UiA University of Agder

Master

Playing the game of Hex with the Tsetlin Machine and tree search

Audun Linjord Simonsen
Ole André Haddeland

SUPERVISOR
Ole-Christoffer Granmo

University of Agder, 2020
Faculty of Engineering and Science
Department of ICT

UiA
University of Agder
Master's thesis

Faculty of Engineering and Science
Department of ICT

Abstract

Hex is an abstract mathematical board game where the players aim to build a connection of pieces, traversing the board from edge to edge. The game requires the use of certain patterns to be played at a high level. Artificially Intelligent Hex players have had success using Monte Carlo tree search and current research efforts have introduced neural networks. This thesis looks into the recent Tsetlin Machine pattern-recognition technique, relying on interpretability, in combination with the Monte Carlo tree search method to play the game of Hex. A supervised learning approach has been employed in an effort to teach the Tsetlin Machine beneficial patterns for winning, resulting in around 91% accuracy, 87% recall and 97% precision. It is demonstrated with a Hex tournament that the Tsetlin Machine is unable to play perfectly on a board of size 6×6 alone, but performs much better in combination with Monte Carlo tree search. Monte Carlo tree search reduced the number of averagely placed piece from around 35.5 down to around 20 and below. The benefit of using the Tsetlin Machine's interpretable clauses and pattern capabilities are that they can provide valuable knowledge needed for gameplay, and appear helpful for ventures into larger unexplored board sizes.

Keywords: Hex, Tsetlin Machine, Monte Carlo tree search, tree search, board evaluation

Preface

This master's thesis is the final work in the subject IKT590 at the University of Agder, Grimstad, Norway. The project idea was defined by our supervisor Ole-Christoffer Granmo, who introduced the Tsetlin Machine.

We would like to thank our professor Ole-Christoffer Granmo for being our supervisor, guiding us during the master thesis period. The thesis would not have been possible without his expertise and availability for questioning. His help and input has proved to be invaluable.

Audun Linjord Simonsen
Ole André Haddeland

Grimstad, May 2020

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The game of Hex is an abstract board game with territorial and connection aspects, features similar to its inspirational sibling game Go [12]. Go is an abstract strategy board game believed to derive from ancient China [47], while Hex was first invented in the 1940s by Danish mathematician Piet Hein [36]. At first glance, Hex and Go appear similar, yet the winning conditions are drastically different and Hex has the unique property of being drawless, guaranteeing a winner.

Hex has been considered difficult to solve due to the sheer amount of complexity the game has, in comparison to games like Chess. Chess was played at above top level with classical heuristic search as early as 1997, with Deep Blue beating Garry Kasparov, the world champion at the time [13]. Chess is played on an 8×8 board and has an approximate complexity of $4.6 * 10^{46}$ legal positions, while 11×11 Hex has $2.4 * 10^{56}$ legal positions, a sizable difference [49, 12].

Go is rooted in Chinese culture, and is played by over 46 million people wordwide [47, 19]. Go is more popular than Hex, and naturally established itself as the complex game to play above top level, instead of Hex. In 2006, the Monte Carlo tree search (MCTS) algorithm was introduced to Go [15, 26], and there was a noticeable strength-gap between programs that used it and those who did not. MCTS

was capable of beating strong amateur human players on 9×9 Go boards [41]. Ten years later in early 2016, Deepmind released their work on the program AlphaGo, an artificially intelligent (AI) player for Go. AlphaGo combined MCTS with deep neural networks to play Go above top level [48]. AlphaGo was the first program to beat a professional human Go player at the full-size game of Go, an achievement thought by some to be unattainable by the state-of-the-art at the time, due to the game's complexity [50].

In 2018, Prof. Ole-Christoffer Granmo at the University of Agder introduced the Tsetlin Machine [27], a novel and promising machine learning tool using propositional logic for pattern recognition. It is an alternative learning technique relying on interpretability, in comparison to black box neural networks [27, 44]. Interpretability is critical for understanding machine learning, especially in the medical field [44, 10].

1.1 Motivation

Current research efforts in improving Hex AI are pursuing the use of neural networks. The recently introduced Tsetlin Machine shows competitive results to neural networks [27] and deep neural networks have achieved great success in competitive Go. However, these networks are difficult to decipher. The benefit of having the Tsetlin Machine for this task is to get a better understanding of how and what the algorithm learns. As such, the Tsetlin Machine is an exciting new tool for exploring game solving. The novelty of the Tsetlin Machine is a motivation for exploring this domain with the game of Hex, a similar game to Go.

1.2 Thesis definition

1.2.1 Thesis Goals

Goal 1: Use the Tsetlin Machine's pattern recognition capabilities to evaluate game boards in Hex, in order to predict the game winner.

Goal 2: Optimize the Tsetlin Machine's parameters to make proficient and accurate evaluations of any given Hex board.

Goal 3: Utilize the Tsetlin Machine's evaluations of game boards to select winning moves.

Goal 4: Combine the Tsetlin Machine with tree search and the Monte Carlo tree search method to make an artificially intelligent Hex player.

1.2.2 Hypotheses

Hypothesis 1: Hex is an abstract game of patterns, where the Tsetlin Machine can learn winning sub-patterns.

Hypothesis 2: The Tsetlin Machine will not be able to play the game at a reasonable level, where the Monte Carlo tree search method will be an effective addition to the Tsetlin Machine.

1.3 Contributions

This thesis examines the recent Tsetlin Machine's ability using supervised learning to evaluate board positions in Hex, looking at accuracy, precision and recall, with the evaluations as pointers for gameplay. Tree search and Monte Carlo tree search algorithms were combined with the Tsetlin Machine to make artificially intelligent Hex players. Using the Tsetlin Machine for board evaluation, the Monte Carlo tree search players are able to play some promising moves.

The provided contributions have set a baseline for the Tsetlin Machine's use-case in playing Hex, which opens up several future possibilities.

1.4 Thesis outline

- Chapter 2 explains necessary theory about the game of Hex, the Tsetlin Machine, tree search and Monte Carlo tree search.
- Chapter 3 presents the state-of-the-art within artificially intelligent Hex players and their concepts.
- Chapter 4 explains the proposed Tsetlin Machine Hex solution, and how Hex was evaluated.
- Chapter 5 will present results and discuss the findings related to the implementations in Chapter 4.
- Chapter 6 concludes the work.
- Chapter 7 propose future work.

Chapter 2

Background

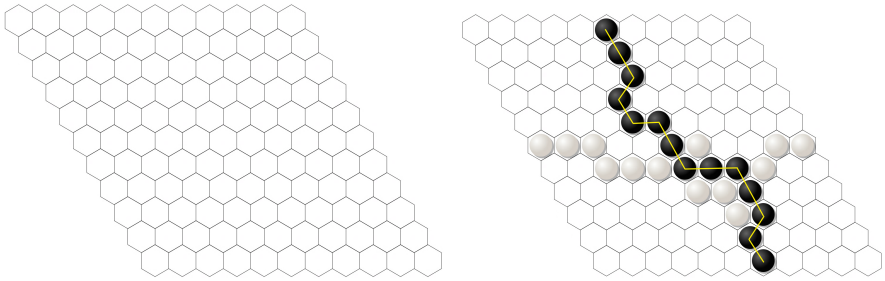
This section will give an overview of concepts and techniques used in this thesis, as theoretical background. Firstly, the game of Hex is explained. Section 2.2 then explains the Testlin Machine. Section 2.3 explains tree search and the Monte Carlo Tree Search algorithm used for playing the game of Hex. Lastly, section 2.4 explains confusion matrices, which are important tables for assessing a classifier's performance.

2.1 The game of Hex

Hex was invented by Danish mathematician Piet Hein in 1942 and later independently re-invented by John Nash in 1948 [35, 37]. Hex is played on a rhombus board where two players take alternating turns putting pieces on empty spots anywhere on the board. A board size of 11×11 is popular because this was used by Hein in 1942, see figure 2.1a, while John Nash advocated using 14×14 boards.

The first player uses black pieces and the second player uses white pieces. The goal for the first player is to form a connection with the pieces from the top edge to the bottom edge, while the second player

will try to form a connection from the left edge to the right edge, see figure 2.1b. In this thesis, "first player" and "black" will be used interchangeably, the same is true for "second player" as "white".



(a) Empty 11x11 Hex board

(b) Finished Hex game where Black is the winner

Figure 2.1: A Hex board showing a win for black that has connected the pieces vertically. The connection is shown with a yellow line.

A game of Hex can never end in a draw. Proving this is equivalent to proving the Brouwer fixed-point theorem for two dimensions, as done by David Gale [20]. As such, the only way to completely block the opponent is to form a connection with one's own pieces. In his invention of the game, John Nash proved that there exists a perfect strategy for black, using a strategy-stealing argument, in order to win every game on boards of size $n \times n$ [25]. Finding these strategies become increasingly difficult with larger board sizes, with current research having solved up to 9×9 boards and also some of the 10×10 openings [42].

Due to this significant favoring of the first player, the game is commonly played with the *swap rule*. This rule states that after black has placed his first piece, white can choose to swap this piece for a white piece. Play then continues with black, which has now effectively become the second player. If white wins after the usage of the swap rule, the game still regards it as a win for the second player.

2.1.1 Compared to the game of Go

As mentioned by Sensei's Library [2], Hex has some similarities with Go:

- Both have territorial and connective aspects.
- Both have non-mobile pieces, different from Chess.
- Similar branching factors at similar board sizes. At 11×11 board size, Hex has $2.4 * 10^{56}$ legal positions while Go has $7.9 * 10^{56}$ legal positions [12, 50].
- Both Hex and Go have been considered difficult for computers due to their complexity.

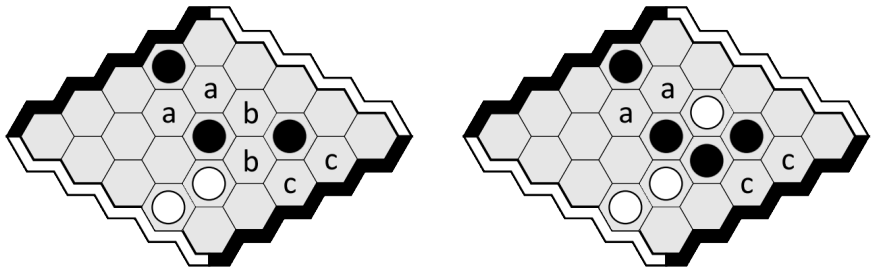
There are also some differences:

- In Hex the pieces are never captured or removed from the board. Once a piece has been placed that position can never be empty again. As a result the game is finite and ends faster.
- Having a piece on any given position is better than not having a piece placed there, unlike Go and Chess where a badly placed piece can be detrimental to a player's game.

2.1.2 Virtual Connections

Hex is an abstract game relying on patterns to be played at a high level. The most crucial ability is to use virtual connections. Virtual connections secure links between pieces without the need to have pieces side by side. The most important virtual connection in Hex is the *bridge* pattern. If two pieces of the same color is separated by a gap of two empty spots, a connection is secured. Figure 2.2 illustrates this pattern with 3 bridges (a, b, c), where the black player can form a link from top to bottom without the white player being able to stop

him. If white tries to block a bridge (either a, b, or c), black can place in the other spot to secure it. Because of virtual connections, a win or loss can be determined in advance because of the inevitability to form a link.



(a) Black player having the bridges a, b and c

(b) White tries to block the b bridge, but black upholds the connection

Figure 2.2: Hex bridge patterns. Figure graphic adapted from Henderson et al. [37]

2.2 The Tsetlin Machine

The Tsetlin Machine (TM) is based on the work of M.L. Tsetlin and his Tsetlin Automaton (TA) [51], as well as clauses from propositional logic [27]. The TM is composed of multiple logical clauses built up of several TA that learn in a decentralized manner. A TM relies on interpretability with logic, and is more efficient without the need for backpropagation, required to train neural networks [27].

2.2.1 The Tsetlin Automaton

A TA is a learning mechanism within reinforcement learning, that adapts based on rewards and penalties and is capable of solving the multi-armed bandit problem [27]. Given a two-action environment the TA will learn which action to take, and is in its essence merely an integer value, keeping track of the *state* of the automaton. Given

N states per action, the automaton can be between state 1 and 2N. Rewards and penalties, which are unknown to the TA and can also change over time, increase and decrease the TA state respectively. This is done with stochasticity to prevent each automaton from becoming equivalent. If the state is between 1 and N the first action is selected, and if the state is between N+1 and 2N the second action is selected. Higher values of N will allow the automaton to become more confident on an action. Figure 2.3 illustrates this learning scheme for a single Tsetlin Automaton with N=2, meaning two states for each action and four states in total.

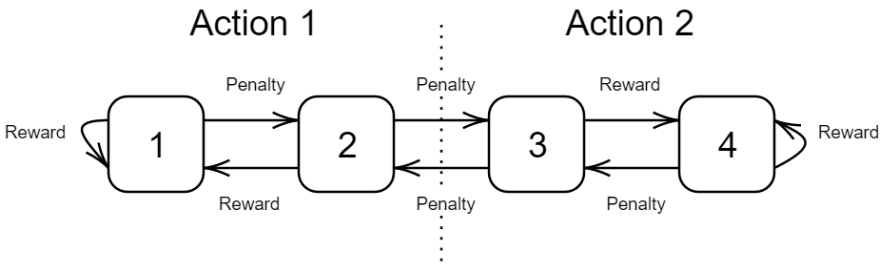


Figure 2.3: Tsetlin Automaton learning between two actions with two states per action.

The problem with the TA is that when making them work together there is an increasing noise with each added TA, an effect referred to as the *vanishing signal-to-noise ratio problem* [27]. However, this problem is overcome with the Tsetlin Machine game [27]. Essentially, the game is played such that all automata individually decide upon an action (1 or 2). They each get reward/penalty based on their collective actions, then the next round of the game is played. Given W automata, there exist 2^W unique action configurations. The TA never interact with each other, meaning the process is decentralized.

2.2.2 Clause

In logic, a clause is an expression composed of literals, negated or non-negated, while a TM clause is a conjunction of several Tsetlin

Automata. A conjunction is also referred to as the AND operator, see table 2.1.

Table 2.1: AND logic operator

x_1	x_2	x_1 **and** x_2
0	0	0
0	1	0
1	0	0
1	1	1

The inputs in the AND operator must all be 1 for the output to be 1. This is identical to multiplying the inputs together as any instances of 0 will make the result be 0.

A TM clause has two automata responsible for each feature. One automaton for the negated literal form and one automaton for the non-negated literal form. Each automaton learns whether to include or exclude (two-action environment) its literal from the clause conjunction. Thus, each clause conjunction makes a pattern based on the features.

A clause pattern can be explained using the binary logic expression XOR as an example. The XOR expression is shown in table 2.2 below, and states that both inputs must be different from each other for it to output 1.

Table 2.2: XOR logic operator

x_1	x_2	x_1 **xor** x_2
0	0	0
0	1	1
1	0	1
1	1	0

A TM clause learning the XOR pattern will assign two TA per feature. This is shown in figure 2.4 where it assigned a TA for x_1, x_2, $\bar{x}_1$ and $\bar{x}_2$. Each TA that has learned the include action is part of the conjunctive clause. In figure 2.4 these are x_1 and $\bar{x}_2$, and as such the conjunction will be $x_1 \wedge \bar{x}_2$. This is a good pattern for output 1 of the XOR problem. Both inputs must be different in XOR and the pattern states that x_2 must be different from x_1 for the conjunction to output 1. Conversely, the other pattern that captures this is $\bar{x}_1 \wedge x_2$.

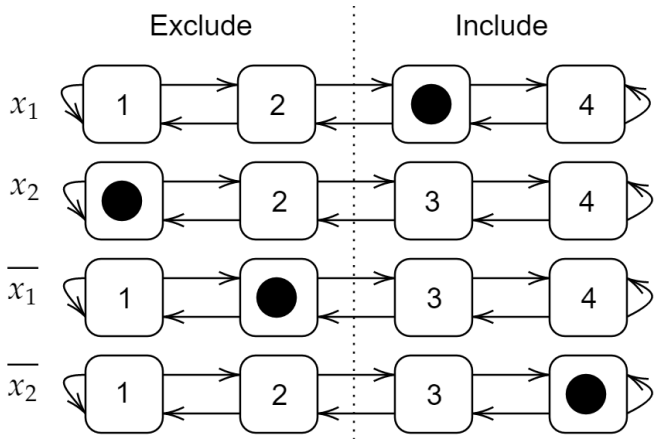


Figure 2.4: A clause with TA for the features x_1 and x_2 that has learned an XOR pattern for output 1. Circles represent the current TA state.

A different configuration of TA states will perhaps learn the pattern $x_1 \wedge x_2$ as in figure 2.5. This is a pattern for XOR output 0 where both features must be equal. The other pattern that captures output 0 is $\bar{x}_1 \wedge \bar{x}_2$.

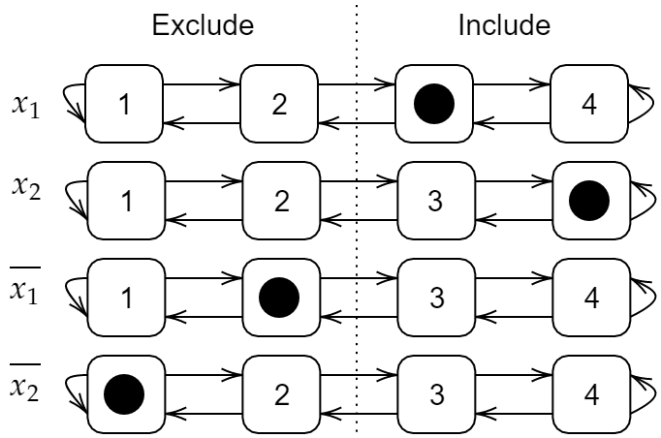


Figure 2.5: A clause with TA for the features x_1 and x_2 that has learned XOR pattern for output 0. Circles represent the current TA state

Using multiple clauses, several patterns will be learned and pattern recognition of new features is possible. Here the TM uses a clause voting system which makes up the TM classifier, voting in favor of a given class or against. Each feature vector in the input will be matched with each clause's pattern. This is done by calculating the conjunction's outcome, either 1 or 0. If the output is 0 the clause will abstain to vote, but if the output is 1 it will vote. Even numbered clauses add positive votes for the given class, while odd numbered clauses add negative votes. Figure 2.4 could be an example of an even numbered clause voting in favor of XOR output 1. Figure 2.5 could be an example of an odd numbered clause voting for XOR output 0. At last, the unit-step function decides the final class. If the total votes $\geqslant 0$ the TM classifier is in favor of class 1, otherwise it outputs 0.

2.2.3 Hyperparameters

A TM requires certain values to be decided before training. This is similar to neural networks where the topology of layers, number of epochs, learning rate and mini-batch size is important [30]. The hyperparameters for the TM is: number of *clauses*, *threshold* and *s-value* [27]. Clauses decide the number of patterns to express, threshold is how easily the available clauses are spent representing each specific sub-pattern and s-value decide the granularity of the clause patterns. The higher the s-value is, the more finely the patterns captured will be.

2.2.4 Multi-Class Tsetlin Machine

A single TM classifier can only decide upon a binary class output. Using several TM classifiers, each deciding on a single class, the TM can be built to execute multi-class predictions [27]. An argmax operator decide upon the TM classifier that has the most votes (most confident) and this will be the selected class [27]. This Multi-Class Tsetlin Machine (MTS) architecture is illustrated in figure 2.6, where the MTM will decide upon win, loss or draw in an arbitrary board game. The figure shows the win and loss classifiers are in agreement for a win, with the loss classifier having a negative amount of votes.

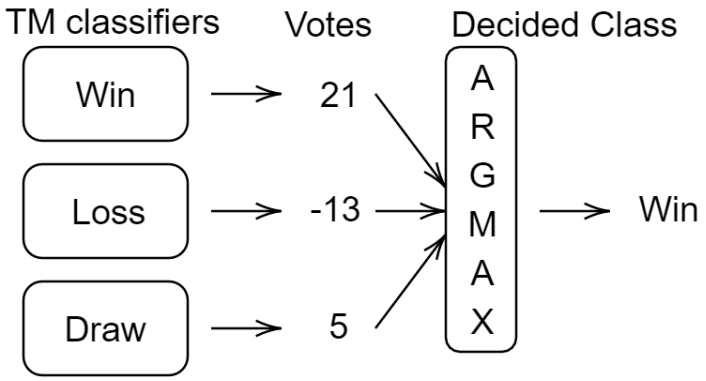


Figure 2.6: Multi-Class Tsetlin Machine architecture [27]

2.2.5 Weighted Tsetlin Machine

The Weighted Tsetlin Machine (WTM) is an improvement upon the MTM that introduces weights, similar to neural networks [43, 5]. The weights can be integers or continuous values and each clause is given a weight coefficient that determines the significance of that particular clause [43, 5]. When voting, each clause will provide as many votes as their weight. The weights are increased and decreased during learning based on clause feedback. True positive patterns have their weight increased and false positive patterns have their weight decreased. This enforces true positive and suppresses false positives [43]. Compared to the MTM, the WTM is shown to achieve similar results with fewer clauses. MTM is strengthened with many clauses, but the weights converge more rapidly [43]. Reducing the amount of clauses also speeds up the TM training and inference [5].

2.3 Tree Search

Given a Hex board of size $n \times n$, the empty starting configuration has $n \times n$ possible moves. Each move gives a new unique configuration of pieces on the board. These configurations can be organized hierarchically in a tree structure. With the starting configuration as the root node, each reachable board position from this position will be children attached to the node. This tree continues down to terminal nodes, positions where the game is finished. Figure 2.7 shows how a tree of 2×2 Hex expands. A full tree have branches for all possible moves in the entire game.

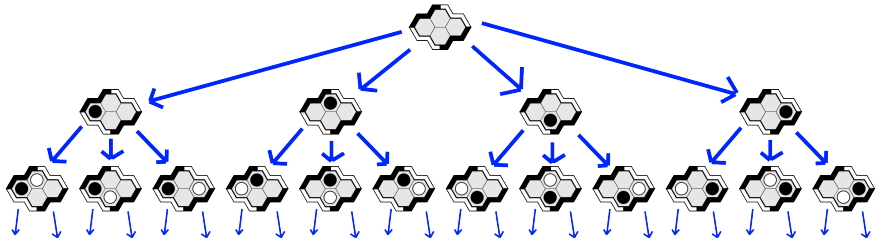


Figure 2.7: Tree structure of 2×2 Hex boards. Figure graphic adapted from Henderson et al. [37]

Tree Search is a method of searching such a structure from a given start position, to find beneficial board positions lower in the tree [45]. Some tree search strategies for exploring the tree are Breadth-First Search and Depth-First Search.

2.3.1 Breadth-First Search

Breadth-First Search starts at the root node and searches through all nodes on a level before searching through nodes on the next level [4]. It guarantees to find the shortest path solution if one exists, with the drawback of high memory usage.

2.3.2 Depth-First Search

Depth-First Search starts at the root node and traverse the tree downwards until it reaches a terminal node, then it backtracks to the parent node whose children are not fully examined. If the tree is very large or infinite, the depth-first search will be stuck spiraling downwards a single path. It does not guarantee to find the shortest path solution unlike Breadth-First-Search, but as a result demands much less memory [4].

2.3.3 Monte-Carlo Tree Search

Many problems are deterministic and in theory solvable using known methods. However, some problems are too complex making them practically unsolvable, because they can not be solved within a reasonable timeframe. Monte Carlo methods are a term for algorithms that combat this problem with stochasticity [28]. Monte Carlo methods work by performing simulations where random samples are drawn from a problem space using a probability distribution [28]. Deterministic methods are used to evaluate the sample. A simulation will approximate a solution, that will be more accurate when several simulations are performed. When the number of simulations tend to infinity, Monte Carlo methods converge to optimal solutions [28].

In 2006, Rémi Coulom applied the Monte Carlo method to tree search, giving it the name Monte Carlo tree search (MCTS) [15]. Since the game trees in Go and Hex have such large branching factors, the Monte Carlo tree search algorithm randomly samples the game tree, instead of searching all node, to find strong continuation paths. A variant of the algorithm was developed by Levente Kocsis and Csaba Szepesvári called the UCT (Upper Confidence bounds applied to Trees) algorithm [39].

Monte Carlo tree search with UCT is an algorithm that builds a hierarchical tree of possible board positions by iterating four stages: Selecting, Expanding, Simulating, and Backpropagation [14]. Selection traverses the tree from the root node to find a node with the highest possibility of winning using the algorithm UCB1 [39]:

$$S_i = \overline{X}_i + C\sqrt{\frac{ln(t)}{n_i}}$$

where;

- S_i = value of a node i
- $\overline{X}_i$ = empirical mean of a node i

- C = exploration constant
- t = total number of simulations

Expanding creates several child nodes from the selected node. Simulation use random decisions down the tree until it reaches a leaf node, giving this leaf node a value. Now that a value has been given to the leaf node the trees values need to be updated, this is done by using backpropagation. The values are then updated from the leaf node all the way up to the root node. After all values are updated step 1 (Selection) begins, creating a loop, a process visualised in figure 2.8

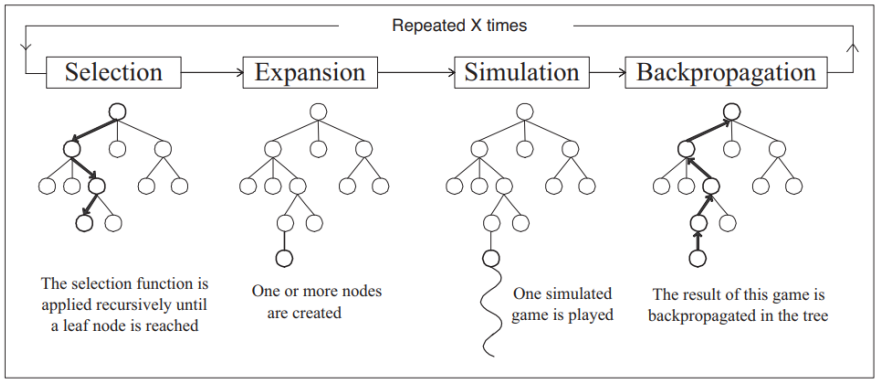


Figure 2.8: Monte Carlo tree search loop. Reprinted from Chaslot et al. [14]

2.4 Confusion Matrix

A confusion matrix is table summarizing a classification process, listing the possible outcomes based on what the true value was and what the classifier predicted [1]. Confusion matrices are commonly used in machine learning when it comes to statistical classification of data and the name refers to the fact that a confusion matrix makes it easy to see when a classifier confuses classes. They are also often called error matrices. Given two classes, positive and negative, the confusion matrix turns into table 2.3.

Table 2.3: Confusion Matrix for positive and negative classes

		True value	
		Positive	Negative
Prediction	Positive	TP	FP
	Negative	FN	TN

With a confusion matrix for Hex, one could say that the positive class is a win and the negative class is a loss. The confusion matrix terminology will then be [1]:

- **Positive:** Win is observed
- **Negative:** Loss is observed
- **True Positive (TP):** Win is observed and win was predicted.
- **False Negative (FN):** Win is observed and loss was predicted.
- **False Positive (FP):** Loss is observed and win was predicted.
- **True Negative (TN):** Loss is observed and loss was predicted.

Given the numbers gained from a confusion matrix some measurements of the classifier's strength can be calculated.

Accuracy is the total number of correctly predicted outcomes divided by the total number of outcomes [1]. The formula for accuracy is:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

Accuracy has some shortcomings with unevenly distributed data and should therefore be paired with Recall and Precision to give a clearer picture of reality. For example, a dataset with 90 observed wins and 10 observed losses would result in an accuracy of 90% if only win was predicted. Such a classifier has not learned anything, but it appears to be a good result.

Recall is the number of true positives divided by the total amount of positives [1]. A high recall number indicates that the positive class is not mistaken for the negative class.

$$Recall = \frac{TP}{TP+FN}$$

Precision is the number of true positives divided by the total number of predicted positives [1]. A high precision indicates that the negative class is not mistaken as the positive class, resulting in the positive predictions being precise.

$$Precision = \frac{TP}{TP+FP}$$

Chapter 3

State-of-the-art

This chapter will explain the current state-of-the-art in the field of artificially intelligent Hex players. Some Hex AI history is also explained leading up to today.

3.1 Early programs

Alpha-beta search [17] was prominent in early Hex programs, exploiting the differences from Go described in section 2.1.1. An analog Hex player was made in the 1950s by C. Shannon and E.F. Moore, using a network of resistors as edges and lightbulbs as vertices [46]. This machine was the inspiration for the program Hexy [3, 6]. Hexy used Shannon's ideas in an alpha-beta search evaluation and also introduced H-search, a way of combining virtual connection (VC) patterns together to build more complex patterns. H-search was shown to be incomplete, and due to this can only find a subset of all VCs [6]. Nevertheless, Hexy was a strong player and won the first Computer Olympiad for Hex in 2000 on a board size of 11×11 [3]. Improving upon Hexy, the program Six won the Computer Olympiads in 2003, 2004 and 2006 [34].

Further improvement in the Hex analysis field is Inferior Cell Analysis (ICA), which finds game positions that are dead, i.e. useless and need not be considered [31, 11]. Figure 3.1 illustrates such dead cells. The program Wolve was developed based on H-search and ICA, and won the Computer Olympiad in 2008 [7].

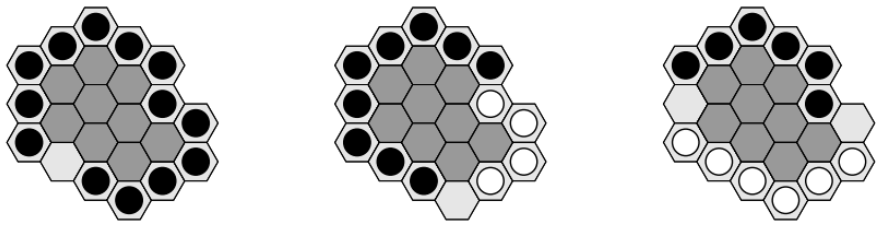


Figure 3.1: Regions of dead cells. Figure reprinted from Henderson et al. [37]

3.2 MoHex

Inspired by Go and MCTS, MoHex was created by Arneson et al. [9] in early 2007 and as part of the Benzene Project codebase[1]. The Benzene Project is a Hex framework built upon Fuego [18], a C++ library developed at the University of Alberta for the game of Go. Both Wolve and MoHex's source codes are available through Benzene. In 2008, MoHex came second in the Computer Olympiad, while in 2009 it was able to beat Wolve and get the gold medal [8].

MoHex uses tricks in order to speed up the MCTS algorithm. It employs tree knowledge, with H-search and ICA, to prune the game tree and make reduced game boards. Monte Carlo simulations are done on these reduced boards [9, 36]. In addition to MCTS, MoHex runs a parallel Hex solver using Depth-first Proof Number search (DFPN) [42], able to find perfect moves if searching long enough. If the DFPN search finishes before the time limit per move, MoHex will play perfectly. MoHex 2.0 is an improvement on MoHex

[1] `http://benzene.sourceforge.net/`

which strengthens the simulations of the MCTS [38]. The simulations are weighted with prior knowledge using learned patterns from a minorization-maximization algorithm [16].

Since 2009, MoHex has won all Computer Olympiads in Hex [32]. Recent wins are however different forms of MoHex, as since 2016 when AlphaGo was introduced, research efforts went in to explore the use of neural networks for the game of Hex. Therefore, in the Computer Olympiad of 2016, MoHexNet based on NeuroHex [33, 52], was introduced and won. Since then MoHex-CNN won in 2017 [32, 21], and MoHex-3HNN - the successor - won in 2018 [23, 22].

As current research efforts explore the use of neural networks, this thesis will focus on the Tsetlin Machine. MoHex uses a custom MCTS algorithm based on various tricks, but the TM will run on a vanilla Monte Carlo tree search implementation, with the focus being on the TM's pattern recognition capabilities.

Chapter 4

Solution

This chapter will explain how supervised learning with the Tsetlin Machine was carried out and evaluated for Hex, and additionally, the process of playing the game with tree search and Monte Carlo tree search.

4.1 Overview

A systematic overview of the presented solution is illustrated in figure 4.1. The method is outlined in parts that capture a thorough insight into the TM implementation. The arrow in the top-left corner marks the start of the process.

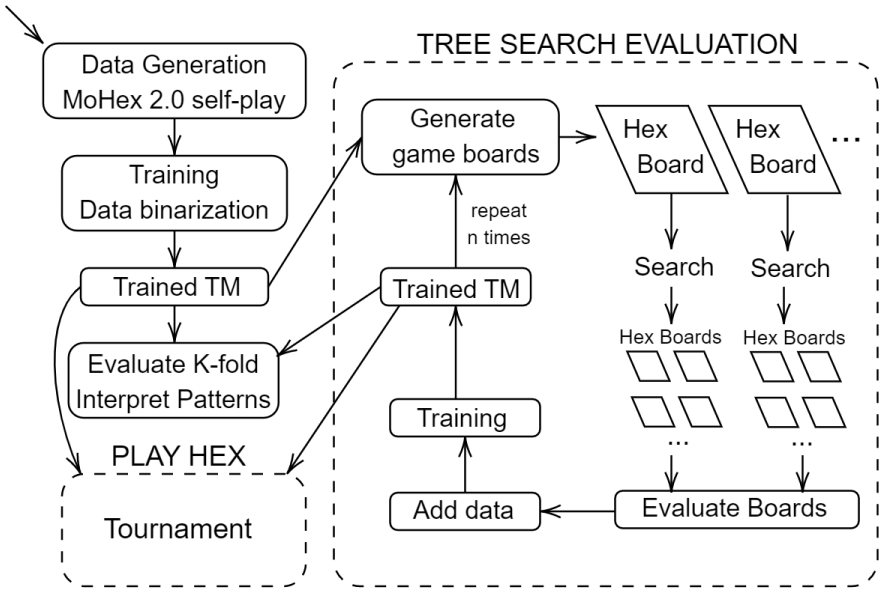


Figure 4.1: Solution Pipeline

The subsequent sections elaborate on the individual steps of the pipeline.

- **Section 4.2: Creating data** describes the first step of acquiring data necessary for supervised learning.
- **Section 4.3: Training the Tsetlin Machine** describes the process of training the TM machine, corresponding to thesis goals 1 and 2, by way of binarizing the input data. The trained TM was analyzed on account of accuracy, precision and recall.
- **Section 4.4: Separating based on moves** refers to an experiment used to find the depth required for tree search evaluation.

- **Section 4.5: Search Evaluation** elaborates on the tree search evaluation process, an important step in relation to thesis goal 3. Tree search encompass all board positions down to a given level, and served as a systematic approach for observing the TM classifier.
- **Section 4.6: Playing the game of Hex** explains how a Hex tournament, between TMs and other player, were conducted. This section corresponds to thesis goals 3 and 4 as well as hypothesis 2.
- **Section 4.7: Interpreting Tsetlin Machine patterns** explains the process of making the TM clause conjunctions illustrated as Hex boards - to be interpretable. This correlates with thesis hypothesis 1.

4.2 Creating data

In order to train the Tsetlin Machine, good data was needed. As mentioned by Gao et al. [24], previous data from MoHex 2.0 training were not available, so they generated their own by MoHex 2.0 self-play. Self-play is the concept of making an algorithm play against itself, instead of having a manual opponent. Self-play enables rapid play without the program having to wait for other input. Unfortunately, the link to the MoHex 2.0 self-play data was broken. As a consequence, a new data generation was performed. An implementation of MoHex 2.0 was found on Github[1] that allowed for modifications to generate new data.

To get winning positions from both players, all the possible starting moves for black and white were iterated over several games. The Benzene project uses x- and y-coordinates to refer to the Hex board positions. This generation process is exemplified in Algorithm 1 with pseudocode. MoHex 2.0 took over and played the rest of the games

[1] `https://github.com/cgao3/benzene-vanilla-cmake`

from move 3 onwards. Each game's board positions were saved to a dataset file on disk.

Algorithm 1 Data generation

Input Text file: *file*, Board size: *bsize*

```
1: procedure GenerateData(file, bsize)
2:     for by ← 1, ..., bsize do                    ▷ Iterate all possible positions
3:         for bx ← 1, ..., bsize do                          ▷ for black first move
4:             for wy ← 1, ..., bsize do            ▷ Iterate all possible positions
5:                 for wx ← 1, ..., bsize do                  ▷ for white first move
6:                     if bx == wx and by == wy then
7:                         continue                 ▷ Prevent placing in same spot
8:                     end if
9:                     game ← NewGame(bsize, bsize)
10:                    game.placePiece(bx, by)          ▷ Place first black piece
11:                    game.placePiece(wx, wy)          ▷ Place first white piece
12:                    winner ← MoHexSelfPlay(game)          ▷ Play out game
13:                    file.write(game.toString(), winner)          ▷ Write data
14:                end for
15:            end for
16:        end for
17:    end for
18: end procedure
```

With a board size of 5×5 there are 25 possible starting moves for black, and 24 possible continuation moves for white. By fixing the starting positions this gave 25*24 = 600 different games. Increasing the board size to 6×6 gave 36*35 = 1260 different games. A board size of 6×6 was selected for the dataset because it was a reasonable size. Using smaller board sizes, than e.g. 11×11, was much faster and MoHex 2.0 could find the perfect moves. Given that current research efforts had analysed up to 10×10 boards [42], a 6×6 board was deemed a reasonable starting point for analysis with the TM.

In the data that was generated, each feature vector represented a board position and each feature in the vector was either 'b' (black), 'w' (white) or 'e' (empty). The feature vectors consisted of features for the first row of the board, then features from the second row and so forth. Each class was the winning player, noted after the game had finished. So, each game was played until the end and every

board position in each game was assigned the class of that winner. The dataset classes were set to 'w' for white win and 'b' for black win. Table 4.1 shows the class distribution over all game boards. The dataset was assigned the name 17k dataset because it had 17880 game boards.

Table 4.1: 17k dataset class distribution

Class	Game Boards
White	1294
Black	16586
Total	17880

4.2.1 Double Dataset

As seen in table 4.1, there is an uneven distribution of the classes. Since black is the starting player he has an advantage, resulting in 16586 / 17880 = 92.8% of the classes to be his wins. This results in the TM needing a prediction accuracy of over 92.8% to be better than just selecting black class. The swap rule in Hex allows the second player to basically become the first player. This means that changing all 'b' features to 'w' and vice verca will result in new theoretically legal positions. Therefore, in order to try and balance this bias towards the first player, the dataset was doubled. By adding all changed positions and classes to the dataset, this became the 35k dataset. The 35k dataset consists of an equal amount of classes, making guessing chance equal 50%.

4.3 Training the Tsetlin Machine

This solution used a weighted multi-class TM with integer weights [5]. Remaining mentions of the TM will refer to this variant. After having generated 17k and 35k datasets as explained in section 4.2, the TM algorithm was trained in Python using the pyTsetlinMachineParallel[2] library. This library is a CPU implementation of the multi-class TM, which has support for multi-threaded performance during training making it more time efficient.

[2] `https://github.com/cair/pyTsetlinMachineParallel`

4.3.1 Data binarization

The TM is based on clauses from logic and requires the input features to be binary. Figure 4.2 shows how this process was applied.

The generated dataset consisted of features with the values, 'b', 'w' and 'e', so they could not directly be changed into bits. Instead, the feature vector was separated in two parts, represented as a composite vector of 72 bits. Two bits for each board position on the 6×6 board. The first 36 positions had bits set to 1 where black had pieces placed, and the last 36 positions had bits set to 1 where white had pieces placed. This resulted in a new feature vector in binary format. An empty position was represented by value 0 in both parts.

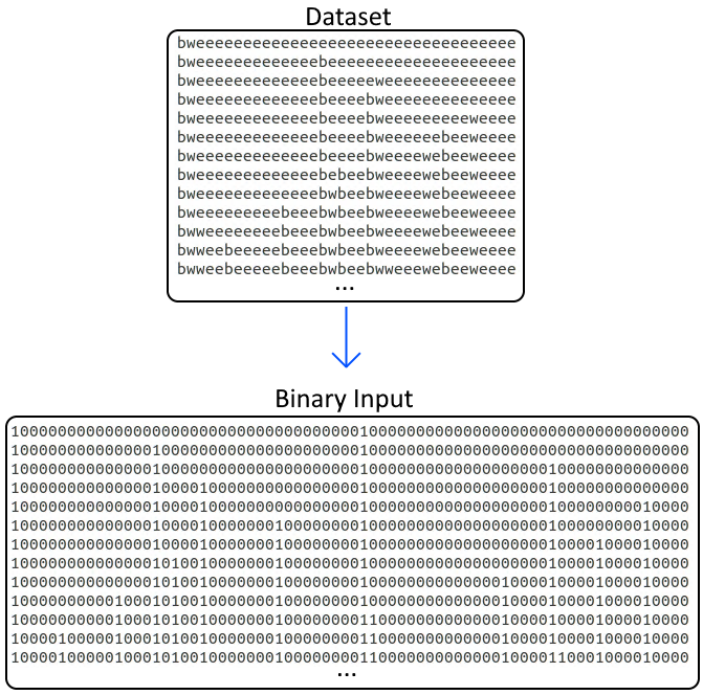


Figure 4.2: A subset of the dataset being transformed into binary format. Each input is doubled, with respect the to black and white players' pieces

The data binarization process can be exemplified with the arbitrary board position given in figure 4.3.

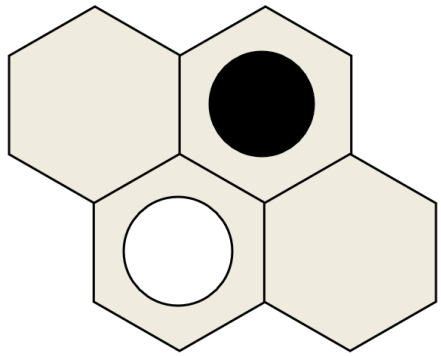


Figure 4.3: 2×2 Hex board position

The dataset from the board position in figure 4.3 is **ebwe** and would transform to binary as follows:

$$\text{ebwe} \rightarrow 0100 \text{ (black)}$$

$$\text{ebwe} \rightarrow 0010 \text{ (white)}$$

$$01000010 \text{ (final)}$$

4.3.2 K-fold Cross Validation

To determine the TM classifier's accuracy, some of the data was seperated as test data. Instead of splitting the data in e.g. 90% training and 10% test, K-fold Cross-Validation (KCV) was used to validate the classifications more robustly [29]. First, the dataset was shuffled, then it was divided into k groups. To avoid having part of a game in training and the rest of that game in testing, the data was split on whole games. This ensured that data from the same game was not split in the cross-validation groups. The testing used a k=10, so 10 groups were made. For each group, a TM was trained on 9 of the groups and tested on the remaining group. The group used for testing

was switched with one of the 9 from training until all groups had been the tested group. The KCV was stratified, meaning each group had the same ratio of classes, to ensure the testing and training data was not split dependent.

4.4 Separating based on moves

The longer a game of Hex goes one, the easier it is to tell who will win. As a first step towards tree search, a Tsetlin Machine with 500 clauses, 6000 threshold and 10 s-value was tested on data where various amounts of pieces were placed on the board. The intention was to get a better understanding of the evaluation strength of sparse or dense game boards. The 17k dataset was divided into separate datasets based on number of pieces. Individual TMs were trained on each of the sets to find their accuracy.

4.5 Search Evaluation

To play Hex with the TM, it needs to be able to help a tree search algorithm find strong continuation paths. It is crucial for the TM to guide the search towards paths that are winning. Figure 4.4 illustrates this evaluation process.

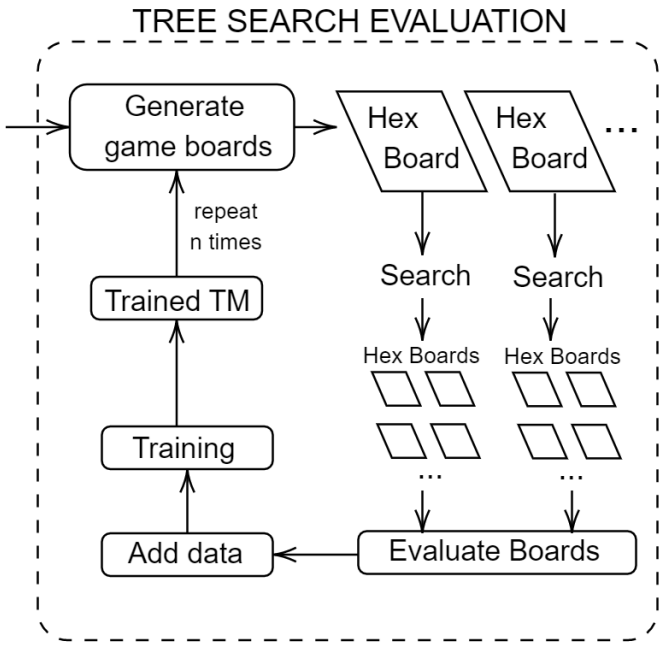


Figure 4.4: Tree Search Process

4.5.1 Dual Tsetlin Machines

So far, the TM has been used to give a binary prediction of who will win some given board position. Going further, trying to guide a search requires the TM to differentiate between strong wins, weak wins and losses. In addition, it must rank positions based on the black and white players' perspective. Currently, only an objective prediction of the winner is given. This lead to two TMs being trained instead of

one; one TM for the black player and one TM for the white player. It is game losing to start the game with a piece in the corner, for both players. Therefore, weak moves for black does not imply strong moves for white, so a single TM solution fails.

OpenSpiel did not have support for the swap rule in the Hex implementation, so the 17k dataset was used. The TM evaluates board positions *after* a player has placed their piece. Therefore, the black TM was trained on board positions where black had just placed (odd number of pieces & white's turn). The white TM was trained on board positions where white had just placed (even number of pieces & black's turn).

In order to rank positions within the same prediction class, it was possible to use the sum of the voting from conjunctive clauses in the TM, a score indicating its confidence. The weighted MTM used one TM classifier for each class; win, loss and each produced a vote total. To get the final score for a position the losing TM's votes were subtracted from the winning TM's votes. This constituted the ranking process of given board positions.

4.5.2 Tree Search

With two TMs it was possible to do tree search, evaluating board positions from both the black and white perspective. The hyperparameters set for these two TMs were 6400 clauses, 12000 threshold and 40 s-value.

The tree search was executed with stochasticity, meaning the starting board positions, where the search was to begin, were randomly created. An integer between 2 and 15 for the number of pieces were randomly determined, and then pieces were placed arbitrarily on the board until it matched this integer. The fastest a game of 6×6 Hex can end is in 11 moves (6 black moves and 5 white moves). Not all games end this quickly, so the top interval value was set a certain amount of moves higher than this. From this position, a breadth-first approach was used to find all possible paths in the tree - on the 3rd level, see figure 4.5.

A depth of 3 was chosen based on some factors. Firstly, the test described in section 4.4. Secondly, the level of the search had to coincide with the game's turn alternations. Because the TM evaluates after a move is done, even numbered depths like 2 or 4 would not work. On these depths the opponent's last move would be evaluated, i.e. not the correct player, meaning the depth had to be an odd number. Lastly, it was computationally feasible - where a higher depth of 5 was to slow to compute. At the start of the game, a depth of 3 will expand to 36*35*34 = 42840 possibilities. Finding all of these possibilities and evaluating them with the TM took around 2 seconds. A depth of 5 however expands to 36*35*34*33*32 = 45239040 possibilities, approximately 1000 times larger.

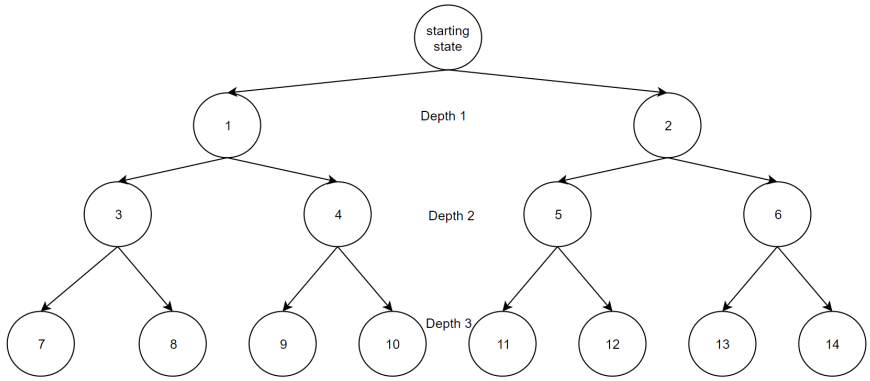


Figure 4.5: Example tree search down to 3rd level. Nodes are numbered in the way they are searched with the breadth-first search

When the 3rd level was reached, the TM evaluated all the possible board positions, giving each of them a score. The top 100 scoring positions were selected for analysis. As such, these were saved to a file on disk. The file contained the top position's score and the TM predicted class. Additionally, MoHex 2.0's DFPN search was utilized to output correct win class to the file.

The tree search was executed in iterations and each iteration was decided to involve 100 stochastic boards. Therefore, each iteration created 100 individual files. Figure 4.6 shows this file structure on disk with some dummy values.

File 1

Score	TM	Mohex
0.989	1	1
0.979	1	1
0.969	1	1
0.959	1	1
0.949	1	1
0.939	1	1
...	...	...

File 2

Score	TM	Mohex
0.888	1	1
0.852	1	1
0.769	1	1
0.732	1	1
0.712	1	1
0.695	1	1
...	...	...

...

File 99

Score	TM	Mohex
0.205	1	1
0.180	1	1
0.143	1	0
0.082	1	1
0.042	0	1
0.002	0	0
...	...	...

File 100

Score	TM	Mohex
-0.345	0	0
-0.362	0	0
-0.385	0	1
-0.421	0	0
-0.456	0	1
-0.467	0	0
...	...	...

Figure 4.6: One iteration of the tree search's file structure on disk

The TM scores are the sum of the TM clause voting process and each clause is multiplied by its corresponding weight coefficient. This

means that the real values can be quite high, often in the tens of thousands, depending on the size of the weights. Therefore, the scores in the files were normalized between -1 and 1, as in the exemplary figure 4.6.

4.5.3 More data

When performing tree search, as explained in section 4.5.2, it was determined the data in the 17k dataset was insufficient. Random search revealed new board positions not previously seen in the dataset, resulting in low and sometimes negative TM scores for (actual) winning positions. To resolve this, the files on disk from tree search was used as additional data. After 27 tree search iterations, where each iteration generated 100 board positions, a total of 2700 were generated. Finding 100 top scoring future positions for each gave 270000 new positions. Combined with the 17k dataset, this became the 287k dataset.

Table 4.2: Class distribution for the 287k dataset

Class	**Game Boards**
Black	175968
White	111826
Total	287794

4.5.4 Tree Search Evaluation

To analyse the TM's ability of finding winning paths in the tree search, recall and precision graphs were made. Recall was calculated with the formula:

$$Recall = \frac{TP}{TP+FN}$$

And precision was calculated with the formula:

$$Precision = \frac{TP}{TP+FP}$$

Since the files from tree search had information about TM prediction and MoHex 2.0 true value, it was possible to calculate a confusion

matrix and derive the values for true positive (TP), false negative (FN) and false positive (FP).

The boards, created stochastically, could start off in losing positions before the tree search even began. With the recall formula these boards were ignored, as recall only focuses on positives (the positions that are winning). Recall will reflect the TM's ability to not predict true wins as loss, and a high value will mean it is adept at this task.

Precision was calculated to show how accurate the TM was at not predicting true loss as win. In turn, a high precision would mean the TM was good at filtering out losses in presented board positions.

Looking at an increasing number of top positions from each file, several values of recall and precision was calculated. The first recall, R_1, and the first precision value, P_1, was calculated by only including the highest scoring position from each of the files. R_2 and P_2 was calculated by looking at the top two positions from each file. This was continued until R_{100} and P_{100} that included all the positions from all the files. Figure 4.7 illustrates how R_5 and P_5 was calculated by including only the top 5 positions:

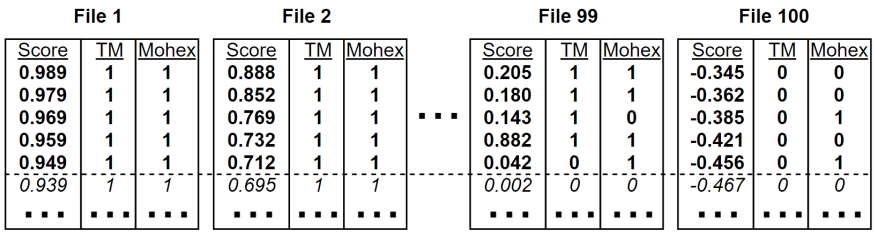


Figure 4.7: R_5 and P_5 including only the top 5 positions from each file

With the dummy values from figure 4.7, a confusion matrix would look like table 4.3.

Table 4.3: Confusion matrix for top 5 scores of file 1, 2, 99 and 100

		True value	
		Win(1)	Loss(0)
Prediction	Win(1)	13 TP	1 FP
	Loss(0)	3 FN	3 TN

Calculating the Recall from this matrix gives:

$$R_5 = \frac{13}{13+3} = 81.25\%$$

Calculating the Precision from this matrix gives:

$$P_5 = \frac{13}{13+1} = 92.86\%$$

4.5.5 Hyperparameter optimization strategy

The method for hyperparameter optimization used for this thesis was:

1. Setting threshold and s-value to some guesstimated numbers.
2. Varying the number of clauses by halving and doubling the value until a satisfactory accuracy was found.
3. Taking that best clause value and varying the threshold and s-value using halving and doubling.

This method was stopped after a reasonable amount of time, because it could be repeated indefinitely with exponentially lower returns, and time was better spent elsewhere.

4.6 Playing the game of Hex

Having finished training the TM with new data, the game could be played. Deepmind's OpenSpiel[3] framework [40] supports the game

[3] `https://github.com/deepmind/open_spiel`

of Hex and was used to get access to legal moves to choose from. Additionally, it provided easy integration with the Monte Carlo tree search algorithm [4]. This implementation is based on the UCT variant by Kocsis and Szepesvári [39].

The evaluator method used to rank non-terminal nodes during the MCTS was overriden in the implementation, in order for the TM to score the nodes during search. Algorithm 2 shows this evaluation procedure, with the UCT constant set to $\sqrt{2}$.

Algorithm 2 MCTS Evaluator

Input Board position: *brd*
Output Board position *score*

```
1: procedure EVALUATE(brd)
2:     pieces_placed ← CountPieces(brd)          ▷ Count non-empty hexes
3:     turn ← (pieces_placed%2 == 0)             ▷ Figure out whose's turn
4:     tm ← SelectTM(turn)                       ▷ Select black TM or white TM
5:     binary_brd ← BinarizeBoard(brd)           ▷ Turn into binary input
6:     predictions, votes ← tm.predict(binary_brd)   ▷ Give input to TM
7:     win_votes ← votes[1]          ▷ Extract votes from TM for each class
8:     loss_votes ← votes[0]
9:     vote_diff ← win_votes − loss_votes
10:    score ← vote_diff /100000                 ▷ Normalize value
11:    score ← max(score, −1.0)
12:    score ← min(score, 1.0)
13:    return score
14: end procedure
```

Since two TMs were employed, it was necessary find whose turn it was to let the respective TM evaluate the position. This was achieved by counting number of pieces on the board, where an even number meant black player's turn and an odd number meant white player's turn.

The TM votes could be in the tens of thousands, so they were normalized between the values of -1.0 and 1.0 by dividing them by 100000. This was set as an upper and lower bound for the votes based on some high votes from tree search (section 4.5). Normalization was a critical

[4] `https://github.com/deepmind/open_spiel/blob/master/docs/algorithms.md`

step as without this step MCTS was behaving very poorly. This was due to board positions being indistinguishable for the algorithm as all were above 1.0, the best score.

4.6.1 Tournament

To evaluate the playing strength of the Tsetlin Machine Hex players, a tournament was played. In the tournament, players using the 287k dataset and the 17k dataset took part, with tree search and MCTS. The MCTS players used 10000 Monte Carlo simulations per move. In addition, since the 287k dataset was created based on randomly generated game boards, the TMs were pitted against two non-TM players. One player doing completely arbitrary moves, and one player using the MCTS algorithm with random rollouts as an evaluation strategy. A rollout is a simulation to the end of the game to find a winner. This information was used to score the evaluated board positions. The random rollout player performed 1000 Monte Carlo simulations and did 20 random rollouts for each board evaluation.

Tree search players participated in the tournament to test hypothesis 2. They played by ranking all board positions on the 3rd level and then selected the one with the highest score. The move on depth 1 from this board position was chosen as the move. For these players it was important that the top scoring board position was a win. Therefore a high first precision value, P_1, was important for these players.

4.7 Interpreting Tsetlin Machine patterns

The last yet important part of the solution was to interpret the TM patterns. The strength of the TM is to clearly visualize the patterns that it learns.

Each clause consists of a team of Tsetlin Automata, with two automata responsible for each position on the Hex board. Tsetlin Automata learns to include or exclude negated and non-negated features.

By querying each Tsetlin Automaton's learned action, the clause conjunction could be extracted.

For example on a 2×2 Hex board, there are 4 possible positions for pieces to be places. In other words the positions can be expressed with a vector of 4 features, with each feature corresponding to a board position. In fact, during the binarization step to fit the input to the TM's requirements, the positions were doubled. Meaning there is one vector for possible positions of black's pieces and one vector for possible positions of white's pieces. This means the composite input vector must have 8 features. As a consequence, there are 8*2 = 16 TA created - one for negated and one for non-negated. Figure 4.8 illustrates how these TA correspond to the Hex board of size 2×2.

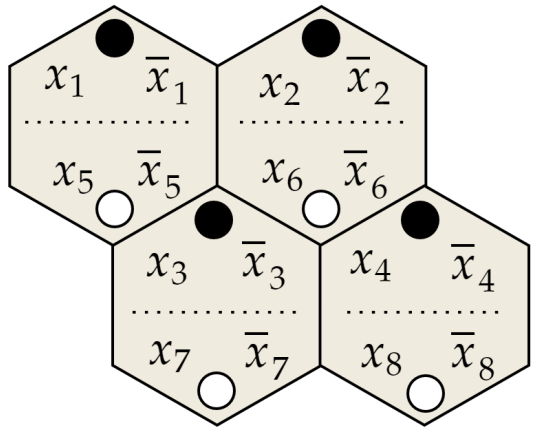


Figure 4.8: The 16 TA for negated and non-negated features corresponding to the positions on the 2×2 Hex board. Features x_{1-4} for black pieces and x_{5-8} for white pieces

As such, a Hex board can be created for each clause to illustrate where the pattern determines it is beneficial to have pieces and where it is inferior. Querying each TA's learned action in figure 4.8 might for example reveal the pattern in figure 4.9

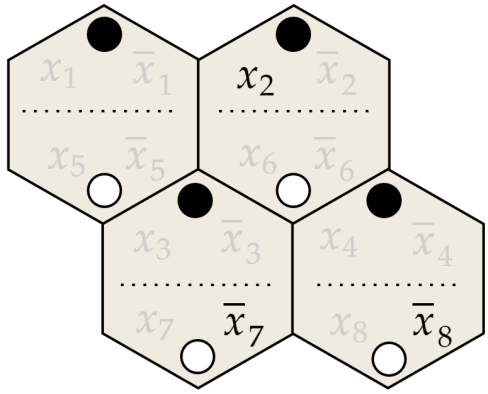


Figure 4.9: Querying the actions for the TA responsible for a 2×2 Hex board reveals the learned conjunction: $x_2 \wedge \bar{x}_7 \wedge \bar{x}_8$

Finally, the pattern can be cleaned up and be illustrated as in figure 4.10. The blank position is ignored by the pattern, while a red X means that this piece must not be in that position.

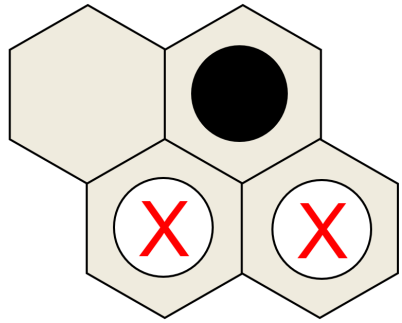


Figure 4.10: A more easily understandable pattern form based on learned actions in figure 4.9

The clauses with high weights are interesting because they have a higher likelihood of being true positives. Here it was important to distinguish even and odd numbered clauses. Only even numbered clauses voted in favor of the class, while odd numbered clauses were counter-examples for the class. Figure 4.10 might be an example of an even numbered clause in favor of win for black.

Chapter 5

Results

This chapter will go over the findings based on the implementations described in Chapter 4. Results are presented on the experiments of prediction rate on a given Hex board, the quality of tree search, pattern interpreting and on the success of playing Hex with the Tsetlin Machine using MCTS and tree search.

5.1 Classification Accuracy

The following subsections will present prediction accuracies, precision and recall with 10-fold cross validation for the datasets explained in Chapter 4. Single TM means one TM trained on all data and then evaluated. Dual TMs means one TM for black player and one for white player. All these TMs have 3 hyperparameters - clauses, threshold and s-value - that must be balanced in order to achieve the optimal results on any given data. This can be time consuming to achieve and the TMs presented in this chapter has not been fully optimized.

5.1.1 Single TM on 17k dataset

The 17k dataset consists of 16586 winning lines for black and 1294 winning lines for white. Guessing chance equals $16586/17880 = 92.8\%$. A single TM is trained and the results for the second step of optimization (section 4.5.5) is shown in table 5.1. The final optimization step is shown in table 5.2. The tables include precision and recall for black being the positive class in the confusion matrix.

Table 5.1: Varying clauses for TM trained on 17k dataset with 6000 Threshold and 10 S-value

Clauses	**Precision**	**Recall**	**Accuracy (%)**
500	0.98	1.00	97.58 ± 0.80
1000	0.98	1.00	97.52 ± 0.93
1500	0.98	0.99	97.37 ± 1.74
1600	0.98	1.00	97.40 ± 1.09
1700	0.98	0.99	97.55 ± 1.04
2000	0.98	0.99	97.74 ± 0.90

Table 5.2: Accuracy with various Thresholds and S-values for 500 clauses trained on 17k dataset

	10 s	**15 s**	**20 s**
5000 t	97.09 ± 1.21	96.92 ± 2.15	96.74 ± 1.68
6000 t	97.59 ± 1.21	97.09 ± 1.61	96.76 ± 1.80
7000 t	97.28 ± 0.82	97.21 ± 0.89	96.80 ± 0.96

5.1.2 Single TM on 35k dataset

Table 5.3 shows a single TM trained on the 35k dataset. The table includes precision and recall for black being the positive class in the confusion matrix.

Table 5.3: Varying clauses for TM trained on 35k dataset with 6000 Threshold and 10 S-value

Clauses	Precision	Recall	Accuracy (%)
500	0.90	0.85	87.73 ± 1.54
1000	0.89	0.88	88.50 ± 1.17
1500	0.97	0.99	96.83 ± 1.78
1600	0.97	0.99	97.11 ± 1.03
1700	0.98	0.99	97.21 ± 0.94
2000	0.97	1.00	96.78 ± 1.43

Table 5.4 shows the final optimization step of table 5.3 for 1700 clauses. Different hyperparameter values are explored to see if the clause with the highest accuracy could be improved upon.

Table 5.4: Accuracy with various Thresholds and S-values for 1700 clauses trained on 35k dataset

	10 s	**15 s**	**20 s**
5000 t	96.54 ± 1.18	96.61 ± 1.45	96.77 ± 1.53
6000 t	97.21 ± 0.94	96.76 ± 0.93	97.10 ± 2.01
7000 t	96.54 ± 2.25	96.74 ± 1.22	96.45 ± 1.08

5.1.3 Dual TMs on 287k dataset

Table 5.5 shows the class distribution for each of the TMs, black and white, split by the 287k dataset.

Table 5.5: Class distribution of Dual TM on 287k dataset

(a) Black TM

Class	Game boards
Win	99645
Loss	39017
Total	138662

(b) White TM

Class	Game boards
Win	71515
Loss	59737
Total	131252

The hyperparameters for the dual TMs are optimized for black and the same values are used for white. Table 5.6 shows the second optimization step for the 287k dataset.

As seen in table 5.6, the TMs that ran for 100 epochs were overfitted. Overfitting means to be overspecialized on the training data, such that the TM would perform poorly on non-identical data, like what was used for testing. The following TMs were therefore trained to 50 epochs and not 100.

The tree search evaluation was performed using 6400 clauses, 12000 threshold and 40 s-value, performing worse than the TM with 9600 clauses and 12800 clauses. The difference between 9600 and 12800 was not as significant as between 6400 and 9600, so the one with 9600 clauses was selected when choosing players for the Hex tournament, as a middle-ground. The highest performing TM with 9600 clauses was selected from the values shown in table 5.7

Table 5.6: Varying clauses for Black TM trained on 287k dataset with 12000 Threshold and 40 S-value

	100th epoch			**50th epoch**
Clauses	**Precision**	**Recall**	**Accuracy (%)**	**Accuracy (%)**
500	0.87	0.92	84.44 ± 0.98	85.09 ± 0.77
1700	0.90	0.93	87.33 ± 1.02	88.21 ± 1.15
3200	0.90	0.94	88.48 ± 1.17	89.60 ± 1.43
6400	0.93	0.92	88.89 ± 2.44	90.39 ± 2.41
9600	0.93	0.92	89.45 ± 1.73	91.19 ± 2.10
12800	0.93	0.93	89.95 ± 2.55	91.28 ± 2.21

Table 5.7: Accuracy for Black TM with various Thresholds and S-values for 9600 clauses trained on 287k dataset up to 50th epoch

	10 s	**15 s**	**20 s**	**40 s**
3000 t	81.41 ± 1.42	84.89 ± 1.42	85.46 ± 1.31	88.13 ± 1.81
6000 t	83.01 ± 2.21	85.73 ± 2.30	86.49 ± 2.57	89.30 ± 2.60
9000 t	83.78 ± 2.99	85.75 ± 3.00	86.68 ± 3.55	90.25 ± 2.54
12000 t	83.61 ± 2.32	86.59 ± 2.06	87.67 ± 1.31	91.19 ± 2.10

5.2 Separating Moves

The results from the experiment on separating moves (section 4.4) is shown in figure 5.1

A TM with 500 clauses, 6000 threshold and 10 s-value was trained on subsets of the 17k dataset, separated by moves. As seen in figure 5.1, the accuracy was low for 2 moves, but quickly increased as more moves were done and more pieces allowed for better clause patterns. The figure decreases as even more moves were done, as data from these number of moves were sparse. The final spike in the figure can thus be explained by the data having few games with 26 moves, and that those that were trained on were similar.

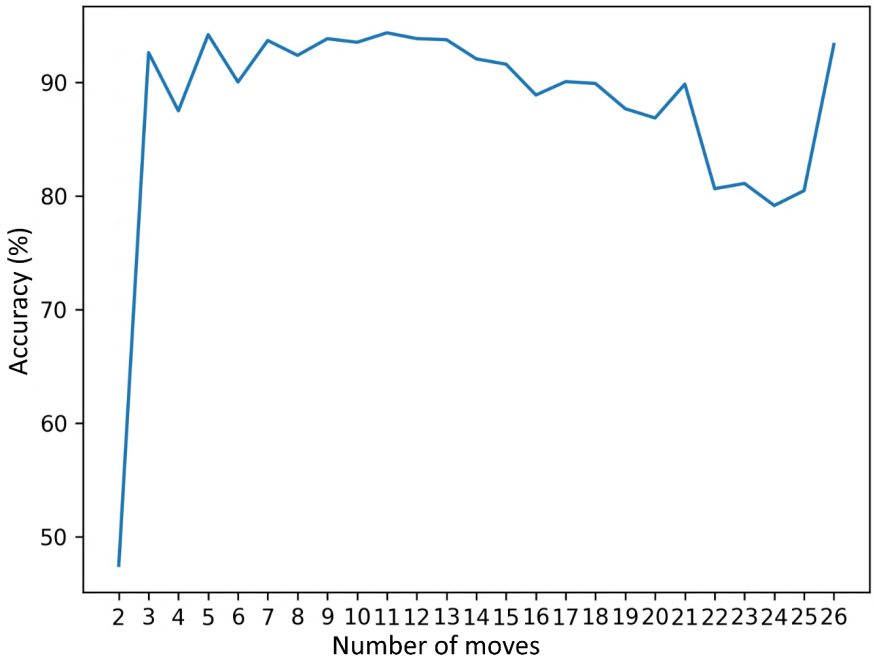


Figure 5.1: Accuracy of TM based on training with separated moves

5.3 Search Evaluation

This section is related to solution section 4.5, tree search, and will display results from this implementation. It will show TM's improvement in search capability, caused by the stochastic generation of new data, shown by the recall and precision graphs.

5.3.1 Recall Graphs

The first iteration of the search was using a TM trained on the 17k dataset. 100 stochastic board positions were generated and classified by the TM. The corresponding recall graph is shown in figure 5.2.

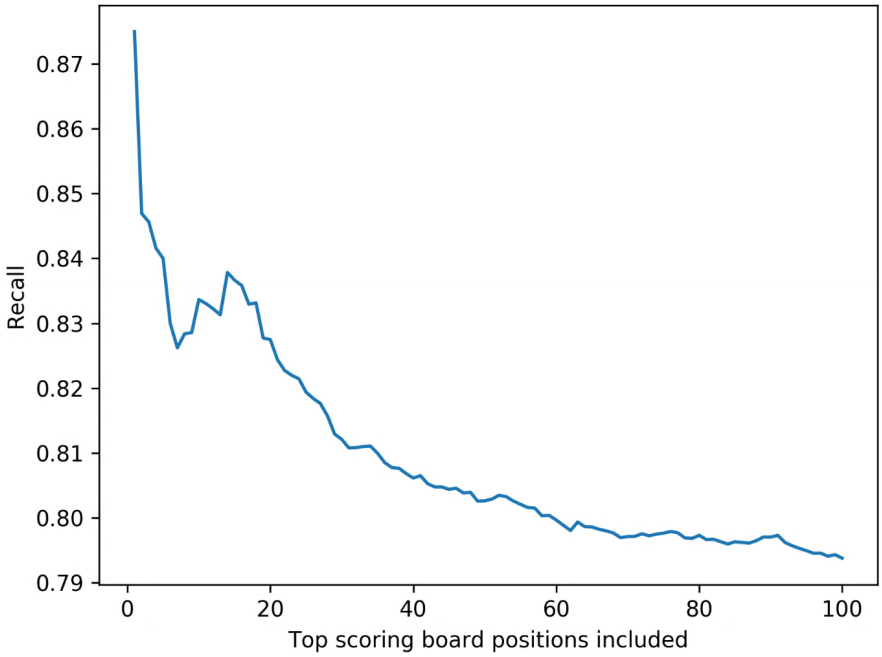


Figure 5.2: Recall graph for 1st iteration of tree search evaluation

As more top positions were included in the recall calculations the percentage value sank, as seen by the slope of figure 5.2 becoming gradually lower. Only the top 100 ranking board positions were evaluated, yet the graph's slope indicates that it converges at some point, given more top board positions. The spike between R_{10} and R_{20} can be explained by randomness. This indicates most of the positions were completely new to the TM at this point, making it unsure.

Figure 5.3 shows the recall graph from the second tree search iteration. This TM was trained on the previous board positions from the first iteration in addition to the 17k dataset. The graph increased with around 10% from figure 5.2, a strong improvement from the previous iteration, indicating more data improves the TM.

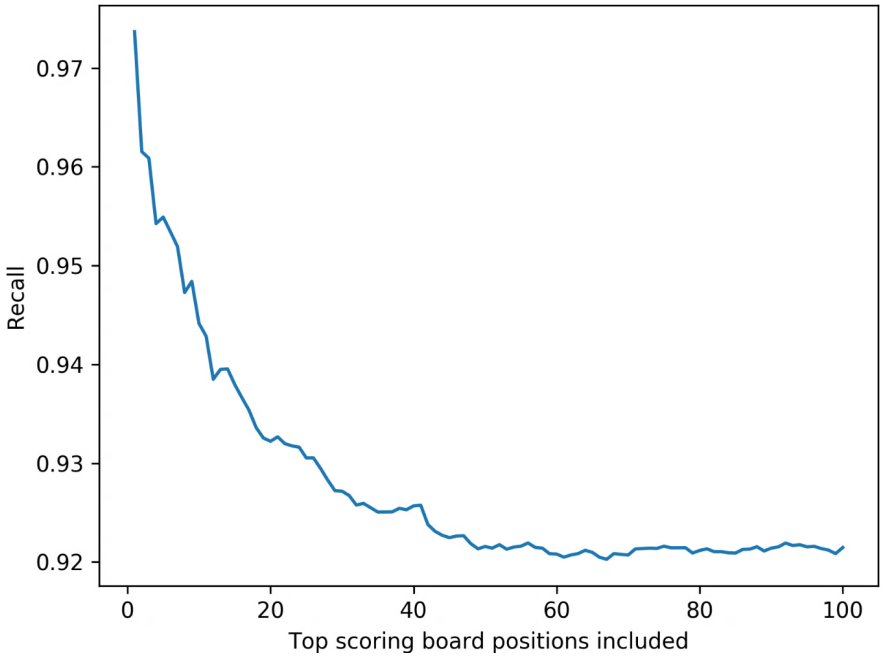


Figure 5.3: Recall graph for 2nd iteration of tree search evaluation

After ten tree search iterations, figure 5.4, the recall appears to not increase. The value is slightly lower than in the second iteration, but this may be attributed to randomness. This TM was trained on the board positions from all previous iterations in addition to the 17k dataset.

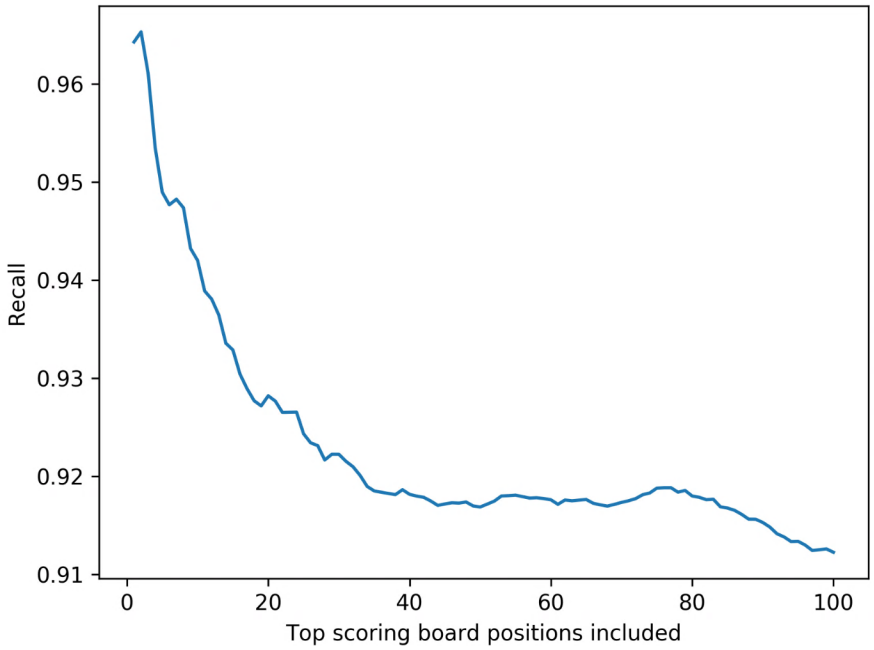


Figure 5.4: Recall graph for 10th iteration of tree search evaluation

Continuing the search to the 27th tree search iteration seen in figure 5.5, indicates the recall value continuously increase - however more subtly. The first recall value, R_1, has taken slower and more granular steps. Nevertheless, the lower values from R_{15-100} has increased a fair bit and the gap between R_1 has narrowed. This may indicate that the TM has become more rigorous in its evaluation. Instead of only correctly classifying a couple of top scoring board positions, it gets more of the winning positions correct. The TM in iteration 27 was trained on the 26 previous iteration's data in addition to the original 17k dataset.

By looking at the overall improvement from the 1st iteration (figure 5.2) to the 27th iteration (figure 5.5), all recall values, R_{1-100}, have improved greatly. This can be interpreted as the TM having learned important patterns to recognize wins for a wider range of board positions. The TM trained on the 17k dataset has a focused awareness of only the best moves to take from MoHex 2.0 self-play. The stochastic board positions in the 287k dataset gives the TM a larger repertoire to form valuable clauses.

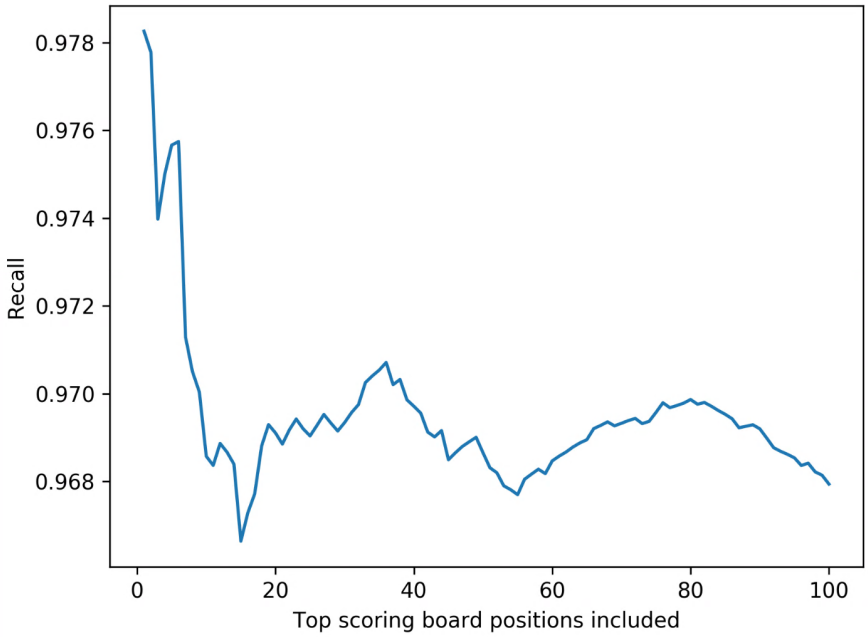


Figure 5.5: Recall graph for 27th iteration of tree search evaluation

5.3.2 Precision Graphs

The following precision graphs are calculated from the same confusion matrices as the recall graphs in subsection 5.3.1. Figure 5.6 has around chance (50%) precision, resulting in the TM not recognising that about half of the top scoring positions were losses and still classified them as wins. This indicates that the TM had not seen these board positions before, and was therefore not able to differentiate win from loss.

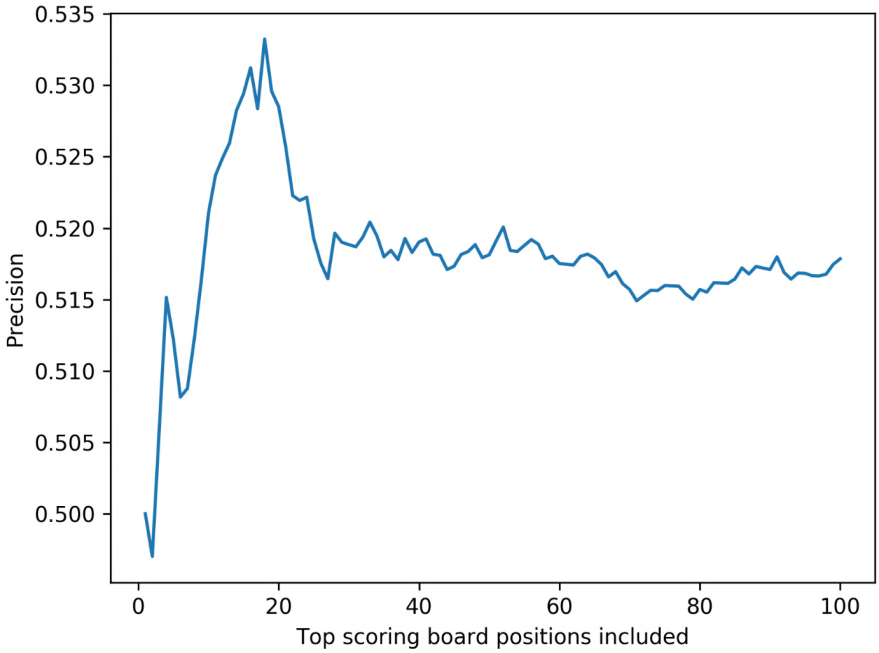


Figure 5.6: Precision graph for 1st iteration of tree search evaluation

Figure 5.7 shows the precision graph from the 2nd tree search iteration. The graph has a 25-30% increase in precision depending on how many positions were included. This indicates that adding generated data helped the TM in not classifying loss as win.

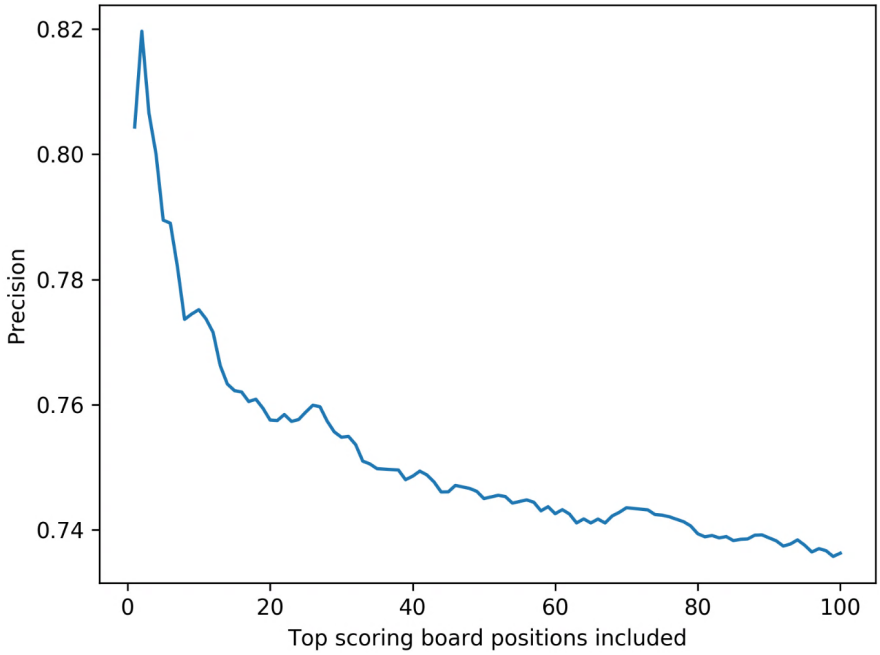


Figure 5.7: Precision graph for 2nd iteration of tree search evaluation

Figure 5.8 shows the graph from the 10th tree search iteration. The precision has increased, but with less gain than from iteration 1 to iteration 2.

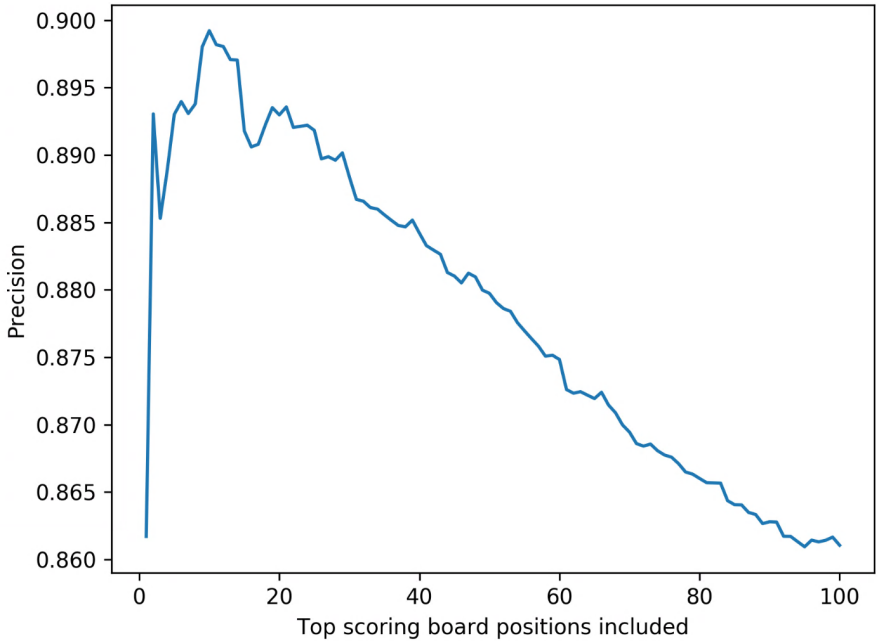


Figure 5.8: Precision graph for 10th iteration of tree search evaluation

Figure 5.9 shows the graph from the 27th tree search iteration. The precision value has not converged yet, indicating that adding more training data would increase precision some more, the same observation made in the 27th recall graph, figure 5.5.

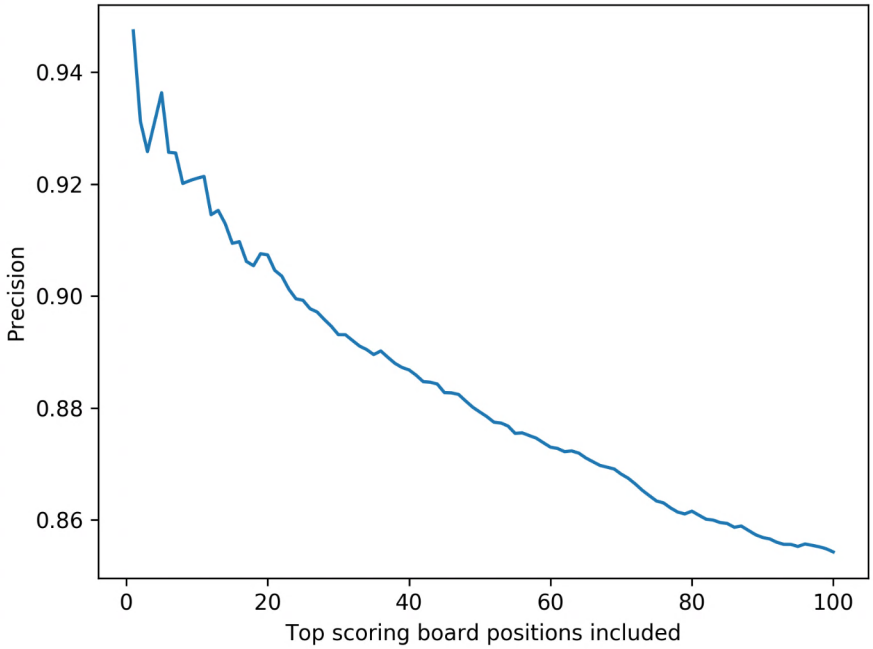


Figure 5.9: Precision graph for 27th iteration of tree search evaluation

5.4 Playing Hex

This section presents the results from the Hex tournament. The non-TM players in the tournament were the following:

- **Random (Rand.):** Arbitrary moves
- **RandomRollout (RR):** Used rollouts as an evaluation strategy in MCTS for non-terminal nodes.

The TM players, with corresponding hyperparameters and playing method, are shown in table 5.8. Each player was pitted against all other players in 10 games, each getting the chance to play as both black and white. Tournament results are shown in table 5.9. Table 5.10 show the average number of pieces placed on the board from the 10 games. Random vs Random and RR vs RR is not of interest and was not done.

Table 5.8: Tsetlin Machine players in the Hex Tournament

TM	Clauses	Threshold	S-value	Dataset	Method
1	500	6000	10	17k	MCTS
2	500	12000	40	287k	MCTS
3	9600	12000	40	287k	MCTS
4	500	6000	10	17k	Tree Search
5	9600	12000	40	287k	Tree Search

Table 5.9: Tournament Results. Playing as black in rows and as white in columns

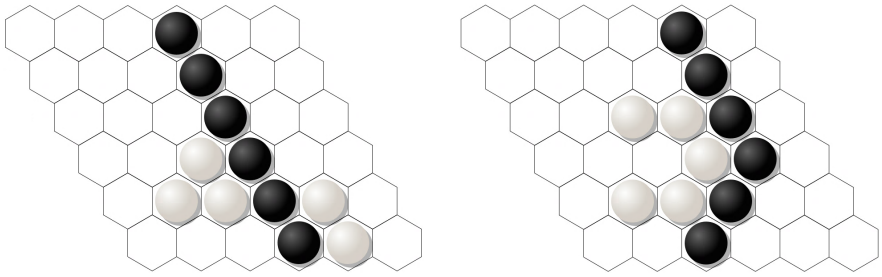
Players		**W H I T E**						
		Rand.	RR	TM1	TM2	TM3	TM4	TM5
B	Rand.	—	0-10	1-9	0-10	0-10	8-2	6-4
L	RR	10-0	—	10-0	10-0	10-0	10-0	10-0
A	TM1	10-0	0-10	10-0	10-0	5-5	10-0	10-0
C	TM2	10-0	0-10	10-0	10-0	10-0	10-0	10-0
K	TM3	10-0	0-10	10-0	10-0	9-1	10-0	10-0
	TM4	5-5	0-10	0-10	0-10	0-10	0-10	0-10
	TM5	5-5	0-10	0-10	0-10	0-10	10-0	10-0

Table 5.10: Average number of pieces placed between players in all 10 games

Players		**W H I T E**						
		Rand.	RR	TM1	TM2	TM3	TM4	TM5
B	Rand.	—	12.0	18.5	16.4	19.2	32.4	34.6
L	RR	11.2	—	11.0	11.6	11.6	11.0	11.0
A	TM1	19.6	12.6	21.2	16.0	20.9	25.8	17.0
C	TM2	11.4	12.4	14.4	11.0	11.0	20.6	11.8
K	TM3	18.6	12.2	17.0	17.0	18.1	23.8	24.8
	TM4	33.9	12.2	16.0	23.6	14.0	36.0	36.0
	TM5	32.3	12.6	17.8	24.8	18.4	35.0	35.0

5.4.1 RandomRollout player

As seen in table 5.9, the RR player played very well and lost no games. Table 5.10 shows that these games finished quickly, usually after 11 or 12 moves. This was the smallest amount of pieces needed on the board to win as black or white respectively. The RR player employed a basic yet effective strategy by building a connection piece by piece. It did not use any VCs like the bridge. Figure 5.10 below shows two finished games played by RR against TM3 and TM4. TM3 was cut off early, while TM4 indicates wanting to capture the center, having no concern for the edge positions.



(a) RR (black) vs TM3 (white) (b) RR (black) vs TM4 (white)

Figure 5.10: Finished games showing RR strategy

5.4.2 Tree Search players

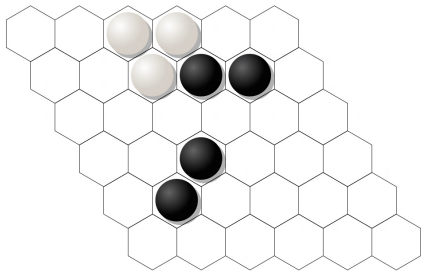
It became clear that the tree search players placed many pieces on the board that resulted in long games. They played an average of 35 and 36 moves against themselves as seen in table 5.10, essentially covering the entire board. Apart from when playing against RR, TM4 had averages exclusively over 20 moves as white. Table 5.9 clearly points out that TM4 and TM5 was the worst players out of all the TM players in the tournament.

TM4 and TM5 placed a similar amount of pieces against Random. As black, Random won the majority of these games, while as white Random won half. Results show that all these games covered at least 32

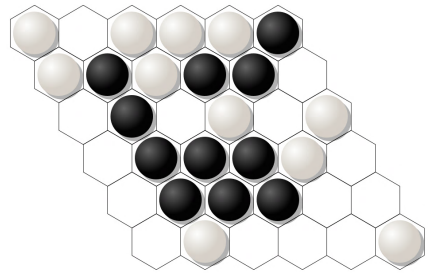
spaces on average, seemingly arbitrary placements, implying neither tree search player had a good winning strategy for ending the game.

The precision graph in figure 5.9 indicates a starting precision of P_1 = 94.7%, a high percentage. However, when performing tree search and selecting only the top scoring board position, the probability for this board position to remain as a win diminishes. For example, doing tree search ten times in a row will give a rough estimate of 0.947^{10} = 0.58. This means that when selecting the tenth move, there was a 58% chance of this move still remaining as a winning position.

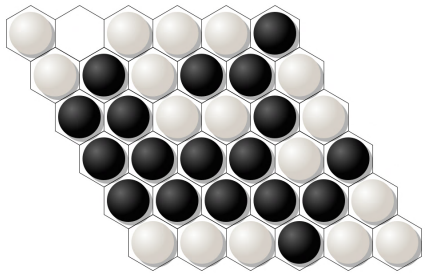
Figures 5.11 and 5.12 look at some the games Random played against TM4 and TM5, in order to get an understanding of their inadequate play. Subfigures a-c show three board positions from a single game, where the tree search player won, and subfigure d shows a game where Random won by luck. The tree search players both played as black.



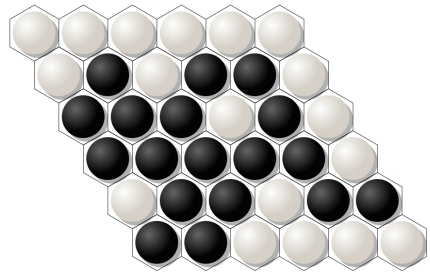
(a) TM4 started out well with bridges in place to secure the win



(b) However it did not capitalize and continued to place in the middle



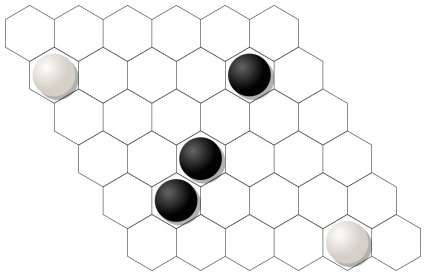
(c) Empty spots got filled one after the other, until TM4 was forced to place on the edge and win



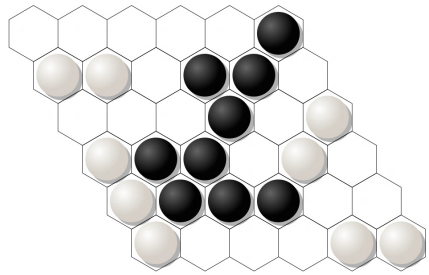
(d) Different game where Random won by making a connection in the top row, filling the board

Figure 5.11: TM4 (black) vs Random (white)

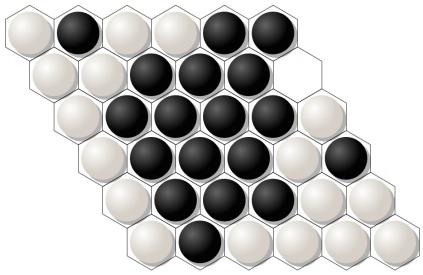
The games indicate that TM4 favored the middle positions when available, mostly disregarding his own edges, as also seen in figure 5.10b. Likewise, figure 5.12 show the same effect for TM5.



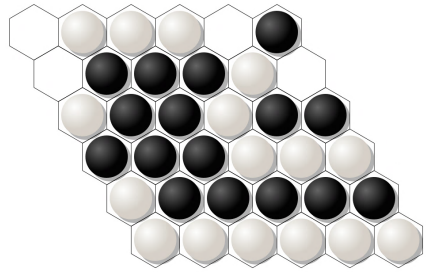
(a) TM5 started well



(b) However the bottom (and winning) edge position was not filled



(c) Only two spots were left before TM5 placed on the bottom edge position to win



(d) Different game where white randomly placed in the bottom row to win by luck

Figure 5.12: TM5 (black) vs Random (white)

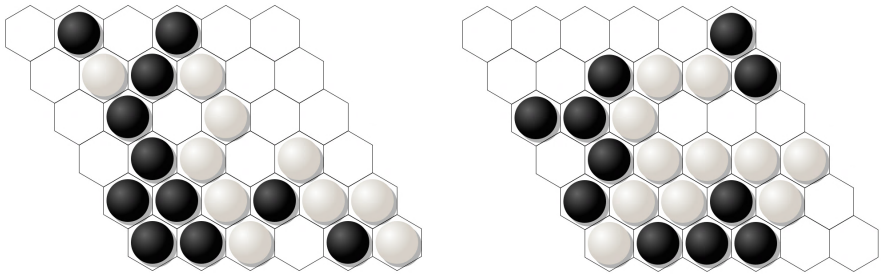
The figures 5.11d and 5.12d shows win by Random on the top and bottom edges. It appears that when the tree search players favored the center, Random would get lucky and place connected pieces on the edges, leading to wins.

5.4.3 MCTS players

Tournament results show the MCTS players performed much better, in comparison to the tree search players, against Random. Apart from one game facing TM1, Random lost all these games. This indicates that combining the MCTS algorithm with the TM was effective at creating more systematic play.

The TM inference is deterministic, so the tree search players played without randomness. Consequently, all the games played by TM4 against TM5 and by TM5 against TM4 were identical. The same trained TM was used in TM3 and TM5, and in TM1 and TM4. This TM gave the same scores for the same board positions. Yet, identical games were not seen by these pairs, with the MCTS players being strengthened by random Monte Carlo simulations. Pure tree search was chosen to show the TM's ability without any search algorithms help. The resulting improvement with MCTS supports thesis hypothesis 2, that a plain TM would not be adequate.

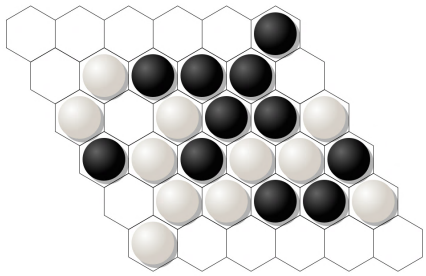
In the tournament, TM1 was one of the players using the smaller 17k dataset and fewer clauses. It lost once to random and only managed to win half of the games against TM3 when playing as black. Figure 5.13a shows the lost game played by TM1 against Random, while figure 5.13b shows one winning game against Random. It seems that the lost game was due to Random placing a blockade in the bottom-left corner. TM1 made a vertical path in the middle in both cases, but in the winning game the bottom-left corner was open.



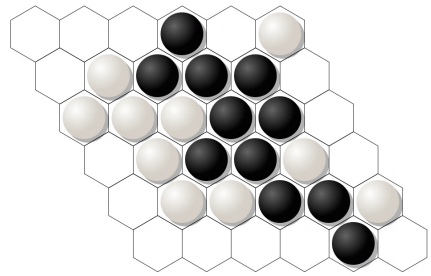
(a) Random (black) vs TM1 (white) (b) Random (black) vs TM1 (white)

Figure 5.13: Finished games played by TM1 against Random

Figure 5.14 shows two finished games between TM1 and TM3, where TM1 played as black. TM1 lost half of these games and the figure shows one loss and one win. It seems that the critical move for black was to connect the upper-middle pieces with the pieces near the bottom. If white placed here first TM3 would win, and TM1 seemed to place in this gap only 50% of the time.



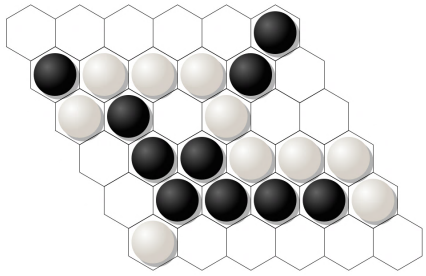
(a) TM1 (black) vs TM3 (white)



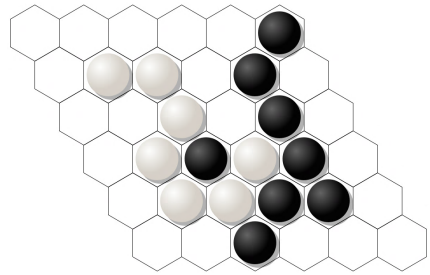
(b) TM1 (black) vs TM3 (white)

Figure 5.14: Finished games played by TM1 against TM3

TM3 lost one game as black playing against itself. Figure 5.15 shows the one lost game and another won game. The loss may be due to randomness with Monte Carlo simulation, where the lost game sampled inferior board positions compared to the won game.



(a) TM3 (black) vs TM3 (white)



(b) TM3 (black) vs TM3 (white)

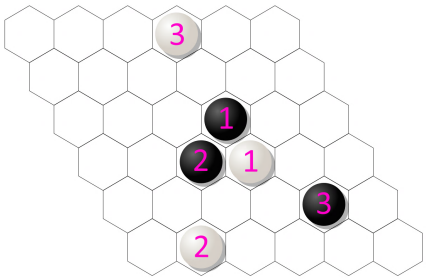
Figure 5.15: Finished games played by TM3 against itself

All these figures imply the MCTS algorithm was able to alleviate the tree search players' problem in that they were too focused on the center. An important difference between the MCTS players and the tree search players, was that MCTS would score finished board positions by giving them the highest or lowest score possible. This meant that positions on the edges, leading to wins, would be picked by MCTS players. On the other hand, tree search would rely solely on the TM's top score. In Hex it is strong to start in the middle, and it was observed that the TM learned the advantage of this, placing

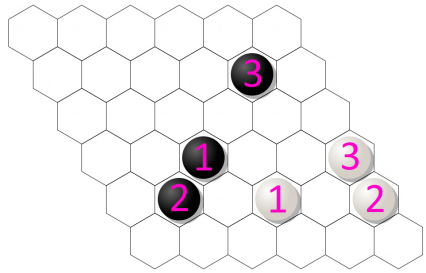
center pieces. However, it did not learn the winning edge positions. This would lead to the tree search players filling up the center at all times, due to being the top score. Some potential solutions are proposed in future work, to improve the Tsetlin Machine's ability to recognize these moves.

Figure 5.16 shows the first moves selected by the TM players in the tournament, in addition to MoHex 2.0 which played perfectly as black and white on the 6×6 board. The first three moves for black and white are shown in order of play. TM4 and TM5 played deterministically, while the MCTS players had some variation due to Monte Carlo simulations, and as such a representative selection of first moves was chosen. As seen in the figure, the TMs have a similar first move to MoHex 2.0, but the subsequent moves differ partially due to the TM not blocking its opponent.

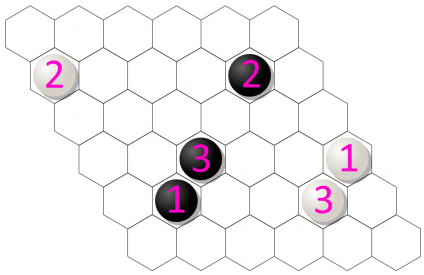
The figure illustrate a pattern between the TM players, showing a tendency to place center pieces on the short diagonal. As black, all TMs have pieces placed in the same positions as TM1's first and third placements.



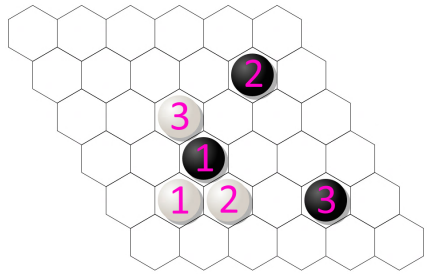
(a) MoHex 2.0 vs MoHex 2.0



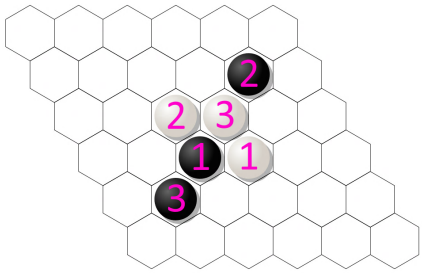
(b) TM1 vs TM1



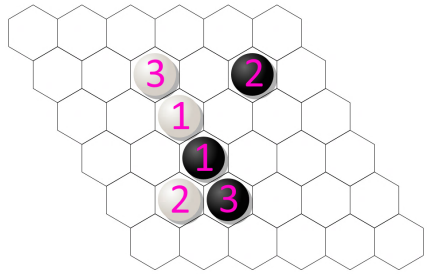
(c) TM2 vs TM2



(d) TM3 vs TM3



(e) TM4 vs TM4



(f) TM5 vs TM5

Figure 5.16: First moves, MoHex 2.0 and the TM players in the tournament

5.5 Learned clause patterns

This section will accentuate the interpretability of the TM by showing explicit examples of learned patterns. All examples used in this section are extracted from even numbered clauses, voting in favor of their given class.

TM1, from the Hex tournament in section 5.4, trained on the 17k dataset had a lot of TA for negated features choosing the include action. This resulted in the two figures 5.17 and 5.18 having several negated pieces in their clauses. The 17k dataset is heavily center focused and this may indicate that the TM is given a higher reward when placing pieces in the middle of the board. Figure 5.17 which represents win is a clear example of how highly the TM values not placing on the edges, due to all negated pieces on the edge positions.

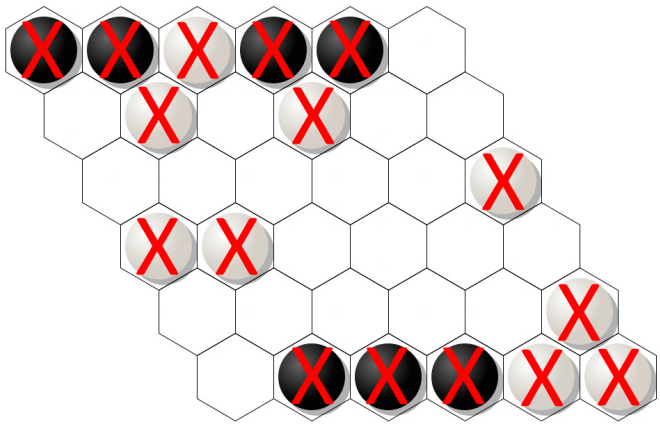


Figure 5.17: Winning clause pattern with weight 291, taken from TM1 trained on 17k dataset

Figure 5.18 represents loss, likely due to black blocking its own path with negated pieces on the bottom edge making the above piece redundant.

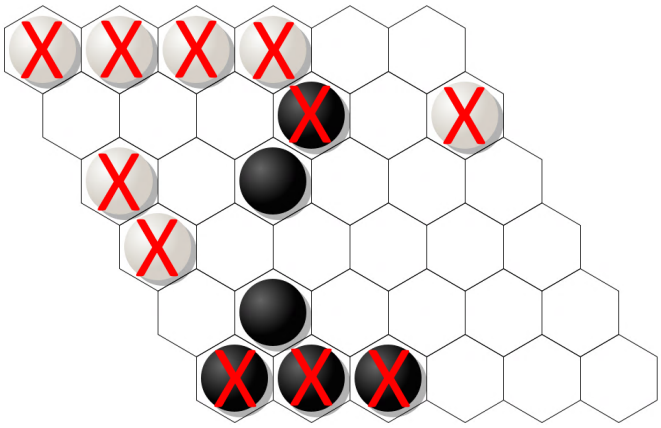


Figure 5.18: Losing clause pattern with weight 244, taken from TM1 trained on 17k dataset

Figure 5.19 is a frequent 287k clause pattern, which is filled up with negated white and black pieces, making it difficult to interpret on its own.

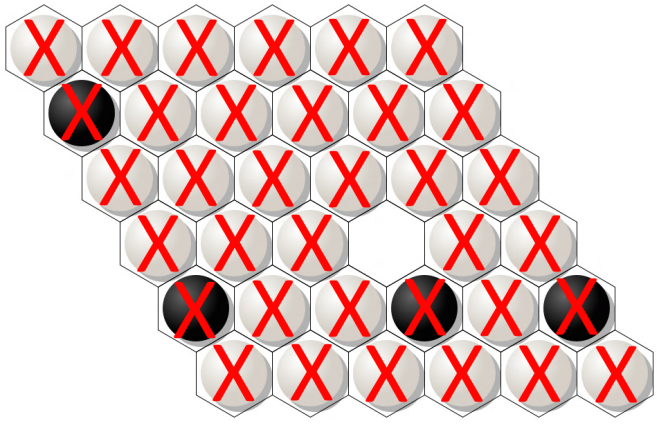


Figure 5.19: Winning clause pattern with weight 247, taken from TM3 trained on 287k dataset.

The clause in figure 5.20 has made a virtual connection (a bridge) for black, recognising that this is a winning strategy for the given player.

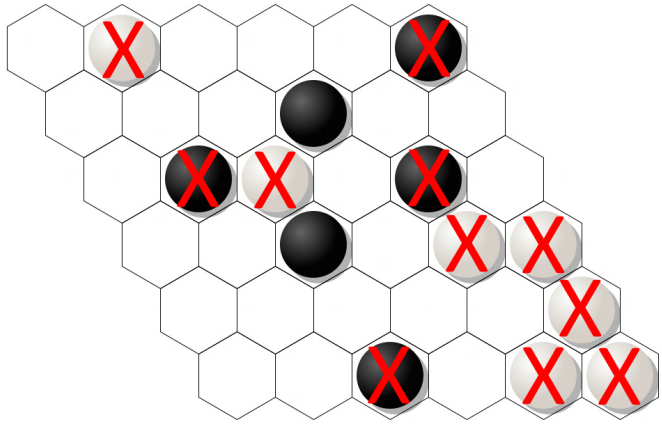


Figure 5.20: Winning clause pattern with weight 164, taken from TM3 trained on 287k dataset

The clause in figure 5.21 represents loss, likely due to blocking one potential bridge, making it challenging to connect previously placed pieces - effectively squandering one piece.

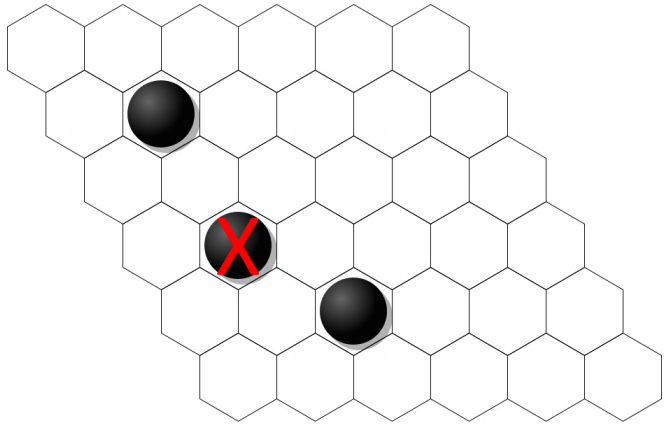


Figure 5.21: Losing clause pattern with weight 223, taken from TM3 trained on 287k dataset

Clauses in the TM work together and it is therefore challenging to select individual clauses that represent what the TM has learned. The clause patterns represented above was chosen based on interpretability and representability of their given class (win/loss) by themselves. This was done in order to make the results more comprehensible.

In a clause, the TA that states a piece must be included in a given board position are much more explicit than the TA for the negated feature. The latter can match with both empty positions and pieces of the other color, making them more versatile. The figures presented in this section include a majority of negated features, rather than non-negated. This may be the reinforcement learning part of the TA attempting to match as many inputs as possible. The TA wanted rewards more often and versatile patterns achieve this.

5.6 Summary

This chapter has presented the findings from the solution with the corresponding results and Hex tournament. A TM trained on the 17k dataset was shown to have weaknesses, when playing the game, apparent from first tree search iteration's precision graph. This weakness was improved with tree search, as the recall and precision graphs show significant improvements on new data derived from stochastic board positions. This is shown step-by-step by the TM using the original data compared to the final TM trained on the 287k dataset. The final TM achieved a first recall value of $R_1 = 97.8\%$ and a first precision value of $P_1 = 94.7\%$, compared to the initial values $R_1 = 87.5\%$ and $P_1 = 50\%$

The Hex tournament show that TM4 and TM5 had problems with finishing their games, due to top score selection being a narrow approach with the given data. The MCTS players were not able to beat random rollouts as an evaluation strategy. Random rollouts rely on playing the game until the end, and appears to be an effective estimate on small boards like 6×6. On larger boards, the effectiveness of a random rollout evaluation may diminish and become more difficult as the number of possible moves grow.

Chapter 6

Conclusion

This chapter overviews the thesis as a whole, concluding the findings, where the game of Hex has been examined using the recently introduced Tsetlin Machine technique.

Initial board evaluations were giving promising results of around 97% accuracy, yet gameplay revealed a lack of knowledge concerning board positions, making the Tsetlin Machine uncertain and prone to make untrained choices. The introduction of data derived from stochastic board positions gave the Tsetlin Machine versatile information, alleviating the gaps in the original data. Transforming the accuracy down to around 91%, but turning the initial recall from $R_1 = 87.5\%$ to $R_1 = 97.8\%$, and precision from $P_1 = 50\%$ to $P_1 = 94.7\%$. This allowed for educated choices during gameplay.

Still, in a conducted Hex tournament the Tsetlin Machine struggled with finishing the game with tree search, due to scoring the center positions higher than the edge positions. The tree search players were shown to fill the entire board before winning. Monte Carlo tree search was therefore introduced along with the Tsetlin Machine, effectively solving the problem of the board being filled, by giving the highest score to board positions that were found to result in a guaranteed win.

The thesis goals are fulfilled and a working Hex player has been demonstrated. Currently, the mixture of the Tsetlin Machine and Monte Carlo tree search was not able to beat a player using random rollouts as an evaluation strategy, on a board of size 6×6. The results are not at a competitive level, but are encouraging and can lead to bigger things. Based on ideas proposed in Chapter 8, future work, it is not unreasonable for the TM to perform competitively on larger board sizes, with further investigations needed to be certain.

We can therefore conclude that the TM is a good baseline that appears to be viable for playing Hex.

Chapter 7

Future Work

This chapter will discuss some interesting ideas sprung forth as we worked on this thesis. The ideas and their related challenges are explained below and should be explored in future research.

- **Extend tree search selection:** Make the tree search select several of the top evaluated board positions instead of only the best one. The search could be extended to go further in depth from these candidates and then finally select the top scoring board position amongst these evaluated boards. This extends the search perimeter to examine additional board positions that could be winning, alleviating evaluation mistakes done by the Tsetlin Machine.
- **Metadata:** The dataset could be extended with the use of metadata, for example to indicate the winning and losing move based on perfect play.
- **Move prediction:** Instead of doing board evaluations, the inference process could be switched to move prediction, as is done with reinforcement learning. The system would then predict the best legal move, instead of ranking whole board positions. This is challenging, requiring significant work, but it is an interesting prospect.

- **Self-play feedback loop:** A natural next step is to employ self-play improvements on the Tsetlin Machine, moving away from the supervised data. Deepmind's AlphaGo improved by being fed it's own best games as training data [48]. By saving the games from the Hex tournament as data, this could be used to re-train the Tsetlin Machine. A process, if repeated, could hopefully improve the pattern capabilities of the Tsetlin Machine over time, to make it sophisticated enough to beat the basic strategy employed by the RandomRollout player.

- **Larger boards:** Results show the Tsetlin Machine is capable of constructing reasonable patterns, but lack the precision of perfect play like MoHex 2.0 on the 6×6 board size. Yet, MoHex 2.0 is incapable of perfect play on much larger board sizes, this should narrow the gap between the methods.

- **Supporting MoHex:** Even though the Tsetlin Machine does not play perfectly, it appears to give good pointers for gameplay. Supporting the state-of-the-art MoHex with beneficial patterns, based on self-play, could be a fitting use-case for the Tsetlin Machine. The Tsetlin Machine could be acting like an *intuition* before search is done.

Bibliography

[1] Confusion matrices. https://www.geeksforgeeks.org/confusion-matrix-machine-learning/. Accessed: 28. May 2020.

[2] Hex at sensei. https://senseis.xmp.net/?Hex. Accessed: 11. May 2020.

[3] Hexy plays hex. http://vanshel.com/Hexy/. Accessed: 11. May 2020.

[4] Tree search. http://how2examples.com/artificial-intelligence/tree-search. Accessed: 05. May 2020.

[5] K Darshana Abeyrathna, Ole-Christoffer Granmo, and Morten Goodwin. Extending the tsetlin machine with integer-weighted clauses for increased interpretability. *arXiv preprint arXiv:2005.05131*, 2020.

[6] Vadim V Anshelevich. A hierarchical approach to computer hex. *Artificial Intelligence*, 134(1-2):101–120, 2002.

[7] Broderick Arneson, Ryan Hayward, and Philip Henderson. Wolve wins hex tournament. *Icga Journal*, 32(1):49–53, 2008.

[8] Broderick Arneson, Ryan Hayward, and Philip Henderson. Mohex wins hex tournament. *ICGA journal*, 32(2):114, 2009.

[9] Broderick Arneson, Ryan B Hayward, and Philip Henderson. Monte carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010.

[10] Geir Thore Berge, Ole-Christoffer Granmo, Tor Oddbjørn Tveit, Morten Goodwin, Lei Jiao, and Bernt Viggo Matheussen. Using the tsetlin machine to learn human-interpretable rules for high-accuracy text categorization with medical applications. *IEEE Access*, 7:115134–115146, 2019.

[11] Yngvi Björnsson, Ryan Hayward, Michael Johanson, and Jack van Rijswijck. Dead cell analysis in hex and the shannon game. In *Graph Theory in Paris*, pages 45–59. Springer, 2006.

[12] Cameron Browne. *Hex Strategy: Making the right connections*. CRC Press, 2009.

[13] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[14] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[15] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[16] Rémi Coulom. Computing "elo ratings" of move patterns in the game of go. *ICGA journal*, 30(4):198–208, 2007.

[17] Daniel James Edwards and TP Hart. The alpha-beta heuristic. 1961.

[18] Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. Fuego—an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[19] The International Go Federation. Go population survey. `http://www.intergofed.org/wp-content/uploads/2016/06/2016_Go_population_report.pdf`. Accessed: 18. May 2020.

[20] David Gale. The game of hex and the brouwer fixed-point theorem. *The American mathematical monthly*, 86(10):818–827, 1979.

[21] Chao Gao, Ryan Hayward, and Martin Müller. Move prediction using deep convolutional neural networks in hex. *IEEE Transactions on Games*, 10(4):336–343, 2017.

[22] Chao Gao, Martin Müller, and Ryan Hayward. Three-head neural network architecture for monte carlo tree search. In *IJCAI*, pages 3762–3768, 2018.

[23] Chao Gao, Kei Takada, and Ryan Hayward. Hex 2018: Mohex3hnn over deepezo. *ICGA JOURNAL*, 41(1):39–42, 2019.

[24] Chao Gao, Siqi Yan, Ryan Hayward, and Martin Müller. A transferable neural network for hex. *ICGA Journal*, 40(3):224–233, 2018.

[25] Martin Gardner. The scientific american book of mathematical puzzles & diversions. simon & sinis ter. *Inc., New York*, 1959.

[26] Sylvain Gelly, Yizao Wang, Olivier Teytaud, Modification Uct Patterns, and Projet Tao. Modification of uct with patterns in monte-carlo go. 2006.

[27] Ole-Christoffer Granmo. The Tsetlin Machine - A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic. *arXiv preprint arXiv:1804.01508*, 2018.

[28] John Hammersley. *Monte carlo methods*. Springer Science & Business Media, 2013.

[29] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[30] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[31] Ryan Hayward, Yngvi Björnsson, Michael Johanson, Morgan Kan, Nathan Po, and Jack van Rijswijck. Solving 7× 7 hex: Virtual connections and game-state reduction. In *Advances in Computer Games*, pages 261–278. Springer, 2004.

[32] Ryan Hayward and Noah Weninger. Hex 2017: Mohex wins the 11x11 and 13x13 tournaments. *ICGA Journal*, 39(3-4):222–227, 2017.

[33] Ryan Hayward, Noah Weninger, Kenny Young, Kei Takada, and Tianli Zhang. Mohex wins 2016 hex 11x11 and 13x13 tournaments.

[34] Ryan B Hayward. Six wins hex tournament. *ICGA Journal*, 29(3):163–165, 2006.

[35] Piet Hein. Vil de laere polygon. *Article in Politiken newspaper*, 26, 1942.

[36] Philip Henderson. Playing and solving the game of hex. 2010.

[37] Philip Henderson, Broderick Arneson, and Ryan B Hayward. Solving 8x8 hex. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[38] Shih-Chieh Huang, Broderick Arneson, Ryan B Hayward, Martin Müller, and Jakub Pawlewicz. Mohex 2.0: a pattern-based mcts hex player. In *International Conference on Computers and Games*, pages 60–71. Springer, 2013.

[39] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[40] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. Openspiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453*, 2019.

[41] Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The computational intelligence of mogo revealed in taiwan's computer go tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, 1(1):73–89, 2009.

[42] Jakub Pawlewicz and Ryan B Hayward. Scalable parallel dfpn search. In *International Conference on Computers and Games*, pages 138–150. Springer, 2013.

[43] Adrian Phoulady, Ole-Christoffer Granmo, Saeed Rahimi Gorji, and Hady Ahmady Phoulady. The weighted tsetlin machine: Compressed representations with clause weighting. *arXiv preprint arXiv:1911.12607*, 2019.

[44] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215, 2019.

[45] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[46] Claude E Shannon. Computers and automata. *Proceedings of the IRE*, 41(10):1234–1241, 1953.

[47] Peter Shotwell. The game of go: speculations on its origins and symbolism in ancient china. *Changes*, 2008, 1994.

[48] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[49] John Tromp. Number of chess diagrams and positions. `https://web.archive.org/web/20110629215923/http://homepages.cwi.nl/~tromp/chess/chess.html`. Accessed: 18. May 2020.

[50] John Tromp and Gunnar Farnebäck. Combinatorics of go. In *International Conference on Computers and Games*, pages 84–99. Springer, 2006.

[51] Michael L Tsetlin. On behaviour of finite automata in random medium. *Avtom I Telemekhanika*, 22(10):1345–1354, 1961.

[52] Kenny Young, Gautham Vasan, and Ryan Hayward. Neurohex: A deep q-learning hex agent. In *Computer Games*, pages 3–18. Springer, 2016.

Appendices

A Hardware Specification

Operating System	Ubuntu 17.10
Processor	Intel i9-9900K
Memory	64GB DDR4

UiA University of Agder
Master's thesis
Faculty of Engineering and Science
Department of ICT