Intelligent Systems Programming, Assignment 1: Connect Four Game

Mark Gray, Johan Sivertsen & Lauge Groes

March 15, 2012

Introduction and approach

To solve the challenge of creating a connect four-playing algorithm we chose a rather experimenting approach. The final result is the best of a number of implementations and failed experiments. We will first briefly describe some of the process and then detail the final implementation.

Our first attempt was a simple depth-first search-implementation of the mini-max algorithm like the one found in the course book on page 166. To implement this a solid and fast data structure was required. We have build a structure that is more or less a 2d-array of boolean objects(*Coin*'s), with some useful methods added. This is the *Board* class. This data structure proved to be rather successful and remaind largely unchanged throughout the process.

In the depth-first search-implementation, we would recursively consider the opponents response to each action. This amounted to recursively calling the MinValue() and MaxValue() methods for each action associated with the state we were in at the time of the call. This implementation was able to play perfectly, as long as the board was limited to a size of 3x3.

On a larger board the depth-first search led to only a very small part of the search space being evaluated. Since we would sequentially start with the first action of the first state and then travel all the way to a terminal state, we would only reach very few terminal states within the time constraints. We tested with a cutoff[1] based on the number of evaluations explored, but even when allowing for 10's of millions of evaluations any larger board would be left largely unexplored.

We then implemented $\alpha - \beta$-pruning. This allowed for a significant performance boost. It meant that boards up to 5x5 could be played reasonably well.

The solution was to introduce depth-limited search and a heuristic evaluation function. Introducing depth-limited search was a change in the way we did our cut-offs worked . Instead of a counter that increased with every recursive call, we introduced a counter for the depth of the search tree. This would mean that instead of following one path to the end, we would evaluated as many options as possible to a certain depth. This is also much closer to the "human" way of evaluation the board.

Without a heuristic the depth-limit made the new AI player worse for larger boards, when we kept the depth of the search tree on the lower end. Since no terminal state was ever reached, the utility of playing a certain action was never evaluated. This shows that a heuristic is extremely important but also that it is the heuristic that decides moves, and thus the logic, for the AI player, until it can reach terminal states in its search. Choosing a good heuristic is thus what this approach to AI boils down to.

Parallel to the implementation of depth-limited search we also implemented a transposition table. This provides a dictionary where the AI player can lookup states already evaluated states. Since a certain state can be reached by taking actions in different orders, one should see improvements speed since we reduce some searches to constant lookups in a hash table.

However, we experienced that the transposition table did not improve running time but rather reduced it. The overhead of putting states into the transposition table seemed to outweigh any

[1] We are talking about when to cut the evaluation process, not cut-offs due to $\alpha - \beta$-pruning.

benefits the constant lookup provided. Our keys in the dictionary are states that are quite large. For the competition board of $6 * 7$ we end up with 3^{42} possible combinations of information for each state. We actually use more since we erred on the side of easy implementation. Each field on the board would be coded as 2 bits so we would end up with 4^{42} possible combinations (2^{42} bits). In the end decided not to use transposition tables at all. The overhead was too large compared to the benefit.

Search and Cut-off Function

The final search is an implementation of a depth limited,$\alpha - \beta$-pruned, Minimax algorithm. It follows the book rather closely, but we will go though to cut-off and other specifics not included in the book.

The main difference between this implementation and the pseudocode on page 170 is the implementation of a depth-limit. This makes is necessary to trace the depth of the return values. We do this using the vAd and bAd arrays(v And Depth, b And Depth). This allows us to choose winning conditions at shallow depths first, and loosing conditions at large depths. The helper functions are gathered in separate classes called *ToolSet* and *State*. *ToolSet* includes functionality for transforming boards and performing calculations like Result() and Max(). *State* includes methods for evaluating states, like Utility() and isTerminal().

To understand the implementation we will walk through the elements of the general algorithm and explain our java-implementation of same element.

A state is implemented as an instance of the class *State*. A state has a Utility and a test to checks if it's terminal. *State* is constructed from an instance of the class *Board* which holds the data-structure for our boards and all information on the placement of *Coins*. There is a very close relationship between a state and a *Board*, but we have split them in order to maintain separation between board and the interpretation of the board. The actions are defined through a query to the *Board*-instance asking for the open columns of the board. This returns an array of integers, signifying the number of a column. The transition between two states can then be carried out through the *Result*() method which takes a *Board*-instance, an integer(the column to play), and the ID of the player adding his coin, it returns a new board with the added information. The helper functions $Max()$ and $Min()$ are part of the *ToolSet* class.

Since the state space is so large the Utility of a state cannon't simply be the "real" minimax value, but must be heuristic in case of a cut-off. The next section details our evaluation function.

Evaluation Function

The evaluation function calculates the heuristic of a given board. It uses a function "calcCoinHeuristic()" to evaluate the individual rows, columns and diagonals. This is the function that determines the heuristic value. Given a coin array the function checks if there is a possibility for a winning condition. The closer the player is to a winning condition, the higher the power of ten times the player value is added to the heuristics. E.g. if playerOne has three coins in a row followed by one null value, the added value to the heuristic is $1 * 10^0 + 1 * 10^1 + 1 * 10^2 = 111$, $-1 * 10^0 + -1 * 10^1 + -1 * 10^2 = -111$ if the coins belong to player 2.

This makes boards with connected friendly coins much more attractive than boards with connected enemy coins. Using the factor ten also means that a board with a single 3-connected

situation will be much more attractive than a large number of 2 or 1-connected situations. A problem is that this does not take into account rows or diagonals with zeros followed by opponent coins and therefor underestimates situations where it is possible to play an unchallenged 3-connection, something that will always lead to a win in one move. This can be improved by including information on player turns, but was beyond our timeframe.

The evaluation function simply sums the heuristic values for all the possible rows, columns and diagonals.