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 in detail the final implementation.

Our first attempt was a simple depth-first search-implementation of the mini-max algorithm. To implement this a solid and fast data structure was required. We have built a structure that is more or less a 2d array of boolean objects(Coins). This data structure proved to be rather successful.

In the the depth-first search-implementation, we would recursively consider the opponents response to each action. This amounted to recursively calling the min and max methods for each action associated with the state we were in at the time of the call. Our experience with this led us to introduce some changes to the final AI player.

The depth-first search led to only a small part of the search space to be evaluated since we would sequentially start with the first action of the first state considered and then the consider the first action the resulting state and hence forward until we reached a terminal state. We would only reach a few terminal states when the board size was large, due to our time constraints.

Even when $\alpha - \beta$-pruning was introduced, the search space was still too large for board sizes larger than 4x4.

The solution was to introduce depth-limited search and a heuristic evaluation function. Introducing depth-limited search amounted to changed how we our cut-offs worked (i.e. not cut-offs due to $\alpha - \beta$-pruning). Instead of a counter that increased with every recursive call, we introduced a counter for the depth of the search tree.

Without a heuristic the changes 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 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.

```java
public class MiniMaxAB {
        int counter =0;
        int cutoff = 8;
        int evals = 0;
        int depth = 0;

        public MiniMaxAB(){
        }

        int ABsearch(Board board){
//                      cutoff=(int) ((float) (189/board.openSlotsLeft()+2.5));

//                      int v = MaxValue(board, Integer.MIN_VALUE, Integer.MAX_VALUE);
                        int choice = ToolSet.Actions(board)[0];
                        int[] vAd = new int[2];
                        vAd[0] = Integer.MIN_VALUE;
                        vAd[1] = 0;

                        for (int a: ToolSet.Actions(board)){
                                counter++;
//                              int b = MinValue(ToolSet.Result(board,a,1),Integer.MIN_VALUE, Integer.MAX_VALUE);
                                int[] bAd = MinValue((ToolSet.Result(board,a,1)), Integer.MIN_VALUE, Integer.MAX_VALUE);
//                              if(b==v) choice=a;
                                if(bAd[0] > vAd[0]) {
                                        choice = a;
                                        vAd = bAd;
                                }
                                else if(bAd[0] == vAd[0]){
                                                if(bAd[0] >= 0) {
                                                        if(bAd[1] < vAd[1]){
                                                                vAd = bAd;
                                                                choice = a;
                                                        }
                                                }
                                                else if(bAd[1] > vAd[1]) {
                                                        vAd = bAd;
                                                        choice = a;
                                                }
                                        }
                                counter--;
                        }

                        System.out.println("At depth "+vAd[1]+", after " + evals+" evaluations, and "+board.openSlotsLeft()+" open slots left in the board.");
                        counter=0;
                        System.out.println("I have chosen to play: "+choice);
                        System.out.println("For an expected outcome of: "+vAd[0]+", at depth:"+vAd[1]);
                        return choice;
                }

        int[] MaxValue(Board board,int alpha,int beta){
                int[] ret = new int[2];
                ret[1] = counter;
                StateEvolved test = new StateEvolved(board);
                if(counter>cutoff){
                        ret[0] = test.getUtility();
                        return ret;
                        }
                evals++;
                if(test.isTerminal()){
                        ret[0] = test.getUtility();
                        return ret;
                }
                int v[] = new int[2];
                v[0] = Integer.MIN_VALUE;
                for (int a:ToolSet.Actions(board)){
                        counter++;
                        int[] minV = MinValue((ToolSet.Result(board,a,1)),alpha,beta);
                        counter--;
                        if(v[0] < minV[0])v = minV;

                        if(v[0] >= beta) return v;
```

We do this using the vAd and bAd arrays(v And Depth, b And Depth). The helper functions are gathered in separate classes called ToolSet and StateEvolved. ToolSet includes functionality for transforming boards and performing calculations like Result() and Max(). StateEvolved 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 a 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 "calcCoin-Heuristic()" 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 $1 * 10^0 + 1 * 10^1 + 1 * 10^2 = 111$ and vice versa $-1 * 10^0 + -1 * 10^1 + -1 * 10^2 = -111$ is added to the heuristic for 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. The "calcCoinHeuristic()" function is inserted below.

```
        private int calcCoinHeuristic(Coin[] coinArr){
                int[] oneHeu = new int[winCondition];
                int[] twoHeu = new int[winCondition];
                int oneCount = 0;
                int twoCount = 0;
                int heuristic = 0;

                for (Coin coin:coinArr){
                        if(coin==null){
                                oneCount++;
                                twoCount++;
                        }
                        else if(coin.isPlayerOne()){
                                oneHeu[oneCount] = 1;
                                oneCount++;
                                twoCount=0;
                        }
                        else if(coin.isPlayerTwo()){
                                twoHeu[twoCount] = -1;
                                twoCount++;
                                oneCount=0;
                                }

                        if(oneCount == winCondition){
                                oneCount = 0;
                                int zeroCount = winCondition;
                                int tempHeuristic = 0;

                                for(int one: oneHeu){
                                        if(one!=0){
                                                int value = (int) Math.pow(10,oneCount);
                                                tempHeuristic += value*one;
                                                oneCount++;
                                                zeroCount--;
                                        }
                                }
                                if(zeroCount!=0){
                                        heuristic += tempHeuristic;
                                }
                                oneCount = 0;
                                oneHeu = new int[winCondition];
                        }

                        if(twoCount == winCondition){
                                twoCount = 0;
                                int zeroCount = winCondition;
                                int tempHeuristic = 0;

                                for(int two: twoHeu){
                                        if(two!=0){
                                        int value = (int) Math.pow(10,twoCount);
                                        heuristic += value*two;
                                        twoCount++;
                                        zeroCount--;
                                        }
                                }
                                if(zeroCount!=0){
                                        heuristic += tempHeuristic;
                                }
                                twoCount = 0;
                                twoHeu = new int[winCondition];
                        }
                }



                return heuristic;
        }
}
```

Figure 2: Excerpt of code