

## Laboration 4: Game Playing

### **Background:**

Games playing is one of the most prominent and popular areas of artificial intelligence that is visible far beyond the community of computer scientists, for example the chess match between the computer Deep Blue and the human former world champion Kasparov (see <http://www.research.ibm.com/deepblue/home/html/b.html>) generated a great deal of interest and media coverage. As soon as computers became programmable, Turing and Shannon wrote the first chess playing programs. Shannon discovered a min-max algorithm by which the best move for any given position can be computed. Thereby it is possible to transform the problem of game playing into a search problem. The process basically represents an arbitrarily large number positions that might result from every possible series of moves, assigns them a numerical score and works backward from this information to derive the best first move. Each step along the chain is called a half move in chess or a ply in computer science. Each new ply causes a branching in the search tree, in which positions are the nodes of the tree and moves are the links between nodes. An evaluation function then scores each end position, assigning for example a “1” to a win, “-1” to a loss and “0” to a draw. As the number of possible positions grows exponentially with the number of moves, an exhaustive search of the entire tree becomes impossible for all but the simplest games such as Tic-Tac-Toe. Therefore, an evaluation function is needed that assigns a score to a given position, based for example on the number of pieces that each opponent has or positional values such as piece placement, pawn structure etc.

### **Assignment:**

Your task is to design a game playing software that is able to compute the best move for arbitrary deterministic (e.g. no dice game such as backgammon) two-player games. You should then apply your game strategy to two concrete games, namely Tic-Tac-Toe and “Connect 4”. You can obtain one extra point (to improve your grade, see grading on the course web-page) if you design and implement improved search strategies such as alpha-beta pruning and ordering moves and in addition apply those strategies to the game checkers. You are supposed to implement the system in the language C++.

### **Expected time:**

20 hours per person

### **Preparation:**

- Study lecture notes on game playing (Monday 12/11/01)
- Read sections 5.1, 5.2, 5.3, 5.4 in the book “Artificial Intelligence – A Modern Approach” Stuart Russell & Peter Norvig, Prentice Hall series

### **Tips:**

Further useful reading and links:

- How Chess Computers Work, Marshall Brain, basic introduction  
<http://www.howstuffworks.com/chess.htm/printable>

- Game trees, min-max search, alpha-beta search  
<http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic11/>
- Game playing, min-max search  
<http://hugsrv.kaist.ac.kr/~kywch0/Storage/aim/topics.htm#Game>
- Game AI page <http://www.gameai.com>
- Checkers rules: [http://www.itsyourturn.com/t\\_help/topic2030.html](http://www.itsyourturn.com/t_help/topic2030.html)
- Connect-4 tutorial: <http://www.ce.unipr.it/~gbe/cn4rules.html>
- Silicon Champions of the Game, Ivars Peterson, popular article  
[http://www.sciencenews.org/sn\\_arc97/8\\_2\\_97/bob1.htm](http://www.sciencenews.org/sn_arc97/8_2_97/bob1.htm)

The template method pattern can be useful to define the skeleton of a general deterministic two-player game. One suggestion is to design abstract classes for *Player*, *Game*, *Move* and *GameState* (board configuration), which provide the interface for the *SearchStrategy* that implements the min-max search algorithm to identify the best move. The concrete games Tic-Tac-Toe and Connect-4 subclass the abstract classes and define concrete subclasses for the particular type of board states, moves and rules. The class *GameState* and its subclasses should have a method *PossibleMoves()* to generate the legal moves that are “playable” in a particular board configuration and an evaluation function *Utility()* that computes the utility or value of a board configuration (for example in checkers counting the number of own pieces minus the number of opponent pieces). Take advantage of the vector and list classes in the standard template library, for example the Tic-Tac-Toe board is composed of `vector< vector<Square*> >` or the method *PossibleMoves* returns a `list<Move*>`.

Conceive a preliminary design model (class diagram with concepts, associations, attributes, methods) before you start with the implementation. You can discuss your design model with the assistant or the course leader.

**Basic rules for connect-4:** Connect-4 is a two player game which is played on a 7x6 rectangular board placed vertically between them. Player A has 21 red, player B 21 orange tokens. Each player can drop a token at the top of the board in any one of the seven columns, as long as this column is not full ( 6 tokens in one column). The objective of the game is to align four connected (neighboring) tokens of the same colour either horizontally, vertically or diagonally. The game ends in a draw if the board is full and all 42 tokens have been played without either player achieving four connected tokens. Players alternate and are forced to move which becomes particular relevant towards the end of the game as more and more columns fill up. Connect-4 has been solved (see <http://www.cs.vu.nl/~victor/connect4.html>) and the player who moves first has a safe winning strategy.

**Basic rules for Tic-Tac-Toe:** Tic-Tac-Toe is played on a 3x3 squares-board, players alternate placing their tokens (Player A red, player B blue) on an empty square. A player wins when he has three tokens aligned in a row, column or diagonal. The game ends in a draw if all squares are occupied and neither player has a winning configuration.

**Assignment:**

Design classes for

1. *Move* : defines a possible move in the game (for example in Tic-Tac-Toe place a red *Token* on *Square A2* )
2. *GameState* : describes a game configuration, a method *PossibleMoves()* generates a list of all legal next moves in that configuration. One can apply a *Move* (method *MakeMove()*) to a given *GameState* which transforms the *GameState* into the configuration resulting from that *Move*. In addition you will find it useful for the tree search algorithm to provide an *UndoMove()* operation that reverses the effect of a *Move* and reconstructs the board configuration (*GameState*) prior to the execution of the *Move*. *GameState* should provide a method *Utility()* that evaluates the utility of a game configuration, for example in connect-4 by counting the number of three token groups in a row, column or diagonal configurations that constitute a potential threat to the opponent as they might result in a four token group winning configuration.
3. *Game* is composed of a *GameState* and in addition manages the switching of *Players*, which alternate executing *Moves*. It keeps tracks of who's turn it is to move and also has methods to determine if the current *GameState* constitutes a *Win()* for either *Player* or a *Draw()*.
4. In most two-player games (including checkers, tic-tac-toe and connect-4) the board is built of *Squares* on which the players can place or move *Tokens*. A *Move* in such a game consist of either placing a *Token* on an empty *Square* (tic-tac-toe and connect-4) or in moving a *Token* from its current *Square* to another *Square* (checkers, chess) possibly capturing an opponents *Token*. Notice, that *Tokens* are usually although not necessarily owned by a particular *Player* who then is allowed to move or place it. For the basic lab assignment you are only required to implement the two similar games tic-tac-toe and connect-4, for the extra point you would have to implement checkers as well. Notice that checkers distinguishes between “man” that move a diagonal distance of one (regular move) or two (jump when capturing) an opponent token and “kings” which can move or jump across an arbitrary distance. Also notice, that in checkers there are forced moves, namely that a player has to jump with a “man” or “king” and cannot choose to move the same or another token instead.
5. Define an abstract class *SearchStrategy* that is able to compute the *BestMove()* (called MiniMax-Decision in the AI book page 126) and the *Value()* (called MiniMax-Value in the AI book page 126) for a particular abstract *GameState*. Concrete subclasses such as *MinMaxSearchStrategy* or *AlphaBetaPruningStrategy* implement a concrete search algorithm. In the basic assignment (no extra lab points) you are only supposed to implement the *MinMaxSearchStrategy*. If you want to get the extra point you also have to implement *AlphaBetaPruningStrategy* with a suitable order scheme of Moves (for example always evaluate moves that capture an opponents token first)

### References:

Artificial Intelligence-A Modern Approach, Stuart Russell, Peter Norvig, Prentice Hall, 1995