Internal Report 2013–01 February 2013

Universiteit Leiden
Opleiding Informatica

Intelligent agents in a large state space

A case study on Magic: The Gathering

Erik Zandvliet

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Intelligent agents in a large state space

A case study on Magic: The Gathering

Erik Zandvliet

February 21, 2013

Contents

Abstract

Many current state of the art algorithms for intelligent agents have been developed with small state spaces in mind. Their performance is said to decrease when state spaces grow larger, but no comparisons have been made to simple strategies. This report describes such a comparison based on a subset of the card game Magic: The Gathering, between a high-level Markov Decision Process-based algorithm and basic algorithms such as Monte-Carlo and beam search.

Chapter 1

Introduction

Many state of the art self-learning intelligent systems have been invented for use on problems with a small statespace. Some of them however, have a broad set-up and are expected to also work on large problems. The main topic of this thesis is to compare such an algorithm — the Q_{MDP} algorithm — with simple problem-specific and random approaches on a large problem set, the trading card game Magic: The Gathering.

The Q_{MDP} algorithm is based on Markov Decision Process theory [8], and — guided by a system of rewards – builds up a model of the state space, recording all interesting states and using these to calculate the best action to be performed. This method is very memory intensive, and in the end there is the possiblity that the model will cover the entire state space to be analyzed, making it into a very inefficient method.

The algorithm tries to prevent this however, by using rewards to determine which actions are good to perform and which actions are not. This way, the algorithm is guided into an area of the problem where it is more likely to find good actions and evade the areas where negative actions are more likely, therefore automatically pruning large parts of the space.

The test problem, Magic: The Gathering [6], is a trading card game developed around 1993, where players construct decks of 60 or more cards and battle against each other. Since the introduction of the game, new sets of cards are released on a regular basis, expanding the total set of available unique cards. Even though the basic rules of the game are fairly simple and straightforward, there exists an extensive rulebook [7] describing — amongst others — how each and every card can influence the rules of the game to their player's benefit. Nearly all basic rules can be changed by text on the cards, and the rules imposed by the cards have a higher priority than the basic rules, making the game complex to understand. Recently, the game has even been proven to be turing complete [3].

The cards being able to change the rules, in combination with various combinations of cards in a single deck, makes the game very unpredictable. This unpredictable behaviour of a deck during a single game is even more extended

by the amount of opportunities and ways to activate certain abilities on the cards.

Even though there are other projects which provide an artificial player for the game as an opponent, no research has been done on how to make the agents as strong as possible. In the end, these frameworks are built as video games which can be won by the player. The same thing holds for MDP-based approaches, which have been shown to work well on small problems (in the order of 10^3), but no results have been made available for large problems. This paper shows the results of an example in the order of 10^18.

This report is written as final product of the Master thesis at Leiden University, supervised by prof. dr. P. Lucas and dr. W.A. Kosters.

In Chapter 2 the game of Magic is explained in a way that people unfamiliar with the game will still be able to understand the remainder of the thesis. In Chapter 3 the notion of Partially Observable Markov Decision Process is defined, and a few algorithms that are built upon this notion, including the Q_{MDP} algorithm are explained. Chapter 4 describes the game model that we have constructed to allow for parsing by the algorithms. Chapter 5 describes our tests, of which the results can be found in Chapter 6. Chapter 7 finally concludes our results, and provides an outlook for future research.

Chapter 2

Magic: The Gathering

Even though it is fairly impossible to describe the complete game of Magic with all its specific rules and options, especially within the boundaries of this report, it will be useful for the reader to have a general understanding of the different aspects that build the game. For more detailed explanations, we refer to [6].

Players of the game are called "Planeswalkers", and can be seen as some kind of magicians. The game is played with a set of custom cards, from which players can build their decks. A deck consists of — depending on the type of game — at least 40 or 60 cards. This deck is shuffled at the beginning of a game, and from that point onward called the library. This library can be seen as an arsenal of spells that a player can cast to defend his life total and to attack the opponents. Each player starts with a life total of 20, and when it drops to 0 or lower, the player loses the game.

With these spells, the player can summon creatures onto a battlefield, to aid in bringing the opponent down, or to help in surviving; they can trigger temporary effects — for example damaging the opponent directly or increasing the strength of creatures — or they can provide constant effects.

However, players will not be able to play spells at will. There are strongly regulated times at which spells can be cast, and on top of that, to actually cast a spell, the player will need to have access to enough "resources" to cast this specific spell. This resource is called *Mana* in the game, there are five types of Mana, and each of these colors can be associated with a different style of cards. For example, red Mana can be associated with a somewhat more aggressive style of playing — Goblins and Dragons are examples of creatures with this color — where green Mana can be associated with natural spells — for example Elves and Snakes are creatures that can be cast through the use of green Mana.

Now we know that we need resources, we have to find a way to gather these. The prime source for obtaining Mana is through so-called Land cards. There are five basic types of lands: Forest, Swamp, Mountain, Island and Plains — corresponding to the five mana types — but during the development of the game more types of lands have been introduced, with different capabilities.

Land cards can be tapped — flipped from portrait to landscape orientation,

an action that is used throughout the entire game to indicate the "usage" of cards or their abilities — to generate Mana. The Mana is then stored in a fictional Manapool, which is drained through the use of other spells.

When a player casts a spell, this spell is first put on a stack to allow for other players to respond on the action. When nobody responds, the spell will resolve with two possible outcomes, based on the type of the spell:

- The spell is a *permanent* (e.g., has the type Land, Creature, Enchantment or Artifact): it will enter the battlefield, and effects may trigger or become active.
- If the spell is not a permanent (e.g., has the type Instant or Sorcery): its effects will take place, and the card will be moved to the graveyard.

The graveyard is a place where used spells and dead creatures reside. If a card is located here, it does not necessarily mean that it will not be used again during the game, some spells will "resurrect" dead creatures, or allow for multiple usages.

Section 2.1 will introduce the layout of the cards used to play Magic. After that, we can continue with the structure of a game in Section 2.2. In Section 2.3 we will then introduce the two decks that we will use in the remainder of this report.

2.1 Card Overview

In this section we will give a basic overview of the different elements on a card. As an example card will we use the Elvish Eulogist depicted in Figure 2.1. The different elements on the card — with between brackets the values for the Elvish Eulogist — are:

1. **Name** (Elvish Eulogist) This uniquely describes the card, and any card with a different name, will have a different set of other properties as well.
2. **Manacost** (G) The amount of Mana that needs to be generated in order to cast this spell. This Mana is mostly generated through the use of Land-cards, but can also be obtained through other spells or the activation of abilities.

   There are five types of Mana in a game: 'White', 'Blue', 'Green', 'Black' and 'Red', each having its own type of land cards. On top of that there is the concept of colorless Mana. Colorless Mana is often used in the Manacost of a spell, and in this situation can be filled in with any color Mana. However, when colorless Mana is generated, we can not substitute this for a type of colored Mana.
3. **Type** (Creature — Elf Shaman) The Type of the card, most of the times in the form 'Main Type — Sub Type', is one of the extensible features in the world of Magic. There are only a few Main Types, but the Sub

Figure 2.1: An example of a card: Elvish Eulogist

Type is a very extensible feature, and there are spells that can target only permanents with a specific type, or trigger on specific types being cast. A card is a member of every Type that is contained within its Main- and Subtypes.

An example of a restriction that is enforced through the Type of a card, is the restriction that a player can play only one Land card per turn. This restriction is enforced on the Main Type Land, and is therefore restricting both the Basic Land and the Land main types.

A quite different way of using the Type of a card we find for example in the card Elvish Eulogist, depicted above. This Creature card has an ability — described in the next point — which allows the player to gain life equal to the amount of Elf cards in his/her graveyard.

4. **Abilities** (Sacrifice... graveyard) The abilities that this card has. These can be either full sentences, fully explaining the ability, or they can be a single keyword. These keyword abilities are a shorthand for common abilities, such as Flying (the creature can not be blocked by non-flying creatures) or Haste (the creature can attack immediately).

   In the game of magic, a discrimination is made between three different ability forms:

   Activated Abilities These are abilities of the form '*Cost* : *Effect*', and denote abilities that need to be activated before their effects take place. Costs can range from simply having to tap a card to sacrificing creatures and paying life. The effects of these abilities in general be-

come more powerful when the costs to activate them become higher, and can range from generating mana to adding tokens to your deck if the bottom card of your library happened to be a ZOMBIE type. These examples do not limit the potential of abilities, as in the game of Magic cards make the rules.

The most common activated abilities used, are those of LAND cards. Even though not depicted on their physical cards, cards of the type 'BASIC LAND' always have an activated ability of the form "Tap : Add <Color> mana to your manapool", in which <Color> is replaced by the specific mana color of the land.

Triggered Abilities Triggered abilities are — as the name already indicates — abilities that trigger when specific events, such as the end of a turn or a new creature entering the battlefield, take place. The types of effects vary in the same range as the activated abilities, but in most cases are more powerful.

Constant Abilities Constant abilities are abilities that are active as long as the card remains on the battlefield. Common effects of this type of abilities are for example (a specific type of) creatures getting $+x/+y$ on power/toughness respectively, or creatures gaining abilities.

5. **Power** (1) The power of this card. This is the amount of damage done to a player and/or creature.

6. **Toughness** (1) The maximal amount of damage a creature can take. If this drops to 0 or lower, the creature will die. Damage taken by a creature will be subtracted from their toughness, until the damage is removed at the end of a turn.

2.2 Game Structure

Magic: The Gathering is a turn-based game; each player has specific moments on which he or she is allowed to play cards and make decisions.

During a single turn, there is exactly one *Active Player*, and one or more "opponents". Opponents have the ability to respond to the Active Player's actions and on the transition to specific turn phases. The exact structure of a turn is depicted in Figure 2.2.

Cards — when played — are put on top of a stack. This stack is then used when other cards are played to handle resolving the effects of cards in the right order. This is for example important when one tries to kill an opponent's creature with a spell, and the opponent wants to reinforce his creature in such a way that it will not die. The stack has to be completely empty before the game can switch to a next step or phase of the turn.

If none of the players decides to respond to a card being put on the stack, the spell *resolves*, e.g., its effects are processed and executed. For creature-type

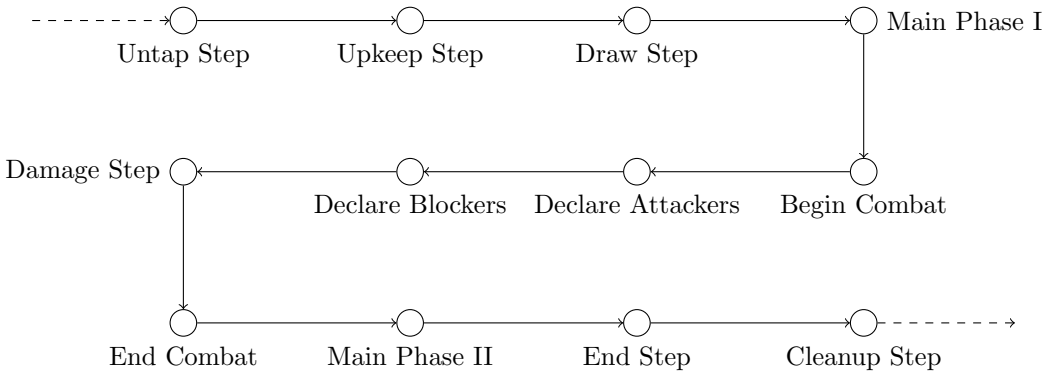

Figure 2.2: The different steps and phases of a turn.

spells this usually means the creature will enter the battlefield, but there might also be some additional effects that are triggered.

Section 2.2.1 through Section 2.2.5 will describe the parts of a turn in more detail. At a lot of these steps, all players will have the opportunity to respond to actions by casting instants or activating abilities on cards. This opportunity will be denoted as 'all players can respond'.

2.2.1 Beginning Phase

The beginning phase is where each turn begins. The most basic way to view it, is as an initializing phase, in which everything is set up such that the new active player can play its moves.

Untap Step

In this step, all permanents that the active player controls become untapped and ready for action. If any creatures are effected by Summoning Sickness, this effect will now be removed.

Upkeep Step

In this step, the abilities on cards that trigger 'at the beginning of your upkeep' take place. They are pushed onto the stack, in order for them to resolve. All players can respond.

Draw Step

The active player draws the top card from his or her library. All players can respond.

2.2.2 First Main Phase

The main phase is where most of the action in a turn will take place. A player can cast his or her spells here, including — in general — at most one LAND card per turn. All players can respond to the casting of spells, as well as to the resolving of a spell from the stack. Creatures entering the battlefield in this step will be affected by Summoning Sickness, unless their abilities state otherwise (e.g., the creature has Haste).

2.2.3 Battle Phase

During this phase, creatures can be assigned to attack other players, and the corresponding other players — which we for ease of use call a *defending player* — can determine whether he or she assigns any creatures to block the incoming attack. If no attackers are assigned by the active player, the remainder of the phase is skipped, as there is no damage to be done.

Beginning of Combat

This step is the last option for players to cast instants and abilities that allow or prevent specific creatures from attacking. Apart from this opportunity, nothing takes place during this step.

Declare Attackers

During this step, the active player declares which of its creatures it wants to attack, and who they want to attack with each creature. Creatures that are tapped or affected by Summoning Sickness can not be declared as attackers. Attacking creatures will become tapped during this step. After declaring attackers, all players can respond. This is also the last option for the active player to prevent certain creatures from blocking.

Declare Blockers

Now every defending player can declare its blocking creatures — if any. The defending player can decide which creature(s) block a creature, but if multiple creatures block one attacking creature, the *attacking player* will decide in which order damage will be given, and the amount of damage done. However, to be able to damage a later blocking creature in the ordering, the attacking creature must deal enough damage to kill the predecessors.

No damage is done yet, and all players will have a last opportunity to respond. For example cards like GIANT GROWTH can be played to increase the attack and defense of a specific creature, to make sure it will survive the attacker, or deal more damage.

Damage Step

During this step all damage is dealt at the exact same time, e.g., each creature deals damage equal to its power to its direct opponent. If at this point any creature has a toughness that is 0 or lower, the creature will be moved to the corresponding player's graveyard. When an attacking creature hasn't been blocked, it will deal damage equal to its power to the defending player's life total. Blocked creatures will not deal damage to the defending player, not even if they have been removed from combat anywhere between its declaration and this step.

There are a lot of situations where this is not the exact way damage is dealt, and an example can be found when creatures have the ability 'Trample'. These creatures deal exactly enough damage to the opposing creature to kill them, and if any power is left, it will still damage the defending players' life total. Another example is found in cards with the ability 'First Strike' or 'Double Strike'. In this case, an extra damage step is introduced, in which only the creatures with either of these abilities deal damage, after which the 'normal' damage step is entered. During this normal damage step, creatures with 'First Strike' will not deal any damage anymore, but creatures with 'Double Strike' will get another round to deal damage.

After dealing damage, all players can respond.

End of Combat

During this step, all players can respond on the end of combat.

2.2.4 Second Main Phase

During this phase, the active player can cast any spells, just as in the First Main Phase, only with the restriction that if a LAND card has been played during the First Main Phase, the player cannot play one during the Second Main Phase. Again, all players can respond.

2.2.5 Ending Phase

During this phase, the turn of the active player is finished. All players will get the opportunity to play the last cards before the turn ends, and the next player will become active.

Beginning of End Step

All abilities that trigger 'at the beginning of your end step', will be put on the stack, and will be resolved. All players can respond on each trigger and resolve action.

Cleanup Step

During this step, the turn will end. If the active player has more than 7 cards in hand, he or she has to choose and discard cards until this constraint is satisfied. After this, all damage will be removed from creatures on the battlefield, together with all effects that last 'this turn' or 'until end of turn'. Players can only respond if an ability has been triggered in this step.

2.3 Elves vs. Goblins

Our test case will revolve around the Duel-Decks pack 'Elves vs. Goblins'. Duel-Decks have been introduced to allow for a relatively simple style of playing as to attract new players, but at the same time consist of two relatively balanced decks, so even advanced players can play a nicely balanced game.

Elves vs. Goblins is the first Duel-Decks created, and has two single-colored decks — Green and Red respectively. Section 2.3.1 will describe the Elves deck in a little more detail, and Section 2.3.2 will describe the Goblins deck. An extensive deck list can be found in Appendix B.

2.3.1 Elves

The Elves deck is a single-colored green deck, and is built up from the cards shown in Table B.1.

Most of the cards in this deck are creatures with the type 'Elf'. These are in general relatively weak creatures with a low power and toughness combination. Most elf cards, however, have added abilities on them.

For example Llanowar Elves is a creature with a power and toughness of both 1, but it has the added ability that one can tap it to generate a unit of green mana, which can then be used to play other spells. Another example in this is the Wellwisher: also a 1/1 creature, but it can be tapped to increase your life total.

The extra abilities of the Elves deck allow for a quick buildup of a defensive layer — which if needed can be strengthened by cards such as Giant Growth which increases both the power and toughness of a single creature with 3 — and for the increase of your life total. The next step is then to wait for the opportunity to obtain stronger creatures — such as the 7/7 elemental creatures generated by Voice of the Woods — that can overwhelm the opponent.

2.3.2 Goblins

The Elves deck is a single-colored red deck, and is built up from the cards shown in Table B.2.

The creatures in this deck mostly have the type 'Goblin', and are annoying creatures for your opponent. The weakness of the individuals is made up for by the amount of goblins that can easily be put onto the battlefield, for they are low-cost cards.

Where the strategy for the Elves deck is mostly awaiting, the Goblins deck dives right in and tries to destroy the opponents life total as quick as possible, before it gets the chance to defend itself. Adding to this strategy are cards such as Tarfire, that damage the opponents creatures, and Spitting Earth that can later on in play be a quick stab in your opponents life total.

The real finishers in this deck are cards such as Skirk Fire Marshall and Flamewave Invoker, dealing heavy damage to your opponent without giving him or her the opportunity to evade the damage.

Chapter 3

Partially Observable Markov Decision Processes

As the main goal of our research is to see whether Partially Observable Markov Decision Processes (POMDPs) can aid in learning to deal with complex and large state spaces, we will first provide informal and formal definitions of the concept. As both the theory and solving of POMDPs rely heavily on the concept of Markov Decision Processes (MDPs), we will give definitions for these as well.

In Section 3.1 we will give a basic overview of Markov Decision Processes and their properties. This gives us a basic understanding to continue in Section 3.2 with the theory on POMDPs. In Section 3.3 we will then give a basic overview on how one can generally solve MDPs and POMDPs through the Forward Search Value Iteration algorithm.

3.1 Markov Decision Processes

A Markov Decision Process (MDP) [8] is a model of an environment that makes use of states and actions. Given a state, the agent in an MDP has several actions that it can execute, and executing an action in a state, will provide the agent with a new observable state. The model intuitively resembles a regular Finite State Machine [4], however, in a MDP it is not needed for the environment to be deterministic.

Suppose we let our agent be a simple robot, with the actions "Up", "Down", "Left" and "Right" for movement around a scenery. Executing "Up" in an arbitrary state will move the robot up one unit in the general case, but when for example the floor is slippery, there is a possibility that the wheels will not have enough grip, and the agent might only move up a quarter of a unit. Instead of encoding this information in the states of the model, MDPs encode a probability in the resulting states for actions.

One of the main motivations of using a simple model such as a Finite State Machine as the basis of the model, is the Markov Property. An environment sat-

isfies the Markov Property, if for each state, the best action to perform depends purely on that state. Using this property, we do not need to keep track of the history of a state and use complex mechanisms to assign a value to this history. All we need to do, is analyze the current state and see which action gives us the best results.

Formally, a MDP is defined by the following tuple:

$$\langle S, A, tr, R \rangle$$

S: States. A state $s \in S$ is a representation of all the relevant information in the world.

A: Actions. All actions that an agent can take. The agent modifies the world by executing these actions.

tr: Transitions. A table that holds for each tuple (s, a, s') the probability that an agent executing an action a in state s will arrive at state s'.

R: Rewards. Rewards direct the agent towards desirable states of the world and keep it away from places it should not visit. Rewards are given to specific states in the world, and are observed upon reaching this state.

Using the rewards we can compute the expected reward for this state by calculating the weighted sum of all rewards. Solving algorithms usually utilize the Average Discounted Reward (ADR) technique to compute the expected reward of a state, making short term rewards more attractive than long term rewards. The ADR can be calculated as follows:

$$ADR = \sum_{t=0}^{\infty} \gamma^t r_t \tag{3.1}$$

with $0 \leq \gamma \leq 1$ a discount factor influencing the effect of long-term rewards, and r_t the reward given at time step t. Note that the reward can also be negative, e.g., a penalty for entering a certain state.

Calculating the ADR needs a policy for determining the next action to execute, which is usually taken to be the action that brings us the highest ADR of the next state.

3.2 Handling Non-observable Information

MDPs assume that the agent has a complete observation e.g., can observe everything that it needs to know to make an informed decision. If we take the robot from the previous example, it requires that the agent knows exactly where the obstacles are. However, for the agent to know this, it needs to ask its sensors whether there is an obstacle at a certain position, and these sensors need not be perfect, e.g., give false positives and false negatives. This results in an uncertainty about the state that agent is in, and this is where Partially Observable Markov Decision Processes (POMDPs) are designed for.

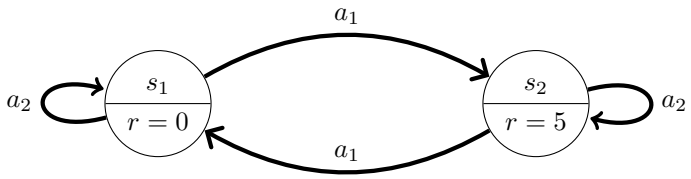

Figure 3.1: A simple environment

In a POMDP, the agent does not know exactly in which state it is, but instead keeps track of the probabilities that it is in a certain state. This probability distribution is known as the *belief state*. Using this belief state, the agent can still determine an optimal strategy, by calculating for each possible state the best action and corresponding ADR, and weighting these, resulting in a piecewise linear function.

Take for example an environment with two states, as depicted in Figure 3.1. The rewards obtained when taking an action are the rewards associated with the destination state. Using these rewards, we can immediately see that in state s_1, action a_1, and in state s_2, action a_2 is the optimal one. The policy in this case should give action a_1 as the most probable action if our probability for being in state s_1 is the highest, but when the probabilities change gradually towards s_2, taking action a_2 will become more probable to give a higher reward.

A POMDP is formally defined as a tuple:

$$\langle S, A, tr, R, \Omega, O, b_0 \rangle$$

S, A, tr, R: The same as the definitions fory u be? a regular MDP, defining the notion of *underlying MDP*.

Ω: Observations. The set of all possible sensor outputs.

O: $O(a, s, o)$. The probability that an agent will observe $o \in \Omega$ after executing a, reaching state s.

b_0. The initial belief state.

Starting in our initial belief state b_0, we need to be able to update our belief according to the action that we have executed, and the observation received. This is done by a τ-function, which assigns to each state s' a new probability, and returns this new distribution b':

$$b' = \tau(b, a, o) \tag{3.2}$$

and is essentially for each $s' \in S$:

$$b'(s') = \frac{O(a, s', o) \sum_s b(s) tr(s, a, s')}{pr(o|b, a)} \tag{3.3}$$

Where we take the probability that this observation corresponds to the new state and the performed action, and multiply it with the probability that we reach the new state. This entity we then divide by the probability that the observation is seen when performing the action in the current belief state:

$$pr(o|b,a) = \sum_{s} b(s) \sum_{s'} tr(s,a,s')O(a,s',o) \tag{3.4}$$

3.3 Forward Search Value Iteration

There exist multiple ways of obtaining an optimal solution for both MDPs and POMDPs, and the results obtained amongst the different algorithms differ. We have chosen for our approach the Forward Search Value Iteration (FSVI) algorithm [9], that makes use of the underlying MDP and its solution as an estimate for training its own solution. Therefore, in Section 3.3.1 we will first explain how to obtain an estimated solution for the underlying MDP through the use of the $\mathrm{Q_{MDP}}$ algorithm [5], before moving on to showing how to solve the POMDP itself through the use of backup functions in Section 3.3.2 and the FSVI algorithm itself in Section 3.3.3.

3.3.1 $\mathbf{Q_{MDP}}$

Because solving a POMDP can be very memory-intensive, and suffers from a state space explosion, the first algorithms developed to deal with them are mere approximation algorithms that hope to find the optimal policy. One of these algorithms is the $\mathrm{Q_{MDP}}$ algorithm [5] utilized by FSVI to obtain an initial approximation.

The $\mathrm{Q_{MDP}}$ algorithm is based on the concept of reinforcement learning [10], and calculates a policy $\hat{\mathrm{Q}}$ that approximates the optimal policy Q^* through the use of the algorithm shown in Algorithm 1.

The values that are now stored in Q, can be used to approximate the value of executing an action a in an arbitrary belief state b in the POMDP:

$$Q_{\mathrm{MDP}}(b,a) = \sum_{s} b(s)Q(s,a) \tag{3.5}$$

If we would now like to compute a policy π for the POMDP system — purely based on the Q_{MDP} values — we simply take for each state the action that is most rewarding:

$$\pi_{Q_{MDP}}(b) = \arg\max_{a \in A} Q_{\mathrm{MDP}}(b,a) \tag{3.6}$$

Using this policy is one of the strategies that we will apply to our test case.

Algorithm 1 Value Iteration

$V' = \emptyset$
Initialize $V(s) = \max_{a \in A} R(s, a)$
while $V \neq V'$ **do**
  $V' \leftarrow V$
  for all $s \in S$ **do**
    for all $a \in A$ **do**
      $Q(s, a) \leftarrow R(s, a) + \gamma \sum_{s'} tr(s, a, s')V(s')$
    end for
    $V(s) \leftarrow \max_{a \in A} Q(s, a)$
  end for
end while
for all $s \in S$ **do**
  $\pi(s) \leftarrow \arg\max_{a \in A} Q(s, a)$
end for

3.3.2 Backup Functions

The FSVI Algorithm belongs to the class of point-based POMDP solvers, and therefore relies heavily on the concept of point-based backups [9]. These backup operations generate α-vectors — storing the approximate reward of an action in a state and store them in a set V.

This set V can in turn be used to define a policy, by selecting the α-vector that generates the highest inner product (and thus the highest expected reward) with regards to the current belief state b:

$$\pi_V(b) = argmax_{a:\alpha_a \in V} \alpha_a \cdot b \tag{3.7}$$

V is initialized with only a single vector: $\vec{0}$, consisting of only 0-entries. In each iteration from then on, we add to V the result of the *backup* function, applied to the current belief state b, increasing the amount of α-vectors.

The *backup*(b) function is mathematically defined by Equation 3.8 through Equation 3.10:

$$backup(b) = \text{argmax}_{g_a^b : a \in A} b \cdot g_a^b \tag{3.8}$$

$$g_a^b = r_a + \gamma \sum_o (\text{argmax}_{g_{a,o}^\alpha : \alpha \in V} (b \cdot g_{a,o}^\alpha)) \tag{3.9}$$

$$g_{a,o}^\alpha(s) = \sum_{s'} O(a, s', o) tr(s, a, s') \alpha(s') \tag{3.10}$$

Equation 3.8 denotes that we calculate and obtain the α-vector g_a^b which maximize the inner product with the current belief state, maximized over all possible actions.

Equation 3.9 denotes the construction of the new α-vector 'candidates'. The idea is to take for each possible observation, the already existing α-vector that again maximizes the inner product with the current belief state, and sum these up. Next, we take a discount factor γ over the summation to assure that current benefits will outperform possible benefits in the future, and at the immediate reward r_a to the complete vector.

Note that if an observation o' is not possible to obtain after executing action a', $g^{\alpha}_{a,o}$ will turn out to be equal to $\vec{0}$, and therefore not add to the summation.

Equation 3.10 finally describes, how we obtain a single component of the summation in Equation 3.9. Given a specific action, observation and an α-vector, we only need to calculate how high the probability is that we will get this observation while being in a state s and executing s'. This value is then multiplied with the value of the α vector at position s', to provide the algorithm with a preference for previously obtained results.

We can — in theory — obtain observation o with a certain probability while arriving in any state s', and thus we need to sum up all these probabilities to obtain a good estimate of how well this action and observation fit together. This in turn can help us increase or decrease our belief for certain states.

Note, that if we are unable to reach s' from s by executing action a, state s' will not add to the summation. The same holds if the α-vector is $\vec{0}$ at the position of s'.

3.3.3 The FSVI Algorithm

The FSVI algorithm combines the concepts in Section 3.3.1 and 3.3.2, by using the Q_{MDP} algorithm to guide its exploration and using backup functions to generate α vectors on the go.

Forward Search Value Iteration uses sampling as the main operation to guide the exploration of the Q_{MDP} algorithm, instead of focusing on the full probability distribution and state space. The algorithm is designed to keep performing actions until the state s is a final state, and then performs backup functions in a backwards fashion, to assure proper propagation of reward data.

Because the algorithm samples from probability distributions, it tends to search more often in directions with higher probabilities. As it is most likely that the agent (during actual execution) will end up in states with higher probabilities, it is an intuitive reasoning that the agent would like to have a better predictive capability for these states when comparing with less-probable situations.

The FSVI algorithm uses a helper function *MDPExplore*(), which is described in Algorithm 2. The initialization and termination properties of the algorithm are described through the while loop in Algorithm 3.

Algorithm 2 shows how we get from one belief state to the next, through use of the solution of the Q_{MDP} algorithm. If our current state s is not a final state, we will determine our next action to perform by taking the best action provided by the Q_{MDP} algorithm. Using this action, we can sample our next state s' from the transitions table. The same way we can obtain an instance of

Algorithm 2 MDPExplore(b, s)

if s is not a goal state **then**
  $a^* \leftarrow argmax_a Q(s, a)$
  Sample s' from $tr(s, a^*, *)$
  Sample o from $O(a^*, s', *)$
  MDPExplore$(\tau(b, a^*, o), s')$
end if
$add(V, backup(b))$

our observation. We calculate the next belief state through the τ function as described in Equation 3.2 and Equation 3.3.

When the algorithm finds a final state s, it will stop exploring, performs an update function and adds it to the full set of α-vectors, before updating previous states. This way all approximate values needed to calculate an α-vector are known at the time that we update this vector, instead of updating the vector with an approximation of 0.

Algorithm 3 FSVI

Initialize V
while V has not converged **do**
  Sample s_0 from the b_0 distribution
  MDPExplore(b_0, s_0)
end while

Algorithm 3 serves as a loop that initializes initial values for the *MDPExplore* algorithm. It checks whether V has converged, and samples the initial state s from b_0. Then it calls Algorithm 2 to do the exploration and update the set V.

Chapter 4

The POMDP-Model

Now that we have a general idea of what a POMDP is, we can start constructing a model for our case study. We will start in Section 4.1 with the states of the model, and more specific the reduction of the huge state space of the game. Section 4.2 will be used to analyse the actions and the concept of perceptual aliasing, and Section 4.3 will describe how to calculate the corresponding results of these actions. Section 4.4 contains a discussion on rewards and how to handle invalid actions. Section 4.5 explains how we make use of the observation elements of the model. Section 4.6 will give an explanation on b_0, the intial belief state. Section 4.7 will then conclude with an explanation of how the model will work in the agent.

4.1 States

The entire state space of our case study would be too large to be feasible. There are 60 cards in either one of the decks, and they can reside — for our specific case study — in four different locations: Library, Hand, Battlefield and the Graveyard. This gives us the size of the state space for one deck to be $4^{60} \approx 1.3 \cdot 10^{36}$. If we then take into account that we have two decks, that both have this — already immense — state space, we get an approximation of the full state space size to be $1.8 \cdot 10^{72}$.

Given that a space of this size is already fairly intractable, we also have the problems that a card on the battlefield has a lot of modifiers to take into account, and that a (human) player could also base its actions on the life total of any player. These two problems actually take the size of the statespace up to an even higher number, especially when we consider — for example – the card Wellwisher, which can increase the lifetotal of its controller considerably.

To overcome the problem of this immense statespace, we have to take some simplificitions into account. Section 4.1.1 through Section 4.1.3 describe these simplifications, and the statespace that they eventually give us.

4.1.1 Permanents Only

The first option for reducing the size of the state space, is by taking only the permanents into account, and not regarding the possibility that cards can be on the stack. This reduces the amount of (interesting) cards in the Elves deck to 55 cards, and for the Goblins deck to 56 cards.

These figures come down to having a statespace size of $4^{55} \approx 1.30 \cdot 10^{33}$ and $4^{56} \approx 5.19 \cdot 10^{33}$ for the Elves and Goblins decks, respectively. This gives us a complete size — though still without the life total problem — of approximately $6.74 \cdot 10^{66}$. Though this is still an intractable size, this simplification reduces the statespace considerably.

4.1.2 Creatures Only

An additional reduction can be obtained when we do take even less cards into account, namely only the Creature cards. This reduces the amount of cards under consideration for the Elves deck to 29 and for the Goblins deck to 30, with a statespace size of $4^{29} \approx 2.88 \cdot 10^{17}$ and $4^{30} \approx 1.15 \cdot 10^{18}$, respectively.

This reduces the total size of the state space size to approximately $3.32 \cdot 10^{35}$, even less than using only permanents in our calculations. This however has as a large drawback that we discard a lot of information concerning the state of the game.

4.1.3 Simplified Creatures

As a last simplification, to again reduce the total size of the state space, we have ordered the creatures in each deck on their Power/Toughness combinations. Within these groups, we furtheron do not distinguish between the different creatures, e.g., having a beginning hand with six Forest cards, and one Llanowar Elves, is the same configuration as a beginning hand with six Forest cards and one Wirewood Herald, for both Llanowar Elves and Wirewood Herald have a Power/Toughness combination of $^1/_1$.

Power/Toughness	Elves	Goblins
$^1/_1$	11	10
$^1/_2$	2	0
$^2/_1$	0	6
$^2/_2$	8	11
$^2/_3$	3	0
$^3/_2$	0	1
$^3/_3$	3	1
$^*/_*$	2	1

Table 4.1: The amount of creatures in each category.

Having these categories, we can serialize a state as a series of digits. For this purpose, we enumerate the used locations:

1. Library
2. Hand
3. Battlefield
4. Graveyard

Using this enumeration, we can easily generate strings that represent all possible configurations for a specific category. Note that, however, each of the cards can reside in any of these categories, we do not discriminate single cards in a category. This has as an effect that the string '1112' indicates the same configuration as the string '1211', or the string '2111': it reflects the fact that we have one creature of this category in our hand, and the others reside in the Library. Since we do not discriminate amongst individual creatures within a group, these denote the same situation. For ease of use, the rest of the paper will use the sorted version to denote these situations.

Using this, we can describe the possible contents of a single category as all the possibilities of a partial ordered set of the digits 1 through 4, where the size of the set equals the amount of cards in this category. This reduces the size of the state space to the numbers in Table 4.2.

	Elves	Goblins
$^1/_1$	364	286
$^1/_2$	10	1
$^2/_1$	1	84
$^2/_2$	165	364
$^2/_3$	20	1
$^3/_2$	1	4
$^3/_3$	20	4
$^*/_*$	10	4
Total	$2.40 \cdot 10^9$	$5.60 \cdot 10^8$

Table 4.2: The possible amount of partial orderings for each category, under the given enumeration.

This ordering in categories, and not discriminating the single cards within a category, renders cards with the same Power/Toughness combination equally dangerous. As much as this diverges from the actual situation, where special abilities on the cards do not necessarily make them an equal threat — especially when considering theme-based decks —, it does reduce the total size of our state space to approximately $1.34 \cdot 10^{18}$, which is beginning to sound like a tractable number. However, the reduction in size does come at the cost of losing information.

4.2 Actions

The actions available to the player can easily be determined at every state of the game, and we can easily generate these actions, given the complete statespace of the game. However, since we have simplified our statespace, there is no one-to-one mapping anymore from the current state in our model, to the actions that we are able to perform at that moment in the game.

This brings us to the concept of *perceptual aliasing* [2]: some states are not distinguishable from each other. In our model, this will for example happen when the player in one form of the state can generate enough mana to play a card, while in another instance of the exact same state — since we do not take into account land cards — the player can not.

Therefore, our actions will always have to be calculated from the current state of the game, and it should be used either as a guideline on deciding which α-vectors are appropriate, or to augment the current model state with all the possible actions.

Both the positive and negative aspects of these approaches will be described in Section 4.2.1. Note however, that even though there are a lot of differences between these options, they have a common set of actions and their results.

There are only a few different actions to be described per card, and not each card will have all of these 8 actions available, some are only available at certain locations, and some are only available during certain steps of the game.

The following is a short summary of all possible actions, and their corresponding results with regards to the current state of our model. We will also talk about restrictions regarding when the actions can be performed and when they can't. We will shortly discuss the possible options for each card:

Play(⟨*Card*⟩ **)** This action describes playing a single card. This action can be performed on every card, though there are a few conditions that have to be fulfilled by the current game state to actually enable performing this action.

The first — fairly straightforward — condition is that the player will actually need to have the card to be played in his hand. It has no use playing a card that already resides on the battlefield and cards that are in the graveyard can — in general — not be played anymore. A player is also not allowed to search for a specific card to play in his Library.

A second condition is the fact that the player will need to be able to generate the mana needed to play the card. LAND cards specifically do not have any costs in the form of mana, but for these we need to take into account whether the current player has already played a card with the LAND type during this turn.

As a last condition, there might be some restrictions on playing a card at a certain point of a turn. In general, only cards of type INSTANT can be played outside of a player's Main Phases.

Activate(⟨*Card*⟩ , ⟨*Ability*⟩ **)** This action describes activating an 'Activated Ability' of a card. Not every card has this type of Abilities, and therefore the

action will not be applicable to every card.

For cards that do have abilities that can be activated, we need to check whether the activation costs can be fulfilled. These may consist of for example 'Tapping' a card, or 'Pay x mana.' For all different types and values activation costs, there are conditions that need to be fulfilled in the current state to activate this ability. The actual constraints per ability can be found in Appendix B.

Attack($\langle Card \rangle$ **)** This action is only available for creatures, and during the 'Declare Attackers' step. When this action is performed on a card, it will be put in 'Attack Mode', such that it will be dealing damage during the 'Assign Damage' step.

Block($\langle Card \rangle$, $\langle OpponentCard \rangle$ **)** This action is only available for creatures, and during the 'Assign Blockers' step. A creature x can be assigned to block a creature y if it has been put in 'Attack Mode'.

Discard($\langle Card \rangle$ **)** This action is only available during the end step of the player's turn, and only when the player has more than seven cards in his or her hand. Upon execution the selected card will be moved from the player's hand to the corresponding graveyard.

4.2.1 Handling Perceptual Aliasing

When discarding a lot of information to reduce the size of the state space, we get multiple states looking identical to the agent. Even though this does not have to be a problem in general — the optimal action to perform in each of these states may be equal —, in many situations it limits the expected power of the policy.

This concept is called *perceptual aliasing*, and it will also limit the exploitation of a computed policy in our case study. For example consider our initial state, where all creature cards reside in the Library. If we now draw our initial hand of seven cards, and assume that there will not be any creature in our hand. Even though the actions that can be performed on these two states, their notation and values are completely identical, and therefore the states are perceptually aliased.

The problem that we may at one state not always have the exact same actions to perform, may be a problem when trying to solve our model. The remainder of this section will be dedicated to two proposed solutions, a guided approach, and an augmented approach.

Guided Approach

Our first proposal for a method is to use the set of states as defined in Section 4.1.3, and extend the FSVI algorithm with methods that can see whether a specific action can be executed. This allows for selecting different actions in different scenario's, while still having the same state representation.

One of the drawbacks of this method, is that the added functions to FSVI will need to be performed on the actual game state rather than on the model state. This makes the algorithm still dependent — although less heavy and indirect — on the full state space. The biggest profit obtained through this method is that we keep the size of the state space relatively small.

The core idea of the approach, is that instead of using all the α vectors to update V and determine a policy, we only use the vectors that correspond to a currently available action. For this, all we need to do is transform Equation 3.7 to take only the available actions into account:

$$\pi_V(b) = argmax_{a:(a \in A_{\text{avail}} \ and \ \alpha_a \in V)} \alpha_a \cdot b \tag{4.1}$$

Here A_{avail} denotes the available set of actions at any given moment. Using this set, we can select the best action currently available, rather than trying to execute an impossible action over and over again.

Augmented Statespace

The second proposed approach tries to remove the problem of perceptual aliasing, by augmenting the current state with the actions that can be taken at that specific state. This allows for a more accurate approximation of the real α-vectors for each configuration, at the expensive of increasing the statespace. The increase of the statespace size is expected to be less than the reduction gained by using an abstract representation instead of the full statespace.

4.3 Transitions

Now that we know the definition of our states and actions, we need to define the means of getting from one state to another by executing an action. Since the transitions in a POMDP model do not need to be deterministic, we need to calculate a probability for each possible resulting state.

Fortunately for us, a lot of actions that we can perform on a state provide a deterministic result. For example, when we play a card we know for certain that it will move from our hand to the battlefield.

Taking into account that our statespace is as large as it is, we need to be able to automatically compute all states s' for which $tr(s, a, s') > 0$. One of the main difficulties in calculating these probabilities for the transtions, will be when the transition is defined by the opponent's turn, e.g., it influences some of the unobservable elements of the state.

Section 4.3.1 through Section 4.3.6 will define the calculations for the transitional probabilities of our model through means of the action taken.

4.3.1 Play($\langle Card \rangle$)

For playing a card, computing the transition probability is deterministic. If we play a creature, the state will be altered by changing an index of 2 to 3 in the

category this creature belongs to. If we do not play a creature we stay in the exact same state, for we derive our model state solely from the information given by creatures.

Algorithm 4 shows how to obtain the transition probabilities of state s, with action a being 'Play($\langle Card \rangle$)', with $\langle Card \rangle$ being any arbitrary card in a player's deck.

Algorithm 4 Play($\langle Card \rangle$)

if Type($< Card >$) = CREATURE **then**
  $s' \leftarrow s$
  $y \leftarrow$ Category($< Card >$)
  $s'[y].replace(2, 3)$ //Moves a card from 'Hand' to 'Battlefield'
  $tr(s, a, s') \leftarrow 1$
else
  $s' \leftarrow s$
  $tr(s, a, s') \leftarrow 1$ //We remain in the same state
end if

4.3.2 Activate($\langle Card \rangle$, $\langle Ability \rangle$)

The result of activating an ability is deterministic, though the effect of it will differ from card to card. As activating an ability can cost — or provide — mana, the set of available actions after activation will change as well. Appendix B contains the effect of each ability.

4.3.3 Attack($\langle Card \rangle$)

The result of attacking with a creature can be determined only after the opponent has assigned all blocking creatures. Also, since multiple creatures can attack during a turn, we can not establish with complete certainty what the result of the action is until after the battle phase. This makes the Attack action undeterministic, and we need to find a way to calculate all possible outcome states, as well as their probabilities.

As a first step in the calculation, we will assume that we have all attacking creatures known at the beginning of our calculation. This assumption does not need any change in the current state, as we do not keep track of specific status modifiers of any cards in our model. The benefit of simplifying the calculations this way, is that all that we need to do is calculate the probability that a certain blocking creature will or will not block an attacking creature.

For this, we need to enumerate all possible configurations for blocking, and assign these an equal probability. For each of these configurations we can then calculate the resulting state by means of the power and toughness of each card.

4.3.4 Block($\langle Card \rangle$, $\langle OpponentCard \rangle$)

Blocking a creature is partially a determinstic action, and partially a non-deterministic action. If a player blocks an attacking creature with a single other creature, the results are easily determined. However, when a player assigns multiple creatures to block a single attacker, the results of the action are not as clear anymore, and more specifically can not be determined when we put one creature into a 'blocking' state.

Therefore we will use the same simplification as in the Attack action, we will assume that we know all blocking creatures are known at the time where we want to compute the result. This makes the result of the action deterministic, for we can easily calculate the damage dealt to each permanent, and see whether it remains on the battlefield, or needs to be moved to the graveyard.

4.3.5 Discard($\langle Card \rangle$)

The result of discarding a card is completely deterministic. The result of the action is given by moving the designated card from the player's hand to the corresponding graveyard. Algorithm 5 describes how to calculate the resulting state s' from s, and sets the correct transition probability

Algorithm 5 Discard($< Card >$)

```
if Type( < Card > ) = CREATURE then
    s' ← s
    y ← Category( < Card > )
    s'[y].replace(2, 4) //Moves a card from 'Hand' to 'Graveyard'
    tr(s, a, s') ← 1
else
    s' ← s
    tr(s, a, s') ← 1 //We remain in the same state
end if
```

4.3.6 Opponent's Turn

The turn that is controlled by the opponent, gives an extra undeterministic transition in both the MDP and POMDP models. As we do not know which cards the opponent has in his or her hand, and which cards he or she draws, we can not determine much about the amount of options that the opponent has. Furthermore, even if we do know the options — as in the MDP version of the model — we can not know the strategy of our opponent, and therefore which actions he or she will take.

As we can calculate all the possible states that the opponent is in at the end of a player's turn — by means of the calculations described in Appendix A.2 — we can still say some things about our opponent's turn. If we assume that our opponent takes actions at random, with each available action having

an equal probability of being selected, we can calculate the probability of being in a specific state after the opponent's turn has been finished.

Even though selecting a randomized strategy as our opponent's policy is in general not a good thing to do, since it will give us incorrect probabilities, we hope that doing this will guide the exploration of the Q_{MDP} and FSVI algorithms into the right direction. By exploring each and every state at an almost equal probability, we hope that our final policy can easily adjust to a more realistic opponent.

4.4 Rewards

Rewards are used in a POMDP model to guide the agent towards good actions, as well as guiding the agent away from the bad actions. The algorithm used to determine optimal actions for the model is influenced for a large part by the rewards (and penalties) that are obtained during the execution of a run.

Since the algorithm is influenced as much by the rewards that are used, this section will contain a discussion about which actions to reward in which way, and the amount of reward that we need to assign to an action.

Section 4.4.1 will first discuss briefly why we can not rely purely on rewards for solving the perceptual aliasing problem described before. Section 4.4.2 will then discuss which actions to reward, and introduces the rewards that we will investigate further during our tests.

4.4.1 Invalid Actions

One of the questions when determining rewards for actions, is how to handle the selection of invalid actions. The first idea that comes to mind is not giving a reward to invalid actions, causing the average discounted reward to decrease.

However, since receiving a reward of 0 is always higher than receiving a negative reward, this introduces the risk of ending up in an infinite loop, for invalid actions will always result in the same state.

Another option is to give a negative reward for executing an invalid action. However — due to perceptual aliasing — it might be possible that the action will be the best action, if it can be executed. Therefore, giving a negative reward will reduce the attractiveness of executing this action when it actually could be executed as a good move.

This leads to the decision to also provide the exact actions that can be executed, in either an augmented fashion, or from an external source. These approaches have been described in Section 4.2.1.

4.4.2 Rewards

When we want to give rewards to guide our agent into the right direction, apart from the values of the rewards and penalties we first need to determine which actions we will and which actions we will not reward/penalize.

An obvious direction to proceed into, is to give every card that enters the battlefield a small reward, as it increases the possibilities that a player has, and therefore the options that he or she has to win the game. However, since our states do not have a notion of non-creature cards, giving these cards a reward upon entering the battlefield yields a certain "random"-like reward for the model. This introduces the danger for self-loops: the agent would maybe prefer to take actions that keep it in a certain state, because of the random rewards that it might get.

Even though this does not need to be a bad property for an agent, prevention of self-loops is at the very least a starting point in selecting the actions that we will reward. When we only look at actions that change the state of the agent, we get the following list of actions:

- The player draws a creature
- The player casts a creature
- The player discards a creature
- A creature controlled by the player dies
- The opponent draws a creature
- The opponent casts a creature
- The opponent discards a creature
- A creature controlled by the opponent dies

This list gives us a nice starting point, but it will probably not be a good list. Rewards should be used to guide the agent into taking the right actions. Because drawing a creature is not an action that the agent can influence easily, it seems like a random reward the agent might get during its turn, and arriving in that state will not necessarily aways give the same reward. It might however look like a good action to the agent, possibly guiding the agent into the wrong direction. Since the agent does not influence the discard choices of his opponent either, setting a reward for this poses the same risk.

Another remark on this list of actions that we reward, is that neither of the winning objectives is encapsulated in the list. We intuitively want to give a penalty for losing the game, and a reward for winning the game, so as to guide the agent towards a winning strategy. Apart from these rewards for winning and losing the game, we will also assign a reward to inflicting damage on the opponent's life total, and a penalty on receiving damage for the player itself. This way the agent hopefully gets directed away from a low life total — even though these are not shown in the state of the model — and towards a low life total for the opponent.

The full list of actions that we need to reward is given in Table 4.3, together with reward values for different strategies. Both the rewards and penalties in the table have been estimated by experienced players as an initial value, with

Reward	No	Low	High
Player creature cast	0	4.6	46.0
Player creature discard	0	−1.6	−16.0
Player creature dies	0	3.3	33.0
Player inflicts damage	0	8.0	80.0
Player wins game	0	7.6	76.0
Opponent creature cast	0	−2.0	−20.0
Opponent creature dies	0	5.1	51.0
Opponent inflicts damage	0	−4.0	−40.0
Opponent wins game	0	−7.6	−76.0

Table 4.3: The reward strategies

a score between −10 and 10 for the low values, and these results are multiplied by 10 for the high values.

4.5 Observations

The observations part for the model is usually utilized for modeling sensors on an agent. Using these sensors the agent can with certain probability determine properties of the state that it resides in. For a magic player, this might mean that we can check whether or not a certain card resides on the battlefield of the opponent. However, all parts that we can determine by modeling such a sensor, we can already see encoded in the state elements of the model.

Therefore, the complete parts of the model dedicated to these Observations can (in our case) be regarded as redundant information. Even so, algorithms working on POMDP models will need this information, and therefore we will define a set of observations and the probability that we will observe them.

As a straightforward solution, each action will have as corresponding observation, the new state s' that the action brings the game to. This makes Ω for our model equal to S, the set of states. The observational probabilities, $O(a, s, o)$ will for the player's actions then be given as follows:

$$O(a, s, o) = \begin{cases} 1 & \text{if } o = s' \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

As the opponent's turn is not deterministic, the result of the opponents turn is not deterministic either. However, since we can calculate $tr(s, a, s')$ for the opponent's turn as well, we can easily determine $O(a, s, o)$ too, since for any action, $o = s'$ needs to hold, giving $O(a, s, o) = tr(s, a, o)$.

4.6 Initial Belief State

The last element of the POMDP model is the initial belief state that we are working on. This is the only part of the POMDP model that we will actually use, and it models at each point of the game the probability for all possible hidden configurations of cards.

With regards to the game of Magic, the initial state is the point where each player has shuffled his or her deck, but has not yet drawn their initial seven cards. At that point in the game, we know of each card where it is: the corresponding player's library. This state for the Elves versus Goblins case study, is depicted in Table 4.4.

$^{\text{Power}}/_{\text{Toughness}}$	Elves	Goblins
$^{1}/_{1}$	11111111111	1111111111
$^{1}/_{2}$	11	-
$^{2}/_{1}$	-	111111
$^{2}/_{2}$	11111111	11111111111
$^{2}/_{3}$	111	-
$^{3}/_{2}$	-	1
$^{3}/_{3}$	111	1
$^{*}/_{*}$	11	1

Table 4.4: The initial (belief) state, before drawing any cards

This initial belief state is the same for each and every single game with these two decks. Once both players have each drawn seven cards, the belief state changes into a multitude of probable states, with all equal probabilities, each one of these states corresponding to a specific configuration of cards in the players' opponent his or her hand, and a fixed configuration of cards in the players' hand.

4.7 Putting it all together

Having determined the model that will be used for training an agent, we still need to determine how we are going to put all of this together and how we are actually going to train our agent.

The Q_{MDP} algorithm essentially builds up a part of the state space, and assigns values to the different states it visits, in order to later on retrieve whether a state was good or should be evaded. The partial state space the model builds up is empty when starting to train, as we have not seen any data yet.

The first step, in any state, is to see which actions are available. By performing these actions we observe an immediate reward, and we can see the value of the resulting state. As the value of the resulting state will initially be 0, it is not of much use for the update function yet. Using the update function given

in the Value Iteration Algorithm (Algorithm 1 on page 16), for ease of use also shown below, we can update this value:

$$Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s'} tr(s,a,s')V(s') \tag{4.3}$$

When we have done this for all possible actions, we have a nice overview of the possible resulting states, and we can deterministically select the best action to perform. If more than one action would get us to a resulting state with the highest value, we can select one at random, and explore whether that action is actually a good action.

During the remainder of the training games, some states will already have values stored in them, and when we encounter them we can propagate these back through the model with a discount factor, making good actions now more beneficial than performing an action that might result in being able to perform a better action later.

Using this method, we try to prune uninteresting parts of the state space in an early part of the training, causing the later training games to be able to focus only on the interesting parts.

Chapter 5

Test Setup

This chapter will describe globally which experiments we are going to run to see how well the Q_{MDP} algorithm performs with regards to other baseline strategies. Because no previous existing research on Magic: The Gathering could be found, we have implemented some simple algorithms ourselves to serve as a comparison.

5.1 Strategies

This section contains a brief description of the various strategies that we will use as a benchmark for the POMDP model that we constructed in Section 4.7. We start off with three hand-made strategies: a random player and two semi-random players that have been adapted to show more intelligent behavior. We have also implemented a Monte-Carlo Tree Search algorithm and a beam search algorithm which are described here as well.

Random This strategy models a player who knows completely nothing about the game, and plays fully at random. All possible moves are generated at each point in the turn where the player needs to make a decision, and this strategy — without performing any further calculations — assigns each of the actions an equal probability, and blindly selects one.

We do not expect this strategy to be intelligent in any form, but it will mainly be used as a baseline to have a comparison method for all other strategies.

Aggressive The aggressive strategy works the same as the random strategy, apart from a few points. As a first, it will always play a LAND card when it has the possibility to. This increases the amount of possibilities the player has later in the game.

The second difference with the pure random strategy is that during the battle phase of each turn, all creatures will be assigned to attack. This allows for easy killing of the less powerful creatures, but hopefully also still inflicts a lot of damage on the opponent.

Defensive The defensive strategy has been implemented as the counterpart of the aggressive strategy. Even though it also plays a LAND card whenever the possibility arises, the main focus of this strategy is to be defensive. A player with this strategy will not attack with any creature that will get killed indefinitely when blocked by one or more creatures on the opponent's battlefield. Even though this strategy is not seen often in strong decks, it is an essential part of specific types of decks (such as mill-decks in which the player tries to defeat his opponent by removing all cards from his or her library).

The hypothesis behind this strategy is that it will work decent for the Elves deck, keeping many of the abilities on the card accessible, as the cards are not removed in the players' attack phase.

Monte-Carlo Monte-Carlo is a general term for algorithms that try to optimize a value by averaging over a lot of different trials. In our case we use an algorithm named Monte-Carlo tree search, where we try to optimize the number of winning games when playing x games at random from the resulting state of each original move. We then execute the move with the highest value.

Beam Search Beam search can best be described as a form of breadth-first-search with an automated layer of pruning. The main idea of the algorithm is to construct a heuristic that can estimate the value of a specific state by purely looking at that single state. We calculate for each action the heuristic of the resulting state, and take the x best states, where x is called the beam width. Then for each of these states we do the same, and again take from the total set of resulting states the x best. The algorithm keeps repeating this until there is only one "source-action" (original action) left in the set of best states, and then it performs this action.

A small example is presented in Figure 5.1 with a beam width of 2. Each vertical level is a new set of expansions. Because the best nodes on level 2 are from the same parent, this action is selected (and therefore printed bold).

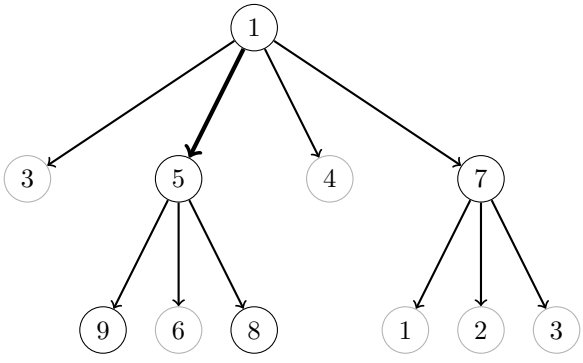

Figure 5.1: A simple beam search example

5.2 Test Cases

To be able to properly compare all strategies with each other, we will let them play 1,000 games against the Random strategy. By doing this we assign a baseline function to the random player, on which the more intelligent strategies should improve. Also, this allows us to immediately see if the more complex algorithms are worthwhile.

As a direct comparison between the semi-random strategies and the POMDP model alone does not give us information regarding the complexity of the algorithm (e.g., there is only one algorithm and we expect it to outperform the semi-random strategies), we have also implemented a basic Monte-Carlo search algorithm [1] and a beam search algorithm [11].

The Monte-Carlo strategy works with a very simple heuristic: In each state, it plays x random games for each available action, and the action with the most wins out of the x games is the action that it performs.

Beamsearch is an extended form of breadth-first-search, and tries to obtain a better result by pruning at each level during the search, keeping only the best x states in memory. As soon as all of these x states have the same ancestor action, it stops the search and executes that action. Another stop criterion is when one of the states is winning for the player, in which case the algorithm immediately executes its ancestor. The key of the algorithm lies in finding a good heuristic in combination with a proper beam width. Determining this heuristic will have to be done by trial and error, and will be described under test results.

Chapter 6

Results

This chapter describes the results obtained from running the tests as described in Chapter 5, as well as comparison between the various methods. Each of the results are obtained from running 1,000 games unless noted otherwise. Section 6.1 will describe the results for the Random and Semi Random agents. Section 6.2 will give the results of a small Monte Carlo test on the game. Section 6.3 will give in addition to the obtained results on the Beam Search algorithm, also some remarks with regards to finding a good heuristic. Finally, in Section 6.4 the results for the full Q_{MDP} algorithm are shown.

6.1 Semi-Random

When looking at the results for the random-based strategies, the first interesting point to look at is how much of the games agents playing fully at random can win. These results are shown in Table 6.1, with the player starting the game being shown on the left. The percentages in this table correspond with the percentage of games that the player that started the game has won. The strategies are shown with only their first letters: R for Random, A for Aggressive and D for Defensive. Results printed in **bold** are results where the player performed really strong, and underlined results are where the player performed exceptionally bad.

		Elves			Goblins		
		R	A	D	R	A	D
Elves	R	48.1%	53.3%	64.4%	53.2%	26.8%	49.8%
	A	68.2%	57.6%	62.2%	68.1%	35.8%	56.8%
	D	57.1%	52.9%	55.1%	65.9%	32.9%	44.6%
Goblins	R	44.6%	50.1%	61.2%	49.4%	26.0%	44.3%
	A	**75.8%**	69.5%	**75.7%**	**73.3%**	52.4%	67.3%
	D	44.3%	55.2%	60.6%	57.3%	34.3%	49.9%

Table 6.1: Semi-Random vs Random

The first point of interest in this table is seeing how the pure random strategy performs on average, and note that in comparison with each other they achieve a win in nearly 50% of the games they started. We will use these values later on as a baseline for comparison with the other algorithms.

The next thing to observe, is that the aggressive strategy works very well, and is very stable for the Goblins deck. Independent of whether the Goblins deck starts or not, as long as it plays the aggressive strategy it wins around approximately 70–75% of the games.

When we look at the strategy that we had developed to work for the elves deck, we find a more interesting result, begin that it doesn't differ as much from the results of the random strategy. Even more, when the Elves deck begins the game, it also appears to be more useful to also play in an aggressive setting. Looking at the situation where the deck is not the beginning player, it is even better to play with a random strategy.

6.2 Monte Carlo

The Monte Carlo algorithm is in our case designed for the sake of evading obvious pitfalls. The results of running either 1 or 10 random games per move are shown in Table 6.2.

		Random	
		Elves	Goblins
Elves	Random	48.1%	53.2%
	MC(1)	49.0%	44.4%
	MC(10)	46.0%*	43.0%*
Goblins	Random	44.6%	49.4%
	MC(1)	59.6%	55.4%
	MC(10)	54.0%*	59.0%*

Table 6.2: Monte Carlo vs Random

The results denoted with a * — those of the games with 10 random games per move — are gathered from running 100 games instead of 1000 due to constraints in computation time.

The obtained results vary widely per setting, but in general we see that when playing against the Elves deck with a random strategy, the probabilities to win a game stay in approximately the same region. When playing against the Goblins deck, we see diverging results. When computing more random games for the Elves deck, it appears to get worse, while doing the same for the Goblins deck increases its chances to win.

6.3 Beam Search

The first part of the the tests for beam search involved determining the base for a good heuristic. The heuristic that we chose to utilize makes use of the form

$$(w_1 \cdot \text{Life}_{\text{self}}) + (w_2 \cdot \text{Library}_{\text{self}}) + (w_3 \cdot \text{Creatures}_{\text{self}}) - (w_4 \cdot \text{Grave}_{\text{self}}) -$$

$$(w_5 \cdot \text{Life}_{\text{opp}}) - (w_6 \cdot \text{Library}_{\text{opp}}) - (w_7 \cdot \text{Creatures}_{\text{opp}}) + (w_8 \cdot \text{Grave}_{\text{opp}})$$

where Life_x is the amount of life that the player x has, Library_x the amount of cards in its library, Creatures_x the amount of creatures on the battlefield and Grave_x the amount of cards in the players' graveyard. Weights w_1 through w_8 can be arbitrary real numbers and weigh the different factors among eachother.

Using this form, by filling in different values for the weights, we found that several parts were better suited for constructing a good agent than others, but that the actual weights themselves did not really make much of a difference, as long as the components were right.

Using either 0 or 1 for the weights, we then ran two sets of experiments, one where we would only take into account weights w_1 through w_4, and in the other one only weights w_5 through w_8. Assuming that good partial solutions would form a decent final set of weights yielded [1 0 1 1] for weights 1 through 4, and [0 0 0 1] as well as [1 0 0 1] for weights 5 through 8.

Combining these for our final tests yields:

$$\text{Config1}: \ \text{Life}_{\text{self}} + \text{Creatures}_{\text{self}} - \text{Grave}_{\text{self}} + \text{Grave}_{\text{opp}}$$

$$\text{Config2}: \ \text{Life}_{\text{self}} + \text{Creatures}_{\text{self}} - \text{Grave}_{\text{self}} - \text{Life}_{\text{opp}} + \text{Grave}_{\text{opp}}$$

The results of these tests are shown in Table 6.3. Because we expect the algorithm to behave near-random in the first few turns of the game, we did not initiate the beam search algorithm until the first players' libary contains 30 or less cards.

		Random	
		Elves	Goblins
Elves	Random	48.1%	53.2%
	Config 1	57.5%	53.7%
	Beam Config 2	57.6%	53.1%
Goblins	Random	44.6%	49.4%
	Config 1	57.6%	53.2%
	Config 2	57.5%	52.2%

Table 6.3: Beam search vs Random

As is clearly visible from this table, the beam search algorithm performs at least on par with the random results, and in most cases a little better. This is probably due to the fact that the algorithm is able in some cases to predict a winning scenario where the random player would perform a different action.

6.4 Q_{MDP}

The final and most complex algorithm amongst the ones tested is of course the Q_{MDP} algorithm. For this algorithm training games are required in order to build up the model, and we have trained a model for each of the four individual games, over multiple runs. We have trained such a model over 250,000 games, whilst checking the progress and quality of the model every 50,000 games. The results are shown in Table 6.4.

		Random	
		Elves	Goblins
Elves	Random	48.1%	53.2%
	Defensive	57.1%	65.9%
	Aggressive	68.2%	68.1%
	50,000	73.3%	74.5%
	100,000	71.9%	71.9%
	150,000	71.9%	68.8%
	200,000	72.2%	72.7%
	250,000	70.7%	69.3%
Goblins	Random	44.6%	49.4%
	Defensive	44.3%	57.3%
	Aggressive	75.8%	73.3%
	50,000	69.6%	69.5%
	100,000	68.1%	68.9%
	150,000	68.4%	69.0%
	200,000	71.1%	69.0%
	250,000	65.7%	67.4%

Table 6.4: Q_{MDP} vs Random

The first thing we notice when looking at this table, is that for the Goblins deck the Aggressive semi-random strategy outperforms the Q_{MDP} model by a large amount, at each level of training. For the Elves deck this is not the case, but the aggressive strategy is nearly on par with the results gained from training the full model.

Another result obtained, is that the model seems to start performing worse when it has been trained over more and more training games. It appears that for each of the 4 situtations, the model performs best after 50,000 training games, and gradually decreases when training it more. We think that this happens because after many games the weights start spreading more evenly over all actions, making the next action to perform a near-random decision in some states. Even though the state space is still large, it can still be the case that some states appear much more often than others. If this is the case, overtraining could also be part of the explanation for this decreasing efficiency.

Also, the model is not capable of generalizing actions and states. When it

has — for example — determined that when it is in a state where it has two creatures on the battlefield and the opponent none, it is a good thing to attack with all creatures, it does not extrapolate this to three or more creatures.

The model does however outperform the other two types of random strategies with ease, so probably with a few adjustments to the trained model it might be possible to obtain even better results.

Chapter 7

Conclusions

In this report we have tried to develop an artificial agent that is able to cope with a large state space through means of contructing a POMDP model. For training and solving the model we have used an algorithm based on reinforcement learning and simple Markov Decision Processess, the Q_{MDP} algorithm. As a test case we have taken a subset of the game Magic: The Gathering. No previous results had been obtained for either the test case or the efficiency of POMDPs on a large state space.

Even though the Q_{MDP} algorithm has a setup that does not limit the size of the state space, using it on a large problem with many actions per state decreases its efficiency. Also, because the algorithm does not take advantage of problem specific characteristics, its usefullness on exploiting these characteristics is limited.

Because of these shortcomings, the algorithm can be outperformed by very simple problem-specific strategies. In the game of Magic: The Gathering however, these problem-specific strategies will in general only work on one specific deck, and a (well-trained and configured) model will probably be more easy to extend towards small adjustments.

Whether it is a good investment to build a complete model and implement a more advanced (self-adaptive) algorithm in the end completely depends on the problem and the situation at hand. When dealing with a fixed problem, of which you know the characteristics, it is probably a better option to build a simple agent which exploits these characteristics rather than building a full model. However, when these characteristics are not well-defined, or the problem situation can change later on, a self-adaptive algorithm can cope with these changes. It is however still a good idea to extend such an algorithm with problem-specific generalisations and characteristics.

7.1 Outlook

A lot of research still needs to be done on POMDP-models for large state spaces, and on the adaptions to the algorithm needed to incorporate things like extrapolation on states with similar characteristics. The Q_{MDP} does not incorporate the Partial Observable parts of the created model, and therefore an algorithm like FSVI based on the model generated from this algorithm might be able to obtain better results by moving away from the exact states.

To develop a good agent to play the game of Magic: The Gathering, much research needs to be done as well, and a more complete engine will have to be implemented in order to play a game with all options that are available to a real player. Also, different algorithms might be better in simulating the human logic for determining a strategy.

Chapter 8

Bibliography

[1] R. Motwani D.Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[2] L. Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 183–188. AAAI Press, 1992.

[3] A. Churchill. Magic: The Gathering is Turing complete. `http://www.toothycat.net/~hologram/Turing/`.

[4] A. Gill. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.

[5] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370. Morgan Kaufmann, 1995.

[6] Wizards of the Coast Inc. Basic rulebook for Magic: The Gathering. `http://www.wizards.com/magic/rules/EN_MTGM12_Rulebook_web.pdf`.

[7] Wizards of the Coast Inc. Extensive rulebook for Magic: The Gathering. `http://www.wizards.com/magic/comprules/MagicCompRules_20120601.pdf`.

[8] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[9] G. Shani, R. I. Brafman, and S. E. Shimony. Forward search value iteration for POMDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 2619–2624, 2007.

[10] R.S. Sutton and A.G. Barto. *Reinforcement Learning*. MIT Press, 1998.

[11] W. Zhang. *State-Space Search: Algorithms, Complexity, Extensions, and Applications*. Springer, 1999.

Appendix A

State Conversion

Since during the process of the FSVI algorithm we work with model states, and during the game simulation we work with game states, it is required that we define a proper transition between both. The transition from game to model state is described in Section A.1, and the transition from model to game state is described in Section A.2.

A.1 From game state to belief state

Even though — through the simplification obtained in Section 4.1.3 — we have obtained a somewhat more tractable statespace, it would not be feasible to enumerate all possible configurations, and assign them a probability, as is done in normal POMDP calculations. However this enumeration is not needed in our case, as at any given moment, most of the entries in our belief state will be equal to zero. For example, when we start playing a game, and each player has drawn his initial seven cards, there are no cards at all on the battlefield or the graveyard yet, so any configuration that describes cards being in either of these locations, has a probability of zero that we are in that specific state.

Thus, we will need to find a way to describe our belief state in terms other than probabilities over the full belief space. However, since the calculations done by the FSVI algorithm described in Section 3.3.2 are done on probabilities, we will also need to find a way to calculate the probability that we are in a specific given state, given our current belief state representation.

A.1.1 Visibility

The first thing that we will need to describe for our belief state, is a notion of visibility. During a game, there will be eight different locations that cards can reside in:

- Library
- Hand
- Battlefield
- Graveyard
- Opponent's Library
- Opponent's Hand
- Opponent'sBattlefield
- Opponent's Graveyard

While intuitively it may seem that a player can only have knowledge about the location of his or her own cards, we luckily have the advantage that we can also state a couple of things about the location of the cards in the opponent's deck.

At first there is the battlefield. During any game of Magic, this location is fully visible to any player, or even to spectators, and therefore it is eligible to use this location in our calculations. Also, since (in general) there are cards that can affect an opponent's or one's own graveyard, both of these locations are visible for either player, though not continuously arranged this way.

Given that we can observe the amount — and even which specific cards — reside in the battlefield and graveyard of the opponent, we can rule out some of the total configurations described in Section 4.1.3. Also, it helps us in calculating probabilities over the things that we do not yet know: what cards the opponent has "in hand".

The last small detail of visibility aids in calculating these probabilities, though still not revealing with a 100% accuracy which cards are in the opponent's hand. What we do need to know however, and what is normally visible, is the amount of cards in the opponent's hand. When humans play a game of Magic, they normally keep their cards up in front of them, and thus one can easily count the amount of cards an opponent has in his hand.

A.1.2 Calculations

We can, through these observations, calculate the probability for each and every given state in the belief space, from observing purely the locations of these cards.

As a first item, any configuration that configures more or less cards in a fully visible location than there actually are, gets a probability of zero. However, since we are completely not interested in the probabilities that are zero, we want to be able to generate the probabilities for non-zero states only, given our visible states.

Let us first illustrate an example of a game that is currently being played. We assume to have the viewpoint of the 'Elves' deck, but the calculations for the 'Goblins' deck are (in essence) the same.

Elves

Library

39 Cards

Hand

2 Forest
1 Llanowar Elves
1 Elvish Eulogist
1 Slate of Ancestry
1 Lys Alana Huntmaster

Battlefield

6 Forest
2 Llanowar Elves
1 Wellwisher
2 Elvish Warrior

Graveyard

2 Giant Growth
1 Elvish Eulogist
1 Wildsize

Goblins

Library

40 Cards

Hand

5 Cards

Battlefield

8 Mountain
2 Mudbutton Torchrunner

Graveyard

2 Tarfire
1 Clickslither
2 Emberwilde Augur

Figure A.1: An example of a visible configuration.

The listings in Figure A.1 give us a nice example to work with while explaining the necessary steps to compute the states that have a non-zero probability.

The first step in calculating these probabilities is generating strings per category, describing on what locations the cards currently reside. As noted before, these strings are expressions of a partial ordering, and will therefore be ordered sequences of digits. The cards that we are not able to see the location of, will be denoted by question marks (?).

	Elves	Goblins
$^{1}/_{1}$	1111122334	????????33
$^{1}/_{2}$	11	
$^{2}/_{1}$		????44
$^{2}/_{2}$	11111111	??????????
$^{2}/_{3}$	133	
$^{3}/_{2}$		?
$^{3}/_{3}$	112	4
$^{*}/_{*}$	11	?

Table A.1: The configuration corresponding to Figure A.1

Furthermore, we can state that each card currently denoted by a question mark is either in the opponent's library, or in it's hand. Having this fact, each configuration where the question marks are — for each category — replaced by a number of 1's followed by a number of 2's would be a valid configuration.

However, we can reduce the calculations even further, by in this stage already eliminating the configurations in which the total amount of cards denoted by a 2 is unequal to the amount of cards currently in the opponent's hand.

As an example of a calculation, we will now calculate the probability that our opponent will have two cards from the category $^{2}/_{2}$, one card from the category $^{1}/_{1}$, one card from the category $^{*}/_{*}$, and one non-creature card in hand.

To calculate the probability that we get — in order — two cards from $^{2}/_{2}$, one card from $^{1}/_{1}$, one card from $^{*}/_{*}$, and one non-creature card, we can simply multiply the corresponding probabilities as follows:

$$\frac{11}{45} \cdot \frac{10}{44} \cdot \frac{8}{43} \cdot \frac{1}{42} \cdot \frac{20}{41} \approx 0.00012$$

A thing that one might recognize however, is that when drawing items from a set, order is important: it would seem that drawing the $^{*}/_{*}$ card from a set of 45 cards has a lower probability than drawing the same card as the last card, when there are only 41 cards available still. However, since the formula consists of only multiplications and divisions, both commutative operations, it does not matter for the total probability of drawing. What *is* true however, is that we can indeed draw these cards in multiple orders, and that this increases the probability that we get this final hand.

Thus, if the order does not matter to calculate the probability, but we do get multiple possibilities of drawing this hand, we can write the total calculation down as calculating a single probability — the formula above in our example — and a multiplier. This multiplier then describes the total amount of possibilities to draw these 5 specific cards.

To be able to calculate this, we need a definition from stochastics theory: combinations. The corresponding formula to this definition calculates for us the amount of possible combinations to draw an amount of n items from a set of k items is:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

We can again combine the different values through multiplication, to obtain our final multiplication. For our example, this gets us

$$\binom{5}{2} \cdot \binom{3}{1} \cdot \binom{2}{1} \cdot \binom{1}{1} = 10 \cdot 3 \cdot 2 \cdot 1 = 60$$

, which makes the total probability of getting the desired configuration $0.00012 \cdot 60 = 0.0072$.

A.2 From belief state to game state

Analog to what is described in Section A.1 for the conversion from a game state to a probability distribution over states in the model, we need to have a transformation the other way around as well. In order to determine which moves the opponent will be able to make during his or her turn, we will need to calculate the corresponding game state. Unfortunately, due to the discarded information during the transform from game to model states, we can not with full certainty determine in which game state the opponent will be — not even if we would be fully certain about the model state he or she is in. Our solution to this problem, is to compute a probability distribution over game states.

To calculate this probability distribution, we have created two algorithms. Algorithm 6 sets up the basic variables that have to with the opponents deck and the visible cards thereof.

Algorithm 6 CalcReal(s)

OppVisible ← Opponent(Visible)
OppLibrary ← Opponent(Library) − *OppVisible*
Distribution ← *EnumerateRecursive*(1, *OppVisible*, s, *OppLibrary*)

After this initialization it calls the recursive function EnumerateRecursive which is described in Algorithm 7.

The first thing this algorithm does, is check whether the model state *state* corresponds with whatever is declared in *Visible* (and thus 'fixed').

If these two states are equal, it stores the current *Probability* value as the probability that model state *state* is equal to what is declared in *Visible*. If they are not equal, we sample the category Y from the list of categories that are not equal between the two.

Next, the algorithm creates a list of possible cards from the library that are in category Y. For each of these cards, we compute the probability that this card is the card in the opponents' hand. This new probability, along with the new *Visible* and *Library* sets are then propagated on to a next recursive call, in which the difference between *state* and *Visible* is effectively decreased by one element.

Algorithm 7 $Result$ = EnumerateRecursive($Probability$, $Visible$, $state$, $Library$)

1: **if** $state = Visible$ **then**
2: $Result[D] \leftarrow P$
3: **else**
4: $Result \leftarrow \emptyset$
5: $Y \in (Visible \cap state)$
6: $X \leftarrow$ PossibleSolutions(Y)
7: **for each** $x \in X$ **do**
8: $Result \leftarrow Result +$ EnumerateRecursive($Probability \cdot P(x)$, $Visible + x$, $state$, $Library - x$)
9: **end for**
10: **end if**

Appendix B

DeckLists

This appendix contains the decklists of both the Elves and Goblins decks.

B.1 Elves

The contents of the Elves deck are the following cards:

19 Forest	2 Tranquil Thicket	1 Wirewood Lodge
1 Ambush Commander	1 Allosaurus Rider	2 Elvish Eulogist
1 Elvish Harbinger	3 Elvish Warrior	2 Gempalm Strider
1 Heedless One	2 Imperious Perfect	3 Llanowar Elves
2 Lys Alana Huntmaster	1 Stonewood Invoker	1 Sylvan Messenger
1 Timberwatch Elf	1 Voice of the Woods	2 Wellwisher
1 Wirewood Herald	1 Wirewood Symbiote	2 Wood Elves
1 Wren's Run Vanquisher	1 Elvish Promenade	2 Giant Growth
1 Harmonize	1 Wildsize	3 Moonglove Extract
1 Slate of Ancestry		

Table B.1: The decklist of the Elves deck

B.2 Goblins

The contents of the Goblins deck are the following cards:

22 Mountain	1 Goblin Burrows	1 Forgotten Cave
1 Siege-Gang Commander	1 Akki Coalflinger	1 Clickslither
3 Emberwilde Augur	1 Flamewave Invoker	1 Gempalm Incinerator
3 Goblin Cohort	1 Goblin Matron	1 Goblin Ringleader
1 Goblin Sledder	1 Goblin Warchief	1 Mogg Fanatic
2 Mogg War Marshal	2 Mudbutton Torchrunner	2 Raging Goblin
1 Reckless One	2 Skirk Drill Sergeant	1 Skirk Fire Marshal
1 Skirk Prospector	1 Skirk Shaman	1 Tar Pitcher
2 Boggart Shenanigans	1 Spitting Earth	3 Tarfire
1 Ib Halfheart, Goblin Tactician		

Table B.2: The decklist of the Goblins deck

Glossary

Ability Describes an action that a card can perform, either by activating it — with for example mana or tapping the card — or as a response to events that happen during the game.

Battlefield The location during the game where most cards of interest appear. This is where lands are put in order for them to generate mana, and where creatures are summoned to defend from, and attack your opponent.

Counters Counters are state-modifiers that can for example give a card more or less power or toughness, but they can also be used to charge certain cards in order to use their stronger abilties. Counters are a permanent modifier, that stay attached to the card until it is moved into the graveyard, or the counters are removed in a different way.

Creature A type of cards. CREATURE cards can be used to attack the opponent, defend your lifetotal, or for any other purpose that can be provided throught it's abilities — such as for example generating mana.

Deck The set of cards that is used by one player during a single game of Magic. The size of the deck is usually 60 cards, but can — under certain circumstances — range from 40 to 200 cards.

Discard This can refer both to effects listed on cards and to the effect that happens at the end of a turn when a player has more than two cards in his or her hand. Discarding is the term used for moving a card from your hand to the graveyard.

Double Strike A keyword ability indicting that a creature deals damage before all others do, as well as when the other creatures deal damage. If there is at least one creature involved in battle — attacking *or* blocking — that has this ability or the "First Strike" ability, an additional damage phase is added before the regular one.

Exile A location where cards are moved when they are removed from play. Cards moved into Exile can — generally — not be returned into play anymore.

First Strike A keyword ability indicating that a creature deals damage before all others do. If there is at least one creature involved in battle — attacking *or* blocking — that has this ability or the "Double Strike" ability, an additional damage phase is added before the regular one.

Graveyard The location during the game where cards are presumed "dead". Instant and Sorcery type cards are moved here after they resolve. Creature, Artifact, Land and Planeswalker type cards are (generally) moved here when they are destroyed, sacrificed or when they die.

Hand When cards reside in this location during game, they are available for casting. Also, some activated abilities such as "Cycling" are available from a card when they are in your hand. (The "Cycling" ability requires discarding of the card — most common in addition to a certain amount of mana — to draw or search for a card.)

Haste A keyword ability indicating that a creature card is not effected by "Summoning Sickness", meaning that the creature can attack and tap during the turn it was cast in.

Library A location in the game where the remainder of your deck is. Cards are drawn from this location, and some abilities or effects allow for searching cards in this location. When this location does not contain any cards, and the player has to draw a card, he loses the game.

Life total A state modifier corresponding to a player. The life total of each player starts at 20, and the main goal of the game is to reduce the opponent's life total to a value 0 or lower.

Land A type of cards. Players can play one card of this type per turn, and Land cards are generally used for the generation of mana, though a wide variety of other abilities have appeared on Land cards as well.

Mana The main source of energy in the game. It can be obtained by — for example — tapping lands, can be used to cast spells and activate abilities.

Permanent A name for cards that stay on the battlefield after being cast. The card types that are targeted as permanents are Land, Artifact, Enchantment, Creature and Planeswalker

Sacrifice Denotes that as a cost for activating an ability, the player must move a card from the battlefield to the graveyard.

Summoning Sickness A state modifier that creatures obtain when they enter the battlefield. The effect of this modifier is that they are not allowed to attack, or activate abilities that require them to become tapped. This modifier is removed from your creatures at the beginning of each turn. Since blocking your opponent's cards does not require tapping, creatures that have this modifier are still allowed to block.

Tap Moving the card from portrait orientation to landscape orientation, in order to show that it has attacked, or an ability that requires the card to tap has been executed.

Trample A keyword ability appearing on creatures, denoting that the remainder of the damage this creature deals, will be dealt on the opponent's life total.