Synthesizing Board Evaluation Functions for Connect4 using Machine Learning Techniques

Master Thesis in Computer Science

Martin Stenmark

Department of Computer Science
Østfold University College

July 2005

Abstract

In this paper, we present the application of two different machine learning techniques, Reinforcement Learning and Automatic Programming, to the domain of game-playing, using the techniques separately to automatically synthesize cutoff-point evaluation functions for the board game Connect4. We show that a substantial amount of relevant learning occurs, and that the resulting functions, combined with game tree search algorithms, are quite adept at playing Connect4.

Table of Contents

1. Introduction

Board games have constituted a popular computer science research area for decades, and many exciting results have emerged, e.g., IBM's chess computer Deep Blue, playing chess at a grandmaster level. When attempting to create skilled game-playing algorithms, numerous methods are available, e.g., brute force search algorithms, hard-coded rules based on human expert knowledge, training neural networks on recorded game databases, and so on. The possible methods, and combinations of them, are seemingly only limited by the human imagination, and new methods continue to emerge.

The following master thesis describes an attempt to synthesize board evaluation functions for the game Connect4, using two different machine learning techniques, i.e., Automatic Programming through the ADATE system [Olsson, 1995], and Reinforcement Learning [Sutton & Barto, 1998]. Both these methods continuously refine an evaluation function based on actual game-playing experience, albeit in quite different ways. The methods are both instances of unsupervised learning, indicating that they are presented with an environment and a goal, and are left alone to explore their environment and reach their goal, through manipulating their respective evaluation functions.

The resulting evaluation functions can easily be used as move selectors, i.e., to select which move to make in a particular game situation/state. This is done by using the functions to evaluate all possible immediate game states resulting from making a move in the game state in question, and then make the move that produces the highest evaluated resulting state. Moreover, the evaluation functions can be combined with game tree search algorithms, looking even further down the game tree, and in the process extending and improving the move selector. The objective of these experiments has been to determine how well these machine learning methods are suited to the game-playing domain, and in the process, try to produce adept Connect4-playing algorithms.

This subject was picked because it seems to be a particularly suitable testing ground for machine learning. A typical game represents a sort of micro cosmos, with clearly defined rules and constraints, and as such can relatively easily be represented accurately by a computer without having to oversimplify the problem. At the same time, the concept of a game is well-known and popular, and this makes it easy to present results that are achieved. Finally, games are entertaining to work with.

The thesis starts with a brief introduction to basic game tree search algorithms, specifically the minimax algorithm and its variations. Next, the game of Connect4 is introduced and described, before the ADATE system and Reinforcement Learning are presented, in separate sections. A thorough description of how the methods were applied to the problem is supplied, as well as the results of the experiments. I studied Reinforcement Learning thoroughly when preparing for my thesis work, and I have programmed the Reinforcement Learning system presented in this thesis, from scratch, including a modifiable neural network. Contrastingly, the ADATE system was developed by my thesis supervisor, Roland Olsson, and I have limited knowledge of the system's inner workings. My dealings with ADATE have first and foremost consisted of crafting the specification file presented to the system. Because of this, The Reinforcement Learning section of the thesis is a great deal more detailed than the ADATE section, and more background theory is presented. Finally, at the end of the thesis, the results from the two machine learning methods are

compared and analyzed.

2. Game theory algorithms

In this section, we introduce the minimax algorithm [Brassard & Bratley, 1996], and an extension of it, namely alpha-beta pruning. We provide the reader with a few examples and illustrations to clarify the nature of the algorithm. The minimax algorithm has been used extensively in the experiments described in this thesis.

2.1. The minimax algorithm

The minimax algorithm is a recursive algorithm which can be used to pick the next move in perfect-information, zero-sum 2 player games, e.g., Chess, checkers, Tic-Tac-Toe, Connect4. A perfect information game is a game in which no information is hidden from the player, and no random events occur. A zero-sum game is a game in which the gain of one player is the loss of the other player. Hence, the sum of loss and gain is zero.

The algorithm is applied to a game tree rooted at the current game state, and by using a depth-first search of the tree, it determines the optimal move of the current player, given the current game state. The opposing player is assumed to make the optimal move in all situations.

In its simplest form, the minimax algorithm traverses the entire game tree rooted at the current game state, each time it searches for the best move. In a game tree, each node represents a game state. The children of a node represents the resulting states following the possible moves from this node (game state). Figure 1 shows a small excerpt of the game tree of Tic-Tac-Toe.

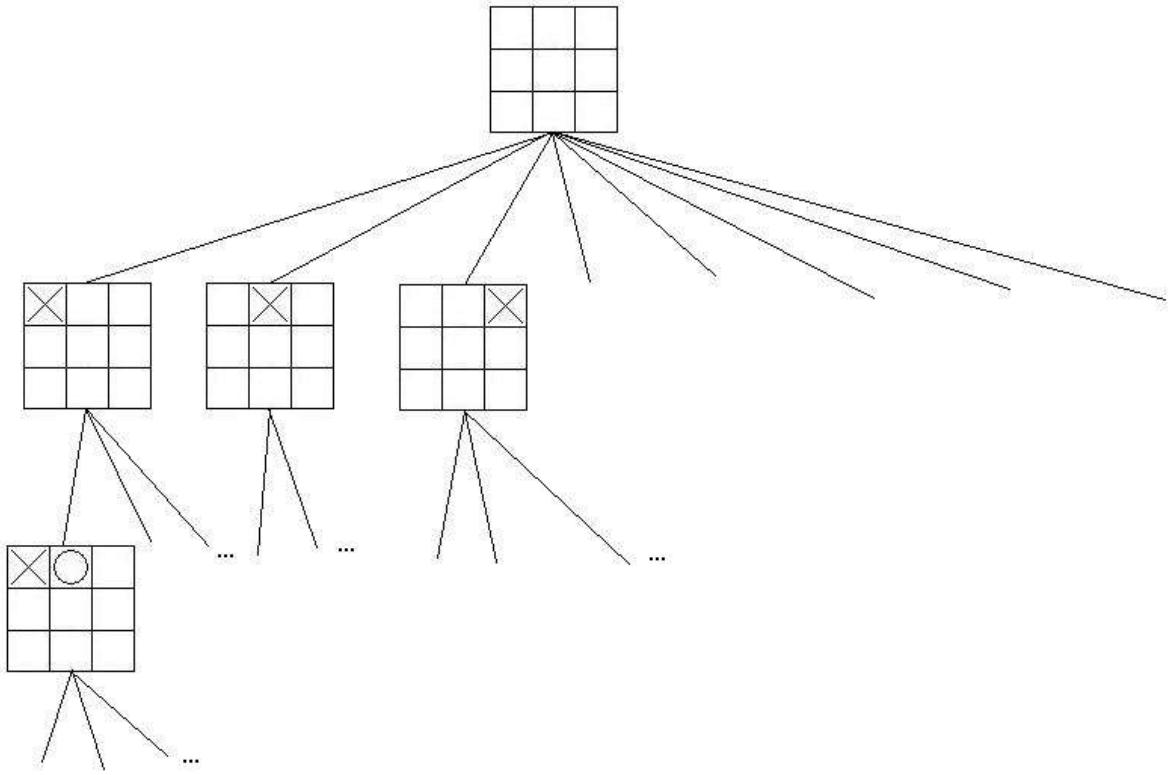

Figure 1. Part of the Tic-Tac-Toe game tree

Let us name the initial player A (places Xs), and the opposing player B (places Os). A is referred to as the maximizing player, and B as the minimizing player. Each game state (node) in the game tree is given an evaluation value, obtained by starting at that node and following each possible combination of successive moves until the leaves of the game tree are reached. The leaves represent end-game states:

- A game state in which A is the winning player, is assigned a value of +INFINITY
- A game state in which B is the winning player, is assigned a value of -INFINITY
- A drawn game state, if possible given the nature of the game, is assigned a value of 0

These values are subsequently backpropagated up the game tree in the following way:

- The value of a node representing a game state where the next move is A's, is the maximum of its childrens' values
- The value of a node representing a game state where the next move is B's, is the minimum of its childrens' values

When the traversal is done, if the current player is A, the move which leads to the child node (game state) with the highest value is chosen. If the current player is B, the move which leads to the child node with the lowest value is chosen.

Below is a block of C-like pseudo code representing the minimax algorithm. The function *minimax* computes the minimax value of a game state. In order to use this function to choose a move from a given game state, the *minimax* function is called with each of the game state's children. The child with the highest score is chosen as the next move. In this example, the function *chooseMove* is responsible for this. *ChooseMove* calls *minimax*:

2.1.1. The chooseMove function

```
board chooseMove(board gamestate, boolean player1) {
      int best_score = 0;
      board best_move = 0;
      if (player1) {
            best_score = -INFINITY;
      }
      else {
            best_score = +INFINITY;
      }

      board[] moves = gamestate.getChildren();

      for (int i=0; i<moves.length; i++) {
            int score = minimax(moves[i], not(player1));
            if (player1) {
                  if (score > best_score) {
                        best_score = score;
                        best_move = moves[i];
                  }
            }
            else if (not(player1)) {
                  if (score < best_score) {
                        best_score = score;
                        best_move = moves[i];
                  }
            }
      }

      return best_move;
}
```

2.1.2. The minimax function

```
int minimax(board gamestate, boolean player1) {

      if (gameOver(gamestate)) {
            if (player A won) {
                  return +INFINITY;
            }
            else if (player B won) {
                  return -INFINITY;
            }
            else if (draw) {
                  return 0;
            }
      }

      int best_score = 0;
      if (player1) {
            best_score = -INFINITY;
      }
      else {
            best_score = +INFINITY;
      }

      board[] moves = gamestate.getChildren();

      for (int i=0; i<moves.length; i++) {
            int score = minimax(moves[i], not(player1));
            if (player1) {
                  if (score > best_score) {
                        best_score = score;
                  }
            }
            else if (not(player1)) {
                  if (score < best_score) {
                        best_score = score;
                  }
            }
      }

      return best_score;
}
```

As an illustrative example, let us construct an extremely simple 2 player game, and follow minimax's traversal of its game tree. The game consists of a game board of three squares forming a row, like this:

The 2 players take alternate moves. A move consists of placing an X or an O in a vacant square on the board. Player A places Xs and player B places Os. Player A always starts. If a player manages to place 2 Xs or Os in a row, he is the winner. If the board is full, and no player has won, the game is a draw. Figure 2 shows the entire game tree of our game:

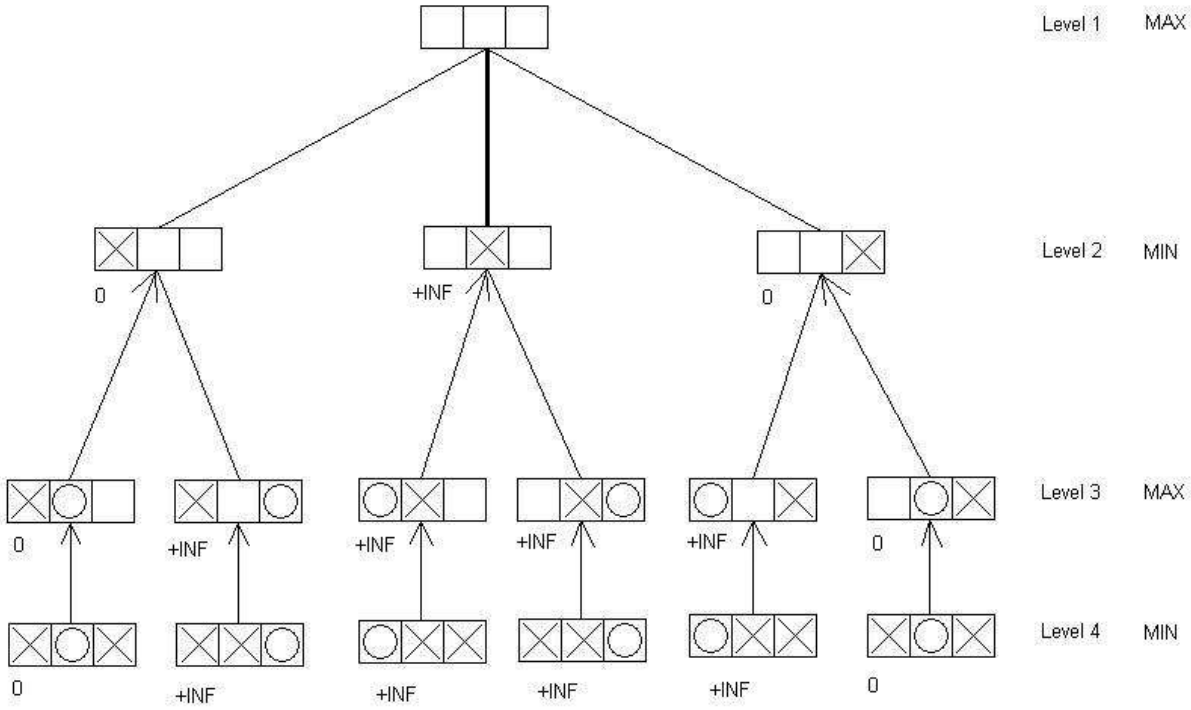

Figure 2

As we can see from the tree, it is impossible for player B (O) to win the game. 4 out of 6 leaves (Level 4) represent a game state in which player A (X) is the winner, and the remaining 2 game states are drawn. The drawn game states are assigned a value of 0, while the game states representing a victory for A, are given a value of +INFINITY. Moving up to Level 3 in the tree, A is the moving player. As A is the maximizing player, he seeks the move which yields the highest reward. Since all nodes on Level 3 only have one child each, there really is no choice to make. All nodes on Level 3 are assigned the value of their only child. On Level 2, B is the moving player, and has 3 possible situations to be in, each providing 2 alternatives for B to choose from. B is the

minimizing player, so each of the 3 nodes on Level 2 are assigned the minimum value of their children. Level 1 of the game tree represents the initial, empty game board, and player A's first choice. A is the maximizing player, and makes the move giving the highest value. This move is placing an X in the center square. From the tree, we can see that this is the only first-move which guarantees victory for player A.

This approach works well with games with small game trees, such as Tic-Tac-Toe. It quickly becomes computationally infeasible, however, as the complexity of the game increases. Chess, for instance, has been estimated to have between 10^{43} and 10^{50} possible game states. In this case a complete game tree cannot be generated in practice. Instead a partial game tree can be built, down to a certain depth, and heuristics can be applied to evaluate the leaves of the tree, even though these leaves aren't real end games. In this case a non-final game state will be given a numerical score according to an evaluation function. Synthesizing this kind of evaluation function is what we describe in this thesis. Using a cutoff evaluation function will of course, almost certainly, compromise the accuracy of the algorithm. There are ways to make minimax more efficient, without sacrificing any accuracy. A well-known such method is alpha-beta pruning. If the game tree is too big to be completely traversed, alpha-beta pruning can still allow the search algorithm to search deeper in the tree, even though it can't search all the way to the leaves.

2.2. Alpha-beta pruning

Alpha-beta pruning is an extension of the minimax algorithm, and its purpose is to avoid searching unnecessary parts of the game tree. This is accomplished by maintaining 2 variables A(Alpha) and B(Beta). A represents the maximizing player's best guaranteed result, at any given time. B represents the minimizing player's best guaranteed result, at any given time. A is initially assigned the value of -INFINITY, and B is assigned the value of +INFINITY. These values are updated during the traversal of the game tree, and are used to “prune” the tree where possible, by cutting off branches which are certain to NOT represent a better alternative than the current best alternative. In order to visualize this algorithm, let us consider a simple game tree, shown in Figure 3.

Each node in the tree represents a game state. The values inside the circles are merely used to identify the nodes in the following explanation. The nodes' scores are located directly beneath the nodes. Black nodes indicate nodes that are never reached, due to pruning. The alpha-beta search is applied as shown in Figure 3.

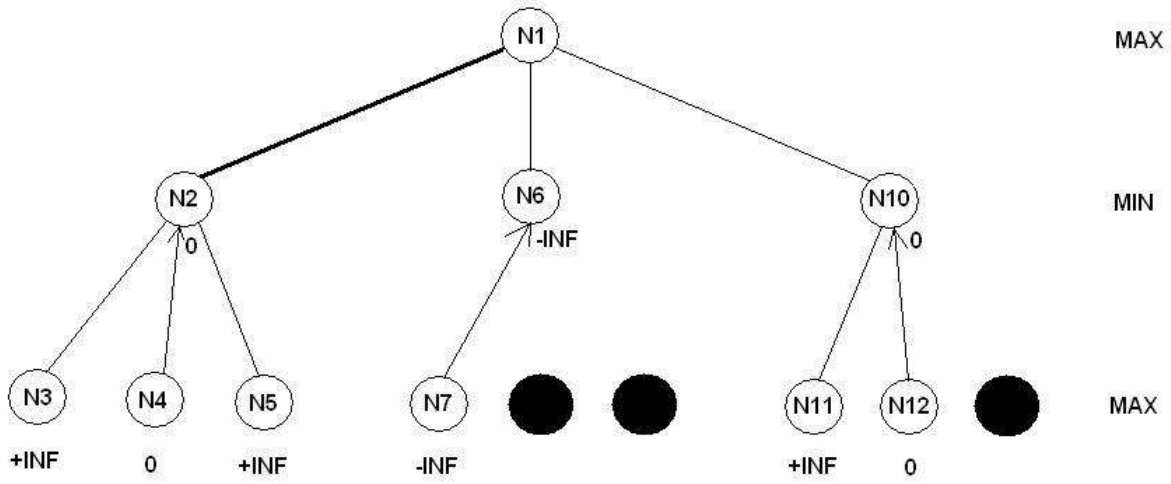

Figure 3. Alpha-beta pruning

The alpha-beta search is applied in the following way:

N1 is a maximizing node, and is therefore “interested” in the child (N2, N6 or N10) with the highest (guaranteed) value. N1 is associated with an alpha value (denoted A) of -INFINITY and a beta value (denoted B) of +INFINITY.

- N1: A = -INFINITY, B = +INFINITY

Since the algorithm employs a depth-first search, it proceeds to N1's first child, namely N2. N2 is associated with the initial values of N1.

- N2: A = -INFINITY, B = +INFINITY

N2 is a minimizing node, and is given the value of its least-valued child. The algorithm recursively calls N2's first child, N3. N3 is a leaf node, with value +INFINITY, indicating a win for the maximizing player. This value is returned to N2. Since N2 is a minimizing node, the value is compared to N2's previous B value, which was +INFINITY. If the new value is smaller than the old, it becomes the new B value of N2. In this case it isn't (+INFINITY = +INFINITY). N2's B value remains unchanged.

Now N2's second child (N4) is called recursively. N4 is a leaf indicating a draw, and therefore returns 0 to N2. N2 compares this to its B value (which is +INFINITY), to see if it is smaller than the old B value. It is, so N2's B value is updated to 0.

- N2: A = -INFINITY, B = 0

Next, N2's final child (N5) is called. N5 is a leaf node indicating a win for the maximizing player, and returns +INFINITY to N2. N2 compares this to its B value. +INFINITY is not smaller than 0, so N2's B value remains unchanged. N2 is given the evaluation value of its B value, which is 0. 0 is returned to N1.

N1 is a maximizing node, and compares the value it received from N2 (0) to its current A value (-INFINITY). If the return value is bigger than the current A value, the new A value is assigned the return value. 0 is bigger than -INFINITY, and so:

- N1: A = 0, B = +INFINITY

N1's second child (N6) is called recursively, and is associated with these values:

- N6: A = 0, B = +INFINITY

As we can see, N6 is passed its parent's (N1's) A value. This is like N1 telling N6 : “N2 has guaranteed me a draw if I choose its path. If you come across a child of yours that doesn't have a better guarantee, stop the search immediately, and don't waste my time.”

N6's first child (N7) is called, and since it represents a loss for the maximizing player, it returns -INFINITY to N6. N6 is a minimizing node, and compares this return value to its B value. If the return value (-INFINITY) is smaller than the current B value (+INFINITY), the B value is assigned the return value. And it is. Hence:

- N6: A = 0, B = -INFINITY

Now the algorithm tests if N6's B value is smaller or equal to its A value. It is, and so the rest of N6's children (N8 and N9) are ignored, and N6's current B value (-INFINITY) is returned to its parent (N1). This is done because N6's B value can never increase in value, only decrease. Therefore, what is returned from N6 to N1 can never be higher than -INFINITY, and in turn never higher than the 0 already returned from N2 to N1. Following this logic, it is unnecessary to examine N8 and N9. These parts of the game tree are pruned. As mentioned, N6's B value (-INFINITY) is returned to N1.

As N1 is a maximizing node, it examines this return value to see if it is greater than its current A value. It is not (return value=-INFINITY, A = 0), so the values associated with N1 are not altered:

- N1: A = 0, B = +INFINITY

Now it is time to examine N1's last child, N10. These are the values associated with N10 in the

recursive call:

- N10: A = 0, B = +INFINITY

N10's first child (N11) is called. It represents a win by the maximizing player, and returns +INFINITY to N10. N10 is a minimizing node, and checks to see if this return value is smaller than its current B value (+INFINITY). It isn't, so N10's values remain unchanged.

Next, N10's second child (N12) is called. N12 indicates a draw, and returns 0 to N10. N10 checks to see if this return value is smaller than its current B value (+INFINITY). It is, so N10's new B value is 0.

- N10: A = 0, B = 0

Now, if N10's B value is smaller than or equal to its A value, its current B value is returned to N1. This means that N13 is ignored, and another pruning has occurred. The reason for this is that N10 can never return anything higher than its current B value 0 , no matter what the value of N13 is. Since N1 has already been guaranteed a draw (0) from N2, no further searching is required in N10's subtree.

The optimal path from N1 is therefore N2. The first path (N2) is chosen over the last one (N10), because N10 is only guaranteed to not give a better result than 0. That doesn't mean that it can't give a worse result than 0.

Below follows a trace of the tree traversal, and the pruning, described in the example above. One level of indentation in the trace indicates one depth-level in the game tree:

```
Start: N1
    Start: N2
        Start: N3
        N3 is a leaf node
        N3 :P1 won; returning +INF
    N2 :received score = +INF from N3
        Start: N4
        N4 is a leaf node
        N4 :Draw; returning 0
    N2 :received score = 0 from N4
    N2 :Updating B from +INF to 0
        Start: N5
        N5 is a leaf node
        N5 :P1 won; returning +INF
    N2 :received score = +INF from N5
    N2 :Returning B = 0
N1 :received score = 0 from N2
N1 :Updating A from -INF to 0
    Start: N6
        Start: N7
        N7 is a leaf node
        N7 :P2 won; returning -INF
    N6 :received score = -INF from N7
    N6 :Updating B from INF to -INF
    N6 : B = -INF, a = 0; Cutting off
N1 :received score = -INF from N6
    Start: N10
        Start: N11
        N11 is a leaf node
        N11 :P1 won; returning INF
    N10 :received score = INF from N11
        Start: N12
        N12 is a leaf node
        N12 :Draw; returning 0
    N10 :received score = 0 from N12
    N10 :Updating B from INF to 0
    N10 : B = 0, A = 0; Cutting off
N1 :received score = 0 from N10
N1 :Returning A = 0
```

Alpha-beta pruning typically reduces the number of nodes examined to the square root of the number of nodes examined using regular minimax. This enables alpha-beta to search twice as deep in the game tree in the same amount of time. The efficiency of alpha-beta depends greatly on the sequence the children of a node are traversed in. The optimal traversal sequence would consist of traversing children from lowest to highest score at a minimizing node, and from highest to lowest score at a maximizing node. Multiple heuristics have been developed to exploit this.

A way to further increase the efficiency of alpha-beta pruning is to consider a narrower search

window (the interval between the Alpha and Beta values), but this compromises the accuracy of the algorithm, as the actual value may reside outside of this window. In this case, the search must be redone, with a wider search window. This is called aspiration search. Other extensions to alpha-beta search also exist.

In addition to this, a number of more complex algorithms exist, which are able to compute minimax values without sacrificing any accuracy. PVS, Negascout and MTD-f are three such algorithms. All algorithms mentioned up to this point employ depth-first search. SSS* is an example of an algorithm that uses best-first search, an optimization of breadth-first search.

Here is a block of C-like pseudo code representing the minimax algorithm extended with alpha-beta pruning (*chooseMove* calls *minimax_ab*):

2.2.1. The chooseMove function

```
board chooseMove(board gamestate, boolean player1) {
      int best_score = 0;
      board best_move = 0;
      if (player1) {
            best_score = -INFINITY;
      }
      else {
            best_score = +INFINITY;
      }

      board[] moves = gamestate.getChildren();

      for (int i=0; i<moves.length; i++) {
            int score = minimax_ab(moves[i], not(player1), -INFINITY,+INFINITY);
            if (player1) {
                  if (score > best_score) {
                        best_score = score;
                        best_move = moves[i];
                  }
            }
            else if (not(player1)) {
                  if (score < best_score) {
                        best_score = score;
                        best_move = moves[i];
                  }
            }
      }

      return best_move;
}
```

2.2.2. The minimax_ab function

```
int minimax_ab(board gamestate, boolean player1, int a, int b) {

      if (gameOver(gamestate)) {
            if (player A won) {
                  return +INFINITY;
            }
            else if (player B won) {
                  return -INFINITY;
            }
            else if (draw) {
                  return 0;
            }
      }


      board[] moves = gamestate.getChildren();

      if (player1) {
            for (int i=0; i<moves.length; i++) {
                  int score = minimax_ab(moves[i], not(player1), a, b);
                  if (score > a) {
                        a = score;
                        if (a>=b) {
                              break;
                        }
                  }
            }
      }

      else if (not(player1)) {
            for (int i=0; i<moves.length; i++) {
                  int score = minimax_ab(moves[i], not(player1), a, b);
                  if (score < b) {
                        b = score;
                        if (b<=a) {
                              break;
                        }
                  }
            }
      }


      if (player1) {
            return a;
      }
      else {
            return b;
      }

}
```

3. The game of Connect4

This thesis is based on an attempt to create adept game-playing algorithms, using machine learning techniques. The game which has been used in the experiments described in the thesis is Connect4, which is based on the classic child's game Tic-Tac-Toe. Other games considered were minesweeper, yatzee, mastermind, battleship and hearts. Connect4 was chosen because its degree of complexity seemed to be suitable for the purposes of this thesis; although Connect4 is far less complex than games like chess or backgammon, it is complex enough to represent a serious challenge for human players. In addition to this, its state space is much too vast for brute-force search techniques to be applied successfully.

3.1. Game details

Connect4 (aka Connect Four or Plot Four) was published in 1974, and is a 2-player game governed by very few and simple rules. The game consists of a game board with 42 circular holes in it (ironically, often called squares), arranged in 6 rows and 7 columns, as shown in Figure 4.

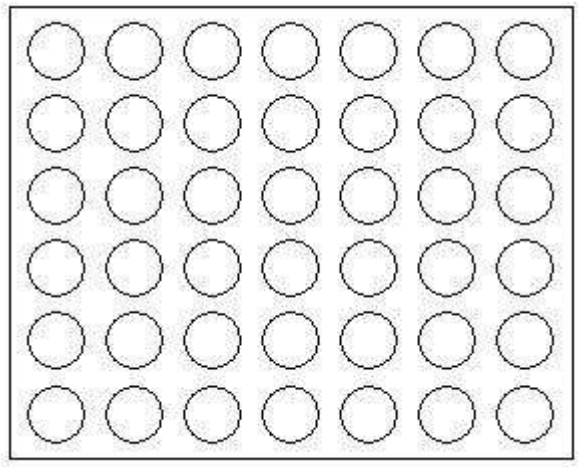

Figure 4. Connect4 board

The game board is placed in an upright position. In other words, Figure 4 shows the board as seen from the front, and not as seen from above. On the top of each of the 7 columns, is a slot. Two sets of 21 small circular discs are given to the two opposing players. The disc sets are colored differently, in order to separate the two players from each other on the game board. The players will now alternate in dropping one disc at a time into one of the 7 slots on top of the columns, providing the column in question is not already filled up. When a disc is dropped into a slot, the disc will drop down to the lowest vacant square in the corresponding column. Thus, gravity is implicitly a key factor in the game. The gameplay consists of the two players alternating in dropping discs into slots of vacant columns. The objective of the game is for a player to connect 4 of his discs in a straight, uninterrupted line on the board. The line can be horizontal, vertical or diagonal. When a player

achieves this, he has won the game. If, however, the game board is filled (all discs are used), and neither of the players have connected 4 discs, the game is a draw. The illustrations in Figure 5 show player 1 (gray player), winning three different games against player 2 (black player), by connecting 4 discs in a horizontal, vertical and diagonal manner. From here on, player 1 will be called the X-player, and player 2 will be called the O-player. When the X-player drops a disc on the board, this is called placing an X, and when the O-player drops a disc, this is called placing an O.

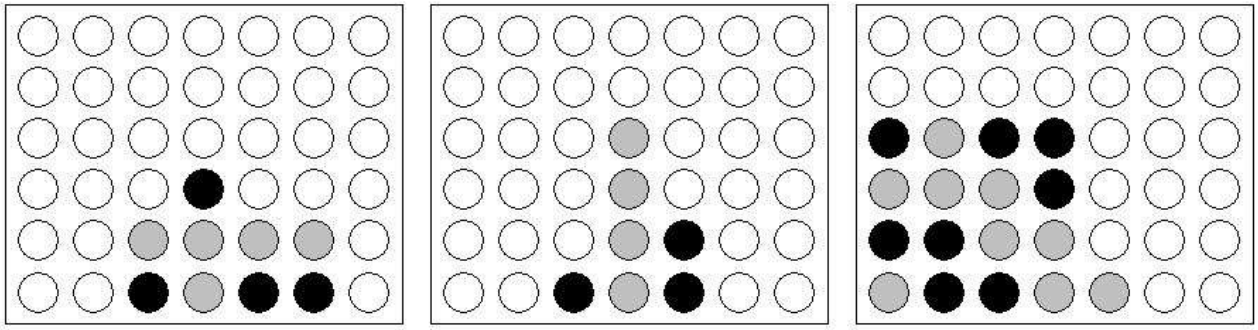

Figure 5. Horizontal, vertical and diagonal winning lines.

Given the structure of the board, we can easily see that squares residing near the middle of the board are more desirable than those closer to the edges. The reason for this is that central squares are much more likely to be part of a winning row. One basic strategy of the game is to somehow try to achieve a double threat to the adversary, i.e., to reach a state in the game where the opponent has to block two squares at the same time in order to prevent you from winning. Clearly, this is impossible, and you are therefore guaranteed a win in such situations.

Another important aspect of the game is that of *Zugzwang*, i.e., the concept of forcing the opponent to make a move he would rather not take if he was given a choice. This feature is prevalent in Connect4 since moves are made in a strictly alternating fashion, and moves cannot be forfeited. The fact that gravity plays an important part in the game (discs are dropped down to the lowest vacant square of the row), often leads to situations where a player is forced to make a move, effectively preparing the win for his opponent. Consider the board shown in Figure 6.

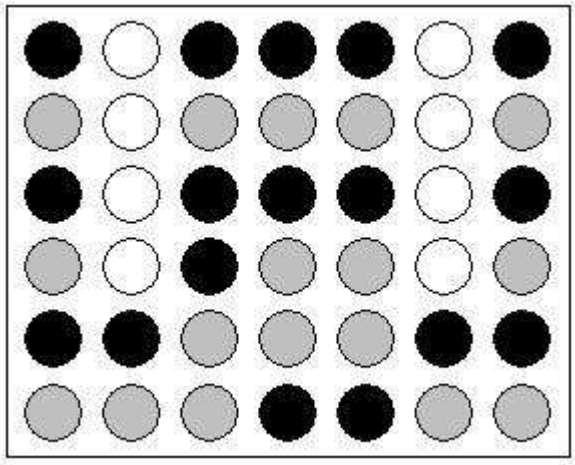

Figure 6. Example of Zugzwang.

Gray is the beginning player (X-player), and since an even number of squares are filled, this means that it is gray's turn to move. Gray has to make a move, and can choose between the second and the sixth column. Clearly, this is no real choice, since either alternative leads to preparing the victory for the O-player. This is a typical example of the concept of Zugzwang.

Although the rules of the game are very simple, becoming a skilled Connect4 player is not a trivial task. More sophisticated strategies consist of planning ahead by counting vacant squares, placing threats on odd and even squares, following your opponent by placing your discs directly on top of his, and so on.

3.2. Complexity of the game

Considering that the Connect4 board consists of 42 squares, and each of these can be in one of three states: empty, occupied by the X-player or occupied by the O-player, we have an upper bound of 3^{42} ($>= 10^{20}$) possible board configurations. However, this rough estimate includes many illegal configurations, which cannot occur when playing the game according to the rules. By excluding some of these states, an upper bound of $7.1 * 10^{13}$ possible states has been found, according to [Allis, 1988]. This makes a brute-force search approach infeasible.

3.3. The optimal strategy

Despite of this, Connect4 has been solved. This happened in 1988, when Victor Allis and James D. Allen solved the game independently of each other. The solution shows that the starting player can always force a win by starting in the center column, i.e., the 4th column, and playing perfectly thereafter. If the starting player starts in columns 2, 3, 5 or 6, the second player will reach a draw by playing perfectly. Finally, if the starting player starts in columns 1 or 7, the second player can force

a win through perfect play. Victor Allis wrote a Connect4-playing program based on his solution [Allis, 1988]. This program was called Victor, and its gameplay used a knowledge-based approach, instead of game tree search. More specifically, it played according to a set of mathematically proven strategic rules. A successor of Victor, called Velena, is available as freeware. Other strong players also exist, which, in contrast to Victor/Velena, employ different combinations of game tree search, heuristic board evaluation functions and stored databases of moves. Examples of these players are Mustrum and TioT.

In our experiments, we have chosen to respectfully disregard the optimal strategy, and concentrate on using machine learning methods to synthesize board evaluation functions and combine them with game tree search algorithms. The primary objective being, as stated earlier, to examine how well these methods are suited to be used in the game-playing domain.

3.4. Evaluating the Connect4 board

When it comes to game tree search, there are numerous ways to improve the tree searching algorithm itself, in terms of search speed. An example of this is using alpha-beta pruning, as mentioned in section 2. A faster algorithm will then be able to search deeper in the game tree in the same amount of time. This will in turn typically improve the player using the algorithm. However, if the evaluation function is poor, the performance of the corresponding search algorithm will suffer. It is therefore important to find a good evaluation function, i.e., a function which takes a Connect4 board as input, and outputs a numerical value indicating the desirability of the board state for the moving player. Considering the relative simplicity of Connect4, an efficient search algorithm, coupled with an evaluation function which captures relevant characteristics of the game board, should be able to play a strong game of Connect4.

How does one construct a good board evaluation function for Connect-4? This is not easily answered, but intuitively the following function description makes sense:

The Connect4 game board is essentially a 7*6 grid (7 columns, 6 rows). There are 69 unique possible winning “lines” on the board. 24 of these are horizontal, 21 are vertical and 24 are diagonal. Given a specific board configuration, each of these 69 lines are examined and given a value, in the following way:

for each of the four squares in the line:

if the square is occupied by the maximizing player, it is given a value of 1

if the square is occupied by the minimizing player, it is given a value of -1

if the square is empty, it is given a value of 0

The value of each line is the sum of the values of its 4 individual squares, and the value of the board is the sum of the values of the 69 potential winning lines. In other words, each square is weighted

according to the number of unique winning lines it can be a member of, illustrated by Figure 7.

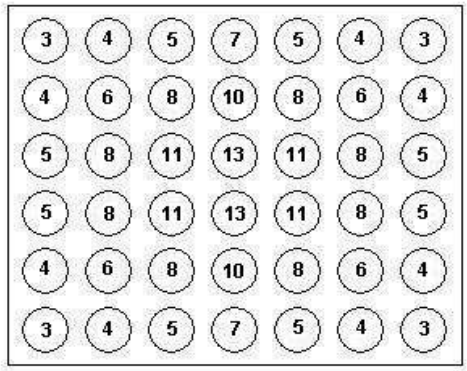

Figure 7. IBEF's weighting of squares.

This evaluation function, combined with a search algorithm, is able to play a quite good game of Connect4. As an example, if we let the minimax algorithm using cutoff depth 1 and this function at the cutoff points, play X, against an opponent making random moves, the X-player will win about 98% of the time. If we let the minimax algorithm using cutoff depth 1 and this function at the cutoff points, play O, against an opponent making random moves, the O-player will win about 90% of the time. In both cases, the winning percentage increases along with the cutoff depth used by the minimax algorithm.

From this point on in the thesis, we will often make the following simplification when describing the playing result of an evaluation function: Instead of writing

"the minimax algorithm using cutoff depth *n* and evaluation function *f* at the cutoff points"

we will write

"the evaluation function *f* using cutoff depth *n*"

In other words, it is always implied that an evaluation function is associated with the minimax algorithm. Although technically imprecise, we have found that this simplification makes the explanations in this thesis clearer and easier to understand.

For brevity's sake, let us denote the evaluation function shown in Figure 7, IBEF (Intuitive Board Evaluation Function). In our experiments, we have utilized IBEF as an opponent for training and evaluation purposes, as well as a starting point for ADATE's function synthesis.

3.5. Applying minimax to Connect4

The game tree search algorithm used in our experiments is the classic minimax algorithm, extended with alpha-beta pruning. As discussed earlier, Connect4 is too complex for its game tree to be completely traversed by a search algorithm, in a reasonable amount of time. To circumvent this problem, the use of board evaluation functions is introduced.

The application of minimax to the game tree of Connect4 is carried out in the same way described in chapter 2, with one notable exception: The minimax function is, in addition to its other parameters, supplied with a value indicating a maximal search depth, or cutoff depth. If this cutoff depth is reached at a non-leaf node in the game tree, the algorithm will not continue the search to the subtree of the node in question, even though the node does not represent a terminal game state. Instead, the algorithm will apply its evaluation function to this node, and return the resulting value, which will serve as a replacement for the real minimax value of the node. As a simple example, consider Figure 8.

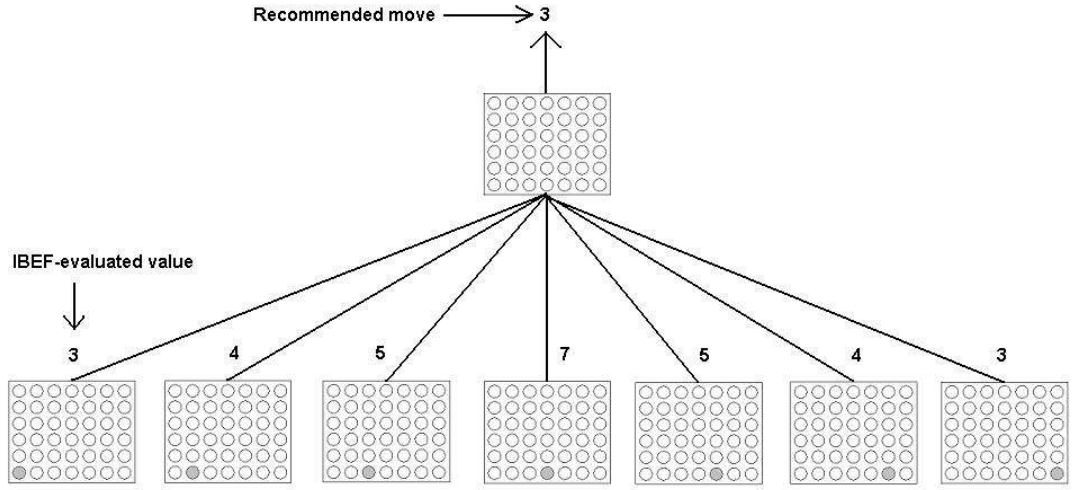

Figure 8. Using minimax with cutoff depth 1 and IBEF evaluation function to determine which move to make.

In Figure 8, the cutoff depth is set to just 1, which means that the minimax algorithm only examines the immediate board states resulting from the possible moves in the current situation. In this example, the board is empty, and it is therefore the X-player's turn to move. The minimax function is called with the empty board and a cutoff depth of 1. Simply put, he minimax function places an X in the leftmost column, and calls itself with the resulting board. In this new activation of the minimax function, the cutoff depth is reached, and thus the evaluation function is applied to the board with an X in the leftmost column. In this case, let us assume that the IBEF evaluation function is used. This means that the board with an X in the leftmost column is assigned a value of 3, according to the diagram shown in Figure 7. This value is returned to the first activation of the minimax function, and is retained in order to compare it to the values of the remaining 6 possible resulting states. Next, the minimax function removes the X from the leftmost column of the board,

and puts it in the second column from the left instead. Then it calls itself with this board, the new activation of minimax applies the IBEF function, returns the resulting value, and so on. As we can see, placing the X in the middle column will result in the highest evaluated resulting board, which is assigned a value of 7. Thus, the minimax algorithm using the IBEF evaluation function and a cutoff depth of 1, will choose to put the first X of the game in the middle column, which is indexed 3. This choice is consistent with the optimal strategy mentioned earlier.

4. The ADATE system

Automatic Programming is a method of generating program code automatically, based on training data, i.e., input/output examples. This approach can yield programs which solve algorithmic problems that would be difficult to solve by writing the programs directly. The aspect of evolution utilized in Automatic Programming is partly based on biological evolution. This means that the program evolves over time according to the demands the environment presents to it. In the beginning, one starts with a single program, and this program is subsequently developed gradually. At any time, a multitude of candidate programs is maintained. These programs are evaluated after their ability solve the algorithmic problem at hand. This ability is called the fitness of the program. New programs are generated by combining properties from the various programs, or by introducing new properties.

Automatic Design of Algorithms Through Evolution (ADATE) [Olsson,1995] is a system for automatic programming i.e., inductive inference of algorithms. ADATE utilizes a combinatorial search of program space by exploring combinations of neutral program transformations. These transformations can be, for example, different kinds of code replacements, abstractions, embedding and so on. By supplying ADATE with a specification file, it can generate algorithms to solve the problem presented in the file, provided sufficient CPU time is available. ADATE generates functional programs in the language of Standard ML, and is designed to solve any algorithmic problem.

4.1. Synthesizing Connect4 evaluation functions using ADATE

Our approach has been to use ADATE to try to find a strong Connect4 board evaluation function, by making ADATE play Connect4 using a minimax algorithm combined with the evaluation function that is to be synthesized. ADATE's opponent is a minimax algorithm which uses the IBEF-function described in section 3. The objective is to synthesize an evaluation function that improves upon the IBEF evaluation function. In order to perform this experiment, we had to write a specification file for ADATE, in fact the largest specification file ever presented to the ADATE system. The file is included in appendix A, and is described in the following section. The code in the file is written in the functional language Standard ML.

4.1.1. The specification file

The specification file, named *playC4.spec*, is structured in the following manner:

First, the necessary data types are defined, representing the game board, game result and so on. 7 data types are defined in this file. Next, a complete Connect4 playing system is built, in the shape of 34 interconnected ML functions. This system, which essentially serves as a function evaluation

system, is comprised of a complete minimax algorithm with alpha-beta pruning, a heuristic evaluation function (IBEF) that is to be used by ADATE's adversary, and ADATE's evaluation function. Initially, ADATE's evaluation function is identical to the adversary's evaluation function, although this is not a prerequisite. This Connect4 playing system enables two players to play Connect4 against each other, both using the minimax algorithm, but with different cutoff point evaluation functions.

Next, the *main* function of the specification file is defined. This function must be present in all specification files in the current version of ADATE. The task of this function is to take the input examples, one at a time, and present them to the evaluation mechanism, which uses these examples to evaluate the synthesized function. In this particular case, the input examples are in the form of game situations from which Connect4 games are to be played out. The evaluation process consists of making ADATE play Connect4, starting at these game situations, using the minimax algorithm with the synthesized function at the game tree cutoff points.

After this, a number of auxiliary functions are included in the file, their task being to prepare the input examples for the *main* function. Next, 155 starting board are included, from which the evaluation games will be played. 556 validation starting boards are also included.

4.1.1.1. The data types

These are the data types used in the specification file:

- `datatype intList  = nill | cons of int * intList`

IntList is a recursive data type which represents a single row of a Connect4 board. In this list of integers, 1 means that the X-player has occupied a square, -1 means that the O-player has occupied the square, and 0 means that the square is empty.

- `datatype intListList = nill' | cons' of intList * intListList`

IntListList is a list of *intLists*, and is thus used to represent an entire Connect4 board. This is merely an intermediate way of representing the board.

- `datatype intPair = intPair of int * int`

IntPair is used in order to be able to return a pair of integers from a function.

- `datatype matrix = matrix of int * int * int uncheckedArray`

Matrix is used by ADATE to represent a Connect4 board.

- `datatype gameResult = gameResult of int * int`

GameResult is used to indicate the result of a finished game of Connect4. The data type includes two integers; the first one shows who won, the second one shows how many moves it took for the game to end, starting at the starting board state.

- `datatype mainInput = mainInput of intListList * bool * bool * int * int`

MainInput is the data type describing the *main* function's input domain (the *main* function will be described later). *MainInput* is comprised of the following:

  - a starting board
  - a boolean value indicating whether ADATE is the X-player or not
  - a boolean value indicating whether it is the X-players move or not
  - an integer indicating the depth of the minimax search ADATE is going to use
  - an integer indicating the depth of the minimax search ADATE's opponent is going to use

- `datatype mainReturn = mainReturn of bool * int`

MainReturn describes the output range of the *main* function. The bool value is true if the ADATE-player won the game at hand, otherwise false. The integer indicates how many moves it took for the ADATE-player to win. If the ADATE-player lost, or the game was drawn, this integer is assigned a value of zero.

4.1.1.2. The Connect4 playing system

As mentioned, the Connect4 playing system consists of 34 interrelated functions, which we have written in Standard ML, and adapted to the limited syntax permitted by ADATE's ML interpreter. The functions and their interrelationships are described in the following.

- `fun not( X : bool ) : bool`

A simple negation function.

- `fun length( Xs : intList ) : int`

This function measures the length of an *intList*, i.e., a row of a Connect4 board.

- `fun length'( Xss : intListList ) : int`

This function measures the length of an *intListList*, i.e., the number of rows in a Connect4 board.

- `fun sub( ( M, Row, Col ) : matrix * int * int ) : int`

This function returns the element, i.e., integer value, situated at the intersection of the row with index *Row* and the column with index *Col*, on the game board.

- `fun update( ( M, Row, Col, X ) : matrix * int * int * int ) : matrix`

This function returns the *matrix*(board), after having updated the matrix element situated at the intersection of the row with index *Row* and the column with index *Col*, assigning it the value of *X*.

- `fun fillInRow( ( Row, Col, Xs, World )`
  `: int * int * intList * matrix ) : matrix`

Auxiliary function used by *fillIn* when converting an *intListList* to a *matrix*.

- `fun fillIn( ( Row, Xss, World ) : int * intListList * matrix ) : matrix`

Auxiliary function used by *toMatrix* when converting an *intListList* to a *matrix*.

- `fun toMatrix( Xss : intListList ) : matrix`

This function converts an *intListList* to a *matrix*.

- `fun getRow( ( Row, Col, M ) : int * int * matrix ) : intList`

Auxiliary function used by *getAll* when converting a *matrix* to an *intListList*.

- `fun getAll( ( Row, M ) : int * matrix ) : intListList`

Auxiliary function used by *fromMatrix* when converting a *matrix* to an *intListList*.

- `fun fromMatrix( M : matrix ) : intListList`

This function converts a *matrix* to an *intListList*.

- `fun eval(board:matrix):int`

This is the static IBEF board evaluation function, programmed in ML code. This evaluation function is used by ADATE's opponent, in cooperation with the minimax algorithm. The success of an ADATE-synthesized function is measured against this function.

- `fun f(board:matrix):int`

This is the dynamic board evaluation function used by ADATE in cooperation with the minimax algorithm. Transforming this function is what ADATE is all about.

- `fun and5((v,w,x,y,z):int*int*int*int*bool):bool`

This function is used to check the board for winning lines. It returns true only the values of v, w, x and y are the same, and z equals true. The function is called by *goHorizontal*, *goVertical* and *goDiagonal*.

- `fun goHorizontal((board,row):matrix*int):bool`

This function checks if there are any horizontal wins present on the game board.

- `fun goVertical((board,column):matrix*int):bool`

This function checks if there are any vertical wins present on the game board.

- `fun goDiagonal(board:matrix):bool`

This function checks if there are any diagonal wins present on the game board.

- `fun bF((board,column,row): matrix*int*int):bool`

This function checks if the game board is full.

- `fun boardFull(board:matrix):bool`

This is a wrapper function for the *bF* function described directly above.

- `fun whoWon(board:matrix):int`

If the game board is full, this function is used to determine if the O-player won, or if the game is a draw. Notice that if the game board is full, the X-player could not have won.

- `fun eR((board,column,row,xs,os): matrix*int*int*int*int):int`

This function is used to determine who won the game, if the game ended without the game board being full.

- `fun endResult(board:matrix):int`

This function determines how the game ended, either by calling *whoWon* or *eR*, depending on the state of the board.

- `fun gameOver(board:matrix):bool`

This function checks if the game has ended or not.

- `fun columnNotFull((board,column):matrix*int):bool`

This function checks if the column with index *column* is full.

- `fun fFR((board,column,row):matrix*int*int):int`

This function finds the lowest vacant row in the column with index *column*, if any.

- `fun findFreeRow((board,column):matrix*int):int`

This is a wrapper function for the *fFr* function described above.

- `fun fUT((board,column,row):matrix*int*int):int`

This function returns the row index of the upmost occupied square in the column with index *column*, on the game board.

- `fun findUpperToken((board,column):matrix*int):int`

This is a wrapper function for the *fUT* described above.

- `fun placeToken((board,column,x):matrix*int*bool):matrix`

This function places a disc in the column with index *column*, and returns the resulting game board.

- `fun removeToken((board,column):matrix*int):matrix`

This function removes the upmost disc in the column with index *column*, and returns the resulting game board.

- `fun minimax((board,x,depth,cutoff):matrix*bool*int*int):int`

This is the function used by ADATE's opponent to find the current best move. The function is heavily recursive, and iterates the game tree by repeatedly calling itself. When using this function to find the best move facing a particular board, *board* is this board configuration, *x* is true if the calling player plays X, otherwise it is false, *depth* is 0, and *cutoff* is the deepest level in the game tree the search is permitted to reach. When/if the search reaches the cutoff level, the cutoff evaluation function *eval* is applied at the cutoff point. The integer value returned from *minimax* is the recommended column to put the next disc in, depending on the cutoff depth and quality of the evaluation function.

- `fun fminimax((board,x,depth,cutoff):matrix*bool*int*int):int`

This is the function used by the ADATE-player to find the current best move. It is identical to the *minimax* function above, except that it uses the dynamic *f* function rather than the static *eval* function at cutoff points in the game tree.

- `fun chooseMove((board,x,cutoff,fPlayer):matrix*bool*int*bool):int`

This function calls *minimax* or *fminimax*, depending on whether the moving player is the ADATE-player or its opponent.

- ```
  fun play((board,fPlaysX,count,xTurn,fSLevel,oSLevel):
      matrix*bool*int*bool*int*int):gameResult
  ```

This recursive function starts a game of Connect4, and plays it to the end. *Board* is the starting board state, *fPlaysX* is true if ADATE plays X, otherwise false, count is an accumulating parameter counting the number of moves made in the game, *xTurn* is true if it is the X-player's turn to move, otherwise false, *fSLevel* is the cutoff level for the tree search for the ADATE-player, and oSLevel is
  ```

the cutoff level for the tree search for ADATE's opponent.

4.1.1.3. The main function

In the case of this specification file, the *main* function looks like this:

```
fun main(input: mainInput ) : mainReturn =
 case input of mainInput(board,fPlaysX,xTurn,fSearchLevel,opponentSearchLevel)=>
 case play(toMatrix(board),fPlaysX,0,xTurn,fSearchLevel,opponentSearchLevel) of
   gameResult(c,r) =>
     case fPlaysX of
       true =>
         (case r = 100000 of
           false => mainReturn(false,0)
           | true => mainReturn(true,c))
       | false =>
         (case r = ~100000 of
           false => mainReturn(false,0)
           | true => mainReturn(true,c))
```

Each call of the *main* function is supplied with an argument; an instance of the *mainInput* data type. *MainInput* includes, as mentioned, a starting board, as well as information about which player is the ADATE-player, whose turn it is to move, as well the cutoff depth level for the ADATE player and its opponent.

Given these starting conditions, *main* initiates a game of Connect4, which is played to completion. *Main* does this by calling the function *play*, starting at the board configuration given by *board*. When the game is done, the result is returned from *play* as an instance of the *gameResult* data type. This game result consists of an integer value representing the number of moves it took for the game to end, as well as an integer value indicating the result of the game. If the X-player won, this value is 100000, if the O-player won, this value is -100000, and if the game is a draw, this value is zero. If the game result shows that the ADATE-player won, the *main* function returns an instance of the data type *mainReturn* containing the value true, along with the number of moves it took for the ADATE-player to win. If, however, the ADATE-player lost or the game was a draw, the *main* function returns an instance of the data type *mainReturn* containing the value false. In this case, the number of moves is ignored, and a value of zero is returned. The return value from *main* is evaluated in the following way: If *main* returns true, the currently evaluated synthesized function is said to have gotten the current input example correct. If *main* returns false, the function has not gotten the current input example correct.

For any synthesized function that is to be evaluated, the procedure described above is performed for each input example. In this way, the number of wins over the set of input examples, indicates the fitness of a given synthesized evaluation function. The more games won by a function, the higher the fitness.

4.1.1.4. The input examples

The starting boards are included in the file as lists of lists, the empty board being represented like this:

```
[
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ]
]
```

We have included the empty board (1), as well as all possible board configurations up to and including search depth 2. This means all possible first moves (7), as well as all possible second moves resulting from the first moves (49). This makes up a total of 57 starting boards. The rest of the boards are picked at random from a transcript obtained by making two random-playing agents playing 100.000 games against each other.

4.1.2. Running the specification

Having prepared the ADATE specification file, it is time to run it. Figure 9 shows a simplified function call chain, illustrating the process of evaluating a synthesized function. Only the central functions of the Connect4 playing system are included in the illustration.

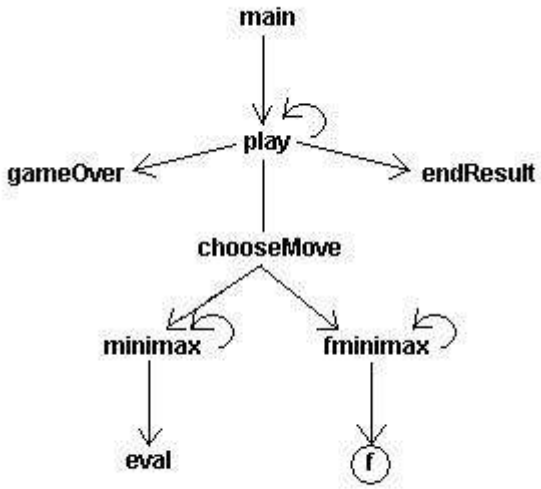

Figure 9. Simplified function call chain for Connect4 playing system in ADATE specification file.

As mentioned, *main* is called with each of the input examples. *Main* calls the recursive function *play* with each input example. *Play*'s task is to play out a game of Connect4, and return the result of the game. Each time *play* is called with a board, it checks the board to see if the game is over. If it is, it finds the end result by calling *endResult*, and returns up the call stack. If the game is not over, *play* places a disc on the board. *Play* knows where to put the disc by calling *chooseMove* with information about whose turn it is. Based on this information, *chooseMove* calls *fminimax* if it is the ADATE-player's turn to move, and *minimax* if it is ADATE's opponent's turn to move. In either case, the minimax algorithm checks all moves and counter moves down the game tree to the specified cutoff depth, by recursively calling itself. As mentioned, in the cases where the cutoff depth is reached before a leaf in the game tree is reached, which is most of the time, the cutoff evaluation function is applied. The *f* evaluation function used by *fminimax* is encircled in the diagram above, indicating that it is the core and product of ADATE's activity. When the *minimax*/*fminimax* function has backtracked up the call stack after reaching its cutoff depth, *chooseMove* can return the recommended next move to *play*, which subsequently places the disc in the recommended column. If the move returned was obtained by calling fminimax, the disc was placed by the ADATE-player, otherwise it was placed by its opponent. *Play* then calls itself with the updated board. This continues in a circular fashion until the game is over, and *play* returns the result of the game.

Following the procedure described above, ADATE's task is to synthesize an evaluation function which enables it to play as well as possible against an adversary using the IBEF function, i.e., win as many games as possible. For each of the 155 starting boards the following takes place:

- ADATE plays X using minimax with cutoff depth 1 against the adversary using minimax with cutoff depth 2
- ADATE plays O using minimax with cutoff depth 1 against the adversary using minimax with cutoff depth 2
- ADATE plays X using minimax with cutoff depth 2 against the adversary using minimax with cutoff depth 2
- ADATE plays O using minimax with cutoff depth 2 against the adversary using minimax with cutoff depth 2

This means that each starting board included in the specification file is transformed into 4 input examples. For each sweep of the training examples, a total of 155*4 = 620 games of Connect4 are presented to *play* by *main*, and played to the end. Each time ADATE wins a game, a win is recorded, as well as the number of moves it took for ADATE to win. Draws and losses are both counted as losses. ADATE will, through inductive inference, aim to maximize the number of wins for each sweep of the training examples, as well as minimize the number of moves needed to win, by continuously transforming the evaluation function it uses. In addition to the training examples, 556 validation starting boards are included. These boards are not taken into consideration when transforming the function, but are used to assure that the function is not overfitted to the training examples. When selecting the best function, this is done based on the function's performance over the validation examples. Much like the training examples, 4 games are played starting at each validation board, i.e., starting board, in the manner described above. This means playing 556*4 = 2224 games.

The relative shallowness of depth used in the game-playing process is due to the extensive computation needed to determine the fitness of a synthesized function (playing 620 + 2224 = 2844 games). For each move in each of these games, the moving player searches *n* levels deep from the board it is faced with, meaning that it considers potentially all possible moves/counter moves *n* levels down the game tree. Adding just one depth level means that the moving player considers (in the worst-case scenario) 7 times more board states. Again, this happens for each move in each game. Specifically, in the worst-case scenario, a playing agent considers 7^n board states, where *n* is the cutoff depth used by the search algorithm. As mentioned earlier, this particular ADATE specification is, at the time this was written, the largest specification ever presented to ADATE, and the fitness evaluation requires approximately ten times as much computing time as the previously largest specification.

4.2. Results

After running a hill-climbing algorithm in ADATE for approximately one week, we picked out the synthesized function which performed best over the set of independent validation data, i.e., the set of starting board states independent of the board states used for training. The function was then combined with the minimax algorithm. One week is a very short running time, but due to time constraints, we had to cut it short. Ideally, ADATE should have had about 1000 times more running time with regards to this problem, given the kind of computing resources we had available. Nevertheless, substantial improvement seems to have occurred. The function, which is presented in its original form in Appendix B, looks very complicated at first sight. We have, however, thoroughly analyzed the function, and discovered its structure, described here:

In the following explanation, $board_{x, y}$ denotes the square on the board intersected by the row with index *x*, counted from the bottom, and the column with index *y*, counted from the left. Considering this, the evaluated value of a Connect4 board according to the ADATE-synthesized function is:

$$
\begin{aligned}
&1 \\
&+ \\
&2 * board_{0,1} + board_{0,3} + 12 * board_{0,4} \\
&+ \\
&2 * board_{1,1} + 5 * board_{1,2} + 5 * board_{1,4} + 4 * board_{1,5} + 4 * board_{1,6} \\
&+ \\
&board_{2,1} + 4 * board_{2,2} + 16 * board_{2,3} + 2 * board_{2,6} \\
&+ \\
&12 * board_{3,1} + 6 * board_{3,2} + 14 * board_{3,3} + 12 * board_{3,4} + 11 * board_{3,5} + 2 * board_{3,6} \\
&+ \\
&2 * board_{4,1} + 6 * board_{4,2} + 6 * board_{4,3} + 2 * board_{4,4} + 4 * board_{4,5} + board_{4,6} \\
&+ \\
&2 * board_{5,0} + 2 * board_{5,2} + 2 * board_{5,3} + 2 * board_{5,4} + 2 * board_{5,5} + board_{5,6}
\end{aligned}
$$

In other words, the squares of the Connect4-board are weighted as shown in Figure 10,

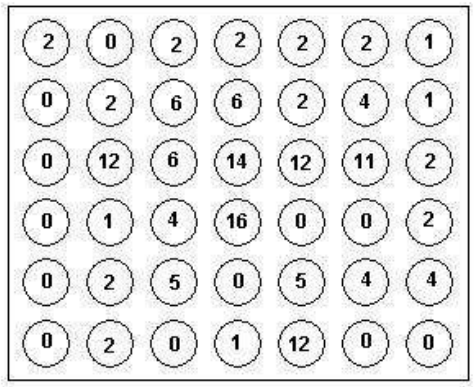

Figure 10. The ADATE-function's weighting of squares.

and a constant of 1 is always added, perhaps reflecting the advantage of being the beginning/maximizing player. Interestingly enough, 12 of the squares on the board are deemed completely unimportant by this evaluation function, and some of these are quite central squares on the board.

4.2.1. The ADATE-function vs. the IBEF-function

When playing X, the ADATE-function emerges as a strong player when facing the IBEF-function. When making the two functions play against each other using equal cutoff depths, up to and including depth 6, the ADATE-function is at a clear advantage at depths 1, 2, 4 and 5. At depth 1 it is superior, winning 100% of the games. At depths 3 and 6 however, it is inferior to the IBEF-function. Aside from this, the results from when ADATE plays X are a bit tricky to analyze, as they seem a bit erratic and lack distinct patterns.

As a contrast, the results are quite clear when the ADATE-function plays O. In this case, the function is particularly strong. Consider this: if we let two random-moving agents play against each other, the X-playing agent will win roughly 55,5% of the games, the O-playing agent will win about 44% of the games, and the remaining 0.5% of the games will be drawn. This indicates the relative advantage of being the X-player. If we play out all cutoff depth-combinations, up to and including depth 6, between the IBEF-function, as X, and the ADATE-function, as O, we get 36 unique game combinations. Considering the winning percentage between the two random-playing agents, the ADATE-function emerges as the clear victor in 31 of these 36 games. It is, when using cutoff depths 2, 3, 4 and 5, able to convincingly beat the X-Playing IBEF-function at all cutoff depths, up to and including depth 6. As an example, the X-Playing IBEF-function with a cutoff depth of 6 wins only 7.7% of the games against the O-Playing ADATE-function with a cutoff depth of 2, which wins 68% of the games.

Considering these results, the ADATE-generated evaluation function represents a distinct

improvement over the original IBEF-function. Some examples of this function's playing ability against the IBEF function are shown in Table 1.

XPlayer/depth	*OPlayer/depth*	*XWins*	*OWins*	*Draws*
ADATE/1	IBEF/1	100,00%	0,00%	0,00%
ADATE/1	IBEF/2	34,00%	65,60%	0,40%
ADATE/2	IBEF/2	61,60%	35,00%	3,40%
ADATE/4	IBEF/4	52,70%	41,00%	6,30%
IBEF/1	ADATE/1	27,40%	71,30%	1,30%
IBEF/4	ADATE/4	11,50%	88,50%	0,00%
IBEF/4	ADATE/2	3,80%	96,20%	0,00%
IBEF/6	ADATE/2	7,70%	68,00%	24,30%

Table 1. Selections from the ADATE-function's winning statistics against the IBEF function.

A more thorough collection of result statistics can be found in Appendix D.

4.3. Further contemplations concerning ADATE

Inspired by this Connect4 example, it is intriguing to imagine possible future uses of the ADATE system. An interesting aspect of the Connect4 example, is that ADATE is used to improve and refine a single module of a larger system. The minimax algorithm ADATE uses, is fixed and given beforehand. This minimax algorithm, however, cooperates with the evaluation function which is constantly changed by ADATE.

In a general setting, ADATE could be used to optimize individual modules of any(?) modular software system, where the fitness of each synthesized function depends on the the performance of the given software system as a whole. Examples of target domains could include production chain systems, elevator dispatch systems, automatic military defense systems (provided the system could run accurate simulations of real-world situations) and so on. Furthermore, one could imagine optimizing all modules of a modular system, one module at a time, aiming towards the best possible overall system behavior. ADATE could even be used to optimize its own source code, using this approach. This is a truly fascinating notion, tapping into the field of self-improving software.

5. Reinforcement Learning

The second machine learning technique we applied to the game-playing problem is Reinforcement Learning (RL) [Sutton & Barto, 1998]. This choice of method is largely inspired by Gerald Tesauro's Backgammon playing program, *TD-Gammon* [Tesauro, 1995]. In the following sections, 5.1 – 5.4, Reinforcement Learning theory is presented. Section 5.5 presents a preliminary RL experiment concerning Tic-Tac-Toe, while the the main RL experiment is presented in sections 5.6 and 5.7.

5.1. Overview

Supervised learning is a machine learning technique suitable for a large group of problems, ranging from spam filters to visual recognition systems. Supervised learning is, as the name implies, dependent on an external teacher/supervisor, in the following way: The teacher supplies the learning algorithm with training data, which in turn consists of input data, paired with desired output data (target data). In this way, the teacher tells the learning algorithm what the desired output should be, given the presented input. The learning algorithm's task is then to generalize a mapping function from these input-output pairs. The goal is to obtain a mapping function which will be able to give correct output for all input in its domain, including input the learning algorithm has not encountered before.

Supervised learning has proven to be successful at many tasks. However, there are problems supervised learning is not as suited for. In these problems, it might be difficult or impossible for a teacher to provide correct output to match the input. An example of this class of problems is learning to play a board game. In this particular case, it would be hard for a supervisor to say which move (output) is the correct response to a given board state (input). It would probably be harder still to pick representative board states to include in the training process, that is, board states which would make it easy for the training algorithm to generalize.

In problems of this kind, a technique called Reinforcement Learning has shown to be a fitting approach. Reinforcement Learning draws inspiration from a natural organism's independent learning through interaction with its environment. An obvious example of this is the human being, who by means of its direct sensori-motor connection to its environment, is able to obtain information about cause and effect through a process of trial-and-error.

5.1.1. The basics

The general Reinforcement Learning system consists of three basic components; an *agent*, an *environment*, and a *goal*. The agent interacts with the environment in order to learn how to reach its goal. In a board game such as chess, the agent would be the game player, the environment would

consist of the chess board and the opposing player, and the goal would be to defeat the opposing player.

The agent is the entity which is faced with the task of learning to make decisions through interacting with its environment. This implies that the agent must be able to sense the *state* of the environment, as well as take *actions* that affect the state of the environment, based on the current state and the agent's goal.

At time step **t**, the environment will be in a state $\mathbf{s_t} \varepsilon \mathbf{S}$, where **S** is the set of all possible states. (**S** may be infinite). In a chess analogy, the state of the environment would consist of the positions of the individual pieces on the board. This state is perceived by the agent, which has a set of actions$\mathbf{a} \varepsilon \mathbf{A(s_t)}$ to choose from, where $\mathbf{A(s_t)}$ denotes all possible actions available in state $\mathbf{s_t}$. When the agent has chosen an action, the environment responds with a new state $\mathbf{s_{t+1}}$, as well as a numeric *reward* $\mathbf{r_{t+1}}$. The reward function is external to the agent, and is a mapping from states to numeric values. Desirable environment states are mapped to high rewards, compared to non-desirable states. The goal of a chess-playing agent would be to reach a state of the environment in which it had won. In chess, we could assign a winning state a reward of 1, a losing state a reward of -1, and draws and all other states a reward of 0. In other words, we indicate which states we want the agent to reach by associating them with high rewards.

5.1.2. Maximizing reward

The general goal of an agent is to maximize the cumulative reward it receives over time. This statement is a bit imprecise. What aspect of a given reward sequence do we wish to maximize? As will be explained in the following, this often depends on the task at hand.

Consider the reward sequence $\mathbf{r_{t+1}} + \mathbf{r_{t+2}} + \mathbf{r_{t+3}} + \mathbf{...}$

This represents a sequence of rewards received by an agent following time step **t**. This sequence may or may not be infinite, depending on the task. We can define a return function $\mathbf{R_t}$ over this sequence, in a way that suits our problem. The agent's goal will then be to maximize the expected return, where the return is denoted $\mathbf{R_t}$.

Reinforcement Learning tasks can be divided into *episodic tasks* and *continual tasks*. An episodic task is a task which has a clearly defined end point, or a terminal state. When this terminal state is reached, the task is over, and a new episode of the task can begin. An example of this class of task is a traditional board game (again, like chess), where an episode corresponds to a game session. That is, an episode is started with the first move, and ends when a terminal state is reached, that is, when one of the players wins, or the game is drawn. In this case, the reward sequence for an agent will look like this:

$$\mathbf{r_{t+1}} + \mathbf{r_{t+2}} + \mathbf{r_{t+3}} + \mathbf{...} + \mathbf{r_T}$$

Given an episodic task, it is usually appropriate to define the return function $\mathbf{R_t}$ as a simple sum of the individual rewards:

$$\mathbf{R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T}$$

Given this $\mathbf{R_t}$, a winning game in chess would give a cumulative reward of 1, losing would give a cumulative reward of -1, and a draw would correspond to 0 (according to the reward assignments chosen earlier in the text).

Continual tasks are tasks which do not break into episodes, but go on without limit. An example of this is a continuous control task, for instance controlling a nuclear reactor. In this case there is no identifiable terminal state, and the sum of rewards could easily amount to infinity. In continual tasks a usual approach is to use a concept called *cumulative discounted reward*:

$$\mathbf{R_t = r_{t+1} + \gamma\, r_{t+2} + \gamma^2\, r_{t+3} + \gamma^3\, r_{t+4} + ... = \sum_{k=0}^{\infty} \gamma^k\, r_{t+k+1}}$$

where γ is a constant $0 <= \gamma <= 1$, called the *discount rate*.

The discount rate determines how much weight future rewards are given at the current time step. A discount rate of 0 only takes into account the immediate reward, that is, the reward one time step into the future. An agent trying to maximize the discounted cumulative reward with a discount rate of 0, would always choose the action whose result state yielded the highest immediate reward, ignoring any future rewards past this point. A higher discount rate makes the associated agent more far-sighted.

5.1.3. Value functions

The concept of *value* is essential in Reinforcement Learning. Estimating value functions is what constitutes the *learning* in Reinforcement Learning. Whereas a reward indicates a state's immediate desirability, a state's value indicates the expected cumulative reward (return) the agent can obtain following this state. Of course, this depends on which choices the agent will take facing future states. In other words, the *policy* of the agent, denoted $\boldsymbol{\pi}$. A policy is a function which maps state **s** and action **a** to the probability of choosing action **a** when in state **s**, given policy $\boldsymbol{\pi}$. In other words:

$$\boldsymbol{\pi}\,(\mathbf{s,a}) = \mathbf{Pr\{\, a = a_t \mid s = s_t \,\}}$$

A value function **V** is therefore associated with a policy $\boldsymbol{\pi}$, and maps states to values. $\mathbf{V^{\pi}(s)}$ denotes the expected return obtained by the agent by starting at state **s** and following policy $\boldsymbol{\pi}$ thereafter. Consequently, $\mathbf{V^{\pi}}$ is called the *state-value function for policy* π. $\mathbf{V^{\pi}(s)}$ is defined as follows:

$$V^{\pi}(s) = E_{\pi} \{ \sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} | s_t = s \}$$

where E_{π} is the expected value providing the agent follows policy π. (In the case of non-discounted cumulative return: k = 1).

Similarly, we can define $Q^{\pi}(s,a)$ as the expected return obtained by the agent by choosing action **a** in response to state **s**, and following policy π thereafter. $Q^{\pi}(s,a)$ is called the *action-value function for policy π*. $Q^{\pi}(s,a)$ is defined as follows:

$$Q^{\pi}(s,a) = E_{\pi} \{ \sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} | s_t = s, a_t = a \}$$

Now, V^{π} (or equivalently, Q^{π}), is involved with policy π in a kind of chicken-and-egg relationship. That is, they depend on each other in a recursive manner. To understand it, consider this:

- $V^{\pi}(s)$ equals the expected return obtained by the agent by starting at state s and following policy π thereafter.
- Policy π chooses action a_t from state s_t by picking the action which results in the s_{t+1} with highest $V^{\pi}(s_{t+1})$

From this, it is easy to understand that V^{π} (or equivalently, Q^{π}), is really the core of the entire learning system, and that estimating it is really what the learning task is all about. V^{π} can be represented in a number of ways. If the environment's state set is sufficiently small, V^{π} can be represented by a table, where each entry consists of a separate state **s** with its corresponding value $V^{\pi}(s)$. Often however, the environment's state set is so large that this approach is infeasible, due to memory constraints. In this case V^{π} can be represented by some form of parameterized function, for example a simple linear function or an advanced Artificial Neural Network (this is an example of Reinforcement Learning utilizing Supervised Learning). Methods for estimating V^{π} (learning algorithms) follow later in this report.

5.1.4. Optimal policies

Intuitively, a policy π is better than or equal to a policy π' if its value function V^{π} gives a greater value than or equal value to $V^{\pi'}$ for all states **s**. That is,

$$\pi >= \pi' \text{ if and only if } V^{\pi}(s) >= V^{\pi'}(s) \text{ , for all } s \in S$$

Consequently, there exists at least one policy which is greater than or equal to all other policies.

This policy is *an optimal policy*, denoted $\boldsymbol{\pi^*}$. There may be more than one optimal policy, but they all share the same state-value function **V***. As could be expected, this is called the o*ptimal state-value function*. Formally, **V*** can be expressed as follows:

$$\mathbf{V^*(s) = \max_{\pi} V^{\pi}(s) \text{ , for all } s \in S}$$

The optimal state-value function gives the expected return by following the optimal policy from state **s**. Of course, there also exists an *optimal action-value function*, **Q*(s,a):**

$$\mathbf{Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \text{ , for all } s \in A(s)}$$

The optimal action-value function gives the expected return by choosing action **a** in response to state **s**, and thereafter following the optimal policy. **Q*** is defined in terms of **V*** in the following way:

$$\mathbf{Q^*(s,a) = E\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s,\ a_t = a\}}$$

(Again, if the return function is a pure summing function; $\gamma = 1$)

When the optimal value function **V*** has been found, the task has been solved, providing we are in possession of a model of the environment. If these conditions are present, this means that the optimal policy also has been found. Now, when the agent faces a state **s**, it first determines the possible successor states of **s,** using the environment model. Then it only has to examine the **V*** values of **s**'s successor states, and perform the action which results in the successor state with the highest **V*** value. This means following the optimal policy. Better still, the agent can apply **Q*(s,a)** to the current state **s** and all possible actions **a ε A(s)**, and pick the action a which gives the highest **Q*** value. In this way, no lookahead and consequently, no model, is necessary at all.

In reality, the optimal policy is almost never found, due to computational cost and memory constraints. This is especially true when facing learning tasks with environments with very large state sets. Chess is a good example of this. This does not mean that a trained agent cannot perform well at its task, however. Although the learned policy might not choose optimal actions from the complete state set, it may perform well over the part of the state set which is most commonly encountered. It is probably of minor importance that an agent chooses poor actions facing states that are rarely ever seen.

5.2. Methods for solving the Reinforcement Learning problem

The three main methods for solving the Reinforcement Learning (RL) problem are Dynamic Programming (DP), Monte Carlo (MC) methods, and Temporal Difference (TD) learning. TD is a central method in RL today, and has been successfully applied to a number of tasks, Gerald Tesauro's world class backgammon player TD-Gammon being an excellent example of this. TD-Gammon is described in [Tesauro, 1995].

Dynamic Programming is a somewhat limited method in practical cases, although studying it can provide a good understanding of general RL methods. What makes DP limited is its need for a complete model of the task's environment, i.e., complete knowledge of the probability of each state's returns and possible successor states, complete knowledge of the probability of the opponent's counter moves, and so on. Such a model is often not available in real-life tasks. An additional limitation of DP springs from the fact that it iterates over the *entire* state set (all possible states) of the environment a potentially large number of times, which makes this approach computationally infeasible in many cases.

In contrast to Dynamic Programming, Monte Carlo Methods learn by interacting directly with an environment or a simulation of the environment. Learning occurs along the agent's state-transition path, and this avoids the potentially enormous computational cost of iterating over all the states. MC methods do not require a complete model of the environment either, as it *samples* rewards from states/actions instead of knowing the exact probability distributions in advance.

Temporal Difference learning incorporates ideas and aspects from both Dynamic Programming and Monte Carlo methods, and is often said to gain the best from both worlds. In the following sections, Dynamic Programming, Monte Carlo methods and Temporal Difference learning will be presented in further detail. First however, let us consider an important general principle called *Generalized Policy Iteration*, which almost all reinforcement learning methods can be described in terms of.

5.2.1. Generalized Policy Iteration

Generalized Policy Iteration (GPI) can be described as the process of iteratively improving a policy, i.e., moving it towards an optimal policy. This process consists of two sub-processes, called *Policy Evaluation* and *Policy Improvement*, which can be said to cooperate in order to reach a mutual goal; an optimal value function and an optimal policy.

5.2.1.1. Policy Evaluation

Policy Evaluation is the process of estimating a value function $\mathbf{V}^{\pi}$, given a policy $\boldsymbol{\pi}$. In other words, the question to ask is: "What does the value function of this policy look like, making it act the way it does?" The way this is done in detail, is largely dependent on the RL method being used. Policy Evaluation is a core concept in RL, and is usually what differs most between the different RL methods.

5.2.1.2. Policy Improvement

Policy Improvement is the process of making a policy greedy according to a value function. In other words, to make the policy, when faced with an environment state, attempt to reach successor states which give the highest value according to the value function, and thereby improving the policy.

Generalized Policy Iteration consists of these two processes (Policy Evaluation and Policy Improvement) interacting with each other. A common way for them to interact, is to alternate in a circular motion. That is, when the first process stops, the second one starts. When that process stops, the first one starts again, and so on. This can be visualized in the following way:

$$\boldsymbol{\pi}_0 \xrightarrow{E} \mathbf{V}^{\pi 0} \xrightarrow{I} \boldsymbol{\pi}_1 \xrightarrow{E} \mathbf{V}^{\pi 1} \xrightarrow{I} \boldsymbol{\pi}_2 \xrightarrow{E} \mathbf{V}^{\pi 2} \xrightarrow{I} \boldsymbol{\pi}_3 \xrightarrow{E} \mathbf{V}^{\pi 3} \xrightarrow{I} \ldots \xrightarrow{I} \boldsymbol{\pi}^* \xrightarrow{E} \mathbf{V}^*$$

where E denotes Policy Evaluation, I denotes Policy Improvement, a numerical subscript indicates a time step, and $\boldsymbol{\pi}^*$ and $\mathbf{V}^*$ are the optimal policy and optimal value function, respectively.

5.2.2. Dynamic Programming

Generally speaking, Dynamic Programming is an algorithmic technique in which an optimization problem is solved by recombining solutions to subproblems, when the subproblems themselves may share sub-subproblems (taken from http://thefreedictionary.com). When Dynamic Programming is utilized as a Reinforcement Learning method, it requires a complete model of its target environment. In addition to this, its computationally expensive Policy Iteration makes it impractical in many cases. Nonetheless, its theoretical foundation is highly relevant in RL.

5.2.2.1. Policy Evaluation in DP

We remember from earlier that Policy Evaluation is the process of estimating the value function of a policy. How is Policy Evaluation performed in DP? In order to answer this properly, let us first define two quantities $\mathbf{P^a_{ss'}}$ and $\mathbf{R^a_{ss'}}$. $\mathbf{P^a_{ss'}}$ denotes the probability of reaching state **s'** in the next time step, when taking action **a** in state **s**. $\mathbf{R^a_{ss'}}$ denotes the expected immediate reward when reaching state **s'** from taking action **a** in state **s**. Formally:

$$\mathbf{P^a_{ss'} = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}}$$

and

$$\mathbf{R^a_{ss'} = E\{r_{t+1} \mid s_t = s, at = a, s_{t+1} = s'\}}$$

In addition to this, we recall that $\boldsymbol{\pi}$**(s,a)** denotes the probability of selecting action **a** from state **s** under policy $\boldsymbol{\pi}$. Now, in order to estimate the value function of the policy, we iterate over the entire state set, updating **V(s)** for one state at a time, according to the following equation:

$$\begin{aligned}\mathbf{V^\pi(s)} &= \mathbf{E_\pi\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \mid s_t = s\}} \\ &= \mathbf{E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s\}} \\ &= \mathbf{\Sigma_a\, \pi(s,a)\, \Sigma_{s'}\, Pa_{ss'}\, [Ra_{ss'} + \gamma V\pi(s')]}\end{aligned}$$

As we can see, the value of state **s** (under policy π) is updated in the following way (expressed in pseudo code):

```
( x = 0  y = 0 )
for each possible action a from state s {
     for each possible result state s' from taking a from s {
          y += P^a_ss' * (R^a_ss' + γV^π(s'))
     }
     x += π(s,a) * y
     y = 0
}
V(s) = x
```

This is called a *full backup*, since the value of a state **s** is assigned to be the expected return from state **s** under policy π, considering *all* possibilities from state **s**, weighted by their probabilities of actually happening. In addition to this, we see that the estimate **V(s)** is based on the estimates of its successor states **s'**. In other words, an estimate based on another estimate. This is called *bootstrapping*.

The entire state set is updated in this fashion, in what is called a *state sweep*. At this point, we assume that the value function **V** is implemented as a table or array, with a separate entry for each state **s**, containing the estimated value of **s**. If we keep the old state values and the new state values in separate arrays, **V(s)** will always be updated with respect to the old values of its successor states. If we only maintain one array however, and replace the contents of its entries as we go, **V(s)** might be updated with respect to *new* values of its successor states, depending on the sequence of states visited in the sweep. This latter approach hos shown to converge faster than the two-array approach, and is often used.

Many state sweeps may be required before $\mathbf{V}^{\pi}$ is estimated with satisfying accuracy. It is usual to terminate the policy evaluation when the maximum difference between **V(s)** (for all **s**) for two consecutive time steps falls between some small positive number. In order to clarify this approach, named *iterative policy evaluation*, consider the following block of pseudo code (adapted from [Sutton & Barto, 1998]). ($\mathbf{S}^{+}$ denotes all states *including* terminal states (states which have no successor states). **S** denotes all states *except* terminal states. All terminal states *must* be initialized to 0 at the beginning of the algorithm).

```
let π be the policy to be evaluated
let V(s) be 0 for all s ε S+
termLimit = a small positive number

do {
     maxDiff = 0
     for all s ε S {
          oldV = V(s)
          V(s) = Σa π(s,a) Σs' Pass' [Rass' + γVπ(s')]
          maxDiff = max(maxDiff, abs(V(s)-oldV))
     }
}
while (maxDiff > termLimit)

output: V -> an estimate of Vπ
```

5.2.2.2. Policy Improvement in DP

When we have estimated a policy's value function, we can use this estimate to improve the policy. This process is called Policy Improvement. The policy is improved by making it greedy with respect to the value function.

Up to this point we have considered stochastic policies on the form $\pi(s,a)$, which indicates the probability of taking action **a** from state **s** under policy π. In order to simplify the further presentation of Policy Iteration, we will from this point on deal with deterministic policies instead, that is policies on the form $\pi(s) \rightarrow a$, which maps a state to an action. Although this makes the further presentation clearer, policy iteration easily extends to deal with the stochastic case.

Now, in order to make a policy π greedy with respect to a value function, we iterate through the state space once. For each state **s** encountered, $\pi(s)$ is updated as follows:

$$\pi(s) = \mathbf{argmax}_a \Sigma_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V^\pi(s')]$$

We simply assign to $\pi(s)$ the action **a** from **s** which gives the maximum expected return in terms of the value function; we make it greedy.

5.2.2.3. Policy Iteration in DP

Policy Iteration in DP describes the process of alternating between Policy Evaluation and Policy Improvement in order to converge to the optimal policy and optimal value function. In its simplest form, this consists of alternating between complete Policy Evaluation (possibly multiple state sweeps) and Policy Improvement. The complete algorithm in pseudo code (adapted from [Sutton & Barto, 1998]):

```
let π be the policy to be evaluated
let V(s) be 0 for all s ε S+, let π(s) be an arbitrary a for all s ε S
termLimit = a small positive number

do {
     //POLICY EVALUATION
     do {
          maxDiff = 0
          for all s ε S {
               oldV = V(s)
               V(s) = Σs' Pπ(s)ss' [Rπ(s)ss' + γV(s')]
               maxDiff = max(maxDiff, abs(V(s)-oldV))
          }
     }
     while (maxDiff > termLimit)
     //POLICY IMPROVEMENT
     policy_stable = true

     for all s ε S {
          oldA =  π(s)
          π(s) = argmaxaΣs' Pass' [Rass' + γV(s')]
          if ( π(s) != oldA ) {
               policy_stable = false
          }
     }
}
while( policy_stable == false )

output :  π, the optimal policy
```

The algorithm starts with Policy Evaluation, and performs as many state sweeps as is necessary for the variable maxDiff to fall beneath the maximum difference allowed. It then proceeds to Policy Improvement, which updates (improves) the policy for all states **s ε S**. For each state, we check if the improved policy chooses a different action than the old policy did. If this is the case, even when it comes to just one single state, the entire Policy Iteration starts over with the improved policy as the target policy. On the other hand, if the improved policy is identical to the old policy (chooses the same actions for all states), the improved policy can in fact be proven to be the optimal policy. The algorithm terminates.

5.2.2.3.1. Alternative GPIs in DP

As mentioned, for each iteration of Policy Iteration, multiple iterations of Policy Evaluation may (and probably will) occur. Since each Policy Evaluation iteration requires a state sweep, this gets computationally expensive. It is therefore common to shorten the Policy Evaluation step in different ways. One approach is to require only one iteration of Policy Evaluation between each policy improvement. This is called Value Iteration, and requires far less computation and time, without sacrificing the guarantee of convergence to the optimal policy. Consider this block of pseudo code to illustrate Value Iteration (adapted from [Sutton & Barto, 1998]):

```
let V(s) be 0 for all s ε S+
termLimit = a small positive number

//POLICY EVALUATION
do {
     maxDiff = 0
     for all s ε S {
          oldV = V(s)
          V(s) = max_a Σ_s' P^a_ss' [R^a_ss' + γV(s')]
          maxDiff = max(maxDiff, abs(V(s)-oldV))
     }
}
while (maxDiff > termLimit)
//POLICY IMPROVEMENT
for all s ε S {
     π(s) = argmax_a Σ_s' P^a_ss' [R^a_ss' + γV(s')]
}
output:  π
```

In cases where even a single state sweep is unfeasible, because of the size of the state set, Asynchronous Dynamic Programming (ADP) might be utilized. The general term ADP includes methods which back up state values in no particular order. Some states may be backed up many times before other states are backed up at all. One might for example wish to focus the attention on a certain area of the state set, and pay less attention to the rest. This will lead to an unequal number of backups for the different states. ADP is a large class of methods, as one can vary the details of the backup "plan" after one's desires. The convergence of these methods to the optimal policy might be uncertain.

5.2.2.4. DP in game playing

How might Dynamic Programming be applied to a game playing task? Due to computational demands (entire state sweeps), DP will in most cases be limited to solving tasks with modest state sets. Tic-Tac-Toe is such a game; with a board consisting of 9 squares and only two types of pieces, the state set consists of a few hundred thousand states. This seems to be a suitable task for DP. In the following section, let us briefly consider how DP methods could be used to get an agent to learn to play Tic-Tac-Toe.

DP requires a perfect model of the environment. In the case of a game, this means a complete probability model of its adversary. Such a model is usually hard to acquire explicitly, especially when facing human opponents. Human beings usually play using their intuition to a large degree, and to assign probabilities to a human's choices when in a state is probably a daunting task, particularly over the entire state set of Tic-Tac-Toe, even though it is a simple game. However, assuming that we are in possession of such a model describing our opponent, whoever or whatever the opponent may be, DP can be used to learn an optimal policy against the opponent.

In the case of games, it is natural to let a state **s** represent a board configuration, and so the state set of the environment consists of all possible board configurations of Tic-Tac-Toe. An action **a** represents a move; in the case of Tic-Tac-Toe this means placing an X or O on a vacant square on the board. Terminal states (states where the game session is over) are assigned a numeric reward, according to their desirability to the learning agent. This reward assignment is important, because it indicates what we want the agent to learn. Desirable states are given positive rewards, and non-desirable states are given negative rewards. In the case of Tic-Tac-Toe, a sensible approach would be to assign a reward of 1 to all terminal states which represent a win for the learning agent, and a reward of -1 to all terminal states which represent a loss. Terminal states representing a draw between the learning agent and its opponent, are assigned a reward of 0. All non-terminal states are also assigned rewards of 0.

The value function **V(s)** must also be initialized, for all states **s**. The non-terminal states can be given arbitrary values, but it is important to assign the terminal states a value of 0. The reason for this is that the value **V(s)** indicates the expected accumulated reward following state **s**. If **s** is a terminal state, then no rewards will be received after this state, since the game is over when reaching it. The final step of the preparation is to initialize the policy the agent is going to use and improve. This policy might also be initialized arbitrarily.

When the environment is set up properly, the "learning" process can begin. This is done by applying some sort of GPI, as described earlier. We might use regular Policy Iteration, Value Iteration or some sort of Asynchronous Dynamic Programming, for instance. In our case, let us consider regular Policy Iteration. The first thing to do then, is to try to estimate the value function corresponding to the current policy, by making (an) entire sweep(s) of the state set. In this context, a state sweep means going through all possible legal board configurations of Tic-Tac-Toe. As we remember, the value of a state s is updated in terms of the following statement

$$\mathbf{V(s) = \Sigma_{s'}\, P^{\pi(s)}_{ss'}\, [R^{\pi(s)}_{ss'} + \gamma V(s')]}$$

For each possible legal board configuration (state **s**), we determine which action a that our policy $\boldsymbol{\pi}$ would respond with. Next we determine which resulting states **s'** this action could produce, and the probability for each of these successor states occurring. These successor states represent game board configurations immediately following the opponent's move. This is the reason why there might be multiple possible successor states. The probability of each of them occurring, is given by the environment model describing the opponent (If the opponent's policy is deterministic, as ours is, there is only one possible successor state).

Next, we go through each possible successor state **s'** (successor game board), sum the immediate reward of **s'** with the estimated state value of **s'**, and multiply this sum by the probability of **s'** being the successor state of **s** given **a**. The weighted sums of these possible successor states are then added together. The resulting sum is the new state value of state **s**. When all game configurations have had their values updated once in this way (one sweep), we check if it is necessary to perform additional sweeps. If it is, the approach above is repeated. If it isn't, we move on to the task of improving the policy according to this newly updated value function.

This improvement step is done by making one sweep through the state set, i.e., all possible legal game board configurations. The objective is to make the improved policy choose the move which gives the highest expected cumulative future reward. In the case of Tic-Tac-Toe, the resulting update statement can be expressed like this:

$$\mathbf{\pi(s) = argmax_a \Sigma_{s'}\, P^{a}_{ss'}\, [R^{a}_{ss'} + \gamma V(s')]}$$

This means: Presented with a game board configuration, make the move that gives the highest sum of immediate reward and value of the board configuration resulting from that move. For all possible legal actions **a** (moves) in response to state **s**, determine all possible successor states **s'**, and sum the immediate reward of **s'** with the state value of **s'**. Multiply this sum with the probability of **s'** being the successor state of **s** given **a**. Add together the weighted sums of all the possible successor states **s'** given action **a**. The action **a** which gives the highest result in this way, is the new action **a** in response to state **s**, according to policy $\boldsymbol{\pi}$. This is called making the policy greedy.

This update is made for all legal board configurations. When this is done, we check to see if the policy is altered in any way from the previous policy. If it is, we start all over with estimating the value function again, but this time with the new and improved policy. If the new policy is the same

as the old one, we are done. The resulting policy should be able to play optimally against an opponent described by the environment model used by the Policy Iteration algorithm.

5.2.3. Monte Carlo methods

Monte Carlo methods are algorithms for solving various kinds of computational problems by using random numbers (taken from http://thefreedictionary.com). In the specific case of Reinforcement Learning, Monte Carlo methods are methods which learn from actual interaction with an environment. As a contrast to Dynamic Programming, MC methods do not require a complete model of the environment, which makes it practically applicable in a larger number of tasks, since an explicit model is often not available. Instead MC methods learn from experience, utilizing *sample backups* (explained later) instead of DP's full backups.

MC methods can also be applied to a simulation of an environment. In this case a partial model is required, to determine which state-action pairs lead to which successor states. The explicit probability distributions of the environment, however, are not required to be included in the model in this case.

5.2.3.1. Policy Evaluation in MC

Given its nature, Monte Carlo Policy Evaluation is defined for episodic tasks only (tasks with identifiable episodes). This is because in MC the value of a state, **V(s)**, is updated (backed up) using the entire *sample return* following state **s** in an episode (sample backup), as opposed to being backed up using the estimate of the successor states. In other words; unlike DP, Monte Carlo methods do not use bootstrapping. Therefore, an entire episode has to be taken to its terminal state before **V(s)** can be backed up. Furthermore, **V(s)** is only updated for the states **s** visited in the episode. Roughly speaking, MC Policy Evaluation (estimation of the value function of a policy) is performed in the following way:

For all states **s** in the entire state set, an average of previously received returns after **s** is maintained. An episode is played out using the policy π to be evaluated. An episode in this case is a set of transitions from state to state until a terminal state is reached:

$$s_0 \xrightarrow{a0} s_1 \xrightarrow{a1} s_2 \xrightarrow{a2} s_3 \xrightarrow{a3} s_4 \xrightarrow{a4} s_5 \xrightarrow{a5} \dots \xrightarrow{aT-1} s_T$$

When the episode is done, we go through each state encountered in the episode, and determine the returns ((discounted) cumulative rewards) received after visiting the states in this episode. These returns are subsequently used to update the average of previously received returns for the states in question. For a state **s**, this average converges to the true **V(s)** as the number of episodes approaches infinity.

But what happens if the same state **s** is encountered multiple times during a single episode? In this

scenario we have two alternative approaches to choose from. One approach is to compute the average of previously received returns from *all* visits to a state **s**. This is called every-visit Monte Carlo. The other approach is to compute the average from only the first visits to a state **s** in each episode. This is called first-visit Monte Carlo. Both methods will give the correct result, but first-visit MC is studied more extensively, and is usually the approach of choice.

To make this explanation clearer, consider this block of pseudo code (first-visit MC):

```
let π be the policy to be evaluated
let V(s) be 0 for all states s ε S+

do {
     play out an episode using policy π
     for each state s encountered in the episode {
          let N be the total number of first visits to s over all
               episodes
          R = return following first occurrence of s
          V(s) = V(s) + ( (1/N) * (R – V(s)) )
     }
}
while (true)
```

This algorithm (adapted from [Sutton & Barto, 1998]) updates each average incrementally, in order to avoid storing each single return for every state. In this case, each return is weighted equally. Depending on the task at hand, it might be desirable to maintain a weighted average, e.g., to give recent returns a relatively bigger importance than earlier returns, in the computation of the average.

5.2.3.2. Policy Improvement in MC

In Dynamic Programming, an optimal value function is all that is required to determine an optimal policy. The reason for this is that a perfect model of the environment's dynamics is present. In order for a policy to pick the best action $\mathbf{a}_t$ from state $\mathbf{s}_t$ in this scenario, DP uses the model to find the successor states of $\mathbf{s}_t$, and then picks the action $\mathbf{a}_t$ that leads to the highest-valued sum of immediate reward $\mathbf{r}_{t+1}$ and $\mathbf{V(s_{t+1})}$. Monte Carlo methods, on the other hand, often deals with situations where no model is present. In this case an optimal value function will not be sufficient to determine the optimal policy, since the look-ahead option is non-existent. To compensate for this, we will attempt to estimate $\mathbf{Q^\pi/Q^*}$ instead of $\mathbf{V^\pi/V^*}$. Remember that $\mathbf{Q^\pi(s,a)}$ denotes the expected return from taking action a in response to state s and thereafter following policy $\boldsymbol{\pi}$. As a result of this, if we have the

optimal action-value function **Q***, all we have to do to choose the optimal action from state s_t, is to consider all possible actions a_t, and take the action which maximizes the expression $Q^*(s_t, a_t)$. And no model is required. In other words, $\pi^*(s) = \text{argmax}_a Q^*(s,a)$.

5.2.3.3. Policy Iteration in MC

Formally, MC Policy Evaluation converges to the value function of the evaluated policy when the number of experienced episodes approaches infinity. In practice, an approach that has shown to be effective, is to play out only a single episode between each update of the policy. The next episode will then be played out with the updated policy, and so on. In this scenario, the algorithm will look something like this (adapted from [Sutton & Barto, 1998]):

```
let π be the policy to be evaluated
let Q(s,a) be 0 for all states s ε S+, a ε A(s)

do {
     play out an episode using policy π
     for each state-action pair s , a encountered in the episode {
          let N be the total number of first visits to s , a over
               all episodes
          R = return following first occurrence of s , a
          Q(s,a) = Q(s,a) + ( (1/N) * (R – Q(s,a)) )
     }
     for each state s encountered in the episode {
          π(s) = argmaxaQ(s,a)
     }
}
while (true)
```

An inherent problem with MC methods is the problem of maintaining exploration, which we can witness in the example above. Since π in this case is a deterministic policy, some actions will never be chosen, and some states will never be visited. As a result, the algorithm will not receive returns following the state-action pairs that are not chosen, and the corresponding action-values **Q(s,a)** will not be backed up. In this way, no exploration is performed, and the policy cannot improve. To work around this problem, we have a few approaches to choose from:

- Exploring starts

The principle of exploring starts implies that each episode is started at a random **Q(s,a)**, i.e., the first state of the episode is **s**, and action **a** is taken from this state, independent of the current policy. After this first action is taken, the current policy takes over. In this way, all action-values are backed up as the number of episodes approaches infinity. This works if the interaction with the environment is simulated, but is naturally difficult to control when interacting with a real-world environment.

- Stochastic policies

As mentioned earlier, a stochastic policy is a policy which assigns probabilities to selecting an action when in a state, i.e. $\pi(s,a) = Pr\{a = a_t \mid s = s_t\}$. This is in contrast to a deterministic policy, which assigns an action to every state, i.e. $\pi(s) = a$. Now, if a stochastic policy is assigned a non-zero probability for all states **s** and actions **a**, exploration is maintained, as long as a sufficient number of episodes are generated. This approach is utilized in two important classes of Monte Carlo methods; On-Policy methods and Off-Policy methods.

5.2.3.3.1. On-Policy MC methods

Characteristic for On-Policy MC Methods is that they follow the same policy that they update. In other words, the policy used for experience (generating episodes) and the policy which is improved after each episode, is one and the same. After each iteration of the algorithm, the new and improved policy is used to generate the next episode.

This approach requires that the policy is always assigned a non-zero probability for all $\pi(s,a)$, $s \in S$, $a \in A(s)$. A policy with this property is said to be *soft*. There are many ways to implement an On-Policy method, but a simple and instructive solution is to make the policy **ε**-greedy. An **ε**-greedy policy is a policy which takes the greedy action in most cases, but takes a random action with a small probability **ε**. That is;

```
for all possible actions a to take in state s {
     if a maximizes Q(s,a) {
          π(s,a) = 1 − ε + (ε/A(s).length)
     }
     else {
          π(s,a) = ε/A(s).length
     }
}
```

//A(s).length denotes the number of possible actions to take when in state **s**

An ε-greedy policy is one type of soft policy. By utilizing this approach, we are guaranteed to move the policy towards the optimal ε-greedy policy. Here is an overview of an On-Policy MC Method (adapted from [Sutton & Barto, 1998]):

```
let π be the policy to be evaluated
let Q(s,a) be 0 for all states s ε S+, a ε A(s)
let ε be some small probability, 0 < ε < 1

do {
     play out an episode using policy π
     for each state-action pair s , a encountered in the episode {
          let N be the total number of first visits to s , a over
               all episodes
          R = return following first occurrence of s , a
          Q(s,a) = Q(s,a) + ( (1/N) * (R – Q(s,a)) )
     }
     for each state s encountered in the episode {
          a* = argmax_a Q(s,a)
          for all a ε A(s) {
               if (a == a*) {
                    π(s,a) = 1 - ε + ε/A(s).length
               }
               else if (a != a*)
                    π(s,a) = ε/A(s).length
               }
          }
     }
}
while (true)
```

A more detailed examination of this algorithm will be provided in section 2.2.3.4.

5.2.3.3.2. Off-Policy MC methods

Unlike On-Policy MC methods, Off-Policy MC methods use different policies for experience and improvement. These methods maintain a *behavior* policy, which is used to generate episodes (experience), as well as an *estimation* policy, which is improved using the experience generated by the behavior policy. This enables the estimation policy to be greedy (deterministic) without sacrificing the sufficient exploration, as long as the behavior policy is soft (stochastic with non-zero probability for all actions).

To be able to use this approach, we must find a way to improve a policy using another policy, because the behavior and estimation policies will, given the nature of the method, not be the same. In fact, we may use a different behavior policy for each iteration of the algorithm, as long as this policy is always soft.

The following algorithm (adapted from [Sutton & Barto, 1998]) shows a Monte Carlo method which incorporates improvement of one policy by following another. Thus, exploration is maintained:

```
let Q(s,a) be 0 for all s ε S+, a ε A(s)
let N(s,a) be 0 for all s ε S+, a ε A(s)
let D(s,a) be 0 for all s ε S+, a ε A(s)
let π be a deterministic policy (the policy to improve)
let T represent the terminal time step

do {
     Play out an episode using a soft policy π'
     let N be the last time step in the episode where π'(s) != π(s)
     for each pair s,a in the episode after time step N {
          t = the time step of the first occurrence of s,a in the
               episode
          w = ∏^(T-1)_(k=t+1) (1/π'(s_k,a_k))
          N(s,a) = N(s,a) + wR_t
          D(s,a) = D(s,a) + w
          Q(s,a) = (N(s,a)/D(s,a))
     }
     for all s ε S {
          π(s) = argmax_a Q(s,a)
     }
}
while(true)
```

This algorithm can be somewhat tricky to understand, so let us try to examine it in more detail. Consider an episode played out by the behavior policy **π'**. We wish to improve the estimation policy **π** from the state transitions in the newly experienced episode. What we need to do first, is to find a continuous sequence of state-transitions lasting till the end of the episode (generated by **π')** , that would have occurred had we followed the estimation policy **π** as well. This sequence of state-transitions is what we are interested in. It is important that the common subsequence lasts until the end of the episode, because in MC the returns are not defined until the end.

Next, we go through this subsequence, and identify the first occurrences (first-visit) of each distinct pair **s**,**a** in the subsequence. For each first-visit of **s**,**a** we update the corresponding action value **Q (s,a)** by weighting each return by its relative probability of occurring under **π** and **π'**. Let us see what happens step by step:

- $\mathbf{w = \prod^{T-1}_{k=t+1} (1/\pi'(s_k,a_k))}$

this expression equals

the probability of the entire sequence following **s**,**a** happening under policy **π**

(Since **π** is deterministic, the probability is 1)
divided by
the probability of the entire sequence following **s**,**a** happening under policy **π'**

in other words, the relative probability of the sequence happening under **π** and **π'**.

- $\mathbf{N(s,a) = N(s,a) + wR_t}$

this expression updates the value of **N(s,a)** by adding it with

the return acquired following **s**,**a** in the latest episode ($\mathbf{R_t}$), weighted by the relative probability of the sequence happening under **π** and **π'** (**w**). **N(s,a**) will be the numerator in the division to the newest estimate of **Q(s,a)**.

- **D(s,a) = D(s,a) + w**

this expression updates the value of **D(s,a)** by adding it with

the relative probability of the sequence happening under **π** and **π'** (**w**). **D(s,a)** will be the denominator in the division to the newest estimate of **Q(s,a)**.

- **Q(s,a) = (N(s,a)/D(s,a))**

the newly updated value of **N(s,a)** is divided by the newly updated value of **D(s,a)** to yield the new estimate of **Q(s,a)**. Simply stated, the expression **N(s,a)**/**D(s,a)** represents an average of all returns

received following **s**,**a** in all episodes experienced up to this point, *weighted* by the probability ratio of the sequence following **s**,**a** happening under $\boldsymbol{\pi}$ and $\boldsymbol{\pi}'$. This ratio is important to include, in order to make the experience gathered from $\boldsymbol{\pi}'$ relevant for $\boldsymbol{\pi}$.

5.2.3.4. MC in game playing

As in the chapter concerning Dynamic Programming, let us also in this chapter consider applying Monte Carlo Methods to a game playing task. For example, Tic-Tac-Toe could be our object of attention in this case as well, although MC methods can be applied to games with larger state sets than DP. The reason for this being that MC methods learn without having to perform systematic state sweeps.

As we recall, MC methods learn from interaction with an environment. If an environment can not be interacted with directly, a model of the environment can be used instead. The model is required to respond to actions in accordance with the behavior of the actual environment. In contrast to the model required by DP methods however, this model is not required to make probability distributions explicitly available to the learning agent. Monte Carlo methods learn by sample transitions, and what is required is what is called a sample model (if the environment can not be interacted with directly, that is). In our concrete case, the learning agent will play against the opponent directly, or with a sample model capturing the opponent's strategy.

As described earlier, there are several different MC methods to choose between. In this case, let us choose a first-visit On-Policy MC method improving an $\boldsymbol{\varepsilon}$-greedy policy. The policy is initialized arbitrarily. We operate with state-action values (**Q(s,a)**) instead of state values (**V(s)**), and initialize **Q(s,a)** for all terminal states to 0. **Q(s,a)** for non-terminal states may be initialized with random values. As the case was with DP, we assign a reward of 1 to terminal states which represent a win for our agent (game boards where our agent has placed three items in a row), a reward of -1 to terminal states which represent a loss for our agent, and a reward of 0 to terminal states representing a draw. All non-terminal states are assigned a reward of 0.

Now, the learning process begins by playing a game session (episode) to the end, resulting in a terminal state. This terminal state returns a reward of 1, -1 or 0, as described above. Next, we iterate through all game boards that faced the learning agent during the episode, paired with the moves the agent made in response to these boards, according to the policy (the state-action pairs). In fact, in this method we only iterate through the first occurrences of the state-action pairs. In other words, if a particular combination of a game board and move occurs more than one time in the episode, we concern ourselves only with the first occurrence of this combination (first-visit).

For each of these pairs, we determine the return (the accumulated reward) following the pair, in the episode at hand. In our case, only the terminal states have the potential to give non-zero rewards, and so the reward of the last state of the episode is the basis of the return. If we are dealing with a non-discounted task (as we probably are in this case), each state-action pair in the episode along the way to the terminal state is presented with a return with the exact same value as the reward of the terminal state. If the task is discounted however, each state-action pair in the episode along the way to the terminal state is presented with a return constituting the reward of the terminal state,

discounted according to the state-action pair's distance to the terminal state, in time steps. In other words, a state-action pair which occurred **n** time steps before the terminal state, is presented with a return with a value of the terminal reward multiplied by γ^n , where γ is the discount rate

0 <= γ <= **1**, as mentioned earlier.

Each state-action pair, (game board-move pair) are presented with this return, which is called a sample return. For each state-action pair, its corresponding return is now used to update the average of all returns this pair has ever been faced with (the returns that have followed this pair). The resulting average becomes the new state-action value (**Q(s,a)**) of the state-action pair in question. In this way, we can see that the **Q(s,a)** values are updated only along the board configurations that occurred during the game session, and all updates are performed after the session is finished.

When a game session is done, and the state-action values are updated, it is time to update and improve the policy, according to the newly updated state-action values. This is done by iterating through all the unique game boards (states) faced in the game session (episode). For each of these game boards, we determine which of the legal moves (actions **a**) from this board (state **s**) gives the highest state-action value. This is the greedy action **a***. Now we update the policy to choose the greedy action **a*** when in state **s** with a large probability. With a small, non-zero probability, we assign the policy to choose a random action when in this state. This ensures sufficient exploration, as mentioned earlier. Now the agent is ready to play again, with the updated and improved policy. When the next game session is over, the entire procedure is repeated, and so on.

5.2.4. Temporal Difference learning

Temporal Difference (TD) Learning is a core concept in Reinforcement Learning today. TD Learning incorporates ideas from both Dynamic Programming and Monte Carlo Methods, and combines the advantages of these two classes of methods. Like DP, TD Learning algorithms employ bootstrapping, i.e., the estimate of a state value is based on the estimate of another state value. Similar to MC methods, TD Learning learns from experience, by interacting with an environment, and so a model of the environment is not required. As a consequence, it also uses sample backups to update the state values, much like MC methods do. Unlike MC methods, however, TD Learning methods do not need to wait until the end of an episode to make backups. TD Learning methods update a state value after only one time step (from the successor state value). Because of this, TD Learning is not restricted to just episodic tasks, but may equally well be applied to continual tasks without identifiable terminal states.

5.2.4.1. Policy Evaluation in TD

As mentioned, when following a policy to be evaluated, TD learning backs up a state value based on the estimated value of the state encountered one time step later. To clarify this, consider the update rule performed by the simplest TD method, called TD(0):

$$\mathbf{V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]}$$

where $\boldsymbol{\alpha}$ is a constant step-size parameter. This shows that when a state-transition leads to a successor state, the value of the original state is adjusted by moving it towards the sum of the immediate reward and the value of the successor state. The magnitude of the update is determined by the parameter $\boldsymbol{\alpha}$.

Since TD methods are often faced with tasks where no model is present, we will concentrate on estimating action-value functions rather than state-value function, like we did with MC methods. Since action values are easily defined in terms of state values, this conversion is relatively straight-forward. The following backup statement, which is derived from the TD(0) update statement above, is employed by a TD method called Sarsa:

$$\mathbf{Q(s_t,a_t) = Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_{t,})]}$$

5.2.4.2. Policy Improvement in TD

The major difference between the various RL methods is found in the way they estimate state- or action-value functions, i.e., in the Policy Evaluation step. When it comes to Policy Improvement, all

methods will improve the policy by making it exploit the value function estimated in the Policy Evaluation step. However, this exploitation must be balanced against the exploration necessary for the policy to improve further. To achieve this, the policy must occasionally take non-greedy actions. This problem of maintaining exploration, as was described in association with Monte Carlo methods, is highly relevant when it comes to Temporal Difference Learning, as well. And like MC methods, TD Learning approaches the problem with two distinct methods; On-Policy and Off-Policy methods.

5.2.4.3. Policy Iteration in TD

In this section we consider full Policy Iteration using Temporal Difference learning. We present two different Policy Iteration method classes which maintain exploration. These are, as mentioned above, On-Policy and Off-Policy methods.

5.2.4.3.1. On-Policy TD learning

As we recall, On-Policy methods update and improve a policy based on experience generated by the same policy. To ensure that sufficient exploration is maintained, the policy must, with non-zero probability, select each action from any given state. In other words, it has to be soft. A usual implementation choice is to let the policy be **ε**-greedy, although other alternatives are possible. There exist a number of sophisticated methods to balance exploration and exploitation. One might for instance utilize an **ε**-greedy policy, which decreases the **ε** parameter over time. The idea behind this is to perform a high degree of exploration early in the learning process, and then gradually move towards a deterministic policy to exploit the result of this early exploration. The following On-Policy TD method uses an **ε**-greedy policy with a stable **ε** value. The update statement used is Sarsa, as shown earlier. In the following presentation (adapted from [Sutton & Barto, 1998]), the algorithm is shown being applied to an episodic task, to simplify the example. Bear in mind, however, that this is not a prerequisite of TD methods. The task could just as well have been continual.

```
Let Q(s,a) be 0 for all s ε S+
for each episode {
     s = starting state
     a =  ε-greedy action in response to state s, with respect to Q
     do {
          take action a in response to state s
          r = the resulting reward
          s' = the resulting state
          a' = ε-greedy action in response to state s', with
               respect to Q
          Q(s,a) = Q(s,a) + α[r + γQ(s',a') − Q(s,a)] //SARSA
          s = s'
          a = a'
     } while(s != terminal state)
}
```

Let us follow one iteration of this algorithm in detail, starting from the first state of an episode. To make this more concrete, consider a generic board game. The starting state represents the initial game board facing our learning agent. Now it is up to the agent to pick a move (action) in response to the board configuration (state). The agent has a number of actions to choose from, and uses the current **Q** estimate to determine which action yields the highest **Q(s,a)** value (**s** = the current state). This action is the greedy action. Now, since the algorithm is **ε**-greedy, the probability of the agent taking the greedy action is **1 - ε + ε/A(s).length** (where **A(s).length** denotes the number of actions available in state **s**). However, with probability **ε/A(s).length** the agent instead takes an action (from **A(s)**) completely at random.

When the agent has taken its action, no matter which, it receives an immediate numeric reward **r**. In addition to this, it is faced with a resulting state **s'**. Again, it is up to the agent to pick an action in response to this state. Then agent consults the estimate of **Q(s',a')**, for all **a' ε A(s')**. It finds the greedy action **a'**, and selects it with probability **1 - ε + ε/A(s').length.** With probability **ε/A(s'). length** the agent instead selects an action (from **A(s')**) at random. The resulting action **a'** is not yet taken however (the state transition is not made). First the Sarsa backup statement is used to update the value of **Q(s,a)** partly based on the value of **Q(s',a')**:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$$

Then, in the next iteration of the algorithm, action **a'** is taken in response to state **s'**, and the state transition is made. This continues along the path of state-action transitions until a terminal state is encountered, and the episode is over. In other words, the policy is updated as the game is being played.

5.2.4.3.2. Off-Policy TD learning

Q-learning is a popular and important Off-Policy TD Learning method, which at the same time is relatively easy to understand and implement. In this section we consider the simplest version of this method, called 1-step Q-learning. The following algorithm (adapted from [Sutton & Barto, 1998]) uses 1-step Q-learning:

```
Let Q(s,a) be 0 for all s ε S+
for each episode {
     s = starting state
     do {
          a =  ε-greedy action in response to state s, with respect
               to Q
          take action a in response to state s
          r = the resulting reward
          s' = the resulting state
          Q(s,a) = Q(s,a) + α[r + γmax_a'Q(s',a') − Q(s,a)]
          s = s'
     } while(s != terminal state)
}
```

As we can see, using this method we update **Q(s,a)** based on the value of the optimal combination of successor state and successor action, independent of the current policy. However, the actions actually taken by the agent, are still chosen in accordance to the policy, in an **ε**-greedy manner, ensuring that the necessary exploration takes place. Summarized, a learning agent using the Q-learning method takes actions in an **ε**-greedy way, but backs up **Q(s,a)** based on the greedy estimates of **Q(s',a')**.

As we did with the On-Policy method in the previous section, let us follow one iteration of this algorithm as well. We will again use the game playing scenario. The learning agent is faced with an initial game board (state), and has to make a choice about which move (action) to make in response to this board. The agent considers all actions **a** possible in state **s**, and finds the greedy one with respect to the current estimate of **Q**. The agent takes the greedy action with probability **1 - ε + ε/A (s).length.** With probability **ε/A(s).length** the agent instead takes an action (from **A(s)**) at random. When the action is taken, the agent receives a reward **r**, and is faced with the resulting state, **s'**. Now, the agent finds the action **a'** in response to state **s'**, which yield the highest value for **Q(s',a')**, given the current estimate of **Q**. That is, the action **a'** taken in state **s** that yields the highest expected cumulative reward from that point on. This maximum value for **Q(s',a')** is then used to backup **Q (s,a)** in the following way:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

After this, state **s'** becomes the current state, and the next iteration starts. The **ε**-greedy policy is applied to the new current state, and so on, until a terminal state is encountered, and the episode is done. The initial statement

Let Q(s,a) be 0 for all s ε S^+

is not really necessary, all that is required is that values for terminal states are 0 (This is also true for DP and MC methods). This makes sense, because terminal states have no successor states, and therefore no accumulated future reward can be acquired from these states. Values for non-terminal states, however, may be initialized to other values, e.g., random values. Sometimes non-terminal states are given high initial values (optimistic values), in order to encourage exploration. However, initializing values for all states to 0, makes it easier to incrementally track the learning algorithm's progress through the state set.

5.2.4.4. TD in game playing

Let us now briefly consider how Temporal Difference Learning methods can be applied to the task of learning how to play a game like Tic-Tac-Toe. The method we will use this time is the simple Off-Policy 1-step Q-learning algorithm, as described earlier. Just like MC methods, this algorithm learns by interacting with its environment, or a sample model of the environment.

First we initialize the state-action values of all legal pairs of game boards and moves (states and actions) arbitrarily. The state-action values involving terminal states must be initialized to 0. When this is done, a game session (episode) can start. When the agent is faced with a game board (state **s**), it determines the move which, paired with the board, has the greatest state-action value. This move is chosen with a large probability, because this is the greedy move. With a small probability however, a move is chosen at random. This is done to ensure sufficient exploration. Now, the agent makes the chosen move (action **a**), whether it is greedy or not. The environment (i.e., the opponent), or the model of the environment, responds with a counter move, and the resulting game board (state **s'**) is therefore the board following the opponent's move.

The agent observes the reward **r** associated with this resulting game board (state **s'**). Now, before the next move is taken by our agent, the state-action value of the previous game board paired with the taken move is updated in the following way:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

This substatement

$$\gamma \max_{a'} Q(s',a')$$

represents the (discounted) maximum state-action value involving the resulting game board configuration (state **s'**). This value is added to the immediate reward associated with state **s'**. The state-action value of the previous game board and the taken action is then subtracted from this sum, like this:

$$r + \gamma \max_{a'} Q(s',a') - Q(s,a)$$

This statement is called the TD error, and is what the value of the original **Q(s,a)** is moved towards. The magnitude of this movement is determined by the size of the step-size parameter **α**, like this:

$$Q(s,a) = Q(s,a) + \alpha * \text{TD-ERROR}$$

When this update is done, the agent makes its next move, according to the same **ε**-greedy policy as described above. The process is is repeated along each state of the game session. When a game board configuration is encountered, the agent selects the move in an **ε**-greedy manner. The opponent makes a counter move, and then it is the agent's turn again. Before making its move, the agent updates the state-action value of the game board and move it encountered and selected, respectively, the last time is was its turn to move. This update is based on the immediate reward and the maximum state-action value of the new game board. In contrast to MC methods, this method learns during a game session, and not just after each episode.

5.3. Using eligibility traces

The TD methods presented so far have all made backups towards the value of a state $\mathbf{s_t}$ based on the immediate reward associated with the successor state ($\mathbf{r_{t+1}}$) *and* the current estimated state value of the successor state ($\mathbf{V(s_{t+1})}$). In other words:

$$\mathbf{V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]}$$

where the substatement

$$\mathbf{r_{t+1} + \gamma V(s_{t+1})}$$

is called the *return*. In this case the return is a 1-step return, named so because the value of a state can be updated after only 1 time step. The TD methods presented so far have all been 1-step methods, i.e. TD methods which make backups of state values based on a 1-step return. In the other end of the spectrum, we find MC methods, which make backups based on the entire sequence of real rewards (i.e., the complete return). In the case of MC methods, the complete return can be expressed like this:

$$\mathbf{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + y^{T-t-1} r_T}$$

Where T is the final time step of the episode. In other words, MC methods do not employ bootstrapping, in contrast to TD methods.

5.3.1. N-step TD methods

Between the two extremes of 1-step TD methods and MC methods is a class of methods called *n-step TD methods* (where n > 1). These methods are also TD methods, but what makes them different from 1-step TD methods, is that they make backups based on an n-step return instead of a 1-step return. But what is an n-step return? In order to answer this, remember that a 1-step return at time step **t** consists of the reward received at time step **t+1** plus the (discounted) current estimated value of the state reached at time step **t+1**, like this (repeated from above):

$$\mathbf{R_t^{(1)} = r_{t+1} + \gamma V(s_{t+1})}$$

An n-step return simply consists of the rewards received at the next **n** time steps plus the current estimated value of the state reached at time step **t+n**, like this:

$$\mathbf{R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n})}$$

For example, a 2-step return is a type of n-step return. The 2-step return can be expressed like this:

$$R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+1})$$

Consequently, the general backup statement used by an n-step TD method would look something like this:

$$V(s_t) = V(s_t) + \alpha[R_t^{(n)} - V(s_t)]$$

n-step TD methods are valid learning methods, and possess the same convergence properties as 1-step TD methods and MC methods. They are, however, somewhat inconvenient to implement in practice, because one has to wait **n** time steps after visiting a state, before that state's estimate can be updated. N-step TD methods have the potential to converge to the correct value function faster than simple 1-step TD methods.

5.3.2. Complex backups

Backups may also be performed based on averages of different n-step returns. For example, we can make backups based on the following return:

$$\tfrac{1}{3}R_t^{(2)} + \tfrac{1}{3}R_t^{(3)} + \tfrac{1}{3}R_t^{(4)}$$

The resulting backup is called a *complex backup*, because it is a combination of two or more simple backups. Any simple backups can be combined in this way, and they may be weighted in any way desirable, provided that the individual weights are positive and sum up to 1. *TD(λ)* is an algorithm that uses complex backups like these in a special way, in order to estimate the value function of a policy. This algorithm updates state values towards a backup based on a special return called the λ-return, R_t^{λ}, defined in the following way:

$$R_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \quad , \qquad 0 <= \lambda <= 1$$

The complete update statement will then look like this:

$$V(s_t) = V(s_t) + \alpha[R_t^{\lambda} - V(s_t)]$$

The special λ-return R_t^{λ} averages *all* n-step returns, and weights each of these returns proportional to λ^{n-1}. The term $(1 - \lambda)$ ensures that the weights sum to 1. In this way, the 1-step return is given the largest weight, equal to

$$(1 - \lambda) * \lambda^0$$
$$= (1 - \lambda) * 1$$
$$= (1 - \lambda)$$

The 2-step return is given the weight $(1 - \lambda) * \lambda$, the 3-step return is given the weight $(1 - \lambda) * \lambda^2$, and so on. If $\lambda = 0$, this means that the 1-step return is given a weight of 1, and all the other returns are ignored. In other words; the λ-return is the same as the 1-step return when $\lambda = 0$. On the other hand, if $\lambda = 1$, the λ-return is the same as the complete return, i.e., the return used by MC-methods. For values of λ larger than 0 but smaller than 1, we get a family of TD methods in the space between 1-step TD and MC methods, which average the n-step returns in different ways. Common for all of these methods is that the weight for each simple composite n-step return decays with increasing **n**. Making backups based on the λ-return often results in more efficient learning than using simple backups.

As mentioned, an n-step return can be inconvenient to compute in practice, and this makes the λ-return especially troublesome to compute in a straightforward way, since it is based on *all* n-step returns. In the next section we will consider a practical way to compute this return, however, and to use this return to estimate the value function of the policy being followed. This is done by the use of eligibility traces.

5.3.3. Eligibility traces

An eligibility trace is a real-valued memory value associated with a state (or state-action pair). This value indicates how eligible the associated state is to be updated in terms of the TD error at a given time. The eligibility trace for state **s** at time **t** is denoted $\mathbf{e_t(s)}$. The eligibility trace for all states are initially 0. When a state is visited during an episode, its value is incremented by 1. In addition, for each time step of the episode, all eligibility traces are decayed by $\gamma\lambda$, where γ is the discount parameter, and λ is called the trace-decay parameter. It is the same parameter introduced in association with complex backups, in the previous section. In other words, for each time step of an episode, eligibility traces are modified in the following way:

```
st = the state visited at the current time step
for all e(s) ,  s ε S {
     if s == st {
          e(s) =  γλ e(s) + 1
     }
     else if s != st {
          e(s) = γλ e(s)
     }
}
```

This kind of trace is common, and is called an accumulating trace, because if a state is visited often, its eligibility trace can accumulate faster than it decays. Replacing trace is another kind of trace, which is reset to 1 each time the associated state is visited. In this way an individual trace can never exceed a value of 1.

In order to visualize how eligibility traces are employed in the task of evaluating the value function of a policy, consider the following policy evaluation algorithm, TD(**λ**):

```
Let V(s) be 0 for all s ε S+
Let e(s) be 0 for all s ε S+
Let π be the policy to be evaluated

for each episode {
     s = first state of episode
     for each step of the episode {
          Let a be the action chosen by π in response to s
          Take action a
          Let r be the resulting reward
          Let s' be the resulting next state
          δ = r + γV(s') − V(s)     //The TD-error
          e(s) = e(s) + 1
          for all s ε S+ {
               V(s) = V(s) + αδe(s)
               e(s) =  γλ e(s)
          }
          s = s'
     }
}
```

In this algorithm (adapted from [Sutton & Barto, 1998]), we see that **δ** represents the 1-step TD error at each time step. The eligibility trace for the state visited at the current time step is increased by 1. Then, the value estimate of all states are updated by multiplying the TD error **δ** with their respective eligibility traces, weighting the resulting product by the step-size parameter **α**. The result is added to the previous value estimate of the state, and thus the update is finished. From this, we can see that more recently visited states (with high-valued eligibility traces) receive a bigger update, which makes sense. Finally, all eligibility traces are decayed according to **γλ**, and we proceed to the next state in the episode. Using eligibility traces result in faster learning than 1-step methods, although it requires more computation.

5.4. Generalizing the value function

So far in this report we have dealt exclusively with cases in which the estimated state- or action value function was represented by a table, with one entry corresponding to each separate state or state-action pair. When facing tasks with large, or possibly infinite, state sets, a table no longer constitutes an appropriate way to represent the value function. One reason for this is the amount of memory needed for all the discrete table entries, as well as the time required to fill and update these entries. Another reason why a table is unsuitable, is that when facing a learning task with a large number of states, each individual state will probably be rarely encountered, and some states might never be visited at all. Clearly, some sort of generalization of experience is needed, so that the encounter of one state might help the learning agent learn about similar states as well.

A common way to escape the disadvantages of table-based value functions, is to represent the tabular value function with some kind of parameterized function. Instead of updating separate table entries, policy evaluation (estimating the value function of a policy) will then consist of adjusting the parameters of this function, in order to make it an approximation of the true value function of the policy at hand. This is called function approximation, and is an instance of the machine learning technique supervised learning.

The parameterized function representing the value function can take many forms. For example, it might be a complex multi-layer artificial neural network, a decision tree, or a simple linear function. By using this kind of value function, an update made based on experience from encountering a state, will affect the estimated value of other states as well. In other words, the value function generalizes from experience. In order to be useful however, the function approximation method used should be able to learn from incrementally-acquired experience, because experience is gathered as the agent explores the environment. It should also be able to handle non-stationary target functions.

In general, function approximation is the act of approximating a target function by observing examples of input-output pairs of this target function. In our case, the target function is the state value function $\mathbf{V}^{\pi}$ of the policy $\boldsymbol{\pi}$. In order to approximate a target function, the approximation method needs to be presented with input-output examples. How are we going to find input-output examples of $\mathbf{V}^{\pi}$?

As described earlier, RL methods make updates to a value function based on backups. Depending on the particular RL method, the backups are computed in different ways. Dynamic Programming methods update the value function based on the following backup (the expected value of the sum of the reward and the discounted estimated value of the successor state, following policy $\boldsymbol{\pi}$):

$$\mathbf{E}_{\pi}\{\mathbf{r}_{t+1} + \gamma \mathbf{V}(\mathbf{s}_{t+1})\}$$

Monte Carlo methods update the value function based on the entire return following a state in an episode:

R

1-step TD methods update the value function based on the backup computed by adding the sample reward and the discounted estimated value of the successor state:

$$\mathbf{r_{t+1} + \gamma V(s_{t+1})}$$

No matter which RL method is used, and how the corresponding backup is computed, a backup signals that the value function should be corrected towards the value of the backup, in some way. This means that a state **s** and its corresponding backup (no matter how it is computed) can serve as an input-output example of the target value function to be approximated, and as such can be presented to the function approximation method. In each example, the state is the input part of the example, and the backup is the output part. For each example presented to the approximation method, the internal parameters of the approximate function are adjusted in a way that will reduce the error, given the current example. How this is performed concretely, depends on the type of learning method used for function approximation.

Learning methods based on the principle of gradient-descent is often used in function approximation in general, and in combination with Reinforcement Learning in particular. There are especially two types of gradient-descent based function approximators that have been extensively used in Reinforcement Learning. The first one is the linear form, where the approximated function is represented as a linear function of the parameters to be adjusted. The second one is in the form of an artificial neural network. In this latter case, the backpropagation algorithm is used to compute the gradients. The parameters to be adjusted in this case, are the network's internal connection weights.

Using function approximation methods to represent the value function in this way, overcomes a number of constraints posed by table based value functions, and has in may cases proven to be successful empirically. The theoretical aspects of this approach, however, are often difficult to analyze, and based on the type of backups used, the distribution of training examples and the type of function approximation applied, the convergence guarantees are often uncertain.

5.5. A preliminary experiment : Tic-Tac-Toe

This experiment was performed in preparation for the main project, concerning Connect4. In this experiment, we attempted to train an agent to play the simpler game Tic-Tac-Toe (TTT) as strongly as possible, utilizing Reinforcement Learning (RL) methods. Tic-Tac-Toe is a two-player game played on a 3x3 grid. The beginning player places Xs in vacant cells on the grid, while the opposing player places Os. The players alternate at making moves. The first player to connect three Xs or Os, is the winner. If noone wins, the game is declared a draw. The training experience used was

gathered exclusively from self-play; i.e., the agent learned to play TTT by playing against itself, and having no prior knowledge of the game, except the rules. In this preliminary experiment, the learned function was not subsequently combined with the minimax algorithm when playing, but merely used, by itself, to choose between the immediate resulting states from a board state.

5.5.1. Using a tabular value function

The implementation consists of two main components; the environment and the learning agent. The environment represents the game board and the rules and rewards of the game, and the agent represents a player tries to improve its game playing skills by interacting with the environment. In this implementation, the state-value function, as explained earlier, is represented by a table, more specifically a hash table, which allows for quick access to stored data. In this hash table, the keys are string representations of various TTT board configurations, and the values are floating point numbers indicating the values of the corresponding boards. In this way, an entry in the hash table might look like this:

```
Key: ”100120000”    Value: 35.04
```

A 1 in the string indicates an X, a 2 indicates an O, and a zero indicates an empty square on the board. The string represents the game board viewed from left to right, top to bottom. As a result, the string above corresponds to the game board shown in Figure 11.

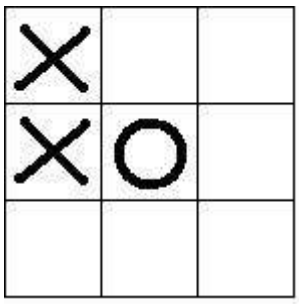

Figure 11

The value associated with a board indicates the board's “desirability”, or more precisely: expected, possibly discounted, cumulative reward following that particular board /state. The environment associates different rewards with different board configurations. A board representing a win is associated with a reward of 100, a loss -100. All other boards are associated with rewards of 0.

In this particular implementation, learning occurs by making two instances of the same agent play against each other, using and updating the same value function. In this way, the first instance always plays X and the other always plays O. Of course, this could have been done by one and the same instance, but we have chosen this approach in order to keep the code more tidy. The X-player updates entries used by X-players in order to decide which move to make, and the O-player updates

entries used by O-players.

Initially, we meant for the agent to work with a table of action values, in which numerical values were associated with a board and a subsequent move, like this:

```
Key: "1000200003"   Value: 35.04
```

In this representation, the first nine elements of the string correspond to the game board, as before, while the tenth element represent the subsequent move. In other words; when the X-player is facing the board shown in Figure 12,

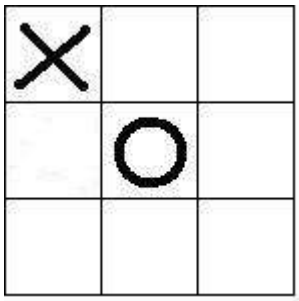

Figure 12

the action of placing an X in the square with index 3 (the leftmost square of the middle row), is associated with a value of 35.04. However, this type of representation leads to a considerable degree of redundancy, as multiple board-move combinations can lead to the same resulting board, as shown in Figure 13.

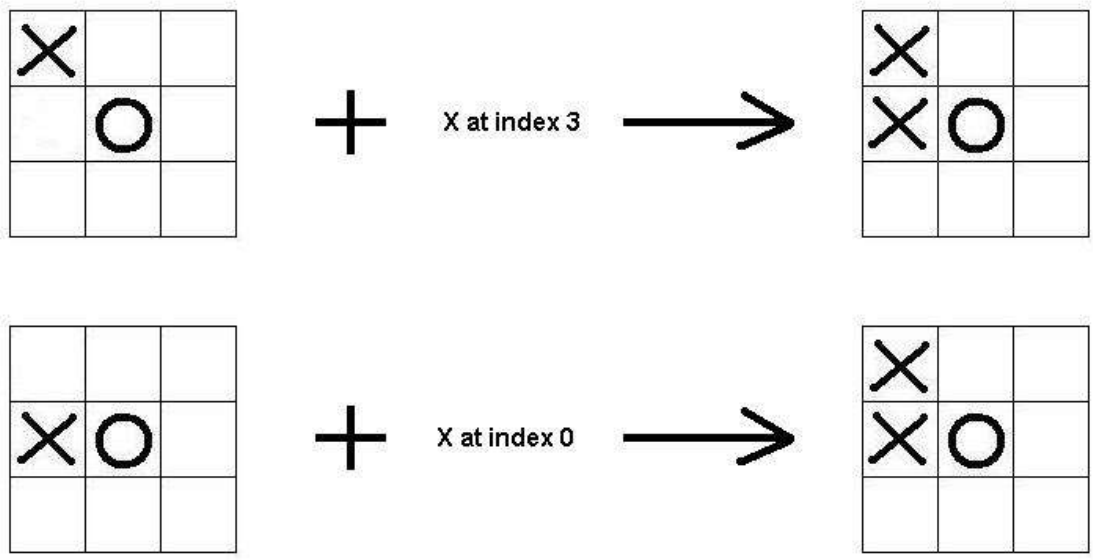

Figure 13. Multiple board-move combinations can lead to the same resulting board.

Now, since the resulting boards of `"1000200003" and "0001200000"` are the same, they should be associated with the same value. Instead of storing these separately, an approach called afterstate values [Sutton & Barto, 1998] was used, in which the *result* of a board and a move is stored as a key in the table. This means that `"1000200003" and "0001200000"` are merged into the new key `"100120000"`. In the case of this experiment, the number of entries in the table were reduced to roughly 1/3 of its former size, without any information being lost. The total number of entries after training is now 5477.

5.5.2. The learning process

Learning takes place by letting the agent play a number of games against itself, where each game corresponds to an episode in RL terminology. The hash table representing the value function (hereby called **V**), is initially empty. The environment is responsible for keeping the game board updated, and to alternate between the players and requesting moves from them. The environment maintains the game board as an integer array of ten elements, where the first nine elements represent the actual board. The tenth element contains the immediate reward associated with the board. When an episode starts, the game board is cleared, and a move is requested from the X-player.

The X-player receives a reference to the game board. It proceeds to examine it to find the vacant squares, i.e., the possible moves it has to choose from. For each vacant square it finds on the board, the agent makes a string representation of the board resulting from filling that square. An example: The X-playing agent receives a board represented by the following array:

```
[1,0,0,0,2,0,0,0,0,0]
```

The corresponding board is shown in Figure 14.

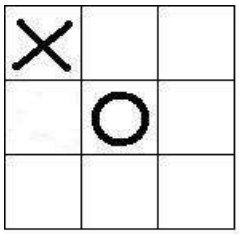

Figure 14

The agent iterates through the board to find the vacant squares. The first objective is to find the best move, i.e., the move with the highest associated value. As we can see, the first square is occupied. The second square is vacant, however. The agent makes a string representation of this possibility; "110020000". Next, the agent checks if this string (key) is already present in **V**. If it isn't, the agent enters it into **V**, and associates it with a value of 0. As we can see, *afterstate* values are stored, i.e., values associated with boards resulting from the agent's potential moves.

On the other hand, if the string is already present in **V**, its associated value is fetched from the table. In this way, all the "candidate moves" are examined. The values are compared, and the best (greedy) move is stored, but not yet made. If multiple moves have the same highest score, a best move is selected randomly from these moves. In this case, let us assume that the best move is to put the X in the third square (upper rightmost corner). The string representation of this is "101020000".

Before a move is made, the previous afterstate chosen by the X-player must be updated towards the value of the optimal move the agent just found. In other words, 1-step Q-learning is used to implement learning in this experiment. In this case, the previous afterstate chosen by the X-player must have been

```
[1,0,0,0,0,0,0,0,0,]
```

or equivalently, like Figure 15.

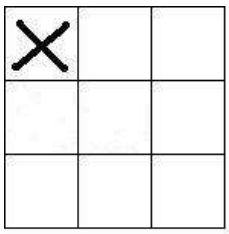

Figure 15

The value associated with the key "100000000" is then updated in the following way:

$$\mathbf{V(\text{“}100000000\text{”}) = V(\text{“}100000000\text{”}) + \alpha[r + \gamma V(\text{“}101020000\text{”}) - V(\text{“}100000000\text{”})]}$$

The **r** in this statement is the immediate reward following the previous afterstate. In other words, **r** is the reward received by the X-player after placing the X in the first square, i.e., the reward associated with the board resulting from O's first move. When the update has been made, the best move, which was found above, is made (sent to the environment) with a large probability. With a small probability however, the move is chosen at random. This is done to ensure sufficient exploration. The algorithm is thereby **ε**-greedy.

The agent then returns its chosen move to the environment, which updates the game board. The environment then checks if the game is over. If it isn't, a reward is assigned to the board, and the O-player is then requested to make a move. In this way, updates are made following each move. One copy of the agent updates entries associated with states following X-moves, another copy updates entries associated with states following O-moves.

5.5.3. Results

In order to test the performance of the learning agent, it was made to play 100.000 TTT games against the minimax algorithm, which was described in section 2. In the case of TTT, the state space is sufficiently small for the minimax algorithm to completely traverse the game tree in a small amount of time. Hence, no cutoff is used. In the case of TTT, the minimax algorithm is unbeatable, so the best possible outcome of a game for the learning agent, is a draw. By using a learning rate, **α**, of 0.1, an agent could be produced, which was able to draw all 100.000 games against the minimax algorithm. This was accomplished by training the agent over 20.000 episodes. Another prerequisite of this result however, was that the **ε**-parameter was reduced linearly, from 1 to 0, during the training process. This means that during the very first episode of training, **ε** was 1, and all moves made by the agent were randomly selected. Likewise, during the last episode, **ε** was 0, and all moves made by the agent were greedy. This enables a large degree of exploration during early episodes, while gradually shifting towards exploitation of learned values during later episodes. The discount rate, **γ**, has been kept at 1.0 at all times during training.

Out of curiosity, we constructed a TTT player which made all moves completely at random, in order to compare how the learning agent and the minimax algorithm performed against this random-agent. This test showed that while the minimax algorithm never lost against the random-agent, the learning agent lost against the random-agent in about 5% of the games when playing O. We were able to improve this by adjusting the training parameters. By decreasing the learning rate, **α**, to 0.01 and training for 1 million episodes, the learning agent no longer lost to the random-agent (and it was still able to avoid loss against the minimax algorithm). The performance results were now as shown in Table 2.

X vs. O	*X wins*	*O wins*	*Draws*
L_Agent vs. Minimax	0,00%	0,00%	100,00%
Minimax vs. L_Agent	0,00%	0,00%	100,00%
R_Agent vs. Minimax	0,00%	80,50%	19,50%
Minimax vs. R_Agent	99,50%	0,00%	0,50%
L_Agent vs. R_Agent	95,00%	0,00%	5,00%
R_Agent vs. L_Agent	0,00%	75,00%	25,00%

Table 2. Results from training an RL agent to play Tic-Tac-Toe, using a tabular value function.

This shows that although the learning agent no longer loses to the random-agent, the statistics of minimax are slightly better than the learning agent's, as the minimax algorithm has a greater percentage of wins compared to draws. The training parameters could probably be adjusted even more, in order to try to achieve an even better result. We are, however, not sure how relevant it is for the learning agent to perform well against a random-agent. Intuitively, it seems to make sense, since if one plays enough games against a random-playing opponent, one is bound to face every defense/counterattack possible from the opponent, at least when it comes to games with small state spaces.

5.5.4. Using a neural network value function

While using a tabular value function seemed to work well when it came to TTT, it was interesting to try to represent the value function with some sort of parameterized function. The choice fell upon using a neural network [Mitchell, 1997] for this task, since this function approximation method has often been used in connection with Reinforcement Learning. We proceeded to implement a neural network application, which in turn permitted us to construct and train neural networks with a variety of architectures, utilizing various transfer functions. However, the application can only handle static, acyclic feedforward networks, and the only learning method used is stochastic gradient descent, by means of the backpropagation algorithm. Aside from this, the user can freely choose the number of inputs to the network, the number of hidden neuron layers, the number of neurons in each layer, the number of output neurons, as well as individual transfer functions for each network layer; sigmoid, tanh or linear.

5.5.5. The learning process - training the network

The learning process when using a neural network is very similar to the tabular case, the key difference being, of course, that the value function is represented by the network instead of the table. This substitution is fairly transparent for the learning agent, whose interface to the network is very much like the interface to the table.

A learning session is started by making two learning agents play against each other. These two agents are both given a reference to the same neural network, which represents the value function. The network has been constructed to receive input in the form of a 9-element array of floating point constants. We have chosen to make the network output a single value from 0.0 to 1.0, i.e., one output-neuron using a sigmoid transfer function. The value of the output indicates the desirability of the board which is "fed" as input to the network; the higher the value, the better the state. Network weights are initialized with random values between -0.5 and 0.5. The agents play against each other by making moves on the basis of the values given by the neural network, at the same time as updating the network, using gradient descent, after each move is made.

The game board is still represented by a 10-element integer array, where the first 9 elements constitute the board itself, and the tenth element contains the reward associated with this board. Let us briefly consider a single step in a learning episode (game):

We are two steps into a game of TTT. The game board currently looks like the board shown in Figure 16.

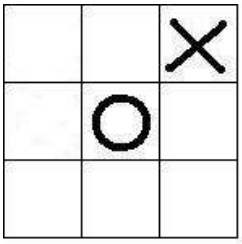

Figure 16

This means that it is the first agent's(X) turn to make a move. The agent receives the following integer array representing the board above. The zeros are empty squares, 1 is X and 2 is O:

```
[0,0,1,0,2,0,0,0,0,0]
```

The next step is to iterate through the elements of this array (except for the last element, containing the reward), in order to find the empty squares. For each empty square, the agent constructs an array of floating point constants, representing the board after that empty square is filled by the X-player (the current agent). These floating point arrays are representations of boards resulting from all possible immediate moves made by the current agent, in response to the current board. In contrast to the integer array representation, the floating point array represents an O with -1 instead of 2. This is due to our assumption that this would be a more suitable input to a neural network. In this particular case, the possible resulting boards would be:

```
[1.0, 0.0, 1.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 1.0, 1.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 1.0, 1.0, -1.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 1.0, 0.0, -1.0, 1.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 1.0, 0.0, -1.0, 0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 1.0, 0.0, -1.0, 0.0, 0.0, 1.0, 0.0]
[0.0, 0.0, 1.0, 0.0, -1.0, 0.0, 0.0, 0.0, 1.0]
```

One by one, the agent propagates these arrays forward through the network. It then compares the corresponding output values and stores the move leading to the highest output value. This is the greedy move. The move is not yet made. First it is time to train the network in terms of the previous move the current agent made, i.e., the move that led to the board

```
[0,0,1,0,0,0,0,0,0,0]
```

To do this, the agent first feeds this board as input to the network, in the form of a floating point array:

```
[0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

The resulting output value is stored in a variable, let us call it **prevValue**. **PrevValue** is the current $\mathbf{V(s_{t-1})}$, i.e., the value of the previous state. Next, the agent feeds the board representing the current optimal next-move, to the network, let us say the optimal move is this:

```
[0.0, 0.0, 1.0, 0.0, -1.0, 0.0, 1.0, 0.0, 0.0]
```

The resulting output value is also stored in a variable, let us call it **curMaxValue**. This is the value of the next state representing an optimal move for the X-player, according to the current internal weights of the neural network. Next, the agent calculates an updated value for the previous state, based on the Q-learning statement. Let us call the new value **newPrevValue**. The statement calculating this value is as follows:

newPrevValue = ((reward+100)/200)+d_rate*curMaxValue-prevValue

The substatement **((reward+100)/200)** is included to scale the reward down to values which the

sigmoid function is able to represent, since the original rewards span from -100 to 100. **D_rate** is the discount rate, which we have chosen to be 1 in all our experiments. The resulting value indicates a target value for the previous state. This means that the previous state, paired with **newPrevValue** can be presented to the neural network as a training example:

nn.train([0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], newPrevValue)

When this is done, it is time for the agent to finally make its move. Which move it makes, depends on the value of the **ε**-parameter, indicating the degree of exploration (if **ε** = 0.1, 10% of the moves made will be random). The agent takes the greedy action found earlier, with probability 1- **ε**, and a random action with probability **ε**. The environment receives the move, updates the board and, providing the game isn't over, passes the board to the opposing agent. And so, the learning process continues.

5.5.6. Results

After running training session experiments for about half a day, we have obtained results which we are quite satisfied with. In our experiments, we have linearly interpolated **ε** between 1 and 0, moving gradually from total exploration to total exploitation. The learning rate, used by the backpropagation algorithm, has been linearly interpolated between 0.2 and 0.01. We have experimented with different network architectures, and the network that has given the best results so far, has got one hidden layer, consisting of 15 neurons. Both the hidden layer neurons and the output neurons use the sigmoid transfer function. This network was trained for 1 million episodes, and performance results are shown in Table 3 (based on 100.000 test games).

X vs. O	***X wins***	***O wins***	***Draws***
L_Agent vs. Minimax	0,00%	0,00%	100,00%
Minimax vs. L_Agent	0,00%	0,00%	100,00%
R_Agent vs. Minimax	0,00%	80,50%	19,50%
Minimax vs. R_Agent	99,50%	0,00%	0,50%
L_Agent vs. R_Agent	90,50%	0,00%	9,50%
R_Agent vs. L_Agent	4,50%	73,50%	22,00%

Table 3. Results from training an RL agent to play Tic-Tac-Toe, using a neural network as the value function.

These results indicate a slight decline in performance compared to the earlier experiments using tabular value-functions. When playing X, the learning agent now wins in 90.5% of the games when facing the random-agent, as opposed to 99% when we used a table. Furthermore, when playing O,

the learning agent now wins in 73.5% of the games when facing the random-agent, in contrast to 91% wins earlier. The learning agent even loses to the random-agent in 4.5% of the games. Earlier it could not be beaten by the random-agent. Its performance against the minimax-algorithm has not changed, however, as all games between these two opponents end in a draw.

Despite of this decline in performance, we feel that the results of using a neural network for representing the value function, are quite good. Learning has clearly taken place. All the training sessions using the network have been performed in the course of a few hours, and there is still a great deal of exploration, various network architectures and parameter adjusting to try, in order to increase performance. In addition, we think it could be rewarding to provide a richer board representation to the network, instead of just indicating the positions. Finally, the results presented here were obtained after training just one network. By training multiple networks with the same parameters, we might have obtained a network that performed better than the one shown here.

5.6. Synthesizing Connect4 evaluation functions using Reinforcement Learning

Given the size of the state space of Connect4, it is not practical to use a tabular state-value function. Our choice has instead been to use an artificial neural network [Mitchell, 1997] for this task, as described in the previous sub section. Using a neural network is more challenging than using a table, since the network has to generalize over the entire state set, and adjusting the network for one input state, potentially changes the output value of all other states as well. As a result, the convergence towards an optimal playing strategy is far more uncertain. In the case of Connect4, using a simple linear representation of the evaluation function might have worked just as well as, or perhaps better than, using a neural network. Both the IBEF evaluation function, and the function generated by ADATE are linear functions. However, the fact that these two functions are linear, made it all the more interesting to explore a function of a different nature. Furthermore, programming the neural network, its interconnected neurons and incremental training procedures was a tempting challenge, as well as a rewarding educational experience.

5.6.1. The Reinforcement Learning system

In order to apply Reinforcement Learning to the task of playing Connect4, we developed a Reinforcement Learning System, including a neural network, using the Java programming language. The main components of the Reinforcement Learning System are:

- The Connect4 module

This module represents the nature of the game, so to speak. It encompasses the game logic, enforces the rules, requests moves of the players when appropriate, receives moves from the players, manages the game board and distributes Reinforcement Learning rewards associated with board states.

- The Reinforcement Learning agent

This is the component which is supposed to learn to play Connect4, by interacting with its environment. This environment consists of the Connect4 module above, and the opposing player, whether this is a learning agent (self-play), or another type of player (e.g., a minimax player). The learning agent uses a state-value function to make decisions (moves), and the learning consists of continually updating this state-value function. In the case of our system, the value function is an artificial neural network. When it comes to self-play, two agent instances each maintain their own separate network, although this is not mandatory; the two agent instances could easily have used and updated the same network.

- The Neural Network (state-value function)

This component is a feed-forward neural network with one hidden layer. The number of inputs, hidden layer neurons and outputs are modifiable, and so are the learning rate and momentum. Transfer functions can be chosen independently for the hidden- and output layer, and can be set to be a hyperbolic tangent, sigmoid or linear function. The training method used is stochastic gradient descent by means of the backpropagation algorithm.

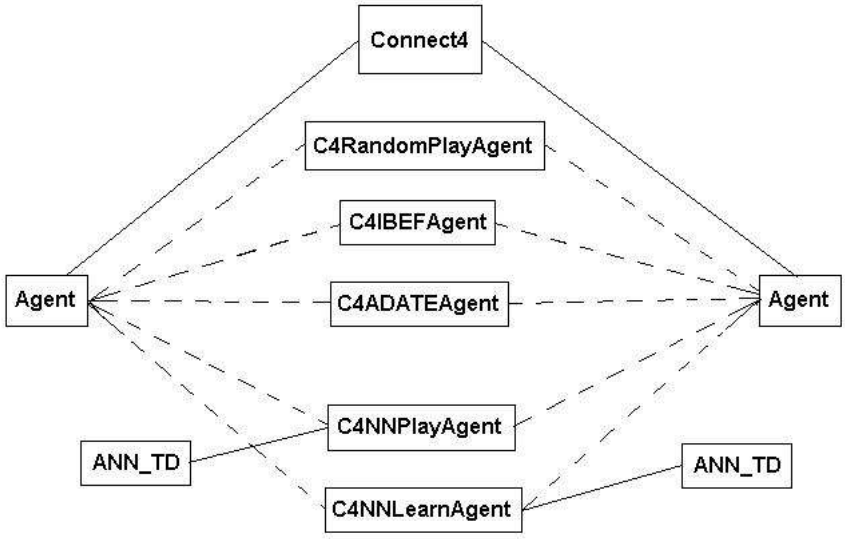

Figure 17. Reinforcement Learning System architecture.

Figure 17 shows an overview of the Reinforcement Learning System. The Connect4 module, encompassing the game logic, must be associated with two playing agents, one of them playing X, and the other playing O. The checkered lines from the Agent modules in the diagram lead to the different kinds of possible agents in the system:

5.6.1.1. Types of agents

- C4RandomPlayAgent

This agent chooses a random move when presented with a game board.

- C4IBEFAgent

This agent plays Connect4 using the minimax algorithm with alpha-beta pruning and the IBEF board evaluation function at the cutoff points in the game tree.

- C4ADATEAgent

This agent plays Connect4 using the minimax algorithm with alpha-beta pruning and an ADATE-generated board evaluation function at the cutoff points in the game tree.

- C4NNPlayAgent

This agent plays Connect4 using the minimax algorithm with alpha-beta pruning and a neural network board evaluation function, obtained from a previous learning session, at the cutoff points in the game tree.

- C4NNLearnAgent

This agent represents the learning activity in the RL system. Like the C4NNPlayAgent, this agent plays Connect4 using a neural network evaluation function, but it also adjusts this very same neural network while playing, according to the experience gathered during the game. In addition to this, the C4NNLearnAgent does not employ the minimax algorithm while playing and learning; it only looks one move ahead in the game. The learning procedure is explained in detail in the next section.

5.6.1.2. Learning scenarios

This system structure gives rise to a number of different learning situations. Of course, it only makes sense if at least one of the agents is a C4NNLearnAgent, otherwise no learning will occur. Here are some possible scenarios:

- A C4NNLearnAgent can play against a C4RandomPlayAgent

In this scenario, the learning agent plays against the agent which makes randomly selected moves. This rarely constitutes a useful training experience for the learning agent, as the random playing agent is usually defeated very early in a given game. Because of this, a vast number of board states are never reached.

- A C4NNLearnAgent can play against a C4IBEFAgent

This is similar to ADATE's training experience described earlier in the thesis, and is the approach that has shown the most promise at this point.

- A C4NNLearnAgent can play against a C4ADATEAgent

This makes it possible for the RL agent to train against an agent which uses an ADATE-generated board evaluation function.

- A C4NNLearnAgent can play against a C4NNPlayAgent

In this scenario, the learning agent plays against an agent which uses a fixed, previously trained neural network board evaluation function. This makes it possible to train networks in a circular fashion, by first training one RL agent against some opponent, and then training another RL agent against the original RL agent, then a third RL agent against the second RL agent, and so on.

- A C4NNLearnAgent can play against another C4NNLearnAgent

In this scenario, two learning agents play against each other while incrementally updating their separate, or alternatively, a shared neural network evaluation function. This concept is usually called self-play.

Of course, it is also possible to combine two or more of the scenarios described above. A C4NNLearnAgent could for example initially be trained using self-play, perhaps in order to capture the basic strategies of the game. After this initial training, the C4NNLearnAgent could be trained against a C4IBEFAgent, in order to further refine the neural network. After this is done, a new C4NNLearnAgent could be introduced and trained against the original C4NNLearnAgent. The possibilities are endless. In addition to this, the cutoff depth of the C4NNLearnAgent's opponents can be varied during training, should it be desirable.

The source code of the central Java classes of the Reinforcement Learning system can be found in Appendix C.

5.6.2. Making moves and learning

When the learning agent receives a game board from the Connect4 module, and is required to make a move, it considers all possible immediate moves from this board. It iterates through all these moves, constructs the resulting game boards (resulting from applying each move to the present game board), and feeds them one by one to the neural network representing the state-value function.

The neural network outputs a double precision value evaluating each board. The move corresponding to the result-state with the highest evaluated value, is marked as the best (greedy) move.

At this point it seems appropriate to describe the input representation of the game board which is fed to the neural network. While playing, the game board is represented as a two-dimensional integer array. When a game board is presented to the network, it is converted from a two-dimensional integer array to a one-dimensional double precision array. It is done in the following manner: Each element in the two-dimensional array is given three entries in the new array, i.e the resulting array consists of (6*7)*3 = 126 elements. As an example, the first element of the first array in the original two-dimensional array (denoted $original_{0,0}$), corresponds to the three first elements in the new one-dimensional array (denoted new_0, new_1, new_2). If the beginning player occupies $original_{0,0}$, i.e., if $original_{0,0}$ is 1, new_0 is set to be 1.0, new_1 is set to be 0.0, and new_2 is also set to be 0.0. If $original_{0,0}$ is 2 however, new_0 is set to be 0.0, new_1 is set to be 1.0, and new_2 is set to be 0.0. If $original_{0,0}$ is 0 (the square is empty), new_0 and new_1 are set to be 0.0, while new_2 is 1.0. The neural network is consequently configured to receive 126 inputs, and present one output. So far we have used sigmoid functions in both hidden- and output layers. The number of hidden layer neurons has been between 5 and 20. Figure 18 serves as a simple illustration of the structure of the neural network.

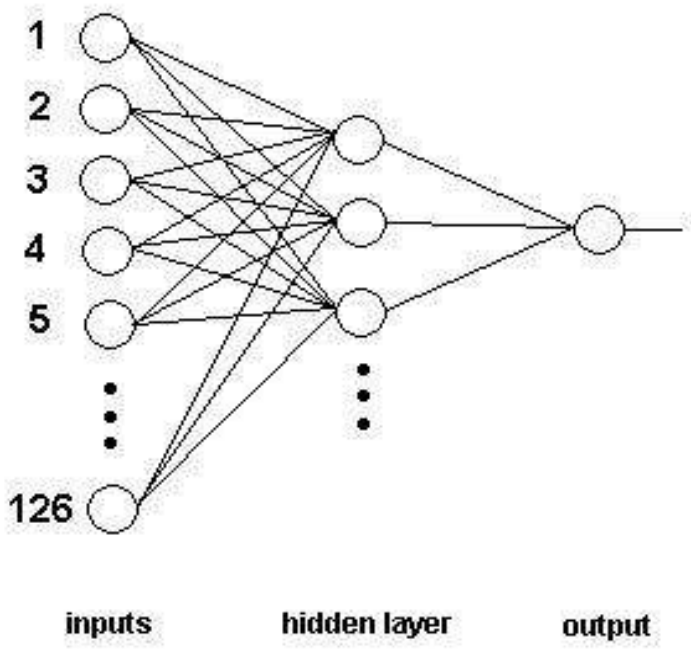

Figure 18. Neural Network structure.

If the current move is not the first move made by the agent in the current episode, the agent will adjust the state value of the resulting state of its previous move, in terms of the greedy move it just found. This is done in four steps:

- 1. Scale the network output (evaluation) corresponding to the current greedy move.

Since the output layer uses a sigmoid transfer function, the network outputs values between 0 and 1, which we scale up to the range -100 <-> 100 using the following formula

```
curValue = (value-0.5)*250
```

The reason we multiply it with 250, is that we train the network with target values between 0.1 and 0.9 instead of 0 and 1, as this is a common practice with neural networks.

- 2. Compute the new estimated value of the resulting board of the previous move made by this agent.

This is done using the following formula (*reward* denotes the immediate reward associated with the board recently received by this agent)

```
prevValue = reward+discount_rate*curValue
```

- 3. Scale the result down to a range appropriate for network training.

That is, between 0.1 and 0.9. This is done with the formula

```
prevValue = (prevValue + 125)/250
```

- 4. Train the network with the acquired value.

The acquired value is a new, and hopefully improved, value estimate of the board resulting from the agent's previous move. The board representation (126-element array) of this board has been saved, and is presented to the network, along with the newly computed value, like this:

```
nn.train(prevBrdAndMove, prevValue)
```

When the update has been made, it is time for the agent to perform the move. It already knows what the greedy move is, and will take this move with a probability corresponding to 1-**ε**. With probability ε it will make a move completely at random. The value of **ε** indicates the degree of exploration.

5.7. Results

Many experiments have been run, adjusting training parameters like number of games played, exploration rate, learning rate for the network, number of neurons in the hidden layer and so on. We have tried training the agent by making it play against a copy of itself, and against the minimax algorithm using IBEF with various search levels. The best results so far have been obtained by playing approximately 100.000 games against the IBEF algorithm. The resulting agents have shown signs of "intelligence", and have proven to be able to play a solid game of Connect4. They can often make naive mistakes, from the point of view of a human being, while seemingly being able to cleverly plan ahead in certain situations. When training the network whose results are presented here, the learning rate, α, was linearly interpolated from 0.2 to 0.01 during the episodes, the exploration rate, ε, was linearly interpolated from 0.5 to 0, the discount rate, γ, was set to 1.0, and the lambda parameter, λ (indicating use of eligibility traces), was set to 0.6. The network has one hidden layer, containing 15 neurons. The network has, as mentioned, been combined with the minimax algorithm.

5.7.1. The RL-function vs. the IBEF-function

As a contrast to the ADATE-generated function, the RL-function is strongest when it is the X-player, against the IBEF-function. When put up against the IBEF-function at equal cutoff depths, up to and including depth 6, the RL-function is superior at depths 1, 3, 4 and 5. Despite a few anomalies, the result statistics of the X-playing RL-function have much fewer irregularities than the the results of the X-playing ADATE-function. Moreover, the results show that the X-playing RL-function is consistently strong against the IBEF-function, and that it in several cases is able to defeat the IBEF-function, when the IBEF-function is using a deeper search than itself. When playing X, the RL-function's weak point seems to be using cutoff depth 2, but aside from this, it constitutes a very strong opponent for the O-Playing IBEF-function.

As an O-Player, the RL-function is a bit weaker than it is when it plays X. When facing an X-Playing IBEF-function at equal cutoff depths, the RL-function is at a strong advantage at depths 1 and 2, while being inferior at depths 3, 4, 5 and 6. However, none of the losses at these four depths can be considered devastating; only at cutoff depth 6 is the RL-function's share of wins below 25%. In fact, out of the 36 game combinations played between the X-playing IBEF-function and the O-playing RL-function, up to and including cutoff depth 6, the RL-function is able to secure a fair share of wins in most cases. Nevertheless, the O-playing RL-function is slightly inferior to the X-playing IBEF-function.

Despite of this, the neural network evaluation function is a strong opponent when facing the IBEF-function. Comparing the two functions is in this case a bit difficult, as the race is somewhat close, but we feel that the RL-function's strong advantage when playing X, justifies us saying that an improvement has been made over the IBEF-function. Some examples of this function's playing ability against the IBEF function, are shown in Table 4.

XPlayer/depth	*OPlayer/depth*	*XWins*	*OWins*	*Draws*
RL/1	IBEF/1	100,00%	0,00%	0,00%
RL/1	IBEF/3	89,80%	10,20%	0,00%
RL/2	IBEF/2	22,00%	68,00%	10,00%
RL/3	IBEF/3	85,80%	0,00%	14,20%
RL/4	IBEF/4	75,10%	24,90%	0,00%
RL/4	IBEF/6	85,00%	15,00%	0,00%
IBEF/1	RL/1	26,30%	73,50%	0,20%
IBEF/2	RL/1	56,60%	43,30%	0,10%
IBEF/2	RL/2	41,70%	52,20%	6,10%
IBEF/3	RL/3	64,40%	32,00%	3,60%
IBEF/4	RL/4	65,70%	25,70%	8,60%

Table 4. Selections from the RL-function's winning statistics against the IBEF function.

A more thorough collection of result statistics can be found in Appendix D.

6. Conclusion

We have now examined the application of two different machine learning techniques to the domain of game-playing. When using the ADATE system, we have utilized Automatic Programming to search for a strong board evaluation function, by using the function's playing ability against the IBEF-function as a measure of fitness. More specifically, the function's fitness is measured by how many games it wins against the IBEF-function, the games starting at a set of various start board states. To accomplish this, we have developed an extensive specification file for the system, containing, among other things, a Connect4 playing system. This has been a novel way of using the ADATE system, and the specification file is the largest and most complicated ever presented to ADATE. This experience has inspired us and given us exciting ideas on the the possible future uses of the ADATE system.

Our use of the second method, Reinforcement Learning, has been inspired by Gerald Tesauro's backgammon playing program, TD-Gammon [Tesauro, 1995]. We have programmed a Connect4 playing system, with attachable playing agents of different kinds, the most important being, of course, the Reinforcement Learning agent. The other types of agents provide training opposition for the RL agent. In our case, the RL agent is attached to a neural network, which it uses to make its moves when playing. At the same time, the agent modifies this network by passing training examples to it, gathered from the playing experience. Thus, the agent learns continuously during the playing process. The neural network resulting from this training experience, is in effect a Connect4 board evaluation function.

6.1. Comparing the two techniques

Both methods used have resulted in functions which are able to play well against the IBEF-function. The ADATE System requires a substantial amount of running time, and the synthesized function presented in this paper, being the result of running the system for a week, is therefore merely a step on the ladder towards the optimal function. Nevertheless, the function synthesized by ADATE represents a solid opponent to the IBEF algorithm, and it seems to be especially strong when playing O, even when it is combined with a low cutoff depth.

Reinforcement Learning required less running time than ADATE, and the function presented in this paper was obtained after running about 45 minutes on a single laptop. The challenge of Reinforcement Learning however, lies in adjusting the various training parameters, the size and details of the neural network, the opposition used for training, and so on. Thus, many possibilities and combinations remain untried. The function presented here is quite complex; a neural network with 126 inputs, 15 hidden neurons and one output, and it is a quite strong player. However, comparing the Reinforcement Learning function with the ADATE-function is not a trivial task, since they seem to have different strengths, and clear result patterns are difficult to extract. For instance, the ADATE-function has a clear advantage over the RL-function, when the RL-function uses cutoff depth 3.

6.2. The ADATE-function vs. the RL-function

When the X-playing RL-function plays against the O-playing ADATE-function, the RL-function is especially strong when playing with cutoff depth 2. In this case, it is able to win all games against the ADATE-function using cutoff depths 1-6, which is worth pointing out. Moreover, the X-playing RL-function with cutoff depth 1 wins all games against the O-playing ADATE-function using cutoff depths 1-3, and about 95% of the games against the O-playing ADATE-function using cutoff depth 4. The X-playing RL-function is also very strong at cutoff depth 4. On the other hand, when the X-playing RL-function uses cutoff depths 3, 5 and 6, the O-playing ADATE-function is clearly at an advantage, strongly dominating the RL-function. The X-playing RL-function is particularly weak when using cutoff depth 3. When making the X-playing RL-function and the O-playing ADATE-function play against each other at equal cutoff depths, up to and including depth 6, the RL-function is at a strong advantage at depths 1, 2, 4, 6, while the opposite is true for depths 3 and 5.

When the X-playing ADATE-function plays against the O-playing RL-function, the playing results contain less anomalies than when the RL-function plays X, and there are fewer surprises present. In most cases, the ADATE-function seems to win a smaller share of games as the cutoff depth of its opponent, in this case the RL-function, increases, and vice versa. Of course, exceptions exist. For example, the X-playing ADATE-function using cutoff depth 2, is able to convincingly defeat the O-playing RL-function using cutoff depths 5 and 6. When making the X-playing ADATE-function play against the O-playing RL-function at equal depths, up to and including depth 6, the ADATE-function is the stronger function at depths 3, 5 and 6, devastatingly so at the two last depths. At depths 1, 2 and 4, however, the RL-function has the upper hand. All in all, when making the ADATE-function and the RL-function play each other, the RL-function seems to have a slight advantage, although the results are a bit irregular and difficult to interpret. On the other hand, the ADATE-function is stronger against the IBEF-function than the RL-function is, even though both of them are better than the IBEF-function.

Some results obtained by making the ADATE-synthesized function play against the RL-function is shown in Table 5.

XPlayer/depth	***OPlayer/depth***	***XWins***	***OWins***	***Draws***
ADATE/1	RL/1	44,70%	55,30%	0,00%
RL/1	ADATE/1	100,00%	0,00%	0,00%
RL/2	ADATE/2	100,00%	0,00%	0,00%
RL/3	ADATE/1	7,30%	92,70%	0,00%
ADATE/3	RL/3	56,00%	29,60%	14,40%
ADATE/4	RL/4	6,00%	68,00%	26,00%
RL/4	ADATE/4	70,00%	29,00%	1,00%
RL/1	ADATE/4	95,70%	4,30%	0,00%
RL/5	ADATE/5	0,00%	100,00%	0,00%

Table 5: Results from letting the ADATE-function and the RL-function play against each other.

A more extensive collection of results can be found in Appendix D.

6.3. Project evaluation

When examining the results presented in this thesis, it is clear that a great deal of learning has occurred, and that both methods have shown to be suitable for the task of synthesizing Connect4 board evaluation functions. Moreover, it seems that further improvements are within reach, given sufficient time and opportunity to run experiments and tweak training parameters and training opposition. The experience of attacking the problem from two different angles like this, has been very interesting and challenging, and valuable insight has been gained into the problem domain, as well as into the machine learning techniques applied to the problem. The programming needed to carry out the project has been demanding, and at times frustrating, but has ultimately constituted a rewarding experience.

6.4. Future work

When considering further possible extensions of the work presented in this thesis, a few key ideas spring to mind. First of all, the ADATE system should be granted substantially more running time, in order to further improve the synthesized evaluation function. We have good reason to believe that a great deal of improvement can be obtained by letting the ADATE system run for a longer period of time. Furthermore, it might be interesting to find a ADATE-synthesized function by using a generational approach, in the following manner: First, synthesize a function by letting it play against some opponent, like we have done in our experiments. Next, take the newly synthesized function, and use it as an opponent for synthesizing a new function, and so on. This type of approach has already been described earlier in the thesis, but in regards to the Reinforcement Learning system.

As mentioned, a myriad of possibilities remain untried when it comes to the Reinforcement Learning system. There are a lot of variables involved in the training process, e.g., the training opposition, the number of training games, the exploration rate, the learning rate, the architecture of the network, and so on. In our experiments, time has limited our exploration of these variables, and a better configuration of training circumstances than those presented here, can surely be conjured. Another possible area of exploration would be the nature of the evaluation function. In our experiments, we have used a neural network, but there are numerous other possible function forms. We have seen that both the IBEF-function and the ADATE-function are linear functions, and they seem to be able to represent the game board in a satisfactory manner. Therefore, it might be interesting to let the Reinforcement Learning agent use a linear function instead of a neural network.

7. References

[Sutton & Barto, 1998] Sutton, R. and Barto, A. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA.

[Brassard & Bratley, 1996] Brassard, G. and Bratley, P. *Fundamentals of Algorithmics.* Prentice-Hall, New Jersey, pp. 317 – 319.

[Tesauro, 1995] Tesauro, G. Temporal Difference Learning and TD-Gammon. *Communications of the ACM, March 1995 / Vol. 38, No. 3.*

[Mitchell, 1997] Mitchell, T. *Machine Learning.* McGraw-Hill, pp. 81 -126.

[Allis, 1988] Allis, V. A Knowledge-based Approach of Connect-Four. *Master Thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands.*

[Olsson, 1995] Olsson, R. Inductive functional programming using incremental program transformation. *Artificial Intelligence, Vol 74, Nr. 1,* pp. 55-83.

8. Acknowledgments

I would like to thank my thesis supervisor, Roland Olsson, for his help and guidance during my work on this master thesis.

A. The ADATE specification file

The following is the ADATE specification file for the Connect4 example, playC4.spec. In this appendix, we have chosen to remove most of the starting/validation board states, for presentation purposes. The file in its entirety is 137 pages long, and we found it unnecessary to include all the board states. The file has been edited two places, and these places are marked by the tag <cut>.

```
datatype matrix = matrix of int * int * int uncheckedArray
datatype intList  = nill | cons of int * intList
datatype intListList = nill' | cons' of intList * intListList
datatype intPair = intPair of int * int
datatype gameResult = gameResult of int * int
datatype mainInput = mainInput of intListList * bool * bool * int * int
datatype mainReturn = mainReturn of bool * int

fun not( X : bool ) : bool =
  case X of false => true | true => false

fun length( Xs : intList ) : int =
  case Xs of
    nill => 0
  | cons( X1, Xs1 ) => 1 + length Xs1

fun length'( Xss : intListList ) : int =
  case Xss of
    nill' => 0
  | cons'( Xs1, Xss1 ) => 1 + length' Xss1

fun sub( ( M, Row, Col ) : matrix * int * int ) : int =
  case M of matrix( NumRows, NumCols, A ) =>
  case Row < NumRows of
    false => raise D2
  | true =>
  case Col < NumCols of
    false => raise D2
  | true =>
  case Row < 0 of
    false => (
      case Col < 0 of
        false => uncheckedArraySub( A, Row * NumCols + Col )
      | true => raise D2 )
  | true => raise D2


fun update( ( M, Row, Col, X ) : matrix * int * int * int ) : matrix =
  case M of matrix( NumRows, NumCols, A ) =>
  case Row < NumRows of
    false => raise D2
  | true =>
  case Col < NumCols of
    false => raise D2
  | true =>
  case Row < 0 of
    false => (
      case Col < 0 of
        false => (
          case uncheckedArrayUpdate( A, Row * NumCols + Col, X ) of Dummy => M
          )
      | true => raise D2 )
  | true => raise D2
```

```
fun fillInRow( ( Row, Col, Xs, World )
               : int * int * intList * matrix ) : matrix =
  case Xs of
    nill => World
  | cons( X1, Xs1 ) =>
      fillInRow( Row, Col+1, Xs1, update( World, Row, Col, X1 ) )

fun fillIn( ( Row, Xss, World ) : int * intListList * matrix ) : matrix =
  case Xss of
    nill' => World
  | cons'( Xs1, Xss1 ) =>
      fillIn( Row+1, Xss1, fillInRow( Row, 0, Xs1, World ) )


fun toMatrix( Xss : intListList ) : matrix =
  case Xss of
    nill' => matrix( 0, 0, uncheckedArray( 0, 0 ) )
  | cons'( Xs1, Xss1 ) =>
  case length' Xss of NumRows =>
  case length Xs1 of NumCols =>
  case matrix( NumRows, NumCols, uncheckedArray( NumRows * NumCols, 0 ) )
  of World => fillIn( 0, Xss, World )


fun getRow( ( Row, Col, M ) : int * int * matrix ) : intList =
  case M of matrix( NumRows, NumCols, A ) =>
  case Col < NumCols of
    false => nill
  | true => cons( sub( M, Row, Col ), getRow( Row, Col+1, M ) )

fun getAll( ( Row, M ) : int * matrix ) : intListList =
  case M of matrix( NumRows, NumCols, A ) =>
  case Row < NumRows of
    false => nill'
  | true => cons'( getRow( Row, 0, M ), getAll( Row+1, M ) )

fun fromMatrix( M : matrix ) : intListList = getAll( 0, M )

fun eval(board:matrix):int =
  let
    fun evalHorizontal((board1, row): matrix * int):int =
      case 5 < row of
        true => 0
        | false =>
          sub(board1,row,0) + sub(board1,row,1) + sub(board1,row,2) +
            sub(board1,row,3) +
          sub(board1,row,1) + sub(board1,row,2) + sub(board1,row,3) +
            sub(board1,row,4) +
          sub(board1,row,2) + sub(board1,row,3) + sub(board1,row,4) +
            sub(board1,row,5) +
          sub(board1,row,3) + sub(board1,row,4) + sub(board1,row,5) +
            sub(board1,row,6) + evalHorizontal(board1, row+1)
        in
          let
            fun evalVertical((board2, column): matrix * int):int =
              case 6 < column of
                true => 0
                | false =>
                  sub(board2,0,column) + sub(board2,1,column) +
                  sub(board2,2,column) + sub(board2,3,column) +
                  sub(board2,1,column) + sub(board2,2,column) +
                  sub(board2,3,column) + sub(board2,4,column) +
                  sub(board2,2,column) + sub(board2,3,column) +
                  sub(board2,4,column) + sub(board2,5,column) +
                  evalVertical(board2, column+1)
```

```
        in
          let
            fun evalDiagonal(board3:matrix):int =
              sub(board3,3,0) + sub(board3,2,1) + sub(board3,1,2) +
                 sub(board3,0,3) +
              sub(board3,4,0) + sub(board3,3,1) + sub(board3,2,2) +
                 sub(board3,1,3) +
              sub(board3,3,1) + sub(board3,2,2) + sub(board3,1,3) +
                 sub(board3,0,4) +
              sub(board3,5,0) + sub(board3,4,1) + sub(board3,3,2) +
                 sub(board3,2,3) +
              sub(board3,4,1) + sub(board3,3,2) + sub(board3,2,3) +
                 sub(board3,1,4) +
              sub(board3,3,2) + sub(board3,2,3) + sub(board3,1,4) +
                 sub(board3,0,5) +
              sub(board3,5,1) + sub(board3,4,2) + sub(board3,3,3) +
                 sub(board3,2,4) +
              sub(board3,4,2) + sub(board3,3,3) + sub(board3,2,4) +
                 sub(board3,1,5) +
              sub(board3,3,3) + sub(board3,2,4) + sub(board3,1,5) +
                 sub(board3,0,6) +
              sub(board3,5,2) + sub(board3,4,3) + sub(board3,3,4) +
                 sub(board3,2,5) +
              sub(board3,4,3) + sub(board3,3,4) + sub(board3,2,5) +
                 sub(board3,1,6) +
              sub(board3,5,3) + sub(board3,4,4) + sub(board3,3,5) +
                 sub(board3,2,6) +
              sub(board3,2,0) + sub(board3,3,1) + sub(board3,4,2) +
                 sub(board3,5,3) +
              sub(board3,1,0) + sub(board3,2,1) + sub(board3,3,2) +
                 sub(board3,4,3) +
              sub(board3,2,1) + sub(board3,3,2) + sub(board3,4,3) +
                 sub(board3,5,4) +
              sub(board3,0,0) + sub(board3,1,1) + sub(board3,2,2) +
                 sub(board3,3,3) +
              sub(board3,1,1) + sub(board3,2,2) + sub(board3,3,3) +
                 sub(board3,4,4) +
              sub(board3,2,2) + sub(board3,3,3) + sub(board3,4,4) +
                 sub(board3,5,5) +
              sub(board3,0,1) + sub(board3,1,2) + sub(board3,2,3) +
                 sub(board3,3,4) +
              sub(board3,1,2) + sub(board3,2,3) + sub(board3,3,4) +
                 sub(board3,4,5) +
              sub(board3,2,3) + sub(board3,3,4) + sub(board3,4,5) +
                 sub(board3,5,6) +
              sub(board3,0,2) + sub(board3,1,3) + sub(board3,2,4) +
                 sub(board3,3,5) +
              sub(board3,1,3) + sub(board3,2,4) + sub(board3,3,5) +
                 sub(board3,4,6) +
              sub(board3,0,3) + sub(board3,1,4) + sub(board3,2,5) +
                 sub(board3,3,6)
        in
          evalHorizontal(board, 0) + evalVertical(board, 0) +
                 evalDiagonal(board)
        end
      end
    end

fun f(board:matrix):int =
  let
    fun evalHorizontal((board1, row): matrix * int):int =
      case 5 < row of
        true => 0
        | false =>
          sub(board1,row,0) + sub(board1,row,1) + sub(board1,row,2) +
            sub(board1,row,3) +
```

```
  sub(board1,row,1) + sub(board1,row,2) + sub(board1,row,3) +
    sub(board1,row,4) +
  sub(board1,row,2) + sub(board1,row,3) + sub(board1,row,4) +
    sub(board1,row,5) +
  sub(board1,row,3) + sub(board1,row,4) + sub(board1,row,5) +
    sub(board1,row,6) + evalHorizontal(board1, row+1)
in
  let
    fun evalVertical((board2, column): matrix * int):int =
      case 6 < column of
        true => 0
        | false =>
          sub(board2,0,column) + sub(board2,1,column) +
          sub(board2,2,column) + sub(board2,3,column) +
          sub(board2,1,column) + sub(board2,2,column) +
          sub(board2,3,column) + sub(board2,4,column) +
          sub(board2,2,column) + sub(board2,3,column) +
          sub(board2,4,column) + sub(board2,5,column) +
          evalVertical(board, column+1)
 in
   let
     fun evalDiagonal(board3:matrix):int =
       sub(board3,3,0) + sub(board3,2,1) + sub(board3,1,2) +
          sub(board3,0,3) +
       sub(board3,4,0) + sub(board3,3,1) + sub(board3,2,2) +
          sub(board3,1,3) +
       sub(board3,3,1) + sub(board3,2,2) + sub(board3,1,3) +
          sub(board3,0,4) +
       sub(board3,5,0) + sub(board3,4,1) + sub(board3,3,2) +
          sub(board3,2,3) +
       sub(board3,4,1) + sub(board3,3,2) + sub(board3,2,3) +
          sub(board3,1,4) +
       sub(board3,3,2) + sub(board3,2,3) + sub(board3,1,4) +
          sub(board3,0,5) +
       sub(board3,5,1) + sub(board3,4,2) + sub(board3,3,3) +
          sub(board3,2,4) +
       sub(board3,4,2) + sub(board3,3,3) + sub(board3,2,4) +
          sub(board3,1,5) +
       sub(board3,3,3) + sub(board3,2,4) + sub(board3,1,5) +
          sub(board3,0,6) +
       sub(board3,5,2) + sub(board3,4,3) + sub(board3,3,4) +
          sub(board3,2,5) +
       sub(board3,4,3) + sub(board3,3,4) + sub(board3,2,5) +
          sub(board3,1,6) +
       sub(board3,5,3) + sub(board3,4,4) + sub(board3,3,5) +
          sub(board3,2,6) +
       sub(board3,2,0) + sub(board3,3,1) + sub(board3,4,2) +
          sub(board3,5,3) +
       sub(board3,1,0) + sub(board3,2,1) + sub(board3,3,2) +
          sub(board3,4,3) +
       sub(board3,2,1) + sub(board3,3,2) + sub(board3,4,3) +
          sub(board3,5,4) +
       sub(board3,0,0) + sub(board3,1,1) + sub(board3,2,2) +
          sub(board3,3,3) +
       sub(board3,1,1) + sub(board3,2,2) + sub(board3,3,3) +
          sub(board3,4,4) +
       sub(board3,2,2) + sub(board3,3,3) + sub(board3,4,4) +
          sub(board3,5,5) +
       sub(board3,0,1) + sub(board3,1,2) + sub(board3,2,3) +
          sub(board3,3,4) +
       sub(board3,1,2) + sub(board3,2,3) + sub(board3,3,4) +
          sub(board3,4,5) +
       sub(board3,2,3) + sub(board3,3,4) + sub(board3,4,5) +
          sub(board3,5,6) +
       sub(board3,0,2) + sub(board3,1,3) + sub(board3,2,4) +
          sub(board3,3,5) +
```

```
              sub(board3,1,3) + sub(board3,2,4) + sub(board3,3,5) +
                 sub(board3,4,6) +
              sub(board3,0,3) + sub(board3,1,4) + sub(board3,2,5) +
                 sub(board3,3,6)
         in
           evalHorizontal(board, 0) + evalVertical(board, 0) +
            evalDiagonal(board)
         end
       end
     end

fun and5((v,w,x,y,z):int*int*int*int*bool):bool =
  case z of
    false => false
    | true =>
      case v = w of
        false => false
        | true =>
          case w = x of
            false => false
            | true =>
              case x = y of
                false => false
                | true => true


fun goHorizontal((board,row):matrix*int):bool =
  case 5 < row of
    true => false
    | false =>
      case and5(sub(board,row,0),sub(board,row,1),sub(board,row,2),
            sub(board,row,3),not(sub(board,row,3)=0)) of
        true => true
        | false =>
          case and5(sub(board,row,1),sub(board,row,2),sub(board,row,3),
                  sub(board,row,4),not(sub(board,row,4)=0)) of
            true => true
            | false =>
              case and5(sub(board,row,2),sub(board,row,3),sub(board,row,4),
                  sub(board,row,5),not(sub(board,row,5)=0)) of
                true => true
                | false =>
                  case and5(sub(board,row,3),sub(board,row,4),sub(board,row,5),
                        sub(board,row,6),not(sub(board,row,6)=0)) of
                    true => true
                    | false =>
                      goHorizontal(board, row+1)


fun goVertical((board,column):matrix*int):bool =
  case 6 < column of
    true => false
    | false =>
      case and5(sub(board,0,column),sub(board,1,column),sub(board,2,column),
            sub(board,3,column),not(sub(board,3,column)=0)) of
        true => true
        | false =>
          case and5(sub(board,1,column),sub(board,2,column),sub(board,3,column),
                  sub(board,4,column),not(sub(board,4,column)=0)) of
            true => true
            | false =>
              case and5(sub(board,2,column),sub(board,3,column),
                  sub(board,4,column),sub(board,5,column),
                  not(sub(board,5,column)=0)) of
                true => true
                | false =>
```

```
                goVertical(board, column+1)

fun goDiagonal(board:matrix):bool =
  case and5(sub(board,3,0),sub(board,2,1),sub(board,1,2),sub(board,0,3),
      not(sub(board,0,3)=0)) of
    true => true
    | false =>
  case and5(sub(board,4,0),sub(board,3,1),sub(board,2,2),sub(board,1,3),
      not(sub(board,1,3)=0)) of
    true => true
    | false =>
  case and5(sub(board,3,1),sub(board,2,2),sub(board,1,3),sub(board,0,4),
      not(sub(board,0,4)=0)) of
    true => true
    | false =>
  case and5(sub(board,5,0),sub(board,4,1),sub(board,3,2),sub(board,2,3),
      not(sub(board,2,3)=0)) of
    true => true
    | false =>
  case and5(sub(board,4,1),sub(board,3,2),sub(board,2,3),sub(board,1,4),
      not(sub(board,1,4)=0)) of
    true => true
    | false =>
  case and5(sub(board,3,2),sub(board,2,3),sub(board,1,4),sub(board,0,5),
      not(sub(board,0,5)=0)) of
    true => true
    | false =>
  case and5(sub(board,5,1),sub(board,4,2),sub(board,3,3),sub(board,2,4),
      not(sub(board,2,4)=0)) of
    true => true
    | false =>
  case and5(sub(board,4,2),sub(board,3,3),sub(board,2,4),sub(board,1,5),
      not(sub(board,1,5)=0)) of
    true => true
    | false =>
  case and5(sub(board,3,3),sub(board,2,4),sub(board,1,5),sub(board,0,6),
      not(sub(board,0,6)=0)) of
    true => true
    | false =>
  case and5(sub(board,5,2),sub(board,4,3),sub(board,3,4),sub(board,2,5),
      not(sub(board,2,5)=0)) of
    true => true
    | false =>
  case and5(sub(board,4,3),sub(board,3,4),sub(board,2,5),sub(board,1,6),
      not(sub(board,1,6)=0)) of
    true => true
    | false =>
  case and5(sub(board,5,3),sub(board,4,4),sub(board,3,5),sub(board,2,6),
      not(sub(board,2,6)=0)) of
    true => true
    | false =>
  case and5(sub(board,2,0),sub(board,3,1),sub(board,4,2),sub(board,5,3),
      not(sub(board,5,3)=0)) of
    true => true
    | false =>
  case and5(sub(board,1,0),sub(board,2,1),sub(board,3,2),sub(board,4,3),
      not(sub(board,4,3)=0)) of
    true => true
    | false =>
  case and5(sub(board,2,1),sub(board,3,2),sub(board,4,3),sub(board,5,4),
      not(sub(board,5,4)=0)) of
    true => true
    | false =>
  case and5(sub(board,0,0),sub(board,1,1),sub(board,2,2),sub(board,3,3),
      not(sub(board,3,3)=0)) of
```

```
    true => true
    | false =>
  case and5(sub(board,1,1),sub(board,2,2),sub(board,3,3),sub(board,4,4),
      not(sub(board,4,4)=0)) of
    true => true
    | false =>
  case and5(sub(board,2,2),sub(board,3,3),sub(board,4,4),sub(board,5,5),
      not(sub(board,5,5)=0)) of
    true => true
    | false =>
  case and5(sub(board,0,1),sub(board,1,2),sub(board,2,3),sub(board,3,4),
      not(sub(board,3,4)=0)) of
    true => true
    | false =>
  case and5(sub(board,1,2),sub(board,2,3),sub(board,3,4),sub(board,4,5),
      not(sub(board,4,5)=0)) of
    true => true
    | false =>
  case and5(sub(board,2,3),sub(board,3,4),sub(board,4,5),sub(board,5,6),
      not(sub(board,5,6)=0)) of
    true => true
    | false =>
  case and5(sub(board,0,2),sub(board,1,3),sub(board,2,4),sub(board,3,5),
      not(sub(board,3,5)=0)) of
    true => true
    | false =>
  case and5(sub(board,1,3),sub(board,2,4),sub(board,3,5),sub(board,4,6),
      not(sub(board,4,6)=0)) of
    true => true
    | false =>
  case and5(sub(board,0,3),sub(board,1,4),sub(board,2,5),sub(board,3,6),
      not(sub(board,3,6)=0)) of
    true => true
    | false => false


fun bF((board,column,row): matrix*int*int):bool =
  case 6 < column of
    true => true
    | false =>
      case 5 < row of
        true => bF(board,column+1,0)
        | false =>
          case sub(board,row,column) = 0 of
            true => false
            | false => bF(board,column,row+1)


fun boardFull(board:matrix):bool =
  bF(board,0,0)


fun whoWon(board:matrix):int =
  case goHorizontal(board,0) of
    true => ~100000
    | false =>
      case goVertical(board,0) of
       true => ~100000
       | false =>
         case goDiagonal(board) of
           true => ~100000
           | false => 0


fun eR((board,column,row,xs,os): matrix*int*int*int*int):int =
  case 6 < column of
```

```
    true =>
      (case os < xs of
        true => 100000
        | false => ~100000)
    | false =>
      case 5 < row of
        true => eR(board,column+1,0,xs,os)
        | false =>
          case(case sub(board,row,column) = 1 of true => xs+1 | false => xs)
            of xx =>
          case(case sub(board,row,column) = ~1 of true => os+1 | false => os)
            of oo =>
            eR(board,column,row+1,xx,oo)

fun endResult(board:matrix):int =
  case boardFull(board) of
    true => whoWon(board)
    | false => eR(board,0,0,0,0)


fun gameOver(board:matrix):bool =
  case boardFull(board) of
    true => true
    | false =>
      case goHorizontal(board,0) of
        true => true
        | false =>
          case goVertical(board,0) of
            true => true
            | false =>
              goDiagonal(board)


fun columnNotFull((board,column):matrix*int):bool =
  sub(board,5,column) = 0


fun fFR((board,column,row):matrix*int*int):int =
  case 5 < row of
    true => ~1
    | false =>
      case sub(board,row,column) = 0 of
        true => row
        | false => fFR(board,column,row+1)

fun findFreeRow((board,column):matrix*int):int =
  fFR(board,column,0)

fun fUT((board,column,row):matrix*int*int):int =
  case row < 0 of
    true => ~1
    | false =>
      case sub(board,row,column) = 0 of
        false => row
        | true => fUT(board,column,row-1)

fun findUpperToken((board,column):matrix*int):int =
  fUT(board,column,5)


fun placeToken((board,column,x):matrix*int*bool):matrix =
  update(board,findFreeRow(board,column),column,(case x of true => 1 | false =>
~1))
```

```
fun removeToken((board,column):matrix*int):matrix =
  update(board,findUpperToken(board,column),column,0)


fun minimax((board,x,depth,cutoff):matrix*bool*int*int):int =
  let
    fun findBestMove((board1,x1,bestScore,bestMove,column,depth1,cutoff1):
          matrix*bool*int*int*int*int*int):intPair =
      case 6 < column of
        true => intPair(bestScore, bestMove)
        | false =>
          case columnNotFull(board1,column) of
            false =>
            findBestMove(board1,x1,bestScore,bestMove,column+1,depth1,cutoff1)
            | true =>
              case minimax(placeToken(board1,column,x1), not(x1), depth1+1,
                  cutoff1) of
                score =>
                  (case x1 of
                    true =>
                      (case bestScore < score of
                        true => findBestMove(removeToken(board1,column),
                              x1,score,column,column+1,depth1,cutoff1)
                        | false => findBestMove(removeToken(board1,column),
                              x1,bestScore,bestMove,column+1,depth1,cutoff1))
                    | false =>
                      (case score < bestScore of
                        true => findBestMove(removeToken(board1,column),
                              x1,score,column,column+1,depth1,cutoff1)
                        | false => findBestMove(removeToken(board1,column),
                              x1,bestScore,bestMove,column+1,depth1,cutoff1)))
  in
    case gameOver(board) of
      true => endResult(board)
      | false =>
        case depth = cutoff of
          true => eval(board)
          | false =>
            case x of
              true =>
                (case findBestMove(board,x,~999999,0,0,depth,cutoff) of
                  intPair(bScore,bMove) =>
                    case depth = 0 of
                      true => bMove
                      | false => bScore)
              | false =>
                (case findBestMove(board,x,999999,0,0,depth,cutoff) of
                  intPair(bScore,bMove) =>
                    case depth = 0 of
                      true => bMove
                      | false => bScore)
  end


fun fminimax((board,x,depth,cutoff):matrix*bool*int*int):int =
  let
    fun ffindBestMove((board1,x1,bestScore,bestMove,column,depth1,cutoff1):
                        matrix*bool*int*int*int*int*int):intPair =
      case 6 < column of
        true => intPair(bestScore, bestMove)
        | false =>
          case columnNotFull(board1,column) of
            false => ffindBestMove
                  (board1,x1,bestScore,bestMove,column+1,depth1,cutoff1)
            | true =>
              case fminimax(placeToken(board1,column,x1), not(x1), depth1+1,
```

```
                cutoff1) of
               score =>
                 (case x1 of
                   true =>
                     (case bestScore < score of
                       true => ffindBestMove(removeToken(board1,column),
                            x1,score,column,column+1,depth1,cutoff1)
                       | false => ffindBestMove(removeToken(board1,column),
                            x1,bestScore,bestMove,column+1,depth1,cutoff1))
                   | false =>
                     (case score < bestScore of
                       true => ffindBestMove(removeToken(board1,column),
                            x1,score,column,column+1,depth1,cutoff1)
                       | false => ffindBestMove(removeToken(board1,column),
                            x1,bestScore,bestMove,column+1,depth1,cutoff1)))
  in
    case gameOver(board) of
      true => endResult(board)
      | false =>
        case depth = cutoff of
          true => f(board)
          | false =>
            case x of
              true =>
                (case ffindBestMove(board,x,~999999,0,0,depth,cutoff) of
                  intPair(bScore,bMove) =>
                    case depth = 0 of
                      true => bMove
                      | false => bScore)
            | false =>
                (case ffindBestMove(board,x,999999,0,0,depth,cutoff) of
                  intPair(bScore,bMove) =>
                    case depth = 0 of
                      true => bMove
                      | false => bScore)
  end


fun chooseMove((board,x,cutoff,fPlayer):matrix*bool*int*bool):int =
  case fPlayer of
    false => minimax(board,x,0,cutoff)
    | true => fminimax(board,x,0,cutoff)


fun play((board,fPlaysX,count,xTurn,fSLevel,oSLevel):
           matrix*bool*int*bool*int*int):gameResult =
  case gameOver(board) of
    true => gameResult(count,endResult(board))
    | false =>
      case xTurn of
        true =>
          (case fPlaysX of
            true => play(placeToken(board,chooseMove(board,xTurn,fSLevel,true),
                        xTurn),fPlaysX,count+1,not(xTurn),fSLevel,oSLevel)
            | false => play(placeToken(board,
                        chooseMove(board,xTurn,oSLevel,false),xTurn),
                        fPlaysX,count+1,not(xTurn),fSLevel,oSLevel))
        | false =>
          (case fPlaysX of
            true => play(placeToken(board,chooseMove(board,xTurn,oSLevel,false),
                        xTurn),fPlaysX,count+1,not(xTurn),fSLevel,oSLevel)
            | false => play(placeToken(board,
                        chooseMove(board,xTurn,fSLevel,true),xTurn),
                        fPlaysX,count+1,not(xTurn),fSLevel,oSLevel))
```

```
fun main(input: mainInput ) : mainReturn =
  case input of mainInput(board,fPlaysX,xTurn,fSearchLevel,opponentSearchLevel)
=>
  case play(toMatrix(board),fPlaysX,0,xTurn,fSearchLevel,opponentSearchLevel) of
    gameResult(c,r) =>
      case fPlaysX of
        true =>
          (case r = 100000 of
            false => mainReturn(false,0)
            | true => mainReturn(true,c))
        | false =>
          (case r = ~100000 of
            false => mainReturn(false,0)
            | true => mainReturn(true,c))



%%

fun fromList [] = nill
  | fromList( X :: Xs ) = cons( X, fromList Xs )

fun fromList' [] = nill'
  | fromList'( Xs :: Xss ) = cons'( fromList Xs, fromList' Xss )

fun toList nill = []
  | toList( cons( X, Xs ) ) = X :: toList Xs

fun toList' nill' = []
  | toList'( cons'( Xs, Xss ) ) = toList Xs :: toList' Xss

fun countXs2([], xs) = xs
  | countXs2(h :: t, xs) = if h = 1 then countXs2(t, xs+1) else countXs2(t, xs)

fun countXs([], xs) = xs
  | countXs(h :: t, xs) = countXs(t, xs+countXs2(h,0))

fun countOs2([], os) = os
  | countOs2(h :: t, os) = if h = ~1 then countOs2(t, os+1) else countOs2(t, os)

fun countOs([], os) = os
  | countOs(h :: t, os) = countOs(t, os+countOs2(h,0))

fun xTurn(board) =
  let
    val xs = countXs(board,0)
    val os = countOs(board,0)
  in
    if xs > os then false else true
  end

fun makeInputList([]) = []
  | makeInputList(h :: t) = mainInput(fromList'(h),true,xTurn(h),1,2) ::
                            mainInput(fromList'(h),false,xTurn(h),1,2) ::
                            mainInput(fromList'(h),true,xTurn(h),2,2) ::
                            mainInput(fromList'(h),false,xTurn(h),2,2) ::
                            makeInputList(t)


val StartBoards =
[
```

```
[
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ]
]


,

[
[ 1,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ]
]


,

[
[ 1,  0,  0,  0,  0,  0,  0 ],
[~1,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ]
]


,

[
[ 1,  0,  0,  0,  0,  0,  0 ],
[ 0,~1,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ]
]


,

[
[ 1,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,~1,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ],
[ 0,  0,  0,  0,  0,  0,  0 ]
]



<cut>


,

[
```

```
[~1, 1, 1,~1,~1, 1, 1 ],
[ 1,~1,~1, 1,~1, 1,~1 ],
[ 1, 0, 1,~1, 0,~1,~1 ],
[ 1, 0, 1,~1, 0, 0, 0 ],
[ 0, 0, 0, 1, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ]
]

,

[
[ 1,~1,~1, 1, 0, 1,~1 ],
[ 0, 1, 1,~1, 0, 1, 1 ],
[ 0, 1, 1,~1, 0, 1, 0 ],
[ 0, 0, 0,~1, 0,~1, 0 ],
[ 0, 0, 0, 0, 0,~1, 0 ],
[ 0, 0, 0, 0, 0,~1, 0 ]
]

]

val Inputs = makeInputList(StartBoards)


val ValBoards =

[

[
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ]
]


,

[
[ 1, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ]
]


,

[
[ 1, 0, 0, 0, 0, 0, 0 ],
[~1, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ]
]


,

[
[ 1, 0, 0, 0, 0, 0, 0 ],
```

```
[ 0,~1, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ]
]


,

[
[ 1, 0, 0, 0, 0, 0, 0 ],
[ 0, 0,~1, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ]
]


,

<cut>

[
[~1, 0, 0, 1, 1,~1, 0 ],
[~1, 0, 0, 0, 0, 0, 0 ],
[ 1, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ]
]


,
[
[~1, 0, 0, 1,~1, 1, 0 ],
[ 0, 0, 0, 1, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ]
]


,
[
[~1, 0, 1, 1, 1,~1, 1 ],
[ 0, 0,~1, 0,~1, 0, 1 ],
[ 0, 0, 1, 0, 1, 0,~1 ],
[ 0, 0,~1, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0 ]
]


]


val Validation_inputs = makeInputList(ValBoards)


val Abstract_types = [ "matrix" ]
val Funs_to_use = [
```

```
  "sub",
  "false", "true",
  "<", "=", "+", "-", "~1", "0", "1", "2", "3", "4", "5", "6"
  ]

val Reject_funs = []

fun restore_transform D = D

structure Grade : GRADE =
struct

type grade = int
val zero = 0
val op+ = Int.+
val comparisons = [ Int.compare ]
val toString = Int.toString
val pack = Int.toString
val unpack = fn S => case Int.fromString S of SOME N => N

val toRealOpt = NONE

val post_process = fn X => X

end

fun output_eval_fun( _, Xss : mainInput, Yss : mainReturn)
  : output_eval_type * Grade.grade =
  case Yss of mainReturn(win,count) =>
    if win then (correct,count) else (wrong,count)


val Max_output_class_card = 0

val Max_time_limit = 30000000
val Time_limit_base = 2.0
```

B. The ADATE-synthesized function

This appendix shows the Connect4 board evaluation function synthesized by ADATE, in its original form, encoded in Standard ML.

```
fun f board =
(
 ( 0 + 1 ) +
  ((
   case sub( board, 1, 4 ) of
    V53FD =>
     ((((
      case sub( board, 3, 5 ) of
       V5417 =>
        ((
         case sub( board, 0, 4 ) of
          V53ED =>
           ((
            case sub( board, 2, 3 ) of
             VAD996 =>
              (((((((
               case sub( board, 4, 5 ) of
                V5401 =>
                 ((
                  case sub( board, 3, 4) of
                   V53EB =>
                    case ((((((
                     case sub( board, 1, 2 ) of
                      V540F =>
                      (((((((( 
                       case sub( board, 3, 3 ) of
                        V53FB =>
                        ((((((((((( 
                         case sub( board, 4, 3 ) of
                          V5419 =>
                          ((
                           case sub( board, 3, 2 ) of
                            V5413 =>
                            (((((((( 
                             case sub( board, 4, 2 ) of
                              V5405 =>
```

```
                    ((((((((((((((((
                     case sub( board, 1, 5 ) of
                      V540B =>
                       ((((((((((((
                        sub( board, 4, 1 ) +
                       V5413 ) +
                      V5405 ) +
                     V53FB ) +
                    V53ED ) +
                   V5405 ) +
                  V53FB ) +
                 V53ED ) +
                V540B ) +
               V53FB ) +
              V53ED ) +
             V540B ) +
            VAD996 )) +
            sub( board, 5, 2 )) +
          VAD996 ) +
         V53EB ) +
        V53EB ) +
       V5419 ) +
      V53EB ) +
      sub( board, 5, 0 )) +
      (
      case sub( board, 1, 6 ) of
       V3DC5 =>
       ( V3DC5 + V3DC5 )
      )) +
     V53FD ) +
    V53FD ) +
   V53FB ) +
   sub( board, 2, 6 )) +
V53ED ) +
(
 case sub( board, 3, 1 ) of
  V409D7 =>
   case ( V409D7 + V409D7 ) of
    V1ABA89 =>
    (
     ( V1ABA89 + V1ABA89 ) +
      V1ABA89
```

```
              ))) +
             V5405 ) +
            sub( board, 5, 3 )
           )) +
          V5417
         ) +
        V5417 ) +
       V5413 ) +
      V5419 ) +
     VAD996 ) +
    V5413 ) +
   V5419 )) +
  sub( board, 5, 4 )
 )) +
V5417 ) +
VAD996 ) +
VAD996 ) +
V53FB ) +
sub( board, 1, 1 ) ) +
V53ED ) +
V53FB ) +
V5417 ) +
(
 case sub( board, 2, 2 ) of
  V1B8B21 =>
   ( V1B8B21 + V1B8B21 )
   )) +
  V53FB ) +
 sub( board, 4, 4 )
)) +
sub( board, 5, 5 )
) +
sub( board, 0, 1 )
) +
V540F ) +
VAD996 ) +
V53EB ) +
V540F ) +
0
)) +
V53EB
) +
```

```
 V5401
 ) +
 VAD996
 ) +
 V53EB
 ) +
V5401 ) of
 V613CD =>
  ( V613CD + V613CD )
   ) +
   (
    sub(board, 5, 6) +
     V5417
      ))) +
      sub( board, 1, 2 )
      ) +
      VAD996 ) +
      V53ED ) +
      V5417 ) +
      VAD996 ) +
      V53ED )) +
      V5417 )) +
      sub( board, 4, 6 )
     )) +
    sub( board, 0, 3 )
   ) +
  V53FD ) +
 sub( board, 2, 1 )
)) +
(
case sub( board, 3, 6 ) of
 V86BDB => ( V86BDB + V86BDB )
)))
```

C. Reinforcement Learning system modules

This appendix presents source code for the main modules (Java classes) of the Reinforcement Learning System. These classes are, in sequence, Connect4.java, C4NNLearnAgent.java, C4NNPlayAgent.java, C4RandomPlayAgent.java, C4IBEFAgent.java, C4ADATEAgent.java and ANN_TD.java. Some of the less central java classes of the system have been excluded from this appendix, for reasons of clarity.

C.1. Connect4.java

```
//game logic class
public class Connect4 implements Environment {

	private Agent a1; //X
	private Agent a2; //O
	private int board[][];
	private boolean a1Turn;
	private int episodes;
	private RLSystem sys;

	public Connect4(Agent a1, Agent a2, RLSystem sys) {
		this.a1 = a1;
		this.a2 = a2;
		board = new int[7][6];
		a1Turn = true;
		episodes = 0;
		this.sys = sys;
	}

	//Prepares a new game of Connect4
	public void newGame() {
		for (int i=0; i<board.length; i++) {
			for (int j=0; j<board[i].length; j++) {
				board[i][j] = 0;
			}
		}
		a1Turn = true;
	}

	public void runEpisodes(int episodes) {
		this.episodes = episodes;
```

```
        go();
    }

    //plays out the correct number of games
    private void go() {
        while(episodes>0) {
            newGame();
            while(true) {
                double reward = eval(board);
                if (reward == 0.0) {
                    if (a1Turn) {
                        updateBoard(board, a1.makeYourMove(board,
                            reward), 1);
                    }
                    else {
                        updateBoard(board, a2.makeYourMove(board,
                            reward), -1);
                    }
                    a1Turn = !a1Turn;
                }
                else if (reward == 100.0) {
                    if (a1Turn) {
                        a1.gameFinished(-100.0);
                        a2.gameFinished(100.0);
                    }
                    else {
                        a1.gameFinished(100.0);
                        a2.gameFinished(-100.0);
                    }
                    break;
                }
                else if (reward == 999.0) {
                    a1.gameFinished(0.0);
                    a2.gameFinished(0.0);
                    break;
                }
            }
            episodes--;
        }
        sys.done();
    }
```

```
//places the correct disc in the correct board position
private void updateBoard(int board[][], int column, int player) {
      for (int i=0; i<board[column].length; i++) {
            if (board[column][i] == 0) {
                  board[column][i] = player;
                  break;
            }
      }
}

//checks the state of the game
private double eval(int b[][]) {
      //check for horizontal wins
      for (int i=0; i<b[0].length; i++) {
            if ((b[0][i]==b[1][i] && b[1][i]==b[2][i] && b[2][i]==b[3][i]
                        && b[3][i]!=0) ||
                (b[1][i]==b[2][i] && b[2][i]==b[3][i] && b[3][i]==b[4][i]
                        && b[4][i]!=0) ||
                (b[2][i]==b[3][i] && b[3][i]==b[4][i] && b[4][i]==b[5][i]
                        && b[5][i]!=0) ||
                (b[3][i]==b[4][i] && b[4][i]==b[5][i] && b[5][i]==b[6][i]
                        && b[6][i]!=0))
            {
                  return 100.0;
            }
      }

      //check for vertical wins
      for (int i=0; i<b.length; i++) {
            if ((b[i][0]==b[i][1] && b[i][1]==b[i][2]&& b[i][2]==b[i][3]
                  && b[i][3]!=0) ||
                (b[i][1]==b[i][2] && b[i][2]==b[i][3]&& b[i][3]==b[i][4]
                  && b[i][4]!=0) ||
                (b[i][2]==b[i][3] && b[i][3]==b[i][4]&& b[i][4]==b[i][5]
                  && b[i][5]!=0))
            {
                  return 100.0;
            }
      }

      //check for diagonal wins
      if ((b[0][3]==b[1][2] && b[1][2]==b[2][1] && b[2][1]==b[3][0]
                  && b[3][0]!= 0) ||
          (b[0][4]==b[1][3] && b[1][3]==b[2][2] && b[2][2]==b[3][1]
                  && b[3][1]!= 0) ||
```

```
(b[1][3]==b[2][2] && b[2][2]==b[3][1] && b[3][1]==b[4][0]
        && b[4][0]!= 0) ||
(b[0][5]==b[1][4] && b[1][4]==b[2][3] && b[2][3]==b[3][2]
        && b[3][2]!= 0) ||
(b[1][4]==b[2][3] && b[2][3]==b[3][2] && b[3][2]==b[4][1]
        && b[4][1]!= 0) ||
(b[2][3]==b[3][2] && b[3][2]==b[4][1] && b[4][1]==b[5][0]
        && b[5][0]!= 0) ||
(b[1][5]==b[2][4] && b[2][4]==b[3][3] && b[3][3]==b[4][2]
        && b[4][2]!= 0) ||
(b[2][4]==b[3][3] && b[3][3]==b[4][2] && b[4][2]==b[5][1]
        && b[5][1]!= 0) ||
(b[3][3]==b[4][2] && b[4][2]==b[5][1] && b[5][1]==b[6][0]
        && b[6][0]!= 0) ||
(b[3][4]==b[4][3] && b[4][3]==b[5][2] && b[5][2]==b[6][1]
        && b[6][1]!= 0) ||
(b[3][5]==b[4][4] && b[4][4]==b[5][3] && b[5][3]==b[6][2]
        && b[6][2]!= 0) ||
(b[0][2]==b[1][3] && b[1][3]==b[2][4] && b[2][4]==b[3][5]
        && b[3][5]!= 0) ||
(b[0][1]==b[1][2] && b[1][2]==b[2][3] && b[2][3]==b[3][4]
        && b[3][4]!= 0) ||
(b[1][2]==b[2][3] && b[2][3]==b[3][4] && b[3][4]==b[4][5]
        && b[4][5]!= 0) ||
(b[0][0]==b[1][1] && b[1][1]==b[2][2] && b[2][2]==b[3][3]
        && b[3][3]!= 0) ||
(b[1][1]==b[2][2] && b[2][2]==b[3][3] && b[3][3]==b[4][4]
        && b[4][4]!= 0) ||
(b[2][2]==b[3][3] && b[3][3]==b[4][4] && b[4][4]==b[5][5]
        && b[5][5]!= 0) ||
(b[1][0]==b[2][1] && b[2][1]==b[3][2] && b[3][2]==b[4][3]
        && b[4][3]!= 0) ||
(b[2][1]==b[3][2] && b[3][2]==b[4][3] && b[4][3]==b[5][4]
        && b[5][4]!= 0) ||
(b[3][2]==b[4][3] && b[4][3]==b[5][4] && b[5][4]==b[6][5]
        && b[6][5]!= 0) ||
(b[2][0]==b[3][1] && b[3][1]==b[4][2] && b[4][2]==b[5][3]
        && b[5][3]!= 0) ||
(b[3][1]==b[4][2] && b[4][2]==b[5][3] && b[5][3]==b[6][4]
        && b[6][4]!= 0) ||
(b[3][0]==b[4][1] && b[4][1]==b[5][2] && b[5][2]==b[6][3]
        && b[6][3]!= 0))
```

```
            {
                  return 100.0;
            }

            else {
                  for (int i=0; i<b.length; i++) {
                        for (int j=0; j<b[i].length; j++) {
                              if (b[i][j] == 0) {
                                    //Game is not over
                                    return 0.0;
                              }
                        }
                  }
                  //Draw
                  return 999.0;
            }
      }
}
```

C.2. C4NNLearnAgent.java

```
//The Reinforcement Learning agent
public class C4NNLearnAgent implements Agent {

      private double l_rate = 0.2;
      private double d_rate = 1.0;
      private double eps = 0.5;
      private double prevBrdAndMove[] = null;
      private double curBrdAndMove[] = null;
      private Environment env;
      private ANN_TD nn;
      private String id;

      public C4NNLearnAgent(String id, ANN_TD nn) {
            this.env = null;
            this.id = id;
            this.nn = nn;
            nn.setLearningRate(l_rate);
      }

      public void setEnvironment(Environment env) {
```

```
        this.env = env;
}

//sets the exploration rate
public void setEpsilon(double e) {
        eps = e;
}

//sets the learning rate
public void setLearningRate(double l) {
        this.l = l;
        nn.setLearningRate(l);
}

//sets the lambda parameter
public void setLambda(double l) {
        nn.setLambda(l);
}

//resets the neural network eligibility traces
public void resetEligibilityTraces() {
        nn.resetE();
}

//returns the appropriate move according to the neural network
public int makeYourMove(int board[][], double reward) {
        double maxValue = Double.NEGATIVE_INFINITY;
        int bestMove = -1;
        for (int i=0; i<board.length; i++) {
                if (board[i][5] == 0) {
                        double[] state = makeState(board, i);
                        double v = (nn.apply(state)[0]-0.5)*250;
                        if (v >= maxValue) {
                                if (v == maxValue) {
                                        if (Math.random() >= 0.5) {
                                                maxValue = v;
                                                bestMove = i;
                                        }
                                }
                                else {
                                        maxValue = v;
                                        bestMove = i;
```

```
                        }
                    }
                }
            }

            curBrdAndMove = makeState(board, bestMove);
            if (prevBrdAndMove != null) {
                double prevValue = reward+d_rate*maxValue;

                nn.train(prevBrdAndMove, new double[]{(prevValue+125.0)/
                    250.0});
            }

            if (Math.random()>=eps) {
                prevBrdAndMove = curBrdAndMove;
                return bestMove;
            }
            else {
                String free = "";
                for (int i=0; i<board.length; i++) {
                    free += (board[i][5]==0?""+i:"");
                }
                double odds = 1.0/free.length();
                int index = (int)(((double)Math.random())/odds);
                prevBrdAndMove = makeState(board, Integer.parseInt
                    (free.substring(index,index+1)));
                return Integer.parseInt(free.substring(index,index+1));
            }

    }

    //called when the game is finished
    public void gameFinished(double reward) {
            double curValue = 0.0;
            double prevValue = reward+d_rate*curValue;

            nn.train(prevBrdAndMove, new double[]{(prevValue+125.0)/250.0});

            prevBrdAndMove = null;
    }


    //converts the current board into an input array for the neural network
```

```
private double[] makeState(int b[][], int column) {
	double state[] = new double[126];
	int xs = 0;
	int os = 0;

	for (int i=0; i<b.length; i++) {
		for (int j=0; j<b[i].length; j++) {
			if (b[i][j]==1) {
				state[(i*18)+(j*3)] = 1.0;
				state[(i*18)+(j*3)+1] = 0.0;
				state[(i*18)+(j*3)+2] = 0.0;
				xs++;
			}
			else if (b[i][j]==-1) {
				state[(i*18)+(j*3)] = 0.0;
				state[(i*18)+(j*3)+1] = 1.0;
				state[(i*18)+(j*3)+2] = 0.0;
				os++;
			}
			else {
				state[(i*18)+(j*3)] = 0.0;
				state[(i*18)+(j*3)+1] = 0.0;
				state[(i*18)+(j*3)+2] = 1.0;
			}
		}

	}

	for (int i=0; i<b[column].length; i++) {
		if (b[column][i] == 0.0) {
			state[(column*18)+(i*3)+2] = 0.0;
			if (xs>os) {
				state[(column*18)+(i*3)+1] = 1.0;
			}
			else if (xs==os) {
				state[(column*18)+(i*3)] = 1.0;
			}
			break;
		}
	}

	return state;
```

```
}
```

C.3. C4NNPlayAgent.java

```
/* agent that plays according to a minimax algorithm using a neural network at
the cutoff points in the game tree, without altering the neural network */
public class C4NNPlayAgent implements Agent {

	private Environment env;
	private String id;
	private boolean x;
	private int cutoff;
	private ANN_TD xnn;
	private ANN_TD onn;

	public C4NNPlayAgent(String id, boolean x, int cutoff, ANN_TD xnn, ANN_TD
		onn) {
		this.env = null;
		this.id = id;
		this.x = x;
		this.cutoff = cutoff;
		this.xnn = xnn;
		this.onn = onn;
	}



	public void setEnvironment(Environment env) {
		this.env = env;
	}

	/* returns a move chosen according to the combination of the minimax
	algorithm and the neural network */
	public int makeYourMove(int board[][], double reward) {
		return chooseMove(board);
	}

	private int chooseMove(int b[][]) {
		return minimax_ab(b, x, 0, -999999, 999999);
	}

	//empty function, inherited from Agent interface
```

```
public void resetEligibilityTraces() {}


//minimax function with aplha-beta pruning
private int minimax_ab(int brd[][], boolean x, int depth, int a, int b) {
      if (gameOver(brd)) {
            return getEndResult(brd);
      }
      if (depth == cutoff) {
            if (x) {
                  return (int)(((xnn.apply(makeState(brd))[0])-0.5)*2500);
            }
            else {
                  return (int)(((onn.apply(makeState(brd))[0])-0.5)*2500);
            }
      }

      int best_move = 999999;
      int best_score = (x?-999999:999999);
      int n_Alpha = -999999;
      int n_Beta = 999999;

      int move_order[] = randomize();

      for (int i=0; i<brd.length; i++) {
            if (brd[move_order[i]][5] == 0) {
                  int j = findFirstVacant(brd[move_order[i]]);
                  brd[move_order[i]][j] = (x?1:-1);
                  if (x) {
                        int score = minimax_ab(brd, !x, depth+1, Math.max
                              (n_Alpha,a),b);
                        if (score > n_Alpha) {
                              n_Alpha = score;
                              best_move = move_order[i];
                              if (n_Alpha>=b) {
                                    brd[move_order[i]][j] = 0;
                                    break;
                              }
                        }
                  }
                  else if (!x) {
                        int score = minimax_ab(brd, !x, depth+1, a,
                              Math.min(b,n_Beta));
```

```
                        if (score < n_Beta) {
                              n_Beta = score;
                              best_move = move_order[i];
                              if (a>=n_Beta) {
                                    brd[move_order[i]][j] = 0;
                                    break;
                              }
                        }
                  }
                  brd[move_order[i]][j] = 0;
            }
      }

      if (depth==0) {
            return best_move;
      }
      else {
            return (x?n_Alpha:n_Beta);
      }

}

//randomizes order of moves considered
private int[] randomize() {
      int mo[] = new int[7];

      for (int i=0; i<mo.length; i++) {
            mo[i] = i;
      }

      for (int i=0; i<mo.length; i++) {
            int n = (int)(Math.random()*mo.length);
            int temp = mo[i];
            mo[i] = mo[n];
            mo[n] = temp;
      }

      return mo;
}

//finds first vacant row of a column
private int findFirstVacant(int column[]) {
```

```
      for (int i=0; i<column.length; i++) {
            if (column[i]==0) {
                  return i;
            }
      }
      return -1;
}

//converts current board into an input array for the neural network
private double[] makeState(int b[][]) {
      double state[] = new double[126];
      int xs = 0;
      int os = 0;

      for (int i=0; i<b.length; i++) {
            for (int j=0; j<b[i].length; j++) {
                  if (b[i][j]==1) {
                        state[(i*18)+(j*3)] = 1.0;
                        state[(i*18)+(j*3)+1] = 0.0;
                        state[(i*18)+(j*3)+2] = 0.0;
                        xs++;
                  }
                  else if (b[i][j]==-1) {
                        state[(i*18)+(j*3)] = 0.0;
                        state[(i*18)+(j*3)+1] = 1.0;
                        state[(i*18)+(j*3)+2] = 0.0;
                        os++;
                  }
                  else {
                        state[(i*18)+(j*3)] = 0.0;
                        state[(i*18)+(j*3)+1] = 0.0;
                        state[(i*18)+(j*3)+2] = 1.0;
                  }
            }

      }

      return state;
}

//checks if the game is over
public boolean gameOver(int b[][]) {
```

```
//check for horizontal wins
for (int i=0; i<b[0].length; i++) {
      if ((b[0][i]==b[1][i] && b[1][i]==b[2][i] && b[2][i]==b[3][i]
                  && b[3][i]!=0) ||
           (b[1][i]==b[2][i] && b[2][i]==b[3][i] && b[3][i]==b[4][i]
                  && b[4][i]!=0) ||
           (b[2][i]==b[3][i] && b[3][i]==b[4][i] && b[4][i]==b[5][i]
                  && b[5][i]!=0) ||
           (b[3][i]==b[4][i] && b[4][i]==b[5][i] && b[5][i]==b[6][i]
                  && b[6][i]!=0))
      {
            return true;
      }
}

//check for vertical wins
for (int i=0; i<b.length; i++) {
      if ((b[i][0]==b[i][1] && b[i][1]==b[i][2]&& b[i][2]==b[i][3]
            && b[i][3]!=0) ||
           (b[i][1]==b[i][2] && b[i][2]==b[i][3]&& b[i][3]==b[i][4]
                  && b[i][4]!=0) ||
           (b[i][2]==b[i][3] && b[i][3]==b[i][4]&& b[i][4]==b[i][5]
                  && b[i][5]!=0))
      {
            return true;
      }
}

//check for diagonal wins
if ((b[0][3]==b[1][2] && b[1][2]==b[2][1] && b[2][1]==b[3][0]
      && b[3][0]!= 0) ||
    (b[0][4]==b[1][3] && b[1][3]==b[2][2] && b[2][2]==b[3][1]
      && b[3][1]!= 0) ||
    (b[1][3]==b[2][2] && b[2][2]==b[3][1] && b[3][1]==b[4][0]
      && b[4][0]!= 0) ||
    (b[0][5]==b[1][4] && b[1][4]==b[2][3] && b[2][3]==b[3][2]
      && b[3][2]!= 0) ||
    (b[1][4]==b[2][3] && b[2][3]==b[3][2] && b[3][2]==b[4][1]
      && b[4][1]!= 0) ||
    (b[2][3]==b[3][2] && b[3][2]==b[4][1] && b[4][1]==b[5][0]
      && b[5][0]!= 0) ||
    (b[1][5]==b[2][4] && b[2][4]==b[3][3] && b[3][3]==b[4][2]
      && b[4][2]!= 0) ||
    (b[2][4]==b[3][3] && b[3][3]==b[4][2] && b[4][2]==b[5][1]
      && b[5][1]!= 0) ||
```

```
    (b[3][3]==b[4][2] && b[4][2]==b[5][1] && b[5][1]==b[6][0]
      && b[6][0]!= 0) ||
    (b[3][4]==b[4][3] && b[4][3]==b[5][2] && b[5][2]==b[6][1]
      && b[6][1]!= 0) ||
    (b[3][5]==b[4][4] && b[4][4]==b[5][3] && b[5][3]==b[6][2]
      && b[6][2]!= 0) ||
    (b[0][2]==b[1][3] && b[1][3]==b[2][4] && b[2][4]==b[3][5]
      && b[3][5]!= 0) ||
    (b[0][1]==b[1][2] && b[1][2]==b[2][3] && b[2][3]==b[3][4]
      && b[3][4]!= 0) ||
    (b[1][2]==b[2][3] && b[2][3]==b[3][4] && b[3][4]==b[4][5]
      && b[4][5]!= 0) ||
    (b[0][0]==b[1][1] && b[1][1]==b[2][2] && b[2][2]==b[3][3]
      && b[3][3]!= 0) ||
    (b[1][1]==b[2][2] && b[2][2]==b[3][3] && b[3][3]==b[4][4]
      && b[4][4]!= 0) ||
    (b[2][2]==b[3][3] && b[3][3]==b[4][4] && b[4][4]==b[5][5]
      && b[5][5]!= 0) ||
    (b[1][0]==b[2][1] && b[2][1]==b[3][2] && b[3][2]==b[4][3]
      && b[4][3]!= 0) ||
    (b[2][1]==b[3][2] && b[3][2]==b[4][3] && b[4][3]==b[5][4]
      && b[5][4]!= 0) ||
    (b[3][2]==b[4][3] && b[4][3]==b[5][4] && b[5][4]==b[6][5]
      && b[6][5]!= 0) ||
    (b[2][0]==b[3][1] && b[3][1]==b[4][2] && b[4][2]==b[5][3]
      && b[5][3]!= 0) ||
    (b[3][1]==b[4][2] && b[4][2]==b[5][3] && b[5][3]==b[6][4]
      && b[6][4]!= 0) ||
    (b[3][0]==b[4][1] && b[4][1]==b[5][2] && b[5][2]==b[6][3]
      && b[6][3]!= 0))
{
      return true;
}

else {
      for (int i=0; i<b.length; i++) {
            for (int j=0; j<b[i].length; j++) {
                  if (b[i][j] == 0) {
                        //Game is not over
                        return false;
                  }
            }
```

```
            }
            //Draw
            return true;
      }
}

//checks the end result of the game
public int getEndResult(int b[][]) {
      //check for horizontal wins
      for (int i=0; i<b[0].length; i++) {
            if ((b[0][i]==b[1][i] && b[1][i]==b[2][i] && b[2][i]==b[3][i]
                  && b[3][i]!=0) ||
                (b[1][i]==b[2][i] && b[2][i]==b[3][i] && b[3][i]==b[4][i]
                  && b[4][i]!=0) ||
                (b[2][i]==b[3][i] && b[3][i]==b[4][i] && b[4][i]==b[5][i]
                  && b[5][i]!=0) ||
                (b[3][i]==b[4][i] && b[4][i]==b[5][i] && b[5][i]==b[6][i]
                  && b[6][i]!=0))
            {
                  return whoWon(b);
            }
      }

      //check for vertical wins
      for (int i=0; i<b.length; i++) {
            if ((b[i][0]==b[i][1] && b[i][1]==b[i][2]&& b[i][2]==b[i][3]
                  && b[i][3]!=0) ||
                (b[i][1]==b[i][2] && b[i][2]==b[i][3]&& b[i][3]==b[i][4]
                  && b[i][4]!=0) ||
                (b[i][2]==b[i][3] && b[i][3]==b[i][4]&& b[i][4]==b[i][5]
                  && b[i][5]!=0))
            {
                  return whoWon(b);
            }
      }

      //check for diagonal wins
      if ((b[0][3]==b[1][2] && b[1][2]==b[2][1] && b[2][1]==b[3][0]
            && b[3][0]!= 0) ||
          (b[0][4]==b[1][3] && b[1][3]==b[2][2] && b[2][2]==b[3][1]
            && b[3][1]!= 0) ||
          (b[1][3]==b[2][2] && b[2][2]==b[3][1] && b[3][1]==b[4][0]
            && b[4][0]!= 0) ||
          (b[0][5]==b[1][4] && b[1][4]==b[2][3] && b[2][3]==b[3][2]
            && b[3][2]!= 0) ||
```

```
    (b[1][4]==b[2][3] && b[2][3]==b[3][2] && b[3][2]==b[4][1]
      && b[4][1]!= 0) ||
    (b[2][3]==b[3][2] && b[3][2]==b[4][1] && b[4][1]==b[5][0]
      && b[5][0]!= 0) ||
    (b[1][5]==b[2][4] && b[2][4]==b[3][3] && b[3][3]==b[4][2]
      && b[4][2]!= 0) ||
    (b[2][4]==b[3][3] && b[3][3]==b[4][2] && b[4][2]==b[5][1]
      && b[5][1]!= 0) ||
    (b[3][3]==b[4][2] && b[4][2]==b[5][1] && b[5][1]==b[6][0]
      && b[6][0]!= 0) ||
    (b[3][4]==b[4][3] && b[4][3]==b[5][2] && b[5][2]==b[6][1]
      && b[6][1]!= 0) ||
    (b[3][5]==b[4][4] && b[4][4]==b[5][3] && b[5][3]==b[6][2]
      && b[6][2]!= 0) ||
    (b[0][2]==b[1][3] && b[1][3]==b[2][4] && b[2][4]==b[3][5]
      && b[3][5]!= 0) ||
    (b[0][1]==b[1][2] && b[1][2]==b[2][3] && b[2][3]==b[3][4]
      && b[3][4]!= 0) ||
    (b[1][2]==b[2][3] && b[2][3]==b[3][4] && b[3][4]==b[4][5]
      && b[4][5]!= 0) ||
    (b[0][0]==b[1][1] && b[1][1]==b[2][2] && b[2][2]==b[3][3]
      && b[3][3]!= 0) ||
    (b[1][1]==b[2][2] && b[2][2]==b[3][3] && b[3][3]==b[4][4]
      && b[4][4]!= 0) ||
    (b[2][2]==b[3][3] && b[3][3]==b[4][4] && b[4][4]==b[5][5]
      && b[5][5]!= 0) ||
    (b[1][0]==b[2][1] && b[2][1]==b[3][2] && b[3][2]==b[4][3]
      && b[4][3]!= 0) ||
    (b[2][1]==b[3][2] && b[3][2]==b[4][3] && b[4][3]==b[5][4]
      && b[5][4]!= 0) ||
    (b[3][2]==b[4][3] && b[4][3]==b[5][4] && b[5][4]==b[6][5]
      && b[6][5]!= 0) ||
    (b[2][0]==b[3][1] && b[3][1]==b[4][2] && b[4][2]==b[5][3]
      && b[5][3]!= 0) ||
    (b[3][1]==b[4][2] && b[4][2]==b[5][3] && b[5][3]==b[6][4]
      && b[6][4]!= 0) ||
    (b[3][0]==b[4][1] && b[4][1]==b[5][2] && b[5][2]==b[6][3]
      && b[6][3]!= 0))
{
      return whoWon(b);
}
```

```
            return 0;
      }
      //checks who won the game
      private int whoWon(int b[][]) {
            int xs = 0;
            int os = 0;

            for (int i=0; i<b.length; i++) {
                  for (int j=0; j<b[i].length; j++) {
                        if (b[i][j]==1) {
                              xs++;
                        }
                        else if (b[i][j]==-1) {
                              os++;
                        }
                  }
            }

            if (xs>os) {
                  return 100000;
            }
            else {
                  return -100000;
            }
      }

      //empty function inherited from Agent interface
      public void setEpsilon(double e) {}

      //empty function inherited from Agent interface
      public void setLearningRate(double l) {}

      //empty function inherited from Agent interface
      public void gameFinished(double reward) {}

      //empty function inherited from Agent interface
      public void setLambda(double l) {}


}
```

C.4. C4RandomPlayAgent.java

```
//agent making moves at random
public class C4RandomPlayAgent implements Agent {

	private String id;

	public C4RandomPlayAgent(String id) {
		this.id = id;
	}

	//empty function inherited from Agent interface
	public void setEnvironment(Environment env){}

	//empty function inherited from Agent interface
	public void gameFinished(double reward){}

	//empty function inherited from Agent interface
	public void setEpsilon(double e){}

	//empty function inherited from Agent interface
	public void setLearningRate(double l){}

	//empty function inherited from Agent interface
	public void resetEligibilityTraces(){}

	//empty function inherited from Agent interface
	public void setLambda(double l) {}

	//returns move chosen at random
	public int makeYourMove(int board[][], double reward){
		String moves = "";

		for (int i=0; i<board.length; i++) {
			if (board[i][5]==0) {
				moves += i;
			}
		}

		double odds = 1.0/moves.length();
		int index = (int)(((double)Math.random())/odds);
```

```
            return Integer.parseInt(moves.substring(index,index+1));
      }

}
```

C.5. C4IBEFAgent.java

```
/* agent making moves according to a minimax algorithm using the IBEF-function
at the cutoff points in the game tree */
public class C4IBEFAgent implements Agent {

      private Environment env;
      private String id;
      private boolean x;
      private int cutoff;

      public C4IBEFAgent(String id, boolean x, int cutoff) {
            this.env = null;
            this.id = id;
            this.x = x;
            this.cutoff = cutoff;
      }

      public C4IBEFAgent(String id, boolean x) {
            this(id, x, 3);
      }

      public void setEnvironment(Environment env) {
            this.env = env;
      }

      /* returns a move chosen according to the combination of the minimax
      algorithm and the IBEF-function */
      public int makeYourMove(int board[][], double reward) {
            return chooseMove(board);
      }

      private int chooseMove(int b[][]) {
            return minimax_ab(b, x, 0, -999999, 999999);
      }
```

```
//empty function inherited from Agent interface
public void resetEligibilityTraces() {}

//minimax algorithm with alpha-beta pruning
private int minimax_ab(int brd[][], boolean x, int depth, int a, int b) {
      if (gameOver(brd)) {
                  return getEndResult(brd);
      }
      if (depth == cutoff) {
            return ibefEval(brd);
      }

      int best_move = 999999;
      int best_score = (x?-999999:999999);
      int n_Alpha = -999999;
      int n_Beta = 999999;

      int move_order[] = randomize();

      for (int i=0; i<brd.length; i++) {
            if (brd[move_order[i]][5] == 0) {
                  int j = findFirstVacant(brd[move_order[i]]);
                  brd[move_order[i]][j] = (x?1:-1);
                  if (x) {
                        int score = minimax_ab(brd, !x, depth+1, Math.max
                              (n_Alpha,a),b);
                        if (score > n_Alpha) {
                              n_Alpha = score;
                              best_move = move_order[i];
                              if (n_Alpha>=b) {
                                    brd[move_order[i]][j] = 0;
                                    break;
                              }
                        }
                  }
                  else if (!x) {
                        int score = minimax_ab(brd, !x, depth+1, a,
                              Math.min(b,n_Beta));
                        if (score < n_Beta) {
                              n_Beta = score;
                              best_move = move_order[i];
                              if (a>=n_Beta) {
                                    brd[move_order[i]][j] = 0;
```

```
                                    break;
                              }
                        }
                  }
                  brd[move_order[i]][j] = 0;
            }
      }

      if (depth==0) {
            return best_move;
      }
      else {
            return (x?n_Alpha:n_Beta);
      }

}

//randomizes order of moves considered
private int[] randomize() {
      int mo[] = new int[7];

      for (int i=0; i<mo.length; i++) {
            mo[i] = i;
      }

      for (int i=0; i<mo.length; i++) {
            int n = (int)(Math.random()*mo.length);
            int temp = mo[i];
            mo[i] = mo[n];
            mo[n] = temp;
      }

      return mo;
}

//finds first vacant row of a column on the board
private int findFirstVacant(int column[]) {
      for (int i=0; i<column.length; i++) {
            if (column[i]==0) {
                  return i;
            }
      }
```

```
      return -1;
}

//checks if the game is over
public boolean gameOver(int b[][]) {
      //check for horizontal wins
      for (int i=0; i<b[0].length; i++) {
            if ((b[0][i]==b[1][i] && b[1][i]==b[2][i] && b[2][i]==b[3][i]
                        && b[3][i]!=0) ||
                (b[1][i]==b[2][i] && b[2][i]==b[3][i] && b[3][i]==b[4][i]
                        && b[4][i]!=0) ||
                (b[2][i]==b[3][i] && b[3][i]==b[4][i] && b[4][i]==b[5][i]
                        && b[5][i]!=0) ||
                (b[3][i]==b[4][i] && b[4][i]==b[5][i] && b[5][i]==b[6][i]
                        && b[6][i]!=0))
            {
                  return true;
            }
      }

      //check for vertical wins
      for (int i=0; i<b.length; i++) {
            if ((b[i][0]==b[i][1] && b[i][1]==b[i][2]&& b[i][2]==b[i][3]
                  && b[i][3]!=0) ||
                (b[i][1]==b[i][2] && b[i][2]==b[i][3]&& b[i][3]==b[i][4]
                        && b[i][4]!=0) ||
                (b[i][2]==b[i][3] && b[i][3]==b[i][4]&& b[i][4]==b[i][5]
                        && b[i][5]!=0))
            {
                  return true;
            }
      }

      //check for diagonal wins
      if ((b[0][3]==b[1][2] && b[1][2]==b[2][1] && b[2][1]==b[3][0]
            && b[3][0]!= 0) ||
           (b[0][4]==b[1][3] && b[1][3]==b[2][2] && b[2][2]==b[3][1]
            && b[3][1]!= 0) ||
           (b[1][3]==b[2][2] && b[2][2]==b[3][1] && b[3][1]==b[4][0]
            && b[4][0]!= 0) ||
           (b[0][5]==b[1][4] && b[1][4]==b[2][3] && b[2][3]==b[3][2]
            && b[3][2]!= 0) ||
           (b[1][4]==b[2][3] && b[2][3]==b[3][2] && b[3][2]==b[4][1]
            && b[4][1]!= 0) ||
           (b[2][3]==b[3][2] && b[3][2]==b[4][1] && b[4][1]==b[5][0]
```

```
      && b[5][0]!= 0) ||
    (b[1][5]==b[2][4] && b[2][4]==b[3][3] && b[3][3]==b[4][2]
      && b[4][2]!= 0) ||
    (b[2][4]==b[3][3] && b[3][3]==b[4][2] && b[4][2]==b[5][1]
      && b[5][1]!= 0) ||
    (b[3][3]==b[4][2] && b[4][2]==b[5][1] && b[5][1]==b[6][0]
      && b[6][0]!= 0) ||
    (b[3][4]==b[4][3] && b[4][3]==b[5][2] && b[5][2]==b[6][1]
      && b[6][1]!= 0) ||
    (b[3][5]==b[4][4] && b[4][4]==b[5][3] && b[5][3]==b[6][2]
      && b[6][2]!= 0) ||
    (b[0][2]==b[1][3] && b[1][3]==b[2][4] && b[2][4]==b[3][5]
      && b[3][5]!= 0) ||
    (b[0][1]==b[1][2] && b[1][2]==b[2][3] && b[2][3]==b[3][4]
      && b[3][4]!= 0) ||
    (b[1][2]==b[2][3] && b[2][3]==b[3][4] && b[3][4]==b[4][5]
      && b[4][5]!= 0) ||
    (b[0][0]==b[1][1] && b[1][1]==b[2][2] && b[2][2]==b[3][3]
      && b[3][3]!= 0) ||
    (b[1][1]==b[2][2] && b[2][2]==b[3][3] && b[3][3]==b[4][4]
      && b[4][4]!= 0) ||
    (b[2][2]==b[3][3] && b[3][3]==b[4][4] && b[4][4]==b[5][5]
      && b[5][5]!= 0) ||
    (b[1][0]==b[2][1] && b[2][1]==b[3][2] && b[3][2]==b[4][3]
      && b[4][3]!= 0) ||
    (b[2][1]==b[3][2] && b[3][2]==b[4][3] && b[4][3]==b[5][4]
      && b[5][4]!= 0) ||
    (b[3][2]==b[4][3] && b[4][3]==b[5][4] && b[5][4]==b[6][5]
      && b[6][5]!= 0) ||
    (b[2][0]==b[3][1] && b[3][1]==b[4][2] && b[4][2]==b[5][3]
      && b[5][3]!= 0) ||
    (b[3][1]==b[4][2] && b[4][2]==b[5][3] && b[5][3]==b[6][4]
      && b[6][4]!= 0) ||
    (b[3][0]==b[4][1] && b[4][1]==b[5][2] && b[5][2]==b[6][3]
      && b[6][3]!= 0))
{
      return true;
}

else {
      for (int i=0; i<b.length; i++) {
            for (int j=0; j<b[i].length; j++) {
```

```
                        if (b[i][j] == 0) {
                              //Game is not over
                              return false;
                        }
                  }
            }
            //Draw
            return true;
      }
}

//checks the end result of the game
public int getEndResult(int b[][]) {
      //check for horizontal wins
      for (int i=0; i<b[0].length; i++) {
            if ((b[0][i]==b[1][i] && b[1][i]==b[2][i] && b[2][i]==b[3][i]
                  && b[3][i]!=0) ||
                (b[1][i]==b[2][i] && b[2][i]==b[3][i] && b[3][i]==b[4][i]
                  && b[4][i]!=0) ||
                (b[2][i]==b[3][i] && b[3][i]==b[4][i] && b[4][i]==b[5][i]
                  && b[5][i]!=0) ||
                (b[3][i]==b[4][i] && b[4][i]==b[5][i] && b[5][i]==b[6][i]
                  && b[6][i]!=0))
            {
                  return whoWon(b);
            }
      }

      //check for vertical wins
      for (int i=0; i<b.length; i++) {
            if ((b[i][0]==b[i][1] && b[i][1]==b[i][2]&& b[i][2]==b[i][3]
                  && b[i][3]!=0) ||
                (b[i][1]==b[i][2] && b[i][2]==b[i][3]&& b[i][3]==b[i][4]
                  && b[i][4]!=0) ||
                (b[i][2]==b[i][3] && b[i][3]==b[i][4]&& b[i][4]==b[i][5]
                  && b[i][5]!=0))
            {
                  return whoWon(b);
            }
      }

      //check for diagonal wins
      if ((b[0][3]==b[1][2] && b[1][2]==b[2][1] && b[2][1]==b[3][0]
            && b[3][0]!= 0) ||
          (b[0][4]==b[1][3] && b[1][3]==b[2][2] && b[2][2]==b[3][1]
```

```
  && b[3][1]!= 0) ||
(b[1][3]==b[2][2] && b[2][2]==b[3][1] && b[3][1]==b[4][0]
  && b[4][0]!= 0) ||
(b[0][5]==b[1][4] && b[1][4]==b[2][3] && b[2][3]==b[3][2]
  && b[3][2]!= 0) ||
(b[1][4]==b[2][3] && b[2][3]==b[3][2] && b[3][2]==b[4][1]
  && b[4][1]!= 0) ||
(b[2][3]==b[3][2] && b[3][2]==b[4][1] && b[4][1]==b[5][0]
  && b[5][0]!= 0) ||
(b[1][5]==b[2][4] && b[2][4]==b[3][3] && b[3][3]==b[4][2]
  && b[4][2]!= 0) ||
(b[2][4]==b[3][3] && b[3][3]==b[4][2] && b[4][2]==b[5][1]
  && b[5][1]!= 0) ||
(b[3][3]==b[4][2] && b[4][2]==b[5][1] && b[5][1]==b[6][0]
  && b[6][0]!= 0) ||
(b[3][4]==b[4][3] && b[4][3]==b[5][2] && b[5][2]==b[6][1]
  && b[6][1]!= 0) ||
(b[3][5]==b[4][4] && b[4][4]==b[5][3] && b[5][3]==b[6][2]
  && b[6][2]!= 0) ||
(b[0][2]==b[1][3] && b[1][3]==b[2][4] && b[2][4]==b[3][5]
  && b[3][5]!= 0) ||
(b[0][1]==b[1][2] && b[1][2]==b[2][3] && b[2][3]==b[3][4]
  && b[3][4]!= 0) ||
(b[1][2]==b[2][3] && b[2][3]==b[3][4] && b[3][4]==b[4][5]
  && b[4][5]!= 0) ||
(b[0][0]==b[1][1] && b[1][1]==b[2][2] && b[2][2]==b[3][3]
  && b[3][3]!= 0) ||
(b[1][1]==b[2][2] && b[2][2]==b[3][3] && b[3][3]==b[4][4]
  && b[4][4]!= 0) ||
(b[2][2]==b[3][3] && b[3][3]==b[4][4] && b[4][4]==b[5][5]
  && b[5][5]!= 0) ||
(b[1][0]==b[2][1] && b[2][1]==b[3][2] && b[3][2]==b[4][3]
  && b[4][3]!= 0) ||
(b[2][1]==b[3][2] && b[3][2]==b[4][3] && b[4][3]==b[5][4]
  && b[5][4]!= 0) ||
(b[3][2]==b[4][3] && b[4][3]==b[5][4] && b[5][4]==b[6][5]
  && b[6][5]!= 0) ||
(b[2][0]==b[3][1] && b[3][1]==b[4][2] && b[4][2]==b[5][3]
  && b[5][3]!= 0) ||
(b[3][1]==b[4][2] && b[4][2]==b[5][3] && b[5][3]==b[6][4]
  && b[6][4]!= 0) ||
(b[3][0]==b[4][1] && b[4][1]==b[5][2] && b[5][2]==b[6][3]
```

```
            && b[6][3]!= 0))
      {
            return whoWon(b);
      }

      return 0;
}

//checks who won the game
private int whoWon(int b[][]) {
      int xs = 0;
      int os = 0;

      for (int i=0; i<b.length; i++) {
            for (int j=0; j<b[i].length; j++) {
                  if (b[i][j]==1) {
                        xs++;
                  }
                  else if (b[i][j]==-1) {
                        os++;
                  }
            }
      }

      if (xs>os) {
            return 100000;
      }
      else {
            return -100000;
      }
}

//the IBEF evaluation function
private int ibefEval(int board[][]) {
      return
      3 * board[0][0] +
      4 * board[1][0] +
      5 * board[2][0] +
      7 * board[3][0] +
      5 * board[4][0] +
      4 * board[5][0] +
      3 * board[6][0] +
```

```
        4 * board[0][1] +
        6 * board[1][1] +
        8 * board[2][1] +
        10 * board[3][1] +
        8 * board[4][1] +
        6 * board[5][1] +
        4 * board[6][1] +
        5 * board[0][2] +
        8 * board[1][2] +
        11 * board[2][2] +
        13 * board[3][2] +
        11 * board[4][2] +
        8 * board[5][2] +
        5 * board[6][2] +
        5 * board[0][3] +
        8 * board[1][3] +
        11 * board[2][3] +
        13 * board[3][3] +
        11 * board[4][3] +
        8 * board[5][3] +
        5 * board[6][3] +
        4 * board[0][4] +
        6 * board[1][4] +
        8 * board[2][4] +
        10 * board[3][4] +
        8 * board[4][4] +
        6 * board[5][4] +
        4 * board[6][4] +
        3 * board[0][5] +
        4 * board[1][5] +
        5 * board[2][5] +
        7 * board[3][5] +
        5 * board[4][5] +
        4 * board[5][5] +
        3 * board[6][5];
}

//empty function inherited from Agent interface
public void setEpsilon(double e) {
}
```

```
	//empty function inherited from Agent interface
	public void setLearningRate(double l) {}

	//empty function inherited from Agent interface
	public void gameFinished(double reward) {}

	//empty function inherited from Agent interface
	public void setLambda(double l) {}

}
```

C.6. C4ADATEAgent.java

```
/* agent making moves according to a minimax algorithm using the ADATE-
synthesized function at the cutoff points in the game tree */
public class C4ADATEAgent implements Agent {

	private Environment env;
	private String id;
	private boolean x;
	private int cutoff;

	public C4ADATEAgent(String id, boolean x, int cutoff) {
		this.env = null;
		this.id = id;
		this.x = x;
		this.cutoff = cutoff;
	}

	public C4ADATEAgent(String id, boolean x) {
		this(id, x, 3);
	}

	public void setEnvironment(Environment env) {
		this.env = env;
	}

	/* returns a move chosen according to the combination of the minimax
	algorithm and the ADATE-function */
	public int makeYourMove(int board[][], double reward) {
		int move = chooseMove(board);
		return move;
```

```
}

private int chooseMove(int b[][]) {
      return minimax_ab(b, x, 0, -999999, 999999);
}

//empty function inherited from Agent interface
public void resetEligibilityTraces() {}

//minimax algorithm with alpha-beta pruning
private int minimax_ab(int brd[][], boolean x, int depth, int a, int b) {
      if (gameOver(brd)) {
                  return getEndResult(brd);
      }
      if (depth == cutoff) {
            return Eval.adateEval(brd);
      }

      int best_move = 999999;
      int best_score = (x?-999999:999999);
      int n_Alpha = -999999;
      int n_Beta = 999999;

      int move_order[] = randomize();

      for (int i=0; i<brd.length; i++) {
            if (brd[move_order[i]][5] == 0) {
                  int j = findFirstVacant(brd[move_order[i]]);
                  brd[move_order[i]][j] = (x?1:-1);
                  if (x) {
                        int score = minimax_ab(brd, !x, depth+1, Math.max
                              (n_Alpha,a),b);
                        if (score > n_Alpha) {
                              n_Alpha = score;
                              best_move = move_order[i];
                              if (n_Alpha>=b) {
                                    brd[move_order[i]][j] = 0;
                                    break;
                              }
                        }
                  }
                  else if (!x) {
```

```
                        int score = minimax_ab(brd, !x, depth+1, a,
                              Math.min(b,n_Beta));
                        if (score < n_Beta) {
                              n_Beta = score;
                              best_move = move_order[i];
                              if (a>=n_Beta) {
                                    brd[move_order[i]][j] = 0;
                                    break;
                              }
                        }
                  }
                  brd[move_order[i]][j] = 0;
            }
      }

      if (depth==0) {
            return best_move;
      }
      else {
            return (x?n_Alpha:n_Beta);
      }

}

//randomized order of considered moves
private int[] randomize() {
      int mo[] = new int[7];

      for (int i=0; i<mo.length; i++) {
            mo[i] = i;
      }

      for (int i=0; i<mo.length; i++) {
            int n = (int)(Math.random()*mo.length);
            int temp = mo[i];
            mo[i] = mo[n];
            mo[n] = temp;
      }

      return mo;
}
```

```
//finds the first vacant row of a column
private int findFirstVacant(int column[]) {
      for (int i=0; i<column.length; i++) {
            if (column[i]==0) {
                  return i;
            }
      }
      return -1;
}

//checks if the game is over
public boolean gameOver(int b[][]) {
      //check for horizontal wins
      for (int i=0; i<b[0].length; i++) {
            if ((b[0][i]==b[1][i] && b[1][i]==b[2][i] && b[2][i]==b[3][i]
                        && b[3][i]!=0) ||
                (b[1][i]==b[2][i] && b[2][i]==b[3][i] && b[3][i]==b[4][i]
                        && b[4][i]!=0) ||
                (b[2][i]==b[3][i] && b[3][i]==b[4][i] && b[4][i]==b[5][i]
                        && b[5][i]!=0) ||
                (b[3][i]==b[4][i] && b[4][i]==b[5][i] && b[5][i]==b[6][i]
                        && b[6][i]!=0))
            {
                  return true;
            }
      }

      //check for vertical wins
      for (int i=0; i<b.length; i++) {
            if ((b[i][0]==b[i][1] && b[i][1]==b[i][2]&& b[i][2]==b[i][3]
                  && b[i][3]!=0) ||
                (b[i][1]==b[i][2] && b[i][2]==b[i][3]&& b[i][3]==b[i][4]
                        && b[i][4]!=0) ||
                (b[i][2]==b[i][3] && b[i][3]==b[i][4]&& b[i][4]==b[i][5]
                        && b[i][5]!=0))
            {
                  return true;
            }
      }

      //check for diagonal wins
      if ((b[0][3]==b[1][2] && b[1][2]==b[2][1] && b[2][1]==b[3][0]
            && b[3][0]!= 0) ||
           (b[0][4]==b[1][3] && b[1][3]==b[2][2] && b[2][2]==b[3][1]
            && b[3][1]!= 0) ||
```

```
(b[1][3]==b[2][2] && b[2][2]==b[3][1] && b[3][1]==b[4][0]
  && b[4][0]!= 0) ||
(b[0][5]==b[1][4] && b[1][4]==b[2][3] && b[2][3]==b[3][2]
  && b[3][2]!= 0) ||
(b[1][4]==b[2][3] && b[2][3]==b[3][2] && b[3][2]==b[4][1]
  && b[4][1]!= 0) ||
(b[2][3]==b[3][2] && b[3][2]==b[4][1] && b[4][1]==b[5][0]
  && b[5][0]!= 0) ||
(b[1][5]==b[2][4] && b[2][4]==b[3][3] && b[3][3]==b[4][2]
  && b[4][2]!= 0) ||
(b[2][4]==b[3][3] && b[3][3]==b[4][2] && b[4][2]==b[5][1]
  && b[5][1]!= 0) ||
(b[3][3]==b[4][2] && b[4][2]==b[5][1] && b[5][1]==b[6][0]
  && b[6][0]!= 0) ||
(b[3][4]==b[4][3] && b[4][3]==b[5][2] && b[5][2]==b[6][1]
  && b[6][1]!= 0) ||
(b[3][5]==b[4][4] && b[4][4]==b[5][3] && b[5][3]==b[6][2]
  && b[6][2]!= 0) ||
(b[0][2]==b[1][3] && b[1][3]==b[2][4] && b[2][4]==b[3][5]
  && b[3][5]!= 0) ||
(b[0][1]==b[1][2] && b[1][2]==b[2][3] && b[2][3]==b[3][4]
  && b[3][4]!= 0) ||
(b[1][2]==b[2][3] && b[2][3]==b[3][4] && b[3][4]==b[4][5]
  && b[4][5]!= 0) ||
(b[0][0]==b[1][1] && b[1][1]==b[2][2] && b[2][2]==b[3][3]
  && b[3][3]!= 0) ||
(b[1][1]==b[2][2] && b[2][2]==b[3][3] && b[3][3]==b[4][4]
  && b[4][4]!= 0) ||
(b[2][2]==b[3][3] && b[3][3]==b[4][4] && b[4][4]==b[5][5]
  && b[5][5]!= 0) ||
(b[1][0]==b[2][1] && b[2][1]==b[3][2] && b[3][2]==b[4][3]
  && b[4][3]!= 0) ||
(b[2][1]==b[3][2] && b[3][2]==b[4][3] && b[4][3]==b[5][4]
  && b[5][4]!= 0) ||
(b[3][2]==b[4][3] && b[4][3]==b[5][4] && b[5][4]==b[6][5]
  && b[6][5]!= 0) ||
(b[2][0]==b[3][1] && b[3][1]==b[4][2] && b[4][2]==b[5][3]
  && b[5][3]!= 0) ||
(b[3][1]==b[4][2] && b[4][2]==b[5][3] && b[5][3]==b[6][4]
  && b[6][4]!= 0) ||
(b[3][0]==b[4][1] && b[4][1]==b[5][2] && b[5][2]==b[6][3]
  && b[6][3]!= 0))
```

```
	{
		return true;
	}

	else {
		for (int i=0; i<b.length; i++) {
			for (int j=0; j<b[i].length; j++) {
				if (b[i][j] == 0) {
					//Game is not over
					return false;
				}
			}
		}
		//Draw
		return true;
	}
}

//checks the end result of the game
public int getEndResult(int b[][]) {
	//check for horizontal wins
	for (int i=0; i<b[0].length; i++) {
		if ((b[0][i]==b[1][i] && b[1][i]==b[2][i] && b[2][i]==b[3][i]
		  && b[3][i]!=0) ||
		    (b[1][i]==b[2][i] && b[2][i]==b[3][i] && b[3][i]==b[4][i]
		  && b[4][i]!=0) ||
		    (b[2][i]==b[3][i] && b[3][i]==b[4][i] && b[4][i]==b[5][i]
		  && b[5][i]!=0) ||
		    (b[3][i]==b[4][i] && b[4][i]==b[5][i] && b[5][i]==b[6][i]
		  && b[6][i]!=0))
		{
			return whoWon(b);
		}
	}

	//check for vertical wins
	for (int i=0; i<b.length; i++) {
		if ((b[i][0]==b[i][1] && b[i][1]==b[i][2]&& b[i][2]==b[i][3]
		  && b[i][3]!=0) ||
		    (b[i][1]==b[i][2] && b[i][2]==b[i][3]&& b[i][3]==b[i][4]
		  && b[i][4]!=0) ||
		    (b[i][2]==b[i][3] && b[i][3]==b[i][4]&& b[i][4]==b[i][5]
		  && b[i][5]!=0))
		{
			return whoWon(b);
```

```
      }
}

//check for diagonal wins
if ((b[0][3]==b[1][2] && b[1][2]==b[2][1] && b[2][1]==b[3][0]
      && b[3][0]!= 0) ||
    (b[0][4]==b[1][3] && b[1][3]==b[2][2] && b[2][2]==b[3][1]
      && b[3][1]!= 0) ||
    (b[1][3]==b[2][2] && b[2][2]==b[3][1] && b[3][1]==b[4][0]
      && b[4][0]!= 0) ||
    (b[0][5]==b[1][4] && b[1][4]==b[2][3] && b[2][3]==b[3][2]
      && b[3][2]!= 0) ||
    (b[1][4]==b[2][3] && b[2][3]==b[3][2] && b[3][2]==b[4][1]
      && b[4][1]!= 0) ||
    (b[2][3]==b[3][2] && b[3][2]==b[4][1] && b[4][1]==b[5][0]
      && b[5][0]!= 0) ||
    (b[1][5]==b[2][4] && b[2][4]==b[3][3] && b[3][3]==b[4][2]
      && b[4][2]!= 0) ||
    (b[2][4]==b[3][3] && b[3][3]==b[4][2] && b[4][2]==b[5][1]
      && b[5][1]!= 0) ||
    (b[3][3]==b[4][2] && b[4][2]==b[5][1] && b[5][1]==b[6][0]
      && b[6][0]!= 0) ||
    (b[3][4]==b[4][3] && b[4][3]==b[5][2] && b[5][2]==b[6][1]
      && b[6][1]!= 0) ||
    (b[3][5]==b[4][4] && b[4][4]==b[5][3] && b[5][3]==b[6][2]
      && b[6][2]!= 0) ||
    (b[0][2]==b[1][3] && b[1][3]==b[2][4] && b[2][4]==b[3][5]
      && b[3][5]!= 0) ||
    (b[0][1]==b[1][2] && b[1][2]==b[2][3] && b[2][3]==b[3][4]
      && b[3][4]!= 0) ||
    (b[1][2]==b[2][3] && b[2][3]==b[3][4] && b[3][4]==b[4][5]
      && b[4][5]!= 0) ||
    (b[0][0]==b[1][1] && b[1][1]==b[2][2] && b[2][2]==b[3][3]
      && b[3][3]!= 0) ||
    (b[1][1]==b[2][2] && b[2][2]==b[3][3] && b[3][3]==b[4][4]
      && b[4][4]!= 0) ||
    (b[2][2]==b[3][3] && b[3][3]==b[4][4] && b[4][4]==b[5][5]
      && b[5][5]!= 0) ||
    (b[1][0]==b[2][1] && b[2][1]==b[3][2] && b[3][2]==b[4][3]
      && b[4][3]!= 0) ||
    (b[2][1]==b[3][2] && b[3][2]==b[4][3] && b[4][3]==b[5][4]
      && b[5][4]!= 0) ||
```

```
            (b[3][2]==b[4][3] && b[4][3]==b[5][4] && b[5][4]==b[6][5]
              && b[6][5]!= 0) ||
            (b[2][0]==b[3][1] && b[3][1]==b[4][2] && b[4][2]==b[5][3]
              && b[5][3]!= 0) ||
            (b[3][1]==b[4][2] && b[4][2]==b[5][3] && b[5][3]==b[6][4]
              && b[6][4]!= 0) ||
            (b[3][0]==b[4][1] && b[4][1]==b[5][2] && b[5][2]==b[6][3]
              && b[6][3]!= 0))
      {
            return whoWon(b);
      }

      return 0;
}

//checks who won the game
private int whoWon(int b[][]) {
      int xs = 0;
      int os = 0;

      for (int i=0; i<b.length; i++) {
            for (int j=0; j<b[i].length; j++) {
                  if (b[i][j]==1) {
                        xs++;
                  }
                  else if (b[i][j]==-1) {
                        os++;
                  }
            }
      }

      if (xs>os) {
            return 100000;
      }
      else {
            return -100000;
      }
}

//the board evaluation function synthesized by ADATE
private int adateEval(int board[][]) {
      return 1 +
```

```
        2 * board[1][0] +
        board[3][0] +
        12 * board[4][0] +
        2 * board[1][1] +
        5 * board[2][1] +
        5 * board[4][1] +
        4 * board[5][1] +
        4 * board[6][1] +
        board[1][2] +
        4 * board[2][2] +
        16 * board[3][2] +
        2 * board[6][2] +
        12 * board[1][3] +
        6 * board[2][3] +
        14 * board[3][3] +
        12 * board[4][3] +
        11 * board[5][3] +
        2 * board[6][3] +
        2 * board[1][4] +
        6 * board[2][4] +
        6 * board[3][4] +
        2 * board[4][4] +
        4 * board[5][4] +
        board[6][4] +
        2 * board[0][5] +
        2 * board[2][5] +
        2 * board[3][5] +
        2 * board[4][5] +
        2 * board[5][5] +
        board[6][5];
}

//empty function inherited by ADATE
public void setEpsilon(double e) {}

//empty function inherited by ADATE
public void setLearningRate(double l) {}

//empty function inherited by ADATE
public void gameFinished(double reward) {}
```

```
	//empty function inherited by ADATE
	public void setLambda(double l) {}

}
```

C.7. ANN_TD.java

```
//implementation of the neural network
public class ANN_TD implements Serializable {

	private double l_rate;
	private double momentum;
	private double lambda;
	private double d_rate;
	private int input;
	private int hidden;
	private String hidden_func;
	private int output;
	private String output_func;
	public double w1[][];
	public double w2[][];
	private double w1_update[][];
	private double w2_update[][];
	private double e1[][][];
	private double e2[][];
	public double b1[];
	public double b2[];
	private double b1_update[];
	private double b2_update[];
	private double b_e1[][];
	private double b_e2[];
	private String type;

	public ANN_TD(String type, double l_rate, double momentum, int input, int
		hidden, String hidden_func, int output, String output_func) {

		if (l_rate <= 0) {
			System.out.println("Learning rate must be greater than
				zero.");
			System.exit(1);
		}
		if (momentum < 0) {
```

```
            System.out.println("Momentum must be greater than or equal to
                  zero.");
            System.exit(1);
      }
      if (input <= 0) {
            System.out.println("The number of inputs must be greater than
                  zero.");
            System.exit(1);
      }
      if (hidden <= 0) {
            System.out.println("The number of neurons on the hidden layer
                  must be greater than zero.");
            System.exit(1);
      }
      if (output <= 0) {
            System.out.println("The number of output units must be greater
                  than zero.");
            System.exit(1);
      }
      if (!(output_func.equals("sigmoid") || output_func.equals("tanh") ||
                  output_func.equals("linear"))) {
            System.out.println("Invalid output transfer function. Choose
                  between sigmoid, tanh and linear.");
            System.exit(1);
      }
      if (!(hidden_func.equals("sigmoid") || hidden_func.equals("tanh") ||
            hidden_func.equals("linear"))) {
            System.out.println("Invalid hidden layer transfer function.
                  Choose between sigmoid, tanh and linear.");
            System.exit(1);
      }

      this.type = type;
      this.l_rate = l_rate;
      this.momentum = momentum;
      this.input = input;
      this.hidden = hidden;
      this.hidden_func = hidden_func;
      this.output = output;
      this.output_func = output_func;
      d_rate = 0.0;
      lambda = 0.0;

      setup();
}
```

```
public ANN_TD(String type, int input, int hidden, String func, int output)
{
      this(type, 0.05, 0.0, input, hidden, func, output, func);
}

public ANN_TD(String type, int input, int hidden, int output) {
      this(type, input, hidden, "sigmoid", output);
}

//sets up the neural network
private void setup() {
      w1 = new double[input][hidden];
      w1_update = new double[input][hidden];
      e1 = new double[input][hidden][output];
      w2 = new double[hidden][output];
      w2_update = new double[hidden][output];
      e2 = new double[hidden][output];

      b1 = new double[hidden];
      b1_update = new double[hidden];
      b_e1 = new double[hidden][output];
      b2 = new double[output];
      b2_update = new double[output];
      b_e2 = new double[output];

      for (int i=0; i<input; i++) {
            for (int j=0; j<hidden; j++) {
                  w1[i][j] = 0.0;
                  w1_update[i][j] = 0.0;
                  for (int k=0; k<output; k++) {
                        e1[i][j][k] = 0.0;
                        w2[j][k] = 0.0;
                        w2_update[j][k] = 0.0;
                        e2[j][k] = 0.0;
                        b_e1[j][k] = 0.0;
                        b2[k] = 0.0;
                        b1_update[k] = 0.0;
                        b_e2[k] = 0.0;
                  }
                  b1[j] = 0.0;
                  b1_update[j] = 0.0;
```

```
            }
      }

}

//resets eligibility traces
public void resetE() {
      for (int j=0; j<hidden; j++) {
            for (int k=0; k<output; k++) {
                  for (int i=0; i<input; i++) {
                        e1[i][j][k] = 0.0;
                  }
                  e2[j][k] = 0.0;
                  b_e1[j][k] = 0.0;
            }
      }

      for (int i=0; i<output; i++) {
            b_e2[i] = 0.0;
      }

}

//sets the learning rate
public void setLearningRate(double l) {
      l_rate = l;
}

//sets the lambda parameter
public void setLambda(double l) {
      if (l<0.0 || l>1.0) {
            System.out.println("Error: lambda parameter must be given a
                  value ranging from 0.0 to 1.0");
      }
      else {
            lambda = l;
      }
}

//sets the discount rate
public void setDiscountRate(double d) {
      if (d<0.0 || d>1.0) {
```

```
            System.out.println("Error: discount rate parameter must be
                  given a value ranging from 0.0 to 1.0");
      }
      else {
            d_rate = d;
      }
}

//initializes network weights to random values
public void randomWeights(double min, double max) {
      for (int i=0; i<w1.length; i++) {
            for (int j=0; j<w1[i].length; j++) {
                  w1[i][j] = (Math.random()*(max-min))+min;
            }
      }

      for (int i=0; i<w2.length; i++) {
            for (int j=0; j<w2[i].length; j++) {
                  w2[i][j] = (Math.random()*(max-min))+min;
            }
      }

      for (int i=0; i<b1.length; i++) {
                  b1[i] = (Math.random()*(max-min))+min;
      }

      for (int i=0; i<b2.length; i++) {
                  b2[i] = (Math.random()*(max-min))+min;
      }

}

//trains the network
public void train(double input[], double[] target_value) {
      if (input.length != this.input) {
            System.out.println("ANN.train: Incorrect number of elements in
                  input vector");
            System.exit(1);
      }
      if (target_value.length != this.output) {
            System.out.println("ANN.train: Incorrect number of elements in
                  target vector");
            System.exit(1);
```

```
}

double outputs[][] = simulate(input);
double errors[] = new double[outputs[1].length];
double deriv[][] = getDerivatives(outputs);
double o = 0.0;

//computing training errors
for (int i=0; i<output; i++) {
      errors[i] = target_value[i] - outputs[1][i];
}

//REGULAR WEIGHTS
//updating eligibility traces for regular weights
for (int j=0; j<hidden; j++) {
      for (int k=0; k<output; k++) {
            e2[j][k] = d_rate*lambda*e2[j][k] +
                  deriv[1][k]*outputs[0][j];
            for (int i=0; i<input.length; i++) {
                  e1[i][j][k] = d_rate*lambda*e1[i][j][k] +
                        deriv[1][k]*w2[j][k]*deriv[0][j]*input[i];
            }
      }
}

//including momentum from previous update regarding hidden layer
//weights
for (int i=0; i<w1_update.length; i++) {
      for (int j=0; j<w1_update[i].length; j++) {
            w1_update[i][j] *= momentum;
      }
}
//including momentum from previous update regarding output layer
//weights
for (int i=0; i<w2_update.length; i++) {
      for (int j=0; j<w2_update[i].length; j++) {
            w2_update[i][j] *= momentum;
      }
}

//computing weight updates
for (int k=0; k<output; k++) {
      for (int j=0; j<hidden; j++) {
```

```
            w2_update[j][k] += l_rate*errors[k]*e2[j][k];
            for (int i=0; i<input.length; i++) {
                  w1_update[i][j] += l_rate*errors[k]*e1[i][j][k];
            }
      }
}

//updating hidden layer weights
for (int i=0; i<w1.length; i++) {
      for (int j=0; j<w1[i].length; j++) {
            w1[i][j] += w1_update[i][j];
      }
}

//updating output layer weights
for (int i=0; i<w2.length; i++) {
      for (int j=0; j<w2[i].length; j++) {
            w2[i][j] += w2_update[i][j];
      }
}



//BIAS WEIGHTS
//updating eligibility traces for bias weights
for (int j=0; j<output; j++) {
      b_e2[j] = d_rate*lambda*b_e2[j] + deriv[1][j] * 1.0;
      for (int i=0; i<hidden; i++) {
            b_e1[i][j] = d_rate*lambda*b_e1[i][j] +
                  deriv[1][j]*w2[i][j]*deriv[0][i] * 1.0;
      }
}

//including momentum from previous update regarding hidden layer
//bias weights
for (int i=0; i<b1_update.length; i++) {
      b1_update[i] *= momentum;
}
//including momentum from previous update regarding output layer //
//bias weights
for (int i=0; i<b2_update.length; i++) {
      b2_update[i] *= momentum;
}
```

```
        //computing bias weight updates
        for (int k=0; k<output; k++) {
              b2_update[k] += l_rate*errors[k]*b_e2[k];
              for (int j=0; j<hidden; j++) {
                    b1_update[j] += l_rate*errors[k]*b_e1[j][k];
              }
        }

        //updating hidden layer bias weights
        for (int i=0; i<b1.length; i++) {
              b1[i] += b1_update[i];
        }

        //updating output layer bias weights
        for (int i=0; i<b2.length; i++) {
              b2[i] += b2_update[i];
        }

}

//computes derivatives for the transfer functions
private double[][] getDerivatives(double outputs[][]) {
        double deriv[][] = new double[outputs.length][];
        deriv[0] = new double[outputs[0].length];
        deriv[1] = new double[outputs[1].length];

        //derivatives for hidden layer
        if (hidden_func.equals("sigmoid")) {
              for (int i=0; i<outputs[0].length; i++) {
                    deriv[0][i] = outputs[0][i]*(1.0-outputs[0][i]);
              }
        }
        else if (hidden_func.equals("tanh")) {
              for (int i=0; i<outputs[0].length; i++) {
                    deriv[0][i] = 1.0-outputs[0][i]*outputs[0][i];
              }
        }
        else {
              for (int i=0; i<outputs[0].length; i++) {
                    deriv[0][i] = 1.0 * outputs[0][i];
              }
        }
```

```
        //derivatives for output layer
        if (output_func.equals("sigmoid")) {
              for (int i=0; i<outputs[1].length; i++) {
                    deriv[1][i] = outputs[1][i]*(1.0-outputs[1][i]);
              }
        }
        else if (output_func.equals("tanh")) {
              for (int i=0; i<outputs[1].length; i++) {
                    deriv[1][i] = 1.0-outputs[1][i]*outputs[1][i];
              }
        }
        else {
              for (int i=0; i<outputs[1].length; i++) {
                    deriv[1][i] = 1.0 * outputs[1][i];
              }
        }

        return deriv;
  }

  //activates the network, returning all network outputs
  public double[][] simulate(double input[]) {
        if (input.length != this.input) {
              System.out.println("ANN_TD.simulate: incorrect number of
                    elements in input vector");
              System.exit(1);
        }
        double intermed[][] = new double[2][];

        intermed[0] = new double[hidden];
        intermed[1] = new double[output];

        for (int i=0; i<intermed.length; i++) {
              for (int j=0; j<intermed[i].length; j++) {
                    intermed[i][j] = 0.0;
              }
        }

        for (int i=0; i<intermed[0].length; i++) {
              for (int j=0; j<input.length; j++) {
                    intermed[0][i] += input[j]*w1[j][i];
              }
```

```
                intermed[0][i] += b1[i]; //correct?
                intermed[0][i] = transfer_func(intermed[0][i], hidden_func);

          }



          for (int i=0; i<output; i++) {
                for (int j=0; j<intermed[0].length; j++) {
                      intermed[1][i] += intermed[0][j]*w2[j][i];
                }
                intermed[1][i] += b2[i];
                intermed[1][i] = transfer_func(intermed[1][i], output_func);
          }
          return intermed;
    }

    //activates the network, returning the network output
    public double[] apply(double input[]) {
          return simulate(input)[1];
    }

    //applies the proper transfer function
    private double transfer_func(double sum, String func) {
          if (func.equals("sigmoid")) {
                return 1/(1+Math.pow(Math.E, -sum));
          }
          else if (func.equals("tanh")) {
                return Math.tanh(sum);
          }
          else {
                return sum;
          }
    }

}
```

D. Playing result statistics

This appendix contains results from making the different agents play against each other with varying cutoff depth levels. The agents that were tested are

- The C4IBEFAgent, in this case denoted IBEFx, where x denotes the cutoff depth level used by the agent.
- The C4ADATEAgent, in this case denoted ADATEx, where x denotes the cutoff depth level used by the agent.
- The C4NNPlayAgent, using a neural network trained by a C4NNLearnAgent. The playing agent is in this case denoted RLx, where x denotes the cutoff depth level used by the agent.
- The C4RandomPlayAgent, int his case denoted RANDOM. The random agent chooses moves at random without searching the game tree, and is therefore not associated with a cutoff depth.

This appendix shows the results of all playing combinations between these four types of agents, up to and including search depth 6. The exception being the random agent, which does not employ game tree search. This makes up a total of 361 combinations. The results are ordered in terms of the X-playing agent.

The lines in this survey are presented in the following format: The first column of the line is the name of the X-player, the second column, after the hyphen, is the name of the O-player. The first number is the percentage of games won by the X-player, the second number is the percentage of games won by the O-player. The last number, enclosed in parentheses, is the percentage of games that were drawn. In other words, like this:

```
XPlayer – OPlayer %Xwins - %OWins (%Draws)
```

*****IBEF1*****

```
IBEF1 - RANDOM 98.4 - 1.6 (0)

IBEF1 - IBEF1 81.5 - 18.5 (0)
IBEF1 - ADATE1 27.4 - 71.3 (1.3)
IBEF1 - RL1 26.3 - 73.5 (0.2)

IBEF1 - IBEF2 57 - 32 (1)
IBEF1 - ADATE2 0 - 100 (0)
IBEF1 - RL2 9 - 91 (0)

IBEF1 - IBEF3 3.1 - 72 (24.9)
IBEF1 - ADATE3 0 - 100 (0)
```

IBEF1 - RL3 22.5 - 70.9 (6.6)

IBEF1 - IBEF4 0.8 - 88.5 (10.7)
IBEF1 - ADATE4 0 - 100 (0)
IBEF1 - RL4 21.4 - 77.2 (1.4)

IBEF1 - IBEF5 0.5 - 85.5 (14)
IBEF1 - ADATE5 0 - 100 (0)
IBEF1 - RL5 13 - 85 (2)

IBEF1 - IBEF6 0 - 29.7 (70.3)
IBEF1 - ADATE6 0 - 100 (0)
IBEF1 - RL6 20 - 79 (1)

*****IBEF2*****

IBEF2 - RANDOM 99.8 – 0.2 (0)

IBEF2 - IBEF1 64.2 - 1.6 (34.2)
IBEF2 - ADATE1 63.2 - 31.6 (5.2)
IBEF2 - RL1 56.6 - 43.3 (0.1)

IBEF2 - IBEF2 21.2 - 34.8 (44)
IBEF2 - ADATE2 45.7 - 48.9 (5.4)
IBEF2 - RL2 52.8 - 45.7 (1.5)

IBEF2 - IBEF3 26.6 - 41.6 (31.8)
IBEF2 - ADATE3 24.6 - 66.9 (8.5)
IBEF2 - RL3 41.3 - 55.1 (3.6)

IBEF2 - IBEF4 3.3 - 61.8 (34.9)
IBEF2 - ADATE4 17.1 - 77 (5.9)
IBEF2 - RL4 62 - 34.6 (3.4)

IBEF2 - IBEF5 6.6 - 68.4 (25)
IBEF2 - ADATE5 13.2 - 79.8 (7)
IBEF2 - RL5 36 - 60 (4)

IBEF2 - IBEF6 9.2 - 75.7 (15.1)
IBEF2 - ADATE6 15.3 - 83.5 (1.2)
IBEF2 - RL6 35 - 60 (5)

*****IBEF3*****

IBEF3 - RANDOM 100 - 0 (0)

IBEF3 - IBEF1 100 - 0 (0)
IBEF3 - ADATE1 89 - 6.5 (4.5)
IBEF3 - RL1 98.5 - 1.5 (0)

IBEF3 - IBEF2 54.4 - 2.8 (42.8)
IBEF3 - ADATE2 0 - 51.3 (48.7)
IBEF3 - RL2 58.6 - 37.6 (3.8)

IBEF3 - IBEF3 57.4 - 17.7 (24.9)
IBEF3 - ADATE3 0 - 52.5 (47.5)
IBEF3 - RL3 64.4 - 32 (3.6)

IBEF3 - IBEF4 0 - 49.4 (50.6)
IBEF3 - ADATE4 0 - 52.8 (47.2)
IBEF3 - RL4 32.2 - 63 (4.8)

IBEF3 - IBEF5 13.5 - 37.3 (49.2)
IBEF3 - ADATE5 0 - 52.4 (47.6)
IBEF3 - RL5 44 - 52 (4)

IBEF3 - IBEF6 12.1 - 16.1 (71.8)
IBEF3 - ADATE6 0 - 100 (0)
IBEF3 - RL6 48 - 48 (4)

*****IBEF4*****

IBEF4 - RANDOM 100 - 0 (0)

IBEF4 - IBEF1 100 - 0 (0)
IBFE4 - ADATE1 92.5 - 5.5 (2)
IBEF4 - RL1 94.7 - 5.2 (0.1)

IBEF4 - IBEF2 75 - 7.8 (17.2)
IBEF4 - ADATE2 3.8 - 96.2 (0)
IBEF4 - RL2 64.8 - 33.7 (1.5)

IBEF4 - IBEF3 64.8 - 12.4 (22.8)
IBEF4 - ADATE3 3.5 - 96.5 (0)
IBEF4 - RL3 90.9 - 4.9 (4.2)

IBEF4 - IBEF4 0.7 - 87.9 (11.4)
IBEF4 - ADATE4 11.5 - 88.5 (0)
IBEF4 - RL4 65.7 - 25.7 (8.6)

IBEF4 - IBEF5 4.1 - 58.7 (37.1)
IBEF4 - ADATE5 4.1 - 95.9 (0)
IBEF4 - RL5 73 - 25 (2)

IBEF4 - IBEF6 12.8 - 4.5 (82.7)
IBEF4 - ADATE6 0 - 100 (0)
IBEF4 - RL6 54 - 37 (9)

*****IBEF5*****

IBEF5 - RANDOM 100 - 0 (0)

IBEF5 - IBEF1 97.8 - 1.8 (0.4)
IBEF5 - ADATE1 98.7 - 1.3 (0)
IBEF5 - RL1 100 - 0 (0)

IBEF5 - IBEF2 100 - 0 (0)
IBEF5 - ADATE2 26 - 59.1 (14.9)
IBEF5 - RL2 95.4 - 3.9 (0.7)

IBEF5 - IBEF3 89.7 - 5.8 (4.5)
IBEF5 - ADATE3 26.6 - 42.8 (30.6)
IBEF5 - RL3 85.2 - 14.6 (0.2)

IBEF5 - IBEF4 37.2 - 36.2 (26.6)
IBEF5 - ADATE4 22.1 - 74.3 (3.6)
IBEF5 - RL4 83 - 13 (4)

IBEF5 - IBEF5 39.6 - 46 (14.4)
IBEF5 - ADATE5 33 - 34.9 (32.1)
IBEF5 - RL5 69 - 26 (5)

IBEF5 - IBEF6 42.4 - 48.4 (9.2)
IBEF5 - ADATE6 30.3 - 27 (42.7)
IBEF5 - RL6 59 - 30 (11)

*****IBEF6*****

IBEF6 - RANDOM 100 - 0 (0)

IBEF6 - IBEF1 100 - 0 (0)
IBEF6 - ADATE1 88.6 - 8.5 (2.9)
IBEF6 - RL1 94.2 - 5.8 (0)

IBEF6 - IBEF2 100 - 0 (0)
IBEF6 - ADATE2 7.7 - 68 (24.3)
IBEF6 - RL2 85.7 - 6.8 (7.5)

IBEF6 - IBEF3 100 - 0 (0)
IBEF6 - ADATE3 8.5 - 68.6 (22.9)
IBEF6 - RL3 100 - 0 (0)

IBEF6 - IBEF4 42.8 - 29.2 (28)
IBEF6 - ADATE4 10.3 - 68.7 (21)
IBEF6 - RL4 55 - 24 (21)

IBEF6 - IBEF5 28 - 43.5 (28.5)
IBEF6 - ADATE5 1.2 - 76.2 (22.6)
IBEF6 - RL5 79 - 9 (12)

IBEF6 - IBEF6 3.9 - 11.1 (85)
IBEF6 - ADATE6 0 - 100 (0)
IBEF6 - RL6 60 - 13 (27)

*****ADATE1*****

ADATE1 - RANDOM 80.9 - 19.1 (0)

ADATE1 - IBEF1 100 - 0 (0)
ADATE1 - ADATE1 58.3 - 38.4 (3.3)

ADATE1 - RL1 44.7 - 55.3 (0)

ADATE1 - IBEF2 34 - 65.6 (0.4)
ADATE1 - ADATE2 20.4 - 62.3 (17.3)
ADATE1 - RL2 10.1 - 89.8 (0.1)

ADATE1 - IBEF3 0 - 100 (0)
ADATE1 - ADATE3 0.2 - 96.6 (3.2)
ADATE1 - RL3 0.5 - 99.4 (0.1)

ADATE1 - IBEF4 0.7 - 99.3 (0)
ADATE1 - ADATE4 6.6 - 90.7 (2.7)
ADATE1 - RL4 7.6 - 91.4 (1)

ADATE1 - IBEF5 1.5 - 98.5 (0)
ADATE1 - ADATE5 1.9 - 96.4 (1.7)
ADATE1 - RL5 29 - 71 (0)

ADATE1 - IBEF6 0 - 100 (0)
ADATE1 - ADATE6 8.6 - 88.8 (2.6)
ADATE1 - RL6 0 - 100 (0)

*****ADATE2*****

ADATE2 - RANDOM 99.5 - 0.4 (0.1)

ADATE2 - IBEF1 100 - 0 (0)
ADATE2 - ADATE1 100 - 0 (0)
ADATE2 - RL1 87.7 - 12 (0.3)

ADATE2 - IBEF2 61.6 - 35 (3.4)
ADATE2 - ADATE2 100 - 0 (0)
ADATE2 - RL2 33.5 - 65.5 (1)

ADATE2 - IBEF3 8.3 - 86.4 (5.3)
ADATE2 - ADATE3 44.8 - 0 (55.2)
ADATE2 - RL3 40.5 - 37.8 (21.7)

ADATE2 - IBEF4 0.9 - 99.1 (0)
ADATE2 - ADATE4 19.7 - 52.8 (27.5)
ADATE2 - RL4 12 - 71 (17)

ADATE2 - IBEF5 8.5 - 82.1 (15)
ADATE2 - ADATE5 27.8 - 42.5 (30.7)
ADATE2 - RL5 49 - 26 (25)

ADATE2 - IBEF6 4 - 85.8 (10.2)
ADATE2 - ADATE6 31.7 - 49.9 (18.4)
ADATE2 - RL6 25 - 6 (69)

*****ADATE3*****

ADATE3 - RANDOM 99.8 - 0.2

ADATE3 - IBEF1 100 - 0 (0)
ADATE3 - ADATE1 100 - 0 (0)
ADATE3 - RL1 90.3 - 9.7 (0)

ADATE3 - IBEF2 67.1 - 26.8 (6.1)
ADATE3 - ADATE2 100 - 0 (0)
ADATE3 - RL2 44 - 54.5 (1.5)

ADATE3 - IBEF3 0.1 - 90 (9.9)
ADATE3 - ADATE3 55.4 - 35 (9.6)
ADATE3 - RL3 56 - 29.6 (14.4)

ADATE3 - IBEF4 54.4 - 37.4 (8.2)
ADATE3 - ADATE4 58.6 - 26.1 (15.3)
ADATE3 - RL4 15 - 58 (27)

ADATE3 - IBEF5 25.3 - 61.1 (13.6)
ADATE3 - ADATE5 40.3 - 56.7 (3)
ADATE3 - RL5 64 - 29 (7)

ADATE3 - IBEF6 5.2 - 83.9 (10.0)
ADATE3 - ADATE6 40.5 - 49.4 (10.1)
ADATE3 - RL6 37 - 17 (46)

*****ADATE4*****

ADATE4 - RANDOM 99.9 - 0.1 (0)

ADATE4 - IBEF1 100 - 0 (0)
ADATE4 - ADATE1 100 - 0 (0)
ADATE4 - RL1 97.9 - 1.9 (0.2)

ADATE4 - IBEF2 73.4 - 22.7 (3.9)
ADATE4 - ADATE2 87.2 - 10 (28)
ADATE4 - RL2 40.1 - 53 (6.9)

ADATE4 - IBEF3 13.2 - 71.3 (15.5)
ADATE4 - ADATE3 69.2 - 30.8 (0)
ADATE4 - RL3 74.7 - 10.1 (15.2)

ADATE4 - IBEF4 52.7 - 41 (6.3)
ADATE4 - ADATE4 66.1 - 28.7 (5.2)
ADATE4 - RL4 6 - 68 (26)

ADATE4 - IBEF5 47.4 - 44.6 (8)
ADATE4 - ADATE5 51.2 - 43.6 (5.2)
ADATE4 - RL5 54 - 23 (23)

ADATE4 - IBEF6 2.4 - 92.5 (5.1)
ADATE4 - ADATE6 43.4 - 39.4 (17.2)
ADATE4 - RL6 71 - 0 (29)

*****ADATE5*****

ADATE5 - RANDOM 99.6 - 0.3 (0.1)

ADATE5 - IBEF1 100 - 0 (0)
ADATE5 - ADATE1 100 - 0 (0)
ADATE5 - RL1 97.6 - 2.4 (0)

ADATE5 - IBEF2 65 - 18.3 (16.7)
ADATE5 - ADATE2 100 - 0 - (0)
ADATE5 - RL2 100 - 0 (0)

ADATE5 - IBEF3 23.6 - 30 (46.4)
ADATE5 - ADATE3 100 - 0 (0)
ADATE5 - RL3 82.2 - 9.6 (8.2)

ADATE5 - IBEF4 70.4 - 22.4 (7.2)
ADATE5 - ADATE4 41.6 - 34.9 (23.5)
ADATE5 - RL4 19.4 - 73.7 (6.9)

ADATE5 - IBEF5 45.6 - 35.5 (18.9)
ADATE5 - ADATE5 28.3 - 53.9 (17.8)
ADATE5 - RL5 100 - 0

ADATE5 - IBEF6 9.1 - 70.5 (20.4)
ADATE5 - ADATE6 55.8 - 38.3 (5.9)
ADATE5 - RL6 12 - 78 (10)

*****ADATE6*****

ADATE6 - RANDOM 100 - 0 (0)

ADATE6 - IBEF1 100 - 0 (0)
ADATE6 - ADATE1 100 - 0 (0)
ADATE6 - RL1 98.5 - 1.4 (0.1)

ADATE6 - IBEF2 82.3 - 16 (1.7)
ADATE6 - ADATE2 96.5 - 0.5 (3)
ADATE6 - RL2 75.9 - 12.5 (11.6)

ADATE6 - IBEF3 16.3 - 59.8 (23.9)
ADATE6 - ADATE3 94.3 - 2.1 (3.6)
ADATE6 - RL3 75.2 - 8.4 (16.4)

ADATE6 - IBEF4 33.3 - 60.6 (6.1)
ADATE6 - ADATE4 29.8 - 35 (15.2)
ADATE6 - RL4 46 - 25 (29)

ADATE6 - IBEF5 36.2 - 58.3 (5.5)
ADATE6 - ADATE5 47.7 - 32.8 (19.5)
ADATE6 - RL5 100 - 0 (0)

ADATE6 - IBEF6 10.3 - 85.2 (4.5)
ADATE6 - ADATE6 44 - 39.4 (16.6)
ADATE6 - RL6 68 - 1 (31)

*****RL1*****

RL1 - RANDOM 76.1 - 23.9 (0)

RL1 - IBEF1 100 - 0 (0)
RL1 - ADATE1 100 - 0 (0)
RL1 - RL1 0 - 100 (0)

RL1 - IBEF2 0 - 100 (0)
RL1 - ADATE2 100 - 0 (0)
RL1 - RL2 0 - 100 (0)

RL1 - IBEF3 89.8 - 10.2 (0)
RL1 - ADATE3 100 - 0 (0)
RL1 - RL3 0 - 100 (0)

RL1 - IBEF4 37.7 - 62.3 (0)
RL1 - ADATE4 95.7 - 4.3 (0)
RL1 - RL4 6 - 65 (29)

RL1 - IBEF5 0 - 100 (0)
RL1 - ADATE5 0 - 100 (0)
RL1 - RL5 0 - 100 (0)

RL1 - IBEF6 0 - 100 (0)
RL1 - ADATE6 0 - 100 (0)
RL1 - RL6 0 - 100 (0)

*****RL2*****

RL2 - RANDOM 96.9 - 2.9 (0.2)

RL2 - IBEF1 20.8 - 79.2 (0)
RL2 - ADATE1 100 - 0 (0)
RL2 - RL1 0 - 100 (0)

RL2 - IBEF2 22 - 68 (10)
RL2 - ADATE2 100 - 0 (0)
RL2 - RL2 4.2 - 94.4 (1.4)

RL2 - IBEF3 66.2 - 31.2 (2.6)
RL2 - ADATE3 100 - 0 (0)
RL2 - RL3 0 - 100 (0)

RL2 - IBEF4 0 - 93.8 (6.2)
RL2 - ADATE4 100 - 0 (0)
RL2 - RL4 25 - 11 (64)

RL2 - IBEF5 27.4 - 67.8 (4.8)
RL2 - ADATE5 100 - 0 (0)
RL2 - RL5 2 - 98 (0)

RL2 - IBEF6 21.7 - 69.1 (9.2)
RL2 - ADATE6 100 - 0 (0)
RL2 - RL6 0 - 100 (0)

*****RL3*****

RL3 - RANDOM 97.8 - 2.1 (0.1)

RL3 - IBEF1 100 – 0 (0)
RL3 - ADATE1 7.3 - 92.7 (0)
RL3 - RL1 63.5 - 0 (36.5)

RL3 - IBEF2 41.2 - 47.3 (11.5)
RL3 - ADATE2 0 - 100 (0)
RL3 - RL2 100 - 0 (0)

RL3 - IBEF3 85.8 - 0 (14.2)
RL3 - ADATE3 0 - 97.2 (2.8)
RL3 - RL3 0 - 100 (0)

RL3 - IBEF4 40 - 48.2 (0)
RL3 - ADATE4 0 - 100 (0)
RL3 - RL4 4 - 73 (23)

RL3 - IBEF5 54.3 - 45.1 (0.6)
RL3 - ADATE5 0 - 100 (0)
RL3 - RL5 12 - 83 (5)

RL3 - IBEF6 50.9 - 49.1 (0)
RL3 - ADATE6 0 - 100 (0)
RL3 - RL6 0 - 100

*****RL4*****

RL4 - RANDOM 99.5 - 0.5 (0)

RL4 - IBEF1 100 - 0 (0)
RL4 - ADATE1 100 - 0 (0)
RL4 - RL1 54 - 26 (20)

RL4 - IBEF2 47.3 - 35.5 (17.2)
RL4 - ADATE2 100 - 0 (0)
RL4 - RL2 60.9 - 8.7 (30.4)

RL4 - IBEF3 77.9 - 22.1 (0)
RL4 - ADATE3 76.3 - 18.7 (5)
RL4 - RL3 90 - 4 (6)

RL4 - IBEF4 75.1 - 24.9 (0)
RL4 - ADATE4 70 - 29 (1)
RL4 - RL4 0 - 100 (0)

RL4 - IBEF5 64 - 34 (2)
RL4 - ADATE5 51 - 30 (19)
RL4 - RL5 18 - 71 (11)

RL4 - IBEF6 85 - 15 (0)
RL4 - ADATE6 90 - 10 (0)
RL4 - RL6 1 - 93 (6)

*****RL5*****

RL5 - RANDOM 100 - 0 (0)

RL5 - IBEF1 100 - 0 (0)
RL5 - ADATE1 76 - 23 (1)
RL5 - RL1 100 - 0 (0)

RL5 - IBEF2 100 - 0 (0)
RL5 - ADATE2 0 - 100 (0)
RL5 - RL2 100 - 0 (0)

RL5 - IBEF3 100 - 0 (0)
RL5 - ADATE3 11 - 81 (8)
RL5 - RL3 100 - 0 (0)

RL5 - IBEF4 82 - 18 (0)
RL5 - ADATE4 16 - 84 (0)
RL5 - RL4 100 - 0 (0)

RL5 - IBEF5 58 - 37 (5)
RL5 - ADATE5 0 - 100 (0)
RL5 - RL5 0 -9 (91)

RL5 - IBEF6 100 - 0 (0)
RL5 - ADATE6 0 - 100 (0)
RL5 - RL6 10 - 87 (3)

*****RL6*****

RL6 - RANDOM 100 - 0 (0)

RL6 - IBEF1 100 - 0 (0)
RL6 - ADATE1 100 - 0 (0)
RL6 - RL1 100 - 0 (0)

RL6 - IBEF2 100 - 0 (0)
RL6 - ADATE2 0 – 100 (0)
RL6 - RL2 100 - 0 (0)

RL6 - IBEF3 79 - 21 (0)
RL6 - ADATE3 0 - 100 (0)
RL6 - RL3 83 - 17 (0)

RL6 - IBEF4 73 - 27 (0)
RL6 - ADATE4 0 - 100 (0)
RL6 - RL4 100 - 0

RL6 - IBEF5 64 - 34 (2)
RL6 - ADATE5 0 - 100 (0)
RL6 - RL5 96 - 0 (4)

RL6 - IBEF6 2 - 98 (0)
RL6 - ADATE6 91 - 9 (0)
RL6 - RL6 100 - 0 (0)

*****RANDOM*****

RANDOM - RANDOM 55.9 - 43.9 (0.2)

RANDOM - IBEF1 9.7 - 90.3 (0)
RANDOM - IBEF2 3.1 - 96.9 (0)
RANDOM - IBEF3 1.6 - 98.4 (0)
RANDOM - IBEF4 0 - 100 (0)
RANDOM - IBEF5 0 - 100 (0)
RANDOM - IBEF6 0 - 100 (0)

RANDOM - ADATE1 34.7 - 65.3 (0)
RANDOM - ADATE2 2.7 - 97.3 (0)
RANDOM - ADATE3 2 - 97.9 (0.1)
RANDOM - ADATE4 0.5 - 99.5 (0)
RANDOM - ADATE5 0.6 - 99.4 (0)
RANDOM - ADATE6 0.1 - 99.9 (0)

RANDOM - RL1 13.1 - 86.9 (0)
RANDOM - RL2 4.9 - 95 (0.1)
RANDOM - RL3 3.3 - 96.7 (0)
RANDOM - RL4 2 - 98 (0)
RANDOM - RL5 1 - 99 (0)
RANDOM - RL6 0 - 100 (0)