UNIVERSITETET I AGDER

Playing Axis & Allies Revised Using Learning Automata

by

Gjermund Karlsen Lie

Thesis submitted in Partial Fulfillment of the

Requirements for the Degree Master of Technology in

Information and Communication Technology

Faculty of Engineering and Science

University of Agder

Grimstad, May 2009

Abstract

The Artificial Intelligence (AI) of opponents in computer games in general, and in strategy games in particular, have been plagued with performance problems of many kinds since they first appeared. Not the least of these problems is the fact that their design schemes often base themselves on predefined ways to play the game, making these opponents predictable and dull to a seasoned player.

In this thesis, we propose using Learning Automata (LA) to create opponents that are able to adapt to any game situation and find a good response, much in the way a player would - by looking ahead in time to see what could happen in the game beyond the immediate next move.

As a suitable environment for these LA, we have chosen the game Axis & Allies Revised. A turn-based war game emulating the second world war, it has many layers of complexity for the LA to struggle with - multiple moves per turn, random outcome of combat, and highly complex rules. To play this game well, the artificial opponent would need not only coordinate all his units into the best combined move each turn, but also to avoid performing moves in the present that it would be punished for during the next turns.

To solve these problems, we propose a two-step solution: First, each unit will be assigned its own, independent LA. Secondly, for each possible action that this unit can select in the next immediate turn, another independent LA will be assigned. This process can then be repeated until a sufficient depth into future moves has been achieved. Each tier of LA in this structure will receive its feedback not from its immediate surroundings - but from the status of the next LA down the tree.

In this thesis we lay the foundation for such a solution by implementing the method on a smaller scale, and by carefully testing its performance in a controlled environment. We find which approaches give the best results, which can only perform under certain conditions, and which are suitable for expanding into larger scale.

The three types of LA chosen for our testing covers most schools of reinforcement learning. The Tsetlin Automata, with its simple, state based structure. The Linear Reward Inaction Automata, with its linear updating scheme. And finally the Bayesian Learning Automata, shaping conjugate distributions in order to determine the optimal action. Each have their own unique strengths and weaknesses, which are recorded in this thesis.

Through thorough testing and careful tuning of these automata, we conclude that while LA may in fact have the potential to perform well in almost any type of scenario, it would still be impractical considering the time spent on deciding on a move. While the speed of decision making of our LA vary, so does its performance, even in our small scale testing.

Nevertheless, we believe that our results should give some insight into the possibilities and benefits, both in performance and design simplicity, of using LA as the decision maker for artificial players.

Preface

This master thesis is submitted in partial fulfillment of the requirements for the degree Master of Science in Information and Communications Technology at the University of Agder, Faculty of Engineering and Science.

The project was given, supported and supervised by associate professor Ole-Christoffer Granmo at the University of Agder, Norway.

I would like to thank my supervisor, Ole-Christoffer Granmo, for his constant feedback, good advice and encouragement during the long months of this project. Giving solid advice both on the subject and on thesis writing in general, this project would never have reached the state in which it is today without his help.

Grimstad, May 2009
Gjermund Karlsen Lie

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The field of machine learning has seen many applications over the years, but it has not yet been thoroughly tested in one of the major fields of Artificial Intelligence (AI) research: Automated opponents for games. While many approaches to game AI have been tried with varying success, we believe that machine learning can be used to create not only good opponents, but also opponents that are interesting and unpredictable to play against. As our game of choice for this project we have chosen Axis & Allies Revised (A&AR), a complex turn-based wargame emulating the second world war, that will give us all the challenge we need for our artificial player.

This chapter will go over the background of AI for computer games, turn-based games in general and A&AR in particular. We will also look at previous work in this field, both similar and completely different approaches, to demonstrate why more research on using machine learning in game AI could be a benefit to the field as a whole.

1.1 Motivation

The most common approaches to AI in computer games involves some kind of predefined actions. Some follow a set of predefined, static procedures, like a strategy-game AI gathering up a set amount of units of a particular combination before attacking, or a racing game AI which always goes for a calculated optimal turn in the track. Others have a set of equally predefined responses to expected player actions, like the strategy-game AI that will build response units based on what units its opponent brings to the battlefield.

For players new to a particular game this is in most cases good enough, but as their playtime increases the problems begin to show. Against an AI using static procedures, most players will eventually realize that the opponent reapeats its actions, or even use an identical approach every game. Beating the AI player is then reduced to devising a plan to beat this particular strategy. Depending on the game type, this might be as simple as using grenades against an opponent that hides behind cover in a shooter-game, or as hard as building the perfect counter to the unit combination the AI is using in a strategy game.

In the case of AI responding to their opponents actions, the human player may be entertained for a while longer, but every response can be responded to as well. Once the player becomes aware of how the AI responds to various actions, he can trick the AI into performing an action for which he is already prepared. In general, this type of AI is vulnerable to traps and other kinds of baiting strategies, like luring opponents into an exposed area to be picked off in a shooter game, or feigning weakness to draw enemy forces out from a safe place in a strategy game.

While the problems with the currently used approaches to game AI is well known, the growth of multiplayer gaming have effectively removed the demand for improving AI opponents in most types of games. Even for newer games, the single player mode have been reduced to a training ground before the players move on to fighting other human opponents. The developers have realized this, and are subsequently spending less of their funds on AI.

Single player gaming, however, is still an important part of what computer games are, and should not be ignored. Some players are unable to play their games against human opponents, or simply prefer not to, for various reasons. Ideally, these players should be given the same level of competition from the AI opponents as a human opponent would give.

With the major flaws in commercially produced game AI being predictability and lack of adaptability, we are naturally searching for approaches that work around this problem. Learning algorithms, reinforcement learning in particular, have already shown promise when it comes to creating adaptive computer opponents, and we believe that with further work, fully autonomous AI opponents that teach themselves how to play any game can be created using this approach.[1]

1.2 Background

1.2.1 Turn-based strategy games

Turn-based strategy games are generally war games, where players take turns moving units, or army tokens, across battlefields split into distinct areas. An example that will most likely be familiar to any reader is the game of chess. Here units of various type, strength and importance fight for a game board divided into 64 squares. Units have different abilities and values, and the ultimate goal usually involves killing the opponent's king. There are of course games with even simpler rules, such as Othello, where you only have one type of unit, and game with much much more complex rules, like the popular Civilization series.[2]

No matter the name of the game, the basic premise is the same, as are the strategies used to play them. Using the resources given to you, you must take control of the playing field. Your opponent is usually given equal resources, and so the outcome depends on who can utilize these resources in the most efficient way.

For research on AI, turn-based games have many interesting properties. The state of the game can be mathematically calculated, in order to give an AI feedback on how well it is performing. There is a finite amount of actions to choose from in each turn, however large this amount may be, and each of them can give both immediate and long-term effects on the state of the game.

In some games, however, a profitable move might not automatically give the desired outcome. In a game of chess you only move one piece each turn, and the outcome of battle is as simple as "the attacker wins". Most turn-based games, however, are many times more complex. Even in a simple wargame such as Risk - where you only use a single type of unit - players can move as many units as they like each turn, and the outcome of battle is decided by throwing dice.[3]

In Risk, this element of randomness makes attempts at calculating the best combined move for all units highly impractical due to the sheer amount of possible outcomes. As a very simple example, imagine a player attacking one area from another. The attacker has 5 armies, and the defender has 3. Even from such a simple setup, there are 18 possible outcomes split between winning. losing and withdrawing in the middle of the battle. Even with a maximum of three dice for the attacker and two for the defender, each of those outcomes is depending on the outcome of anywhere from 5 to 18 dice throws in addition to the choice of the attacker to continue, making it more or less impossible to predict - one can only give a rough estimate using statistics.

The AI opponents usually implemented in games such as these are heavily scripted to the point where a seasoned player can predict the outcome of any given situation. There are benefits to this kind of approach, both in precision and efficiency. If done properly, a scripted AI will have a good response to most common states of the game, and might even have a reasonable response to scenarios not directly predicted by the developers. In addition, an AI using prepared strategies will require little in the way of computational power to perform its job, allowing the developers to use more power on other elements of the game.

There are, however, problems with this approach, as have partly been explained in chapter 1.1. Predictability eventually makes computer opponents dull, as strategies for beating them are discovered by the player. While a game focused on multi-player may be excused for not challenging the player in single player mode, some games are simply not multi-player friendly. For these games, good AI is a selling point, and a reputation for having a bad AI may lead directly to loss of sales.

The focus of our work is therefore on reducing the predictability of AI opponents, while keeping their choice of action as beneficial as possible. Using turn-based games for this work comes naturally: Turn-based games can be simulated with relative ease, compared to many other games. The state of the game, and thus the performance of the AI, can be calculated numerically, and there is a fixed amount of actions the AI can choose from, even if this number may be high.

1.2.2 Axis & Allies Revised

Originally a board game dating back to the early 1980's, the game of Axis & Allies has seen many variations and upgrades over the years. There are several board games with various extension packs, and both turn-based and real-time computerized versions. The version we will refer to for our rules and unit data in this thesis is the "Revised" version, released in 2004, that have also been computerized in both commercial and open-source implementations. The open source implementation, named "TripleA", is the source of the graphics and maps used throughout this paper to explain various game states.[4, 5]

Each game is played on the a world map divided into zones of varying importance and value. These areas are then divided among the up to five players, each controlling a major power during the second world war. These five powers work together as, and can be combined into, the two sides - the Axis and the Allies - which makes it possible to play the game with as few as two people.

In these areas, each player is given a predetermined number of units representing armies of different types.[6] Ranging from simple infantry batallions to bomber aircraft and battleships, each unit has different cost, properties and special abilities. In additon the player is given factory structures, representing points where he can position new units.

The gameplay is relatively straightforward. Each turn, the player is given an amount of production credits, called "Industrial Production Certificates" (IPC). These can then be spent on researching technology to improve his units, or producing more of them. Next comes the combat move phase, in which the player decides where to move each of his units, and where to engage enemy units in order to take over their territories.

When a combat phase move ends with units in opponent territory, actual combat starts. One battle at a time is resolved by dice throws from both players, and goes on until either the attacker decides to retreat, or one side loses all its units. After the combat moves, the player can perform non-combat moves, followed by an economic phase where the player positions his produced units and collects his IPC. When all of this is done, the next player starts his turn.

The goal of the game depends on how long you want to play, but usually involves capturing an agreed upon amount of the twelve "Victory Cities", of which both sides start out with six. Capturing 8 of the cities is refered to as a minor victory, 10 a major victory, while 12 is refered to as a total victory.[7]

On the way there, however, the foundations for winning a game comes from the following cycle: The player must spend his IPC in the most efficient way to take control of more lands, which will then give him more IPC to use on units. All other things being equal, the player with the highest amount of IPC controlled, whether it be in units and land, will come out on top.

The current implementation of AI into TripleA is unfortunately very weak. It works based on simple rules to decide where to attack and with how many units, and how to produce, place and move its units into position for such an attack. The rules which it follows are strict enough to be predicted by the human player, and yet does not provide the AI with an overall plan that is good enough for systematically pressing for a victory. It simply looks for opportunity in the present, and then moves toward this opportunity, ignoring possible counterattacks, traps or other consequences.

For a better approach to an AI for A&AR, we are proposing a learning-based approach. To be an improvement over the existing AI, such an approach would need to both respond better to immediate threats, and to have better responses to likely enemy moves in the future. While all of the tasks involved in the game, even the choice between production and research and the placement of additional structures and units, should be possible using a learning-based approach, to keep the size of the project at a manageable level, we have chosen to limit ourselves to handling the movement and combat phases.

Even while avoiding some of the management choices, we are still left with a challenging task. Each player controls a large amount of units, each having their own strengths and weaknesses, and each having a selection of possible actions. In addition, with combat being decided by dice throw, even what seemed a good choice one round might turn out to be a bad one the next round. The total amount of possible outcomes of any scenario is enormous, even by only looking at the single next turn.

To implement a learning-based AI into this type of game environment, we would need the ability to simulate each turn over and over. Unfortunately, TripleA does not support this type of "undo" feature natively, and for our purposes, implementing it into TripleA would be needlessly complicated. Thus, we will be looking at creating a simplified game engine from scratch, which can handle simulating an arbitrary number of rounds.

1.2.3 Artificial Intelligence in strategy games

To establish why we have chosen to implement our AI using a learning algorithm, we will first look at a selection of other methods for creating good computer opponents for strategy games.

Chess

Likely the most popular turn-based strategy game in existence, the game of chess has naturally been the subject of AI research for many years.[8]

A short summary of the rules for those not familiar with the game: The game board is divided into 8x8 squares, on which each of the two players have 16 pieces. There are 6 different types of pieces, each with their own rules for movement, and thus different sets of possible actions. Each turn, the player can move one piece according to its own rules. Entering a square containing an opponent unit will remove the occupying piece from the game, and the entering piece will take its place. The goal of the game is to defeat the opponent King, a slow unit that needs protecting by the other units.

Many attempts have been made at making chess AI that can compete with humans, and one of the better approaches in terms of performance is a Minimax search with Alpha-beta pruning.[9]This algorithm works by searching a node tree where each branch represents a possible action by either the player or opponent in a 2-player game. Each action is evaluated on what would be the worst outcome from it, depending on opponent actions, and the algorithm then attempts to find the node with the least bad outcome. If an action is found to yield a worse result than the expected minimum from the tree as a whole, it is "pruned", and the search will not go deeper into that branch. While the Minimax search in itself is expensive performance-wise, the Alpha-beta pruning improves performance without compromising precision.

This works well for chess, because even if there are a large amount of possible actions to choose from, each action yields the same result each time. When one chess piece enters the area of another chesspiece, the first chess piece will win every time, there is no random factor. A Minimax search can therefore explore the node tree as deep as computational power or time constraints allows, in order to give an accurate picture of what would be the best move. The Minimax tree includes every move open to the opponent aswell, so the method works regardless of what actions the opponent would take.

For Axis & Allies, however, the Minimax search with Alpha-beta pruning is not a possible approach. While Minimax searching would theoretically be possible, this would have to be done using average values for the outcome of each combat scenario because of the random outcome of battles. That would involve mathematically calculating the average IPC-gain value for every single possible move, immediately requiring many times as much computational power as the Minimax algorithm itself. In addition, there is the problem that for each turn, unlike in chess, the player can move as many units as he would like, drastically increasing the amount of branches from each node in the tree.

Go

Like chess, go is a turn-based strategy game for two players, but gets its complexity in an entirely different way. The game only has one type of playing piece, that can only be placed once on the game board, much like Othello. The game board, however, is usually 19x19 squares, and the players are free to place their pieces in any unoccupied position on the board. This sets the possible actions for each player each turn to a flat 19x19 = 361 minus pieces already placed.

While the scoring rules vary somewhat, the winner of the game is generally the player with the largest areas of the game board surrounded by his pieces. Removal of pieces from the game board only occurs when they are completely surrounded by enemy pieces, and is a rare occurence in most professional games. For more in-depth go rules, see [10].

Go AI is considered much more challenging than chess AI, an assumption backed up by the relative performance of go AI and chess AI playing against top-level human opponents. Whereas the best chess players in the word have been beaten by autonomous computer opponents, even the best algorithms for playing Go, using many times more computational power, still require huge handicaps to play on par with the best human players. Some of these algorithms, in particular Monte Carlo based algorithms such as MoGo, do however do fairly well on smaller playing boards.[11, 12]

The Monte Carlo method is based on a simple premise:

- The player has a certain set of possible actions. In the case of go, he would have 361 options minus pieces already placed.
- After randomly selecting a move from the set, the status of the game is calculated.
- After attempting a given amount of random moves, the result is given as the choice giving the best game state found during this search.

The benefits of this method includes that it works better given more time and processing power - shown by MoGos good performance against humans when given adequate processing power. As an example, for the first ever win against a professional player, MoGo employed 15 Teraflops ($15 * 10^{12}$ floating point operations per second) worth of computational power.

Translated to Axis & Allies, it might at first appear a possible approach. The set of actions possible actions in Axis & Allies is, in fact, finite - even if the ability to move multiple units every round might quickly yield a much larger set than Go. When it comes to calculating the state of the game, however, Axis & Allies starts giving the method problems. A single set of actions might give a wide selection of different outcomes, depending on what player the dice favors during the combat phase. Simply picking the best outcome based on a game state caused by dice throws, the chance of actually picking an actually beneficial move is minimal.

Diplomacy

Diplomacy is a game with many similar properties to Axis & Allies, but also many important differences. Like Axis & Allies, the game is played on a map divided into areas, through which multiple units can be moved to attack or defend each turn. In contrast to Axis & Allies, however, each player does not take his turn in the traditional fashion. For each turn, every player selected his own set of moves without knowing what moves the opponents will pick - making it hard even for a human player to calculate what moves would be beneficial.

As might be expected, this frequently leads to the situation wherein two, or even more, players attempts to move their pieces into the same area. This situation is known as a standoff. A standoff in practice means that each unit moves back to their original positions, having achieved nothing that turn. However, there are ways to get around this situation. The third movement option for each unit is called support. When supporting, a unit lends its strength to another unit performing another move. When a standoff occurs, the unit which have the most supporting units, regardless of their nationality, wins the standoff.

The simple fact that players can support each others units in combat, and that all moves are determined in secret, ensures that this game has truly earned its name - backstabbing and secret alliances are not only possible, but almost mandatory, in order to win the game.

For a more in-depth explanation of the rules of Diplomacy, see the official rulebook.[13]

When it comes to AI development for Diplomacy, many approaches have been attempted. As our example, we have chosen a method using pattern-weights.[14] This approach is based on creating a database of features, information of interest when deciding upon what actions to pick. These patterns are then weighted through experience, determining what importance they are relative to one another.

As a very basic example, a feature could be: Who occupies the neighboring territory? The weights for each possible option - the player himself, another player, or no one - could be adjusted either by an expert player, or simply by according to what occurs when playing randomly.

In [14], learning is used to update these weights through self-play. After each play through until the finish, the weights involved in deciding which moves to pick each turn are updated depending on whether the player was victorious or not.

This learning-based approach worked well for Diplomacy, but for Axis & Allies, there are some distinguishing features which could cause problems for an approach based on long-term victory or loss. The most critical of these features is the random combat. Whereas combat in Diplomacy is determined simply by who brings the largest amount of troops, Axis & Allies combat can have a wide array of outcomes depending on the combination of troops and the luck of the dice. Through an entire game, simple luck might be the only reason why a player won or lost, and thus an update based on this win or loss would usually lead down a wrong path.

1.2.4 Learning Automata

The first study concerning the concept of Learning Automata (LA) was published by M.L. Tsetlin in 1973.[15] His work focused on machines determining the best behavior in a random environment by using learning algorithms, and has since been added upon and updated regularly. The general principles of his learning methods, however, still applies.

The field of machine learning has seen use in many fields recently, the most highly profiled ones including some kind of pattern recognition. Search engines, spam detection and text or image classification are just some of many possible uses.[16, 17, 18]

In this thesis we will be focusing on an area of machine learning known as reinforcement learning.[19] The basic premise of reinforcement learning is this: Having a set of environment states, a set of actions, and a set of rewards, the algorithm attempts to choose the action from the set giving it the highest possible reward. The LA needs only know how many possible actions it can choose from, and how profitable it was choosing its last action. Each action performed changes the environment state, and the reward given depends on this state. The automata then process this reward, usually by changing the probability for picking the last selected action in subsequent runs.

A central property of LA is that they have no prior knowledge of which action in the set is the best, in fact they usually have no knowledge of any property concerning the environment. Yet they are still designed to find the optimal action, simply by trial and error. Because of this, reinforcement learning sometimes gives unexpected results, both of the good and the bad kind, simply because how the algorithm sees the environment might be different than our preconceived notions.

Translated to the setting of our game, we note that Axis & Allies includes many features that appear suitable for reinforcement learning. We have a game state defined by the position and amounts of various units in their areas. We have a set of actions defined by where each of these units can attack or move, and combat rounds resolved using random numbers, which changes the game state. Finally we have the possibility to calculate a set of rewards determined by the relative strengths of the player and opponent.

Since an LA does not require any information on how a particular game works, implementing it as a decision maker into any given game is, at least in theory, rather simple. All you need is some way of connecting the automata to some game entity, define what its set of actions means in game terms, and decide how to translate the result of said actions into a reward.

To be effective, such an LA would need to simulate the outcome of its actions before actually taking them. In some computer implementations of simpler games, an "undo" feature exists, but in the case of A&AR, we will need to create our own simulation engine.

1.3 Research Questions

Having established why we have chosen a learning-based approach, we move on to the definition of our thesis. The goal of our research can be condensed into the following research questions. Note that we are referring to the concepts of Decentralized and Multi-Tiered Learning Automata. These, and our reasoning for using them, will be thoroughly explained in chapter 2.

1.3.1 Main Question

Can we, using Learning Automata, achieve a good balance between reactivity and proactivity in an Artificial Player?

In general, the weaknesses of Artificial Players comes from being too focused on either reactivity or proactivity in its playing strategy. An overly reactive player will react to every move the opponent does without an overall strategy. An overly proactive player will have a strict plan that it will follow no matter what its opponent does, and may fall apart should its plan be disrupted. It is therefore our goal to determine if LA can be used to find a good balance point between these two extremes, improving the performance of the Artificial Player.

1.3.2 Sub-Questions

Does a Decentralized Learning Automata perform adequately as a reactive player?

While creating a purely reactive Artificial Player is as simple as mathematically determining what move currently produces the greatest gain, we need to determine whether an LA-based player can react properly to situations where a short-term threat needs to be prioritized over the long-term goal. As opposed to an Artificial Player that analyze the entire playing field at once, we intend to use a separate LA for every unit. For the LA-based player to be successful, the unit-LAs will need to work together to find the best combined move in a situation, without relying on internal communication between automata. As a reactive player, we need to test if a decentralized LA can determine the best move while only looking at the next move, ignoring the consequences. We then need to test if this reactivity is preserved when we proceed to looking several moves ahead.

Can a Multi-Tiered Learning Automata substitute the need for a long-term plan?

Even if our Artificial Player is not entirely reactive, an LA-based player would never have a predefined plan for victory. Even so, occasionally a situation will arise in which looking more than one turn ahead is needed. We need to determine if an LA-based player can look past the immediate feedback of the next turn, and respond to threats in the turns beyond, whether they consist of a trap, a new threat arriving, or simply an opportunity arising due to actions in other areas of the map.

1.4 Report Outline

This thesis is organized as follows:

Chapter 1 explained the motivation behind this project and introduced the reader to previous approaches used in turn-based game AI in order to explain why we have chosen a learning-based approach. It also gave a brief introduction to the field of learning automata.

Chapter 2 thoroughly explains the learning methods we have used in our approach and the functionality of the various types of LA.

Chapter 3 explains our implementation, both of the simulation engine created to substitute the actual game engine, and the implementation of the various types of LA.

Chapter 4 describes the scenarios we have created for our testing, as well as a thorough statistical analysis of some key possible actions to determine what results should be the goal of our LA.

Chapter 5 shows statistics on the results of the various types of LA playing the scenarios described in chapter 4, and an analysis of the performance of each type of LA.

Chapter 6 contains the conclusion of this thesis, in addition to proposing further possible work related to the topic and to our approach.

2 Methodology

In this chapter we will describe the learning method we have used in our approach, the Multi-Tier, Decentralized Learning Automata. We will also explain the function, update methods and action selection methods for our three types of LA.

2.1 Decentralized Learning Automata

Because we are dealing with several individual units that needs to cooperate in order to accomplish an overall goal, we are naturally going to use a decentralized learning scheme. Each units needs their own automata taking individual decisions, otherwise the complexity, the sheer amount of movement combinations, would quickly become overwhelming. Even the units in a single area of the map could easily have millions of possible movement combinations, and using only one automata to teach itself which combination would be the best would be impractically difficult, if not downright impossible. Decentralizing the learning automata gives each automata a more manageable amount of actions to choose from, and allows for better response from each round of feedback from the environment.

Because the performance of an individual unit might have little or no connection to the performance of the player side as a whole, all automatas need to receive the same feedback. This might need different implementation depending on the type of automata, but should lead to a similar outcome for each of them. A good example related to Axis & Allies gameplay would be the cheap, expendable Infantry units. When the opponent score hits, the player is allowed to choose his own casualties.[7] The player would in almost any practical case start by removing one of his infantry units as a casualty, seeing as they are the cheapest and usually also the weakest unit in any given battle. This unit would then be marked as "dead", and if it were to be rewarded based on individual performance, it would obviously be punished for getting killed. In practice, the player side as a whole might be performing well, partly because the infantry was there to take the hits. Without it, more expensive units might have been killed before doing damage, changing the outcome of the battle as a whole.

2.2 Multi-Tiered Learning Automata

In order to allow our artificial player to see the greater picture, not just the immediate benefits of a particular set of moves, we need to allow for simulating some turns ahead. For a simple estimate on what action to choose, simply simulating the current turn repeatedly might give a rough idea on which moves are good or bad. It would not, however, help against opponents with the ability to plan even a few moves ahead, as most humans would. An automata only concerned with the immediate next move would be susceptible to even the simplest traps, like a player leaving a valuable area underdefended in order to thin out the opponent's units over a more zones.

To achieve this, we need to set up not only one automata for each turn that we are playing, but one for each possible action. Each automata will then only need to concern itself with the various changes to the world state when the unit it is connected to picks that particular move. To clarify: Every time one automata makes the decision to move from one particular zone to another in a particular turn, the same automata handles the feedback from the system. This regardless of what other units are doing. This automata then learns what would be the best average move from this position depending on the choices of other automata during this round.

Now, implementing this method is simple enough for the first turn forwards, but the size of the tree of automata quickly grows each turn. First, branches for each possible action would be added to the single root automata choosing the first action. For the next turn, each automata on the end of those branches would have branches on their own for every possible action, and so on through the turns. If the amount of possible actions remains constant throughout the turns, the amount of automata required will grow exponentially, limiting how many turns ahead we can practically simulate.

In addition to the potentially overwhelming amount of automata at later turns, we are faced with the problem of getting useful information from said automata back to the starting automata, in order for it to make a good decision. With the outcome of each battle being random, an average performance of each round of simulation after learning has been done would be returned to the parent node in the tree. To make this work, we will be doing the following:

- Each depth level of automata will simulate their moves and combat actions, and move on to the next depth level until max depth is reached.
- At the last depth level the automatas will simulate their move, learning from the feedback from the system. The average status of the game after this learning will be returned to the depth level above.
- Each depth level will receive feedback from the deeper levels, and learn from it. The average status of the game after this learning will also be returned to the depth level above.
- The above will be repeated until the top level of automata have learned, after which we will have our conclusion.

For each level of depth we go down, the exact situation that is currently being simulated upon gets less likely to happen. At the first depth level, there is a good probability for repeating the previous action and get a similar result, but once we get lower in depth, this probability decreases rapidly. For example: One Tank unit moving from friendly area A to another friendly area B during the first turn would give the same outcome every time, and depending on how many possible actions the Tank has, may happen often. On the other hand, an Infantry unit moving between two areas the first turn, attacking a defended opponent area the next, and finally moving back the next turn, would be a string of events that would be very rarely repeated.

Because each scenario deeper into the future turns is very unlikely to be repeated exactly, we have chosen to use less resources on exploring how profitable each of them would be. In general, this would not cost much in the way of precision, but yield a large benefit in terms of efficiency. For similar reasons we will look into how deep into the tree of automata there is actually any benefit to simulate, as we expect there to be a cutoff point where a deeper search would not affect what action should be chosen on the first turn.

Combining the multi-tiered approach with the need for a decentralized learning approach, we get a structure as seen in figure 1.

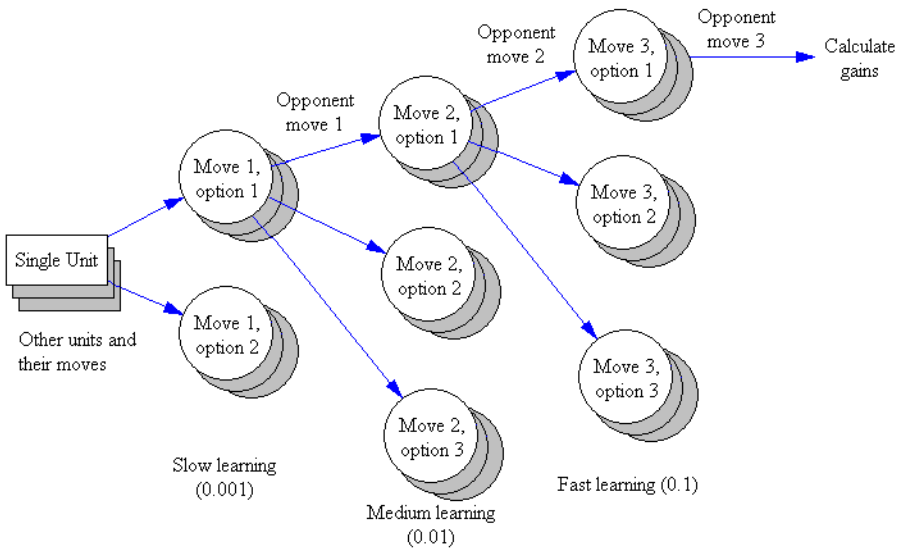

Figure 1: Multi-Tiered, Decentralized Automata

In our example structure, we have the starting move, the one which we are actually intending to perform in the real game, and then two more simulated moves. When both the player and the opponent have chosen their third move, and all combat calculations have been performed, the total gain for each player is calculated from the current game state. This is then passed back down to the automata deciding on what move to pick as the third move for each unit, updating the automata.

The simulator repeats the last move either a set amount of times or until all automata have converged, depending on the type of automata aswell as performance concerns. The average of the last, and thus hopefully best, part of the calculated gain is then returned to the automata deciding on what move to pick as the second move. Another set of second-tier moves are selected, and then the simulation start over on the third move. This is repeated as many times as necessary for the first automata to come to a decision.

Depending on what type of automata is used in the various tiers, there may be changes in learning rate as we move through the tiers. This is, as mentioned before, because the precision of the learning is not as critical when it comes to the later turns.

2.3 Various types of Learning Automata

For this project, we have chosen three quite different learning automata, both in function, update mechanisms, and action selection. Trying the same scenarios with different types of automata, even different combinations of automata, gives us a good picture of the capabilities of different approaches, and of LA in general.

2.3.1 Tsetlin Automata

Based on the original idea of LA, the Tsetlin automata is one of the simpler ones, both to understand and implement.[15, 20] A Tsetlin automata in itself is only required to store two variables - its current choice, and its "depth". When the LA is rewarded for its current choice, it keeps the choice variable unchanged and increases the depth variable up towards a maximum value. When it is punished, on the other hand, it reduces its depth until it reaches zero, and then changes its choice.

A good feature in Tsetlin automata is that they are easy to configure to your liking. For example, increasing the maximum value for the depth variable will reduce switching between states. This would lead to a more decisive automata, but it could also increase the chance of missing a good move. One can also put artificial limits on how often the automata will do its reward or punishment, in order to enforce a more explorative or a more decisive automata, respectively. In practice this either means a limit on what number range from a set of rewards actually translates to either a reward or a penalty, or setting a percentage limit on how often a reward or punishment actually is performed after the decision has been made. We will be using the second method in our implementation, as can be seen in section 3.2.1.

The downside of the Tsetlin automata is that they have no memory of previous rewards or penalties, in fact no memory at all beyond what action it currently percieves as the best. This means that no real convergence can be obtained, and that as the amount of choices go up, in particular the amount of choices with similar rewards, the probability that the automata will find the optimal choice is lowered.

A simple form of Tsetlin automata, with only two choices and a maximum depth of 3, can be illustrated as seen in figure 2. Rewarding the automata will move its decision further away from the line between the choices. Penalizing it will move it closer, and when in the central states, across it.

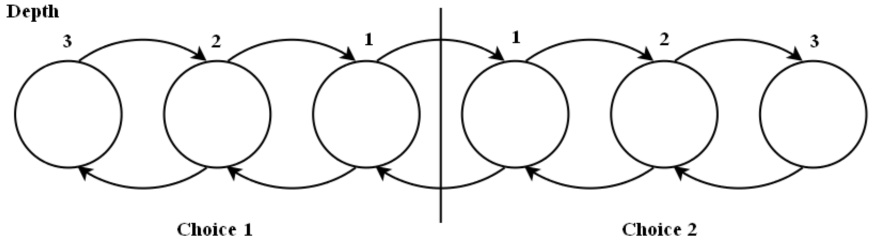

Figure 2: Simple Tsetlin automata

In our case, each LA has a number of choices depending on the amount of possible moves for each unit. In the case of aircraft units capable of multiple moves per turn on a larger map, this might go up to or beyond 10, but for most land based units, which is what we will be focusing on, it would be an average of 4. This gives us a slightly different kind of structure, but the end result is the mostly the same, as can be seen in figure 3. Whenever the automata changes its choice, it simply picks another at random. This means adding another variable for the amount of choices to pick from for when the automata needs to change its choice.

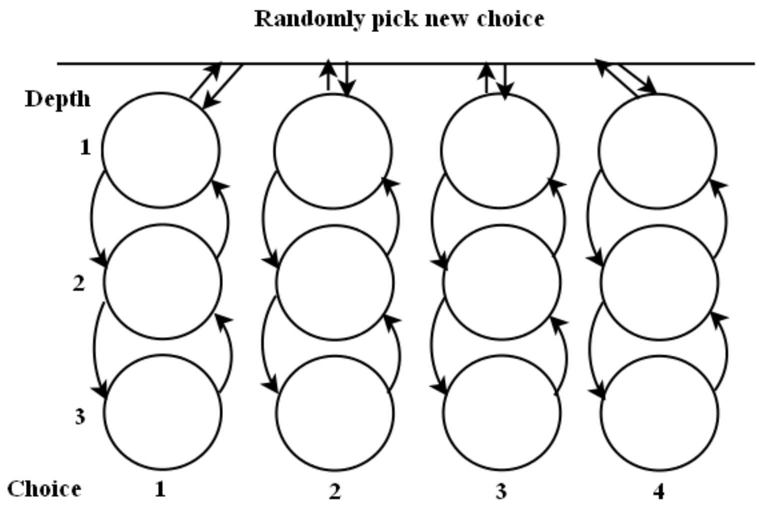

Figure 3: Multiple-Choice Tsetlin automata

2.3.2 Linear Reward Inaction Automata

Linear updating automata is a well known and thoroughly researched type of automata, and for this project we have chosen to use the Linear Reward Inaction (LRI) automata.[20, 21]

Simply put, this automata contains only as many variables as it has choices, plus a single factor determining its learning speed, wich we will refer to as α. Starting out, each choice is given an equal chance at being picked at random, usually referred to as P_n. We will be using the very simplest updating scheme, working as follows. Each time the automata performs action x and receives a reward, the following will occur:

$$P'_x = P_x + \alpha$$

$$P'_n = P_n * (1 - \alpha)$$

Where P_n represents the probabilities for picking all actions, x included. The net effect is that the chance to repeat the performed action increases, but the total probability remains constant.

When the automata performs action x and gets punished, however, no change occurs. This usually ensures that “good” choices are not as affected by bad luck, as they will only be reduced in relative strength when another good choice is found. For our purpose, where streaks of bad luck caused by the often large amount of dice throws might otherwise cause good choices to be incorrectly seen as bad, this is a large benefit.

Again, the basic concept is best explained by illustration, as seen in figure 4. Initially having four equally probably choices, the automata eventually goes toward converging on Choice 1. It should be clear from the figure that this approach is equally suited for both small and large numbers of possible actions, but in general more actions will require a higher number of simulation runs.

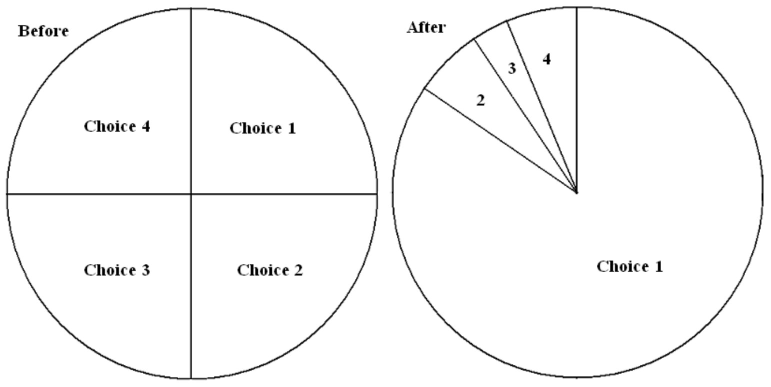

Figure 4: LRI - Distribution of chance between choices

Instead of increasing the amount of simulation runs to get a conclusion from a larger amount of choices, we could increase the learning rate. This would naturally cause the LA to converge faster, saving simulation cycles, but could also lower the precision of the automata as a whole. This comes from the fact that we are basing our rewards on random numbers, and with too high learning rate, a string of random numbers could trick the automata into converging on a less than optimal move. An LRI converged on a bad decision will end up in a state of inactivity, because only picking bad choices will cause penalizing, and penalizing does not affect the automata. It is therefore important to balance the need for learning speed with the need for learning the optimal action.

When given a reasonably low learning rate and enough time, an LRI will eventually converge. If more than one action is more or less equally beneficial, the LRI might have trouble deciding between these two, and might return either of them - but the probability to pick any other action would still be marginalized. This ensures that the LRI picks a good option even when there are more than one of them. If only one option is profitable the LRI will always converge to picking this option each simulation run, as long as the learning rate is not high enough to give false positives.

The standard implementation of LRI is more or less ready to use for our purpose, but it does require a high amount of simulated rounds to converge, and might thus not be suitable for use in the later tiers of the tree of automata.

2.3.3 Bayesian Learning Automata

In contrast to the automata above, which are simple both in concept and implementation, we have chosen the Bayesian Learning Automata (BLA) as our final LA.

Although computationally intractable in many cases, Bayesian methods provide a standard for optimal decision making. The BLA is inherently Bayesian in nature, yet avoids computational intractability by relying simply on updating the hyper parameters of sibling conjugate distributions, and on random sampling from these distributions.[22]

The probability that BLA selects a specific action can be interpreted as the probability that the specific action is the optimal one, given the feedback thus far received from the system. Thus the BLA gradually shift its selection focus towards the action which most likely is the optimal one, as the observations are received.

One possible explanation of the superiority of the BLA is the following: The BLA is the only one of our methods that maintains a Bayesian posterior distribution of the possible reward probabilities of the arms. Thus, when the BLA makes a decision, it is based on the best possible knowledge about the move "value", given the feedback that have been received thus far.

Instead of making a decision simply by determining where in our set of weighted choices a given random number is, the BLA functions by finding the move which gives the highest mean value from a normal distribution where the parameters determining the shape is updated by our automata. Given a random number generator r, we get the mean value using the following formula:

$$Mean = \mu + N(r, \sqrt{\frac{\tau}{G(r,\alpha,\beta)}})$$

Where N returns a random number from a Normal (Gaussian) distribution, G returns a random number from a Gamma distribution, τ, α and β determines the shape and scale of our distributions, and μ is the expected mean value based on feedback from our system. Now, to select an action from our set, we simply select the action connected to the highest *Mean* value, where new random samplings from each distributions are taken for each choice.

The Normal distribution function normalized to a mean value of 0 looks like this, with the shape determined by the size of σ - referred to as the standard deviation.[23]

$$n(x,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-x^2}{2\sigma^2}}$$

As σ grows, the graph moves from a flat, wide curve into a tall, thin peak. When the BLA receives good feedback, the value of σ is increased for a particular move, eventually converging to the point where the tallest peak is placed on top of a particular move, at which point only this move will be selected.

The Gamma distribution function used to yield a randomly selected σ based on the feedback looks like this. The α variable governs the shape of the curve, the β variable governs the scale, and $\Gamma(n) = (n-1)!$.[24]

$$g(x,\alpha,\beta) = \frac{x^{\alpha-1}e^{-x/\beta}}{\Gamma(\alpha)\beta^{\alpha}}$$

By constantly increasing the value of α, while increasing β based on how well the automata is doing, we end up with a tight peak, where the height is determined by how beneficial feedback the automata received.

To update the variables used in the above formulas, the following formulas are used. S is the feedback received from the system, given as a number between 0 and 1:

$$\alpha' = \alpha + \frac{1}{2}$$

$$\beta' = \frac{\beta + \tau(\mu - S)^2}{2(\tau+1)}$$

$$\mu' = \frac{\tau\mu + S}{\tau + 1}$$

$$\tau' = \tau + 1$$

Our BLA have been derived from a solution for solving the Two-Armed Bernoulli Bandit problem, where a BLA using a beta distribution is shown. For a more in depth explanation on the inner workings of the BLA, we will direct you to this paper.[21]

Because the functionality of the BLA requires us to keep incrementing the variables, and some of these variables are used for calculations of *n!* complexity, the longer time we spend on learning, the higher the computation time becomes. As we expect to need a longer period of learning each time we increase the amounts of units on our A&AR game board, we may end up with a set of automata with very good accuracy, but with poor computational efficiency.

3 Implementation

If we could simply use an existing A&AR engine to do our simulations, we would have saved ourselves a lot of time and effort. Unfortunately, the existing game engines do not support retracing your steps backwards in time, a critical requirement for our method. Because of this, we had to reimplement enough of the game functionality into a new engine to allow the our two sides of LA both to both learn and simulate actual gameplay correctly. This meant replicating some of the major non-graphical features of the game engine: Unit and map properties, movement and combat simulations. In addition we needed to allow for keeping track of previous moves, so that we could go backwards through the turns without losing information, and of course our LA functionality.

3.1 The Simulation Engine

Our engine consists of three main parts. The first is the setup phase, reading map information from files, setting up the areas of the map with the units they contain, and giving every unit their set of automata, whether it be learning or not. Almost all memory allocation is performed in this step to improve performance. If the scenario has a predefined set of moves for the opponent to use instead of LA, these are added to the automata.

For units not controlled by these predefined moves, every unit, now connected to an unique automata, has their move set calculated. Each move in each of these sets are connected to new automata, and this is repeated for as many turns ahead as the scenario will be running. For our scenarios, two or three turns have proven to be enough.

The second part is the main loop. This is a recursive loop, keeping track of which round it is currently simulating, and returning the performance of later rounds for use in updating the automata in the earlier round. Each time a new round starts, all learning data is reset for the automata involved in that round, and for every round of the loop, all unit states are returned to what they were before simulating that round.

To keep the computational demands down, we have chosen to only implement the multi-tiered LA approach in one of our automated players. The second opponent, in the single scenario where our opponent is actually learning based, will be looking only at the current turn when deciding on its move. For the scenario where this limitation is in effect, however, the effect should be neglible, as the outcome of the scenario can mostly be determined after the first round has been played.

Each loop of the simulator has the following steps:

1. Reset all unit and map states to either the positions generated at map creation or the state of the last round - depending on what round we are currently simulating. This is equivalent of turning the game clock one turn backwards after simulating a move, and is the main reason why we could not simply use the already existing game engine. Each unit has a list of locations, containing its position in each depth level of the simulation. It also has information on whether it is alive or dead in any given round, and some units have stored information concerning their special abilities.
2. The player-automata picks a set of moves for its units, the simulator detects unit collisions and then performs the necessary combat simulations.
3. The opponent either runs its own set of learning rounds to determine a good response, or plays out the predetermined moves if the scenario has any. Collision detection and combat simulations again.
4. If we're at the end of our simulation depth, we simply calculate the average performance of the last move set, and return it to the automata in the level above. If not, we recursively start a new round of simulations, increasing depth by one.

Translated to code and heavily simplified, the simulation loop looks like algorithm 1. Each unit contains a set of LA, and keeps track of which automata is currently active for each level of depth. The update methods for each automata will be described in chapter 3.2, and each depth level can contain its own type of automata, allowing for a high amount of different potential combinations.

As explained before - when the opponent side is not controlled by predetermined moves, denoted as "scripting", it runs its own learning scheme - only learning from the current turn. This fact leads to a large difference in average time consumption per run of the simulator between scenarios where the opponent is required to learn and where it can rely on predetermined moves.

The variable denoting maximum simulation rounds is determined through testing, in order to allow the automata to complete its learning, or converge, where applicable.

Algorithm 1 Main simulator loop

```
playRounds(maximumDepth){
  foreach Unit in AllUnits:
    Unit->resetAutomata
  do until maxSimRounds {
    foreach Unit in AllUnits:
      Unit->resetLocationAndAlive
    foreach Area in CurrentMap:
      Area->resetOwner
    foreach Unit in PlayerUnits:
      Unit->Automata->selectMove
    calculateCombat
    if (opponentIsScripted) {
      foreach Unit in OpponentUnits:
        Unit->Automata->selectMove
      calculateCombat
    } else {
      foreach Unit in OpponentUnits:
        Unit->resetAutomata
      do until maxSimRounds {
        foreach Unit in OpponentUnits:
          Unit->Automata->selectMove
        calculateCombat
        getStatus
        foreach Unit in OpponentUnits:
          Unit->updateAutomata
        foreach Unit in AllUnits:
          Unit->resetLocationAndAlive
      }
      foreach Unit in OpponentUnits:
        Unit->Automata->selectTopMove
      calculateCombat
    }
    if ( currentDepth < maximumDepth ) {
      currentDepth++
      status = playRounds(maximumDepth)
      currentDepth--
      foreach Unit in PlayerUnits:
        Unit->updateAutomata
    } else {
      getStatus
      foreach Unit in PlayerUnits:
        Unit->UpdateAutomata
    }
    if ( currentSimRound > maxSimRounds * 0.8 ) {
      statusSum += status
    }
  }
  return ( statusSum / maxSimRounds * 0.2 )
}
```

The last part of the engine is the unit collision and combat simulation methods. Unit collision is simple enough, at least in principle. When both the player and the opponent has units within the same area after a move, there will be combat, and when combat ends, either one or both side will have no units left in that area. If the attacker is the one with remaining units, the ownership of the area is changed, otherwise it remains in the hands of the defender.

For the actual combat simulations, several different methods had to be implemented. First of all - units have a variable value, and the player losing units can choose what units to remove from the game board. This means that units would have to be sorted according to value, to make sure that units would be removed in the appropriate order (Infantry first, then Artillery, then Tanks, for example).

Next, by throwing dice, we need to aggregate how many hits each side scores per round, and then remove that amount of units from the board. This process is then repeated until either side loses all units. While in a real game of A&AR, players can choose to retreat at any time, we have chosen to enforce all-out attacks in order to reduce the amount of options for the automata to decide on.

Translated to simplified code, the combat simulations look like algorithm 2. Attacking units use their attack strength, for infantry this also includes possible support by artillery, and defenders use their defense strength. Counting hits is a simple aggregating how many out of one sides units have a hit score equal to or below the result of a dice throw. The return value of this method is either true, in which case the area in which combat occured changes owner, or false, in which case it does not.

We will go into more detail on the gameplay of A&AR in chapter 4.

Algorithm 2 Combat simulation

```
combatSimulation(Attackers, Defenders) {
  do until oneSideIsDead {
    foreach Unit in Attackers:
      Unit->getAttackStrength
    foreach Unit in Defenders:
      Unit->getDefenseStrength
    sortAttackersByCost
    sortDefendersByCost
    attackHits = countHits(Attackers)
    defenseHits = countHits(Defenders)
    foreach attackHit:
      removeCheapestUnit(Defenders)
      if ( Defenders is empty ) {
        DefendersKilled = true
        oneSideIsDead = true
      }
    foreach defenseHit:
      removeCheapestUnit(Attackers)
      if ( Attackers is empty ) {
        AttackersKilled = true
        oneSideIsDead = true
      }
  }
  if (DefendersKilled == true) {
    if (AttackersKilled == true) {
      // Both sides dead, area owner remains the same
      return false
    } else {
      // Attacker wins, area changes owner
      return true
    }
  } else {
    // Defender wins, area owner remains the same
    return false
  }
}
```

3.2 The Learning Algorithms

As explained earlier in chapter 2, all learning algorithms work on similar principles. All our automata are used to chose between a set number of actions, and are updated according to the game status as calculated by the simulator engine. For each automata, the set of possible actions is obtained in the setup phase described in chapter 3.1. This is trivial for units that can only move one area each turn, but should we want to extend our scenarios to contain flying units, which can move multiple moves, obtaining the set of actions for these would required a tree search. Each automata then has its own method for initializing, updating itself and returning a choice, as will be described in this chapter.

3.2.1 Tsetlin Automata

As explained earlier, this is a rather simple algorithm to implement. Each automata requires only a set of N actions, a choice value ranging from 1-N, and a depth value. We have chosen to use a depth range of 1-4 striking a balance between the need for changing the choice reasonably fast and the need for some stability once a good choice have been found.

Upon initialization, a choice value is randomly selected from the size of the set of possible actions, and the depth value is set to 1. To update this automata, we use the result from the state of the game after the last moveset. Comparing this number to a randomly generated number for each automata, each will choose to reward or penalize individually. Rewarding the automata is as simple as increasing the depth towards the maximum, after which it does nothing. Penalizing it decreases depth, until it reaches 0, where it will randomly pick another choice value.

We wanted to make our Tsetlin-automata more decisive, as we would like to ensure that when a really good choice is found, a few rounds of bad luck won't immediately change the choice again. To do this, we put a random limiter on how often a punishment at 1 depth leads to a change in choice. This would be a number between 0 and 1, which we could change at will in order to optimize the performance of the automata. Through testing, we have found the optimal value of this parameter for our scenarios to be in the range 0.6-0.8, meaning that the automata will only enforce a penalty in 60-80% of the cases when the feedback suggests it. Shown in algorithm 3, which is a simplified version of the updating algorithm, this limit is the L constant. The *status* variable is the feedback from the system - a number between 0 and 1, where higher is better.

When selecting a move through this automata, no algorithm is needed - the automata simply returns the current choice value.

Algorithm 3 Tsetlin Automata - Update method

```
updateTsetlin(status){
  r = random(0-1)
  limiter = random(0-1)
  if (r < status){
    if (depth < 4){
      depth++
    } else {
      // At maximum depth, no change
    }
  } else {
    if ( limiter < L ) {
      if (depth > 1){
        depth--
      } else {
        choice = random(1-number of actions)
      }
    }
  }
}
```

3.2.2 Linear Reward Inaction Automata

For this type of automata, all automata playing in a particular round is updated the same way each round. Whether the set of automata is rewarded or not is done by comparing a single random number with the status of the current game.

Initializing the automata is as simple as giving each possible action a weight equal to 1 / amount of actions. When deciding on whether to reward the automata, a random number is compared to the feedback from the game state, and all automata are either rewarded or together.

When receiving a reward, as explained in chapter 2.2.2, each active automata is updated by increasing the chance of picking the last selected move, while reducing the chance of picking each of the others. When recieving a penalty, the automata remains unchanged. Algorithm 4 shows the update method in a simplified version. The update rate for an LRI automata can be modified in order to change the speed of learning. In our implementation, we have found that an update rate of between 0.02 and 0.05 gives us the best combination of speed and accuracy for the first set of turns, while increasing the learning rate for each depth level allows for using less simulation rounds as depth increases without significant loss of precision.

Algorithm 4 LRI Automata - Update method

```
updateLRI(reward) {
  if (reward == true) {
    foreach Weight in Weights:
      Weight *= (1 - LRI_updateRate)
    Weight[currentChoice] += LRI_updateRate
  } else {
    // No change when penalized
  }
}
```

Unlike the Tsetlin automata, there is no fixed choice for the automata to return when asked for a choice of action. Instead, the LRI depends on choosing a number through a weighted random choice. Simply put, a random number is chosen between 0 and 1, which is the total sum of our weights. The weights are then added together until we reach a sum of weights larger than this random number. The action corresponding to the last weight added is then the choice returned as the automata's decision. Algorithm 5 shows how this would look in simplified code.

Algorithm 5 LRI Automata - Selection of action

```
selectActionLRI() {
  sum, choice = 0
  r = random(0-1)
  while ( r > sum ){
    choice++
    sum += Weights[choice]
  }
  currentChoice = choice
  return moveSet[choice]
}
```

3.2.3 Bayesian Learning Automata

Implementing the BLA into our automata once we had the formulas, as described in chapter 2.3.3, was naturally very simple. We used the Gnu Scientific Library to return random numbers from the Normal (Gaussian) and Gamma distributions,[26] and the rest of the implementation was derived more or less directly from our formulas.

Initializing the automata was straightforward: We randomly select a choice from the set of possible actions, and then initialize each of the variables α, β, μ and τ for each action. Each of these variables are then updated according to the formulas described in chapter 2.3.3, which translated to simplified code looks like algorithm 6. The *status* variable is once again the feedback from the game state, a number between 0 and 1, where higher is better.

Algorithm 6 BLA - Update method

```
updateBLA(status) {
  alpha[choice] += 0.5
  beta[choice] = 1.0 / ( beta[choice] + ( tau[choice] *
      (mu[choice]-status)^2 ) / ( 2*(tau[choice]+1) ))
  beta[choice] = 1.0 / beta[choice]
  mu[choice] = ( tau[choice] * mu[choice] + status ) /
      (tau[choice] + 1)
  tau[choice] += 1.0
}
```

For selecting the optimal action, we again implemented the formula more or less directly. For each possible action, a random number is selected from the Normal distribution via another random number selected from the Gamma distribution, both shaped by our α, β and τ variables. This number is then added to the expected value, μ, giving us a random number around the expected value for this move, denoted as the *Mean* value. The *Mean* value for each possible action is then compared, and the move connected to the highest value is returned as the action of choice for that particular automata. Translated to simplified code, this process looks like algorithm 7, where the *gamma()* and *gaussian()* functions are calls to the Gnu Scientific Library, returning a random number from the appropriate distribution, and *rng* is a random number generator included in the same library.[26]

Algorithm 7 BLA - Selection of action

```
selectActionBLA() {
  initialize maximum, mean
  foreach Move in Moveset:
    variance = 1.0 / gamma(rng, alpha[Move], beta[Move])
    mean = mu[Move] + gaussian(rng, squareRoot(variance /
        tau[Move]))
    if (mean >= max) {
      max = mean
      choice = Move
    }
  return MoveSet[choice]
}
```

4 Experiments

4.1 Introduction

To understand the foundations of our simulations, we first need to go a little deeper into the gameplay of A&AR. For a more thorough explanation of rules, see the A&AR Manual.[7]

Each scenario has two main types of components: The individual units of different kinds, and the land for which they are fighting. Land areas come with an attached value, which in the actual game reflects its production potential. Units come with a range of different attributes, for some even unit-spesific abilities. Each unit has a production cost, which will be used by our LA in the calculations of gain and loss.

The subset of units we will be using for our preset scenarios, to test the abilities of our LA, have the following properties:

Unit	Attack	Defense	Cost	Notes
Infantry	1 (2)	2	3	+1 with artillery
Artillery	2	2	4	Can support infantry
Tank	3	3	5	Can Blitz*

*Blitz: These units can normally only move from one area to the next each turn, but if the area a Tank is moving into is friendly or unoccupied, they get another, free move.[7]

Table 1: Basic Unit Properties

In addition to these units, there are two kinds of buildings available. One is the factory, which in the game is used to mark what areas on the map you can place reinforcements on. The placement of reinforcements, and therefore the function of factories, have not been included in our testing. Even so, factories still have a high value and their ownership gets transferred upon conquering a territory, so we look at them as simply increasing the value of the territory they are in. The second is the anti-air unit, which is not included in our testing.

Combat occurs when units of any faction move into an area containing units from the opposing faction, and ends either when the attacker withdraws or all units on one side have been destroyed. The actual combat calculations, as would be expected from a board game, is done by dice throws. The attack and defense strengths of each unit is equal to the highest dice number on which they will score a hit. For example, an attacking Tank unit will score a hit on any dice result from 1 through 3, while a defending Infantry unit will score a hit on a dice result of 1 or 2. For each hit, except in the case of a special ability not used in our subset, the opposing player will remove one unit of his own choice from the game board.

The opposing side gets the same chance of causing damage to the player, however, before the pieces are removed. This means that bringing an excessively large force will not totally overrun the enemy without taking casualties, but it will likely be victorious in fewer rounds of combat. In quite a few cases, depending on how well balanced the two sides are, this rule leads to mutual destruction - neither side has units remaining. When this occurs, the map remains as it was before the combat started, giving the defending side a slight edge, as the attacker is required to have units surviving combat to take control of the attacked area.

Figure 5 shows an example of combat, in a screenshot taken from the TripleA implementation, showing a Russian (Red) force attacking a defended area owned by the Germans (Grey).[5]

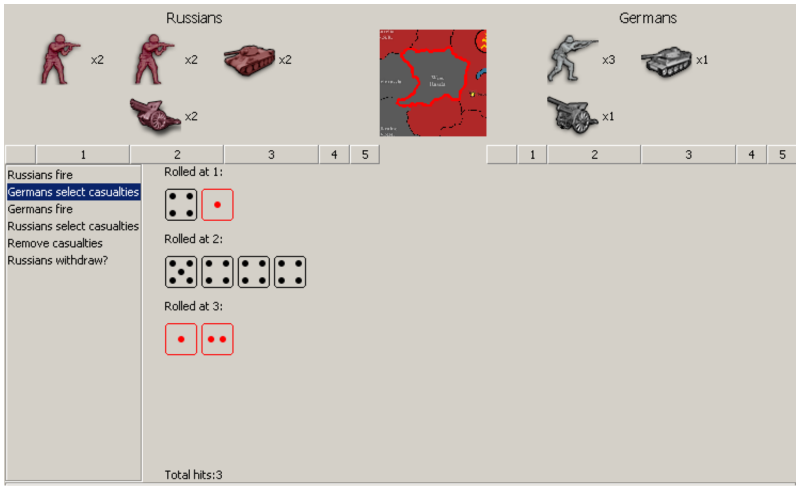

Figure 5: Combat calculation example

In this example, we can see several properties of combat calculations. Two of the attacking Russian Infantry units are supported by Artillery and thus gain an attack strength of 2. The other two Infantry has an attack strength of 1, while the defending German Infantry have a defense strength of 2. The two unsupported Russian Infantry scores one hit with a dice throw of 1, and the two Tanks, each with an attack strength of 3, scores two hits with two dice of 1 and 2. The Germans lose three of their units, and will remove three infantry units from the map after having returned fire. After both sides have thrown their dice and units have been removed, the Russians get the choice to keep attacking or to retreat. If they choose to continue the attack, the formula above is repeated.

4.2 About Scenario Analysis

In order to demonstrate what should be expected from our LA, or even the optimal play for some scenarios, we have chosen to do some statistical analysis. The outcome of various combat scenarios have been estimated using an online combat simulator by Daniel Rempel, simulating each combat scenario 10,000 times.[25] The average expected IPC gain from each move has been calculated by hand, and some assumptions had to be done in order to get a fixed number. As such, the probabilities for the outcomes are precise within tenths of a percent, but the average IPC values might vary slightly.

We have included graphic representations for most of the possible combat scenarios, and analyzed how each outcome from these fights works out in terms of points gained and lost. A positive average point value means the outcome is beneficial to our automated player, while a negative value means it is beneficial to the opponent, whether it is an automata or scripted. In the graphic representation pie charts showing the distribution of winning and losing, the red color types always represent the attacking player, while the blue color types represent the defending player.

Finally, the maps which we have retrieved from the TripleA implementation of A&AR allow us to use real areas of the game as the background for our scenarios. However, the values of these territories in our scenarios does not reflect their value in the actual game, but have been altered in order to suit our purposes.

4.3 Scenario 1: Simple test map

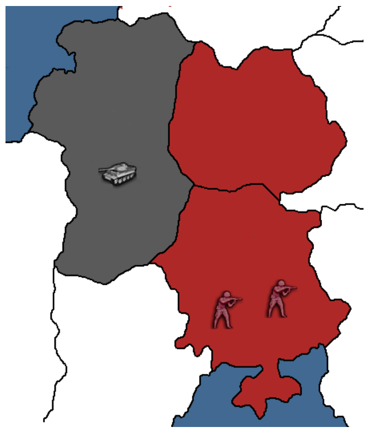

Figure 6: Scenario 1 - Simple test map

One team of LA, from now on referred to as the player, controls two infantry units and two zones, playing the Russians (Red). Another LA, referred to as the opponent, controls one tank unit and one zone, playing the Germans (Grey). Zone worth is equal at one point each. The objective is to ensure that more than one LA can properly assess a relatively simple situation over more than one turn, and show that it is capable of determining the appropriate reaction to a basic situation. In particular, we are looking for the LA controlling each of the infantry units to cooperate in beating the tank. The LA controlling the opponent, as explained in chapter 3.1, is limited to only learning from the current turn, while LA controlling the infantry can recieve feedback from the two turns ahead we are simulating this scenario.

4.3.1 Scenario Analysis

The first round of this scenario, where the player, our multi-tiered LA, makes its move, essentially gives 4 different move combinations, each with a set of responses for the opponent.

- Both infantry either stay in place or moves to the northern area to defend, which is essentially the same outcome. The opponent can then choose either to stand still, essentially delaying the combat phase until next round, move into the empty territory for a minor gain in points, or attack the two defending Infantry.
- One infantry moves north while the other stay put, to defend both areas. The opponent can then either stand still, delaying combat another round, or attack one of the defending infantry. Should the tank attack and destroy one of the infantry, there would be another set of choices for the next round.
- Both infantry attack the tank, after which the opponent's options depend on whether they win or lose. If the infantry won, the opponent has no options. If they lost, the tank can take both areas at its leisure.
- One infantry attacks the tank while the other defends either the southern or northern area. As above, if the infantry wins, the opponent has no options. Otherwise, the tank can either attack the remaining infantry, occupy the empty territory or stand still.

4.3.2 Both infantry defend, tank attacks

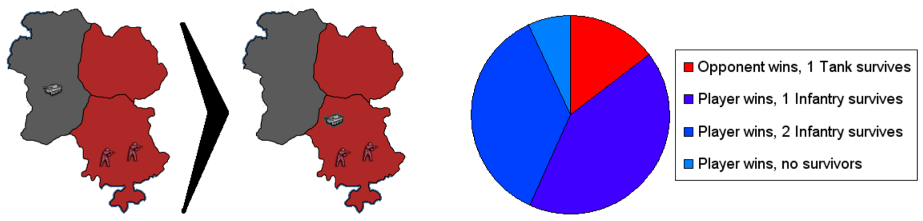

Figure 7: 1 Tank attacking 2 Infantry

First up, we have both infantry defending the same territory. For this example, we have used the southern territory in which they start out, but defending the northern territory would naturally give the same outcome. The only action which gives a combat scenario from this choice of action is if the tank choose to attack the defended territory, which is shown in figure 7.

If attacked by a tank, there is an 85.3% chance that the player will keep its territory, defeating the attacker. There is a 78.4% chance that at least one of the infantry will survive to be able to take over the opponent's territory during the next round. If the tank wins, which happens 14.7% of the time, it can capture the last territory during the next turn.

If the opponent wins, he will have gained 6 points for killing the infantry and 1 point per territory for a total of 8. If one infantry survives, the player will gain 5 points for killing the tank, lose 3 points for losing the infantry, and gain 1 point for taking the territory the next turn - a total gain of 3. If both survive, the player would not lose the 3 points, gaining a total of 6.

If no units survive, the player will have lost 6 points worth of infantry and gained 5 for killing the tank. If the tank used the blitz ability to move through and capture the unoccupied zone, the player would lose another 1 point, for a total gain of either -1 or -2.

Combining the percentages with the gains, the average gain for the player for this particular outcome comes to 2.69 points. This means that defending together against an attacking tank is highly beneficial. However, as the opponent is also controlled by an LA, he should be able to determine that attacking two defending infantry with his single tank is not in his interest, and either stay put or take the empty territory instead. Taking the empty territory would leave the player with a -1 gain from losing it, but during the next round it could simply move its infantry into the now empty western territory, essentially putting buth non-agressive options at 0 point gain for either side.

4.3.3 Infantry splits up, tank attacks

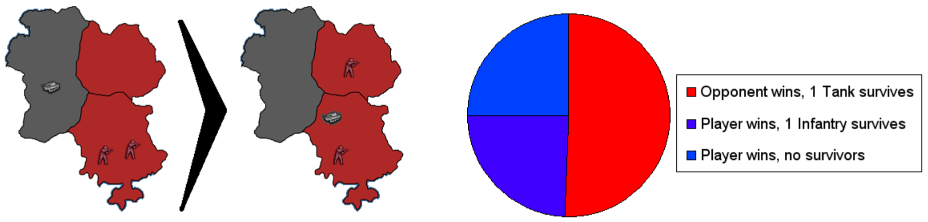

Figure 8: 1 Tank attacking 1 Infantry

With one infantry defending each territory, the LA-controlled tank is much more likely to attack. Given a lot better odds than in the last scenario, as seen in figure 8, there is a 50.5% chance that the tank will beat one infantry. The player has a 24.2% chance that the infantry will win, and another 25.3% chance both will be killed, leaving the area in the hands of the player.

If the tank wins, it can attack the other infantry during the next round, for the same odds, and perhaps end up in control of the entire scenario. If the infantry wins, or at least destroys the tank, the player can take and keep all 3 areas. The tank losing the first round will give the player a gain of 5 points for the tank and 1 point for the western area . If the tank wins, however, we must calculate in the average for another round of combat - this time with the player starting at -3 gain, having lost an infantry unit. The tank in this position has the same choices as before - he can either attack or stay.

Attacking will give the same odds as in figure 8, with slightly different point gains. Starting with -4 points for the player, the tank winning another battle will leave the player with -8 points. The infantry winning gives the player the opportunity to recapture the taken lands for a total of 3 points gained, while both units being destroyed gives a gain of -2. Assuming that the tank attacks again if it wins the first fight, we get an average gain from that round of -3.82 points.

With the final option of both units being destroyed in the first fight also giving the player automata a gain of 3 points, we end up with a total average of 0.28 points when including a second turn of combat. Should the tank instead choose to stay during the second turn, we end up with an average of 0.19 points.

Summing up, splitting the infantry might immediately appear as weakening the position, while it is in fact beneficial to the player. The reason for this is primarily the fact that tanks are more valuable than infantry, and yet does not get a significant advantage in small-scale conflicts. In this fight in particular, we see the strength of defense in small combat scenarios. The rule that leaves the area in the hands of the defenders when all units are destroyed gives the defending infantry in this scenario twice the chance to defend successfully.

4.3.4 Both infantry attack

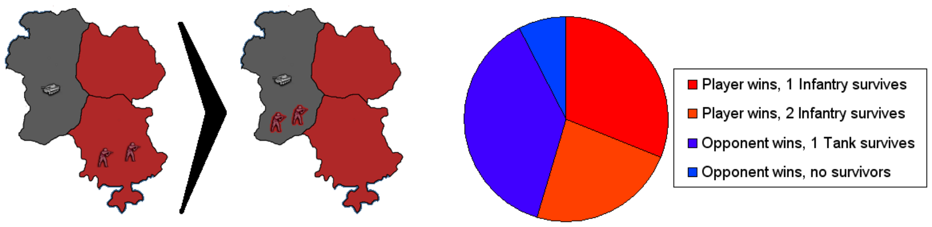

Figure 9: 2 Infantry attacking 1 Tank

Figure 9 shows a preemptive strike - attacking with both infantry - which gets reasonably good odds considering the low attack value of Infantry. There is a 54.4% chance of defeating the tank and taking the territory, while the tank has a 38% chance of survival.

Beating the tank gives the player complete control of the scenario, while losing gives it to the opponent, as the tank is free to take the other two areas. One infantry surviving gives the player a gain of 5 points for the tank, -3 points for lost infantry, and 1 for territory, a total of 3. Two surviving infantry yields a gain of 6 points. If the tank wins, the player gain is -6 points for the lost infantry and -2 for the lost areas, totaling to -8. If all units are destroyed, the gain is -6 for lost infantry and 5 for the tank, totalling to -1.

The average gain from picking this action comes to -0.78. As expected, attacking a good defensive unit with two weak offensive units is not beneficial, but at the same time the difference between a good and a bad choice is really not the largest in this particular scenario.

4.3.5 One infantry attack alone

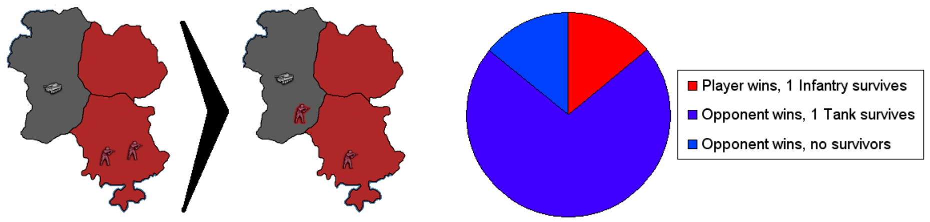

Figure 10: 1 Infantry attacking 1 Tank

To illustrate what is clearly a bad choice, figure 10 shows the last option open to the player LA. Attacking with only one of its infantry gives 14.3% chance of victory, but in total 28.5% chance of destroying the tank. In that case the player has control of the entire board, even if the infantry may be lost. The more likely scenario, at 71.5% chance, is that the tank survives, to be able to attack the remaining infantry unit at the same odds as shown in figure 8.

The infantry winning gives the player a gain of 6, and both the tank and infantry getting destroyed gives it a gain of 2. If the tank wins, we get the same calculation as shown under figure 8 - assuming the tank follows up with an attack on the remaining infantry - the average gain is -3.82 points.

The average gain from picking this scenario, still assuming the tank follows up with an attack, totals up to -1.44 points. If the tank decides to just take the empty territory instead and await the next move, we get a total of -1.58 points instead. All in all, clearly the worst move open to the player LA, and hopefully one it will avoid as much as possible.

4.3.6 Summary

To give an overview on the various outcomes of this scenario, table 2 shows the average gain on various movement combinations ordered by their expected performance.

With automata playing both sides in this scenario, we are expecting the average results to be as close to 0 as possible, as both sides will go for the situation that gives them the best outcome on average. If both infantry defend together, the tank LA is highly unlikely to pick the attacking move, as the average gain for it would be awful. As both sides idling would give no gains for

Movement choice	Average gain
Infantry defend together, tank attacks	2.69 points
Infantry split up, tank attacks twice	0.19 points
No attack by either side	0 points
Both infantry attack	-0.78 points
One infantry attack, tank responds	-1.44 points

Table 2: Scenario 1 - Average point gain by move choice

either side, we expect the infantry LA to split up in an attempt to lure the tank LA into attacking. While this would on average be a slightly worse move than standing still, the difference might be small enough that the tank AI would go for it anyway.

With both sides of automata working properly, we are expecting minimal use of either end-point in this table. In fact, if both sides of automata play optimally, we expect to see a complete standstill, with neither side performing an attack.

4.4 Scenario 2: Rushing to the defense

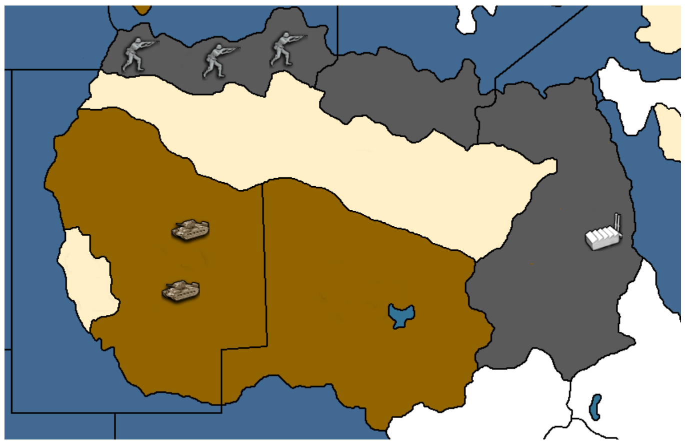

Figure 11: Scenario 2 - Rushing to the defense

Playing the Germans (Grey), the Learning Automata side, referred to as the player, controls three infantry and three zones, one of which is valuable. the area itself is worth 15 points, and contains a factory worth another 15 points. The opponent, playing the British (Brown), controls two tanks and two areas, and is preprogrammed to move both tanks one move east per turn. The values of the four western territories is neglible at 1 point each. If the opponent tanks reaches the area with the factory first, the infantry will not realistically be able to recapture it, due to the low attack value of infantry compared to the defense value of tanks.

Hence, the goal of this scenario that the LA learns that it must move all three infantry to the valuable area in the two first turns in order to protect it from the tanks inevitable attack. While in an actual game, the tanks would be able to use the Blitz ability to reach the valuable zone in one move, we have set them to move one area each turn for simplicity: It is the relative attack and defense strength of the units that are interesting, not how far away the tanks would have to be in a realistic setup. This scenario have been set up to determine if our multi-tiered LA can overcome the fact that no action taken during the first turn will give any reward - the only rewards possible comes from performing the correct set of moves through two turns.

The choice and numbers of units are carefully calibrated to ensure that the LA has the best possible feedback to work with once the second turn is played - one or two defending infantry will most likely lose against two tanks, but with all three infantry together, there is a good chance that the area remains in the players hands. If the area is not defended immediately, however, the chances of recapturing it with the units present are slim to none. To make sure that all options can be tested, including attempts at retaking the territory on the third turn, this scenario will be simulated three turns ahead.

4.4.1 Scenario Analysis

While this map might look more complex, it is in fact a lot simpler to break down than scenario 1, because the opponent has been preprogrammed in order to properly test the automata. No matter what moves the LA picks, in its second move, the opponent will move both its Tank units into the valuable area and stay there.

This setup gives us two different types of outcomes, based on what the LA has done with its own moves. Either it is attacking, because it was too slow in moving its units into the valuable area, or it is defending, because it got there first. The outcomes of the actual combat phases, depending on what the LA has decided to do, are as follows:

- If the LA gets to the valuable zone first, we get 2 Tank units attacking either 1, 2 or 3 Infantry units.
- If the opponent gets to the valuable zone first, we get 1, 2 or 3 Infantry units attacking the 2 Tank units. The LA might also decide to do nothing, seeing that the defense is too powerful.

Because of the setup of the map, point gains in this scenario might look a little off. The best outcome the player might hope for, destroying both the tanks and taking the southern area, only gives 12 points gain. The absolute worst case, seeing as the tanks are forced to stay in the valuable area, would be losing all units and the valuable area. This would give -36 points of gain. As might be expected, the average gains are all skewed into the negatives, but this does not make them less useful for the automata when determining which one is the optimal.

For simplicity, we have omitted the full graphical presentation of some of the least beneficial moves.

4.4.2 Two infantry defending

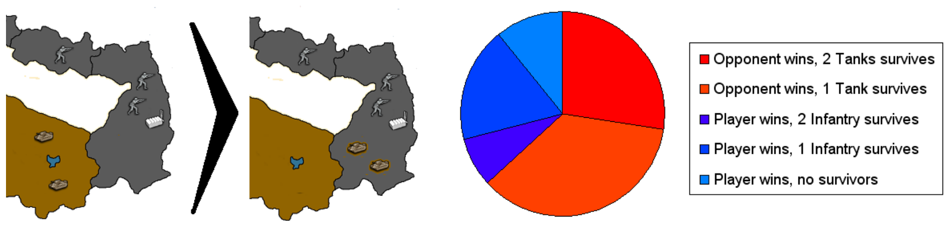

Figure 12: 2 Tanks attacking 2 Infantry

As can be seen in figure 12, defending the profitable area with only two infantry only gives the player a small chance of keeping it safe. The tanks have 63.1% chance to take the area, and then either a 85.7% chance of defending the area from the single infantry - like we saw in figure 10 - if one tank survives, or 99.3% chance to defend if both tanks survive.

Average point gains are as follows: With two tanks surviving, the player gain is -36, assuming the last infantry does not suicide itself. With one tank surviving, the gain is -31 under the same condition. Both infantry surviving gives a gain of 12 points, with infantry capturing the last two zones, while only one survivor gives a gain of 9. The no survivors option here leaves the player LA with a single infantry left behind, which can take the empty areas for a total of 6 points of gain.

Summing up we get an average gain of -17.7 points from defending with only two of the three infantry.

4.4.3 Three infantry defending

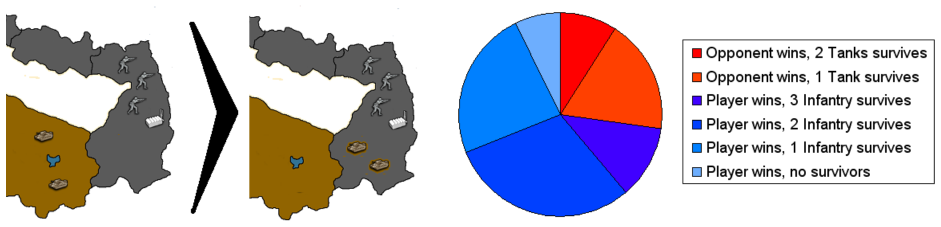

Figure 13: 2 Tanks attacking 3 Infantry

Bringing all three infantry in the defense, as seen in figure 13, is really the only way the player LA can expect victory. With a total chance of 72.9% to destroy both tanks and keep the area safe, it yields much better results than only bringing two.

Average point gains are as follows: Two tanks surviving gives a gain of -39, while one tank gives -34. All infantry surviving, on the other hand, gives the player the chance to take the empty areas for a total of 12 points of gain. Similarly, two infantry remaining gives 9 points of gain, and one remaining gives 6 points. No survivors in this scenario evens out at 1 point of gain, with two tanks killed worth 10 points and three infantry worth 9.

Combined with the chances for each outcome, we get an average gain of -4 points, quite a bit better than only defending with two infantry.

Note that we have omitted the choice of only defending the territory with one infantry. Considering that it only gives a chance of 4.8% to successfully defend the territory, and a full 60.5% chance that both tanks would survive, it should not be necessary to include a figure to show why this would be a bad choice.

4.4.4 Three infantry attacking

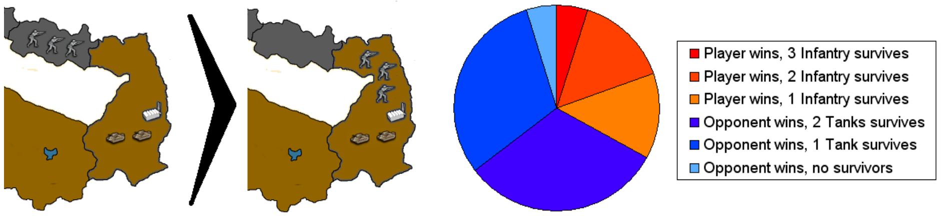

Figure 14: 3 Infantry attacking 2 Tanks

To demonstrate why waiting for the tanks to enter the territory before attacking would be a bad idea, we have included the best case scenario from this possibility. Figure 14 shows the outcome of three infantry attacking two tanks, only getting a 33% chance of success. As the tanks have already taken the area worth 30 points, we get the following average gains:

Three, two and one infantry surviving and able to take the empty territories, gives 12, 9 and 6 points respectively. Two tanks surviving gives -39 points, and one tank gives -34 points. No survivors in this case leaves the area in the hands of the opponent, giving a gain of -29 points.

Combined with the chances for each outcome, this gives us an average point gain of -21.4, clearly a lot worse than using the three infantry to defend rather than attack.

Again note that we have omitted the choice of only attacking the defending tanks with one or two infantry. These options have a success rate of 0.7% and

9.9%, respectively, and average point gains in the negative thirties, and a figure should not be needed to explain why either would be a bad choice.

4.4.5 Summary

An overview of the average point gains for various actions ordered by expected performance is shown in table 3.

Movement choice	Average gain
Move three infantry to defend	-4 points
Move two infantry to defend	-17.7 points
Move three infantry to attack	-21.4 points
No movement	-30 points

Table 3: Scenario 2 - Average point gain by move choice

All other options gives even worse average gains, and should not require looking at further. It should be clear that the only beneficial choice for the player side LA is to work together, bringing all three infantry into the valuable area before the opponent attacks. If this was a one-turn battle, with infantry and tanks in the neighboring areas, this would be an easy task for any automata type. By design, we are forcing the automata to see beyond the immediate next turn, hopefully showing that our multi-tiered structure can process this information correctly and display basic proactivity.

4.5 Scenario 3: Reacting to opportunity

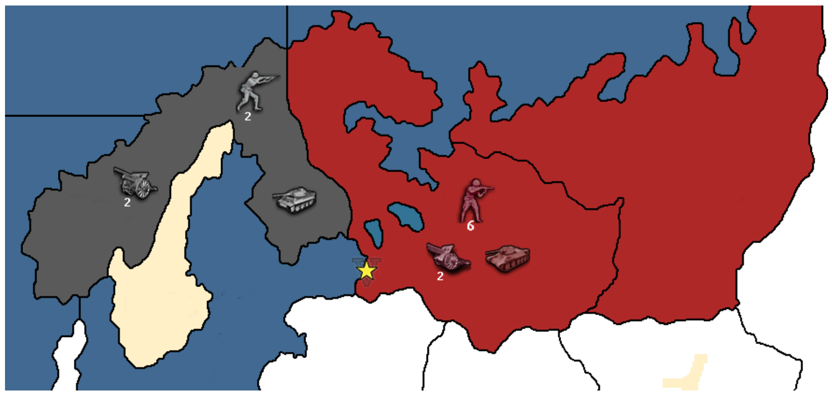

Figure 15: Scenario 3 - Reacting to opportunity

In this quite a bit more complex scenario, both the our LA player, playing the Germans (Grey) and the preprogrammed opponent, playing the Russians (Red) controls a varied force, with the opponent having four more infantry units.

The LA controls one zone, while the opponent has two, of which one contains a "Victory Point" - making it more valuable - a total of 10 points. After the first move by the LA, the opponent is programmed to respond to an imagined threat to the east and move some of its force - 3 infantry, 1 artillery and the tank - into the eastern territory. The goal of the scenario is to ensure that the LA is able to see and take opportunities as they arise, by attacking a valuable territory when the opponent leaves it underdefended. To achieve this, the LA first has to conclude that any actual move during the first round would be a bad idea, as the defending force is too big, and then attack on the next turn when the opponent is weaker.

4.5.1 Scenario analysis

An even more complex scenario, this requires the LA to coordinate five units to stand and move as one, or lose. For simplicity, we will only show the two major outcomes of this particular battle, in order to demonstrate why deviating from the path is not really profitable. The LA has two groups of options for this scenario, and only one of them have a realistic chance of a reward. The options are as follows:

- Attack immediately during the first turn with any amount of units.
- Hold position with all units during the first turn, and then attack with any amount of units after the opponent has moved his units.

Because of the scripted nature of the opponent, there is no risk of losing the originally owned area. The worst outcome for the player LA is thus losing all its units, worth 19 points. The moved opponent units will stay in the eastern area - picture them as lost to an external threat - and the best case scenario involves killing the units remaining in the middle zone and taking it over - worth 13 points for the units and 10 for the area for a total of 23. Technically, killing all enemy units and taking both areas would be best possible case at 42 points, but as we will see, this is not a realistic goal.

As both of these options in practise only requires two turns to be played out, this is how far ahead we will simulate for this area.

4.5.2 Attacking during the first turn

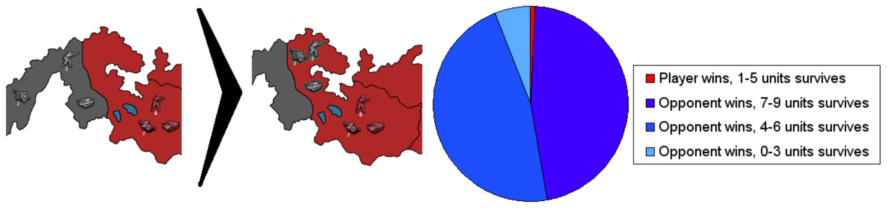

Figure 16: Attacking during the first turn

To demonstrate just what a bad move attacking during the first turn is, figure 16 shows the chances for various outcomes. Due to the large amount of units involved, we had to simplify the counters for surviving units. With a mere 0.7% chance of victory, and even then most likely with only a single unit remaining, attacking before the opponent moves out of the zone is truly a bad choice.

The precise counts of point gains for each and every single outcome of this scenario would be needlessly long and tedious - so we will only note the actual end results here. With a maximum gain of 42, as noted above, and a minimum of -19, this move receives a meager average at -10.3 points.

Note that attacking with less units will only lead to even worse probabilities and possible gains, as fewer units would have a minimal chance at even surviving a single round of combat against such odds.

4.5.3 Attacking after opponent move

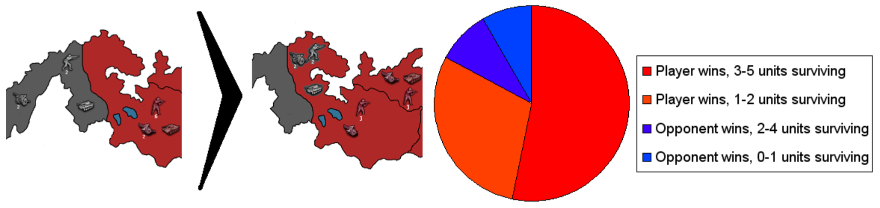

Figure 17: Attacking after opponent move

By waiting until the opponent have moved a portion of his forces out of the middle area, and then attacking with all units, we get scenario shown in figure 17. Instead of a more or less guaranteed loss, we are now looking at a more or less secure victory - 83.1% chance of winning. The opponent is scripted to not respond by attacking the middle area, as that is not what this scenario was designed to test. If this seems unrealistic, the reader can feel free to explain the removal of the amount of units shown above in another way instead - perhaps they were destroyed in an attack on an enemy not shown in our scenario. For our purposes, it is sufficient that they are no longer part of the defense force, and we will not need to account for them attempting to recapture the middle area.

That being said, we will examine the average gains for this set of moves. As in the previous set, counting points for every outcome would be long and tedious, and we will again show only the end results. With a maximum gain of 23 points, since most of the enemy force have moved out and we will not be taking the eastern area, and again a minimum gain of -19, we get a much more beneficial average of 11.6 points gained.

Again, note that attacking with less units will quickly reduce this gain to the point where it is no longer beneficial - as small a change as a single unit staying behind, be it an infantry or an artillery, will reduce the players chance of victory to an average of 56.5%, with an average gain of 5.6 points. Removing an additional unit takes that number down to 27.4%, with an average gain of a meager 0.3 points. In addition, these numbers are based on the fact that the opponent would not retaliate should he win, which in a real scenario would be highly unlikely. Again, it is clear that the biggest challenge for the player team of LA in this scenario is getting all the LA to cooperate over more than one turn, in order to get the optimal rewards.

4.5.4 Summary

An overview of the average point gains for various actions ordered by expected performance is show in table 4.

Movement choice	Average gain
All units attack during the second turn	11.6 points
All units but one non-tank attack during the second turn	5.6 points
No movement	0 points
All units attack during the first turn	-10.3 points

Table 4: Scenario 3 - Average point gains by move choice

We observe that even taking no action at all is better than any form of attack during the first round, and that the best gain clearly comes from performing combined move we designed this scenario for. A single infantry or artillery staying behind may give some reward, but the difference in reward between that option and the correct choice is large enough that there should be no real reason for the automata to use it, should it first have decided to wait during the first turn set.

5 Results

Our results are relatively straightforward to understand. For each scenario and each type of LA, we have followed the same formula. Through testing, we determine what amount of simulations is required to complete learning, or converge, depending on the type of automata.

For an assessment of the performance of each algorithm when it comes to time and computational power required, we have noted the average run times of each scenario, which have been run on a dedicated 3.0Ghz Intel E8400 CPU. This will then be used in our discussion on whether this approach has practical applications on current hardware.

For each scenario, we have then aggregated empirical data over several hundreds of complete simulation runs to gain a statistical figure on how each type of automata performs. To ease the explaining of each result, we have given a graphical presentation. Finally, we have analyzed the performance of the various LA, in an attempt to explain the reasons behind each outcome.

5.1 Scenario 1

Because several possible move combinations in this scenario are close to each other when it comes to how many points are to be gained, our automata were divided in what move to choose from the set described in chapter 4.3. As such, we will devote a section to each type of LA for this scenario. Each type of LA is playing against an opponent controlled by the same type of LA, with the goal being that they would converge onto the choice of moves closest to equal gains for both sides. As the only truly equal gain in this scenario comes from both automata defending - as the value of the areas are not high enough to risk an attack - the optimal outcome would yield a net gain of 0.

As explained in chapter 4.3, any outcome of this scenario can be determined after two turns of both players selecting their moves, and so this is the depth of simulation we have chosen to use.

Tsetlin

Through testing, we determined that the best results for this type of LA was obtained when the first turn was simulated at least 500 times, for each which the second turn was simulated at least 300 times. Average time consumption per run was 115 seconds.

Figure 18 shows two things. First, how often the two infantry units picked each combination of moves throughout our test runs. Second, how often the tank unit chose each of the two possible responses to each of the choices where the infantry are defending - either the two of them together, or one in each territory. The tanks response to the last two options, meaning what it chooses to do after having been attacked, has been omitted for simplicity, simply because it gave the same results every time: Depending on the outcome of that battle, the tank would be either dead, in which case it would do nothing, or it would attack, seeing as at least one of the infantry would already be dead.

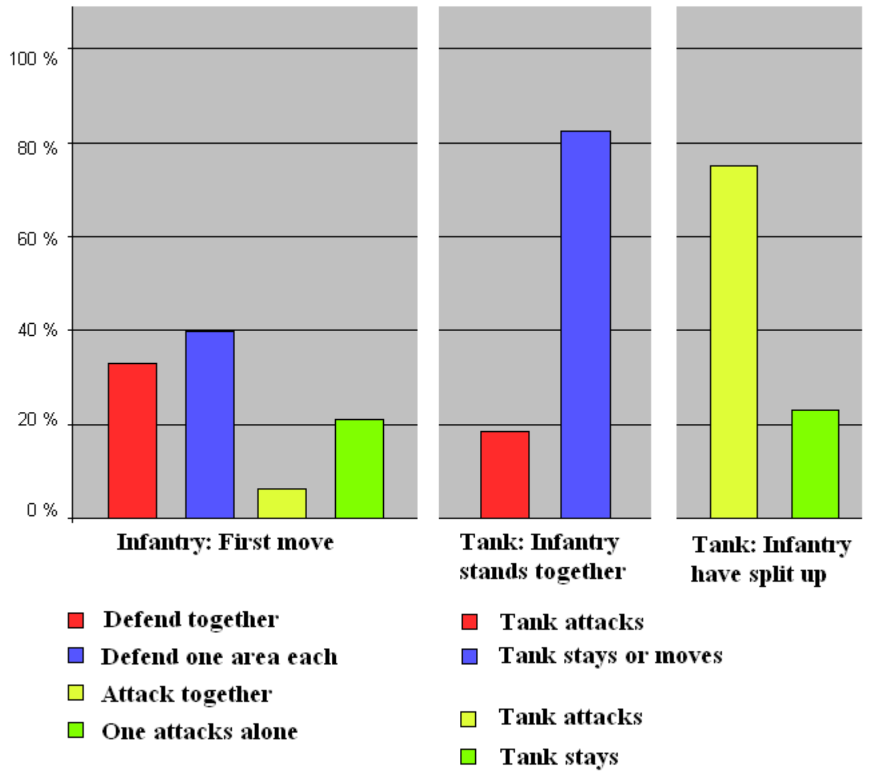

Figure 18: Results - Scenario 1 - Tsetlin - Graphical representation

As can be seen from the graphic, the Tsetlin, even with its reduced chance of performing a penalty, is not particularly decisive. While it is understandable that it has problems choosing between the defense scenarios, it is more worrying that it seems to have a preference for the worst move of all - attacking with a single infantry, even above the slightly better choice of attacking with both together. The opponent has its own share of problems - attacking when the infantry are defending together is possibly the worst move of all - however it does not pick this one very often. We note that the opponent has a preference for attacking when the infantry split up, even if we have found that this is slightly less beneficial than standing still, but the difference in point gains between these two options is minimal, as explained in chapter 4.3.3.

LRI

In order to converge properly in this particular scenario, the LA required the first turn to be simulated at least 800 times, for each which the second turn was simulated at least 400 times. Average time consumption for each run was 260 seconds.

Figure 19 shows, as for the Tsetlin Automata, how often each movement combinationn was chosen after learning had completed. Again, the final option - what the tank would choose after having been attacked by either one or two infantry - have been omitted, for the same reasons as above.

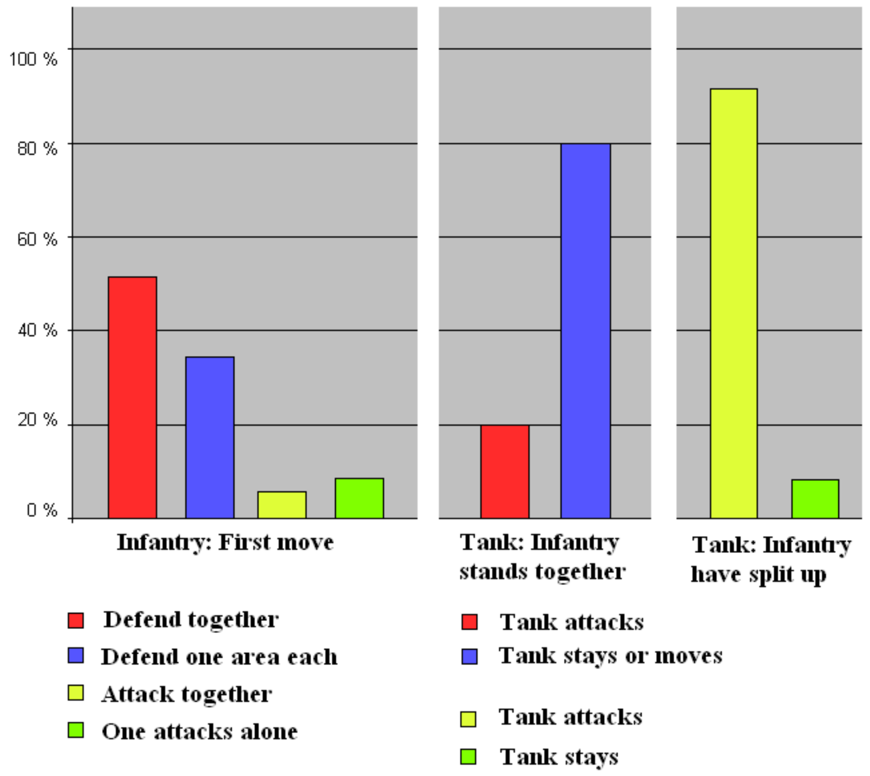

Figure 19: Results - Scenario 1 - LRI - Graphical representation

The infantry controlled by the LRI is performing quite a bit better than those controlled by the Tsetlin, picking the better choices - either of the defense combinations - more often. The LRI automata controlling the tank is performing in a similar way as above, with a higher preference on attacking in both outcomes compared to the Tsetlin. In total, the infantry is doing slightly better overall, while the performance of the tank is slightly worse.

BLA

In order to converge properly, the BLA required the first turn to be simulated at least 700 times, for each which the second turn was simulated at least 300 times. Average time consumption for each run was 170 seconds.

Figure 20 shows which combination of moves the three BLA-controlled units chose at the end of the learning process. Unlike for the LRI or Tsetlin Automatas, there are really no need for the secondary columns here - the units only perform two different moves, both which yield the exact same outcome.

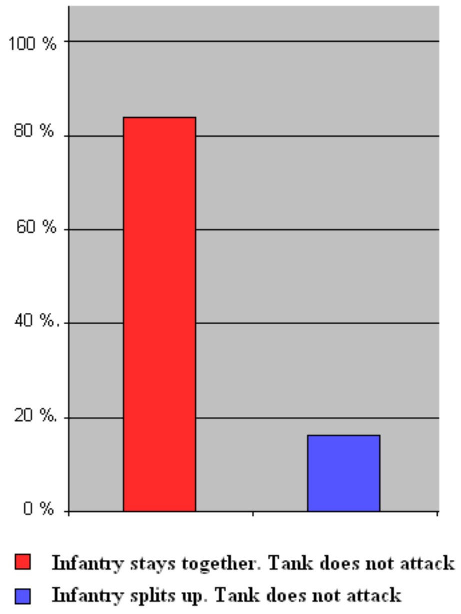

Figure 20: Results - Scenario 1 - BLA - Graphical representation

As can be seen from the graphic, the BLA is highly decisive compared to the other two types of LA. Even if it might appear to have alternated between two options, they are in essence the same, giving the same outcome. Either the infantry defends together, or they split up to defend, and in both cases the tank decided to stay, seeing that any attack would lead to a minor loss in points. The outcome of all scenario runs using a BLA on both sides is thus 0 gain for either side. Seeing as any deviation from this course of action would cause either automata to lose points, it is fair to assume that the BLA is in fact precise enough to detect even the small variations between each outcome when given feedback from the game.

Analysis

Overall performance for each type of automata is much as expected - varying in precision. The Tsetlin, with its random searching between choices to find one which yields consistent rewards, end up with the worst performance, deciding on the worst move combination possible for the two infantry - a single infantry attacking - in 21.2% of the runs. Similarly, the tank chose an even worse move - attacking two defending infantry - 18.2% of the runs where the infantry chose this combination. The most likely explanation of these deviations from the expected results, not only for the Tsetlin, but overall for this scenario, is that the differences between an optimal move and a less than optimal move is small in terms of calculated game state. This is especially true for the LA controlling the tank, which only looks at a single turn when making its decision.

The performance of the LRI automata may look similar, but is in fact a fair bit better, at least for the side in charge of the infantry. Compared to the Tsetlin deciding to use either form of attack in 27.3% of runs, the LRI only attacks in 13.2% of runs. The tank, however, is actually doing slightly worse - attacking more in both outcomes where the infantry defends. The reason for this might very well be that several choices for each automata are in fact the same, and thus the LRI might have trouble deciding upon just one of them.

For example: Whether the tank choose to move or stay when two infantry are standing together are in essence a nearly identical decision - the change of points caused by the opponent controlling two very low-value territories is neglible when it comes to rewards. The same goes for deciding which of the infantry to attack when they split up, defending one territory each - the effect is identical, but the LRI will still attempt to converge upon one of them.

The performance of the BLA is in its own category - seemingly cooperating to achieve the best outcome for both sides. With the best two moves for the infantry involving not attacking, and the least damaging move for the tank in either outcome actually being to not attack - the BLA seems to have precisely estimated the statistics behind the outcomes of this scenario. As long as it is given the time to converge, the BLA shows itself as a highly precise tool, detecting even the minute differences between moves described in chapter 4.3.

And the time to converge is an important factor here. The Tsetlin, by design, needs less rounds and thus less time before giving a conclusion. More suprisingly is that the complicated BLA actually converges faster both in turns and in time than the simpler LRI automata. For scenarios as small as this, the BLA is not as inefficient as we first thought, but as the amount of simulation rounds required to converge increases, the BLA gradually works slower, as described in chapter 2.3.3.

5.2 Scenario 2

As could be seen in chapter 4.4, there are really only two options for outcome of this scenario: The LA controlled side either does what we intended it to do, or it does not. The difference in average point gain between choosing the moves we intended it to learn, namely using the two turns available to move all three of its infantry into the valuable zone, and other combination of moves, is simply too big.

Unlike in scenario 1, the countermoves chosen by the opponent is not of interest - we already know what moves it will pick. Instead, it is interesting to us to see what move the automata have selected as its first and second set of moves during simulations. In order to select the correct second move, first all three LA controlling the infantry need to cooperate, moving into the second territory. In order to get rewarded for this move, however - the automata also need to choose the correct second move - otherwise the first set of moves would be penalized.

To receive feedback concerning all realistic options in this scenario, we need to allow the LA-controlled automata to play through three turns - allowing it to attempt attacking the valuable territory after the tanks have entered it. Because of this, the search depth of this scenario is three turns for all automata.

Also in contrast to scenario 1, in addition to using each type of automata for all three turns, we also test the performance of combinations of automata. For these tests, we have used the slower LRI and BLA automata for the first tier of automata, and the Tsetlin for the second and third turn.

In order to complete learning, and in the case of the LRI and BLA, to converge properly, each of our automata required their own minimum amount of simulation rounds for each depth level, giving each of them their own average time consumption per run, both of which are shown in table 5.

Type of Automata	Turn 1	Turn 2	Turn 3	Avg. time
Tsetlin	500	200	100	78 sec.
LRI	700	400	200	405 sec.
BLA	600	300	150	436 sec.
LRI + Tsetlin	700	200	100	116 sec.
BLA + Tsetlin	600	200	100	118 sec.

Table 5: Scenario 2 - Required amount of simulation runs and time consumption

Requiring a similar or lower amount of rounds during the first and second round as in scenario 1 can be attributed to a combination of two factors: This scenario is much clearer when it comes to what move choices are profitable, but at the same time, our LA is required to coordinate an extra unit over another move. The final turn requires significantly less simulations, seeing as its role is simply to teach the automata that any sort of attack on the defensive position is far from optimal - thus the feedback is more likely to be precise.

We can also see the expected outcome from our performance tests using combinations of automata. While using the LRI and BLA for all tiers takes many times longer than the Tsetlin alone, only using either the LRI or BLA for the first turn cuts large amounts of run time.

Figure 21 shows the percentage of the runs in which each LA, or combination of LA, have chosen the correct movement combination after learning have finished. Note that the percentage is not the average amount of units in each simulation that learned to perform the correct combined move, but the average amount in which all three units chose the correct combined move.

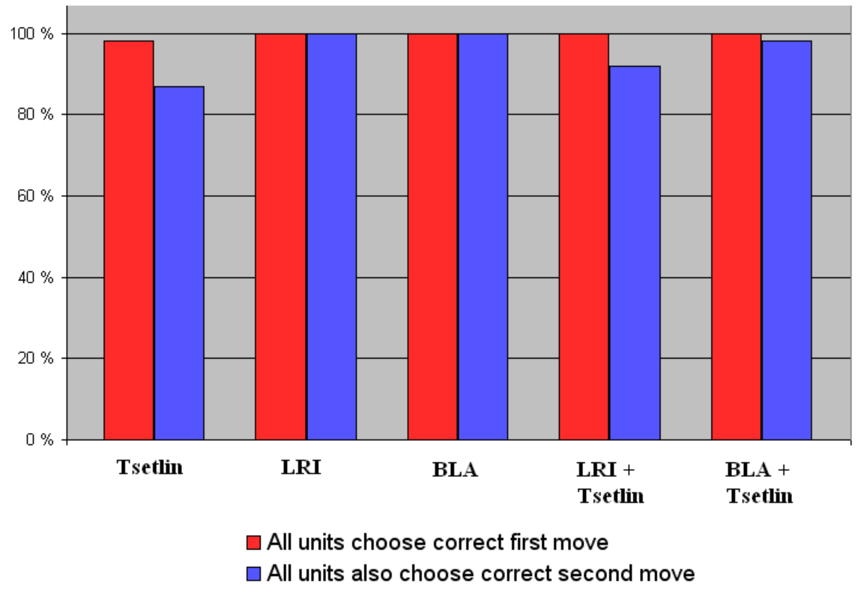

Figure 21: Results - Scenario 2 - Graphic representation

We see clearly that both the LRI-automata and the BLA have no problems with converging on the correct set of moves for all their units. The Tsetlin automata, which cannot converge at all, still picks the correct combination of moves in the majority of runs. Even with the Tsetlin giving the LRI automata and the BLA slightly variable feedback from the second and third turn, both the LRI automata and the BLA converges on the correct choice every time.

Analysis

The optimal performance for this scenario, as described in chapter 4.4, is moving all three infantry units to the high-value area. The LRI automata and BLA, when used for all three turns required to search this scenario thoroughly, have 100% success rate in finding this optimal set of moves, given time to converge. And it is the time issue that concerns us here: Even the Tsetlin, which does not converge at all, is able to come to the same conclusion in 87% of runs, and accomplishes this much faster. When using the Tsetlin automata for the second and third turn, and either the LRI or BLA for the first, performance improves significantly, at a slight loss in overall precision.

The interesting point of these three-turn simulations is that they are just that - simulations. When playing an actual game, determining the first move would be enough, as the simulation would be done again for the next move - and for this purpose a combination of automata would seem to be a better solution: Much faster, and minimal or no loss of precision.

5.3 Scenario 3

This scenario comes with a much larger amount of units on the map than the previous, and thus have a much larger amount of possible outcomes. However, the amount of units, and thus LA that is required to cooperate, is not that much larger than in scenario 2.

As in scenario 2, there are again only one strictly correct set of moves. If all units does not stay in their original territory during the first round, the LA will most likely end up with a loss in points. During the second turn, the best move by far is for all units to attack together, as this gives the best gain in points. However, a single unit, whether it be an infantry or an artillery unit, who acted differently will not directly lead to a point loss - merely a lesser gain. For this reason, we have included a third field in our graphic, showing at what percentage the automata performs an “adequate” move - at most one unit deviates from the correct choice.

Seeing as all combat should be resolved by the second turn, we are again only simulating these two turns ahead in this scenario.

As in scenario 2, we will test not only options where one type of LA controls both turns, but also combinations of the LRI or BLA and the Tsetlin automata. The amount of simulation runs for each turn required for the various types of LA and their combinations, along with the average time consumption per run, can be seen in table 6.

Type of Automata	Turn 1	Turn 2	Avg. time
Tsetlin	500	300	5 sec.
LRI	700	400	8 sec.
BLA	800	400	17 sec.
LRI + Tsetlin	700	300	6 sec.
BLA + Tsetlin	800	300	9 sec.

Table 6: Scenario 3 - Required amount of simulation runs and time consumption

Again, there is time to be saved by using a combination of automata, although the difference is less apparent than in scenario 2. Note that the average time consumption for each run is many times lower than in scenario 2, with the only major change being not simulating a third round.

Figure 22 shows the percentage of the resulting moves after learning have finished in which each LA have chosen the correct movement combination. Again, note that the percentage is not the average amount of units in each simulation that learned to perform the correct move, it is the percentage of runs in which all units concluded on choosing the correct combined move. In addition, we are showing the percentage of which at most one unit have chosen another move. As explained above, this move would most likely lead to a reward, although a reward smaller than the optimal.

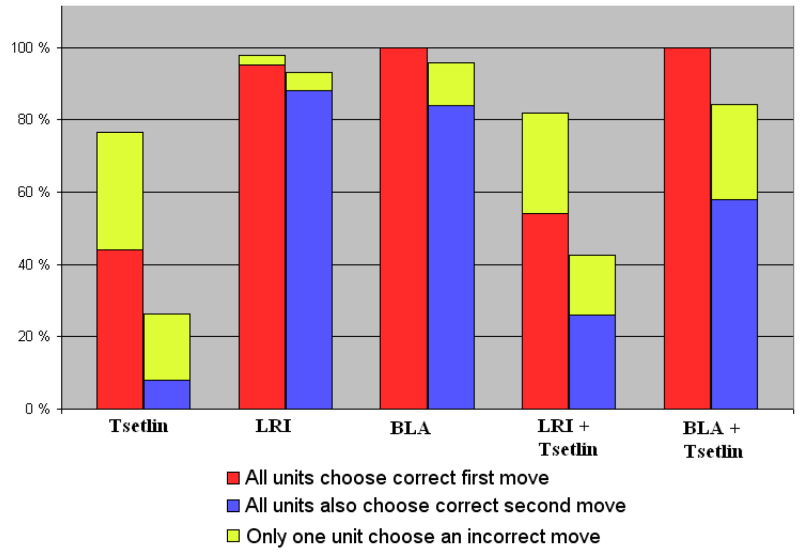

Figure 22: Results - Scenario 3 - Graphic representation

Unlike scenario 2, neither the LRI automata nor the BLA can show a perfect performance in this scenario. The BLA performs correctly during the first turn every time, but is not quite as certain on the second turn. The LRI automata has a tighter spread - while they would fail the first turn now and then, they usually pick the correct set of moves during the second turn if the first move set was done correctly.

The Tsetlin-automata, however, does no longer perform as well. Only occasionally managing to get all units to work together properly, and only very rarely making the correct decision for both moves. When coupled with the LRI, it even reduce the performance of the LRI automata, most likely by giving it bad feedback due to its own lacking performance. The BLA seems not as bothered by the bad feedback of the Tsetlin, managing a perfect score, and even improving the average performance of the Tsetlin over its standalone performance.

As we expected in chapter 4.5.4, some of the incorrect moves involved only a single unit acting on its own, which is likely to have given the automata some false positives. In the cases where this happens during the first turn, the automata usually have a hard time performing a proper second move combination. When the first turn is played correctly, however, a single error may simply be attributed to the team's automata getting fooled by the false positives, being rewarded for performing above average even when there is a potentially higher reward to be gained.

Analysis

As expected, giving the team of automata a tougher challenge in terms of more units and a more complex situation gives a more varied result. Both the LRI automata and the BLA still performs very well, and decides upon their actions in a reasonable time frame. However, this time frame is smaller only because there was no real need to simulate beyond the second turn, and so we did not. Adding a third turn to the simulation - for the scenario at hand completely pointless - we are faced with the same performance problems as seen in scenario 2. During preliminary testing whether two or three rounds would be needed in order to get a precise result, adding a third round increased the run time for pure LRI and BLA by a hundred times or more, and for the Tsetlin twenty times or more, without any apparent increase in precision.

Finally - for the Tsetlin, we seem to have reached the limit on how many units it is able to coordinate comfortably, only finding the correct combination of moves a mere 8% of the time when playing both turns by itself. We could speculate that this is because the Tsetlin is particularly vulnerable to false positives. A less than optimal reward might still cause the Tsetlin to become stuck in a particular choice, as it has no memory of what choices gave it the highest chance of rewards. However, this should have caused the Tsetlin to have ended up with a choice where all units but one pick the wrong move, which we are not seeing.

6 Summary

6.1 Conclusion

In this thesis we presented Learning Automata (LA) of various types for use as players in the game Axis & Allies Revised (A&AR). In particular, we have studied how each of them respond to predefined scenarios within the game, in order to get a measurement of what performance we might expect should they be implemented as a full-scale artificial player.

We have compared the theoretical benefits of LA and Reinforcement Learning to traditional approaches for Artificial Intelligence (AI), and explained the reasoning behind these benefits.

To test the performance of a variety of distinct types of LA, we have implemented the Tsetlin Automata, the Linear Reward Inaction Automata, and the Bayesian Learning Automata. We then implemented a simulation engine for use in teaching our LA how to play a subset of A&AR, and created scenarios with which to challenge our LA to determine their accuracy and performance. Each type of LA have been tested against themselves and against scripted opponent in these scenarios, in order to determine if each of them can achieve the desired combination of reactivity and proactivity.

In order to achieve reactivity for the multiple individual units which populate the world map in A&AR, we chose to implement the Decentralized Learning Automata approach. By giving each individual unit their own, independent Learning Automata, we aimed to let our artificial players teach themselves what combination of moves would be the most beneficial in any given situation.

In order to achieve proactivity for these Decentralized Learning Automata, we chose to implement a Multi-Tier Learning Automata approach. We connected a separate, independent LA to each action that could be chosen by the original LA, and if necessary for each action that could be chosen by each of *these* LA. Instead of recieving feedback from its immediate environment, each LA selecting actions for the units during the earlier turns would receive feedback based on the performance of other LA during the later turns. Through this, we aimed to let our teams of LA teach themselves not only what combination of moves would be beneficial immediately, but also what combination of moves would be most beneficial in the near future.

For each scenario, we have given a complete statistical breakdown of what feedback could be expected for our LA when performing any of a range of possible movement combinations. This was done to give an overview of the desired outcomes of learning, and in order to have the required background information in order to measure the actual performance of each type of Automata.

The Tsetlin Automata, a very simple form of LA, with no memory of past rewards or penalties, simply a choice and how many times selecting this choice have been reinforced with a reward, have performed surprisingly well in our testing. In its best scenario, it chose the correct movement combinations over two turns in a respectable 87% of the total runs, ensuring cooperation between three individual units. In scenarios where the optimal move was less apparent, however, the Tsetlin Automata had a harder time finding it. The main attraction of the Tsetlin Automata, however, is not its accuracy. Whereas more complex types of LA may yield better results, the Tsetlin regularly performs “well enough”, and does so with much greater speed.

The Linear Reward Inaction Automata, another theoretically simple form of LA, responds only to rewards, and thus can only change its preferred choice when a better choice have been found. This approach works very well for our scenarios where the one combination of moves is clearly better than the others, as might be expected. The performance of the automata is worse, however, when the difference between a good and a bad choice is small. Given enough time and one combination of moves clearly better than the others, and the LRI will converge on the correct one every time. If several other choices are also beneficial, the LRI could conclude that a lesser move is the optimal, depending on the random outcomes when the LRI explores each of them.

The Bayesian Learning Automata, which learns by shaping the curves of Normal (Gaussian) and Gamma distributions through feedback receieved from the system, is a complex approach to learning, used as a contrast to our other, simpler types of LA. It performs near perfectly in all our scenarios, only slightly less certain when required to coordinate five individual units in a scenario where there are multiple rewarding actions with different average rewards. Both when it comes to reactivity and proactivity, the BLA performs admirably, and it has seemingly little trouble coordinating between several Automata at once - nor with recieving feedback from sub-automata.

For accuracy and overall performance in our small-scale tests, the BLA performs well. However, the design of the BLA involves incrementing parameters during each iteration, which are then used in computations of *n!* complexity. As the complexity of a particular scenario goes up, we see that more iterations would be needed for any LA to converge. At some point, the BLA becomes computationally impractical, and even with its high precision, it seems unlikely that a practical implementation for larger scale experiments can be achieved.

In contrast to this, the LRI updating and selection methods are computationally indifferent to how many iterations the simulation has been going through, and as such would likely outperform the BLA in regard to speed at an increasing rate as the complexity of the scenarios go up.

Overall, our testing shows that the various LA can achieve reasonable precision in smaller scenarios, which improves when the scenarios have clear outcomes. Even for smaller scenarios, however, the time spent on simulating in order to allow our LA to learn, even when an average computer CPU is dedicated to this process alone, leads us to the conclusion that for practical application in actual games, this approach still needs work. There would be much to gain through code optimization, as this has not been our main focus in this project, and simpler games than this would most likely also allow for faster learning, but if we were to aim at creating a fully capable artificial player using this approach, more computational power would be among the first requirements.

Fittingly, this conclusion relates well to other approaches within this field - the best artificial players for games such as chess or go employ computational powers far beyond the capabilities of the average home computer in order to be competitive in real-time. If our approach were to be expanded to controlling the entire game board of A&AR, while keeping precision at a level comparable to humans, there is no reason why we should expect the average home computer to be able to conclude on its decision with the speed expected from a human opponent.

We believe that our approach, while currently impractical for implementations designed for home use, and as such for any commercial games, gives a good baseline for further development on learning-based artificial players for turn-based games. We also believe that this approach can be utilized in artificial alayers in other types of games where the state of the game at any given time can be quantified in order to give feedback to a LA.

6.2 Further work

Even if the approach described in this thesis has shown its qualities in our subset of the game A&AR, the work required to make a fully useable and competitive artificial player using it is a massive undertaking. While we believe that the core concepts of this approach - the Decentralized and Multi-Tiered Learning Automata - are sound, both these and other facets of our implementation present ample opportunities for further work.

Testing a wider range of Learning Automata types

While the three selected types of LA included in our implementation span a wide amount of different classes of LA, we cannot realistically assume that any of them are neccesarily optimal for this method. As such, testing the approach using a wide variety of different types of LA would be an area open to further study.

Expanding the size and complexity of testing scenarios

In this thesis, we have used only a small subset of the complete unit gallery within A&AR and our scenarios have had relatively few choices for each unit to decide between. Expanding to more units and larger scenarios would likely improve the general understanding on the functionality of the approach, and give further insight into its limitations.

Expanding the search depth of the Multi-Tiered Learning Automata

As we designed our scenarios in a way so as to be able to get correct results with a limited search depth, we did not need to search further than three turns ahead. For a larger scale scenario, this may very well be required in order for an artificial player to respond to a situation correctly.

Improving distinction between states giving similar feedback

As we saw in particular during the testing of our first scenario, our approach is currently at its weakest when there are several outcomes which yield similar results. Methods to distinguish more closely between these outcomes would most likely improve both the precision and learning speed of the LA in such cases.

References

[1] Kok, E., *Adaptive reinforcement learning agents in RTS games.* Utrecht, The Netherlands, 2008

[2] *The History of Civilization.* Available online at http://www.gamasutra.com/view/feature/1523/the_history_of_civilization.php?print=1

[3] *40th Anniversary Collector's edition of the Risk rules.* Available online at http://www.hasbro.com/common/instruct/RiskCollector%27s40thAnniversaryEdition.PDF

[4] *Axis & Allies Official Home Page.* Available online at http://www.wizards.com/default.asp?x=ah/aa/welcome

[5] TripleA, an open-source Axis & Allies implementation. Available online at http://triplea.sourceforge.net/mywiki

[6] *Axis & Allies Reference Charts.* Available online at http://calmdragon.net/refcharts4.html

[7] *The Axis & Allies Revised Manual.* Available online at http://www.wizards.com/avalonhill/rules/axis2004.pdf

[8] Shannon, C.E., *Programming a Computer for Playing Chess*, Philosophical Magazine, Ser.7, Vol. 41, No. 314, 1950

[9] Marsland, T.A., *Computer Chess Methods*, Encyclopedia of Artificial Intelligence, 1987

[10] *Official American Go Association Rules of Go.* Available online at http://www.usgo.org/resources/downloads/completerules.pdf

[11] Brügmann, B., *Monte Carlo Go*, München, Germany, 1993

[12] Gelly, S., Wang, Y., Munos, R & Teytand, O., *Modification of UCT with Patterns in Monte-Carlo Go.* Technical Report 6062, INRIA, France, 2006

[13] *Rules of diplomacy, 4th edition.* Available online at http://www.wizards.com/avalonhill/rules/diplomacy_rulebook.pdf

[14] Shapiro, A., Fuchs, G. Levinson, R., *Learning a Strategy Game using Pattern-Weights and Self-Play*, Santa Cruz, USA, 2002

[15] Tsetlin, M.L., *Automaton Theory and Modeling of Biological Systems.* New York: Academic Press, 1973

[16] McCallum, A., Nigam, K., Rennie, J. & Kristie, S., *Building Domain-Specic Search Engines with Machine Learning Techniques,* Proc. AAAI-99 Spring Symposium on Intelligent Agents in Cyberspace, 1999.

[17] Bevilacqua-Linn, M., *Machine Learning for Naive Bayesian Spam Filter Tokenization,* 2003

[18] Stensby, A.M., *Stochastic Learning-Based Estimation Methods for Pattern-Recognition and Its Applications to Topic Detection and Tracking.* Grimstad, Norway, May 2008

[19] Kaelbing, L.P., Moore, A.W. & Littman, M.L., *Reinforcement Learning, A Survey.* Journal of Artificial Intelligence Research 4, 1996

[20] Oommen, B.J. & Ma, Y.O., *Deterministic Learning Automata Solutions to the Equipartitioning Problem,* IEEE Transactions on Computers, vol. 37, no. 1, 1988

[21] Granmo, O.C., *A Bayesian Learning Automata for Solving Two-Armed Bernoulli Bandit Problems,* Grimstad, Norway, 2008

[22] Fink, D., *A Compendium of Conjugate Priors,* 1995

[23] Weusstein, E. W., *Normal Distribution.* Available online at http://mathworld.wolfram.com/NormalDistribution.html

[24] Weisstein, E. W., *Gamma Distribution.* Available online at http://mathworld.wolfram.com/GammaDistribution.html

[25] Rempel, D., *Axis & Allies Combat Simulator.* Available online at http://frood.net/aacalc/

[26] *Gnu Scientific Library*, Available online at http://www.gnu.org/software/gsl/