Intelligent Systems Programming, Assignment 1: Connect Four Game

Mark Gray, Johan Sivertsen & Lauge Groes

March 15, 2012

Introduction

We have developed an implementation of a connect four agent that uses a combination of the minimax algorithm, depth-limited search, $\alpha - \beta$-pruning, a transposition table and a heuristic evaluation function.

Search and Cut-off Function

Our first attempt was a simple depth-first search-implementation of the mini-max algorithm, which included a cut-off after a certain number recursive calls and $\alpha - \beta$-pruning.

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 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 evaluated since we would sequantially start with the first action of the first state considereder and then first action the resulting and hence forward until we reached a terminal state. We would only reach a few terminal states when the board size was large enough, 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 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 of with 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 is coded as 2 bits so we end up with 4^42 possible combinations (2^42 bits). We tried our hands on our own hash functions implementations but in the end we used Javas.

Archicture and Implementation

To better understand our solution one needs to understand our chosen architectural design and how we have implemented the methods.

MatrixLogix is our implementation of the IGameLogic interface. We have opted to push most of the logic to other clases for a cleaner implementation.

The AI player uses a Board object to store states in. Board objects are passed between methods that require information about a certain state.

Evaluation Function

```java
import java.util.Arrays;


public class Board implements Cloneable{

        public Column[] board = new Column[0];
        private int rows = 0;
        private int columns = 0;
        private int winCondition = 4;
        private boolean isEmpty=true;
        private int playerID = 1;


        /**
         * Constructs the board.
         * @param column
         * @param row
         */
        public Board(int column, int row){
                this.board = new Column[column];
                this.rows = row;
                this.columns = column;
                for (int i=0;i<column;i++){
                        this.board[i]=new Column(row);
                }
        }

        public Board(int column, int row, int playerID){
                this.board = new Column[column];
                this.rows = row;
                this.playerID = playerID;
                this.columns = column;
                for (int i=0;i<column;i++){
                        this.board[i]=new Column(row);
                }
        }

        /**
         * The clone implementation for a board.
         */
        @Override
        protected Object clone(){
                Board clone = new Board(this.columns,this.rows);
                clone.board = new Column[this.colCount()];
                for(int i=0;i<this.colCount();i++){
                                clone.board[i] = (Column) this.board[i].clone();
                }

                return clone;
        }

    /**
    *
    * @param object
    * @return Returns true if the object passed to the method is equal to
    * to the caller object.
```

Figure 1: Excerpt of code