigned a base value, according to a static array[2]. This is due to the fact that some positions will initially always be better than others. For example, placing the first coin of the game in the middle column of the board gives more opportunities than placing it in the left- or rightmost column. By utilizing this base value, it is ensured that the algorithm will pick the most favorable column if multiple columns result in the same evaluation value when considering the two other features alone.

The static array that determines the value of a position has only been defined for a board with seven columns and six rows. Due to this fact, this evaluation feature is not generic and thus our evaluation function is only working optimally for a board of this size.

Max Coins in a Row

Coins can be placed in rows on four different axes, namely horizontal, vertical, diagonal going down from left to right (diagonalOne), and diagonal going up from left to right (diagonalTwo). When a coin is placed in a column, it can become a part of multiple of these rows simultaneously. For each possible row, the feature calculates the amount of coins that are positioned in succession of each other in that row, including the most recently placed coin (i.e., the coin placed in the column that we are currently evaluating). The return value is the maximum of these values.

The most critical return value from this feature is one that is equal to the win condition. If this value is returned, the algorithm should place a coin in the corresponding column. This is ensured by returning the max utility value from the entire evaluation function. The result will be either of the following outcomes:

- The algorithm connects four coins and wins the game
- The algorithm effectively blocks the opponent from connecting four coins

[2] http://goo.gl/5JT5P4 (stackexchange.com)

Returning a value equal to the win condition also constitutes the upper bound of the range of values that the feature can return. The feature will never return a greater value, since the game would already be over in such a state.

If the feature returns a value smaller than the win condition, the value is treated as a sub-score only if the algorithm itself placed the last coin (if this is not the case, the value is dismissed). Furthermore, the value is being multiplied by six to ensure a higher evaluation value than another column that might initially have a higher base value.

Win Combinations Count

In isolation, the "Max Coins in a Row" feature is inadequate in computing an accurate score for the column. To enhance the score, we also calculate how many of the rows in which the coin takes part that can actually result in a win. Consider the following board:

```
 1 2 3 4 5 6
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
|O| |X|X|O| |
```

The "Max Coins in a Row" feature would return a value of three for column two. However, placing a coin in this column will never result in a win on the horizontal axis, though it might result in winning on the vertical or diagonalOne axis at some point in the future. The "Win Combinations Count" feature takes this into account, by calculating the amount of rows that can possibly turn out as winning rows in the future and returning this value. Consider the same board as above, but at a later point in the game:

```
 1 2 3 4 5 6
| | | | | | |
| | | | | | |
| | | | | | |
|O| | |X|X| |
|O| |X|X|O|O|
```

When examining column three, the "Max Coins in a Row" feature will once more return a value of three. However, this column should be preferred to column two, since it can result in more winning rows in the future, namely the horizontal, vertical, and diagonalOne axis.

As a consequence, the "Win Combinations Count" feature will return a value of two for column two, and three for column three. In turn, the evaluation function adds this value to the return value of the "Max Coins in a Row" feature, resulting in a preliminary total score of five for column two and six for column three, thus weighing column three highest.