UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Aleksei Loos

Machine Learning for k-in-a-row Type Games Using Random Forest and Genetic Algorithm

Master's Thesis (30 EAP)

Supervisor: prof. Jaak Vilo

Author: ……..……..…….. "……" May 2012

Supervisor: ……..……..…….. "……" May 2012

Approved for defence:

Professor: ……..……..…….. "……" May 2012

TARTU 2012

Table of contents

Introduction

The field of computer intelligence in common games has been studied since the beginning of computer science. One of the notable achievement examples is computer program Deeper Blue designed by IBM that was able to defeat world chess champion in 1997 [1]. Such computer programs are mostly based on designing algorithms that use human expertise as their knowledge base – the important game patterns, situations, threats and defenses are all predefined and provided by humans. In 1995 it was demonstrated by D. Fogel that it is possible to train computer intelligence also without any human expertise. This was done with the help of neural networks and evolutionary learning on a simple Tic-tac-toe game [2]. This method was further explored in 1997 where the AI program Blondie24 was trained to play checkers and achieved expert level results against human players online [3]. The Blondie24 project was also inspiration for this given work, but Random Forests are used instead of neural networks.

The main objective of the thesis is to explore the viability of combination multiple machine learning techniques in order to train Artificial Intelligence for k-in-a-row type games. The techniques under observation are following:

- Decision Trees [4]
- Random Forest (consisting of Decision Trees) [5]
- Minimax Algorithm [6]
- Genetic Algorithm [7]

The main engine for training AI is Genetic Algorithm where a set of individuals are evolved towards better playing computer intelligence. In the evaluation step, series of games are done where individuals compete in series of games against each other – the results are recorded and the evaluation score of the individuals are based on their performance in the games. During a game, heuristic game tree search algorithm Minimax is used as player move advisor. Each of the competing individuals has a Random Forest attached that is used as the heuristic function in Minimax. The key idea of the training is to evolve as good Random Forests as possible.

During the first chapter of the thesis, an overview of games used is given, along with their rules and theoretical complexity. A more detailed explanation of algorithms and their procedures is given in Chapter 2. How the algorithms are implemented and combined is explained in Chapter 3. To speed up the training process multiple optimizations were done as described in Chapter 4. Achieved results are given in Chapter 5.

1 K-in-a-row games

The family of connecting games has a long history and has been studied in computer science comprehensively [8], [9], [10]. The connection games are played on grid-like boards between two players who are altering moves. The general idea is to place stones on the game board and achieve a consecutive line of *k* stones either vertically, horizontally or diagonally. The most famous and simple connection game known is Tic-tac-toe. Such games have generalized definition of (*m*,*n*,*k*)-games where *m* and *n* represent the size of the board and *k* represents number of consecutive stones needed in a line to win a game. Tic-tac-toe represents a (*3*,*3*,*3*)-game by the definition. Recently, around 2004, a further extension was introduced to (*m*,*n*,*k*)-games – as addition, each player places *p* stones each turn with exception of first turn where player places *q* stones. Such games are defined as (*m*,*n*,*k*,*p*,*q*)-games. This extension was required to define now quite popular worldwide game Connect6 with the definition of (*m*,*n*,*6*,*2*,*1*)-game.

1.1 Gomoku

Gomoku is a another classic example of *k-in-a-row* type games that was played typically with Go [11] board and game pieces. The aim of the game is to achieve five consecutive stones in a row on a standard Gomoku board size of 15x15 grid. The name originated from Japanese language where the game is called *gomokunarabe* - *go* means five, *moku* is a counter word for pieces and *narabe* means line-up.

State space complexity of the game is 3^{225} and game tree complexity is approximately 10^{70}, assuming an average game length of 30 moves [9]. One of the earlier successful implementations for the game is "Vertex" which achieved gold medal during ICGA Tournaments in 1992. It used Alpha-Beta game tree search with depth of 16 plies where in each ply only the best 14 moves were searched deeper. Further study by L.V.Allis in 1994 has shown Gomoku to be solved game where first player wins with perfect game [9]. A combination of threat-space search, proof-number search and database construction was used in order to show this.

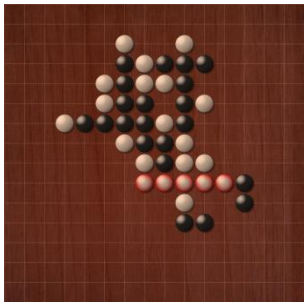

Figure 1-1: Example Gomoku board

During this project, a Gomoku with board size of 10x10 will be used in order to explore the viability and performance of proposed thesis subject. This reduces the state space com-

plexity to 3^{100} and game tree complexity to approximately 10^{59} while still giving good overview of the method effectiveness.

1.2 Connect Four

Connect Four is yet another connection game where 4 stones have to be placed in a row. It has additional move restriction rule, where gravity applies to each move – once a stone has been placed, it will be sent to the lowest possible position currently on board. Connect Four is considered as a solved game on a classical 6x7 board [12]:

- If player starts in the middle, he can force a win
- If player starts in one of the edge positions, opponent can force a win
- If player starts in any other position, opponent can force a draw

The game has quite low game tree complexity compared with Gomoku as there are always at maximum of 7 different positions to move. A comparison between game tree complexities used during this work is illustrated in [Figure 1-2].

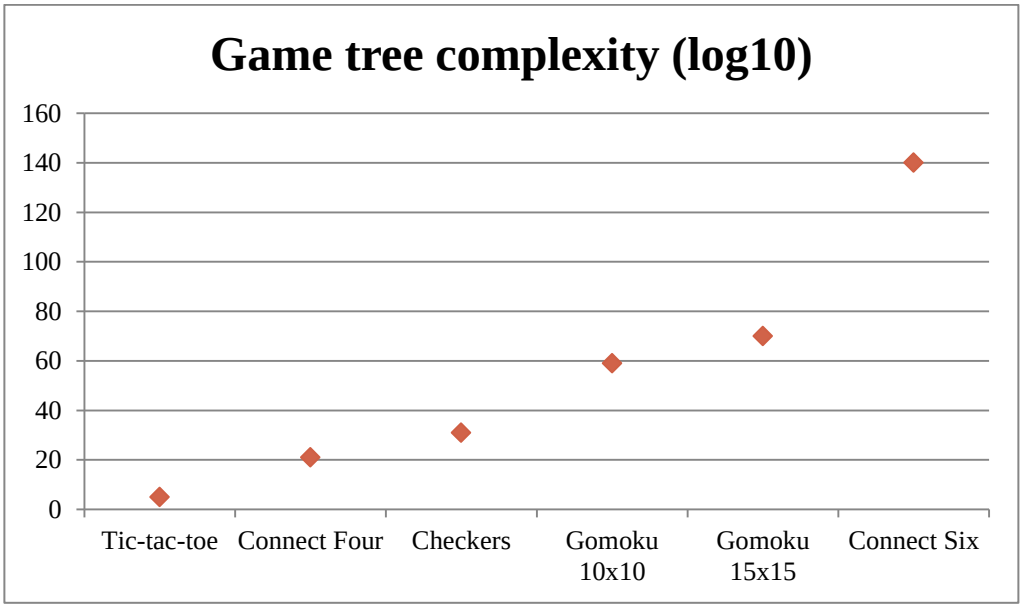

Figure 1-2: Game tree complexity

2 Algorithms used

2.1 Minimax

Minimax algorithm is a recursive search algorithm that traverses on a game tree and tries to minimize maximum possible loss. It is widely used in turn-taking, two-player, zero-sum games with perfect information [13]. This means that the there is no hidden information and the game states are always opposite – if one player wins, then the other player automatically loses. Common examples of such games are checkers, chess and k-in-a-row games (e.g. Tic-tac-toe). The algorithm uses depth-first approach to recursively scan all possible game states until all terminal nodes are found or a certain depth has achieved. Such depth is often defined as *look-ahead* number of turns where is each depth level is referred to as *ply*. In game tree such nodes are considered to be tree leaves and when a leaf-node is found it will be evaluated according to a heuristic function. When a game state is found where the player wins, it is evaluated automatically with positive infinity and with negative infinity if it is a loss for the player. All other non-terminal states are evaluated based on heuristic function giving estimation of how good the game state is in perspective of the player. During this work, Random Forests are used as the heuristic function.

The input of the algorithm is a game state from where all possible next game states are expanded and the output is the move with maximum evaluation score that player is guaranteed to achieve from the input state. Each ply of the tree alternates between MAX levels, where the goal is to benefit the player maximizing the evaluated score of a game state, and MIN levels, where the goal is to benefit the opponent by minimizing the evaluated score of a game state [14]. The algorithm relies on accuracy of heuristic function and on assumption that opponent does not make any mistakes. The pseudocode of the algorithm is described in [Pseudocode 2-1].

Minimax pseudocode

```
function minimax( node, depth )
    if node is a terminal node or depth <= 0:
        return the heuristic value of node

    α = -∞
    foreach child in node:
        α = max( a, -minimax( child, depth-1 ) )

    return α
```

Pseudocode 2-1: Minimax

The provided pseudocode is also enhanced by Negamax code optimization [14]. Normal Minimax pseudocode would require two separate code blocks, first for minimizing and second for maximizing. Negamax takes into account that $max(a,b) = -min(-a,-b)$ and seeks always from perspective of currently acting player by maximizing the negative values of sub game state evaluations. The outcome of Negamax does not differ in any way compared with Minimax and the only aim is to reduce the implementation code footprint.

An example fictional game tree is illustrated in [Figure 2-1] where all nodes have been already fully analyzed. The nodes that lead to player victory are evaluated with positive infinity and nodes that lead to opponent victory are evaluated with negative infinity. Maximizing player is represented by circles (who chooses best evaluated move from sub-node values) and minimizing player is represented by squares (who chooses worst evaluated sub-node values). Starting from bottom-to-top it is possible to see that the smallest guaranteed evaluation score from left sub-tree is -10 and from right sub-tree it is -7. So in order to achieve guaranteed game board state of -7, player should choose the move represented by right sub-tree.

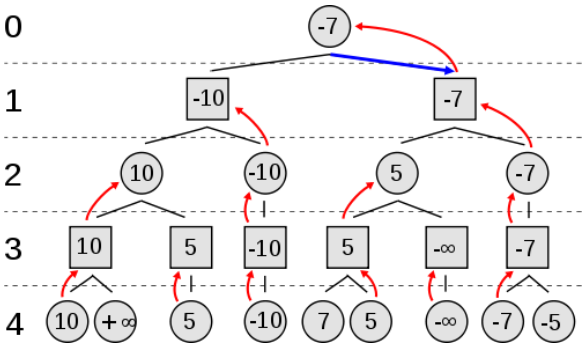

Figure 2-1: Example Minimax game tree [15]

The naive Minimax algorithm can be further improved with Alpha-Beta pruning [13]. Not all states of the game tree have to be examined during the game tree search – if a game state is found that is worse than previously found state, then it can be completely discarded as it has no way of improving maximum possible result. On average, discarding such states reduces the game tree branching factor by 25%, thus also improving the algorithm speed. Additional speed-up can be achieved if more likely better child-nodes are explored first. This can be done by simply sorting child nodes before next level recursion.

To implement Alpha-Beta pruning, the algorithm is extended with two additional variables (α and β) that pass information recursively to next levels. Alpha keeps track of minimum value that the player can achieve (initiated with negative infinity) and beta keeps track of maximum value that player can achieve (initiated with maximum infinity). During the search

process, these values are modified based on the heuristic function evaluation for game states. The pseudocode for the Alpha-Beta pruning is described in [Pseudocode 2-2].

Alpha-Beta pruning pseudocode

```
function negamax( node, depth, α, β, color )

    if node is a terminal node or depth = 0
        return color * the heuristic value of node

    foreach child of node
        value = -negamax( child, depth-1, -β, -α, -color )

        if value ≥ β
            return value /** Alpha-Beta cut-off */

        if value ≥ α
            α = value

    return α
```

Pseudocode 2-2: Alpha-Beta pruning

The provided pseudocode is enhanced again with Negamax code optimization, to reduce the code footprint and the result does not differ from regular Alpha-Beta pruning.

2.2 Decision Tree

In machine learning Decision Trees are used as models to predict an item target value based on observed attributes from given dataset. More commonly such trees are called classification and regression trees (CART) and they map observations about an item to conclusions. The structure of Decision Tree consists of tree nodes and tree leaves. Tree nodes represent a single attribute on a data item and tree leaves represent the prediction value. An example Decision Tree is illustrated in [Figure 2-2] which describes survival of passengers on the Titanic. For example, if a person on the ship was male under 9.5 years old, had more than 2.5 siblings or spouses on board (represented by *sibsp* parameter), then this person had 89% of survival rate. Total of 2% observations are represented by this tree leaf.

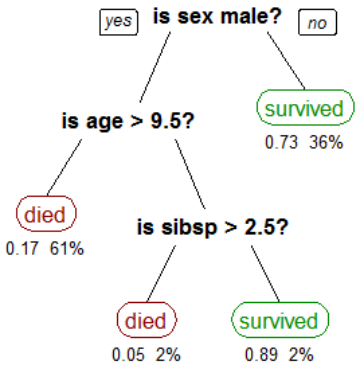

Figure 2-2: Survival rate of Titanic[16]

The main idea of Decision Tree is to split observed information into subsets based on single attributes. A common simple tree learning algorithm is ID3 that splits data into subsets starting from attributes with best information gain based on entropy that is used in information theory. Such recursive partitioning is done until subset under a tree node has the same prediction value or if the splitting gives no more statistical value to the predictions.

There are many advantages for Decision Trees such as:

- Simple to understand and interpret – white box models
- Requires little data preparation
- Able to handle both numerical and categorical data
- Robust and it is possible to validate their effectiveness using statistical tests
- Can model large data sets without performance issues.

But there are also disadvantages such as:

- Constructing optimal Decision Trees is known to be NP-hard problem
- Decision Trees can fail to generalize data correctly (overfit) – this problem can be overcome with pruning techniques

2.3 Random Forest

Random Forest is an ensemble classifier that uses multiple Decision Trees as described in the previous chapter. The idea is to collect target value prediction information from many Decision Trees and the mean of all trees would be the output from Random Forest. Each of the trees in such forest has different subset of decision attributes on which the tree makes the prediction. The overall procedure for training a Random Forest is as follows:

1. Let N be the number of trees to be learned and M total number of variables; the process is repeated until required number of trees have been learned
2. A bootstrap sample from the training set is chosen randomly
3. A Decision Tree is learned on the chosen bootstrap
4. At each node of the tree, a random subset of m input variables are chosen that are used to determine decision at a node of the tree (m should be much smaller than M) and the tree is split based on the best variable
5. The resulted tree is added to random forest next to other already trained trees
6. Examples not chosen into bootstrap set are used to estimate the error of the trained tree and evaluating the prediction value

7. The resulted tree is not pruned and kept as is.

There are many advantages for Random Forest compared to a single Decision Tree such as:

- It is one of the most accurate classifier algorithms currently known [17]
- Robust to outliers and noise
- Has high performance on large dataset size and large number of variables

With two main disadvantages:

- Random Forests have been observed to overfit when learning on some noisy datasets [18]
- Compared to Decision Trees, they are harder for human to interpret

Random Forests were chosen for Minimax algorithm heuristic function as an alternative option to Artificial Neural Networks (as used in Blondie24 for example [3]). Also, it is possible to convert already trained networks into form of Decision Trees [19], which are the core of Random Forests.

2.4 Genetic Algorithm

The Minimax algorithm relies heavily on the heuristic function which is used to evaluate game states. Such heuristic functions are in most cases developed with help of human expert knowledge. With inaccurate evaluation, the game tree search becomes inefficient and does not provide good move suggestions. During this thesis, the process of finding and training accurate Random Forests for Minimax heuristic function is obtained with help of Genetic Algorithm. This allows creating a heuristic function from scratch without any human expert knowledge, which is also one of the aims to achieve during this thesis.

Genetic Algorithm is a search heuristic that mimics natural evolution process. The aim of the algorithm is to evolve towards a solution by going through same steps as observed in nature. The algorithm is initiated with a completely random population where each individual is a possible solution candidate. Once such population is generated, the algorithm starts with evaluation where each individual is assigned a fitness score – how good they are in solving the problem. After this, natural selection occurs similar to the 'survival of the fittest' in nature where only strongest individuals are transferred to next step: reproduction. Each individual has genetic information that encodes solution information, for example binary arrays

('0101100100') or strings ('UAGACGGAG'). During the reproduction step, two individuals are selected from the survived population and go through crossover process – again, mimicking nature, new child individual is formed from combination of genetic information in both parents. There is also a small probability that newly formed individual will go through mutation, where the genetic information is slightly altered. The objective of the mutation in terms of search heuristic is to overcome local optima and help find a better solution. After new population is generated from crossover step, the evaluation and new iteration of the algorithm begins. Overview of the Genetic Algorithm procedure is following:

1. Initial population
2. Evaluation
3. Fitness assignment
4. Natural selection
5. Reproduction
6. Crossover
7. Mutation
8. Repeat from step 2.

After each iteration the individuals evolve into better solution and the algorithm stops once a satisfactory result solution is achieved or predefined number of iterations has passed. There can be also other end criteria such as manual observation or filling allocated timeslot.

3 Implementation overview

The implemented version of the software consists around 9000 lines of code. This includes implementation of Genetic Algorithm, Random Forest, Decision Trees and Minimax algorithms. There are many threats when writing software program in short timespan with volume such size – software errors are inevitable, which is backed up by the fact that during the testing and training period numerous defects were found (and hopefully correctly fixed). Yet there could be always more still undetected issues that could potentially bias the training results [20].

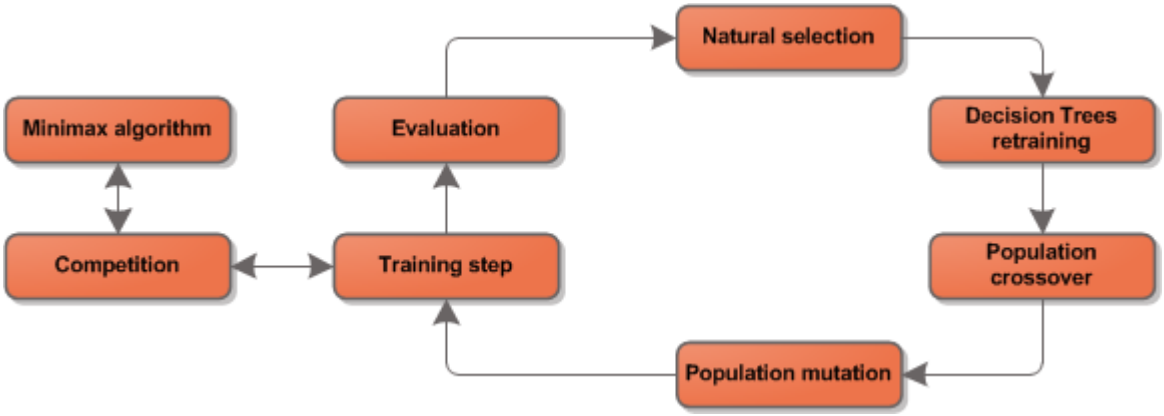

Figure 3-1: Overall training workflow

The overall training procedure is illustrated in [Figure 3-1] and it is separated mainly into two main procedures: training step and genetic step. These steps and other different parts of the whole process are described during this chapter.

3.1 Training step

The training process starts always with Genetic Algorithm step – new population for training is created there and the individuals from this population are then added to competition consisting of games where they compete against each other. In each game Minimax algorithm is used based on Random Forest attached to each individual. There are two strategies that can be used to compete individuals against each other: full and limited. With full training, each individual is competed against every other individual. With this strategy the number of games that have to be played has quadratic growth based on the population size. Full training strategy gives the best evaluation information for the individuals.

With the second, limited strategy, it is possible to define number of games that will be done for each individual. This allows reducing number of games to a smaller (linear) growth.

The idea is to compete survived population from previous iteration against new population generated from mating process in Genetic Algorithm. This method does not give as precise estimation as full training, but still gives good estimation for an individual if it had better performance than one of its predecessors – better performing individuals will survive by replacing their predecessors, thus making population stronger.

Once all games are finished, the results are analyzed based on each game outcome and each individual is then given predefined performance score, with default values as follows:[1]

- 2 points for each victory
- 1 points for each draw
- -3 points for each loss

The point values are configurable, but seem to suit for population size of 5 to 10. It was possible to observe more efficient evolving of individuals if points reduced by a loss was greater than points given for a victory. Otherwise some individuals would "sacrifice" some games in order to get more victory points.

3.2 Genetic Algorithm step

Genetic Algorithm has two separate parts implemented called Brain and Knowledge. The Brain conducts the process of Genetic Algorithm - that is natural selection, population retraining and mating. The Knowledge on the other hand has implementation of algorithms that process the data or delegates tasks to other parts of the project. For example the retraining process is done separately in order it would be possible to use multithreading.

As a first step in Genetic Algorithm, natural selection takes place. The process takes input as already evaluated population from previous training step and returns only best performed individuals during the competitions. All other individuals are completely discarded.

Once the best individuals are selected the retraining process starts. During this step all Decision Trees are extracted from all Random Forests attached to each survived individual. Then each extracted tree is retrained based on the Decision Tree algorithm, but using the same decision attributes as previously. The retraining process is important, since after each competition step, there is new available information stored from results of each game. Retraining process takes this new information into account and retrained Decision Trees will have potentially better knowledge.

[1] No claim of optimal settings is done here – the chosen values were solely chosen as a result from series of mini tests and trainings.

After best individuals are retrained, the mating and crossover process starts. There are two possible strategies for mating process: full and partial. In full mating, each individual goes through crossover against each other individual. With this strategy the number of maximum individuals has quadratic growth based on input population size. The second strategy is to use partial mating where crossover is done against two individuals selected randomly from the full population until configured maximum population size is reached. This allows limit the quadratic population size growth to a linear size growth and reduces overall training time significantly. An example comparison between full and limited training is shown in [Table 3-1].

	Full training and crossover	Limited training and crossover
Initial population	10	10
Maximum population	100	20
Number of total games	9,900	200

Table 3-1: Full and limited training comparison

During the crossover process, genes are considered to be the trees in the Random Forest for each individual. New individual is created by using randomly chosen trees from both parents. The resulted new forest has the same amount of trees as their parents and they are attached to the new child individual.

During the crossover process there is also a predefined probability of mutation. During the mutation process the genes are again considered to be the trees in the Random Forest. If a child individual is mutated, then for each tree in the forest there is the same predefined probability that the tree is discarded and a new tree is trained instead based on the Decision Tree algorithm. This ensures that new potentially better decision attributes (game board patterns) are introduced into the population while some potentially less efficient are removed.

3.3 Minimax usage

The implementation of Minimax follows the general definition of the algorithm, with addition of Alpha-Beta pruning and Negamax code optimization as described in Chapter 2.1. There are also few modifications to the algorithm pseudo code – all possible next game board states are pre-calculated and sorted based on board evaluation heuristic before the game tree search begins. This ensures that more likely better subtrees are searched first and giving higher chance of earlier Alpa-Beta cutting.

The heuristic that evaluates board states for sorting is quite simple: a game board is scanned for multiple predefined patterns and each of the patterns has also assigned score value. When a pattern is found in the game board state, the overall board score is either added or subtracted based on for which player the pattern was found. The patterns that are scanned can be seen on [Figure 3-2], where each pattern is visualized from left to right, one pattern per line: cross symbolizes player stone and dash an empty field. These patterns are also mirrored vertically and diagonally. The scanning process is using Boyer-Moore-Horspool [21] pattern search algorithm and board is scanned one pattern at a time. A possible improvement would be to implement multi-pattern search algorithm here so all patterns could be scanned at the same time, thus also improving the performance. This heuristic is used only for sorting the boards for Alpha-Beta cutting process and does not affect Minimax algorithm output. The actual board evaluation for Minimax algorithm comes from Random Forest output.

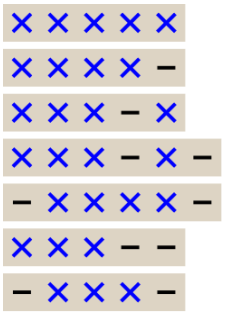

Figure 3-2: Alpha-Beta sorting heuristic patterns

3.4 Random Forest usage

By definition the Random Forest algorithm classification output is mean of all Decision Trees attached to the forest. This output would then be used as heuristic function of a board state during Minimax algorithm and using such method would have only three possible categorization: player A wins, player B wins, players tie. This gives very little information as in most cases, mean classification from trees would be "player does not win". This is caused by the procedure how boards are evaluated – each tree scans all possible locations on the game board (also rotated and mirrored) and it actually makes sense that it finds most scanned locations as non-winning.

Due to nature of this project, the output is collected and interpreted differently - instead of mean classification from trees, all information from all trees categorization is collected and used to evaluate a single game board state. The key idea is to find out how many locations of the board were found as:

- Winning location
- Losing location
- Draw location

Once such information is found, the values can be normalized and then used to calculate board evaluation. Normalization is required for Minimax algorithm so all boards would have similar score. The difference comes from number of total classifications – when a tree scans board locations and all the decision fields are empty, the classification is ignored. Final board score would then be calculated based on proportions of the classifications multiplied by the prediction weight as follows:

$$\boldsymbol{win\ prediction} = \frac{win\ predictions\ * 100}{total\ predictions} * win\ weight$$

$$\boldsymbol{loss\ prediction} = \frac{loss\ predictions\ * 100}{total\ predictions} * loss\ weight$$

$$\boldsymbol{draw\ prediction} = \frac{draw\ predictions\ * 100}{total\ predictions} * draw\ weight$$

$$\boldsymbol{board\ evaluation\ score} = win\ prediction + loss\ prediction + draw\ prediction$$

Default weights that were used during this project are listed in [Table 3-2]. Using all information provided from the Decision Trees with combination of provided formulas, the heuristic algorithm becomes more informative and offers more than just three possible target values.[2]

Prediction type	Weight
Win prediction weight	2.0
Draw prediction weight	1.0
Loss prediction weight	-0.5

Table 3-2: Default weights for predictions

3.5 Random Forest training

The implementation of Random Forest training follows general definition of the algorithm, with one notable difference: an extra level of Decision Tree abstraction is added, called 'Random Woods' as seen in [Figure 3-3]. This abstraction replaces Decision Trees by definition and consists itself of many Decision Trees. The key difference between Decision Tree and Random Woods is that woods consist of the same Decision Tree that is replicated and transformed over whole game board. The aim of Random Woods is to cover all possible locations on the board with the same single pattern from the Decision Tree when they are used during Minimax board evaluation process. This is a key feature when the size of defined game board grows. Such feature is not needed on smaller boards, for example Tic-tac-toe or even for game boards where width is smaller than 5~7 fields – a single Decision Tree could already cover all possible locations in this case.

[2] No claim of optimal settings is done here – the chosen values were solely chosen as a result from series of mini tests and trainings.

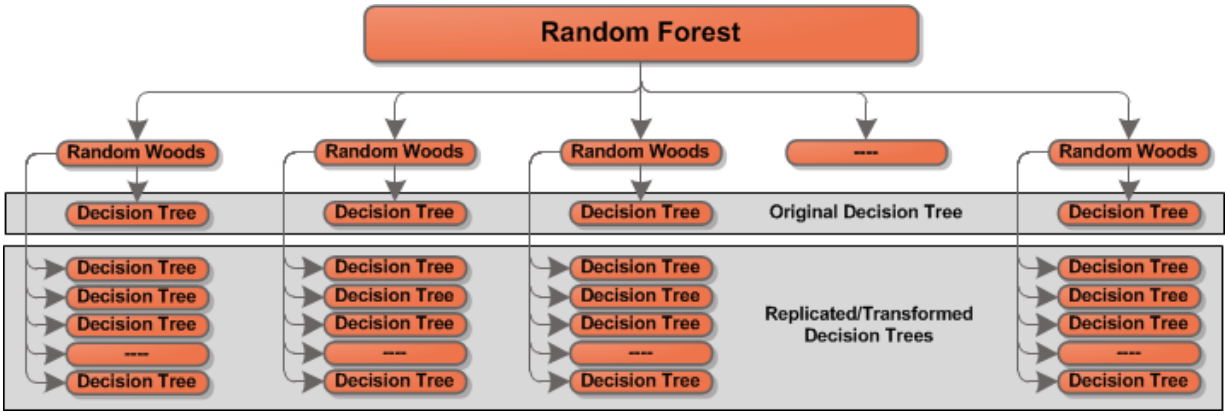

Figure 3-3: Random Forest with Random Woods abstraction

When the first Decision Tree is trained with random decision attributes it is attached to the Random Woods. After this, the trained Decision Tree is replicated over all possible game board locations. In addition, following operations are performed to the tree decision attributes pattern:

- Mirroring vertically
- Mirroring horizontally
- Mirroring vertically and horizontally
- Transposed
- Transposed with all mirroring operations again

Once such transformations are completed, they are propagated into Decision Tree structure and then are attached to corresponding Random Woods. Essentially these Decision Trees are deciding on same features, but on different locations of the board. Detailed example how trees are transformed is shown in [Table 3-3] as a minimum example with pattern radius of 4 and target board size 5x5 fields. For each such pattern a separate Decision Tree is formed. This ensures that Random Forest will be able to analyze all features in all possible locations on the board and report the result accordingly.

One of the main advantages of pattern separation to cover whole game board on Decision Tree level is possibility to add caching to the evaluation process. This is discussed in more details during Chapter 4.3.

Source	Replication - Original			
- - X X - - - - - X - X X - - -	- - X X - - - - - - - X - X - X - - - - - - - - -	- - - - - - - X X - - - - - - - X - X - X - - - -	- - - X X - - - - - - - X - X - X - - - - - - - -	- - - - - - - - X X - - - - - - - X - X - X - - -
	Replication – Mirrored horizontally			
X - - - - X - X - - - - - - X X	X - - - - - X - X - - - - - - - - X X - - - - - -	- - - - - X - - - - - X - X - - - - - - - - X X -	- X - - - - - X - X - - - - - - - - X X - - - - -	- - - - - - X - - - - - X - X - - - - - - - - X X
	Replication – Mirrored vertically			
X X - - - - - - X - X - - - - X	X X - - - - - - - - X - X - - - - - X - - - - - -	- - - - - X X - - - - - - - - X - X - - - - - X -	- X X - - - - - - - - X - X - - - - - X - - - - -	- - - - - - X X - - - - - - - - X - X - - - - - X
	Replication – Mirrored horizontally and vertically			
- - - X X - X - - - - - X X - -	- - - X - X - X - - - - - - - X X - - - - - - - -	- - - - - - - - X - X - X - - - - - - - X X - - -	- - - - X - X - X - - - - - - - X X - - - - - - -	- - - - - - - - - X - X - X - - - - - - - X X - -
	Replication - Transposed			
- - - X - - X - X - - - X - X -	- - - X - X - X - - - - - - - X X - - - - - - - -	- - - - - - - - X - X - X - - - - - - - X X - - -	- - - - X - X - X - - - - - - - X X - - - - - - -	- - - - - - - - - X - X - X - - - - - - - X X - -
	Replication – Transposed and mirrored horizontally			
X - X - X - - - - - X - - - - X	X - X - - X - - - - - - X - - - - - X - - - - - -	- - - - - X - X - - X - - - - - - X - - - - - X -	- X - X - - X - - - - - - X - - - - - X - - - - -	- - - - - - X - X - - X - - - - - - X - - - - - X
	Replication – Transposed and mirrored vertically			
X - - - - X - - - - - X - X - X	X - - - - - X - - - - - - X - - X - X - - - - - -	- - - - - X - - - - - X - - - - - - X - - X - X -	- X - - - - - X - - - - - - X - - X - X - - - - -	- - - - - - X - - - - - X - - - - - - X - - X - X
	Replication – Transposed, mirrored horizontally and vertically			
- X - X - - - X - X - - X - - -	- X - X - - - - X - - X - - - X - - - - - - - - -	- - - - - - X - X - - - - X - - X - - - X - - - -	- - X - X - - - - X - - X - - - X - - - - - - - -	- - - - - - - X - X - - - - X - - X - - - X - - -

Table 3-3: Pattern transformation and replication

3.6 Decision Tree learning

In training step when a game between individuals ends, the result state is stored in the database. This includes following key information:

- Winning player
- Game board field values around last move performed by player

The winning player information is used as the Decision Tree prediction task based on the game field states. The game fields do not include full game board information, instead during

Decision Tree learning process, only near game end fields are explored and analyzed. This ensures that Decision Tree learns more relevant information what helped to end the game. Also such method reduces possible number of Decision Tree attribute combinations and fastens the Genetic Algorithm in process of finding better patterns. This process is illustrated in [Figure 3-4] and goes through following steps:

1) Winning move is done by a player – last move location is memorized
2) The surrounding fields of last move location are extracted
3) Fields are transformed to linear array form
4) Winning player information is added
5) Field values are converted to byte data type
6) Array is stored in database

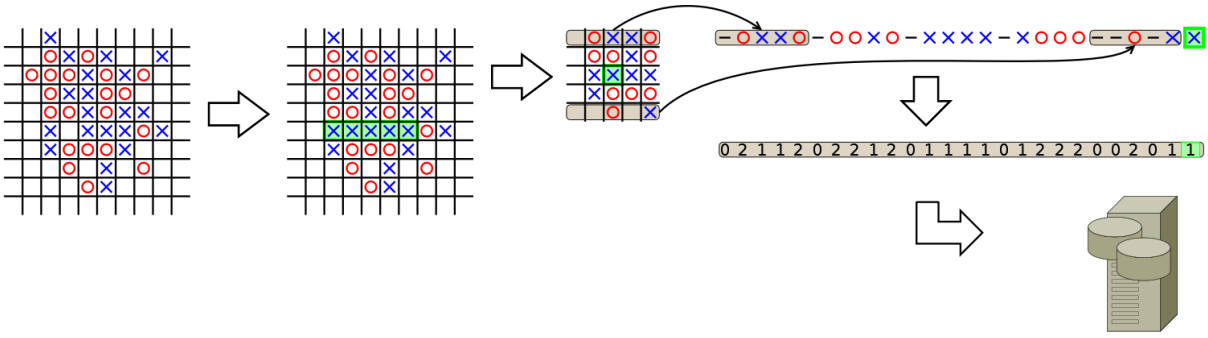

Figure 3-4: Game board state extraction

After each training step iteration, there is more information stored in database than previously for Decision Tree learning. This information is used during Genetic Algorithm retraining step (all trees are retrained with same decision attributes) and during mutation step (mutated trees are trained with new random decision attributes).

Initial information for Decision Tree learning is generated with help of games between random moving players and simple heuristic players. This ensures that Decision Trees have some sort of starting point to quick start the whole training process. The simple heuristic player uses same heuristic function as Alpha-Beta pruning method to sort board states. Such player has no chance to win against Minimax based player, but is good enough to beat randomly moving players.

4 Performance optimization

Multiple steps of the project are multithread oriented. Currently, following steps have been implemented to work in parallel in multiple threads:

- Training step process
- Minimax algorithm
- Retraining process during Genetic Algorithm

To achieve multithreading, Java native API was used. A thread pool with fixed number of threads is created with help of Executor class – one thread per single task. If there are more tasks than the thread pool size, then they are added to queue and will be executed one-by-one once a thread is released back into the pool after previous task has finished. This queue is managed by ExecutorService class.

4.1 Threaded training step process

During the training step, one of the main sub processes is the competition between individuals where games are taken place between them. This process is most time consuming in the whole project and takes easily up to 95% of the whole training time. Implementing multithreaded competition is very straight forward – one game between two individuals is considered as one task and has full thread to use from the thread pool. Once the game is finished, the game state/result is stored and the used thread is released so a next game in queue can start. The situation where there are fewer games on going than number of usable threads is covered by threaded Minimax, discussed in next chapter.

4.2 Threaded Minimax

One of the problems with threaded training step process is the situation where there are fewer games ongoing than the number of usable threads. This kind of situation can increase training step time in some cases twice, for example when the last game in queue will take long time due to even opponents who are unable to win quickly. In order to still exploit free usable threads this project implemented basic threaded Minimax [22]. This is achieved by splitting top level game tree search into sub processes.

Threaded Minimax is slower since alpha-beta cutting is not as efficient as non-threaded version. This is caused by multiple threads searching in game tree at the same time – if some thread finds a good alpha-beta value to perform cutting, then this value will not be propagated

into other on-going searches. These threads will still perform game tree search based on the initial alpha-beta when the search started. Although, once a sub game tree search has finished, then a better alpha-beta value is stored and any new starting threads will request and continue their search based on these values.

Since using threaded Minimax is slower, it is not useful to use it at all times. A Minimax-Manager class was implemented that follows two key patterns:

- Singleton pattern
- Observer pattern

The main objective of the manager is to decide when to use regular and when to use threaded Minimax. For this, MinimaxManager is added as an observer to training process step. The training process task is to update the manager with currently running number of games. When manager detects, that there are less running games than number of available threads, it returns instance of threaded Minimax, otherwise just regular Minimax. Once a threaded Minimax instance is created, the search process starts. The possible number of concurrently ongoing searches is also taken from MinimaxManager as it has knowledge of currently running games and free available threads to use. This means that number of searches at the same time can increase during the Minimax algorithm when another game has finished and a new thread is released for usage. There is also a constraint that only one threaded Minimax can be running at the same time. This is done with help of synchro-nous Threaded Minimax activation between threads and ensures that there is only one task per thread.

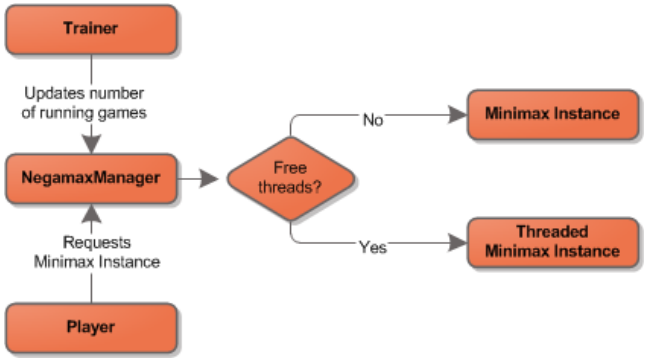

Figure 4-1: Threaded Minimax usage

4.3 Cached Decision Tree

One of the best performance gains was achieved by creating cached Decision Trees. Nor-mal Decision Tree takes a game state as an input and returns result based on the decision attributes. Cached tree on the other hand takes also game state as an input, but instead of traversing tree structure, a hash lookup table is used to store and retrieve cached results.

Building such hash lookup table is quite simple – after training a Decision Tree, we can pre-calculate all possible combinations, feed them as an input to the tree and store the results. There are three possible single field states (empty, player, opponent) so memory consumption of such method grows in n^3, where n is number of decision attributes. Structural difference between regular and cached usage of Decision Trees is illustrated in [Figure 4-2].

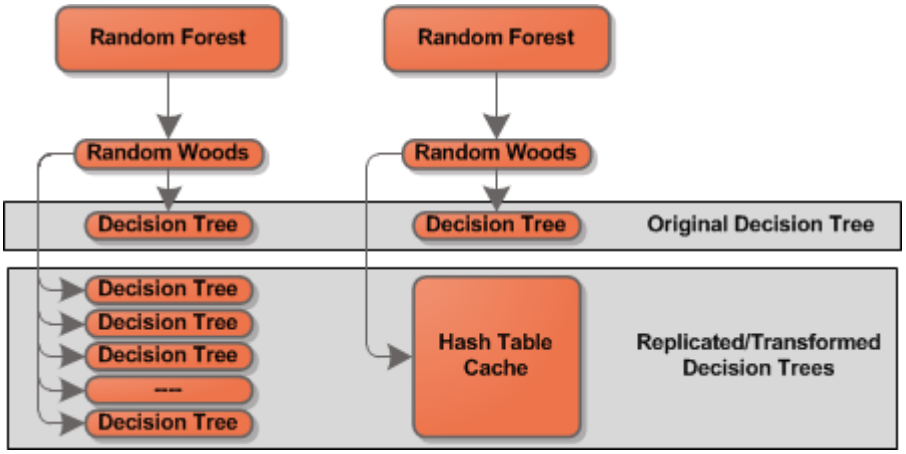

Figure 4-2: Normal and cached Random Woods

Using cached Decision Trees improved the performance of evaluation speed about 5 times on average, thus also improving performance of Minimax algorithm and overall training time.

4.4 Game tree branching limitation

In order to fasten the training process, another optimization was achieved by limiting number of possible moves where a player using Minimax algorithm can move. This essentially reduces game tree branching factor and lowers the game tree exponential growth. One of the options for limiting moves is allowing moves only within certain radius of already done moves. An example move limitation with radius 2 is illustrated in [Figure 4-3] where green marks legal moves positions and red marks moves that are not explored during the Minimax algorithm. Such limitation lowers the number of legal moves while still keeping most (if not all) of the strategical aspects.

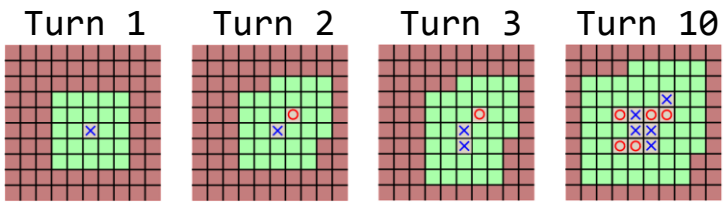

Figure 4-3: Limiting legal moves

5 Results

The main outcome of the thesis is the implemented software to train and evolve Random Forests as described in Chapter 2.4. Also multiple forests were trained in order to get overview how effective this process is. The most amount of time was spent to train Gomoku-playing Random Forest on a 10x10 game board. The resulted forests were competed against each other and three chosen instances went through series of games against human players.

To improve player gaming experience a minimal graphical user interface was created alongside with command line interface in order to conduct human tests on server side. Example screenshots of both interfaces can be seen in [Figure 5-1]. The CLI was colorized with help of ANSI color codes to indicate human player (always blue) and opponent (always red) colors. Also last move done by computer AI is indicated by green color.

Command line user interface **Graphical UI**

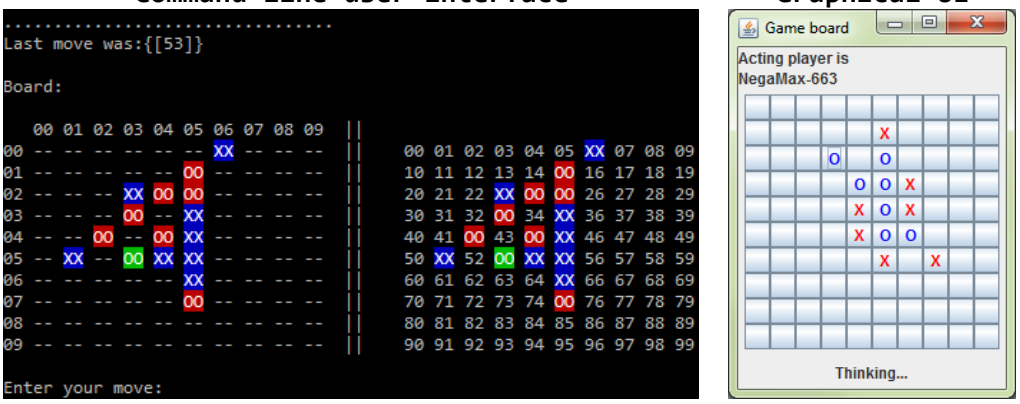

Figure 5-1: User interfaces

5.1 Training environments

All of the trainings were conducted in four different locations as follows:

1. Author's personal computer
   - CPU: AMD Phenom II X4 955 @ 3.20 GHz
   - 4 cores/4 threads
   - 8GB of RAM
   - Usage: Development and series of mini-tests, also training Tic-tac-toe
2. High Performance Computing Center of University of Tartu
   - 2x Intel Xeon Quad-core L5420 @ 2.5 GHz
   - 8 cores / 8 threads each, total of 16 cores / 16 threads
   - 32GB of RAM

- Usage: Tuning software and series of mini-tests

3. High Performance Computing Center of University of Tartu (BIIT/EXCS servers)
   - 8x Intel Xeon E7 2860 @ 2.27 GHz
   - 10 cores / 20 threads each, total of 80 cores / 160 threads
   - 1024 GB of RAM
   - Usage: Connect Four training and series of human tests against AI
     Training of Tic-tac-toe
4. Additional 8 separate servers with virtual machines were used
   - Intel Xeon E7 2830 @ 2.13 GHz
   - 8 cores / 16 threads, VM was given 7 cores / 14 threads
   - Speed 2.13 GHz
   - 10 GB of RAM
   - Usage: Training of Gomoku on 10x10 boards

5.2 Tic-tac-toe

As a proof of concept a Random Forest was trained for Tic-tac-toe on 3x3 game boards. Training process was conducted in training environment #3. The resulted forest was able to play draws when not using Minimax algorithm for move decisions (achieved by setting maximum depth equal to zero). This shows that Random Forest was able to recognize important patterns solely on its knowledge base and did not need to rely on Minimax algorithm.

When increasing Minimax depth level to 3, the resulted Random Forest was also able to win the game in most cases when the opponent made a mistake. With depth level 6, the number of winnings increases even further on opponent mistake and gameplay by Random Forest was near perfect based on conducted human tests. This tendency can be also observed when competing resulted forest against completely randomly moving player – these test results are given in [Table 5-1].

		As first			As second		
Iteration 102	Total games	Wins	Draws	Loss	Wins	Draws	Loss
Depth 0	200	97	0	3	72	0	28
Depth 1	200	95	5	0	81	17	2
Depth 2	200	100	0	0	78	15	7
Depth 3	200	99	0	1	76	13	11
Depth 4	200	99	1	0	90	10	0
Depth 5	200	100	0	0	95	5	0

Table 5-1: Random Forest against randomly moving player

The Random Forest was trained with Genetic Algorithm population size of 20 where forests attached to each individual had 25 trees and using 4 random attributes of the game board (patterns). Total of 102 iterations were trained within 6 hours and 30 minutes using 100 threads.[3] The average individual evaluation score during each iteration is shown in [Figure 5-2] – a rapid growth can be seen already within first 10 iterations, which stabilizes later due to most games ending with draw result.

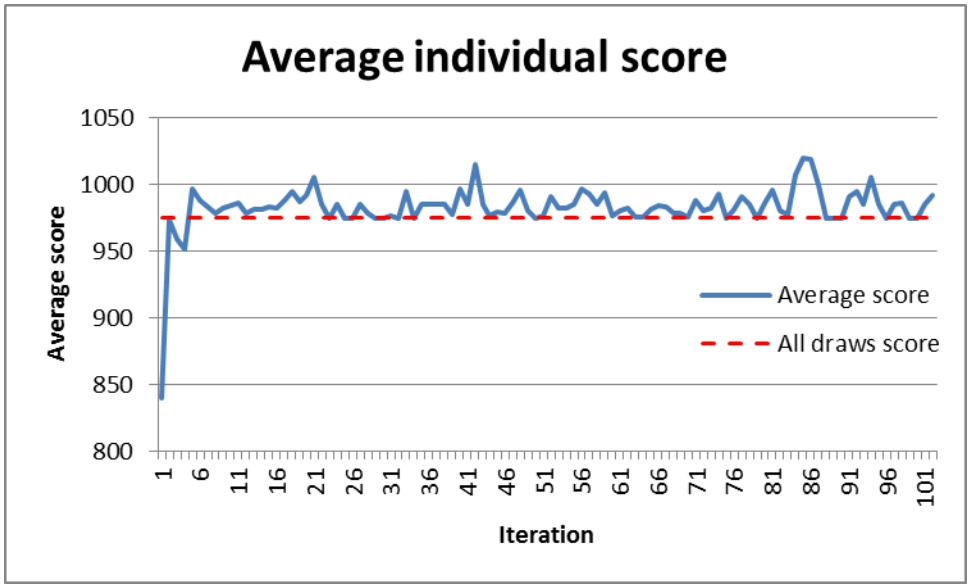

Figure 5-2: Average population score

5.3 Connect Four

A single Random Forest was trained for playing Connect Four on 7x7 boards. Training process was conducted in environment #3. The resulted forest was trained only using Minimax depth level 6 and it was able to play a victory in many cases by building dual or triple forks, while human player was able to win only by end game forcing moves. This was caused due to computer unable to detect and block some open moves that were created by the human player in the start of the game. One of such situations is illustrated in [Table 5-2].

[3] Due to speed of the games all threads and CPU cores were possibly not fully utilized.

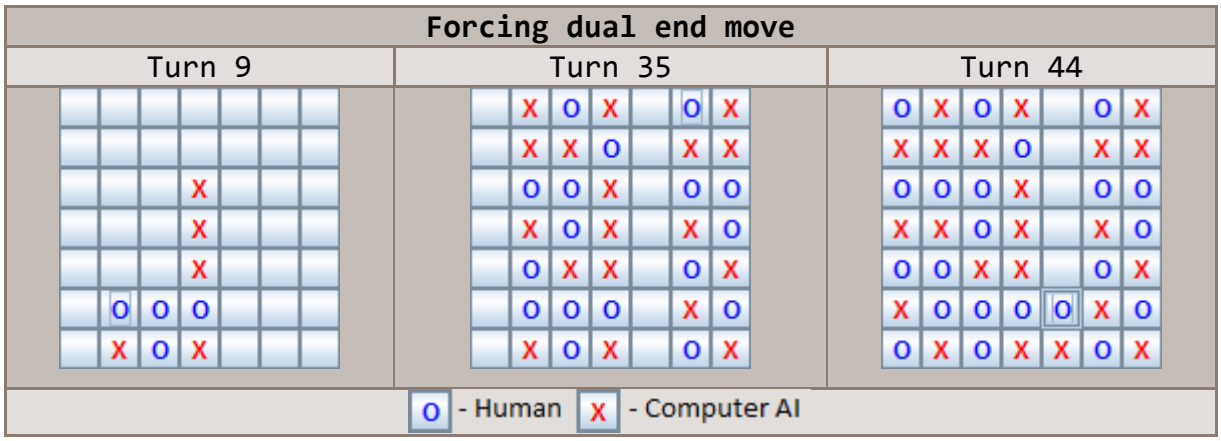

Table 5-2: Forcing dual end move

All test games ended by either player win and no draws were achieved. Detailed results can be found in Appendix [10.3] and summary of test results are given in [Table 5-3].

Connect Four	Total games	AI Wins	Draws	AI Loss
Depth 6	20	12	0	8

Table 5-3: Connect four game results

The Random Forest was trained with only with 50 trees, where each tree used 5 random attributes of the game board (patterns). Total of 107 iterations were completed within 4 days of training time using 40 threads.

5.4 Gomoku

As higher proof of concept, the main effort in training was done with Gomoku on 10x10 boards. Training process was conducted in training environment #4. On each server out of eight, one Random Forest was trained. The forests went through a series of competitions from where 3 better performing were taken further for human tests. Latest competition results are described in [Table 5-4], where each candidate played against each other candidate twice (as first player and as second player).

		SCORE						
	Iteration	**Depth 0**	**Depth 1**	**Depth 2**	**Depth 3**	**Depth 4**	**TOTAL**	**WEIGHTED**
Test #87	172	8	4	4	4	12	**32**	**16.8**
Test #90	58	-8	4	0	0	-8	**-12**	**-8**
Test #92	55	-4	0	12	4	4	**16**	**9.2**
Test #93	47	-8	8	-4	4	8	**8**	**9.6**
Test #94	36	12	4	-4	8	0	**20**	**4.8**
Test #96	24	0	-20	-8	-8	4	**-32**	**-6.4**

Table 5-4: Final competition results

Human tests were conducted with help of mainly 5 persons (including author of the work), each of who played series of games on different Minimax depth settings when starting as first and second player. For each Random Forest 3 tests were done in each setting. Overview of players' feedback on the game results is following:[4]

Depth 1

Computer was not able to resists against human players at all and movements seemed to be quite random. Not blocking player straight rows and it was possible to win by placing 5 stones in a row right away.

Users average rating: completely useless

Depth 2

Still not able to resist against human players, but random movement reduced to more sensible and blocking some of the player straight wins.

Users average rating: useless, sometimes can block

Depth 3

Number of randomly seeming moves decreased drastically and computer was able to block player straight open moves thanks to Minimax algorithm – this resulted in longer games on average. Some players had trouble winning and lost many games due to human error (not seeing computer winning possibility). Still it is quite easy to defeat computer by building a simple double fork. Computer was not able to avoid mostly useless moves near edges of the board. Some of the human tests were done on this level also with a 10 year old 2nd grade child and he was not able to win.

Users average rating: average

Depth 4

Most players had trouble winning computer on this level. While it is still possible using double fork, computer was able to detect many of them without Minimax help. Also computer started building actively forks with help of Minimax algorithm. Minimal amount of randomly seeming moves were done by computer. Average game length increased significantly.

Users average rating: good

[4] Detailed results were lost during the human tests due to scripting error. This was partially caused by unpredictable internet connectivity problems right before the testing process started. Players' feedback was collected in written free form and with author's own observations during the games.

Depth 5

Not possible to defeat computer with simple double work thanks to Minimax algorithm and each move by human players had to done very carefully. In order to win, player must construct a complex fork system. Most games won by computer as it was constructing actively forks and blocking player good moves. Games take quite many moves to finish and in the end computer wins in most cases.

Users average rating: very good

The results overall are above expectations and received positive feedback from players with one exception – each game takes long time starting from depth level 4 and is quite unreasonably slow already for level 5 (one move for computer took on average 15 minutes when using 4 threads in training environment #3). A production level application would require either server-side solution or heavy optimization, with one possibility being cache where all moves are already pre-calculated.

In order to measure training time performance, a test group of 4 trainings (Test #92, Test #93, Test #94 and Test #96) was conducted with identical training parameters except for number of trees in a single forest. The main object of the test group was to observe the impact on training times with different number of trees in each forest. All the trainings were done in same amount of time (13 days) and it's clear that higher number of trees makes training process longer (as expected). To illustrate the difference, refer to [Figure 5-3] where each iteration training time is shown. Training configuration details are described in [Table 5-5].

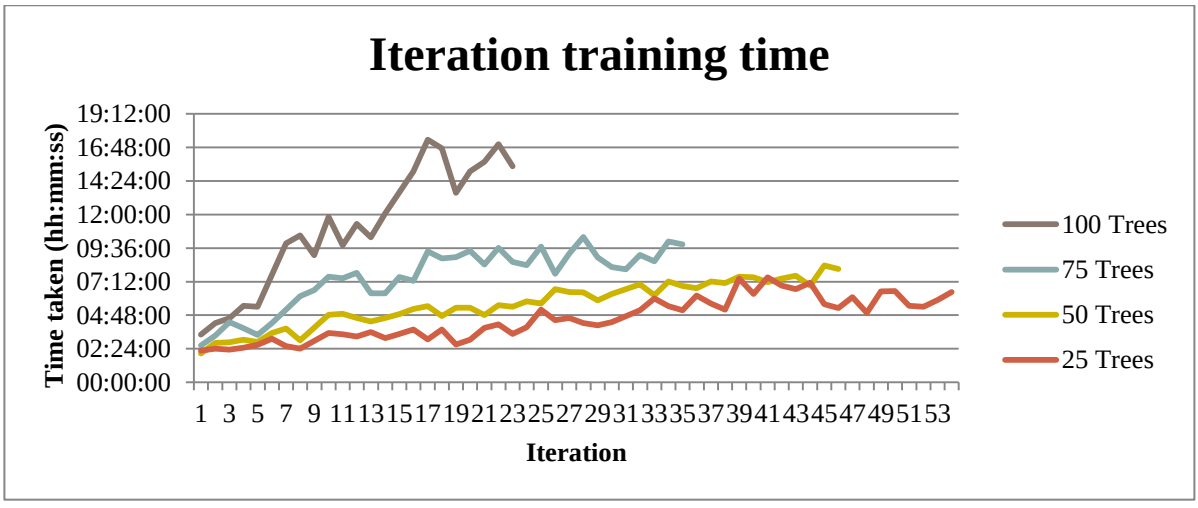

Figure 5-3: Iteration training time

Configuration property	Value
Genetic population	5
Maximum population size	10
Genetic mutation rate	0.15 (15%)
Games per one individual	5+5
Total number of games	50
Win evaluation score	2.0
Draw evaluation score	1.0
Loss evaluation score	-3.0
Decision Tree number of attributes	8
Decision Tree attributes pattern radius	7
Decision Tree win prediction weight	2.0
Decision Tree draw prediction weight	1.5
Decision Tree loss prediction weight	-0.5
Decision Trees in each forest	Test #92 – 25 trees Test #93 – 50 trees Test #94 – 75 trees Test #96 – 100 trees
Number of iterations trained	Test #92 – 55 iterations Test #93 – 47 iterations Test #94 – 36 iterations Test #96 – 24 iterations
Total training time per test	13 days

Table 5-5: Performance test training configuration

6 Future work

There were several observations done during the implementation of the algorithms that could potentially improve the training speed and efficiency, but due to time limitations were not explored in detail.

The naive Threaded Minimax implementation separates game tree search only at the top level of the tree. This means that concurrent game tree search will not be able to fully use Alpha-Beta pruning information. A possible research topic would be to implement fully concurrent Minimax where Alpha-Beta values are synchronized between running threads and full information is used.

Another possible improvement would be to add weights for each Decision Tree pattern – currently all patterns are with equal weight and give same proportion to the evaluation score. There could be several ways how to implement this – for example by adding tracking to Minimax algorithm in order to check which patterns helped to decide towards good move. Completely different approach would be to use Genetic Algorithm to find good pattern weights to already predefined patterns, i.e. instead of evolving patterns, the algorithm would evolve the weights.

During the evaluation process, each individual is competed against other individuals. This takes most of the whole training process time due to speed of Minimax algorithm. The evaluation process could be instead to complete many various tasks in predefined scenarios, for example:

- Defend a losing position
- Attack a good position
- Form a forcing move for opponent
- Find a winning move on a board in 3 moves
- Games against randomly moving player
- Games against external implemented A.I. players

This would fundamentally differ from proposed thesis topic as this requires a lot of human expertise for creating such scenarios – current idea is that Random Forests are trained from scratch with as little human expertise as possible.

7 Summary

The aim of this thesis was to explore the viability of combining multiple machine learning techniques in order to train k-in-a-row type games-playing Artificial Intelligence. These algorithms are following: Genetic Algorithm, Random Forests and Minimax.

The main idea for Genetic Algorithm was to find as efficient game board patterns as possible for Decision Trees used in Random Forests. The objective for Random Forests was to act as a heuristic function in Minimax algorithm. Minimax itself was used during Genetic Algorithm evaluation step where each individual competed against each other individual. Better performing individuals were transferred to next iteration and new individuals were produced from them as per Genetic Algorithm. Such training process was repeated until pre-defined time limit or satisfactory result was reached.

The algorithms were implemented and with the developed software multiple Random Forests were trained to evaluate the effectiveness of such method. The trained forests were then taken into series of games against human players in various settings with results above expectations. Important to note that Random Forests for different games were trained with same method by just adjusting slightly either board size or game rules, indicating that this method is quite universal. Although it is slow in normal environments, the speed of training process can be increased by just using more CPU power – considering that there are many high performance computing centers nowadays, it should not be ruled out. Also multiple algorithm performance optimizations were suggested as possible future work.

8 Summary in Estonian – Masinõpe k-ritta mängude õppimiseks

Antud töö põhieesmärgiks oli uurida kui efektiivne ja mõistlik on kombineerida mitu erinevat masinõppe meetodit, et treenida tehisintellekti *k-in-a-row* tüüpi mängudele. Need meetodid on järgnevad: geneetiline algoritm, juhumetsad (koos otsustuspuudega) ning Minimax algoritm. Eriliseks teeb sellise meetodi asjaolu, et kogu intelligents treenitakse ilma inimese ekspert teadmisteta ning kõik vajaliku informatsiooni peab arvuti ise endale omandama.

Geneetiline algoritm imiteerib looduses toimuvat evolutsiooni ning otsib optimaalseid lahendusi probleemidele, mis kombineeritakse olemasolevatest lahendustest (nagu loodeses kahe vanema abil sünnib uus laps) ja kus tugevamad jäävad ellu. Iga indiviidi tugevuse määrab nende omavaheline võistlus mängudes, kus võitjad pääsevad edasi ning kaotajad langevad treeningust välja.

Minimax algoritm on otsing mängupuul, mida kasutatakse tihti kahe mängija mängudes ja kus on olemas täielik informatsioon mängu seisude kohta (puudub peidetud info). Tegemist on otsinguga, mis leiab sisend mängu seisust järgmise käigu, millel on garanteeritud maksimaalse väärtus. Ehk siis algoritmi põhiidee on hoida kaotuse võimalust võimalikult madalal. Minimax kasutab heuristilist funktsiooni, et hinnata mänguseise ning see hinnangu funktsioon tulebki kolmandast kasutatust meetodist – juhumetsadest.

Juhumetsad on komplekt-klassifikaator (või ennustaja), mis kasutab mitmeid otsustus puid korraga. Igal otsustus puul on kindlaks määratud tunnused, mida sisend andmetes vaadeldakse. Antud töös on juhumetsade sisendiks mängude laua seis, millest iga üksik puu vaatab erinevaid omadusi (niiöelda positsioonide mustreid) ning erinevate puude tulemused summeeritakse. Juhumetsa väljundit kasutatakse Minimax algoritmis ning selle peamine eesmärk on ennustada kui hea mingi laua seis on. Geneetilise algoritmi eesmärk on leida võimalikult effektiivsed omaduste valikud (mustrid) otsustuspuudele.

Kõik nimetatud algoritmid said implementeeritud ja valmis tarkvaraga treeniti mitmeid erinevaid juhumetsasid. Valminud metsad pandi omavahel võistlema ning parimad omakorda suunati edasi mängima inimeste vastu. Tulemused olid loodetust paremad – inimestel tekkis raskusi arvuti vastu juba kasutades Minimax otsingupuu sügavust 3.

Kuigi selline algoritm on üsna aeglane, siis arvestades erinevaid optimiseerimise või tänapäeva tehnika võimalusi, saab seda protsessi alati kiirendada.

9 References

[1] M. Tran, "Deeper blue dip for Kasparov," *The Guardian*. p. 9, 1997.

[2] D. B. Fogel, *Evolutionary computation: toward a new philosophy of machine intelligence*. IEEE Press, 1995.

[3] K. Chellapilla and D. B. Fogel, "Evolution, neural networks, games, and intelligence," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1471-1496, 1999.

[4] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*, vol. 19, no. Book, Whole. Wadsworth, 1984, p. 368.

[5] L. Breiman, "Random Forests," *[[Machine Learning (journal)|Machine Learning]]*, vol. 45, no. 1, pp. 5 - 32, 2001.

[6] M. J. Osborne, *A Course in Game Theory*, vol. 11, no. 1. The MIT Press, 1995, pp. 93-100.

[7] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, vol. Addison–We. Addison-Wesley, 1989, p. 432.

[8] H. Van Den Herik, "Games solved: Now and in the future," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 277-311, 2002.

[9] L. V. Allis and others, *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen, 1994.

[10] M. Y. Hsieh and S.-C. Tsai, "On the fairness and complexity of generalized -in-a-row games," *Theoretical Computer Science*, vol. 385, no. 1–3, pp. 88-100, Oct. 2007.

[11] A. Smith, *The game of go : the national game of Japan*. Rutland Vt.: C.E. Tuttle Co., 1956.

[12] V. Allis, "A Knowledge-based Approach of Connect-Four," Vrije Universiteit, 1988.

[13] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, vol. 60, no. 4. Prentice Hall, 2003, pp. 269-72.

[14] G. T. Heineman, G. Pollice, and S. Selkow, *Algorithms in a Nutshell*, vol. 10, no. 6. O'Reilly Media, 2010, pp. 358-66.

[15] N. Nogueira, "Minimax algorithm," *2006*. [Online]. Available: http://en.wikipedia.org/wiki/File:Minimax.svg. [Accessed: 07-May-2012].

[16] S. Milborrow, “An example of a CART classification tree,” 2011. [Online]. Available: http://en.wikipedia.org/wiki/File:CART_tree_titanic_survivors.png. [Accessed: 07-May-2012].

[17] R. Caruana, N. Karampatziakis, and A. Yessenalina, “An empirical evaluation of supervised learning in high dimensions,” *Proceedings of the 25th International Conference on Machine Learning (2008)*, pp. 96-103, 2008.

[18] M. R. Segal, “Machine Learning Benchmarks and Random Forest Regression.” UC San Francisco: Center for Bioinformatics and Molecular Biostatistics, 2003.

[19] O. Boz, “Converting a trained neural network to a decision tree.,” Lehigh University, 2000.

[20] O. Widder, “Geek And Poke: Good Coders,” 2011. [Online]. Available: http://geekandpoke.typepad.com/geekandpoke/2011/07/good-coders.html. [Accessed: 07-May-2012].

[21] R. N. Horspool, “Practical fast searching in strings,” *Software Practice and Experience*, vol. 10, no. 6, pp. 501-506, 1980.

[22] P. Borovska and M. Lazarova, “Efficiency of parallel minimax algorithm for game tree search,” in *Proceedings of the 2007 international conference on Computer systems and technologies CompSysTech 07*, 2007, p. 1.

10 Appendices

10.1 Forest XML files DTD

This appendix describes document type definition how Random Forests are stored in XML file format. For each forest one separate file would be created.

```
<!DOCTYPE RandomForest [
 <!ELEMENT RandomForest ( RandomWoods )+ >
 <!ELEMENT RandomWoods ( TrainingAttributesList, ForestTree ) >
 <!ELEMENT TrainingAttributesList ( TrainingAttributes )+ >
 <!ELEMENT TrainingAttributes EMPTY>
 <!ELEMENT ForestTree ( TreeRoot ) >
 <!ELEMENT TreeRoot ( TreeNode | TreeLeaf )+ >
 <!ELEMENT TreeNode ( TreeNode | TreeLeaf )+ >
 <!ELEMENT TreeLeaf EMPTY >
 <!ATTLIST TrainingAttributes attributeIndxs CDATA "" >
 <!ATTLIST ForestTree decisionAttr CDATA "" >
 <!ATTLIST TreeRoot headerIndx CDATA "" >
 <!ATTLIST TreeNode branchValue CDATA "" >
 <!ATTLIST TreeNode headerIndx CDATA "" >
 <!ATTLIST TreeLeaf branchValue CDATA "" >
 <!ATTLIST TreeLeaf categoryValue CDATA "" >
]>
```

10.2 Deployment guide

The source code of the work can be found in Appendix [10.3]. Ant tools and build script is used to create the binary Java application archives. Already prebuilt JAR files are also available as listed in [Table 10-1]. The applications can be started using standard Java command:

java -jar *applicationname* [parameters]

The source code has been developed using JavaSE 7u4 and requires at least Java runtime environment of level 7.

Application	Comment
competition.jar	Used to compete different Random Forests against each other
dbimport.jar	Used to import game results from one DB to other
humantests.jar	Used to test single Random Forest human or random player using CLI
startgui.jar	Starts GUI application for testing Random Forests
connect-four.jar	Starts training Connect Four based on connect-four.properties file
connect-six.jar	Starts training Connect6 based on connect-six.properties file
gomoku.jar	Starts training Gomoku based on gomoku.properties file
tic-tac-toe.jar	Starts training Tic-tac-toe based on tictactoe.properties file

Table 10-1: Applications prebuilt

The full binary distribution folder content is as follows:

+	dist		binary distribution folder
|	-	tic-tac-toe.jar	application file
|	-	connect-four.jar	application file
|	-	gomoku.jar	application file
|	-	connect-six.jar	application file
|	-	dbimport.jar	application file
|	-	humantest.jar	application file
|	-	competition.jar	application file
|	-	startgui.jar	application file
+	-	etc	configuration files for training and logging, misc files
	|	-	RandomForest.dtd
	|	-	connect-four.properties
	|	-	connect-o-bot.properties
	|	-	connect-six.jardesc
	|	-	connect-six.properties
	|	-	footballtree.txt
	|	-	gomoku.properties
	|	-	ideas
	|	-	log4j-aitest.properties
	|	-	log4j-gui.properties
	|	-	log4j-minimax.properties
	|	-	log4j.properties
	|	-	tictactoe.properties
+	-	data	database location during training
+	-	gui	graphical user interface configurations and forests
	|	-	connect-four.properties	game configuration file for GUI
	|	-	gomoku10.properties	game configuration file for GUI
	|	-	gomoku7.properties	game configuration file for GUI
	+	-	connectfour	connect four forest files
		|	-	forest01.xml
	+	-	gomoku	gomoku forest files
		|	-	forest01.xml
+	-	lib	external libraries used
	|	-	antlr-3.4-complete-no-antlrv2.jar	used by byteseek
	|	-	byteseek-1.2.jar	pattern search algorithms on byte level
	|	-	hsqldb.jar	HSQLDB for storing game states

	|	-	jansi-1.8.jar	used to colorize command line interface
	|	-	junit-4.10.jar	some basic tests with JUnit
	|	-	log4j-1.2.16.jar	logging management
+	-	log		logging target directory
	|	-	forests	random forest XML files are placed here during the training
	|	-	minimax	separate logging folder used by GUI and other debugging classes parts
+	-	test		random forests location for human and other tests
	|	-	negamaxevaltest.xml	Negamax evaluation test
	|	-	visualise.xml	forest visualization test
	|	-	humanaitest.xml	forest XML that is used by humatest.jar application
	|			
	+	-	competition	forest contained in this folder are used by competition.jar application

10.3 Source code and log files

Attached physical storage device in the form of DVD. Including source code, binary distribution files, logs of conducted experiments and trainings.